Ach, ne! - WS2017/18

Design Documentation

Andre Brand, Matthis Keppner, Mike Wüstenberg, Sebastian Brückner

Version 0.4

Revision History

Revision	**Date**	**Author(s)**	**Description**
0.1	02.10.2017	Sebastian Brückner	Template
0.2	02.10.2017	Sebastian Brückner	Added CPU Selection
0.3	32.10.2017	Andre Brand	Added OS Selection
0.4	13.02.2018	Mike Wüstenberg	Added riscv-poky section
0.4	23.10.2017	Matthis Kepp-ner	Added Guide for plan B

Contents

1 Introduction

This project aims to develop a CPU and OS for educational purpose. Main goal is to create a CPU/OS combination wich demonstrates OS and Computer architecture, but is still relatively simple to understand. The System should be capable of running C/Assembly written programs with and without the OS and provide a least a shell. To make this possible, there should be a cross compiler/assembler and toolset to flash the programs.

The Project originates at the HAW Hamburg in the course of creating a new Major.

2 CPU Architecture Selection

2.1 RISC vs CISC

The CPU in this project is an RISC CPU, because

- a CISC CPU might be to complex to be easily understood
- modern CPU tend to be RISC CPUs
- a CISC might be to complex to implement within the scope of this project

2.2 Instruction sets

There are plenty of choices when it comes to instruction sets, this list compiles the most obvious.

- ARM
- Thumb-2
- AVR
- OpenRisc
- RISC-V

While ARM (and Thumb-2) are well documented, the ARM is not a RISC by the book. Also there a plenty of ARM devices around and this project aims to break fresh ground.

AVR is not really suited for an CPU that is supposed to run an OS. Its better suited for small microcontrollers.

Also these architectures are nor open source. This is the main advantage of OpenRISC and RISC-V.

RISC-V was chosen over OpenRISC because:

- a very good documentation
- modular Instruction Set Architecture with multiple well defined extensions
- existing compiler (gcc toolchain)
- existing QEMU simulator to develop the OS on

- not many existing CPU designs (being capable to break fresh ground)
- the RISC-V Foundation has strong backers (e.g. Google and IBM) [1]

[1]https://riscv.org/membership/

3 OS Selection

3.1 Operating System

For our project we need an Operating System which is free, doesn't need many ressources and is easy to extend. Because we don't have the ability and time to write our own OS, we decided to use an existing open source OS.

3.2 Available Operating Systems

In our first meeting we figured out which Operating Systems are available for our project and the outcome was:

- Linux
- BSD
- Riot
- Minix

Linux and FreeBSD are way to huge and it would take to much time to customize it for our needs.
Riot as an OS which is partly invented at the HAW would be a great one to use.
But we decided to use the Minix OS because it is very small and light weighted, well documented and easy to extend.
Aditionally Minix as an Operating System for teaching perfectly fits to our project.

3.3 Update

During our third meeting we figured out, that there already is an existing Linux-Port for the RISC-V architecture. So we decided to use that Linux-Version for our project, but also we want to keep the idea of a running minix system for our project. Moreover we planned to build our own HAW-Shell for that system.
Due to our investigations we figured out, that a minix-portation would be far to expensive. First we have to make shure to understand the whole build-system of the minix-architecture, the code-dependencies can be found at:

- `http://svn.freebsd.org/base/head/sys/conf/files.riscv`

Makefile:

- `http://svn.freebsd.org/base/head/sys/conf/Makefile.riscv`

LDscript:

- `http://svn.freebsd.org/base/head/sys/conf/ldscript.riscv`

Options:

- `http://svn.freebsd.org/base/head/sys/conf/options.riscv`

4 OS Riscv Tools

4.1 Operating System

To compile and test a linux operating system based on riscv32i, we need to install some tools to get started.

4.2 Packages needed

4.2.1 Ubuntu

```
$ sudo apt-get install autoconf automake autotools-dev curl
 device-tree-compiler libmpc-dev libmpfr-dev libgmp-dev
 libusb-1.0-0-dev gawk build-essential bison flex texinfo
 gperf libtool patchutils bc zlib1g-dev pkg-config git ncurses-dev openssl
```

4.2.2 Fedora

```
$ sudo dnf install autoconf automake @development-tools curl
 dtc libmpc-devel mpfr-devel gmp-devel gawk build-essential
 bison flex texinfo gperf libtool patchutils bc zlib-devel
```

4.3 Installing the toolchain

Choose a place in your hard drive with big expanse space, and let's call that $TOP. Change to the directory you want to install in, and then set the $TOP environment variable accordingly:

```
$ export TOP=$(pwd)
```

4.3.1 Toolchain components

- riscv-gnu-toolchain, a RISC-V cross-compiler
- riscv-pk, a proxy kernel that services system calls generated by code built and linked with RISC-V Newlist port (this does not apply to Linux, as it handles the system calls)
- riscv-tests, a set of assembly tests and benchmarks
- others,...

4.3.2 Obtaining and compiling the sources

Change directory and clone the tools from the riscv-tools GitHub repository:

```
$ cd $TOP
$ git clone https://gitub.com/riscv/riscv-tools.git
```

Updating submodules

```
$ cd $TOP/riscv-tools
$ git submodule update --init --recursive
```

Before we start installation, we need to set the $RISCV environment variable.
The variable is used throughout the build script process to identify where to install the new tools.
(This value is used as the argument to the –prefix configuration switch.)

```
$ export RISCV=$TOP/riscv
```

Add bin folder to $PATH variable

```
$ export PATH=$PATH:$RISCV/bin
```

Now, we can run the build script. You may edit it for your needs. Change –host=riscv64-unknown-elf to –host=riscv32-unknown-elf for building 32 bit compiler. There is also a special instruction set script, you may edit this and use it instead. If you wanna use 32 bit with only integer instructions set, you may edit the file "build-rv-32ima.sh" and the change the commands: –with-isa and –with-arch like this: –with-isa=rv32i –with-arch=rv32i Finally run the "build-rv-32ima.sh" script

```
$ ./build-rv-32ima.sh
```

4.3.3 Testing your toolchain

Now when you installed the toolchain, we can start by testing it with a simple "Hello world" programm. Exit the riscv-tools directory and write you "Hello world" programm.

```
$ cd $TOP
$ echo -e '#include <stdio.h>\n int main(void) { printf("Hello world!\\n");
  return 0; }' > hello.c
```

Then, build your programm with riscv32-unknown-elf-gcc.

```
$ cd $TOP
$ riscv-32-unknow-elf-gcc -o hello hello.c
```

The “Hello wolrd” programm involves a system call, which couldn’t be handled by out x86 system.
We’ll have to run it within the proxy kernel ’pk’, this can be run via spike, the RISC-V architectural simulator. Run your “hello wolrd” programm via spike like so:

```
$ spike pk hello
```

4.4 Installing Linux/RISC-V

4.4.1 Building riscv64-unknown-linux-gnu-gcc

The cross-compiler is used to build binaries linked to GNU C Library (glibc) insteadof the Newlibi libary. To build linux you can use the already installed riscv32-unknown-elf-gcc, but for cross-compiling applications, we now build the riscv-32-unknown-linux-gnu-gcc.

First we need to enter the riscv-gnu-toolchain directory und run the configure script to generate the “Makefile”.

Here if you wanna use the 32 bit version with only integer instructions set, you have to add options like so.

```
$ ./configure --with-arch=rv32i --prefix=$RISCV
```

4.5 Building linux kernel

Get the linux kernel source code. (This way tested on version 4.15-rc3)

```
$ cd $TOP
$ git clone https://github.com/riscv/riscv-linux.git
```

Configuring the linux kernel
Applying default configuration and edit it with a text-based GUI (ncurses)
Don’t forget the CROSS COMPILE option!!

```
$ make -j4 ARCH=riscv defconfig
       CROSS_COMPILE=$RISCV/bin/riscv32-unknown-linux-gnu-
$ make -j4 ARCH=riscv menuconfig
       CROSS_COMPILE=$RISCV/bin/riscv32-unknown-linux-gnu-
```

When done configuration, you can now start building the kernel. -j [number] is optional and it uses multiple cores to speed up the process.

```
$ make -j4 ARCH=riscv
       CROSS_COMPILE=$RISCV/bin/riscv32-unknown-linux-gnu- vmlinux
```

The created vmlinux is the cross-compiled linux kernel image.

5 Riscv Poky

5.1 Poky

Riscv-poky is a port of the Yocto project. It's a full Linux distribution which enables user to cross-compile and package software as file system.

5.1.1 Yocto project

The open source project Yocto provides tools, templates and methods to create Linux based systems. The package creation is based on recipe. A recipe describes how to cross-compile and fetch a package. This allows to build recipes for a wide range of build targets. The complete Linux images is build with the BitBake tool.

5.1.2 BitBake

BitBake is a efficient execution engine which works in parallel and executes Python tasks. The instructions on what tasks BitBake should run are stored in recipe(.bb), configuration(.conf) and class(.bbclass) files.
The recipes includes:

- Description
- Recipe version
- Dependencies
- Where to get the Source code
- patch for the source code
- Compile instructions
- install location

A list of all BitBake recipe versions can be shown with the "bitbake -s" command.

The Configuration file defines various configurations. This includes user configuration options, compiler tuning options and more. The Class files contains information which are useful the to share between files. For more information take a look at the Yocto documentation.

- Yocto Project: http://www.yoctoproject.org/docs/current/ref-manual/ref-manual.html

- BitBake: http://www.yoctoproject.org/docs/1.6/bitbake-user-manual/bitbake-user-manual.html

5.2 Quickstart

First donwload the riscv-poky repository and initialize the OpenEmbedded environment.

```
git clone https://github.com/riscv/riscv-poky.git
cd riscv-poky
source oe-init-build-env
```

This takes you to the build directory. The ”conf/local.conf” file is preconfigured to build for riscv64. In case you want to use a already installed tool-chain you can change the configuration. Once the compilation has finished you will find is inside the ”tmp” folder. To start the build process run the following command.

```
bitbake core-image-riscv
```

This will build a a riscv-v image which includes Python, Perl, SSH and other tools. Since this will download a large amount of data this can take several hours. But you can on completion run a fully operational Linux. Test your Linux with the following command.

```
runspike riscv64
```

5.3 Adding recipes

To add a recipe to your image you need to change the image .bb file. The file can be found in the following path.

```
/riscv-porky/meta-riscv/recipes-core/image
```

The default image has the name ”core-image-riscv.bb”. Copy the default which creates a basic Linux. Now go edit the new file. You can add new recipes with.

```
IMAGE_INSTALL += "git"
```

As long as there is a recipe for the package you want to install this will install it on your new image. Build your Linux image with the ”bitbake” command and the new image recipe you just created. BitBake will use previous build software to speed up the process.

5.4 Troubleshooting

5.4.1 Failed to fetch URL

In case a commit got orphaned and is not ”visible” from any branch you can add nobranch=1; add to the fetch command.
Source: https://github.com/riscv/riscv-poky/issues/15

```
WARNING: qemu-native-2.5.0-r0 do_fetch: Failed to fetch URL gitsm://github.com/riscv/risc
ERROR: qemu-native-2.5.0-r0 do_fetch: Fetcher failure: Unable to find revision 9bfcd4717b
ERROR: qemu-native-2.5.0-r0 do_fetch: Fetcher failure for URL: 'gitsm://github.com/riscv/
ERROR: qemu-native-2.5.0-r0 do_fetch: Function failed: base_do_fetch
ERROR: Logfile of failure stored in: /home/user/RISC-V/riscv-poky/build/tmp/work/x86_64-l
ERROR: Task (virtual:native:/home/user/RISC-V/riscv-poky/build/../meta-riscv/recipes-devt

In the file meta-riscv/recipes-devtools/qemu/qemu_2.5.0.bb replace SRC_URI as follows:
SRC_URI = "git://github.com/riscv/riscv-qemu.git;nobranch=1;destsuffix=${S}
```

6 Quick guide for Plan B

For Plan B we are using an existing implementation for an RISC-V architecture with a linux distribution. This is find in the following repository. https://github.com/ucb-bar/fpga-zynq

6.1 Step one

Clone the repository.

6.2 Step two

Switch to zedboard folder in terminal and run the command make fetch-images.

6.3 Step three

Switch to the fpga-images-zedboard folder and copy *boot.bin, devicetree.dtb, uImage and uramdisk.image.gz* to a **fap32** formatted sd-card.

6.4 Step four

Set the Jumper Position of JP11-JP7 to the following position to set as SD-BOOT mode:

JP11	0
JP10	1
JP9	1
JP8	0
JP7	0

Connect the micro-usb to the uart connector and power up the zedboard.

6.5 Step five

Run the following command:

screen /dev/tty.usbmodem1411115200,cs8,-parenb,-cstopb

The linux should power on on the ARM Processor. The username and the password are **root**.

6.6 Step six

Run the following command to start a linux on the RISC-V processor:

```
./fesvr-zynq bbl
```