RISC-V - Getting Started Guide

RISC-V Foundation

2020-08-12

Introduction

CHAPTER 1

Introduction

Additional tips

Look out for these floating sidebar boxes, they can provide additional information related to the topic at hand. While not strictly necessary to complete the guide, they may give some background for the instructions provided.

This Getting Started Guide will explain how to get started with developing for the free and open RISC-V ISA (Instruction Set Architecture), both in simulation and on physical implementations.

The Guide focuses on running standard operating systems - *Zephyr* and *Linux* - on popular RISC-V platforms with minimum effort.

It will be expanded with time to cover more platforms and scenarios.

1.1 About RISC-V

RISC-V (pronounced "risk-five") is an open, free ISA enabling a new era of processor innovation through open standard collaboration. It's both academia- and industry friendly, open to scrutiny, built from scratch with security and modern use cases in mind. The standard is driven by a Foundation with more than 130 members, including Google, Western Digital, NVIDIA, NXP and many other industry leaders.

For details about the ISA and the Foundation, see the RISC-V website.

1.2 Contributing

The source code for this Guide can be found on the RISC-V Foundation's GitHub - while its compiled version is automatically generated using Read the Docs.

We encourage you to contribute - see the project's README for details.

CHAPTER 2

Zephyr

The Zephyr OS is a popular security-oriented RTOS with a small-footprint kernel designed for use on resource-constrained and embedded systems.

It is fully open source, highly configurable and modular, making it perfect for developers building everything from simple embedded environmental sensors and LED wearables to sophisticated embedded controllers, smart watches, and IoT wireless applications.

The Zephyr OS is managed by the vendor neutral Zephyr Project which is part of the Linux Foundation.

Zephyr-enabled platforms currently described in the Getting Started Guide include:

- *SiFive HiFive1*
- *LiteX SoC with VexRiscv CPU* running on the Future Electronics Avalanche board with a Microsemi PolarFire FPGA or in the Renode simulation framework

And the winner is...

The VexRiscv CPU, which is also capable of running Linux in FPGA, is the winner of the first edition of the RISC-V Soft CPU contest due to its very effective implementation in FPGA using the author's own Scala-based HDL generator language, SpinalHDL. The LiteX soft SoC, developed in MiGen/Python that VexRiscv can be - and often is - combined with scales from simple designs with UART or SPI, I2C to complex setups with Ethernet, USB, PCIe, DDR controllers etc. Those two projects illustrate the active, software-driven community around RISC-V.

There is also a generic *QEMU simulation target* supporting RISC-V.

For a full list of supported boards and details, see the Zephyr documentation.

2.1 Why Zephyr?

Zephyr is a generic, open source, cross-platform and vendor-independent RTOS, with a well-constructed governance structure and extensive and active community. Just like RISC-V, it has security and flexibility in mind, and comes with 'batteries included', enabling a wide array of applications thanks to a number of different configurations available.

Zephyr follows standard coding guidelines, best practices - shared with the wider Linux ecosystem and based on lessons learned from over 25 years of OS development - and a community approach for which it is especially praised.

CHAPTER 3

Getting Zephyr

Note that a complete Getting Started Guide including installation instructions for different OSs is available in the Zephyr Project documentation.

Here, we will focus on a Ubuntu Linux based environment for simplicity and clarity.

First, you will need to install the following prerequisities:

```
sudo apt-get install --no-install-recommends git cmake ninja-build gperf \
  ccache dfu-util device-tree-compiler wget python3-pip python3-setuptools \
  python3-wheel xz-utils file make gcc gcc-multilib
```

Then, download the Zephyr source code and install additional dependencies:

```
git clone https://github.com/zephyrproject-rtos/zephyr
cd zephyr
pip3 install --user -r scripts/requirements.txt
```

Set up the environment (do this always for a shell where you will be compiling/running Zephyr):

```
export ZEPHYR_TOOLCHAIN_VARIANT=zephyr
export ZEPHYR_SDK_INSTALL_DIR="/opt/zephyr-sdk/"
. ./zephyr-env.sh
```

Download and install Zephyr SDK (note that you can use a different directory for the SDK installation by changing the shell variable set in the snippet above; the value used here is just a sane default):

```
wget https://github.com/zephyrproject-rtos/sdk-ng/releases/download/v0.10.0/zephyr-sdk-0.
↪10.0-setup.run
sudo sh zephyr-sdk-0.10.0-setup.run -- -d $ZEPHYR_SDK_INSTALL_DIR
```

CHAPTER 4

Running Zephyr on QEMU

4.1 Setting up the environment

Please remember to *get the sources and setup the environment* first.

4.2 Compiling an example

Create a build directory and run following commands:

```
mkdir build-example
cd build-example
cmake -DBOARD=qemu_riscv32 $ZEPHYR_BASE/samples/hello_world
make -j $(nproc)
```

4.3 Running an example

To run an example, simply run:

```
make run
```

You can exit QEMU with `C-a x` key strokes.

CHAPTER 5

Running Zephyr on SiFive HiFive1

SiFive's HiFive1 is an Arduino-Compatible development kit featuring the Freedom E310, the industry's first commercially available RISC-V SoC.

It's a very good starting point if you want to get Zephyr running on a physical chip/board. SiFive provides open source schematics, an Altium Designer PCB project, BOM, and - of course - tooling for the HiFive1.

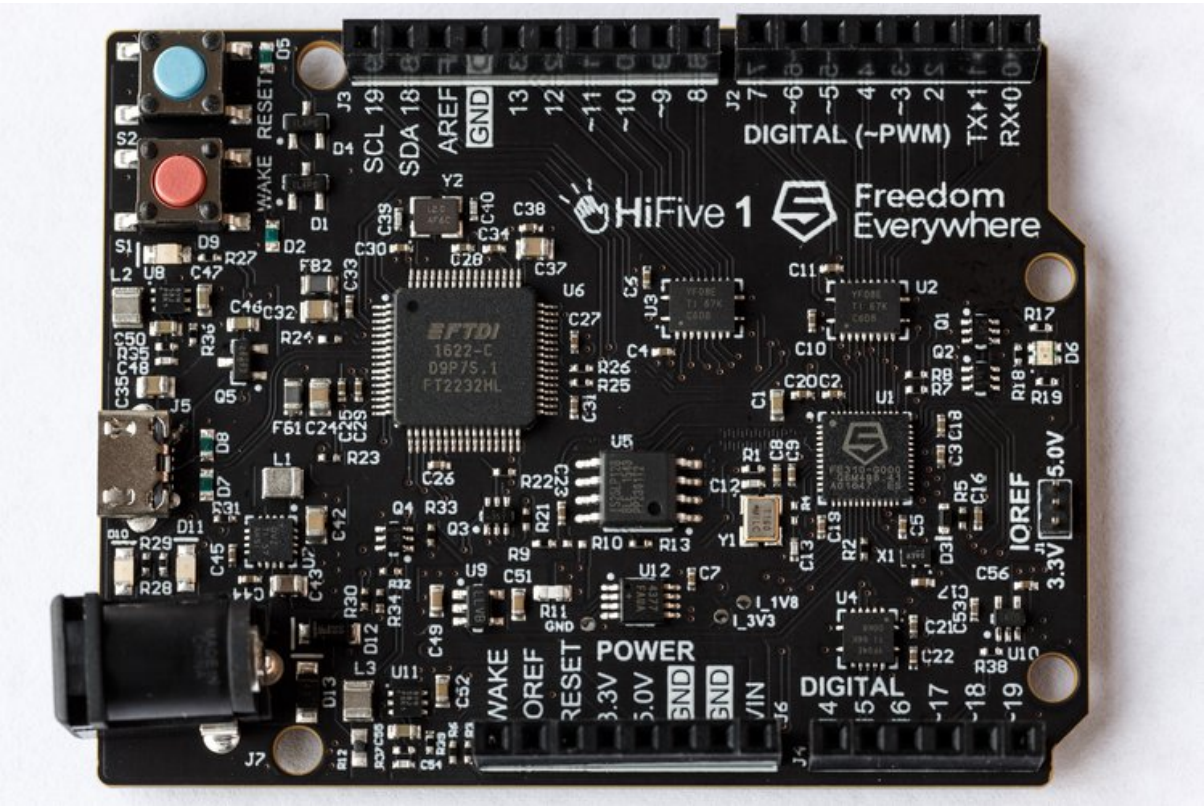

Fig. 5.1: The HiFive1 board - top.

5.1 Setting up the enviroment

Please remember to *get the sources and setup the environment* first.

5.2 Getting tools for HiFive1 board

Download and extract a prebuilt OpenOCD and GDB with RISC-V from SiFive's website:

```
# GDB
wget https://static.dev.sifive.com/dev-tools/riscv64-unknown-elf-gcc-2018.07.0-x86_64-
↪linux-ubuntu14.tar.gz
tar xzfv riscv64-unknown-elf-gcc-2018.07.0-x86_64-linux-ubuntu14.tar.gz

# OpenOCD
wget https://static.dev.sifive.com/dev-tools/riscv-openocd-2018.7.0-x86_64-linux-ubuntu14.
↪tar.gz
tar xzfv riscv-openocd-2018.7.0-x86_64-linux-ubuntu14.tar.gz
```

SiFive provides an open source SDK for their Freedom E platform.

Download Freedom E SDK and move previously downloaded prebuilt tools to their respective directories:

```
git clone https://github.com/sifive/freedom-e-sdk
mv riscv64-unknown-elf-gcc-2018.07.0-x86_64-linux-ubuntu14/* freedom-e-sdk/riscv-gnu-
↪toolchain
mv riscv-openocd-2018.7.0-x86_64-linux-ubuntu14/* freedom-e-sdk/openocd
```

Note: If you wish to build the toolchain yourself, please refer to the instructions on SiFive's GitHub.

5.3 Compiling an example

Create a build directory (we will use `build-example` here) and compile an example binary inside it with the following commands:

```
mkdir build-example
cd build-example
cmake -DBOARD=hifive1 $ZEPHYR_BASE/samples/hello_world
make -j $(nproc)
cd ..
```

5.4 Flashing

Move to your Freedom E SDK directory and connect to the board with OpenOCD:

```
cd freedom-e-sdk
sudo openocd/bin/openocd -f bsp/env/freedom-e300-hifive1/openocd.cfg
```

Leave OpenOCD running and connect to the board with GDB, disable flash protection and load the binary (assuming it's in the `build-example` directory you've created earlier):

```
riscv-gnu-toolchain/bin/riscv64-unknown-elf-gdb
(gdb) set remotetimeout 240
(gdb) target extended-remote localhost:3333
(gdb) monitor reset halt
(gdb) monitor flash protect 0 64 last off
(gdb) load build-example/zephyr/zephyr.elf
(gdb) monitor resume
```

Finally, you can connect with picocom to the serial console:

```
sudo picocom -b 115200 /dev/ttyUSBx  # substitute "x" with appropriate port number
```

After resetting the board, a hello world message should appear. You can quit picocom using the `C-a C-q` key strokes.

CHAPTER 6

Running Zephyr on LiteX/VexRiscv on Avalanche board with Microsemi PolarFire FPGA

This section contains a tutorial on how to build and run a shell sample for the Zephyr RTOS on the LiteX soft SoC with an RV32 VexRiscv CPU on the Future Electronics Avalanche Board with a PolarFire FPGA from Microsemi (a Microchip company) as well as in the Renode open source simulation framework.

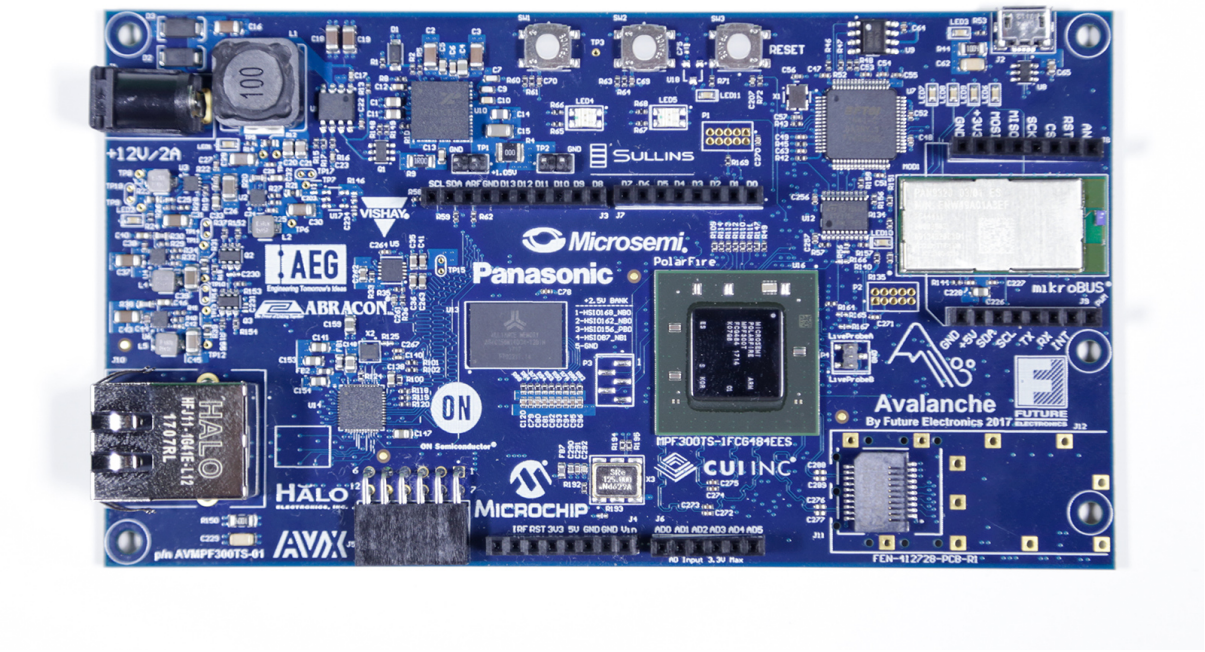

Fig. 6.1: The Future Electronics Avalanche board - top.

6.1 Building your system

6.1.1 Zephyr

First, *prepare your environment and get the Zephyr RTOS sources*.

If you want to skip building Zephyr manually, you can download a precompiled ELF file and binary file.

Building a shell sample

Generate the project build files:

```
cd samples/subsys/shell/shell_module
mkdir build
cd build
cmake -DBOARD=litex_vexriscv ..
```

Build it:

```
make -j $(nproc)
```

As a result, you should find `zephyr.elf` and `zephyr.bin` in the `zephyr` folder.

6.2 Running

6.2.1 Preparing the platform

Hardware

Download a pregenerated bitstream of LiteX with VexRiscv and BIOS preloded to RAM:

```
wget https://github.com/riscv/risc-v-getting-started-guide/releases/download/tip/
↪bitstream-litex-vexriscv-avalanche-zephyr.job
```

Load it onto the Avalanche board using the PolarFire FlashPro tool. You can refer to the "Creating a Job Project from a FlashPro Express Job" section of the tool's official User Guide.

Renode

Note: Support for LiteX is available in Renode since version 1.7 - download pre-built packages from GitHub. Refer to the Renode README for more detailed installation instructions.

Start Renode and create a simulated instance of LiteX+VexRiscv:

```
mach create "litex-vexriscv"
machine LoadPlatformDescription @platforms/cpus/litex_vexriscv.repl
```

6.2.2 Loading Zephyr

Hardware

In this example, Zephyr will be loaded onto the board over a serial connection. Download and run the `litex_term.py` script (shipped with LiteX) on your host computer and connect it to the board via serial:

```
wget https://raw.githubusercontent.com/enjoy-digital/litex/master/litex/tools/litex_term.
↪py
chmod u+x litex_term.py

./litex_term.py --serial-boot --kernel zephyr.bin /dev/ttyUSB1
```

Renode

To load the binary onto the simulated platform, just do:

```
sysbus LoadELF @zephyr.elf
```

Note: LiteX bios plays a role of a bootloader and is required on hardware to run Zephyr.

In Renode, however, you can load an ELF file to RAM and set the CPU PC to its entry point, so there is no need for a bootloader.

6.2.3 Running Zephyr

Hardware

Reset the board.

You should see the following output:

```
[TERM] Starting....

        __   _ __      _  __
       / /  (_) /____ | |/_/
      / /__/ / __/ -_)>  <
     /____/_/\__/\__/_/|_|

 (c) Copyright 2012-2019 Enjoy-Digital
 (c) Copyright 2012-2015 M-Labs Ltd

 BIOS built on Apr  9 2019 14:40:45
 BIOS CRC passed (8c8ddc55)

--============ SoC info ================--
CPU:       VexRiscv @ 100MHz
ROM:       32KB
SRAM:      32KB
L2:        8KB
MAIN-RAM:  262144KB

--========== Peripherals init ===========--
```

(continues on next page)

(continued from previous page)

```
Memtest OK

--========== Boot sequence ==============--
Booting from serial...
Press Q or ESC to abort boot completely.
sL5DdSMmkekro
[TERM] Received firmware download request from the device.
[TERM] Uploading zephyr.bin (57912 bytes)...
[TERM] Upload complete (7.6KB/s).
[TERM] Booting the device.
[TERM] Done.
Executing booted program at 0x40000000




uart:~$
```

Renode

Open a UART window and start the Renode simulation:

```
showAnalyzer sysbus.uart
start
```

As a result, in the UART window you will see the shell prompt:

```
uart:~$
```

Now you can use the UART window to interact with the shell, e.g.:

```
uart:~$ help
Please press the <Tab> button to see all available commands.
You can also use the <Tab> button to prompt or auto-complete all commands or its␣
↪subcommands.
You can try to call commands with <-h> or <--help> parameter for more information.
Shell supports following meta-keys:
Ctrl+a, Ctrl+b, Ctrl+c, Ctrl+d, Ctrl+e, Ctrl+f, Ctrl+k, Ctrl+l, Ctrl+u, Ctrl+w
Alt+b, Alt+f.
Please refer to shell documentation for more details.

uart:~$ kernel version
Zephyr version 1.14.0
```

CHAPTER 7

Linux

Linux and related tools are - for the most part - already in the upstream repositories of the respective projects. As noted on the Debian RISC-V wiki (with some updates):

- binutils: upstreamed (2.28 is the first release with RISC-V support)
- gcc: upstreamed (7.1 is the first release with RISC-V support)
- glibc: upstreamed (2.27 is the first release with RISC-V support)
- linux kernel: upstreamed (the architecture core code went into kernel 4.15; kernel 4.19 contains all drivers necessary for booting a simulated system to userland)
- gdb: upstreamed in master (in the release process)
- qemu: upstreamed (2.12 is the first release with RISC-V support)

Linux-enabled platforms currently described in the Getting Started Guide include:

- *SiFive HiFive Unleashed*
- *LiteX SoC with VexRiscv CPU* running on the Future Electronics Avalanche board with a Microsemi PolarFire FPGA or in the Renode simulation framework

There is also a generic *QEMU 64-bit RISC-V simulation target running Linux*.

Debian, Fedora, and openSUSE ports are also available, for more information see:

- Debian RISC-V wiki
- Fedora RISC-V wiki
- openSUSE RISC-V wiki

CHAPTER 8

Running 64- and 32-bit RISC-V Linux on QEMU

This is a "hello world" example of booting Linux on RISC-V QEMU. This guide covers some basic steps to get Linux running on RISC-V. It is recomended that if you are interested in a specific distrubution you follow their steps. For example if you are interested in running Debian, they have instructions on their wiki https://wiki.debian.org/RISC-V. Most distrobutions (Debian, Fedora, OpenEmbedded, buildroot, OpenSUSE, FreeBSD and others) support RISC-V.

8.1 Prerequisites

Running Linux on QEMU RISC-V requires you to install some prerequisites. Find instructions for various Linux distributions as well as macOS below:

Ubuntu/Debian

Note: This has been tested on Ubuntu 18.04.

```
sudo apt install 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 git
```

Fedora/CentOS/RHEL OS

```
sudo yum install autoconf automake libmpc-devel mpfr-devel gmp-devel gawk bison flex \
                 texinfo patchutils gcc gcc-c++ zlib-devel expat-devel git
```

macOS

```
brew install gawk gnu-sed gmp mpfr libmpc isl zlib expat
```

8.2 Getting the sources

First, create a working directory, where we'll download and build all the sources.

64-bit

```
mkdir riscv64-linux
cd riscv64-linux
```

32-bit

```
mkdir riscv32-linux
cd riscv32-linux
```

Then download all the required sources, which are:

- QEMU
- Linux
- Busybox

```
git clone https://github.com/qemu/qemu
git clone https://github.com/torvalds/linux
git clone https://git.busybox.net/busybox
```

You will also need to install a RISC-V toolchain. It is recomendded to install a toolchain from your distro. This can be done by using your distro's installed (apt, dnf, pacman or something similar) and searching for riscv64 and installing gcc. If that doesn't work you can use a prebuilt toolchain from: https://toolchains.bootlin.com.

Build QEMU with the RISC-V target:

64-bit

```
cd qemu
git checkout v5.0.0
./configure --target-list=riscv64-softmmu
make -j $(nproc)
sudo make install
```

32-bit

```
cd qemu
git checkout v5.0.0
./configure --target-list=riscv32-softmmu
make -j $(nproc)
sudo make install
```

Build Linux for the RISC-V target. First, checkout to a desired version:

64-bit

```
cd linux
git checkout v5.4.0
make ARCH=riscv CROSS_COMPILE=riscv64-unknown-linux-gnu- defconfig
```

32-bit

```
cd linux
git checkout v5.4.0
make ARCH=riscv CROSS_COMPILE=riscv32-unknown-linux-gnu- defconfig
```

Then compile the kernel:

64-bit

```
make ARCH=riscv CROSS_COMPILE=riscv64-unknown-linux-gnu- -j $(nproc)
```

32-bit

```
make ARCH=riscv CROSS_COMPILE=riscv32-unknown-linux-gnu- -j $(nproc)
```

Build Busybox:

```
cd busybox
CROSS_COMPILE=riscv{{bits}}-unknown-linux-gnu- make defconfig
CROSS_COMPILE=riscv{{bits}}-unknown-linux-gnu- make -j $(nproc)
```

8.3 Running

Go back to your main working directory and run:

64-bit

```
sudo qemu-system-riscv64 -nographic -machine virt \
     -kernel linux/arch/riscv/boot/Image -append "root=/dev/vda ro console=ttyS0" \
     -drive file=busybox,format=raw,id=hd0 \
     -device virtio-blk-device,drive=hd0
```

32-bit

```
sudo qemu-system-riscv32 -nographic -machine virt \
     -kernel linux/arch/riscv/boot/Image -append "root=/dev/vda ro console=ttyS0" \
     -drive file=busybox,format=raw,id=hd0 \
     -device virtio-blk-device,drive=hd0
```

CHAPTER 9

Running 64-bit RISC-V Linux on SiFive HiFive Unleashed

SiFive's HiFive Unleashed is a Linux capable development board featuring the Freedom U540, the industry's first commercially available Linux-capable RISC-V SoC.

It's a very good starting point if you want to get Linux running on a physical chip/board. Just like with the HiFive1, SiFive provides open source schematics, an Altium Designer PCB project, BOM, and - of course - tooling for the HiFive Unleashed.

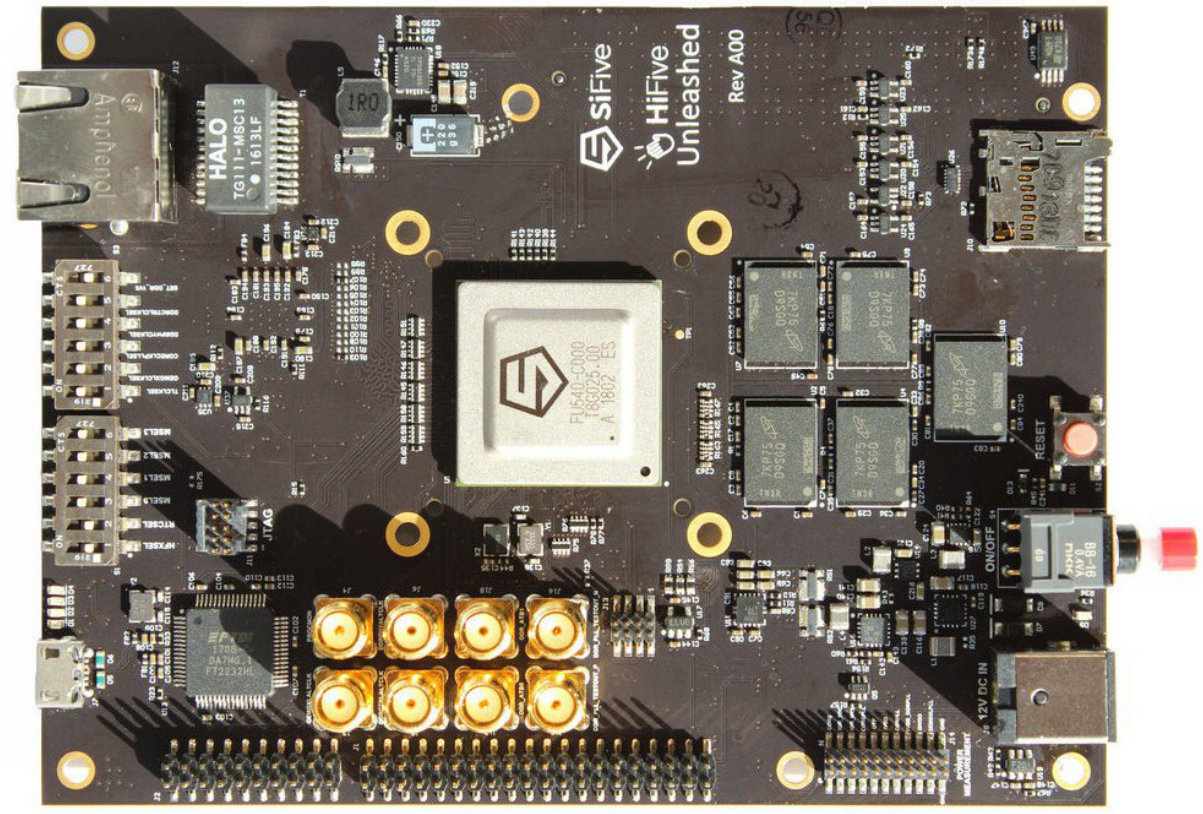

Fig. 9.1: The HiFive Unleashed board - top.

Note: This chapter targets Debian-based Linux flavors, and has been tested on Ubuntu 18.04.

9.1 Prerequisites

Running Linux on the HiFive Unleashed board requires you to install some prerequisites.

```
sudo apt install autoconf automake autotools-dev bc bison build-essential curl flex \
                 gawk gdisk git gperf libgmp-dev libmpc-dev libmpfr-dev libncurses-dev \
                 libssl-dev libtool patchutils python screen texinfo unzip zlib1g-dev
```

9.2 Getting the sources

Clone the official Freedom Unleashed SDK repository and checkout to appropriate version:

```
git clone --recursive https://github.com/sifive/freedom-u-sdk.git
cd freedom-u-sdk
git checkout hifive-unleashed
cd buildroot
git checkout sifive
cd ..
```

Note: Recursively cloning the Freedom Unleashed SDK repository may take a while. Should it fail, use `git submodule update --init --recursive` inside the `freedom-u-sdk` folder to make sure you have all the relevant files.

9.3 Building

Simply run:

```
make -j $(nproc)
```

Note: The build process may take some time, do not be discouraged by the wait.

9.4 Flashing

Insert an empty SD card and run the following commands:

```
umount /dev/sdx     # change "x" hereinafter with a letter corresponding to your SD card!
sgdisk --clear \
       --new=1:2048:67583  --change-name=1:bootloader --typecode=1:2E54B353-1271-4842-
↪806F-E436D6AF6985 \
```

(continues on next page)

(continued from previous page)

```
      --new=2:264192:      --change-name=2:root       --typecode=2:0FC63DAF-8483-4772-
↪8E79-3D69D8477DE4 \
      /dev/sdx
dd if=freedom-u-sdk/work/bbl.bin of=/dev/sdx1 bs=4096
mke2fs -t ext3 /dev/sdx2
```

9.5 Running

Connect HiFive's USB serial console to your PC. On the PC, open your favourite serial terminal and connect to the board's serial console using e.g picocom:

```
picocom -b 115200 /dev/ttyUSBX
# where X is the device number - this can be obtained from e.g. your system log
```

Note: The UART connection parameters are: baud rate 115200 bps, no flow control, 8bit words, no parity bits. The /dev/ttyX device may not appear in the system until the HiFive board is powered.

Power up the board and wait until Linux boots. The default Linux credentials are:

username root

password sifive

CHAPTER 10

Running 32-bit Linux on LiteX/VexRiscv on Avalanche board with Microsemi PolarFire FPGA

This section contains a tutorial on how to build and run 32-bit Linux on the LiteX soft SoC with an RV32 VexRiscv CPU on the Future Electronics Avalanche Board with a PolarFire FPGA from Microsemi (a Microchip company) as well as in the Renode open source simulation framework.

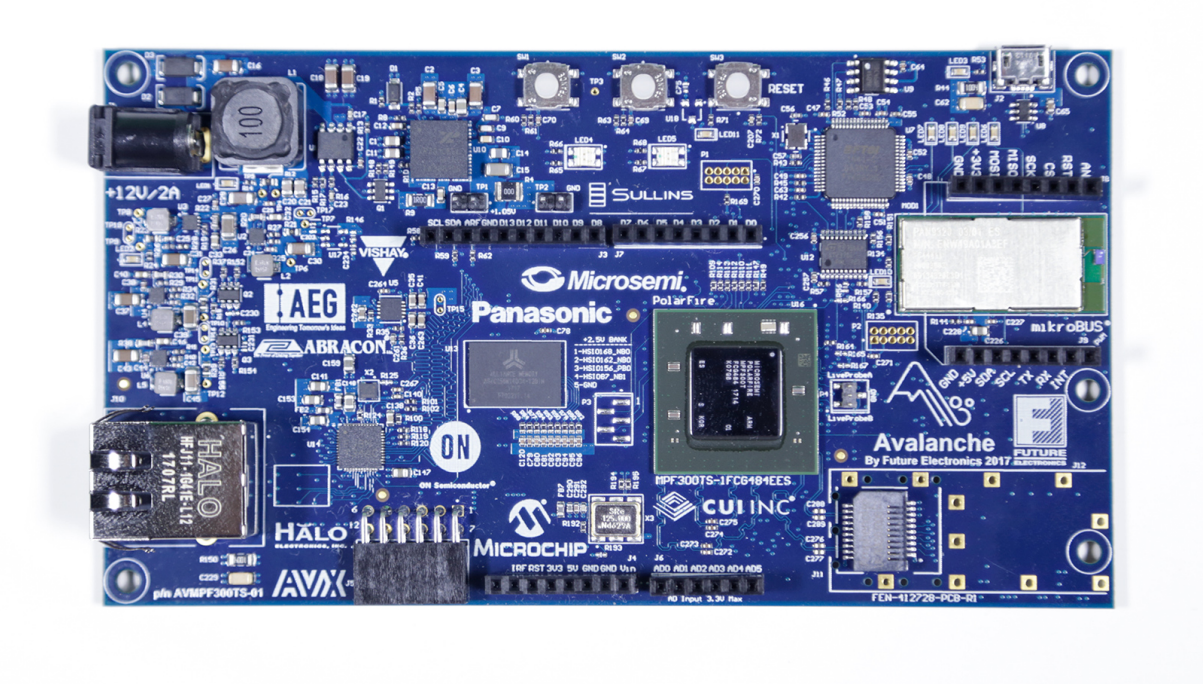

Fig. 10.1: The Future Electronics Avalanche board - top.

Note: This chapter targets Debian-based Linux flavors, and has been tested on Ubuntu 18.04.

10.1 Prerequisites

Running Linux on the Litex/VexRiscv platform requires you to install some prerequisites.

```
sudo apt install autoconf automake autotools-dev bc bison build-essential curl cpio flex \
                 gawk gdisk git gperf libgmp-dev libmpc-dev libmpfr-dev libncurses-dev \
                 libssl-dev libtool patchutils python rsync screen texinfo unzip zlib1g-
↪dev
```

10.2 Getting the sources

Clone the official Buildroot repository and apply a `patch adding VexRiscv support`

```
git clone https://github.com/buildroot/buildroot
cd buildroot
git checkout 653bf9383721a2e2d3313ae08a3019b864326
git am ../../files/0001-Add-Litex-VexRiscv-support.patch
```

10.3 Building

Simply run:

```
mkdir ~/vexriscv-bins
make litex_vexriscv_defconfig
make -j`nproc`
cp output/images/Image output/images/rootfs.cpio ~/vexriscv-bins
```

Note: The build process may take some time, do not be discouraged by the wait. The resulting binaries (`rootfs.cpio` and `Image`) will be written in the `output/images` folder.

10.4 Running

10.4.1 Preparing the platform

Hardware

Download a pregenerated bitstream of Litex with VexRiscv and BIOS preloaded to RAM:

```
wget https://github.com/riscv/risc-v-getting-started-guide/releases/download/tip/
↪bitstream-litex-vexriscv-avalanche-linux.job
```

Load it onto the Avalanche board using the PolarFire FlashPro tool. You can refer to the "Creating a Job Project from a FlashPro Express Job" section of the tool's official User Guide.

Renode

Note: Support for Linux-enabled LiteX with VexRiscv is available in Renode since version 1.7.1 - download pre-built packages from GitHub. Refer to the Renode README for more detailed installation instructions.

Start Renode and create an simulated instance of Linux-enabled LiteX+VexRiscv:

```
mach create "litex-vexriscv-linux"
machine LoadPlatformDescription @platforms/cpus/litex_vexriscv_linux.repl
```

10.4.2 Loading Linux images

First, download pre-built binaries of two files needed for running Linux on the platform, the Machine Mode emulator and the device tree:

```
cd ~/vexriscv-bins
wget https://github.com/riscv/risc-v-getting-started-guide/releases/download/tip/
↪devicetree-litex-vexriscv-avalanche-linux.dtb
wget https://github.com/riscv/risc-v-getting-started-guide/releases/download/tip/emulator-
↪litex-vexriscv-avalanche-linux.bin
```

Hardware

The Avalanche board can be flashed using the UART interface. Flashing can be done using the `litex_term` tool

```
# get the litex_term tool
wget https://raw.githubusercontent.com/enjoy-digital/litex/
↪190ff89aaa120cc983ccaeb1077ba1d23f00e37c/litex/tools/litex_term.py
wget https://raw.githubusercontent.com/antmicro/risc-v-getting-started-guide/master/
↪source/files/images-litex-vexriscv-avalanche-linux.json
chmod +x litex_term.py
# assuming the board serial has been registered as ttyUSB0
./litex_term.py --images images-litex.json /dev/ttyUSB0
```

Renode

To load all the binaries onto the simulated platform, execute the following commands:

```
sysbus LoadBinary @Image        0xc0000000
sysbus LoadBinary @rootfs.cpio  0xc0800000
sysbus LoadFdt    @devicetree-litex-vexriscv-avalanche-linux.dtb 0xc1000000
sysbus LoadBinary @emulator-litex-vexriscv-avalanche-linux.bin   0x20000000

# start executing directly from the Machine Mode emulator
cpu PC 0x20000000
```

Note: The LiteX bios plays a role of a bootloader and is required on hardware to run Linux.

In Renode, however, you can load binaries to RAM directly and set the CPU PC to its entry point, so there is no need for a bootloader.

10.4.3 Running Linux

Hardware

Reset the board. You should see the following output:

```
[LXTERM] Starting....

   __   _ __      _  __
  / /  (_) /____ | |/_/
 / /__/ / __/ -_)>  <
/____/_/\__/\__/_/|_|

(c) Copyright 2012-2019 Enjoy-Digital
(c) Copyright 2012-2015 M-Labs Ltd

BIOS built on May 10 2019 22:08:04
BIOS CRC passed (183ff024)

--============ SoC info ================--
CPU:        VexRiscv @ 100MHz
ROM:        32KB
SRAM:       32KB
L2:         8KB
MAIN-RAM:   262144KB

--========== Peripherals init ===========--
Memtest OK

--========== Boot sequence =============--
Booting from serial...
Press Q or ESC to abort boot completely.
sL5DdSMmkekro
[LXTERM] Received firmware download request from the device.
[LXTERM] Uploading Image to 0xc0000000 (2726132 bytes)...
[LXTERM] Upload complete (7.7KB/s).
[LXTERM] Uploading rootfs.cpio to 0xc0800000 (4055552 bytes)...
[LXTERM] Upload complete (7.7KB/s).
[LXTERM] Uploading rv32.dtb to 0xc1000000 (1866 bytes)...
[LXTERM] Upload complete (7.6KB/s).
[LXTERM] Uploading emulator.bin to 0x20000000 (9028 bytes)...
[LXTERM] Upload complete (7.7KB/s).
[LXTERM] Booting the device.
[LXTERM] Done.
Executing booted program at 0x20000000
```

Renode

Open a UART window and start the Renode simulation:

```
showAnalyzer sysbus.uart
start
```

Now you should see the following log of booting Linux:

```
VexRiscv Machine Mode software built May 13 2019 14:14:12
--========== Booting Linux =============--
```

(continues on next page)

(continued from previous page)

```
[    0.000000] No DTB passed to the kernel
[    0.000000] Linux version 5.0.14 (user@host) (gcc version 8.3.0 (Buildroot 2019.05-rc1-
↪00022-g653bf93)) #1 Mon May 13 10:22:15 CEST 2019
[    0.000000] Initial ramdisk at: 0x(ptrval) (8388608 bytes)
[    0.000000] Zone ranges:
[    0.000000]   Normal   [mem 0x00000000c0000000-0x00000000c7ffffff]
[    0.000000] Movable zone start for each node
[    0.000000] Early memory node ranges
[    0.000000]   node   0: [mem 0x00000000c0000000-0x00000000c7ffffff]
[    0.000000] Initmem setup node 0 [mem 0x00000000c0000000-0x00000000c7ffffff]
[    0.000000] elf_hwcap is 0x1101
[    0.000000] Built 1 zonelists, mobility grouping on.  Total pages: 32512
[    0.000000] Kernel command line: mem=128M@0x40000000 rootwait console=hvc0 root=/dev/
↪ram0 init=/sbin/init swiotlb=32
[    0.000000] Dentry cache hash table entries: 16384 (order: 4, 65536 bytes)
[    0.000000] Inode-cache hash table entries: 8192 (order: 3, 32768 bytes)
[    0.000000] Sorting __ex_table...
[    0.000000] Memory: 119052K/131072K available (1958K kernel code, 90K rwdata, 317K␣
↪rodata, 104K init, 184K bss, 12020K reserved, 0K cma-reserved)
[    0.000000] SLUB: HWalign=64, Order=0-3, MinObjects=0, CPUs=1, Nodes=1
[    0.000000] NR_IRQS: 0, nr_irqs: 0, preallocated irqs: 0
[    0.000000] clocksource: riscv_clocksource: mask: 0xffffffffffffffff max_cycles:␣
↪0x114c1bade8, max_idle_ns: 440795203839 ns
[    0.000751] sched_clock: 64 bits at 75MHz, resolution 13ns, wraps every 2199023255546ns
[    0.006686] Console: colour dummy device 80x25
[    0.205255] printk: console [hvc0] enabled
[    0.213843] Calibrating delay loop (skipped), value calculated using timer frequency..␣
↪150.00 BogoMIPS (lpj=300000)
[    0.230054] pid_max: default: 32768 minimum: 301
[    0.289177] Mount-cache hash table entries: 1024 (order: 0, 4096 bytes)
[    0.300773] Mountpoint-cache hash table entries: 1024 (order: 0, 4096 bytes)
[    0.562839] devtmpfs: initialized
[    0.777903] clocksource: jiffies: mask: 0xffffffff max_cycles: 0xffffffff, max_idle_
↪ns: 7645041785100000 ns
[    0.793367] futex hash table entries: 256 (order: -1, 3072 bytes)
[    1.514347] clocksource: Switched to clocksource riscv_clocksource
[    2.964577] Unpacking initramfs...
[   11.940415] Initramfs unpacking failed: junk in compressed archive
[   12.049860] workingset: timestamp_bits=30 max_order=15 bucket_order=0
[   13.575690] Block layer SCSI generic (bsg) driver version 0.4 loaded (major 254)
[   13.588037] io scheduler mq-deadline registered
[   13.593169] io scheduler kyber registered
[   22.876345] random: get_random_bytes called from init_oops_id+0x4c/0x60 with crng_
↪init=0
[   23.017785] Freeing unused kernel memory: 104K
[   23.025710] This architecture does not have kernel memory protection.
[   23.036765] Run /init as init process
mount: mounting tmpfs on /dev/shm failed: Invalid argument
mount: mounting tmpfs on /tmp failed: Invalid argument
mount: mounting tmpfs on /run failed: Invalid argument
Starting syslogd: OK
Starting klogd: OK
Initializing random number generator... [   36.389928] random: dd: uninitialized urandom␣
↪read (512 bytes read)
done.
Starting network: ip: socket: Function not implemented
```

(continues on next page)

(continued from previous page)

```
ip: socket: Function not implemented
FAIL

Welcome to Buildroot
buildroot login: root


   __   _                                    __   _ __      _  __   _   __          ___  _
     / /  (_)__  __ ____ _________  ___  ____/ /  (_) /____ | |/_/__| | / /____ __ / _ \(_
↪)__ _____  __
    / /__/ / _ \/ // /\ \ /___/ _ \/ _ \/___/ /__/ / __/ -_)>  </___/ |/ / -_) \ // , _/␣
↪(_-</ __/ |/ /
   /____/_/_//_/\_,_//_\_\    \___/_//_/   /____/_/\__/\__/_/|_|    |___/\__/_\_\/_/|_/_/_
↪__/\__/|___/

                          32-bits VexRiscv CPU with MMU integrated in a LiteX SoC

login[48]: root login on 'hvc0'
root@buildroot:~#
```

The default Linux credentials are:

username root

password (no pass)