TECHNISCHE UNIVERSITÄT DRESDEN

Fakultät Informatik Institut für Technische Informatik Professur für Compilerbau

Diplomarbeit

zum Thema

Simulation of RISC-V based Systems in gem5

vorgelegt von	Robert Scheffel
im Studiengang	Informationssystemtechnik, Jg. 2012
geboren am	04.01.1994 in Görlitz

zur Erlangung des akademischen Grades eines

Diplomingenieurs
(Dipl.-Ing.)

Betreuer:	Dipl.-Ing. Christian Menard
	Dipl.-Ing. Gerald Hempel
Verantwortlicher Hochschullehrer:	Prof. Dr.-Ing. Jeronimo Castrillon
Tag der Einreichung:	20.08.2018

TECHNISCHE UNIVERSITÄT DRESDEN

cfaed CENTER FOR ADVANCING ELECTRONICS DRESDEN

Fakultät Informatik Institut für Technische Informatik, Professur Compilerbau

Aufgabenstellung für die Diplomarbeit

für	**Robert Scheffel**
Studiengang:	Informationssystemtechnik, Jahrgang 2012
Matrikel-Nr.:	3873256
Thema:	**Simulation of RISC-V based Systems in gem5**

RISC-V is a modern and open instruction set architecture (ISA) that has a lot of traction in research as well as industry. Especially for embedded applications, RISC-V is of high interest due to its energy-efficient design and its extensible ISA for application-specific costumization.

In the process of designing and evaluating new systems, virtual prototypes (VPs) are an essential tool. Raising the abstraction level from the register transfer level (RTL) to the system view, allows for both faster design space exploration and faster simulation of systems. In order to research and evaluate RISC-V based systems, it is crucial to create VPs of RISC-V cores and to integrate them with a full system simulator. The objective of this diploma thesis is to design and implement such a simulation environment.

gem5 is a virtual prototyping and system simulation framework that is widely used in research and industry due to its openness and flexibility. However, gem5 currently only provides limited support for the simulation of RISC-V cores. In this diploma thesis, the RISC-V support in gem5 shall be improved such that a full system running an embedded firmware can be simulated. Furthermore, gem5 shall be extended such that it supports simulation of user-defined custom instructions. In order to make the work available to a broad community, the changes shall be submitted to the mainline gem5 repository.

The structure of the work in this thesis will be as follows:

1. Familiarization with the RISC-V ISA and the ecosystem (i.e., build tools).
2. Familiarization with the gem5 simulation framework and the current status the support for RISC-V.
3. Prototype implementation of a full system simulation that executes a simple firmware without using any interrupts or periphery.
4. Extension of the prototype to support simulation of a realistic firmware including interrupts and basic periphery devices (e.g., Timer, UART). This includes the design of use-cases that illustrate the correctness of the simulation.
5. Definition and implementation of an interface for defining custom instructions in gem5.
6. Thorough testing of the full system simulation and the realization of custom instructions.
7. (Optional) Extension of the prototype to support simulation of a full operating system.
8. (Optional) Comparison of the gem5 simulation to a real RISC-V core and calibration of the gem5 model.
9. (Optional) Implementation of a mechanism for extending the RISC-V tool-chain by custom instructions.
10. Writing the actual thesis (in English) documenting all the above points.

Betreuer:	Christian Menard und Gerald Hempel
1. Prüfer:	Prof. Dr.-Ing. Jeronimo Castrillon
2. Prüfer:	Prof. Dr.-Ing. Akash Kumar (Vorschlag)

Ausgehändigt: 12. März, 2018 Einzureichen: 20. August, 2018

[illegible] [illegible]

Prof. Dr.-Ing. habil. Martin Wollschlaeger
Vorsitzender des Prüfungsausschusses

Prof. Dr.-Ing. Jeronimo Castrillon
Verantwortlicher Hochschullehrer

Acknowledgements

I would like to thank my supervisors Christian Menard and Gerald Hempel for their help and guideline. Their door was always open whenever I had a question about my research or writing.

I would like to express my gratitude to Professor Castrillon for the opportunity to write this thesis at the Chair for Compiler Construction.

I wish to acknowledge the help provided by CommSolid. The opportunity to use their FPGA board enriched this thesis with a more detailed accuracy analysis. My special thanks are extended to the staff of CommSolid for providing me help and consulted me during my work with the board and the collection of my data.

TECHNISCHE
UNIVERSITÄT
DRESDEN

Fakultät Informatik Institut für Technische Informatik Professur für Compilerbau

Simulation of RISC-V based Systems in gem5

The open and free instruction set RISC-V has properties ideal for building embedded systems. In this domain, full system simulators, such as gem5, are used for fast design space exploration. The absence of this type of simulators for the RISC-V architecture limit its usability in academia. This thesis presents a solution for closing this gap by introducing the implementation of the Full-System mode in gem5 for the RISC-V architecture. Furthermore, the RISC-V Extension Parser is introduced, offering the possibility to define custom extensions. These implementations enable full system simulation of RISC-V based embedded systems with arbitrary custom instructions and registers. Moreover, the accuracy of full system simulation in gem5 is evaluated.

Tutor:	Dipl.-Ing. Christian Menard
	Dipl.-Ing. Gerald Hempel
Supervisor:	Prof. Dr.-Ing. Jeronimo Castrillon
Day of Submission:	20.08.2018

DIPLOMA THESIS Author: Robert Scheffel

Contents

List of Figures

List of Tables

List of Listings

Abbreviations

AHB	advanced high-performance bus
ALU	arithmetic logic unit
API	application programming interface
AST	abstract syntax tree
CISC	complex instruction set computing
CPU	central processing unit
DSL	domain-specific language
FFT	fast Fourier transform
FS	Full-System
GCC	GNU Compiler Collection
glibc	GNU C Library
IoT	internet of things
ISA	Instruction Set Architecture
M-mode	machine-mode
MPSoC	multi-processor system-on-chip
PLIC	platform-level interrupt controller
PMP	physical memory protection
RISC	reduced instruction set computing
RTL	register-transfer level
TTM	Time-To-Market
SE	Syscall-Emulation
SoC	system-on-a-chip

TLB	translation lookaside buffer
UART	universal asynchronous receiver-transmitter
U-mode	user-mode
VP	Virtual Prototype

1 Introduction

In today's industry, companies have to deal with increasing complexity of hardware and software architectures [21]. To decrease Time-To-Market (TTM), costs, and efforts, it is more efficient to develop hardware and software concurrently instead of sequentially. Therefore, Virtual Prototypes (VPs) are used to simulate the exact behaviour of real hardware [57]. Whereas the industry offers SystemC based CPU models, in academia open source simulation frameworks are needed. A requirement on these frameworks is the possibility to evaluate a wide variety of architectures and facilities including network communication and standard IO devices [11]. These demands can be fulfilled with *full system* simulators [26]. Such simulators are capable of simulating not only a central processing unit (CPU) and memories, but also a whole system-on-a-chip (SoC) including peripheral devices.

A full system simulator already used in academia is the gem5 simulator [11]. This open and free project allows easy collaboration. Due to its different simulation modes, the support of many different Instruction Set Architectures (ISAs), and the simpleness of defining new system models, it is a useful tool for architectural evaluations. Furthermore, this simulator offers the possibility to do cycle-accurate simulations with feasible accuracy.

The open and free ISA RISC-V [83] is designed to solve problems in existing architectures. It has a simple base instruction set and is designed extensible to better integrate efficient accelerators close to the core. Around the ISA itself, an ecosystem containing a core generator [5], a hardware construction language [7], a toolchain [13, 30], and simulators [9, 42, 72] has arisen. Due to these properties and availability of open source tools, RISC-V is ideal for research purposes.

Currently, for simulating RISC-V based systems, two different approaches are available. The first is to use simulators in the RISC-V ecosystem. These are instruction-accurate ISA simulators that lack full system capabilities. Cycle-accurate simulations are possible with VHDL models mapped to FPGA for detailed register-transfer level (RTL) simulation. However, the possibility for full system simulation of RISC-V based systems is missing. Moreover, existing simulators are not able to conveniently capture custom extensions defined by users.

This absence of full system simulators limits the possibilities of architectural evaluations in research using this instruction set. Additionally, missing possibilities to define custom instructions and registers in simulation models prevents the usage of one of the key features of RISC-V. A solution to overcome this deficiency is presented in this work by extending the support for RISC-V in gem5 and enabling the full system simulation mode. Additionally, an interface is designed making defined custom extensions available for use in simulation. This offers researchers the possibility to use the advantages of the RISC-V ISA in full system simulation for architectural evaluations.

This thesis is structured as follows. In Chapter 2 the ISA RISC-V and the simulator gem5 are introduced and Chapter 3 motivates the benefit of using both together. In Chapter 4 use-cases for embedded systems are defined and from them requirements on the implementation are derived. Additionally, the implementation of features needed to fulfil these requirements is described. Chapter 5 introduces the verification of the implementation. Furthermore, an accuracy evaluation of the gem5 simulators is described. Additionally, an example on accelerating algorithms by utilising the RISC-V Extension Parser is given. After that, Chapter 6 presents relevant projects regarding RISC-V and gem5. In Chapter 7 ideas for future work based on the outcomes of this thesis are described. Finally, Chapter 8 concludes this work.

2 Background

This chapter introduces the ISA RISC-V and the simulator gem5. Moreover, related work is described highlighting the interest in both projects.

2.1 RISC-V

This section introduces the ISA RISC-V. It shortly explains the architecture itself as well as available software tools. Also, a short comparison to similar architectures is given explaining the advantage of using RISC-V.

2.1.1 Overview

RISC-V is an open and free ISA, that was introduced in 2010 by the University of California, Berkeley. The name originates from the history of ISA projects, whereas RISC-V is the fifth major reduced instruction set computing (RISC) project of the university [83].

As the name implies, the design of the ISA is based on principles of RISC and developed to fit the demands of nearly every computing device, from embedded devices to desktop computers [4, 48, 83]. Therefore, RISC-V is designed to be highly extensible and offers the possibility to extend the base instruction set with already defined standard extensions, as well as user-defined custom extensions [80].

Most important is the openness of the architecture, which enables competition and innovation [3, 83] and allows RISC-V to be used in research as well as in the industry [56]. For example Nvidia plans to use a RISC-V core on their GeForce graphics card [85].

2.1.2 ISA Specification

This subsection will shortly introduce the architecture specification itself. The specification is structured in two parts, the user-level ISA [80] and the privileged architecture [81]. “The RISC-V Instruction Set Manual Volume I: User-Level ISA” [80] explains the base instruction sets and standard extensions. The second part, “The RISC-V Instruction Set Manual Volume II: Privileged Architecture” [81] discusses instructions that occur beyond user-level as well as the purpose of machine registers and their behaviour at certain events. Where the user space ISA is fixed, the privileged architecture is still a draft and applications have to consider that some parts may change.

User-Level ISA

To be able to be used in both small, embedded devices and personal computers, without being over-optimized for a special use-case, the architecture is modular and highly extensible [83]. RISC-V supports both 32 Bit and 64 Bit address spaces, where their base instruction sets differ in the register width. Support for 128 Bit architectures is in development [80].

RISC-V is, as many other RISC architectures, a load-store architecture [29]. There, it is differentiated between instructions for arithmetic logic unit (ALU) operations and memory accesses. The RISC principles lead to few instructions of a fixed size that consume similar time in execution [77]. In the case of RISC-V, the instructions are 32 Bit wide. Due to the similar execution time, pipelines can be fully utilized and less structural and data hazards occur [10]. Also there are only four core instruction formats, which can be seen in Figure 2.1.

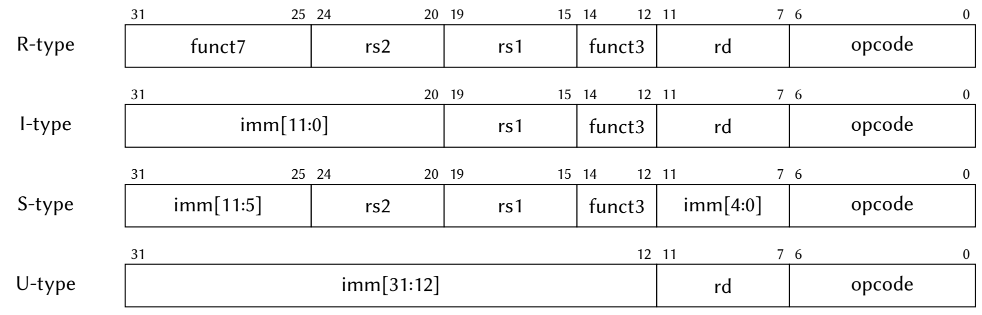

Figure 2.1: Base instruction formats of the RISC-V user-level ISA (adapted from [80]).

The base instruction set is sufficient enough to implement a fully functional general-purpose computer [83]. Aside of it, many standard extensions, such as compressed instructions and integer multiplication, are specified [80]. To describe the actual implementation, each extension is encoded with a letter and than described as "RV32IMC" or "RV64I". Table 2.1 gives an overview of defined standard extensions and their encoding. If a computer has the extensions "IMAFD", it can be also referred to as "G", which stands for "general purpose".

Especially in the embedded domain, applications run with limited clock frequencies in order to save power. That leads to poor performance and computation

Table 2.1: RISC-V standard extensions that are not expected to be changed in future editions of the user-level ISA.

Extension	Description
M	integer multiplication and division
A	atomic instructions
F	single-precision floating point
D	double-precision floating point
Q	quad-precision floating point
C	compressed instructions

intensive calculations often need to be supported by additional hardware [35, 45]. To address this issue, certain operation codes, where a user can define custom instructions are reserved in the ISA [80]. Table 2.2 shows these reserved opcodes.

Table 2.2: RISC-V opcode map for RVG instructions (adapted from [80]). In the highlighted fields the operation codes for custom instructions can be seen.

inst[4:2]	000	001	010	011	100	101	110
inst[6:5]							
00	LOAD	LOAD-FP	custom-0	MISC-MEM	OP-IMM	AUIPC	OP-IMM-32
01	STORE	STORE-FP	custom-1	AMO	OP	LUI	OP-32
10	MADD	MSUB	NMSUB	NMADD	OP-FP	*reserved*	custom-2/rv128
11	BRANCH	JALR	*reserved*	JAL	SYSTEM	*reserved*	custom-3/rv128

RISC-V has 32 general purpose registers *x0-x31*, where 31 of them are integer registers and *x0* contains a constant *0*. Every register is as wide as the address space of the implementation, for example 32 Bit for RV32I. Additionally, the register *pc* holds the address of the current instruction [80].

Privileged Architecture

Every aspect of the ISA beyond user-level is described in the privileged instruction set specification, including functionalities to attach devices and run operating systems. This part of the specification is still a draft, hence sections of it may change. It is designed to be completely independent of the user-level ISA and is specified to support commonly used operating systems [81].

To fulfil the design goals of RISC-V the “The RISC-V Instruction Set Manual Volume II: Privileged Architecture” has to support several types of systems. Therefore, three different privilege modes are defined: machine, supervisor and user. Table 2.3 gives an overview about their abbreviations and encodings. The only mandatory level for a RISC-V hardware platform is the machine level having the highest privileges. Code running in machine-mode (M-mode) has access to the registers of the machine itself and is therefore completely trusted [81]. Since the machine level is the only mandatory privilege level, simple embedded systems implementations might provide only M-mode. However, secure embedded systems, that intend to protect the system from untrusted user code, will at least provide hardware with two privilege levels, user-mode (U-mode) and M-mode. In such a scenario, the application code runs in a lower privilege level and only traps are handled in machine-mode.

Table 2.3: Overview of the different privilege levels in RISC-V, including encodings and abbreviations.

Level	Encoding	Name	Abbreviation
0	00	User/Application	U
1	01	Supervisor	S
2	10	*reserved*	
3	11	Machine	M

In the RISC-V architecture, a trap causes the transfer of control to a trap handler. The cause of a trap can be synchronous or asynchronous. Synchronous traps are called exceptions. These are associated with an instruction and therefore occur synchronously to the program flow [80]. On the other side, interrupts have an asynchronous character, as they refer to external events forcing the current RISC-V thread to an action. Such events could be a timer alarm [81]. For both, interrupts and exceptions, the privileged architecture specification defines, which machine registers have to be altered and how the transfer of control happens. It is also specified, what actions have to be done, once the trap handler finishes its work.

Other aspects covered in the specification are memory systems and memory protection. Memory protection is useful for secure embedded systems, as they may need to protect memory-mapped registers, machine mode code and data from user access.

2.1.3 RISC-V Ecosystem

Although RISC-V is a new ISA, necessary software tools for design and development of RISC-V based hardware and software are available. In terms of tool support, a developer does not need to go without commonly used programs.

This subsection gives a general overview of the available software stack, as seen in Figure 2.2, with some extended comments on compilers and implementations.

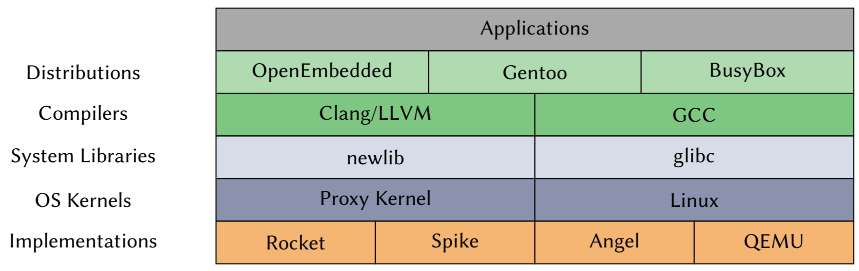

Figure 2.2: Overview of the RISC-V software stack (adapted from [43]).

Compiler Support

For development of software for RISC-V based systems, the two popular compilers GNU Compiler Collection (GCC) and Clang/LLVM are available. Whereas GCC supports all RISC-V ISA variants [19], Clang/LLVM only supports the base instruction set for 32 Bit address mode [13]. Furthermore two standard libraries can be used with GCC. The GNU C Library (glibc) is a full blown implementation of the *C* standard library and intended to use with the RISC-V Linux port. Embedded systems use Newlib, the second available standard library [16]. The RISC-V port of Clang/LLVM only targets Newlib [19].

Simulators and Emulators

The ecosystem contains two ISA simulators, Spike [72] and Angel [42], whereas the latter is not under active development any more. Spike is the golden RISC-V model [19]. It implements the latest versions of the user-level specification and privileged architecture, and can therefore be used as a reference when implementing the RISC-V ISA. Spike models a core and a cache system and is capable of booting Linux. Besides ISA simulation, full system emulation is also possible with QEMU. The ecoystem offers a port for this open source machine emulator and virtualiser. QEMU translates system calls to instructions on the host system, leading to fast simulations on instruction-level accuracy [68].

RISC-V Hardware

When implementing a RISC-V based system, sooner or later, the developer has to build actual hardware. Due to its openness, one does not need to start from scratch, but can use either core generation tools or reuse actual implementations that are publicly available.

Having a look at the software stack in Figure 2.2, another RISC-V implementation is Rocket [5]. Rocket is an open source SoC generator developed at UC Berkeley. It consists of different generator libraries that can be used to design custom SoCs. Among them, different core generators offer the possibility to build multi-core RISC-V systems. Therefore, Rocket is, like the ISA itself, capable of generating a wide variety of computer systems.

Actual RISC-V silicon is also available. The company SiFive offers multiple products, including single chips, SoCs and whole development boards. Such a commercially available development board is the SiFive HiFive1 [38]. The architecture of this board is introduced later in Section 5.2.1, as it is used for evaluating the accuracy of the gem5 simulator.

2.1.4 Comparison to other ISAs

When implementing a new hardware system, a developer faces the question, which ISA the CPU should realize. Basically he can decide, whether he wants to build a RISC or complex instruction set computing (CISC) architecture. CISC tries to achieve a high performance with executing fewer instructions per program. This approach leads to rather complex decoders, as they need to decode a lot of different instructions [63]. Therefore, RISC cores are preferred in the embedded domain, due to their lower area consumption. Taking a look into the industry provides evidence for this statement. Currently, about 99% of the microprocessors implement RISC architectures [63].

This section compares RISC-V with other popular RISC ISAs. Advantages against other ISAs will be highlighted. For comparison, the ISAs ARM, MIPS and SPARC are analysed, as they are among the most popular and widely used RISC architectures [23, 60].

ARM

The ISA ARM was developed in 1985 by the British company Acorn Computer [23]. Later, the name was changed to Advanced RISC Machines Ltd, which is now ARM Ltd.. The most recent version is ARMv8 [2]. It was announced by ARM Ltd. in 2011 and is a complete redesign of the previous version *ARMv7* [83]. It adds an optional 64 Bit architecture, referred to as AArch64, offering compatibility to 32 Bit architectures, known as AArch32. Moreover, it allows 32 Bit applications to be run in 64 Bit operating systems. Currently, ARM processors are used in mobile devices and automotive [60].

Both versions, 32 and 64 Bit, differ quite sever in their specification. The 32 Bit mode defines 16 32 Bit wide registers, where 12 of them are general purpose registers. The added 64 Bit mode in ARMv8 defines 32 64 Bit wide registers. AArch64 is fully compliant to the IEEE floating point standard with its NEON extension. Compressed instructions are supported in AArch32 but not in AArch64 [23]. Also the privilege levels are complicated. The ISA defines two different base modes, user and privilege mode. The privilege mode is further divided into system mode and exception mode. In system mode is in fact a user mode with higher privileges and therefore more access rights. Under the exception mode, five more modes are defined. The first is the supervisor mode that is entered on reset or if a software interrupt instruction is executed. Second, the abort mode is defined, handling memory access violations. The third mode is the undefined mode handling undefined instructions. For occurrence of interrupts, two further modes are defined, the interrupt mode and the fast interrupt mode. The interrupt mode is entered if a normal interrupt occurs. On occurrence of high priority interrupts the fast interrupt mode is entered. In each of the exception modes, the registers have a different meaning. Overall, this leads to 9 different modes with different register conventions [60].

MIPS

The MIPS [34] architecture was introduced in 1982 by the University of Stanford and heavily influenced by the IBM 801 minicomputer [83]. The architecture was originally designed to support personal computers. Later, MIPS processors were used in gaming consoles such as Nintendo 64, Sony's PlayStation and PlayStation2 [36]. Today, the architecture is mainly used in embedded systems and MIPS based chips are among the most shipped architectures [23, 63]. MIPS is a proprietary instruction set, as the architecture is developed by MIPS Technologies. They had a patent on misaligned load and store instructions. Although the patent is already expired [33], the architectures remains a trademark of Imagination Technologies and compatibility to the architecture cannot be claimed without their permission [83].

MIPS has 32 32 Bit wide registers, where the register $0 is hard-wired to zero. Additionally, the architecture provides two special registers, *HI* and *LO*, which are used by the integer multiplication and division instructions. These instructions run asynchronously, allowing them to be executed separately from and concurrently with other instructions [23]. MIPS supports only two operating modes, kernel and user mode. To kernel mode is switched at power up and on occurrence of an interrupt or exception. The user mode has lower privileges than kernel mode and it prevents different users from interfering with one another [60]. For floating-point operations MIPS intends to use a separate coprocessor, which complicates single-chip designs [83].

SPARC

SPARC [39] was developed by Sun Microsystems and first released in 1987. Its design was strongly influenced by the projects RISC-I [64] and RISC-II [44], proposed by the University of Berkeley [83]. The architecture's first 32 Bit implementation was used in Sun's Sun-4 workstation and server system. Later, field of application moved towards server systems and SPARC is considered as a high performance architecture for systems that rely on high throughput [60]. SPARC is completely open, free and non-proprietary.

The main difference of the SPARC architecture compared with others is its register file model. Unlike in MIPS, ARM and RISC-V the model is not flat but is an overlapping register window, which is arranged in a circular buffer [23]. In the SPARC architecture 37 32 Bit wide registers are available and divided into four groups. Similar to MIPS, SPARC has two different privilege levels, namely user and supervisor mode. The instruction set of the architecture is rather simple and includes 90 instructions. Hardware support for floating-point computation adds another 50 instructions.

Comparison

The criteria used for the comparison consider typical requirements on ISAs in general and in context of embedded systems according to Waterman [83]. An open and free instruction set gives academia and industry the opportunity to evaluate designs without paying license fees. Furthermore, support for 32 Bit and 64 Bit wide addresses allows more flexible processor designs. Compressed instructions and a separate privileged ISA are features used in embedded devices. In a domain requiring for small designs in order to save costs, the code size can be reduced with compressed instructions. Additional gates can be saved by not being forced to support privileged architecture specifications. A last criterion is the hardware support for the IEEE standard for floating-point arithmetic, which leads to more performance while processing floating-point values.

An overview of the compared architectures and the criteria is shown in Table 2.4. All selected ISAs support both, 32 Bit and 64 Bit address widths. The RISC-V ISA is the only architecture fulfilling all chosen criteria. RISC-V and SPARC are the only two open source ISAs. Of all compared ISAs, RISC-V is the only architecture that has a separate privileged architecture specification. This feature offers flexibility to adapt the ISA to its specific use-case. For example, in small embedded systems no privilege levels have to be implemented, therefore a lot of special registers can be saved. In contrast, the privileged architecture is implemented when designing a system used as general purpose computer running different applications that are not trustworthy. Both examples are still conform to the RISC-V base instruction set.

A drawback of the RISC-V ISA is the unreleased architectural specification. Although most parts of it are unlikely to change, current implementations need to be aware that changes might happen. Another problem is the poor support of custom extensions in the toolchain. It is possible to add new assembly instructions, but this requires patching of files deep inside the project. The missing support in the toolchain exacerbates working with one of the key features of the architecture, its extensibility.

In conclusion, without RISC-V a computer architect has to evaluate the needed features carefully. Consequently he chooses the ISA offering the most desired features. The introduction of RISC-V supersedes this evaluation as RISC-V supports all these features. Moreover it is free and open, allowing to explore its features complimentary.

Table 2.4: Summary of several ISAs' support for desirable architectural features.

	ARM	MIPS	RISC-V	SPARC
Free and Open			✓	✓
32 Bit and 64 Bit	✓	✓	✓	✓
Compressed Instructions	✓	✓	✓	
Separate Privileged ISA			✓	
IEEE 754-2008	✓		✓	

2.2 The gem5 Simulator

This section gives an introduction to gem5 in general, introduce the usage of the simulator and explain its ISA parser, a crucial part of the system. Also, a more detailed comparison to other simulation frameworks is given. After that, related work regarding calibration of simulators is introduced.

2.2.1 Overview

The open source system simulator gem5 [11] originates from M5 [12] and GEMS [55]. M5 focuses on network simulation, whereas GEMS is a timing simulator for multiprocessor systems. Gem5 merges the aspects of both of its predecessors to enable computer-system architecture research. It is not only used in academia, but in industry, i.e. ARM and AMD use gem5 internally and contribute to the project. It supports many different ISAs, which are varyingly strong supported and maintained, as seen in Table 2.5.

Gem5 offers two different modes, the Syscall-Emulation (SE) mode and the Full-System (FS) mode. Within the SE mode, every system call gets translated to its equivalent on the host system. This call is executed by the host and the answer is then taken as response. In SE mode the behaviour of a binary can be investigated, that is run on a system of a specific architecture. Due to translation to a system call of the host system, this mode is faster but not as accurate as the FS mode. Thereby, a full system model with many different components, such as caches, interrupts and I/O devices, is simulated. Furthermore, the execution of the operating system is simulated and unmodified binaries can be executed. [11, 59]

The simulator offers different models for CPUs and memory with different levels of detail and complexity. Each CPU, as well as every other component, is

Table 2.5: Overview of the supported architectures in gem5.

ISA	Level of ISA support	Full-system OS support
Alpha	high	Linux
ARM	high	Linux, BSD, Android
MIPS	low	none
Power	low	none
RISC-V	medium	none
SPARC	low	none
x86	medium	Linux, BSD

implemented independently from an ISA. That allows the use of nearly every component in every ISA [11, 59].

Usage of gem5

To use gem5, two things are required. First, a binary of the simulator is needed. This binary is compiled specifically for one ISA, which is RISC-V in our case. Second. the definition of a system is required. A system definition describes of which parts the SoC consists and how they are connected [11].

The most basic system, on which a binary can be run, is displayed in Figure 2.3. It only contains a CPU and a memory block. Both are connected via a memory bus, that allows communication. As this simple system uses no caches, the ports for data and instruction cache are directly connected to the bus.

The functionality of the simulator is implemented in *c++*. Functions describing the behaviour of the components are defined. Furthermore, the simulator provides a configuration interface. Components used to build a system can be configured in *python*. Therefore, every configurable object provides a *c++* implementation and a *python* definition. Concluding, parameters and specifications are defined in the *python* class and the description of the whole system is done in a *python* script [59].

The gem5 Decoder

The decoder is a crucial part of gem5, where ISA independent CPU models meet ISA specific binary instructions. To achieve this independence, all CPU

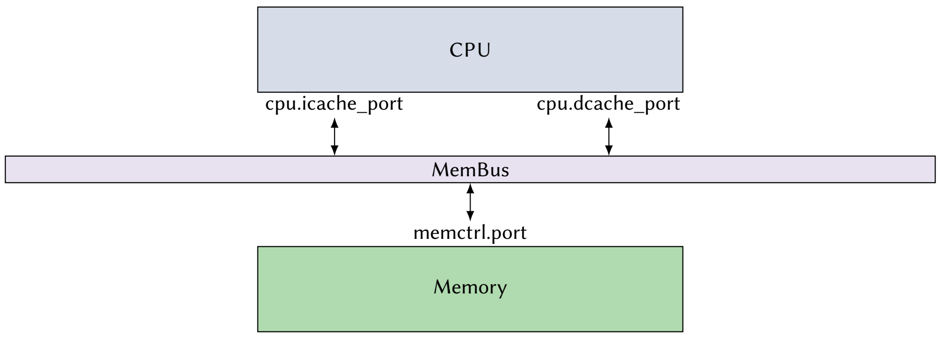

Figure 2.3: The simplest system, that is able to execute a binary.

```
decode OPCODE {
  0: add({{ Rc = Ra + Rb; }});
  1: sub({{ Rc = Ra - Rb; }});
}
```

Listing 2.1: Example definition of instructions in the gem5 decoder.

implementations inherit from a base class. In the implementation of opcodes, virtual functions of the base class are overwritten [11].

For defining ISAs, gem5 provides a domain-specific language (DSL) allowing to overwrite the mentioned virtual functions. When compiling, the ISA description language is parsed and translated to proper *c++* code. An example definition of two instructions in this DSL can be seen in Listing 2.1. There, the semantics of an *add* and a *sub* instruction are described. It also shows how the decode tree is built. With the keyword *decode*, defined bit fields can be observed. Depending on its value, further decoding is done or instructions are assigned. In this example case, decoding the field *OPCODE* is enough to determine, whether *add* or *sub* needs to be executed.

For every instruction defined with the ISA DSL, a *c++* class is generated providing certain functions. Most important to mention is the *execute()* method, which defines the behaviour of the ISA instruction. The code generation of these classes and methods can be controlled with formats. They describe the code that is omitted when parsing the ISA description. These formats are then used in the decode section, where a decode tree is built. There, the affiliation of an operational code to an instruction is defined. Furthermore, the format of an instruction is set, which as a consequence describes the omitted *c++* code for this instruction. Basically from the decode tree a *switch* statement is generated, that decodes the binary code, finds the proper instruction to every opcode and creates a new class object, that corresponds to this instruction. Then, its *execute()* method can be called, that leads to the realisation of the RISC-V instruction [11].

2.2.2 Comparison to other Simulators

As seen in Table 2.5, gem5 offers basic support for RISC-V. Currently, RISC-V binaries can only be simulated with the SE mode, which is faster, but lacks accuracy. Additional work needs to be done in order to simulate unmodified binaries on RISC-V based systems in gem5. This subsection compares other simulators to gem5, highlighting the advantages of the latter. It shows, why gem5 was chosen as simulator, although porting of the FS mode to RISC-V means effort.

One of the first full-system simulators was SimpleScalar [14]. This open source project was initially designed to simulate MIPS architectures. Later versions enabled functional and timing accurate simulation of ARM and Alpha systems [25]. Compared to gem5, no RISC-V support is available and therefore, this simulator is not used for this project, as the effort to implement the ISA support is to high.

Later on, focus moved towards the ISA x86. Several simulators where released, that support only the specific CISC architecture [53, 62, 86]. Even though some of them where cycle-accurate [53, 86] their restrictions regarding the simulated architecture limit their wide use in research.

OVPsim [41] is a full-system simulator, which is freely available for non commercial use and marketed by Imperas. It offers support for many different architectures, for example ARC, ARM, MIPS and RISC-V. Through its public application programming interfaces (APIs), users can create own processor and platform models, and contribute them to the community. OVPsim offers different models for RISC-V processors as well, by implementing different bit widths and standard extensions. This simulator models an instruction-accurate processor, which is not as accurate as cycle-accurate simulation. Compared to gem5, which offers cycle-accurate full-system simulation, this is a disadvantage. Though, this simulator has feasible RISC-V support, the lack of cycle-accurate simulations is disadvantageous in comparison to gem5.

SystemC [40] is an *IEEE* standard that extends *c++* with macros and classes. There is a lack of free, accurate and realistic *SystemC* models for modern CPUs [57]. However, existing models are shipped as binary libraries, which prevents them from being modified. Therefore, *SystemC* is not used as simulation framework for this thesis.

As seen in Section 2.1.3, the RISC-V ecosystem already provides simulators.

The Spike simulator only models a core and a cache system and is therefore only feasible for simulation of RISC-V binaries. On the other hand, QEMU provides full-system support. The disadvantage of QEMU is, that it translates every instruction of the guest CPU to an instruction of the host CPU. This leads to simulations with less details as in gem5. Furthermore, parameters of system parts and devices can not be configured conveniently as in gem5.

2.2.3 Calibration of gem5 Components

To be more flexible and achieve more accurate simulation results, most gem5 components can be configured and calibrated. These configurations can be done in the *python* script of the system definition. Via *python* bindings, these parameters are given to the *c++* world, where the functionality is implemented. This mechanism offers the possibility to reconfigure system parameters without recompilation. Examples for parameters that can be configured is the frequency of the system, the latency of buses, memories and caches. Furthermore the latencies of IO devices can be configured.

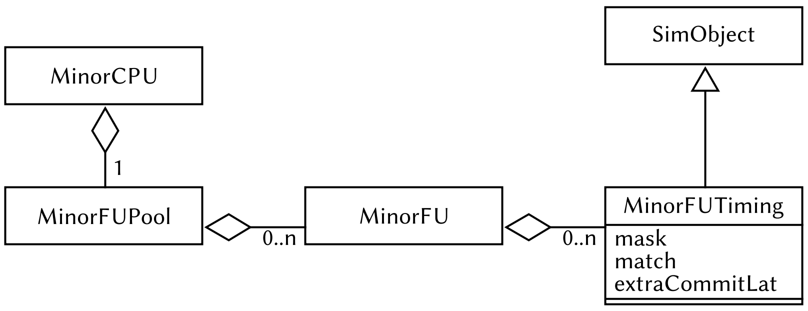

Figure 2.4: Class diagram of the MinorCPU class and its functional units.

The in-order pipelined CPU model in gem5 can be configured as well. This CPU has four pipeline stages and therefore differs from the classical textbook five stage pipeline [73]. The stages are called *fetch1*, *fetch2*, *decode* and *execute*, whereas the *execute* stage implements the functionality for instruction issue, execution and write back. For each of the stages, parameters like latency of the stage and data width can be configured. Most interesting for this work are the parameters of the *execute* stage. Therein, functional units define how many cycles the execution of an instruction lasts and how big the latency until an operation of the same operation class can be issued again. To provide a more detailed configuration of the timings of instructions, functional units can have multiple functional unit timing objects. In these objects, extra commit latencies can be defined [73].

A class diagram illustrating this structure can be seen in Figure 2.4. The *MinorCPU* has a pool of functional units. These again have a pool of functional unit timings. Therein, masks and matches can be defined to specify these timings for one single instruction of the ISA. Those masks and matches can be defined in the python script of the system definition and do not need to be set at compile time.

3 Motivation

Due to its properties, such as its openness and extensibility, RISC-V [83] is of high interest in academia. Researchers can download the specification and evaluate architectural designs without paying any license fees. The modularity and extensibility of RISC-V allows convenient adaptation to specific use-cases of the system.

For better and faster evaluations of architectures, ISA simulators are needed, being able to simulate the ISA of interest. As described earlier in Section 2.1.3, simulators for RISC-V are already available. But there is a lack of full system simulators, especially when it comes to open source frameworks. Currently, RISC-V based systems can not be simulated with feasible accuracy in reasonable speed. Researchers must choose between RTL-level simulation on FPGA boards and instruction-level ISA simulation. RTL-level simulation is slow but highly accurate, whereas ISA simulators deliver their results faster but their accuracy is limited on instruction-level.

Additionally, none of the available simulators offers the possibility to conveniently define arbitrary custom extensions for simulated systems. In conclusion, the academia is forced to use older ISAs for the evaluation of complex systems due to this absence. Another possibility is to rely on rather inaccurate simulation results. However, the advantages and the full potential of RISC-V can not be exploited in architectural research at the moment.

The open and free gem5 simulator [11] is a simulation framework offering a full system simulation mode. As described in Section 2.2.1, the modular design decouples ISAs from simulation models. With that approach, devices, such as CPUs and memories, can be used independent of the underlying ISA. Moreover, gem5 already has basic support for RISC-V by the availability of the Syscall-Emulation mode. Therefore, the ISA decoder is already implemented.

With extending this support and enabling the Full-System mode, this thesis closes the gap between RTL-level and instruction-level simulation of RISC-V based systems. Furthermore, due to the design of gem5, existing simulation models can be reused. Additionally, this work minds the extensibility of RISC-V as one of its key features. It makes convenient definition of custom extensions for the RISC-V ISA possible. The usage of these extensions in software and simulation of customized systems is enabled. With that, more expressive virtual prototypes can be implemented and the full potential of the RISC-V ISA can be exploited.

To delimit this work, goals are defined of which systems shall be simulated in gem5. As currently no support for full system simulation of RISC-V based systems in gem5 exists, the starting point is simulating very basic systems. The most basic system that can be build with the RISC-V ISA is an embedded system. Since these systems have a dedicated task within a larger system they have severe constraints regarding size, memory and energy consumption for the purpose of saving costs [74]. In consequence the hardware and the software of these systems is reduced in complexity and optimized for a dedicated function. The reduced complexity of both, software and hardware, makes embedded system ideal to implement full system support for RISC-V based systems in gem5.

In conclusion, the goal of this thesis is to enable full system simulation for RISC-V based embedded systems in gem5. An interface is designed, allowing the definition of custom extensions and makes them available for usage in RISC-V based software and in simulation. Besides the actual implementation of missing features, the functional correctness is verified. The accuracy of gem5 is evaluated by comparing the simulation to real hardware.

4 Concept and Implementation

This chapter first introduces use-cases for embedded applications. From that requirements are derived for features, that the gem5 simulator must support in order to simulate systems with these use-cases. After that concepts and implementation of the features is described.

4.1 Use-cases for Embedded Applications

As the goal is to have a working full system simulator for RISC-V based embedded systems. As shortly mentioned in Chapter 3, embedded systems are build and highly optimized for one specific function within a larger system. Hence, their architecture differ quite strong. To delimit the range of system architectures that gem5 shall be able to simulate, typical use-cases for embedded systems are introduced. These use-cases lead to features, that the simulator shall support.

To find intersections in a vast domain, that covers many different systems, definitions of Zöbel [88] and Fan [27] are taken as reference. From that, use-cases are derived, that are introduced hereinafter. To every use-case, an appropriate system is modelled. Taken together, a definition of a system arises, which gem5 shall be able to simulate.

4.1.1 Communication via UART

Embedded systems rely on the communication with the outside world to either retrieve or distribute information. Commonly, communication partners are other systems and a dedicated peripheral device is needed [27].

To solve this problem, the most basic system that can be seen in Figure 2.3 is enriched with a universal asynchronous receiver-transmitter (UART) module. UART enables asynchronous data transmission over wire and establishes a serial connection. While transmitting, the module receives data by the CPU in the form of whole words. It adds start and stop bits and creates a serial bit stream. The receiving UART device reverts the operations and recreates the parallel data packet [28].

A system, that reflects this use-case can be seen in figure 4.1. Besides the CPU and the memory, a UART device needs to be attached to the bus. In order to

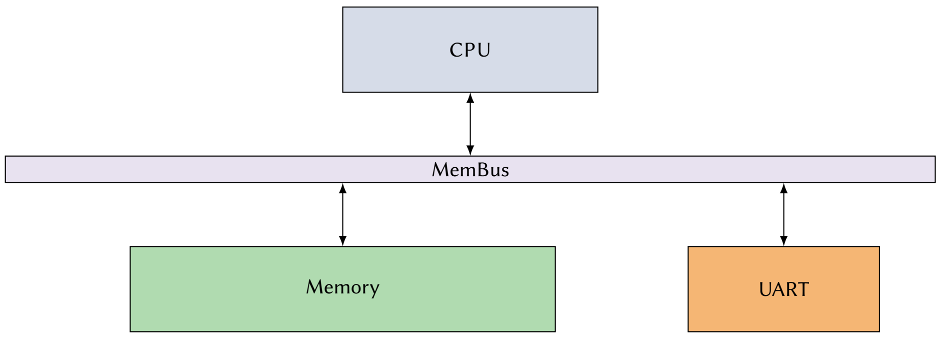

Figure 4.1: Basic system with UART device.

communicate, the CPU has to access the UART module at its address.

4.1.2 Execution of Time-Controlled Tasks

Very crucial parts of embedded systems are timers. These are used by CPUs to measure elapsed time. Another use-case is to time the execution of tasks. For example embedded systems, and other systems as well, want to prevent busy waiting. Instead of polling the whole time, going to sleep and poll at certain points in time is much more efficient, as it saves a lot of energy. This use-case would not be possible without timers, as the alarm is used as a wake-up event. That means, that a CPU timer has to be added to the system, which can be seen in figure 4.2.

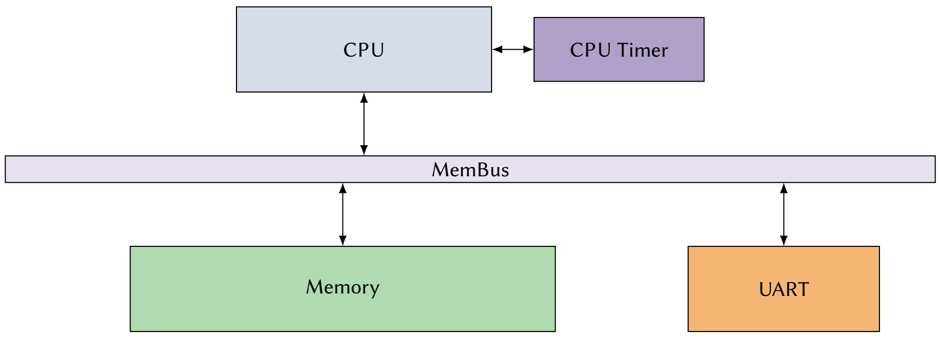

Figure 4.2: Basic system with UART device and a CPU timer.

With this periphery device comes the need for the system to handle interrupts. If a timer is set and the alarm goes off, the current execution of the application is interrupted. After handling the cause of the alarm, the interrupted program is restored and continued.

4.1.3 Performance Increase through Additional Hardware

Embedded systems often have strict constraints regarding energy consumption to ensure long-lasting battery life [51]. To achieve this, CPUs run with a slow clock speed, which leads to low computational performance.

The low throughput of the processor leads to long execution times when processing complex algorithms To approach this problem, bottle necks are often accelerated with additional hardware [54, 71]. Custom Instructions are used to group logical operations and run them directly on hardware instead of executing them in the ALU. Custom Registers are used to prevent intermediate values from being wrote back into memory.

In contrast to Shao et al. [71] who assumed additional hardware to be a separate unit on the core with its own memory, in this thesis additional instructions are assumed to be a functional unit in the CPU. Custom register are also within the CPU and additional to the general purpose registers defined in the RISC-V architecture specification [80, 81]. This fits in the modular concept of RISC-V and is similar to the approach of Mach et al. [54].

A system, that reflects this scenarios is shown in figure 4.3. The CPU has access to custom registers and custom instructions. Binaries, that run on the system can call these instructions with assembly instructions and can also access additional registers.

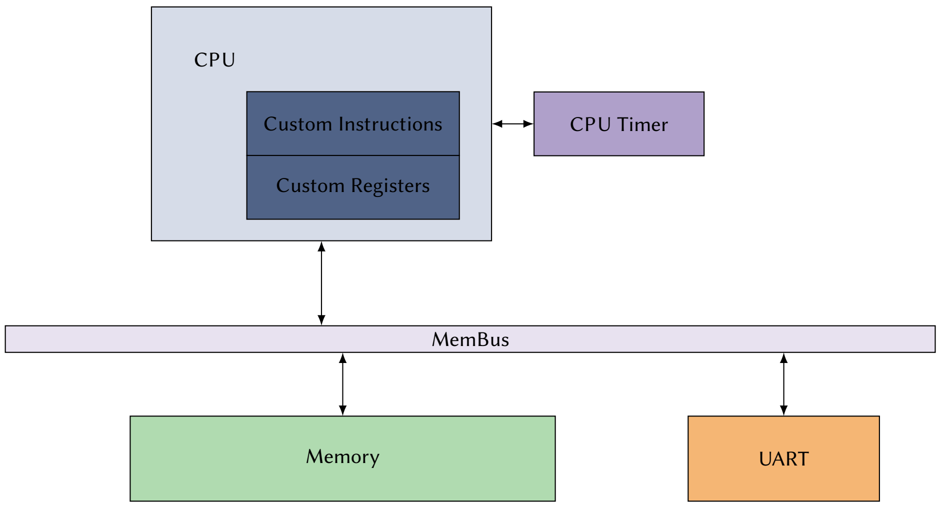

Figure 4.3: System with peripheral devices and custom extensions.

The system sketched in figure 4.3 covers all the devices and modules needed by the use-cases. It serves as a definition of a system gem5 shall be able to simulate.

4.2 Requirements

In this section, the just presented use-cases are analysed to derive requirements of which features gem5 must support. These features than lead to tasks of implementations that needs to be done in order to fulfil the requirements.

With the use-cases the definition of an embedded system was made, that shall be simulated in the Full-System mode in gem5. As described in section 2.2.1, this mode is the most accurate of the modes gem5 offers. Moreover, it allows the simulation of an unmodified binary and is therefore well-suited for architectural evaluation and virtual prototyping. Currently, the FS mode is not supported for the RISC-V ISA, as seen in table 2.5 [11]. Therefore, the first requirement is to make this mode available and enable the simulation of a rudimentary binary.

With taking a look at the use-cases, one requirement is to be able to simulate systems, that uses UART and timer devices. Therefore it is required, that objects are implemented in gem5, that reflect the behaviour of the corresponding peripheral device. More precise, a UART and a timer device is needed, that can be added in the definition of the system that shall be simulated.

The implementation of the timer object leads to another feature, that gem5 must support. Similar to real timers, the timer object can be accessed by applications and an alarm can be set. When the set alarm goes off, the timer object posts an interrupt to the CPU. The simulator must be able to handle these interrupts and take the same actions as real hardware. Currently, no support for interrupts and exceptions is available in the implementation of the RISC-V ISA in gem5. Therefore, the simulation of the use-case in Section 4.1.2 requires implementation of it.

The last use-case in Section 4.1 describes, how embedded systems gain performance through custom hardware instructions and registers. This requires that custom extensions can be defined and convenient working with them is possible. Therefore, an interface is defined where users can add make these definitions. To be able to simulate custom hardware, gem5 must be extended. Furthermore, the RISC-V toolchain also needs to be altered to enable convenient usage of the defined extensions in RISC-V binaries. Therefore, it comes clear, that behind the requirement of simulating custom instructions and registers requirements emerge, that concern more than just gem5.

As stated earlier, full-system simulation is used for architectural evaluation and

Table 4.1: Overview of requirements on gem5 and the interface for custom extensions.

Extending gem5	Interface for Custom Extensions
Full-System Mode for RISC-V	Definition of Custom Extensions
UART and timer devices	Parsing and Information Retrieval
Support for Interrupts and Exceptions	Automatically extend gem5
RV32IMC Support	Patch RISC-V Toolchain

virtual prototyping. To be able to use simulation for this purposes, a requirement is a good accuracy of the simulation. To evaluate the accuracy of the simulation of RISC-V based systems in gem5 a comparison to real hardware is done. For this purpose, two RISC-V boards are available. These platforms are 32 bit architectures, which is also more common in embedded systems. Therefore, these systems also have to be supported in gem5. The goal is to support the rv32imc mode in gem5 to do an accuracy evaluation by comparing the simulation to real hardware.

To summarise the requirements on the simulation framework, Table 4.1 gives an overview. In the left column all tasks are listed, that require implementations in gem5. This regards the Full-System mode for RISC-V, the availability of UART and timer devices. The RISC-V ISA implementation in gem5 needs to support interrupts and exceptions and furthermore it needs to support the simulation of 32 bit architectures. In the right column of Table 4.1 the requirements on the interface for custom extensions are listed. There, it must be possible to conveniently define new instructions and registers. These definitions needs to be parsed and information for further actions needs to be retrieved. Finally, the RISC-V toolchain and the gem5 simulator needs to be extended using the retrieved information.

In the following concepts and implementation of the just described requirements are introduced.

4.3 Extending Full-System Support in gem5

This section is structured as follows. First, the enabling of the RISC-V FS mode in gem5 is described. Then, comments on peripheral devices, namely Timer and UART, are made. After that, the concept and implementation of exceptions and interrupt support is described. In the end, enabling of 32 Bit mode is introduced.

4.3.1 Enable Full-System mode

Enabling FS support gives the opportunity to enrich the defined system with peripheral devices including UART and timers. As currently gem5 supports the RISC-V ISA only in SE mode, the first step is to achieve the simulation of a very rudimentary bare-metal application.

To simulate a system in gem5 a system configuration script is required, as described in section 2.2.1. In this script a system object needs to be created containing all system-level information. Furthermore, devices including memories and peripherals can only be added to such a system object. For the RISC-V ISA two system definitions exists, the classes *RiscvSystem* and *RiscvLinuxSystem*. Both classes have currently no functionality implemented and are only stubs.

The missing functionality to initialise a system needs to be implemented in order to enable the Full-System mode. Thereby, the possibility shall be obtained, to have support for other types of systems, like Linux, too. As the class *RiscvLinuxSystem* inherits from the *RiscvSystem* class, functionality that is only valid for bare-metal specific systems cannot be implemented there. Therefore, a new class *BareMetalSystem* is introduced, that inherits from the *RiscvSystem* class. The class structure for this concept is shown in Figure 4.4. Similar to the *LinuxRiscvSystem* the *BareMetalRiscvSystem* describes a type of a system. Functionality needed to instantiate this system type and is only valid for this type of system is implemented there. Further functionality that all RISC-V system types have in common are implemented in the *RiscvSystem* class. With checking of the system type, complex implementations including address translation in the translation lookaside buffer (TLB) can be skipped for bare-metal systems. Also, developers can implement desired features without influencing bare-metal systems, when implementing the support for other systems.

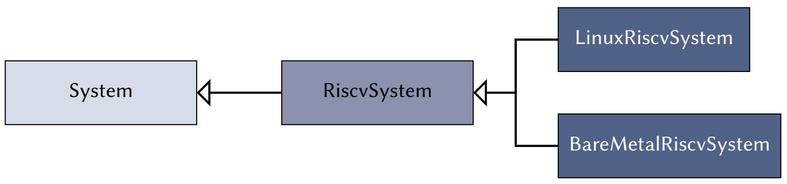

Figure 4.4: Inharitance structure of the different system classes.

Having defined a system, its components need to be initialised during the starting phase of the simulation. Therefore, a *python* script instantiates all components, that are defined in the configuration file. The sequence diagram in Figure 4.5 gives a rough overview about the processes during the system initialisation. Important for the description of the implementation are the system object and the CPU object. Hence, the sequence diagram only considers calls by the *python* script to these classes. The first task of the script is to instantiate all objects, that are defined in the system configuration script. After that it calls the *init()* method and the *initState()* of each of them. Within the method *init()* of the CPU class an ISA the function *initCPU()* is called. This function has the task to take actions hardware would do on a reset according to the corresponding ISA specification. As described earlier, the RISC-V system classes are only stubs as well as the RISC-V specific *initCPU()* method. Therefore, these functions have to be implemented to enable the start-up of a RISC-V FS simulation.

The function *initCPU()* is implemented according to the description in the privileged architecture specification [81]. During the method *initState()*, the binary, that is executed on the simulated system, is loaded into the simulated memory. This includes clearing all interrupts and set the program counter to a implementation defined reset vector. It has to be mentioned that interrupts are currently not supported at all. However, appropriate functions for clearing them are called to keep the implementation consistent for later changes.

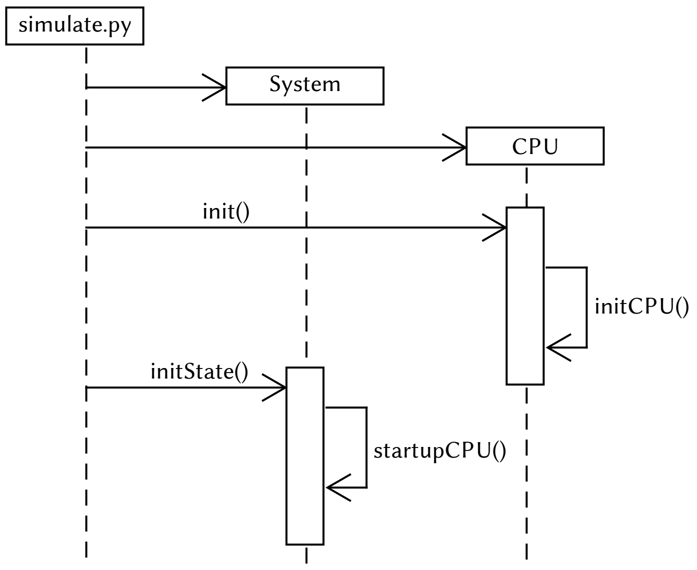

Figure 4.5: Sequence diagram of the initialisation of the system class and the CPU class.

With these implementations, a simple binary containing start-up code and a rudimentary main application can be executed and simulated. This can be done on a most basic system, as shown in Figure 4.6. With that, rudimentary FS support for RISC-V is enabled.

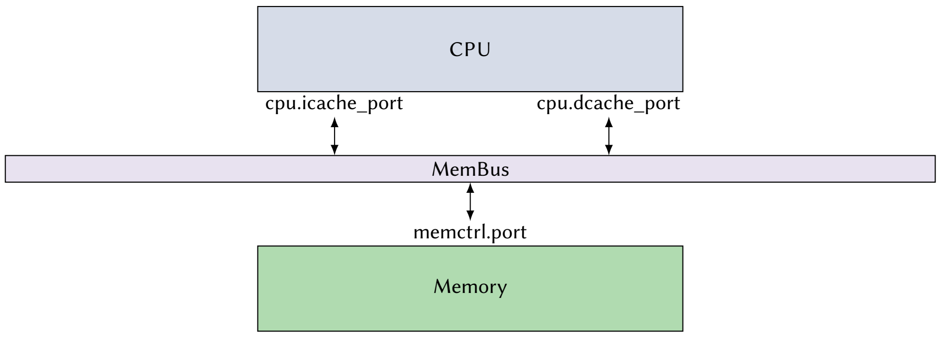

Figure 4.6: The simplest system, that is able to execute a binary.

4.3.2 Support of Peripheral Devices

The second requirement on the gem5 simulator is to support peripheral devices. In the following, usage of the UART device and concept and implementation of a timer device are described.

UART

Implementations for UART devices already exists in gem5. Therefore, it is only necessary to add this device to the system and connect the ports. This leads to a memory mapped UART register, that can be accessed by writing data directly to the configured address.

Gem5 offers an implementation of a terminal, that can be used as communication partner for the UART device. This terminal is connected to a telnet port. With that, it can be accessed from outside of the simulation with establishing a telnet connection from a local console to a certain port. This way, characters send via UART to the terminal can be read by a user.

The resulting system definition for gem5 to simulate can be seen in Figure 4.7. The UART device is connected to the memory bus, so it shares the same access delays. With the possibility to send characters, an example of communicating with the outside world was implemented. With that, simulation of the use-case presented in Section 4.1.1 is possible.

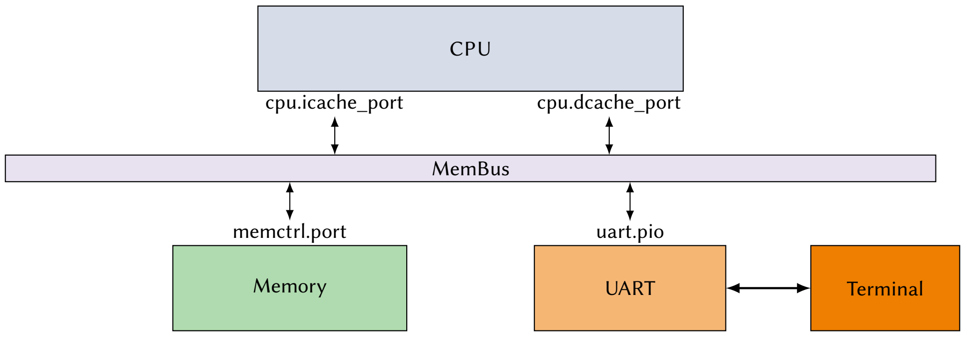

Figure 4.7: The system definition in gem5 with a UART device connected to a terminal.

Timer

Besides other system wide timers, the privileged architecture specification [81] defines a CPU timer, which is only accessible by the corresponding CPU. This timer has two memory mapped registers, namely *mtime* and *mtimecmp*, which are 64 Bit wide, as seen in Figure 4.8. The register *mtime* is a real-time counter, that runs at constant speed. To compare the time with a value the register *mtimecmp* is used. A timer interrupt is posted, if the value in *mtime* is equal or greater than the value in the *mtimecmp* register. [81]

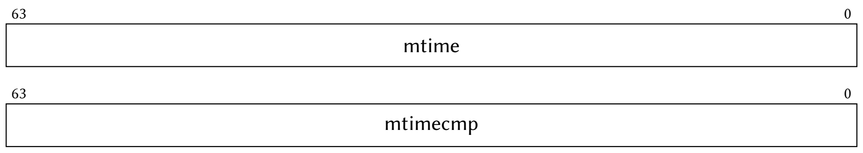

Figure 4.8: The memory mapped *mtime* and *mtimecmp* registers, as in [81].

A device is implemented that models memory mapped registers. For that, the abstract class *PioDevice* is used as base class from which the implementation inherits. This class offers templates for *read()* and *write()* functions and attributes for a start address and an address range to which the device shall respond. The gem5 framework automatically assigns memory accesses within address range to the *read()* and *write()* methods of the corresponding object. To reflect the affiliation to one specific CPU, the device class gets a pointer to its belonging CPU. With this approach, the timer device can post an interrupt directly to the CPU once the alarm goes off. This set-up modelled in UML can be seen in Figure 4.9.

To model the behaviour of a timer according to the specification, an interrupt

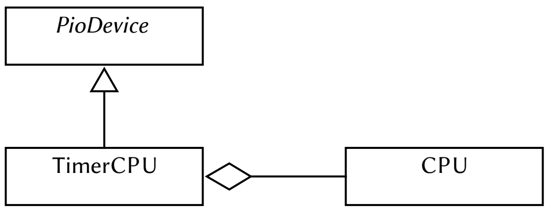

Figure 4.9: Rudimentary class diagram of the TimerCPU class.

needs to be posted when the *mtime* register and the *mtimecmp* register have the same value. Establishing a check every cycle would be inefficient, as it would slow down the simulation. Instead, an event is scheduled for the time when the registers *mtimecmp* and *mtime* would have the same value. For that purpose, gem5 offers an event manager that is inherited by the CPU-Timer object implicitly. In the scheduled function an interrupt is posted to the CPU indicating a timer alert.

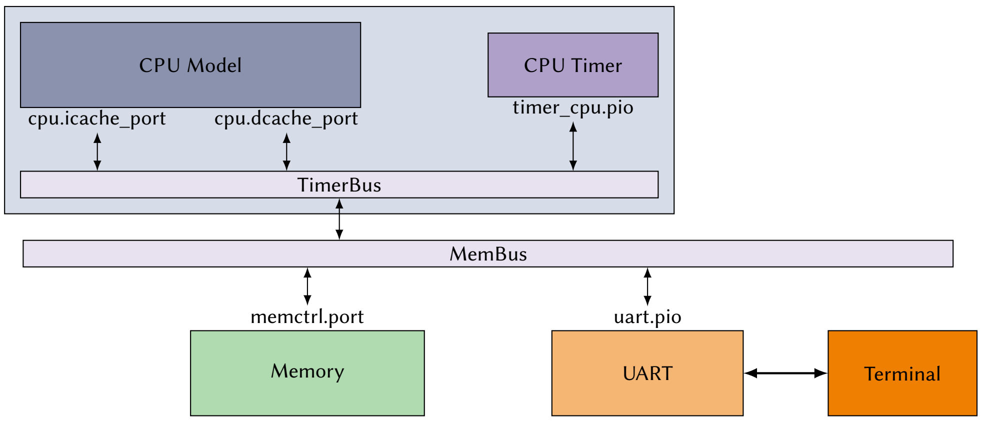

Figure 4.10: The system definition in gem5 with peripheral devices.

These implementations enable the simulation of the execution of time-controlled tasks. With the system in Figure 4.10, an application has the possibility to set up a timer, that posts an interrupt to the CPU, if the alarm goes off. A separate bus, the *TimerBus*, is used because the timer device can only be connected to the system using its *PIO* port. Furthermore, with a separate bus access latencies can be adjusted and the timer does not have to share the same bus latency with the connected memory and UART device.

4.3.3 Support for Interrupts and Exceptions

One requirement for full system simulation of embedded system in gem5 is the support for interrupts and exceptions. Currently, there is no support for neither of them implemented. In the following, the phrases *exceptions* and *interrupts* are used according to the RISC-V manual [80]. Interrupts occur asynchronously to the program flow, while exceptions are associated with an instruction. On occurrence of both, control is given to a *trap handler.*

This subsection shows the implementation of support for exceptions, interrupts and the return from a trap handler in gem5. This enables the usage of the CPU-Timer in simulation.

Exceptions

Support for exceptions is needed as the access of the timer registers shall only be possible in machine mode, according to the privileged architecture specification. Therefore, user mode applications have to make a request to the supporting execution environment. This is done with a system call, which is the *ecall* instruction in RISC-V. Furthermore, on occurrence of an exception, its exact cause is written to the *mcause* register. After that, the privilege level is changed and the current program counter is saved. Than, the next instruction set to the address of the trap vector, which is specified in the *mtvec* register. The bit fields of the *mcause* and *mtvec* registers can be seen in Figure 4.11 and Figure 4.12.

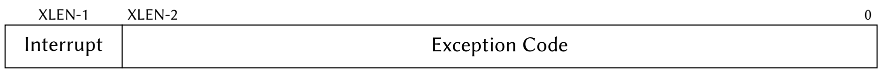

Figure 4.11: The mcause special register, as in [81].

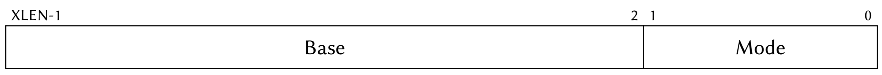

Figure 4.12: The mtvec special register, as in [81].

Currently, gem5 is able to decode the *ecall* instruction, but functionality is completely missing. That means, the alternations to special registers have to be implemented. For that, two different approaches are possible, which are explained in the following.

The first approach directly describes the changes in the functional description of the *ecall* instruction and system call is interpreted by the CPU like every other instruction. A sequence diagram for this approach can be seen in Figure 4.13. When decoded the bit stream as a *ecall* instruction, the CPU executes its functionality. With that, all actions, that needs to be done on a system call, are processed.

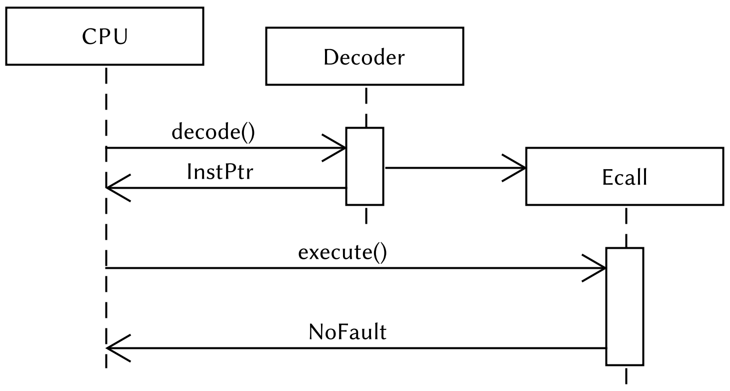

Figure 4.13: First approach for implementing system calls in gem5.

The disadvantage of this concept is, that this implementation is directly associated with the *ecall* instruction, which generates only one specific type of exception. In the RISC-V ISA all kinds of traps require nearly the same actions on special registers. The only difference is the exact cause. Gathering these actions can save code duplicates and with implementing system call exceptions, other exceptions become available as well.

The mechanism in gem5, that offers the possibility to join common code for traps, is the concept of faults. Having again a look at Figure 4.13 it can be seen, that the *execute()* function has a return value of type *FaultBase*. The definition of this class is shown in Listing 4.1. It shows that every fault has to implement an *invoke()* method, which is called by the CPU, if an instruction returned a fault. Having a closer look at the definition of the *invoke()* function, it can be seen, that one of the arguments is of type *ThreadContext*. This class provides an interface for accessing a state inside a CPU, including reading and writing special registers and altering the program counter. With that possibilities, the *invoke()* method provides the necessary functionality, to map the RISC-V specification to gem5.

Faults are used whithin the second approach to implement a common base class for all traps. There, common code for altering the special registers is shared among all kind of traps. Taking the class *FaultBase* as a base, the resulting class diagram can be seen in Figure 4.14. Functionality for altering the program counter and special registers is implemented in the class *RiscvFault*. The exact cause of

```
class FaultBase
{
  public:
    virtual const char* name() const = 0;
    virtual void invoke(ThreadContext * tc, const
        StaticInstPtr &inst =
                        StaticInst::nullStaticInstPtr);
};
```

Listing 4.1: Class definition of the base class for all faults.

the trap is than determined by the instantiated class, for example *SyscallFault.*

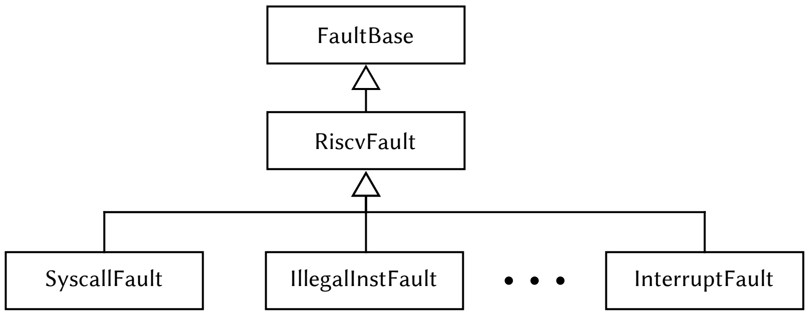

Figure 4.14: Class inheritance for RISC-V faults.

A sequence diagram that illustrates the second approach is shown in Figure 4.15. It starts with the decoding of the *ecall* instruction and the decoder returns a pointer to the CPU to the just decoded instruction. The *execute()* method, that every instruction in gem5 has, is now called. This time, this method implements no functionality at all, but a *SyscallFault* is instantiated and returned to the CPU. Then the CPU invokes this fault, which leads to the described actions. After that, the program counter points to the trap vector and execution of the program continues there.

For implementation of support for exceptions in gem5 the second approach is chosen. This approach offers the possibility to reuse code when implementing support for interrupts. Furthermore, it is conform to the implementation of exception support for other ISAs in the gem5 project.

Interrupts

In the following the concept and implementation for the support of asynchronous traps in gem5 is described.

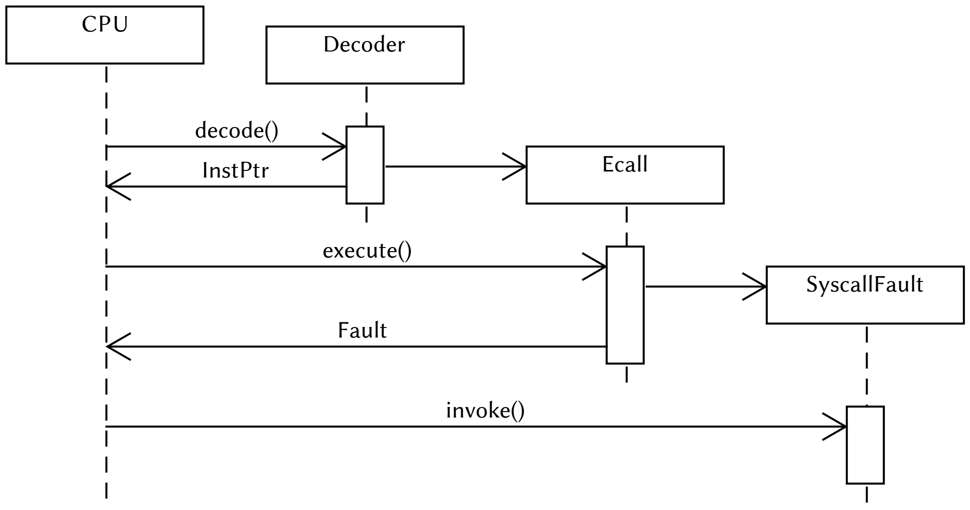

Figure 4.15: Sequence diagram for executing a system call.

The key class for interrupt support in gem5 is the interrupt controller, that every CPU model has. This controller is ISA dependent and offers necessary functions to get the information about a new interrupt and let the CPU know about it. The relationship between the CPU and the interrupts controller is expressed in Figure 4.16. In this class diagram can be seen, that the interrupt controller is a class named *Interrupts* Also pictured are the most important functions of this class. With the function *post()* the CPU can inform the controller that an interrupt occurred and needs to be stored. Every simulated cycle, the CPU checks with the method *checkInterrupts()*, if in the meantime an interrupt was posted. If so, it asks the interrupt controller with the function *getInterrupt()* to return the actual interrupt. Therein, the controller has the task to select the highest prioritised interrupt, if concurrent interrupts are pending. A closer look into the class diagram in Figure 4.16 tells, that this function returns a fault. So this concept of an interrupt controller in gem5 matches perfectly with the approach to model all traps as faults. After that, the method *updateIntrInfo()* is called, where machine registers are updated. Currently, the interrupt controller in the RISC-V ISA is just a stub. Therefore, enabling interrupt support requires the implementation of the functionality.

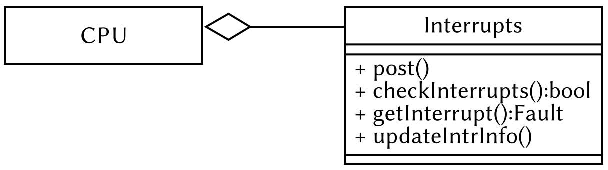

Figure 4.16: Relationship between the CPU class and the Interrupt class.

The implementation of the functionality of the interrupt controller is now described by reference to the procedure of processing an interrupt generated by the CPU-Timer. The procedure of the generation of an interrupt in gem5 can be seen in the sequence diagram in Figure 4.16. The timer alarm goes off asynchronously, which causes the CPU-Timer to call the *postInterrupt()* method of the CPU. Hence, the core immediately calls the *post()* method of the interrupt controller. This method just stores the occurring interrupt locally. At the beginning of the next simulated tick, the CPU checks, if in the meantime an interrupt was posted, what happened indeed. This is done with the function *checkInterrupts()*. Therein, the interrupt controller checks, if the previously posted interrupt is masked or not. Therefore, the special registers *mstatus* and *mie* are analysed to see if interrupts are globally and locally enabled. For this example, it is assumed, that both checks are true, so that the *checkInterrupts()* function returns also true. Thus, the CPU asks for the specific interrupt and expects a fault type, as described in Listing 4.1 as answer. An *InterruptFault* is generated, that holds the exact interrupt reason. After this whole process, the CPU now has a pointer to a class, that represents the interrupt. Next, the special register *mip*, that holds information about all pending interrupts, must be updated. This is done in the function *updateIntrInfo()*. After that, the *InterruptFault* can be invoked by the CPU. This leads to the same actions as if an exception occurs. That means, that the privilege level is changed to machine mode, the register *mcause* is updated with the reason of the interrupt and the program counter is set to the trap vector.

Concluding, this implementations offers the possibility for the CPU-Timer to post an interrupt to the CPU. This causes the interruption of the current program flow and the system is set to the trap vector, where the interrupt is handled.

Return from a Trap

To return from a trap, RISC-V defines three instructions, generally referred to as *xret*. Depending on the privilege level *mret*, *sret* or *uret* are used to restore the program state as it was before the occurrence of the trap. This instruction is

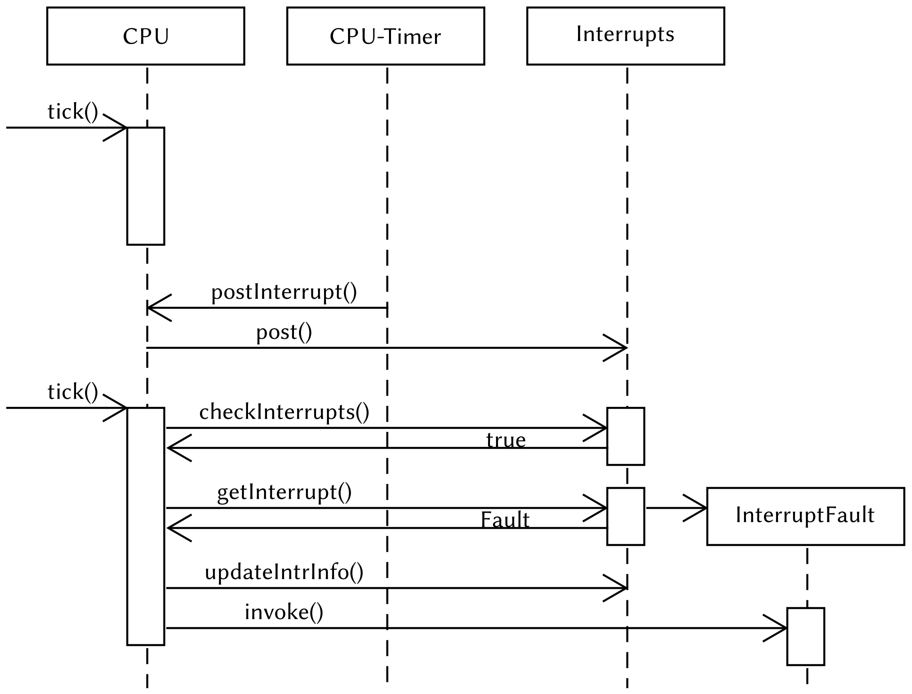

Figure 4.17: Procedure of posting an interrupt to the CPU until it finally is executed.

usually used at the end of the trap handler of RISC-V kernels. In the following, implementation of the *xret* instructions is described.

Currently, the RISC-V decoder in gem5 is able to decode *mret*, *sret* and *uret* instructions. However, they lack all functionality and have to be implemented.

The mechanism used to implement the *xret* instructions is the possibility to alter the CPU state within classes representing instructions. The alternations to special registers are implemented according to the privileged architecture specification of RISC-V. This includes the enabling of interrupts, the change of the privilege level to user mode and setting to the program counter stored by occurrence of the trap.

A sequence diagram of the decoding and execution of an *xret* instruction can be seen in Figure 4.18. When decoding the instruction, the decoder creates an instance of the class representing the instructions. A pointer to this instance is returned to the CPU, which then calls the *execute()* method of this class. Therein, the just described alternations to the special registers are done.

Having implementations for the *xret* instructions in place, gem5 is now able to simulate a CPU-Timer. It is possible to access the memory mapped timer registers from a user application with system calls. If the timer alarm goes off,

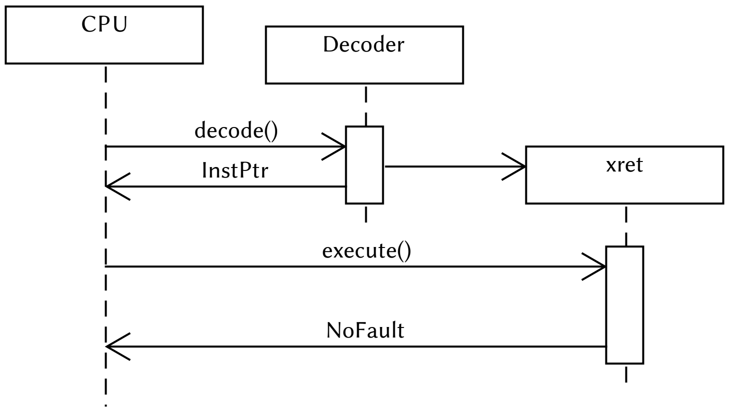

Figure 4.18: Sequence diagram for executing a return from interrupt.

an interrupt is posted by gem5 to the simulated CPU. Thereby, the current program flow is interrupted and the program counter is saved. After handling the interrupt, the *xret* instructions makes it possible to continue execution on the saved program counter. Simulation of a system according to the use-case presented in Section 4.1.2 is now possible.

4.3.4 32 Bit support

The last requirement for the gem5 simulator is the possibility to simulate 32 Bit systems. Currently, gem5 only supports 64 Bit architectures. Therefore, the register widths, instruction semantics and operational codes are all fit for the 64 Bit specification of the RISC-V ISA.

Though most of the instructions do the same, supporting the 32 bit mode in gem5 requires a second decoder tree. To reason the need for this, the definition of the 64bit *add* instruction, which can be seen in Listing 4.2, is taken as an example. Important in the instruction definition in Listing 4.2 is the operand type *_sd* for the output register *Rd* and the input registers *Rs1* and *Rs2*. The operand type defines the integer type in the generated *c++* code, which in consequence lead to the bitwidth of the operand registers. It becomes clear, that these types have to be adapted for 32 Bit architectures to generate 32 Bit wide registers.

The *c++* code that is omitted by the gem5 ISA parser is defined by *formats*. There, the omitted class declaration corresponding to the instruction and the

```
0x0: ROp::add({{
    Rd_sd = Rs1_sd + Rs2_sd;
}});
```

Listing 4.2: Add instruction definition for 64 bit architectures.

```
class Add : public RegOp
{
  public:
    /// Constructor.
    Add(MachInst machInst);
    Fault execute(ExecContext *, Trace::InstRecord *) const
        override;
    using RegOp::generateDisassembly;
};
```

Listing 4.3: Generated class for the 64 bit add instructions.

execute() method are defined. An example for the usage of *formats* can be seen in Listing 4.2. There, the format *ROp* tells the gem5 ISA parser, that the instruction *add* is a register-register operation. The reason for the necessity to adapt all formats is the omitted declaration of the *c++* class defined by the formats. There, the name of the defined instruction is taken as the class name. This is clarified by the comparison of the definition of an instruction in Listing 4.2 and its generated class in Listing 4.3. The *add* instruction generates a class named *Add*. As the instructions in 32 and 64 bit architectures have the same names, two decoder files lead to two generated classes with the same name. With the introduction of new formats for 32 bit instructions, this problem is solved. Every generated *c++* class, that corresponds to a rv32 instruction, is placed inside a namespace. Hence, no duplicated names occur.

With the implementation two different decode trees are available. Gem5 must chose the right decode tree for the application. Therefore, the simulator needs to know the architecture type of the system. Gem5 already provides features to analyse binaries, which are extended. With using this utilities, the system gets the information about the architecture type and stores it. This information now has to be passed to the decoder. Moreover, the bit width of the architecture is chosen based on the binary. This offers the possibility to implement a switching of architectures during simulation.

The class representing the decoder and its most important functions can be seen in Listing 4.4. With the function *moreBytes()* the CPU gives data to the

```
class Decoder
{
 protected:
    //The extended machine instruction being generated
    ExtMachInst emi;

  public:
    StaticInstPtr decodeInst(ExtMachInst mach_inst);
    void moreBytes(const PCState &pc, Addr fetchPC,
        MachInst inst);
};
```

Listing 4.4: Most important functions of the decoder class.

decoder. More precise, this function is used to fill the protected member *emi* with the bytes of the instruction, that shall be decoded. The bytes of this instruction are delivered with the argument *inst.* More important information, like alignment, address and compression of this instruction are stored within the program counter object *pc* of type *PCState.* The member variable *emi* is than given to the method *decodeInst()*, which is the generated by the ISA parser, as described in Section 2.2.1. That means, the only way to get the information about the bit width of the architecture into the decoder is within the *PCState.* Than it has to be stored inside the *ExtMachInst.* This type is 64 bit wide and every instruction in RISC-V is only 32 bit wide, independent of the underlying architecture. Therefore, the upper 32 bit of *ExtMachInst* can be used to store additional information. To do so, the type is changed from plain *uint64_t* to a bit field definition, which is possible in gem5 through predefined macros.

Within the *decodeInst()* function the architecture bit field is analysed. Depending on that, the corresponding decode tree is chosen. The implementation of the ISA description file, that is parsed by gem5 to such a decoder is shown in Listing 4.5. Behind these include statements, two separate ISA description files define the rv64 and rv32 instructions.

With this implementation it is possible to simulate 32 bit architectures, additional to already existing possibility to simulate 64 bit architectures. Together with the previously described concepts and changes to gem5 it fulfils all requirements that were introduced in Section 4.2.

```
decode ARCH default Unknown::unknown() {
    0:
    ##include "rv64.isa"
    1:
    ##include "rv32.isa"
}
```

Listing 4.5: Choice of the right decoder depending on the architecture bit field within the ExtMachInst.

4.4 Interface for Custom Extensions

The second group of tasks regards the simulation of custom extensions. In summary, it is required to have an interface where users can define instructions and registers. These definitions shall be parsed to retrieve necessary information for extensions of gem5 and the RISC-V toolchain. Therefore, a project is created, the *RISC-V Custom Extension Parser.* This section introduces the concept and implementation of crucial parts of it.

4.4.1 The RISC-V Extension Parser

In the following, an overview of the project itself is given. Additionally, the integration in build process of gem5 is described.

Overview of the RISC-V Extension Parser Project

The requirements on design of the *RISC-V Custom Extension Parser* from Table 4.1 in Section 4.2 are visualised in Figure 4.19. This diagram shows the steps from defining an extension until the simulation of a binary using the defined extension is possible.

Starting point is the definition of an extension. This definition is given to the *Extension Parser.* As the name indicates, it has the task to parse the defined models and to gather information about instructions and registers. These information are processed to generate toolchain patches offering the possibility to access custom instructions with assembly code. Additionally, a plug-in for the gem5 simulator is created containing decoding and timing information of added instructions. Moreover, with this plug-in custom registers can be used in gem5.

Figure 4.20 shows the structure of the project, most important folders and files. Folders are illustrated in light blue, files in white. Due to the amount of different file types, the files are sorted by these. Central part is the *python* folder

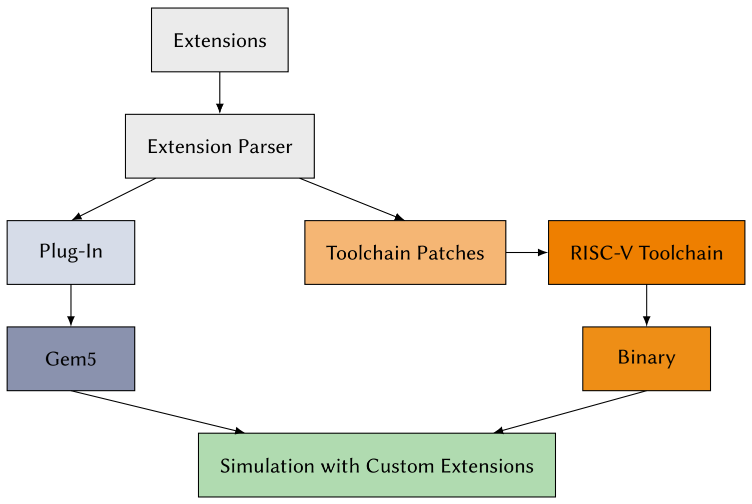

Figure 4.19: The RISC-V custom extension parser in context of toolchain and gem5.

containing scripts with all needed functionality and an extensive unit testing framework. Located in this folder is the file *extensionparser.py*. It is the entry point for adding custom extensions to the toolchain and the simulator. With this script the parsing of the models is started. This process can be configured with the file *config.ini*, which is checked in the beginning of the parsing process. For example, a configuration option is to define the folder, in which the custom extensions are, which is by default the folder *extensions*. When processing the custom extensions, the *riscv-opcodes* [67] project is used to conveniently generate matches and masks for the custom instructions. These are then used by the patches for the RISC-V toolchain. Additionally, a plug-in for gem5 is generated as well. While generating this plug-in, different files have to be created. These are placed in the *build* folder. Together with the files in the *include* and *src* directory, these are used to compile the plug-in. For a convenient use, the *SConscipt* file enables integration in the gem5 build process.

In the following a closer look into the class structure of the *python* project is taken,as it is the central part. Implementing the functionality in *python* enables convenient parsing of *c++* files. Moreover, due to its object orientation, the language is sufficient to express the different targets, namely RISC-V toolchain and the gem5 simulator, and the sources, namely custom extensions, as objects. A rough overview of the class hierarchy is given in Figure 4.21. The *ExtensionParser* creates a parser, which processes the defined instructions and registers. With the gained information, it creates an object *Compiler* containing functions to extend header files, source files and generate intrinsic instructions. Furthermore, the

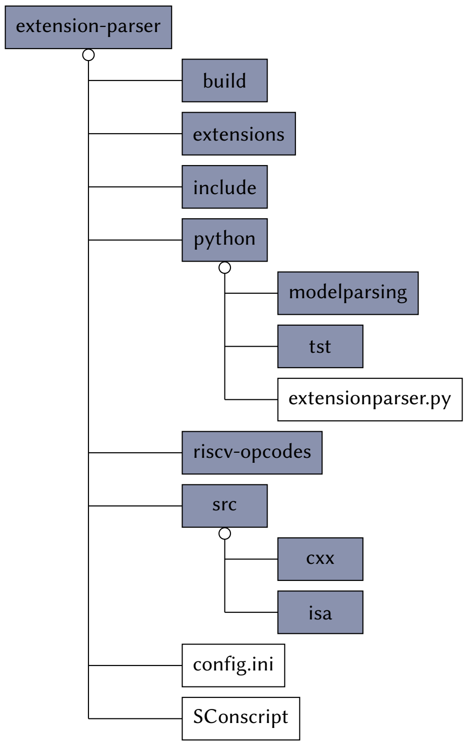

Figure 4.20: The extension parser package and its project structure.

parser instantiates an object, that represents the gem5 simulator, which is called *Gem5*. This class implements functions to generate a custom decoder and let gem5 generate appropriate *c++* files with help of the ISA parser. Moreover this class provides a function to process the timing information of instructions and creates custom functional unit timings, used by the CPU model in gem5.

Integration in gem5 and Build Process

The processing of custom extensions is integrated in the gem5 build process. Therefore, the whole project is placed in the folder *ext* of the gem5 project. Therein, less-common external packages needed to build gem5 are located. This hierarchy is expressed in Figure 4.22. Gem5 uses the build tool *scons*, which is based on *python* and enables the invocation and execution of *python* code. When building, the *SConstruct* file of gem5 searches the external folder for *SConscript* files. In the *SConscript* file of the *RISC-V Extension Parser* project the *python* class *ExtensionParser* is created and called. The placement of these files can also be seen in Figure 4.22. The *SConstruct* file is placed in the top level folder of the

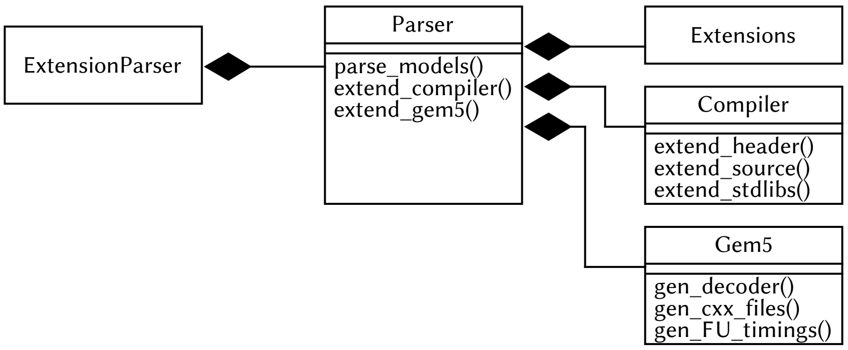

Figure 4.21: Class diagram of the extension parser.

```
[DEFAULT]
MODELPATH = ~/projects/gem5_cc/ext/riscv-custom-extension/
    extensions
TOOLCHAIN = ~/projects/riscv-gnu-toolchain
```

Listing 4.6: Content of the configuration file for the extension parser.

gem5 project and within the extension parser project the *SConscript* file is found.

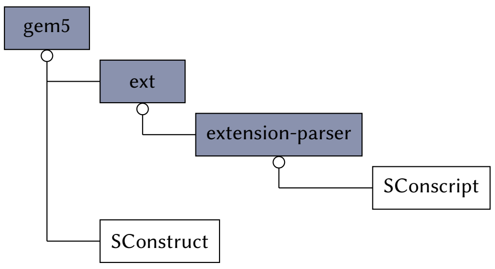

Figure 4.22: Folder structure of the extension parser project within gem5.

To specify certain paths, like the path of the RISC-V toolchain and the folder, where custom extensions are defined, a configuration file is placed in the project and analysed be the *ExtensionParser* class. This file is of type *.ini* and an example can be seen in Listing 4.6.

Having stored the configurable parameters, the extension parser instantiates an object of type *Parser*. Than the three methods of the *Parser* class, which can be seen in Figure 4.21 are invoked. When done, the *SConscript* takes the generated files to build a static library. After that, gem5 can be linked against it.

In the following, all broached topics are described in more detail, giving an fine-grained insight of the processing of custom extensions until they can be used in RISC-V binaries and simulated in gem5.

4.4.2 Defining Custom Extensions

This section describes how users can define custom extensions. Furthermore the choice of the language in which these definitions happen is explained.

The requirements on configurable parameters of custom extensions are described in the following. When defining an instruction, users want to assign a custom name and of course define a certain semantic. Furthermore, the possibility to chose an instruction format, as in Figure 2.1, shall be supported. This gives the user additional flexibility for the usage of the custom instruction. Besides defining the instruction itself, users shall be able to state further information, namely operational codes and the cycles that the instruction shall consume. Another requirement is the possibility to define custom registers by their name and index. Furthermore, these registers need to be easily accessible. For that, functions for reading and writing needs to be provided.

For the definition of custom extensions, a language is required. Due to various advantages *c++* was chosen. First, parsers for *c++* are already available. Because many information has to be retrieved from the model definition, it has to be parsed. To reduce the effort to write a parser for a custom description language, it is more efficient to use a language where parsers are already available. Another advantage is the feasibility of *c++*. Having defined a model, that is already compilable, hardware designer can include the function in testbenches. There, the model serves as a reference to test the hardware implementation of the custom extension against it and compare the produced outputs [26]. Last but not least, the gem5 decoder uses pure *c++* to describe the semantics of a function. Having obtained the definition of the custom instruction, it can be directly included in the gem5 ISA description without translation. That, again, saves a lot of effort.

One custom instruction is defined by one *c++* function. Furthermore, the base format of the instruction is defined by the name of the *c++* function arguments. Thereby, the base instruction formats *R-Type* and *I-Type* are supported.

As stated before, operational codes and cycle counts also needs to be ascertained. these information are added directly in the *c++* file, where the instruction is defined Rather than having an additional configuration file in yet another language, this approach ensures that every information is in place and not distributed over a lot different files.

```
uint8_t cycles = 2;     // cycle count for this instruction
uint8_t opc    = 0x02;  // opc, 5 bits
uint8_t funct3 = 0x00;  // funct3, 3 bits
uint8_t funct7 = 0x00;  // funct7, 7 bits

void foo(uint32_t Rd, uint32_t Rs1, uint32_t Rs2)
{
    // function body
}
```

Listing 4.7: Example definition of a custom instruction.

```
#include <cstdint>

#define c0 0x800  // read and write
#define c1 0xcc0  // read only

uint32_t READ_CUSTOM_REG(uint32_t reg);
void WRITE_CUSTOM_REG(uint32_t reg, uint32_t val);
```

Listing 4.8: Example definition of custom registers.

Listing 4.7 illustrates the definition of a custom instruction. Here, a R-Type instruction named *foo* is defined consuming two cycles and has certain operational codes.

Custom registers are defined in a *c++*-header file with *#define* statements. It is done this way to keep a consistency between defining registers and instructions. Listing 4.8 shows an example of defining two custom registers. The index conventions follow the official RISC-V specification in [81]. There is defined that non-standard read-only user register start with the index *0xcc0* and readable and writeable user registers with index *0x800*. The functions *READ_CUSTOM_REG()* and *WRITE_CUSTOM_REG()* are place-holders whose implementation is later generated by the backend.

4.4.3 Parsing of the Extension Models

The *Parser* class has the task to analyse the *c++* models for custom extensions. This includes parsing the custom instructions as well as the custom register file. Therefore, a closer look into the class *Parser* of the class diagram in Figure 4.21 is taken. A more detailed extract is shown in Figure 4.23. There, the class *Model* can be seen. One instance of this class represents a model of a custom instruction, which means one function defined in a *c++* file. In this class functions exist, to compile and analyse a model. The *Parser* also creates an *Extensions* object, wherein all custom instructions and registers are gathered. Additionally, the registers file is analysed and depending on the definition of custom registers an object *Registers* is instantiated, that saves all names and indexes.

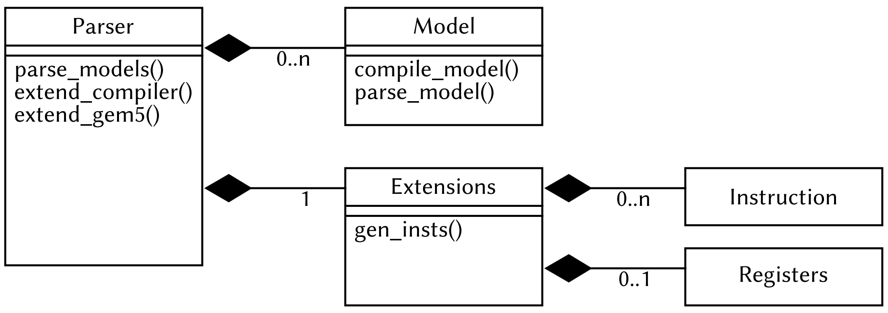

Figure 4.23: Class diagram of the Parser class and its properties.

To illustrate this process, Figure 4.24 shows the parsing of a single model of an instruction until all necessary information are in place to generate toolchain patches and build the plug-in for gem5. For every model, that is found in the configured extensions folder an instance of the class *Model* is generated. This is done in the *parse_models()* method. Therein, every *c++* file is at first compiled, which ensures a semantically correct model. After that, the model is parsed within the *parse_model()* function. The tool used for parsing and analysing is the *libclang* project [31]. This project provides a *c*-language API for accessing the abstract syntax tree (AST) of a parsed source file. *Libclang* has *python* bindings as well, which enables the use of this project from *python* scripts. These bindings provide functions and methods to traverse the nodes of the AST representing the *c++* model of the custom extensions [70]. The nodes are analysed and all necessary information are retrieved. Having parsed the models, an *Extensions* object is generated. With the method *gen_insts() Instruction* objects are generated for all models. Therein, information obtained by parsing are transformed and extended with masks and matches for each instruction. The matches and masks are obtained by invoking the *riscv-opcodes* project [67] and are used by the toolchain patches. Masks and matches are required for defining assembly instructions in the toolchain. Their purpose is to separate instructions. With the *mask* variable parts of the instruction word, such as input registers, output registers and immediate values, are masked. With that, only the opcodes remain. If these remaining bit pattern is equal to the *match*, the corresponding instruction is found.

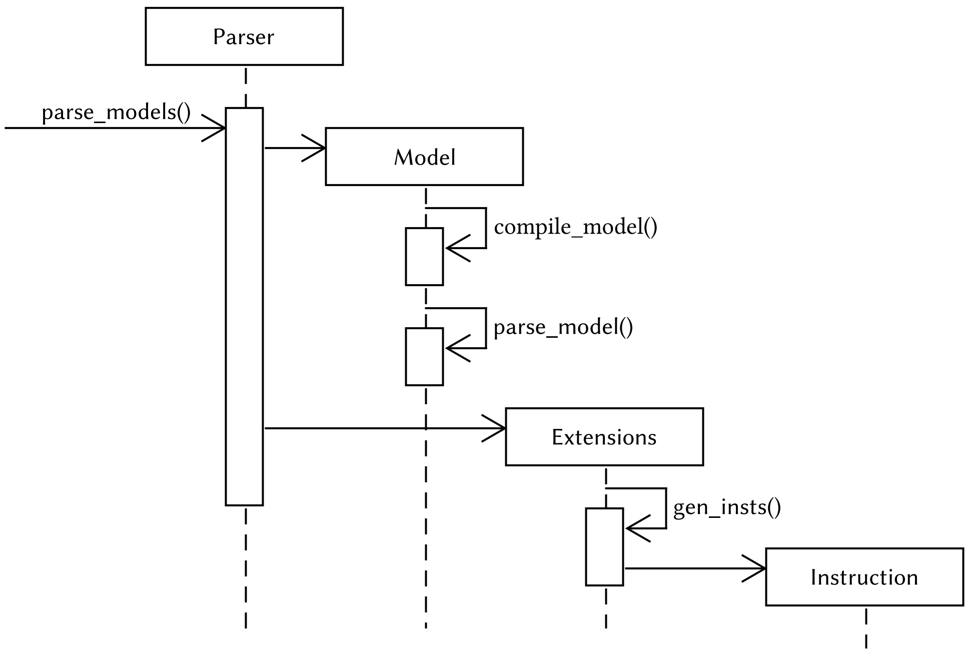

Figure 4.24: Sequence diagram of the parser processing information from a given model to generate a class instance, that represents the custom instruction.

4.4.4 Implementation of the gem5 Plug-In

To be able to simulate applications using custom extensions, gem5 must know about additional instructions and registers. That requires gem5 to be able to decode added instructions and execute them with respect to the configured timing information. Furthermore, the simulator must know about additional registers.

Having again a look into the class diagram of the *python* project, the class that is responsible for the described tasks is called *Gem5*. As seen in Figure 4.25, this class has access to the recently parsed information via the *Extensions* object. The processing of the parsed information to enable simulation of custom instructions and custom registers is described in the following.

Custom Instructions

The gem5 decoder must be able to recognise the newly added operational codes and needs know the functional semantics of the custom instructions. Patching

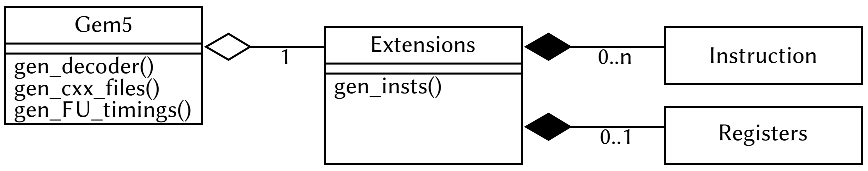

Figure 4.25: Class diagram of the class Gem5 and its aggregations.

the decoder description file in gem5 with these information would lead to the rebuild of a lot of gem5 components. To fasten this process and prevent gem5 components from being recompiled, another approach is chosen. This approach is to generate a stand-alone decoding function, which can be achieved by invoking the gem5 ISA parser. After that, a library is built that gem5 calls as a fall-back, if the standard decoder is not able to recognize the operational code.

Being able to decode the instruction is not enough for efficient and realistic design of integrated circuits. It is crucial to consider timing information about the consumed cycles of each function. Therefore the plug-in must enrich the MinorCPU of gem5 with timing information devices for each custom instruction. As described in Section 2.2.3, altering the pool of timing devices for a functional unit of the CPU can be done during execution time. That means, this change to gem5 by the plug-in does not result in recompiling the simulator.

To summarise this, Figure 4.26 gives an overview of components in the plug-in. The functional unit timings of the MinorCPU will use additional custom timings, that are generated for each instruction. Moreover the ISA decoder has a fall-back to the generated custom decoder. The generation of the custom decoder and the generation of timing information for custom instructions is now described in more detail.

Generation of the Custom Decoder. The goal is to build a library that contains a decoder for the custom instructions. This is achieved by treating the custom instructions as whole instruction set and providing an ISA description that gem5 is able to parse within its ISA parser. This parser is implemented in *python* and as a result it can be called conveniently from the *RISC-V Custom Extension* project. The gem5 ISA parser takes one single file of type *.isa* as input, the *main.isa*. In this file, all other *.isa* files, such as bit field definitions, operand types and instruction formats, has to be included. The file *main.isa* of the RISC-V custom instruction decoder reuses most required *.isa* descriptions from the RISC-

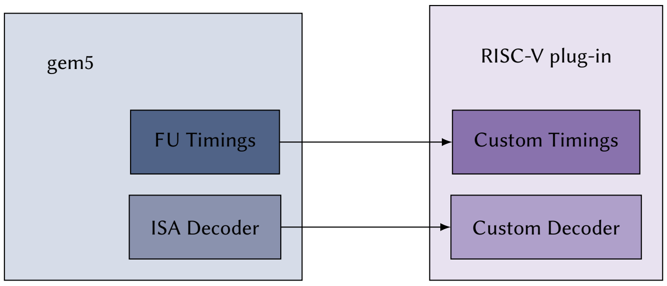

Figure 4.26: Gem5 components interacting with corresponding components in the RISC-V custom extensions library.

```
StaticInstPtr
decodeCustomInst(ExtMachInst mach_inst)
{
    RiscvcustomISA::Decoder decoder;
    return decoder.decodeInst(mach_inst);
}
```

Listing 4.9: Extract of the source file custom_decoder.cc.

V ISA definition in gem5. Furthermore, a decode tree must be included in this file. This tree is auto-generated by class *Gem5* of the *python* project. Furthermore, this class also invokes the gem5 ISA parser, which leads to the generation of *c++* files. These files contain the definition of the decoder for custom instructions.

Gem5 accesses the custom decoder with the function *decodeCustomInst()*, whose definition can be seen in Listing 4.9. This *c++* function instantiates the custom decoder and calls the method *decodeInst()*. It has to be mentioned that the definition of this method originates from the gem5 *isa* parser.

Figure 4.27 illustrates the usage of the custom decoder by the gem5 RISC-V decoder. In this sequence diagram, the decoding of an example custom instruction, named *customInst*, is shown. First, the RISC-V decoder in gem5 is called for decoding the bit stream. As the bit stream represents a custom instruction, the decoder will not find the corresponding instruction. But instead of throwing and unknown instruction exception, it invokes the function *decodeCustomInst()* and gives control to the *RISC-V Custom Extension* plug-in. This function instantiates the *CustomDecoder* and calls its method *decodeInst()*. There, the corresponding instruction to the operational code is found. As every instruction is represented by a *c++* class, the object *CustomInst* is instantiated. Finally, a pointer to this class is then returned to the *RISC-V Decoder*.

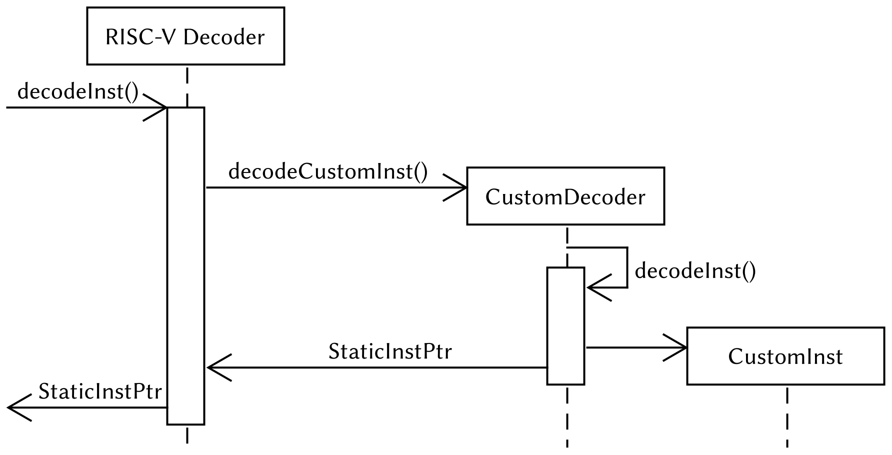

Figure 4.27: Sequence diagram of the process of decoding a custom instruction in gem5.

This approach leads to an execution of custom instructions within the execution pipeline of the CPU model.

Timing Information for Custom Instructions. With the just described implementation, gem5 is able to recognize a custom instruction and execute it. What is missing is the information about how many cycles a custom instruction consumes. Currently, every custom instruction is assumed to take only one cycle. This is not practicable, as instructions, that accumulate a lot of logical operations, would limit the clock frequency. Due to the limit of speed, that logical components have, the clock frequency could only be as fast, as it takes to process the input in the custom extension. Furthermore, some calculations may require several steps and therefore need to be pipelined. With splitting the calculation over multiple cycles, the user is able to define the clock frequency more flexible. As seen in Listing 4.7, a user can define the cycles, that a custom instruction needs, until its result is available. To model this behaviour in simulation, the in-order CPU of gem5 offers the possibility to assign set of timing objects to a functional unit, as described in Section 2.2.3. Therefore, a functional unit is created, that is responsible for all custom instructions. In this unit, the possibility exists to create a set of *MinorFUTiming* objects. This timing objects inherit from the *SimObject* class, which means, that this object can be configured from *python* scripts. Therein, matches and masks can be defined to assign the timing information to one specific instruction. Only of the instruction logically anded with the mask is equal to the match, the additional timing information are applied. This mechanism is used to generate timing objects for each custom instruction

```
class MinorFUTimingFoo(MinorFUTiming):
    description = 'CustomFoo'
    match = 0xb
    mask = 0xfe00707f
    extraCommitLat = 1
```

Listing 4.10: Generated timing object for a custom instruction.

using the matches and masks that are generated while the custom extensions are parsed. Such a generated timing object can be seen in Listing 4.10. This object is generated from the instruction model in Listing 4.7. The value stored in *extraCommitLat* is added to the commit latency specified in the functional unit. In the *python* project, the *Gem5* class creates one functional unit timing object for each custom instruction that has a commit latency greater than 1 cycle.

Custom Registers

Apart from custom instructions, gem5 must also be able to recognize custom registers. In contrast to other general purpose registers, a user can be sure, that values in custom registers are not written back to memory. This allows a fast access of the content in a defined amount of time. The simulator must be able to store the values, that are written to these registers, and retrieve this value, if an application reads the custom register. Two different approaches are evaluated in the following.

Memory-mapped Custom Registers. The first approach is to make custom registers memory mapped. This allows to access a register by writing or reading from the address it is mapped to. To simulate memory mapped custom registers in gem5, a concept similar to the CPU-Timer is implemented. The corresponding class diagram can be seen in Figure 4.28. There, the class *CustomReg* inherits from the abstract class *PioDevice*, which gives the possibility to be attached to a port and receive packets from the memory backend of gem5. To process these packets *read()* and *write()* functions have to be overwritten. These functions access the *RegisterMap*, where the addresses and values of all custom registers are maintained.

The resulting definition of the system to be simulated in gem5 can be seen in Figure 4.29. Similar to the CPU-Timer, the *CustomRegisters* object is attached to the *TimerBus*, which ensures delay-free access. With this approach, RISC-V binaries can access a defined custom register with writing to or reading from the

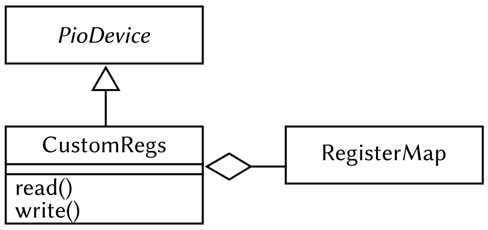

Figure 4.28: Class diagram the concept for memory mapped custom registers.

configured address. In the simulation, this triggers the invocation of the memory backend of gem5.

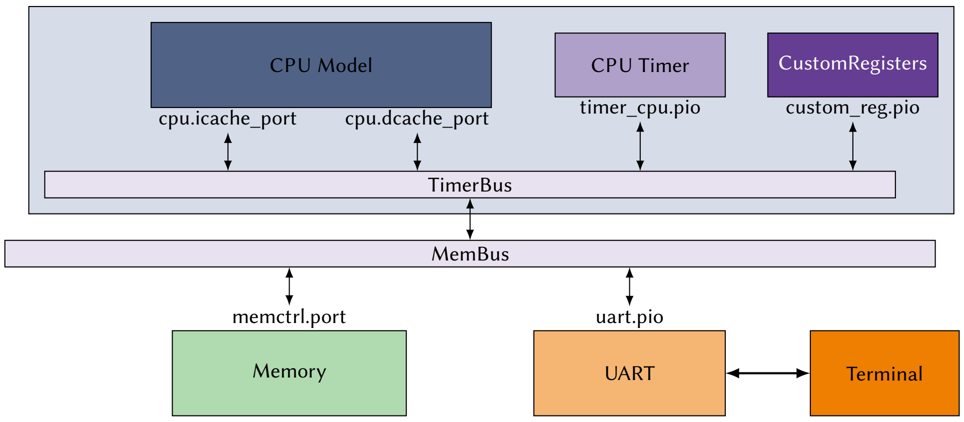

Figure 4.29: System definition in gem5 with memory mapped custom registers.

The disadvantage of this approach is the realisation of accesses of custom registers within custom instruction. Thereby, the memory backend has to be invoked to access the simulation object representing custom registers. This requires the creation of a memory packet and waiting until gem5 has processed it. All this logic has to be implemented in the custom instruction, where the register value is needed. This approach is therefore not practicable.

Index-mapped Custom Registers. The second approach is to make custom registers index-mapped. In the privileged architecture specification of the RISC-V ISA [81], indexes for special registers are defined. The indexes from $0x800$ to $0x8FF$ are reserved for non-standard registers and $0xCC0$ to $0xCFF$ is reserved for non-standard read-only registers.

In gem5 the class *Isa* represents the underlying ISA itself. This class already provides functions to access special registers by its indexes, which can be seen in

```
#define READ_CUSTOM_REG(reg) ({uint32_t val; val = xc->
    readMiscReg(reg); val;})

#define WRITE_CUSTOM_REG(reg, val) (xc->setMiscReg(reg,val)
    )
```

Listing 4.11: Generated macros for read and write access to a custom register.

the class diagram in Figure 4.30. The functions *readMiscReg()* and *setMiscReg()* access the defined special registers, called *MiscReg*. To represent custom registers, a map is added in the class *Isa*, here illustrated as class *CustomReg*. Access of this map is implemented in the functions *setMiscReg()* and *readMiscReg()*. Furthermore, Figure 4.30 shows that the class *Isa* inherits from the abstract class *SimObject*. As stated earlier, this implies that the class has an equivalent *python* class and their parameters can be configured from *python* in run time. Therefore the entries of the custom register map can be defined without recompiling gem5.

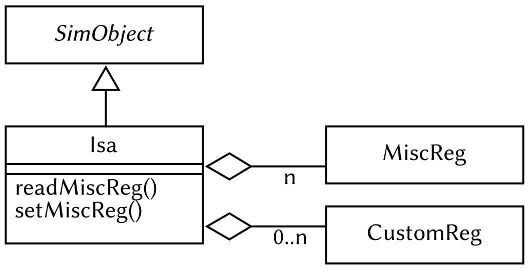

Figure 4.30: Class diagram the concept for index mapped custom registers.

The realisation of accesses of custom registers within custom instructions was identified as disadvantage of the approach of memory mapped registers. With index mapped registers, this use-case can be implemented conveniently. Within instructions, registers can be accessed by just calling the *readMiscReg()* and *setMiscReg()* functions of the class *Isa* due to the inheritance of the *ExecContext* variable. It offers the possibility to access function of the class *Isa*. As the function body from the model definition is directly taken as function description in the custom decoder, a mechanism has to be provided to ensure convenient definition of instructions accessing custom registers, without knowing about gem5 mechanics. To solve this problem, a header file containing macros is generated. These macros can be seen in Listing 4.11.

Listing 4.12 shows an example of a custom instruction accessing custom registers. This code hides the gem5 mechanisms and the generated macros, as in

```
#include "registers.hh"

void cust_inst( uint32_t Rd, uint32_t Rs1, uint32_t Rs2)
{
    uint32_t var = READ_CUSTOM_REG(c0);

    // some calculations

    WRITE_CUSTOM_REG(c0, var);
}
```

Listing 4.12: Definition of a custom instruction using custom registers.

Listing 4.11, are invoked when gem5 executes the functionality of the custom instruction.

4.4.5 Generation of Toolchain Patches

Having parsed the *c++* models of custom extensions and having built a plug-in for gem5, the last missing piece is to ensure convenient development of RISC-V based applications using custom instructions and custom registers. For that, the *Parser* class of the *RISC-V Custom Extension* project creates a member of type *Compiler*, which can be seen in Figure 4.21. In the following, patching the toolchain to use custom instructions in assembly code is described. After that it is shown, how custom registers can be accessed conveniently in RISC-V based binaries.

Custom Instructions

Though, extending the RISC-V ISA with custom instructions is supported, extending the toolchain requires the patching of different files yet. However, the user shall be able to call its new instruction with assembly code. Extracting the required information from custom extensions to patch the files in the toolchain automatically is already done. What is left is adding these information in the right format The files, that have to be patched, can be seen in Figure 4.31. In the header file *riscv-opc.h*, where the mask and match to every instruction is defined. The source file *riscv-opc.c* contains an array, where all instructions are defined. Again, an entry in this array is a struct, that combines the name of an instruction, its operands and its masks and matches. Hence, adding instructions there makes them available in assembly code.

In the implementation, the *python* class providing functions to patch the source

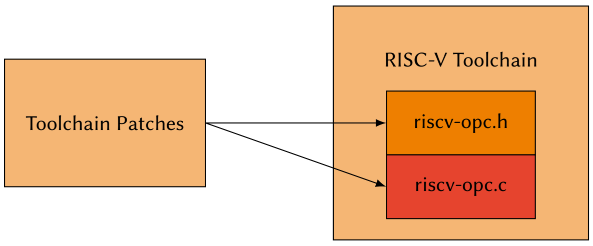

Figure 4.31: The RISCV custom extension plug-in.

and the header files is the class *Compiler*. A class diagram of it can be seen in Figure 4.32. Similar to the class *Gem5*, that is responsible for the plug-in generation, this class also has access to the generated instructions and registers via the *Extensions* object.

The functions *extend_header()* is responsible for patching *riscv-opc.h*. First, it creates a copy of the original file. Than it creates a custom header file and patches *riscv-opc.h* to include this custom header. Therein, the generated masks and matches for the custom instructions are defined. Additionally, this implementation allows a convenient restoration of the original, unchanged header file.

With the function *extend_source* the source file *riscv-opc.c* is patched. Again, the original file is copied and stored. Than, the array, that contains the structs, as described earlier, is extended. For every custom instruction, that is defined, this function generates the appropriate struct definition and adds it in the file.

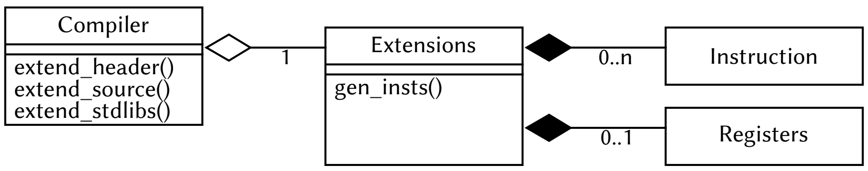

Figure 4.32: Diagram of the class Compiler.

Custom Registers

For the use of custom registers in user programs, no toolchain files have to be patched. Although the *riscv-opcodes* project, that is used to generate matches and masks, also generates define statements for special registers, these are hard coded into the project. Therefore, adding the index and names of custom registers there would again require to patch the project itself. Instead another approach is chosen. As soon as at least one custom register is defined, two additional custom instructions are generated, that enable read and write access to custom registers

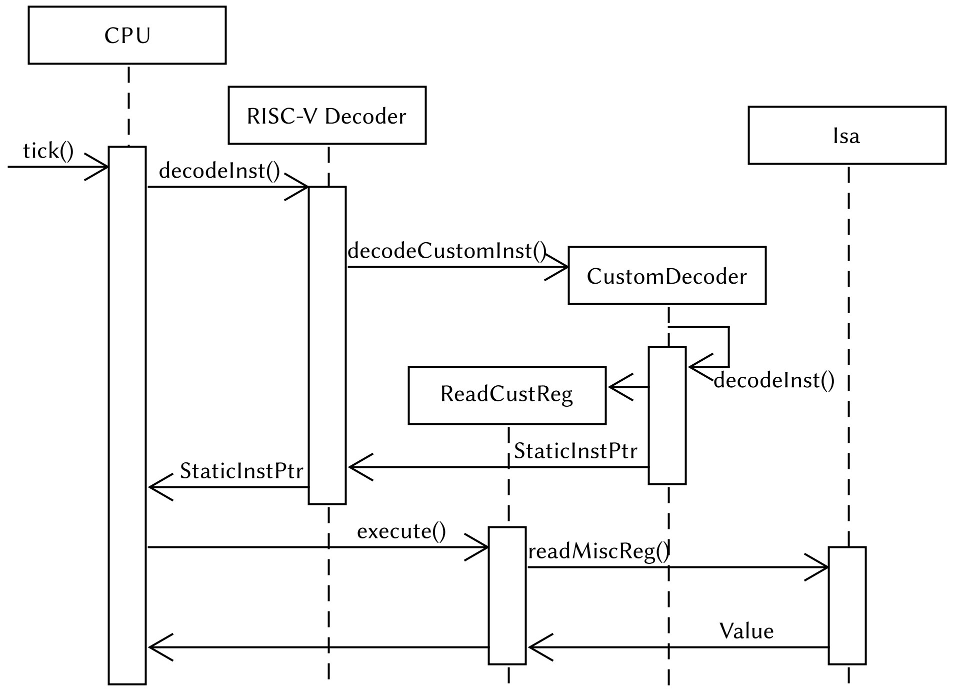

Figure 4.33: Sequence diagram showing the decoding and execution of a read of a custom register.

The process for simulating the access of a custom register in gem5 is shown in Figure 4.33. It shows the interaction of the generated plug-in and gem5 components to access a custom register. First, the CPU invokes the RISC-V decoder to decode a bit stream. As it finds no matching instruction to the given operational code, it calls the fall-back function to the custom decoder. The *CustomDecoder* finds the instruction *ReadCustReg* and instantiates it. The pointer to this class is then given to the CPU, which invokes the *execute()* function. Therein, the function *readMiscReg()* of class *Isa* is called. The *Isa* looks in its custom register map, that is shown in the class diagram in Figure 4.30 for the called index and returns its value to the CPU.

```c
#include <riscvintr.h>

int main()
{
    // set special register
    WRITE_CUSTOM_REG(c0, 5);
    int a;
    a = READ_CUSTOM_REG(c0);
    return 0;
}
```

Listing 4.13: Include of the header for the generated RISC-V intrinsic instructions.

RISC-V Intrinsic Instructions

To access custom instructions, a user has the possibility to use assembly code. Hence, this would require the use of in-line assembly in *c* code, which is a rather ungainly coding style. Therefore, intrinsic instructions, that wrap the assembly code are generated. These are defined in a header file, namely *riscvintr.h*, that is placed in the *include* directory inside the compiled toolchain. Having again a look into the class diagram in Figure 4.32, the function *extend_stdlibs()* is responsible for the creation and placement of this file. This way, the header can be included just like standard libraries. The Listing 4.13 shows the use of generated intrinsic instructions for reading and writing custom registers.

5 Verification and Evaluation

This chapter describes the verification of the previously described implementations. Furthermore, this chapter evaluates the accuracy of gem5 by comparing the simulator to real RISC-V hardware. Finally, the usage of the RISC-V Extension Parser for accelerating algorithms by using custom extensions is demonstrated with an example.

5.1 Verification of the Implementation

Tests were developed, in order to verify the implementations and ensure its functional correctness. In this section, these tests are described. This covers the verification of the correctness of the RISC-V FS mode and the RISC-V 32 bit mode in gem5. Furthermore, this section describes the testing of the interface for custom extensions.

5.1.1 Verification of the RISC-V Full-System Mode

A custom RISC-V binary holding the current feature status is developed simultaneously to the implementations in the RISC-V FS mode in gem5. This application is simulated in gem5, and the output is observed and interpreted as a test result. This way, it is checked whether an application behaves as expected and if gem5 is able to simulate the use-cases which were described in Section 4.1. Therefore, the binary prints characters via UART, sets up a timer, waits for the interrupt, and finally uses defined custom instructions and custom registers.

The binary has more than just a testing purpose. Due to the sparse documentation about the FS mode and crucial functions, it also has the purpose of discovering missing function implementations in gem5.

The advantage of writing an own kernel instead of using existing kernels is its customisability for the current features implemented in gem5. In context of this thesis, the term *kernel* refers to a small implemented library containing start-up code and trap handling. With every implemented feature, the bootloader implementation is extended as well.

To be able to conveniently maintain the files and add new implementations, an own project is created. In the following, the project structure and the build process are introduced. After that, important features of the kernel implementation are highlighted.

Project Structure and Build Process

With implementing more features to gem5 and therefore to the testing binary, the number of files grew. Hence, it is appropriate to create an independent project for the RISC-V binary. To ease adding of new files and features, *CMake* is chosen as the build tool.

While building, the kernel containing the start-up code is compiled into a static library. Every user application links against this library in order to use this kernel. Therefore, many different applications for test purposes can be developed, conveniently added to the project and integrated in the build process.

Additionally, a configuration script is added to the project, which automates the build process. With this script it can be configured, if the binary shall be compiled for a 32 bit oder 64 bit architecture. Accordingly, the right compiler is chosen and a build directory is created. After that, the generation of the *Makefile* with the tool *cmake* is done and the build is started.

Features of the Custom Kernel

Along with the implementations in gem5, the custom kernel grows and is extended with more and more features. These features reflect the use-cases for embedded systems, that are presented in Section 4.1.

The kernel comes with its own linker script, wherein, the memory origins and sizes are defined. Furthermore, input sections are assigned to output sections, which again are assigned to the defined memories. Using a custom linker script gives the opportunity to adapt the memory definitions to the settings of the hardware systems.

Important features of the kernel regard the loading of the main application and handling traps. On start-up, the kernel zero-fills every integer register, enables interrupts globally and writes the trap vector in the *mtvec* register. After that, it initialises the stack pointer and jumps to the main application. Next, the kernel only is active on occurrence of a trap. Responsible for this task is the trap handler. It saves the current CPU state by writing the inputs of all registers to a dedicated save area. Then, it analyses the cause of the exception and handles it accordingly. After that, the register state is restored and the interrupted execution continued. Both, the entry code and the trap handler, are implemented in assembly and

```c
inline int32_t
add(int32_t rs1, int32_t rs2)
{
    int32_t rd = 0;
    ROP("add", rd, rs1, rs2);
    return rd;
}
```

Listing 5.1: Wrapper for the add assembly instruction.

verified against the implementation in the RISC-V Berkeley Bootloader [82] and in the RISC-V Linux port [66].

Other important features that the custom kernel provides, enable the use of peripheral devices, namely access of the UART device and setting up a timer. To redirect all prints to the UART register, the *_write()* function is overwritten. Routines in *libc* use this function for output to all files including *stdout*. Additionally, a library is provided allowing applications to set up a timer. The user mode application calls the provided functions, wherein a system call is done. This call is handled by the kernel in machine mode, which allows access to the memory mapped timer registers. As physical memory protection is not implemented in gem5, the access of the timer register happens in machine mode to represent a realistic RISC-V system.

Additionally, the custom kernel is able to support both, 64 bit and 32 bit architectures. To use the same custom kernel for tests systems of both bit widths, architecture specific load and store instructions were replaced by macros. This saves effort and prevents code duplication. The macro is then replaced with the real instruction depending on the chosen toolchain. With that, it is possible to use the same kernel for 32 bit and 64 bit RISC-V applications.

5.1.2 Verification of the RISC-V 32 Bit Mode

The enabling of the 32 Bit address mode described in Section 4.3.4 added the possibility to decode the RV32IMC assembly instructions. Therefore, the implementation of the functional semantics of every instruction needs to be tested.

In the test program, every instruction of the 32 bit standard instruction set and the extensions for multiplication,division and compressed instructions is tested. To do so, *c* functions are implemented as wrappers around the corresponding inline assembly instruction. This can be seen in Listing 5.1 for the *add* instruction. The actual test can be seen in Listing 5.2. The expected result and the assembly function, that shall be tested, is given to a function evaluating the output.

```
// add
expect<int32_t>(16638, [](){return I::add(0x3fff, 0xff);}, "
    add");
expect<int32_t>(-1,
    [](){return I::add(0x7fffffff, 0x80000000);},
    "add, overflow");
```

Listing 5.2: Test of the 32 bit add instruction.

The test is based on the already existing instruction test for the 64 bit RISC-V instructions, which exists as test program in the gem5 repository. This 32 bit instruction test is run in FS mode. The results are printed on the gem5 terminal, which is connected to the UART device.

5.1.3 Verification of the Custom Extension Parser

Apart from the implementations in gem5, the interface for custom extensions developed to a rather complex piece of software with numerous features. As described in Section 4.4.2, the interface, where users can define instructions and registers, is restricted in terms of variable names and required information. Therefore, the parser script has to consider many different error cases and do extensive sanity checks.

Unit tests verify the correctness of every function in every class. The overall functioning is verified with test binaries, that either uses custom registers or custom instructions. These applications uses the previously described custom kernel and are simulated in gem5. As test result are print statements evaluated.

5.2 Accuracy Evaluation of gem5

With the introduced tests it is attested that the simulator is able to run typical embedded applications. To be used for virtual prototyping or architectural studies, more than just pure functionality is important. Another vital attribute is the accuracy of the simulator. Only if systems are simulated with sufficient precision, assumptions on their architectural behaviour can be made.

In this section, an accuracy evaluation of gem5 is done. The simulator is compared against two hardware systems, the *SiFive HiFive1* board, a RISC-V core on real silicon, and the *CommSolid Samara* FPGA board. Both architectures differ quite severely, which gives the opportunity to analyse the adaptivity to different system architectures of gem5. The architecture of these RISC-V systems is first introduced before the benchmarks, being used to measure the accuracy, are explained. After that the results are shown and evaluated.

5.2.1 Architectures of the RISC-V Systems

To be able to rebuild the RISC-V systems in gem5, a closer look on their architecture is given. Each hardware is analysed and then rebuild in python scripts with components provided by gem5. The attributes of these components are than adapted to the constraints of the systems in order to achieve an accurate model.

CommSolid PicoRV32

The RISC-V CPU on the FGPA board by CommSolid [18] is based on the open source project PicoRV32 [84]. It is a size optimised RISC-V core implementing the 32 bit instruction set with the multiplication, division, and compressed instructions standard extensions. Additionally, the core is highly configurable to save size by implementing the multiplier and divider in hardware. Although the multiplier is able to deliver the result in one cycle, the divider is a rather simple implementation processing one bit per cycle. Therefore, its execution latency is 32 cycles. For the purpose of being small, the core does not implement a pipeline and has no caches. It directly fetches data and instruction from SRAM. This fetch lasts two cycles. One cycle is consumed for loading the operands from the registers and one more cycle is consumed for write-back. Hence, the latency of an instruction is four cycles plus the latency of the execute state. For example, the execute state of the ALU consumes one cycle. Therefore, an ALU operation, such as *add* or *sll*, is expected to consume five cycles. As the multiplier is an own hardware unit, the core needs one additional cycle to invoke this unit. For this reason, the result of a multiplication is expected to have a six cycle latency.

The core is connected to an advanced high-performance bus (AHB), just like the SRAM. The equivalent system definition is similar to the most basic system that can be defined in gem5. This system is shown in Figure 2.3. Just a CPU, a memory bus and a memory are needed. The memory bus models the AHB and the memory simulates the SRAM.

The assigned memory for the RISC-V core is only 64 KiB. Therefore, only small programs can be executed and they must not be linked against the standard library, as it consumes too much memory. This causes the *printf* statement not to be available for information retrieval. Hence, the cycle and instruction counts are stored in variables and read by executing the program with the debugger *GDB*. After processing the benchmark, an *ebreak* instruction is executed, which causes the *GDB* to halt. By inspecting the local variables, the cycle and instruction counts can be retrieved and the benchmark evaluated.

SiFive HiFive1

The second available system is the SiFive HiFive1 board, whose architecture is introduced in the following. It is an open source RISC-V development kit, that uses the industry's first commercially available RISC-V SoC *SiFive FE310-G000* [37, 38]. It implements a 32 bit RISC-V CPU with the standard extensions for multiplication and division, atomic and compressed instructions. A schematic layout of the core is shown in Figure 5.1. On the left side, the core complex and its functional units can be seen, whereas on the ride hand side interfaces and memories are shown.

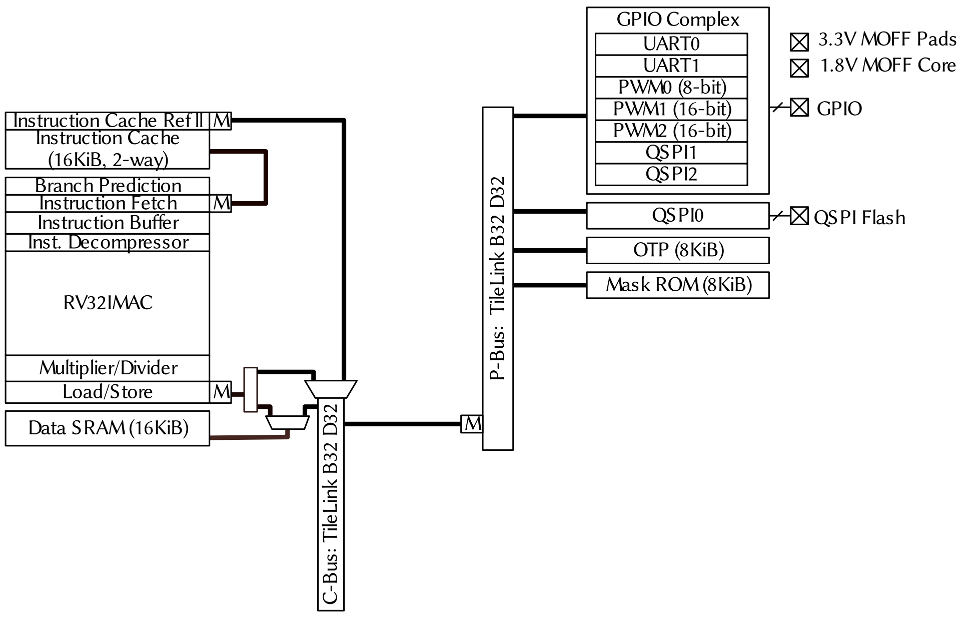

Figure 5.1: Core complex of the RISC-V CPU on the SiFive HiFive1, according to [37].

The core implements a single issue in-order CPU with a five stage pipeline, that leads to a one cycle result latency for most instructions. It is capable of running with up to 320 MHz clock frequency and is configured to run with 256 MHz. In the architectural overview in Figure 5.1 it can be seen, that the RISC-V CPU has one load-store unit, one arithmetical unit and a hardware multiplier and divider. The multiplier has a five cycle result latency in the pipeline and the divider is implemented with an early out, which means its latency varies

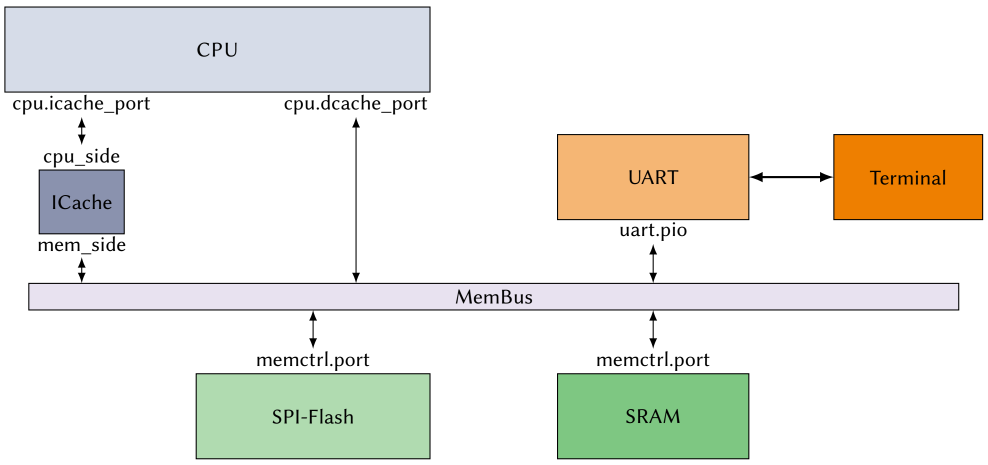

Figure 5.2: The system definition in gem5, that reflects the SiFive HiFive1 board.

between two and 33 cycles. Furthermore, the CPU has a 16 KiB 2-way associative instruction cache being connected to the instruction fetch unit. Another part of the core is a GPIO complex with UART ports, which gives the possibility to use print statements in RISC-V applications [37]. The UART device is also used to print the measured values within the benchmarks.

The board has several memories for different purposes. User applications are stored in the SPI-Flash memory, which has a size of 512 MB. Typical for flash memories are high access latencies compared to other memory types. The latency of the SPI-Flash memory of this board is not documented in the manual, and therefore has to be measured. Data in the binary are loaded on start-up by the bootloader into the *Data SRAM*, which has two cycles latency for reading full words. This memory has a size of 16 KiB and prevents data from being written back into the slow flash memory. The mentioned bootloader is stored in *ROM*, that can be seen on the right hand side of Figure 5.1.

For this board, an open source software development kit is available, that offers the possibility to compile programs and flash them into the memory of the board. Additionally, debug possibilities are available for the convenient development of RISC-V applications. [38]

To simulate this system, a corresponding system definition was created in gem5. This definition can be seen in Figure 5.2. Noticeable is the central memory bus, being used to connect the memories and the UART device with the CPU and cache. This is required, because gem5 uses this bus to load the data segments of the binary into the memories. The *dcache_port* of the CPU is connected directly to the memory bus. As the architecture of the *SiFive FE310-G000* also has no data cache, this sufficiently reflects the real board. A *SimpleMemory* is used to model the SRAM and a second *SimpleMemory* is used to model the SPI-Flash. For the core the *MinorCPU* model is chosen, as it also implements an in-order pipeline.

5.2.2 Approach for the Accuracy Evaluation

The approach to evaluate the accuracy is to run benchmarks on hardware and in the simulator. The execution times of these benchmarks are compared. For that, the same, unmodified binary is run on the hardware board and on the calibrated system model created in gem5.

The execution time of the benchmark cannot be retrieved directly, but needs to be calculated. For that, Equation (5.1) is used, where n is the number of consumed cycles and f is the frequency. The frequency of the systems is known, while the number of consumed cycles can be measured. For that, the control register *mcycle* is read before and after the execution of the benchmark. Calculating the difference of both values results in the number of cycles, that the benchmark consumed.

$$t = \frac{n}{f} \tag{5.1}$$

The second value being directly measured in the binary is the instruction count. To retrieve this count, the control register *minstret* is accessed. Similar to the cycle count, the instruction count is also measured before and after execution of the benchmark. The consumed instructions are then calculated by subtracting both values.

Both values are used to compare simulation to hardware. If the instruction count is the same on both, it is ensured that the same amount of instructions are run. This indicates that the simulator executes the same instructions as the hardware boards. Comparing the calculated execution time indicates how much time the program consumed. Differences in this time are then analysed and explained.

5.2.3 Calibration of the System Models

Gem5 offers pre-defined objects for every component on a chip for system definition. These objects can be configured through parameters. A calibration is done to adapt parameters of the objects in the modelled system to the characteristics

Table 5.1: Cycles per assembler instruction for the PicoRV32 core on the CommSolid Samara board.

PicoRV32	add	c.add	mul	div	lw	sw
Cycles per Instruction	5.04	4.06	6.03	39.42	7.99	8.33

of the hardware boards. For that, assembler instructions are benchmarked to evaluate the parameters of the functional units and memories in gem5.

Every functional unit of the CPU is calibrated by measuring the used cycles per instruction for an assembly instruction executed by the corresponding unit. For that, the *add* instruction the ALU is utilised. In each of the hardware systems, the multiplier and divider are self-reliant functional units. To evaluate this, the instructions *mul* and *div* are benchmarked. Furthermore, the two boards support the standard extension for compressed instructions. To measure the impact of compressed instructions on the instruction fetch, the *c.add*, which is the compressed add instruction, is benchmarked as well. Finally, the load and store units and the memory latencies need to be calibrated. Therefore, *lw* and *sw*, the instructions for loading and storing a word, are analysed. Each of the mentioned assembler instructions are run 1000 times and the consumed instructions and cycles are measured. This is done in multiple rounds to measure the impact of the instruction cache in the *SiFive HiFive1* core. With the obtained values, the consumed cycles per assembly instruction are calculated. Based on this value the corresponding functional unit in gem5 is calibrated.

CommSolid PicoRV32

In the following, the calibration of the system definition reflecting the *CommSolid* FPGA board is described. The challenge of this architecture is to find the most feasible CPU model in gem5. Therefore, the *PicoRV32* core is benchmarked by the execution of assembly instructions, where the results can be seen in Table 5.1. The measured cycles per assembly instructions are shown. It can be seen, that the compressed instruction consumes one cycle less than the *add* instruction. This results from the CPU fetching two instructions with one word. Therefore, in every second instruction the fetch state can be skipped, which saves two cycles. The *lw* and *sw* both consume around 8 cycles per instruction, whereas the load is slightly faster.

Based on these results, two gem5 CPU models were chosen and evaluated to find the most feasible. Thereby, the challenge is to model the non-pipelined

Table 5.2: Calibrated latencies of the CommSolid system model components.

Component	Default Latency	Calibrated Latency
SimpleMemory	$30ns$	$63ns$
MemBus	2 cycles	0 cycles
ALU	3 cycles	5 cycles
Multiply Unit	3 cycles	6 cycles
Division Unit	9 cycles	40 cycles
Load-Store Unit	1 cycle	8 cycles

PicoRV32 core. Gem5 provides a non pipelined CPU model, that considers timing information of the memory. The drawback of the *TimingCPU* is its lack of functional units. This results in every instruction consuming one cycle in the execution stage. The other CPU that is evaluated is the *MinorCPU*, whose functional units are calibrated on the just measured cycle counts of the assembly instructions. Table 5.2 shows the calibrated latencies for the memory models and the functional units of the in-order CPU model. The latency of the memory is calculated by the measured cycle count of the *lw* instruction and with Equation (5.1). In this equation, the number of cycles is set to 4 as the execute state of the instruction consumes 4 cycles. This leads to the assumption that the memory latency is 4 cycles as well. The frequency on the FPGA board is $64MHz$. Therefore, the calculated latency of the memory is roughly $63ns$.

The results for both CPU models can be seen in Table 5.3. The mismatch of the *TimingCPU* is at least 25% except for the store instruction. The highest mismatch occurs in the case of the *div* instruction. As it has the highest result latency in the *PicoRV32* core with 39 cycles, the impact of the one cycle execution stage in the *TimingCPU* model is the biggest. This results in a mismatch of 92.25% for the *div* instruction.

By calibrating the functional units in the *MinorCPU*, feasible results for the assembler instructions can be achieved. For all instructions except *c.add* and *sw* the mismatch is below one percent. The high mismatch of the compressed instruction of 23.97% can not be reduced. In gem5, the *c.add* instruction is also assigned to the functional unit being responsible for ALU operations. More specific, the *c.add* instruction cannot be calibrated separately and therefore shares the same configured latency as the add instruction. That means, it is defined that an operation in the ALU has a five cycle result latency. Furthermore, the pipeline

Table 5.3: Cycles per assember instruction and mismatch to the PicoRV32 core.

Cycles per Instruction	add	c.add	mul	div	lw	sw
MinorCPU	5.03	5.03	6.01	39.41	7.98	7.98
Mismatch	0.25%	23.97%	0.21%	0.03%	0.07%	4.22%
TimingCPU	3.05	3.05	3.05	3.05	6.00	8.31
Mismatch	39.44%	24.94%	49.32%	92.25%	24.84%	0.24%

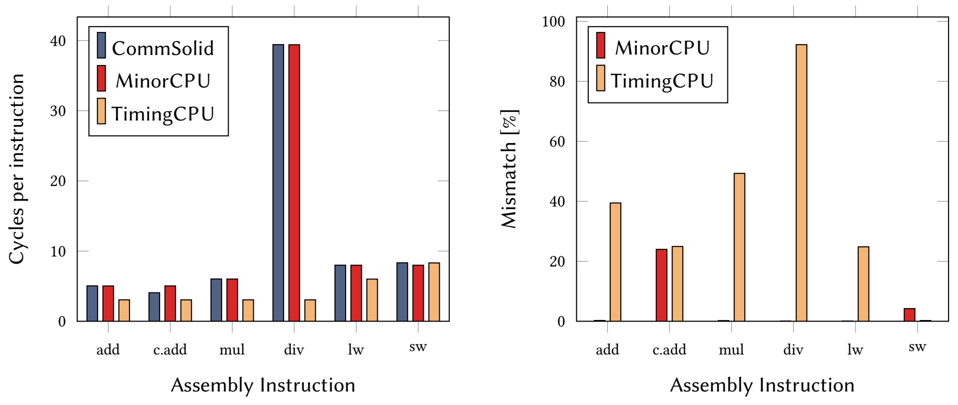

Figure 5.3: Cycle counts for the CommSolid Samara board.

in the in-order CPU model fetches an instruction in every cycle, even though this instruction has already been fetched. Therefore, compressed instructions do not cause the fetch stage to be skipped, and no cycle is saved.

To summarise the results of the assembly instruction benchmarks, Figure 5.3 shows the measured cycles per instruction of the *CommSolid* core compared to the two CPU models in gem5. When taking a look at the mismatches between the CPU models and the *CommSolid PicoRV32*, it becomes clear that the *MinorCPU* is more feasible for simulating the *CommSolid* FPGA board.

SiFive HiFive1

In this part, the calibration of the components in the system definition corresponding to the *SiFive HiFive1* board is shown. As described in Section 5.2.1, the RISC-V board implements the in-order pipelined CPU *FE310-G000*. The corresponding model in gem5 is the highly configurable *MinorCPU*, which models an in-order CPU. To calibrate the functional units, the assembler instruction benchmarks are

Table 5.4: Cycles per assember instruction and mismatch to the FE310-G000 core compared to the gem5 MinorCPU.

Assembly Instruction	add	c.add	mul	div	lw	sw
HiFive1	1.04	1.04	6.73	6.73	1.99	1.04
MinorCPU	1.02	1.02	6.91	8.88	2.48	1.99
Mismatch	2.48%	1.56%	2.69%	31.87%	24.69%	90.93%

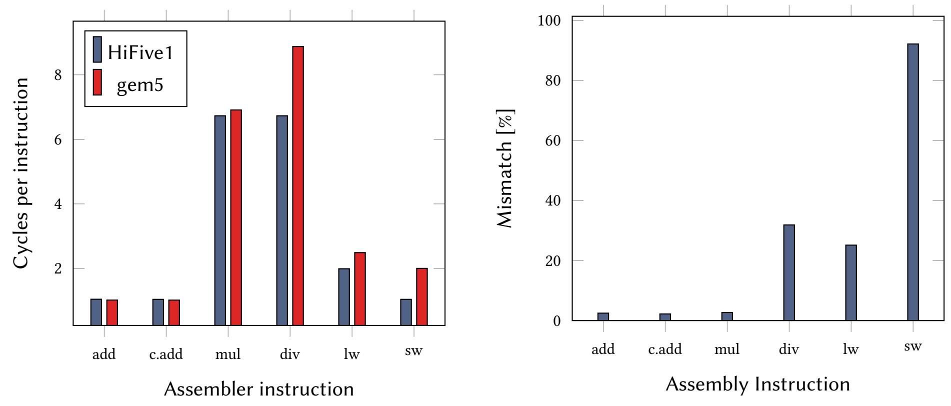

Figure 5.4: Cycle counts using the instruction cache for the SiFive HiFive1 board.

run and evaluated.

Table 5.4 lists the cycles each instruction consumes with a warmed up cache and Figure 5.4 illustrates these results. Especially the calibration ALU and the multiplier leads to accurate results with a maximum of only 2.96% mismatch between simulation and reality. The calibration of the divider is a challenge, as the HiFive1 board implements an early out. In gem5, the default divider configuration was left unchanged leading to a 9 cycle result latency. This leads to an error of 31.87% for the *div* instruction. This mismatch is not constant due to the early out. Therefore, the mismatch depends on the numerator and the divisor. The *lw* instruction achieves a mismatch of 24.96% and the *sw* instruction has an even higher error with 90.93% mismatch between simulation and real hardware. The corresponding functional unit is calibrated on the lowest possible result latency and therefore a higher accuracy is not possible.

As stated earlier, the latency of the SPI-Flash of the SiFive HiFive1 board is not specified in the manual. To calibrate the memory that models the SPI-Fash a separate benchmark is created. This benchmark measures the number of cycles

Table 5.5: Cycles per assembler instruction and mismatch to the FE310-G000 core compared to the gem5 MinorCPU.

Cache Line Fetches		1	2	3	4	5
Number of Instructions		9	17	25	33	41
Cycles	HiFive1	4632	9258	13884	18516	23144
	gem5	4631	9258	13883	18512	23139
Mismatch		0.02%	0%	0.01%	0.02%	0.02%

for an instruction cache miss by executing several *nop* instructions. Each *nop* is executed in the ALU, and therefore the pipeline is expected to deliver one result per cycle, if no cache miss occurs. If a miss occurs, the instruction is fetched from the flash memory. By measuring the cycle counts for this action, the latency can be calculated with the formula Equation (5.1). Therein, t is the time of the SPI flash access latency, n is the number of cycles and f is the frequency of the core. In Table 5.5 the consumed cycles for different numbers of instructions can be seen. The numbers of instructions are chosen according to the number of cache misses they lead to. For one cache miss within 9 instructions, 4632 cycles are needed. This implies, that the instruction where the cache miss occurs lasts 4624 cycles. The *MinorCPU* has a 4-stage pipeline, and it is assumed that the execute stage, the decode stage and the fetch2 stage consume one cycle each. Therefore, the fetch1 stage consumes 4621 cycles, which is the cycle count n in Equation (5.1). This calculation leads to a calibrated latency of the SPI flash of $18\mu s$.

Figure 5.5 shows the calibrated gem5 system in comparison to the SiFive HiFive1 board. Therein, the consumed cycles for various numbers of instructions are shown. With the adjusted latency of the flash memory, the results are within 0.02% mismatch.

Table 5.6 summarises the calibrated latencies of the components in the system model. The *SimpleMemory*, modelling the data RAM, is calibrated to $4ns$ as the memory responds within one cycle. This set-up achieves feasible accuracy on the single-instruction benchmarks and reflects the real hardware board well.

5.2.4 Benchmarks

To evaluate the overall accuracy of gem5, several benchmarks are implemented. These benchmarks have the purpose to primarily analyse the accuracy and configurability of the gem5 CPUs, its functional units and the memory system including

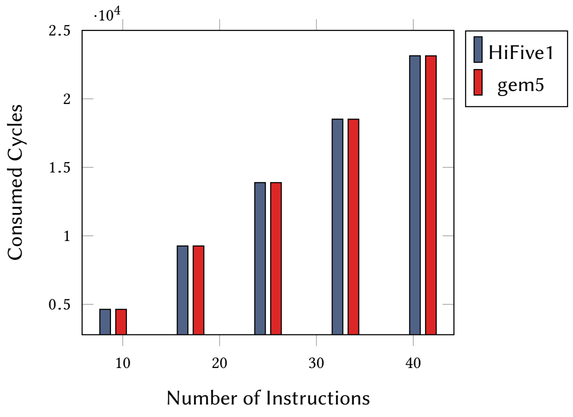

Figure 5.5: Cycle counts for multiple cache line fetches.

Table 5.6: Calibrated latencies of the HiFive1 system model components.

Component	Default Latency	Calibrated Latency
SimpleMemory (Flash)	30ns	18μs
SimpleMemory (RAM)	30ns	4ns
Cache	-	1 cycle
MemBus	2 cycles	0 cycles
ALU	3 cycles	1 cycle
Multiply Unit	3 cycles	7 cycles
Division Unit	9 cycles	9 cycles
Load-Store Unit	1 cycle	1 cycle

Table 5.7: Execution time for the calculation of the Mandelbrot set for different pixel sizes.

Number of Pixels		2x2	5x5	10x10	20x20	50x50	100x100
Execution	CommSolid	0.008	0.039	0.15	0.58	3.44	13.634
Time [s]	gem5	0.007	0.035	0.133	0.51	3.02	11.958
Mismatch		10.71%	11.6%	11.91%	12.12%	12.23%	12.29%

caches. The latency of peripheral devices and other system components are not evaluated.

After calibrating the systems to achieve a high accuracy on the different assembler instructions, more complex calculations are run. Therefore, three algorithms with different properties are benchmarked.

First, the calculation of a Mandelbrot set for different numbers of pixels is executed, which predominantly consists of arithmetical operations. These numbers vary from 2x2 to 100x100 and the number of iterations is set to 100 for all cases.

Second, a matrix multiplication is done for different matrix sizes. This benchmark is intended to stress the load-store unit and to evaluate the accuracy of memory latency as well. In this test, the size of the matrices varies from 2x2 to 64x64.

Finally, the calculation of the N-point fast Fourier transform (FFT) is benchmarked, which unifies the focus of the previous tests. The mismatch between the execution time for different numbers of points N is evaluated. The number of points range from 8 to 1024.

CommSolid PicoRV32

The results for the Mandelbrot set benchmark are shown in Table 5.7 and illustrated in Figure 5.6. The results show, that the simulator is faster in any case. The assumed reasons for that are the pipeline and different functional units in the *MinorCPU* model. These allow to execute instructions concurrently, which leads to a performance increase and a faster execution time. With taking a look at the mismatch, it can be seen that the error rate increases with the number of pixels from 10.71% to 12.29%. But the diagram in Figure 5.6 shows, that the slope of the mismatch is decreasing, which means, that the error rate has an upper bound. Even for large processing times, such as $13s$, the mismatch is in a reasonable magnitude.

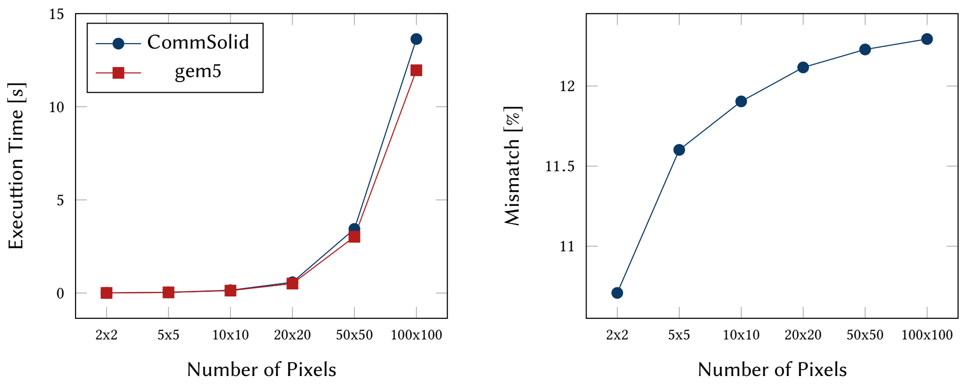

Figure 5.6: Execution time for the Mandelbrot set benchmark on the CommSolid Samara board and simulated in gem5.

Table 5.8: Execution time for the calculation of the multiplication of matrices of different sizes.

Matrix Size		2x2	4x4	8x8	16x16	32x32	64x64
Execution Time [ms]	CommSolid	0.04	0.23	1.6	12.14	94.59	738.58
	gem5	0.03	0.2	1.44	11.05	86.15	697.88
Mismatch		8.53%	10.75%	10.03%	8.96%	8.92%	7.95%

The results of the matrix multiplication benchmark are shown in Table 5.8 and Figure 5.7. The highest error of 10.75% occurs by multiplying two 4x4 matrices. When processing larger matrices, the error rate decreases and is 7.95% for the 64x64 entries. As the space for two input matrices and the output matrix is allocated on the stack, multiplying larger matrices is not possible due to the limited memory. From this benchmark can be concluded, that algorithms loading and storing values often can be simulated with reasonable accuracy, even though a pipelined CPU model is used in gem5.

The N-point FFT is the third of the executed benchmarks for evaluating the overall accuracy of gem5. The results can be seen in Table 5.9. The longest execution time of about $704ms$ on the FPGA board occures for the 1024-point FFT. The mismatch for this 1024-point FFT is 10.87% and it is also the highest occurring. Similar to the error for the Mandelbrot set benchmark, the slope of the error for this benchmark also is decreasing. The temporal progress of the graph for the mismatch in Figure 5.8 indicates that the mismatch has an upper

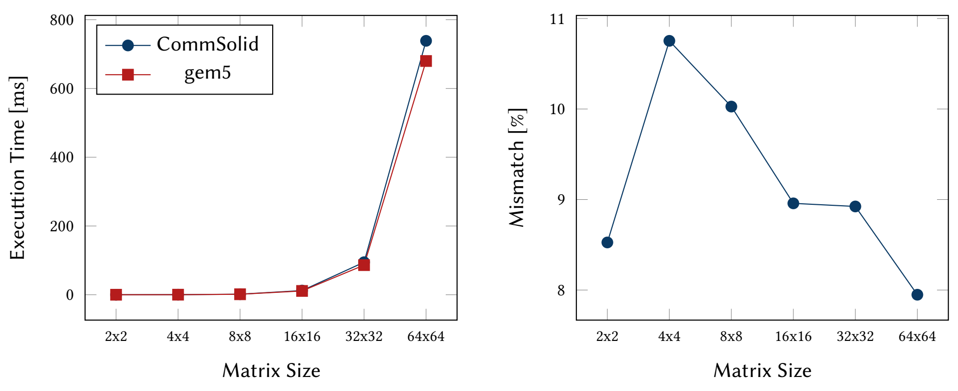

Figure 5.7: Execution time for the matrix multiplication benchmark on the CommSolid Samara board and simulated in gem5.

Table 5.9: Execution time for the FFT benchmark of different sizes.

FFT Points		8	32	128	512	1024
Execution Time [ms]	CommSolid	3.18	17.14	77.61	338.8	704.01
	gem5	2.96	15.45	69.4	302.1	627.47
Mismatch		7.15%	9.83%	10.59%	10.83%	10.87%

bound and will not increase further for larger execution times.

To summarise the results of the benchmarked algorithms it can be said, that even though a pipelined CPU model needs to be used to simulate a non-pipelined core, the simulation results have a reasonable accuracy to evaluate the hardware system based on the simulation. The highest observed error is 12.29% for the Mandelbrot set calculation of 100x100 pixels. This error occurs by the longest execution time of 13.63*s*. The error averages at around 10%. Within the particular benchmarks, the average error is highest for the Mandelbrot set benchmark, whereas the average error in the matrix multiplication benchmark is the lowest. This leads to the assumption, that computation intensive algorithms are simulated with a higher error than memory-access intensive algorithms. The simulation is always faster than the hardware, which can be explained by the pipeline in the CPU model. This pipeline allows concurrent execution of instructions and achieves therefore a higher performance than a non-pipelined CPU. It is assumed, that the pipeline in the CPU model is the reason for this.

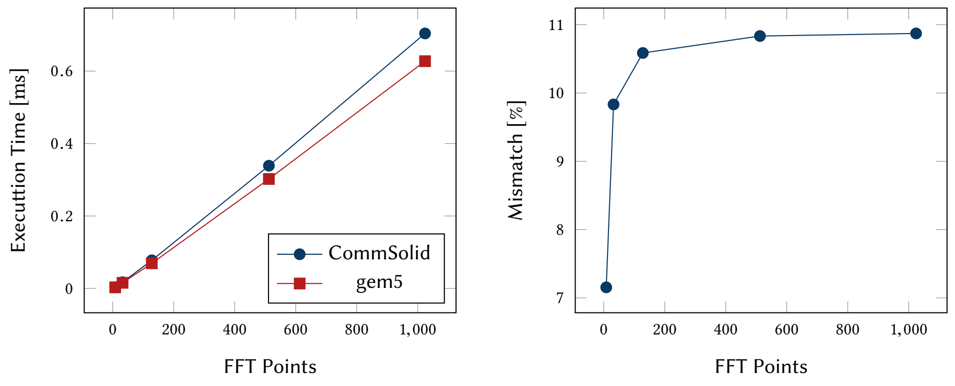

Figure 5.8: Execution time for the FFT benchmark on the CommSolid Samara board and simulated in gem5.

Table 5.10: Execution time for the calculation of the multiplication of matrices of different sizes.

Matrix Size		2x2	4x4	8x8	16x16	32x32
Execution	HiFive1	0.23	0.24	0.35	1.08	6.37
Time [ms]	gem5	0.14	0.15	0.24	0.88	5.73
Mismatch		36.42%	35.64%	31.69%	18.13%	10.00%

SiFive HiFive1

In Table 5.10 the results for the matrix multiplication benchmark are shown. With a mismatch of 36.42% for the multiplication of two 2x2 matrices the simulation is inaccurate for short calculations. With a greater execution time, the mismatch gets smaller and is 10% for the multiplication of two 32x32 matrices. Due to the limited memory, it is not possible to multiply larger matrices as they are allocated on the stack.

The graphs in Figure 5.9 illustrate this observation. Even though the execution time increases exponentially, the mismatch decreases. This leads to the assumption, that a part in the algorithm initialisation is simulated with low accuracy. This part has a higher influence for small matrices and is qualified for longer execution times.

The results for the calculation of the Mandelbrot set and FFT are shown in Table 5.11 and Table 5.12. Additionally, these values are graphically illustrated in Figure 5.10 and Figure 5.11. For the Mandelbrot set calculation, the mismatch

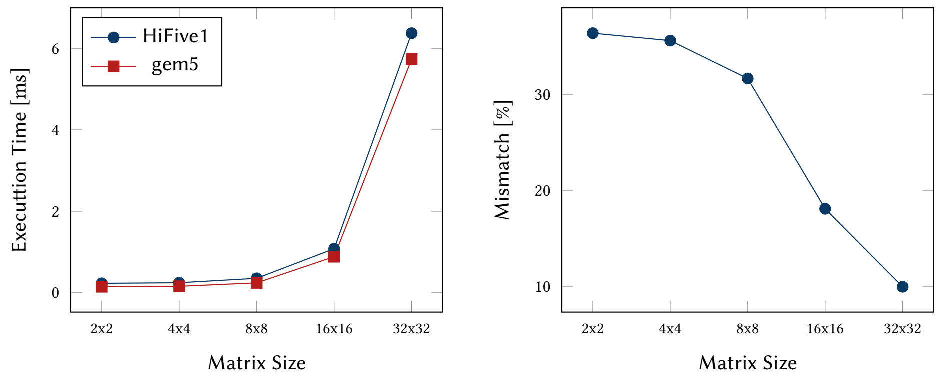

Figure 5.9: Execution time for the matrix multiplication benchmark on the SiFive HiFive1 board and simulated in gem5.

Table 5.11: Execution time for the calculation of the Mandelbrot set for different pixel sizes.

Number of Pixels		2x2	5x5	10x10	20x20	50x50	100x100
Execution	HiFive1	0.008	0.039	0.15	0.58	3.44	13.634
Time [ms]	gem5	0.007	0.035	0.133	0.51	3.02	11.958
Mismatch		102.31%	124.83%	164.28%	180.56%	177.74%	176.13%

varies from 100% to 180%. The mismatch of the 1024-point FFT is even higher with 190%. After the calibration and results from the matrix multiplication benchmark, these mismatches are remarkable and requires further investigation.

5.2.5 Analysing the HiFive1 Simulation Model

The gem5 simulation model of the SiFive HiFive1 board achieved high mismatches for the FFT benchmark and the calculation of the Mandelbrot set, although the calibration leads to a good accuracy of single instruction benchmarks. Furthermore, the achieved accuracy for the matrix multiplication is sufficient with 10.00% for 32x32 matrices. In the following, the process of finding the reason for the high mismatches in the FFT and Mandelbrot set benchmark is described. It is chosen to outline this analysis as the location of the reason for the high errors was not straight forward.

First, the influence of the cache is assumed to be a potential issue. Since the CommSolid PicoRV32 core has no caches, and the mismatch between it and

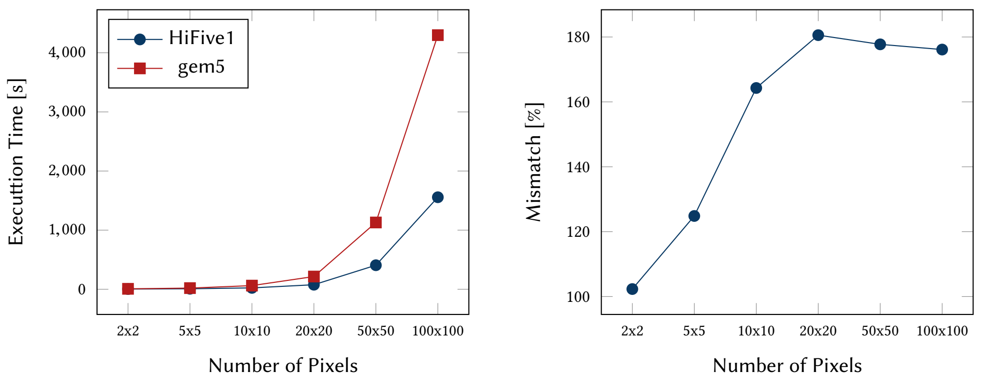

Figure 5.10: Execution time for the Mandelbrot set benchmark on the SiFive HiFive1 board and simulated in gem5.

Table 5.12: Execution time for the FFT benchmark of different sizes.

FFT Points		8	32	128	512	1024
Execution	HiFive1	5.93	7.69	15.65	46.98	89.96
Time [ms]	gem5	6.63	12.57	36.15	132.44	261.25
Mismatch		11.85%	63.43%	130.95%	181.87%	190.41%

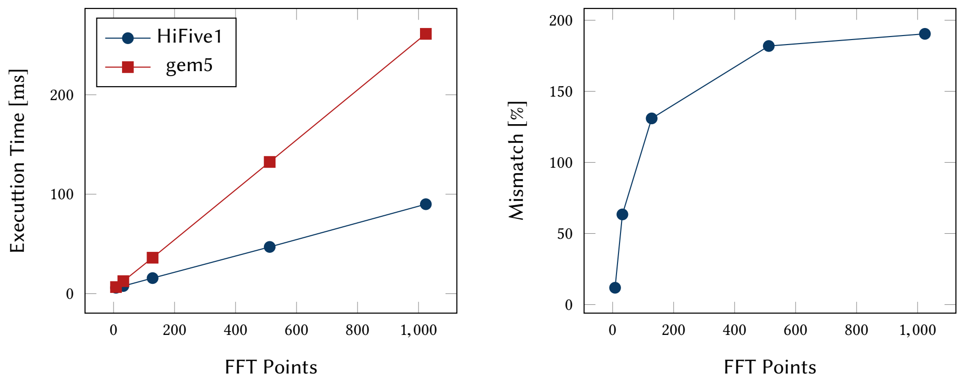

Figure 5.11: Execution time for the FFT benchmark on the SiFive HiFive1 board and simulated in gem5.

```
li a7, 1000
0:
beqz a7, end
addi a7, a7, -1
j 0b
end:
```

Listing 5.3: Instruction sequence of the testcase producing high mismatches.

its simulation is in a sufficient range, issues with the cache seems plausible. Therefore, the cache misses for the FFT benchmark are analysed. The cache misses are approximately constant for all numbers of points in the FFT. More precisely, the 8-point FFT produced the same amount of cache misses as the 1024-point FFT. This leads to the assumption that the cache is not the reason for the high mismatches.

To investigate further potential causes, a look into the linker script of the SiFive HiFive1 software development kit is done. There, the output section *.rodata* aggregating read-only data is mapped into the SPI-Flash memory. Furthermore, it is unpacked by the bootloader into flash memory as well. That implies, read-only data is located in the flash memory during execution of the user-application. The board has only an instruction cache and accesses to flash memory are slow with a calculated latency of $18\mu s$. Moreover, only the FFT and the Mandelbrot set benchmarks contained read-only data. Therefore, a difference in the access time of read-only data could be an issue. To investigate this assumption, another benchmark is created. A constant array is defined that is mapped into the section *.rodata* by the linker. Its values are accessed and the consumed cycles measured. The benchmark leads to mismatches of less than 5%, which disproves the assumption.

The occurrence of floating-point values and calculations in former algorithms is another difference of the Mandelbrot set and FFT benchmark compared to the vector multiplication. As the board has no floating-point unit, calculations and conversions are done in software. For that, the library *glibc* provides predefined functions. Through several measurements of the execution time of these functions, the method *__muldf3* is found to cause a high mismatch between execution on the board and in the gem5 simulator. Investigation of its structure shows, that besides arithmetic instructions unconditional jumps occur in a significant amount. To analyse this further, a benchmark was designed were defined amounts of unconditional jumps are executed. Listing 5.3 shows the assembly sequence that causes high mismatches between the HiFive1 board and the gem5 simulator.

Table 5.13: Consumed cycles for different numbers of unconditional jumps.

Number of Jumps		1000	2000	3000	4000	5000
Number of Instruction		3018	4018	5018	6018	7018
Consumed Cycles	HiFive1	3066	4069	5045	6069	7057
	gem5	8040	16038	10040	18038	12040
Mismatch		162.23%	294.15%	99.01%	197.22%	70.61%

Table 5.14: Execution time for the FFT benchmark of different sizes without the initialisation phase.

FFT Points		8	32	128	512	1024
Execution Time [ms]	HiFive1	0.89	1.16	2.55	9.40	19.63
	gem5	0.75	1.00	2.28	8.58	18.03
Mismatch		15.67%	13.44%	10.79%	8.76%	8.17%

Table 5.13 shows the results for this benchmark. It compares the consumed instructions and cycles for unconditional jumps. Additionally, the mismatch between the gem5 simulator and the SiFive HiFive1 board is shown. This mismatch varies from 70.61% for 5000 jumps to 294.15% for 2000 jumps.

To analyse the exact cause for this mismatches, the execution pipeline of the CPU model in gem5 is observed using the *pipeline viewer*. This tool is part of the gem5 project and visualises the pipeline. It can be comprehended how a single assembly instruction is processed through each stage of the pipeline. Utilizing this tool shows, that the branch predictor experiences difficulties while processing backward jumps. If a backward jump occurs, the pipeline is stalled for 6 cycles. The measurements listed in Table 5.13 indicate that the pipeline in the SiFive HiFive1 board is fully utilized during the whole benchmark.

The FFT benchmark uses floating-points only for initialising an array that represents the input signal. To prove the just presented observations, a second measurement was done excluding the initialisation step. Table 5.14 and Figure 5.12 show the results. After an analyse, it occurs that the mismatch is highest for the 8-point FFT and lowest for the 1024-point FFT. The highest occurring error is 15.67% for the 8-point FFT. It decreases to 8.17% for 1024 points. Similar to the vector multiplication benchmark, short execution times cause high errors. For

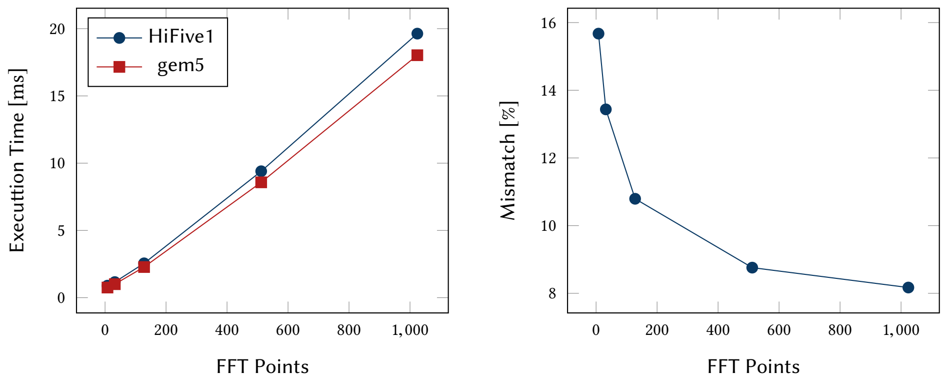

Figure 5.12: Execution time for the FFT benchmark without the initialisation phase on the SiFive HiFive1 board and simulated in gem5.

longer execution times a higher accuracy is achieved.

5.2.6 Discussion of the Benchmark Results

The benchmark results for both system models show a sufficient simulation accuracy for severely different hardware architectures. It is possible to simulate a non-pipelined CPU using the in-order CPU model of gem5. This causes the simulation to be faster than the hardware. However, the speed-up is in a feasible range with the highest occurring error of 12.29%. The average error for this use-case is around 10% and slightly higher for long lasting calculations. These results are reliable for different types of applications as the occurring errors for all three benchmarked algorithms are within similar range.

The accuracy for simulating a pipelined board depends on the program flow. The analysis in Section 5.2.5 shows, that the occurrence of backward jumps in the program flow influence the accuracy severely. Using the *MinorCPU* for simulation of hardware with pipelined CPUs is feasible for long running applications where few backward jumps occur. In this scenario, the accuracy is feasible with an error below 10%. This requires knowledge about the structure of the program, that is run on the simulation model. If the program structure is not known and many jumps occur, the simulation results are not reliable in terms of cycle counts and execution time. Especially if the application processes floating-point operations in software, high errors can be expected.

Gem5 is sufficient to study the behaviour of applications running on a specific system. The simulator is build to study the interaction of components of a SoC while running unmodified binaries and operating systems on the system model. If the exact architecture of the CPU matters, it is better to implement a corresponding model in gem5.

5.3 Acceleration of Algorithms Using the RISC-V Extension Parser

This section gives an example on how the RISC-V Extension Parser supports the acceleration of algorithms by using additional hardware. It shows, that using the tool together with gem5 allows fast evaluation of the impact of custom instructions on software algorithms. For this purpose the FFT algorithm is chosen, because it is already available in *c*-code due to its use as benchmark. The algorithm is compiled with the compiler option "-O2" for speed optimisations, as system model serves the definition of the SiFive HiFive1 board.

Deep inside in the algorithm, a fixed-point multiply function is called. Therein, two values of type *short* are multiplied and the result is right shifted by 15 Bit. Listing 5.4 shows the function definition. In the target architecture, the type *short* is 16 Bit wide. As a consequence, the multiplier in the execution pipeline of the SiFive HiFive1 model multiplies two 16 Bit values, though it is capable of processing 32 Bit. This multiplier has a result latency of 7 cycles. Building a fixed-point multiplier directly in hardware that only processes 16 Bit input values is assumed to deliver the result within one cycle. Moreover, the right shift is achieved in hardware by ignoring the lower 15 Bit. Consequently accelerating this function with a custom instruction can increase the performance of the FFT.

To evaluate this statement, a fixed-point multiply instruction with a result latency of one cycle is defined in the RISC-V Extension Parser. After defining the instruction, the FFT algorithm is adapted to access the additional hardware when multiplying fixed-point values.

```c
#define FIX_MPY(DEST,A,B) DEST = ((long)(A) * (long)(B))
    >>15

fixed fix_mpy(fixed a, fixed b)
{
    FIX_MPY(a,a,b);
    return a;
}
```

Listing 5.4: Definition of the fixed-point multiply function used inside of the FFT algorithm.

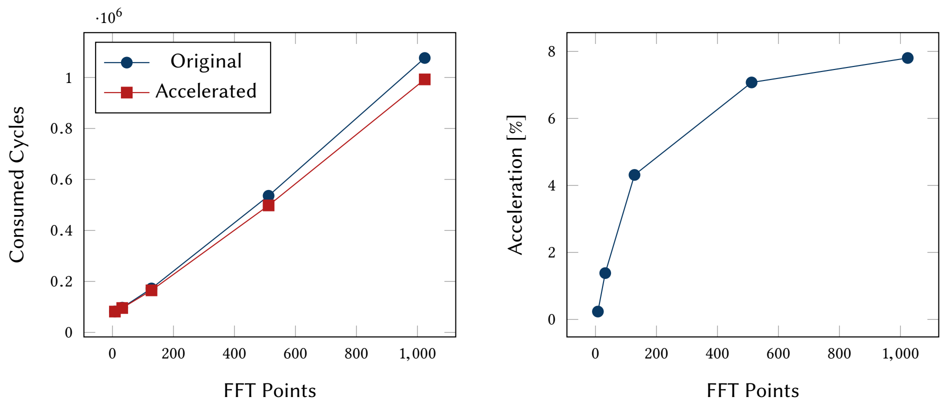

Figure 5.13: Cycle counts of the FFT algorithm in *c* and accelerated with a custom instruction.

Figure 5.13 shows the consumed cycles for the original FFT compared to the accelerated algorithm. Additionally, the gained acceleration in percent is illustrated. For the 8-point FFT the gained acceleration is below 1% as the fixed-point multiply is not called often. Larger sizes of the FFT causes more fixed-point multiplication, hence the custom instruction is utilised more often. It can be seen that the accelerated version of the 1024-point FFT consumes nearly 7.8% less cycles than the original *c*-implementation.

6 Related Work

This chapter highlights relevant projects based on RISC-V and gem5.

6.1 RISC-V

RISC-V is of high interest for both researchers and the industry due to its advantages against other open source risc architectures [3]. This section gives an overview about existing RISC-V projects.

Very soon since RISC-V has been proposed by Asanović and Patterson [3] in 2010 the University of Berkeley used the ISA in own projects and teaching [47]. Lee [47] used a simple unpipelined RISC-V core for tutorials on RTL synthesis. Also the first more complex studies on CPU designs based on RISC-V came from Berkeley. In 2014 Lee et al. [48] proposed a 64 Bit dual-core. This architecture consists of two *Rocket* cores that both have a six stage pipeline with single-issue in-order execution. The core uses the high extensibility of the RISC-V architecture and implements custom vector accelerators. Furthermore, it meets the requirements to boot modern operating systems including Linux [48]. This core achieves better energy efficiency by smaller size than the ARM Cortex-A5. These results show that RISC-V is competitive in the area of low power devices, including smartphones and IoT devices.

In 2015 Asanovic, Patterson, and Celio [4] proposed *BOOM*, a superscalar out-of-order RISC-V core. It is intended to serve as baseline implementation of a out-of-order micro-architecture for education, research and industry [4]. The core supports branch speculation and branch prediction. Benchmark results show that this core is competitive in performance and area compared to commercially available embedded out-of-order cores [4].

The two presented RISC-V cores lead to the *Rocket Chip Generator* proposed by Asanovic et al. [5]. This project is an open-source SoC generator, that emits synthesizable RTL. The framework is a library of generators for cores, caches and interconnects and composes them to a SoC. It is base for several projects that lead to tape-outs and has therefore proven its practicability [49, 50].

Due to the properties of RISC-V that make the ISA ideal for academic use, other universities realised projects based on it. Popular among them is the *PULPino* project by ETH Zürich [76]. The goal of Traber et al. [76] is to build an open-source ultra-low-power processor for computing in internet of things (IoT). Because of its *FreeBSD* license, its compressed instructions and its extensibility, the group chose RISC-V. The outcome is an open source, 32 Bit single-core SoC [75] that is competitive against the *ARM Cortex-M4* [76]. Because of its *Advanced Debug Unit* that enables debugging over JTAP and SPI and its peripheral devices including UART, timer and *GPIO*, this project is ideal as SoC in embedded devices. [76]

The *PULPino* project is the base for other academic works, which focus on embedded systems and IoT as well. Cheikh et al. [17] proposed a multi-threaded RISC-V core family. This work approaches the execution of concurrent threads in IoT end-nodes. It solves this problem by enabling multi-threading. With that, multiple applications can be run on a single core. It is a 32 bit architecture, that is compliant with the *PULPino* SoC and can therefore be used as processing core in the system. [17]

In embedded computing energy efficiency is required while being able to efficiently process complex computations involve floating point operations [54]. Mach et al. [54] use the high extensibility of RISC-V and implement a floating point unit in the *RI5CY* core of the *PULPino* SoC. This FP unit optimizes the system regarding energy efficiency by implementing SIMD operations for floating point formats and is directly integrated in the execution stage of the pipeline of the core. With that implementation the *PULPino* core is up to 18% more efficient by up to 25% more performance.

Both, the *Rocket Chip Generator* and the *PULPino* SoC are important projects used as starting point for works that approach problems in embedded systems. For instance, these problems include the enhancement of security in embedded systems [22, 69, 87].

Not only in academia, but also in industry RISC-V becomes more and more popular. The first commercially available RISC-V SoC is produced by *SiFive* [37]. This system is part of the *SiFive HiFive1*, a cheap and small arduino compatible RISC-V development kid [38]. Therein, a 32 bit six stage pipelined processor is implemented. The company also offers an open source software development kid for convenient implementation, building and debugging of RISC-V based software.

To summarise the just presented academic works, RISC-V is very famous in the domain of low-power embedded devices. Due to its extensibility, custom hardware units can be implemented to optimize designs regarding energy efficiency and power consumption. The results of this approaches are shared among researches and taken as starting point for new projects. This is only possible, because RISC-V is an open source ISA.

6.2 gem5

The gem5 simulator [11] is of high interest in the academia. This chapter summarises important works regarding accuracy evaluations using the gem5 simulator. Furthermore, usage of gem5 in context of RISC-V and embedded system is summarised.

As accuracy is an important property for simulators, several works have evaluated the accuracy of gem5. These evaluations mostly targeted ARM cores. Butko et al. [15] analysed the execution time of benchmarks against the hardware development kit *Snowball SKYS9500-ULP-C01*. This board implements a dual-core *ARM Cortex-A9* with instruction and data L1 caches and a shared L2 cache. The mismatch of the execution time varies between 1.39% and 17.94%. However, Butko et al. [15] used the *Timing Simple* CPU model that does not simulate at the micro-architectural level.

Endo, Couroussé, and Charles [25] directly focused on in-order and out-of-order *ARM* microprocessors. The researchers used the out-of-order CPU model of gem5 to simulate *ARM Cortex-A8* and *ARM Cortex-A9* cores. As the *O3* model was the only functional timing accurate CPU model in gem5 at that time, they configured it to model an in-order pipeline. With that Endo, Couroussé, and Charles [25] achieved accuracy results of only 7% mismatch.

Another work regarding accuracy optimization of gem5 for an *ARM* architecture was proposed by Gutierrez et al. [32]. They analysed the sources of mismatches between simulation and reality for not only the overall system but also for single components. For that, they compared the simulation with the development board *ARM Versatile Express TC2*.

Further investigations were done regarding the accuracy of full system simulators including gem5. Nowatzki et al. [61] evaluated errors in several full system simulators. In gem5 they highlighted several flaws in the *O3* CPU model. However, these flaws were only verified against the x86 architecture and reported to the gem5 community. This evaluation was only possible due to the openness of gem5. With contributions by the community, the gem5 simulator improves steadily, as flaws are found and fixed.

This thesis uses the RISC-V implementation for gem5 as foundation. The implementation was contributed by Roelke and Stan [68] in 2017. Prior to this implementation, simulators were either slow but highly accurate RTL level simulations or fast but low-accurate binary translation. With gem5 this gap is closed enabling fast but accurate simulation of RISC-V based systems. The implementations done in this work will extend the features contributed by Roelke and Stan [68].

Key features of RISC-V are the highly extensibility with standard extensions as well as custom extensions. The simulator must support this feature as well. Especially in embedded system, custom extensions are used to enhance the performance of the chip. To achieve accurate results, simulators must be able to work with such hardware accelerators. For gem5 an implementation was proposed by Shao et al. [71] for co-design of extensions and SoC interfaces. They assumed that hardware accelerators are often separate IP blocks within the SoC and consists of multiple customized datapath lanes and local memories [71]. Especially the local memories require data movement and coherence management. In their work Shao et al. [71] enhanced gem5 to capture interactions between the SoC and accelerators and achieved an accuracy within 6% against real hardware.

This thesis proposes an interface for defining custom extensions to accelerate system by additional hardware. The presented approach in Section 4.4 places the additional hardware as functional unit within the pipeline of the CPU model. This is conform to work by Mach et al. [54], who also placed their hardware accelerator within the pipeline. Therefore, the work by Shao et al. [71] is not used in this thesis.

Gem5 has a modular and open design that allows to combine the simulator with other frameworks. In the following, two projects are introduced that used this approach in order to increase the number possibilities for system level design space exploration.

Menard et al. [57] contributed an implementation that allows interoperability between the gem5 and *SystemC* framework. *SystemC* [40] is an *IEEE* standard that extends *c++* with macros and classes. With that an event-driven simulation kernel is provided. Coupling gem5 and *SystemC* offers the possibility to use and connect models from both frameworks. Furthermore, this project offers the possibility to simulate ISA-heterogeneous systems, which increases the number of systems that can be simulated.

Another approach that also allows to simulate multiple, distributed cores, is the *dist-gem5* project [58]. It is a distributed version of gem5 proposed by Mohammad et al. [58]. Dist-gem5 reduces the time a simulation lasts by offering the possibility to distribute the work load on multiple simulation hosts. Hence, the project was intended to support design space exploration for HPC systems, the approach can be used to simulate multi-processor system-on-chip (MPSoC) architectures. Because every host runs its own gem5 instance, these instances can be compiled for different ISAs. This enables the simulation of ISA-heterogeneous computing.

To enrich design space exploration with gem5 with information about area and power consumption, Endo, Couroussé, and Charles [24] combined gem5 and McPAT. McPAT [52] is a multi-core power and area simulator, that estimates power and area values for system definitions. In their work, the researchers implemented a parser that enables system definition translation between these two frameworks. With that, system definitions from gem5 can be used as input for the McPAT simulator. This allows to estimate the power area consumption of systems defined in gem5.

7 Future Work

This chapter summarises ideas for future projects using this work as base. First, conceptions for further implementations in the gem5 simulator are presented. These offer the possibility to simulate a wider range of use-cases. Afterwards, ideas for projects using implementations presented in this thesis as foundation.

7.1 RISC-V Support in gem5

The RISC-V support in gem5 needs to be extended in order to support a wider range of systems. This regards better support of peripherals as well as implementation of more standard extensions. In the following, open tasks in gem5 are discussed.

7.1.1 Evaluation of the Out-Of-Order CPU Model

The accuracy evaluation in Section 5.2.4 shows that the in-order CPU model has difficulties with fully utilizing the pipeline on occurrence of jumps. Topics for future projects can be the evaluation of the implementation of this CPU model and the adaptation of the branch prediction.

Gem5 offers also an out-of-order CPU. This model is not tested for the RISC-V ISA. In a future project, this CPU can be evaluated and analysed if it achieves a higher accuracy on simulating the SiFive HiFive1 board.

7.1.2 Platform-Level Interrupt Controller

Currently, local timer interrupts with the implemented CPU-Timer are the only possibility to interrupt the current program. As the name implies, this timer is bound to one specific CPU. To simulate more realistic SoCs, global interrupts have to be supported. Therefore, the platform-level interrupt controller (PLIC) has to be implemented.

The PLIC is defined in the privileged architecture specification of the RISC-V ISA [81]. This device connects global interrupt sources to interrupt targets and its purpose is to find an available hardware thread where an occurring interrupt can be processed. Thereby, interrupt sources are usually I/O devices and interrupt targets are usually CPUs. On occurrence, global interrupts are sent to the PLIC. In the controller, the interrupt is processed and the interrupt enable bits of each of the CPUs are checked. The PLIC selects a target, where the pending interrupt is enabled. The chosen hardware thread then processes the interrupt.

7.1.3 Physical Memory Protection

Privilege levels are not checked in the current implementation of the RISC-V ISA in gem5. Consequently, user programs can access machine mode registers and every memory region being defined in the system. Therefore, only insecure and simple applications can be simulated. In current embedded systems, different applications run on the same processor and memory protection from user-mode applications is common. To simulate this use-case in gem5, physical memory protection and different privilege levels are needed.

To support secure processing by limiting the physical addresses accessible by software, the RISC-V privileged architecture specification [81] defines physical memory protection (PMP). The optional PMP unit adds additional machine mode control registers. PMP entries are described by an 8 Bit configuration register and an address register of the width of the architecture. The access rights of different privilege levels for address spaces are defined. In fact, PMP is a partitioning of the available memory and is also compatible to virtual memory.

Currently, in gem5 the control registers are implemented but the check within memory access is missing. One approach to implement these checks is within the TLB of the RISC-V architecture. Its methods for converting virtual to physical addresses are always called, no matter if the system actually has virtual addresses. Implementing the PMP check in the TLB ensures memory protection on every memory access.

7.1.4 Support of Standard Extensions

The implementation of the 64 Bit version of RISC-V in gem5 supports the standard extensions for floating-point and atomic instructions. Implementing both in the 32 Bit decoder would enable the simulation of a wider range of systems.

The standard extensions for single-precision floating point adds additional instructions and registers and is compliant with the 2008th revision of the IEEE 754 arithmetic standard [89]. The 33 additional registers are 32 Bit wide, where 32 of them are general purpose and one is a status register [80]. Furthermore, this extension adds 26 instructions.

Atomic instructions are required for supporting the synchronization between multiple RISC-V hardware threads running in the same memory space. Therefore, this standard extension contains 11 additional instructions for atomically read-modify-write memory.

The concept of enabling these extensions in RISC-V is similar to the implementations of the standard described in Section 4.3.4. Single-precision floating-point values are 32 Bit wide, which offers the possibility to reuse the already defined instruction formats. To enable the floating-point extension for 32 Bit architectures, the instructions have to be added in the decode tree. For the standard extension of atomic instructions existing instruction formats have to be adapted in order to reflect the 32 Bit width of the input and output registers. Furthermore, the RISC-V 32 Bit decoder tree has to be extended with the operational codes and the functional descriptions of the new instructions.

7.2 Further Projects

The implementations introduced in Chapter 4 offer the possibility to simulate RISC-V based systems with custom extensions in Full-System mode in gem5. This section presents ideas for research projects using this possibility as foundation.

7.2.1 Automatic Generation of Peephole Optimisations

Accelerating hot-spots in computation-intensive algorithms with custom hardware is very common in the domain of embedded systems. Examples are acceleration of floating point operations [54] or accelerating the calculation of cryptographic algorithms [65, 78].

Due to its extensibility, the RISC-V ISA is ideal for building SoCs enriched with hardware accelerators. The outcomes of the implementations presented in this thesis support this feature by offering an interface, where arbitrary custom extension can be defined. Furthermore, these extensions are directly usable in the gem5 simulator allowing researchers the possibility to conveniently evaluate the impact of accelerators in terms of computational performance.

Currently, the custom instructions have to be called by the programmer explicitly. Utilising the RISC-V Extension Parser can support research by developing automatic generation of peephole optimisations. Peephole optimisers in compilers recognize patterns and substitute them with functionally equal, but faster instructions [20]. Creating these optimisations and replacement rules by hand is time consuming. Furthermore, not all opportunities for optimisations may be exploited [8]. Therefore, researchers proposed ideas for automated generation of peephole optimisers [1, 8, 20].

Utilising the RISC-V Extension Parser offers the possibility to develop automated generation of peephole optimisers for the RISC-V compiler. The tool can be used to create arbitrary custom instructions and the capabilities of the gem5 simulator allows to analyse the efficiency of the generated optimisers.

7.2.2 Heterogeneous Computing

Especially in embedded systems, multiple cores of different ISAs are used to achieve instruction level parallelism [46]. One type of heterogeneous computing is having multiple cores of different ISAs on a single chip [79]. Each core is built to fulfil one specific task and its ISA is chosen according to its properties in fulfilling this task. These special-purpose solutions have large performance and energy advantages over general purpose solutions [6].

The extensibility and modularity of RISC-V is well-suited for the application as a coprocessor with a dedicated task. For example, one small RISC-V core with custom extensions built to process security algorithms is used as a coprocessor besides an ARM application processor with multiple cores.

As the gem5 binary is compiled for exactly one ISA, the simulation of ISA-heterogeneous systems is natively not possible. The SystemC coupling by Menard et al. [57] can solve this limitation and enables simulation of heterogeneous MPSoCs. With the coupling multiple gem5 instances compiled for different ISAs can be connected. Every instance represents one core of the MPSoC system and is connected to the other cores with SystemC objects.

The academia is already aware of challenges of MPSoC architectures. Asmussen et al. [6] proposed an microkernel-based operating system specifically designed to support heterogeneous architectures with custom accelerators. The implementations done in this thesis offer the possibility to include RISC-V cores with definable custom extensions into MPSoC simulations. These can be used to support development of tools, such as compilers and debuggers, for efficient and convenient software development for MPSoC systems.

8 Conclusion

This thesis described a solution to overcome the deficiency of full system simulators for the RISC-V ISA. The basic RISC-V support in the simulator gem5 was extended and the full system simulation mode was enabled. Furthermore, the possibility was implemented to simulate 32 Bit architectures. A concept and an implementation of a module enabling the definition of additional instructions and registers was described. The RISC-V Extension Parser automatically parses the definitions, patches the toolchain, and generates a plug-in for gem5. Defined custom instructions are directly included in the execution pipeline of the gem5 in-order CPU model. This offers the possibility to evaluate architectures enriched with custom extensions with sufficient accuracy in a feasible amount of time.

In this thesis, the accuracy of the simulation of RISC-V based systems in gem5 was evaluated. Although these available hardware systems differ quite severely in their architecture, it was able to achieve sufficient accuracy results for both systems. For the non-pipelined architecture of the CommSolid PicoRV32 core, a mismatch between 7.15% and 12.29% was measured. It was shown that due to the functionality of the in-order CPU model of gem5 the accuracy of the simulation of pipelined architectures depends on the structure of the program. For arithmetical calculations with few occurring backward jumps the mismatch between hardware and simulator is below 10%.

This thesis presented an example on the possibility to use the RISC-V Extension Parser for architectural evaluations. A custom instruction was created, whose usage within the FFT algorithm increased the performance and lead to 7.8% less consumed cycles.

Parts of the implementation have been contributed to the gem5 main release and are available for use.

References

[1] Farhana Aleen, Vyacheslav P Zakharin, Rakesh Krishnaiyer, Garima Gupta, David Kreitzer, and Chang-Sun Lin. “Automated compiler optimization of multiple vector loads/stores”. In: *International Journal of Parallel Programming* 46.2 (2018), pp. 471–503.

[2] *ARMv8-A architecture reference manual.* Tech. rep. ARM Ltd., 2015.

[3] Krste Asanović and David A Patterson. “Instruction sets should be free: The case for risc-v”. In: *EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2014-146* (2014).

[4] Krste Asanovic, David A Patterson, and Christopher Celio. *The berkeley out-of-order machine (boom): An industry-competitive, synthesizable, parameterized risc-v processor.* Tech. rep. University of California, Berkeley, 2015.

[5] Krste Asanovic, Rimas Avizienis, Jonathan Bachrach, Scott Beamer, David Biancolin, Christopher Celio, Henry Cook, Daniel Dabbelt, John Hauser, Adam Izraelevitz, et al. “The rocket chip generator”. In: *EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2016-17* (2016).

[6] Nils Asmussen, Marcus Völp, Benedikt Nöthen, Hermann Härtig, and Gerhard Fettweis. “M3: A hardware/operating-system co-design to tame heterogeneous manycores”. In: *ACM SIGPLAN Notices.* Vol. 51. 4. ACM. 2016, pp. 189–203.

[7] Jonathan Bachrach, Huy Vo, Brian Richards, Yunsup Lee, Andrew Waterman, Rimas Avižienis, John Wawrzynek, and Krste Asanović. “Chisel: constructing hardware in a scala embedded language”. In: *Design Automation Conference (DAC), 2012 49th ACM/EDAC/IEEE.* IEEE. 2012, pp. 1212–1221.

[8] Sorav Bansal and Alex Aiken. “Automatic generation of peephole superoptimizers”. In: *ACM Sigplan Notices.* Vol. 41. 11. ACM. 2006, pp. 394–403.

[9] Fabrice Bellard. “QEMU, a fast and portable dynamic translator.” In: *USENIX Annual Technical Conference, FREENIX Track.* Vol. 41. 2005, p. 46.

[10] Dileep Bhandarkar and Douglas W Clark. “Performance from architecture: comparing a RISC and a CISC with similar hardware organization”. In: *ACM SIGARCH Computer Architecture News.* Vol. 19. 2. ACM. 1991, pp. 310–319.

[11] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R Hower, Tushar Krishna, Somayeh Sardashti, et al. “The gem5 simulator”. In: *ACM SIGARCH Computer Architecture News* 39.2 (2011), pp. 1–7.

[12] Nathan L Binkert, Ronald G Dreslinski, Lisa R Hsu, Kevin T Lim, Ali G Saidi, and Steven K Reinhardt. “The M5 simulator: Modeling networked systems”. In: *IEEE Micro* 26.4 (2006), pp. 52–60.

[13] Alex Bradbury. *RISC-V LLVM status update.* http://lists.llvm.org/pipermail/llvm-dev/2017-August/116709.html. Accessed: 24.04.2018. Aug. 2017.

[14] Doug Burger and Todd M Austin. “The SimpleScalar tool set, version 2.0”. In: *ACM SIGARCH computer architecture news* 25.3 (1997), pp. 13–25.

[15] Anastasiia Butko, Rafael Garibotti, Luciano Ost, and Gilles Sassatelli. “Accuracy evaluation of gem5 simulator system”. In: *Reconfigurable Communication-centric Systems-on-Chip (ReCoSoC), 2012 7th International Workshop on.* IEEE. 2012, pp. 1–7.

[16] Steve Chamberlain, Roland Pesch, Red Hat Support, and Jeff Johnston. *The Red Hat newlib C Library.* Tech. rep. Red Hat Inc., Dec. 2014.

[17] Abdallah Cheikh, Gianmarco Cerutti, Antonio Mastrandrea, Francesco Menichelli, and Mauro Olivieri. “The microarchitecture of a multi-threaded RISC-V compliant processing core family for IoT end-nodes”. In: *arXiv preprint arXiv:1712.04902* (2017).

[18] *CommSolid by Goodix.* Accessed: 15.08.2018. URL: `https://commsolid.com/`.

[19] Palmer Dabbelt. “RISC-V Software Ecosystem”. Accessed: 24.04.2018.

[20] Jack W Davidson and Christopher W Fraser. *Automatic generation of peephole optimizations.* Vol. 19. 6. ACM, 1984.

[21] Tom De Schutter. *Better Software. Faster!: Best Practices in Virtual Prototyping*. Happy About, 2014.

[22] Ghada Dessouky, Shaza Zeitouni, Thomas Nyman, Andrew Paverd, Lucas Davi, Patrick Koeberl, N Asokan, and Ahmad-Reza Sadeghi. "LO-FAT: Low-overhead control flow attestation in hardware". In: *Proceedings of the 54th Annual Design Automation Conference 2017*. ACM. 2017, p. 24.

[23] Sarah El Kady, Mai Khater, and Merihan Alhafnawi. "MIPS, ARM and SPARC-an architecture comparison". In: *Proceedings of the World Congress on Engineering*. Vol. 1. 2014.

[24] Fernando A Endo, Damien Couroussé, and Henri-Pierre Charles. "Micro-architectural simulation of embedded core heterogeneity with gem5 and mcpat". In: *Proceedings of the 2015 Workshop on Rapid Simulation and Performance Evaluation: Methods and Tools*. ACM. 2015, p. 7.

[25] Fernando A Endo, Damien Couroussé, and Henri-Pierre Charles. "Micro-architectural simulation of in-order and out-of-order ARM microprocessors with gem5". In: *Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS XIV), 2014 International Conference on*. IEEE. 2014, pp. 266–273.

[26] Jakob Engblom. "Full-system simulation". In: *Proceedings of the European Summer School on Embedded Systems (ESSES 2003)* (2003).

[27] Xiaocong Fan. *Real-Time Embedded Systems: Design Principles and Engineering Practices*. Newnes, 2015.

[28] Yi-yuan Fang and Xue-jun Chen. "Design and simulation of UART serial communication module based on VHDL". In: *Intelligent Systems and Applications (ISA), 2011 3rd International Workshop on*. IEEE. 2011, pp. 1–4.

[29] Michael J Flynn. *Computer architecture: Pipelined and parallel processor design*. Jones & Bartlett Learning, 1995. ISBN: ISBN 0867202041.

[30] *GNU toolchain for RISC-V, including GCC*. Accessed: 27.07.2018. 2017. URL: `https://github.com/riscv/riscv-gnu-toolchain`.

[31] Christopher Guntli. "Architecture of clang". In: *Analyze an open source compiler based on LLVM* (2011).

[32] Anthony Gutierrez, Joseph Pusdesris, Ronald G Dreslinski, Trevor Mudge, Chander Sudanthi, Christopher D Emmons, Mitchell Hayenga, and Nigel Paver. "Sources of error in full-system simulation". In: *Performance Analysis of Systems and Software (ISPASS), 2014 IEEE International Symposium on.* IEEE. 2014, pp. 13–22.

[33] Craig C Hansen and Thomas J Riordan. *RISC computer with unaligned reference handling and method for the same.* US Patent 4,814,976. 1989.

[34] John Hennessy, Norman Jouppi, Steven Przybylski, Christopher Rowen, Thomas Gross, Forest Baskett, and John Gill. "MIPS: A microprocessor architecture". In: *ACM SIGMICRO Newsletter.* Vol. 13. 4. IEEE Press. 1982, pp. 17–22.

[35] Alireza Hodjat and Ingrid Verbauwhede. "Interfacing a high speed crypto accelerator to an embedded CPU". In: *Signals, Systems and Computers, 2004. Conference Record of the Thirty-Eighth Asilomar Conference on.* Vol. 1. IEEE. 2004, pp. 488–492.

[36] Lee W Howes, Paul Price, Oskar Mencer, Olav Beckmann, and Oliver Pell. "Comparing FPGAs to graphics accelerators and the PlayStation 2 using a unified source description". In: *Field Programmable Logic and Applications, 2006. FPL'06. International Conference on.* IEEE. 2006, pp. 1–6.

[37] SiFive Inc. *SiFive FE310-G000 Manual.* Tech. rep. SiFive Inc., Oct. 2017.

[38] SiFive Inc. *SiFive HiFive1 Getting Started Guide.* Tech. rep. SiFive Inc., Jan. 2017.

[39] SPARC International Inc and David L Weaver. *The SPARC architecture manual.* Prentice-Hall, 1994.

[40] Open SystemC Initiative et al. "IEEE standard SystemC language reference manual". In: *IEEE Computer Society* (2006), pp. 1666–2005.

[41] Open Virtual Platform Initiative et al. *OVPsim instruction set simulators.*

[42] *JavaScript RISC-V ISA Simulator. Boots linux in a web-browser.* Accessed: 26.07.2018. URL: `https://github.com/riscv/riscv-angel`.

[43] Sagar Karandikar. *Structure of the RISC-V So0ware Stack.* Accessed: 30.07.2018. Jan. 2015. URL: `https://riscv.org/wp-content/uploads/`

2015/01/riscv-software-stack-bootcamp-jan2015.pdf.

[44] Manolis GH Katevenis, Robert W Sherburne Jr, David A Patterson, and Carlo H Séquin. “The RISC II micro-architecture”. In: *Advances in VLSI and Computer Systems* 1.2 (1984), pp. 138–152.

[45] Manuel "Koschuch, Joachim Lechner, Andreas Weitzer, Johann Großschädl, Alexander Szekely, Stefan Tillich, and Johannes Wolkerstorfer. “Hardware/-Software Co-design of Elliptic Curve Cryptography on an 8051 Microcontroller”. In: *Cryptographic Hardware and Embedded Systems - CHES 2006*. Ed. by Louis Goubin and Mitsuru Matsui. 2006, pp. 430–444.

[46] Rakesh Kumar, Dean M Tullsen, Norman P Jouppi, and Parthasarathy Ranganathan. “Heterogeneous chip multiprocessors”. In: *Computer* 38.11 (2005), pp. 32–38.

[47] Yunsup Lee. “RTL-to-Gates Synthesis using Synopsys Design Compiler”. In: (2010).

[48] Yunsup Lee, Andrew Waterman, Rimas Avizienis, Henry Cook, Chen Sun, Vladimir Stojanović, and Krste Asanović. “A 45nm 1.3 GHz 16.7 double-precision GFLOPS/W RISC-V processor with vector accelerators”. In: *European Solid State Circuits Conference (ESSCIRC), ESSCIRC 2014-40th*. IEEE. 2014, pp. 199–202.

[49] Yunsup Lee, Andrew Waterman, Henry Cook, Brian Zimmer, Ben Keller, Alberto Puggelli, Jaehwa Kwak, Ruzica Jevtic, Stevo Bailey, Milovan Blagojevic, et al. “An agile approach to building risc-v microprocessors”. In: *IEEE Micro* 36.2 (2016), pp. 8–20.

[50] Yunsup Lee, Brian Zimmer, Andrew Waterman, Alberto Puggelli, Jaehwa Kwak, Ruzica Jevtic, Ben Keller, Stevo Bailey, Milovan Blagojevic, Pi-Feng Chiu, et al. “Raven: A 28nm risc-v vector processor with integrated switched-capacitor dc-dc converters and adaptive clocking”. In: *Hot Chips 27 Symposium (HCS), 2015 IEEE*. IEEE. 2015, pp. 1–45.

[51] Qing Li and Caroline Yao. *Real-time concepts for embedded systems*. CRC Press, 2003.

[52] Sheng Li, Jung Ho Ahn, Richard D Strong, Jay B Brockman, Dean M Tullsen, and Norman P Jouppi. “McPAT: an integrated power, area, and timing modeling framework for multicore and manycore architectures”. In: *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*. ACM. 2009, pp. 469–480.

[53] Gabriel H Loh, Samantika Subramaniam, and Yuejian Xie. “Zesto: A cycle-level simulator for highly detailed microarchitecture exploration”. In: *Performance Analysis of Systems and Software, 2009. ISPASS 2009. IEEE International Symposium on*. IEEE. 2009, pp. 53–64.

[54] Stefan Mach, Davide Rossi, Giuseppe Tagliavini, Andrea Marongiu, and Luca Benini. “A Transprecision Floating-Point Architecture for Energy-Efficient Embedded Computing”. In: *Circuits and Systems (ISCAS), 2018 IEEE International Symposium on*. IEEE. 2018, pp. 1–5.

[55] Milo MK Martin, Daniel J Sorin, Bradford M Beckmann, Michael R Marty, Min Xu, Alaa R Alameldeen, Kevin E Moore, Mark D Hill, and David A Wood. “Multifacet’s general execution-driven multiprocessor simulator (GEMS) toolset”. In: *ACM SIGARCH Computer Architecture News* 33.4 (2005), pp. 92–99.

[56] *Members at a Glance*. Accessed: 14.04.2018. URL: `https://riscv.org/members-at-a-glance/`.

[57] Christian Menard, Jeronimo Castrillon, Matthias Jung, and Norbert Wehn. “System simulation with gem5 and systemC: the keystone for full interoperability”. In: *Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS), 2017 International Conference on*. IEEE. 2017, pp. 62–69.

[58] Alian Mohammad, Umur Darbaz, Gabor Dozsa, Stephan Diestelhorst, Daehoon Kim, and Nam Sung Kim. “dist-gem5: Distributed simulation of computer clusters”. In: *Performance Analysis of Systems and Software (ISPASS), 2017 IEEE International Symposium on*. IEEE. 2017, pp. 153–162.

[59] Fuentes Morales and Jose Luis Bismarck. *Evaluating Gem5 and QEMU Virtual Platforms for ARM Multicore Architectures*. Accessed: 05.05.2018. 2016.

[60] Apurv Nerlekar, Rishabh Sreedhara, Rishikesh Nagare, and Shivakumar Soppannavar. *Comparison between RISC architectures: MIPS, ARM and SPARC.* Accessed: 28.07.2018. May 2015. URL: `https://www.slideshare.net/ApurvNerlekar1/cmpe-200-ppt-47929145`.

[61] Tony Nowatzki, Jaikrishnan Menon, Chen-Han Ho, and Karthikeyan Sankaralingam. "gem5, gpgpusim, mcpat, gpuwattch," your favorite simulator here" considered harmful". In: (2014).

[62] Avadh Patel, Furat Afram, Shunfei Chen, and Kanad Ghose. "MARSS: a full system simulator for multicore x86 CPUs". In: *Proceedings of the 48th Design Automation Conference.* ACM. 2011, pp. 1050–1055.

[63] David Patterson. "Reduced Instruction Set Computers Then and Now". In: *Computer* 50.12 (2017), pp. 10–12.

[64] David A Patterson and Carlo H Sequin. "RISC I: A reduced instruction set VLSI computer". In: *Proceedings of the 8th annual symposium on Computer Architecture.* IEEE Computer Society Press. 1981, pp. 443–457.

[65] Srivaths Ravi, Anand Raghunathan, Paul Kocher, and Sunil Hattangady. "Security in embedded systems: Design challenges". In: *ACM Transactions on Embedded Computing Systems (TECS)* 3.3 (2004), pp. 461–491.

[66] *RISC-V Linux Port.* Accessed: 11.07.2018. URL: `https://github.com/riscv/riscv-linux`.

[67] *RISC-V Opcodes.* Accessed: 30.06.2018. URL: `https://github.com/riscv/riscv-opcodes`.

[68] Alec Roelke and Mircea R Stan. "Risc5: Implementing the RISC-V ISA in gem5". In: *First Workshop on Computer Architecture Research with RISC-V (CARRV).* 2017.

[69] Laurent Sauvage, Sofiane Takarabt, and Youssef Souissi. "Secure silicon: Towards virtual prototyping". In: *Electromagnetic Compatibility-EMC EUROPE, 2017 International Symposium on.* IEEE. 2017, pp. 1–5.

[70] Stephen Schaub and Brian A Malloy. "Comprehensive Analysis of C++ Applications using the libClang API". In: *International Society of Computers and Their Applications (ISCA)* (2014).

[71] Yakun Sophia Shao, Sam Likun Xi, Vijayalakshmi Srinivasan, Gu-Yeon Wei, and David Brooks. “Co-designing accelerators and soc interfaces using gem5-aladdin”. In: *Microarchitecture (MICRO), 2016 49th Annual IEEE/ACM International Symposium on*. IEEE. 2016, pp. 1–12.

[72] *Spike, a RISC-V ISA Simulator*. Accessed: 26.07.2018. URL: `https://github.com/riscv/riscv-isa-sim`.

[73] Nitish Srivastava. *A Tutorial on the Gem5 Minor CPU Model*. Accessed: 23.07.2018. July 2017. URL: `https://nitish2112.github.io/post/gem5-minor-cpu/`.

[74] Jürgen Teich and Christian Haubelt. *Digitale Hardware/Software-Systeme: Synthese und Optimierung*. Springer-Verlag, 2007.

[75] Andreas Traber and Michael Gautschi. “PULPino: Datasheet”. In: *ETH Zurich and University of Bologna* 63 (2016), p. 64.

[76] Andreas Traber, F Zaruba, S Stucki, A Pullini, G Haugou, E Flamand, FK Gürkayank, and L Benini. “PULPino: A small single-core RISC-V SoC”. In: *3rd RISCV Workshop*. 2016.

[77] Leon Urbas. “Mikrorechentechnik 1, Befehlssatzarchitektur”. Accessed: 16.04.2018. URL: `https://www.et.tu-dresden.de/ifa/uploads/media/MRT1-003_Befehlssatzarchitektur_01.pdf`.

[78] Michael Vai, Ben Nahill, Josh Kramer, Michael Geis, Dan Utin, David Whelihan, and Roger Khazan. “Secure architecture for embedded systems”. In: *High Performance Extreme Computing Conference (HPEC), 2015 IEEE*. IEEE. 2015, pp. 1–5.

[79] Ashish Venkat and Dean M Tullsen. “Harnessing ISA diversity: Design of a heterogeneous-ISA chip multiprocessor”. In: *ACM SIGARCH Computer Architecture News* 42.3 (2014), pp. 121–132.

[80] Andrew Waterman and Krste Asanovic. “The RISC-V Instruction Set Manual Volume I: User-Level ISA”. In: *CS Division, EECE Department, University of California, Berkeley* (May 2017).

[81] Andrew Waterman and Krste Asanovic. “The RISC-V Instruction Set Manual Volume II: Privileged Architecture”. In: *CS Division, EECE Department, University of California, Berkeley* (May 2017).

[82] Andrew Waterman, Yunsup Lee, and Christopher Celio. *EA RISC-V proxy kernel and boot loader*. Tech. rep. Tech. rep. EECS Department University of California Berkeley, 2015.

[83] Andrew Shell Waterman. *Design of the RISC-V instruction set architecture*. University of California, Berkeley, 2016.

[84] Clifford Wolf. *PicoRV32 - A Size-Optimized RISC-V CPU*. Accessed: 16.07.2018. URL: `https://github.com/cliffordwolf/picorv32`.

[85] Joe Xie. “NVIDIA RISC V Evaluation Story”. Accessed: 16.04.2018. July 2016. URL: `https://www.youtube.com/watch?v=gg1lISJfJI0`.

[86] Matt T Yourst. “PTLsim: A cycle accurate full system x86-64 microarchitectural simulator”. In: *Performance Analysis of Systems & Software, 2007. ISPASS 2007. IEEE International Symposium on*. IEEE. 2007, pp. 23–34.

[87] Shaza Zeitouni, Ghada Dessouky, Orlando Arias, Dean Sullivan, Ahmad Ibrahim, Yier Jin, and Ahmad-Reza Sadeghi. “Atrium: Runtime attestation resilient under memory attacks”. In: *Proceedings of the 36th International Conference on Computer-Aided Design*. IEEE Press. 2017, pp. 384–391.

[88] Dieter Zöbel. *Echtzeitsysteme: Grundlagen der Planung*. Springer-Verlag, 2008.

[89] Dan Zuras, Mike Cowlishaw, Alex Aiken, Matthew Applegate, David Bailey, Steve Bass, Dileep Bhandarkar, Mahesh Bhat, David Bindel, Sylvie Boldo, et al. “IEEE standard for floating-point arithmetic”. In: *IEEE Std 754-2008* (2008), pp. 1–70.

Selbstständigkeitserklärung

Hiermit versichere ich, Robert Scheffel, geboren am 04.01.1994 in Görlitz, dass ich die vorliegende Diplomarbeit zum Thema

Simulation of RISC-V based Systems in gem5

ohne unzulässige Hilfe Dritter und ohne Benutzung anderer als der angegebenen Hilfsmittel angefertigt habe; die aus fremden Quellen direkt oder indirekt übernommenen Gedanken sind als solche kenntlich gemacht. Bei der Auswahl und Auswertung des Materials sowie bei der Herstellung des Manuskripts habe ich Unterstützungsleistungen von folgenden Personen erhalten:

Dipl.-Ing. Christian Menard, Dipl.-Ing. Gerald Hempel

Weitere Personen waren an der geistigen Herstellung der vorliegenden Diplomarbeit nicht beteiligt. Mir ist bekannt, dass die Nichteinhaltung dieser Erklärung zum nachträglichen Entzug des Diplomabschlusses (Masterabschlusses) führen kann.

Dresden, den 20.08.2018

....

Unterschrift