Getting started with ORPSoC on the ATLYS board

Anton Fosselius, Per Lenander

June 1, 2012

Contents

1 Introduction **3**

2 Hardware **3**

3 Installing Software **3**

3.1 Xilinx ISE 3

3.2 Subversion and Git 4

3.3 ORPSoC 4

3.4 ork1sim 4

3.5 OpenRISC toolchain 5

3.6 SPI Flash 5

3.7 Board specific library for bare metal 6

3.8 Toolchain Done! 6

4 Building and Flashing **6**

4.1 Bare metal ”hello world” 7

4.2 Running the ”hello world” on or1ksim 7

4.3 Running the ”hello world” on the FPGA 7

4.4 Building the Linux kernel 8

4.5 Running the Linux kernel in or1ksim 8

4.6 Running the Linux kernel on the FPGA 8

4.6.1 Bad way 8

4.6.2 Good way 9

5 Extra **10**

5.1 UART 10

5.2 Adding a RTL Module 10

5.3 Bare metal light LEDs in assembly 10

5.4 DDR2 RAM 11

5.5 Memory Mapping 12

6 References **12**

1 Introduction

This document will explain how to set up an development environment in Ubuntu Linux for ORPSoC on the Atlys development board. It will give detailed descriptions on where to get the correct software and how to use it.

The document contains various examples on how to configure the hardware and software to work correctly with the ORPSoC build system, from simply getting a led to blink to compiling a custom Linux kernel to run on the OpenRISC processor.

2 Hardware

To be able to follow every step of this article you will need the following hardware:

- One Computer with a recent edition of Ubuntu installed (11.10 or later)
- One Atlys FPGA board
- One Computer with Windows XP or later installed

The Windows computer is only used to do SPI Flash with the application Adept from Digilent. The authors of this document have to their regret not found any reliable way to do this in Linux. The Xilinx iMPACT programming tool can be used to program the FPGA itself, but is very unreliable and slow when programming the SPI Flash.

3 Installing Software

To be able to boot Linux on ORPSoC with the Atlys board, you will have to install a wide range of tools. This will take some time and lot of disk space (close to 20GB).

3.1 Xilinx ISE

Note: The trial version of Xilinx ISE does not work because it do not allow you to generate bitstreams.

The first thing you want to do is to sign up at Xilinx homepage and download the ISE webpack edition. The ISE webpack file is about 6Gbyte big, it will take a while to download. When the ISE is installed it will take about 13-15Gbyte of disk space. Make sure you have enough disk space available!

http://www.xilinx.com/products/design-tools/ise-design-suite/ise-webpack.html

Now when the ISE is downloading its time to download a webpack license, this is also done on the Xilinx homepage. When the ISE download is finished run the installer as ”sudo” and follow the instructions.

3.2 Subversion and Git

To follow this document you need to have both subversion and git installed on your computer.

```
sudo apt-get install subversion git
```

3.3 ORPSoC

Checkout the ORPSoC directory from OpenCores:

```
git clone git://git.openrisc.net/stefan/orpsocv2
```

You have to build the PDF file first to be able to read the documentation for orpsocv2. First cd into the orpsocv2/doc folder and type the following:

```
./configure
make pdf
```

You can then read the pdf by running:

```
evince orpsoc.pdf
```

3.4 ork1sim

To install the OpenRISC toolchain we first need to install the or1ksim simulator. First check out the subversion repository to get the latest version.

```
svn co http://opencores.org/ocsvn/openrisc/openrisc/trunk/or1ksim
```

Create a build folder and cd into it.

```
mkdir builddir_or1ksim
cd builddir_or1ksim
```

Now build and install the or1ksim with the following commands:

```
../or1ksim/configure –target=or32-elf –prefix=/opt/or1ksim
make all
make install
export PATH=/opt/or1ksim/bin:$PATH
```

To test the Simulator you will need to install the OpenRISC toolchain. If DejaGNU and the OpenRISC GNU tool chain are installed, the build can be tested as follows.

```
make check
```

All tests should pass.

3.5 OpenRISC toolchain

Checkout the gnu-src subversion directory:

```
svn co http://opencores.org/ocsvn/openrisc/openrisc/trunk/gnu-src
```

We can build uClibc and Linux along with the toolchain. This will download more then 500mb, it might take a while.

```
cd gnu-src
git clone git://git.openrisc.net/jonas/uClibc
git clone git://git.openrisc.net/stefan/linux
```

You will need a bunch of other tools to be able to build everything correctly.

```
sudo apt-get -y install build-essential make gcc g++ flex bison patch
texinfo libncurses-dev libmpfr-dev libgmp3-dev libmpc-dev libzip-
dev iverilog
```

Now we got everything! run the following command:

```
./bld-all.sh –force –prefix /opt/openrisc –or1ksim-dir /opt/or1ksim
–uclibc-dir uClibc –linux-dir linux
```

Add the following to your .bashrc file located in (/home/USERNAME/.bashrc). Open .bashrc with a editor and then add the export statement at the end of the file.

```
gedit .bashrc
```

Add this:

```
export PATH=$PATH:/opt/openrisc/bin
```

Start a new terminal and type "or" and double tab. If everything works you will see a list of or32-elf and or32-linux tools.

3.6 SPI Flash

Note: You can try to use SPI Flash with iMPACT in Linux, for this to work at all you will have to set the JP11 jumper (next to the USB port). The SPI Flash will now start but it is SLOW and will most likely fail.

This instructions are for installing Digilent Adept in Windows, no good solution for Linux have been found.

You need to download and install Digilent Adept and the Digilent plugin. Adept can be downloaded from:

http://www.digilentinc.com/Products/Detail.cfm?Prod=ADEPT

The plugin can be downloaded from:

http://www.digilentinc.com/Products/Detail.cfm?NavPath=2,66,768&Prod=DIGILENT-PLUGIN

Run the installs and follow the instructions.

For Linux to recognise the board we have to set some udev rules.

Copy the udev-rules from the xilinx folder to your udev folder.

```
sudo cp /xilinix/13.4/ISE_DS/common/bin/lin64/digilent/\
digilent.adept.runtime_2.7.4-x86_64/52-digilent-usb.rules /etc/udev/rules.d/

sudo cp /opt/Xilinx/13.4/ISE_DS/common/bin/lin64/\
xusbdfwu.rules /etc/udev/rules.d/

sudo /etc/init.d/udev restart
```

Done!

3.7 Board specific library for bare metal

The or32-elf toolchain needs to know how your board is structured to properly compile elfs for it. **NOTE:** These steps are not needed to build Linux applications, only bare metal. If your board is supported here[1] you don't need to take these steps, just make sure to call or32-elf-gcc/g++ with the correct flags.

In the gnu-src/newlib-1.18.0/libgloss/or32 folder of the toolchain, find the ml501.S file. This file contains basic definitions for the board, and will be used to build libboard.a. Make a copy of the file called atlys.S and edit the copy with the following changes:

- _board_mem_size is the size of connected RAM. In the case of the atlys, it should be changed to 0x8000000 (128MB DDR2).
- _board_clk_freq is the clock frequency in hertz. For the atlys, it should be 50000000.

UART settings should already be correct.

Build libboard.a from the file and move it to the path where the OpenRISC toolchain is installed:

```
or32-elf-as -o atlys atlys.S
or32-elf-ar -q libboard.a atlys
sudo mkdir /opt/openrisc/or32-elf/lib/boards/atlys
sudo cp libboard.a /opt/openrisc/or32-elf/lib/boards/atlys/
```

This process should be similar for other boards, check their specification for correct memory size and clock speed.

3.8 Toolchain Done!

Now you have a working toolchain! congratulations!

4 Building and Flashing

When building and flashing your FPGA or your flash you will be faced with some different file types.

[1]http://opencores.org/openrisc,newlib_toolchain

- Makefile, this file contains building instructions.
- file.v is a Verilog source code file
- file.vhd is a VHDL source code file
- file.bit is a Xilinx bitstream file, The .bit is used to program the FPGA
- file.mcs is a Intel MCS-86 Hexadecimal Object. The MCS can be loaded to the flash.

4.1 Bare metal "hello world"

Writing a hello world for the processor is simple:

FIX: REDO EXAMPLE (NEEDS PRINTF.H)

```
#include "printf.h"

int main()
{
    printf("Hello world!\n");
    return 0;
}
```

This file needs the board-specific headers to compile.

Build it with:

```
or32-elf-gcc -mboard=atlys hello.c -o hello
or32-elf-objcopy -O binary hello
bin2binsizeword hello hello-bsw.bin
```

Note: for the -mboard=atlys flag to work, libboard.a has to have been built and moved to the correct directory, see section 3.7.

Load the hello-bsw.bin to the simulator or the FPGA as described below.

4.2 Running the "hello world" on or1ksim

Change your directory to your or1ksim folder and run:

```
./sim -f sim.cfg /path/to/your/file/hello
```

Now you will see a lot of funny text and if you look closely you will see a "hello world" in the middle of all the output.

4.3 Running the "hello world" on the FPGA

```
make orpsoc.mcs BOOTLOADER_BIN=hello-bsw.bin
```

Load the mcs and reset the FPGA.

4.4 Building the Linux kernel

To get a linux kernel specific for the OpenRISC platform and the Atlys board, check out:

```
git clone git://git.openrisc.net/stefan/linux
```

Build the kernel by entering:

```
make atlys_defconfig
make ARCH=openrisc menuconfig
make ARCH=openrisc
```

(The first line loads a default configuration for the atlys board, which can be overwritten by running the second line. The third line actually builds the kernel.) Something that can be interesting to change is the modeline sent to the VGA module on boot. This can be changed in the menuconfig by entering the *Processor type and features* menu, and changing the line at the very bottom of the list.

This will build vmlinux.bin, which will be loaded to the FPGA. To change what basic programs are loaded with the linux dist, add your own programs to arch/openrisc/support/initramfs. These must be compiled with the or32-linux tools.

4.5 Running the Linux kernel in or1ksim

Same procedure as for ordinary elf. Linux will open a telnet interface you can connect to **AT PORT NR?**.

4.6 Running the Linux kernel on the FPGA

The OpenRISC processor need the linux image to be in the correct format, so convert the vmlinux.bin with the bin2binsizeword found in orpsoc/sw/utils:

```
cd sw/utils
make
./bin2binsizeword /path/to/image/vmlinux.bin /path/to/new/im-
age/vmlinux.bin
```

4.6.1 Bad way

Put the word size bin in the boards/xilinx/atlys/backend/par/run folder. Then load it as a "bootloader" into the mcs file by typing:

```
make orpsoc.mcs BOOTLOADER_BIN=vmlinux.bin
```

This will program the linux image into the mcs file, which should be about 15MB in size by now. Load this file to the FPGA SPI-flash using iMPACT or Digilent Adept.

4.6.2 Good way

Load a bootloader into the BOOTLOADER_BIN section and load the linux image from flash or from an external memory. We use u-boot for this. Download it by running:

```
git clone git://openrisc.net/stefan/u-boot
```

Then run:

```
make atlys_config
make
```

To build u-boot.bin for the atlys board (several other orpsoc board configurations exist). Convert it to word size with:

```
bin2binsizeword u-boot.bin u-boot-wordsize.bin
```

Load it to flash using the instructions above (BOOTLOADER_BIN flag).

Use the mkimage program in the tools directory to build a u-boot compatible image to boot:

1. Linux image:

   ```
   tools/mkimage -n 'Linux for OpenRISC' -A or1k -O linux -T
   kernel -C none -a 0 -e 0x100 -d vmlinux.bin uImage
   ```

2. Bare metal image:

   ```
   tools/mkimage -A or1k -T standalone -C none -a 0 -e 0x100 -n
   helloWorld -d hello.bin uImage
   ```

Connect to the board through a serial terminal (connect to the UART microUSB port, connect other end to computer USB port). The device will probably show up as ttyACM0, but check dmesg to verify. Start a serial terminal such as gtkterm and connect to the board using a baudrate of 115200, 8 data bits, 1 stop bit, no parity. Type help and you should get a list of available commands.

You must have a network connection to a computer with nfs installed (sudo apt-get install nfs-kernel-server). Export a directory (add a line to /etc/exports) containing the uImage you want.

On the board (through the u-boot uart interface) set the ipaddr and serverip. These can be saved to flash with **saveenv**. Set the *bootfile* to /the/path/to/your/export/uImage. Type **nfs**. The image should load (check network connection if fails) in around 10-20 seconds. Type **bootm**. Image should boot.

To make it easier to boot, write:

```
setenv bootcmd nfs 0x100000 192.168.1.10:/export/share/uImage
bootm 100000
saveenv
```

The only command needed to be run through the uart terminal is boot. *Currently this doesn't work so good for some reason. Should though...*

5 Extra

5.1 UART

You can get UART output from the MicroUSB port located beside the USB port, In Ubuntu it shows as /dev/ttyACM0. use a baudrate of 115200 with 8 databit, 1 stop bit, no parity. GtkTerm is a good tool for reading from the uart.

```
sudo apt-get install gtkterm
```

5.2 Adding a RTL Module

A good way to start making your own RTL modules is to first check how the other RTL modules are structured. The GPIO module is simple and a good module to start with. Make a copy of the GPIO folder (orpsocv2/boards/xilinx/atlys/rtl/verilog) with your wanted module name. Now modify the content of the folders to your needs and make sure to follow all the naming conventions. The folder, the verilog file and the module must have the same name.

Files that need to be updated:

- atlys/rtl/verilog/orpsoc_top/orpsoc_top.h
- atlys/rtl/verilog/include/orpsoc-defines.v
- atlys/rtl/verilog/orpsoc_testbench.v
- atlys/backend/par/bin/atlys.ucf
- If any wishbone buses are used, many things need to be updated:
  - atlys/rtl/verilog/arbiter/(the bus you are adding to).v
  - atlys/rtl/verilog/orpsoc_top/orpsoc_top.h
  - atlys/rtl/verilog/include/orpsoc_params.h

atlys.ucf only needs to be updated if any IO pins where modified.

5.3 Bare metal light LEDs in assembly

This is an example to test to set the gpio register (through the wishbone bus) on the board. The LED register on GPIO module lights the LEDs according to the input. You need to "cd" to the following directory orpsocv2/boards/xilinx/atlys/sw/bootrom/

```
##### name pio.S
l.movhi r3,0x9100 # Set the memory address for the LEDs (GPIO module)
l.addi r4,r0,0xff # Set r4 to 0xff (set all GPIO to high)

l.sb 0x0(r3),r4 # sb = set byte, sets register at address r3 with value r4
l.sb 0x1(r3),r4 # sets the GPIO register to output

l.j 0
l.nop
l.nop
```

To compile this, run the following:

```
or32-elf-as -o pio pio.S
or32-elf-objcopy -O binary pio
```

Now, lets do some magic. (the magic of pipes updates the bootrom.v)

```
bin2vlogarray < pio
bin2vlogarray < pio > bootrom.v
```

Now you need to rebuild your entire system then flash the FPGA with the new image. When you have loaded the .mcs file to your flash and restarted your board, the LEDs should be lit!

All available assembly instructions and their implementation are explained in the openrisc_arch.pdf document.

5.4 DDR2 RAM

The Atlys board contains a DDR2 RAM that allows for simultaneous reads and writes. The DDR2 RAM contains 6 ports, 3 Read/Write ports 2 Read only and 1 Write only. The Write only port can be combined with a Read only to form an extra Read/Write port.

5.5 Memory Mapping

OpenRISC Reference Platform (ORP) Address Space

Start adr	**End adr**	**cached**	**Size(Mb)**	**Content**
0xf000_0000	0xffff_ffff	Cached	256	ROM
0xc000_0000	0xefff_ffff	Cached	768	Reserved
0xb800_0000	0xbfff_ffff	Uncached	128	Reserved for custom devices
0xa600_0000	0xb7ff_ffff	Uncached	288	Reserved
0xa500_0000	0xa5ff_ffff	Uncached	16	Debug 0-15
0xa400_0000	0xa4ff_ffff	Uncached	16	Digital Camera Controller 0-15
0xa300_0000	0xa3ff_ffff	Uncached	16	I2C Controller 0-15
0xa200_0000	0xa2ff_ffff	Uncached	16	TDM Controller 0-15
0xa100_0000	0xa1ff_ffff	Uncached	16	HDLC Controller 0-15
0xa000_0000	0xa0ff_ffff	Uncached	16	Real-Time Clock 0-15
0x9f00_0000	0x9fff_ffff	Uncached	16	Firewire Controller 0-15
0x9e00_0000	0x9eff_ffff	Uncached	16	IDE Controller 0-15
0x9d00_0000	0x9dff_ffff	Uncached	16	Audio Controller 0-15
0x9c00_0000	0x9cff_ffff	Uncached	16	USB Host Controller 0-15
0x9b00_0000	0x9bff_ffff	Uncached	16	USB Func Controller 0-15
0x9a00_0000	0x9aff_ffff	Uncached	16	General-Purpose DMA 0-15
0x9900_0000	0x99ff_ffff	Uncached	16	PCI Controller 0-15
0x9800_0000	0x98ff_ffff	Uncached	16	IrDA Controller 0-15
0x9700_0000	0x97ff_ffff	Uncached	16	Graphics Controller 0-15
0x9600_0000	0x96ff_ffff	Uncached	16	PWM/Timer/Counter Controller 0-15
0x9500_0000	0x95ff_ffff	Uncached	16	Traffic COP 0-15
0x9400_0000	0x94ff_ffff	Uncached	16	PS/2 Controller 0-15
0x9300_0000	0x93ff_ffff	Uncached	16	Memory Controller 0-15
0x9200_0000	0x92ff_ffff	Uncached	16	Ethernet Controller 0-15
0x9100_0000	0x91ff_ffff	Uncached	16	General-Purpose I/O 0-15
0x9000_0000	0x90ff_ffff	Uncached	16	UART16550 Controller 0-15
0x8000_0000	0x8fff_ffff	Uncached	256	PCI I/O
0x4000_0000	0x7fff_ffff	Uncached	1024	Reserved
0x0000_0000	0x3fff_ffff	Cached	1024	RAM

6 References

OpenRISC: http://opencores.org/or1k/Main_Page
ORPSoC: http://opencores.org/openrisc,orpsocv2
or1ksim: http://opencores.org/openrisc,or1ksim
Toolchan: http://opencores.org/openrisc,gnu_toolchain