



Bachelors Thesis

Simon Vinding Brodersen

RISC-V based computers in the data center

Advisor: Phillippe Bonnet

May 7, 2024

Contents

1	Introduction	3
2	Background	4
2.1	Accelerator-based Computer Architecture	4
2.2	RISC-V	4
3	Implementation	5
3.1	Dependencies	5
3.2	Bare metal C application	5

1 Introduction

2 Background

2.1 Accelerator-based Computer Architecture

The concept of offloading is nothing new. In a team where each member specializes, it seems logical when discussing our day-to-day work environment. Effective communication between entities while focusing on what we do best seems logical when speaking about work. However, when it comes to computer architecture, we have heavily relied on the Central Processing Unit (CPU). An accelerator is a separate substructure designed with different objectives than the base processor. With this design, the substructure can be optimized for its specific task, often leading to both performance increases and less energy consumption[7]. Prominent examples of accelerators include the Graphics Processing Unit (GPU), which is a major component in most computers today.

reduce the barrier of entry and greatly increase innovation along with affordability[8].

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, 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 more highly specialised.

2.2 RISC-V

Reduced Instruction 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

3 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 toolchain technologies, most notably for this project is the clang compiler and lldb debugger. Furthermore, qemu will be used extensively while testing the implementations.

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[3]. 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[5].¹

3.1.2 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

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

freeRTOS. A fix to this issue is to explicitly 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`
- `-target=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 development of the bare metal C version. First, we need information about the development 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 necessary machine info by running:

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

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

Listing 1: Installing QEMU

```
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 infrastructure 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;lld" \
    -DCMAKE_INSTALL_PREFIX=/opt/llvm \
    -DLLVM_TARGETS_TO_BUILD="RISCV"
sudo cmake --build . --target install
popd
```

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 Device-tree 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 0x80000000>
};
```

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 $0x00 + 0x80000000$ bytes, which is equivalent to 128MB.

3.2.2 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[6]. 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))
{
    *(.rodata)
    *(.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.²

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` vari-

²In general the global pointer is used together with an offset in much the same manner as a stack pointer and offset.

able later, and as such it should be aligned aswell. 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.

3.2.3 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 process

```

First we specify that externaly 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 adressing 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.

then afterwards define a `uart_put` function, which takes an unsigned character and sends it to the receive buffer register. The `putchar` function also makes sure that we do not overwrite a previous character by busy looping until the FIFO is clear.

3.2.4 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 opensource baremetal versions of the `printf` function, which makes the effort of printing a lot easier. One such example is 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 3.2.1 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, `0x10000000`, and it states that it is `ns16550a` compatible [9]. The specifications states the addresses of the different registers seen above. This allows us to first initialize the `uart`, and

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 register
#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 receiver buffer register
    uart_put(c);
}
```

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] RISC-V Foundation. *RISC-V Getting Started Guide*, 2018-2020.
- [6] llvm. Linker script implementation notes and policy.
- [7] Sanjay Patel & Wen mei W. Hwu. Accelerator architectures. *IEEE Computer Society*, 2008.
- [8] 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.
- [9] SIDSA. *UART 16550 IP*, 2000.