UNIVERSITY OF COPENHAGEN
DEPARTMENT OF COMPUTER SCIENCE

Bachelors Thesis

Simon Vinding Brodersen

RISC-V based computers in the data center

Advisor: Phillippe Bonnet

March 15, 2024

Contents

1 Introduction

2 Background

2.1 Accelerator-based Computer Architecture

The concept of offloading is nothing new in the world. Working in a team where each person has their speciality seems logical when we are talking about our day to day work environment. Having good comunication between each entity yet working on what we are best at seems logical when speaking about work. However, when it comes to the architectecture of computers, we have heavily relied on the shoulders of the base Centrel Processing Unit(CPU). An acellerator is a seperate substructure that is architected using a different set of objectives than the base processor. With this design, the substructure can be optimized for it's specific task, often leading to both performance increases and less energy consumption. [4] Prominent examples of acellerators include the Graphic Processing Unit(GPU), which is a part major part of most computers today.

2.2 RISC-V

Reduced Intruction Set Computer(RISC), more specifically the fifth version (RISC-V). Is an Instruction Set Architecture(ISA), that aims to make the process of making custom processors targeting a variety of end applications more feasible. Previous ISAs have often been created by private companies, which leads to patents and a need for a license to develop a specialised processor. These licenses could often take months to negotiate without mentioning the large sum of money involved. It is assumed that creating a free and open sourced ISA could reduce the barrier of entry and greatly increase innovation along with afforability.[5]

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 specialised accelerators. In RISC-V there are two primary base integer variants, RV32I and RV64I, whcih 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 gereral, RISC-V provides standard and non-standard extensions, where standard extensions should not conflict with other standard extensions, and the non-standard extensions are more highly specialised.

3 Implementation

The implmentations created as part of this bachorlor thesis aimed to make use of the LLVM compiler infastructure. LLVM is a collection of modular and reusable compiler and toolchain technologies, most notably for this project is the clang compiler and lldb debugger. Furthermore, cmake and qemu will be heavily used when automating the building process and virtualising a RISC-V 32-bit architecture.

3.1 Dependencies

3.1.1 QEMU

. QEMU is a system emulator, which has the capabilities of emulating both a 32-bit and 64-bit RISC-V CPU.[1] 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.[1][3]

3.1.2 Installing LLVM compiler infastructure

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 expletely tell clang to make use of the RISC-V gnu toolchain 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 toolchain 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

3.2 Bare metal C application

3.2.1 Getting system information

Now that there is a working toolchain, we can move on to the developement of the bare metal C version. First, we need information about the delvelopemnt environment we are currently working on, such that we are able to setup a stack for the bare metal C program. With QEMU it is possible to get the necesarry machine info by running:

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

This creates a Devicetree Blob(dtb) datafile, which contains information about

[1]Once installed make sure to add both llvm and riscv gnu toolchain to path. Both should be installed in the /opt/ folder.

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
git clone https://github.com/riscv-collab/riscv-gnu-toolchain  # clone
riscv-gnu-toolchain
cd riscv-gnu-toolchain  # change directory
./configure --prefix=/opt/riscv --enable-multilib
# prefix is install path used by llvm
# --enable-multilib allows us to compile for 32-bit
sudo make -j$(nproc)
cd ..

# Installing LLVM
git clone https://github.com/llvm/llvm-project.git # clone llvm-project
cd llvm-project
mkdir build
pushd build
sudo cmake -S ../llvm -G Ninja \
-DCMAKE_BUILD_TYPE="Release" -DBUILD_SHARED_LIBS=True \
-DLLVM_BUILD_TESTS=False \
-DLLVM_ENABLE_PROJECTS="clang;clang-tools-extra;\
lldb;lld;polly" \
-DCMAKE_INSTALL_PREFIX=/opt/llvm \
-DDEFAULT_SYSROOT="/opt/riscv/riscv64-unknown-elf" \
-DGCC_INSTALL_PREFIX="/opt/riscv" \
-DLLVM_TARGETS_TO_BUILD="RISCV" \
-DLLVM_USE_LINKER=lld
sudo cmake --build . --target install
popd
```

the virt qemu-system-riscv32 virtual machine. This format is not usuable 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 Device-tree specification states, that the memory node descirbes 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”. [2] 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.

3.2.2 Creating a linker script

The linker script is used to combine a number of object files together to create a single object or executable file. Now creating an entire linker script from scratch is quite cumbersome, and beyond the scope of this project. However, luckily for us, we are able to modify the default linker script quite easily to incorperate our information about the qemu virtual machine. First, to access the default linker script of lld run the following command.

```
ld.lld --verbose > riscv.ld
```

This creates a new file called riscv.ld, which contains the default linker script used. From here, we have to introduce a new variable, _stack_top, and define the memory section, which we previously found. Within the linker script, we add the following lines, to define the memory section.

```
MEMORY {
RAM(rwx) : ORIGIN = 0x80000000, LENGTH=128M
}
```

References

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

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

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

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

[5] 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.