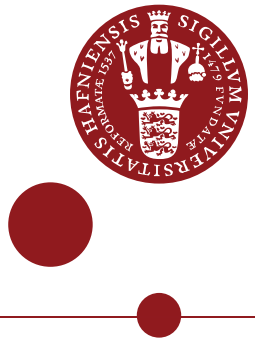
UNIVERSITY OF COPENHAGEN
DEPARTMENT OF COMPUTER SCIENCE



Bachelors Thesis

Simon Vinding Brodersen

RISC-V based computers in the data center

Advisor: Philippe Bonnet

May 25, 2024

Contents

1 Introduction **3**
- 1.1 Context . . . 3
- 1.2 Problem . . . 3
- 1.3 Approach . . . 4
- 1.4 Contribution . . . 4
- 1.5 Related work . . . 4

2 Background **4**
- 2.1 Accelerator-based Computer Architecture . . . 4
- 2.2 RISC-V . . . 5
- 2.3 Computational Storage . . . 5
- 2.4 Toolchain . . . 6

3 Design **7**
- 3.1 Single vs Multicore . . . 7
- 3.2 Context switching . . . 8
- 3.3 Memory . . . 8

4 Implementation **11**
- 4.1 Dependencies . . . 11
- 4.2 Creating a linker script . . . 12
- 4.3 Getting into the main function . . . 14
- 4.4 Printing with UART . . . 14
- 4.5 libucontext . . . 16

5 Evaluation **16**
- 5.1 Testing . . . 16

6 Conclusion **16**

1 Introduction

1.1 Context

Data centres are becoming increasingly essential in the IT sector. Whether it is Google's cloud platform, Microsoft Azure, or Amazon's web services, news about new data centres seems like a daily occurrence. With such scale comes an ever-growing need for custom solutions and cutting-edge technologies to both reduce power consumption and improve overall performance.

Historically, solid-state drives (SSDs) were a drop-in replacement for the magnetic disks of the past. They would implement a similar interface, allowing for seamless integration. But the use of SSDs came with multiple improvements over the magnetic disks of the past, which were hindered by said interface. As such, there was a rapid movement towards Open-channel SSDs that do not have a firmware Flash Translation Layer and instead leave the management of the physical SSD to the computer's operating system. This solves the issue mentioned previously but introduces further data transferring between the CPU and the SSD. However, in recent years, the discrepancy between a storage device's READ and WRITE operations and a CPU's ability to perform READ and WRITE memory operations has been ever increasing. If this trend continues, the CPU will soon become a bottleneck for performance in the data centers.

A solution to the problem would be to offload the CPU and provide computation at the SSD level. Such a solution has been described as a computational storage device (CSD). This would involve implementing the most commonly used data manipulations, such as indexing into an SSD or more complex manipulations like sorting. Within this thesis, the issue of implementing a high-performance sorting algorithm running on a stand-alone bare metal processor has been investigated.

1.2 Problem

For computational storage to be a viable solution for meeting the ever-growing demand for massive data computations, it is essential to investigate whether implementing a processor designed for such a purpose is feasible. Consequently, several open questions remain unanswered. (1) What type of computation should be performed by a storage device? (2) Is it possible to implement such computation on a bare-metal processor?

What computation should be handled by a storage device?

Although there are multiple cases of large data transfers between a CPU and an SSD, one of the more prominent is that of sorting a given array. Sorting plays an integral part in multiple programming scenarios. From being an integral part of many searching algorithms to its use in data science, fast sorting is a necessity for fast performance. With a running time of O(n log n), merge sort was the algorithm chosen for further investigation. Not only that, but parallel versions of the merge sort algorithm should be possible on bare-metal.

Is it feasible to implement such a computation on a bare metal RISC-V processor?

As the main goal is to offload the primary CPU, we must investigate whether it is at all possible to create a high-performance sorting algorithm without the need of an underlying operating system.

1.3 Approach

For this thesis, an experimental approach was taken. First, a feasible design developed for implementing on a bare metal processor is introduced. Secondly, an implementation of said designed is presented. Third, the viability and validity of the implementation is evaluated. Lastly, shortcomings and proposed further research are presented. These implementations will be carried out on a QEMU virtual machine where the code is loaded via a general loader.

1.4 Contribution

1.5 Related work

2 Background

2.1 Accelerator-based Computer Architecture

The notion of offloading has long been established in specialized teams, where each member focuses on their area of expertise. This concept seems inherently logical when discussing day-to-day work environments. Effective communication between entities, with an emphasis on performing tasks best suited to our skills, appears to be the foundation of efficient collaboration. Contrary, computer architecture relies heavily on the Central Processing Unit

(CPU) for executing various operations. An accelerator serves as a separate substructure designed with distinct objectives compared to the CPU itself. By offloading the CPU, accelerators can optimize performance and reduce energy consumption[8]. A prime example of an accelerator is the Graphics Processing Unit (GPU), a crucial component in contemporary computers. This thesis aims to explore the feasibility of adopting a similar design approach for creating computational storage devices.

2.2 RISC-V

Reduced Instruction Set Computing (RISC), particularly its fifth iteration, RISC-V, represents an Instruction Set Architecture (ISA) designed to simplify the development of custom processors for various applications. Unlike proprietary ISAs created by private companies, RISC-V offers a free and open-source solution that minimizes intellectual property concerns and reduces entry barriers, promoting innovation and affordability in processor development.[9].

RISC-V aims to provide a small core of instructions which compilers, assemblers, linkers, and operating systems can generally rely on, while still being extendable for more specialized accelerators. In RISC-V there are two primary base integer variants, RV32I and RV64I, which provide the 32-bit and 64-bit user-level address spaces respectively. However, RISC-V is already in the works with a RV128I variant which would provide the foundation needed for a 128-bit user address space in the future. In general, RISC-V provides standard and non-standard extensions, where standard extensions should not conflict with other standard extensions, and the non-standard extensions are highly specialized.

With the rise of ARM[1] based machines with comparble and in some cases better performance than that of a Complex Instruction Set Computing(CISC) alternative.[5] RISC-V aims to provide the same benefits in an open sourced environment. With this RISC-V, more specifically the 32-bit version, was chosen as the ISA for developement in this thesis.

2.3 Computational Storage

Computational storage can be seen as a subsection of Accelerator-based Computer architecture. Firstly, it aims to offload the host processor as described in Section 2.1 by providing a secondary processors optimized for specific computational tasks. Secondly, it aims to reduce data movement between the

[1]short for Advanced RISC Machine

storage device and the host processor. This would allow the read and writes to be distributed among multiple RAM sections rather than a single processor. This could be an integral part of the issues presented in Section 1.1, as a computational storage device would be scalable with the ever-growing need for large volumes of data processing.

2.4 Toolchain

2.4.1 QEMU

QEMU is a system emulator, which has the capabilities of emulating both a 32-bit and 64-bit RISC-V processor [3]. With QEMU I am able to create code intented for a processor running the RISC-V instruction set even if my development environment is running a different ISA. For the puposes of this thesis it is the RISC-V 32-bit version of the qemu virtual machine that will be used.

2.4.2 LLVM and RISC-V GNU Toolchain

The LLVM project is a collection of reusable compiler and toolchain technologies. Most noteably for the context of this thesis clang. Clang is a gcc compatibile frontend compiler, which aims to provide fast compile times and low memory use. In tandem with the LLVM compiler back end, clang provides a library-based architecture such that the compiler can work together with other tool. This allows for the use of more sophisticated development environments such as an Language Server Protocol(LSP). Generally clang also provides more sophisticated error reports making the overall debugging easier. Moreover, clang provides a crosscompiler capable of targeting the RISC-V 32-bit architecture.

At the time of writing, the lldb debugger connection to the RISC-V QEMU machine was inadequete for the needs of this thesis. As such the GNU gdb debugger for the RISC-V target was compiled for use as a debugger for the implementation section. Furthermore, the RISC-V GNU toolchain provides necessary header files for the stdlib, which allows for some rudimentary implementations of algorithms for the compiler to use, one such instance is that of memcpy.

3 Design

3.1 Single vs Multicore

When designing for bare metal implementation, implementing context switching on a single core would result in slower running algorithms than a standard implementation of mergesort. Context switching requires each thread to have an allocated stack, which takes time to set up. Additionally, each context switch requires time to set the context for a specific core to that of the function it now needs to run. Therefore, working on a single core did not seem viable for the purpose of this thesis, as it would likely lead to inefficiencies and decreased performance.

3.1.1 Working on multiple cores

The first method for implementing a multicore merge sort involved using threading and a scheduler. When we split a given list into two halves, we would create a thread assigned to sort each sublist. These two lists would be added to the queue of available threads, after which the job of merging the two lists could be added to the back of the queue. The merge job would have to check whether the two sublists have finished being sorted, but otherwise, assuming a round-robin scheduler, it would automatically allow for the correct ordering for the parallelized merge sort algorithm. However, this approach introduces multiple race conditions; the primary one being implementing a queue capable of handling concurrent access. The simplest method would be to implement a lock, allowing mutual exclusion when adding to and removing from the queue. The downside of implementing a locking queue is that synchronization can lead to performance issues. Another method would be to implement a lock-free queue, which should remove any synchronization issues and be a viable solution. However, I ran into problems with a child thread (created to sort a sublist) notifying its parent when it has finished sorting the sublist. Each core would need some way of keeping track of the current thread running and telling the parent thread (whose job is to sort the child's list and another sublist) when it has finished. Although this approach is possible, it was scrapped for the following design due to these challenges.

To simplify initialization, a single core would have the job of splitting the initial array into sublists until every core has a single sublist to work on. While doing the splitting, it would also create threads which have the job of merging the sublists once they are finished. This approach would remove the need for a queue and scheduler in the first place, as each core would, through its own core ID, know what thread it would have to complete. The issue of

communication with the parent merge would still exist, but the same index used to find the thread initially could be used to find the parent as well. The implementation of this approach can be seen in the Implementation section.

3.2 Context switching

Context switching is an integral part of multithreading. It is the act of storing the state of the process so that it can later be restored and resume execution at a later point. Not only can one save the state of the current process, one can also modify the context such that instead of continuing at the point of initialization it instead continues execution at a target function. Modifying specific registers would also allow for preset values to be loaded as function parameters. When creating the thread structure mentioned previously, it would then be possible to create context for computing both the mergesort and merge for a given section of a sublist.

3.3 Memory

3.3.1 Getting system information

To properly use the memory, we need some information about the system we are working on. As this thesis is created on a QEMU system, we are able to get the system information by running the following:

```
qemu-system-riscv32 -machine virt \
-machine dumpdtb=riscv32.dtb
```

This creates a Device Tree Blob (dtb) data file, which contains information about the virt qemu-system-riscv32 virtual machine. This format is not usable by us at the moment, but by using the Device Tree Compiler (dtc) package, we can convert it from the binary dtb format to a human-readable dts format.

```
sudo apt install dtc
dtc -I dtb -O dts -o riscv32.dts riscv32.dtb
```

Opening the file up in your favorite text editor you should see a lot of information regarding the qemu-system-riscv32 virtual machine. First we note, that the Devicetree specification states, that the memory node describes the physical memory layout for the system. As we want the programs stack to live within the memory section, this is section we should find information about starting address and length of the memory section. The memory node has two required sections, first the device_type, which must simply be 'memory', and secondly the reg value. The reg value "Consists of an arbitrary number of address and size pairs that specify the physical address and size of the

memory ranges' [4]. Furthermore, it is stated, that the property name reg has the value encoded as a number of (address, length) pairs. It also states, that the number of <u32> cells required to specify the address and length are bus-specific and are specified by the #address-cells and #size-cells properties in the parent of the device node. Looking through our riscv32.dts file, we find the relevant information to be:

```
#address-cells = <0x02>;
#size-cells = <0x02>;

memory@80000000 {
  device_type = "memory";
  reg = <0x00 0x80000000 0x00 0x8000000>
};
```

With the information previously provided, we know that the starting address of the memory section is at address $0x00 + 0x80000000 = 0x80000000$ and has a size of $0x00 + 0x8000000$ bytes, which is equivalent to 128MB. To allow space for saving static values such as .bss and .data sections

3.3.2 Memory Layout

As mentioned, all created threads need to have a separate stack for context switching to work. Thus, when creating a thread, we have to allocate some location in RAM to the task the thread has to perform. In Figure 1, this memory area is denoted with the "thread x STACK" area. As a design choice, I chose to separate the thread stacks in the opposite end of RAM from where the core stacks would be allocated. That way, if I ran into a thread stack overflow, I would know it was caused by the threads themselves and vice versa with the core stacks. Different sizes of thread stacks have not been tested, but a size of 1024 seemed to work without issues on relatively small lists.

At the end of the RAM section is where the individual core stacks would be allocated. Again, the specific size has not been tested, but with 8 cores and the chosen stack size of 2048 bytes, assuming a thread stack size of 1024 bytes, we would be able to have approximately 130.032 individual thread stacks without the two different stack areas overlapping.[2]

[2] $0x88000000 - 8 * 0x800 = 0x87ffc000$ would be the end of core stacks. $0x87ffc000 - x * 0x400 = 0x80100000 \implies x = 130.032$. However, as the .data, .bss and .text sections might be saved in the RAM area by the linker script, we don't know the definitive value of _stack_size until the program is fully compiled.

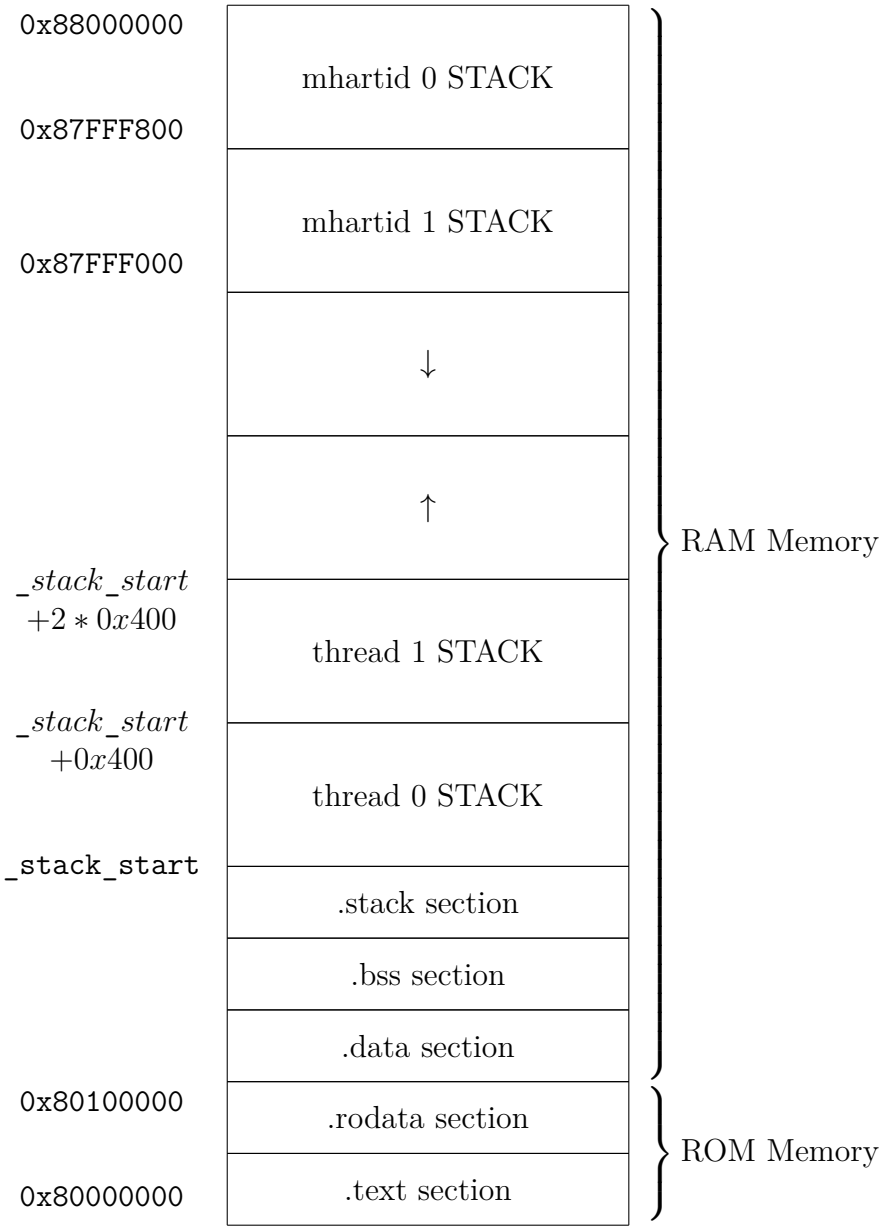


Figure 1: Memory layout of the QEMU virt machine with a core stack size of 2048 bytes and a thread stack size of 1024

Listing 1: Installing QEMU

```
git clone git clone https://github.com/qemu/qemu  # Clone the qemu repo
./configure --target-list=riscv32-softmmu  # Configure the 32-bit RISC-V target
make -j $(nproc)  # build the project with all num cores jobs
sudo make install
```

Listing 2: Installing LLVM compiler infastructure with RISC-V 32-bit as native target.

```
# Dependencies
sudo apt-get -y install \
  binutils build-essential libtool texinfo \
  gzip zip unzip patchutils curl git \
  make cmake ninja-build automake bison flex gperf \
  grep sed gawk python bc \
  zlib1g-dev libexpat1-dev libmpc-dev \
  libglib2.0-dev libfdt-dev libpixman-1-dev

# Installing the RISC-V-gnu-toolchain with llvm support
git clone https://github.com/riscv-collab/riscv-gnu-toolchain  # clone
riscv-gnu-toolchain
cd riscv-gnu-toolchain  # change directory
./configure --prefix=/opt/riscv --with-arch=rv32gc -disable-linux --enable-llvm
# prefix is install path used by llvm
sudo make -j$(nproc)
cd ..
popd
```

4 Implementation

The implementations created as part of this bachelor thesis aimed to make use of the LLVM compiler infrastructure. LLVM is a collection of modular and reusable compiler and tool chain technologies, most notably for this project is the clang compiler. Furthermore, QEMU will be used extensively while testing the implementations.

4.1 Dependencies

4.1.1 QEMU

Following the instructions by RISC-V's getting started guide we can build the QEMU RISC-V system emulators by running the code provided in Listing 1[6].

4.1.2 Installing LLVM compiler infrastructure

When it comes to clang there are two methods of installing, that are relevant to this project. If running on a Debian based system, then you

can simply install llvm-tools package. The issue with this approach is that the general build is for use with the current system installation is on, which unless you are running a RISC-V computer architecture natively will lead to issues when trying to cross compile if the given targets use any of the standard libraries, such as freeRTOS. A fix to this issue is to explicitly tell clang to make use of the RISC-V gnu tool chain on every compilation.

The second approach is to build LLVM with the RISC-V 32-bit target as the native target. This approach is documented in Listing 2. After installation it is important to add both clang build and RISC-V gnu tool chain to PATH. However, adding the following flags to compilation should lead to the same results, although the second approach is used throughout this project.

- –sys-root=Path to RISC-V install/riscv64-unknown-elf
- –taget=riscv32
- –gcc-toolchain=Path to RISC-V install

4.2 Creating a linker script

The linker script is used to tell the linker which parts of the file to include in the final output file, as well as where each section is stored in memory. As we are working on an embedded system, we have to stray from the default and create our own linker script. The clang uses the LLVM lld linker, which is compatible with the general linker scripts implementations of the GNU ld linker [7]. Thus, we can make use of the GNU ld manual for modifying the linker script in freeRTOS for our bare metal application instead of writing the entire thing from scratch [2].

```
OUTPUT_ARCH('riscv')
ENTRY(_start)

MEMORY
{
/* Fake ROM area */
rom (rxa) : ORIGIN = 0x80000000, LENGTH = 1M
ram (wxa) : ORIGIN = 0x80100000, LENGTH = 127M
}
```

First, we must specify that we want the RISC-V architecture and designate the entry point of the program at a function named '_start,' which we will define later. Second, we define the MEMORY area to consist of both a writable memory region and a read-only memory region. We name these regions 'ram' and 'rom,' respectively. With that we move on to define the SECTIONS element of the linker script.

```
SECTIONS
{
.text : ALIGN(CONSTANT(MAXPAGESIZE))
{
  *(.text .text.*)
}

.rodata : ALIGN(CONSTANT(MAXPAGESIZE))
{
  *(.rdata)
  *(.rodata .rodata.*)
}

.data : ALIGN(CONSTANT(MAXPAGESIZE))
{
  *(.data .data.*)
  /*RISCV convention to have __global_pointer
  aligned to 8 bytes*/
  . = ALIGN(8);
  PROVIDE( __global_pointer$ = . + 0x800 );
}

.bss : ALIGN(CONSTANT(MAXPAGESIZE))
{
  *(.bss .bss.*)
}

/* It is standard to have
the stack aligned to 16 bytes*/
. = ALIGN(16);
_end = .;

.stack : ALIGN(CONSTANT(MAXPAGESIZE))
{
  _stack_top = ORIGIN(ram) + LENGTH(ram);
}
}
```

The text, rodata, data and bss sections follows the same general procedure. We align the section to the maximum size of a page, and the match all the data which we care about for the given sections. I have opted to disregard specifying where the linker has to save all the data, and instead opted to let the linker itself find a suited location looking at the attributes we gave to memory previously. In the data section, we also provide a global pointer, which is used to access global variables within our later code implementation. [3]

Then I align the end with 16 bytes as is the custom. Reason this is moved outside the stack is that I use the _end variable later, and as such it should be aligned as well. Then within the stack section we define the _stack_top as being the end of the random access memory section (ram), as the stack grows downwards.

[3] In general the global pointer is used together with an offset in much the same manner as a stack pointer and offset.

4.3 Getting into the main function

In the linker script we specified the entry point of our program as _start. Next up is implementing said entry point in assembly. Within a new assembly file we add the following.

```
.extern main
.section .init
.globl _start
.type _start,@function

_start:
  .cfi_startproc
  .cfi_undefined ra
  .option push
  .option norelax
  la gp, __global_pointer$
  .option pop
  // load _stack_top into the sp register
  la sp, _stack_top
  add s0, sp, zero

  // argc, argv, envp is 0 and jump to main
  li  a0, 0
  li  a1, 0
  li  a2, 0
  jal main
  .cfi_endproc // We end the proccess
```

First we specify that externally there will be implemented a main entry point. Next we tell the linker to save the following code in the .init section and initialize a global label _start, and note that it is a function.

Next the _start is defined, and define .cfi_startproc such that we have an entry in the .eh_frame. Next we define the return address register(ra) as being undefined, as we are in the start of the entire program. Since the linker usually relaxes addressing sequences to shorter GP-relative sequences when possible, the initial load of GP must not be relaxed [1]. However, we do not need the same for loading the _stack_top into the sp register, and then also save it in the s0 register, which stores the frame pointer. Then the last step is loading 0 into argc, argv and envp and the jump to the externally defined main function.

4.4 Printing with UART

As we are working with bare metal, the standard printf function will not work. However, QEMU allows the use of universal asynchronous receiver / transmitter (UART) protocol, which allows us to implement a printf function, that writes to the terminal without the need for the QEMU GUI interface. Furthermore, there exists open source bare-metal versions of the printf function, which makes the effort of printing a lot easier. One such example is

Listing 3: Implementation of putchar of stdarg lib

```
#include <stdint.h>
#define UART_ADDR 0x10000000
#define LCR 0x03          // Line control register
#define LSR 0x05          // Line status register
#define FCR 0x02          // FIFO control reigster
#define RBR 0x00          // Receiver buffer register
#define IER 0x01          // Interrupt enable register
#define LSR_THRE 0b110000 //
void uart_init(void) {
  volatile uint8_t *ptr = (uint8_t *)UART_ADDR;

  // Set word length to 8 (LCR[1:0])
  *(ptr + LCR) = 0b11;

  // Enable FIFO (FCR[0])
  *(ptr + FCR) = 0b1;

  // Enable receiver buffer interrupts (IER[0])
  *(ptr + IER) = 0b1;
}

static void uart_put(uint8_t c) { *(uint8_t *)(UART_ADDR + RBR) = c; }
static uint8_t uart_get(uintptr_t addr) { return *(uint8_t *)(addr); }

void putchar(unsigned char c) {
  volatile uintptr_t ptr = (uintptr_t)UART_ADDR;
  // make sure there is nothing else in FIFO
  while ((uart_get(ptr + LSR) & LSR_THRE) == 0) {
    // do nothing
  }
  // add the char to reciver buffer register
  uart_put(c);
}
```

Georges Menie's stdarg, which depends on a single function called putchar, which has to take a character and place it somewhere. One such implementation can be seen in Listing 3. Again this code is a modified version of what is seen in freeRTOS. From the previous dtc file created in Section 3.3.1 there is also information regarding the UART configuration.

```
serial@10000000 {
  interrupts = <0x0a>;
  interrupt-parent = <0x03>;
  clock-frequency = "\08@";
  reg = <0x00 0x10000000 0x00 0x100>;
  compatible = "ns16550a";
};
```

First it states the address of the uart is, 0x10000000, and it states that it is ns16550a compatible [10]. In Listing 3 you can see the needed implementations for the uart to work, which is a slightly modified version of what is seen in freeRTOS. First, we define the register values as stated in the uart documentation, and initialize the uart. Next, we define af uart_put and uart_get function to read from and write to addresses. With this we are able to define the putchar function. It works by busy waiting until the FIFO queue is empty, and then gives the character to the reciver buffer register.

4.5 libucontext

5 Evaluation

5.1 Testing

NUM CORES HAS TO BE A POWER OF TWO

6 Conclusion

References

[1] *GNU Binutils*, 2.31 edition.

[2] *LD Linker Scripts*, 2.42 edition.

[3] QEMU Project Developers. *QEMU RISC-V System emulator*, 2023.

[4] devicetree.org. *Devicetree Specifications*, v0.4 edition, 2023.

[5] Blem et. al. Power struggles: Revisiting the risc vs. cisc debate on contemporary arm and x86 architectures. *University of Wisconsin - Madison*, 2013.

[6] RISC-V Foundation. *RISC-V Getting Started Guide*, 2018-2020.

[7] llvm. Linker script implementation notes and policy.

[8] Sanjay Patel & Wen mei W. Hwu. Accelerator architectures. *IEEE Computer Society*, 2008.

[9] Krste Asanović & David A. Patterson. Intruction sets should be free: The case for risc-v. *Electrical Engineering and Computer Sciences University of California at Berkeley*, 2014.

[10] SIDSA. *UART 16550 IP*, 2000.