Overview of Chipyard

Dennis Frank

Installation of Chipyard

WORK IN PROGRESS

The installation of Chipyard is described in `https://chipyard.readthedocs.io/en/latest/Chipyard-Basics/Initial-Repo-Setup.html`. Some additional steps are added by ChatGPT. These steps were performed on a system with Ubuntu 24.04.3 LTS, a differnt configuration could require additional steps!

1. Step: Dependencies:

Scala:

```
    curl -O -L https://github.com/chipsalliance/chisel/releases/latest/download/
    chisel-example.scala

    curl -sSLf https://scala-cli.virtuslab.org/get | sh
```

Termurin:

```
    sudo apt install -y wget gpg apt-transport-https

    echo "deb https://packages.adoptium.net/artifactory/deb
    $(awk -F= '/^VERSION_CODENAME/{print$2}' /etc/os-release) main" |
    sudo tee /etc/apt/sources.list.d/adoptium.list

    wget -qO - https://packages.adoptium.net/artifactory/api/gpg/key/public |
    gpg --dearmor | sudo tee /etc/apt/trusted.gpg.d/adoptium.gpg > /dev/null

    apt update

    apt install temurin-17-jdk
```

Ubuntu Packages:

```
sudo apt-get install autoconf automake autotools-dev curl libmpc-dev libmpfr-dev libgmp-dev libusb-1.0-
sudo apt install libguestfs-tools
```

The following steps are neccesary:

```
    wget https://repo.anaconda.com/miniconda/Miniconda3-latest-Linux-x86_64.sh

    bash Miniconda3-latest-Linux-x86_64.sh

    source ~/miniconda3/etc/profile.d/conda.sh
```

After this clone the repository from chipyard and execute the installation, caution a lot of space is needed! Follow the steps:

```
    git clone https://github.com/ucb-bar/chipyard.git

 cd chipyard

 ./scripts/init-submodules-no-riscv-tools.sh

 source  ./env.sh

 ./build-setup.sh riscv-tools
```

Simulation

RTL Simulation

The RTL simulation is done with **Verilator**. With this simulation the actual hardware architecture of Rocket Chip is simulated. Every instruction is executed just like the hardware working in cycles. That's why the behavior is cycle-accurate. The main disadvantage is, that the simulation takes a lot of time, especially for more komplex code or bigger software.

First of all you will generate an executable which is used to simulate the chip, in this case the Rocket Chip (but there are other configurations available).
The following command generates the executable, use this command in the chipyard directory. After this the executable can be found within chipyard/sims/verilator:

```
make -C sims/verilator verilog CONFIG=RocketConfig
```

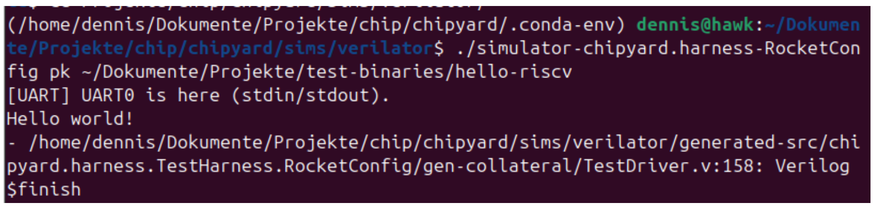
After this it is possible to simulate own code written in C/C++. Therefore you need to compile your code to code that is useable for the simulation. In general you can use the following command, it will create another file with ".elf" as ending:

```
riscv64-unknown-elf-gcc -o <name>.elf <name>.c
```

Now you can use the generated file within the simulation. For this enter sims/verilator and execute the following command:

```
./simulator-chipyard.harness-RocketConfig pk
    "the-whole-path-to-your-.elf-file-starting-at-home-directory"
```

The last command includes "pk" which stands for "Proxy Kernel". It is a tiny OS provided by the RISCV group and handles basic tasks like providing an evironment to deal with system calls. With these steps it is possible for example to write a simple "Hello World" program an run it with the verilator simulation. The result could look like this:



```
(/home/dennis/Dokumente/Projekte/chip/chipyard/.conda-env) dennis@hawk:~/Dokumen
te/Projekte/chip/chipyard/sims/verilator$ ./simulator-chipyard.harness-RocketCon
fig pk ~/Dokumente/Projekte/test-binaries/hello-riscv
[UART] UART0 is here (stdin/stdout).
Hello world!
- /home/dennis/Dokumente/Projekte/chip/chipyard/sims/verilator/generated-src/chi
pyard.harness.TestHarness.RocketConfig/gen-collateral/TestDriver.v:158: Verilog
$finish
```

There are already different examples of code within chipyard, that can be run by verilator. More about this can be found under (2.1.5):
https://chipyard.readthedocs.io/en/stable/Simulation/Software-RTL-Simulation.html
It is possible to run the example code with the last command mentioned above. There is actually no use of "pk", you can leave this out in this case.

NOTE: It could be neccesary to execute *cmake .* after executing *make* in the "tests" directory of chipyard. After this you can go on with the instructions in the documentation.

Functional Simulation

This simulation can be accomplished with **Spike** (or with Qemu). In this case only the RISCV instruction-architecture is simulated. This means that the instructions will be executed following the RISCV specification. It shows if the simulated program works correctly in terms of logic and architecture.
The command to use spike is:
spike pk "the-whole-path-to-your-.elf-file-starting-at-home-directory"

You don't need to add the whole path if you are in the directory which contains the file you want to execute. In this case you just need to add the file. If you want to execute the example code in tests from the chipyard directory with spike, leave out the "pk".

Algorithms in Simulation

Fast Fourier Transformation

The Fast Fourier Transformation was implemented with the help of FFTW (`https://www.fftw.org/`). In my case, it was neccesary to install it manually with the following commands:

```
wget http://www.fftw.org/fftw-3.3.10.tar.gz

tar xzf fftw-3.3.10.tar.gz

cd fftw-3.3.10

export CC=riscv64-unknown-elf-gcc

export AR=riscv64-unknown-elf-ar

export RANLIB=riscv64-unknown-elf-ranlib

export HOST=riscv64-unknown-elf

./configure --host=$HOST --enable-static --disable-shared --prefix=$(pwd)/install

make -j4

make install
```

To follow the instructions you need the following files: image.py, fft_output.txt, image_data.h, fftw.c, gen_freq_pic.c and an image (e.g. test.png). All of these files should be in the same directory.
The first step is to use the Python script to transform a picture into an array of values with the following command. You will be asked for a name of an image. You have to enter the whole name of the image including the ending (e.g. ".png").

```
python3 image.py
```

The data created by the python script is added in the "image_data.h" file. Everytime you choose a new image, the data in this file will be overwritten.
After you generated the data for the image, you can compile the FFT code with the following

command:

```
riscv64-unknown-elf-gcc -static
-I./fftw-3.3.10/install/include
-L./fftw-3.3.10/install/lib
-o fftw.elf fftw.c -lfftw3 -lm
```

This will generate the executeable file for the simulation. Note that the conda environment needs to be activated and execute "source env.sh" in the chipyard directory. In the next step you can simulate this with spike:

```
spike pk fftw.elf
```

The output of the execution is written to the file "fft_output.txt", which contains a huge amount of lines.
The code in "gen_freq_pic.c" serves the visualiztion of the simulation output. Therefore first compile this code with (only neccesary once):

```
gen_freq_pic.c -o gen_freq_pic -lfftw3 -lm
```

And then execute with:

```
./gen_freq_pic
```

Edge Detection

To perform edge detection in simulation you need the following files in the same directory: image.py, edge.c, image_data.h, stb_image.h, stb_image_write.h and an image (like test.png).
The files stb_image.h and stb_image_write.h are added via:

```
wget https://raw.githubusercontent.com/nothings/stb/master/stb_image.h
wget https://raw.githubusercontent.com/nothings/stb/master/stb_image_write.h
```

The first step is to use the Python script to transform a picture into an array of values with the following command. You will be asked for a name of an image. You have to enter the whole name of the image including the ending, in this case the endig needs always to be ".jpg" for the simulation.

```
python3 image.py
```

The data created by the python script is added in the "image_data.h" file. Everytime you choose a new image, the data in this file will be overwritten.
After you generated the data for the image, you can compile the FFT code with the following command:

```
riscv64-unknown-elf-gcc -static  -o edge.elf edge.c  -lm
```

This will generate the executeable file for the simulation. Note that the conda environment needs to be activated and execute "source env.sh" in the chipyard directory. In the next step you can simulate this with spike:

```
spike pk edge.elf
```

The simulation will automatically generate a new image which shows the the result of the edge detection. The image for the result is saved in "edges.png".

Optical Flow

- measuring the movement of object in a scene, approximation of movement between to frames
- assumptions are neccesary:
1. brightness of an image point remains constant over time
2. displacemant and time step are very small
- maybe use of OpenCV
- Lucas Kanade Method: Assumption: for each pixel assume Motionfield (hence Optical Flow) is constant within a small neighborhood

Further Notes

Use Rocket Chip on FPGA:
- Rocket Chip can be configured but as defualt it has AXI connections (and a debug connection)
- it is possible to define a differnt configuration and with this you can change periphery
- it is neccessary to convert the connections (e.g. AXI to PCIe) when using a FPGA
- firesim has predefined configurations for specfic FPGAs

source: `https://github.com/chipsalliance/rocket-chip/issues/1594` - use of "Vivado" (Installation neccessary)
- within chipyard it is possible to build a bitstream

-> chisel is converted to verilog, this runs through vivado to create a bitstream