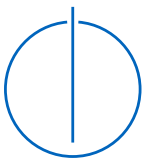
DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics

Extending the RISC-V Memory Management Unit by Support for a Real-Time Capable Checkpoint/Restore Mechanism

Aaron Dietz



DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics

Extending the RISC-V Memory Management Unit by Support for a Real-Time Capable Checkpoint/Restore Mechanism

Erweiterung der RISC-V Memory Management Unit um Unterstützung für einen echtzeitfähigen Checkpoint/Restore Mechanismus

Author:	Aaron Dietz
Supervisor:	Prof. Dr. Uwe Baumgarten
Advisor:	Sebastian Eckl, M.Sc.
Submission Date:	15.02.2018

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

Munich, 15.02.2018 Aaron Dietz

Acknowledgments

First of all, I want to thank Prof. Dr. Uwe Baumgarten for making this bachelor thesis possible in the first place. I am also very grateful for the support of Sebastian Eckl and Alexander Weidinger for their continued support of my work. Thanks to the people of the RISC-V and RISCV-BOOM mailing lists, especially to Bruce Hoult, Farzad Farshchi, Abraham Gonzalez and Andrew Waterman as they greatly helped me to understand the RISC-V ISA and the Rocket Chip SoC generator better. Thanks a lot to Tabea Pape for carefully reading my work and making insightful comments.

Abstract

Modern processors are used in nearly every application today, and they are getting more and more important. User depend increasingly on the correct function of modern machines. The software of a self-driving car, for example, is a very critical application that should not completely crash. The same holds for modern medical devices and other mission-critical appliances. To reduce the impact of hardware failures and other unforeseen events systems are build with redundant hardware. In the case of failure, the redundant part of the system can take over. Depending on the system this step can take valuable time, as the system needs to initialize and boot up to the state where the previous system has failed. Checkpoint/Restore mechanisms are put in place to reduce the impact of such defects further, as a checkpointed system only loses the progress since the last checkpoint in case of a crash.

As most of the current Checkpoint/Restore mechanisms are implemented in software, they introduce certain bottlenecks into the system. Especially the tracking of changes to checkpointed memory and the copying of such memory to conserve previous states has proven to be costly. The most currently available hardware merely is not made with Checkpoint/Restore systems in mind. To accelerate critical systems hardware acceleration have since been used. Hardware modules that are connected over the system bus or other connections might not be fast enough for high-speed systems. Thus this thesis will look in the possibility to directly modify components in the CPU to accelerate Checkpoint/Restore algorithms further. To freely modify a CPU, one needs an open CPU architecture. In this thesis, the upcoming RISC-V instruction set architecture was chosen as it looks promising and can be freely obtained. The RISC-V ISA is a free blueprint of a processor architecture and was initially developed by the University of California, Berkeley. As for the implementation part of this thesis, we will use three kinds of Xilinx FPGA boards, the included development suite Xilinx Vivado and the RISC-V implementation Rocket Chip as it appears to be the most popular RISC-V project as of the time of this thesis. Later this thesis will present a Checkpoint/Restore mechanism that builds on the modification of the memory management unit (MMU) in the CPU. We will explore how the Rocket Chip project has integrated the MMU in their Rocket cores and how we could apply the proposed concept. Due to time constraints, the concept could not be implemented.

Contents

1 Introduction

Developing a website has become very easy in recent years. A basic model of a personal site can be made with a decent amount of work and an HTML template. Even such complicated software as a whole operation system (Linux for example) is open-sourced and available to everybody with an internet connection. Linux is even the most used operating system for web servers worldwide [80]. Developing a CPU on the other side is an entirely different story. Almost all CPU architectures are closed source and owned by large companies. The execution of most modern CPUs cannot be tracked, in fact, many contain a so-called secure zone or trusted zone that handles DRM content and security-related work like storing cryptography keys [45]. A general distrust against these proprietary zones has risen among enthusiast users that motivated some users so far that they researched the Intel Active-Management-Technology (AMT) and found a security vulnerability by which they could completely disable the functionality [18]. However, these more or less official hidden layers are of small concern when we consider the thread of undocumented backdoors built into the hardware. Modern x86 processors contain hidden subprocessors that might have unrestricted access to the execution of the main processor. One security researcher found such a backdoor and used it to gain root access on an old VIA x86 system [11]. Furthermore, the undocumented features are not the only problem of the current generation of the processor: in January of 2018, two new security vulnerabilities surfaced that deeply shattered the security of many processors. Spectre [40] hit AMD processors, Intel processors and many other chips, that use out-of-order execution and Meltdown [42] hit manly Intel processors with speculative execution (starting from 1995, nearly all processors). These flaws, for the most part can be fixed with an update, but the current implementation of the fixes lowers the performance of the system and are thus not ideal. New hardware mitigations have to be implemented. Apart from the user perspective, the situation for producers of electronic components is not ideal either. Many manufacturers need processors to power their products, of which a prominent example is Nvidia. Graphics cards have become very complex and thus also need co-processors. Developing a new processor for every new product is not feasible for these companies. Currently, they need to buy processors from vendors. The problem with this approach is that the bought architecture often cannot be easily extended with a needed extension or an accelerator which a new product needs let alone it is even possible to extend it. These

bought custom solutions are not cheap either.

Many companies and associations have tried to solve the issues of developing a flexible and open ISA, but no other has reached as far as the Berkeley University of California with their RISC-V design. It is designed to be easily extendable and said to fit microcontrollers, as well as to many core massive computing processors like the SuperMUC NG in Garching uses. Safety concerns can also be addressed if the code of a RISC-V implementation is published and audited. No Spectre or Meltdown vulnerability has yet been seen in a RISC-V implementation although this statement heavily relies on the correctness of the implementation. This bachelor thesis will dive deeper into the extendability of the RISC-V design and the implementation of the design from the Berkeley University. With the new possibility of extending a CPU design, we want to look into the probability of implementing a Checkpoint/Restore mechanism in hardware.

2 Motivation

The chair of operating systems at the Technical University Munich is currently refining an operating system called Genode that should be fault tolerant and hardened against attackers. The system should be able to support real-time applications in the field of car electronics. One essential part of making this system fault tolerant is to implement the technique of Checkpoint/Restore for the RAM system. This technique makes it possible to manage a complete failure of one unit with little to no noticeable downtime and small additional computational expenses as long as a mirror unit is available. The previous implementations of the Checkpoint/Restore technique have been implemented in software and thus are slowing the system down, as fast access to memory is crucial for modern computer systems [6]. When searching for new ways to implement a hardware solution for this technique, one soon comes across the RISC-V architecture. This instruction set architecture (ISA) is completely open source and can be modified by everybody. It is built to be modular and expandable. Major big companies like Nvidia [54] and Western Digital [55] joined the RISC-V consortium and want to use the new ISA extensively. Western Digital as a Platinum member has already developed and open sourced a RISC-V instruction set simulator called Whisper [22], [30] and a RISC-V implementation called SweRV Core [82], [29]. Western Digital ships more than one billion processors each year and plans to migrate the ARM-based processors to RISC-V [57]. With the new possibility of directly customizing a processor new ways of implementing a Checkpoint/Restore mechanism can be found, that do not suffer from drastic performance reductions. Even though custom hardware modules exist, a direct implementation in the CPU is beneficial, as it can be faster.

Taking the traction the RISC-V project has gained in recent years this topic looks very promising.

3 Related work

In this chapter, we want to revisit previous papers that already have found a hardware or software solution for the Checkpoint/Restore Mechanism and discuss how well their solution could fit a RISC-V approach. We first want to analyze the basic principle of the approaches and then classify them for the use in this thesis. The selection in inherently limited, as the number of hardware-focused approaches to Checkpoint/Restore is limited.

3.1 Sheaved Memory

3.1.1 Analysis

Sheaved memory is discussed in a paper from 1989 "Sheaved memory: architectural support for state saving and restoration in pages systems" [70] that was published by Mark E. Staknis from the Northeastern University of Boston, Massachusetts. Sheaved memory is based on grouped copies of memory pages called sheaves and the concept of "read-one/write-many paged memory". Before the execution starts, a count of levels has to be specified. Available are two, four or eight levels which translate to 2-way, 4-way, and 8-way sheaved memory. The levels determine the count of page files that are group together to form on "sheave", and thereby the amount of available checkpoint states the system can hold. Sheaved memory can be turned on or off process-wise. At the start of execution, all levels are "grouped" together, which means all write operations are executed on all levels in a sheave. Reads are not influenced by this and are executed on the top level of a sheave. If a program wants to create a checkpoint, one level is excluded from the group of levels and will not be written in a future write operation. The state in this level of the sheave is conserved. If the future execution of the program requires a new checkpoint, it depends on the previously chosen count of levels if another checkpoint can be saved or a previous checkpoint has to be sacrificed for a new checkpoint. Returning to a previous checkpoint is as simple as making a previous level the new top level. All future operations will be executed on this new level. If the previous checkpoint needs to be retained, a copy has to be made which should be ungrouped from the new top level.

3.1.2 Classification

Sheaved memory makes it easy for the operating system to checkpoint and restore memory. Read operations should not be slower compared to an unaltered system and write operations should also not suffer if the sheaved memory module can perform the write operation in parallel. Otherwise, the write performance could suffer by the factor of levels that need to be written. The model has two major downfalls: memory consumption and inflexibility. Every checkpoint that might be used in the future consumes memory from the start of the application. In the worst case, eight times the memory of the usual memory requirement is needed to run an application, turning 8GB of available RAM into 1GB. Also, the initially set count of levels cannot be changed, and thus the count of available checkpointing slots is inherently limited. With improvements to the memory consumption and the flexibility of the amount of available checkpointing slots, this implementation could be the basis of the RISC-V approach to the Checkpoint/Restore Mechanism.

3.2 Checkpoint/Restore with Hardware Cache

3.2.1 Analysis

One big problem of Checkpoint/Restore techniques is the additional use of memory for checkpointed data elements. A patent with the number US2016357645 [36] with the name "HARDWARE-ASSISTED APPLICATION CHECKPOINTING AND RESTORING" defines a physical Checkpoint Cache that is connected to the I/O system of the processor. This Checkpoint Cache is used to speed up the process of checkpointing a section of memory. The patent defines no technique like copy-on-write, so all the original data of the checkpoint is copied out of the RAM and into an external medium. If a restore of the data is needed the data is copied back from the external cache into the RAM. This method conserves RAM space in exchange for slower checkpoint and restore operations.

3.2.2 Classification

A full copy of the checkpointed data is only beneficial if the area of memory is changing very heavily. If a less active or mixed activity part of the memory is checkpointed, a copy-on-write approach could be beneficial. However, the idea of copying the checkpointed data of the volatile RAM is favorable if the system suffers a hardware defect or any other event that interrupts the execution.

3.3 Compiler-based Checkpoint/Restore

3.3.1 Analysis

If the source code of a program is available and editable compiler-based Checkpoint/Restore can be used. This method does not involve changes to the hardware, but hokes into the process of the compilation of the program and inserts logic that performs the checkpoint and restore process. A programmer should understand the process to implement it correctly. The tracking process of changes can be done in many ways, but we will list three of them here [79]:

Page-granular Checkpointing

This classic approach utilizes a feature of modern memory management units (MMUs) called Copy-on-Write (COW). This way only write operations trigger a significant additional memory usage for a checkpoint. Read operations on a copied unaltered memory page that was duplicated by the CoW operation are simply redirected to the original memory page the new page was copied from. A simple way to implement the actual checkpointing process on a Unix system is to execute the system call fork in the execution of the program. The child process should then go into a sleep mode, waiting for a restore command. All COW semantic is delegated to the kernel which makes this approach easy to implement if the operating system and the host hardware is capable of COW.

Undo Log

The undo log checkpoint technique keeps a record of all previous write operations and intercepts future read operations which are redirected to the newly written value. If a checkpoint should be restored the log history has to be processed for changes to the memory. The log history can grow to a considerable size as it needs to contain the memory address, size of the change and the content of the previous memory entries.

Shadow State

The method of shadow state splits the memory into a primary state (which contains the original data), a shadow state (changed data since the last checkpoint) and a tagmap which keeps track of the changes. The tagmap contains tags that describe if a certain part in the shadow state memory contains a changed copy of the data of the original state. New write operations are performed on the shadow copy and marked as modified in the tagmap. The shadow copy can either be completely initialized as a copy of the

original state (COW is recommended to save memory) or can manually grow with an algorithm designed by the developer that allocates memory for every piece of data that has not already been in the shadow state.

3.3.2 Classification

Compiler-based approaches to Checkpoint/Restore are interesting, as they do not require a change to the existing hardware. An approach with the possibility of changing the hardware, like this thesis, can still use the core concepts of tracking changes, that are included in the compiler-based Checkpoint/Restore methods.

3.4 Incremental Checkpoint/Restore

3.4.1 Analysis

The tracking of checkpointed memory is one of the key challenges of a Checkpoint/Restore system. A system can, for example, keep a big undo log dataset like we have seen in the previous section of compiler-based Checkpoint/Restore mechanism or find another way to track the changed parts of the memory. An interesting approach is presented in the paper "Comparing different approaches for Incremental Checkpointing: The Showdown" [77]. They propose to repurpose the Write or Dirty bit of a page table entry to intercept the normal read and write operations of the system.

Dirty Bit

The dirty bit is normally responsible for marking a page as modified. If such a page is to be swapped out, it needs to be written back to the secondary memory. A clean page that exists in the secondary memory can be deleted without harm, as it has not been modified. In the presented approach the kernel is extensively modified to disable the dirty page check, and the bit is reused to checkpoint memory pages [77]. The approach is called Berkeley Lab Checkpoint/Restart (BLCR).

Write Bit

The write bit of page table entry normally marks a page as writable to the system. If a write attempt is performed on a page with a disabled write bit, a soft page fault is risen. This mechanism can be exploited for the tracking of modifications to checkpointed memory pages. Every page that should be tracked gets a deactivated write bit. Every following write access to the page can now be tracked by catching the soft page fault. The Linux system itself uses this mechanism to implement the Copy-on-Write (CoW)

mechanism. CoW allows the system to copy a page without duplicating the data it contains virtually. Read operations will be redirected to the original page table entry until a write operation is performed, which then triggers the full duplication of the data to the new location.

3.4.2 Classification

The approaches are interesting, as they utilize the already existing mechanisms to make the tracking of changes to checkpointed pages easier and faster. Sadly, they come with performance penalties, as the write and dirty bit are normally utilized by the system to speed up certain memory operations that now need to be emulated. If the hardware itself can be modified, these previous methods could inspire a new approach.

3.5 FPGA accelerated Checkpoint/Restore

3.5.1 Analysis

Sebastian Bachmaier presents in his master thesis a new way of accelerating Checkpoint/Restore operations [6]. After he identified the bottlenecks of previous implementations, he developed an FPGA coprocessor for the main ARM processor of a Xilinx board. The coprocessor is connected over the AXI interconnect with the ARM processor. All memory access operations that would normally be handled directly by the memory are redirected through the AXI connection to the FPGA. The FPGA then fulfills the memory accesses and handles Checkpoint/Restore operations.

3.5.2 Classification

As Bachmaier in his master thesis already found, the main bottleneck of the system is the AXI interconnect, as now every memory operations need to be rerouted. This introduces a delay and bandwidth limitation to every memory access performed by the system. If one could accelerate the AXI interconnect or replace it with a faster technique this method could be useful for processors with integrated accelerator modules.

3.6 VM live migration

3.6.1 Analysis

Moving virtual machines (VMs) from host to host is a common occurrence in modern data centers. Overload conditions, hardware, and software maintenance are reasons

that make this technique very valuable. Even energy can be saved if for example one host can be freed of VMs and then be shut down. To minimize downtime different techniques have emerged to make transfers as fast as possible.

Pre-Copy

The method of Pre-Copy tries to transfer as much data as possible before the complete transfer to the new machine starts while the VM is still running on the old host. In the beginning, one complete RAM image will be transferred to the new host. As the VM on the host is still running the content of the RAM will change. This new changed partitions of the RAM will be transferred again and again to the new host until a certain threshold of already transferred and freshly changed RAM on the old host is reached. After this threshold is reached, the VM is halted, the last bits of memory are transferred, and the execution is continued on the new host. If for some reason the new host becomes unavailable the VM can be simply continued on the old host. [64]

Post-Copy

The method of Post-Copy starts with transferring a minimal set of data to the new host, continues the VM on the new host and sequentially transfers the RAM content from the old host. When the VM now accesses the RAM it will always hit a page fault because only a minimal information set has already been transfered to the new host. The page faults will be resolved by a daemon that prioritizes the pages that are needed. The VM performance will most likely suffer at the start of the transfer as the transmission of data introduces a delay. Also if the new host fails for any reason, a restore is not easily possible. The advantage of this method is that the data has to be transferred only once [65].

3.6.2 Classification

As the techniques are specialized for migrating memory and not checkpointing it, they can not directly be adopted. Still, the method of Pre-Copy could be evaluated in the sense that one might want to consider doing as many nonblocking things before one impacts the execution of the program. Post-Copy could be used in the sense that one might start a checkpoint process but does not stop the execution. Instead, one could lock the memory access and process the resulting page locks by prioritizing the corresponding memory sections in the checkpointing process.

4 Foundation

4.1 Memory Management Unit (MMU)

[71] The memory management unit is a hardware component that manages the mapping of virtual addresses to physical addresses. The CPU passes all memory references through the MMU and thus has not to manage the address translation. This principle can be graphically observed in figure 4.1. The component is mostly placed directly in the CPU but can also be a separate integrated chip (mainly old models). Modern MMUs partition the virtual address space into pages of size power of 2. As the MMU handles all translations of page addresses for the CPU, a sufficiently fast translation is important. To achieve this at least for a limited amount of highly accessed pages in many implementations a so-called translation lookaside buffer (TLB) is used, as can be seen in figure 4.1.

Most MMU implementations use a so-called *page table* to map the virtual pages to the physical memory. One entry of the page table is called *page table entry* and references the virtual page to a page frame in the physical memory. Virtual pages can be swapped out to disk if there is no physical space left. Such *page table entries* do have an empty reference. If the CPU tries to access such a virtual page, the MMU notices this access and raises a trap called page fault. This trap causes the operating system to take a previously used page frame to be written to the disc and replace the previous content

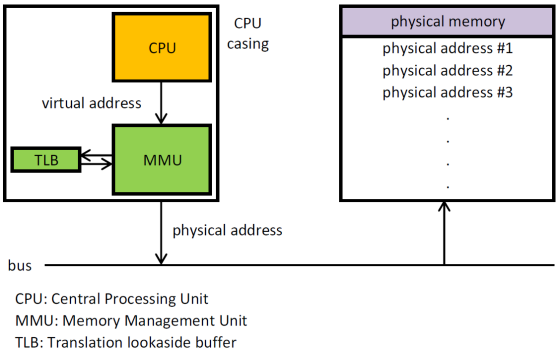


Figure 4.1: Simplified view on a MMU [71, p. 186ff]

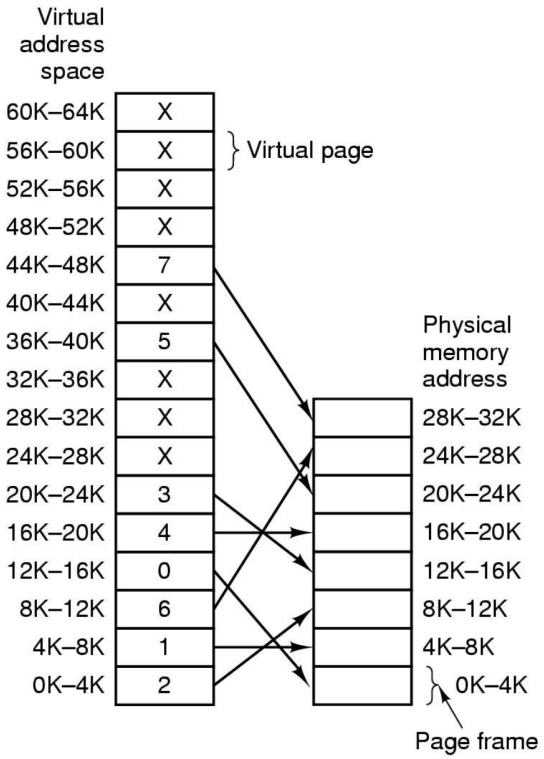


Figure 4.2: Page table representation [71, p. 188]

of this physical memory space to be replaced with new content. This principle allows for a nearly unlimited virtual address space and is especially beneficial for systems with many concurrent tasks that do not need to run at the same time. A representation of a simplified page table can be seen in figure 4.2. Each page in this example contains 4096 addresses.

4.2 Checkpoint/Restore Technique

Checkpoint/Restore [9], in general, describes the process of capturing a certain state of a system and being able to restore this state at a later stage. Checkpointing a system is especially beneficial for processes that are executed for a long time and are very critical. In a checkpointed system only the progress since the last checkpoint is lost. This technique can mitigate hardware failures and other transient errors, but there are limits. Design and programming errors cannot be solved, as the system returns to the previous state before the crash and continues the same execution that led to a crash

previously. The system will be stuck in an infinite loop. Checkpoint/Restore can be used for the whole system or just parts of it. A good example for the former one is the snapshot function of a hypervisor software like Oracle Virtualbox, QEMU or many other hypervisor programs which also checkpoint the state of the file system.

4.3 FPGA

To better understand what an FPGA is it will be compared to an ASIC and a CPU. A Central Processing Unit (CPU) is a very generic processor and loads its program code from memory. It is, in its basic form, not optimized for specific tasks but made for an extensive range of tasks. Parallelization can only be achieved by adding more CPU cores. Application-specific integrated circuits (ASICs) are highly optimized for a minimal scope of predefined tasks. Algorithms are hardwired and cannot be changed. ASICs often contain multiple circuits with the same functionality to speed up the execution. The combination of hard wired circuits and massive parallel execution make ASICs very energy efficient. Many processors are paired with an ASIC to speed up a certain use case. Video acceleration circuits, for example, can be found in nearly every mobile processors [4].

An FPGA is composed out of so-called configurable logic blocks (CLBs) that performs a certain calculation operation. In figure 4.3, an example of a CLB can be seen. The schematic contains two triple input Lookup tables (LUTs), a full adder (FA) with a carry in and carry out signal and a D-type Flip Flop (DFF) with a controlling clock signal (clk). The LUTs in this example define which input (a, b or c) is forwarded to the FA. Therefore it defines the combinatorial logic of the computing block. The FA sums up the two signals plus the carry in signal it receives and puts out a value plus the carry out signal for future calculations in other blocks. As the last step, the DFF reacts to the clk and sends the result of the calculation of the FA out. This is one example of a CLB, other implementations are possible. To process data, an FPGA needs to group many CLBs and connect many of them in a meaningful matter. A simplified schematic of such an FPGA can be seen in figure 4.4. Let us have a look at one of the many quartets of blocks out of which the core of this FPGA is formed. One quartet is formed out of one configurable logic block (CLB) which we already know, two connect boxes (CBs) and one switch box (SB). The CBs selectively connect the outputs and inputs of CLBs or SBs to interconnection channels. The SBs also connect channels but are more flexible, as they allow horizontal and vertical connections. At the edges of the FPGA core, input-output elements (I/Os) can be seen, which are responsible for inputting and outputting data into and from the FPGA core. Other setups are possible. The SBs could, for example, be exchanged for more CLBs.

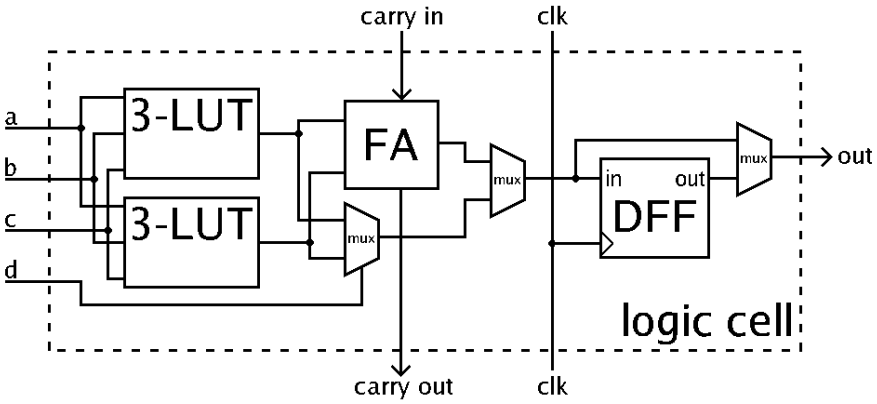


Figure 4.3: Example of a configurable logic block (CLB) [47]

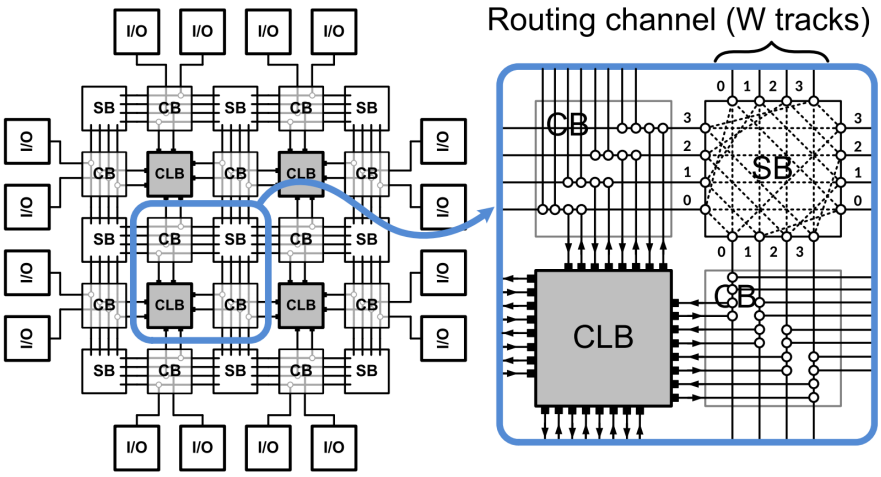


Figure 4.4: Simplified FPGA [50, p. 6]

When an FPGA is started, no command can be executed before the instructions for the CLBs, CBs, SBs, and the I/O elements are loaded from some form of memory or manually loaded by a programming adapter. Through this initial step, an internal network is formed.

5 RISC-V

RISC-V is a free and open-source instruction set architecture (ISA) that was from the start designed to be modular and extendable. In contrast to many other ISAs, it can be used for commercial and non-commercial projects and is freely editable. This is possible because the ISA is released under the Berkley Software Distribution (BSD) licenses which do allow free usage of the ISA and does not require anyone to release sources for a custom build RISC-V implementation [53]. RISC-V is remarkable because it is designed for a vast range of applications. The architecture can be tuned for microcontrollers, that operate in conditions of limited resources, as well as for complex multicore systems with custom extensions [7]. The architecture comes with rich software support including simulators, compilers (GCC, LLVM, ...), kernels (seL4, Linux, ...), operating systems (Genode, Debian, ...) and many other things. More about the software in chapter 5.4.

5.1 History

A predecessor of RISC-V was the academical RISC instruction set DLX. It was introduced in 1996 by Hennessy and Patterson and was mainly targeted teaching purposes [8].

In 2010 a new architecture emerged at the University of Berkeley. Krste Asanović decided to develop and publish a new architecture that was not only aimed at research but also for broader applications [5]. The project gained traction and grew over the years. In 2015 the RISC-V Foundation was founded which continued to develop the RISC-V ISA. Today the organization has over 100 members. A small selection of some platinum members:

- Berkeley Architecture Research [1]
- Google [2]
- NVIDIA [3]

[1] `https://riscv.org/membership/1564/berkeley-architecture-research/`
[2] `https://riscv.org/membership/999/google/`
[3] `https://riscv.org/membership/1437/nvidia/`

- Qualcomm [4]
- Samsung [5]
- SiFive [6]
- Western Digital [7]

[56]

5.2 Base ISA

The general architecture of RISC-V is based on a load-store architecture, which means that "only load and store instructions access memory and arithmetic instructions only operate on CPU registers" [73, p. 18]. The minimal set of instructions is called the ISA base. In the following table 5.1 all current bases can be seen.

Name	Description	Version	Status
RV32I	Base Integer Instruction Set, 32-bit	2.0	Frozen
RV32E	Base Integer Instruction Set (embedded), 32-bit, 16 registers	1.9	Open
RV64I	Base Integer Instruction Set, 64-Bit	2.0	Frozen
RV128I	Base Integer Instruction Set, 128-Bit	1.7	Open

Table 5.1: RISC-V ISA base [73, p. i & v-vi]

5.2.1 RV32I

RV32I is the basic instruction set for a 32-bit operating system. The "I" at the end of the name indicates that this set includes integer operations. The architecture specifies 32 general-purpose registers in integer size and a 32-bit user address space. RV32I is frozen, which means that the design will not change in the future.

5.2.2 RV32E

RV32E is the embedded implementation of RV32I and an attempt to "provide an even smaller base core for embedded microcontrollers" [73, p. 27]. The main difference is

[4] https://riscv.org/membership/1585/qualcomm/
[5] https://riscv.org/membership/1849/samsung/
[6] https://riscv.org/membership/1017/sifive/
[7] https://riscv.org/membership/1020/western-digital/

that the register count is reduced to 16 registers and not all extensions can be applied to the design. The following extensions can be applied:

- Atomic instructions (A)
- Compressed instructions (C)
- Integer multiplication & division instructions (M)

RV32E is not frozen and can be subject to change in the future.

5.2.3 RV64I

The RV64I instruction set builds on RV32I and extends it with larger registers and a larger user address space (both 64-bit). Some additional instructions are included to operate on 32-bit values explicitly. RV64I is already frozen.

5.2.4 RV128I

A higher address space mostly means a higher amount of addressable memory. As 64 bit can already address 2^{64} bytes = 16 exabytes currently there is no real need for a 128-bit variant, but in the future, this might be a useful extension. The 128-bit variant of the base instruction set extends RV64I like RV64I did extend RV32I. It is not frozen, as currently, the expertise for 128-bit real-world applications is missing.

5.3 Extensions

As the base ISAs are kept very minimal (even integer multiplication is excluded) to enable the implementation of very minimal cores, standard extensions have been defined to enable faster hardware implementations. If an implementation includes certain extensions, the letter is suffixed behind the base ISA (e.g. RV32IC for a standard 32-bit base ISA with the compressed instructions extension). The available standard extensions (their version and their status can be seen in table 5.2) followed by the explanations of the extensions.
The following information was taken from the RISC-V Instruction Set Manual Volume I: User-Level ISA [73].

5.3.1 Atomic instructions

The base versions of RISC-V already support the sharing of memory between multiple hardware threads (harts). The atomic instructions extension adds additional commands

Name	Description	Version	Status
A	Atomic instructions	2.0	Frozen
B	Bit manipulation instructions	0.36 [20]	Open
C	Compressed instructions	2.0	Frozen
D	Double-precision floating-point instructions	2.0	Frozen
F	Single-Precision floating-Point instructions	2.0	Frozen
G	General (combination out of I,M,A,F,D)	2.0	Frozen
J	Dynamically translated languages	0.0	Open
L	Decimal floating point instructions	0.0	Open
M	Integer multiplication & division instructions	2.0	Frozen
N	User-level interrupt instructions	1.1	Open
P	Packed-SIMD instructions	0.1	Open
Q	Quad-precision floating-point instructions	2.0	Frozen
T	Transactional memory instructions	0.0	Open
V	Vector operation instructions	0.2	Open

Table 5.2: RISC-V ISA extensions [73, p. i & vi-viii]

that help to make handling conditions easier and faster directly in the hardware. The specification has been frozen. Two groups of commands are added:

- Load-Reserved/Store-Conditional instructions
- Atomic memory operations

Load-Reserved/Store-Conditional instructions

These instructions implement commands to place a reservation on a memory word to avoid race conditions and undefined states. These reservations can be broken by other harts, but the original hart will be notified that his lock is no longer valid, when it tries to perform a write operation. This prevents deadlock situations.

Atomic memory operations

This instruction group abstracts the steps of performing a modification of value into one command. They are called read-modify-write operations. AMOADD, for example, takes in a source address and an integer value. The operation will load the value from the source address into a register, apply an add operation on the value in the register and the given integer value and writes the result back to the source address.

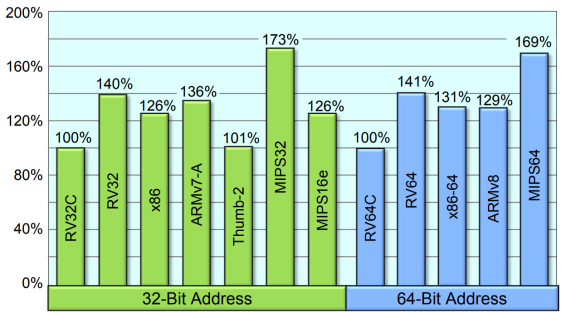


Figure 5.1: Comparison of SPECint2006 compilation sizes for different ISAs [38, p. 4], [60, p. 16]

5.3.2 Bit manipulation instructions

The B extension, until now, is a placeholder for instructions that operate on bit size level. These instructions are not finalized and subject to change.

5.3.3 Compressed instructions

Compressing instructions has two major benefits:

- Smaller binary size
- Smaller size in cache

The smaller size of the binary files especially is beneficial if the target environment has resource restrictions (small embedded devices). The second benefit is important for high-speed computing applications. High-speed computers, in general, depend on extensively fast connected storage. This storage is called cache memory and quite limited in size, as it is expensive to build. Compressed instructions make it possible to store more information in this predefined memory space. This accelerates the execution. In figure 5.1 a comparison of the compiled binary file size of the benchmarking program SPECint2006 can be seen.

5.3.4 Single-Precision floating-Point instructions

Describing this before the Double-precision floating-point instructions as the double-precision extension builds on the single-precision extension.
This extension implements the classic 32-bit floating-point logic compliant with the IEEE 754-2008 arithmetic standard [35]. To execute the calculations 32 32-bit registers and one status register are added.

5.3.5 Double-precision floating-point instructions

This extension builds on the F extension and widens it from a 32-bit to 64-bit size. To hold the larger word sizes the 32 registers from the F extension are expanded to 64-bit. The F registers are now capable of holding either 32-bit or 64-bit. The D extension requires the F extension.

5.3.6 General

As a large number of extensions can be overwhelming and could lead to fragmentation of implementations on the market, the "general-purpose" ISA extension General was created. General does not add any additional instructions but groups the extensions IMAFD together, so a "general-purpose" RISC-V CPU includes:

- Base integer instructions
- Integer multiplication & division instructions
- Atomic instructions
- Single-Precision floating-Point instructions
- Double-precision floating-point instructions

5.3.7 Dynamically translated languages

The dynamically translated languages extension is a placeholder for future standard extensions to support dynamically translated languages like Java and Javascript. "These languages can benefit from additional ISA support for dynamic checks and garbage collection" [73, p. 87].

5.3.8 Decimal floating-point instructions

The decimal (base-10) floating-point instructions have not been further specified and the L extension until now is a placeholder. Still, it is defined that the extension should be implemented according to the 754-2008 arithmetic standard [35].

5.3.9 Integer multiplication & division instructions

These instructions were separated from the base ISAs to enable low-end implementations that either emulate multiplication and division instructions or source them out to separate chipsets. The instructions in these extensions are basic integer multiplication and division instructions.

5.3.10 User-level interrupt instructions

This extension enables the ISA to forward interrupts and exceptions directly to user-level without triggering a context switch. These instructions could be used for garbage collection barriers, integer overflow or floating-point traps.

5.3.11 Packed-SIMD instructions

The Single instruction multiple data (SIMD) instructions allow massive parallel computing on homogeneous data types (for example: adjusting the brightness of a picture). This extension does reuse the floating-point registers and thus does not need additional registers but depends on the F extension. The extension is not finished and might be dropped for a standardized form of the V extension for SIMD operations.

5.3.12 Quad-precision floating-point instructions

The quad-precision floating-point extension widens the floating-point registers of the previous floating-point extensions to 128-bit. The extension requires the RV64I base ISA and the extensions F and D.

5.3.13 Transactional memory instructions

This instructions are a placeholder and thus not frozen. Transactional memory is supposed to make working with multiple threads easier, as syncing memory between threads can be a big challenge [32].
"Despite much research over the last twenty years and initial commercial implementations, there is still much debate on the best way to support atomic operations involving multiple addresses. Our current thoughts are to include a small limited-capacity transactional memory buffer along the lines of the original transactional memory proposals" [73, p. 89].

5.3.14 Vector operation instructions

This extension is in an early stage and represents a proposal for a RISC-V vector extension. "The vector extension supports a configurable vector unit, to trade off the number of architectural vector registers and supported element widths against available maximum vector length" [73, p. 93]. This enables different sizes of implementations to handle the same binary code. The minimal implementations can be built small, as they can split up the commands into multiple smaller commands internally. Larger implementations can utilize parallelism to speed up the execution.

5.3.15 Future extensions

At the Santa Cara Convention Center on the 2018-12-04 Krste Asanovic listed some future extensions that will come to the RISC-V ISA: [61, p. 14].

- Fast interrupts (CLIC) - stable, ratifiable in 2019?
- V extension (Vectors) - getting stable, ratifiable in 2019?
- Hypervisor extension - stable, ratifiable in 2019?
- Crypto (builds on vectors) - builds on vectors
- TEE (trusted execution environments) - refocused
- B extension (Bit Manipulation) - recently restarted
- J extension (dynamic translation support) - ongoing discussion
- P extension (DSP extensions) - recently started
- Sv128 (Large secure address space) - recently started

As the one-character naming scheme limits the amount of possible names for future extensions, multiple characters for an abbreviation of an extension are now allowed.

5.4 Software

The amount of software that was built or extended for RISC-V has grown tremendously. In the following section, some projects that support RISC-V will be presented. This list is far from containing all RISC-V capable software, as a complete list with full descriptions would be out of scope for this bachelor thesis. A larger list can be found here: [8].

5.4.1 Simulators

To develop and test an architecture, simulators are an important quick start option. For RISC-V already exist eleven simulators with varying functionality [59]. Some of these are described in the following.

[8] `https://riscv.org/software-status/`

QEMU

One prominent emulator is the open source machine emulator quick emulator (QEMU) which also powers large virtualization solution like Proxmox [49]. QEMU can be obtained from the project GitHub site and built on most Linux machines [24]. QEMU is a functional emulator for the RISC-V ISA which means it directly translates RISC-V instructions to the host CPU instructions [66]. The downside of this approach is that the emulation does not provide an instruction-by-instruction trace. This would normally help to debug the code.

Spike

A (outside of the RISC-V world) less know emulator is the Spike emulator. The Spike emulator is considered the "golden reference" simulator [66] for the RISC-V ISA. It is working on a trace accurate basis, which means the emulator can give the developer hints if debugging is needed. Additionally, it is built to be easily editable. The speed compared to QEMU is about ten times slower (QEMU: 100 million to >1 billion instructions per second versus Spike: 10 to 100 million instructions per second [66]).

FireSim

Another approach is lead by FireSim. FireSim is an "FPGA-Accelerated Cycle-Exact Scale-Out System Simulation in the Public Cloud" [39] paper title. FireSim uses FPGAs in the Amazon cloud to simulate one or multiple RISC-V implementations like Berkeley Out-of-Order Machine (BOOM), Rocket-Chip and more. As FPGAs often run below the clock rates of the target tape-out environment, FireSim supports delaying I/O operations to simulate correct hardware behavior. A schematic of a simulated RISC-V quadcore can be seen in figure 5.2. The core simulation runs at 150 MHz with a 40 MHz network simulation. The processor sees itself as running at 3.2 GHz. The RAM access and the network access are scaled down to the speed of the simulation. The observed time of the processor runs $3200MHz/150MHz = 21,\overline{3}$ times slower than real time. This project enables developers to test their implementation on costly FPGA equipment for a low lease price. Small instances with one Xilinx UltraScale+ FPGA [83] start at an "On-Demand" price of 1.65 USD [2]. In the presentation at the Chisel Community Conference a whole datacenter simulation was shown that runs 4096 RISC-V cores grouped on 1024 "servers" on 256 FPGAs [10, min. 12:08]. FireSim is an open-source project.

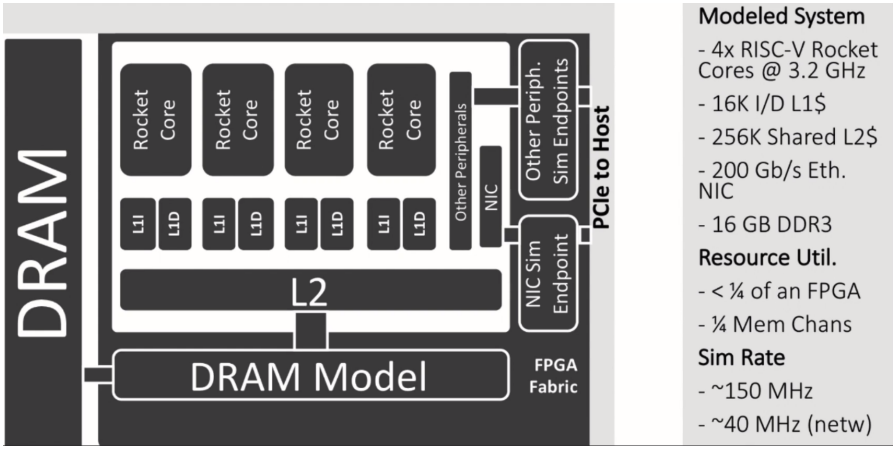


Figure 5.2: Schematic of a RISC-V processor simulated in FireSim [10, min. 9:00]

Web-Emulators

With the advancements of JavaScript over the last years it is even possible to run a RISC-V emulation in the web browser. The projects riscv-angel [25] can boot a Linux kernel with Busybox installed on an RV64IMA implementation directly in the browser in about 3 seconds. The system is very minimal but still functional. Another project named JSLinux [15] can even boot Fedora with 3D output, although the system takes a very long time to boot up. The low performance might stem from the fact that the developer implemented an RV128IMAFDQC core. A 128-bit architecture is hard to emulate, especially in the environment of JavaScript which is well optimized but still introduces some limitations.

5.4.2 Toolchain, compilers and libraries

Official RISC-V toolchain

The official RISC-V GNU Compiler Toolchain [21] bundles a good set of tools to compile and emulate C and C++ code for RISC-V. It contains:

- QEMU, (Emulator)
- Newlib (A C library)
- Glibc (GNU C library)
- GCC (GNU C compiler)

- DejaGnu (Test framework)
- GDB (GNU Debugger)

One can download this toolchain and compile it with the argument *--with-arch=rv32gc* for a 32-bit version of the ISA with the extensions general (I,M,A,F,D) and compressed. The toolchain can be set up to link the C libraries Newlib and Glibc automatically. Newlib is needed if the program should run on an FPGA with I/O devices, but no Linux environment and Glibc is needed if the program should run under Linux in user mode [43]. The resulting product of the compilation can be executed in the included QEMU version, Spike or other emulators. To eliminate bugs and find problems the GNU debugger (GDB) and the test framework DejaGnu can be used.

LLVM

Apart from the standard C compiler GCC the widely used LLVM compiler can be used. The developer itself recommends to use GCC until the RISC-V LLVM support is included in the official LLVM release as this makes the process "slightly more user friendly" [23] at "Should I be compiling my code with Clang and the RISC-V LLVM backend?". However, the development of the implementation can already compile and run the "GCC torture suite", which shows that it is indeed usable [23].

5.4.3 Kernels

Linux Kernel

The base of Linux is the Linux Kernel. Starting with version 4.15 in January 2018 the RISC-V version of the kernel was labeled "stable" [33] and with version 4.19 additional patches were added to support timers and first-level interrupt controllers [48].

seL4

seL4 is a microkernel that has been formally proven to be functional correct on ARM systems [63]. This kernel port is especially interesting for the chair of operating systems at the Technical University Munich as they heavily work with the related L4 microkernel. Such a kernel makes it possible to ensure high security and fail-safety, as microkernel separate nearly all processes from the kernel. This way, the access right of a process can be limited, and a failed process can easily be restarted.

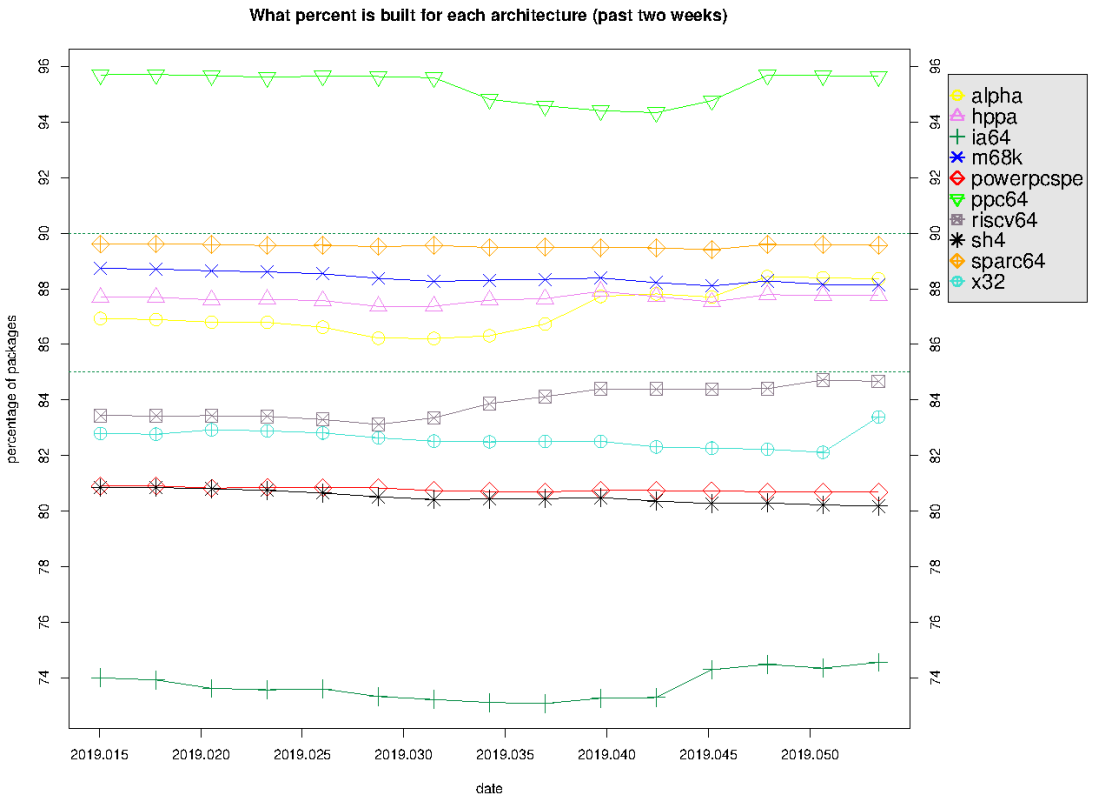


Figure 5.3: Debian-ports architectures [13, "Two weeks report"] accessed on 2019-01-22

5.4.4 Operating systems

Linux kernel based OSs

Many efforts have been made to port operating systems to RISC-V on the base of the ported Linux kernel. Fedora, Debian, OpenSUSE, Gentoo, and OpenWRT have at least partially been ported to the RISC-V environment. The developers of the port of the distribution Debian have also started to port the packages to RISC-V. The progress can be seen in figure 5.3. Until today (2019-01-22) already 84 percent of all Debian source packages have been ported. About 9000 of the 13000 packages that are architecture dependent are already building fine, which is already higher than the number of packages that are available for the Intel Itanium architecture (ia64 in figure 5.3) [86, min. 5:35].

Genode

Genode is a free operating system framework that can run on many kernels:

- Linux kernel
- NOVA microhypervisor
- Base hardware implementation
- Mue separation kernel
- seL4 microkernel
- Fiasco.OC microkernel
- L4ka::Pistachio microkernel
- OKL4 microkernel

[17]
For RISC-V the base hardware implementation of Genode has been ported. It should also be possible to use the Linux kernel as it is also being ported to RISC-V. The Genode team described the porting of Genode as a "rough ride" [16]. They encountered multiple issues with the RISC-V toolchain being out of sync with the specification and additional problems with newer pushed versions of GCC that they could not use for their project [16]. It seems that in the early stages of development when Genode started working on the port for RISC-V (2015), the ISA was under heavy development. The most recent presentation of "RISC-V State of the Union" at the Santa Clara Convention Center by Krste Asanovic [61] suggests that in 2019 this situation will be better, as the RV32 and RV64 bases and the General extension have been frozen.

5.5 Implementations

There exist many implementations ranging from minimal educational implementations to larger scalable multi-core ones, even Western Digital recently released their new SweRV kernel which will be used in SSD in the future [3]. This section will mainly talk about the implementations from the University of Berkeley because they cover the complete spectrum from simple microcontrollers to more advanced out-of-order processors. The University of Berkeley currently develops three different implementations of the RISC-V ISA.

- Sodor (Collection of simple educational processors)

- Rocket (In-Order processor)
- Berkeley Out-of-Order Machine (Out-of-order Processor)

All of the projects use the Constructing Hardware in a Scala Embedded Language (Chisel) as a programming language. As the name indicates, this language is implemented in Scala.

5.5.1 Scala

Scala is a general-purpose programming language with a strong object-oriented static type system which it inherited from Java [1]. The language uses the Actor model to simplify concurrent programming and make use of many threads, hence the name Scala from scalable.

Actor model [34]: The actor model is a model that enables concurrent computing. In the model, one process is an isolated actor. An actor can:

- Make local decisions and computations
- Create more actors
- Send messages to other actors

One actor can not directly modify the state of another actor. He can only send them messages.

Scala builds on Java and thus can use the wide ranges of Java libraries. It also runs on the JVM. Programming in Scala is expressive and might sometimes give a feeling of functional programming. A simple print loop, for example, looks like this

```
for (i <- Array(1,2,3)) println(i)
```

Modification of the Array is achieved via the yield *command* or the *map* command

```
for (i <- Array(1,2,3)) yield i * 2
-> Array(2,4,6)

for (i <- Array(1,2,3)).map(_ * 2)
-> Array(2,4,6)
```

Examples taken from the brilliant Scala Cookbook [1, p. xiv-xv].
Other functions like *.filter* are also available. Scalas first version 1 was released in 2003

and has since been developed to version 2.12. Many companies use Scala to power their website, including Twitter, Netflix, Tumblr, LinkedIn, Foursquare, and many more [1, p. xiii]. The language is best used with the Simple Build Tool (sbt) that can automate the build process. It builds on the folder structure of Maven and thus can use existing Maven packages.

5.5.2 Chisel

With Chisel it is possible to develop an FPGA design without writing one line of Verilog or VHDL but, in the end, have a synthesized design in one of these languages. Chisel can transform the code into Verilog, VHDL, and C++. Until now the development focus was the Verilog ouput [9]. The language supports three primitive types:

- UInt (Unsigned integer)
- SInt (Signed integer)
- Bool (Boolean value)

which can be declared as variables like this:

```
val number = UInt(42)
val answer = Bool(true)
```

A user can expand this limited types with so-called bundles:

```
class MyFloat extends Bundle {
    val sign = Bool()
    val exponent = UInt(width=8)
    val significand = UInt(width=23)
}
```

Example taken from [52, p. 11]
The class in the example above extends itself from the base class bundle and defines three values. Two of the variables define an unsigned integer of the size 8 and 23. If a new data type is created, new included functions can also be defined. Below a implementation of a 2D-vector can be seen:

```
class 2D-Vector(val first: SInt, val second: SInt) extends Bundle {
    def + (b : 2D-Vector): 2D-Vector =
        new 2D-Vector(first + b.first, second + b.second)
    def - (b : 2D-Vector): 2D-Vector =
```

[9] https://stackoverflow.com/questions/49587973/converting-chisel-to-vhdl-and-systemc

```
        new 2D-Vector(first - b.first, second - b.second)
    ...
}

val x = new 2D-Vector(SInt(20), SInt(32))
val y = new 2D-Vector(SInt(10), SInt(45))
val result = x - y
-> 2D-Vector(10, -13)
```

The class above defines two operations, namely addition + and subtraction -. The definitions are executed below the definition. For more complex procedures a module can be defined:

```
class Stack(val depth: Int) extends Module {
    val io = new Bundle {
        val push = Bool(INPUT)
        val pop = Bool(INPUT)
        val en = Bool(INPUT)
        val dataIn = UInt(INPUT, 32)
        val dataOut = UInt(OUTPUT, 32)
    }
    val stack_mem = Mem(UInt(width = 32), depth)
    val sp = Reg(init = UInt(0, width = log2Up(depth+1)))
    val dataOut = Reg(init = UInt(0, width = 32))
    when (io.en) {
        when(io.push && (sp < UInt(depth))) {
            stack_mem(sp) := io.dataIn
            sp := sp + UInt(1)
        } .elsewhen(io.pop && (sp > UInt(0))) {
            sp := sp - UInt(1)
        }
        when (sp > UInt(0)) {
            dataOut := stack_mem(sp - UInt(1))
        }
    }

    io.dataOut := dataOut
}
```

This is an implementation of a stack with a data element width of 32 bit. At first, the *io*

variable with definitions of output and input wires are defined, then the stack memory with a predefined size (*depth*) is declared. At last we define the stack pointer (*sp*) with a width of $\log_2 n$ (so its values can address the complete *stack_mem*) and the data output (*dataOut*) as register variables (*Reg()*, variables retain values till they are changed). After the definition of variables the stack logic on the enable signal (*io.en*) is defined. This stack implementation can also be tested directly in Chisel.

```
class StackTests(c: Stack) extends Tester(c) {
    var nxtDataOut = 0
    val stack = new ScalaStack[Int]()
    for (t <- 0 until 16) {
        val enable = rnd.nextInt(2)
        val push = rnd.nextInt(2)
        val pop = rnd.nextInt(2)
        val dataIn = rnd.nextInt(256)
        val dataOut = nxtDataOut
        if (enable == 1) {
            if (stack.length > 0)
                nxtDataOut = stack.top
            if (push == 1 && stack.length < c.depth) {
                stack.push(dataIn)
            } else if (pop == 1 && stack.length > 0) {
                stack.pop()
            }
        }
        poke(c.io.pop, pop)
        poke(c.io.push, push)
        poke(c.io.en, enable)
        poke(c.io.dataIn, dataIn)
        step(1)
        expect(c.io.dataOut, dataOut)
    }
}
```

Example taken from [52, p. 45]
A significant feature of Chisel is the great configurability. More on that in subsection 5.5.4.

5.5.3 Sodor

Sodor is a group of educational implementations of the RISC-V ISA from the university Berkeley. The five implementations employ the basic RV32I variant of the base ISA and support no virtual memory (to keep them as simple as possible). The cores can be differentiated by the count of stages they use:

- 1-stage (essentially an ISA simulator)
- 2-stage (demonstrates pipelining in Chisel)
- 3-stage (uses sequential memory)
- 5-stage (can toggle between fully bypassed or fully interlocked)
- "bus"-based micro-coded implementation

All Sodor core use Chisel as a programming language and can be downloaded on the GitHub page [26]. The 3-stage Sodor implementation was renamed in 2015 to Z-scale [62] and was developed further as a smaller alternative to the Rocket Chip SoC generator. Since 2017-03-30, the repository has received no update, and the project is deprecated [31]. According to an user on StackOverflow [69], a similar result to a Z-scale core can be generated by running the Rocket Chip Generator with the "TinyConfig" configuration.

```
cd vsim
make verilog CONFIG=TinyConfig
```

5.5.4 Rocket Chip

Rocket Chip is a highly flexible open source generator written in Chisel and maintained by SiFive and the University of Berkeley. It generates so-called tiles that each contains one Rocket core, private caches and optional ROCC accelerators that surround the core. To ensure communication a fitting Uncore module with coherence agents shared caches, DMA engines and memory controllers are added, and everything is "glued" together [58].

Tiles

A tile forms one minimal computing complex in a Rocket Chip. A quad-core design, for example, will contain four tiles. Every tile will have their own private instruction and data L1 caches, as can be seen in figure 5.4. The variables *sets* and *ways* in the

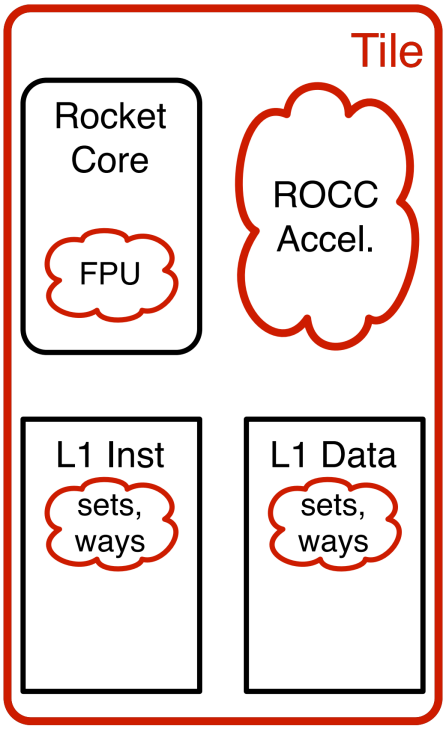


Figure 5.4: A Tile with a Rocket core inside [58, p. 3]

cloud in the figure define how large the cache is and how the connection is handled. A typical configuration would be:

```
case CacheName("L1D") => CacheConfig(
nSets = 64,
nWays = 4,
rowBits = site(L1toL2Config).beatBytes*8,
nTLBEntries = 8,
cacheIdBits = 0,
splitMetadata = false)
```

To speed up certain computation tasks an FPU can be added directly in the Rocket core, or a ROCC accelerator is included in the tile. A nice example of such an accelerator is shown by the Celerity [12], [46] that was created from 20 graduate students in a combined effort of the University of Michigan, UC San Diego, the University of Washington and the Cornell University. Their design houses 511 RISC-V cores with different levels of complexity. 5 Linux capable, general-purpose RV64G Rocket cores control the system and execute the operating system. Next to them, 496 RV32IM Rocket cores are packaged in a mesh tiled array together to form the manycore tier. Ten additional low voltage cores are placed next to the manycore package. To accelerate the computation further, the Accelerating Binarized Neural Networks are added as a third specialization tier, which can communicate with the manycore tier and the general-purpose tier. The construction can be seen in 5.5. The ROCC interface makes it possible for developers to easily include accelerators in a Rocket Chip architecture.

Interconnects

As it can be seen in figure 5.5 general-purpose tier tiles are connected with NASTI interfaces to the rest of the hardware on the system (memory, I/O, ...). **N**ot **A** **ST**andard **I**nterface (NASTI, pronounced nasty) is the implementation of the University of Berkeley of the AXI-standard from Xilinx. It is necessary to use this interface if one wants to develop designs on Xilinx FPGA hardware, but it is not the only interface available:

- TileLink
- NASTI (pronounced nasty)
- HASTI (pronounced hasty)
- POCI (pronounced pokey)

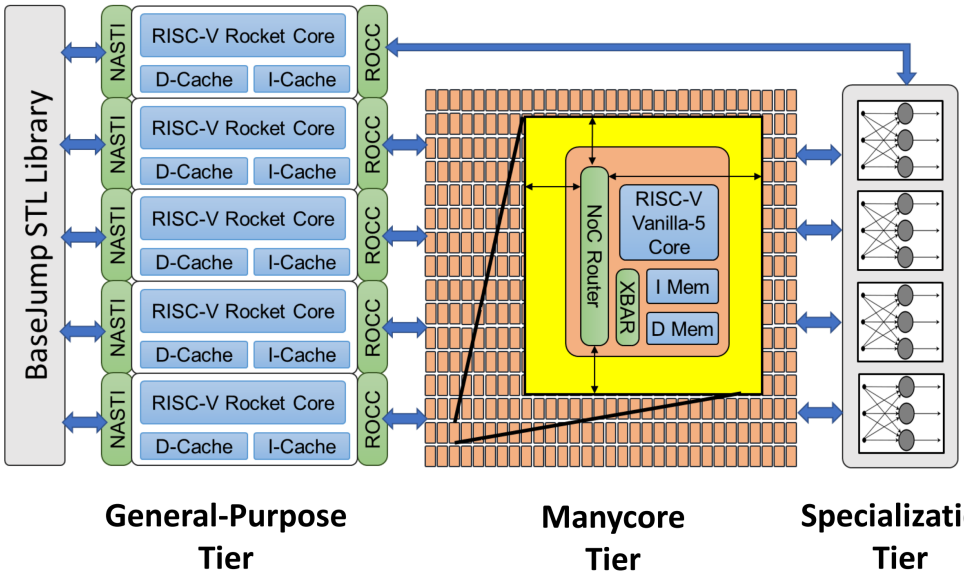


Figure 5.5: Overview of the Celerity architecture [46]

TileLink will be explained in the next subsection. Highly Advanced System Transport Interface (HASTI) is the implementation of the University of Berkeley for the AHB-Lite standard from ARM. A similar implementation is the Peripheral Oriented Connection Interface (POCI) which is the implementation of the AHB standard, from ARM. The last two standards are used for the communication with ARM cores that are bound to the AHB/AHB-lite standard [62].

TileLink

When the development of Rocket-Chip started the developers were confronted with a problem. They needed a bus system that fulfills the following requirements:

- Open standard
- Easy to implement
- Cache-coherent block motion (Moving cache blocks around)
- Multiple cache layers
- Reusable on- and off-chip
- High performance

Table 5.3: Comparison of TileLink conformance levels [68, p. 10]

	TL-UL	TL-UH	TL-C
Read/Write operations	x	x	x
Multibeat Messages		x	x
Atomic operations		x	x
Hint (prefetch) operations		x	x
Cache block transfers			x
Priorities B+C+E			x

They considered AHB, Wishbone, AXI4, ACE and CHI as standards for the bus system, but none of them fulfilled all requirements that were previously set. Additionally, if they would rely on a standard from a competitor (ARM: AHB, CHI, ACE; Xilinx: AXI4, Opencores(Silicore): Wishbone), they could change the standard or hold back documents to slow down the development. So the team decided to reuse an academic already existing bus protocol called TileLink and develop it further. TileLink is a Master-Slave protocol, with 5 priorities (A,B,C,D,E). Messages can be sent out-of-order with optional ordering. It was designed to be composable and deadlock free. TileLink comes in three versions that build on each other, shown in table 5.3. The written-out names of the conformance levels are:

- TL-UL: TileLink uncached lightweight
- TL-UH: TileLink uncached heavyweight
- TL-C: TileLink cached

It is possible to use multiple levels on the same chip for different components like it is shown in the schematic of the SiFive FU500 design in figure 5.6.

5.5.5 Berkeley Out-of-Order Machine (BOOM)

Berkeley Out-of-Order Machine (BOOM) builds on top of the Rocket Chip SoC ecosystem and expands it with out-of-order execution. BOOM consists of about 16 thousand lines of Chisel code and is, as the Rocket Chip generator, an open-source chip generator. BOOMv1.0 was first released on 14. July 2017 and replaced by BOOMv2 on 16. August 2017. Since then the v2 version has matured to version 2.2.0. The use of out-of-order execution, should in the future, speed up the computations of the processor.

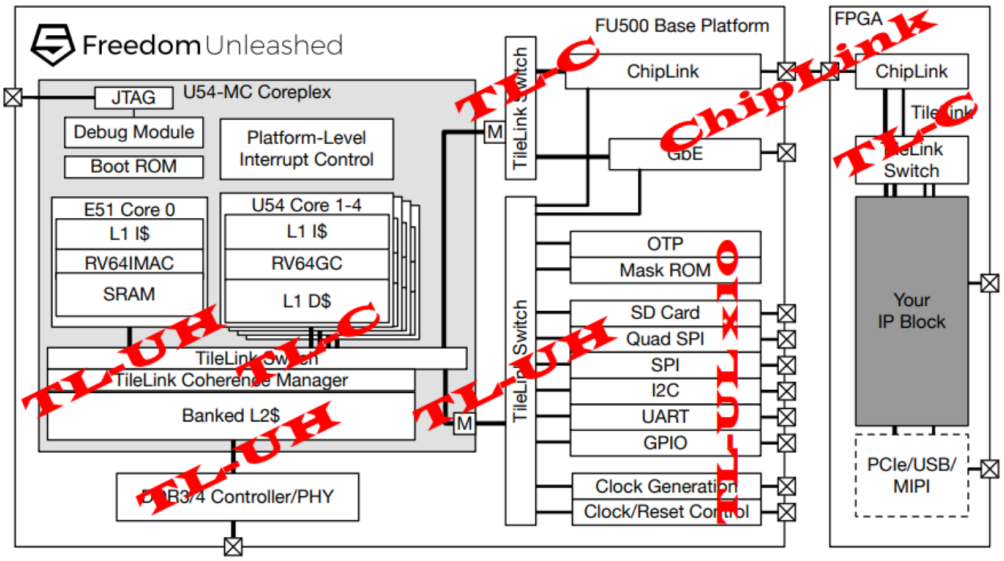


Figure 5.6: Schematic of SiFive FU500 [68, p. 25]

5.6 Comparing implementations

As the interest in RISC-V grows, so does the count of implementations of the RISC-V ISA in software and hardware. In the following diagram mainly RISC-V implementations are compared against ARMs solutions and one custom soft microprocessor solution from the manufacturer Xilinx. Processors need to adapt to the problems they need to solve, and as there is an uncountable amount of problems, a comparison between the corresponding processors could be infinitely large as well. The following comparison groups the processors by their included features to get a better overview:

1. No FPU, No MMU: Very basic microcontroller, mostly used in applications where power is very restricted.
2. FPU, No MMU: Limited processor for embedded systems that need a hardware acceleration but often are focused on a limited set of tasks.
3. FPU, No MMU but PMP: Larger processor for embedded systems that runs mostly more than one process and thus needs a certain level of delimitation between processes.
4. FPU, MMU: Processors capable of running a general purpose operating system like Linux that uses virtual address spaces.

Abbreviations used:
FPU = Floating Point Unit, MMU = Memory Management Unit, PMP = Physical Memory Protection

Table 5.4: Comparison of chip implementations

Implementation	No FPU No MMU	FPU No MMU	FPU No MMU but PMP	FPU MMU
ARM M0, M0+, M1, M3	X			
ARM M4, M7, M33, M35P	X	(optional)		
ARM R-Series	X	(optional)	(optional)	
ARM A-Series				X
SiFive E-Series	X	(optional)		
SiFive S-Series			X	
SiFive U-Series				X
Pulp Platform Micro-Riscy, Zero-Riscy	X			
Pulp Platform RI5CY		X	(optional)	
Pulp Platform Ariane				X
Greenwaves GAP8	X			
Kendryte K210				X
Xilinx MicroBlaze (no RISC-V)				X

5.7 Competition

As the market for processors architecture has grown over the years, RISC-V is faced with many competitors. Some of these are described in the following subsections.

5.7.1 ARM

ARM is one of the largest competitors of RISC-V. Many smaller ARM chips are built into embedded systems, which can be replaced by a RISC-V processor. ARM is away of its new competitor and launched a marketing campaign website against RISC-V in July 2018 which promoted the disadvantages of the RISC-V architecture. The campaign website has since been taken down after internal and external protests [72]. The website has also been removed from *web.archive.org* [10], were a cached version was still accessible shortly after the reports were published.

5.7.2 MIPS

MIPS is like RISC-V a RISC architecture and has been acquired in June 2018 by the AI Startup Wave Computing [75]. Wave Computing announced that in the first quarter of 2019 they would open up the ISA for other developers and further development will happen under the new MIPS Open initiative [81]. In contrast to RISC-V Wave computing still requires a sign up to access the ISA and extensions to it. The MIPS ISA could be a significant competitor to RISC-V, as it has already been taped out many times, also runs the Linux kernel and is now free to use.

5.7.3 x86 and other high-performance architectures

RISC-V currently has not yet achieved a tap-out of high performance. The Linux capable SiFive HiFive Unleashed board with its four cores at 1.5GHz [51] is already a step in the right direction but far from a desktop pc grade chip. SiFive also has announced the U74-MC processor with up to eight cores, which is said to perform equally to an ARM-A55 (that is used in many mobile phones) [41]. RISC-V will be a competitor for high-performance chips, but the implementation of this new version are not ready yet.

[10] `https://web.archive.org/web/20180708231736/https://riscv-basics.com/`

6 Modify a Rocket Chip RISC-V

When it comes to implementing a feature in hardware one needs to choose a modifiable processor implementation. In this thesis, the Rocket Chip project was chosen, as it seems to be most popular RISC-V implementation as of the time of this thesis. The University of Berkeley and SiFive maintain the Rocket Chip SoC generator and the Rocket core. Thus the project has great support from the founders and from a growing company that already has taped out the Rocket cores twice. The programming language of the Rocket Chip project Chisel also seems to be a good fit for the job of modifying the processor. The wide configurability of the SoC generator also allows us to work on a minimal base processor without additional distractions. The rocket-chip GitHub repository can be found here [1].

6.1 SiFive web configuration

For a company that wants to evaluate RISC-V processors the entry point of the programmed Chisel configuration might be too high. SiFive is creating a solution to this problem. They are currently developing a product called SiFive Core Designer which makes configuring a processor easy. In the current state of the website, we can choose from three series of processors with different submodels. (ordered from low to high performance)

- E20, E21, E24, E2 custom
- E31, E34, E3 custom
- S51, S54, S5 custom

[67]
These models predefine the size of the design and approximate which ARM chip is comparable to it. The website is still in the preview phase, and thus not all SiFive processors are available. Especially the higher performance models will be available in the future. When we selected a processor type, the following site presents us with a UI in which we can choose what capabilities the processor should have. A screenshot of the site can be seen in figure 6.1. We are presented with six menu items which allow us

[1] `https://github.com/freechipsproject/rocket-chip`

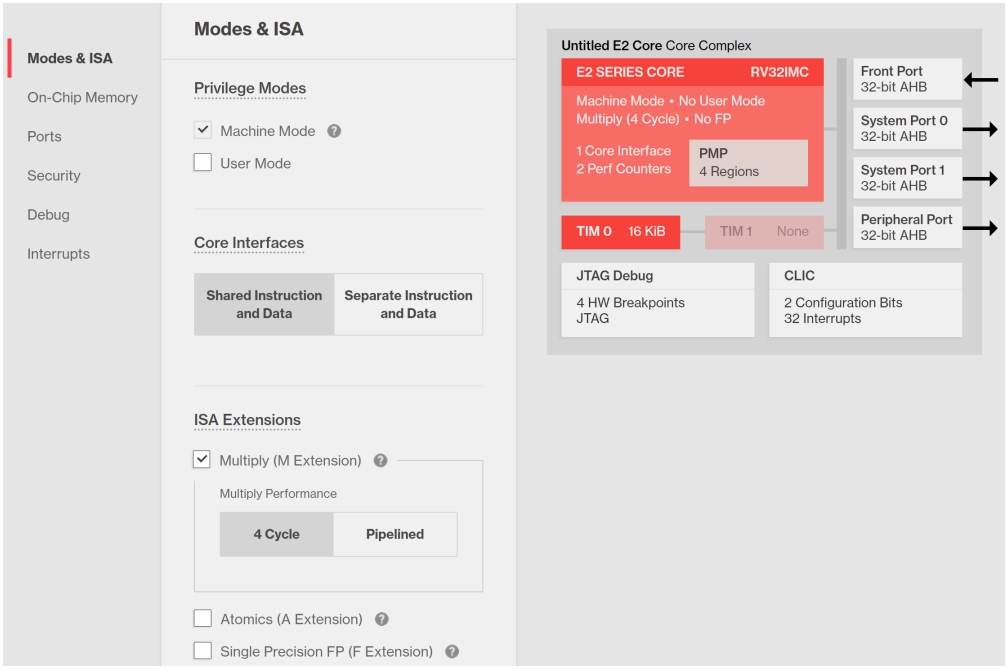


Figure 6.1: The SiFive Core Designer [67, Model: E20, connections added]

to for example define what I/O ports the design should have. Changes in the program display live in the interactive presentation

6.2 Standard Configuration

If one wants to dive further in customizing RISC-V designs, the standard configuration written in Chisel is the easiest way to modify the generated SoC. The configuration file which is located at *rocket-chip/src/main/scala/system/Configs.scala* provides the developer with configurations that already have been proven to work. To generate a standard single Rocket core the configuration

```
class DefaultConfig extends Config(new WithNBigCores(1) ++ new BaseConfig)
```

can be used. A quadcore processor is generated if the developer changes the 1 to a 4. To get a better understanding of the configuration process let us inspect the class WithNBigCores(n: Int).

```
class WithNBigCores(n: Int) extends Config((site, here, up) => {
    case RocketTilesKey => {
```

```
      val big = RocketTileParams(
        core = RocketCoreParams(mulDiv = Some(MulDivParams(
          mulUnroll = 8,
          mulEarlyOut = true,
          divEarlyOut = true))),
        dcache = Some(DCacheParams(
          rowBits = site(SystemBusKey).beatBits,
          nMSHRs = 0,
          blockBytes = site(CacheBlockBytes))),
        icache = Some(ICacheParams(
          rowBits = site(SystemBusKey).beatBits,
          blockBytes = site(CacheBlockBytes))))
    List.tabulate(n)(i => big.copy(hartId = i))
    }
})
```

In the piece of code above a standard configuration is described. We define a freely named variable big, which resembles one tile in our SoC design. Following we define, that the Rocket core should have a multiply and divide module (mulDiv). We continue with the size of the data cache (dcache) and instruction cache (icache). Additionally, we define that the data cache should be blocking (nMSHRs = 0). The last line in the example code duplicates the previously defined tile n times and assigns each new tile an ascending hardware thread number (hartId). The Rocket core above will be an all-purpose core which can, for example, execute Linux. In comparison, we can also configure a Rocket core for the use in a microcontroller.

```
class DefaultSmallConfig extends Config(
    new WithNSmallCores(1) ++
    new BaseConfig
)

class WithNSmallCores(n: Int) extends Config((site, here, up) => {
    case RocketTilesKey => {
    val small = RocketTileParams(
        core = RocketCoreParams(useVM = false, fpu = None),
        btb = None,
        dcache = Some(DCacheParams(
            rowBits = site(SystemBusKey).beatBits,
            nSets = 64,
            nWays = 1,
```

```
            nTLBEntries = 4,
            nMSHRs = 0,
            blockBytes = site(CacheBlockBytes))),
        icache = Some(ICacheParams(
            rowBits = site(SystemBusKey).beatBits,
            nSets = 64,
            nWays = 1,
            nTLBEntries = 4,
            blockBytes = site(CacheBlockBytes))))
    List.tabulate(n)(i => small.copy(hartId = i))
    }
})
```

Again we define a tile but this time it should be smaller. To achieve this, we disable the creation of the FPU and set the parameter useVM to false. The name useVM might be a bit misleading here. It does not reference the hypervisor capability of the Rocket core but controls the creation of an MMU. Thus this processor will not have the capability of using a virtual address space. As in our previous design, we now define the sizes of the data and instruction cache, but override the standard value *nWays*=4 with 1 to get a 1-way associated cache, which results in slimmer data interconnects and thus a simpler CPU. Again, as the last step, the design is duplicated and the according hartId is assigned.

6.3 Rocket Custom Coprocessor (RoCC)

If the above methods do not fulfill the requirements of the use case, more custom accelerators can be added to the design. A comparably easy way to do this is to use the Rocket Custom Coprocessor (RoCC) interface. The RoCC interface defines how the accelerator is connected and how it can be called. In figure 6.2 a simplified Rocket core with a connected RoCC accelerator can be seen. Although the accelerator in the figure is only connected to the L1 data cache, a connection to the L2 cache or another component block is possible. Again as in the previous section, everything is designed and configured with Chisel.

6.4 Standard extension

If additional changes to the execution have to be made, we need to go deeper into the code of Rocket Chip. All changes that need to be made to the existing code should be

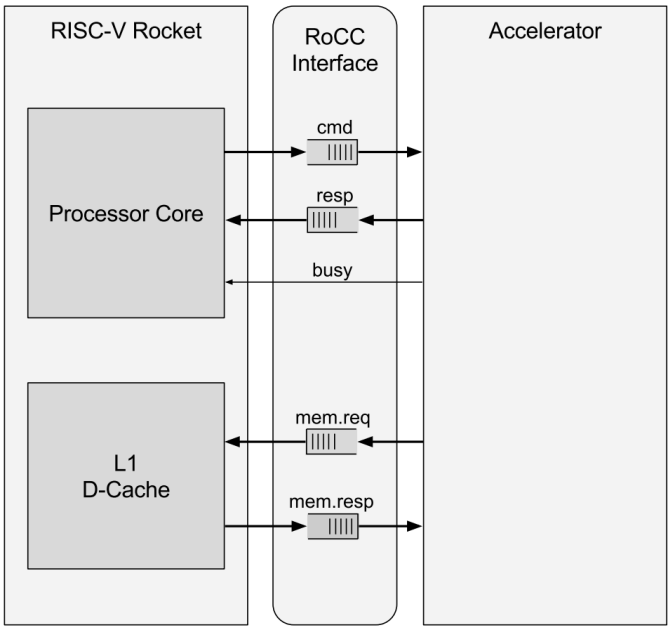


Figure 6.2: Simplified view on a processor with RoCC accelerator [37, p. 6]

made optional. If we, for example, need additional registers for our task, we would need to place checks for our extension in the code. If the extension is deactivated, the generated design does not contain any hints to the existence of the extension.

6.5 Non-standard extensions

If a change to the code cannot be made optional, the last resort is to generate a non-standard extension patch for the Rocket Chip project. The design of the generated chip is defined in the code after all, and as the code is licensed freely, everthing can be done with it. The downside to this approach is, that future version and other standards of the ISA might no longer be compatible to the non-standard changes.

7 Concept

As we have seen in the previous chapters 5 and 6 the Rocket Chip SoC generator is build to be extensible and adaptable. This modularity allows for new concepts when it comes to implementing Checkpoint/Restore mechanisms. The following sections will present one way of implementing a Checkpoint/Restore in a customizable architecture like RISC-V.

7.1 Virtual sliced Checkpoint/Restore

A big problem of previous implementations of Checkpoint/Restore mechanism is often the tracking of checkpointed pages (or any other size one chooses to checkpoint) and the link to the working copy that should be used for write operations which occur after the checkpoint was put in place. The operations take computing time and memory.

7.1.1 Extending the page table entries and the memory management unit

We want to start by adding two Boolean entries to the already existing page table entries (PTEs).

- Checkpoint active (CA)
- Checkpoint written (CW)

When the CA flag is active write operations for this memory page will be redirected to the working copy of the page. If the CW flag is not set a copy operation will be performed before the write operation to copy the complete data set to the location of the working copy. A read operation will only be redirected when the CW flag is set to true. This system does work for a single checkpoint but also works for multiple checkpoints, as the redirected PTE on its own contains the two flags which again can trigger a redirect to a new PTE. A check should be included to avoid endless loops. If the prefix overflows from 1...1 to the original base prefix 0...0 an error should be raised.

7.1.2 Addressing the working copy

To make addressing a working copy easier and less complicated we use the large virtual address space of a 64-bit processor and split it up into n by power of two pieces of the same size. The value of n defines the maximum count of possible checkpoints (including the current working copy) the system can hold. Thanks to the splitting of the virtual address space we now can address working copies with a simple prefix. The system will at startup only be presented with the first block with the prefix 0...0 and is unable to directly write to other blocks. To illustrate the process better, let us assume we pick $n = 2$. A system will, in the normal operation, work on the virtual memory part with the prefix 0. Let us assume further that our system sets the checkpointing flag for the virtual page 00...4242 ("..." short for 0s) and wants to execute a write operation on this page. As the CA flag on the PTE for 00...4242 is set, but the CW flag is not, the MMU will trigger a trap, called Copy-on-Write (CoW) interruption. This trap will cause the content of page 00...4242 will be copied to the page 10...4242. After this operation, every write access to the PTE 00...4242 will be redirected to the PTE 10...4242 as long as the CA flag is set in PTE. This system can easily be applied to $n = 2^i, i \in \mathbb{N}_{\neq 0}$ as the redirects will recursively lead to the current working copy PTE.

7.1.3 Restore process

To restore the previously checkpointed state of the pages, the system should block the access to the address ranges. This lock will prevent the possibility of corrupt data during the restore process. A full halt of the program execution might not be needed. A waitForCheckpoint trap, which is thrown when a certain memory page is accessed, should be sufficient. This trap then waits for the CA flag to be false. To increase the performance, it might be useful that the trap can also signal a request for prioritization to the restore process.

7.1.4 Deletion process

A deletion process can be easily implemented, if the processor supports locking PTEs (RISC-V T extension, 5.3.13). Then the original PTE can be locked, while the data of the working copy PTE is copied back. If the CPU does not provide the support for locking a page, we need to implement a similar lock like in the previous subsection.

7.1.5 Expected performance and limitations

Virtual address space

The first obvious limitation is the splitting of the addressable virtual memory space. If or if not this impacts the execution of a system depends on the work task and the available virtual memory space the processor does provide. 32-bit implementations might often be ruled out, as splitting the already limited space of 32-bit into smaller pieces limits the applications the processor can be used for. As for 64-bit applications, this problem is far less serious, as long as the number of needed checkpointing slots is limited. The RISC-V 64-bit implementation defines up to 48 bits of addressable memory space [74, p. 63], which translates to $2^{48} = 281,474,976,710,656$. For a system with a predefined value of $n = 4$ still $2^{48}/4 = 2^{46} = 70,368,744,177,664$ addresses will be available and four system states can be maintained. This equals roughly 70TB of addressable memory. If this limit is reached, one could reduce the number of available checkpoints or switch to a 128-bit implementation of a processor (like the RISC-V 128-bit base) which provides huge addressable memory spaces.

Additional memory usage

As we add two boolean values to the PTEs, we decrease the number of elements the translation lookaside buffer (TLB) of the MMU can hold. Even if the PTE is only 64 bit in size, we increase the size to only 66, a $66/44 - 1 = 3,125\%$ of additional memory. A bigger problem could be the trashing of the TLB. When checkpoints are created future read and write operations need to find the new working copy that should be used recursively. This generates the problem that for a single memory page n PTEs are necessary to get to the actual working copy. To negate this problem a processor might increase the size of the TLB, which can increase the performance of the system not only in checkpointing use cases but also in the normal operation.

Expected performance

The performance of PTE checkpointed memory page operations solely relies on the implementation of the MMU. If the MMU can check the checkpointed recursive PTE paths fast enough, there will be only an insignificant loss of performance due to the increased size of the PTEs.

8 Implementation

8.1 FPGAs

This thesis started with the fpga-zynq repository [28] which provides a version locked edition of the rocket-chip repository, a Linux front-end which enables us to communicate with the RISC-V core on the FPGA and rocket-chip build configurations for the interconnect between the front-end and the RISC-V processor. The project is deprecated, and thus the functionality might be limited.

8.1.1 Xilinx Zybo Zynq-7000

The first attempt was to put the rocket chip Verilog code on the FPGA of a Xilinx Zybo Zynq-7000 with the Xilinx Vivado software. The Xilinx Zybo board combines an FPGA with 28,000 logic cells with an ARM dual-core [84]. The rocket-chip fpga-zynq configuration from the project defines the following code:

```
...
class ZynqSmallConfig extends Config(
    new WithZynqAdapter ++
    new DefaultSmallConfig
)
...
```

The configuration creates a small rocket-core with an attached ZynqAdapter, which uses the Xilinx AXI4 protocol to communicate with the ARM cores.

After compiling the rocket-chip Verilog code with the Xilinx Vivado suite to a binary, the software reported that the design would not fit on the Zybo board. The number of logic cells of the board was not high enough, although the repository states compatibility with the Zybo board in the repository description.

8.1.2 Xilinx Zedboard

The Zedboard by Xilinx was the next larger board which was available at the chair of operating systems at the Technical University Munich. It houses 85,000 logic cells [14]

and could be programmed with the rocket chip design. After the compilation of the design, one needs to prepare an SD card, which contains all files the Zedboard needs to boot. Normally all of these files can be obtained by executing the download scripts in the fpga-zynq repository but as the project is deprecated some files are missing or are no longer valid. The Linux kernel and the U-boot (the boot manager for the project) had to be manually compiled as they were no longer available. Also, the integrated RISC-V compiler in the Xilinx suite is deprecated and needs to be replaced with *riscv64-unknown-elf-gcc* from the compiled version of the risc-toolchain [21]. In general, it was not easy to follow the tutorials as many things were deprecated, steps were missing, files were no longer available, and links lead to 404 pages. After everything was put in place on the SD card, the Zedboard would not boot past the U-boot bootloader. Even combined efforts with other colleges from the chair and multiple RAM load configurations could not get the boot process past the u-boot command line. In the issues tab of the fpga-zynq repository multiple users report that they have difficulties to get the project running [1, 2, 3]. The user zxhero speculates that the problem might be the communication between the rocket-chip and the debug module [3]. It is certainly possible to bring a Rocket core on the Zedboard, but this requires greater efforts, which are out of the scope of this thesis. The connection problems might also be connected to the issue "AXI4 MEM port: wrong write address timing" in the rocket-chip repository [4].

8.1.3 Xilinx Arty A7: Artix-7

After some research, the next best option to get a running rocket-chip core on an FPGA was the Xilinx Arty A7: Artix-7. The board features only an FPGA which eliminates the previously problematic AXI4 ZynqAdapter between the FPGA and the ARM processor and replaces it with a simple JTAG connector. It is used by SiFive to prototype their first RISC-V microcontroller, the HiFive1. SiFive created the Freedom platform on GitHub with configuration files and additional extensions to make the setup of the board easier [19]. Although the setup of the Freedom platform was easier and much more up-to-date than the fpga-zynq project, some parts were still missing. An alternative complete guide [5] on how to set up the Freedom E300 platform from SiFive on the Xilinx Arty A7 board was created, to help future projects. All problems that occurred during the setup and how to mitigate them are listed in the guide. After the setup, the board booted the rocket core without problems, and the integrated programs can be run. However,

[1] `https://github.com/ucb-bar/fpga-zynq/issues/92`
[2] `https://github.com/ucb-bar/fpga-zynq/issues/97`
[3] `https://github.com/ucb-bar/fpga-zynq/issues/101`
[4] `https://github.com/freechipsproject/rocket-chip/issues/1778`
[5] `https://github.com/Mixermachine/setup-freedom-e300-on-arty-A7`

upon further inspection of the rocket core, it came apparent, that the version for the Arty board will not be sufficient for this Bachelor thesis. One can spot this in the Chisel configuration:

```
class DefaultFreedomEConfig extends Config (
    ...
    new WithJtagDTM ++
    new TinyConfig
)
```

The configuration defines the core to be "tiny", which sets the "useVM" boolean to false and thus disables the creation of the MMU, which is essential for the Checkpoint/Restore approach described in chapter 7. As the E300 platform is not usable for this thesis and larger FPGAs are very costly (the larger SiFive U500 platform use the Xilinx Virtex-7 FPGA VC707, which costs about 3,500 dollar [19], [85]), further steps are done in the simulator.

8.2 Rocket-chip

The rocket-chip repository on GitHub is often denoted as "cutting-edge" and currently has some build issues [6, 7]. The release feature of GitHub is also not used by the rocket-chip repository, so finding a stable version might not always be easy. In the following sections the referenced rocket-chip version of the freedom repository *b21c7879b3ea22f69cb8457109561f37c225f8ea* is used, as it seems to be stable. To clone the repository we need to execute the following code:

```
# Checkout the repository and the connected submodule
git clone https://github.com/sifive/freedom.git
cd freedom
git submodule update --init --recursive

# Build the included version of the RISC-V GNU Compiler Toolchain
cd rocket-chip
mkdir toolchain
cd toolchain
export RISCV=\$(pwd)
cd ../riscv-tools/
./build.sh
```

[6] https://github.com/freechipsproject/rocket-chip/issues/1709
[7] https://github.com/freechipsproject/rocket-chip/issues/1644

```
cd ..

# To run Verilater simulation tests we need to build Verilator and the tests
cd emulator
# if we need waveform output add the prefix 'debug' to make
make
```

The bash commands above will download the source code, compile the compiler toolchain and the test suite with the Verilog simulator Verilator. Verilator compiles Verilog code to C or C++ code [78]. After we compiled the test suite, we will find *.riscv* files that contain the tests.

8.2.1 Finding the MMU

As the approach of this thesis on Checkpoint/Restore heavily relies on the memory management unit it is critical to find the code for the MMU. The rocket-chip code does not contain any class that is labeled "MMU" and does contain in general nearly no comments, which made it hard to find references to the actual position of the MMU. After consulting the mailing list, first a hint to the Translation Lookaside Buffer (TLB) Scala file was given by Bruce Holt [8]. A second hint was given by Abraham Gonzalez and Farzad Farshchi [9]. They recommended to

- read the documentation at [10, 11].
- build the Verilog and open it in a schematic viewer
- run 'risc-v tests' and look into the waveform
- read section 4.3 and 4.4 of the privileged spec

and mentioned, that the MMU in the rocket-chip project is called page table walker (PTW). A quick search in the project revealed the file *rocket-chip/src/main/scala/rocket/PTW.scala* to be the object of desire. It contains classes like *PTWReq (PTW request), PTWResp (PTW response), TLBPTWIO (TLB PTW communication)* and one particularly interesting class *PTE (page table entry)*

[8] https://groups.google.com/a/groups.riscv.org/forum/?utm_medium=email&utm_source=footer#!msg/isa-dev/vf33dd6ulh0/mu3AaPTvDQAJ
[9] https://groups.google.com/forum/?utm_medium=email&utm_source=footer#!msg/riscv-boom/jZ1gC4Qs35Q/8HltWll_CgAJ
[10] https://github.com/cnrv/rocket-chip-read
[11] https://www.lowrisc.org/docs/tagged-memory-v0.1/rocket-core/

```
class PTE(implicit p: Parameters) extends CoreBundle()(p) {
    val ppn = UInt(width = 54)
    val reserved_for_software = Bits(width = 2)
    val d = Bool()
    val a = Bool()
    val g = Bool()
    val u = Bool()
    val x = Bool()
    val w = Bool()
    val r = Bool()
    val v = Bool()

    def table(dummy: Int = 0) = v && !r && !w && !x
    def leaf(dummy: Int = 0) = v && (r || (x && !w)) && a
    def ur(dummy: Int = 0) = sr() && u
    def uw(dummy: Int = 0) = sw() && u
    def ux(dummy: Int = 0) = sx() && u
    def sr(dummy: Int = 0) = leaf() && r
    def sw(dummy: Int = 0) = leaf() && w && d
    def sx(dummy: Int = 0) = leaf() && x
}
```

The class PTE defines one page table entry, but no comment is given in the definition. In general, comments in the project are quite sparse, the 385 lines *PTW.scala* file contains only five comments. After some research, the variables could be resolved with the help of the RISC-V Privileged ISA Specification [74, p. 59-61]. "Virtual page" is abbreviated by vp.

- d: Dirty, vp has been written
- a: Accessed, vp has been read, written or fetched
- g: Global, vp exists in all address spaces
- u: User, vp is accessible to the user mode
- x: Executable, vp is executable
- w: Writable, vp is writeable
- r: Readable, vp is readable

- v: Valid, vp is valid
- table: vp is pointer to next level of page table.
- leaf: vp is valid, at least fetched and can be read or executed
- ur: vp is sr and readable by user
- uw: vp is sw and writable by user
- ux: vp is sx and executable by user
- sr: vp is leaf and readable
- sw: vp is leaf, writable and dirty
- sx: vp is leaf and executable

To help future developers, a fork and a new branch of the project has been created on GitHub which adds comments to the definitions of the *PTW.scala* file [12]. The patches will be packed into a patch and submitted to the original project in a pull request. Now that we know where the PTW is defined we need to find out how it works. A first hint on how the PTW is connected with the TLB can be seen in figure 8.1 from the lowRISC project. The figure shows the L1 instruction cache of a Rocket core. On the right side of the schematic, we can see that a TLB element is built directly into the Cache module, most certainly to speed up the lookup of frequently used virtual pages. This TLB can trigger a refill which is fulfilled by the PTW. For the further setup of the system a question regarding the setup of the TLB and PTW on the mailing list was asked [13]. Andrew Waterman replied to the inquiry and clarified the setup: The Rocket core is connected over the L1 instruction cache to the TLB that resolves often requested PTEs. If the TLB does not have a specific entry, it forwards the request to the PTW. The Rocket core also has a direct connection to the PTW, but this is only used to get the page-table base register. This register is responsible for providing each process with a separated logic address space [76, "Page Tables" -> "The PTBR"]. The L1 data cache has a similar TLB element. The PTW has a small cache for nonleaf nodes (pointer to other page tables) and a larger L2 TLB for PTEs.

[12] `https://github.com/Mixermachine/rocket-chip/blob/documentation/commentsPTW/src/main/scala/rocket/PTW.scala`

[13] `https://groups.google.com/a/groups.riscv.org/forum/?utm_medium=email&utm_source=footer#!msg/hw-dev/_RqboAwU8pQ/T8rvsE6eGAAJ`

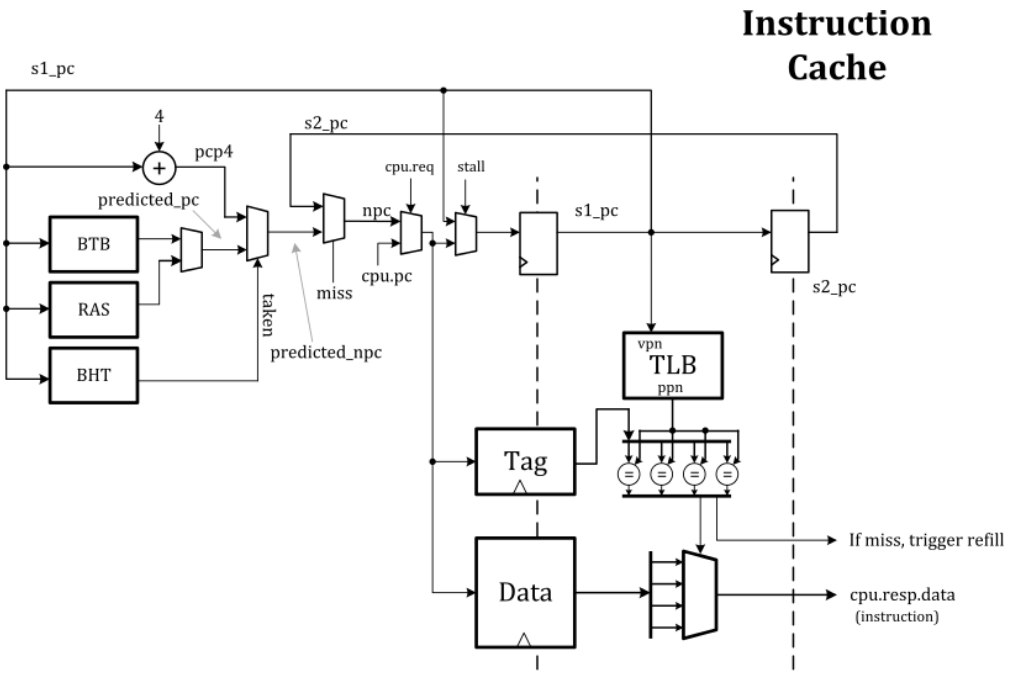


Figure 8.1: Schematic of the instruction cache of a Rocket core [44]

8.2.2 Thoughts on the implementation of the concept

The RISC-V setup of cascaded TLBs certainly provides faster address resolution but makes the implementation of the Checkpoint/Restore concept from chapter 7 harder.

L1 cache

The complete Checkpoint/Restore algorithm implementation could happen directly in the L1 caches, as this is the first location a checkpointed PTE would hit a possible resolution. This approach is beneficial in regards to the execution performance, as every processor would be connected to its own Checkpoint/Restore element. The downside is that this approach would enlarge the die size of the Cache memory. L1 cache is extraordinarily expensive to build, as it needs to run at high clock speeds.

PTW

A more cost-effective approach would be to implement the mechanism mainly in the PTW. A checkpointed PTE then would first need to be processed in the PTW, which then needs to either synchronize or invalidate all existing copies in the TLBs. In regard

to the TLBs there are multiple possible solutions. The TLBs in the standard form do not know how to handle checkpointed PTEs.

- TLBs do not store checkpointed PTEs
- TLBs get a changed PTE with different physical address
- TLBs contain a minimal Checkpoint/Restore logic

The first solution is the cheapest one but could hurt the performance of the system, as all checkpointed PTEs can no longer be cached in the L1 cache and every operation needs to be performed by the PTW.

In the second solution the PTW invalidates all previous copies of PTEs in the L1 Caches and then replaces them with manipulated PTEs. This PTEs would still have the virtual address out of the first virtual memory block (prefix 0) but the physical address points to the new working copy.

The last solution would need a change in the TLBs. A minimal Checkpoint/Restore logic could be implemented, which only handles the CA, CW bits in the PTEs. The rest of the logic for creating and managing checkpoints can be outsourced to the PTW. A new *PTWResp* format might be necessary as the TLBs need to receive multiple PTEs for one *PTWReq* in order to process a checkpoint correctly.

Due to time constraints, this thesis can not go further in the implementation of the concept.

8.2.3 riscv-tests

For future efforts in the direction of implementing the Checkpoint/Restore concept of this thesis or in general to understand the logic of the processor better the riscv-tests repository is beneficial, as the simulation can output waveform files which contain all wire states of the CPU run. The riscv-tests system [27] is designed to be as modular as possible. The included test virtual machine (TVM) hides the differences between the implementations by restricting the operation of the test. The TVM can define:

- The set of registers and instructions that can be used.
- Which portions of memory can be accessed.
- The way the test program starts and ends execution.
- The way that test data is input.
- The way that test results are output.

[27]
Currently, the following TVMs are already implemented. All of them feature only one hardware thread:

- rv32ui: RV32 user-level, integer only
- rv32si: RV32 supervisor-level, integer only
- rv64ui: RV64 user-level, integer only
- rv64uf: RV64 user-level, integer and floating-point
- rv64uv: RV64 user-level, integer, floating-point, and vector
- rv64si: RV64 supervisor-level, integer only
- rv64sv: RV64 supervisor-level, integer and vector

[27]
An additional layer of testing flexibility is added with the target environment configuration:

- p: virtual memory is disabled, only core 0 boots up
- pm: virtual memory is disabled, all cores boot up
- pt: virtual memory is disabled, timer interrupt fires every 100 cycles
- v: virtual memory is enabled

[27]
Testing programs are written in an assembly language, which includes the RISC-V ISA and additional commands for the test environment like *RVTEST_CODE_BEGIN, RVTEST_PASS, RVTEST_CODE_END, and RVTEST_FAIL*. A nice example is the following assembly code take from [27].

```
#include "riscv_test.h"

RVTEST_RV64U # Define TVM used by the program.

# Test code region.
RVTEST_CODE_BEGIN # Start of test code.
       lw x2, testdata
       addi x2, 1 # Should be 42 into \$2.
```

```
       sw x2, result # Store result into memory overwriting 1s.
       li x3, 42 # Desired result.
       bne x2, x3, fail # Fail out if doesn't match.
       RVTEST_PASS # Signal success.
fail:
       RVTEST_FAIL
RVTEST_CODE_END # End of test code.

# Input data section.
# This section is optional, and this data is NOT saved in the output.
.data
       .align 3
testdata:
       .dword 41

# Output data section.
RVTEST_DATA_BEGIN # Start of test output data region.
       .align 3
result:
       .dword -1
RVTEST_DATA_END # End of test output data region.
```

At the start of the program, the *riscv_test.h* header file needs to be included, which defines the macros that are used by the TVM. After the include, we define the TVM which should be used for our test. In the test run, we load some test data (32-bit unsigned integer 41) and add 1 to it. The reasonable answer for this calculation is of course 42, which is checked in the test by comparing the two registers x2 and x3. If this check does for some reason fail, we jump to RVTEST_FAIL which triggers the failure of the test. This test can be used to get to know the PTW better, as the command *sw* stores the register value in the memory. Maybe a reduced version that only executes:

```
#include "riscv_test.h"

RVTEST_RV64U

RVTEST_CODE_BEGIN
    lw x2, testdata
    sw x2, result
    RVTEST_PASS
RVTEST_CODE_END
```

```
testdata:
    .dword 42
```

might be a good place to start.

9 Evaluation

In the course of this bachelor thesis, we explored how the RISC-V ISA is set up and how the implementation of the University of Berkeley rocket-chip can be extensively configured to fit many use-cases. The start of the bachelor thesis was mainly dominated by the setup of the Xilinx boards the included Xilinx Vivado development suite. Sadly the fpga-zynq repository is no longer up to date, and thus the attempt to bring a working Rocket-core on an FPGA-ARM board did not succeed. Even after combined efforts with colleagues from the chair of operating systems could not get the Linux image to boot on the Xilinx boards. The experience with the Xilinx Arty board in combination with the up-to-date Freedom repository from SiFive was better but not still not perfect. A guide has been written, which should make the setup of the E300 platform on the Xilinx Arty board easier. After the setup up of the board, it was discovered, that it does contain a large enough FPGA to support a Rocket Chip MMU. Nonetheless, the gained knowledge of working with the Freedom platform can still be used for later attempts of bringing the larger SiFive U500 platform on the new Xilinx VC707 FPGA board at the chair. The included quad-core in the U500 platform contains an MMU and is capable of running Linux. For further research, the included emulator was used. The following phase in section 8.2 was dominated by the studies of the Rocket Chip project and the structure of the Rocket core. As the concept of the hardware construction language Chisel is new and the Rocket Chip project contains only a small number of comments the initial barrier to understanding the structure was high. With the help of the people from the RISC-V and BOOM mailing list, initial questions could be resolved, and the system was easier to understand. After further analysis of the Rocket Core, one found the caching structure to be more complicated than previously anticipated but well thought out. The scope of the implementation will need to include some change to the translation lookaside buffers, located in the L1 caches of the Rocket tiles. The concept in chapter 7 could not be implemented, as time was restricted.

For future approaches, it might be beneficial to start first with a virtual emulator instead of a physical FPGA attempt, to speed up the initial discovery process of the Rocket-Chip generator.

10 Limitations

10.1 Xilinx boards and Vivado

In the course of this thesis many problems were encountered. The first problem that has been met was the deprecation of the fpga-zynq repository, which made the attempt to bring a working RISC-V core on the Xilinx boards Zybo Zynq-7000 and Zedboard fail. The compatibility of the Xilinx Vivado development suite sometimes seems to be unclear. The SiFive Freedom repository, for example, recommends the version 2017.1 or newer for the E300 platform as "old versions are known to fail". The recommended version for the U500 platform, on the other hand, is 2016.04 or older as "newer versions are known to fail" [1].

Also, in the course of this Bachelor thesis, a patch update from 2018.2 to 2018.2.2 provoked a failure in the compilation process. Reverting to the previous version fixed the issue.

10.2 Rocket-chip repository

As we have seen in the chapter 8 the Rocket-chip project file *PTW.scala* did not contain many lines of comments. After some further inspection of the Rocket-chip repository and the use of the Scalastyle inspection [2] in IntelliJ it is apparent, that the project contains a meager number of comments. This makes it hard for beginners to understand the logic of the project. One developer in the mailing list recommended to execute the virtual tests and analyze the resulting waveform file as the good way to understand the logic [3].

[1] https://github.com/sifive/freedom

[2] http://www.scalastyle.org/

[3] `https://groups.google.com/forum/?utm_medium=email&utm_source=footer#!msg/riscv-boom/jZ1gC4Qs35Q/8HltWll_CgAJ`

10.3 Concept

The presented concept mainly contains two limitations. First, there is the reduction of addressable virtual memory, as the proposed method splits the virtual memory space into n equally sized parts. This can limit larger applications, especially if many checkpoint slots are necessary. The second limitation is the performance penalty for recursively searching the current working copy of a page table entry, as every new incremental checkpoint adds a new page table entry. The last limitation is the higher memory requirement for the TLB, which first needs to hold two more bits per PTE and secondly has to store n PTEs in the worst case to access one working copy for one memory page.

11 Future work

The future work for this thesis will include the implementation of the Checkpoint/Restore concept which was presented earlier. As the RISC-V architecture is steadily growing, this might be a worthwhile task for future processors and applications. In subsection 8.2.2 we already discovered, that the memory system of a Rocket core is more complex as previously anticipated. We have found two ways to implement the concept of which the first one was expensive and the second one contained a performance penalty. For the future work, the author recommends the second option, as firstly the performance penalty only hits when a checkpoint is created, and secondly, the performance should only be mildly impaired as only small sets of data need to be renewed, and the resolution of page tables is, in general, a very accelerated task.

12 Conclusion

Looking back at the beginning of this thesis, the journey through Xilinx ARM-FPGA boards, binary files, the RISC-V ISA and the many RISC-V implementations was very interesting. The Rocket-chip was complicated at first glance, but presents a unique and modern way of solving hardware design problems. Chisel and its base language Scala allow the use of reusable, highly configurable classes which later turn into synthesizable Verilog.

The original goal of finding and implementing a real-time capable Checkpoint/Restore mechanism was not fully reached, as the setup of the Xilinx boards at the start of the thesis took time. However, this thesis found a new approach to Checkpoint/Restore systems and presented it in chapter 7.

The LaTeX source code of this document can be obtained at [1].

[1] `https://github.com/Mixermachine/ba-thesis-risc-v-checkpoint-restore`

List of Figures

List of Tables

Bibliography

[1] A. Alexander. *Scala Cookbook: Recipes for Object-Oriented and Functional Programming*. 1st. O'Reilly Media, Inc., 2013. ISBN: 1449339611.

[2] *Amazon, Amazon EC2 F1-Instances*. https://aws.amazon.com/de/ec2/instance-types/f1/. Accessed: 2019-01-20.

[3] *Anandtech, Western Digital Reveals SweRV RISC-V Core, Cache Coherency over Ethernet Initiative*. https://www.anandtech.com/show/13678/western-digital-reveals-swerv-risc-v-core-and-omnixtend-coherency-tech. Accessed: 2019-02-07.

[4] *ARM, GPU Compute accelerated HEVC decoder on ARM® MaliTM-T600 GPUs*. https://www.arm.com/files/event/Mali_Partner_Track_4_GPU_Compute_accelerated_HEVC_decoder_on_ARM_Mali-T600_GPUs.pdf. Accessed: 2019-01-10.

[5] K. Asanović and D. A. Patterson. *Instruction Sets Should Be Free: The Case For RISC-V*. Tech. rep. UCB/EECS-2014-146. EECS Department, University of California, Berkeley, Aug. 2014.

[6] S. Bachmaier. "Optimization of a real-time capable Checkpoint/Restore mechanism for L4 Fiasco.OC/Genode by hardware-assisted memory tracing and copyin." MA thesis. TU Munich, 2019.

[7] *Berkeley University, Video: Instruction Sets Want To Be Free: A Case for RISC-V*. https://www.youtube.com/watch?v=mD-njD2QKN0. Accessed: 2019-01-18.

[8] U. Brinkschulte and T. Ungerer. *Mikrocontroller und Mikroprozessoren (eXamen.press) (German Edition)*. Springer, 2010. ISBN: 3540468013.

[9] *Carnegie Mellon University, John DeVale, Checkpoint-Recovery*. https://users.ece.cmu.edu/~koopman/des_s99/checkpoint/. Accessed: 2019-01-03.

[10] *Chisel Community Conference 2018, FireSim Intensive (Architecture Track)*. https://www.youtube.com/watch?v=S3OriQnJXYQ. Accessed: 2019-01-20.

[11] *Christopher Domas, GOD MODE UNLOCKED - Hardware Backdoors in x86 CPUs*. https://www.youtube.com/watch?time_continue=3058&v=_eSAF_qT_FY. Accessed: 2019-01-12.

[12] S. Davidson, S. Xie, C. Torng, K. Al-Hawai, A. Rovinski, T. Ajayi, L. Vega, C. Zhao, R. Zhao, S. Dai, A. Amarnath, B. Veluri, P. Gao, A. Rao, G. Liu, R. K. Gupta, Z. Zhang, R. Dreslinski, C. Batten, and M. B. Taylor. "The Celerity Open-Source 511-Core RISC-V Tiered Accelerator Fabric: Fast Architectures and Design Methodologies for Fast Chips." In: *IEEE Micro* 38.2 (Mar. 2018), pp. 30–41. ISSN: 0272-1732.

[13] *Debian, Official architectures - Debian-ports architectures*. `https://buildd.debian.org/stats/`. Accessed: 2019-01-22.

[14] *eLinux, Zedboard*. `https://elinux.org/Zedboard`. Accessed: 2019-02-02.

[15] *Fabrice Bellard, JSLinux*. `https://bellard.org/jslinux/`. Accessed: 2019-01-22.

[16] *Genode, How Genode came to RISC-V*. `https://genode.org/documentation/articles/riscv`. Accessed: 2019-01-28.

[17] *Genode, How to use Genode with different kernels*. `https://genode.org/documentation/platforms/index`. Accessed: 2019-01-28.

[18] *Github, Bartblaze, Disable Intel AMT repository*. `https://github.com/bartblaze/Disable-Intel-AMT/`. Accessed: 2018-12-16.

[19] *Github, Freedom*. `https://github.com/sifive/freedom`. Accessed: 2019-02-03.

[20] *Github, RISC-V B extension releases repository*. `https://github.com/cliffordwolf/xbitmanip/releases`. Accessed: 2019-01-17.

[21] *Github, RISC-V GNU Compiler Toolchain repository*. `https://github.com/riscv/riscv-gnu-toolchain`. Accessed: 2019-01-22.

[22] *Github, RISC-V instruction set simulator Whisper repository*. `https://github.com/westerndigitalcorporation/swerv-ISS`. Accessed: 2018-12-29.

[23] *Github, RISC-V LLVM repository*. `https://github.com/lowrisc/riscv-llvm`. Accessed: 2019-01-22.

[24] *Github, RISC-V QEMU repository*. `https://github.com/riscv/riscv-qemu`. Accessed: 2019-01-20.

[25] *Github, riscv-angel repository*. `https://github.com/riscv/riscv-angel`. Accessed: 2019-01-22.

[26] *Github, riscv-sodor repository*. `https://github.com/ucb-bar/riscv-sodor`. Accessed: 2019-01-23.

[27] *Github, riscv-tests*. `https://github.com/riscv/riscv-tests`. Accessed: 2019-02-04.

[28] *Github, Rocket Chip on Zynq FPGAs*. `https://github.com/ucb-bar/fpga-zynq`. Accessed: 2019-02-02.

[29] *Github, SweRV RISC-V CoreTM from Western Digital*. `https://github.com/westerndigitalcorporation/swerv_eh1`. Accessed: 2019-02-05.

[30] *Github, Western Digital's Open Source RISC-V SweRV Instruction Set Simulator*. `https://github.com/westerndigitalcorporation/swerv-ISS`. Accessed: 2019-02-05.

[31] *Github, Z-scale Microarchitectural Implementation of RV32 ISA repository*. `https://github.com/ucb-bar/zscale`. Accessed: 2019-01-26.

[32] *GNU GCC, Transactional Memory in GCC*. `https://gcc.gnu.org/wiki/TransactionalMemory`. Accessed: 2019-01-20.

[33] *Golem, RISC-V LLVM*. `https://www.golem.de/news/cpu-architektur-risc-v-portierung-in-den-linux-kernel-aufgenommen-1711-131182.html`. Accessed: 2019-01-22.

[34] C. Hewitt, P. Bishop, and R. Steiger. "A Universal Modular ACTOR Formalism for Artificial Intelligence." In: *Proceedings of the 3rd International Joint Conference on Artificial Intelligence*. IJCAI'73. Stanford, USA: Morgan Kaufmann Publishers Inc., 1973, pp. 235–245.

[35] "IEEE Standard for Floating-Point Arithmetic." In: *IEEE Std 754-2008* (Aug. 2008), pp. 1–70.

[36] L. Igor and G. R. a. "HARDWARE-ASSISTED APPLICATION CHECKPOINTING AND RESTORING." US2016357645. Dec. 2016.

[37] *James Martin, RISC-V, Rocket, and RoCC*. `https://inst.eecs.berkeley.edu/~cs250/sp17/disc/lab2-disc.pdf`. Accessed: 2019-01-30.

[38] D. Kanter. *RISC-V OFFERS SIMPLE, MODULAR ISA*. Tech. rep. Mar. 2016.

[39] S. Karandikar, H. Mao, D. Kim, D. Biancolin, A. Amid, D. Lee, N. Pemberton, E. Amaro, C. Schmidt, A. Chopra, Q. Huang, K. Kovacs, B. Nikolic, R. Katz, J. Bachrach, and K. Asanovic. "FireSim: FPGA-Accelerated Cycle-Exact Scale-Out System Simulation in the Public Cloud." In: *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*. June 2018, pp. 29–42. DOI: 10.1109/ISCA.2018.00014.

[40] P. Kocher, J. Horn, A. Fogh, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom. "Spectre Attacks: Exploiting Speculative Execution." In: *40th IEEE Symposium on Security and Privacy (S&P'19)*. 2019.

[41] *Linux Gizmos, SiFive unveils new RISC-V designs, including two Linux-ready models*. http://linuxgizmos.com/sifive-unveils-octa-core-risc-v-designs-including-two-linux-ready-models/. Accessed: 2019-02-05.

[42] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, A. Fogh, J. Horn, S. Mangard, P. Kocher, D. Genkin, Y. Yarom, and M. Hamburg. "Meltdown: Reading Kernel Memory from User Space." In: *27th USENIX Security Symposium (USENIX Security 18)*. 2018.

[43] *lowRISC, Compile and install RISC-V cross-compiler*. https://www.lowrisc.org/docs/untether-v0.2/riscv_compile/. Accessed: 2019-01-22.

[44] *lowRISC, Rocket core overview*. https://www.lowrisc.org/docs/tagged-memory-v0.1/rocket-core/. Accessed: 2019-02-06.

[45] *Microcontrollertips. Arm TrustZone explained*. https://www.microcontrollertips.com/embedded-security-brief-arm-trustzone-explained/. Accessed: 2018-12-13.

[46] *Opencelerity, Open-Source RISC-V Tiered Accelerator Fabric SoC*. http://opencelerity.org/. Accessed: 2019-01-25.

[47] *Petter Källström, FPGA cell example*. https://commons.wikimedia.org/wiki/File:FPGA_cell_example.png. Accessed: 2019-01-15.

[48] *Phoronix, RISC-V's Linux Kernel Support Is Getting Into Good Shape, Userspace Starting To Work*. https://www.phoronix.com/scan.php?page=news_item&px=Linux-4.19-RISC-V. Accessed: 2019-01-22.

[49] *Proxmox, Qemu/KVM Virtual Machines*. https://pve.proxmox.com/wiki/Qemu/KVM_Virtual_Machines. Accessed: 2019-01-20.

[50] T. Qin, S. J. Bleiker, S. Rana, F. Niklaus, and D. Pamunuwa. "Performance Analysis of Nanoelectromechanical Relay-Based Field-Programmable Gate Arrays." In: *IEEE Access* 6 (2018), pp. 15997–16009. ISSN: 2169-3536. DOI: 10.1109/ACCESS.2018.2816781.

[51] *RISC-V, Bit-tech Article: SiFive announces 64-bit 1.5GHz RISC-V HiFive Unleashed SBC*. https://riscv.org/2018/02/bit-tech-article-sifive-announces-64-bit-1-5ghz-risc-v-hifive-unleashed-sbc/. Accessed: 2019-02-05.

[52] *RISC-V, Chisel – Accelerating Hardware Design*. https://riscv.org/wp-content/uploads/2015/01/riscv-chisel-tutorial-bootcamp-jan2015.pdf. Accessed: 2019-01-23.

[53] *RISC-V, Frequently Asked Questions about RISC-V*. https://riscv.org/faq/. Accessed: 2019-01-18.

[54] *RISC-V, Member Directory Nvidia*. https://riscv.org/membership/1437/nvidia/. Accessed: 2018-12-29.

[55] *RISC-V, Member Directory Western Digital*. https://riscv.org/membership/1020/western-digital/. Accessed: 2018-12-29.

[56] *RISC-V, Members at a Glance*. https://riscv.org/members-at-a-glance/. Accessed: 2019-01-18.

[57] *RISC-V, News: SiliconANGLE Article: China, Blockchain And Chips: Western Digital Looks Beyond Storage*. https://riscv.org/2018/06/siliconangle-article-china-blockchain-and-chips-western-digital-looks-beyond-storage/. Accessed: 2018-12-29.

[58] *RISC-V, RISC-V "Rocket Chip" SoC Generator in Chisel*. https://riscv.org/wp-content/uploads/2015/01/riscv-rocket-chip-generator-workshop-jan2015.pdf. Accessed: 2019-01-25.

[59] *RISC-V, RISC-V Software Ecosystem Overview*. https://riscv.org/software-status/. Accessed: 2019-01-20.

[60] *RISC-V, RISC-V Compressed Extension Workshop June 2015*. https://riscv.org/wp-content/uploads/2015/06/riscv-compressed-workshop-june2015.pdf. Accessed: 2019-01-19.

[61] *RISC-V, State of the Union, Santa Clara Convention Center, Santa Clara, CA*. https://content.riscv.org/wp-content/uploads/2018/12/8.40-Asanovic-RISC-V-State-of-the-Union.pdf. Accessed: 2019-01-28.

[62] *RISC-V, Z-scale: Tiny 32-bit RISC-V Systems*. https://riscv.org/wp-content/uploads/2015/06/riscv-zscale-workshop-june2015.pdf. Accessed: 2019-01-25.

[63] T. Sewell, S. Winwood, P. Gammie, T. Murray, J. Andronick, and G. Klein. "seL4 Enforces Integrity." In: *Interactive Theorem Proving*. Ed. by M. van Eekelen, H. Geuvers, J. Schmaltz, and F. Wiedijk. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 325–340. ISBN: 978-3-642-22863-6.

[64] S. Sharma and M. Chawla. "A three phase optimization method for precopy based VM live migration." In: *SpringerPlus* 5.1 (July 2016), p. 1022. DOI: 10.1186/s40064-016-2642-2.

[65] A. Shribman and B. Hudzia. "Pre-Copy and Post-Copy VM Live Migration for Memory Intensive Applications." In: *Euro-Par 2012: Parallel Processing Workshops*. Ed. by I. Caragiannis, M. Alexander, R. M. Badia, M. Cannataro, A. Costan, M. Danelutto, F. Desprez, B. Krammer, J. Sahuquillo, S. L. Scott, and J. Weidendorfer. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 539–547. ISBN: 978-3-642-36949-0.

[66] *SiFive, RISC-V QEMU Part 1: Privileged ISA v1.10, HiFive1 and VirtIO*. `https://www.sifive.com/blog/risc-v-qemu-part-1-privileged-isa-hifive1-virtio`. Accessed: 2019-01-22.

[67] *SiFive, SiFive Core Designer*. `https://scs.sifive.com/core-designer/`. Accessed: 2019-01-30.

[68] *SiFive, TileLink: A free and open-source, highperformance scalable cache-coherent fabric designed for RISC-V*. `https://content.riscv.org/wp-content/uploads/2017/12/Wed-1154-TileLink-Wesley-Terpstra.pdf`. Accessed: 2019-01-25.

[69] *StackOverFlow, How to build a Zscale core? (RISC-V, rocket-chip)*. `https://stackoverflow.com/questions/38732041/how-to-build-a-zscale-core-risc-v-rocket-chip`. Accessed: 2019-01-26.

[70] M. E. Staknis. "Sheaved Memory: Architectural Support for State Saving and Restoration in Pages Systems." In: *SIGARCH Comput. Archit. News* 17.2 (Apr. 1989), pp. 96–102. ISSN: 0163-5964. DOI: 10.1145/68182.68191.

[71] A. S. Tanenbaum. *Modern Operating Systems*. 3rd. Upper Saddle River, NJ, USA: Prentice Hall Press, 2007. ISBN: 9780136006633.

[72] *The Register, Up in arms! Arm kills off its anti-RISC-V smear site after own staff revolt*. `https://www.theregister.co.uk/2018/07/10/arm_riscv_website/`. Accessed: 2019-02-05.

[73] *The RISC-V Instruction Set Manual, Volume I: User-Level ISA*. Version 2.2. CS Division, EECS Department, University of California, Berkeley. Mar. 2017.

[74] *The RISC-V Instruction Set Manual, Volume II: Privileged Architecture*. Version 1.10. CS Division, EECS Department, University of California, Berkeley. Mar. 2017.

[75] *Top500, MIPS Acquired by AI Startup Wave Computing*. `https://www.top500.org/news/mips-acquired-by-ai-startup-wave-computing/`. Accessed: 2019-02-05.

[76] *University of Minnesota Duluth, Virtual Memory Address Translation*. `https://www.d.umn.edu/~gshute/os/address-translation.xhtml`. Accessed: 2019-02-06.

[77] M. Vasavada, F. Mueller, and P. Hargrove. "Comparing different approaches for Incremental Checkpointing : The Showdown." In: 2011.

[78] *Verilator, Introduction to Verilator*. https://www.veripool.org/wiki/verilator. Accessed: 2019-02-04.

[79] D. Vogt, C. Giuffrida, H. Bos, and A. Tanenbaum. "Techniques for Efficient In-Memory Checkpointing." In: vol. 48. Nov. 2013. DOI: 10.1145/2524224.2524236.

[80] *W3techs. Usage of operating systems for websites*. https://w3techs.com/technologies/overview/operating_system/all. Accessed: 2018-12-09.

[81] *Wave Computing, MIPS Open™*. https://wavecomp.ai/mipsopen. Accessed: 2019-02-05.

[82] *Western Digital, RISC-V implementation SweRV Core*. https://www.westerndigital.com/company/innovations#risc-v. Accessed: 2018-12-29.

[83] *Xilinx, Amazon picks Xilinx UltraScale+ FPGAs to accelerate AWS*. https://forums.xilinx.com/t5/Xcell-Daily-Blog-Archived/Amazon-picks-Xilinx-UltraScale-FPGAs-to-accelerate-AWS-launches/ba-p/735873. Accessed: 2019-01-20.

[84] *Xilinx, Digilent Zybo Zynq-7000 ARM/FPGA SoC Trainer Board*. https://www.xilinx.com/products/boards-and-kits/1-4azfte.html. Accessed: 2019-02-02.

[85] *Xilinx, Xilinx Virtex-7 FPGA VC707 Evaluation Kit*. https://www.xilinx.com/products/boards-and-kits/ek-v7-vc707-g.html. Accessed: 2019-02-03.

[86] *Youtube, Linux Kernel on RISC-V: Where Do We Stand*. https://www.youtube.com/watch?v=6X6i0kcy3GA. Accessed: 2019-01-26.