Getting Started with FreeBSD/RISC-V

BY MITCHELL HORNE

Introducing FreeBSD's newest CPU architecture: RISC-V. This article will bring you up to speed on what you need to know about the architecture and how you can build and run your very own FreeBSD/RISC-V system using QEMU.

Introduction to the RISC-V Architecture

RISC-V is an open and extendable ISA (Instruction-Set Architecture) developed by UC Berkeley over the last decade. It is a RISC architecture that was designed to learn from some of the pitfalls and mistakes made by existing computer architectures.

RISC-V has two main claims to fame. The first is that the ISA's specifications are released as open source under a BSD license, which means that RISC-V is completely free to use in both commercial and academic contexts. This, combined with the fact that it can be implemented quite simply, makes RISC-V attractive for use in academia, both for research implementations and as a tool for teaching computer architecture. For Silicon companies looking to avoid the heavy licensing fees they might pay for other architectures such as ARM, RISC-V provides a tempting cost-free alternative. Finally, there are projects like lowRISC (https://www.lowrisc.org/our-work/) and OpenHW Group's CORE-V (https://www.openhwgroup.org/news/2019/12/10/openhw-group-announces-core-v-chassis-soc-project-and-issues-industry-call-for-participation/) that aim to produce fully open-source, Unix-capable, RISC-V-based SoCs. This will make it possible to run systems that are completely free and open-source—from the OS and applications running on it, all the way down to the CPU cores themselves.

The RISC-V instruction set is also modular by design. The idea here is that a RISC-V implementation requires only a small number of integer instructions (the base ISA), and depending on the desired use-case, more instructions can be added in the form of official or unofficial ex-

tensions. As a surprising example, multiplication and division instructions are not required by the base integer instruction set. Instead, they come as part of the official "M" extension. The thinking is that a small, single-core, embedded core might not require this functionality, so by making it optional, the implementer is given the freedom to omit it. More complicated SMP systems might require additional functionality like atomics or hardware floating-point, and these are provided as official extensions as well. The "G" (general) extension includes several extensions necessary for running most Unix-like operating systems. The hope of the RISC-V designers is that by making the RISC-V freely available and easily extended, it can be adopted for a wide range of use cases—from the smallest microprocessors to large multi-CPU server platforms.

FreeBSD and RISC-V

RISC-V is FreeBSD's most recent and experimental supported architecture. Work on this port began in 2015, led by Ruslan Bukin (br@). In 2016, it was officially imported into the FreeBSD source tree. Currently, FreeBSD's RISC-V support is classified as Tier-3, which means that it is still considered to be under development. As a result, no guarantees are made about feature availability, ABI stability, or support from the security, ports, or release engineering teams. This is not to say that the RISC-V port is unusable; on the contrary, the last few years have seen slow but steady improvements, and the majority of the base system is fully functional.

In particular, the RISC-V hardware ecosystem is still young, and Unix-capable, RISC-V SoCs are not readily available. FreeBSD supports SiFive's HiFive Unleashed, one of the few boards of this type on the market today, but its high price and lack of availability make it an impractical choice for most consumers. For now, much of the development and testing of FreeBSD/RISC-V is done using simulators such as QEMU or Spike. As RISC-V matures and its adoption increases, there will be opportunities for FreeBSD's support for it to improve as well.

Building a FreeBSD/RISC-V Image

All right, that's enough information for now, so let's get to the fun part. We will build a 64-bit RISC-V system from source. This can be done from an amd64 host running any supported version of FreeBSD.

First, we must obtain a RISC-V toolchain. This includes the cross-compiler, linker, and other utilities required to build the FreeBSD operating system from source. We will be using the GNU toolchain since it has the most mature RISC-V support at the moment. You can install the RISC-V GNU toolchain using `pkg(8)`:

```
pkg install riscv64-xtoolchain-gcc
```

This will install the `devel/riscv64-binutils` and `devel/riscv64-gcc` packages. This preconfigured toolchain should be everything you need to cross-build FreeBSD. All of FreeBSD/RISC-V's sources are in HEAD, so, using your favorite version control utility, grab a copy of FreeBSD's sources and start building:

```
git checkout https://github.com/freebsd/freebsd.git freebsd-riscv
cd freebsd-riscv

# First, build the userland libraries and utilities
make -j4 CROSS_TOOLCHAIN=riscv64-gcc TARGET=riscv buildworld
```

```
# Next, the kernel. We're building the QEMU kernel config.
make -j4 CROSS_TOOLCHAIN=riscv64-gcc TARGET=riscv KERNCONF=QEMU buildkernel
```

Optionally, you can edit `sys/riscv/conf/QEMU` before compiling and set `ROOTDEVNAME=/dev/vtbd0p1` line to point to the correct root filesystem for this setup.

Note: for those who are a little adventurous, FreeBSD's in-tree version of clang and lld should have the necessary support to build FreeBSD/RISC-V, if you're running CURRENT. Try it out with `make -j4 TARGET=riscv buildworld`, *but your mileage may vary!*

Wait a little time (or a long time) until compilation has finished. If all went well, then you can move on to the next step. We want to install the newly built FreeBSD/RISC-V root filesystem to some user-accessible directory so we can later generate an image file. You can specify the directory you want with the `DESTDIR` make variable.

```
# NO_ROOT allows us to install files as a regular user.
make TARGET=riscv -DNO_ROOT DESTDIR=$HOME/riscv-root installworld
make TARGET=riscv -DNO_ROOT DESTDIR=$HOME/riscv-root distribution
make TARGET=riscv -DNO_ROOT DESTDIR=$HOME/riscv-root installkernel
```

Now that we've installed all the necessary files, we want to create a disk image that can be read by QEMU. We will first generate a ufs root filesystem using `makefs(8)`, and then create the image including a swap partition using `mkimg(1)`.

Change to the root filesystem directory we just populated. You can use the following script to generate the image.

```
#!/bin/sh

# Create /etc/rc.conf and append to METALOG
echo 'hostname="qemu"' > etc/rc.conf
s=$(($(cat etc/rc.conf | wc -c)))
echo "./etc/rc.conf type=file uname=root gname=wheel mode=0644 size=$s" >> METALOG

# Create /etc/fstab and append to METALOG
echo "/dev/vtbd0p1 / ufs rw,noatime 1 1" > etc/fstab
echo "/dev/vtbd0p2 / swapsw  0 0" >> etc/fstab
s=$(($(cat etc/fstab | wc -c)))
echo "./etc/fstab type=file uname=root gname=wheel mode=0644 size=$s" >> METALOG

# Create FreeBSD ufs root partition
makefs -D -B little \
    -o label=freebsd_root \
    -o version=2 \
    -s 20g -f 65% \
    riscvroot.ufs METALOG

# Create the final .img
mkimg -s gpt -p freebsd-ufs:=riscvroot.ufs -p freebsd-swap::4G -o riscvroot.img
```

Booting FreeBSD

Now that we have built our own FreeBSD image, it's time to test it out. As mentioned, we will be using QEMU. Please install the `emulators/qemu-devel` package from ports. Note that the regular `emulators/qemu` is enough to run FreeBSD, but there have been many improvements to the RISC-V QEMU platforms, so `emulators/qemu-devel` is recommended.

You will also need to install OpenSBI via `sysutils/opensbi`. OpenSBI will act as the bootloader, and it provides low-level firmware functions required by the FreeBSD kernel via its Supervisor Binary Interface (SBI).

With OpenSBI installed, you can now boot FreeBSD using the following command:

```
qemu-system-riscv64 -machine virt -smp 2 -m 2G -nographic \
     -kernel $HOME/riscv-root/boot/kernel/kernel \
     -bios /usr/local/share/opensbi/platform/qemu/virt/firmware/fw_jump.elf \
     -drive file=/path/to/riscvroot.img,format=raw,id=hd0 \
     -device virtio-blk-device,drive=hd0
```

If everything went well, you should see FreeBSD begin to start up. If you skipped the optional step earlier, the system will fail to mount the root filesystem. When that happens, simply enter `ufs:/dev/vtbd0p1` at the prompt.

You might be surprised not to see the familiar `loader(8)` prompt. This is because `loader(8)` has not yet been ported to RISC-V, and so the kernel boots directly from OpenSBI. For now, you won't have the benefit of loader tweaks or tunables, but this will certainly be available in the future.

You should see the system boot to the login prompt. Log in as root, and change the root password with `passwd(1)`.

Networking

Most systems aren't all that useful without a network connection, so let's power-off and fix that. Add the following arguments to the commandline when launching QEMU:

```
-netdev user,hostfwd=tcp::10000-:22,id=net0 -device virtio-net-device,netdev=net0
```

As you can see, we are forwarding TCP port 10000 from the host machine to port 22 of the guest. Port 22 is the default port used by `ssh(1)`. Before we can connect, we must enable `sshd(8)` on the guest by appending the following to `/etc/rc.conf`:

```
sshd_enable="YES"
```

Now, start the `sshd(8)` service with:

```
service sshd start
```

You should now be able to log in to your QEMU guest; just provide the proper port.

```
ssh -p 10000 mhorne@localhost
```

If you have a `tap(4)` device bridged to a real network card, then you can use that too. Append the following arguments to QEMU instead:

```
-netdev tap,ifname=tap0,script=no,id=net1 -device virtio-net-device,netdev=net1
```

A `tap(4)` interface will allow your guest to appear to the rest of your network as any other host would. Connect using the IP address as it appears in the output of `ifconfig(8)` on the guest:

```
ssh mhorne@10.0.1.25
```

Updating Your System

Let's say some time has gone by, and you want to update your FreeBSD/RISC-V system to take advantage of some new fix or feature that just landed in HEAD. For a typical FreeBSD machine, you might do a self-hosted build, i.e., build and install using the same machine you are updating. Unfortunately, since QEMU is being used to run FreeBSD/RISC-V, we're at the mercy of the emulator, and that means that any self-hosted build will be SLOW.

You could, of course, follow the steps from the previous sections after updating the source tree, and you would end up with a fresh, new FreeBSD/RISC-V system; but, any configuration you've done would be lost. Let's look at how you might update the system within an existing root image file.

First, we will perform the same steps as before to cross-compile an updated version of FreeBSD, targeted for riscv64.

```
# Update the source tree, assuming git
cd freebsd-riscv
git pull

# Now, rebuild and install
make -j4 CROSS_TOOLCHAIN=riscv64-gcc TARGET=riscv buildworld
make -j4 CROSS_TOOLCHAIN=riscv64-gcc TARGET=riscv KERNCONF=QEMU buildkernel
```

Now the updated system is ready to be installed. However, instead of installing to some destination directory on the host, this time we want to update the contents of the image file that we generated earlier. We can make use of `mdconfig(8)`, which will allow us to create and mount our image file as a memory disk.

Note: This, and the steps that follow, require superuser privilege. To prevent concurrent access to the filesystem, please ensure you have shut down any instance of QEMU that is using the image file before continuing.

```
# Create the memory device
mdconfig -a -f /path/to/riscvroot.img
```

The name of the new memory device will be output to the console. Remember that the image we generated contains two partitions: a ufs root partition and a swap partition. We want to mount the ufs partition so that we can install the updated system there.

```
# For memory device /dev/md0
mount /dev/md0p1 /mnt

# Install the system to the mounted partition
make TARGET=riscv DESTDIR=/mnt installworld
make TARGET=riscv DESTDIR=/mnt installkernel

# Optionally delete stale files
make TARGET=riscv DESTDIR=/mnt delete-old
make TARGET=riscv DESTDIR=/mnt delete-old-libs

# Now, unmount
umount /dev/md0p1
```

And you're done! You can boot up the freshly updated system using the same image file you created before. One thing to note is that the kernel specified on the QEMU commandline is the one that will be booted, rather than the one installed to the image filesystem. Therefore, you might need to search for it in the `/usr/obj` directory or do an `installkernel` to a local folder on the host.

Conclusion

The intent of this article is to introduce the RISC-V CPU architecture and to allow easy experimentation with FreeBSD/RISC-V. Hopefully you now feel confident in your ability to build, install, and run your own FreeBSD/RISC-V system. Interested users are encouraged to continue tracking changes and updating their system, now that they have it set up. For further reading, check out the RISC-V page on the FreeBSD wiki (https://wiki.freebsd.org/riscv), and subscribe to the freebsd-riscv mailing list (https://lists.freebsd.org/mailman/listinfo/freebsd-riscv).

Certainly, it is still early days for RISC-V as a whole. The lack of availability on the hardware side and incomplete support on the software side mean that RISC-V's usefulness is, at present, mainly limited to research and specialized computing work. The many improvements to software support and increased adoption of RISC-V over the last few years signify that this will not always be the case and that it may soon emerge as a more accessible platform for general computing. As RISC-V continues to grow,

---

MITCHELL HORNE is a university student currently finishing his undergrad at the University of Waterloo, in Canada. He is a FreeBSD src committer and over the last year has become one of the main contributors to the FreeBSD/RISC-V port.