Compile Linux on RISC-V processor

Iman Rasouli Parto

University of Tehran

***Abstract***—***This article provides instructions for compiling 32-bit Linux on RISC-V processor and test the program.***

***Keywords***—***config, make, Makefile, RISC-V toolchain, GCC***

Introduction: In embedded systems, the operating system (OS) communicates with the processor, manages resources, executes programs and performs other tasks. Therefore, understanding how an operating system compiles on a processor is essential. To test this process, a RISC-V processor is a suitable choice as it provides the appropriate tools for the task. In this article we are going to compile Linux on Spike simulator for RISC-V processor.

At the beginning, we need to update the system using the command below:

$ sudo apt update && sudo apt upgrade

Before compiling the Linux, we will need some required build tools. The following command will install them:

$ sudo apt-get install device-tree-compiler autoconf automake autotools-dev curl libmpc-dev \ libmpfr-dev libgmp-dev gawk build-essential bison flex texinfo gperf libtool patchutils bc zlib1g-dev \ libexpat-dev python2-dev python3-dev unzip libglib2.0-dev libpixman-1-dev git rsync wget cpio \ libncurses-dev

# Step 1: Get the simple repository from git

$ git clone <https://github.com/riscv-zju/riscv-rss-sdk.git>

Then, go to the directory and run quickstart:

$ cd riskv-rss-sdk/

$ sh quickstart.sh

After that, fetch all the system dependencies for the code:

$ git submodule update --init --recursive --progress

# Step 2: Set the RISC-V director as a variable and add it to PATH

$ export RISCV=/home/(your username)/riscv-rss-sdk

$ export PATH=$PATH:$RISCV/bin

Note: Replace “your\_username” with your actual username. Be sure to use your own username in place of “your\_username” to avoid any confusion.

Variables will be lost after a reboot. To prevent them from being lost after a reboot, open the “bashrc” file using a text editor and add these two commands to it. Make sure you are in your home directory.

$ gedit .bashrc

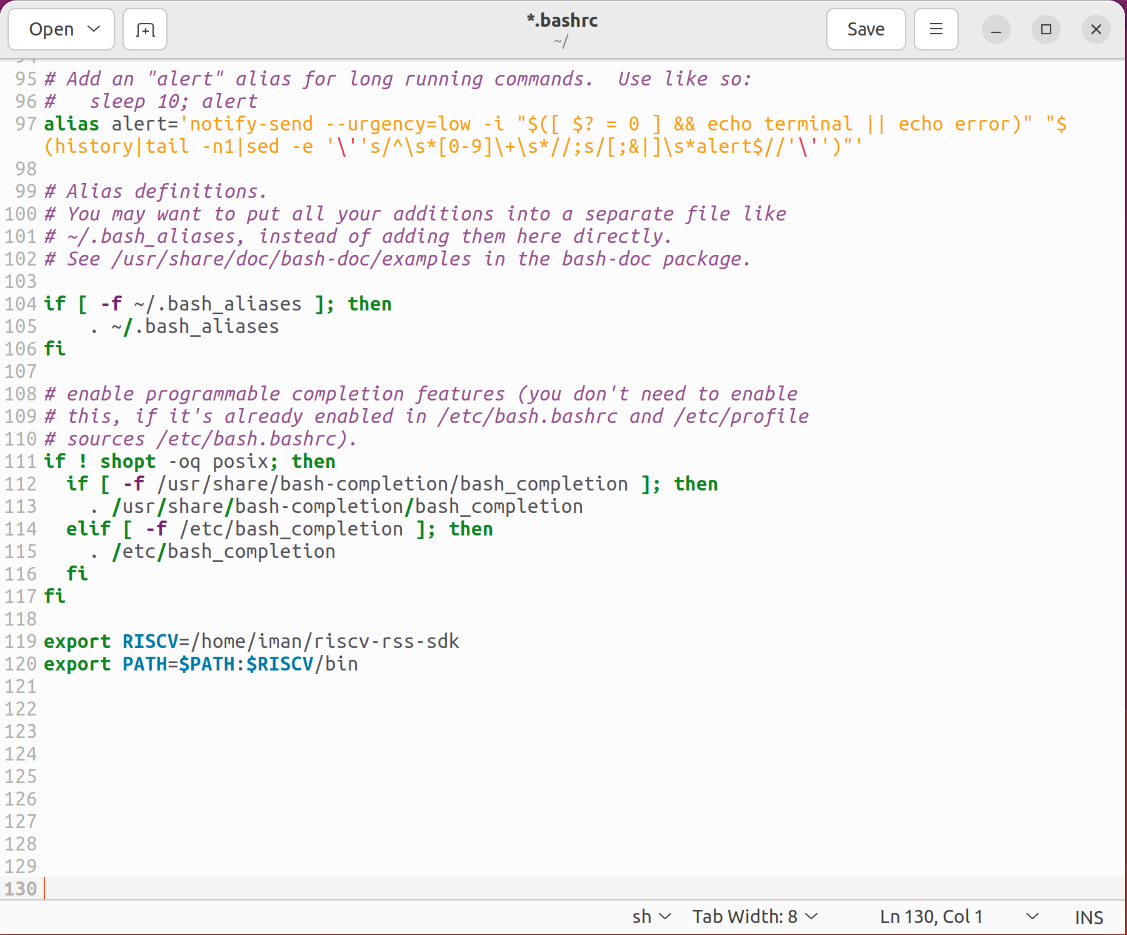


Figure 1. Add variables in bashrc file

Finally, save the file and exit.

# Step 3: Cross compile toolchain

$ cd $RISCV/repo/riscv-gnu-toolchain/

$ mkdir build && cd build

$ ../configure --prefix=$RISCV --with-arch=rv32imafdc --with-abi=ilp32d

$ make

The process will take some time.

Note: **Do not** execute any command with root access (don’t use “sudo”). It could fail the process.

Once the compilation is complete, the GCC compiler will be installed. You can find GCC compiler and its libraries in the “RISCV” and “RISCV/bin” directories. Commands that start with “riscv32-unknown-elf-” are used for cross-compile for the RISC-V processor. In addition, we need to cross compile GCC for Linux kernel with the command below:

$ make linux

After compilation, the compiler for Linux will be appear in “RISCV/bin” directory with start in name “riscv32-unknown-linux-gnu-”. Now you have all tools for compile all we need for simulations.

# Step 4: Build the Proxy kernel

Execute following instructions to build the Proxy kernel:

$ cd $RISCV/repo/riscv-pk

$ mkdir build && cd build

$ ../configure --prefix=$RISCV --host=riscv32-unknown-elf

$ make

$ make install

# Step 5: Build Spike simulator

Execute following commands to compile Spike:

$ cd $RISCV/repo/riscv-pk

$ mkdir build && cd build

$ ../configure --prefix=$RISCV --enable-histogram

$ make

$ make install

# Step 6: Verify the simulator

To verify the compilation, run this simple code with the commands:



$ riscv32-unknown-elf-gcc test.c

$ $RISCV/repo/riscv-isa-sim/build/spike --isa=RV32I $RISCV/repo/riscv-pk/build/pk a.out

If “Hello world!” appeared in the terminal it’s mean you have everything for compilation and simulations for linux kernel.

# Step 7: Configure the Linux to compile

We should configure some features in config menu to compile Linux correctly. With command below open the config menu:

$ cd $RISCV

$ make buildroot\_initramfs-menuconfig

A menu will appear in terminal that needs to change a little bit.

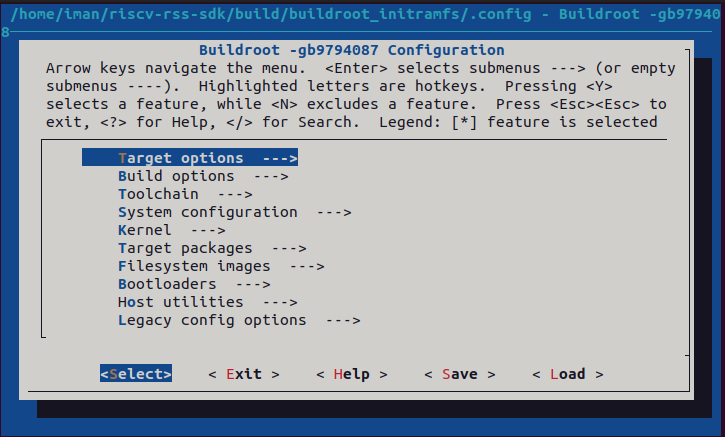


Figure 2. Config menu

In “Target options” section, enter “Target ABI” and select the “lp32” version. In “Toolchain” section, enter “Toolchain type” and select “Buildroot toolchain”. Next, enter “C library” and choose “glibc” (stay in “Toolchain” tab). Again in “Toolchain” section, enter “Kernel Headers” and choose “5.15.x” version. Then go to “System configuration” section, enable “Enable root login with password” and set a password in “Root password”. Exit the menu and save the changes, finally compile the Linux on simulator with the following command:

$ make

The compilation takes a huge time, after completed, you’ll see a Linux terminal (It’s not your system terminal). Login with root and the password you set in configuration.

# Step 8: Test the compiled Linux

To test the system, create a directory and use the “ls” command to view the directory. If the directory is created successfully, it indicates that the system is functioning properly.

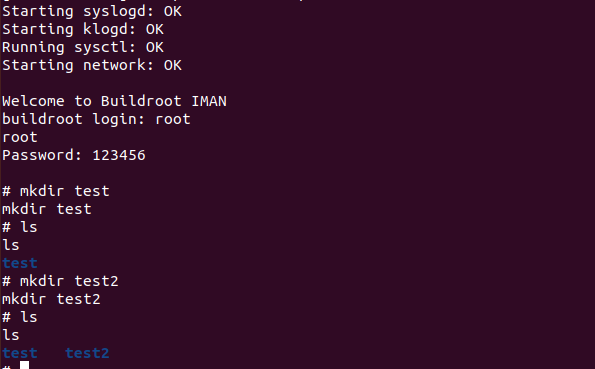


Figure 3. Test compiled Linux on simulator

# Appendix

* **Buildroot**: Buildroot is a set of [Makefiles](https://en.wikipedia.org/wiki/Makefile" \o "Makefile) and [patches](https://en.wikipedia.org/wiki/Patch_(Unix)) that simplifies and automates the process of building a complete and bootable [Linux](https://en.wikipedia.org/wiki/Linux) environment for an [embedded system](https://en.wikipedia.org/wiki/Embedded_system), while using [cross-compilation](https://en.wikipedia.org/wiki/Cross_compiler) to allow building for multiple target platforms on a single Linux-based development system. Buildroot can automatically build the required cross-compilation [toolchain](https://en.wikipedia.org/wiki/Toolchain), create a [root file system](https://en.wikipedia.org/wiki/Root_file_system), compile a [Linux kernel](https://en.wikipedia.org/wiki/Linux_kernel) image, and generate a [boot loader](https://en.wikipedia.org/wiki/Boot_loader) for the targeted embedded system, or it can perform any independent combination of these steps. For example, an already installed cross-compilation toolchain can be used independently, while Buildroot only creates the root file system.

Buildroot is primarily intended to be used with small or embedded systems based on various [computer architectures](https://en.wikipedia.org/wiki/Computer_architectures) and [instruction set architectures](https://en.wikipedia.org/wiki/Instruction_set_architecture) (ISAs), including [x86](https://en.wikipedia.org/wiki/X86), [ARM](https://en.wikipedia.org/wiki/ARM_architecture), [MIPS](https://en.wikipedia.org/wiki/MIPS_architecture) and [PowerPC](https://en.wikipedia.org/wiki/PowerPC).

Multiple [C standard libraries](https://en.wikipedia.org/wiki/C_standard_library) are supported as part of the toolchain, including the [GNU C Library](https://en.wikipedia.org/wiki/GNU_C_Library), [uClibc](https://en.wikipedia.org/wiki/UClibc" \o "UClibc) and [musl](https://en.wikipedia.org/wiki/Musl" \o "Musl), as well as the C standard libraries that belong to various preconfigured development environments, such as those provided by [Linaro](https://en.wikipedia.org/wiki/Linaro" \o "Linaro). Buildroot's build configuration system internally uses [Kconfig](https://en.wikipedia.org/wiki/Kconfig" \o "Kconfig), which provides features such as a menu-driven interface, handling of [dependencies](https://en.wikipedia.org/wiki/Dependency_(computer_science)), and [contextual help](https://en.wikipedia.org/wiki/Contextual_help); Kconfig is also used by the Linux kernel for its source-level configuration.

* **Busybox**: BusyBox is a [software suite](https://en.wikipedia.org/wiki/Software_suite) that provides several [Unix utilities](https://en.wikipedia.org/wiki/List_of_Unix_commands) in a single [executable file](https://en.wikipedia.org/wiki/Executable_file). It runs in a variety of [POSIX](https://en.wikipedia.org/wiki/POSIX) environments such as [Linux](https://en.wikipedia.org/wiki/Linux), [Android](https://en.wikipedia.org/wiki/Android_(operating_system)), and [FreeBSD](https://en.wikipedia.org/wiki/FreeBSD), although many of the tools it provides are designed to work with interfaces provided by the [Linux kernel](https://en.wikipedia.org/wiki/Linux_kernel). It was specifically created for embedded operating systems with very limited resources.

# References

[1] [https://en.wikipedia.org/wiki/Buildroot](https://en.wikipedia.org/wiki/Buildroot%23:~:text=Buildroot%20can%20automatically%20build%20the,independent%20combination%20of%20these%20steps.)

[2] [https://en.wikipedia.org/wiki/BusyBox](https://en.wikipedia.org/wiki/BusyBox#:~:text=Article%20Talk,provided%20by%20the%20Linux%20kernel.)