Structure of the RISC-V Software Stack

Sagar Karandikar
UC Berkeley
skarandikar@berkeley.edu
Outline

- Targets, Compilers, Kernels, Usage Patterns
- Overview of riscv-tools
- (Cross) Compiling a C Program for RISC-V
- Running Programs with Spike + Proxy Kernel
- Building RISC-V Linux
- Running Programs with QEMU + Linux
- Other RISC-V Software Tools
Meta: Following Along

- No time to wait for many of the compiles
  - This material will only be in the slides
- Anything I type into a console, you should be able to follow along with in real-time
- Lab time from 3pm-5pm in case you get stuck
RISC-V Software Target Machines

- **ANGEL**
  - Goal: Ease-of-use, outreach, education

- **Spike**
  - Goal: Match ISA specification, “golden standard”

- **QEMU**
  - Goal: High-speed JIT-ing simulation
RISC-V Compilers

- **gcc**
  - 4.6 - old ABI
    - old names, like riscv-gcc
  - 4.9 - new ABI
    - follows standard triplet: cpu-vendor-os-gnu-gcc

- **llvm**
  - 3.3 - new ABI
  - trunk - new ABI

- **Standard Libraries:**
  - newlib
    - use with proxy kernel
  - glibc
    - use with riscv-linux

- Today, we’ll use gcc + newlib and gcc + glibc
- See http://github.com/ucb-bar/riscv-llvm for more about llvm
RISC-V Kernels

- **SEE**: Supervisor Execution Environment
- **AEE**: Application Execution Environment
- **SBI**: System Binary Interface
- **ABI**: Application Binary Interface
- Proxy Kernel with fesvr
  - ABI only
- **Linux**
  - ABI + SBI
Common Usage Patterns

- Small programs/embedded systems:
  - Compile: riscv64-unknown-elf-gcc (newlib)
  - Kernel: Proxy Kernel
  - Target: Spike

- Programs that require a full-blown OS:
  - Compile: riscv64-unknown-linux-gnu-gcc (glibc)
  - Kernel: Linux
  - Target: Spike or QEMU
Stack Overview - “riscv-tools”

- Repo with submodules for every release-ready component in the RISC-V Software Toolchain
- Available on GitHub: [http://github.com/ucb-bar/riscv-tools](http://github.com/ucb-bar/riscv-tools)
- Build status monitored with Travis CI
- Pre-installed on your VM
- Install instructions for Ubuntu in README.md
- Install on OS X with homebrew:
  
  $ brew tap ucb-bar/riscv
  
  $ brew install riscv-tools
The RISC-V Software Toolchain

- riscv-fesvr – The RISC-V Frontend Server
- riscv-gnu-toolchain
- riscv-isa-sim – “Spike”, the golden reference ISA Simulator
- riscv-llvm
- riscv-pk – “Proxy Kernel”, a lightweight kernel
- riscv-qemu – QEMU Port – “Fast” ISA Simulation
- riscv-tests
- (riscv-linux)
riscv-fesvr – The Frontend Server

- Facilitates communication between a host machine and a RISC-V target
- ELF loading, peripheral device emulation over HTIF
- Host Target Interface (HTIF)
  - Communication bus for test hardware
- Used on FPGA development boards (fesvr-zynq), test chips, and Spike
- Where to begin:
  - Software: see spike
  - FPGA Boards: $ fesvr-zynq -h
riscv-pk – Proxy Kernel

- Lightweight kernel, proxies syscalls to host over HTIF
- Together with fesvr, provides an ABI + AEE (application execution environment)
- Binary on your VM in ~/riscv-tools-4.9/riscv64-unknown-elf/bin/pk

Where to begin:

$ spike pk BINARY
Host-Target Interface (HTIF)

- Non-Standard Berkeley Extension: *Not* part of the ISA, but defined in riscv-fesvr
- Non-blocking FIFO Interface, communicates non-zero values
- In RV64 implementations, two 64-bit communication registers in CSR space:
  - *fromhost* - Host writes, RISC-V target reads
  - *tohost* - RISC-V target writes, Host reads
- Existing driver/device implementations:
  - Console
  - Block-devices
  - Networking
HTIF in Action – Proxy Kernel

```c
long frontend_syscall(long n, long a0, long a1, long a2, long a3, long a4)
{
    static volatile uint64_t magic_mem[8];
    [...] // spinlock

    magic_mem[0] = n; magic_mem[1] = a0; magic_mem[2] = a1;

    mb(); // fence
    write csr(tohost, magic_mem);
    while (swap csr(fromhost, 0) == 0);
    mb(); // fence

    long ret = magic_mem[0];
    [...] // spinlock release
    return ret;
}
```

From riscv-pk/pk/frontend.c
void syscall_t::dispatch(reg_t mm) {
    reg_t magicmem[8];
    memIf->read(mm, sizeof(magicmem), magicmem);

    reg_t n = magicmem[0];
    if (n >= table.size() || !table[n])
        […] //error

    magicmem[0] = (this->*table[n])(magicmem[1],
                               magicmem[2], magicmem[3],
                               magicmem[4], magicmem[5]);

    memIf->write(mm, sizeof(magicmem), magicmem);
}

From riscv-fesvr/fesvr/syscall.cc
riscv-isa-sim – “Spike”

- Golden Reference Functional ISA Simulator
- Full system emulation or proxied emulation with HTIF
- Supports single-stepping, viewing register/memory contents (-d)
- Supports a variable amount of memory and number of CPUs
- Can provide SEE or AEE
- Where to begin:
  
  $ spike -h
riscv-qemu – QEMU RISC-V Target

- Full-system RV64 Simulation
  - Does not work with pk (provides only SEE)
  - Use with some OS, e.g. riscv-linux
- Currently the fastest RISC-V Implementation
- Device Support:
  - 8250 UART (for console)
  - Virtio Disk/Networking
  - HTIF Disk
- Used extensively in OpenEmbedded Port
- Where to begin:
  $ qemu-system-riscv --help
Installing the Toolchain

$ git clone https://github.com/ucb-bar/riscv-tools
$ cd riscv-tools
$ git submodule update --init --recursive
$ export RISCV=/path/to/install
$ export PATH=$PATH:$RISCV/bin
$ ./build.sh

This takes a while, so riscv-tools is preinstalled on your VM
Now in $RISCV/bin$

skarandikar@a8:$RISCV/bin$  ls
elf2hex                      riscv64-unknown-elf-gcc-ar
fesvr-eth                   riscv64-unknown-elf-gcc-nm
fesvr-rs232                 riscv64-unknown-elf-gcc-ranlib
fesvr-zedboard              riscv64-unknown-elf-gcov
qemu-ga                     riscv64-unknown-elf-gprof
qemu-img                    riscv64-unknown-elf-ld
qemu-io                     riscv64-unknown-elf-ld.bfd
qemu-nbd                    riscv64-unknown-elf-nm
qemu-system-riscv           riscv64-unknown-elf-objcopy
riscv64-unknown-elf-addr2line riscv64-unknown-elf-objdump
riscv64-unknown-elf-ar      riscv64-unknown-elf-ranlib
riscv64-unknown-elf-as      riscv64-unknown-elf-readelf
riscv64-unknown-elf-c++     riscv64-unknown-elf-size
riscv64-unknown-elf-c++filt riscv64-unknown-elf-strings
riscv64-unknown-elf-cpp     riscv64-unknown-elf-strip
riscv64-unknown-elf-elfedit spike
riscv64-unknown-elf-g++     spike-dasm
riscv64-unknown-elf-gcc     termios-xspike
riscv64-unknown-elf-gcc-4.9.2 xspike
2 Compilers, 2 ways to run:
- `riscv64-unknown-elf-gcc`: produces code for use with pk (newlib)
- `riscv64-unknown-linux-gnu-gcc`: produces code for use with the Linux kernel

Proxy Kernel (binary is `$RISCV/riscv64-unknown-elf/bin/pk$`):

```bash
$ riscv64-unknown-elf-gcc -o hello hello.c
$ spike pk hello
Hello World!
```
Building riscv-linux

- Assumes you have riscv-tools installed
- Overview:
  A. Build `riscv64-unknown-linux-gnu-gcc` *(links against glibc)*
  B. Build the Linux Kernel
  C. Build Dynamically-Linked BusyBox
  D. Create Root Disk Image
- Rest of the tutorial supposes we have a directory `$TOP`, with riscv-tools cloned into it
- No need to follow along here, we’ve included pre-built copies in your VM - this is just FYI
The SYSROOT concept:
- Need to populate directory with C standard library and header files
- Shared libraries will be placed here, will need to copy to disk image
- \texttt{riscv64-unknown-linux-gnu-*} build process will populate it for us
- Located in $\text{RISCV}/\text{sysroot64}$
  - Shared libraries in $\text{RISCV}/\text{sysroot64}/\text{lib}$
Enter the riscv-gnu-toolchain directory:

$ cd $TOP/riscv-tools/riscv-gnu-toolchain
Building riscv-linux-gcc (3/3)

- Configure and compile
  
  $ ./configure --prefix=$RISCV
  $ make linux

- Installs riscv64-unknown-linux-gnu-gcc (and friends) in $RISCV/bin

- Populates $RISCV/sysroot64

- Will already be on your PATH in the VM
Added to $RISCV/bin

skarandikar@a8:$RISCV/bin$  ls
riscv64-unknown-linux-gnu-addr2line
riscv64-unknown-linux-gnu-ar
riscv64-unknown-linux-gnu-as
riscv64-unknown-linux-gnu-c++
riscv64-unknown-linux-gnu-c++filt
riscv64-unknown-linux-gnu-cpp
riscv64-unknown-linux-gnu-elfedit
riscv64-unknown-linux-gnu-g++
riscv64-unknown-linux-gnu-gcc
riscv64-unknown-linux-gnu-gcc-4.9.2
riscv64-unknown-linux-gnu-gcc-ar
riscv64-unknown-linux-gnu-gcc-nm
riscv64-unknown-linux-gnu-gcc-ranlib
riscv64-unknown-linux-gnu-gcov
riscv64-unknown-linux-gnu-gfortran
riscv64-unknown-linux-gnu-gprof
riscv64-unknown-linux-gnu-ld
riscv64-unknown-linux-gnu-ld.bfd
riscv64-unknown-linux-gnu-nm
riscv64-unknown-linux-gnu-objcopy
riscv64-unknown-linux-gnu-objdump
riscv64-unknown-linux-gnu-ranlib
riscv64-unknown-linux-gnu-readelf
riscv64-unknown-linux-gnu-size
riscv64-unknown-linux-gnu-strings
riscv64-unknown-linux-gnu-strip
Added to $RISCV/sysroot64/lib$

skarandikar@a8:$RISCV/sysroot64/lib$  ls
ld-2.20.so               libmemusage.so               libnss_nisplus-2.20.so
ld.so.1                   libm.so.6                   libnss_nisplus.so.2
libanl-2.20.so            libnsl-2.20.so                   libnss_nis.so.2
libanl.so.1               libnsl.so.1                   libnss_nis.so.2
libBrokenLocale-2.20.so   libnss_compat-2.20.so             libpcpProfile.so
libBrokenLocale.so.1      libnss_compat.so.2                libpthread-2.20.so
libc-2.20.so              libnss_db-2.20.so            libpthread.so.0
libcidn-2.20.so           libnss_dns-2.20.so                  libresolv-2.20.so
libcidn.so.1              libnss_dns.so.2                   libresolv.so.2
libcrypt-2.20.so          libnss_files-2.20.so               librt-2.20.so
libcrypt.so.1             libnss_files.so.2                  librt.so.1
libc.so.6                 libnss_hesiod-2.20.so               libSegFault.so
libdl-2.20.so             libnss_hesiod.so.2                libthread_db-1.0.so
libdl.so.2                libnss_nis-2.20.so                   libthread_db.so.1
libm-2.20.so              libnss_nis-2.20.so                   libutil-2.20.so
libm-2.20.so
Here, we build for QEMU - use master branch to build for spike

Obtain upstream sources:

$ cd $TOP
Overlay RISC-V specific subtree:

```bash
$ cd linux-3.14.15
$ git init
$ git remote add origin git@github.com:ucb-bar/riscv-linux.git
$ git fetch
$ git checkout -f -t origin/qemu-coredump-demo
```
Building the Linux Kernel (3/3)

- Populate/modify `.config`:
  
  $ make ARCH=riscv defconfig

  $ make ARCH=riscv menuconfig

- Build:

  $ make ARCH=riscv
Use BusyBox for `init` and an `ash` prompt

```
$ cd $TOP
$ curl -L http://busybox.net/downloads/busybox-1.21.1.tar.bz2 | tar -xj
$ cd busybox-1.21.1
```
Building BusyBox (2/2)

- Use recommended configuration:
  
  ```
  $ curl -L http://riscv.org/install-guides/busybox-riscv-qemu.config > .config
  $ make menuconfig
  $ make
  ```
Creating a Root Disk Image (1/4)

- Build root image with our sysroot + programs
  
  ```bash
  $ dd if=/dev/zero of=root.bin bs=1M count=64
  $ mkfs.ext2 -F root.bin
  ```

- Mount:

  ```bash
  $ mkdir mnt
  $ sudo mount -o loop root.bin mnt
  ```
Creating a Root Disk Image (2/4)

- Add required directories:
  
  ```
  $ cd mnt
  $ mkdir -p bin etc dev lib proc sbin tmp usr usr/bin usr/lib usr/sbin
  ```

- Add BusyBox:
  
  ```
  $ cp $TOP/busybox-1.21.1/busybox bin
  $ ln -s ..../bin/busybox sbin/init
  ```
Creating a Root Disk Image (3/4)

- **Add initramfs:**
  
  ```bash
  $ curl -L http://riscv.org/install-guides/linux-initramfs-qemu > etc/initramfs
  ``

- **Add shared libraries:**
  
  ```bash
  $ cp $RISCV/sysroot64/lib/* lib/
  ```
- Add your program[s], unmount: *
  $ cd ..
  $ riscv64-unknown-linux-gnu-gcc -o hello hello.c
  $ sudo cp hello hello mnt/hello
  $ sudo umount mnt
Using QEMU – Method 1

- Devices: 8250 UART Console + HTIF Disk
- Build:
  
  $ ./configure --target-list=riscv-softmmu --prefix=YOUR_LOCATION
  $ make && make install

- Boot:
  
  $ qemu-system-riscv -kernel vmlinux -hda root.bin -nographic
  $ [...] // booting Linux
  riscv# ./YOUR_BINARY
Using QEMU – Method 2

- Devices: 8250 UART Console + Virtio Disk + Virtio Networking
- Build:
  
  ```
  $ ./configure --target-list=riscv-softmmu
  --prefix=YOUR_LOCATION --disable-htif
  $ make && make install
  ```
- Boot:
  
  ```
  $ qemu-system-riscv ...
  ```

See OpenEmbedded talk for more
Other RISC-V Software Tools

- **C++ RTL Simulation:**
  - Chisel can generate C++ code that simulates a hardware design - Rocket, BOOM, etc.
  - Slower than functional implementations (designed to simulate hardware details)

- **ANGEL:**
  - Browser-based ISA Simulator (in JavaScript)
  - Mainly for education/outreach
  - Demo: riscv.org/angel
Questions?

- We’re happy to answer them during lab time from 3-5pm today