

---

# **QuantumRisc-VM**

**Harald Heckmann**

**Mar 30, 2021**



# CONTENTS

<b>1</b>	<b>Introduction</b>	<b>1</b>
1.1	What is this project? . . . . .	1
1.2	Goals . . . . .	1
1.3	Contents . . . . .	2
<b>2</b>	<b>Using a QuantumRisc-VM</b>	<b>3</b>
2.1	Prerequisites . . . . .	3
2.2	Setup . . . . .	3
2.3	Usage . . . . .	12
<b>3</b>	<b>Tool build- and install scripts</b>	<b>15</b>
3.1	Prerequisites . . . . .	15
3.2	Tool build and install scripts . . . . .	15
3.3	Fully automated and configurable tools and projects install script . . . . .	17
<b>4</b>	<b>Creating a QuantumRisc-VM</b>	<b>21</b>
4.1	Prerequisites . . . . .	21
4.2	Preparing the VM . . . . .	21
4.3	Configuring and running the fully automatic install procedure . . . . .	24
4.4	Shrinking the VM . . . . .	24
<b>5</b>	<b>Extending the install scripts</b>	<b>25</b>
5.1	Single tool build and install script . . . . .	25
5.2	Fully configurable tools and project installation script . . . . .	32
<b>6</b>	<b>Script and configuration index</b>	<b>43</b>
6.1	build_tools . . . . .	43
6.2	icestorm . . . . .	53
6.3	ghdl . . . . .	56
6.4	openocd_vexriscv . . . . .	62
6.5	ujprog . . . . .	65
6.6	openocd . . . . .	68
6.7	libraries . . . . .	71
6.8	verilator . . . . .	78
6.9	spinalhdl . . . . .	81
6.10	rust_riscv . . . . .	82
6.11	gtkterm . . . . .	83
6.12	icarusverilog . . . . .	86
6.13	nextpnr . . . . .	89
6.14	riscv_tools . . . . .	94
6.15	cocotb . . . . .	102

6.16	yosys . . . . .	102
6.17	trellis . . . . .	105
6.18	spike . . . . .	108
6.19	fujprog . . . . .	111
6.20	gtkwave . . . . .	114

## INTRODUCTION

QuantumRisc is a project that aims to extend RiscV CPUs by post-quantum secure cryptography. This enables the future users of such extended RiscV CPUs to securely execute cryptography on classical computers, irrespective of the actuality that strong quantum computers exist.

### 1.1 What is this project?

This project offers an out-of-the-box usable Virtual Machine (VM) that includes many tools required for hardware and software development within the QuantumRisc project. This VM can be created by anyone by using build and install scripts, which are supplied in this project. Those scripts are configurable and depending on the configuration completely automatically install the tools. Every tool has its own script. Those scripts can be invoked one-by-one, alternatively another script can be used though, which installs and configures all tools and projects as specified in a simple configuration file.

### 1.2 Goals

The major goals were defined before the VM was specified and ultimately led to the creation of this QuantumRisc-VM project. The goals include, but are not limited to:

- A team should be able to work on a whole set of tools with identical versions. This allows progress to be shared and executed in a way that ensures that no difference in tool versions leads to errors.
- New project members should be able to start working in the project in a fast and uncomplicated manner, eliminating the effort to build and install every tool in the correct version by themselves.
- In regards to future publications, with view on the mentioning of the used development environment, a VM with a set of tools with fixed versions (which easily can be retrieved) is convenient.
- A platform-independent development environment is required to allow any project member to choose their favorite operating system.
- Single tools and complete VMs should be setup fully automatically, reducing the preliminaries to adjusting a configuration file.

## 1.3 Contents

In this section the single components of this project (QuantumRisc-VM) are summarized. This project can be used on three layers:

1. User - Hardware or Software developer in the QuantumRisc project (chapter [Using a QuantumRisc-VM](#))
2. Configurator - Usage of build and install scripts (chapter [Creating a QuantumRisc-VM](#) and [Tool build- and install scripts](#))
3. Developer - Extension of build and install scripts (chapter [Tool build- and install scripts](#), [Extending the install scripts](#) and [Script and configuration index](#))

### 1.3.1 Tool installation scripts

Any tool that is required for hardware or software development within the QuantumRisc can be installed using a fully automated installation script. Those scripts can be used independently from the VM to install the tools. Explanation on how to use these scripts is given in chapter [Tool build- and install scripts](#). All scripts and their configuration files are listed in chapter [Script and configuration index](#).

### 1.3.2 QuantumRisc-VM build script

The QuantumRisc-VM build script is a configurable builder/installer of all tools for which an installation script exists. It was made with two priorities:

1. It should be easily configurable and executable
2. The operator should be able to leave the machine and come back to a fully configured VM in a couple of hours

In a configuration file every tool and project that the script will configure, build and if desired install, can be configured. After the script has been launched and possibly after answering some prompts, the script will work autonomously. A detailed description is given in chapter [Creating a QuantumRisc-VM](#).

### 1.3.3 QuantumRisc-VM

RheinMain University offers an out-of-the-box usable VM that includes any tools required to work in the QuantumRisc project. The VM includes tools for open-source FPGA development from source code to simulation or programming of a real FPGA. This includes compilation of SpinalHDL code to Verilog or VHDL, synthesis, place and route, bitstream creation, bitstream programming for lattice fpgas, simulation and debugging. The VM also includes tools for RiscV CPU extension development which enable compiling, simulating and debugging. Finally, the VM includes projects that assist during the development of hardware-software-co-designs. It also includes a hello world project to test the available tools. The structure and usage of the QuantumRisc-VM is described in chapter [Using a QuantumRisc-VM](#).

### 1.3.4 Documentation

The installation scripts and QuantumRisc-VM build scripts are kept up to date in this documentation. Any remote changes will be automatically build and updated, so that the most recent changes are transparent. Users of the VM, users of the build scripts and developers who extend those scripts all should be able to get a majority of their relevant questions answered here.

## USING A QUANTUMRISC-VM

This Chapter deals with the download, setup and usage of a QuantumRisc-VM. The list of tools and projects included in the VM might vary from version to version, but should be included at the download page. Additionally, the tools are listed in chapter *Script and configuration index* and in a version file at the desktop of the VM.

### 2.1 Prerequisites

- QuantumRisc-VM
- [VirtualBox](#) (tested with version 6.1.10\_Ubuntu r138449)
- ~27GB hard disk space (~21GB VM image, ~6GB archive)

### 2.2 Setup

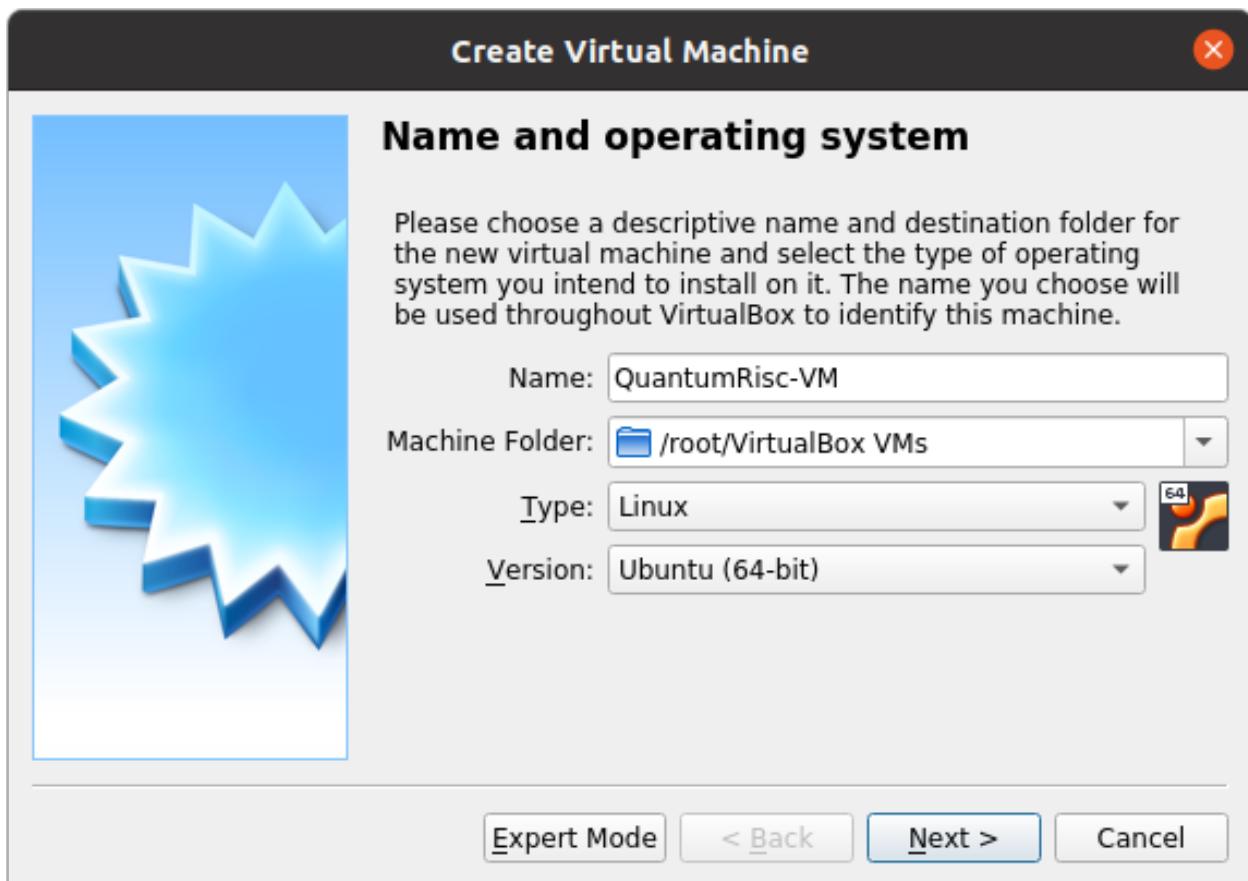
Download and extract the QuantumRisc-VM from the link mentioned in *Prerequisites*. You can get yourself a coffee or a tee, because the archive is relatively large and the extraction can take half an hour. Also install VirtualBox, following the instructions given at the vendors page.

### 2.2.1 Setting up VirtualBox

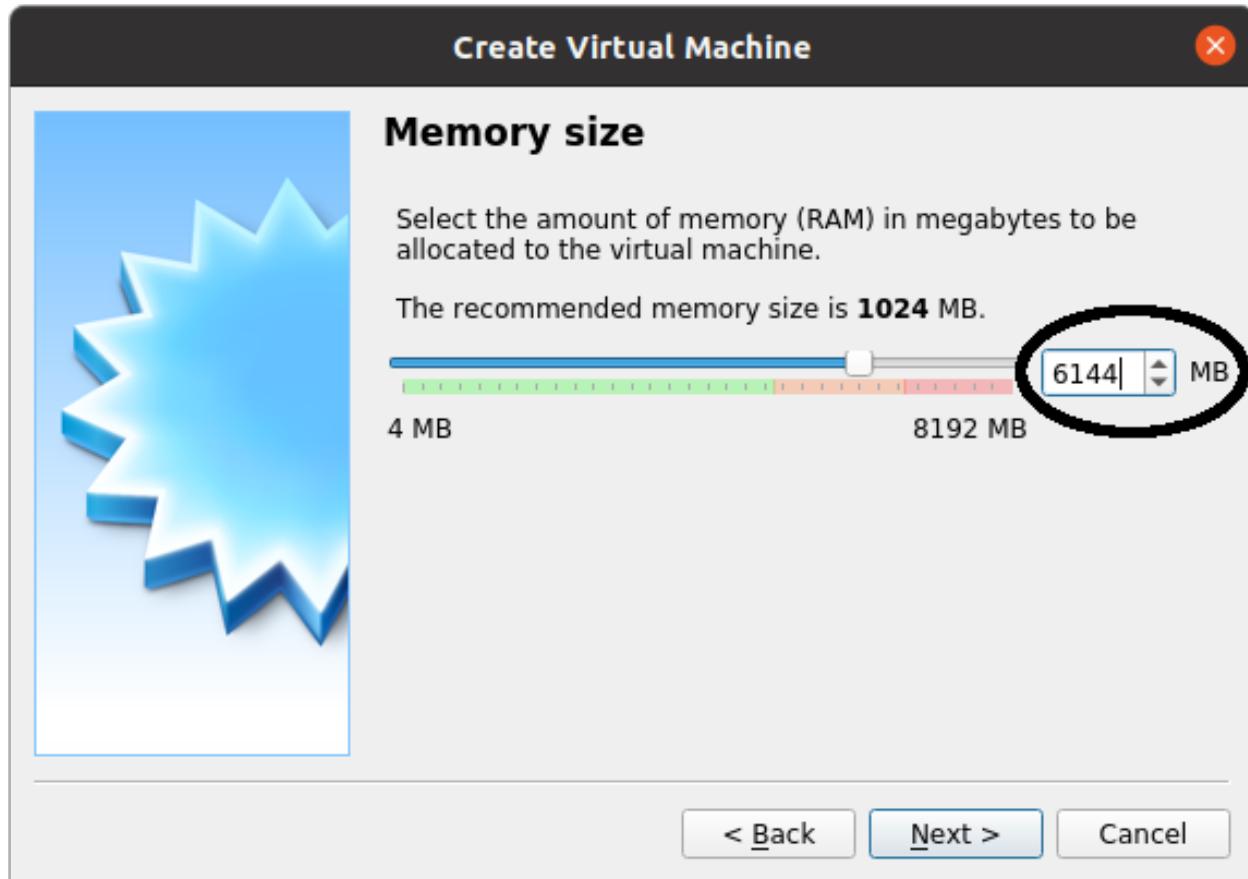
Start VirtualBox and select “new” in the toolbar to add the QuantumRisc-VM image:



Give the Virtual Machine a name and data folder:



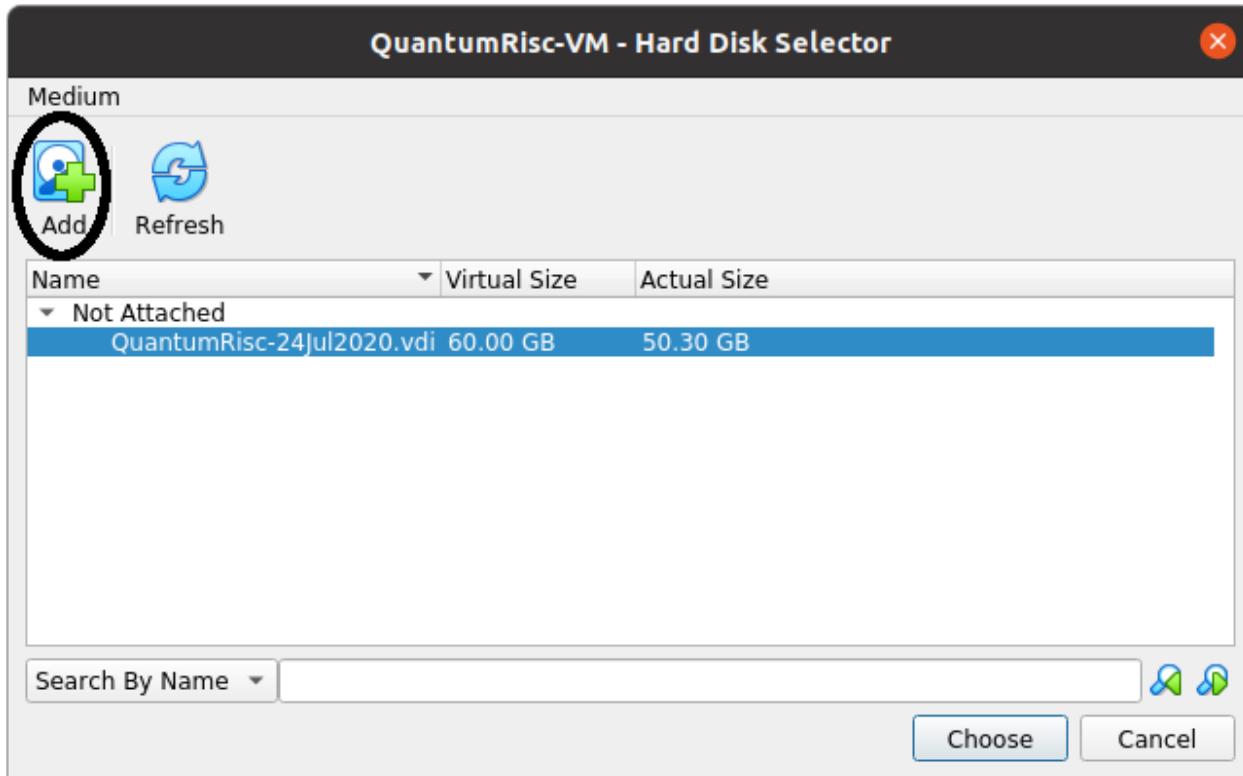
Select the amount of RAM to assign to the VM (this can be changed later). A value too low can lead either to a dysfunctional VM or massive swapping of RAM contents to the hard drive, which slows the machine down. A value too high has the same effects on the hosting machine. To use the VM, 4GB should be enough. Be aware though that building the VM requires 6GB or more of RAM, otherwise the build will fail at the RiscV toolchain (more information at chapter [Tool build- and install scripts](#)).



Select “use an existing virtual hard disk file” and press on the folder icon:



A new dialogue should open. Select “Add” and select the previously downloaded and extracted QuantumRisc-VM image:

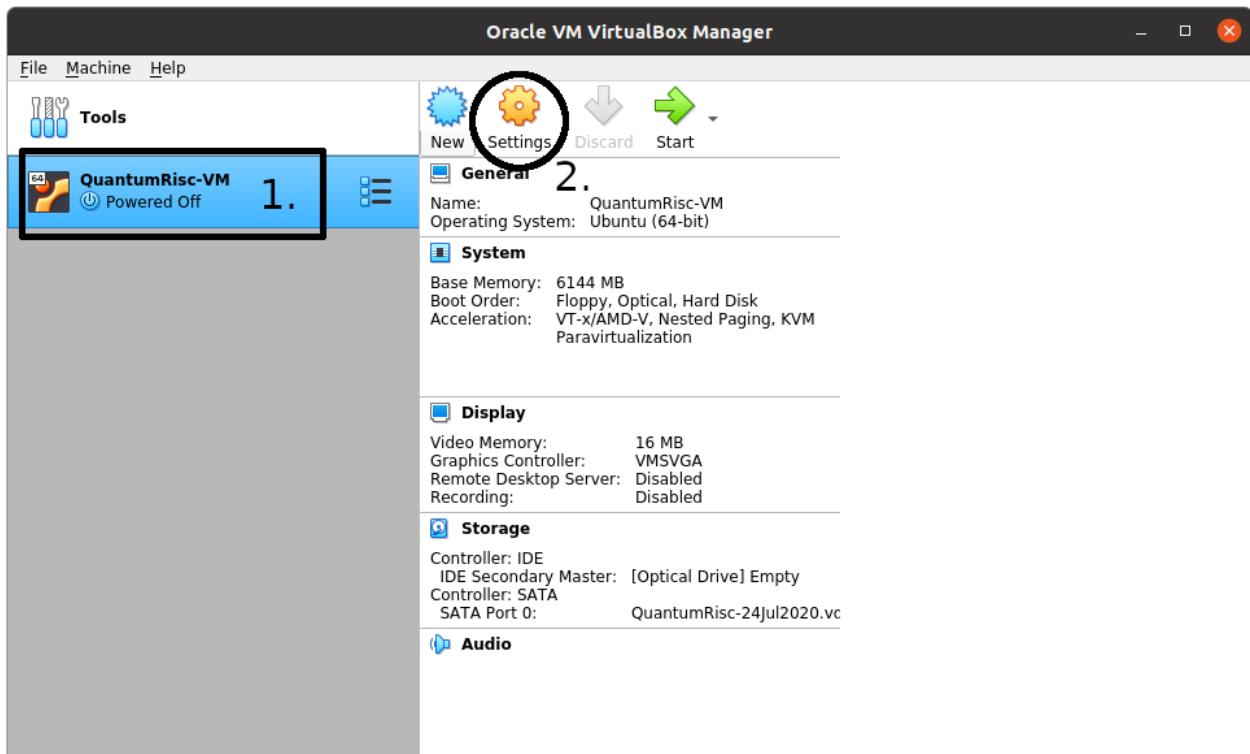


Press on “create”. Your VM has now been created and can be used. Before you use it, you should configure it as instructed in the section [Setting up QuantumRisc-VM](#).

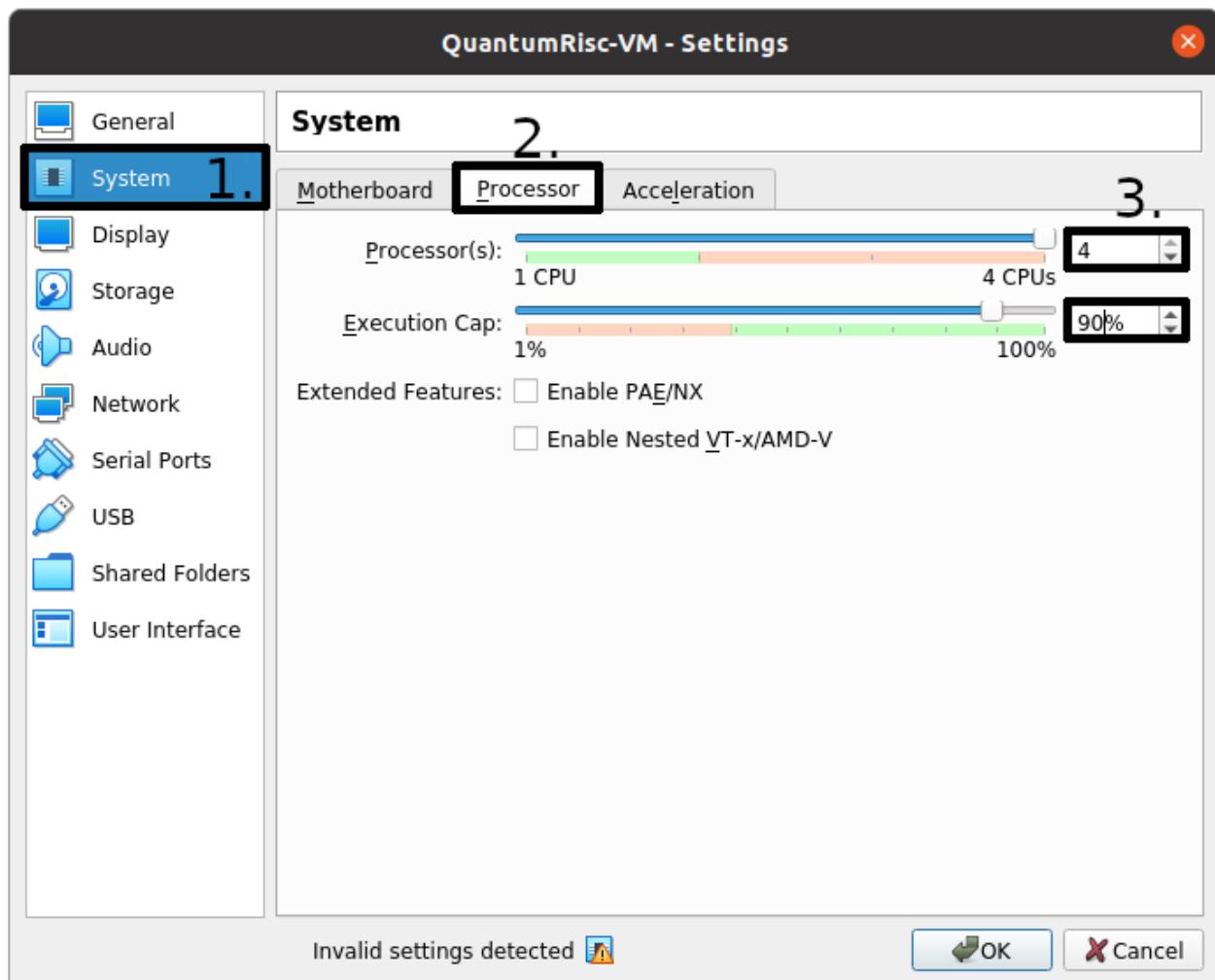
## 2.2.2 Setting up QuantumRisc-VM

After finishing the steps provided to setup VirtualBox as specified in [Setting up VirtualBox](#), a virtual machine that mounts the QuantumRisc-VM image has been created. Now we are going to assign processors and the execution cap, video memory and USB access.

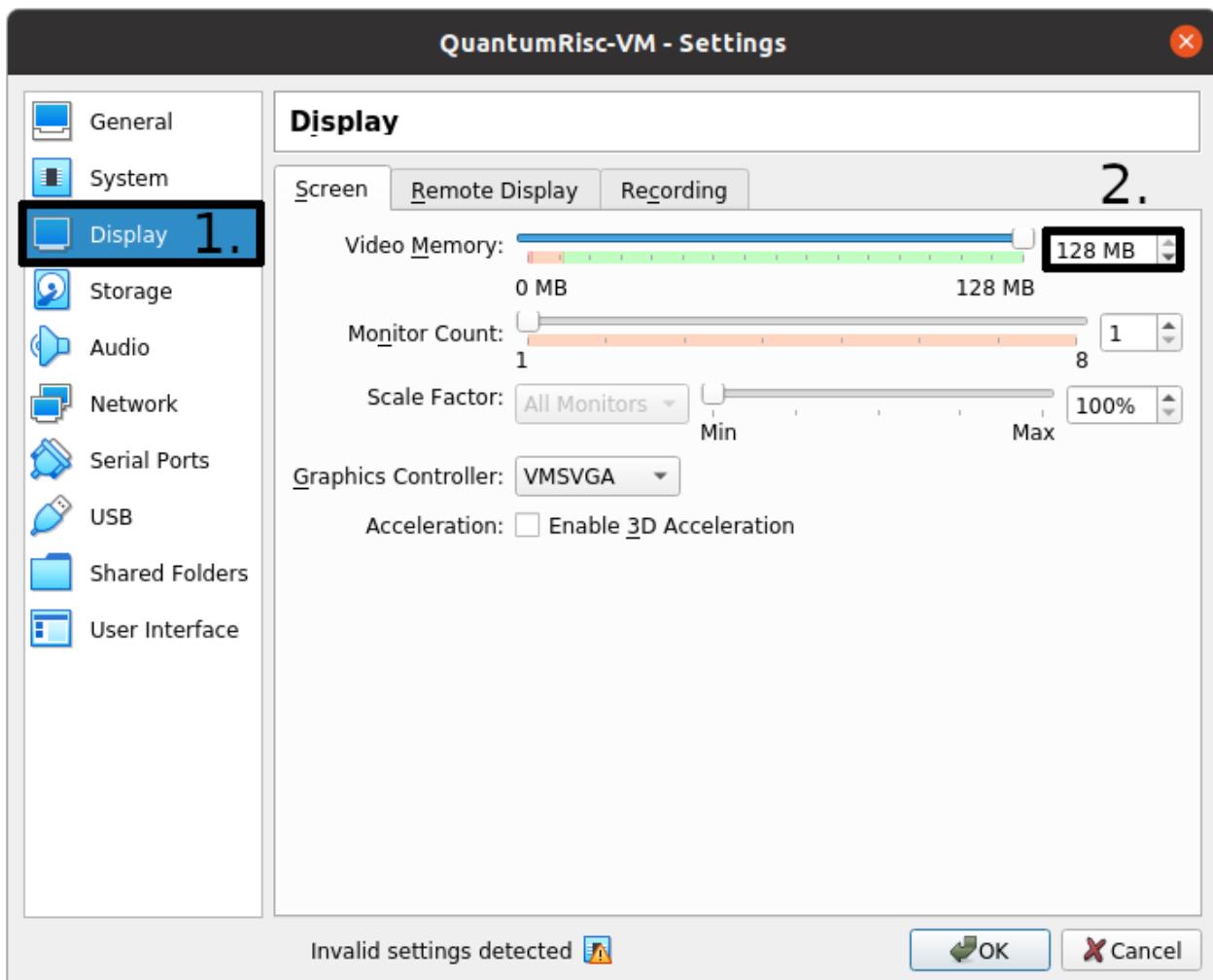
Start by selecting the VM from the list of available VMs and click on the cogwheel icon:



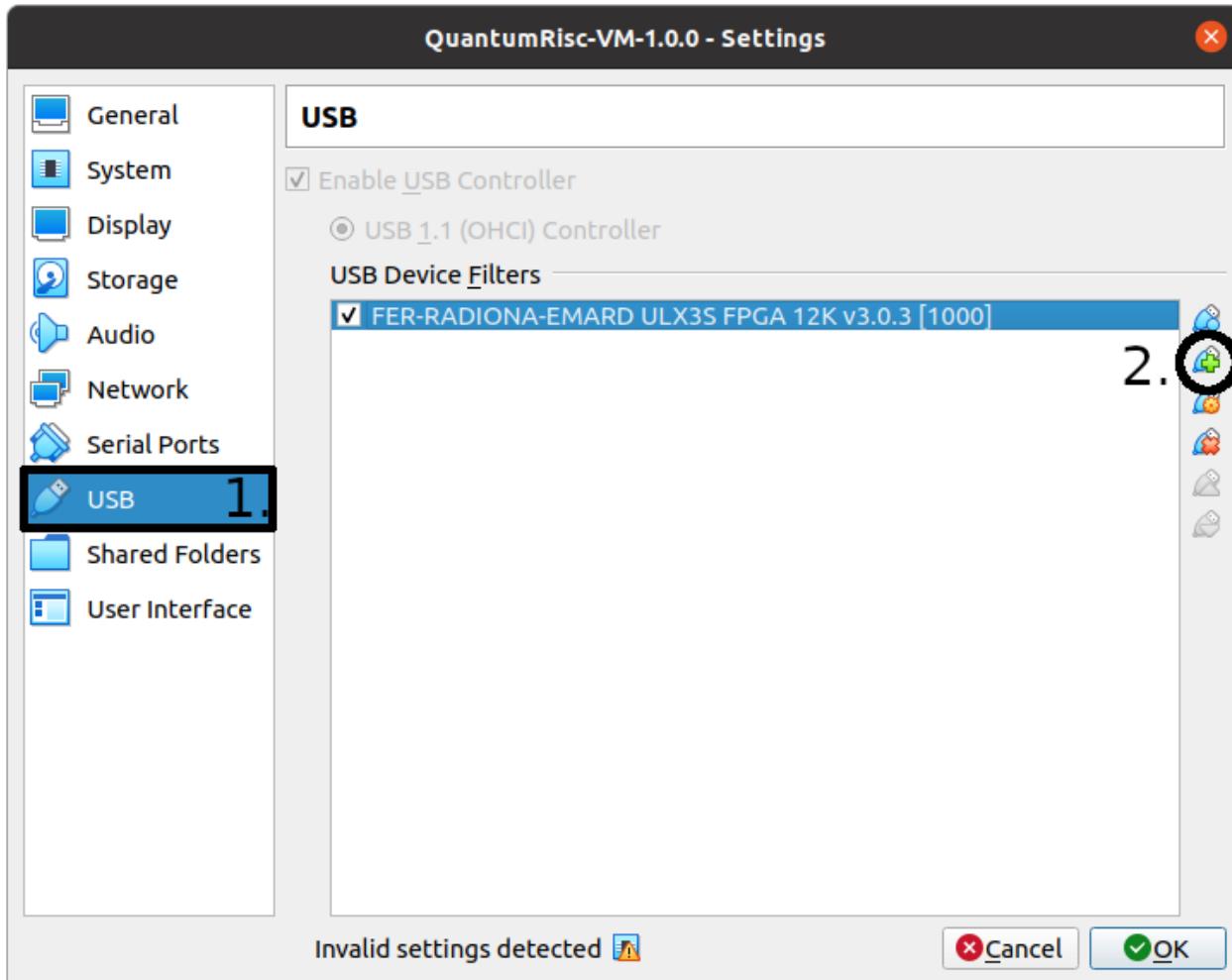
To configure the processor count and usage cap, click on “System” in the left list of categories. Select the “Processor” tab. You can specify the number of processors and the execution cap. You might not want to select 100% execution cap in case you have selected all available processors, because that might slow down or even temporarily freeze your host system.



Next select the “Display” category and specify the video memory. To avoid graphical lags you should assign as much as you can provide. You can also configure multiple monitors in that dialogue.



Complete the configuration by making sure that USB connections are passed through to your VM. This is only relevant if you want to work with devices connected over USB, for example to flash a FPGA. You have to pass through each USB device or create a filter that matches a group of devices. To permanently pass through an USB device, select the USB icon that contains a green + sign on it in the USB dialog:

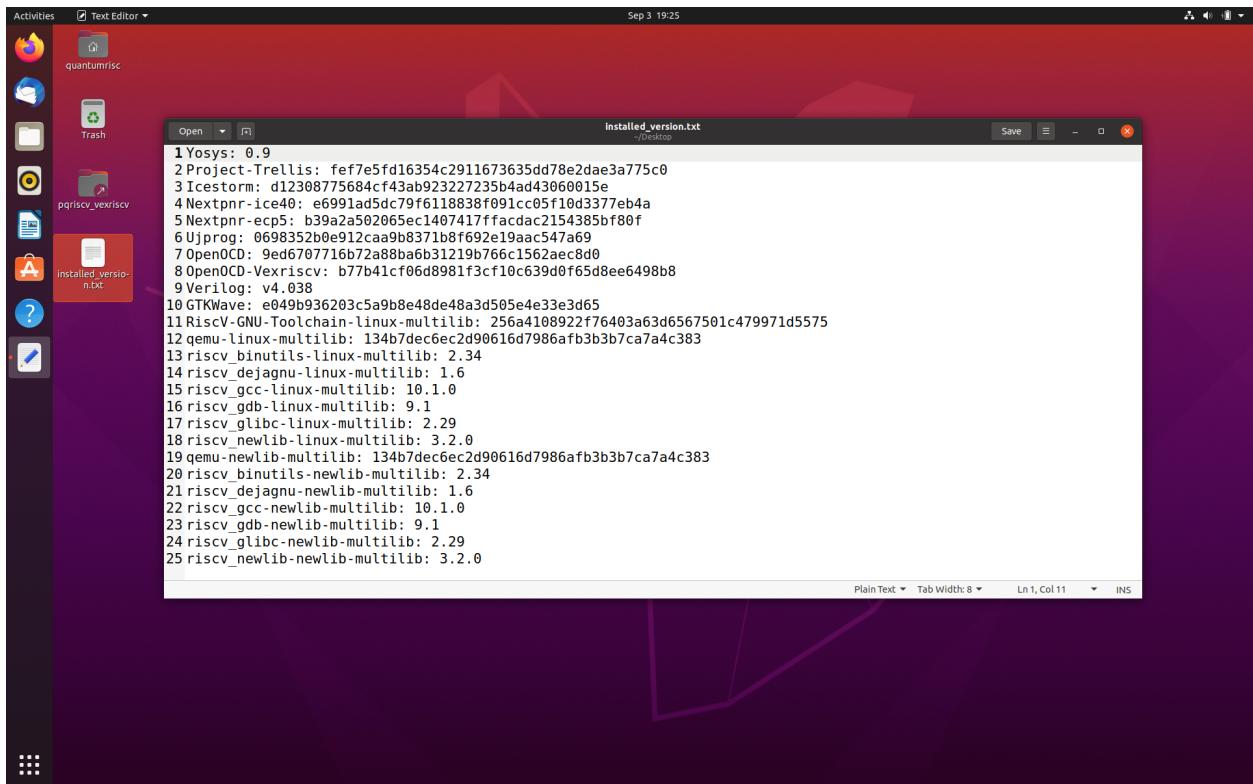


Hint: You can also add and remove permissions to pass through your USB devices during the execution of the VM. To do so, click on *Devices* -> *USB* in the menu of the running VM.

## 2.3 Usage

After setting VirtualBox and the QuantumRisc-VM up, the VM is ready to use. Start the VM, the superuser credentials can be found at the QuantumRisc-VM download page. If you can only see a black screen, press *right CTRL + F* twice. You might want to change the display resolution. This can be achieved by clicking on “Activities” in the top left corner, typing “displays” and pressing enter. You can switch between fullscreen and scaled mode by pressing hostkey + F and hostkey + S respectively. By default, the hostkey is mapped to right CTRL. If you experience graphical issues, switching to scaled mode (hostkey + S) and configuring the displays within the VM might resolve the issues.

After launching the VM you see the desktop containing a version file and symbolic links to folders:



The version file contains a version dump of all the tools that are available on the VM. All these tools are already configured and installed properly and can be used out of the box. The symbolic links to folders are links to projects that have been selected to be included into the VM by default. Those are usually projects that are being developed currently or assist during development. One of the default projects is an “Hello World” project, which serves as testkit to automatically test most of the tools that are available on the VM. This project is described in the next section usage-hello-world



## TOOL BUILD- AND INSTALL SCRIPTS

The entire project consists mainly of folders, which contain two scripts and sometimes a configuration file. The folder is named after the tool or the collection of tools, which are installed by the scripts contained within. One script does install the build essentials, using the apt package manager as it's primary source. The other script pulls, configures, builds and installs the tool in question. All scripts can be found in this documentation in *Script and configuration index*. The usage of those tool build and install scripts is described in section *Tool build and install scripts*.

In addition to scripts for every single tool, a major fully configurable script exists, which automatically builds and installs all tools and projects, for which a tool build script exists and for which the installation flag is toggled in the configuration file. For more details, skip to section *Fully automated and configurable tools and projects install script*

### 3.1 Prerequisites

- Ubuntu (tested with version 20.04 LTS)
- Build tools
- Bash (tested with version 5.0.17)
- Apt package manager (tested with version 2.0.2ubuntu0.1)

### 3.2 Tool build and install scripts

This section describes how to configure and use the tool build and install scripts.

#### 3.2.1 Preparation

Before attempting to install the tools, you have to install some build-essentials like make, compilers and the python interpreter. You only have to execute this script once on a specific machine. Locally browse to *build\_tools* and execute *install\_build\_essentials.sh* as a superuser:

```
sudo ./install_build_essentials.sh
```

### 3.2.2 Usage

The scripts are structured similarly and most of the time offer identical configuration options. Let us simulate the usage of one tool together, using explanations of the configuration options and what the script does internally. Browse to *build\_tools/verilator*. This folder contains the two script:

1. *install\_verilator\_essentials.sh*
2. *install\_verilator.sh*

This is a common naming pattern in this project, you can replace *verilator* by the names of other tools supported by this project. Both scripts require superuser privileges. To install the build essentials, the apt install command is used, that requires superuser privileges. Furthermore to install the built script, superuser privileges are required. The script could be designed such that superuser privileges are requested when required. By using this alternative approach, a fully automatic sequential installation of all tools would not be possible if the user does forget to run the scripts as superuser, because after a certain time the user must type in the superuser credentials again. You should install the software required to build the tool before building it by invoking the *install\_<toolname>.sh* script, in this case:

```
sudo ./install_verilator_essentials.sh
```

After the build essentials have been installed, we can build and install the tool. Let's check out the parameters by executing the script with the *-h* option:

```
./install_verilator.sh -h
```

This prints the following output (for verilator):

```
install_verilator.sh [-h] [-c] [-d dir] [-i path] [-t tag] -- Clone latested tagged  
→verilator  
version and build it. Optionally select the build directory and version, install  
→binaries and  
cleanup setup files.  
  
where:  
  -h      show this help text  
  -c      cleanup project  
  -d dir  build files in "dir" (default: build_and_install_verilator)  
  -i path  install binaries to path (use "default" to use default path)  
  -t tag   specify version (git tag or commit hash) to pull (default: Latest tag)
```

The *-c*, *-d*, *-i* and *-t* options are default options that are available for every tool build and install script.

The script creates a build folder, in which the source code for the project is being pulled into and in which temporary files might be stored. The name of the build folder can be specified by using the *-d* flag.

The source code version that should be pulled can be specified by using the *-t* flag. You can specify a branch name, tag, commit hash or one of the following options:

- default/latest: Pulls the default branch
- stable: Pulls the latest tag

The default behaviour (in case *-t* was not specified) is to pull the default branch. Before using the *stable* option, be sure to check whether the repository stopped to use tags at some point in time. If this is the case, the script will pull and use an outdated version, because it does not check timestamps. If no tags are found, the default branch is used.

The scripts only builds the tools by default. To also install them (using the default path specified in the tool itself), execute the script with the *-i* flag. The *-i* flag takes one parameter, which is used to specify the install path. Set it to default to use the default install path preconfigured within the tool in question.

The last default flag is the `-c` flag, which deletes all files after the tool has been successfully installed. It is only relevant if the `-i` flag is supplied at the same invocation. Otherwise a tool that was build but not installed would be removed, which is obviously pointless because it is equivalent to no changes at all.

Some tools have additional parameters which should be documented well enough in the output of the `-h` flag.

If the tool build essentials have been installed and the invocation of the tool is realized with superuser privileges and correct parameters, the script will fully automatically install the tool in question. Note that the build and/or installation process can be canceled by the SIGINT or SIGTERM signals, the default behavior of the scripts is to remove any files created by the script though. Therefore any progress will be lost.

## 3.3 Fully automated and configurable tools and projects install script

This section describes how to configure and use the major tools and projects install script.

### 3.3.1 Preparation

The script depends on a configuration file, which specifies which tools and projects should be installed and how they are configured. This file is located in `build_tools/config.cfg`. The configuration parameters should be commented well enough to be understood, but let's take a look at Verilators configuration section

#### Tool configuration

```
## Verilator
# Build and (if desired) install Verilator?
VERILATOR=true
# Build AND install Verilator?
VERILATOR_INSTALL=true
# Install path (default = default path)
VERILATOR_INSTALL_PATH=default
# Remove build directory after successful install?
VERILATOR_CLEANUP=true
# Folder name in which the project is built
VERILATOR_DIR=default
# Specify project version to pull (default/latest, stable, tag, branch, hash)
VERILATOR_TAG=default
```

The configuration parameter names for tools follow the name conception `TOOLNAME_PARAMETER=VALUE`. The `TOOL=true` flag specifies whether this tool should be build and optionally installed or whether it should be ignored. Other than that, the four basic tool build and install script flags, that were described in [Tool build and install script parameters](#), are mirrored by the config parameters followed by `TOOL=true`. This is the minimal configuration, at the same time it is the complete set of configuration parameters for most of the tools.

## Project configuration

Beside configuration entries for tools, projects can also be configured. The configuration is identical for every project and looks like this:

```
## Pqvexriscv project
# Download git repository
PQRISCV_VEXRISCV=false
# Git URL
PQRISCV_VEXRISCV_URL="https://github.com/mupq/pqriscv-vexriscv.git"
# Specify project version to pull (default/latest, stable, tag, branch, hash)
PQRISCV_VEXRISCV_TAG=default
# If default is selected, the project is stored in the documents folder
# of each user listed in the variable PQRISCV_VEXRISCV_USER
PQRISCV_VEXRISCV_LOCATION=default
# Space separated list of users (in quotation marks) to install the project for
# in /home/$user/Documents (if PQRISCV_VEXRISCV_LOCATION=default).
# default = all logged in users. Linking to desktop is also based on this list.
PQRISCV_VEXRISCV_USER=default
# Symbolic link to /home/$user/Desktop
PQRISCV_VEXRISCV_LINK_TO_DESKTOP=true
```

The configuration parameter names for projects follow the name conception *PROJECT\_PARAMETER=VALUE*. You can toggle whether you'd like the project to be installed by specifying *PROJECT=true*. Currently the projects are limited to projects that can be pulled by using git. The git repository url can be specified as an HTTP-link in the *PROJECT\_URL=HTTPURL* parameter. The state of the git repository that should be used is reflected in the *PROJECT\_TAG=STATE* parameter. *STATE* can take the same values as the *-t* flag from the *Tool build and install script parameters*. By specifying *PROJECT\_LOCATION=PATH* you can control where the project is copied to. Leaving this value at *default* does use the documents folder inside the home directory of the user specified in the variable *PROJECT\_USER=USER*. If *PROJECT\_USER* is default, any logged on user will be regarded. Finally, it is possible to configure whether the project is linked to the desktop of the user by specifying *PROJECT\_LINK\_TO\_DESKTOP=BOOL*.

### 3.3.2 Usage

After configuring the tools and projects that shall be installed by adjusting *config.cfg*, execute the install script *install\_everything.sh* and toggle the *-h* parameter (note that the real execution requires superuser privileges):

```
./install_everything.sh -h
```

It should emit the following output:

```
install_everything.sh [-c] [-h] [-o] [-p] [-v] [-d dir] -- Build and install_
↪QuantumRisc
toolchain.

where:
  -c      cleanup, delete everything after successful execution
  -h      show this help text
  -o      space seperated list of users who shall be added to dialout
          (default: every logged in user)
  -p      space seperated list of users for whom the version file shall
          be copied to the desktop (default: every logged in user)
  -v      be verbose (spams the terminal)
  -d dir  build files in "dir" (default: build_and_install_quantumrisc_tools)
```

The parameters *-c* and *-d* are equal to the default parameters mentioned in *Tool build and install script parameters*.

The `-o` parameter is used to specify the users who are added to the dialout group. By default (if `-o` is not set), the install script installs all tools and projects for every user who is logged in during the installation process. `-o` can be used in a scenario where the install script is configured to install the tools and projects for a single user or a set of users.

The `-p` parameter lets us control which users get a copy of the version file. This file is explained in the following section [Version file](#). Identical to the behavior of `-o`, `-p` does target all logged on users by default.

The `-v` parameter enables or disables the verbose output. By default, only the current operations are printed to the console. This keeps the console relatively clean. Note that errors are still logged in a file (see [Error file](#)). By setting the `-v` parameter, every output is passed to the console. This includes compiler logs, which spam the console.

The default behavior of the script in case it receives SIGINT or SIGTERM signals, is to leave everything as it was before receiving the signal and to terminate the script. Nevertheless, the tool build script will delete the tool build folder in that case.

## Version file

Every single tool installation script does log the version the tool was build for in a file called `installed_version.txt`. The major tools and projects installation script, that is covered in this chapter, does collect the information from the version file of every tool that was build into a file called `installed_versions.txt`. The file is copied to the desktop of each user, who was specified by the `-p` parameter (every logged on user by default). This file can be used for instance when releasing a new QuantumRisc-VM version or when publishing a paper. The contents of the version file look like this:

```

Yosys: 0.9
Project-Trellis: fef7e5fd16354c2911673635dd78e2dae3a775c0
Icestorm: d12308775684cf43ab923227235b4ad43060015e
Nextpnr-ice40: e6991ad5dc79f6118838f091cc05f10d3377eb4a
Nextpnr-ecp5: b39a2a502065ec1407417ffacdac2154385bf80f
Ujprog: 0698352b0e912caa9b8371b8f692e19aac547a69
OpenOCD: 9ed6707716b72a88ba6b31219b766c1562aec8d0
OpenOCD-Vexriscv: b77b41cf06d8981f3cf10c639d0f65d8ee6498b8
Verilog: v4.038
GTKWave: e049b936203c5a9b8e48de48a3d505e4e33e3d65
RiscV-GNU-Toolchain-linux-multilib: 256a4108922f76403a63d6567501c479971d5575
qemu-linux-multilib: 134b7dec6ec2d90616d7986afb3b3b7ca7a4c383
riscv_binutils-linux-multilib: 2.34
riscv_dejagnu-linux-multilib: 1.6
riscv_gcc-linux-multilib: 10.1.0
riscv_gdb-linux-multilib: 9.1
riscv_glibc-linux-multilib: 2.29
RiscV-GNU-Toolchain-newlib-multilib: 256a4108922f76403a63d6567501c479971d5575
qemu-newlib-multilib: 134b7dec6ec2d90616d7986afb3b3b7ca7a4c383
riscv_binutils-newlib-multilib: 2.34
riscv_dejagnu-newlib-multilib: 1.6
riscv_gcc-newlib-multilib: 10.1.0
riscv_gdb-newlib-multilib: 9.1
riscv_newlib-newlib-multilib: 3.2.0

```

## Error file

Any errors that occur during the execution of the *install\_everything.sh* script are logged in the build directory, whose name is specified by the *-d* or whose name is set to the default value “build\_and\_install\_quantumrisc\_tools” if *-d* was not set. The file is named “errors.log”. If *-v* is not set, the error messages are only redirected to this file. If *-v* is set, the error messages are additionally printed in the console.

## Checkpoints

The *install\_everything.sh* script does remember which tools or projects have been successfully installed. By default, this information is stored inside the build directory in a file that’s called “latest\_success\_tools.txt”. For projects, by default a file named “latest\_success\_projects.txt” is used. If the execution of this script is canceled by the user or an error, the script remembers the state and during the next execution offers the user to continue were it stopped. The user can either decide to go on or start over. If the script terminated successfully, the user can only decide to install the latest tool or project in case the build directory was not cleaned up (id est *-c* was not set).

## Projects

All projects are only downloaded using the version that was specified in the configuration file *config.cfg*. The downloaded files are placed in the “Documents” folder inside the home folder of all users who were specified in the configuration file. In addition, a symbolic link to the projects is placed on the desktop. Currently this part only works on English systems, because the folder names “Documents” and “Desktop” are hard-coded.

## Deriving a configuration file with fixed tool versions

The project offers a script that does derive a configuration from another configuration file and a version file. The final configuration file is a duplicate of the original configuration file, but it locks the tool versions to those used in the version file.

Once you installed a set of tools using the *install\_everything.sh* script and the corresponding *config.cfg* file, a file called *installed\_versions.txt* is created in the build folder and in the same folder the *install\_everything.sh* file is located in. A script called *versiondump\_to\_config.sh* is located in the *misc\_tools* folder. This script can be invoked to derive the configuration file that locks the tool versions: *versiondump\_to\_config.sh [-h] versionfile configfile outfile*

Given a specific git commit hash or tag of the QuantumRisc-VM build tools and the derived configuration file, anybody can recreate the exact same set of tools on their own system.

## CREATING A QUANTUMRISC-VM

In this section you can learn how to setup a virtual machine, how to configure the tool and project installation script and finally how to start the fully automatic QuantumRisc-VM setup process.

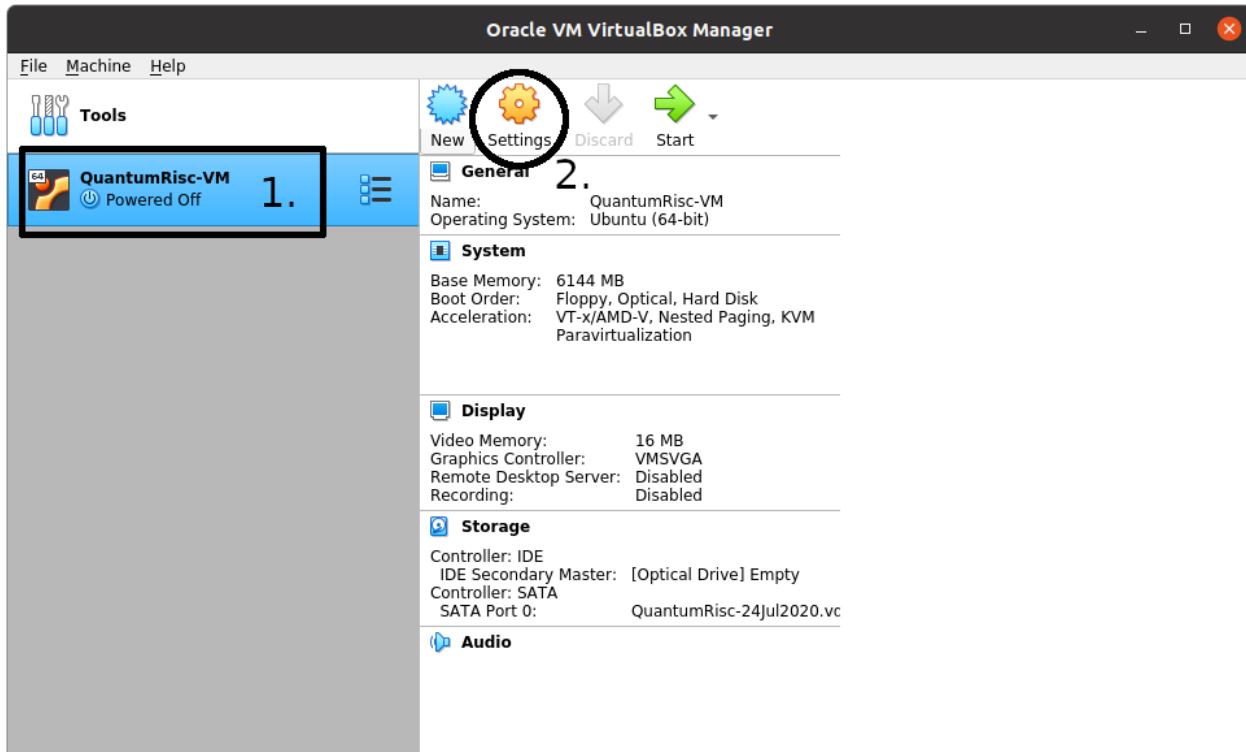
### 4.1 Prerequisites

- [VirtualBox](#) (tested with version 6.1.10\_Ubuntu r138449)
- [VirtualBox Guest Additions](#)
- [Ubuntu 20.04 LTS setup iso](#)
- [VM build tools](#)
- >6GB RAM
- >85GB hard disk space (~80GB for the VM, ~5GB to archive it)

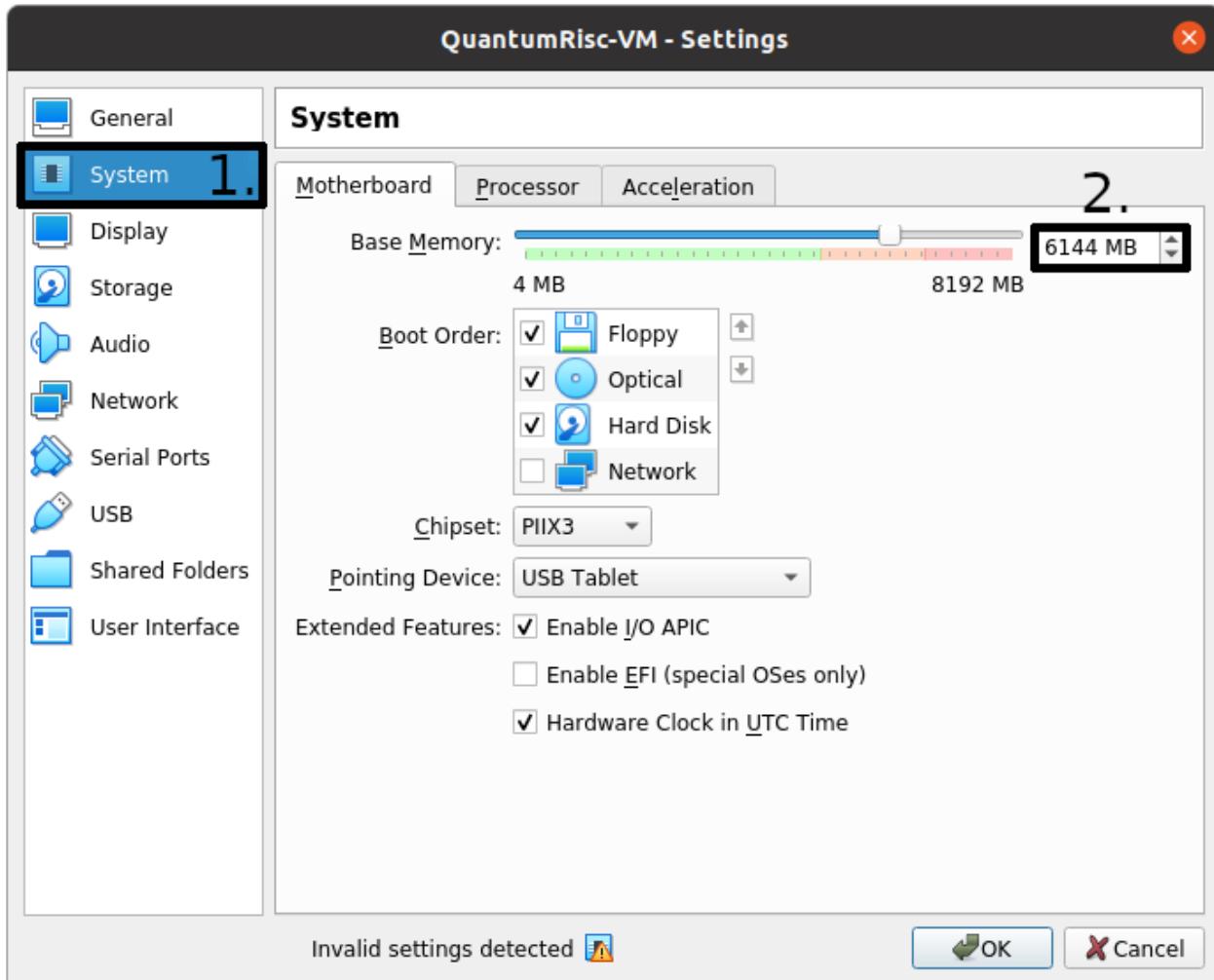
### 4.2 Preparing the VM

Follow the instructions on [how to install Ubuntu 20.04 LTS](#), but instead of allocating 30GB of disk space, choose at least 85GB (100GB recommended). You can set the username and password both to “quantumrisc”. After the successful installation of Ubuntu and all tools and projects, about 76 GB are used up. During the installation of the tools and projects, the virtual hard drive image will use up to almost 80 GB temporarily. After the VM setup is complete, we will shrink the virtual hard drive image to about 21GB.

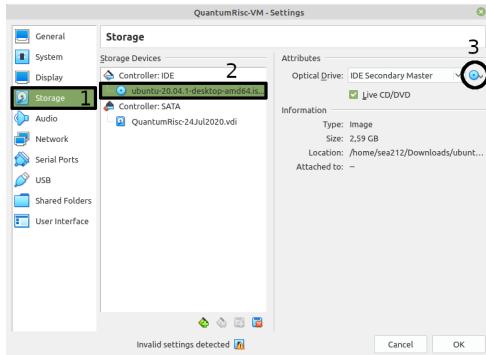
After the successful installation of [Ubuntu 20.04 LTS](#) and the [VirtualBox Guest Additions](#) on the VM, shutdown the VM and follow the instructions from section [Setting up QuantumRisc-VM](#). In addition to those instructions, you also have to raise the available memory for the VM to at least 6GB. To achieve this, select the VM and enter the *Settings* dialogue:



Switch to the *System* tab in the left menu and set the base memory to 6144 MB or more:



If you have not already removed the Ubuntu iso image from the virtual optical drive, the virtual machine will try to boot from it first. You can remove it in the *Storage* section of the *Settings* dialogue. Click on the image under the IDE Controller, next click on the disk image in the *Attributes* section and finally select “Remove Disk from Virtual Drive” in the dialogue. Since no virtual disk or floppy is detected now, the VM will boot from the virtual hard drive:



If you still experience issues booting your VM, try to change the *Boot Order* in the *System* section of the *Settings* dialogue. Give the *Hard Disk* the highest priority (top) and see if your VM boots. Note that your VM will now ignore virtual floppy or disk images during the boot process.

Start your VM and upgrade any packages on it and the kernel if desired:

```
sudo apt update && sudo apt upgrade -y && sudo apt dist-upgrade && sudo apt autoremove -y
```

## 4.3 Configuring and running the fully automatic install procedure

Copy the folder `build_tools` from the `QuantumRisc-VM` git project to `/opt/QuantumRisc-Tools`. Change the current directory to `/opt/QuantumRisc-Tools/build_tools`.

Configure the fully automated and configurable tools and project install script as desired. Instruction can be found on section *Fully automated and configurable tools and projects install script* of the scripts chapter. After an adequate configuration was created, run the install script with the desired flags (usually `-c` is enough), as explained in section *Usage* of the scripts chapter:

```
sudo ./install_everything.sh -c
```

Finally, clean up traces you left during the setup:

- browser history
- temporary files which are not required anymore
- command line history by using the command `history -c && history -w`

## 4.4 Shrinking the VM

Files deleted within the VM are not freed from the allocated space of the hard drive file on the host system. This has to be done manually. After the VM was configured completely, follow the tutorial at [howtogeek.com](http://howtogeek.com) to shrink the VM disk image file. You might find yourself with an error that the Linux partition is still mounted when executing `zerofree` as shown in the tutorial. In this case, insert the Ubuntu live CD iso image in the virtual disk like when you installed Ubuntu. After Ubuntu started from the iso image, click on “Try out” in the dialogue. Open a terminal and install `zerofree`: `sudo apt-get install zerofree`. Use the `fdisk` command to find the Linux partition `sudo fdisk -l`. Finally, use `zerofree` to zero unused disk space: `sudo zerofree -v /dev/sd<X><Y>`

## EXTENDING THE INSTALL SCRIPTS

This section covers the most difficult task of this project: Extending the install scripts. Please read the chapter [Tool build- and install scripts](#) and get familiar with the folder structure and scripts before we dive deep into the structure of the single scripts, the relationship of the scripts and configuration files and a workflow that allows usage of generic code patterns.

### 5.1 Single tool build and install script

Inside the folder `build_tools` are many other folders, all named after a single tool or a collection of tools. Each of those folders contains at least 2 scripts and optionally configuration files. One script, `install_<toolname>_essentials.sh` does install all the required libraries to build the tools. The other script, `install_<toolname>.sh`, is a parametrisable fetch, configure, build and install script for `<toolname>`.

#### 5.1.1 Extending a tool script

Since all of the tool build and install scripts are very similar, it should be sufficient to explain the structure using one specific example. In this section, we will use `build_tools/verilator/verilator` as an example.

The easiest and probably most common extension is to add (new) missing dependencies. Refer to [Missing dependencies](#) to understand how this is done.

All the scripts follow a specific code structure. We will disassemble `build_tools/verilator/install_verilator.sh` to explain the code. If you want to understand how a complete script is structured and functioning, you can just go on with this section. Alternatively, you can select one specific segment of the code:

- *Default variable initialization*
- *Parameter parsing*
- *Function section*
- *Error handling and superuser privilege enforcement*
- *Tool fetch and initialization*
- *Configuration and build*
- *Installation*
- *Cleanup*

## Missing dependencies

Take a look at `build_tools/verilator/install_verilator_essentials.sh`:

```
# require sudo
if [[ $UID != 0 ]]; then
    echo "Please run this script with sudo:"
    echo "sudo $0 $*"
    exit 1
fi

# exit when any command fails
set -e

# required tools
TOOLS="git perl python3 make g++ libfl2 libfl-dev zlib zlib1g zlib1g-dev \
      ccache libgoogle-perf-tools-dev numactl git autoconf flex bison"

# install and upgrade tools
apt-get update
apt-get install -y $TOOLS
apt-get install --only-upgrade -y $TOOLS
```

This script is rather simple. It updates the apt cache, installs all packages specified within the `TOOLS` variable and upgrades all packages that were already installed and were therefore skipped during the installation. If you want to add new dependencies, extend the `TOOLS` variable by a space followed by the package name:

```
# required tools
TOOLS="git perl python3 make g++ libfl2 libfl-dev zlib zlib1g zlib1g-dev \
      ccache libgoogle-perf-tools-dev numactl git autoconf flex bison MY-NEW-VALID-
      ↵PACKAGE"
```

Be careful though that the package exists, otherwise APT will throw an error which in return will cancel the execution of the script.

## Default variable initialization

Every tool build and install script begins with the initialization of default variables, which are either constant values or values that might be overwritten by a parameter that was passed with a flag during the invocation of the script. Take a look at the following default variable initialization section of `build_tools/verilator/install_verilator.sh`:

```
RED='\033[1;31m'
NC='\033[0m'
REPO="https://github.com/verilator/verilator.git"
PROJ="verilator"
BUILD_FOLDER="build_and_install_verilator"
VERSIONFILE="installed_version.txt"
TAG="latest"
INSTALL=false
INSTALL_PREFIX="default"
CLEANUP=false

USAGE="--snip--"
```

Currently constants and variables cannot be distinguished, it would be a good practice to add this information to the variable name in the future. This examples are the most common default variables. `RED`, `NC`, `REPO`, `PROJ`, `VERSIONFILE` and `USAGE` are constants. `RED` and `NC` are color codes, that allow you to color your console output red

(*RED*) or to reset the color (*NC*). *REPO* contains the Git URL to the project. It's important that this URL begins with *https://*, otherwise the user must supply a key. *PROJ* contains the relevant folder. Most of the time it is just the project name, sometimes it is a path to a folder within the project, like in *build\_tools/gtkwave/install\_gtkwave.sh*. *VERSION-FILE* contains the name of the file the version number is written into. The major *build\_tools/install\_everything.sh* script relies on the circumstance that all scripts use the same version filename, so it's best to never change this value and just to adapt it or change it in every single script altogether. *USAGE* contains a help string that can be printed when the program invocation was invalid.

*BUILDFOLDER*, *TAG*, *INSTALL*, *INSTALL\_PATH* and *CLEANUP* are default variables that might be altered by parameters that were supplied during the invocation of the tool build and install script. If a parameter is not passed during invocation, the script uses the value that is assigned to the corresponding default variable during initialization. Check out *Tool build and install script parameters* to learn more about tool build and install script parameters.

## Parameter parsing

The first functional action of the script is to parse arguments. Let's take a look how *install\_verilator.sh* does that:

```
while getopts ':hi:cd:t:' OPTION; do
    case $OPTION in
        i) INSTALL=true
            INSTALL_PREFIX="$OPTARG"
            echo "-i set: Installing built binaries to $INSTALL_PREFIX"
            ;;
    esac
done

OPTIND=1
```

The script checks the flags and parameters two times, because some parameters have a causal connection (e.g. cleaning up freshly built files is only reasonable if those file already have been installed/copied). The code snippet above shows the first iteration. The scripts uses *getopts* to parse the flags and parameters. The *getopts* command takes at least two parameters: A string, in this case '*:hi:cd:t:*', containing all valid flags and the information whether they expect a parameter, and a variable name to stored the flag that is currently processed. The string containing the flags '*:hi:cd:t:*' starts with a colon followed by flag letters and an optional colon after the flag letter. Every letter is a valid flag, every colon after the letter indicates that the flag is followed by a parameter. In a switch-case statement, every flag can be processed. The current parameter is stored in *\$OPTARG*. After the flags have been processed, the 'flag pointer' *OPTIND* that indicates which flag is currently processed is reset to the first flag. After that the flags are parsed a second time:

```
while getopts ':hi:cd:t:' OPTION; do
    case "$OPTION" in
        h) echo "$USAGE"
            exit
            ;;
        c) if [ $INSTALL = false ]; then
                >&2 echo -e "${RED}ERROR: -c only makes sense if the built binaries were installed before (-i)${NC}"
                exit 1
            fi
            CLEANUP=true
            echo "-c set: Removing build directory"
            ;;
        d) echo "-d set: Using folder $OPTARG"
            BUILDFOLDER="$OPTARG"
            ;;
        t) echo "-t set: Using version $OPTARG"
```

(continues on next page)

(continued from previous page)

```

        TAG="$OPTARG"
        ;;
:) echo -e "${RED}ERROR: missing argument for -$OPTARG\n${NC}" >&2
echo "$USAGE" >&2
exit 1
;;
\?) echo -e "${RED}ERROR: illegal option: -$OPTARG\n${NC}" >&2
echo "$USAGE" >&2
exit 1
;;
esac
done

shift "$((OPTIND - 1))"
```

It is important that both iterations use identical “flag strings”, otherwise some flags might be ignored. One difference to the previous run of parsing flags is that two additional cases that do not represent a specific flag are used: : and \?. The first one handles the case that a flag that requires a parameter was specified without one, the second one handles the case that a flag that is not contained in the “flag string” was passed. This is also the first output of an error messages we encounter in this section. It is printed in *RED* and redirected to stderr >&2. After the flags have been parsed, they are popped (removed) using the *shift* command.

## Function section

After the flag and parameters parsing section functions are defined. Common operations or complex operations are sourced out into functions. This increases the readability of the functional core section that configures, builds and installs the tool. Furthermore it increases the reusability in different context. Example:

```

# This function does checkout the correct version and return the commit hash or tag
# name
# Parameter 1: Branch name, commit hash, tag or one of the special keywords default/
# latest/stable
# Parameter 2: Return variable name (commit hash or tag name)
function select_and_get_project_version {
    # --snip--
}
```

For someone who is not familiar with shell scripting it might be worth mentioning that a return value (other than a return code [int]) must be passed back to the caller using a parameter that contains the variable name to store the result in.

## Error handling and superuser privilege enforcement

After the function section behavior in error cases and superuser privilege enforcement are defined:

```

# exit when any command fails
set -e

# require sudo
if [[ $UID != 0 ]]; then
    echo -e "${RED}Please run this script with sudo:"
    echo "sudo $0 $*"
    exit 1

```

(continues on next page)

(continued from previous page)

**fi**

```
# Cleanup files if the programm was shutdown unexpectedly
trap 'echo -e "${RED}ERROR: Script was terminated unexpectedly, cleaning up files...${NC}" && pushd -0 > /dev/null && rm -rf $BUILD FOLDER' INT TERM
```

The error handling is straightforward: If an error occurs, stop the execution (*set -e*). Since the script sequentially executes interdependent steps, this approach seems fine. If the project could not be downloaded, the version can't be set, it can be configured, build or installed. If the version could not be checked out, it won't go on and build the tool, using a wrong version. If it can't be configured, there is no point in building it. If nothing was build, nothing is to be installed. Either the user has to fix the error by himself (for example specify a correct project version) or to contact the developers. If the script receives a *SIGINT* or *SIGTERM* signal, it stops the execution and deletes any file it created (*trap* command).

Only one command might require superuser privileges (install), but to avoid that long-lasting scripts ask the user after an indefinite amount of time to enter superuser credentials, the script enforces superuser privileges (*\$UID == 0*).

## Tool fetch and initialization

The next snippet fetches the git project and checks out the specified version:

```
# fetch specified version
if [ ! -d $BUILD FOLDER ]; then
    mkdir $BUILD FOLDER
fi

pushd $BUILD FOLDER > /dev/null

if [ ! -d "$PROJ" ]; then
    git clone --recursive "$REPO"
fi

pushd $PROJ > /dev/null
select_and_get_project_version "$TAG" "COMMIT_HASH"
```

First it creates a workspace by creating a folder name *\$BUILD FOLDER*, which is controlled by the *-d* flag. This approach renders a simultaneous execution of multiple instances of the script possible, for example to build different versions at the same time. After that the directory is changed to the workspace. All the scripts use *pushd* and *popd*, which uses a rotatable directory stack to keep track of visited directories. The git project is fetched if the git project does not exist in the workspace yet. The *--recursive* flag is ignored if no submodules are existent, therefore it is supplied every time *git clone* is invoked. If submodules are added to the git project in the future, the script still remains functioning. At last the git project version is changed to *\$TAG*, which is controlled by the *-t* flag. If it is a valid tag, it is stored in the variable *COMMIT\_HASH*. If it is not, the commit hash is stored in *COMMIT\_HASH*. This code block is highly flexible and can be used for most if not every git project.

## Configuration and build

Next the project is configured and built, which is a part that differs from project to project:

```
# build and install if wanted
# unset var
if [ -n "$BASH" ]; then
    unset VERILATOR_ROOT
else
    unsetenv VERILATOR_ROOT
fi

autoconf

if [ "$INSTALL_PREFIX" == "default" ]; then
    ./configure
else
    ./configure --prefix="$INSTALL_PREFIX"
fi

make -j$(nproc)
```

This part of the script is basically a copy of different instructions from the build instruction of the tool in question that are weld together in a causally correct order. In this case the parameter within *INSTALL\_PREFIX*, which is either a default value or the parameter of the *-i* flag, is specified. This can happen here or later, when the command that triggers the tool installation is executed. Be sure to always supply the *-j\$(nproc)* flag to take full advantage of multi threading during the build process.

## Installation

```
if [ $INSTALL = true ]; then
    make install
fi
```

Here the tool is installed, depending on whether the *-i* flag was set. Sometimes the install location must be supplied here, this depends on the project. This is the only code segment that potentially requires superuser privileges.

## Cleanup

At the end of the project, irrelevant data can be removed:

```
# return to first folder and store version
pushd -0 > /dev/null
echo "Verilator: $COMMIT_HASH" >> "$VERSIONFILE"

# cleanup if wanted
if [ $CLEANUP = true ]; then
    rm -rf $BUILD FOLDER
fi
```

We make use of the directory stack here that comes with *pushd* and *popd*. By executing *pushd -0*, we rotate the oldest folder from the bottom to the top of the stack. Remember that the commit hash or tag was stored during the git project retrieval? At this point it is stored in a version file, which will be created at the root directory, more specifically the directory where the scripts are located. This is important if multiple people work on the same project (to ensure consistency regarding the tools) and for publications. The fully automatic and configurable tools and projects

installation script, `install_everything.sh`, collects all the tool versions in one single file. If the script was invoked with the `-c` flag, the workspace is removed completely.

### 5.1.2 Creating a tool script

Creating a tool build and install script might be easier than you think right now. Most of the time it requires only minor adaption to one of the existing scripts to create a new fully functional tool build and install script. In most cases even the integration in the major tools and projects installation script (`install_everything.sh`) only takes some minutes.

#### Step 1: Naming conventions

The naming convention is very important, because the major tools and projects installation script (`install_everything.sh`) uses them to find the scripts. Create a new folder in the `build_tools` directory which will contain the new scripts. You can give it any name, but for convenience reasons we suggest using the tool name or the collection name that are going to be installed. We'll use `<toolname>` as the name of the folder. The scripts within must be named `install_<toolname>.sh` and `install_<toolname>_essentials.sh`.

#### Step 2: Copying a template

Copy the `build_tools/verilator/install_verilator.sh` and `build_tools/verilator/install_verilator_essentials.sh` scripts to your freshly created folder `build_tools/<toolname>`. After that replace `verilator` in the name of the scripts with `<toolname>`. If your `<toolname>` is `yosys` for example, the scripts should be named `install_yosys.sh` and `install_yosys_essentials.sh`

#### Step 3: Adjusting dependencies

Lookup the dependencies on the project page and find appropriate packages in the apt packet manager. If you have a list of all dependencies, adjust the `install_<toolname>_essentials.sh` file to only install relevant apt packages, as described in section [Missing dependencies](#)

#### Step 4: Changing relevant constants

The next step encompasses the adjustment of some constants. You can view all default variables and constants at section [Default variable initialization](#). You have to change the repository url, the folder where the relevant project lies and the default value for the build folder (workspace):

```
REPO="https://github.com/verilator/verilator.git"
PROJ="verilator"
BUILD FOLDER="build_and_install_verilator"
```

At this point, your script already can parse the default flags `-c`, `-d`, `-i` and `-t`, interpret them, create a workspace based on `-d`, download the correct git project and checkout the desired version based on `-t`.

### Step 5: Adding additional flags

Adding additional flags is not difficult by itself, however, if new flags are added, the major install script `install_everything.sh` must be adjusted to process those new flags. Refer to section [Fully configurable tools and project installation script](#) for more information. If you have to add additional flags, [Parameter parsing](#) elucidates how parameters are registered, received and handled.

### Step 6: Adjusting the configure, build and install section

Depending on the project, the build process is initialized and configured differently. Get to know how to configure and build the project and reflect that knowledge in the [Configuration and build](#) segment of the script. At last, adjust the code segment that installs the project ([Installation](#)).

### Step 7: Adding the script to the major install script

This last step includes the tool install script into the major install script `install_everything.sh`. Besides potential adjustments of that script to incorporate new flags and parameters (id est any flags except `c`, `d`, `i` and `t`), the script must be registered in the major script and a config section must be created. Refer to section [Adding a tool to the script](#) to learn how this is done. After working through that section, you are done. You now have a fully functioning tool build and install script and it is integrated into the major install script, well done!

## 5.2 Fully configurable tools and project installation script

This section explains how the major install script `build_tools/install_everything.sh` is structured and how to add tool build and install scripts and projects to it.

### 5.2.1 Adding a tool to the script

Let's assume you have created a tool install script in `build_folder/<toolname>`. To add the script to the major install script, append `<TOOLNAME>` in uppercase to the following variable within the `install_everything.sh` script:

```
SCRIPTS="YOSYS TRELLIS ICESTORM NEXTPNR_ICE40 NEXTPNR_ECP5 UJPROG OPENOCD \
OPENOCD_VEXRISCV VERILATOR GTKWAVE RISCV_NEWLlib RISCV_LINUX <TOOLNAME>"
```

After that, open the configuration file for the major install script, `config.cfg`, and append the tool configuration section by a copy of the verilator configuration:

```
## Configure tools

# --snip--

## Verilator
# Build and (if desired) install Verilator?
VERILATOR=true
# Build AND install Verilator?
VERILATOR_INSTALL=true
# Install path (default = default path)
VERILATOR_INSTALL_PATH=default
# Remove build directory after successful install?
VERILATOR_CLEANUP=true
# Folder name in which the project is built
```

(continues on next page)

(continued from previous page)

```
VERILATOR_DIR=default
# Specify project version to pull (default/latest, stable, tag, branch, hash)
VERILATOR_TAG=default
```

now simply replace VERILATOR by <TOOLNAME> in uppercase and specify your desired default configuration:

```
### Configure tools

# --snip--

## <Toolname>
# Build and (if desired) install <Toolname>?
<TOOLNAME>=true
# Build AND install <Toolname>?
<TOOLNAME>_INSTALL=true
# Install path (default = default path)
<TOOLNAME>_INSTALL_PATH=default
# Remove build directory after successful install?
<TOOLNAME>_CLEANUP=true
# Folder name in which the project is built
<TOOLNAME>_DIR=default
# Specify project version to pull (default/latest, stable, tag, branch, hash)
<TOOLNAME>_TAG=default
```

## Registering additional parameters

In short, the configuration file `build_tools/config.cfg` is *sourced*, which means that every variable within it is included in the current environment. Since you followed the naming convention and included the name of your tool in the `SCRIPTS` list, the variable names that were supplied in `config.cfg` can be derived for the default configuration flags `-c`, `-d`, `-i` and `-t`. Let's take a look at the function that decides which flags and parameters are used based on the sourced `config.cfg`:

```
# Process common script parameters
# Parameter $1: Script name
# Parameter $2: Variable to store the parameters in
function parameters_tool {
    # Set "i" parameter
    if [ "$(eval "echo \$`echo \$1`_INSTALL")" = true ]; then
        eval "$2=\${!2} -i $(eval "echo \$`echo \$1`_INSTALL_PATH")\""
    fi

    # Set "c" parameter
    if [ "$(eval "echo \$`echo \$1`_CLEANUP")" = true ]; then
        eval "$2=\${!2} -c\""
    fi

    # Set "d" parameter
    local L_BUILD_DIR="$(eval "echo \$`echo \$1`_DIR")"

    if [ -n "$L_BUILD_DIR" ] && [ "$L_BUILD_DIR" != "default" ]; then
        eval "$2=\${!2} -d \"\$L_BUILD_DIR\"\""
    fi

    # Set "t" parameter
    local L_BUILD_TAG="$(eval "echo \$`echo \$1`_TAG")"
```

(continues on next page)

(continued from previous page)

```

if [ -n "$L_BUILD_TAG" ] && [ "$L_BUILD_TAG" != "default" ]; then
    eval "$2=\${!2} -t \"\$L_BUILD_TAG\"\\\""
fi

# Set "b" for Yosys only
if [ $1 == "YOSYS" ]; then
    local L_BUILD_COMPILER="\$(eval echo \$`echo \$1`_COMPILER)"
    if [ -n "\$L_BUILD_COMPILER" ]; then
        eval "$2=\${!2} -b \"\$L_BUILD_COMPILER\"\\\""
    fi
fi

# Append special parameters for gnu-riscv-toolchain and nextpnr variants
if [ "${1::5}" == "RISCV" ]; then
    parameters_tool_riscv "$1" "$2"
elif [ "${1::7}" == "NEXTPNR" ]; then
    parameters_tool_nextpnr "$1" "$2"
fi
}

```

Since every tool build and install script must follow the naming convention and support the default flags `-c`, `-d`, `-i` and `-t`, and in addition must supply the corresponding entries in `config.cfg`, the script can just derive the variable name that was specified in `config.cfg` and controls a specific flag.

Let's work through one example. You have added a tool called `MYTOOL` which support the four basic flags. In addition, you have added the configuration entry in `config.cfg`:

```

## Mytool
# Build and (if desired) install Mytool?
MYTOOL=true
# Build AND install Mytool?
MYTOOL_INSTALL=true
# --snip--

```

At some point the `install_everything.sh` script does source the configuration file, so all the variables within are now in the environment of the current instance of `install_everything.sh`, including the configuration variables for `MYTOOL`. Now at some point the `install_everything.sh` script must figure out which flags and parameters have to be set, which is done in the `parameters_tool` function in the code snippet above. The function is called like that: `parameters_tool 'MYTOOL' 'RESULT'`. First it scans the configuration variables that control the common default flags, for example for `-i`:

```

# Set "i" parameter
if [ "\$(eval echo \$`echo \$1`_INSTALL)" = true ]; then
    eval "$2=\${!2} -i \$(eval echo \$`echo \$1`_INSTALL_PATH)\\\""
fi

```

In this example the variable `$1` contains our tool name, `MYTOOL`. Within the if-statement, the eval command `\$(eval echo \$`echo \$1`_INSTALL)"` evaluates to `"$MYTOOL_INSTALL"`. This is exactly the variable name we assigned in the configuration `config.cfg` and which the script already sourced in its own environment. If the flag is set, the parameter list, which is stored in the variable name contained within `$2`, is appended by `"-i $MYTOOL_INSTALL_PATH"`. This is repeated for every default value, which the scripts resolves to the variables `MYTOOL_CLEANUP`, `MYTOOL_BUILD_DIR` and `MYTOOL_TAG`.

If you want to add a custom parameter, let's assume `MYTOOL` does now allow a `-z` flag, which builds a specific feature, you have to add it to the configuration file `config.cfg` and you have to write some custom code to handle that parameter

in addition to the default parameters. You added a configuration variable:

```
MYTOOL_NICE_FEATURE=true
```

Take a look at the end of the *parameters\_tools* function:

```
# Append special parameters for gnu-riscv-toolchain and nextpnr variants
if [ "${1::5}" == "RISCV" ]; then
    parameters_tool_riscv "$1" "$2"
elif [ "${1::7}" == "NEXTPNR" ]; then
    parameters_tool_nextpnr "$1" "$2"
fi
```

For each tool that uses additional parameters, it calls a specific function that can handle those parameters. The ` \${1::X}` command reads the first X characters from the variable \$1. It is only required if multiple tools with the same prefix use the same additional parameter function. In our case, it is sufficient to add another *elif* branch that compares the complete name:

```
elif [ "$1" == "MYTOOL" ]; then
    parameters_tool_mytool "$1" "$2"
fi
```

Create a new function *parameters\_tool\_mytool* that handles the additional parameters:

```
# Process additional mytool script parameters
# Parameter $1: Script name
# Parameter $2: Variable to store the parameters in
function parameters_tool_mytool {
    # set -z flag
    if [ "$(eval "echo \$`echo \$1`_NICE_FEATURE")" = true ]; then
        eval "$2=\$!2 -z\""
    fi
}
```

Just as for the other default flags, the if-statement checks the value of *MYTOOL\_NICE\_FEATURE* and appends the parameter string \$2 by -z if it is set to true. Congratulations, you have successfully added a custom parameters to the configuration.

## 5.2.2 Adding a project to the script

To add a project to the major install script, two steps are required:

1. Copy and adapt an existing configuration for a project from *config.cfg*
2. Add the project name to the *PROJECTS* variable in *install\_everything.sh*

Step 1: Open *config.cfg* and duplicate the last project configuration, in this case it is *DEMO\_PROJECT\_ICE40*:

```
## Hello world demo application
# Download git repository
DEMO_PROJECT_ICE40=false
# Git URL
DEMO_PROJECT_ICE40_URL="https://github.com/ThorKn/icebreaker-vexriscv-helloworld.git"
# Specify project version to pull (default/latest, stable, tag, branch, hash)
DEMO_PROJECT_ICE40_TAG=default
# If default is selected, the project is stored in the documents folder
# of each user listed in the variable DEMO_PROJECT_ICE40_USER
```

(continues on next page)

(continued from previous page)

```
DEMO_PROJECT_ICE40_LOCATION=default
# Space separated list of users (in quotation marks) to install the project for
# in /home/$user/Documents (if DEMO_PROJECT_ICE40_LOCATION=default).
# default = all logged in users. Linking to desktop is also based on this list.
DEMO_PROJECT_ICE40_USER=default
# Symbolic link to /home/$user/Desktop
DEMO_PROJECT_ICE40_LINK_TO_DESKTOP=true
```

Replace DEMO\_PROJECT with the project you want to add and adjust the configuration values as you desire:

```
## Hello world demo application
# Download git repository
<YOUR_PROJECT>=false
# Git URL
<YOUR_PROJECT>_URL=<YOUR_PROJECT_GIT_HTTPS_URL>
# Specify project version to pull (default/latest, stable, tag, branch, hash)
<YOUR_PROJECT>_TAG=default
# If default is selected, the project is stored in the documents folder
# of each user listed in the variable <YOUR_PROJECT>_USER
<YOUR_PROJECT>_LOCATION=default
# Space separated list of users (in quotation marks) to install the project for
# in /home/$user/Documents (if <YOUR_PROJECT>_LOCATION=default).
# default = all logged in users. Linking to desktop is also based on this list.
<YOUR_PROJECT>_USER=default
# Symbolic link to /home/$user/Desktop
<YOUR_PROJECT>_LINK_TO_DESKTOP=true
```

Double check every configuration parameter, especially the *URL* and if *<YOUR\_PROJECT>* is set to *true*.

Step 2: Open *install\_everything.sh* and look for the definition of the *PROJECTS* variable in the constant/default variable initialization section of the code:

```
PROJECTS="PQRISCV_VEXRISCV DEMO_PROJECT"
```

Append your project name to list, using a space as a separator:

```
PROJECTS="PQRISCV_VEXRISCV DEMO_PROJECT <YOUR_PROJECT>"
```

The major install script should now download and copy your project.

### 5.2.3 Extending the install script

The script is designed in a generic way to allow smooth integration of additional tool build and install scripts. By using naming conventions, the major install script is able to find the tool install scripts, find their configuration and invoke them with default parameters. In this section, we'll walk through the structure of the script and explain each segment.

#### Default variable initialization

The major install script first initializes default variables and constants, just like the tool build and install scripts do:

```
RED='\033[1;31m'
NC='\033[0m'
CONFIG="config.cfg"
BUILD FOLDER="build_and_install_quantumrisc_tools"
VERSIONFILE="installed_version.txt"
SUCCESS_FILE_TOOLS="latest_success_tools.txt"
SUCCESS_FILE_PROJECTS="latest_success_projects.txt"
DIALOUT_USERS=default
VERSION_FILE_USERS=default
CLEANUP=false
VERBOSE=false
SCRIPTS="YOSYS TRELLIS ICESTORM NEXTPNR_ICE40 NEXTPNR_ECP5 UJPROG OPENOCD \
OPENOCD_VEXRISCV VERILATOR GTKWAVE RISCV_NEMLIB RISCV_LINUX"
PROJECTS="PQRISCV_VEXRISCV DEMO_PROJECT"
```

Some constants and default variables are equivalent to those of a tool build and install script, refer to section *Default variable initialization* to get an explanation about their function.

*CONFIG*, *SUCCESS\_FILE\_TOOLS*, *SUCCESS\_FILE\_PROJECTS*, *SCRIPTS* and *PROJECTS* are new constants. *CONFIG* specifies the location of the configuration file. *SUCCESS\_FILE\_TOOLS* defines the name of the file that contains the latest successfully installed script. *SUCCESS\_FILE\_PROJECTS* does the same for projects. Those files contain all the information required for the checkpoint mechanism used in this script. *SCRIPTS* contains a space separated list of tool install scripts. By using naming conventions, the major install script is able to find the location of the tool build scripts and configuration values within *CONFIG*. *PROJECTS* contains a space separated list of projects, which the script uses to find the configuration for each project listed there.

In addition to those constants, some default values are defined: *DIALOUT\_USERS*, *VERSION\_FILE\_USERS* and *VERBOSE*. *DIALOUT\_USERS* contains a space separated list of users that are added to the dialout group. It is modified by the parameter of the *-o* flag. By default every logged in user is added. *VERSION\_FILE\_USERS* contains a space separated list of users for whom a copy of the final version file is placed on their desktop. The default behavior is to add the version file to the desktop of every logged in user. It is modified by the parameter of the *-p* flag. *VERBOSE* contains a boolean that toggles whether warning and errors are printed to stdout. It is toggled by the *-v* flag.

#### Parameter parsing

Refer to section *Parameter parsing* for more information.

## Function section

Please refer to section [Function section](#) before continuing in this section.

This script contains many more functions than the tool build scripts. A method that is used often in those function is the deduction of other variable names. Section [Registering additional parameters](#) explains how to add additional parameters, which includes the explanation of two important functions that use variable name deduction.

## Error handling and superuser privilege enforcement

Refer to section [Error handling and superuser privilege enforcement](#) for more information. In contrast to the tool build and install scripts, the major install script does not delete the workspace (*BUILDFOLDER*) when SIGINT or SIGTERM signals are received. This decision was made because a checkpoint mechanism was implemented, which uses files within the workspace. If the workspace would be deleted, the *install\_everything.sh* script would not know the previous progress. Running tool build and install scripts are killed and their workspace is still removed though.

## Initialization

Before the tool build and install scripts are invoked, the workspace is set up and the configuration is parsed:

```
# Read config
echo_verbose "Loading configuration file"
source config.cfg

# create and cd into buildfolder
if [ ! -d $BUILDFOLDER ]; then
    echo_verbose "Creating build folder \${BUILDFOLDER}\\"
    mkdir $BUILDFOLDER
fi

cp -r install_build_essentials.sh $BUILDFOLDER
pushd $BUILDFOLDER > /dev/null
ERROR_FILE="\$(pwd -P)/errors.log"

# Potentially create and empty errors.log file
echo '' > errors.log
echo "Executing: ./install_build_essentials.sh"
exec_verbose "./install_build_essentials.sh" "$ERROR_FILE"
```

Parsing the configuration file *build\_tools/config.cfg* is really simple. Since it only contains variable assignments in the form *VAR=value*, it is enough to *source* the configuration file. Now the script can use all the variables defined within *config.cfg*.

Just like for tool build and install scripts, a *BUILDFOLDER* is created to serve as a workspace. All builds will happen within it and every script will temporarily be copied into that workspace. Within that folder an error file *errors.log* is created. This file is going to contain any warnings and errors. The last step of the initialization includes the execution of the *install\_build\_essentials.sh* script, which install packages that deliver the functionality to download from git, configure, build and install projects.

## Handling the tools

At the core of the script lies one for loop, that iterates through every *SCRIPT* and utilizes the functions which were defined to build and eventually install the scripts:

```
echo -e "\n--- Installing tools ---\n"
get_latest "$SCRIPTS" "$SUCCESS_FILE_TOOLS" "tool" "SCRIPTS"

# Process scripts
for SCRIPT in $SCRIPTS; do
    # Should the tool be build/installed?
    if [ "${!SCRIPT}" = true ]; then
        echo "Installing $SCRIPT"
        PARAMETERS=""
        parameters_tool "$SCRIPT" "PARAMETERS"
        COMMAND_INSTALL_ESSENTIALS=""
        COMMAND_INSTALL=""
        find_script "$SCRIPT" "COMMAND_INSTALL_ESSENTIALS" "COMMAND_INSTALL"
        COMMAND_INSTALL="${COMMAND_INSTALL} ${PARAMETERS}"
        echo "Executing: $COMMAND_INSTALL_ESSENTIALS"
        exec_verbose "$COMMAND_INSTALL_ESSENTIALS" "$ERROR_FILE"
        echo "Executing: $COMMAND_INSTALL"
        exec_verbose "$COMMAND_INSTALL" "$ERROR_FILE"
        echo "$SCRIPT" > $SUCCESS_FILE_TOOLS
    fi
done
```

Before the scripts iterates over the tool build and install scripts, it checks whether some of the scripts already have successfully been installed during a previous invocation in the same workspace. The *get\_latest* function takes a list of tool build and install script names *\$SCRIPTS*, checks at which position the script contained within the checkpoint file *\$SUCCESS\_FILE\_TOOLS* is in that list, offers the users to start over or go on from there and finally stores the modified list in the last parameter, which is also called *SCRIPTS* here.

The for loop iterates over the modified list of tool build and install script names. Remember that the configuration file only contains variable assignments and the naming convention to enter *<TOOLNAME>\_PARAMETER=value*? This circumstance is used now to evaluate the tool configuration. In each iteration, the *SCRIPT* variable contains the current tool name. The command “\${!SCRIPT}” evaluates the variable that has the name that is stored in *\$SCRIPT*. So effectively the if statement looks like this in every iteration:

```
if [ "$TOOLNAME" = true ]; then
```

Since we have parsed config.cfg before, which contains “TOOLNAME=value” for any tool, we effectively have tested one element of our configuration. If the tool was configured to be build, we enter the body, which first does evaluate the configuration (using the same trick line in the if-statement) and creates a string containing the flags and parameters:

```
PARAMETERS=""
parameters_tool "$SCRIPT" "PARAMETERS"
```

After that it copies the *install\_<toolname>\_essentials.sh* script and the *install\_<toolname>.sh* script into the current workspace and appends the flags and parameters after the *install\_<toolname>.sh* script path:

```
COMMAND_INSTALL_ESSENTIALS=""
COMMAND_INSTALL=""
find_script "$SCRIPT" "COMMAND_INSTALL_ESSENTIALS" "COMMAND_INSTALL"
COMMAND_INSTALL="${COMMAND_INSTALL} ${PARAMETERS}"
echo "Executing: $COMMAND_INSTALL_ESSENTIALS"
```

At this point the naming convention is important again. The *find\_script* function assumes that the naming convention was incorporated. It copies the tool build and install script folder *<toolname>* to the current workspace and returns a path in the current workspace to *<toolname>/install\_<toolname>.sh* and *<toolname>/install\_<toolname>\_essentials.sh*. In addition, it copies an additional configuration file within the tool folder if it exists, that must be named *versions.cfg* (this will likely be changed to an arbitrary amount of config files with arbitrary names).

Everything is prepared now to execute the scripts, respecting the configuration:

```
echo "Executing: $COMMAND_INSTALL_ESSENTIALS"
exec_verbose "$COMMAND_INSTALL_ESSENTIALS" "$ERROR_FILE"
echo "Executing: $COMMAND_INSTALL"
exec_verbose "$COMMAND_INSTALL" "$ERROR_FILE"
```

At last, the current tool name *\$SCRIPT* is stored in the checkpoint file. If the next tool script should fail, this script will know where to continue.

## Handling the projects

In comparison to handling the tools, handling the projects is much simpler. Basically a project differs from tools by not requiring to be built or installed. So projects are only fetched from the web in the desired version and copied to some locations:

```
echo -e "\n--- Setting up projects ---\n"
get_latest "$PROJECTS" "$SUCCESS_FILE_PROJECTS" "project" "PROJECTS"

for PROJECT in $PROJECTS; do
    if [ "${!PROJECT}" = true ]; then
        echo "Setting up $PROJECT"
        install_project "$PROJECT"
        echo "$PROJECT" > $SUCCESS_FILE_PROJECTS
    fi
done
```

Just as for tools, a checkpoint mechanism is used for projects. Same logic, just a different file name. The configuration trick is the same here as well. *PROJECT* contains the name of the current project, *\$/!PROJECT* checks its value, which previously was defined in the configuration file in the form of *<PROJECT>=value*. If the project was configured to be installed, the body of the for loop is entered:

```
echo "Setting up $PROJECT"
install_project "$PROJECT"
echo "$PROJECT" > $SUCCESS_FILE_PROJECTS
```

The function *install\_project* is called, which downloads and configures the project based on the configuration. The project is placed at the users documents folder and if desired, linked to desktop. After the projects was successfully installed, it is stored in the projects checkpoints file.

## Cleanup

Before cleaning up the workspace (-c), that means deleting it, the version file is copied out of the workspace and into the same folder the `install_everything.sh` script lies. Additionally, it is copied to the desktop of the users specified in the variable `VERSION_FILE_USERS`:

```
# secure version file before it gets deleted (-c)
pushd -0 > /dev/null

if [ -f "${BUILD FOLDER}/${VERSIONFILE}" ]; then
    cp "${BUILD FOLDER}/${VERSIONFILE}" .
fi

# --snip--

# copy version file to users desktop
if [ "$VERSION_FILE_USERS" == "default" ]; then
    copy_version_file "$(pwd -P)/${VERSIONFILE}" `who | cut -d: -f1`
else
    copy_version_file "$(pwd -P)/${VERSIONFILE}" "$VERSION_FILE_USERS"
fi
```

In addition, a set of users contained within the variable `DIALOUT_USERS` is copied to the dialout group:

```
# add users to dialout
if [ "$DIALOUT_USERS" == "default" ]; then
    for DIALOUT_USER in `who | cut -d: -f1`; do
        usermod -a -G dialout "$DIALOUT_USER"
    done
else
    for DIALOUT_USER in "$DIALOUT_USERS"; do
        usermod -a -G dialout "$DIALOUT_USER"
    done
fi
```

After that the workspace is deleted, if the -c flag was set.



## SCRIPT AND CONFIGURATION INDEX

### 6.1 build\_tools

#### 6.1.1 install\_everything.sh

```
1 #!/bin/bash
2
3 # Author: Harald Heckmann <mail@haraldheckmann.de>
4 # Date: Jul. 23 2020
5 # Project: QuantumRisc (RheinMain University) <Steffen.Reith@hs-rm.de>
6
7 # constants
8 RED='\033[1;31m'
9 NC='\033[0m'
10 LIBRARY="libraries/library.sh"
11 CONFIG="config.cfg"
12 BUILDFOLDER="build_and_install_quantumrisc_tools"
13 VERSIONFILE="installed_version.txt"
14 SUCCESS_FILE_TOOLS="latest_success_tools.txt"
15 SUCCESS_FILE_PROJECTS="latest_success_projects.txt"
16 DIALOUT_USERS=default
17 VERSION_FILE_USERS=default
18 CLEANUP=false
19 VERBOSE=false
20 SCRIPTS="YOSYS TRELLIS ICESTORM NEXTPNR_ICE40 NEXTPNR_ECP5 UJPROG OPENOCD \
21 OPENOCD_VEXRISCV VERILATOR GTKTERM GTKWAVE RISCV_NEWSLIB RISCV_LINUX FUJPROG \
22 SPINALHDL ICARUSVERILOG SPIKE GHDL COCOTB RUST_RISCV"
23 PROJECTS="PQRISCV_VEXRISCV DEMO_PROJECT_ICE40"
24
25
26 # parse arguments
27 USAGE="$basename \"$0\" [-c] [-h] [-o] [-p] [-v] [-d dir] -- Build and install \
28 #QuantumRisc toolchain.
29 where:
30     -c      cleanup, delete everything after successful execution
31     -h      show this help text
32     -o      space separated list of users who shall be added to dialout
33                  (default: every logged in user)
34     -p      space separated list of users for whom the version file shall
35                  be copied to the desktop (default: every logged in user)
36     -v      be verbose (spams the terminal)
37     -d dir  build files in \"dir\" (default: ${BUILDFOLDER})"
```

(continues on next page)

(continued from previous page)

```

38
39 while getopts ':chopvd:' OPTION; do
40     case "$OPTION" in
41         c) echo "-c set: Cleaning up everything in the end"
42             CLEANUP=true
43             ;;
44         d) echo "-d set: Using folder $OPTARG"
45             BUILDFOLDER="$OPTARG"
46             ;;
47         h) echo "$USAGE"
48             exit
49             ;;
50         o) echo "-o set: Adding users \"${OPTARG}\" to dialout"
51             DIALOUT_USERS="$OPTARG"
52             ;;
53         p) echo "-o set: Copying version file to desktop of \"${OPTARG}\\""
54             VERSION_FILE_USERS="$OPTARG"
55             ;;
56         v) echo "-v set: Being verbose"
57             VERBOSE=true
58             ;;
59         :) echo -e "${RED}ERROR: missing argument for -$OPTARG\n${NC}" >&2
60         echo "$USAGE" >&2
61         exit 1
62         ;;
63     \?) echo -e "${RED}ERROR: illegal option: -$OPTARG\n${NC}" >&2
64         echo "$USAGE" >&2
65         exit 1
66         ;;
67     esac
68 done
69 shift $((OPTIND - 1))
70
71 # exit when any command fails
72 set -e
73
74 # require sudo
75 if [[ $UID != 0 ]]; then
76     echo -e "${RED}Please run this script with sudo:\n${NC}"
77     echo "sudo $0 $*"
78     exit 1
79 fi
80
81 # load shared functions
82 source $LIBRARY
83
84 # Read config
85 echo_verbose "Loading configuration file"
86 source config.cfg
87
88 # create and cd into buildfolder
89 if [ ! -d $BUILDFOOLDER ]; then
90     echo_verbose "Creating build folder \"${BUILDFOOLDER}\\""
91     mkdir $BUILDFOOLDER
92 fi
93
94 cp -r install_build_essentials.sh $BUILDFOOLDER

```

(continues on next page)

(continued from previous page)

```

95 pushd $BUILDFOLDER > /dev/null
96 ERROR_FILE=$(pwd -P)/errors.log
97
98 # Potentially create and empty errors.log file
99 echo '' > errors.log
100 echo "Executing: ./install_build_essentials.sh"
101 exec_verbose "./install_build_essentials.sh" "$ERROR_FILE"
102
103 # Cleanup files if the programm was shutdown unexpectedly
104 # trap 'echo -e "${RED}ERROR: Script was terminated unexpectedly, cleaning up files...
105 #       rm -rf $BUILDFOLDER' INT TERM
106
107 echo -e "\n--- Installing tools ---\n"
108 get_latest "$SCRIPTS" "$SUCCESS_FILE_TOOLS" "tool" "SCRIPTS"
109
110 # Process scripts
111 for SCRIPT in $SCRIPTS; do
112     # Should the tool be build/installed?
113     if [ "${!SCRIPT}" = true ]; then
114         echo "Installing $SCRIPT"
115         PARAMETERS=""
116         parameters_tool "$SCRIPT" "PARAMETERS"
117         COMMAND_INSTALL_ESSENTIALS=""
118         COMMAND_INSTALL=""
119         find_script "$SCRIPT" "COMMAND_INSTALL_ESSENTIALS" "COMMAND_INSTALL"
120
121         # Execute any file that is available
122         # build essentials
123         if [ -f "$COMMAND_INSTALL_ESSENTIALS" ]; then
124             echo "Executing: $COMMAND_INSTALL_ESSENTIALS"
125             exec_verbose "$COMMAND_INSTALL_ESSENTIALS" "$ERROR_FILE"
126         else
127             echo -e "Warning: ${COMMAND_INSTALL_ESSENTIALS##*/} not found"
128         fi
129
130         # tool install script
131         if [ -f "$COMMAND_INSTALL" ]; then
132             echo "Executing: ${COMMAND_INSTALL} ${PARAMETERS}"
133             exec_verbose "${COMMAND_INSTALL} ${PARAMETERS}" "$ERROR_FILE"
134             echo "$SCRIPT" > $SUCCESS_FILE_TOOLS
135         else
136             echo -e "Warning: ${COMMAND_INSTALL##*/} not found"
137         fi
138     done
139
140     echo -e "\n--- Setting up projects ---\n"
141     get_latest "$PROJECTS" "$SUCCESS_FILE_PROJECTS" "project" "PROJECTS"
142
143     for PROJECT in $PROJECTS; do
144         if [ "${!PROJECT}" = true ]; then
145             echo "Setting up $PROJECT"
146             install_project "$PROJECT"
147             echo "$PROJECT" > $SUCCESS_FILE_PROJECTS
148         fi
149     done

```

(continues on next page)

(continued from previous page)

```

151
152 # secure version file before it gets deleted (-c)
153 pushd -0 > /dev/null
154
155 if [ -f "${BUILD FOLDER}/ ${VERSIONFILE}" ]; then
156   cp "${BUILD FOLDER}/ ${VERSIONFILE}" .
157 fi
158
159 # add users to dialout
160 if [ "$DIALOUT_USERS" == "default" ]; then
161   for DIALOUT_USER in `who | cut -d' ' -f1` ; do
162     usermod -a -G dialout "$DIALOUT_USER"
163   done
164 else
165   for DIALOUT_USER in "$DIALOUT_USERS"; do
166     usermod -a -G dialout "$DIALOUT_USER"
167   done
168 fi
169
170 # copy version file to users desktop
171 if [ "$VERSION_FILE_USERS" == "default" ]; then
172   copy_version_file "${(pwd -P)}/${VERSIONFILE}" `who | cut -d' ' -f1`
173 else
174   copy_version_file "${(pwd -P)}/${VERSIONFILE}" "$VERSION_FILE_USERS"
175 fi
176
177 # cleanup
178 if [ $CLEANUP = true ]; then
179   echo_verbose "Cleaning up files"
180   rm -rf $BUILD FOLDER
181 fi
182
183 echo "Script finished successfully."

```

### 6.1.2 install\_build\_essentials.sh

```

1#!/bin/bash
2
3# Author: Harald Heckmann <mail@haraldheckmann.de>
4# Date: Jun. 23 2020
5# Project: QuantumRisc (RheinMain University) <Steffen.Reith@hs-rm.de>
6
7# require sudo
8if [[ $UID != 0 ]]; then
9  echo "Please run this script with sudo:"
10  echo "sudo $0 $*"
11  exit 1
12fi
13
14# exit when any command fails
15set -e
16
17# required tools
18TOOLS="build-essential git clang gcc meson ninja-build g++ python3-dev \
19      make flex bison libc6 binutils gzip bzip2 tar perl autoconf m4 \

```

(continues on next page)

(continued from previous page)

```

20      automake gettext gperf dejagnu expect tcl xdg-user-dirs \
21      python-is-python3"
22
23  # install and upgrade tools
24  apt-get update
25  apt-get install -y $TOOLS
26  apt-get install --only-upgrade -y $TOOLS

```

### 6.1.3 config.cfg

```

1  ### Configure tools
2
3
4  ## Yosys
5  # Build and (if desired) install Yosys?
6  YOSYS=true
7  # Build AND install yosys?
8  YOSYS_INSTALL=true
9  # Install path (default = default path)
10 YOSYS_INSTALL_PATH=default
11 # Remove build directory after successful install?
12 YOSYS_CLEANUP=true
13 # Folder name in which the project is built
14 YOSYS_DIR=default
15 # Compiler (gcc or clang)
16 YOSYS_COMPILER=clang
17 # Specify project version to pull (default/latest, stable, tag, branch, hash)
18 YOSYS_TAG=default
19
20
21 ## Project Trellis
22 # Build and (if desired) install Project Trellis?
23 TRELLIS=true
24 # Build AND install Project Trellis?
25 TRELLIS_INSTALL=true
26 # Install path (default = default path)
27 TRELLIS_INSTALL_PATH=default
28 # Remove build directory after successful install?
29 TRELLIS_CLEANUP=true
30 # Folder name in which the project is built
31 TRELLIS_DIR=default
32 # Specify project version to pull (default/latest, stable, tag, branch, hash)
33 TRELLIS_TAG=default
34
35
36 ## Icestorm
37 # Build and (if desired) install Icestorm?
38 ICESTORM=true
39 # Build AND install Icestorm?
40 ICESTORM_INSTALL=true
41 # Install path (default = default path)
42 ICESTORM_INSTALL_PATH=default
43 # Remove build directory after successful install?
44 ICESTORM_CLEANUP=true
45 # Folder name in which the project is built

```

(continues on next page)

(continued from previous page)

```

46 ICESTORM_DIR=default
47 # Specify project version to pull (default/latest, stable, tag, branch, hash)
48 ICESTORM_TAG=default
49
50
51 ## NextPNR-Ice40
52 # Build and (if desired) install NextPNR-ice40?
53 NEXTPNR_ICE40=true
54 # Build AND install NextPNR-ice40?
55 NEXTPNR_ICE40_INSTALL=true
56 # Install path (default = default path)
57 NEXTPNR_ICE40_INSTALL_PATH=default
58 # Remove build directory after successful install?
59 NEXTPNR_ICE40_CLEANUP=false
60 # Folder name in which the project is built
61 NEXTPNR_ICE40_DIR=default
62 # Specify project version to pull (default/latest, stable, tag, branch, hash)
63 NEXTPNR_ICE40_TAG=default
64 # Use chip dbs from the following path (default = fetch latest chip dbs)
65 NEXTPNR_ICE40_CHIPDB_PATH=default
66
67
68 ## NextPNR-Ecp5
69 # Build and (if desired) install NextPNR-Ecp5?
70 NEXTPNR_ECP5=true
71 # Build AND install NextPNR-NextPNR-Ecp5?
72 NEXTPNR_ECP5_INSTALL=true
73 # Install path (default = default path)
74 NEXTPNR_ECP5_INSTALL_PATH=default
75 # Remove build directory after successful install?
76 NEXTPNR_ECP5_CLEANUP=true
77 # Folder name in which the project is built
78 NEXTPNR_ECP5_DIR=default
79 # Specify project version to pull (default/latest, stable, tag, branch, hash)
80 NEXTPNR_ECP5_TAG=default
81 # Use chip dbs from the following path (default = fetch latest chip dbs)
82 NEXTPNR_ECP5_CHIPDB_PATH=default
83
84
85 ## Fujprog
86 # Build and (if desired) install Fujprog?
87 FUJPROG=true
88 # Build AND install Fujprog?
89 FUJPROG_INSTALL=true
90 # Install path (default = default path)
91 FUJPROG_INSTALL_PATH=default
92 # Remove build directory after successful install?
93 FUJPROG_CLEANUP=true
94 # Folder name in which the project is built
95 FUJPROG_DIR=default
96 # Specify project version to pull (default/latest, stable, tag, branch, hash)
97 FUJPROG_TAG=default
98
99
100 ## Ujprog
101 # Build and (if desired) install Ujprog?
102 UJPROG=true

```

(continues on next page)

(continued from previous page)

```

103 # Build AND install Ujprog?
104 UJPROG_INSTALL=true
105 # Install path (default = default path)
106 UJPROG_INSTALL_PATH=default
107 # Remove build directory after successful install?
108 UJPROG_CLEANUP=true
109 # Folder name in which the project is built
110 UJPROG_DIR=default
111 # Specify project version to pull (default/latest, stable, tag, branch, hash)
112 UJPROG_TAG=default
113
114
115 ## OpenOCD
116 # Build and (if desired) install OpenOCD?
117 OPENOCD=true
118 # Build AND install OpenOCD?
119 OPENOCD_INSTALL=true
120 # Install path (default = default path)
121 OPENOCD_INSTALL_PATH=default
122 # Remove build directory after successful install?
123 OPENOCD_CLEANUP=true
124 # Folder name in which the project is built
125 OPENOCD_DIR=default
126 # Specify project version to pull (default/latest, stable, tag, branch, hash)
127 OPENOCD_TAG=default
128
129
130 ## OpenOCD-VexRiscV
131 # Build and (if desired) install OpenOCD-VexRiscV?
132 OPENOCD_VEXRISCV=true
133 # Build AND install OpenOCD-VexRiscV?
134 OPENOCD_VEXRISCV_INSTALL=true
135 # Install path (default = default path)
136 OPENOCD_VEXRISCV_INSTALL_PATH=default
137 # Remove build directory after successful install?
138 OPENOCD_VEXRISCV_CLEANUP=true
139 # Folder name in which the project is built
140 OPENOCD_VEXRISCV_DIR=default
141 # Specify project version to pull (default/latest, stable, tag, branch, hash)
142 OPENOCD_VEXRISCV_TAG=default
143
144
145 ## Verilator
146 # Build and (if desired) install Verilator?
147 VERILATOR=true
148 # Build AND install Verilator?
149 VERILATOR_INSTALL=true
150 # Install path (default = default path)
151 VERILATOR_INSTALL_PATH=default
152 # Remove build directory after successful install?
153 VERILATOR_CLEANUP=true
154 # Folder name in which the project is built
155 VERILATOR_DIR=default
156 # Specify project version to pull (default/latest, stable, tag, branch, hash)
157 VERILATOR_TAG=default
158
159

```

(continues on next page)

(continued from previous page)

```

160 ## GTKTerm
161 # Build and (if desired) install GTKTerm?
162 GTKTERM=true
163 # Build AND install GTKTerm?
164 GTKTERM_INSTALL=true
165 # Install path (default = default path)
166 GTKTERM_INSTALL_PATH=default
167 # Remove build directory after successful install?
168 GTKTERM_CLEANUP=true
169 # Folder name in which the project is built
170 GTKTERM_DIR=default
171 # Specify project version to pull (default/latest, stable, tag, branch, hash)
172 GTKTERM_TAG=default
173
174
175 ## GTKWave
176 # Build and (if desired) install GTKWave?
177 GTKWAVE=true
178 # Build AND install GTKWave?
179 GTKWAVE_INSTALL=true
180 # Install path (default = default path)
181 GTKWAVE_INSTALL_PATH=default
182 # Remove build directory after successful install?
183 GTKWAVE_CLEANUP=true
184 # Folder name in which the project is built
185 GTKWAVE_DIR=default
186 # Specify project version to pull (default/latest, stable, tag, branch, hash)
187 GTKWAVE_TAG=default
188
189
190 ## RiscV-GNU-Toolchain Newlib Multilib
191 # build and install RiscV-GNU-Toolchain?
192 RISCV_NEolib=true
193 # Remove build directory after successful install?
194 RISCV_NEolib_CLEANUP=false
195 # Folder name in which the project is built
196 RISCV_NEolib_DIR=default
197 # Specify project version to pull (default/latest, stable, tag, branch, hash)
198 # Note: You can specify the version of every single tool of this toolchain in
199 # ./riscv_tools/versions.cfg
200 RISCV_NEolib_TAG=default
201 # Build with experimental vector extensions?
202 RISCV_NEolib_VECTOR=false
203 # Extend PATH by RiscV-GNU-Toolchain path?
204 RISCV_NEolib_EXTEND_PATH=true
205 # Specify user to install the toolchain for (default = everybody)
206 # Note: this only makes sense if PATH is extended (RISCV_NEolib_EXTEND_PATH)
207 RISCV_NEolib_USER=default
208 # Specify install path (default: /opt/riscv)
209 RISCV_NEolib_INSTALL_PATH=default
210
211
212 ## RiscV-GNU-Toolchain Linux Multilib
213 # build and install RiscV-GNU-Toolchain?
214 RISCV_LINUX=true
215 # Remove build directory after successful install?
216 RISCV_LINUX_CLEANUP=true

```

(continues on next page)

(continued from previous page)

```

217 # Folder name in which the project is built
218 RISCV_LINUX_DIR=default
219 # Specify project version to pull (default/latest, stable, tag, branch, hash)
220 # Note: You can specify the version of every single tool of this toolchain in
221 # ./riscv_tools/versions.cfg
222 RISCV_LINUX_TAG=default
223 # Build with experimental vector extensions?
224 RISCV_LINUX_VECTOR=false
225 # Extend PATH by RiscV-GNU-Toolchain path?
226 RISCV_LINUX_EXTEND_PATH=true
227 # Specify user to install the toolchain for (default = everybody)
228 # Note: this only makes sense if PATH is extended (RISCV_LINUX_EXTEND_PATH)
229 RISCV_LINUX_USER=default
230 # Specify install path (default: /opt/riscv)
231 RISCV_LINUX_INSTALL_PATH=default
232
233
234 ## SpinalHDL
235 # install SpinalHDL scala libraries?
236 SPINALHDL=true
237
238
239 ## Cocotb
240 # install Cocotb python package?
241 COCOTB=true
242
243
244 ## Rust and RiscV targets
245 # install Rust and RiscV targets?
246 RUST_RISCV=true
247
248
249 ## Spike
250 # Build and (if desired) install Spike?
251 SPIKE=true
252 # Build AND install Spike?
253 SPIKE_INSTALL=true
254 # Install path (default = default path)
255 SPIKE_INSTALL_PATH=default
256 # Remove build directory after successful install?
257 SPIKE_CLEANUP=true
258 # Folder name in which the project is built
259 SPIKE_DIR=default
260 # Specify project version to pull (default/latest, stable, tag, branch, hash)
261 SPIKE_TAG=default
262
263
264 ## Icarusverilog
265 # Build and (if desired) install Icarusverilog?
266 ICARUSVERILOG=true
267 # Build AND install Icarusverilog?
268 ICARUSVERILOG_INSTALL=true
269 # Install path (default = default path)
270 ICARUSVERILOG_INSTALL_PATH=default
271 # Remove build directory after successful install?
272 ICARUSVERILOG_CLEANUP=true
273 # Folder name in which the project is built

```

(continues on next page)

(continued from previous page)

```

274 ICARUSVERILOG_DIR=default
275 # Specify project version to pull (default/latest, stable, tag, branch, hash)
276 ICARUSVERILOG_TAG=default
277
278
279 ## Ghdl
280 # Build and (if desired) install Ghdl?
281 GHDL=true
282 # Build AND install Ghdl?
283 GHDL_INSTALL=true
284 # Install path (default = default path)
285 GHDL_INSTALL_PATH=default
286 # Remove build directory after successful install?
287 GHDL_CLEANUP=true
288 # Folder name in which the project is built
289 GHDL_DIR=default
290 # Specify project version to pull (default/latest, stable, tag, branch, hash)
291 GHDL_TAG=default
292 # Note: At least one of the following three backends must be build
293 # Build mcode backend?
294 GHDL_MCODE=true
295 # Build LLVM backend?
296 GHDL_LLVM=true
297 # Build GCC backend?
298 GHDL_GCC=true
299 # Build GHDL plugin for yosys? (requires yosys in PATH)
300 GHDL_YOSYS=false
301
302
303 ### Configure projects
304
305 ## Pqvexriscv project
306 # Download git repository
307 PQRISCV_VEXRISCV=false
308 # Git URL
309 PQRISCV_VEXRISCV_URL="https://github.com/mupq/pqriscv-vexriscv.git"
310 # Specify project version to pull (default/latest, stable, tag, branch, hash)
311 PQRISCV_VEXRISCV_TAG=default
312 # If default is selected, the project is stored in the documents folder
313 # of each user listed in the variable PQRISCV_VEXRISCV_USER
314 PQRISCV_VEXRISCV_LOCATION=default
315 # Space separated list of users (in quotation marks) to install the project for
316 # in /home/$user/Documents (if PQRISCV_VEXRISCV_LOCATION=default).
317 # default = all logged in users. Linking to desktop is also based on this list.
318 PQRISCV_VEXRISCV_USER=default
319 # Symbolic link to /home/$user/Desktop
320 PQRISCV_VEXRISCV_LINK_TO_DESKTOP=true
321
322 ## Hello world demo application
323 # Download git repository
324 DEMO_PROJECT_ICE40=false
325 # Git URL
326 DEMO_PROJECT_ICE40_URL="https://github.com/ThorKn/icebreaker-vexriscv-helloworld.git"
327 # Specify project version to pull (default/latest, stable, tag, branch, hash)
328 DEMO_PROJECT_ICE40_TAG=default
329 # If default is selected, the project is stored in the documents folder
330 # of each user listed in the variable DEMO_PROJECT_ICE40_USER

```

(continues on next page)

(continued from previous page)

```

331 DEMO_PROJECT_ICE40_LOCATION=default
332 # Space separated list of users (in quotation marks) to install the project for
333 # in /home/$user/Documents (if DEMO_PROJECT_ICE40_LOCATION=default).
334 # default = all logged in users. Linking to desktop is also based on this list.
335 DEMO_PROJECT_ICE40_USER=default
336 # Symbolic link to /home/$user/Desktop
337 DEMO_PROJECT_ICE40_LINK_TO_DESKTOP=true

```

## 6.2 icestorm

### 6.2.1 install\_icestorm\_essentials.sh

```

1 #!/bin/bash
2
3 # Author: Harald Heckmann <mail@haraldheckmann.de>
4 # Date: Jun. 24 2020
5 # Project: QuantumRisc (RheinMain University) <Steffen.Reith@hs-rm.de>
6
7 # require sudo
8 if [[ $UID != 0 ]]; then
9     echo "Please run this script with sudo:"
10    echo "sudo $0 $*"
11    exit 1
12 fi
13
14 # exit when any command fails
15 set -e
16
17 # required tools
18 TOOLS="build-essential clang bison flex libreadline-dev \
19      gawk tcl-dev libffi-dev git mercurial graphviz \
20      xdot pkg-config python python3 libftdi-dev \
21      qt5-default python3-dev libboost-all-dev cmake libeigen3-dev"
22
23 # install and upgrade tools
24 apt-get update
25 apt-get install -y $TOOLS
26 apt-get install --only-upgrade -y $TOOLS

```

### 6.2.2 install\_icestorm.sh

```

1 #!/bin/bash
2
3 # Author: Harald Heckmann <mail@haraldheckmann.de>
4 # Date: Jun. 24 2020
5 # Project: QuantumRisc (RheinMain University) <Steffen.Reith@hs-rm.de>
6
7 # constants
8 RED='\033[1;31m'
9 NC='\033[0m'
10 LIBRARY="../libraries/library.sh"

```

(continues on next page)

(continued from previous page)

```
11 REPO="https://github.com/cliffordwolf/icestorm.git"
12 PROJ="icestorm"
13 BUILDFOLDER="build_and_install_icestorm"
14 VERSIONFILE="installed_version.txt"
15 RULE_FILE="/etc/udev/rules.d/53-lattice-ftdi.rules"
16 # space separate multiple rules
17 RULES=('ACTION=="add", ATTR{idVendor}=="0403", ATTR{idProduct}=="6010", MODE=="666"')
18 TAG="latest"
19 INSTALL=false
20 INSTALL_PREFIX="default"
21 CLEANUP=false
22
23
24 # parse arguments
25 USAGE="$basename \"$0\" [-h] [-c] [-d dir] [-i path] [-t tag] -- Clone latested_
26 ↪tagged ${PROJ} version and build it. Optionally select the build directory and_
27 ↪version, install binaries and cleanup setup files.
28
29 where:
30     -h          show this help text
31     -c          cleanup project
32     -d dir      build files in \"dir\" (default: ${BUILDFOLDER})
33     -i path     install binaries to path (use \"default\" to use default path)
34     -t tag      specify version (git tag or commit hash) to pull (default: Latest tag)
35
36
37 while getopts ':hi:cd:t:' OPTION; do
38     case $OPTION in
39         i) INSTALL=true
40             INSTALL_PREFIX="$OPTARG"
41             echo "-i set: Installing built binaries to $INSTALL_PREFIX"
42             ;;
43         esac
44     done
45
46 OPTIND=1
47
48 while getopts ':hi:cd:t:' OPTION; do
49     case "$OPTION" in
50         h) echo "$USAGE"
51             exit
52             ;;
53         c) if [ $INSTALL = false ]; then
54             >&2 echo -e "${RED}ERROR: -c only makes sense if the built binaries_
55 ↪were installed before (-i)""
56             exit 1
57         fi
58         CLEANUP=true
59         echo "-c set: Removing build directory"
60         ;;
61         d) echo "-d set: Using folder $OPTARG"
62             BUILDFOLDER="$OPTARG"
63             ;;
64         t) echo "-t set: Using version $OPTARG"
65             TAG="$OPTARG"
66             ;;
67     esac
68 done
```

(continues on next page)

(continued from previous page)

```

64      :) echo -e "${RED}ERROR: missing argument for -$OPTARG\n${NC}" >&2
65      echo "$USAGE" >&2
66      exit 1
67      ;;
68  \?) echo -e "${RED}ERROR: illegal option: -$OPTARG\n${NC}" >&2
69      echo "$USAGE" >&2
70      exit 1
71      ;;
72  esac
73 done
74
75 shift "$((OPTIND - 1))"
76
77 # exit when any command fails
78 set -e
79
80 # require sudo
81 if [[ $UID != 0 ]]; then
82     echo -e "${RED}Please run this script with sudo:"
83     echo "sudo $0 $*"
84     exit 1
85 fi
86
87 # cleanup files if the programm was shutdown unexpectedly
88 trap 'echo -e "${RED}ERROR: Script was terminated unexpectedly, cleaning up files..."' \
89       && pushd -0 > /dev/null && rm -rf $BUILDFOLDER' INT TERM
90
91 # load shared functions
92 source $LIBRARY
93
94 # fetch specified version
95 if [ ! -d $BUILDFOLDER ]; then
96     mkdir $BUILDFOLDER
97 fi
98
99 pushd $BUILDFOLDER > /dev/null
100
101 if [ ! -d "$PROJ" ]; then
102     git clone --recursive "$REPO" "${PROJ%/*}/"
103 fi
104
105 pushd $PROJ > /dev/null
106
107 select_and_get_project_version "$TAG" "COMMIT_HASH"
108
109 # build and install if wanted
110 make -j$(nproc)
111
112 if [ $INSTALL = true ]; then
113     if [ "$INSTALL_PREFIX" == "default" ]; then
114         make install
115     else
116         make install PREFIX="$INSTALL_PREFIX"
117     fi
118 fi
119
# allow any user to access ice fpgas (no sudo)

```

(continues on next page)

(continued from previous page)

```

120 touch "$RULE_FILE"
121
122 for RULE in "${RULES[@]}"; do
123     if ! grep -q "$RULE" "$RULE_FILE"; then
124         echo -e "$RULE" >> "$RULE_FILE"
125     fi
126 done
127
128 # return to first folder and store version
129 pushd -0 > /dev/null
130 echo "${PROJ##*/}: $COMMIT_HASH" >> "$VERSIONFILE"
131
132 # cleanup if wanted
133 if [ $CLEANUP = true ]; then
134     rm -rf $BUILDFOLDER
135 fi

```

## 6.3 ghdl

### 6.3.1 install\_ghdl.sh

```

1#!/bin/bash
2
3# Author: Harald Heckmann <mail@haraldheckmann.de>
4# Date: Oct. 23 2020
5# Project: QuantumRisc (RheinMain University) <Steffen.Reith@hs-rm.de>
6
7# constants
8RED='\033[1;31m'
9NC='\033[0m'
10LIBRARY="../libraries/library.sh"
11REPO="https://github.com/ghdl/ghdl.git"
12REPO_LIBBACKTRACE="https://github.com/ianlancetaylor/libbacktrace.git"
13REPO_GCC="https://gcc.gnu.org/git/gcc.git"
14REPO_GHDL_YOSYS_PLUGIN="https://github.com/ghdl/ghdl-yosys-plugin"
15PROJ="ghdl"
16PROJ_GHDL_YOSYS_PLUGIN="ghdl-yosys-plugin"
17BUILDFOLDER="build_and_install_ghdl"
18VERSIONFILE="installed_version.txt"
19TAG="latest"
20INSTALL=false
21INSTALL_PREFIX="default"
22CLEANUP=false
23BUILD_MCODE=false
24BUILD_LLVM=false
25BUILD_GCC=false
26BUILD_YOSYS_PLUGIN=''
27DEFAULT_PREFIX='/usr/local'
28GHDL_GCC_SUFFIX='-ghdl'
29BUILD_GCC_DEFAULT_CONFIG="--enable-languages=c,vhdl --disable-bootstrap \
30--disable-lto --disable-multilib --disable-libssp --program-suffix=${GHDL_GCC_SUFFIX}"
31
32# parse arguments

```

(continues on next page)

(continued from previous page)

```

33 USAGE="$(basename "$0"") [-h] [-c] [-l] [-m] [-g] [-y] [-d dir] [-i path] [-t tag] --_
→Clone latest tagged ${PROJ} version and build it. Optionally select the build_
→directory and version, install binaries and cleanup setup files.

34
35 where:
36     -h      show this help text
37     -c      cleanup project
38     -g      build GCC backend
39     -l      build LLVM backend
40     -m      build mcode backend
41     -y      build ghdl-yosys-plugin
42     -d dir  build files in \"dir\" (default: ${BUILDFOLDER})
43     -i path  install binaries to path (use \"default\" to use default path)
44     -t tag   specify version (git tag or commit hash) to pull (default: Latest tag)
→"
45
46
47 while getopts ':hcglmyd:i:t:' OPTION; do
48     case $OPTION in
49         i) INSTALL=true
50             INSTALL_PREFIX="$OPTARG"
51             echo "-i set: Installing built binaries to $INSTALL_PREFIX"
52             ;;
53     esac
54 done

55
56 OPTIND=1

57
58 while getopts ':hcglmyd:i:t:' OPTION; do
59     case "$OPTION" in
60         h) echo "$USAGE"
61             exit
62             ;;
63         c) if [ $INSTALL = false ]; then
64             >&2 echo -e "${RED}ERROR: -c only makes sense if the built binaries_
→were installed before (-i)""
65             exit 1
66         fi
67         CLEANUP=true
68         echo "-c set: Removing build directory"
69         ;;
70         g) echo "-g set: Building GCC backend for GHDL"
71             BUILD_GCC=true
72             ;;
73         l) echo "-l set: Building LLVM backend for GHDL"
74             BUILD_LLVM=true
75             ;;
76         m) echo "-m set: Building MCODE backend for GHDL"
77             BUILD_MCODE=true
78             ;;
79         y) echo "-y set: Building ghdl yosys plugin"
80             BUILD_YOSYS_PLUGIN='--enable-synth'
81             ;;
82         d) echo "-d set: Using folder $OPTARG"
83             BUILDFOLDER="$OPTARG"
84             ;;
85         t) echo "-t set: Using version $OPTARG"

```

(continues on next page)

(continued from previous page)

```

86         TAG="$OPTARG"
87         ;;
88     : ) echo -e "${RED}ERROR: missing argument for -$OPTARG\n${NC}" >&2
89     echo "$USAGE" >&2
90     exit 1
91     ;;
92     \?) echo -e "${RED}ERROR: illegal option: -$OPTARG\n${NC}" >&2
93     echo "$USAGE" >&2
94     exit 1
95     ;;
96
97     esac
98 done
99
100 shift "$((OPTIND - 1))"
101
102 function build_mcode {
103     mkdir -p 'build_mcode'
104     pushd 'build_mcode' > /dev/null
105
106     if [ $INSTALL = true ]; then
107         if [ "$INSTALL_PREFIX" == "default" ]; then
108             INSTALL_PREFIX="$DEFAULT_PREFIX"
109         fi
110     else
111         INSTALL_PREFIX="$(pwd -P)/build"
112     fi
113
114     mkdir -p "$INSTALL_PREFIX"
115     ./configure $BUILD_YOSYS_PLUGIN --prefix="$INSTALL_PREFIX"
116     make -j$(nproc)
117
118     # ugly workaround: makefile offers no option to add a suffix, therefore
119     # two variants of ghdl (e.g. mcode and gcc) overwrite each other.
120     cp ghdl_mcode ghdl_mcode-mcode
121     make install EXEEXT='mcode'
122
123     popd > /dev/null
124 }
125
126 function build_llvm {
127     mkdir -p 'build_llvm'
128     pushd 'build_llvm' > /dev/null
129
130     if [ $INSTALL = true ]; then
131         if [ "$INSTALL_PREFIX" == "default" ]; then
132             INSTALL_PREFIX="$DEFAULT_PREFIX"
133         fi
134     else
135         INSTALL_PREFIX="$(pwd -P)/build"
136     fi
137
138     # compile latest libbacktrace.a to compile ghdl-llvm with stack backtrace support
139     if [ ! -d './libbacktrace' ]; then
140         git clone --recursive "$REPO_LIBBACKTRACE" 'libbacktrace'
141     fi
142
143     # build libbacktrace

```

(continues on next page)

(continued from previous page)

```

143 pushd 'libbacktrace' > /dev/null
144 ./configure
145 make -j$(nproc)
146 local L_LIBBACKTRACE_PATH=$(pwd -P)/.libs/libbacktrace.a"
147 popd > /dev/null
148
149 # build ghdl-llvm
150 ./configure $BUILD_YOSYS_PLUGIN --with-llvm-config --with-backtrace-lib="$L_
151 ↪LIBBACKTRACE_PATH" --prefix="$INSTALL_PREFIX"
152 make -j$(nproc)
153
154 # ugly workaround: makefile offers no option to add a suffix, therefore
155 # two variants of ghdl (e.g. mcode and gcc) overwrite each other.
156 cp ghdl_llvm ghdl_llvm-llvm
157 make install EXEEXT='-llvm'
158 popd > /dev/null
159 }
160
161 function build_gcc {
162     # download GCC sources
163     if [ ! -d 'gcc' ]; then
164         git clone --recursive "$REPO_GCC" 'gcc'
165     fi
166
167     # checkout latest release and build prerequisites
168     pushd 'gcc' > /dev/null
169     local L_GCC_SRC_PATH=`pwd -P`
170     select_and_get_project_version 'stable' 'THROWAWAY_VAR' 'releases/*'
171     ./contrib/download_prerequisites
172     popd > /dev/null
173     # configure ghdl-gcc
174     mkdir -p 'build_gcc'
175     pushd 'build_gcc' > /dev/null
176
177     if [ $INSTALL = true ]; then
178         if [ "$INSTALL_PREFIX" == "default" ]; then
179             INSTALL_PREFIX="$DEFAULT_PREFIX"
180         fi
181     else
182         INSTALL_PREFIX=$(pwd -P)/build"
183     fi
184
185     ./configure $BUILD_YOSYS_PLUGIN --with-gcc="$L_GCC_SRC_PATH" --prefix="$INSTALL_
186 ↪PREFIX"
187     local L_GCC_CONFIG="--prefix=${INSTALL_PREFIX}"
188
189     make -j$(nproc) copy-sources
190     mkdir -p 'gcc-objs'
191     pushd 'gcc-objs' > /dev/null
192
193     # check if the gcc used to compile ghdl-gcc uses default pie
194     if [ `gcc -v 2>&1 | grep -c -- "--enable-default-pie" ` -gt 0 ]; then
195         L_GCC_CONFIG="${L_GCC_CONFIG} --enable-default-pie"
196     fi
197
198     # compile gcc
199     $L_GCC_SRC_PATH/configure $L_GCC_CONFIG $BUILD_GCC_DEFAULT_CONFIG

```

(continues on next page)

(continued from previous page)

```

198     make -j$(nproc)
199     make install
200     popd > /dev/null
201     # compile ghdl
202     make -j$(nproc) ghdllib
203     make install
204     popd > /dev/null
205 }
206
207
208 # exit when any command fails
209 set -e
210
211 # require sudo
212 if [[ $UID != 0 ]]; then
213     echo -e "${RED}Please run this script with sudo:"
214     echo "sudo $0 $*"
215     exit 1
216 fi
217
218 # cleanup files if the programm was shutdown unexpectedly
219 trap 'echo -e "${RED}ERROR: Script was terminated unexpectedly, cleaning up files..."' INT TERM
220         && pushd -0 > /dev/null && rm -rf $BUILDFOLDER' INT TERM
221
222 # invalid configuration
223 if [ $BUILD_MCODE = false ] && [ $BUILD_LLVM = false ] && [ $BUILD_GCC = false ]; then
224     echo -e "${RED}ERROR: Invalid configuration (at least one of -m, -l and -g must_
225     be specified) ${NC}"
226     exit 2
227 fi
228
229 # load shared functions
230 source $LIBRARY
231
232 # fetch specified version
233 if [ ! -d $BUILDFOLDER ]; then
234     mkdir $BUILDFOLDER
235 fi
236
237 pushd $BUILDFOLDER > /dev/null
238
239 if [ ! -d "$PROJ" ]; then
240     git clone --recursive "$REPO" "${PROJ%%/*}"
241 fi
242
243 pushd $PROJ > /dev/null
244 select_and_get_project_version "$TAG" "COMMIT_HASH"
245
246 # build and install if wanted
247 if [ $BUILD_MCODE = true ]; then
248     build_mcode
249     PLUGIN_VARIANT='ghdl-mcode'
250 fi
251 if [ $BUILD_LLVM = true ]; then
252     build_llvm
253     PLUGIN_VARIANT='ghdl-llvm'
254 fi

```

(continues on next page)

(continued from previous page)

```

253 if [ $BUILD_GCC = true ]; then
254     build_gcc
255     PLUGIN_VARIANT='ghdl'
256 fi
257
258 # build ghdl plugin for yosys if wanted
259 if [ -n "$BUILD_YOSYS_PLUGIN" ]; then
260     # clone
261     if [ ! -d "$PROJ_GHDL_YOSYS_PLUGIN" ]; then
262         git clone --recursive "$REPO_GHDL_YOSYS_PLUGIN" "${PROJ_GHDL_YOSYS_PLUGIN%/*}"
263     fi
264
265 pushd $PROJ_GHDL_YOSYS_PLUGIN > /dev/null
266
267 # build
268 GHDL="${INSTALL_PREFIX}/bin/${PLUGIN_VARIANT}"
269 make -j$(nproc) GHDL="$GHDL"
270
271 # install
272 make install GHDL="$GHDL"
273 fi
274
275 # return to first folder and store version
276 pushd -0 > /dev/null
277 echo "${PROJ##*/}: ${COMMIT_HASH}" >> "$VERSIONFILE"
278
279 # cleanup if wanted
280 if [ $CLEANUP = true ]; then
281     rm -rf $BUILDFOLDER
282 fi

```

### 6.3.2 install\_ghdl\_essentials.sh

```

1 #!/bin/bash
2
3 # Author: Harald Heckmann <mail@haraldheckmann.de>
4 # Date: Oct. 23 2020
5 # Project: QuantumRisc (RheinMain University) <Steffen.Reith@hs-rm.de>
6
7 # require sudo
8 if [[ $UID != 0 ]]; then
9     echo "Please run this script with sudo:"
10    echo "sudo $0 $*"
11    exit 1
12 fi
13
14 # exit when any command fails
15 set -e
16
17 # required tools
18 TOOLS="build-essential git make gcc gnat llvm clang flex libc6 binutils gzip \
19 bzip2 tar perl autoconf m4 automake gettext gperf dejagnu expect tcl \
20 autogen guile-3.0 ssh texinfo"
21

```

(continues on next page)

(continued from previous page)

```

22 # install and upgrade tools
23 apt-get update
24 apt-get install -y $TOOLS
25 apt-get install --only-upgrade -y $TOOLS

```

## 6.4 openocd\_vexriscv

### 6.4.1 install\_openocd\_vexriscv.sh

```

1 #!/bin/bash
2
3 # Author: Harald Heckmann <mail@haraldheckmann.de>
4 # Date: Jun. 25 2020
5 # Project: QuantumRisc (RheinMain University) <Steffen.Reith@hs-rm.de>
6
7 # constants
8 RED='\033[1;31m'
9 NC='\033[0m'
10 LIBRARY="../libraries/library.sh"
11 REPO="https://github.com/SpinalHDL/openocd_riscv.git"
12 PROJ="openocd_vexriscv"
13 BUILDFOLDER="build_and_install_openocd_vexriscv"
14 VERSIONFILE="installed_version.txt"
15 TAG="latest"
16 INSTALL=false
17 INSTALL_PREFIX="default"
18 CLEANUP=false
19
20 CONFIGURE_STRING="--prefix=/usr/local --program-suffix=-vexriscv
--datarootdir=/usr/local/share/vexriscv --enable-maintainer-mode
--disable-werror --enable-ft232r --enable-ftdi --enable-jtag_vpi"
21
22
23
24
25 # parse arguments
26 USAGE="$basename \"$0\" [-h] [-c] [-d dir] [-i path] [-t tag] -- Clone latested_
↪ tagged ${PROJ} version and build it. Optionally select the build directory and_
↪ version, install binaries and cleanup setup files.
27
28 where:
29     -h          show this help text
30     -c          cleanup project
31     -d dir      build files in \"dir\" (default: ${BUILDFOLDER})
32     -i path     install binaries to path (use \"default\" to use default path)
33     -t tag      specify version (git tag or commit hash) to pull (default: Latest tag)
↪ "
34
35
36 while getopts ':hi:cd:t:' OPTION; do
37     case $OPTION in
38         i)  INSTALL=true
39             # Adjust configure string
40             if [ "$OPTARG" != 'default' ]; then
41                 CONFIGURE_STRING="${CONFIGURE_STRING} // /usr/local/" "$OPTARG" ;;

```

(continues on next page)

(continued from previous page)

```

42         fi
43         # INSTALL_PREFIX="$OPTARG"
44         echo "-i set: Installing built binaries to $OPTARG"
45         ;;
46     esac
47 done

48

49 OPTIND=1

50

51 while getopts ':hi:cd:t:' OPTION; do
52     case "$OPTION" in
53         h) echo "$USAGE"
54             exit
55             ;;
56         c) if [ $INSTALL = false ]; then
57                 >&2 echo -e "${RED}ERROR: -c only makes sense if the built binaries were installed before (-i)""
58             exit 1
59         fi
60         CLEANUP=true
61         echo "-c set: Removing build directory"
62         ;;
63         d) echo "-d set: Using folder $OPTARG"
64         BUILD FOLDER="$OPTARG"
65         ;;
66         t) echo "-t set: Using version $OPTARG"
67         TAG="$OPTARG"
68         ;;
69         :) echo -e "${RED}ERROR: missing argument for -$OPTARG\n${NC}" >&2
70         echo "$USAGE" >&2
71         exit 1
72         ;;
73         \?) echo -e "${RED}ERROR: illegal option: -$OPTARG\n${NC}" >&2
74         echo "$USAGE" >&2
75         exit 1
76         ;;
77     esac
78 done

79 shift "$((OPTIND - 1))"

80 # exit when any command fails
81 set -e

82

83 # require sudo
84 if [[ $UID != 0 ]]; then
85     echo -e "${RED}Please run this script with sudo:"
86     echo "sudo $0 $*"
87     exit 1
88 fi

89

90 # cleanup files if the programm was shutdown unexpectedly
91 trap 'echo -e "${RED}ERROR: Script was terminated unexpectedly, cleaning up files..."'
92     && pushd -0 > /dev/null && rm -rf $BUILD FOLDER' INT TERM

93

94 # load shared functions
95 source $LIBRARY

```

(continues on next page)

(continued from previous page)

```

97
98 # fetch specified version
99 if [ ! -d $BUILDFOLDER ]; then
100     mkdir $BUILDFOLDER
101 fi
102
103 pushd $BUILDFOLDER > /dev/null
104
105 if [ ! -d "${PROJ%%/*}" ]; then
106     git clone --recursive "$REPO" "${PROJ%%/*}"
107 fi
108
109 pushd $PROJ > /dev/null
110 select_and_get_project_version "$TAG" "COMMIT_HASH"
111 # build and install if wanted
112 ./bootstrap
113 ./configure $CONFIGURE_STRING
114
115 make -j$(nproc)
116
117 if [ $INSTALL = true ]; then
118     make install
119 fi
120
121 # return to first folder and store version
122 pushd -0 > /dev/null
123 echo "${PROJ##*/}: $COMMIT_HASH" >> "$VERSIONFILE"
124
125 # cleanup if wanted
126 if [ $CLEANUP = true ]; then
127     rm -rf $BUILDFOLDER
128 fi

```

## 6.4.2 install\_openocd\_vexriscv\_essentials.sh

```

1 #!/bin/bash
2
3 # Author: Harald Heckmann <mail@haraldheckmann.de>
4 # Date: Jun. 25 2020
5 # Project: QuantumRisc (RheinMain University) <Steffen.Reith@hs-rm.de>
6
7 # require sudo
8 if [[ $UID != 0 ]]; then
9     echo "Please run this script with sudo:"
10    echo "sudo $0 $*"
11    exit 1
12 fi
13
14 # exit when any command fails
15 set -e
16
17 # required tools
18 TOOLS="build-essential git gcc make libtool pkg-config autoconf automake \
19         texinfo libftdi-dev libusb-1.0-0-dev libyaml-dev"
20

```

(continues on next page)

(continued from previous page)

```

21 # install and upgrade tools
22 apt-get update
23 apt-get install -y $TOOLS
24 apt-get install --only-upgrade -y $TOOLS

```

## 6.5 ujprog

### 6.5.1 install\_ujprog.sh

```

1 #!/bin/bash
2
3 # Author: Harald Heckmann <mail@haraldheckmann.de>
4 # Date: Jun. 25 2020
5 # Project: QuantumRisc (RheinMain University) <Steffen.Reith@hs-rm.de>
6
7 # constants
8 RED='\033[1;31m'
9 NC='\033[0m'
10 LIBRARY="../libraries/library.sh"
11 REPO="https://github.com/f32c/tools.git"
12 PROJ="tools/ujprog"
13 BUILDFOLDER="build_and_install_ujprog"
14 VERSIONFILE="installed_version.txt"
15 TAG="latest"
16 INSTALL=false
17 INSTALL_PREFIX="default"
18 CLEANUP=false
19
20
21 # parse arguments
22 USAGE="$ (basename "$0") [-h] [-c] [-d dir] [-i path] [-t tag] -- Clone latested_
23 ↪tagged ${PROJ} version and build it. Optionally select the build directory and_
24 ↪version, install binaries and cleanup setup files.
25
26 where:
27   -h          show this help text
28   -c          cleanup project
29   -d dir      build files in \"dir\" (default: ${BUILDFOLDER})
30   -i path     install binaries to path (use \"default\" to use default path)
31   -t tag      specify version (git tag or commit hash) to pull (default: Latest tag)
32
33 while getopts ':hi:cd:t:' OPTION; do
34   case $OPTION in
35     i)  INSTALL=true
36         INSTALL_PREFIX="$OPTARG"
37         echo "-i set: Installing built binaries to $INSTALL_PREFIX"
38         ;;
39   esac
40 done
41 OPTIND=1

```

(continues on next page)

(continued from previous page)

```

42
43 while getopts ':hi:cd:t:' OPTION; do
44     case "$OPTION" in
45         h) echo "$USAGE"
46             exit
47             ;;
48         c) if [ $INSTALL = false ]; then
49             >&2 echo -e "${RED}ERROR: -c only makes sense if the built binaries were installed before (-i)" 1>&2
50             exit 1
51         fi
52         CLEANUP=true
53         echo "-c set: Removing build directory"
54         ;;
55         d) echo "-d set: Using folder $OPTARG"
56         BUILDFOLDER="$OPTARG"
57         ;;
58         t) echo "-t set: Using version $OPTARG"
59         TAG="$OPTARG"
60         ;;
61         :) echo -e "${RED}ERROR: missing argument for -$OPTARG\n${NC}" >&2
62         echo "$USAGE" >&2
63         exit 1
64         ;;
65         \?) echo -e "${RED}ERROR: illegal option: -$OPTARG\n${NC}" >&2
66         echo "$USAGE" >&2
67         exit 1
68         ;;
69     esac
70 done
71
72 shift "$((OPTIND - 1))"
73
74 # exit when any command fails
75 set -e
76
77 # require sudo
78 if [[ $UID != 0 ]]; then
79     echo -e "${RED}Please run this script with sudo:"
80     echo "sudo $0 $*"
81     exit 1
82 fi
83
84 # cleanup files if the programm was shutdown unexpectedly
85 trap 'echo -e "${RED}ERROR: Script was terminated unexpectedly, cleaning up files...\n${NC}" && pushd -0 > /dev/null && rm -rf $BUILDFOLDER' INT TERM
86
87 # load shared functions
88 source $LIBRARY
89
90 # fetch specified version
91 if [ ! -d $BUILDFOLDER ]; then
92     mkdir $BUILDFOLDER
93 fi
94
95 pushd $BUILDFOLDER > /dev/null
96

```

(continues on next page)

(continued from previous page)

```

97 if [ ! -d "$PROJ" ]; then
98     git clone --recursive "$REPO" "${PROJ##*/}"
99 fi
100
101 pushd $PROJ > /dev/null
102 select_and_get_project_version "$TAG" "COMMIT_HASH"
103
104 # build and install if wanted
105 cp Makefile.linux Makefile
106
107 # Adjust path if required
108 if [ "$INSTALL_PREFIX" != "default" ]; then
109     mkdir -p "${INSTALL_PREFIX}/bin"
110     sed -i "s /usr/local ${INSTALL_PREFIX} g" Makefile
111 fi
112
113 make -j$(nproc)
114
115 if [ $INSTALL = true ]; then
116     make install
117 fi
118
119 # return to first folder and store version
120 pushd -0 > /dev/null
121 echo "${PROJ##*/}: $COMMIT_HASH" >> "$VERSIONFILE"
122
123 # cleanup if wanted
124 if [ $CLEANUP = true ]; then
125     rm -rf $BUILDFOLDER
126 fi

```

## 6.5.2 install\_ujprog\_essentials.sh

```

1 #!/bin/bash
2
3 # Author: Harald Heckmann <mail@haraldheckmann.de>
4 # Date: Jun. 25 2020
5 # Project: QuantumRisc (RheinMain University) <Steffen.Reith@hs-rm.de>
6
7 # require sudo
8 if [[ $UID != 0 ]]; then
9     echo "Please run this script with sudo:"
10    echo "sudo $0 $*"
11    exit 1
12 fi
13
14 # exit when any command fails
15 set -e
16
17 # required tools
18 TOOLS="libftdi-dev libftdi1-dev libusb-dev build-essential clang make"
19
20 # install and upgrade tools
21 apt-get update
22 apt-get install -y $TOOLS

```

(continues on next page)

(continued from previous page)

```
23 apt-get install --only-upgrade -y $TOOLS
```

## 6.6 openocd

### 6.6.1 install\_openocd\_essentials.sh

```
1 #!/bin/bash
2
3 # Author: Harald Heckmann <mail@haraldheckmann.de>
4 # Date: Jun. 25 2020
5 # Project: QuantumRisc (RheinMain University) <Steffen.Reith@hs-rm.de>
6
7 # require sudo
8 if [[ $UID != 0 ]]; then
9     echo "Please run this script with sudo:"
10    echo "sudo $0 $*"
11    exit 1
12 fi
13
14 # exit when any command fails
15 set -e
16
17 # required tools
18 TOOLS="build-essential git gcc make libtool pkg-config autoconf automake \
19      texinfo libftdi-dev libusb-1.0-0-dev"
20
21 # install and upgrade tools
22 apt-get update
23 apt-get install -y $TOOLS
24 apt-get install --only-upgrade -y $TOOLS
```

### 6.6.2 install\_openocd.sh

```
1 #!/bin/bash
2
3 # Author: Harald Heckmann <mail@haraldheckmann.de>
4 # Date: Jun. 25 2020
5 # Project: QuantumRisc (RheinMain University) <Steffen.Reith@hs-rm.de>
6
7 # constants
8 RED='\033[1;31m'
9 NC='\033[0m'
10 LIBRARY="..libraries/library.sh"
11 REPO="https://git.code.sf.net/p/openocd/code"
12 PROJ="openocd"
13 BUILDFOLDER="build_and_install_openocd"
14 VERSIONFILE="installed_version.txt"
15 TAG="latest"
16 INSTALL=false
17 INSTALL_PREFIX="default"
18 CLEANUP=false
```

(continues on next page)

(continued from previous page)

```

19
20
21 # parse arguments
22 USAGE="${basename "$0"} [-h] [-c] [-d dir] [-i path] [-t tag] -- Clone latested_
23 ↪tagged ${PROJ} version and build it. Optionally select the build directory and_
24 ↪version, install binaries and cleanup setup files.
25
26 where:
27     -h      show this help text
28     -c      cleanup project
29     -d dir   build files in \"dir\" (default: ${BUILDFOLDER})
30     -i path   install binaries to path (use \"default\" to use default path)
31     -t tag    specify version (git tag or commit hash) to pull (default: Latest tag)
32
33 while getopts ':hi:cd:t:' OPTION; do
34     case $OPTION in
35         i) INSTALL=true
36             INSTALL_PREFIX="$OPTARG"
37             echo "-i set: Installing built binaries to $INSTALL_PREFIX"
38             ;;
39     esac
40 done
41 OPTIND=1
42
43 while getopts ':hi:cd:t:' OPTION; do
44     case "$OPTION" in
45         h) echo "$USAGE"
46             exit
47             ;;
48         c) if [ $INSTALL = false ]; then
49             >&2 echo -e "${RED}ERROR: -c only makes sense if the built binaries_
50 ↪were installed before (-i)""
51             exit 1
52         fi
53         CLEANUP=true
54         echo "-c set: Removing build directory"
55         ;;
56         d) echo "-d set: Using folder $OPTARG"
57             BUILDFOLDER="$OPTARG"
58             ;;
59         t) echo "-t set: Using version $OPTARG"
60             TAG="$OPTARG"
61             ;;
62         :) echo -e "${RED}ERROR: missing argument for -$OPTARG\n${NC}" >&2
63         echo "$USAGE" >&2
64         exit 1
65         ;;
66         \?) echo -e "${RED}ERROR: illegal option: -$OPTARG\n${NC}" >&2
67         echo "$USAGE" >&2
68         exit 1
69         ;;
70     esac
71 done

```

(continues on next page)

(continued from previous page)

```

72 shift "$((OPTIND - 1))"

73

74 # exit when any command fails
75 set -e

76

77 # require sudo
78 if [[ $UID != 0 ]]; then
79     echo -e "${RED}Please run this script with sudo:"
80     echo "sudo $0 $*"
81     exit 1
82 fi

83

84 # cleanup files if the programm was shutdown unexpectedly
85 trap 'echo -e "${RED}ERROR: Script was terminated unexpectedly, cleaning up files..."' INT TERM
86             ↪&& pushd -0 > /dev/null && rm -rf $BUILDFOLDER' INT TERM

87 # load shared functions
88 source $LIBRARY

89

90 # fetch specified version
91 if [ ! -d $BUILDFOLDER ]; then
92     mkdir $BUILDFOLDER
93 fi

94

95 pushd $BUILDFOLDER > /dev/null

96

97 if [ ! -d "$PROJ" ]; then
98     git clone --recursive "$REPO" "${PROJ%%/*}"
99 fi

100

101 pushd $PROJ > /dev/null

102

103 # avoid version 10.0 when stable is selected (majorly outdated)
104 if [ "$TAG" == "stable" ]; then
105     TAGLIST=`git rev-list --tags --max-count=1` 

106

107     if [ -n "$TAGLIST" ] && [ `git describe --tags $TAGLIST` == "v0.10.0" ]; then
108         git checkout --recurse-submodules $(git symbolic-ref refs/remotes/origin/HEAD_
109             ↪| sed 's@^refs/remotes/origin/@@')
110         COMMIT_HASH=$(git rev-parse HEAD)
111         >&2 echo -e "${RED}WARNING: No git tags found, using default branch${NC}"
112     else
113         select_and_get_project_version "$TAG" "COMMIT_HASH"
114     fi
115 else
116     select_and_get_project_version "$TAG" "COMMIT_HASH"
117 fi

118 echo "selected TAG: $COMMIT_HASH"

119

120 # build and install if wanted
121 ./bootstrap

122

123 if [ "$INSTALL_PREFIX" == "default" ]; then
124     ./configure
125 else
126     ./configure --prefix="$INSTALL_PREFIX"

```

(continues on next page)

(continued from previous page)

```

127 fi
128
129 make -j${nproc}
130
131 if [ $INSTALL = true ]; then
132     make install
133 fi
134
135 # return to first folder and store version
136 pushd -0 > /dev/null
137 echo "${PROJ##*/}: $COMMIT_HASH" >> "$VERSIONFILE"
138
139 # cleanup if wanted
140 if [ $CLEANUP = true ]; then
141     rm -rf $BUILDFOLDER
142 fi

```

## 6.7 libraries

### 6.7.1 library.sh

```

1 #!/bin/bash
2
3 # Author: Harald Heckmann <mail@haraldheckmann.de>
4 # Date: Oct. 22 2020
5 # Project: QuantumRisc (RheinMain University) <Steffen.Reith@hs-rm.de>
6
7 # This file contains functions that are shared by the build and install scripts.
8
9 # constants
10 RED='\033[1;31m'
11 NC='\033[0m'
12
13 # This function does checkout the correct version and return the commit hash or tag_
14 # name
15 # Parameter 1: Branch name, commit hash, tag or one of the special keywords default/
16 # latest/stable
17 # Parameter 2: Return variable name (commit hash or tag name)
18 # Parameter 3: (OPTIONAL) glob string filter for stable tag list
19 function select_and_get_project_version {
20     # Stable selected: Choose latest tag if available, otherwise use default branch
21     if [ "$1" == "stable" ]; then
22         if [ -n "$3" ]; then
23             local L_TAGLIST=`git rev-list --tags="$3" --max-count=1`  

24         else
25             local L_TAGLIST=`git rev-list --tags --max-count=1`  

26         fi
27
28         # tags found?
29         if [ -n "$L_TAGLIST" ]; then
30             local L_COMMIT_HASH=`git describe --tags $L_TAGLIST`  

31             git checkout --recurse-submodules "$L_COMMIT_HASH"  

32             return 0

```

(continues on next page)

(continued from previous page)

```

31      else
32          git checkout --recurse-submodules $(git symbolic-ref refs/remotes/origin/
33          ↵HEAD | sed 's@^refs/remotes/origin/@@')
34          local L_COMMIT_HASH=$(git rev-parse HEAD)
35          >&2 echo -e "${RED}WARNING: No git tags found, using default branch${NC}"
36      fi
37  else
38      # Either checkout default/stable branch or use custom commit hash, tag or
39      ↵branch name
40      if [ "$1" == 'default' ] || [ "$1" == 'latest' ]; then
41          git checkout --recurse-submodules $(git symbolic-ref refs/remotes/origin/
42          ↵HEAD | sed 's@^refs/remotes/origin/@@')
43      else
44          # Check if $1 contains a valid tag and use it as the version if it does
45          git checkout --recurse-submodules "$1"
46      fi
47
48      local L_COMMIT_HASH=$(git rev-parse HEAD)
49
50  fi
51
52      # Set return value to tag name if available
53  local L_POSSIBLE_TAGS=`git tag --points-at $L_COMMIT_HASH`
54
55  if [ -n "$L_POSSIBLE_TAGS" ] && [ "$L_POSSIBLE_TAGS" != "nightly" ]; then
56      L_COMMIT_HASH="${L_POSSIBLE_TAGS%%[$'\n']*}"
57  fi
58
59      # Apply return value
60  eval "$2=\\"$L_COMMIT_HASH\\\""
61
62}
63
64# Prints only if verbose is set
65function echo_verbose {
66    if [ $VERBOSE = true ]; then
67        echo "$1"
68    fi
69}
70
71# Prints only errors from executed commands if verbose is set
72# Parameter $1: Command to execute
73# Parameter $2: Path to error file
74function exec_verbose {
75    if [ $VERBOSE = false ]; then
76        $1 > /dev/null 2>> "$2"
77    else
78        $1 2>> "$2"
79    fi
80}
81
82# Read latest executed tool/project/etc.
83# Parameter $1: tool/project/etc. list
84# Parameter $2: success file
85# Parameter $3: string containing list element type (tool/project/etc.)
86# Parameter $4: Return variable name
87function get_latest {
88    if [ ! -f "$2" ]; then
89        return 0
90    fi
91}

```

(continues on next page)

(continued from previous page)

```

85   fi
86
87   local LATEST_SCRIPT=`cat $2`
88   local SCRIPTS_ADAPTED=`echo "$1" | sed "s/.*/ ${LATEST_SCRIPT} //"`
89
90   if [ "$SCRIPTS_ADAPTED" == "$1" ]; then
91     local AT_END=true
92     echo -e "\nThe script detected a checkpoint after the last ${3}. This means_
93     ↪that all ${3}s already have been checked and installed if configured that way. Do_
94     ↪you want to check every ${3} and install them again if configured that way (y/n)?"
95     else
96       local AT_END=false
97       echo -e "\nThe script detected a checkpoint. Do you want to install every ${3}_
98     ↪from the checkpoint onwards (y) if configured that way or do you want to start_
99     ↪over from the beginning (n)?"
100    echo "${3}s yet to be check for installation after the checkpoint: ${SCRIPTS_"
101    ↪ADAPTED}"
102    fi
103
104    local DECISION="z"
105
106    while [ $DECISION != "n" ] && [ $DECISION != "y" ]; do
107      read -p "Decision(y/n): " DECISION
108
109      if [ -z $DECISION ]; then
110        DECISION="z"
111      fi
112    done
113
114    echo -e "\n"
115
116    if [ $DECISION == "n" ]; then
117      if [ $AT_END = true ]; then
118        eval "$4=\\"\\\""
119      fi
120    else
121      eval "$4=\\"$SCRIPTS_ADAPTED\\\""
122    fi
123  }
124
125  # Process riscv_gnu_toolchain script parameters
126  # Parameter $1: Script name
127  # Parameter $2: Variable to store the parameters in
128  function parameters_tool_riscv {
129    # set -n flag
130    if [ "${1:6}" == "NEWLIB" ]; then
131      eval "$2=\\"${!2} -n\\\""
132    fi
133
134    # Set "e" parameter
135    if [ "$(eval "echo \$`echo $1`_EXTEND_PATH")" = true ]; then
136      eval "$2=\\"${!2} -e\\\""
137    fi
138
139    # Set "v" parameter
140    if [ "$(eval "echo \$`echo $1`_VECTOR")" = true ]; then
141      eval "$2=\\"${!2} -v\\\""
142    fi
143
144  }

```

(continues on next page)

(continued from previous page)

```

137   fi
138
139
140   # set "u" parameter
141   local L_BUILD_USER=$(eval "echo \$`echo \$1`_USER")"
142
143   if [ -n "$L_BUILD_USER" ] && [ "$L_BUILD_USER" != "default" ]; then
144     eval "$2=\${!2} -u \\"$L_BUILD_USER\\\""
145   fi
146
147   # set "p" parameter
148   local L_BUILD_INSTALL_PATH=$(eval "echo \$`echo \$1`_INSTALL_PATH")"
149
150   if [ -n "$L_BUILD_INSTALL_PATH" ] && [ "$L_BUILD_INSTALL_PATH" != "default" ]; then
151     eval "$2=\${!2} -p \\"$L_BUILD_INSTALL_PATH\\\""
152   fi
153 }
154
155 # Process nextpnr script parameters
156 # Parameter $1: Script name
157 # Parameter $2: Variable to store the parameters in
158 function parameters_tool_nextpnr {
159   # set -e flag
160   if [ "${1:8}" == "ECP5" ]; then
161     eval "$2=\${!2} -e\""
162   fi
163
164   local L_BUILD_CHIPDB=$(eval "echo \$`echo \$1`_CHIPDB_PATH")"
165
166   if [ -n "$L_BUILD_CHIPDB" ] && [ "$L_BUILD_CHIPDB" != "default" ]; then
167     eval "$2=\${!2} -l \\"$L_BUILD_CHIPDB\\\""
168   fi
169 }
170
171 # Process ghdl script parameters
172 # Parameter $1: Script name
173 # Parameter $2: Variable to store the parameters in
174 function parameters_tool_ghdl {
175   # Set "g" flag
176   if [ "$(eval "echo \$`echo \$1`_GCC")" = true ]; then
177     eval "$2=\${!2} -g\""
178   fi
179
180   # Set "l" flag
181   if [ "$(eval "echo \$`echo \$1`_LLVM")" = true ]; then
182     eval "$2=\${!2} -l\""
183   fi
184
185   # Set "m" flag
186   if [ "$(eval "echo \$`echo \$1`_MCODE")" = true ]; then
187     eval "$2=\${!2} -m\""
188   fi
189
190   # Set "y" flag
191   if [ "$(eval "echo \$`echo \$1`_YOSYS")" = true ]; then
192     eval "$2=\${!2} -y\""

```

(continues on next page)

(continued from previous page)

```

193   fi
194 }
195
196 # Process common script parameters
197 # Parameter $1: Script name
198 # Parameter $2: Variable to store the parameters in
199 function parameters_tool {
200   # Set "i" parameter
201   if [ "$(eval echo \$`echo \$1`_INSTALL)" = true ]; then
202     eval "$2=\${!2} -i $(eval echo \$`echo \$1`_INSTALL_PATH)\""
203   fi
204
205   # Set "c" parameter
206   if [ "$(eval echo \$`echo \$1`_CLEANUP)" = true ]; then
207     eval "$2=\${!2} -c\""
208   fi
209
210   # Set "d" parameter
211   local L_BUILD_DIR="$(eval echo \$`echo \$1`_DIR)"
212
213   if [ -n "$L_BUILD_DIR" ] && [ "$L_BUILD_DIR" != "default" ]; then
214     eval "$2=\${!2} -d \"\$L_BUILD_DIR\"\""
215   fi
216
217   # Set "t" parameter
218   local L_BUILD_TAG="$(eval echo \$`echo \$1`_TAG)"
219
220   if [ -n "$L_BUILD_TAG" ] && [ "$L_BUILD_TAG" != "default" ]; then
221     eval "$2=\${!2} -t \"\$L_BUILD_TAG\"\""
222   fi
223
224   # Set "b" for Yosys only
225   if [ $1 == "YOSYS" ]; then
226     local L_BUILD_COMPILER="$(eval echo \$`echo \$1`_COMPILER)"
227
228     if [ -n "$L_BUILD_COMPILER" ]; then
229       eval "$2=\${!2} -b \"\$L_BUILD_COMPILER\"\""
230     fi
231   fi
232
233   # Append special parameters
234   if [ "${1::5}" == "RISCV" ]; then
235     parameters_tool_riscv "$1" "$2"
236   elif [ "${1::7}" == "NEXTPNR" ]; then
237     parameters_tool_nextpnr "$1" "$2"
238   elif [ "$1" == "GHDL" ]; then
239     parameters_tool_ghdl "$1" "$2"
240   fi
241 }
242
243 # Copies the project to documents and creates a symbolic link if desired
244 # Parameter $1: Project name
245 # Parameter $2: User name
246 # Parameter $3: Create symbolic link (bool)
247 # Parameter $4: Project directory (where to copy it)
248 function install_project_for_user {
249   xdg-user-dirs-update

```

(continues on next page)

(continued from previous page)

```

250 local L_PROJECT="$1"
251 local L_USER="$2"
252
253 # User not found (link to desktop impossible)
254 if [ $3 = true ] || [ "$4" == "default" ]; then
255     if ! runuser -l $L_USER -c "xdg-user-dir"; then
256         echo -e "${RED}ERROR: User ${L_USER} does not exist.${NC}"
257         return
258     fi
259 fi
260
261 # Lookup Documents and Desktop and create if not existant
262 if [ "$4" == "default" ]; then
263     local L_DESTINATION=`runuser -l $L_USER -c "xdg-user-dir DOCUMENTS"`
264 else
265     # Strip last possible "/" path
266     if [ "${4:-1}" == "/" ]; then
267         local L_DESTINATION="${4:: -1}"
268     else
269         local L_DESTINATION="$4"
270     fi
271 fi
272
273 # Copy project
274 mkdir -p "$L_DESTINATION"
275 cp -r "$L_PROJECT" "$L_DESTINATION"
276 chown -R "${L_USER}:${L_USER}" "$L_DESTINATION"
277
278 # Create symbolic link to desktop if desired
279 if $3; then
280     local L_USER_DESKTOP=`runuser -l $L_USER -c "xdg-user-dir DESKTOP"`
281     ln -s "${L_DESTINATION}/${L_PROJECT}" "${L_USER_DESKTOP}/${L_PROJECT}"
282 fi
283 }
284
285 # Install project ("configure projects" section in config.cfg)
286 # Parameter $1: Project name
287 function install_project {
288     if [ "${!1}" = false ]; then
289         return 0
290     fi
291
292     local L_NAME_LOWER=`echo "$1" | tr [A-Z] [a-z]`
293
294     # Clone
295     if [ ! -d "$L_NAME_LOWER" ]; then
296         exec_verbose "git clone --recurse-submodules ""$(eval "echo \$`echo $1`_URL")"
297         ↪ "" "$L_NAME_LOWER"" ""$ERROR_FILE"
298     fi
299
300     # Checkout specified version
301     local L_TAG="$(eval "echo \$`echo $1`_TAG")"
302
303     if [ "$L_TAG" != "default" ]; then
304         pushd $L_NAME_LOWER > /dev/null
305         exec_verbose "select_and_get_project_version ""$L_TAG"" ""L_COMMIT_HASH"" "
306         ↪$ERROR_FILE"

```

(continues on next page)

(continued from previous page)

```

305     popd > /dev/null
306 fi
307
308 local L_LINK=$(eval "echo \$`echo \$1`_LINK_TO_DESKTOP")"
309 # Get users to install the projects for
310 local L_USERLIST=$(eval "echo \$`echo \$1`_USER")"
311 # Get project install location
312 local L_INST_LOC=$(eval "echo \$`echo \$1`_LOCATION")"
313
314 if [ "$L_USERLIST" == "default" ]; then
315     for L_USER in `who | cut -d' ' -f1`; do
316         install_project_for_user "$L_NAME_LOWER" "$L_USER" $L_LINK "$L_INST_LOC"
317     done
318 else
319     for L_USER in "$L_USERLIST"; do
320         install_project_for_user "$L_NAME_LOWER" "$L_USER" $L_LINK "$L_INST_LOC"
321     done
322 fi
323
324 rm -rf "$L_NAME_LOWER"
325 }
326
327 # Moves script folder into build folder and returns script path
328 # Parameter $1: Script name
329 # Parameter $2: Variable to store the script path for requirements script in
330 # Parameter $3: Variable to store the script path for installation script in
331 function find_script {
332     if [ "${SCRIPT:::5}" == "RISCV" ]; then
333         cp -r ../riscv_tools .
334         eval "$2=\$(pwd -P)/riscv_tools/install_riscv_essentials.sh\""
335         eval "$3=\$(pwd -P)/riscv_tools/install_riscv.sh\""
336         cp "\$(pwd -P)/riscv_tools/versions.cfg" .
337     elif [ "${SCRIPT:::7}" == "NEXTPNR" ]; then
338         cp -r ../nextpnr .
339         eval "$2=\$(pwd -P)/nextpnr/install_nextpnr_essentials.sh\""
340         eval "$3=\$(pwd -P)/nextpnr/install_nextpnr.sh\""
341     else
342         local L_NAME_LOWER=`echo "$1" | tr [A-Z] [a-z]`'
343         cp -r ../${L_NAME_LOWER} .
344         eval "$2=\$(pwd -P)}/${L_NAME_LOWER}/install_${L_NAME_LOWER}_essentials.sh\""
345         eval "$3=\$(pwd -P)}/${L_NAME_LOWER}/install_${L_NAME_LOWER}.sh\""
346         local L_CFG_FILES=`find "\$(pwd -P)}/${L_NAME_LOWER}" -iname "*.cfg"``'
347
348         for CFG_FILE in $L_CFG_FILES; do
349             cp "$CFG_FILE" .
350         done
351     fi
352 }
353
354 # Copies version file $1 to the desktop of the users specified in $2
355 # Parameter $1: Version file path
356 # Parameter $2: User list
357 function copy_version_file {
358     if [ ! -f "$1" ]; then
359         echo -e "${RED}ERROR: File ${1} does not exist.${NC}"
360         return
361     fi

```

(continues on next page)

(continued from previous page)

```

362
363     xdg-user-dirs-update
364
365     for L_USER in $2; do
366         if ! local L_DESKTOP=`runuser -l $L_USER -c "xdg-user-dir DESKTOP"`;
367             then echo -e "${RED}ERROR: User ${L_USER} does not exist.${NC}"
368             continue
369         fi
370
371         cp "$1" "$L_DESKTOP"
372     done
373 }
```

## 6.8 verilator

### 6.8.1 install\_verilator\_essentials.sh

```

1 #!/bin/bash
2
3 # Author: Harald Heckmann <mail@haraldheckmann.de>
4 # Date: Jun. 25 2020
5 # Project: QuantumRisc (RheinMain University) <Steffen.Reith@hs-rm.de>
6
7 # require sudo
8 if [[ $UID != 0 ]]; then
9     echo "Please run this script with sudo:"
10    echo "sudo $0 $*"
11    exit 1
12 fi
13
14 # exit when any command fails
15 set -e
16
17 # required tools
18 TOOLS="git perl python3 make g++ libfl2 libfl-dev zlib zlib1g zlib1g-dev \
19           ccache libgoogle-perf-tools-dev numactl git autoconf flex bison"
20
21 # install and upgrade tools
22 apt-get update
23 apt-get install -y $TOOLS
24 apt-get install --only-upgrade -y $TOOLS
```

### 6.8.2 install\_verilator.sh

```

1 #!/bin/bash
2
3 # Author: Harald Heckmann <mail@haraldheckmann.de>
4 # Date: Jun. 25 2020
5 # Project: QuantumRisc (RheinMain University) <Steffen.Reith@hs-rm.de>
6
7 # constants
```

(continues on next page)

(continued from previous page)

```

8  RED='\033[1;31m'
9  NC='\033[0m'
10 LIBRARY="../libraries/library.sh"
11 REPO="https://github.com/verilator/verilator.git"
12 PROJ="verilator"
13 BUILDFOLDER="build_and_install_verilator"
14 VERSIONFILE="installed_version.txt"
15 TAG="latest"
16 INSTALL=false
17 INSTALL_PREFIX="default"
18 CLEANUP=false
19
20
21 # parse arguments
22 USAGE="$basename \"$0\" [-h] [-c] [-d dir] [-i path] [-t tag] -- Clone latested_
23 ↪tagged ${PROJ} version and build it. Optionally select the build directory and_
24 ↪version, install binaries and cleanup setup files.
25
26 where:
27     -h          show this help text
28     -c          cleanup project
29     -d dir      build files in \"dir\" (default: ${BUILDFOLDER})
30     -i path     install binaries to path (use \"default\" to use default path)
31     -t tag      specify version (git tag or commit hash) to pull (default: Latest tag)
32
33 while getopts ':hi:cd:t:' OPTION; do
34     case $OPTION in
35         i) INSTALL=true
36             INSTALL_PREFIX="$OPTARG"
37             echo "-i set: Installing built binaries to $INSTALL_PREFIX"
38     ;;
39     esac
40 done
41 OPTIND=1
42
43 while getopts ':hi:cd:t:' OPTION; do
44     case "$OPTION" in
45         h) echo "$USAGE"
46             exit
47     ;;
48         c) if [ $INSTALL = false ]; then
49             >&2 echo -e "${RED}ERROR: -c only makes sense if the built binaries_
50 ↪were installed before (-i)""
51             exit 1
52         fi
53         CLEANUP=true
54         echo "-c set: Removing build directory"
55     ;;
56         d) echo "-d set: Using folder $OPTARG"
57             BUILDFOLDER="$OPTARG"
58     ;;
59         t) echo "-t set: Using version $OPTARG"
60             TAG="$OPTARG"
61     ;;

```

(continues on next page)

(continued from previous page)

```

61      :) echo -e "${RED}ERROR: missing argument for -$OPTARG\n${NC}" >&2
62      echo "$USAGE" >&2
63      exit 1
64      ;;
65  \?) echo -e "${RED}ERROR: illegal option: -$OPTARG\n${NC}" >&2
66      echo "$USAGE" >&2
67      exit 1
68      ;;
69  esac
70 done
71
72 shift "$((OPTIND - 1))"
73
74 # exit when any command fails
75 set -e
76
77 # require sudo
78 if [[ $UID != 0 ]]; then
79   echo -e "${RED}Please run this script with sudo:"
80   echo "sudo $0 $*"
81   exit 1
82 fi
83
84 # cleanup files if the programm was shutdown unexpectedly
85 trap 'echo -e "${RED}ERROR: Script was terminated unexpectedly, cleaning up files..."' \
86       && pushd -0 > /dev/null && rm -rf $BUILDFOLDER' INT TERM
87
88 # load shared functions
89 source $LIBRARY
90
91 # fetch specified version
92 if [ ! -d $BUILDFOLDER ]; then
93   mkdir $BUILDFOLDER
94 fi
95
96 pushd $BUILDFOLDER > /dev/null
97
98 if [ ! -d "$PROJ" ]; then
99   git clone --recursive "$REPO" "${PROJ}%%/*"
100 fi
101
102 pushd $PROJ > /dev/null
103 select_and_get_project_version "$TAG" "COMMIT_HASH"
104
105 # build and install if wanted
106 # unset var
107 if [ -n "$BASH" ]; then
108   unset VERILATOR_ROOT
109 else
110   unsetenv VERILATOR_ROOT
111 fi
112
113 autoconf
114
115 if [ "$INSTALL_PREFIX" == "default" ]; then
116   ./configure
117 else

```

(continues on next page)

(continued from previous page)

```

117   ./configure --prefix="$INSTALL_PREFIX"
118 fi
119
120 make -j$(nproc)
121
122 if [ $INSTALL = true ]; then
123   make install
124 fi
125
126 # return to first folder and store version
127 pushd -0 > /dev/null
128 echo "${PROJ##*/}: $COMMIT_HASH" >> "$VERSIONFILE"
129
130 # cleanup if wanted
131 if [ $CLEANUP = true ]; then
132   rm -rf $BUILDFOLDER
133 fi

```

## 6.9 spinalhdl

### 6.9.1 install\_spinalhdl\_essentials.sh

```

1 #!/bin/bash
2
3 # Author: Harald Heckmann <mail@haraldheckmann.de>
4 # Date: Jun. 23 2020
5 # Project: QuantumRisc (RheinMain University) <Steffen.Reith@hs-rm.de>
6
7 # require sudo
8 if [[ $UID != 0 ]]; then
9   echo "Please run this script with sudo:"
10  echo "sudo $0 $*"
11  exit 1
12 fi
13
14 # exit when any command fails
15 set -e
16
17 # required tools
18 TOOLS="openjdk-8-jdk scala sbt"
19
20 # install and upgrade tools
21 echo "deb https://dl.bintray.com/sbt/debian /" | sudo tee -a /etc/apt/sources.list.d/
22   ↪sbt.list
22 apt-key adv --keyserver hkp://keyserver.ubuntu.com:80 --recv 642AC823
23 apt-get update
24 apt-get install -y $TOOLS
25 apt-get install --only-upgrade -y $TOOLS

```

## 6.10 rust\_riscv

### 6.10.1 install\_rust\_riscv.sh

```
1 #!/bin/bash
2
3 # Author: Harald Heckmann <mail@haraldheckmann.de>
4 # Date: Oct. 24 2020
5 # Project: QuantumRisc (RheinMain University) <Steffen.Reith@hs-rm.de>
6
7 # require sudo
8 if [[ $UID != 0 ]]; then
9     echo "Please run this script with sudo:"
10    echo "sudo $0 $*"
11    exit 1
12 fi
13
14 # exit when any command fails
15 set -e
16
17 BUILDFOLDER="install_rust_riscv"
18 VERSIONFILE="installed_version.txt"
19 LIBRARY="../libraries/library.sh"
20 # required tools
21 TOOLS="curl"
22 # available rust targets
23 DEFAULT_TARGETS="riscv32i-unknown-none-elf riscv32imac-unknown-none-elf \
24 riscv32imc-unknown-none-elf riscv64gc-unknown-linux-gnu \
25 riscv64gc-unknown-none-elf riscv64imac-unknown-none-elf"
26
27 source $LIBRARY
28
29 # install and upgrade tools
30 apt-get update
31 apt-get install -y $TOOLS
32 apt-get install --only-upgrade -y $TOOLS
33
34 # install rust
35 mkdir -p $BUILDFOLDER
36 pushd $BUILDFOLDER > /dev/null
37 curl --proto '=https' --tlsv1.2 -sSf https://sh.rustup.rs > 'rustup_installer.sh'
38 chmod +x './rustup_installer.sh'
39
40 # Install rust in the context of every logged in user
41 for RUST_USER in `who | cut -d' ' -f1`; do
42     # check if rustup is installed
43     RUSTUP_SCRIPT="$PWD/rustup_installer.sh"
44
45     if ! runuser -l $RUST_USER -c "command -v rustup" &> /dev/null; then
46         runuser -l $RUST_USER -c "$RUSTUP_SCRIPT -y"
47         RUSTUP="\$HOME/.cargo/bin/rustup"
48     else
49         RUSTUP=`runuser -l $RUST_USER -c "command -v rustup"`
50     fi
51
52     # update rust
```

(continues on next page)

(continued from previous page)

```

53 runuser -l $RUST_USER -c "$RUSTUP install stable"
54 runuser -l $RUST_USER -c "$RUSTUP install nightly"
55 runuser -l $RUST_USER -c "$RUSTUP update"
56 runuser -l $RUST_USER -c "$RUSTUP update nightly"
57
58 # add riscv target
59 # scan for available targets first, if it fails use DEFAULT_TARGETS
60 PRRT=`runuser -l $RUST_USER -c "$RUSTUP target list | grep riscv"`
61 DEFAULT_TC=`runuser -l $RUST_USER -c "rustup default"`
62 DEFAULT_TC="${DEFAULT_TC// (default) }"
63
64 if [ -n "$PRRT" ]; then
65     DEFAULT_TARGETS=`echo $PRRT`
66 fi
67
68 runuser -l $RUST_USER -c "$RUSTUP target add --toolchain ${DEFAULT_TC} ${DEFAULT_
69 →TARGETS// (installed) }"
70 runuser -l $RUST_USER -c "$RUSTUP target add --toolchain nightly ${DEFAULT_"
71 →TARGETS// (installed) }"
72
73 # add some useful dev components
74 runuser -l $RUST_USER -c "$RUSTUP component add --toolchain ${DEFAULT_TC} rls_
75 →rustfmt rust-analysis clippy"
76 runuser -l $RUST_USER -c "$RUSTUP component add --toolchain nightly rls rustfmt_
77 →rust-analysis clippy"
78 done
79
80 # cleanup
81 popd > /dev/null
82 rm -r $BUILD FOLDER
83
84 VER_DEFAULT=`runuser -l $RUST_USER -c "$RUSTUP run ${DEFAULT_TC} rustc --version"`
85 VER_DEFAULT="${VER_DEFAULT//(*)}"
86 VER_DEFAULT="${VER_DEFAULT#rustc }"
87
88 VER_NIGHTLY=`runuser -l $RUST_USER -c "$RUSTUP run nightly rustc --version"`
89 VER_NIGHTLY="${VER_NIGHTLY//(*)}"
90 VER_NIGHTLY="${VER_NIGHTLY#rustc }"
91 echo -e "rustc (${DEFAULT_TC}, with riscv targets): ${VER_DEFAULT}\nrustc (nightly, "
92 →with riscv targets): ${VER_NIGHTLY}" >> "$VERSIONFILE"
```

## 6.11 gtkterm

### 6.11.1 install\_gtkterm\_essentials.sh

```

1 #!/bin/bash
2
3 # Author: Harald Heckmann <mail@haraldheckmann.de>
4 # Date: Jun. 25 2020
5 # Project: QuantumRisc (RheinMain University) <Steffen.Reith@hs-rm.de>
6
7 # require sudo
8 if [[ $UID != 0 ]]; then
```

(continues on next page)

(continued from previous page)

```

9    echo "Please run this script with sudo:"
10   echo "sudo $0 $*"
11   exit 1
12 fi
13
14 # exit when any command fails
15 set -e
16
17 # required tools
18 TOOLS="libgtk-3-dev libvte-2.91-dev intltool libgudev-1.0 meson ninja-build"
19
20 # install and upgrade tools
21 apt-get update
22 apt-get install -y $TOOLS
23 apt-get install --only-upgrade -y $TOOLS

```

## 6.11.2 install\_gtkterm.sh

```

1 #!/bin/bash
2
3 # Author: Harald Heckmann <mail@haraldheckmann.de>
4 # Date: Oct. 08 2020
5 # Project: QuantumRisc (RheinMain University) <Steffen.Reith@hs-rm.de>
6
7 # constants
8 RED='\033[1;31m'
9 NC='\033[0m'
10 LIBRARY="../libraries/library.sh"
11 REPO="https://github.com/Jeija/gtkterm"
12 PROJ="gtkterm"
13 BUILDFOLDER="build_and_install_gtkterm"
14 VERSIONFILE="installed_version.txt"
15 TAG="latest"
16 INSTALL=false
17 INSTALL_PREFIX="default"
18 CLEANUP=false
19
20
21 # parse arguments
22 USAGE="$ (basename "$0") [-h] [-c] [-d dir] [-i path] [-t tag] -- Clone latested_
23 ↪tagged ${PROJ} version and build it. Optionally select the build directory and_
24 ↪version, install binaries and cleanup setup files.
25
26 where:
27   -h          show this help text
28   -c          cleanup project
29   -d dir      build files in \"dir\" (default: ${BUILDFOLDER})
30   -i path     install binaries to path (use \"default\" to use default path)
31   -t tag      specify version (git tag or commit hash) to pull (default: Latest tag)
32
33 while getopts ':hi:cd:t:' OPTION; do
34   case $OPTION in
35     i)  INSTALL=true

```

(continues on next page)

(continued from previous page)

```

35         INSTALL_PREFIX="$OPTARG"
36         echo "-i set: Installing built binaries to $INSTALL_PREFIX"
37         ;;
38     esac
39 done
40
41 OPTIND=1
42
43 while getopts ':hi:cd:t:' OPTION; do
44     case "$OPTION" in
45         h) echo "$USAGE"
46             exit
47             ;;
48         c) if [ $INSTALL = false ]; then
49             >&2 echo -e "${RED}ERROR: -c only makes sense if the built binaries were installed before (-i)""
50             exit 1
51         fi
52         CLEANUP=true
53         echo "-c set: Removing build directory"
54         ;;
55         d) echo "-d set: Using folder $OPTARG"
56         BUILDFOLDER="$OPTARG"
57         ;;
58         t) echo "-t set: Using version $OPTARG"
59         TAG="$OPTARG"
60         ;;
61         :) echo -e "${RED}ERROR: missing argument for -$OPTARG\n$NC" >&2
62         echo "$USAGE" >&2
63         exit 1
64         ;;
65         \?) echo -e "${RED}ERROR: illegal option: -$OPTARG\n$NC" >&2
66         echo "$USAGE" >&2
67         exit 1
68         ;;
69     esac
70 done
71
72 shift "$((OPTIND - 1))"
73
74 # exit when any command fails
75 set -e
76
77 # require sudo
78 if [[ $UID != 0 ]]; then
79     echo -e "${RED}Please run this script with sudo:"
80     echo "sudo $0 $*"
81     exit 1
82 fi
83
84 # cleanup files if the programm was shutdown unexpectedly
85 trap 'echo -e "${RED}Script was terminated unexpectedly, cleaning up files..."'
86     && pushd -0 > /dev/null && rm -rf $BUILDFOLDER' INT TERM
87
88 # load shared functions
89 source $LIBRARY

```

(continues on next page)

(continued from previous page)

```

90 # fetch specified version
91 if [ ! -d $BUILDFOLDER ]; then
92     mkdir $BUILDFOLDER
93 fi
94
95 pushd $BUILDFOLDER > /dev/null
96
97 if [ ! -d "$PROJ" ]; then
98     git clone --recursive "$REPO" "${PROJ}%%/*}"
99 fi
100
101 pushd $PROJ > /dev/null
102 select_and_get_project_version "$TAG" "COMMIT_HASH"
103
104 if [ "$INSTALL_PREFIX" == "default" ]; then
105     meson build
106 else
107     meson build -Dprefix="$INSTALL_PREFIX"
108 fi
109
110 if [ $INSTALL = true ]; then
111     ninja -C build install
112 else
113     ninja -C build
114 fi
115
116 # return to first folder and store version
117 pushd -0 > /dev/null
118 echo "${PROJ##*/}: $COMMIT_HASH" >> "$VERSIONFILE"
119
120 # cleanup if wanted
121 if [ $CLEANUP = true ]; then
122     rm -rf $BUILDFOLDER
123 fi

```

## 6.12 icarusverilog

### 6.12.1 install\_icarusverilog\_essentials.sh

```

1#!/bin/bash
2
3# Author: Harald Heckmann <mail@haraldheckmann.de>
4# Date: Oct. 24 2020
5# Project: QuantumRisc (RheinMain University) <Steffen.Reith@hs-rm.de>
6
7# require sudo
8if [[ $UID != 0 ]]; then
9    echo "Please run this script with sudo:"
10   echo "sudo $0 $*"
11   exit 1
12fi
13
14# exit when any command fails

```

(continues on next page)

(continued from previous page)

```

15 set -e
16
17 # required tools
18 TOOLS="make g++ autoconf flex bison gperf autoconf libbz2-1.0 libc6 libgcc-s1 \
19     libreadline8 libstdc++6 zlib1g"
20
21 # install and upgrade tools
22 apt-get update
23 apt-get install -y $TOOLS
24 apt-get install --only-upgrade -y $TOOLS

```

## 6.12.2 install\_icarusverilog.sh

```

1#!/bin/bash
2
3# Author: Harald Heckmann <mail@haraldheckmann.de>
4# Date: Oct. 24 2020
5# Project: QuantumRisc (RheinMain University) <Steffen.Reith@hs-rm.de>
6
7# constants
8RED='\033[1;31m'
9NC='\033[0m'
10LIBRARY="../libraries/library.sh"
11REPO="https://github.com/steveicarus/iverilog.git"
12PROJ="iverilog"
13BUILDFOLDER="build_and_install_iverilog"
14VERSIONFILE="installed_version.txt"
15TAG="latest"
16INSTALL=false
17INSTALL_PREFIX="default"
18CLEANUP=false
19
20
21# parse arguments
22USAGE="$ (basename "$0") [-h] [-c] [-d dir] [-i path] [-t tag] -- Clone latested_
23    ↪tagged ${PROJ} version and build it. Optionally select the build directory and_
24    ↪version, install binaries and cleanup setup files.
25
26where:
27    -h          show this help text
28    -c          cleanup project
29    -d dir      build files in \"dir\" (default: ${BUILDFOLDER})
30    -i path     install binaries to path (use \"default\" to use default path)
31    -t tag      specify version (git tag or commit hash) to pull (default: Latest tag)
32
33while getopts ':hi:cd:t:' OPTION; do
34    case $OPTION in
35        i) INSTALL=true
36            INSTALL_PREFIX="$OPTARG"
37            echo "-i set: Installing built binaries to $INSTALL_PREFIX"
38            ;;
39    esac
done

```

(continues on next page)

(continued from previous page)

```

40
41 OPTIND=1
42
43 while getopts ':hi:cd:t:' OPTION; do
44     case "$OPTION" in
45         h) echo "$USAGE"
46             exit
47             ;;
48         c) if [ $INSTALL = false ]; then
49             >&2 echo -e "${RED}ERROR: -c only makes sense if the built binaries were installed before (-i)" 1>&2
50             exit 1
51         fi
52         CLEANUP=true
53         echo "-c set: Removing build directory"
54         ;;
55         d) echo "-d set: Using folder $OPTARG"
56         BUILD FOLDER="$OPTARG"
57         ;;
58         t) echo "-t set: Using version $OPTARG"
59         TAG="$OPTARG"
60         ;;
61         :) echo -e "${RED}ERROR: missing argument for -$OPTARG\n${NC}" >&2
62         echo "$USAGE" >&2
63         exit 1
64         ;;
65         \?) echo -e "${RED}ERROR: illegal option: -$OPTARG\n${NC}" >&2
66         echo "$USAGE" >&2
67         exit 1
68         ;;
69     esac
70 done
71
72 shift "$((OPTIND - 1))"
73
74 # exit when any command fails
75 set -e
76
77 # require sudo
78 if [[ $UID != 0 ]]; then
79     echo -e "${RED}Please run this script with sudo:"
80     echo "sudo $0 $*"
81     exit 1
82 fi
83
84 # cleanup files if the programm was shutdown unexpectedly
85 trap 'echo -e "${RED}ERROR: Script was terminated unexpectedly, cleaning up files...\n${NC}" && pushd -0 > /dev/null && rm -rf $BUILD FOLDER' INT TERM
86
87 # load shared functions
88 source $LIBRARY
89
90 # fetch specified version
91 if [ ! -d $BUILD FOLDER ]; then
92     mkdir $BUILD FOLDER
93 fi

```

(continues on next page)

(continued from previous page)

```

95 pushd $BUILDFOLDER > /dev/null
96
97 if [ ! -d "$PROJ" ]; then
98     git clone --recursive "$REPO" "${PROJ%%/*}"
99 fi
100
101 pushd $PROJ > /dev/null
102 select_and_get_project_version "$TAG" "COMMIT_HASH"
103
104 # build and install if wanted
105 autoconf
106
107 if [ "$INSTALL_PREFIX" == "default" ]; then
108     ./configure
109 else
110     ./configure --prefix="$INSTALL_PREFIX"
111 fi
112
113 make -j$(nproc)
114
115 if [ $INSTALL = true ]; then
116     make install
117 fi
118
119 # return to first folder and store version
120 pushd -0 > /dev/null
121 echo "${PROJ##*/}: $COMMIT_HASH" >> "$VERSIONFILE"
122
123 # cleanup if wanted
124 if [ $CLEANUP = true ]; then
125     rm -rf $BUILDFOLDER
126 fi

```

## 6.13 nextpnr

### 6.13.1 install\_nextpnr\_essentials.sh

```

1 #!/bin/bash
2
3 # Author: Harald Heckmann <mail@haraldheckmann.de>
4 # Date: Jun. 24 2020
5 # Project: QuantumRisc (RheinMain University) <Steffen.Reith@hs-rm.de>
6
7 # require sudo
8 if [[ $UID != 0 ]]; then
9     echo "Please run this script with sudo:"
10    echo "sudo $0 $*"
11    exit 1
12 fi
13
14 # exit when any command fails
15 set -e
16

```

(continues on next page)

(continued from previous page)

```

17 # required tools
18 TOOLS="clang-format qt5-default libboost-dev libboost-filesystem-dev \
19     libboost-thread-dev libboost-program-options-dev libboost-python-dev \
20     libboost-iostreams-dev libboost-dev libeigen3-dev python3-dev cmake"
21
22 # install and upgrade tools
23 apt-get update
24 apt-get install -y $TOOLS
25 apt-get install --only-upgrade -y $TOOLS

```

## 6.13.2 install\_nextpnr.sh

```

1#!/bin/bash
2
3# Author: Harald Heckmann <mail@haraldheckmann.de>
4# Date: Jun. 25 2020
5# Project: QuantumRisc (RheinMain University) <Steffen.Reith@hs-rm.de>
6
7# constants
8RED='\033[1;31m'
9NC='\033[0m'
10LIBRARY="../libraries/library.sh"
11REPO="https://github.com/YosysHQ/nextpnr.git"
12PROJ="nextpnr"
13CHIP="ice40"
14BUILDFOLDER="build_and_install_nextpnr"
15VERSIONFILE="installed_version.txt"
16TAG="latest"
17LIBPATH=""
18INSTALL=false
19CLEANUP=false
20# trellis config
21TRELLIS_LIB="/usr"
22TRELLIS_REPO="https://github.com/SymbiFlow/prjtrellis"
23TRELLIS_PROJ="prjtrellis"
24# icestorm config
25ICESTORM_REPO="https://github.com/cliffordwolf/icestorm.git"
26ICESTORM_PROJ="icestorm"
27ICESTORM_LIB="/usr/local/share/icebox"
28ICESTORM_ICEBOX_DIR="icestorm"
29
30
31# parse arguments
32USAGE="$ (basename "$0") [-h] [-c] [-e] [-d dir] [-i path] [-l path] [-t tag] -- Clone \
33    ↪ latested tagged ${PROJ} version and build it. Optionally select the build directory, \
34    ↪ chip files, chipset and version, install binaries and cleanup setup files.
35
36where:
37    -h      show this help text
38    -c      cleanup project
39    -e      install NextPNR for ecp5 chips (default: ice40)
40    -d dir   build files in \"dir\" (default: ${BUILDFOLDER})
41    -i path   install binaries to path (use \"default\" to use default path)
42    -l path   use local chip files for ice40 or ecp5 from \"path\" (use empty \
43    ↪ string for default path in ubuntu)

```

(continues on next page)

(continued from previous page)

```

41      -t tag      specify version (git tag or commit hash) to pull (default: Latest tag)
42      ↵"
43
44 while getopts ":hecd:i:t:l:" OPTION; do
45   case $OPTION in
46     i) INSTALL=true
47         INSTALL_PREFIX="$OPTARG"
48         echo "-i set: Installing built binaries to $INSTALL_PREFIX"
49         ;;
50     e) echo "-e set: Installing NextPNR for ec5 chipset"
51         CHIP="ec5"
52         ;;
53   esac
54 done
55
56 OPTIND=1
57
58 while getopts ':hecd:i:t:l:' OPTION; do
59   case "$OPTION" in
60     h) echo "$USAGE"
61         exit
62         ;;
63     c) if [ $INSTALL = false ]; then
64         >&2 echo -e "${RED}ERROR: -c only makes sense if the built binaries were_"
65         ↵installed before (-i)"
66         exit 1
67     fi
68     CLEANUP=true
69     echo "-c set: Removing build directory"
70     ;;
71     d) echo "-d set: Using folder $OPTARG"
72         BUILDFOLDER="$OPTARG"
73         ;;
74     t) echo "-t set: Using version $OPTARG"
75         TAG="$OPTARG"
76         ;;
77     l) echo "-l set: Using local chip data"
78     if [ -z "$OPTARG" ]; then
79       if [ "$CHIP" = "ice40" ]; then
80         LIBPATH="$ICESTORM_LIB"
81       else
82         LIBPATH="$TRELLIS_LIB"
83       fi
84     else
85       if [ ! -d "$OPTARG" ]; then
86         echo -e "${RED}ERROR: Invalid path \\"$OPTARG\\\""
87         exit 1
88       fi
89       LIBPATH="$OPTARG"
90     fi
91     ;;
92   :) echo -e "${RED}ERROR: missing argument for -$OPTARG\n$NC" >&2
93   echo "$USAGE" >&2
94   exit 1
95   ;;

```

(continues on next page)

(continued from previous page)

```

96     \?) echo -e "${RED}ERROR: illegal option: -$OPTARG\n${NC}" "$OPTARG" >&2
97     echo "$USAGE" >&2
98     exit 1
99     ;;
100   esac
101 done
102
103 shift "$((OPTIND - 1))"
104
105 # exit when any command fails
106 set -e
107
108 # require sudo
109 if [[ $UID != 0 ]]; then
110   echo -e "${RED}Please run this script with sudo:"
111   echo "sudo $0 $*"
112   exit 1
113 fi
114
115 # cleanup files if the programm was shutdown unexpectedly
116 trap 'echo -e "${RED}ERROR: Script was terminated unexpectedly, cleaning up files..."' \
117       && pushd -0 > /dev/null && rm -rf $BUILDFOLDER' INT TERM
118
119 # load shared functions
120 source $LIBRARY
121
122 # fetch specified version
123 if [ ! -d $BUILDFOLDER ]; then
124   mkdir $BUILDFOLDER
125 fi
126
127 pushd $BUILDFOLDER > /dev/null
128
129 if [ ! -d "$PROJ" ]; then
130   git clone --recursive "$REPO" "${PROJ%/*}"
131 fi
132
133 pushd $PROJ > /dev/null
134
135 select_and_get_project_version "$TAG" "COMMIT_HASH"
136
137 # build and install if wanted
138 # chip ice40?
139 if [ "$CHIP" = "ice40" ]; then
140   # is icestorm installed?
141   if [ -n "$LIBPATH" ]; then
142     if [ "$INSTALL_PREFIX" == "default" ]; then
143       cmake -DARCH=ice40 -DICEBOX_ROOT=${LIBPATH}
144     else
145       cmake -DARCH=ice40 -DICEBOX_ROOT=${LIBPATH} -DCMAKE_INSTALL_PREFIX="$INSTALL_PREFIX".
146     fi
147   else
148     echo "Note: Pulling Icestorm from Github."
149
150   if [ ! -d "$ICESTORM_PROJ" ]; then
151     git clone $ICESTORM_REPO "$ICESTORM_ICEBOX_DIR"

```

(continues on next page)

(continued from previous page)

```

151   fi
152
153   NEXTPNR_FOLDER=`pwd -P`
154   # build icebox (chipdbs)
155   pushd "${ICESTORM_ICEBOX_DIR}/icebox" > /dev/null
156   make -j$(nproc)
157   make install DESTDIR=$NEXTPNR_FOLDER PREFIX=''
158   popd +0 > /dev/null
159   # build icetime (timing)
160   pushd "${ICESTORM_ICEBOX_DIR}/icetime" > /dev/null
161   make -j$(nproc) PREFIX=$NEXTPNR_FOLDER
162   make install DESTDIR=$NEXTPNR_FOLDER PREFIX=''
163   popd +0 > /dev/null
164   # build nextpnr-ice40 next
165
166   if [ "$INSTALL_PREFIX" == "default" ]; then
167     cmake -j$(nproc) -DARCH=ice40 -DICEBOX_ROOT="${NEXTPNR_FOLDER}/share/
168   ↪icebox" .
169   else
170     cmake -j$(nproc) -DARCH=ice40 -DICEBOX_ROOT="${NEXTPNR_FOLDER}/share/
171   ↪icebox" -DCMAKE_INSTALL_PREFIX="$INSTALL_PREFIX" .
172   fi
173 fi
174 # chip ecp5?
175 else
176   # is project trellis installed?
177   if [ -d "$LIBPATH" ]; then
178     if [ "$INSTALL_PREFIX" == "default" ]; then
179       cmake -j$(nproc) -DARCH=ecp5 -DTRELLIS_INSTALL_PREFIX=${LIBPATH} .
180     else
181       cmake -j$(nproc) -DARCH=ecp5 -DTRELLIS_INSTALL_PREFIX=${LIBPATH} -DCMAKE_
182   ↪INSTALL_PREFIX="$INSTALL_PREFIX" .
183     fi
184   else
185     echo "Note: Pulling Trellis from Github."
186
187     if [ ! -d "$TRELLIS_PROJ" ]; then
188       git clone --recursive $TRELLIS_REPO
189     fi
190
191     TRELLIS_MAKE_PATH="$(pwd -P)/${TRELLIS_PROJ}/libtrellis"
192     pushd "$TRELLIS_MAKE_PATH" > /dev/null
193     cmake -j$(nproc) -DCMAKE_INSTALL_PREFIX="$TRELLIS_MAKE_PATH" .
194     make -j$(nproc)
195     make install
196     popd +0 > /dev/null
197
198     if [ "$INSTALL_PREFIX" == "default" ]; then
199       cmake -j$(nproc) -DARCH=ecp5 -DTRELLIS_INSTALL_PREFIX="$TRELLIS_MAKE_PATH"
200   ↪" .
201     else
202       cmake -j$(nproc) -DARCH=ecp5 -DTRELLIS_INSTALL_PREFIX="$TRELLIS_MAKE_PATH"
203   ↪" -DCMAKE_INSTALL_PREFIX="$INSTALL_PREFIX" .
204     fi
205   fi
206 fi

```

(continues on next page)

(continued from previous page)

```

203 make -j$(nproc)
204
205 if [ $INSTALL = true ]; then
206     make install
207 fi
208
209 # return to first folder and store version
210 pushd -0 > /dev/null
211
212 if [ "$CHIP" == "ice40" ]; then
213     echo "${PROJ##*/}-ice40: $COMMIT_HASH" >> "$VERSIONFILE"
214 else
215     echo "${PROJ##*/}-ecp5: $COMMIT_HASH" >> "$VERSIONFILE"
216 fi
217
218 # cleanup if wanted
219 if [ $CLEANUP = true ]; then
220     rm -rf $BUILD FOLDER
221 fi

```

## 6.14 riscv\_tools

### 6.14.1 install\_riscv.sh

```

1 #!/bin/bash
2
3 # Author: Harald Heckmann <mail@haraldheckmann.de>
4 # Date: Jul. 02 2020
5 # Project: QuantumRisc (RheinMain University) <Steffen.Reith@hs-rm.de>
6
7 # constants
8 RED='\033[1;31m'
9 NC='\033[0m'
10 LIBRARY="../libraries/library.sh"
11 REPO="https://github.com/riscv/riscv-gnu-toolchain.git"
12 PROJ="riscv-gnu-toolchain"
13 BUILDFOLDER="build_and_install_riscv_gnu_toolchain"
14 VERSIONFILE="installed_version.txt"
15 TOOLCHAIN_SUFFIX="linux-multilib"
16 TAG="latest"
17 NEWLIB=false
18 # INSTALL=false
19 INSTALL_PATH="/opt/riscv"
20 PROFILE_PATH="/etc/profile"
21 CLEANUP=false
22 EXPORTPATH=false
23 VECTOREXT=false
24 VECTORBRANCH='rvv-intrinsic'
25
26 VERSION_FILE_NAME="versions.cfg"
27 VERSION_FILE='## Define sourcecode branch
# default = use predefined versions from current riscv-gnu-toolchain branch
# or any arbitrary git tag or commit hash
28
29'

```

(continues on next page)

(continued from previous page)

```

30 # note that in most projects there is no master branch
31 QEMU=default
32 RISCV_BINUTILS=default
33 RISCV_DEJAGNU=default
34 RISCV_GCC=default
35 RISCV_GDB=default
36 RISCV_GLIBC=default
37 RISCV_NEWLIB=default
38
39 ## Define which RiscV architectures and ABIs are supported (space seperated list
40 ↪"arch-abi")
41
42 # Taken from Sifive:
43 # https://github.com/sifive/freedom-tools/blob/
44 ↪120fa4d48815fc9e87c59374c499849934f2ce10/Makefile
45 NEWLIB_MULTILIBS_GEN="\
46     rv32e-ilp32e--c \
47     rv32ea-ilp32e--m \
48     rv32em-ilp32e--c \
49     rv32eac-ilp32e-- \
50     rv32emac-ilp32e-- \
51     rv32i-ilp32--c,f,fc,fd,fdc \
52     rv32ia-ilp32-rv32ima,rv32iaf,rv32imaf,rv32iafd,rv32imafd- \
53     rv32im-ilp32--c,f,fc,fd,fdc \
54     rv32iac-ilp32--f,fd \
55     rv32imac-ilp32-rv32imafc,rv32imafdc- \
56     rv32if-ilp32f--c,d,dc \
57     rv32iaf-ilp32f--c,d,dc \
58     rv32imf-ilp32f--d \
59     rv32imaf-ilp32f-rv32imafd- \
60     rv32imfc-ilp32f--d \
61     rv32imafc-ilp32f-rv32imafdc- \
62     rv32ifd-ilp32d--c \
63     rv32imfd-ilp32d--c \
64     rv32iafd-ilp32d-rv32imafd,rv32iafdc- \
65     rv32imafdc-ilp32d-- \
66     rv64i-lp64--c,f,fc,fd,fdc \
67     rv64ia-lp64-rv64ima,rv64iaf,rv64imaf,rv64iafd,rv64imafd- \
68     rv64im-lp64--c,f,fc,fd,fdc \
69     rv64iac-lp64--f,fd \
70     rv64imac-lp64-rv64imafc,rv64imafdc- \
71     rv64if-lp64f--c,d,dc \
72     rv64iaf-lp64f--c,d,dc \
73     rv64imf-lp64f--d \
74     rv64imaf-lp64f-rv64imafd- \
75     rv64imfc-lp64f--d \
76     rv64imafc-lp64f-rv64imafdc- \
77     rv64ifd-lp64d--c \
78     rv64imfd-lp64d--c \
79     rv64iafd-lp64d-rv64imafd,rv64iafdc- \
80     rv64imafdc-lp64d--"
81
82 # Linux install (cross-compile for linux)
83 # Default value from riscv-gcc repository
84 GLIBC_MULTILIBS_GEN="\
85     rv32imac-ilp32-rv32ima,rv32imaf,rv32imafd,rv32imafc,rv32imafdc- \

```

(continues on next page)

(continued from previous page)

```

85    rv32imafdc-ilp32d-rv32imafd- \
86    rv64imac-lp64-rv64ima,rv64imaf,rv64imafd,rv64imafc,rv64imafdc- \
87    rv64imafdc-lp64d-rv64imafd-''
88
89
90 # parse arguments
91 USAGE=$(basename "$0") [-h] [-c] [-n] [-d dir] [-t tag] [-p path] [-u user] -- Clone_ \
92   ↪ latested ${PROJ} version and build it. Optionally select compiler (buildtool), \
93   ↪ build directory and version, install binaries and cleanup setup files.
94
95 where:
96   -h          show this help text
97   -c          cleanup project
98   -e          extend PATH in by RiscV binary path (default: /etc/profile)
99   -n          use \"newlib multilib\" instead of \"linux multilib\" cross-compiler
100  -v          install with experimental vector extensions (uses rvv-intrinsic_ \
101    ↪branch)
102  -d dir      build files in \"dir\" (default: ${BUILDFOLDER})
103  -t tag      specify version (git tag or commit hash) to pull (default: default_ \
104    ↪branch)
105  -p path     choose install path (default: /opt/riscv)
106  -u user     install RiscV tools for user \"user\". (default: install globally)"
107
108 while getopts ':hcenvd:t:u:p:' OPTION; do
109   case "$OPTION" in
110     h) echo "$USAGE"
111     exit
112     ;;
113     c) if [ $INSTALL = false ]; then
114       >&2 echo -e "${RED}ERROR: -c only makes sense if the built binaries_ \
115     ↪were installed before (-i)""
116       exit 1
117     fi
118     CLEANUP=true
119     echo "-c set: Removing build directory"
120     ;;
121     e) EXPORTPATH=true
122     echo "-e set: Extending PATH by RiscV binary path"
123     ;;
124     n) echo "-n set: Using newlib cross-compiler"
125     NEWLIB=true
126     TOOLCHAIN_SUFFIX="newlib-multilib"
127     ;;
128     v) echo "-v set: Installing with experimental vector extensions"
129     VECTOREXT=true
130     ;;
131     d) echo "-d set: Using folder $OPTARG"
132     BUILDFOLDER="$OPTARG"
133     ;;
134     t) echo "-t set: Using version $OPTARG"
135     TAG="$OPTARG"
136     ;;
137     p) echo "-p set: Using install path $OPTARG"
138     INSTALL_PATH="$OPTARG"
139     ;;
140     u) echo "-u set: Installing for user $OPTARG"
141     PROFILE_PATH=$(grep $OPTARG /etc/passwd | cut -d ":" -f6)/.profile"
142
143
144
145
146
147
148
149
150
151
152
153
154
155
156
157
158
159
160
161
162
163
164
165
166
167
168
169
170
171
172
173
174
175
176
177
178
179
180
181
182
183
184
185
186
187
188
189
190
191
192
193
194
195
196
197
198
199
200
201
202
203
204
205
206
207
208
209
210
211
212
213
214
215
216
217
218
219
220
221
222
223
224
225
226
227
228
229
230
231
232
233
234
235
236
237
238
239
240
241
242
243
244
245
246
247
248
249
250
251
252
253
254
255
256
257
258
259
260
261
262
263
264
265
266
267
268
269
270
271
272
273
274
275
276
277
278
279
280
281
282
283
284
285
286
287
288
289
290
291
292
293
294
295
296
297
298
299
300
301
302
303
304
305
306
307
308
309
310
311
312
313
314
315
316
317
318
319
320
321
322
323
324
325
326
327
328
329
330
331
332
333
334
335
336
337
338
339
340
341
342
343
344
345
346
347
348
349
350
351
352
353
354
355
356
357
358
359
360
361
362
363
364
365
366
367
368
369
370
371
372
373
374
375
376
377
378
379
380
381
382
383
384
385
386
387
388
389
390
391
392
393
394
395
396
397
398
399
400
401
402
403
404
405
406
407
408
409
410
411
412
413
414
415
416
417
418
419
420
421
422
423
424
425
426
427
428
429
430
431
432
433
434
435
436
437
438
439
440
441
442
443
444
445
446
447
448
449
450
451
452
453
454
455
456
457
458
459
460
461
462
463
464
465
466
467
468
469
470
471
472
473
474
475
476
477
478
479
480
481
482
483
484
485
486
487
488
489
490
491
492
493
494
495
496
497
498
499
500
501
502
503
504
505
506
507
508
509
510
511
512
513
514
515
516
517
518
519
520
521
522
523
524
525
526
527
528
529
530
531
532
533
534
535
536
537
538
539
540
541
542
543
544
545
546
547
548
549
550
551
552
553
554
555
556
557
558
559
559
560
561
562
563
564
565
566
567
568
569
570
571
572
573
574
575
576
577
578
579
579
580
581
582
583
584
585
586
587
588
589
589
590
591
592
593
594
595
596
597
598
599
599
600
601
602
603
604
605
606
607
608
609
609
610
611
612
613
614
615
616
617
618
619
619
620
621
622
623
624
625
626
627
628
629
629
630
631
632
633
634
635
636
637
638
639
639
640
641
642
643
644
645
646
647
648
649
649
650
651
652
653
654
655
656
657
658
659
659
660
661
662
663
664
665
666
667
668
669
669
670
671
672
673
674
675
676
677
678
679
679
680
681
682
683
684
685
686
687
688
689
689
690
691
692
693
694
695
696
697
698
699
699
700
701
702
703
704
705
706
707
708
709
709
710
711
712
713
714
715
716
717
718
719
719
720
721
722
723
724
725
726
727
728
729
729
730
731
732
733
734
735
736
737
738
739
739
740
741
742
743
744
745
746
747
748
749
749
750
751
752
753
754
755
756
757
758
759
759
760
761
762
763
764
765
766
767
768
769
769
770
771
772
773
774
775
776
777
778
779
779
780
781
782
783
784
785
786
787
788
789
789
790
791
792
793
794
795
796
797
798
799
799
800
801
802
803
804
805
806
807
808
809
809
810
811
812
813
814
815
816
817
818
819
819
820
821
822
823
824
825
826
827
828
829
829
830
831
832
833
834
835
836
837
838
839
839
840
841
842
843
844
845
846
847
848
849
849
850
851
852
853
854
855
856
857
858
859
859
860
861
862
863
864
865
866
867
868
869
869
870
871
872
873
874
875
876
877
878
879
879
880
881
882
883
884
885
886
887
887
888
889
889
890
891
892
893
894
895
896
897
898
899
899
900

```

(continues on next page)

(continued from previous page)

```

137
138     if [ ! -f "$PROFILE_PATH" ]; then
139         echo -e "${RED}ERROR: No .profile file found for user \\"${OPTARG}\\$"
140         ↪{NC}" >&2
141         exit 1;
142         fi
143     ;;
144     :)
145         echo -e "${RED}ERROR: missing argument for -$OPTARG\\n${NC}" >&2
146         echo "$USAGE" >&2
147         exit 1
148     ;;
149     \?())
150         echo -e "${RED}ERROR: illegal option: -$OPTARG\\n${NC}" >&2
151         echo "$USAGE" >&2
152         exit 1
153     ;;
154     esac
155 done
156 shift $((OPTIND - 1))
157
158 # exit when any command fails
159 set -e
160
161 # require sudo
162 if [[ $UID != 0 ]]; then
163     echo -e "${RED}Please run this script with sudo:"
164     echo "sudo $0 $*"
165     exit 1
166 fi
167
168 # cleanup files if the programm was shutdown unexpectedly
169 trap 'echo -e "${RED}ERROR: Script was terminated unexpectedly, cleaning up files...\\n"
170 ↪>&2 && pushd -0 > /dev/null && rm -rf $BUILDFOLDER' INT TERM
171
172 # load shared functions
173 source $LIBRARY
174
175 # does the config exist?
176 if [ ! -f "$VERSION_FILE_NAME" ]; then
177     echo -e "${RED}Warning: No version.cfg file found. Generating file and using\u
178 ↪default versions${NC}";
179     echo "$VERSION_FILE" > "$VERSION_FILE_NAME"
180 fi
181
182 source "$VERSION_FILE_NAME"
183 CFG_LOCATION=`pwd -P`
184
185 # fetch specified version
186 if [ ! -d $BUILDFOLDER ]; then
187     mkdir $BUILDFOLDER
188 fi
189
190 pushd $BUILDFOLDER > /dev/null
191
192 if [ ! -d "$PROJ" ]; then
193     git clone ${([ "$VECTOREXT" = true ] && echo "--branch $VECTORBRANCH" || echo "")} -
194     ↪--recursive --depth 1 "$REPO" "${PROJ}%%/*"
195 fi

```

(continues on next page)

(continued from previous page)

```

190
191 pushd $PROJ > /dev/null
192
193 # fetch correct commit
194 if [ $VECTOREXT = false ]; then
195   select_and_get_project_version "$TAG" "COMMIT_HASH"
196 else
197   if [ "$TAG" != 'latest' ]; then
198     BRANCHES=`git branch --contains "$TAG"` || true
199
200     if [ 'rvv-intrinsic' == ${BRANCHES:*(-13)} ]; then
201       git checkout --recurse-submodules "$TAG"
202       COMMIT_HASH="$TAG"
203     else
204       echo -e "${RED}WARNING: Commit hash \"$TAG\" is either not present in rvv-
205       →intrinsic branch or is present in multiple branches. Using latest commit in rvv-
206       →intrinsic branch instead.${NC}"
207       sleep 5s
208     fi
209   COMMIT_HASH=`git rev-parse HEAD`
210 fi
211
212 VERSIONLIST="RiscV-GNU-Toolchain-${TOOLCHAIN_SUFFIX}: $COMMIT_HASH"
213
214 # fetch versions for all subrepos (as specified in versions.cfg)
215 while read LINE; do
216   if [ -n "$LINE" ] && [ "${LINE:0:1}" != "#" ]; then
217     SUBREPO=`echo "$LINE" | sed "s/[=].*$/"`
218     if [ -n "${!SUBREPO}" ]; then
219       SUBREPO_LOWER=`echo "$SUBREPO" | tr [A-Z,_] [a-z,-]`
220       if [ -d "$SUBREPO_LOWER" ]; then
221         pushd $SUBREPO_LOWER > /dev/null
222
223         if [ "${!SUBREPO}" != "default" ]; then
224           git checkout --recurse-submodules ${!SUBREPO}
225         fi
226
227         SUBREPO_COMMIT_HASH=$(git rev-parse HEAD)"
228
229         # set return value to tag name if available
230         # we have to cheat here: Since riscv-collaborators used branch names_
231         →instead
232         # of tag names (why?!), we have to check both and hack the version a_
233         →bit to
234         # indicate that.
235         POSSIBLE_TAGS=`git tag --points-at $SUBREPO_COMMIT_HASH`
236
237         if [ -n "$POSSIBLE_TAGS" ]; then
238           SUBREPO_COMMIT_HASH="${POSSIBLE_TAGS%%[$'\n'*]}"
239         else
240           # check branches
241           POSSIBLE_BRANCHES=`git branch -r --points-at $SUBREPO_COMMIT_HASH`
242           if [ -n "$POSSIBLE_BRANCHES" ]; then
243             ONE_BRANCH=${POSSIBLE_BRANCHES%%[$'\n'*]}
244             # this is hacky. Extracts the number and anything after the_
245             →number

```

(continues on next page)

(continued from previous page)

```

243                         # matching the the pattern d.d, where d is an arbitrary long_
244                         ↵number
245                         SUBREPO_COMMIT_HASH="$({echo "$ONE_BRANCH" | grep -Po '\d+\.\'
246                         ↵d+.*') (${SUBREPO_COMMIT_HASH})"
247                         fi
248                         fi
249                         popd > /dev/null
250                         VERSIONLIST="$VERSIONLIST\n${SUBREPO_LOWER}-$TOOLCHAIN_SUFFIX": $
251                         ↵${SUBREPO_COMMIT_HASH}"
252                         fi
253                         fi
254 done < "${CFG_LOCATION}/${VERSION_FILE_NAME}"
255
256 # build and install if wanted
257 PATH="${INSTALL_PATH}:$PATH"
258
259 if [ $NEWLIB = true ]; then
260     ./configure --prefix=$INSTALL_PATH --enable-multilib --disable-linux
261     # activate custom multilibs
262     pushd "riscv-gcc/gcc/config/riscv" > /dev/null
263     chmod +x ./multilib-generator
264     ./multilib-generator ${NEWLIB_MULTILIBS_GEN} > t-elf-multilib
265     popd > /dev/null
266     NEWLIB_MULTILIB_NAMES=`echo ${NEWLIB_MULTILIBS_GEN} | sed "s/-\(\rv\|32\|64\)\[a-zA-
267     ↵Z\]*,\*\)-\(\[a-zA-Z\]*,\*\)*//g"`
268     echo "Building newlib-multilib for \"${NEWLIB_MULTILIB_NAMES}\""
269     # build
270     make -j$(nproc) NEWLIB_MULTILIB_NAMES="${NEWLIB_MULTILIB_NAMES}"
271 else
272     ./configure --prefix=$INSTALL_PATH --enable-multilib --enable-linux
273     # activate custom multilibs
274     pushd "riscv-gcc/gcc/config/riscv" > /dev/null
275     chmod +x ./multilib-generator
276     ./multilib-generator ${GLIBC_MULTILIBS_GEN} > t-linux-multilib
277     popd > /dev/null
278     GLIBC_MULTILIB_NAMES=`echo ${GLIBC_MULTILIBS_GEN} | sed "s/-\(\rv\|32\|64\)\[a-zA-Z\]*,
279     ↵\*\)-\(\[a-zA-Z\]*,\*\)*//g"`
280     echo "Building linux-multilib for \"${GLIBC_MULTILIB_NAMES}\""
281     # build
282     make -j$(nproc) GLIBC_MULTILIB_NAMES="${GLIBC_MULTILIB_NAMES}" linux
283 fi
284
285 # extend path
286 if [ $EXPORTPATH = true ]; then
287     PATH_STRING="\n# Add RiscV tools to path
288 if [ -d "${INSTALL_PATH}/bin" ]; then
289     PATH="${INSTALL_PATH}/bin:$PATH"
290 fi"
291
292     if ! grep -q "PATH=\"${INSTALL_PATH}/bin:$PATH\" \"$PROFILE_PATH\""; then
293         echo -e "$PATH_STRING" >> "$PROFILE_PATH"
294     fi
295 fi

```

(continues on next page)

(continued from previous page)

```

295 # return to first folder and store version
296 pushd -0 > /dev/null
297 echo -e "$VERSIONLIST" >> "$VERSIONFILE"
298
299 # cleanup if wanted
300 if [ $CLEANUP = true ]; then
301     rm -rf $BUILDFOLDER
302 fi

```

## 6.14.2 install\_riscv\_essentials.sh

```

1 #!/bin/bash
2
3 # Author: Harald Heckmann <mail@haraldheckmann.de>
4 # Date: Jul. 02 2020
5 # Project: QuantumRisc (RheinMain University) <Steffen.Reith@hs-rm.de>
6
7 # require sudo
8 if [[ $UID != 0 ]]; then
9     echo "Please run this script with sudo:"
10    echo "sudo $0 $*"
11    exit 1
12 fi
13
14 # exit when any command fails
15 set -e
16
17 # required tools
18 TOOLS="autoconf automake autotools-dev curl python3 libmpc-dev libmpfr-dev \
19         libgmp-dev gawk build-essential bison flex texinfo gperf libtool \
20         patchutils bc zlib1g-dev libexpat-dev python-is-python3"
21
22 # install and upgrade tools
23 apt-get update
24 apt-get install -y $TOOLS
25 apt-get install --only-upgrade -y $TOOLS

```

## 6.14.3 versions.cfg

```

1 ## Define sourcecode branch
2
3 # default = use predefined versions from current riscv-gnu-toolchain branch
4 # or any arbitrary git tag or commit hash
5 # note that in most projects there is no master branch
6 QEMU=default
7 RISCV_BINUTILS=default
8 RISCV_DEJAGNU=default
9 RISCV_GCC=default
10 RISCV_GDB=default
11 RISCV_GLIBC=default
12 RISCV_NEWLIB=default
13

```

(continues on next page)

(continued from previous page)

```

14
15 ## Define which RiscV architectures and ABIs are supported (space seperated list
16 # ↪ "arch-abi")
17
18 # Taken from Sifive:
19 # https://github.com/sifive/freedom-tools/blob/
20 # ↪ 120fa4d48815fc9e87c59374c499849934f2ce10/Makefile
21 NEWLIB_MULTILIBS_GEN="\
22     rv32e-ilp32e--c \
23     rv32ea-ilp32e--m \
24     rv32em-ilp32e--c \
25     rv32eac-ilp32e-- \
26     rv32emac-ilp32e-- \
27     rv32i-ilp32--c,f,fc,fd,fdc \
28     rv32ia-ilp32-rv32ima,rv32iaf,rv32imaf,rv32iafd,rv32imafdc- \
29     rv32im-ilp32--c,f,fc,fd,fdc \
30     rv32iac-ilp32--f,fd \
31     rv32imac-ilp32-rv32imafc,rv32imafdc- \
32     rv32if-ilp32f--c,d,dc \
33     rv32iaf-ilp32f--c,d,dc \
34     rv32imf-ilp32f--d \
35     rv32imaf-ilp32f-rv32imafdc- \
36     rv32imfc-ilp32f--d \
37     rv32imafc-ilp32f-rv32imafdc- \
38     rv32ifd-ilp32d--c \
39     rv32imfd-ilp32d--c \
40     rv32iafd-ilp32d-rv32imafdc,rv32iafdc- \
41     rv32imafdc-ilp32d-- \
42     rv64i-lp64--c,f,fc,fd,fdc \
43     rv64ia-lp64-rv64ima,rv64iaf,rv64imaf,rv64iafd,rv64imafdc- \
44     rv64im-lp64--c,f,fc,fd,fdc \
45     rv64iac-lp64--f,fd \
46     rv64imac-lp64-rv64imafc,rv64imafdc- \
47     rv64if-lp64f--c,d,dc \
48     rv64iaf-lp64f--c,d,dc \
49     rv64imf-lp64f--d \
50     rv64imaf-lp64f-rv64imafdc- \
51     rv64imfc-lp64f--d \
52     rv64imafc-lp64f-rv64imafdc- \
53     rv64ifd-lp64d--c \
54     rv64imfd-lp64d--c \
55     rv64iafd-lp64d-rv64imafdc,rv64iafdc- \
56     rv64imafdc-lp64d--"
57
58 # Linux install (cross-compile for linux)
59 # Default value from riscv-gcc repository
60 GLIBC_MULTILIBS_GEN="\
61     rv32imac-ilp32-rv32ima,rv32imaf,rv32imafdc,rv32imafc,rv32imafdc- \
62     rv32imafdc-ilp32d-rv32imafdc- \
63     rv64imac-lp64-rv64ima,rv64iaf,rv64imaf,rv64iafd,rv64imafc,rv64imafdc- \
64     rv64imafdc-lp64d-rv64imafdc-"

```

## 6.15 cocotb

### 6.15.1 install\_cocotb\_essentials.sh

```
1 #!/bin/bash
2
3 # Author: Harald Heckmann <mail@haraldheckmann.de>
4 # Date: Oct. 24 2020
5 # Project: QuantumRisc (RheinMain University) <Steffen.Reith@hs-rm.de>
6
7 # require sudo
8 if [[ $UID != 0 ]]; then
9     echo "Please run this script with sudo:"
10    echo "sudo $0 $*"
11    exit 1
12 fi
13
14 # exit when any command fails
15 set -e
16
17 # required tools
18 TOOLS="g++ python3 python3-pip"
19
20 # install and upgrade tools
21 apt-get update
22 apt-get install -y $TOOLS
23 apt-get install --only-upgrade -y $TOOLS
24 # install cocotb
25 python3 -m pip install --upgrade pip
26 python3 -m pip install cocotb
```

## 6.16 yosys

### 6.16.1 install\_yosys.sh

```
1 #!/bin/bash
2
3 # Author: Harald Heckmann <mail@haraldheckmann.de>
4 # Date: Jun. 24 2020
5 # Project: QuantumRisc (RheinMain University) <Steffen.Reith@hs-rm.de>
6
7 # constants
8 RED='\033[1;31m'
9 NC='\033[0m'
10 LIBRARY="../libraries/library.sh"
11 REPO="https://github.com/YosysHQ/yosys.git"
12 PROJ="yosys"
13 BUILDFOLDER="build_and_install_yosys"
14 VERSIONFILE="installed_version.txt"
15 COMPILER="clang"
16 TAG="latest"
17 INSTALL=false
18 INSTALL_PREFIX="default"
```

(continues on next page)

(continued from previous page)

```

19 CLEANUP=false
20
21
22 # parse arguments
23 USAGE="$basename \"$0\" [-h] [-c] [-d dir] [-b buildtool] [-i path] [-t tag] -- Clone_"
24 #> latest tagged ${PROJ} version and build it. Optionally select compiler_
25 #> (buildtool), build directory and version, install binaries and cleanup setup files.
26
27 where:
28     -h          show this help text
29     -c          cleanup project
30     -d dir      build files in \"dir\" (default: ${BUILDFOLDER})
31     -i path     install binaries to path (use \"default\" to use default path)
32     -b compiler specify compiler (default: ${COMPILER}, alternative: gcc)
33     -t tag      specify version (git tag or commit hash) to pull (default: default_
34 #> branch)"
35
36
37 while getopts ':hi:cd:b:t:' OPTION; do
38     case $OPTION in
39         i) INSTALL=true
40             INSTALL_PREFIX="$OPTARG"
41             echo "-i set: Installing built binaries to $INSTALL_PREFIX"
42             ;;
43     esac
44 done
45
46 OPTIND=1
47
48 while getopts ':hi:cd:b:t:' OPTION; do
49     case "$OPTION" in
50         h) echo "$USAGE"
51             exit
52             ;;
53         c) if [ $INSTALL = false ]; then
54                 >&2 echo -e "${RED}ERROR: -c only makes sense if the built binaries_"
55 #> were installed before (-i)"
56                 exit 1
57             fi
58             CLEANUP=true
59             echo "-c set: Removing build directory"
60             ;;
61         d) echo "-d set: Using folder $OPTARG"
62             BUILDFOLDER="$OPTARG"
63             ;;
64         b) echo "-b set: Using compiler $OPTARG"
65             COMPILER="$OPTARG"
66             ;;
67         t) echo "-t set: Using version $OPTARG"
68             TAG="$OPTARG"
69             ;;
70         :) echo -e "${RED}ERROR: missing argument for -$OPTARG\n$NC" >&2
71             echo "$USAGE" >&2
72             exit 1
73             ;;
74     \?) echo -e "${RED}ERROR: illegal option: -$OPTARG\n$NC" >&2
75             echo "$USAGE" >&2
76     esac
77 done

```

(continues on next page)

(continued from previous page)

```

72         exit 1
73     ;;
74     esac
75 done
76
77 shift "$((OPTIND - 1))"
78
79 # exit when any command fails
80 set -e
81
82 # require sudo
83 if [[ $UID != 0 ]]; then
84     echo -e "${RED}Please run this script with sudo:${NC}"
85     echo "sudo $0 $*"
86     exit 1
87 fi
88
89 # cleanup files if the programm was shutdown unexpectedly
90 trap 'echo -e "${RED}ERROR: Script was terminated unexpectedly, cleaning up files...${NC}" && pushd -0 > /dev/null && rm -rf $BUILDFOLDER' INT TERM
91
92 # load shared functions
93 source $LIBRARY
94
95 # fetch specified version
96 if [ ! -d $BUILDFOLDER ]; then
97     mkdir $BUILDFOLDER
98 fi
99
100 pushd $BUILDFOLDER > /dev/null
101
102 if [ ! -d "$PROJ" ]; then
103     git clone --recursive "$REPO" "${PROJ%/*}/"
104 fi
105
106 pushd $PROJ > /dev/null
107 select_and_get_project_version "$TAG" "COMMIT_HASH"
108
109 # build and install if wanted
110 make config-$COMPILER
111 make -j$(nproc)
112
113 if [ $INSTALL = true ]; then
114     if [ "$INSTALL_PREFIX" == "default" ]; then
115         make install
116     else
117         make install PREFIX="$INSTALL_PREFIX"
118     fi
119 fi
120
121 # return to first folder and store version
122 pushd -0 > /dev/null
123 echo "${PROJ##*/}: $COMMIT_HASH" >> "$VERSIONFILE"
124
125 # cleanup if wanted
126 if [ $CLEANUP = true ]; then
127     rm -rf $BUILDFOLDER

```

(continues on next page)

(continued from previous page)

128 **fi**

## 6.16.2 install\_yosys\_essentials.sh

```

1  #!/bin/bash
2
3  # Author: Harald Heckmann <mail@haraldheckmann.de>
4  # Date: Jun. 23 2020
5  # Project: QuantumRisc (RheinMain University) <Steffen.Reith@hs-rm.de>
6
7  # require sudo
8  if [[ $UID != 0 ]]; then
9      echo "Please run this script with sudo:"
10     echo "sudo $0 $*"
11     exit 1
12 fi
13
14 # exit when any command fails
15 set -e
16
17 # required tools
18 TOOLS="build-essential clang bison flex libreadline-dev gawk tcl-dev \
19     libffi-dev git graphviz xdot pkg-config python3 libboost-system-dev \
20     libboost-python-dev libboost-filesystem-dev zlib1g-dev"
21
22 # install and upgrade tools
23 apt-get update
24 apt-get install -y $TOOLS
25 apt-get install --only-upgrade -y $TOOLS

```

## 6.17 trellis

### 6.17.1 install\_trellis.sh

```

1  #!/bin/bash
2
3  # Author: Harald Heckmann <mail@haraldheckmann.de>
4  # Date: Jun. 25 2020
5  # Project: QuantumRisc (RheinMain University) <Steffen.Reith@hs-rm.de>
6
7  # constants
8  RED='\033[1;31m'
9  NC='\033[0m'
10 LIBRARY="..libraries/library.sh"
11 REPO="https://github.com/SymbiFlow/prjtrellis"
12 PROJ="prjtrellis/libtrellis"
13 BUILD FOLDER="build_and_install_trellis"
14 VERSIONFILE="installed_version.txt"
15 TAG="latest"
16 INSTALL=false
17 INSTALL_PREFIX="default"

```

(continues on next page)

(continued from previous page)

```

18 CLEANUP=false
19
20
21 # parse arguments
22 USAGE="$basename \"$0\" [-h] [-c] [-d dir] [-i path] [-t tag] -- Clone latested_
23 ↪tagged ${PROJ} version and build it. Optionally select the build directory and_
24 ↪version, install binaries and cleanup setup files.
25
26 where:
27   -h      show this help text
28   -c      cleanup project
29   -d dir  build files in \"dir\" (default: ${BUILDFOLDER})
30   -i path  install binaries to path (use \"default\" to use default path)
31   -t tag   specify version (git tag or commit hash) to pull (default: Latest tag)
32
33 while getopts ':hi:cd:t:' OPTION; do
34   case $OPTION in
35     i) INSTALL=true
36     INSTALL_PREFIX="$OPTARG"
37     echo "-i set: Installing built binaries to $INSTALL_PREFIX"
38     ;;
39   esac
40 done
41
42 OPTIND=1
43
44 while getopts ':hi:cd:t:' OPTION; do
45   case "$OPTION" in
46     h) echo "$USAGE"
47     exit
48     ;;
49     c) if [ $INSTALL = false ]; then
50       >&2 echo -e "${RED}ERROR: -c only makes sense if the built binaries_
51 ↪were installed before (-i)""
52       exit 1
53     fi
54     CLEANUP=true
55     echo "-c set: Removing build directory"
56     ;;
57     d) echo "-d set: Using folder $OPTARG"
58     BUILDFOLDER="$OPTARG"
59     ;;
60     t) echo "-t set: Using version $OPTARG"
61     TAG="$OPTARG"
62     ;;
63     :) echo -e "${RED}ERROR: missing argument for -$OPTARG\n$NC" >&2
64     echo "$USAGE" >&2
65     exit 1
66     ;;
67     \?) echo -e "${RED}ERROR: illegal option: -$OPTARG\n$NC" >&2
68     echo "$USAGE" >&2
69     exit 1
70     ;;
71   esac
72 done

```

(continues on next page)

(continued from previous page)

```

71
72 shift "$((OPTIND - 1))"
73
74 # exit when any command fails
75 set -e
76
77 # require sudo
78 if [[ $UID != 0 ]]; then
79     echo -e "${RED}Please run this script with sudo:$"
80     echo "sudo $0 $*"
81     exit 1
82 fi
83
84 # cleanup files if the programm was shutdown unexpectedly
85 trap 'echo -e "${RED}ERROR: Script was terminated unexpectedly, cleaning up files..."' \
86       && pushd -0 > /dev/null && rm -rf $BUILDFOLDER' INT TERM
87
88 # load shared functions
89 source $LIBRARY
90
91 # fetch specified version
92 if [ ! -d $BUILDFOLDER ]; then
93     mkdir $BUILDFOLDER
94 fi
95 pushd $BUILDFOLDER > /dev/null
96
97 if [ ! -d "$PROJ" ]; then
98     git clone --recursive "$REPO" "${PROJ%%/*}"
99 fi
100
101 pushd $PROJ > /dev/null
102 select_and_get_project_version "$TAG" "COMMIT_HASH"
103
104 # build and install if wanted
105 if [ "$INSTALL_PREFIX" == "default" ]; then
106     cmake -DCMAKE_INSTALL_PREFIX=/usr .
107 else
108     cmake -DCMAKE_INSTALL_PREFIX="$INSTALL_PREFIX" .
109 fi
110
111 make -j$(nproc)
112
113 if [ $INSTALL = true ]; then
114     make install
115 fi
116
117 # return to first folder and store version
118 pushd -0 > /dev/null
119 echo "${PROJ##*/}: $COMMIT_HASH" >> "$VERSIONFILE"
120
121 # cleanup if wanted
122 if [ $CLEANUP = true ]; then
123     rm -rf $BUILDFOLDER
124 fi

```

## 6.17.2 install\_trellis\_essentials.sh

```
1 #!/bin/bash
2
3 # Author: Harald Heckmann <mail@haraldheckmann.de>
4 # Date: Jun. 25 2020
5 # Project: QuantumRisc (RheinMain University) <Steffen.Reith@hs-rm.de>
6
7 # require sudo
8 if [[ $UID != 0 ]]; then
9     echo "Please run this script with sudo:"
10    echo "sudo $0 $*"
11    exit 1
12 fi
13
14 # exit when any command fails
15 set -e
16
17 # required tools
18 TOOLS="build-essential clang cmake python3 python3-dev libboost-all-dev git"
19
20 # install and upgrade tools
21 apt-get update
22 apt-get install -y $TOOLS
23 apt-get install --only-upgrade -y $TOOLS
```

## 6.18 spike

### 6.18.1 install\_spike\_essentials.sh

```
1 #!/bin/bash
2
3 # Author: Harald Heckmann <mail@haraldheckmann.de>
4 # Date: Oct. 24 2020
5 # Project: QuantumRisc (RheinMain University) <Steffen.Reith@hs-rm.de>
6
7 # require sudo
8 if [[ $UID != 0 ]]; then
9     echo "Please run this script with sudo:"
10    echo "sudo $0 $*"
11    exit 1
12 fi
13
14 # exit when any command fails
15 set -e
16
17 # required tools
18 TOOLS="device-tree-compiler"
19
20 # install and upgrade tools
21 apt-get update
22 apt-get install -y $TOOLS
23 apt-get install --only-upgrade -y $TOOLS
```

## 6.18.2 install\_spike.sh

```

1 #!/bin/bash
2
3 # Author: Harald Heckmann <mail@haraldheckmann.de>
4 # Date: Oct. 24 2020
5 # Project: QuantumRisc (RheinMain University) <Steffen.Reith@hs-rm.de>
6
7 # constants
8 RED='\033[1;31m'
9 NC='\033[0m'
10 LIBRARY="../libraries/library.sh"
11 REPO="https://github.com/riscv/riscv-isa-sim.git"
12 PROJ="spike"
13 BUILDFOLDER="build_and_install_spike"
14 VERSIONFILE="installed_version.txt"
15 TAG="latest"
16 INSTALL=false
17 INSTALL_PREFIX="default"
18 CLEANUP=false
19
20
21 # parse arguments
22 USAGE="$ (basename "$0") [-h] [-c] [-d dir] [-i path] [-t tag] -- Clone latested_
23 ↪tagged ${PROJ} version and build it. Optionally select the build directory and_
24 ↪version, install binaries and cleanup setup files.
25
26 where:
27     -h      show this help text
28     -c      cleanup project
29     -d dir   build files in \"dir\" (default: ${BUILDFOLDER})
30     -i path   install binaries to path (use \"default\" to use default path)
31     -t tag    specify version (git tag or commit hash) to pull (default: Latest tag)
32
33 while getopts ':hi:cd:t:' OPTION; do
34     case $OPTION in
35         i) INSTALL=true
36             INSTALL_PREFIX="$OPTARG"
37             echo "-i set: Installing built binaries to $INSTALL_PREFIX"
38         ;;
39     esac
40 done
41 OPTIND=1
42
43 while getopts ':hi:cd:t:' OPTION; do
44     case "$OPTION" in
45         h) echo "$USAGE"
46             exit
47             ;;
48         c) if [ $INSTALL = false ]; then
49             >&2 echo -e "${RED}ERROR: -c only makes sense if the built binaries_
50 ↪were installed before (-i)">&2
51             exit 1
52     fi

```

(continues on next page)

(continued from previous page)

```

52         CLEANUP=true
53         echo "--c set: Removing build directory"
54         ;;
55     d) echo "--d set: Using folder $OPTARG"
56         BUILDFOLDER="$OPTARG"
57         ;;
58     t) echo "--t set: Using version $OPTARG"
59         TAG="$OPTARG"
60         ;;
61     :) echo -e "${RED}ERROR: missing argument for -$OPTARG\n${NC}" >&2
62         echo "$USAGE" >&2
63         exit 1
64         ;;
65     \?) echo -e "${RED}ERROR: illegal option: -$OPTARG\n${NC}" >&2
66         echo "$USAGE" >&2
67         exit 1
68         ;;
69     esac
70 done
71
72 shift "$((OPTIND - 1))"
73
74 # exit when any command fails
75 set -e
76
77 # require sudo
78 if [[ $UID != 0 ]]; then
79     echo -e "${RED}Please run this script with sudo:"
80     echo "sudo $0 $*"
81     exit 1
82 fi
83
84 # cleanup files if the programm was shutdown unexpectedly
85 trap 'echo -e "${RED}ERROR: Script was terminated unexpectedly, cleaning up files...\n${NC}" && pushd -0 > /dev/null && rm -rf $BUILDFO
86
87 # load shared functions
88 source $LIBRARY
89
90 # fetch specified version
91 if [ ! -d $BUILDFO
92     mkdir $BUILDFO
93 fi
94
95 pushd $BUILDFO > /dev/null
96
97 if [ ! -d "$PROJ" ]; then
98     git clone --recursive "$REPO" "${PROJ%/*}"
99 fi
100
101 pushd $PROJ > /dev/null
102 select_and_get_project_version "$TAG" "COMMIT_HASH"
103
104 # build and install if wanted
105 mkdir -p 'build'
106 pushd 'build' > /dev/null
107

```

(continues on next page)

(continued from previous page)

```

108 if [ "$INSTALL_PREFIX" == "default" ]; then
109     ./configure
110 else
111     ./configure --prefix="$INSTALL_PREFIX"
112 fi
113
114 make -j$(nproc)
115
116 if [ $INSTALL = true ]; then
117     make install
118 fi
119
120 # return to first folder and store version
121 pushd -0 > /dev/null
122 echo "${PROJ##*/}: $COMMIT_HASH" >> "$VERSIONFILE"
123
124 # cleanup if wanted
125 if [ $CLEANUP = true ]; then
126     rm -rf $BUILD_FOLDER
127 fi

```

## 6.19 fujprog

### 6.19.1 install\_fujprog\_essentials.sh

```

1 #!/bin/bash
2
3 # Author: Harald Heckmann <mail@haraldheckmann.de>
4 # Date: Oct. 23 2020
5 # Project: QuantumRisc (RheinMain University) <Steffen.Reith@hs-rm.de>
6
7 # require sudo
8 if [[ $UID != 0 ]]; then
9     echo "Please run this script with sudo:"
10    echo "sudo $0 $*"
11    exit 1
12 fi
13
14 # exit when any command fails
15 set -e
16
17 # required tools
18 TOOLS="libftdi1-dev libusb-dev cmake make build-essential"
19
20 # install and upgrade tools
21 apt-get update
22 apt-get install -y $TOOLS
23 apt-get install --only-upgrade -y $TOOLS

```

## 6.19.2 install\_fujprog.sh

```

1 #!/bin/bash
2
3 # Author: Harald Heckmann <mail@haraldheckmann.de>
4 # Date: Oct. 23 2020
5 # Project: QuantumRisc (RheinMain University) <Steffen.Reith@hs-rm.de>
6
7 # constants
8 RED='\033[1;31m'
9 NC='\033[0m'
10 LIBRARY="../libraries/library.sh"
11 REPO="https://github.com/kost/fujprog.git"
12 PROJ="fujprog"
13 BUILDFOLDER="build_and_install_fujprog"
14 VERSIONFILE="installed_version.txt"
15 RULE_FILE="/etc/udev/rules.d/80-fpga-ulx3s.rules"
16 # space separate multiple rules
17 RULES=(
18     'SUBSYSTEM=="tty", ATTRS{idVendor}=="0403", ATTRS{idProduct}=="6015", MODE=="664", \
19     ↪GROUP="dialout"'
20     'ATTRS{idVendor}=="0403", ATTRS{idProduct}=="6015", GROUP="dialout", MODE=="666"'
21 )
22 TAG="latest"
23 INSTALL=false
24 INSTALL_PREFIX="default"
25 CLEANUP=false
26
27 # parse arguments
28 USAGE="$basename \"$0\" [-h] [-c] [-d dir] [-i path] [-t tag] -- Clone latested, \
29 ↪tagged ${PROJ} version and build it. Optionally select the build directory and, \
30 ↪version, install binaries and cleanup setup files.
31 where:
32     -h          show this help text
33     -c          cleanup project
34     -d dir      build files in \"dir\" (default: ${BUILDFOLDER})
35     -i path     install binaries to path (use \"default\" to use default path)
36     -t tag      specify version (git tag or commit hash) to pull (default: Latest tag)
37
38 while getopts ':hi:cd:t:' OPTION; do
39     case $OPTION in
40         i) INSTALL=true
41             INSTALL_PREFIX="$OPTARG"
42             echo "-i set: Installing built binaries to $INSTALL_PREFIX"
43             ;;
44     esac
45 done
46
47 OPTIND=1
48
49 while getopts ':hi:cd:t:' OPTION; do
50     case "$OPTION" in
51         h) echo "$USAGE"

```

(continues on next page)

(continued from previous page)

```

52         exit
53         ;;
54     c) if [ $INSTALL = false ]; then
55         >&2 echo -e "${RED}ERROR: -c only makes sense if the built binaries were installed before (-i)""
56         exit 1
57     fi
58     CLEANUP=true
59     echo "-c set: Removing build directory"
60     ;;
61     d) echo "-d set: Using folder $OPTARG"
62     BUILD FOLDER="$OPTARG"
63     ;;
64     t) echo "-t set: Using version $OPTARG"
65     TAG="$OPTARG"
66     ;;
67     :) echo -e "${RED}ERROR: missing argument for -$OPTARG\n${NC}" >&2
68     echo "$USAGE" >&2
69     exit 1
70     ;;
71     \?) echo -e "${RED}ERROR: illegal option: -$OPTARG\n${NC}" >&2
72     echo "$USAGE" >&2
73     exit 1
74     ;;
75   esac
76 done
77
78 shift "$((OPTIND - 1))"
79
80 # exit when any command fails
81 set -e
82
83 # require sudo
84 if [[ $UID != 0 ]]; then
85     echo -e "${RED}Please run this script with sudo:""
86     echo "sudo $0 $*"
87     exit 1
88 fi
89
90 # cleanup files if the programm was shutdown unexpectedly
91 trap 'echo -e "${RED}ERROR: Script was terminated unexpectedly, cleaning up files..."'
92     && pushd -0 > /dev/null && rm -rf $BUILD FOLDER' INT TERM
93
94 # load shared functions
95 source $LIBRARY
96
97 # fetch specified version
98 if [ ! -d $BUILD FOLDER ]; then
99     mkdir $BUILD FOLDER
100 fi
101 pushd $BUILD FOLDER > /dev/null
102
103 if [ ! -d "$PROJ" ]; then
104     git clone --recursive "$REPO" "${PROJ%%/*}"
105 fi

```

(continues on next page)

(continued from previous page)

```

107 pushd $PROJ > /dev/null
108 select_and_get_project_version "$TAG" "COMMIT_HASH"
109
110 # build and install if wanted
111 if [ "$INSTALL_PREFIX" == "default" ]; then
112     cmake .
113 else
114     cmake -DCMAKE_INSTALL_PREFIX="${INSTALL_PREFIX}" .
115 fi
116
117 make -j$(nproc)
118
119 if [ $INSTALL = true ]; then
120     make install
121 fi
122
123 # allow any user to access ulx3s fpgas (no sudo)
124 touch "$RULE_FILE"
125
126 for RULE in "${RULES[@]}"; do
127     if ! grep -q "$RULE" "$RULE_FILE"; then
128         echo -e "$RULE" >> "$RULE_FILE"
129     fi
130 done
131
132 # return to first folder and store version
133 pushd -0 > /dev/null
134 echo "${PROJ##*/}: $COMMIT_HASH" >> "$VERSIONFILE"
135
136 # cleanup if wanted
137 if [ $CLEANUP = true ]; then
138     rm -rf $BUILD FOLDER
139 fi

```

## 6.20 gtkwave

### 6.20.1 install\_gtkwave.sh

```

1 #!/bin/bash
2
3 # Author: Harald Heckmann <mail@haraldheckmann.de>
4 # Date: Jun. 25 2020
5 # Project: QuantumRisc (RheinMain University) <Steffen.Reith@hs-rm.de>
6
7 # constants
8 RED='\033[1;31m'
9 NC='\033[0m'
10 LIBRARY="..libraries/library.sh"
11 REPO="https://github.com/gtkwave/gtkwave.git"
12 PROJ="gtkwave/gtkwave3-gtk3"
13 BUILD FOLDER="build_and_install_gtkwave"
14 VERSIONFILE="installed_version.txt"
15 TAG="latest"

```

(continues on next page)

(continued from previous page)

```

16 INSTALL=false
17 INSTALL_PREFIX="default"
18 CLEANUP=false
19
20
21 # parse arguments
22 USAGE="$basename \"$0\" [-h] [-c] [-d dir] [-i path] [-t tag] -- Clone latested_
→tagged ${PROJ} version and build it. Optionally select the build directory and_
→version, install binaries and cleanup setup files.
23
24 where:
25     -h          show this help text
26     -c          cleanup project
27     -d dir      build files in \"dir\" (default: ${BUILDFOLDER})
28     -i path     install binaries to path (use \"default\" to use default path)
29     -t tag      specify version (git tag or commit hash) to pull (default: Latest tag)
→"
30
31
32 while getopts ':hi:cd:t:' OPTION; do
33     case $OPTION in
34         i) INSTALL=true
35             INSTALL_PREFIX="$OPTARG"
36             echo "-i set: Installing built binaries to $INSTALL_PREFIX"
37             ;;
38     esac
39 done
40
41 OPTIND=1
42
43 while getopts ':hi:cd:t:' OPTION; do
44     case "$OPTION" in
45         h) echo "$USAGE"
46             exit
47             ;;
48         c) if [ $INSTALL = false ]; then
49             >&2 echo -e "${RED}ERROR: -c only makes sense if the built binaries_
→were installed before (-i)""
50             exit 1
51         fi
52         CLEANUP=true
53         echo "-c set: Removing build directory"
54         ;;
55         d) echo "-d set: Using folder $OPTARG"
56         BUILDFOLDER="$OPTARG"
57         ;;
58         t) echo "-t set: Using version $OPTARG"
59         TAG="$OPTARG"
60         ;;
61         :) echo -e "${RED}ERROR: missing argument for -$OPTARG\n${NC}" >&2
62         echo "$USAGE" >&2
63         exit 1
64         ;;
65         \?) echo -e "${RED}ERROR: illegal option: -$OPTARG\n${NC}" >&2
66         echo "$USAGE" >&2
67         exit 1
68         ;;

```

(continues on next page)

(continued from previous page)

```

69      esac
70  done
71
72 shift "$((OPTIND - 1))"
73
74 # exit when any command fails
75 set -e
76
77 # require sudo
78 if [[ $UID != 0 ]]; then
79     echo -e "${RED}Please run this script with sudo:${NC}"
80     echo "sudo $0 $*"
81     exit 1
82 fi
83
84 # cleanup files if the programm was shutdown unexpectedly
85 trap 'echo -e "${RED}ERROR: Script was terminated unexpectedly, cleaning up files...${NC}" && pushd -0 > /dev/null && rm -rf $BUILDFOLDER' INT TERM
86
87 # load shared functions
88 source $LIBRARY
89
90 # fetch specified version
91 if [ ! -d $BUILDFOLDER ]; then
92     mkdir $BUILDFOLDER
93 fi
94
95 pushd $BUILDFOLDER > /dev/null
96
97 if [ ! -d "$PROJ" ]; then
98     git clone --recursive "$REPO" "${PROJ%%/*}"
99 fi
100
101 pushd $PROJ > /dev/null
102 select_and_get_project_version "$TAG" "COMMIT_HASH"
103
104 # build and install if wanted
105 if [ "$INSTALL_PREFIX" == "default" ]; then
106     ./configure
107 else
108     ./configure --prefix="$INSTALL_PREFIX"
109 fi
110
111 make -j$(nproc)
112
113 if [ $INSTALL = true ]; then
114     make install
115 fi
116
117 # return to first folder and store version
118 pushd -0 > /dev/null
119 echo "${PROJ##*/}: $COMMIT_HASH" >> "$VERSIONFILE"
120
121 # cleanup if wanted
122 if [ $CLEANUP = true ]; then
123     rm -rf $BUILDFOLDER
124 fi

```

### 6.20.2 install\_gtkwave\_essentials.sh

```
1 #!/bin/bash
2
3 # Author: Harald Heckmann <mail@haraldheckmann.de>
4 # Date: Jun. 25 2020
5 # Project: QuantumRisc (RheinMain University) <Steffen.Reith@hs-rm.de>
6
7 # require sudo
8 if [[ $UID != 0 ]]; then
9     echo "Please run this script with sudo:"
10    echo "sudo $0 $*"
11    exit 1
12 fi
13
14 # exit when any command fails
15 set -e
16
17 # required tools
18 TOOLS="build-essential git gcc make debhelper libgtk2.0-dev zlib1g-dev \
19      libbz2-dev flex gperf tcl-dev tk-dev liblzma-dev libjudy-dev \
20      libgconf2-dev"
21
22 # install and upgrade tools
23 apt-get update
24 apt-get install -y $TOOLS
25 apt-get install --only-upgrade -y $TOOLS
```