UNIVERSITY OF COPENHAGEN
DEPARTMENT OF COMPUTER SCIENCE

Bachelors Thesis

Simon Vinding Brodersen

RISC-V based computers in the data center

Advisor: Philippe Bonnet

June 4, 2024

Contents

1 Introduction

1.1 Context

Data centers 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) served as drop-in replacements for magnetic disks, utilizing similar interfaces to ensure 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 Central Processing Unit(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?

1. **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. Moreover, parallel versions of the merge sort algorithm should be feasible on bare-metal[1].

2. **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 on a bare-metal processor.

1.3 Approach

For this thesis, an experimental approach was taken. First, a feasible design 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

For this thesis, the following contributions have been made:

1. Present available design patterns when developing a computational storage device
2. Designed and implemented a specific merge sort algorithm meant for running on a bare-metal processor as described in Section 3.2.3.
3. Evaluate the implementation on lists of varying sizes.
4. Evaluate the viability of custom bare-metal applications for later use as a Computational Storage Device.

[1]Bare metal is described further in Section 3.2.3.

1.5 Related work

Marcelino et al. [11] evaluate three hardware sorting units implemented with specific Field Programmable Gate Arrays (FPGAs). One of these being the FIFO[2]-based merge sorting machine, where they present a merge sorting structure for a FIFO embedded system merging. This implementation assumes two sorted input lists and they later propose a hybrid solution using Insertion and FIFO based merge sorting. With this they find that the FPGA hybrid insertion and FIFO-based merge sorting sees speed-ups between 1.6 and 15 time compared to a quick sort pure software solution.

Jackson et al. [8] look for faster sorting algorithms used for flash memory embedded devices. They provide a merge sort for sorting with minimal memory usage which aims to reduce the number of WRITES to flash memory. This implementation would only need two memory buffers. They find that when sorting large data sets with small memory the proposed algorithms reduces I/Os and execution time by about 30%.

Lobo et al. [10] compare the performance of different types of merge sort algorithms. Within their testing they find that the serial and parallel merge sort discussed within this paper have a similar amount of resource utilization, but find the delay of the parallel merge sort to be a lot smaller than the serial counterpart. As such, they theorize that this implies the parallel execution to be much faster than the serial.

[2]Short for First In First Out

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 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[12]. 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.[13].

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[3] based machines with comparable and in some cases better performance than that of a Complex Instruction Set Computing(CISC) alternative.[6] RISC-V aims to provide the same benefits in an

[3]Short for Advanced RISC Machine.

open sourced environment. With this RISC-V, more specifically the 32-bit version, was chosen as the ISA for development 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 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

QEMU

QEMU is a system emulator, which has the capabilities of emulating both a 32-bit and 64-bit RISC-V processor [4]. With QEMU I am able to create code intended for a processor running the RISC-V instruction set even if my development environment is running a different ISA. For the purposes of this thesis it is the RISC-V 32-bit version of the QEMU virtual machine that will be used.

LLVM and RISC-V GNU Toolchain

The LLVM project is a collection of reusable compiler and toolchain technologies. Most notably for the context of this thesis clang. Clang is a gcc compatible 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 cross-compiler 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 inadequate 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 Requirements

As described in Section 1 the design goal is to implement a merge sort algorithm running on a bare-metal RISC-V processor. The implementation should be usable for lists of varying sizes, and should be easily customisable given different hardware specifications within the RISC-V ecosystem. This should all be done without the need of an operating system as described in Section 3.2.3.

3.2 Development platform

3.2.1 freeRTOS

RTOS stands for real-time operating system. The goal of an RTOS is to provide a small and simple design, which is easy to port to different architectures. Furthermore, freeRTOS provides fast execution speeds and methods for multi-threading, mutexes, semaphores, software timers, and more. FreeRTOS specifically is a leading open-source RTOS, fitting well with the open-source idea provided with RISC-V as well. Using an RTOS like freeRTOS would allow for creating a similar implementation of a parallelized merge sort as provided in this thesis. It would also facilitate easier portability across various architectures. However, using freeRTOS would always provide a small overhead compared to a complete bare-metal implementation and as the aim of this thesis was to assess the viability of high-performance processors in the data center, a more custom implementation was instead chosen.

3.2.2 Unikernel

A unikernel can be seen as a small footprint single-address space kernel. It allows a single application to run as if a fully-fledged operating system is in the background. Thus, it removes many of the obstacles associated with developing on bare-metal, as described in Section 3.2.3. Generally, these are used to create specialized images, which bridge the gap between having a fully-fledged operating system and working directly on bare-metal. However, in the field of RISC-V, the most prominent seems to be Nanos. Nanos aims to provide an alternative than the Linux operating system when creating images meant for virtual machines. This could be cloud computing where something like Docker might be used together with Linux today. Although unikernels can be created for specific hardware applications, it seems sparse in the RISC-V development environment.

3.2.3 Bare-metal

Also known as embedded system programming, bare-metal programming involves developing applications meant to run without an underlying operating system. This allows for interlinking with specific hardware and enables a more customizable program tailored to the hardware's specifications. However, this approach comes with drawbacks, such as the lack of a standard library. Default memory allocation functions like malloc in C are not implemented on bare-metal systems because they require an operating system to function. Consequently, developers must handle memory allocation and deallocation themselves for more advanced programs. Additionally, standard printing for debugging purposes is left up to the developer to implement, requiring them to understand hardware specifications, such as UART printing. [14] [4]

The management of memory often leads to some odd errors when attempting to debug. Most notably, where one might generally be used to gdb[5] to detect, when we have reached a stack overflow that is not possible when working on bare-metal. It is the operating system, which usually tells a user-level program, that it is outside of its allocated memory space, but as we have no operating system unless we ourselves implement some error checking, then there is none. Instead errors show up as variables getting updated without intentionally meaning too.

However, with the benefits of customization, bare-metal was chosen as the development platform.

3.3 Single vs Multicore

The merge sort algorithm can be executed both with serial and parallel computing. Merge sort, in itself, has a recursive pattern of dividing the list into subsections and sorting each subsection for itself. This, combined with parallelized algorithms generally showing better performance than their serial counterpart, led to the choice of a multicore development for the implementation in this thesis. [3] This decision provides new challenges not present in serial computing, as the goal is to access the viability of custom sorting algorithms on bare-metal.

[4]The QEMU virt machine is compatible with UART, redirecting any output from stdout to the terminal instance running the QEMU virt machine by default. In Section 5.2, this redirection is used for debugging the implementation, with output being redirected to a file instead.

[5]gdb is the GNU Project Debugger later used for debugging in Section 5.1.

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 sub list. 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 sub lists 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 sub list) notifying its parent when it has finished sorting the sub list. 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 sub list) 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 sub lists until every core has a single sub list to work on. While doing the splitting, it would also create threads which have the job of merging the sub lists 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.

3.4 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. This is done by saving the values of the registers, such that they all can be restored at a later point to continue execution.

When creating the thread structure, as mentioned previously, it would then be possible to create context for computing both the merge sort and merge for a given section of a sublist.

3.5 Memory

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, 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 the 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 [5]. Furthermore, it is stated that the property name reg has the value encoded as a number of <u32> cells required to specify the address and length, which 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

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.[6]

[6] $0x8800000 - 8 * 0x800 = 0x87ffc000$ 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.

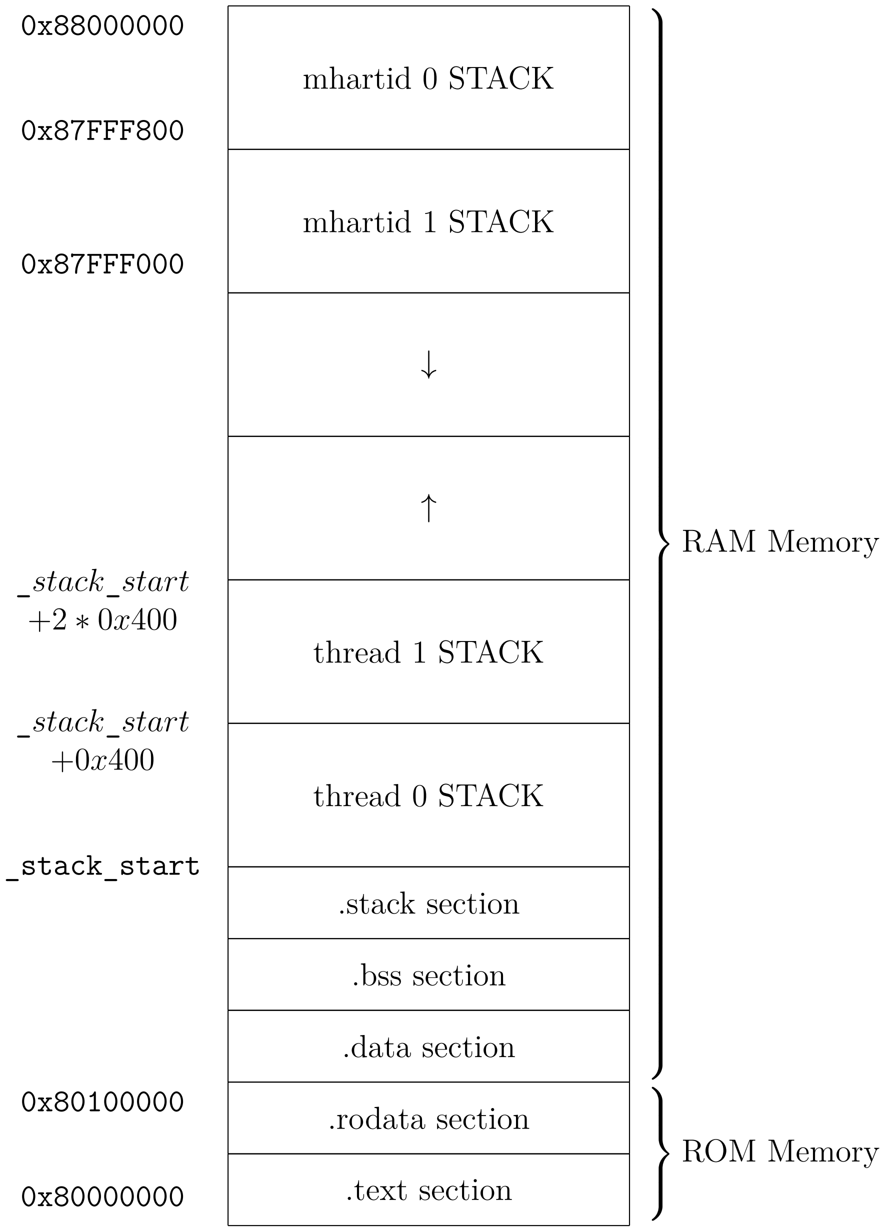


Figure 1: Memory layout of the QEMU virt machine with a core stack size of 2048 bytes and a thread stack size of 1024

Listing 1: Installing QEMU

```
git clone git clone https://github.com/qemu/qemu  # Clone the QEMU repo
cd qemu
./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

git clone https://github.com/riscv-collab/riscv-gnu-toolchain
cd riscv-gnu-toolchain  # change directory
./configure --prefix=/opt/riscv --with-arch=rv32gc -disable-linux --enable-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

QEMU

Following the instructions from RISC-V's getting started guide, we can build the QEMU RISC-V system emulators by running the code provided in Listing 1[7].

LLVM and RISC-V-gnu-toolchain

Although the LLVM Clang compiler comes with an available cross-compiler, I found that it often caused issues with missing header files compatible with my implementation. Furthermore, the LLDB debugger was unable to provide a working debugger for multicore remote debugging on QEMU. These are issues which might only be affecting me, as information revolving around the issues was scarce. As such, the following steps of building LLVM and the RISC-V 32-bit GDB may be obsolete but are left here as a known working toolchain. Running the code in Listing 2 installs a RISC-V compatible Clang compiler and GDB debugger in the /opt/riscv/ directory. For use outside this folder, make sure to add it to PATH.

libucontext

Libucontext is an open-sourced library that provides the ucontext.h C API. Most notably for the project of this thesis, it can deploy on bare metal RISC-V 32-bit with newlib. Building the library from scratch led to some issues on my end, and as such, the necessary files were copied and linked together with my implementation upon building. With this, I am able to use getcontext, makecontext, and setcontext, which allows me to do the necessary context switching described within Section 3.

4.2 Creating a linker script

The linker script is used to inform 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 deviate from the default and create our own linker script. The clang uses the LLVM lld linker, which is compatible with general linker scripts implementations of the GNU ld linker [9]. Therefore, 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].

Listing 3: Memory area defined in linker script

```
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.

Listing 4: Linker scripts SECTIONS.

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

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

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

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

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

  .stack : ALIGN(CONSTANT(MAXPAGESIZE))
  {
    . = ALIGN(8);
    PROVIDE(_stack_start = .);
    PROVIDE(_stack_top = ORIGIN(ram) + LENGTH(ram));
  } > ram
}
```

The text, rodata, data and bss sections follow the same general procedure. We align the section to the maximum size of a page and match all the data which we care about for the given sections. By specifying the > rom, we tell the linker to save the given section in the ROM section and the same is true for the > ram. From the Figure 1, we can see the RAM and ROM correspond to the RAM and ROM section of the figure.[7]

In the data section, we also provide a global pointer, which is used to access global variables within our later code implementation. The global pointer is used together with an offset to save global variables. As such we

[7]ROM stands for Read-Only Memory, and RAM stands for Random-Access Memory. Generally it is not necessarily needed to split the two up as done here, but it is a good practice to separate what can change and what cannot change in memory.

allow for 0x800=2048 bytes of global variables. With the implementation being quite reliant on global variables, it might need to be increased for lists of large sizes.

The last section is the .stack section. We align the starting of the _stack_start with 8 bytes. Generally not necessary in this instance, but still a good custom. This is the point from where each thread stack will be allocated. Next, we specify that the _stack_top will reside at the end of the RAM section such that we can allocate a stack for each core with an 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 this entry point in assembly. In a new assembly file, we add the following:

Listing 5: Assembly code for getting to main function.

```
.extern main
.extern secondary_main
.globl _start
.type _start,@function
#include "../include/defines.h"

_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
  csrr a0, mhartid
  bnez a0, 2f
  1:
    // argc, argv is 0 and jump to main
    li  a0, 0
    li  a1, 0
    jal main
  1:
    // loop
    j 1b
  2:
    la t1, STACK_SIZE
    li t0, 0
  1:
    andi sp, sp, -16
    beq a0, t0, 1f
    sub sp, sp, t1
    addi t0, t0, 1
    j 1b
  1:
    // argc, argv is 0 and jump to main
    li  a0, 0
    li  a1, 0
```

```
39      jal secondary_main
40    1:
41      // loop
42      j 1b
43      .cfi_endproc  // We should never really reach this
```

Lines 1-5 provide the setup for the assembly file. We specify that a main and secondary_main label will be defined outside of the file, that _start is a global label and that _start is a function type. At last, we include the defines.h file, which includes definitions of the STACK_SIZE.

On lines 7-13 we provide call frame information (CFI) for there being no return address and that the process begins here. Then when initializing the global pointer, we must specify options push, norelax, and pop as described in GNU binutils. [1] After linker relaxation, this would produce the anticipated code:

```
auipc gp, %pcrel_hi(__global_pointer$)
addi gp, gp, %pcrel_lo(__global_pointer$)
```

On lines 15-17, we load the value of _stack_top, which the linker provides through the linker script defined previously, and save it into the stack pointer (sp) register. We read the current machine hart identifier (mhartid), which contains a unique identifier for each core on the processor. This is what allows for differentiation between the different cores. Line 17 moves execution to line 27 if the machine hart identifier is not 0. As such, only mhartid 0 will be allowed to continue execution to line 22, where it jumps to the main function.

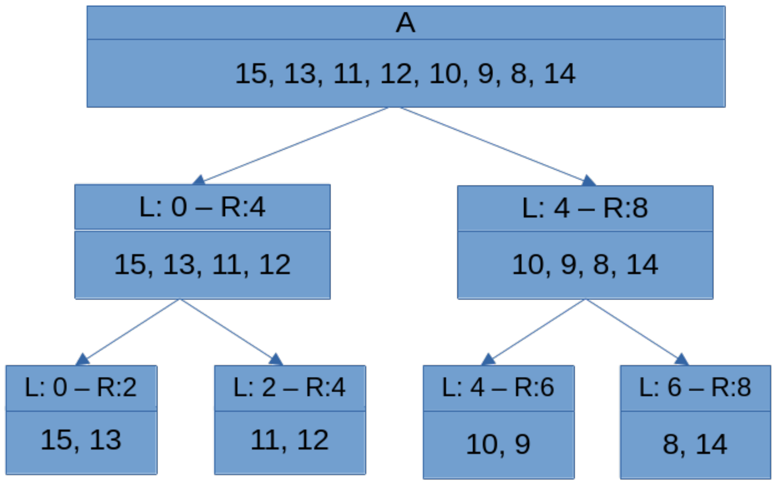
All other cores continue execution at line 27, where they load the value STACK_SIZE defined in defines.h into the temporary register t1. We also load immediately(li) the value 1 into the temporary register r1. Line 30 aligns the current value in the stack pointer to 16. Afterwards, we compare the value in register a0, which holds the value of the mhartid, to the value in t0. If they are equal, we jump to line 35, which makes us jump to the externally defined secondary_main function. Otherwise, we continue on line 32, where we subtract the register t1 (STACK_SIZE) from the value stored in the sp register. We increment the value in t0 by one, and jump back to line 30. With this loop, we are setting up a stack of size STACK_SIZE for all the different cores defined, such that we get the desired memory layout shown at the top of Figure 1.

4.4 Initializing the thread jobs

A pseudocode implementation for initializing the parallel merge sort algorithm is provided in Algorithm 1. Initially, it should be noted that it is assumed NUM_CORES is a multiple of 2. Next, as described in Section 3.3, it is never in our interest to perform unnecessary context switching when

Algorithm 1 Initialization of the threads

```
Require: threads[] # Empty thread array
 1: procedure INITIALIZE_THREADS($A$)
 2:     $depth \leftarrow MostSignificantBit(NUM\_CORES)$
 3:     $idx \leftarrow 0$
 4:     for $i = 0, i < depth$ do
 5:         if $i == 0$ then
 6:             # Top level thread
 7:             $ct \leftarrow threads[idx]$
 8:             mid ←Length(A)/2
 9:             $thread\_create(ct, parallel\_merge)$
10:             $ct.l = 0$
11:             $ct.mid = mid$
12:             $ct.r = Length(A)$
13:             idx++
14:             continue
15:         end if
16:         $k \leftarrow 2^k$
17:         for $j = 0, j < k$ do
18:             $parent\_thread \leftarrow threads[(idx - 1)/2]$
19:             $ct \leftarrow thread[idx]$
20:             if $j\%2 == 0$ then
21:                 $ct.l = parent\_thread.l$
22:                 $ct.r = parent\_thread.mid$
23:             else
24:                 $ct.l = parent\_thread.mid$
25:                 $ct.r = parent\_thread.r$
26:             end if
27:             $ct.mid = ct.l + (ct.r - ct.l)/2$
28:             if $i == depth - 1$ then
29:                 # Just regular merge sort
30:                 thread_create(ct, merge sort)
31:             else
32:                 # Merge given section
33:                 thread_create(ct, parallel_merge)
34:             end if
35:             idx++
36:         end for
37:     end for
38: end procedure
```



THREADS := [(A, 0, 8, MERGE), (A, 0, 4, MERGE), (A, 4, 8, MERGE), (A, 0, 2, MERGESORT), (A, 2, 4, MERGESORT), (A, 4, 6, MERGESORT), (A, 6, 8, MERGESORT)]

Figure 2: Example of partitioning a random list using Algorithm 1

working on a single core. Consequently, we can calculate the depth desired for our merge tree by employing the number of cores. This may be achieved by taking $\log_2(NUM\_CORES)$ or, equivalently, by examining the most significant bit of NUM_CORES since it is a multiple of 2. At line 5, an early escape mechanism is implemented which takes care of the single thread that will have no parent thread. If we are not positioned at the very top level of the merge tree, we assign the variable 'k' as the number of threads for the given level and partition the list to create threads targeting a specific sub list. Lines 20 - 27 address the task of determining which subsection of the list each thread is responsible for. Finally, line 28 ensures that we do not generate more active threads than what the available number of cores can effectively handle. Upon completion of the algorithm, a threads array will be produced containing $2 \cdot NUM\_CORES - 1$ different threads, each assigned with a specific subsection of the list to either merge or perform sequential merge sort on.

4.5 Reducing synchronization

The idea behind initializing the threads array, as described in Section 4.4, is that we can reduce the need for synchronization between threads once the partitioning of the list is done. An example can be seen in Figure 2 with the corresponding finished THREADS array. This example would be on a system with 4 cores. As seen in Section 4.3, each core has a unique **id**, mhartid, from 0 to 3. With this, each core can index into the list with the number of jobs it has already completed and the length of the array list such that:

$$index = LENGTH(A) - \text{NUM\_COMPLETED\_JOBS} \cdot id \qquad (1)$$

With this, each thread is capable of retrieving a thread job without having to consider the state of any other thread, as the job is preassigned during the initialization. There are two caveats to this. Each merge must still wait on the child thread (the direction of the arrow in Figure 2) to finish before starting a merge.

5 Evaluation

5.1 Testing

Testing has been accomplished by providing a random_numbers.py file. With three separate integer parameters, it creates a random list using Python's standard random library. The inputs include a lower and upper bound for the list to generate, together with the length of the list. Once the list is created, running make will create a .elf file, which contains the parallel merge sort algorithm with the unsorted list hard-coded within. Once the .elf file is loaded on a RISC-V processor, it will immediately begin sorting the hard-coded list. The implementation also needs the value NUM_CORES defined within the Makefile, where it both defines a constant NUM_CORES for the .elf file to use, and the same value is used for running the QEMU virtual machine.

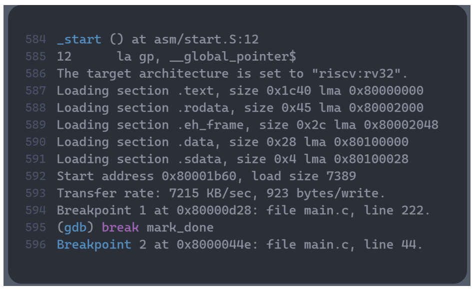
This gives the following work flow for creating and running a test:

- Change directory to src/bare_metal.
- Run random_numbers.py to generate alist.c with an unsorted list.
- Change the NUM_CORES variable in the Makefile to the desired number of cores.
- Run "make clean" to remove all files built with previous settings.
- Run "make test" to generate the .elf file and host QEMU. This step will create a test.txt file, as the stdout of QEMU is redirected to said file.
- Run "python3 validate.py". This reads the test.txt file, which sorts the unsorted array with Python's built-in sort function, and compares that sorted array with the one produced by the .elf file running in QEMU.

Debugging

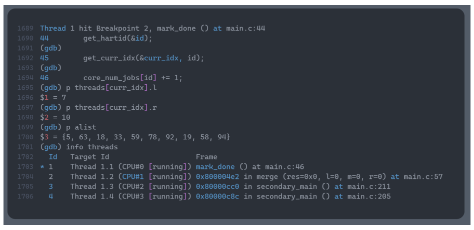
To access and debug the multicore system, QEMU provides a way for a remote gdb-server to connect at the start of execution.[4] In a system where each core is equivalent, gdb will automatically detect the cores, but will display them as threads. This allows one to debug the QEMU virtual machine with the same methodology used when debugging multithreaded execution. The workflow shown in Figure 3 is as follows:

- Change directory to src/bare_metal.



```
_start () at asm/start.S:12
12      la gp, __global_pointer$
The target architecture is set to "riscv:rv32".
Loading section .text, size 0x1c40 lma 0x80000000
Loading section .rodata, size 0x45 lma 0x80002000
Loading section .eh_frame, size 0x2c lma 0x80002048
Loading section .data, size 0x28 lma 0x80100000
Loading section .sdata, size 0x4 lma 0x80100028
Start address 0x80001b60, load size 7389
Transfer rate: 7215 KB/sec, 923 bytes/write.
Breakpoint 1 at 0x80000d28: file main.c, line 222.
(gdb) break mark_done
Breakpoint 2 at 0x8000044e: file main.c, line 44.
```

(a) Connecting gdb by running "riscv32-unknown-elf-gdb and breaking at mark_done"



```
Thread 1 hit Breakpoint 2, mark_done () at main.c:44
44          get_hartid(&id);
(gdb)
45          get_curr_idx(&curr_idx, id);
(gdb)
46          core_num_jobs[id] += 1;
(gdb) p threads[curr_idx].l
$1 = 7
(gdb) p threads[curr_idx].r
$2 = 10
(gdb) p alist
$3 = {5, 63, 18, 33, 59, 78, 92, 19, 58, 94}
(gdb) info threads
  Id   Target Id                     Frame
* 1    Thread 1.1 (CPU#0 [running]) mark_done () at main.c:46
  2    Thread 1.2 (CPU#1 [running]) 0x800004e2 in merge (res=0x0, l=0, m=0, r=0) at main.c:57
  3    Thread 1.3 (CPU#2 [running]) 0x80000cc0 in secondary_main () at main.c:211
  4    Thread 1.4 (CPU#3 [running]) 0x80000c8c in secondary_main () at main.c:205
```

(b) Hit breakpoint, where alist is sorted by core 1 from index 7 to 10 (10 exclusive).

Figure 3: Debugging the QEMU virtual machine with 4 cores.

- Initialize the alist.c file with an array and a given size.
- Run "make debug" to initialize the QEMU virtual machine.
- In a separate terminal instance run "riscv32-unknown-elf-gdb". This will automatically run the commands in the ".gdbinit" file and connect to the QEMU virtual machine.
- Break at mark_done. This function is run whenever a given thread job is finished.
- Run "info threads" to get an overview of all threads.

As an example, Figure 3 shows a list of size 10, where core 1 has sorted the subsection of the list from index 7 to 10. This section corresponds to one fourth of the list, as the number of cores available is 4.

5.2 Validation

When running a test, the ELF file first prints the unsorted array, and then once sorting is done, it prints it again. When executing "make test", the QEMU virtual machine outputs the stdout to a file called test.txt. Subsequently, invoking python3 validate.py reads this file to ascertain if sorting was performed correctly. In Table 6, an overview of some tests I conducted can be seen. On the left, the formatting is specified as the lower bound of randomly selected numbers, the upper bound, and the number of random elements in the list. A value of 'pass' signifies that the validate.py file executed without throwing an assertion error. The tests were initially performed with a global

Table 1: Table of tests run

	NUM_CORES					
Lower:Upper:Number	2	4	8	16	32	64
-100:0:100	pass	pass	pass	pass	pass	pass
0:100:100	pass	pass	pass	pass	pass	pass
-50:50:100	pass	pass	pass	pass	pass	pass
-1000:1000:1000	pass*	pass*	pass*	pass*	pass*	pass*

*This run initially failed due to stack overflow. After increasing the size of STACK_SIZE to 8192 bytes, THREAD_STACK_SIZE to 8192 and GLOBAL_STACK_SIZE 10MB bytes it worked.

stack of 0x800,[8] a STACK_SIZE of 2048 and a THREAD_STACK_SIZE of 10,240. If a failure occurred, modifications were made to these three values in an attempt to pass the test.

5.3 Future work

The implementation proposed in this thesis is more a proof of concept, and as such comes with a few shortcomings. Firstly, with the current implementation there is a heavy reliance on the GLOBAL_STACK_SIZE. The array to be sorted is hard-coded and stored on the global stack, which can occupy substantial space when dealing with large arrays. Furthermore, the array of thread jobs is also saved on the global stack. With the current implementation each thread type has a size of 848 bytes. Thus, a solution that reduces overall usage of the global stack, or a better method of detecting the size of the global stack would allow for a more reliable solution.

Whenever a merge sort splits the array into two halves, there is created a new thread with an allocated stack size of THREAD_STACK_SIZE. The top-level thread job, which has to sort the entire array, must first create a copy of the array before it can sort in place. The same goes for all other thread jobs, but the amount of the array they need to copy is halved for each level we move down in the merge sort tree shown in Figure 2. In theory, this allows for halving the size of the allocated thread stack for each level we move down. This does not take into account the constant space needed for the declared variables, so the thread stack size might instead be represented

[8]defined in ram.ld. Equates to 2048 bytes.

by:

$$\text{THREAD\_STACK\_SIZE} = \text{INITIAL\_STACK\_SIZE}/2^i + C \qquad (2)$$

Where i is the depth level, and C is some constant used to store the variables needed during computation, this constant C would then be different depending on whether the given thread has to perform merge sort or just merge over the array. Experiments determining the size of C could be carried out to optimize the memory better.

Furthermore, in regards to memory, the application does not check whether memory is out of bounds. Rather, it assumes that the sorting can be done with the allocated memory, and simply lets the application run if there is a stack overflow on any of the cores or thread stacks. The bounds checking would be extra overhead, but might still be something worth looking into.

It is not possible for performance testing as the application is running on a QEMU version. Implementing a timing library might be needed to access whether the performance of multiple cores is significant.

As it stands, the implementation also requires that the number of cores is a power of 2. This is due to how the initialization builds the threads up. This issue has a few ways it can be fixed. Firstly, instead of stopping at the same level for all the merge sorts, it could theoretically split only a few of the sub lists given, such that instead of having to be a power of 2, it would instead only require that it is a multiple of 2. This would still allow for all cores to work on a sublist at the same time. Another approach would be to allow a subset of the number of cores to busy loop until the two child threads as described in Section 4.4. The former approach seems more promising, but is left as something which can later be addressed.

As it stands, the only method of loading an array onto the processor is by hard coding the list into the .elf file. Although this works to show the entire sorting algorithm, it would not be useful in an actual data center unless some sort of communication protocol is established between the host CPU and the computational storage device. This communication is left as a problem that later has to be solved.

6 Conclusion

6.1 Summary of results

This thesis discusses the viability of implementing custom-made solutions for computational storage devices. In the first part of the thesis, the problem is presented along with the background. Next, the approach of implementing a bare-metal merge sort application on the QEMU virtual machine is proposed. A design for a merge sort algorithm that does not make use of any queue is suggested to reduce the need for synchronization. With this, a method of memory allocation is discussed and the design is carried out making use of context switching. The implementation of the merge sort algorithm is tested with varying list sizes as well as a varying number of cores utilized. Finally, possible improvements are presented in future work and left as open research questions.

6.2 Takeaways

In Section 1.2 two questions were proposed and aimed to be answered throughout this thesis. As such, these will build the foundation of the following conclusion.

What computation should be handled by a storage device?

Within the context of an academic thesis, it is demonstrated that intricate computations, such as sorting, are indeed feasible to implement on a bare-metal storage device. However, this endeavor does not come without its share of complications. These include crafting a custom linker script and devising custom assembly scripts, in addition to other obstacles left unmentioned within the scope of this thesis. These challenges may deter the average consumer; nonetheless, when viewed from the perspective of large-scale data centers, the potential for computational storage remains bright. As the CPU increasingly becomes a bottleneck in data transfer between a storage device and its host, computational storage has the potential to go beyond just sorting capabilities. Consequently, this thesis proposes that it is not the complexity of the computation at hand but rather the magnitude of demand from large data centers that will ultimately determine the success of optimized solutions within this realm.

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

The bare-metal approach presents numerous challenges that are not encountered when utilizing an underlying operating system. Despite these limitations, with perseverance, it is possible to overcome them and optimize a system specifically tailored to meet certain demands. In the context of large data centers, developing a custom computational storage device is both viable and advantageous. Nevertheless, it remains uncertain whether the performance enhancements justify the investment in implementing such systems.

References

[1] *GNU Binutils*, 2.31 edition.

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

[3] Darko Božidar and Tomaž Dobravec. Comparison of parallel sorting algorithms. Technical report, University of Ljubljana, 2015.

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

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

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

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

[8] Riley Jackson and Ramon Lawrence. Faster sorting for flash memory embedded devices. In *2019 IEEE Canadian Conference of Electrical and Computer Engineering (CCECE)*, pages 1–5, 2019.

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

[10] Joella Lobo and Sonia Kuwelkar. Performance analysis of merge sort algorithms. In *2020 International Conference on Electronics and Sustainable Communication Systems (ICESC)*, pages 110–115, 2020.

[11] Neto H. Marcelino, R. and J.M.P Cardoso. Sorting units for fpga-based embedded systems. *IFIP International Federation for Information Processing*, 271, 2008.

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

[13] 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.

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

Glossary

CISC	Complex Instruction Set Computing.
CPU	Central Processing Unit.
CSD	Computational Storage Device.
dtb	Device Tree Blob.
dtc	Device Tree Compiler, can convert a .dtb file to human readable format.
GCC	GNU Project Compiler. Compiler to make executable file.
GDB	GNU Project Debugger. A program used for debugging.
GPU	Graphics Processing Unit.
ISA	Instruction Set Architecture.
QEMU	Used to create a virtual machine running the RISC-V 32-bit ISA.
RISC	Reduced Instruction Set Computing.
SSD	Solid-state drives.