QuantumRisc-VM

Harald Heckmann

Oct 08, 2020

CONTENTS

CHAPTER

ONE

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 usuable 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.

CHAPTER

TWO

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

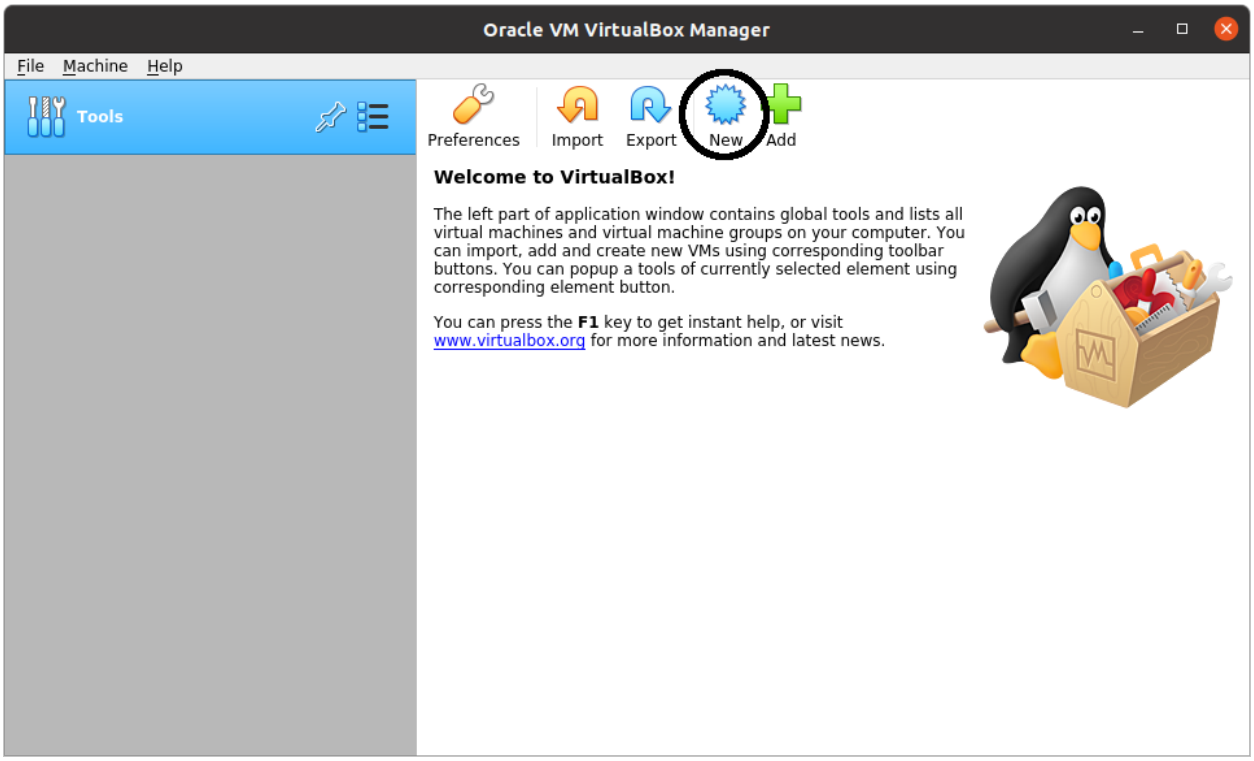
- QuantumRisc-VM
- VirtualBox (tested with version 6.1.10_Ubuntu r138449)
- >80GB hard disk space

2.2 Setup

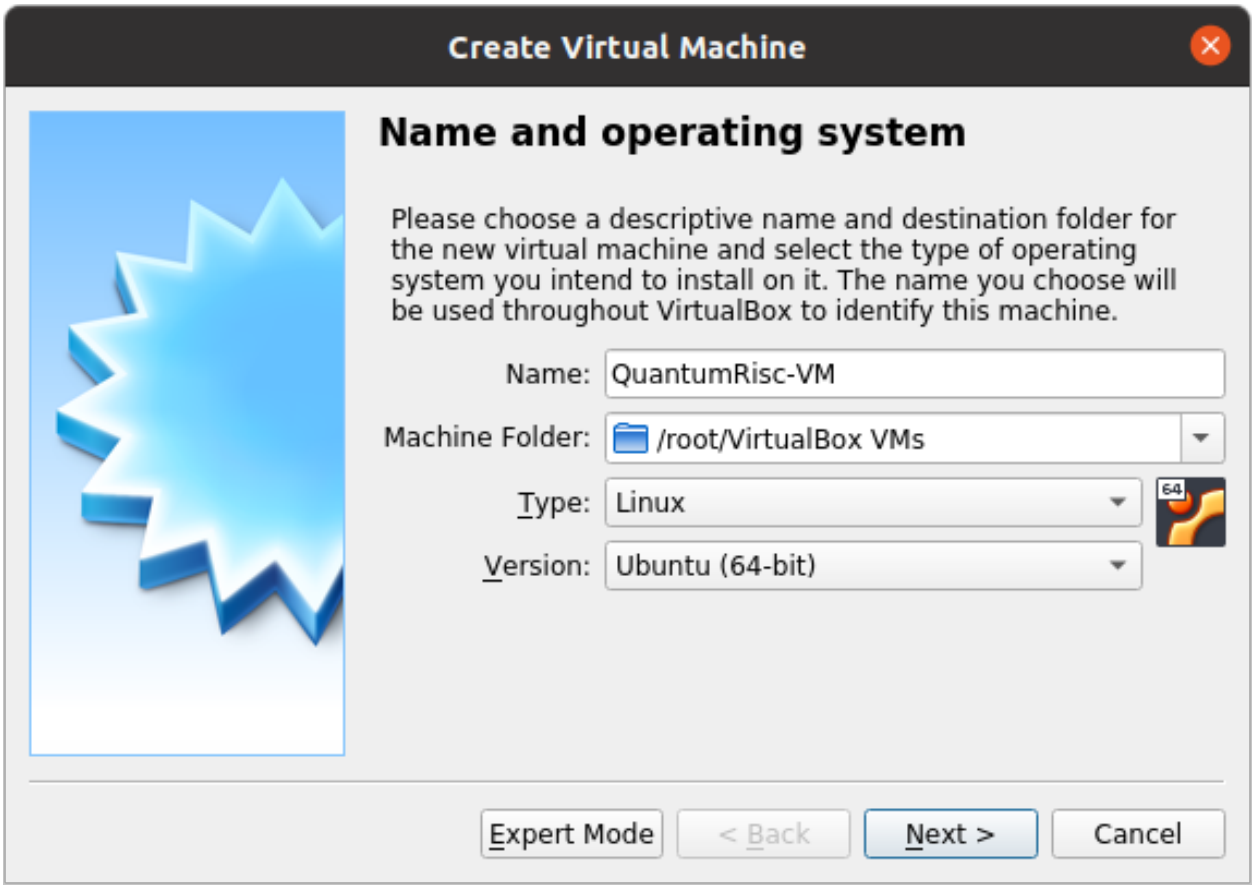
Download and extract the QuantumRisc-VM from the link mentioned in *Prerequisites*. You can get yourself a coffee or a tee, because the archive is >10GB large and the extraction takes more than half an hour. Also install VirtualBox, following the instructions given at the vendors page.

2.2.1 Setting up VirtualBox

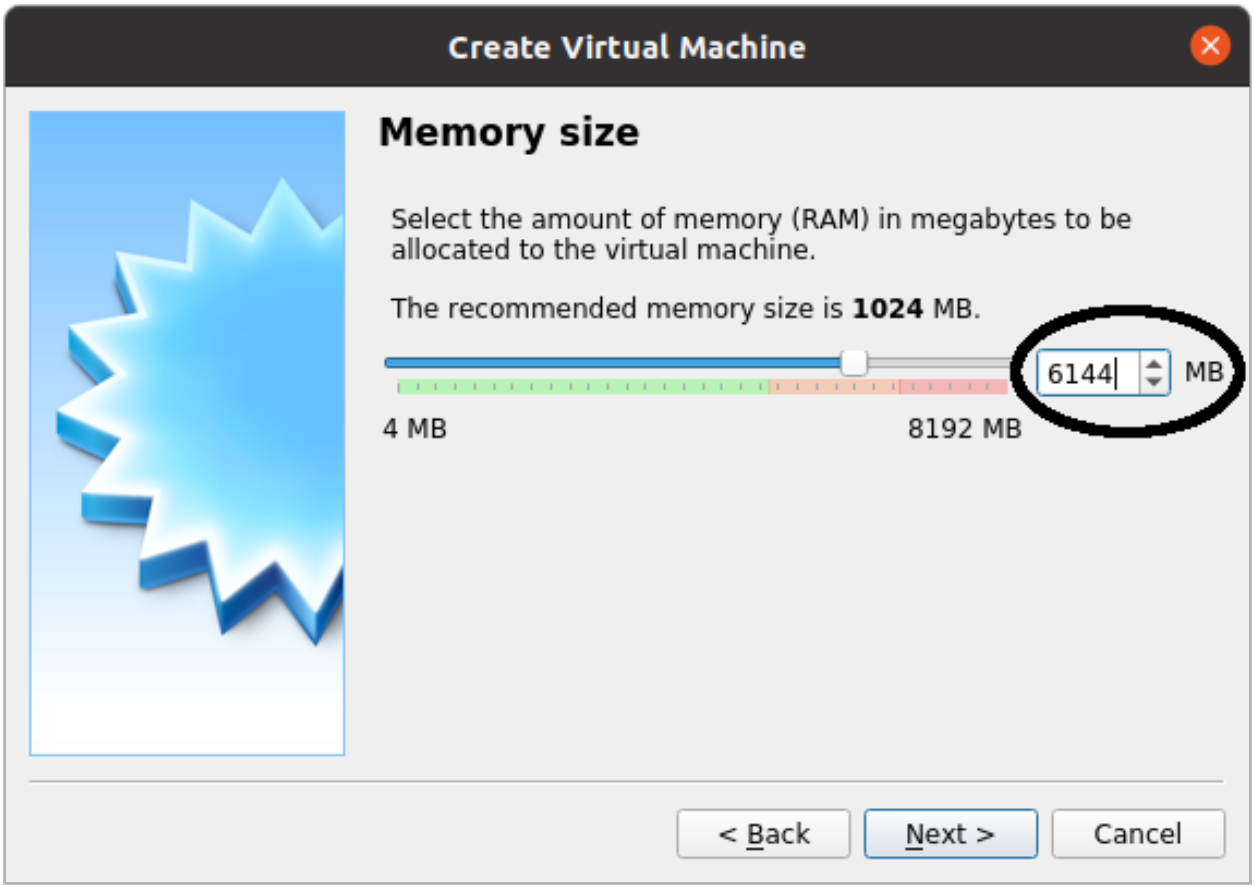
Start VirtualBox and select “new” in the toolbar to add the QuantumRisc-VM image:



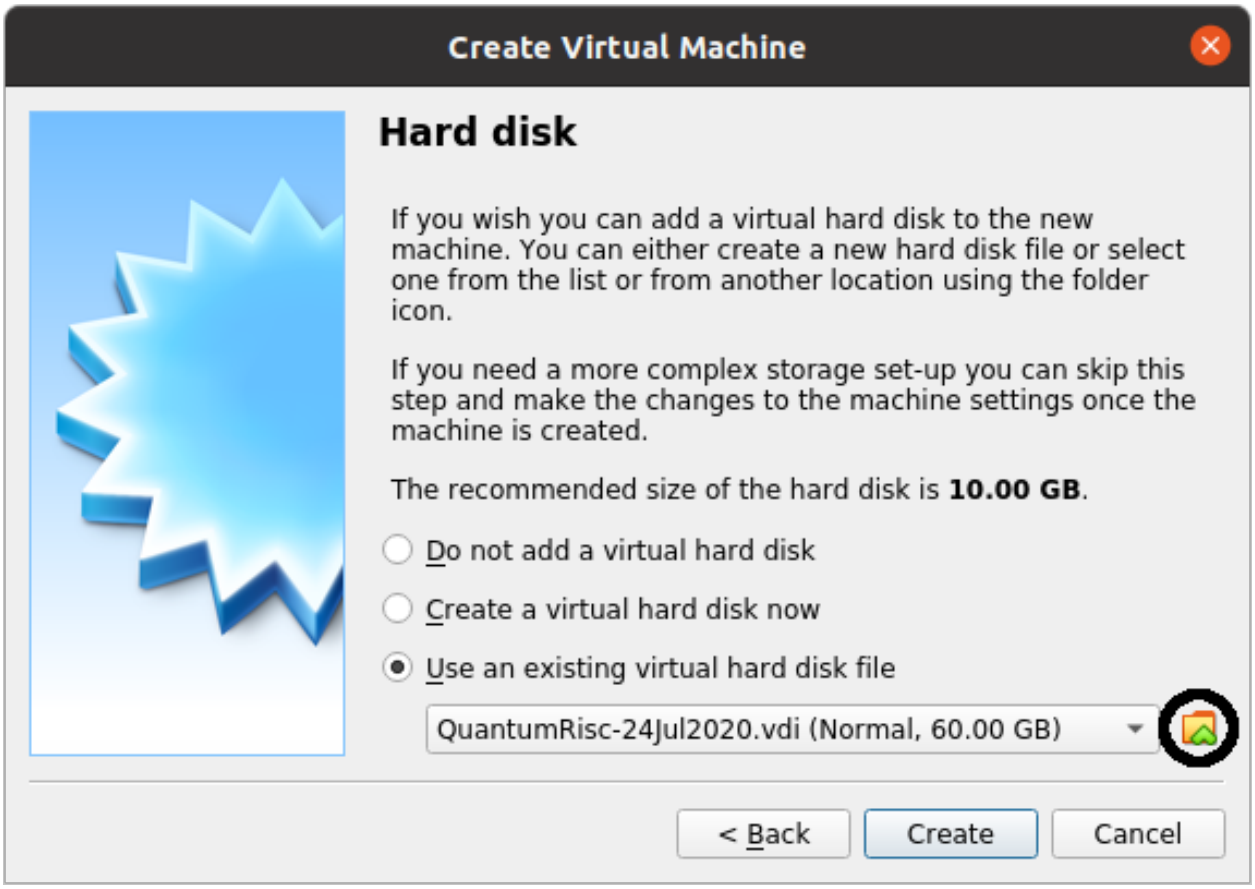
Give the Virtual Machine a name and data folder:



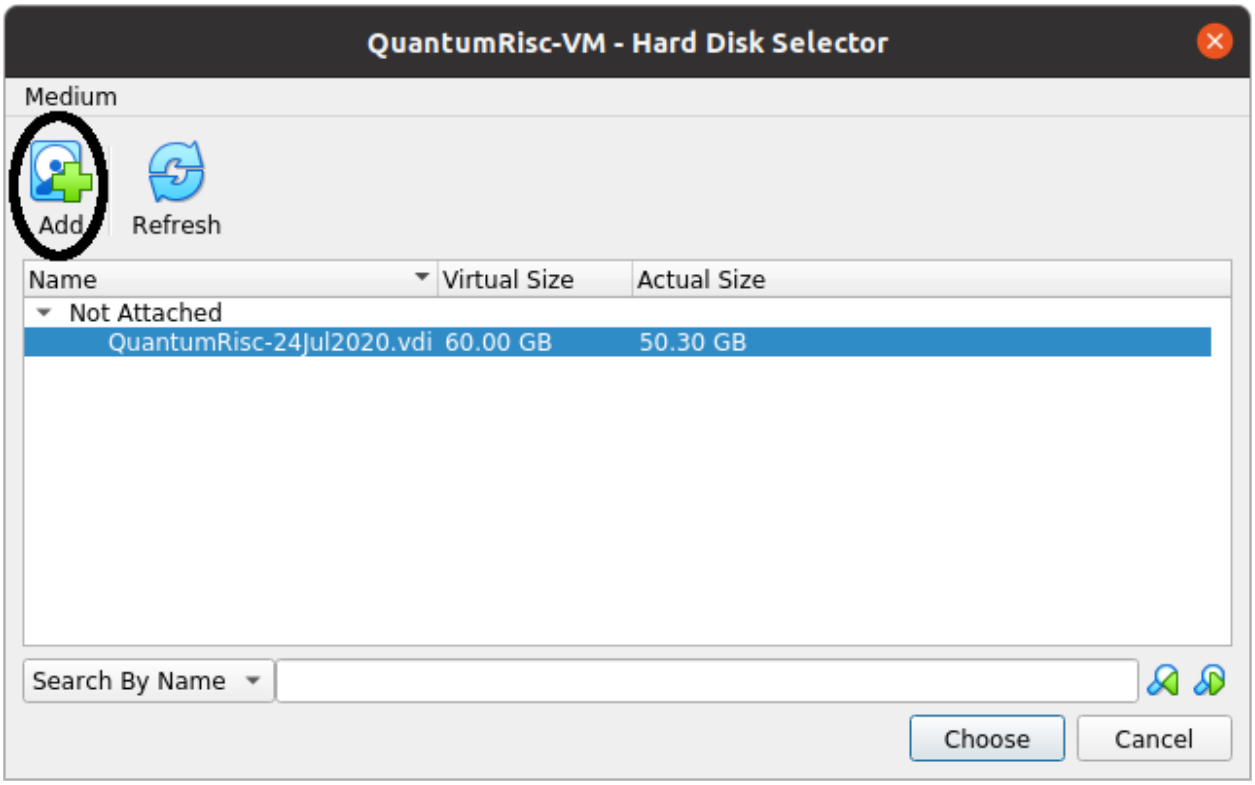
Select the amount of RAM to assign to the VM (this can be changed later). A value too low can lead either to a dysfunctional VM or massive swapping of RAM contents to the hard drive, which slows the machine down. A value too high has the same effects on the hosting machine. To use the VM, 4GB should 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:

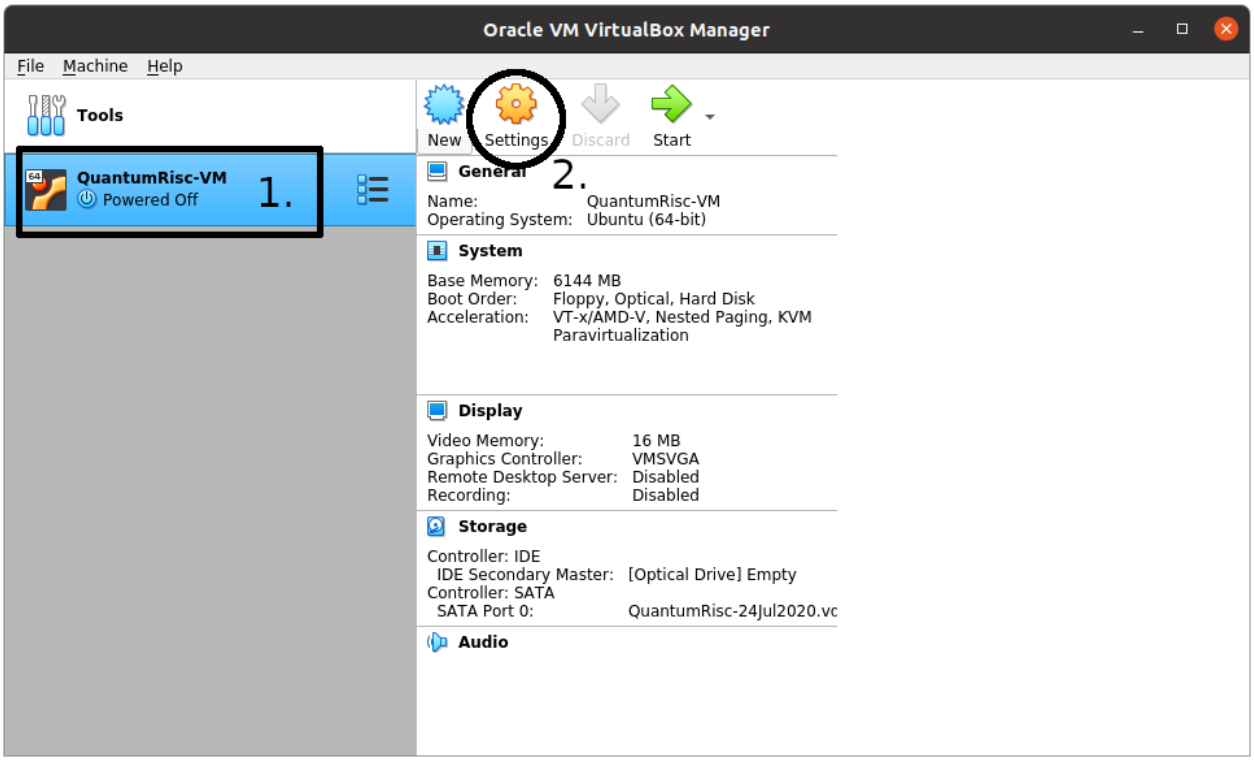


Press on “create”. Your VM has now been created and can be used. Before you use it, you should configure it as instructed in the section *Setting up QuantumRisc-VM*.

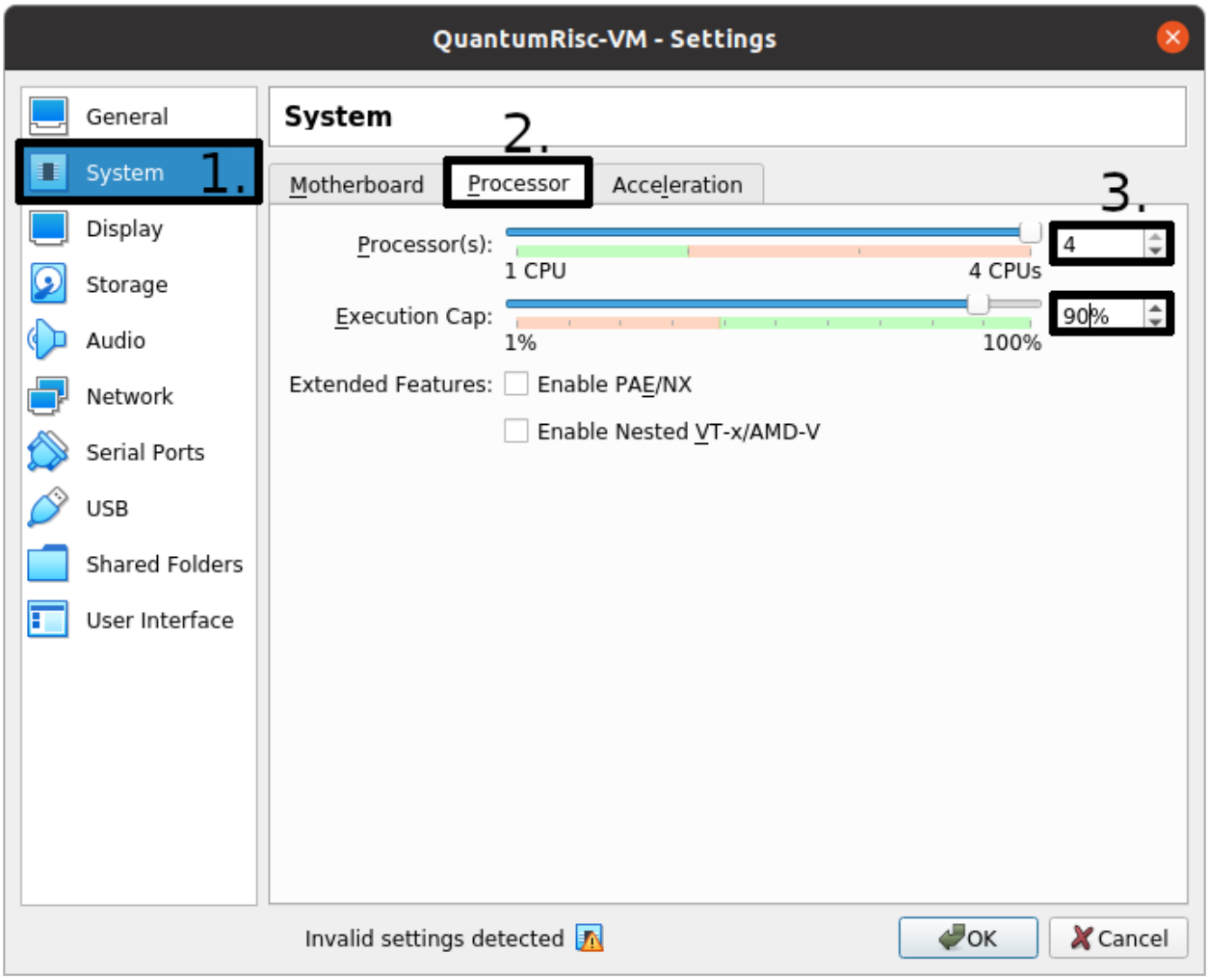
2.2.2 Setting up QuantumRisc-VM

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.

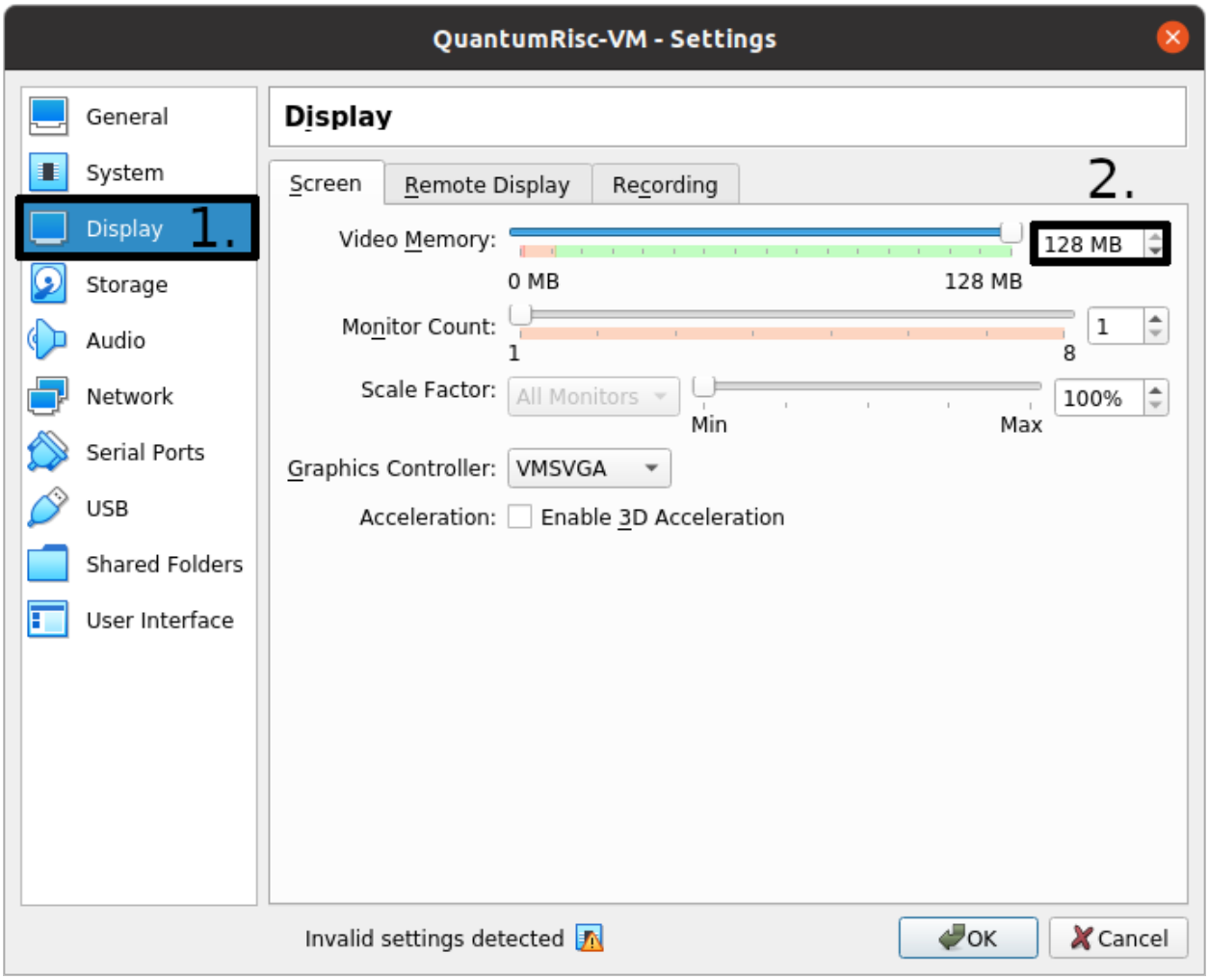
Start by selecting the VM from the list of available VMs and click on the cogwheel icon:



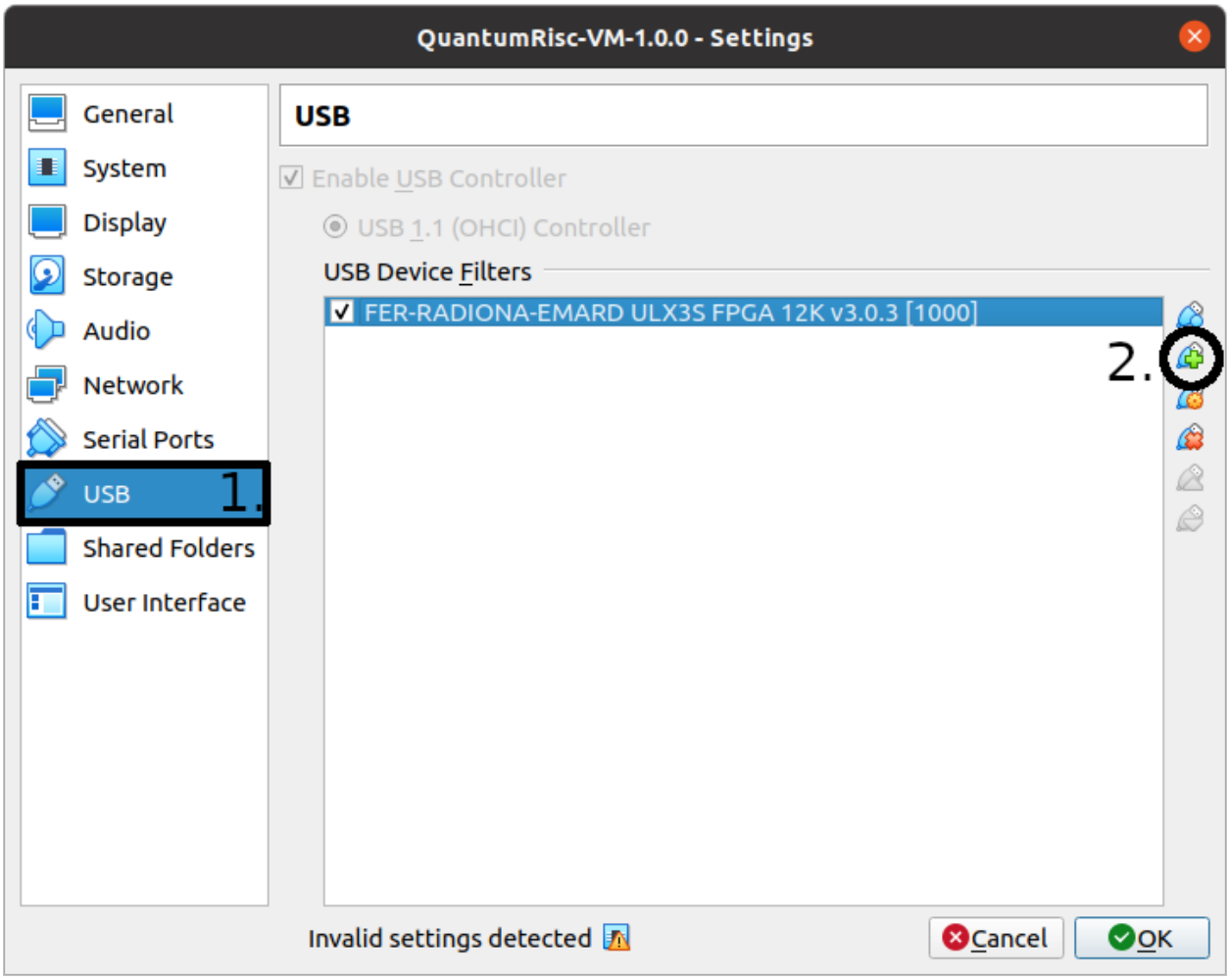
To configure the processor count and usage cap, click on “System” in the left list of categories. Select the “Processor” tab. You can specify the number of processors and the execution cap. You might not want to select 100% execution cap in case you have selected all available processors, because that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

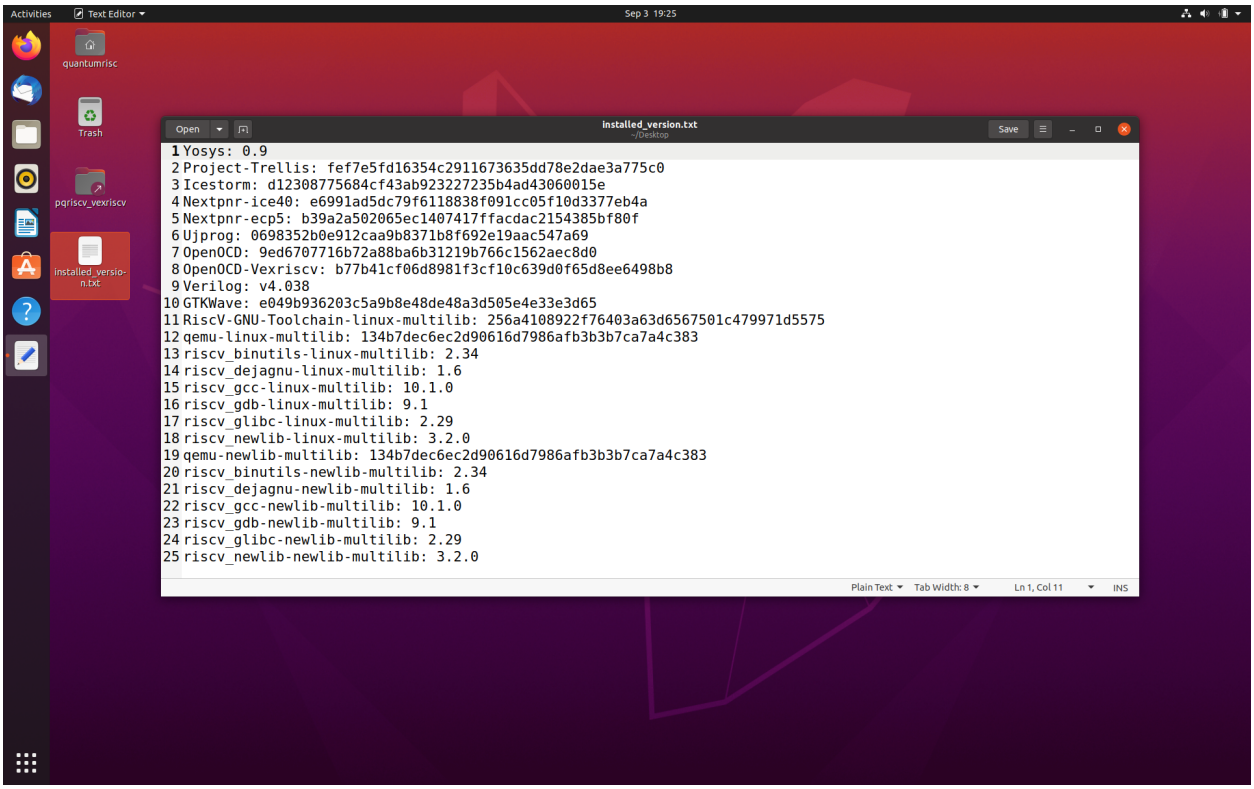


Hint: You can also add and remove permissions to pass through your USB devices during the execution of the VM. To do so, click on *Devices -> USB* in the menu of the running VM.

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 *Hello World*

2.3.1 Hello World

A demo application which uses most if not all of the tools on the VM will be included in the next version.

CHAPTER

THREE

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>_essentials.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 repostiory
PQRISCV_VEXRISCV=true
# 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
# Space seperated list of users (in quotation marks) to install the project for
# in /home/$user/Documents. default = all logged in users
PQRISCV_VEXRISCV_USER=default
# Symbolic link to /home/$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*.

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 by 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.

CHAPTER

FOUR

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.

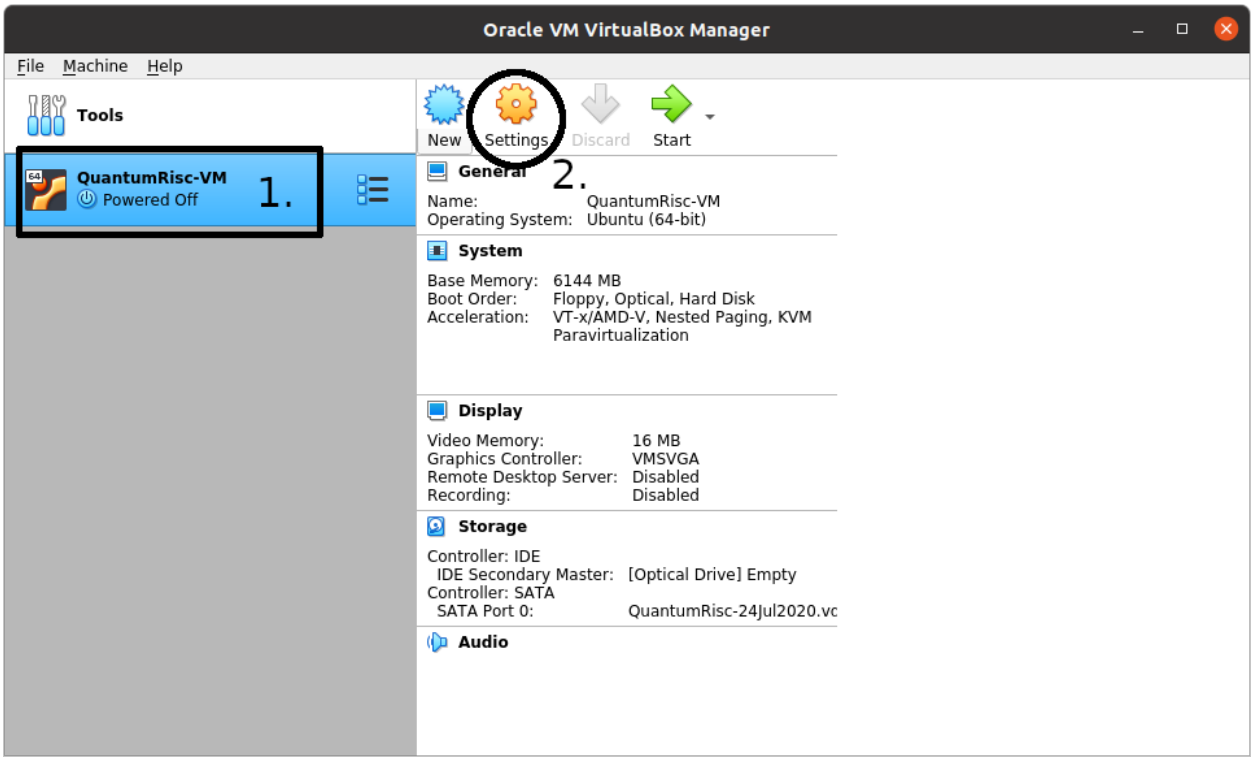
4.1 Prerequisites

- VirtualBox (tested with version 6.1.10_Ubuntu r138449)
- VirtualBox Guest Additions
- Ubuntu 20.04 LTS setup iso
- VM build tools
- >6GB RAM
- >70GB hard disk space (~54GB for the VM, ~12GB to archive it)

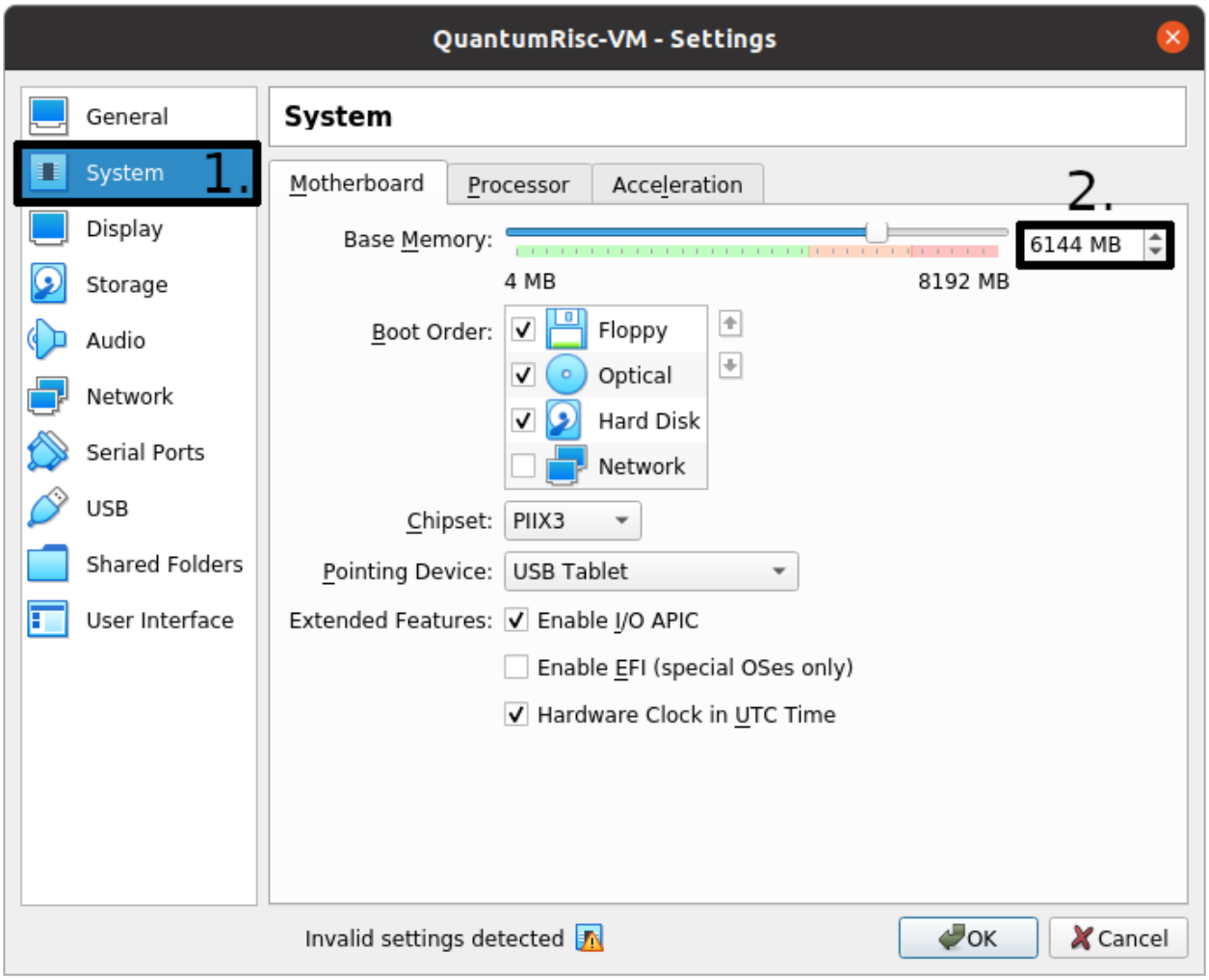
4.2 Preparing the VM

Follow the instructions on how to install Ubuntu 20.04 LTS, but instead of allocating 30GB of disk space, choose at least 60GB. You can set the username and password both to "quantumrisc". After the successful installation of Ubuntu and all tools and projects, about 54 GB are used up. In case you selected 60GB, about 6GB are still available for the end user to download and install addtional software. During the installation of the tools and projects, the disk will use up to almost 60 GB temporarily.

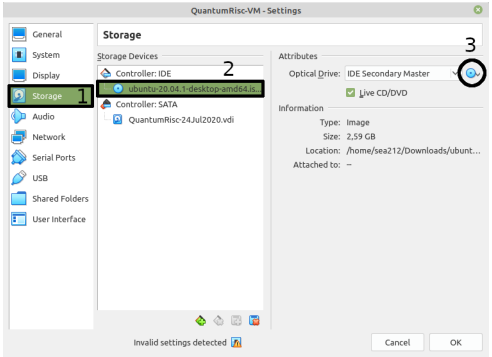
After the successful installation of Ubuntu 20.04 LTS and the VirtualBox Guest Additions on the VM, shutdown the VM and follow the instructions from section *Setting up QuantumRisc-VM*. In addition to those instructions, you also have to raise the available memory for the VM to at least 6GB. To achieve 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`

CHAPTER

FIVE

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 zlibc zlib1g zlib1g-dev \
       ccache libgoogle-perftools-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 zlibc zlib1g zlib1g-dev \
       ccache libgoogle-perftools-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"
BUILDFOLDER="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*. *VERSIONFILE* 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)"
                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..."␣
↪&& pushd -0 > /dev/null && rm -rf $BUILDFOLDER' 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 requires 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 $BUILDFOLDER ]; then
    mkdir $BUILDFOLDER
fi

pushd $BUILDFOLDER > /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 *$BUILDFOLDER*, which is controlled by the *-d* flag. This approach renders a simultaneous execution of multiple instances of the script possible, for example to build different versions at the same time. After that the directory is changed to the workspace. All the scripts use *pushd* and *popd*, which uses a rotatable directory stack to keep track of visited directories. The git project is fetched if the git project does not exist in the workspace yet. The *–recursive* flag is ignored if no submodules are existent, therefore it is supplied every time *git clone* is invoked. If submodules are added to the git project in the future, the script still remains functioning. At last the git project version is changed to *$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 $BUILDFOLDER
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"
BUILDFOLDER="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_NEWLIB 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:

```
## Hello world demo application
# Download git repostiory
DEMO_PROJECT=true
# Git URL
DEMO_PROJECT_URL="https://github.com/ThorKn/icebreaker-vexriscv-helloworld.git"
# Specify project version to pull (default/latest, stable, tag, branch, hash)
DEMO_PROJECT_TAG=default
# Space seperated list of users (in quotation marks) to install the project for
# in /home/$user/Documents. default = all logged in users
```

(continues on next page)

(continued from previous page)

```
DEMO_PROJECT_USER=default
# Symbolic link to /home/$user/Desktop
DEMO_PROJECT_LINK_TO_DESKTOP=true
```

Replace DEMO_PROJECT with the project you want to add and adjust the configuration values as you desire:

```
## Description
# Download git repostiory
<YOUR_PROJECT>=true
# 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
# Space seperated list of users (in quotation marks) to install the project for
# in /home/$user/Documents. default = all logged in users
<YOUR_PROJECT>_USER=default
# Symbolic link to /home/$user/Desktop
<YOUR_PROJECT>_LINK_TO_DESKTOP=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 it:

```
RED='\033[1;31m'
NC='\033[0m'
CONFIG="config.cfg"
BUILDFOLDER="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_NEWLIB 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 toggles 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
```

(continues on next page)

(continued from previous page)

```
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 "${BUILDFOLDER}/${VERSIONFILE}" ]; then
    cp "${BUILDFOLDER}/${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.

CHAPTER

SIX

SCRIPT AND CONFIGURATION INDEX

6.1 build_tools

6.1.1 install_build_essentials.sh

```
#!/bin/bash

# Author: Harald Heckmann <mail@haraldheckmann.de>
# Date: Jun. 23 2020
# Project: QuantumRisc (RheinMain University) <Steffen.Reith@hs-rm.de>

# 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="build-essential git clang gcc meson ninja-build g++ python3-dev \
       openjdk-8-jdk scala sbt make"

# install and upgrade tools
echo "deb https://dl.bintray.com/sbt/debian /" | sudo tee -a /etc/apt/sources.list.d/
↪sbt.list
apt-key adv --keyserver hkp://keyserver.ubuntu.com:80 --recv 642AC823
apt-get update
apt-get install -y $TOOLS
apt-get install --only-upgrade -y $TOOLS
```

6.1.2 install_everything.sh

```
#!/bin/bash

# Author: Harald Heckmann <mail@haraldheckmann.de>
# Date: Jul. 23 2020
# Project: QuantumRisc (RheinMain University) <Steffen.Reith@hs-rm.de>

# constants
RED='\033[1;31m'
NC='\033[0m'
CONFIG="config.cfg"
BUILDFOLDER="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 GTKTERM GTKWAVE RISCV_NEWLIB RISCV_LINUX"
PROJECTS="PQRISCV_VEXRISCV DEMO_PROJECT_ICE40"


# parse arguments
USAGE="$(basename "$0") [-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: ${BUILDFOLDER})"

while getopts ':chopvd:' OPTION; do
    case "$OPTION" in
        c)  echo "-c set: Cleaning up everything in the end"
            CLEANUP=true
            ;;
        d)  echo "-d set: Using folder $OPTARG"
            BUILDFOLDER="$OPTARG"
            ;;
        h)  echo "$USAGE"
            exit
            ;;
        o)  echo "-o set: Adding users \"${OPTARG}\" to dialout"
            DIALOUT_USERS="$OPTARG"
            ;;
        p)  echo "-o set: Copying version file to desktop of \"${OPTARG}\""
            VERSION_FILE_USERS="$OPTARG"
            ;;
        v)  echo "-v set: Being verbose"
```

(continues on next page)

(continued from previous page)

```
            VERBOSE=true
            ;;
        :)  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))

# Prints only if verbose is set
function echo_verbose {
    if [ $VERBOSE = true ]; then
        echo "$1"
    fi
}

# Prints only errors from executed commands if verbose is set
# Parameter $1: Command to execute
# Parameter $2: Path to error file
function exec_verbose {
    if [ $VERBOSE = false ]; then
        $1 > /dev/null 2>> "$2"
    else
        $1 2>> "$2"
    fi
}

# 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 {
    # Stable selected: Choose latest tag if available, otherwise use default branch
    if [ "$1" == "stable" ]; then
        local L_TAGLIST=`git rev-list --tags --max-count=1`

        # tags found?
        if [ -n "$L_TAGLIST" ]; then
            local L_COMMIT_HASH="`git describe --tags $L_TAGLIST`"
            git checkout --recurse-submodules "$L_COMMIT_HASH"
        else
            git checkout --recurse-submodules $(git symbolic-ref refs/remotes/origin/HEAD | sed 's@^refs/remotes/origin/@@')
            local L_COMMIT_HASH="$(git rev-parse HEAD)"
            >&2 echo -e "${RED}WARNING: No git tags found, using default branch${NC}"
        fi
    else
        # Either checkout defaut/stable branch or use custom commit hash, tag or branch name
        if [ "$1" == "default" ] || [ "$1" == "latest" ]; then
            git checkout --recurse-submodules $(git symbolic-ref refs/remotes/origin/HEAD | sed 's@^refs/remotes/origin/@@')
```

(continues on next page)

(continued from previous page)

```
            local L_COMMIT_HASH="$(git rev-parse HEAD)"
        else
            # Check if $1 contains a valid tag and use it as the version if it does
            git checkout --recurse-submodules "$1"
            local L_COMMIT_HASH="$(git rev-parse HEAD)"

            for CUR_TAG in `git tag --list`; do
                if [ "$CUR_TAG" == "$1" ]; then
                    L_COMMIT_HASH="$1"
                    break
                fi
            done
        fi
    fi

    # Apply return value
    eval "$2=\"$L_COMMIT_HASH\""
}

# Read latest executed tool/project/etc.
# Parameter $1: tool/project/etc. list
# Parameter $2: success file
# Parameter $3: string containing list element type (tool/project/etc.)
# Parameter $4: Return variable name
function get_latest {
    if [ ! -f "$2" ]; then
        return 0
    fi

    local LATEST_SCRIPT=`cat $2`
    local SCRIPTS_ADAPTED=`echo "$1" | sed "s/.*${LATEST_SCRIPT} //"`

    if [ "$SCRIPTS_ADAPTED" == "$1" ]; then
        local AT_END=true
        echo -e "\nThe script detected a checkpoint after the last ${3}. This means that all ${3}s already have been checked and installed if configured that way. Do you want to check every ${3} and install them again if configured that way (y/n)?"
    else
        local AT_END=false
        echo -e "\nThe script detected a checkpoint. Do you want to install every ${3} from the checkpoint onwards (y) if configured that way or do you want to start over from the beginning (n)?"
        echo "${3}s yet to be check for installation after the checkpoint: $SCRIPTS_ADAPTED"
    fi

    local DECISION="z"

    while [ $DECISION != "n" ] && [ $DECISION != "y" ]; do
        read -p "Decision(y/n): " DECISION

        if [ -z $DECISION ]; then
            DECISION="z"
        fi
    done

    echo -e "\n"
```

(continues on next page)

(continued from previous page)

```

    if [ $DECISION == "n" ]; then
        if [ $AT_END = true ]; then
            eval "$4=\"\""
        fi
    else
        eval "$4=\"$SCRIPTS_ADAPTED\""
    fi
}

# Process riscv_gnu_toolchain script parameters
# Parameter $1: Script name
# Parameter $2: Variable to store the parameters in
function parameters_tool_riscv {
    # set -n flag
    if [ "${1:6}" == "NEWLIB" ]; then
        eval "$2=\"${!2} -n\""
    fi

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

    # set "u" parameter
    local L_BUILD_USER="$(eval "echo $`echo $1`_USER")"

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

    # set "p" parameter
    local L_BUILD_INSTALL_PATH="$(eval "echo $`echo $1`_INSTALL_PATH")"

    if [ -n "$L_BUILD_INSTALL_PATH" ] && [ "$L_BUILD_INSTALL_PATH" != "default" ];␣
↪then
        eval "$2=\"${!2} -p \"$L_BUILD_INSTALL_PATH\"\""
    fi
}

# Process nextpnr script parameters
# Parameter $1: Script name
# Parameter $2: Variable to store the parameters in
function parameters_tool_nextpnr {
    # set -e flag
    if [ "${1:8}" == "ECP5" ]; then
        eval "$2=\"${!2} -e\""
    fi

    local L_BUILD_CHIPDB="$(eval "echo $`echo $1`_CHIPDB_PATH")"

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

# Process common script parameters
```

(continues on next page)

(continued from previous page)

```
# 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")"

    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
}

# Copies the project to documents and creates a symbolic link if desired
# Parameter $1: Project name
# Parameter $2: User name
# Parameter $3: Create symbolic link (bool)
function install_project_for_user {
    # Get user and home directory
    local L_USER_HOME=$(getent passwd "$2" | cut -d: -f6)

    # User not found
    if [ -z "$L_USER_HOME" ]; then
        echo -e "${RED}ERROR: User ${L_USER} does not exist (home directory not␣
↪found).${NC}"
        exit 1;
```

(continues on next page)

(continued from previous page)

```
    fi

    # Lookup Documents and Desktop and create if not existant
    local L_USER_DOCUMENTS="${L_USER_HOME}/Documents"
    local L_USER_DESKTOP="${L_USER_HOME}/Desktop"

    # TODO: Improve for multiple languages (currently only en support)
    if [ ! -d "${L_USER_DOCUMENTS}" ]; then
        mkdir $L_USER_DOCUMENTS
        chown -R ${L_USER}:$L_USER $L_USER_DOCUMENTS
    fi

    if [ ! -d "${L_USER_DESKTOP}" ]; then
        mkdir $L_USER_DESKTOP
        chown -R ${L_USER}:$L_USER $L_USER_DESKTOP
    fi

    # Copy project to Documents
    cp -r "$1" "$L_USER_DOCUMENTS"
    chown -R ${L_USER}:$L_USER "${L_USER_DOCUMENTS}/$1"

    # Create symbolic link if desired
    if [ "$3" = true ]; then
        ln -s "${L_USER_DOCUMENTS}/$1" "${L_USER_DESKTOP}/$1"
    fi
}

# Install project ("configure projects" section in config.cfg)
# Parameter $1: Project name
function install_project {
    if [ "${!1}" = false ]; then
        return 0
    fi

    local L_NAME_LOWER=`echo "$1" | tr [A-Z] [a-z]`

    # Clone
    if [ ! -d "$L_NAME_LOWER" ]; then
        exec_verbose "git clone --recurse-submodules ""$(eval "echo $`echo $1`_URL")"
↪" ""$L_NAME_LOWER""" "$ERROR_FILE"
    fi

    # Checkout specified version
    local L_TAG="$(eval "echo $`echo $1`_TAG")"

    if [ "$L_TAG" != "default" ]; then
        pushd $L_NAME_LOWER > /dev/null
        exec_verbose "select_and_get_project_version ""$L_TAG"" ""L_COMMIT_HASH""" "
↪$ERROR_FILE"
        popd > /dev/null
    fi

    local L_LINK="$(eval "echo $`echo $1`_LINK_TO_DESKTOP")"

    # Get users to install the projects for
    local L_USERLIST="$(eval "echo $`echo $1`_USER")"

```

(continues on next page)

(continued from previous page)

```
    if [ "$L_USERLIST" == "default" ]; then
        for L_USER in `who | cut -d: -f1`; do
            install_project_for_user "$L_NAME_LOWER" "$L_USER" $L_LINK
        done
    else
        for L_USER in "$L_USERLIST"; do
            install_project_for_user "$L_NAME_LOWER" "$L_USER" $L_LINK
        done
    fi

    rm -rf "$L_NAME_LOWER"
}

# Moves script folder into build folder and returns script path
# Parameter $1: Script name
# Parameter $2: Variable to store the script path for requirements script in
# Parameter $3: Variable to store the script path for installation script in
function find_script {
    if [ "${SCRIPT::5}" == "RISCV" ]; then
        cp -r ../riscv_tools .
        eval "$2=\"$(pwd -P)/riscv_tools/install_riscv_essentials.sh\""
        eval "$3=\"$(pwd -P)/riscv_tools/install_riscv.sh\""
        cp "$(pwd -P)/riscv_tools/versions.cfg" .
    elif [ "${SCRIPT::7}" == "NEXTPNR" ]; then
        cp -r ../nextpnr .
        eval "$2=\"$(pwd -P)/nextpnr/install_nextpnr_essentials.sh\""
        eval "$3=\"$(pwd -P)/nextpnr/install_nextpnr.sh\""
    else
        local L_NAME_LOWER=`echo "$1" | tr [A-Z] [a-z]`
        cp -r ../${L_NAME_LOWER} .
        eval "$2=\"$(pwd -P)/${L_NAME_LOWER}/install_${L_NAME_LOWER}_essentials.sh\""
        eval "$3=\"$(pwd -P)/${L_NAME_LOWER}/install_${L_NAME_LOWER}.sh\""

        # TODO: Extend to automatically find all configuration files
        if [ -f "$(pwd -P)/${L_NAME_LOWER}/versions.cfg" ]; then
            cp "$(pwd -P)/${L_NAME_LOWER}/versions.cfg" .
        fi
    fi
}

# Copies version file $1 to the desktop of the users specified in $2
# Parameter $1: Version file path
# Parameter $2: User list
function copy_version_file {
    if [ ! -f "$1" ]; then
        return
    fi

    for L_USER in "$2"; do
        local L_VERSION_USER_DESKTOP="$(getent passwd "$L_USER" | cut -d: -f6)/Desktop
↪"

        # TODO: add multiple language support
        if [ ! -d "$L_VERSION_USER_DESKTOP" ]; then
            mkdir "$L_VERSION_USER_DESKTOP"
        fi

```

(continues on next page)

(continued from previous page)

```
        cp "$1" "$L_VERSION_USER_DESKTOP"
    done
}

# 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
fi

# 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"

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

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
```

(continues on next page)

(continued from previous page)

```
done


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

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

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

# 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

# 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

# cleanup
if [ $CLEANUP = true ]; then
    echo_verbose "Cleaning up files"
    rm -rf $BUILDFOLDER
fi

echo "Script finished successfully."
```

6.1.3 config.cfg

```
### Configure tools


## Yosys
# Build and (if desired) install Yosys?
YOSYS=true
# Build AND install yosys?
YOSYS_INSTALL=true
# Install path (default = default path)
YOSYS_INSTALL_PATH=default
# Remove build directory after successful install?
YOSYS_CLEANUP=true
# Folder name in which the project is built
YOSYS_DIR=default
# Compiler (gcc or clang)
YOSYS_COMPILER=clang
# Specify project version to pull (default/latest, stable, tag, branch, hash)
YOSYS_TAG=default


## Project Trellis
# Build and (if desired) install Project Trellis?
TRELLIS=true
# Build AND install Project Trellis?
TRELLIS_INSTALL=true
# Install path (default = default path)
TRELLIS_INSTALL_PATH=default
# Remove build directory after successful install?
TRELLIS_CLEANUP=true
# Folder name in which the project is built
TRELLIS_DIR=default
# Specify project version to pull (default/latest, stable, tag, branch, hash)
TRELLIS_TAG=default


## Icestorm
# Build and (if desired) install Icestorm?
ICESTORM=true
# Build AND install Icestorm?
ICESTORM_INSTALL=true
# Install path (default = default path)
ICESTORM_INSTALL_PATH=default
# Remove build directory after successful install?
ICESTORM_CLEANUP=true
# Folder name in which the project is built
ICESTORM_DIR=default
# Specify project version to pull (default/latest, stable, tag, branch, hash)
ICESTORM_TAG=default


## NextPNR-Ice40
# Build and (if desired) install NextPNR-ice40?
NEXTPNR_ICE40=true
# Build AND install NextPNR-ice40?
NEXTPNR_ICE40_INSTALL=true
```

(continues on next page)

(continued from previous page)

```
# Install path (default = default path)
NEXTPNR_ICE40_INSTALL_PATH=default
# Remove build directory after successful install?
NEXTPNR_ICE40_CLEANUP=false
# Folder name in which the project is built
NEXTPNR_ICE40_DIR=default
# Specify project version to pull (default/latest, stable, tag, branch, hash)
NEXTPNR_ICE40_TAG=default
# Use chip dbs from the following path (default = fetch latest chip dbs)
NEXTPNR_ICE40_CHIPDB_PATH=default


## NextPNR-Ecp5
# Build and (if desired) install NextPNR-Ecp5?
NEXTPNR_ECP5=true
# Build AND install NextPNR-NextPNR-Ecp5?
NEXTPNR_ECP5_INSTALL=true
# Install path (default = default path)
NEXTPNR_ECP5_INSTALL_PATH=default
# Remove build directory after successful install?
NEXTPNR_ECP5_CLEANUP=true
# Folder name in which the project is built
NEXTPNR_ECP5_DIR=default
# Specify project version to pull (default/latest, stable, tag, branch, hash)
NEXTPNR_ECP5_TAG=default
# Use chip dbs from the following path (default = fetch latest chip dbs)
NEXTPNR_ECP5_CHIPDB_PATH=default


## Ujprog
# Build and (if desired) install Ujprog?
UJPROG=true
# Build AND install Ujprog?
UJPROG_INSTALL=true
# Install path (default = default path)
UJPROG_INSTALL_PATH=default
# Remove build directory after successful install?
UJPROG_CLEANUP=true
# Folder name in which the project is built
UJPROG_DIR=default
# Specify project version to pull (default/latest, stable, tag, branch, hash)
UJPROG_TAG=default


## OpenOCD
# Build and (if desired) install OpenOCD?
OPENOCD=true
# Build AND install OpenOCD?
OPENOCD_INSTALL=true
# Install path (default = default path)
OPENOCD_INSTALL_PATH=default
# Remove build directory after successful install?
OPENOCD_CLEANUP=true
# Folder name in which the project is built
OPENOCD_DIR=default
# Specify project version to pull (default/latest, stable, tag, branch, hash)
OPENOCD_TAG=default
```

(continues on next page)

(continued from previous page)

```
113
114
115 ## OpenOCD-VexRiscV
116 # Build and (if desired) install OpenOCD-VexRiscV?
117 OPENOCD_VEXRISCV=true
118 # Build AND install OpenOCD-VexRiscV?
119 OPENOCD_VEXRISCV_INSTALL=true
120 # Install path (default = default path)
121 OPENOCD_VEXRISCV_INSTALL_PATH=default
122 # Remove build directory after successful install?
123 OPENOCD_VEXRISCV_CLEANUP=true
124 # Folder name in which the project is built
125 OPENOCD_VEXRISCV_DIR=default
126 # Specify project version to pull (default/latest, stable, tag, branch, hash)
127 OPENOCD_VEXRISCV_TAG=default
128
129
130 ## Verilator
131 # Build and (if desired) install Verilator?
132 VERILATOR=true
133 # Build AND install Verilator?
134 VERILATOR_INSTALL=true
135 # Install path (default = default path)
136 VERILATOR_INSTALL_PATH=default
137 # Remove build directory after successful install?
138 VERILATOR_CLEANUP=true
139 # Folder name in which the project is built
140 VERILATOR_DIR=default
141 # Specify project version to pull (default/latest, stable, tag, branch, hash)
142 VERILATOR_TAG=default
143
144
145 ## GTKTerm
146 # Build and (if desired) install GTKTerm?
147 GTKTERM=true
148 # Build AND install GTKTerm?
149 GTKTERM_INSTALL=true
150 # Install path (default = default path)
151 GTKTERM_INSTALL_PATH=default
152 # Remove build directory after successful install?
153 GTKTERM_CLEANUP=true
154 # Folder name in which the project is built
155 GTKTERM_DIR=default
156 # Specify project version to pull (default/latest, stable, tag, branch, hash)
157 GTKTERM_TAG=default
158
159
160 ## GTKWave
161 # Build and (if desired) install GTKWave?
162 GTKWAVE=true
163 # Build AND install GTKWave?
164 GTKWAVE_INSTALL=true
165 # Install path (default = default path)
166 GTKWAVE_INSTALL_PATH=default
167 # Remove build directory after successful install?
168 GTKWAVE_CLEANUP=true
169 # Folder name in which the project is built
```

(continues on next page)

(continued from previous page)

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


## RiscV-GNU-Toolchain Newlib Multilib
# build and install RiscV-GNU-Toolchain?
RISCV_NEWLIB=true
# Remove build directory after successful install?
RISCV_NEWLIB_CLEANUP=false
# Folder name in which the project is built
RISCV_NEWLIB_DIR=default
# Specify project version to pull (default/latest, stable, tag, branch, hash)
# Note: You can specify the version of every single tool of this toolchain in
# ./riscv_tools/versions.cfg
RISCV_NEWLIB_TAG=default
# Extend PATH by RiscV-GNU-Toolchain path?
RISCV_NEWLIB_EXTEND_PATH=true
# Specify user to install the toolchain for (default = everybody)
# Note: this only makes sense if PATH is extended (RISCV_NEWLIB_EXTEND_PATH)
RISCV_NEWLIB_USER=default
# Specify install path (default: /opt/riscv)
RISCV_NEWLIB_INSTALL_PATH=default


## RiscV-GNU-Toolchain Linux Multilib
# build and install RiscV-GNU-Toolchain?
RISCV_LINUX=true
# Remove build directory after successful install?
RISCV_LINUX_CLEANUP=true
# Folder name in which the project is built
RISCV_LINUX_DIR=default
# Specify project version to pull (default/latest, stable, tag, branch, hash)
# Note: You can specify the version of every single tool of this toolchain in
# ./riscv_tools/versions.cfg
RISCV_LINUX_TAG=default
# Extend PATH by RiscV-GNU-Toolchain path?
RISCV_LINUX_EXTEND_PATH=true
# Specify user to install the toolchain for (default = everybody)
# Note: this only makes sense if PATH is extended (RISCV_LINUX_EXTEND_PATH)
RISCV_LINUX_USER=default
# Specify install path (default: /opt/riscv)
RISCV_LINUX_INSTALL_PATH=default


### Configure projects

## Pqvexriscv project
# Download git repostiory
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
# Space seperated list of users (in quotation marks) to install the project for
# in /home/$user/Documents. default = all logged in users
PQRISCV_VEXRISCV_USER=default
```

(continues on next page)

(continued from previous page)

```
# Symbolic link to /home/$user/Desktop
PQRISCV_VEXRISCV_LINK_TO_DESKTOP=true

## Hello world demo application
# Download git repostiory
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
# Space seperated list of users (in quotation marks) to install the project for
# in /home/$user/Documents. default = all logged in users
DEMO_PROJECT_ICE40_USER=default
# Symbolic link to /home/$user/Desktop
DEMO_PROJECT_ICE40_LINK_TO_DESKTOP=true
```

6.2 openocd_vexriscv

6.2.1 install_openocd_vexriscv.sh

```
#!/bin/bash

# Author: Harald Heckmann <mail@haraldheckmann.de>
# Date: Jun. 25 2020
# Project: QuantumRisc (RheinMain University) <Steffen.Reith@hs-rm.de>

# constants
RED='\033[1;31m'
NC='\033[0m'
REPO="https://github.com/SpinalHDL/openocd_riscv.git"
PROJ="openocd_vexriscv"
BUILDFOLDER="build_and_install_openocd_vexriscv"
VERSIONFILE="installed_version.txt"
TAG="latest"
INSTALL=false
INSTALL_PREFIX="default"
CLEANUP=false

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"


# parse arguments
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.

where:
    -h          show this help text
    -c          cleanup project
    -d dir      build files in \"dir\" (default: ${BUILDFOLDER})
    -i path     install binaries to path (use \"default\" to use default path)
```

(continues on next page)

(continued from previous page)

```
    -t tag      specify version (git tag or commit hash) to pull (default: Latest tag)"


while getopts ':hi:cd:t:' OPTION; do
    case $OPTION in
        i)  INSTALL=true
            # Adjust configure string
            if [ "$OPTARG" != 'default' ]; then
                CONFIGURE_STRING="${CONFIGURE_STRING//"/usr/local"/"$OPTARG"}"
            fi
            # INSTALL_PREFIX="$OPTARG"
            echo "-i set: Installing built binaries to $OPTARG"
            ;;
    esac
done

OPTIND=1

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)"
                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 $OPTARG"
            TAG="$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))"

# 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 {
```

(continues on next page)

(continued from previous page)

```
    # Stable selected: Choose latest tag if available, otherwise use default branch
    if [ "$1" == "stable" ]; then
        local L_TAGLIST=`git rev-list --tags --max-count=1`

        # tags found?
        if [ -n "$L_TAGLIST" ]; then
            local L_COMMIT_HASH="`git describe --tags $L_TAGLIST`"
            git checkout --recurse-submodules "$L_COMMIT_HASH"
        else
            git checkout --recurse-submodules $(git symbolic-ref refs/remotes/origin/
↪HEAD | sed 's@^refs/remotes/origin/@@')
            local L_COMMIT_HASH="$(git rev-parse HEAD)"
            >&2 echo -e "${RED}WARNING: No git tags found, using default branch${NC}"
        fi
    else
        # Either checkout defaut/stable branch or use custom commit hash, tag or␣
↪branch name
        if [ "$1" == "default" ] || [ "$1" == "latest" ]; then
            git checkout --recurse-submodules $(git symbolic-ref refs/remotes/origin/
↪HEAD | sed 's@^refs/remotes/origin/@@')
            local L_COMMIT_HASH="$(git rev-parse HEAD)"
        else
            # Check if $1 contains a valid tag and use it as the version if it does
            git checkout --recurse-submodules "$1"
            local L_COMMIT_HASH="$(git rev-parse HEAD)"

            for CUR_TAG in `git tag --list`; do
                if [ "$CUR_TAG" == "$1" ]; then
                    L_COMMIT_HASH="$1"
                    break
                fi
            done
        fi
    fi

    # Apply return value
    eval "$2=\"$L_COMMIT_HASH\""
}

# 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
fi

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

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

(continues on next page)

(continued from previous page)

```

pushd $BUILDFOLDER > /dev/null

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

pushd $PROJ > /dev/null
select_and_get_project_version "$TAG" "COMMIT_HASH"
# build and install if wanted
./bootstrap
./configure $CONFIGURE_STRING

make -j$(nproc)

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

# return to first folder and store version
pushd -0 > /dev/null
echo "${PROJ##*/}: $COMMIT_HASH" >> "$VERSIONFILE"

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

6.2.2 install_openocd_vexriscv_essentials.sh

```
#!/bin/bash

# Author: Harald Heckmann <mail@haraldheckmann.de>
# Date: Jun. 25 2020
# Project: QuantumRisc (RheinMain University) <Steffen.Reith@hs-rm.de>

# 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="build-essential git gcc make libtool pkg-config autoconf automake \
       texinfo libftdi-dev libusb-1.0-0-dev libyaml-dev"

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

6.3 gtkwave

6.3.1 install_gtkwave.sh

```
#!/bin/bash

# Author: Harald Heckmann <mail@haraldheckmann.de>
# Date: Jun. 25 2020
# Project: QuantumRisc (RheinMain University) <Steffen.Reith@hs-rm.de>

# constants
RED='\033[1;31m'
NC='\033[0m'
REPO="https://github.com/gtkwave/gtkwave.git"
PROJ="gtkwave/gtkwave3-gtk3"
BUILDFOLDER="build_and_install_gtkwave"
VERSIONFILE="installed_version.txt"
TAG="latest"
INSTALL=false
INSTALL_PREFIX="default"
CLEANUP=false


# parse arguments
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.

where:
    -h          show this help text
    -c          cleanup project
    -d dir      build files in \"dir\" (default: ${BUILDFOLDER})
    -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)
"


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

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)"
```

(continues on next page)

(continued from previous page)

```
                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 $OPTARG"
            TAG="$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))"

# 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 {
    # Stable selected: Choose latest tag if available, otherwise use default branch
    if [ "$1" == "stable" ]; then
        local L_TAGLIST=`git rev-list --tags --max-count=1`

        # tags found?
        if [ -n "$L_TAGLIST" ]; then
            local L_COMMIT_HASH="`git describe --tags $L_TAGLIST`"
            git checkout --recurse-submodules "$L_COMMIT_HASH"
        else
            git checkout --recurse-submodules $(git symbolic-ref refs/remotes/origin/HEAD | sed 's@^refs/remotes/origin/@@')
            local L_COMMIT_HASH="$(git rev-parse HEAD)"
            >&2 echo -e "${RED}WARNING: No git tags found, using default branch${NC}"
        fi
    else
        # Either checkout defaut/stable branch or use custom commit hash, tag or branch name
        if [ "$1" == "default" ] || [ "$1" == "latest" ]; then
            git checkout --recurse-submodules $(git symbolic-ref refs/remotes/origin/HEAD | sed 's@^refs/remotes/origin/@@')
            local L_COMMIT_HASH="$(git rev-parse HEAD)"
        else
            # Check if $1 contains a valid tag and use it as the version if it does
            git checkout --recurse-submodules "$1"
            local L_COMMIT_HASH="$(git rev-parse HEAD)"

            for CUR_TAG in `git tag --list`; do
```

(continues on next page)

(continued from previous page)

```
                if [ "$CUR_TAG" == "$1" ]; then
                    L_COMMIT_HASH="$1"
                    break
                fi
            done
        fi
    fi

    # Apply return value
    eval "$2=\"$L_COMMIT_HASH\""
}

# 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
fi

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

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

pushd $BUILDFOLDER > /dev/null

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

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

# build and install if wanted
if [ "$INSTALL_PREFIX" == "default" ]; then
    ./configure
else
    ./configure --prefix="$INSTALL_PREFIX"
fi

make -j$(nproc)

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

# return to first folder and store version
pushd -0 > /dev/null
echo "${PROJ##*/}: $COMMIT_HASH" >> "$VERSIONFILE"

```

(continues on next page)

(continued from previous page)

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

6.3.2 install_gtkwave_essentials.sh

```
#!/bin/bash

# Author: Harald Heckmann <mail@haraldheckmann.de>
# Date: Jun. 25 2020
# Project: QuantumRisc (RheinMain University) <Steffen.Reith@hs-rm.de>

# 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="build-essential git gcc make debhelper libgtk2.0-dev zlib1g-dev \
       libbz2-dev flex gperf tcl-dev tk-dev liblzma-dev libjudy-dev \
       libgconf2-dev"

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

6.4 nextpnr

6.4.1 install_nextpnr_essentials.sh

```
#!/bin/bash

# Author: Harald Heckmann <mail@haraldheckmann.de>
# Date: Jun. 24 2020
# Project: QuantumRisc (RheinMain University) <Steffen.Reith@hs-rm.de>

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

# exit when any command fails
```

(continues on next page)

(continued from previous page)

```
set -e

# required tools
TOOLS="clang-format qt5-default libboost-dev libboost-filesystem-dev \
       libboost-thread-dev libboost-program-options-dev libboost-python-dev \
       libboost-iostreams-dev libboost-dev libeigen3-dev python3-dev cmake"

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

6.4.2 install_nextpnr.sh

```
#!/bin/bash

# Author: Harald Heckmann <mail@haraldheckmann.de>
# Date: Jun. 25 2020
# Project: QuantumRisc (RheinMain University) <Steffen.Reith@hs-rm.de>

# constants
RED='\033[1;31m'
NC='\033[0m'
REPO="https://github.com/YosysHQ/nextpnr.git"
PROJ="nextpnr"
CHIP="ice40"
BUILDFOLDER="build_and_install_nextpnr"
VERSIONFILE="installed_version.txt"
TAG="latest"
LIBPATH=""
INSTALL=false
CLEANUP=false
# trellis config
TRELLIS_LIB="/usr"
TRELLIS_REPO="https://github.com/SymbiFlow/prjtrellis"
TRELLIS_PROJ="prjtrellis"
# icestorm config
ICESTORM_REPO="https://github.com/cliffordwolf/icestorm.git"
ICESTORM_PROJ="icestorm"
ICESTORM_LIB="/usr/local/share/icebox"
ICESTORM_ICEBOX_DIR="icestorm"


# parse arguments
USAGE="$(basename "$0") [-h] [-c] [-e] [-d dir] [-i path] [-l path] [-t tag] -- Clone␣
↪latested tagged ${PROJ} version and build it. Optionally select the build directory,
↪ chip files, chipset and version, install binaries and cleanup setup files.

where:
    -h          show this help text
    -c          cleanup project
    -e          install NextPNR for ecp5 chips (default: ice40)
    -d dir      build files in \"dir\" (default: ${BUILDFOLDER})
    -i path     install binaries to path (use \"default\" to use default path)
    -l path     use local chip files for ice40 or ecp5 from \"path\" (use empty␣
↪string for default path in ubuntu)
```

(continues on next page)

(continued from previous page)

```
    -t tag      specify version (git tag or commit hash) to pull (default: Latest tag)"


while getopts ":hecd:i:t:l:" OPTION; do
    case $OPTION in
        i)  INSTALL=true
            INSTALL_PREFIX="$OPTARG"
            echo "-i set: Installing built binaries to $INSTALL_PREFIX"
            ;;
        e)  echo "-e set: Installing NextPNR for ecp5 chipset"
            CHIP="ecp5"
            ;;
    esac
done

OPTIND=1

while getopts ':hecd:i:t:l:' 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)"
            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 $OPTARG"
        TAG="$OPTARG"
        ;;
    l)  echo "-l set: Using local chip data"
        if [ -z "$OPTARG" ]; then
            if [ "$CHIP" = "ice40" ]; then
                LIBPATH="$ICESTORM_LIB"
            else
                LIBPATH="$TRELLIS_LIB"
            fi
        else
            if [ ! -d "$OPTARG" ]; then
                echo -e "${RED}ERROR: Invalid path \"${OPTARG}\""
                exit 1
            fi

            LIBPATH="$OPTARG"
        fi
        ;;
    :)  echo -e "${RED}ERROR: missing argument for -${OPTARG}\n${NC}" >&2
        echo "$USAGE" >&2
        exit 1
        ;;
```

(continues on next page)

(continued from previous page)

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

shift "$((OPTIND - 1))"

# This function does checkout the correct version and return the commit hash or tag
# 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 {
    # Stable selected: Choose latest tag if available, otherwise use default branch
    if [ "$1" == "stable" ]; then
        local L_TAGLIST=`git rev-list --tags --max-count=1`

        # tags found?
        if [ -n "$L_TAGLIST" ]; then
            local L_COMMIT_HASH="`git describe --tags $L_TAGLIST`"
            git checkout --recurse-submodules "$L_COMMIT_HASH"
        else
            git checkout --recurse-submodules $(git symbolic-ref refs/remotes/origin/HEAD | sed 's@^refs/remotes/origin/@@')
            local L_COMMIT_HASH="$(git rev-parse HEAD)"
            >&2 echo -e "${RED}WARNING: No git tags found, using default branch${NC}"
        fi
    else
        # Either checkout defaut/stable branch or use custom commit hash, tag or branch name
        if [ "$1" == "default" ] || [ "$1" == "latest" ]; then
            git checkout --recurse-submodules $(git symbolic-ref refs/remotes/origin/HEAD | sed 's@^refs/remotes/origin/@@')
            local L_COMMIT_HASH="$(git rev-parse HEAD)"
        else
            # Check if $1 contains a valid tag and use it as the version if it does
            git checkout --recurse-submodules "$1"
            local L_COMMIT_HASH="$(git rev-parse HEAD)"

            for CUR_TAG in `git tag --list`; do
                if [ "$CUR_TAG" == "$1" ]; then
                    L_COMMIT_HASH="$1"
                    break
                fi
            done
        fi
    fi

    # Apply return value
    eval "$2=\"$L_COMMIT_HASH\""
}

# exit when any command fails
set -e

```

(continues on next page)

(continued from previous page)

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

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

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

pushd $BUILDFOLDER > /dev/null

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

pushd $PROJ > /dev/null

select_and_get_project_version "$TAG" "COMMIT_HASH"

# build and install if wanted
# chip ice40?
if [ "$CHIP" = "ice40" ]; then
    # is icestorm installed?
    if [ -n "$LIBPATH" ]; then
        if [ "$INSTALL_PREFIX" == "default" ]; then
            cmake -DARCH=ice40 -DICEBOX_ROOT=${LIBPATH} .
        else
            cmake -DARCH=ice40 -DICEBOX_ROOT=${LIBPATH} -DCMAKE_INSTALL_PREFIX="$INSTALL_PREFIX" .
        fi
    else
        echo "Note: Pulling Icestorm from Github."

        if [ ! -d "$ICESTORM_PROJ" ]; then
            git clone $ICESTORM_REPO "$ICESTORM_ICEBOX_DIR"
        fi

        NEXTPNR_FOLDER=`pwd -P`
        # build icebox (chipdbs)
        pushd "${ICESTORM_ICEBOX_DIR}/icebox" > /dev/null
        make -j$(nproc)
        make install DESTDIR=$NEXTPNR_FOLDER PREFIX=''
        popd +0 > /dev/null
        # build icetime (timing)
        pushd "${ICESTORM_ICEBOX_DIR}/icetime" > /dev/null
        make -j$(nproc) PREFIX=$NEXTPNR_FOLDER
        make install DESTDIR=$NEXTPNR_FOLDER PREFIX=''
        popd +0 > /dev/null
        # build nextpnr-ice40 next

```

(continues on next page)

(continued from previous page)

```
        if [ "$INSTALL_PREFIX" == "default" ]; then
            cmake -j$(nproc) -DARCH=ice40 -DICEBOX_ROOT="${NEXTPNR_FOLDER}/share/
↪icebox" .
        else
            cmake -j$(nproc) -DARCH=ice40 -DICEBOX_ROOT="${NEXTPNR_FOLDER}/share/
↪icebox" -DCMAKE_INSTALL_PREFIX="$INSTALL_PREFIX" .
        fi
    fi
# chip ecp5?
else
    # is project trellis installed?
    if [ -d "$LIBPATH" ]; then
        if [ "$INSTALL_PREFIX" == "default" ]; then
            cmake -j$(nproc) -DARCH=ecp5 -DTRELLIS_INSTALL_PREFIX=${LIBPATH} .
        else
            cmake -j$(nproc) -DARCH=ecp5 -DTRELLIS_INSTALL_PREFIX=${LIBPATH} -DCMAKE_
↪INSTALL_PREFIX="$INSTALL_PREFIX" .
        fi
    else
        echo "Note: Pulling Trellis from Github."

        if [ ! -d "$TRELLIS_PROJ" ]; then
            git clone --recursive $TRELLIS_REPO
        fi

        TRELLIS_MAKE_PATH="$(pwd -P)/${TRELLIS_PROJ}/libtrellis"
        pushd "$TRELLIS_MAKE_PATH" > /dev/null
        cmake -j$(nproc) -DCMAKE_INSTALL_PREFIX="$TRELLIS_MAKE_PATH" .
        make -j$(nproc)
        make install
        popd +0 > /dev/null

        if [ "$INSTALL_PREFIX" == "default" ]; then
            cmake -j$(nproc) -DARCH=ecp5 -DTRELLIS_INSTALL_PREFIX="$TRELLIS_MAKE_PATH
↪" .
        else
            cmake -j$(nproc) -DARCH=ecp5 -DTRELLIS_INSTALL_PREFIX="$TRELLIS_MAKE_PATH
↪" -DCMAKE_INSTALL_PREFIX="$INSTALL_PREFIX" .
        fi
    fi
fi

make -j$(nproc)

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

# return to first folder and store version
pushd -0 > /dev/null

if [ "$CHIP" == "ice40" ]; then
    echo "${PROJ##*/}-ice40: $COMMIT_HASH" >> "$VERSIONFILE"
else
    echo "${PROJ##*/}-ecp5: $COMMIT_HASH" >> "$VERSIONFILE"
fi

```

(continues on next page)

(continued from previous page)

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

6.5 openocd

6.5.1 install_openocd.sh

```
#!/bin/bash

# Author: Harald Heckmann <mail@haraldheckmann.de>
# Date: Jun. 25 2020
# Project: QuantumRisc (RheinMain University) <Steffen.Reith@hs-rm.de>

# constants
RED='\033[1;31m'
NC='\033[0m'
REPO="https://git.code.sf.net/p/openocd/code"
PROJ="openocd"
BUILDFOLDER="build_and_install_openocd"
VERSIONFILE="installed_version.txt"
TAG="latest"
INSTALL=false
INSTALL_PREFIX="default"
CLEANUP=false


# parse arguments
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.

where:
    -h          show this help text
    -c          cleanup project
    -d dir      build files in \"dir\" (default: ${BUILDFOLDER})
    -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)
↪"


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

```

(continues on next page)

(continued from previous page)

```
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)"
                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 $OPTARG"
            TAG="$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))"

# 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 {
    # Stable selected: Choose latest tag if available, otherwise use default branch
    if [ "$1" == "stable" ]; then
        local L_TAGLIST=`git rev-list --tags --max-count=1`

        # tags found?
        if [ -n "$L_TAGLIST" ] && [ "$L_TAGLIST" != "v0.10.0" ]; then
            local L_COMMIT_HASH="`git describe --tags $L_TAGLIST`"
            git checkout --recurse-submodules "$L_COMMIT_HASH"
        else
            git checkout --recurse-submodules $(git symbolic-ref refs/remotes/origin/HEAD | sed 's@^refs/remotes/origin/@@')
            local L_COMMIT_HASH="$(git rev-parse HEAD)"
            >&2 echo -e "${RED}WARNING: No git tags found, using default branch${NC}"
        fi
    else
        # Either checkout defaut/stable branch or use custom commit hash, tag or branch name
        if [ "$1" == "default" ] || [ "$1" == "latest" ]; then
            git checkout --recurse-submodules $(git symbolic-ref refs/remotes/origin/HEAD | sed 's@^refs/remotes/origin/@@')
```

(continues on next page)

(continued from previous page)

```
            local L_COMMIT_HASH="$(git rev-parse HEAD)"
        else
            # Check if $1 contains a valid tag and use it as the version if it does
            git checkout --recurse-submodules "$1"
            local L_COMMIT_HASH="$(git rev-parse HEAD)"

            for CUR_TAG in `git tag --list`; do
                if [ "$CUR_TAG" == "$1" ]; then
                    L_COMMIT_HASH="$1"
                    break
                fi
            done
        fi
    fi

    # Apply return value
    eval "$2=\"$L_COMMIT_HASH\""
}

# 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
fi

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

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

pushd $BUILDFOLDER > /dev/null

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

pushd $PROJ > /dev/null

select_and_get_project_version "$TAG" "COMMIT_HASH"

# build and install if wanted
./bootstrap

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

```

(continues on next page)

(continued from previous page)

```
make -j$(nproc)

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

# return to first folder and store version
pushd -0 > /dev/null
echo "${PROJ##*/}: $COMMIT_HASH" >> "$VERSIONFILE"

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

6.5.2 install_openocd_essentials.sh

```
#!/bin/bash

# Author: Harald Heckmann <mail@haraldheckmann.de>
# Date: Jun. 25 2020
# Project: QuantumRisc (RheinMain University) <Steffen.Reith@hs-rm.de>

# 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="build-essential git gcc make libtool pkg-config autoconf automake \
       texinfo libftdi-dev libusb-1.0-0-dev"

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

6.6 yosys

6.6.1 install_yosys_essentials.sh

```
#!/bin/bash

# Author: Harald Heckmann <mail@haraldheckmann.de>
# Date: Jun. 23 2020
# Project: QuantumRisc (RheinMain University) <Steffen.Reith@hs-rm.de>
```

(continues on next page)

(continued from previous page)

```

# 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="build-essential clang bison flex libreadline-dev gawk tcl-dev \
       libffi-dev git graphviz xdot pkg-config python3 libboost-system-dev \
       libboost-python-dev libboost-filesystem-dev zlib1g-dev"

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

6.6.2 install_yosys.sh

```
#!/bin/bash

# Author: Harald Heckmann <mail@haraldheckmann.de>
# Date: Jun. 24 2020
# Project: QuantumRisc (RheinMain University) <Steffen.Reith@hs-rm.de>

# constants
RED='\033[1;31m'
NC='\033[0m'
REPO="https://github.com/YosysHQ/yosys.git"
PROJ="yosys"
BUILDFOLDER="build_and_install_yosys"
VERSIONFILE="installed_version.txt"
COMPILER="clang"
TAG="latest"
INSTALL=false
INSTALL_PREFIX="default"
CLEANUP=false


# parse arguments
USAGE="$(basename "$0") [-h] [-c] [-d dir] [-b buildtool] [-i path] [-t tag] -- Clone␣
↪latested tagged ${PROJ} version and build it. Optionally select compiler␣
↪(buildtool), 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: ${BUILDFOLDER})
    -i path     install binaries to path (use \"default\" to use default path)
    -b compiler specify compiler (default: ${COMPILER}, alternative: gcc)
    -t tag      specify version (git tag or commit hash) to pull (default: default␣
↪branch)"
```

(continues on next page)

(continued from previous page)

```

while getopts ':hi:cd:b: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

while getopts ':hi:cd:b: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)"
                exit 1
            fi
            CLEANUP=true
            echo "-c set: Removing build directory"
            ;;
        d)  echo "-d set: Using folder $OPTARG"
            BUILDFOLDER="$OPTARG"
            ;;
        b)  echo "-b set: Using compiler $OPTARG"
            COMPILER="$OPTARG"
            ;;
        t)  echo "-t set: Using version $OPTARG"
            TAG="$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))"

# 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 {
    # Stable selected: Choose latest tag if available, otherwise use default branch
    if [ "$1" == "stable" ]; then
        local L_TAGLIST=`git rev-list --tags --max-count=1`
```

(continues on next page)

(continued from previous page)

```
85
86          # tags found?
87          if [ -n "$L_TAGLIST" ]; then
88              local L_COMMIT_HASH="`git describe --tags $L_TAGLIST`"
89              git checkout --recurse-submodules "$L_COMMIT_HASH"
90          else
91              git checkout --recurse-submodules $(git symbolic-ref refs/remotes/origin/
    ↪HEAD | sed 's@^refs/remotes/origin/@@')
92              local L_COMMIT_HASH="$(git rev-parse HEAD)"
93              >&2 echo -e "${RED}WARNING: No git tags found, using default branch${NC}"
94          fi
95      else
96          # Either checkout defaut/stable branch or use custom commit hash, tag or␣
    ↪branch name
97          if [ "$1" == "default" ] || [ "$1" == "latest" ]; then
98              git checkout --recurse-submodules $(git symbolic-ref refs/remotes/origin/
    ↪HEAD | sed 's@^refs/remotes/origin/@@')
99              local L_COMMIT_HASH="$(git rev-parse HEAD)"
100         else
101             # Check if $1 contains a valid tag and use it as the version if it does
102             git checkout --recurse-submodules "$1"
103             local L_COMMIT_HASH="$(git rev-parse HEAD)"
104
105             for CUR_TAG in `git tag --list`; do
106                 if [ "$CUR_TAG" == "$1" ]; then
107                     L_COMMIT_HASH="$1"
108                     break
109                 fi
110             done
111         fi
112     fi
113
114     # Apply return value
115     eval "$2=\"$L_COMMIT_HASH\""
116 }
117
118 # exit when any command fails
119 set -e
120
121 # require sudo
122 if [[ $UID != 0 ]]; then
123     echo -e "${RED}Please run this script with sudo:"
124     echo "sudo $0 $*"
125     exit 1
126 fi
127
128 # Cleanup files if the programm was shutdown unexpectedly
129 trap 'echo -e "${RED}ERROR: Script was terminated unexpectedly, cleaning up files..."␣
    ↪&& pushd -0 > /dev/null && rm -rf $BUILDFOLDER' INT TERM
130
131 # fetch specified version
132 if [ ! -d $BUILDFOLDER ]; then
133     mkdir $BUILDFOLDER
134 fi
135
136 pushd $BUILDFOLDER > /dev/null
137
```

(continues on next page)

(continued from previous page)

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

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

# build and install if wanted
make config-$COMPILER
make -j$(nproc)

if [ $INSTALL = true ]; then
    if [ "$INSTALL_PREFIX" == "default" ]; then
        make install
    else
        make install PREFIX="$INSTALL_PREFIX"
    fi
fi

# return to first folder and store version
pushd -0 > /dev/null
echo "${PROJ##*/}: $COMMIT_HASH" >> "$VERSIONFILE"

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

6.7 spinalhdl

6.7.1 install_spinalhdl_essentials.sh

```
#!/bin/bash

# Author: Harald Heckmann <mail@haraldheckmann.de>
# Date: Jun. 23 2020
# Project: QuantumRisc (RheinMain University) <Steffen.Reith@hs-rm.de>

# 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="openjdk-8-jdk scala sbt"

# install and upgrade tools
echo "deb https://dl.bintray.com/sbt/debian /" | sudo tee -a /etc/apt/sources.list.d/
↪sbt.list
```

(continues on next page)

(continued from previous page)

```
apt-key adv --keyserver hkp://keyserver.ubuntu.com:80 --recv 642AC823
apt-get update
apt-get install -y $TOOLS
apt-get install --only-upgrade -y $TOOLS
```

6.8 ujprog

6.8.1 install_ujprog.sh

```
#!/bin/bash

# Author: Harald Heckmann <mail@haraldheckmann.de>
# Date: Jun. 25 2020
# Project: QuantumRisc (RheinMain University) <Steffen.Reith@hs-rm.de>

# constants
RED='\033[1;31m'
NC='\033[0m'
REPO="https://github.com/f32c/tools.git"
PROJ="tools/ujprog"
BUILDFOLDER="build_and_install_ujprog"
VERSIONFILE="installed_version.txt"
TAG="latest"
INSTALL=false
INSTALL_PREFIX="default"
CLEANUP=false


# parse arguments
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.

where:
    -h          show this help text
    -c          cleanup project
    -d dir      build files in \"dir\" (default: ${BUILDFOLDER})
    -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)
↪"


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

```

(continues on next page)

(continued from previous page)

```
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)"
                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 $OPTARG"
            TAG="$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))"

# 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 {
    # Stable selected: Choose latest tag if available, otherwise use default branch
    if [ "$1" == "stable" ]; then
        local L_TAGLIST=`git rev-list --tags --max-count=1`

        # tags found?
        if [ -n "$L_TAGLIST" ]; then
            local L_COMMIT_HASH="`git describe --tags $L_TAGLIST`"
            git checkout --recurse-submodules "$L_COMMIT_HASH"
        else
            git checkout --recurse-submodules $(git symbolic-ref refs/remotes/origin/HEAD | sed 's@^refs/remotes/origin/@@')
            local L_COMMIT_HASH="$(git rev-parse HEAD)"
            >&2 echo -e "${RED}WARNING: No git tags found, using default branch${NC}"
        fi
    else
        # Either checkout defaut/stable branch or use custom commit hash, tag or branch name
        if [ "$1" == "default" ] || [ "$1" == "latest" ]; then
            git checkout --recurse-submodules $(git symbolic-ref refs/remotes/origin/HEAD | sed 's@^refs/remotes/origin/@@')
```

(continues on next page)

(continued from previous page)

```
            local L_COMMIT_HASH="$(git rev-parse HEAD)"
        else
            # Check if $1 contains a valid tag and use it as the version if it does
            git checkout --recurse-submodules "$1"
            local L_COMMIT_HASH="$(git rev-parse HEAD)"

            for CUR_TAG in `git tag --list`; do
                if [ "$CUR_TAG" == "$1" ]; then
                    L_COMMIT_HASH="$1"
                    break
                fi
            done
        fi
    fi

    # Apply return value
    eval "$2=\"$L_COMMIT_HASH\""
}

# 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
fi

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

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

pushd $BUILDFOLDER > /dev/null

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

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

# build and install if wanted
cp Makefile.linux Makefile

# Adjust path if required
if [ "$INSTALL_PREFIX" != "default" ]; then
    mkdir -p "${INSTALL_PREFIX}/bin"
    sed -i "s /usr/local ${INSTALL_PREFIX} g" Makefile
fi

make -j$(nproc)
```

(continues on next page)

(continued from previous page)

```

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

# return to first folder and store version
pushd -0 > /dev/null
echo "${PROJ##*/}: $COMMIT_HASH" >> "$VERSIONFILE"

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

6.8.2 install_ujprog_essentials.sh

```
#!/bin/bash

# Author: Harald Heckmann <mail@haraldheckmann.de>
# Date: Jun. 25 2020
# Project: QuantumRisc (RheinMain University) <Steffen.Reith@hs-rm.de>

# 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="build-essential clang make"

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

6.9 verilator

6.9.1 install_verilator_essentials.sh

```
#!/bin/bash

# Author: Harald Heckmann <mail@haraldheckmann.de>
# Date: Jun. 25 2020
# Project: QuantumRisc (RheinMain University) <Steffen.Reith@hs-rm.de>

# require sudo
```

(continues on next page)

(continued from previous page)

```
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 zlibc zlib1g zlib1g-dev \
       ccache libgoogle-perftools-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
```

6.9.2 install_verilator.sh

```
#!/bin/bash

# Author: Harald Heckmann <mail@haraldheckmann.de>
# Date: Jun. 25 2020
# Project: QuantumRisc (RheinMain University) <Steffen.Reith@hs-rm.de>

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


# parse arguments
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.

where:
    -h          show this help text
    -c          cleanup project
    -d dir      build files in \"dir\" (default: ${BUILDFOLDER})
    -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)
↪"


while getopts ':hi:cd:t:' OPTION; do
    case $OPTION in
```

(continues on next page)

(continued from previous page)

```
        i)  INSTALL=true
            INSTALL_PREFIX="$OPTARG"
            echo "-i set: Installing built binaries to $INSTALL_PREFIX"
            ;;
    esac
done

OPTIND=1

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)"
                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 $OPTARG"
            TAG="$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))"

# 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 {
    # Stable selected: Choose latest tag if available, otherwise use default branch
    if [ "$1" == "stable" ]; then
        local L_TAGLIST=`git rev-list --tags --max-count=1`

        # tags found?
        if [ -n "$L_TAGLIST" ]; then
            local L_COMMIT_HASH="`git describe --tags $L_TAGLIST`"
            git checkout --recurse-submodules "$L_COMMIT_HASH"
        else
            git checkout --recurse-submodules $(git symbolic-ref refs/remotes/origin/HEAD | sed 's@^refs/remotes/origin/@@')
```

(continues on next page)

(continued from previous page)

```
            local L_COMMIT_HASH="$(git rev-parse HEAD)"
            >&2 echo -e "${RED}WARNING: No git tags found, using default branch${NC}"
        fi
    else
        # Either checkout defaut/stable branch or use custom commit hash, tag or branch name
        if [ "$1" == "default" ] || [ "$1" == "latest" ]; then
            git checkout --recurse-submodules $(git symbolic-ref refs/remotes/origin/HEAD | sed 's@^refs/remotes/origin/@@')
            local L_COMMIT_HASH="$(git rev-parse HEAD)"
        else
            # Check if $1 contains a valid tag and use it as the version if it does
            git checkout --recurse-submodules "$1"
            local L_COMMIT_HASH="$(git rev-parse HEAD)"

            for CUR_TAG in `git tag --list`; do
                if [ "$CUR_TAG" == "$1" ]; then
                    L_COMMIT_HASH="$1"
                    break
                fi
            done
        fi
    fi

    # Apply return value
    eval "$2=\"$L_COMMIT_HASH\""
}

# 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
fi

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

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

pushd $BUILDFOLDER > /dev/null

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

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

# build and install if wanted
```

(continues on next page)

(continued from previous page)

```
# 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)

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

# return to first folder and store version
pushd -0 > /dev/null
echo "${PROJ##*/}: $COMMIT_HASH" >> "$VERSIONFILE"

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

6.10 gtkterm

6.10.1 install_gtkterm_essentials.sh

```
#!/bin/bash

# Author: Harald Heckmann <mail@haraldheckmann.de>
# Date: Jun. 25 2020
# Project: QuantumRisc (RheinMain University) <Steffen.Reith@hs-rm.de>

# 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="libgtk-3-dev libvte-2.91-dev intltool libgudev-1.0 meson ninja-build"

```

(continues on next page)

(continued from previous page)

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

6.10.2 install_gtkterm.sh

```
#!/bin/bash

# Author: Harald Heckmann <mail@haraldheckmann.de>
# Date: Oct. 08 2020
# Project: QuantumRisc (RheinMain University) <Steffen.Reith@hs-rm.de>

# constants
RED='\033[1;31m'
NC='\033[0m'
REPO="https://github.com/Jeija/gtkterm"
PROJ="gtkterm"
BUILDFOLDER="build_and_install_gtkterm"
VERSIONFILE="installed_version.txt"
TAG="latest"
INSTALL=false
INSTALL_PREFIX="default"
CLEANUP=false


# parse arguments
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.

where:
    -h          show this help text
    -c          cleanup project
    -d dir      build files in \"dir\" (default: ${BUILDFOLDER})
    -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)"


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

while getopts ':hi:cd:t:' OPTION; do
    case "$OPTION" in
        h)  echo "$USAGE"
            exit
```

(continues on next page)

(continued from previous page)

```
            ;;
        c)  if [ $INSTALL = false ]; then
                >&2 echo -e "${RED}ERROR: -c only makes sense if the built binaries were installed before (-i)"
                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 $OPTARG"
            TAG="$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))"

# 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 {
    # Stable selected: Choose latest tag if available, otherwise use default branch
    if [ "$1" == "stable" ]; then
        local L_TAGLIST=`git rev-list --tags --max-count=1`

        # tags found?
        if [ -n "$L_TAGLIST" ]; then
            local L_COMMIT_HASH="`git describe --tags $L_TAGLIST`"
            git checkout --recurse-submodules "$L_COMMIT_HASH"
        else
            git checkout --recurse-submodules $(git symbolic-ref refs/remotes/origin/HEAD | sed 's@^refs/remotes/origin/@@')
            local L_COMMIT_HASH="$(git rev-parse HEAD)"
            >&2 echo -e "${RED}WARNING: No git tags found, using default branch${NC}"
        fi
    else
        # Either checkout defaut/stable branch or use custom commit hash, tag or branch name
        if [ "$1" == "default" ] || [ "$1" == "latest" ]; then
            git checkout --recurse-submodules $(git symbolic-ref refs/remotes/origin/HEAD | sed 's@^refs/remotes/origin/@@')
            local L_COMMIT_HASH="$(git rev-parse HEAD)"
        else
            # Check if $1 contains a valid tag and use it as the version if it does
```

(continues on next page)

(continued from previous page)

```
            git checkout --recurse-submodules "$1"
            local L_COMMIT_HASH="$(git rev-parse HEAD)"

            for CUR_TAG in `git tag --list`; do
                if [ "$CUR_TAG" == "$1" ]; then
                    L_COMMIT_HASH="$1"
                    break
                fi
            done
        fi
    fi

    # Apply return value
    eval "$2=\"$L_COMMIT_HASH\""
}

# 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
fi

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

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

pushd $BUILDFOLDER > /dev/null

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

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

if [ "$INSTALL_PREFIX" == "default" ]; then
    meson build
else
    meson build -Dprefix="$INSTALL_PREFIX"
fi

if [ $INSTALL = true ]; then
    ninja -C build install
else
    ninja -C build
fi

# return to first folder and store version
```

(continues on next page)

(continued from previous page)

```
pushd -0 > /dev/null
echo "${PROJ##*/}: $COMMIT_HASH" >> "$VERSIONFILE"

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

6.11 icestorm

6.11.1 install_icestorm.sh

```
#!/bin/bash

# Author: Harald Heckmann <mail@haraldheckmann.de>
# Date: Jun. 24 2020
# Project: QuantumRisc (RheinMain University) <Steffen.Reith@hs-rm.de>

# constants
RED='\033[1;31m'
NC='\033[0m'
REPO="https://github.com/cliffordwolf/icestorm.git"
PROJ="icestorm"
BUILDFOLDER="build_and_install_icestorm"
VERSIONFILE="installed_version.txt"
RULE_FILE="/etc/udev/rules.d/53-lattice-ftdi.rules"
# space seperate multiple rules
RULES='ACTION=="add", ATTR{idVendor}=="0403", ATTR{idProduct}=="6010", MODE:="666"'
TAG="latest"
INSTALL=false
INSTALL_PREFIX="default"
CLEANUP=false


# parse arguments
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.

where:
    -h          show this help text
    -c          cleanup project
    -d dir      build files in \"dir\" (default: ${BUILDFOLDER})
    -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)
↪"


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"
```

(continues on next page)

(continued from previous page)

```
            ;;
    esac
done

OPTIND=1

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)"
                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 $OPTARG"
            TAG="$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))"

# 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 {
    # Stable selected: Choose latest tag if available, otherwise use default branch
    if [ "$1" == "stable" ]; then
        local L_TAGLIST=`git rev-list --tags --max-count=1`

        # tags found?
        if [ -n "$L_TAGLIST" ]; then
            local L_COMMIT_HASH="`git describe --tags $L_TAGLIST`"
            git checkout --recurse-submodules "$L_COMMIT_HASH"
        else
            git checkout --recurse-submodules $(git symbolic-ref refs/remotes/origin/HEAD | sed 's@^refs/remotes/origin/@@')
            local L_COMMIT_HASH="$(git rev-parse HEAD)"
            >&2 echo -e "${RED}WARNING: No git tags found, using default branch${NC}"
```

(continues on next page)

(continued from previous page)

```
        fi
    else
        # Either checkout defaut/stable branch or use custom commit hash, tag or branch name
        if [ "$1" == "default" ] || [ "$1" == "latest" ]; then
            git checkout --recurse-submodules $(git symbolic-ref refs/remotes/origin/HEAD | sed 's@^refs/remotes/origin/@@')
            local L_COMMIT_HASH="$(git rev-parse HEAD)"
        else
            # Check if $1 contains a valid tag and use it as the version if it does
            git checkout --recurse-submodules "$1"
            local L_COMMIT_HASH="$(git rev-parse HEAD)"

            for CUR_TAG in `git tag --list`; do
                if [ "$CUR_TAG" == "$1" ]; then
                    L_COMMIT_HASH="$1"
                    break
                fi
            done
        fi
    fi

    # Apply return value
    eval "$2=\"$L_COMMIT_HASH\""
}

# 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
fi

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

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

pushd $BUILDFOLDER > /dev/null

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

pushd $PROJ > /dev/null

select_and_get_project_version "$TAG" "COMMIT_HASH"

# build and install if wanted
make -j$(nproc)
```

(continues on next page)

(continued from previous page)

```

if [ $INSTALL = true ]; then
    if [ "$INSTALL_PREFIX" == "default" ]; then
        make install
    else
        make install PREFIX="$INSTALL_PREFIX"
    fi
fi

# allow any user to access ice fpgas (no sudo)
touch "$RULE_FILE"

for RULE in "$RULES"; do
    if ! grep -q "$RULE" "$RULE_FILE"; then
      echo -e "$RULE" >> "$RULE_FILE"
    fi
done

# return to first folder and store version
pushd -0 > /dev/null
echo "${PROJ##*/}: $COMMIT_HASH" >> "$VERSIONFILE"

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

6.11.2 install_icestorm_essentials.sh

```
#!/bin/bash

# Author: Harald Heckmann <mail@haraldheckmann.de>
# Date: Jun. 24 2020
# Project: QuantumRisc (RheinMain University) <Steffen.Reith@hs-rm.de>

# 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="build-essential clang bison flex libreadline-dev \
       gawk tcl-dev libffi-dev git mercurial graphviz   \
       xdot pkg-config python python3 libftdi-dev \
       qt5-default python3-dev libboost-all-dev cmake libeigen3-dev"

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

6.12 riscv_tools

6.12.1 install_riscv_essentials.sh

```
#!/bin/bash

# Author: Harald Heckmann <mail@haraldheckmann.de>
# Date: Jul. 02 2020
# Project: QuantumRisc (RheinMain University) <Steffen.Reith@hs-rm.de>

# 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="autoconf automake autotools-dev curl python3 libmpc-dev libmpfr-dev \
      libgmp-dev gawk build-essential bison flex texinfo gperf libtool \
      patchutils bc zlib1g-dev libexpat-dev"

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

6.12.2 install_riscv.sh

```
#!/bin/bash

# Author: Harald Heckmann <mail@haraldheckmann.de>
# Date: Jul. 02 2020
# Project: QuantumRisc (RheinMain University) <Steffen.Reith@hs-rm.de>

# constants
RED='\033[1;31m'
NC='\033[0m'
REPO="https://github.com/riscv/riscv-gnu-toolchain.git"
PROJ="riscv-gnu-toolchain"
BUILDFOLDER="build_and_install_riscv_gnu_toolchain"
VERSIONFILE="installed_version.txt"
TOOLCHAIN_SUFFIX="linux-multilib"
TAG="latest"
NEWLIB=false
# INSTALL=false
INSTALL_PATH="/opt/riscv"
PROFILE_PATH="/etc/profile"
CLEANUP=false
EXPORTPATH=false

```

(continues on next page)

(continued from previous page)

```
VERSION_FILE_NAME="versions.cfg"
VERSION_FILE='## Define sourcecode branch
# default = use predefined versions from current riscv-gnu-toolchain branch
# or any arbitrary git tag or commit hash
# note that in most projects there is no master branch
QEMU=default
RISCV_BINUTILS=default
RISCV_DEJAGNU=default
RISCV_GCC=default
RISCV_GDB=default
RISCV_GLIBC=default
RISCV_NEWLIB=default

## Define which RiscV architectures and ABIs are supported (space seperated list "arch-abi")

# Taken from Sifive:
# https://github.com/sifive/freedom-tools/blob/120fa4d48815fc9e87c59374c499849934f2ce10/Makefile
NEWLIB_MULTILIBS_GEN="\
    rv32e-ilp32e--c \
    rv32ea-ilp32e--m \
    rv32em-ilp32e--c \
    rv32eac-ilp32e-- \
    rv32emac-ilp32e-- \
    rv32i-ilp32--c,f,fc,fd,fdc \
    rv32ia-ilp32-rv32ima,rv32iaf,rv32imaf,rv32iafd,rv32imafd- \
    rv32im-ilp32--c,f,fc,fd,fdc \
    rv32iac-ilp32--f,fd \
    rv32imac-ilp32-rv32imafc,rv32imafdc- \
    rv32if-ilp32f--c,d,dc \
    rv32iaf-ilp32f--c,d,dc \
    rv32imf-ilp32f--d \
    rv32imaf-ilp32f-rv32imafd- \
    rv32imfc-ilp32f--d \
    rv32imafc-ilp32f-rv32imafdc- \
    rv32ifd-ilp32d--c \
    rv32imfd-ilp32d--c \
    rv32iafd-ilp32d-rv32imafd,rv32iafdc- \
    rv32imafdc-ilp32d-- \
    rv64i-lp64--c,f,fc,fd,fdc \
    rv64ia-lp64-rv64ima,rv64iaf,rv64imaf,rv64iafd,rv64imafd- \
    rv64im-lp64--c,f,fc,fd,fdc \
    rv64iac-lp64--f,fd \
    rv64imac-lp64-rv64imafc,rv64imafdc- \
    rv64if-lp64f--c,d,dc \
    rv64iaf-lp64f--c,d,dc \
    rv64imf-lp64f--d \
    rv64imaf-lp64f-rv64imafd- \
    rv64imfc-lp64f--d \
    rv64imafc-lp64f-rv64imafdc- \
    rv64ifd-lp64d--c \
    rv64imfd-lp64d--c \
    rv64iafd-lp64d-rv64imafd,rv64iafdc- \
    rv64imafdc-lp64d--"


```

(continues on next page)

(continued from previous page)

```
# Linux install (cross-compile for linux)
# Default value from riscv-gcc repository
GLIBC_MULTILIBS_GEN="\
    rv32imac-ilp32-rv32ima,rv32imaf,rv32imafd,rv32imafc,rv32imafdc- \
    rv32imafdc-ilp32d-rv32imafd- \
    rv64imac-lp64-rv64ima,rv64imaf,rv64imafd,rv64imafc,rv64imafdc- \
    rv64imafdc-lp64d-rv64imafd-"'


# parse arguments
USAGE="$(basename "$0") [-h] [-c] [-n] [-d dir] [-t tag] [-u user] [-p path] -- Clone latested ${PROJ} version and build it. Optionally select compiler (buildtool), build directory and version, install binaries and cleanup setup files.

where:
    -h          show this help text
    -c          cleanup project
    -n          use \"newlib multilib\" instead of \"linux multilib\" cross-compiler
    -e          extend PATH in by RiscV binary path (default: /etc/profile)
    -d dir      build files in \"dir\" (default: ${BUILDFOLDER})
    -t tag      specify version (git tag or commit hash) to pull (default: default branch)
    -u user     install RiscV tools for user \"user\". (default: install globally)
    -p path     choose install path (default: /opt/riscv)"

while getopts ':hcend:t:u:p:' 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)"
                exit 1
            fi
            CLEANUP=true
            echo "-c set: Removing build directory"
            ;;
        e)  EXPORTPATH=true
            echo "-e set: Extending PATH by RiscV binary path"
            ;;
        n)  echo "-n set: Using newlib cross-compiler"
            NEWLIB=true
            TOOLCHAIN_SUFFIX="newlib-multilib"
            ;;
        d)  echo "-d set: Using folder $OPTARG"
            BUILDFOLDER="$OPTARG"
            ;;
        t)  echo "-t set: Using version $OPTARG"
            TAG="$OPTARG"
            ;;
        p)  echo "-p set: Using install path $OPTARG"
            INSTALL_PATH="$OPTARG"
            ;;
        u)  echo "-u set: Installing for user $OPTARG"
            PROFILE_PATH="$(grep $OPTARG /etc/passwd | cut -d ":" -f6)/.profile"

```

(continues on next page)

(continued from previous page)

```
            if [ ! -f "$PROFILE_PATH" ]; then
                echo -e "${RED}ERROR: No .profile file found for user \"${OPTARG}\"${NC}" >&2
                exit 1;
            fi
            ;;
        :)  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))

# 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 {
    # Stable selected: Choose latest tag if available, otherwise use default branch
    if [ "$1" == "stable" ]; then
        local L_TAGLIST=`git rev-list --tags --max-count=1`

        # tags found?
        if [ -n "$L_TAGLIST" ]; then
            local L_COMMIT_HASH="`git describe --tags $L_TAGLIST`"
            git checkout --recurse-submodules "$L_COMMIT_HASH"
        else
            git checkout --recurse-submodules $(git symbolic-ref refs/remotes/origin/HEAD | sed 's@^refs/remotes/origin/@@')
            local L_COMMIT_HASH="$(git rev-parse HEAD)"
            >&2 echo -e "${RED}WARNING: No git tags found, using default branch${NC}"
        fi
    else
        # Either checkout defaut/stable branch or use custom commit hash, tag or branch name
        if [ "$1" == "default" ] || [ "$1" == "latest" ]; then
            git checkout --recurse-submodules $(git symbolic-ref refs/remotes/origin/HEAD | sed 's@^refs/remotes/origin/@@')
            local L_COMMIT_HASH="$(git rev-parse HEAD)"
        else
            # Check if $1 contains a valid tag and use it as the version if it does
            git checkout --recurse-submodules "$1"
            local L_COMMIT_HASH="$(git rev-parse HEAD)"

            for CUR_TAG in `git tag --list`; do
                if [ "$CUR_TAG" == "$1" ]; then
                    L_COMMIT_HASH="$1"
                    break
                fi
            done
        fi
```

(continues on next page)

(continued from previous page)

```
    fi

    # Apply return value
    eval "$2=\"$L_COMMIT_HASH\""
}

# 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
fi

# cleanup files if the programm was shutdown unexpectedly
trap 'echo -e "${RED}ERROR: Script was terminated unexpectedly, cleaning up files..." >&2 && pushd -0 > /dev/null && rm -rf $BUILDFOLDER' INT TERM

# does the config exist?
if [ ! -f "$VERSION_FILE_NAME" ]; then
    echo -e "${RED}Warning: No version.cfg file found. Generating file and using default versions${NC}";
    echo "$VERSION_FILE" > "$VERSION_FILE_NAME"
fi

source "$VERSION_FILE_NAME"
CFG_LOCATION=`pwd -P`

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

pushd $BUILDFOLDER > /dev/null

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

pushd $PROJ > /dev/null
select_and_get_project_version "$TAG" "COMMIT_HASH"
VERSIONLIST="RiscV-GNU-Toolchain-${TOOLCHAIN_SUFFIX}: $COMMIT_HASH"

# fetch versions for all subrepos (as specified in versions.cfg)
while read LINE; do
    if [ -n "$LINE" ] && [ "${LINE:0:1}" != "#" ]; then
        SUBREPO=`echo "$LINE" | sed "s/[=].*$//"`
        if [ -n "${!SUBREPO}" ]; then
            SUBREPO_LOWER=`echo "$SUBREPO" | tr [A-Z,_] [a-z,-]`

            if [ "${!SUBREPO}" != "default" ]; then
                if [ -d "$SUBREPO_LOWER" ]; then
                    pushd $SUBREPO_LOWER > /dev/null
                    git checkout --recurse-submodules ${!SUBREPO}
                    VERSIONLIST="${VERSIONLIST}\n${SUBREPO_LOWER}-${TOOLCHAIN_SUFFIX}: ${!SUBREPO}"
```

(continues on next page)

(continued from previous page)

```
                    popd > /dev/null
                fi
            else
                if [ -d "$SUBREPO_LOWER" ]; then
                    pushd $SUBREPO_LOWER > /dev/null
                    VERSIONLIST="${VERSIONLIST}\n${SUBREPO_LOWER}-${TOOLCHAIN_SUFFIX}
↪: $(git rev-parse HEAD)"
                    popd > /dev/null
                fi
            fi
        fi
    fi
done < "${CFG_LOCATION}/${VERSION_FILE_NAME}"


# build and install if wanted
PATH="${INSTALL_PATH}:${PATH}"

if [ $NEWLIB = true ]; then
    ./configure --prefix=$INSTALL_PATH --enable-multilib --disable-linux
    # activate custom multilibs
    pushd "riscv-gcc/gcc/config/riscv" > /dev/null
    chmod +x ./multilib-generator
    ./multilib-generator $NEWLIB_MULTILIBS_GEN > t-elf-multilib
    popd > /dev/null
    NEWLIB_MULTILIB_NAMES=`echo $NEWLIB_MULTILIBS_GEN | sed "s/-\(rv\(32\|64\)[a-zA-
↪Z]*,*\)*-\([a-zA-Z]*,*\)*//g"`
    echo "Building newlib-multilib for \"$NEWLIB_MULTILIB_NAMES\""
    # build
    make -j$(nproc) NEWLIB_MULTILIB_NAMES="$NEWLIB_MULTILIB_NAMES"
else
    ./configure --prefix=$INSTALL_PATH --enable-multilib --enable-linux
    # activate custom multilibs
    pushd "riscv-gcc/gcc/config/riscv" > /dev/null
    chmod +x ./multilib-generator
    ./multilib-generator $GLIBC_MULTILIBS_GEN > t-linux-multilib
    popd > /dev/null
    GLIBC_MULTILIB_NAMES=`echo $GLIBC_MULTILIBS_GEN | sed "s/-\(rv\(32\|64\)[a-zA-Z]*,
↪*\)*-\([a-zA-Z]*,*\)*//g"`
    echo "Building linux-multilib for \"$GLIBC_MULTILIB_NAMES\""
    # build
    make -j$(nproc) GLIBC_MULTILIB_NAMES="$GLIBC_MULTILIB_NAMES" linux
fi

# extend path
if [ $EXPORTPATH = true ]; then
    PATH_STRING="\n# Add RiscV tools to path
if [ -d \"${INSTALL_PATH}/bin\" ]; then
  PATH=\"${INSTALL_PATH}/bin:\$PATH\"
fi"

    if ! grep -q "PATH=\"${INSTALL_PATH}/bin:\$PATH\"" "$PROFILE_PATH"; then
      echo -e "$PATH_STRING" >> "$PROFILE_PATH"
    fi
fi

# return to first folder and store version
```

(continues on next page)

(continued from previous page)

```
pushd -0 > /dev/null
echo -e "$VERSIONLIST" >> "$VERSIONFILE"

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

6.12.3 versions.cfg

```
## Define sourcecode branch

# default = use predefined versions from current riscv-gnu-toolchain branch
# or any arbitrary git tag or commit hash
# note that in most projects there is no master branch
QEMU=default
RISCV_BINUTILS=default
RISCV_DEJAGNU=default
RISCV_GCC=default
RISCV_GDB=default
RISCV_GLIBC=default
RISCV_NEWLIB=default


## Define which RiscV architectures and ABIs are supported (space seperated list
↪"arch-abi")

# Taken from Sifive:
# https://github.com/sifive/freedom-tools/blob/
↪120fa4d48815fc9e87c59374c499849934f2ce10/Makefile
NEWLIB_MULTILIBS_GEN="\
    rv32e-ilp32e--c \
    rv32ea-ilp32e--m \
    rv32em-ilp32e--c \
    rv32eac-ilp32e-- \
    rv32emac-ilp32e-- \
    rv32i-ilp32--c,f,fc,fd,fdc \
    rv32ia-ilp32-rv32ima,rv32iaf,rv32imaf,rv32iafd,rv32imafd- \
    rv32im-ilp32--c,f,fc,fd,fdc \
    rv32iac-ilp32--f,fd \
    rv32imac-ilp32-rv32imafc,rv32imafdc- \
    rv32if-ilp32f--c,d,dc \
    rv32iaf-ilp32f--c,d,dc \
    rv32imf-ilp32f--d \
    rv32imaf-ilp32f-rv32imafd- \
    rv32imfc-ilp32f--d \
    rv32imafc-ilp32f-rv32imafdc- \
    rv32ifd-ilp32d--c \
    rv32imfd-ilp32d--c \
    rv32iafd-ilp32d-rv32imafd,rv32iafdc- \
    rv32imafdc-ilp32d-- \
    rv64i-lp64--c,f,fc,fd,fdc \
    rv64ia-lp64-rv64ima,rv64iaf,rv64imaf,rv64iafd,rv64imafd- \
    rv64im-lp64--c,f,fc,fd,fdc \
    rv64iac-lp64--f,fd \
```

(continues on next page)

(continued from previous page)

```
    rv64imac-lp64-rv64imafc,rv64imafdc- \
    rv64if-lp64f--c,d,dc \
    rv64iaf-lp64f--c,d,dc \
    rv64imf-lp64f--d \
    rv64imaf-lp64f-rv64imafd- \
    rv64imfc-lp64f--d \
    rv64imafc-lp64f-rv64imafdc- \
    rv64ifd-lp64d--c \
    rv64imfd-lp64d--c \
    rv64iafd-lp64d-rv64imafd,rv64iafdc- \
    rv64imafdc-lp64d--"

# Linux install (cross-compile for linux)
# Default value from riscv-gcc repository
GLIBC_MULTILIBS_GEN="\
    rv32imac-ilp32-rv32ima,rv32imaf,rv32imafd,rv32imafc,rv32imafdc- \
    rv32imafdc-ilp32d-rv32imafd- \
    rv64imac-lp64-rv64ima,rv64imaf,rv64imafd,rv64imafc,rv64imafdc- \
    rv64imafdc-lp64d-rv64imafd-"
```

6.13 trellis

6.13.1 install_trellis.sh

```
#!/bin/bash

# Author: Harald Heckmann <mail@haraldheckmann.de>
# Date: Jun. 25 2020
# Project: QuantumRisc (RheinMain University) <Steffen.Reith@hs-rm.de>

# constants
RED='\033[1;31m'
NC='\033[0m'
REPO="https://github.com/SymbiFlow/prjtrellis"
PROJ="prjtrellis/libtrellis"
BUILDFOLDER="build_and_install_trellis"
VERSIONFILE="installed_version.txt"
TAG="latest"
INSTALL=false
INSTALL_PREFIX="default"
CLEANUP=false


# parse arguments
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.

where:
    -h          show this help text
    -c          cleanup project
    -d dir      build files in \"dir\" (default: ${BUILDFOLDER})
    -i path     install binaries to path (use \"default\" to use default path)
```

(continues on next page)

(continued from previous page)

```
    -t tag      specify version (git tag or commit hash) to pull (default: Latest tag)"


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

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)"
                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 $OPTARG"
            TAG="$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))"

# 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 {
    # Stable selected: Choose latest tag if available, otherwise use default branch
    if [ "$1" == "stable" ]; then
        local L_TAGLIST=`git rev-list --tags --max-count=1`

```

(continues on next page)

(continued from previous page)

```
        # tags found?
        if [ -n "$L_TAGLIST" ]; then
            local L_COMMIT_HASH="`git describe --tags $L_TAGLIST`"
            git checkout --recurse-submodules "$L_COMMIT_HASH"
        else
            git checkout --recurse-submodules $(git symbolic-ref refs/remotes/origin/HEAD | sed 's@^refs/remotes/origin/@@')
            local L_COMMIT_HASH="$(git rev-parse HEAD)"
            >&2 echo -e "${RED}WARNING: No git tags found, using default branch${NC}"
        fi
    else
        # Either checkout defaut/stable branch or use custom commit hash, tag or branch name
        if [ "$1" == "default" ] || [ "$1" == "latest" ]; then
            git checkout --recurse-submodules $(git symbolic-ref refs/remotes/origin/HEAD | sed 's@^refs/remotes/origin/@@')
            local L_COMMIT_HASH="$(git rev-parse HEAD)"
        else
            # Check if $1 contains a valid tag and use it as the version if it does
            git checkout --recurse-submodules "$1"
            local L_COMMIT_HASH="$(git rev-parse HEAD)"

            for CUR_TAG in `git tag --list`; do
                if [ "$CUR_TAG" == "$1" ]; then
                    L_COMMIT_HASH="$1"
                    break
                fi
            done
        fi
    fi

    # Apply return value
    eval "$2=\"$L_COMMIT_HASH\""
}

# 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
fi

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

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

pushd $BUILDFOLDER > /dev/null

if [ ! -d "$PROJ" ]; then
```

(continues on next page)

(continued from previous page)

```
    git clone --recursive "$REPO" "${PROJ%%/*}"
fi

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

# build and install if wanted
if [ "$INSTALL_PREFIX" == "default" ]; then
    cmake -DCMAKE_INSTALL_PREFIX=/usr .
else
    cmake -DCMAKE_INSTALL_PREFIX="$INSTALL_PREFIX" .
fi

make -j$(nproc)

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

# return to first folder and store version
pushd -0 > /dev/null
echo "${PROJ##*/}: $COMMIT_HASH" >> "$VERSIONFILE"

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

6.13.2 install_trellis_essentials.sh

```
#!/bin/bash

# Author: Harald Heckmann <mail@haraldheckmann.de>
# Date: Jun. 25 2020
# Project: QuantumRisc (RheinMain University) <Steffen.Reith@hs-rm.de>

# 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="build-essential clang cmake python3 python3-dev libboost-all-dev git"

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

CHAPTER

SEVEN

FURTHER REFERENCES

- Project page
- VM page
- Further reads

CHAPTER

EIGHT

CHANGELOG

8.1 1.0.0

First release