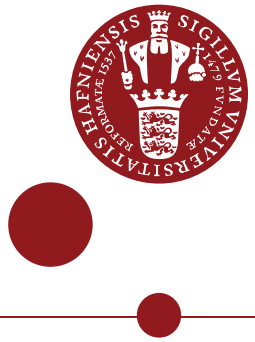
UNIVERSITY OF COPENHAGEN
DEPARTMENT OF COMPUTER SCIENCE



Bachelors Thesis

Simon Vinding Brodersen

RISC-V based computers in the data center

Advisor: Philippe Bonnet

May 25, 2024

Contents

1 Introduction

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. 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 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.

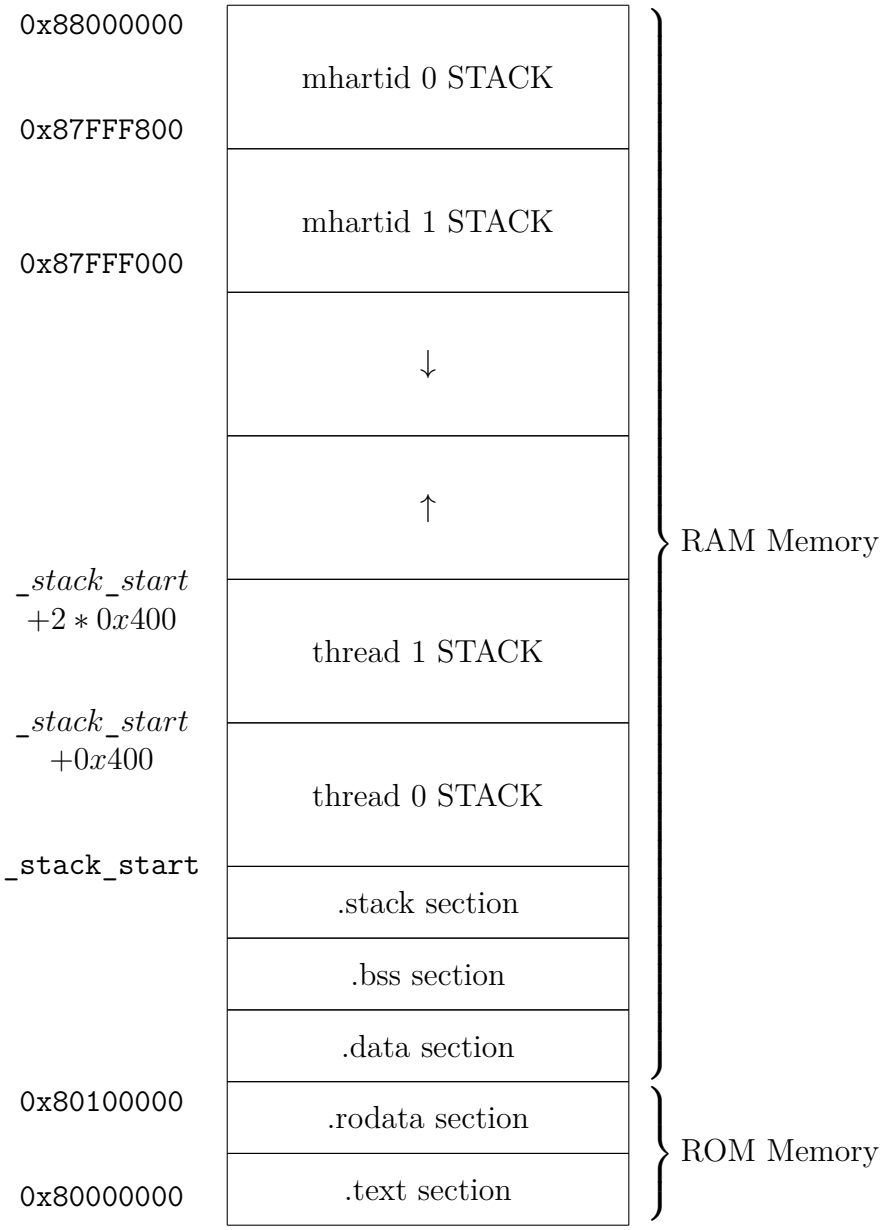


Figure 1: Memory layout of the QEMU virt machine with a core stack size of 2048 bytes and a thread stack size of 1024

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
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 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].

4.1.1 Installing LLVM compiler infrastructure

LLVM and RISC-V-gnu-toolchain

Although the LLVM clang compiler comes with an available crosscompiler, 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 the multicore remote debugging on QEMU. These are issues, which might only be affecting me, as information revolving the issues were scarce. As such, the following steps of building llvm and the RISC-V 32-bit gdb might 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 the use outside this folder, make sure to add it to PATH.

4.1.2 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.*)
  } > 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 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. 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 ROM and RAM section of the figure.[3]

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 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 necesarry in this instance, but stil a good custom. This is 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 with an offset can allocate a stack for each core.

[3]rom stands for read-only memory, and ram stands for random-access memory. Generally it is not necesarrily needed to split the two up as done here, but it is a good practise to seperate what can change and what can not change in memory.

4.2 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.

```
1  .extern main
2  .extern secondary_main
3  .globl _start
4  .type _start,@function
5  #include "../include/defines.h"
6
7  _start:
8    .cfi_startproc
9    .cfi_undefined ra
10   .option push
11   .option norelax
12   la gp, __global_pointer$
13   .option pop
14   // load _stack_top into the sp register
15   la sp, _stack_top
16   csrr a0, mhartid
17   bnez a0, 2f
18   1:
19     // argc, argv is 0 and jump to main
20     li  a0, 0
21     li  a1, 0
22     jal main
23   1:
24     // loop
25     j 1b
26
27   2:
28     la t1, STACK_SIZE
29     li t0, 0
30   1:
31     andi sp, sp, -16
32     beq a0, t0, 1f
33     sub sp, sp, t1
34     addi t0, t0, 1
35     j 1b
36   1:
37     // argc, argv is 0 and jump to main
38     li  a0, 0
39     li  a1, 0
40     jal secondary_main
41   1:
42     // loop
43     j 1b
44
45     .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. Atlast, we include the defines.h file, which includes definitions of the STACK_SIZE.

On lines 7-13 we give call frame information (cfi) for there being no return address and that the process starts 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 expected 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 28 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 28, 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 1. Line 31 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 36, which makes us jump to the externally defined secondary_main function. Otherwise, we continue on line 33, 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 31. 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.

4.3 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.