# Building a Minimal OpenRISC System

Matthew Hicks
*University of Illinois at Urbana-Champaign*

To use the OR1200 OpenRISC processor we need to combine it with several other peripherals, using their wishbone interfaces, creating a system-on-chip (SoC). A basic SoC contains the OR1200 processor with JTAG-based debug interface, an internal memory, and a UART. This allows for small programs to be loaded into memory using the Xilinx USB programming cable which uses one of the FPGA's user JTAG access ports. The processor runs the program from memory, outputting console text to the UART port. This system is sufficient enough to learn how to build a system, compile software for that system, attach a debugger, and run a "Hello World" program.

## 1 Prerequisites

- Xilinx XUP-V5 development board (a.k.a. ML509)
- Linux
- Xilinx ISE 12 with bin directory added to the PATH
- RS-232 to USB adapter
- cutecom

## 2 Building the hardware

1. Download the svn repositories for each core in the SoC into a common folder
   (a) svn co `http://opencores.org/ocsvn/adv_debug_sys/adv_debug_sys/trunk` adv_debug_sys
   (b) svn co `http://opencores.org/ocsvn/openrisc/openrisc/trunk/or1200` or1200
   (c) svn co `http://opencores.org/ocsvn/uart16550/uart16550/trunk` uart16550
   (d) svn co `http://opencores.org/ocsvn/minsoc/minsoc/trunk` minsoc
2. Configure the cores
   (a) The Verilog options file for the Xilinx JTAG user interface of adv_debug_sys needs to be updated with the correct model FPGA. The file is located in …adv_debug_sys/Hardware/xilinx_internal_jtag/rtl/verilog. Uncomment the VIRTEX5 line and ensure all other lines are commented-out.
   (b) The Verilog defines file for the debug interface (adv_debug_sys) needs to be updated with the correct parameters. The file is located in …adv_debug_sys/Hardware/adv_dbg_if/rtl/verilog. Comment-out the definition of DBG_JSP_SUPPORTED and ADBG_JSP_SUPPORT_MULTI.
   (c) The Verilog defines file for the top-level design needs to be configured so that the minimal SoC is implemented correctly. The file is located in …minsoc/rtl/verilog. Uncomment and comment lines as required to build a SoC with only debug, UART, and internal memory peripherals, for a Xilinx Virtex 5 FPGA, running at 100 MHZ (input clock is 100 MHz), with negative reset. It is key that ETHERNET and STARTUP are left undefined.
3. Prepare the Makefile

   (a) Copy the Makefile to the directory where all the svn repos of the cores were checked-out into
   (b) Copy the Makefile support files (.prj and .xst) folder buildSupport to the directory where all svn repos were checked-out into
   (c) Update the ROOT directory in Makefile to the path of the core repo containing directory

4. Verify the existence or create a constraints (.ucf) file for your development board in . . . minsoc/backend

5. Copy the blackBoxes folder to the folder where the cores are checked-out into
   (a) In order to reduce synthesis times and keep the system more modular, this project synthesizes each core separately, then connects the resulting netlists together, at the top level, using a WISHBONE compliant bus. To do this, the top level treats each core as a black box during synthesis. This requires an empty, interface only version of each core's top-level design file. The blackBoxes folder contains an interface version of the UART, OR1200, and debug cores.
   (b) Adding other peripherals to the SoC requires creating an interface version of that peripheral's top level file and then adding it to the blackBoxes folder. To do this, remove all logic from the peripheral's top-level HDL file, keeping only the input/output description.

6. make all - synthesizes, translates, maps, places and routes, then generates a bitstream for the SoC
   (a) make help - provides more information about other make targets

7. Problems
   (a) Missing Verilog files due to core designer adding or removing files from the design
   (b) Change in Xilinx command line tools accepted switches

## 3 Programming the FPGA

Make sure that the FPGA is configured to load configuration data from JTAG and use Xilinx's Impact tool to load the bit file (minsoc.bit) onto the FPGA.

## 4 Installing or1ksim, the OpenRISC Simulator

or1ksim is an instruction-level simulator for the OpenRISC platform. It is capable of booting Linux and running bare metal programs. or1ksim is required for building the OpenRISC GCC toolchain from source.

1. svn co `http://opencores.org/ocsvn/openrisc/openrisc/trunk/or1ksim` or1ksim
2. mkdir or1ksimbuild
3. cd or1ksimbuild
4. `../or1ksim/configure --target=or32-elf --prefix=/opt/or1ksim`
5. make all
6. sudo make install
7. cd ..
8. rm -rf or1ksimbuild
9. Add /opt/or1ksim/bin to your PATH variable

## 5 Install the OpenRISC GCC Toolchain

1. Checkout the toolchain source - svn co `http://opencores.org/ocsvn/openrisc/openrisc/trunk/gnu-src` orGNU
2. Verify that the entire repository was transfered using "svn update"
3. Libraries required for building the toolchain
   (a) build_essential
   (b) make
   (c) gcc
   (d) g++
   (e) flex
   (f) bison
   (g) patch
   (h) texinfo
   (i) libncurses-dev
   (j) libmpfr-dev
   (k) libgmp3-dev
   (l) libmpc-dev
   (m) libzip-dev
4. Download C library and Linux kernel sources
   (a) Verify git is installed
   (b) cd orGNU
   (c) git clone `git://git.openrisc.net/jonas/uClibc`
   (d) git clone `git://git.openrisc.net/jonas/linux`
5. In the orGNU folder, build both bare metal (newlib, or32-elf-) and Linux (uLibC, or32-linux-) OpenRISC toolchains by executing the command

```
sudo ./bld-all.sh --force --prefix /opt/openrisc --uclibc-dir uClibc

--linux-dir linux --or1ksim-dir /opt/or1ksim
```

6. Update your PATH variable to include /opt/openrisc/bin
7. rm -rf orGNU
8. If you have problems visit `http://opencores.org/openrisc,gnu_toolchain`

## 6 Write and Compile a "Hello World" Program

1. cd . . . minsoc/sw/utils
2. make clean
3. make all
4. cd . . . minsoc/sw/support

5. In both except.S and reset.S, remove all underscore prefixes from all referenced functions
6. Update board.h with the correct frequency “define IN_CLK 100000000”
7. make clean
8. make all
9. cd . . . minsoc/sw/drivers
10. make clean
11. make all
12. cd . . . minsoc/sw/uart
13. make clean
14. make all

## 7 Compile the Debug Interface

1. cd . . . adv_debug_sys/software/adv_jtag_bridge
2. Configure the Makefile
   (a) Change “prefix” to the path where the binary will be stored, in a bin folder
   (b) Select the correct option for “BUILD_ENVIRONMENT”
   (c) Change “INCLUDE_JSP_SERVER=true” to “INCLUDE_JSP_SERVER=false”
3. make
4. sudo make install
5. Add “prefix”/bin to you PATH variable

## 8 Gather the JTAG Device Files

1. Copy the bsdlFiles folder to the directory where all the cores were checked-out into
   (a) To find the bsdl files for all the devices on the JTAG chain of the FPGA, first create a list of devices using Xilinx’s Impact tool
   (b) Locate the bsd file for each device in the Xilinx ISE installation folders and copy it to the bsdlFiles folder

## 9 Debugging

1. Start the JTAG bridge software
   (a) cd . . . bsdlFiles
   (b) adv_jtag_bridge xpc_usb
2. Start cutecom
   (a) Change to the cutecom folder
   (b) sudo ./cutecom
   (c) Set the baud rate to 115200

    (d) Use device /dev/ttyUSB0
    (e) Connect to the board

3. Load and run the uart program using gdb
    (a) cd . . . minsoc/sw/uart
    (b) or32-elf-gdb uart-nocache.or32
    (c) set remotetimeout 10
    (d) target remote:9999
    (e) load
    (f) c

## 10 Hello World

If everthing is setup correctly, you should see “Hello World” printed in the cutecom window. Additionally, whatever text you send to the board using cutecom will be returned, but using the next ASCII character.