OpTiMSoC User Guide

December 29, 2015

Document Changes

2015.1

- Fresh restart with package-based installation (S. Wallentowitz)
- Update old tutorials (S. Wallentowitz)
- Extend tutorial for FPGA (P. Wagner and S. Wallentowitz)

June 20, 2013

- Add installation and configuration description (S. Wallentowitz)
- Update old tutorials (S. Wallentowitz)
- Add host software (GUI) in tutorials (S. Wallentowitz)
- Add development tutorials (P. Wagner)

January 28, 2013

- Initial version of the document
- First tutorial steps for distributed memory systems

OpTiMSoC Documentation Overview

The OpTiMSOC documentation is organized in four different categories:

User Guide The User Guide describes the general usage of the OpTiMSoC elements in a tutorial style. It covers the basic building processes and how to get stuff running. The User Guide is related to releases and distributed via the website and can be built in the repository.

API documentation The software components are documented using Doxygen[1]. The generated API documentations serve the users when programming software for OpTiMSoC. The API documentation is also related to releases and can be automatically generated in the repository and are also distributed via the website. At the moment the following API documentation is available:

- OpTiMSoC Baremetal Libraries API
- OpTiMSoC gzll Libraries API
- OpTiMSoC Host Software API
- OpTiMSoC SystemC Library

[1]`http://www.doxygen.org`

Reference Manual The Reference Manual covers all topics in detail. It gives a better insight in how OpTiMSoC is organized and how things work. It primarily serves developers as source of information when extending OpTiMSoC.

Technical Reports The Technical Reports are released separately and cover implementation details. They therefore serve as documentation of components and source of information for developers. They do not only cover technical details but most importantly also present *why* something works as it does.

Contents

1 Introduction **1**

2 Installation & Configuration **3**

2.1 Prerequisites 3
2.2 Get OpTiMSoC – Quick Start 3
2.3 Get OpTiMSoC – Build yourself 4
2.3.1 Pre-requisites 5
2.3.2 Get the sources 5
2.3.3 Run installer 5

3 Tutorials **6**

3.1 Starting Small: Compute Tile and Software (Simulated) 6
3.2 Going Multicore: Simulate Multicore Compute Tiles 8
3.3 Tiled Manycore SoC: Simulate a Small 2x2 Distributed Memory System 8
3.4 The Look Inside: Introducing the Debug System 9
3.5 Verilator: Compiled Verilog simulation 10
3.6 Going to the FPGA: ZTEX Boards 11
3.6.1 Prepare: Simulate the Complete System 11
3.6.2 Synthesis 12
3.6.3 Testing on the FPGA 13

4 Develop OpTiMSoC **15**

4.1 Choosing an Editor/IDE 15
4.1.1 Eclipse 15
4.1.2 Emacs 17
4.1.3 Verilog-mode 17
4.1.4 Qt Creator for GUI Development 19

1 Introduction

Open Tiled Manycore System-on-Chip (OpTiMSoC) is library-based framework that allows you to build you own Manycore. So called "Tiled Manycores" are based on a regular Network-on-Chip to which different tiles are connected. Such tiles can contain processing elements, memory, I/O devices etc.

OpTiMSoC is based on LISNoC, an open source Network-on-Chip and other open source hardware components. In future, this set of components will be continuously extended and you can easily integrate your own components. A variety of target platforms, such as FPGA boards, emulation platforms and simulations will be supported.

For a general introduction about the project goals and elements of OpTiMSoC please refer to our permanently updated preprint on ArXiv.org[1].

This document documents the projects from a user point of view. Starting with the description of how to get and install OpTiMSoC it describes the different kinds of simulation or syntheses systems currently supported by OpTiMSoC in the style of step-by-step tutorials.

In the following, we will give you a short overview of the different components that are part of OpTiMSoC or other dependencies.

OpTiMSoC Toolchain

The toolchain of OpTiMSoC currently consists of the standard OpenRISC crosscompiler and a newlib libc port (based on the OpenRISC newlib) plus a few small utility programs and scripts.

You will need to have it installed in most cases as it is necessary to build all the software running on the OpTiMSoC systems.

SoC Libraries

The system libraries provide all functionalities of the OpTiMSoC platforms to the applications. This includes hardware drivers, runtime support, communication APIs and task management APIs.

RTL Simulation

In case you are developing hardware in OpTiMSoC, RTL simulation is the starting point of development. Beside this, RTL simulation is used to execute small pieces of software for testing or development of drivers and the runtime system.

Unfortunately, we do not know of any high-quality open source RTL simulation tool, so that we here rely on commercial EDA tools. At the moment this is Mentor's Modelsim.

[1]`http://arxiv.org/abs/1304.5081`

Verilator

Verilator is an open source tool that compiles Verilog code to SystemC and allows users without a commercial RTL simulator license to run system simulations.

Synthesis

Currently we focus on FPGA synthesis for Xilinx FPGA. There will always be a supported board that can be used with the free Xilinx WebPack, but in general you will of course need a Xilinx license for this.

Beside Xilinx tools, we also support Synopsys Synplify as this a more sophisticated synthesis tool.

Host Software

Host software is needed for control and diagnosis/debug of the running systems.

Host GUI

A graphical user interface provides easy control and observability of the running systems.

2 Installation & Configuration

Before you get started with OpTiMSoC you should notice that external tools and libraries might be required that are in some cases proprietary and cost some money. Although OpTiMSoC is developed at an university with access to many EDA tools, we aim to always provide tool flows and support for open and free tools, but especially when it comes to synthesis such alternatives are even not available.

2.1 Prerequisites

Throughout this document some packages are required in your Linux distribution. OpTiMSoC should principally work on all common Linux distributions. In case you encounter problems in your system we highly encourage you to contact the OpTiMSoC maintainers to fix these problems. Nevertheless, we recommend Ubuntu 12.04 or 14.04 LTS as development system and can ensure OpTiMSoC will work on it as we also work on it. In the following we will refer to Ubuntu/Debian commands to install packages, that will run under Ubuntu 12.04 and 14.04 LTS.

Independent of the OpTiMSoC components you plan to use, you will need some packages to be installed:

```
sudo apt-get -y install git python build-essential automake autoconf
```

2.2 Get OpTiMSoC – Quick Start

The most simple way to get started is with the release packages. You can find the OpTiMSoC releases here: `https://github.com/optimsoc/sources/releases`. With the release you can find the distribution packets that need to be extracted to `/opt/optimsoc`. There are two packages: the `base` package contains the programs, libraries and tools to get started. The `examples` package contains prebuilt verilator simulations for the real quick start.

For example take the 2015.1 release and download both `base` and `examples`. Then create the base folder and extract the package:

```
sudo mkdir /opt/optimsoc
sudo chown $USER /opt/optimsoc
tar -xzf optimsoc-2015.1-base.tgz -C /opt/optimsoc
tar -xzf optimsoc-2015.1-examples.tgz -C /opt/optimsoc
```

You can now use the installation after setting the environment with the pre-installed environment source script:

```
source /opt/optimsoc/2015.1/setup.sh
```

We encourage you to put this line into your `~/.bashrc`.

Now you are near to get started, but you need some programs to build software to run in OpTiMSoC and execute the verilator-based simulations. Those are: the `or1k-elf-multicore` toolchain, `SystemC` and `Verilator`. All those tools are free, but are (except for an outdated `Verilator` version) not part of the Linux package systems. Hence you need to built those tools as described in the Reference Manual, or you can simply download some prebuilt-versions.

To do so simply run a script for your version:

```
wget https://raw.githubusercontent.com/optimsoc/prebuilts/master/optimsoc
    ↪ -prebuilt-deploy.py
chmod a+x optimsoc-prebuilt-deploy.py
./optimsoc-prebuilt-deploy -d /opt/optimsoc systemc verilator or1kelf
```

You may of course leave out any of the tools if you already have it installed. Finally, you will get another source script to set up the environment for the prebuilt tools:

```
source /opt/optimsoc/setup_prebuilt.sh
```

You are now ready to go to the tutorials.

2.3 Get OpTiMSoC – Build yourself

You can also build OpTiMSoC from the sources. This options usually becomes standard if you start developing for or around OpTiMSoC. The build is done from one git repository: `https://github.com/optimsoc/sources`.

It is generally a good idea to understand git, especially when you plan to contribute to OpTiMSoC. Nevertheless, we will give a more detailed explanation of how to get your personal copies of OpTiMSoC and (potentially) update them.

The start of a successful built is to install the tools `Verilator`, `SystemC`, and the `or1k-elf-multicore` toolchain. The most simple way is to start with the prebuilt tools as described above, then set the environment for the tools.

Note

During the installation, you'll frequently encounter three types of directories.

- **The source directory.** This is the place where the uncompiled source code files are stored. Usually, that is the folder that you cloned from the git repository.

  The `$OPTIMSOC_SOURCE` environment variable should point to the root of the source directories.

- **The build or object directory.** For different components such a directory is used to build the component. The installer performs the build process in these directories and you perform individual builds there if you develop for OpTiMSoC (for example: if you develop the libraries).

Sometimes, this directory is equal to the source directory, but most of the time, we create a new directory called `build` inside the source directory. Doing so has a great benefit: if something in the build process went wrong, you can simply delete the build directory and start all over again.

- **The installation directory.** This is the target directory where results of the build process are stored for further use. It is used by the installer or you when running `make install` to store the files generated by the build process. The environment variable `$OPTIMSOC` points to the root of the installation directory.

2.3.1 Pre-requisites

You will need some programs to build OpTiMSoC, e.g., on Ubuntu 14.04:

```
sudo apt-get install autoconf automake libtool tcl texlivetexlive-latex-
    ↪ extra texlive-fonts-extra
```

2.3.2 Get the sources

Start with checking out the repository:

```
git clone https://github.com/optimsoc/sources.git optimsoc-sources
cd optimsoc-sources
```

2.3.3 Run installer

Now simply run the installer to install:

```
./tools/install.py -d /opt/optimsoc/2015.1
```

3 Tutorials

The best way to get started with OpTiMSoC after you've prepared your system as described in the previous chapter is to follow some of our tutorials. They are written with two goals in mind: to introduce some of the basic concepts and nomenclature of manycore SoC, and to show you how those are implemented and can be used in OpTiMSoC.

Some of the tutorials (especially the first ones) build on top of each other, so it's recommended to do them in order. Simply stop if you think you know enough to implement your own ideas!

3.1 Starting Small: Compute Tile and Software (Simulated)

It is a good starting point to simulate a single compute tile of a distributed memory system. Therefore a simple example is included and demonstrates the general simulation approach and gives an insight in the software building process.

Simulating only a single compute tile is essentially an OpenRISC core plus memory and the network adapter, where all I/O of the network adapter is not functional in this test case. It can therefore only be used to simulate local software.

You can find this example in `$OPTIMSC/examples/dm/compute_tile`.

In default mode they start a server to connect the host software to, but you can use the parameter `--standalone` to run them in standalone. If you start the simulation now

```
$OPTIMSOC/examples/dm/compute_tile/tb_compute_tile --standalone
```

you will get this output

```
%Error: ct.vmem:0: $readmem file not found
Aborting...
Aborted (core dumped)
```

The simulations always expect vmem files that initialize the memories. This needs to be generated from the compiled source code.

Our demonstration software is available in an extra repository:

```
git clone https://github.com/optimsoc/baremetal-apps
cd baremetal-apps
```

Build a simple "Hello World" example:

```
cd hello
make
```

You will then find the executable elf file as `hello/hello.elf`. Furthermore some other files are build:

- `hello.dis` is the disassembly of the file
- `hello.bin` is the elf representation of the binary file
- `hello.vmem` is a textual copy of the binary file

You can now run the example again, this time with a different memory initialization file:

```
$OPTIMSOC/examples/dm/compute_tile/tb_compute_tile --standalone --meminit
   ↪ =hello.vmem
```

the simulation should terminate with:

```
[157801] Core 0 has terminated
[157801] All cores terminated. Exiting..
```

Furthermore, you will find a file called `stdout.000` which shows the actual output:

```
Hello World! Core 0 of 1 in tile 0, my absolute core id is: 0
There are 1 compute tiles:
 rank 0 is tile 0
```

But how does the actual printf-output get there when there is no UART or similar?

OpTiMSoC software makes excessive use of a useful part of the OpenRISC ISA. The "no operation" instruction `l.nop` has a parameter `K` in assembly. This can be used for simulation purposes. It can be used for instrumentation, tracing or special purposes as writing characters with minimal intrusion or simulation termination.

The termination is forced with `l.nop 0x1`. The instruction is observed and a trace monitor terminates when it was observed at all cores (shortly after `main` returned).

With this method you can simply provide constants to your simulation environments. For variables this method is extended by putting data in further registers (often `r3`). This still is minimally intrusive and allows you to trace values. The printf is also done that way (see newlib):

```
void sim_putc(unsigned char c) {
  asm("l.addi\tr3,%0,0": :"r" (c));
  asm("l.nop %0": :"K" (NOP_PUTC));
}
```

This function is called from printf as write function. The trace monitor captures theses characters and puts them to the stdout file.

You can easily add your own "traces" using a macro defined in `baremetal-libs/src/libbaremetal/include/optimsoc-baremetal.h`:

```
#define OPTIMSOC_TRACE(id,v)                    \
   asm("l.addi\tr3,%0,0": :"r" (v) : "r3"); \
   asm("l.nop %0": :"K" (id));
```

3.2 Going Multicore: Simulate Multicore Compute Tiles

Next you might want to build an actual multicore system. In a first step, you can just start simulations of compute tiles with multiple cores:

```
$OPTIMSOC/examples/dm/compute_tile-dual/tb_compute_tile --standalone --
    ↪ meminit=hello.vmem
$OPTIMSOC/examples/dm/compute_tile-quad/tb_compute_tile --standalone --
    ↪ meminit=hello.vmem
$OPTIMSOC/examples/dm/compute_tile-octa/tb_compute_tile --standalone --
    ↪ meminit=hello.vmem
```

You can observe, the simulation runs become longer. After each run, inspect the `stdout.*` files.

3.3 Tiled Manycore SoC: Simulate a Small 2x2 Distributed Memory System

Next we want to run an actual NoC-based tiled manycore system-on-chip, with the examples you get `system_2x2_cccc`. The nomenclature in all pre-packed systems first denotes the dimensions and then the instantiated tiles, here `cccc` as four compute tiles.

Execute it again to get the hello world experience:

```
$OPTIMSOC/examples/dm/system_2x2_cccc/tb_system_2x2_cccc --standalone --
    ↪ meminit=hello.vmem
```

In our simulation all cores in the four tiles run the same software. Before you shout "that's boring:" You can still write different code depending on which tile and core the software is executed. A couple of functions are useful for that:

- `optimsoc_get_numct()`: The number of compute tiles in the system
- `optimsoc_get_numtiles()`: The number of tiles (of any type) in the system
- `optimsoc_get_ctrank()`: Get the rank of this compute tile in this system. Essentially this is just a number that uniquely identifies a compute tile.

There are more useful utility functions like those available, find them in the file `baremetal-libs/src/libbaremetal/include/optimsoc-baremetal.h`.

A simple application that uses those functions to do message passing between the different tiles is `hello\_mpsimple`. This program uses the simple message passing facilities of the network adapter to send messages. All cores send a message to core 0. If all messages have been received, core 0 prints a message "Received all messages. Hello World!".

```
cd ../hello_mpsimple
make
$OPTIMSOC/examples/dm/system_2x2_cccc/tb_system_2x2_cccc --standalone --
    ↪ meminit=hello_mpsimple.vmem
```

Have a look what the software does (you find the code in `$OPTIMSOC_APPS/baremetal/hello_mpsimple/hello_mpsimple.c`). Let's first check the output of core 0.

```
$> cat stdout.000
Wait for 3 messages
Received all messages. Hello World!
```

Finally, let's have a quick glance at a more realistic application: `heat_mpsimple`. You can find it in the same place as the previous applications, `hello` and `hello_mpsimple`. The application calculates the heat distribution in a distributed manner. The cores coordinate their boundary regions by sending messages around.

Can you compile this application and run it? Don't get nervous, the simulation can take a couple of minutes to finish. Have a look at the source code and try to understand what's going on. Also have a look at the `stdout` log files. Core 0 will also print the complete heat distribution at the end.

3.4 The Look Inside: Introducing the Debug System

Note

This and the following sections have not been tested for this release, and may most probably not run as described. But as a reference, they can serve you to better understand OpTiMSoC. They will be rewritten for the upcoming release. Thanks for your understanding!

In the previous tutorials you have seen some software running on a simple OpTiMSoC system. Until now, you have only seen the output of the applications, not how it works on the inside.

This problem is one of the major problems in embedded systems: you cannot easily look inside (especially as soon as you run on real hardware as opposed to simulation). In more technical terms, the system's observability is limited. A common way to overcome this is to add a debug and diagnosis infrastructure to the SoC which transfers data from the system to the outside world, usually to a PC of a developer.

OpTiMSoC also comes with an extensive debug system. In this section, we'll have a look at this system, how it works and how you can use it to debug your applications. But before diving into the details, we'll have a short discussion of the basics which are necesssary to understand the system.

Many developers know debugging from their daily work. Most of the time it involves running a program inside a debugger like GDB or Microsoft Visual Studio, setting a breakpoint at the right line of code, and stepping through the program from there on, running one instruction (or one line of code) at a time.

This technique is what we call run-control debugging. While it works great for single-threaded programs, it cannot easily be applied to debugging parallel software running on possibly heterogenous many-core SoC. Instead, our solution is solely based on tracing, i.e. collecting information from the system while it is running and then being able to look at this data later to figure out the root cause of a problem.

The debug system consists of two main parts: the hardware part runs on the OpTiMSoC system itself and collects all data. The other part runs on a developer's PC (often also called host PC) and controls the debugging process and displays the collected data. Both parts are connected using either a USB connection (e.g. when running on the ZTEX boards), or a TCP connection (when running OpTiMSoC in simulations).

3.5 Verilator: Compiled Verilog simulation

At the moment running "verilated" simulations is the best supported way of observing the system traces. We will therefore run the examples from before using a verilated simulation and observing the system in the graphical user interface.

In the following we will have a look at building such a system and how to observe it with the GUI. In `tbench/verilator/dm` you find systems identical to the RTL simulation. We will directly start with the `system_2x2_cccc`. In the base folder you should simply make it:

```
$> make
```

The command will first generate the verilated version of the 2x2 system. Finally it builds the toplevel files and links to `tb_system_2x2_cccc` and `tb_system_2x2_cccc-vcd`. The latter generates a full VCD trace file of the hardware, which is much slower and also easily takes up tens of GB.

Similar to the steps described above you will need to build the software, e.g., the heat example. Again you need to link the `vmem` file. Now start the simulation:

```
$> ./tb_system_2x2_cccc
```

It will start a debug server and wait for connections:

```

             SystemC 2.3.0-ASI --- Feb 11 2013 12:54:17
        Copyright (c) 1996-2012 by all Contributors,
        ALL RIGHTS RESERVED

Listening on port 22000
```

In another console now start the OpTiMSoC GUI:

```
$> optimsocgui
```

In the first dialog window you need to set the debug backend to *Simulation TCP Interface* and proceed then. After the GUI started you need to connect using *Target System→Connect*. The system view should change to a 2x2 system.

The last step is to run the system by *Target System→Start CPUs*. The execution trace on the bottom of the window will start showing execution sections and events. By moving the mouse over the section you will find the description of the section. Similarly for the events you find a short description of the event.

3.6 Going to the FPGA: ZTEX Boards

The recommended platform for software development or any other system which needs no I/O is the ZTEX boards[1]. Various variants exist, the supported boards are the 1.15 series version b and d, where the latter is twice as large as the former and can therefore contain more processor cores. The 2x2 example works with both boards.

3.6.1 Prepare: Simulate the Complete System

Before we go to the actual board we want to simulate the entire system on the FPGA to see if the debug system works correctly and the clocks works correct.

The distribution therefore contains a SystemC module that functionally behaves like the USB chip on the ZTEX boards. The host tools can connect to the debug system via this module using a TCP socket.

The system can be found at `tbench/rtl/dm/system_2x2_cccc_ztex`. Build the system running `make`. Before you simulate the system you will now need to provide a `modelsim.ini` either globally or in the system's folder that contains the Xilinx libraries. Once you have it, you can start the system using

```
$> make sim-noninteractive
```

The simulation will start and you can now connect to the system in a different shell by using the command line interface:

```
$> optimsoc_cli -bdbgnoc -oconn=tcp,host=localhost,port=23000
```

The command line interface will connect to the system and enumerate all debug modules:

```
Connected to system.
System ID: 0x0000ce75
Module summary:
addr.   type    version name
0x02    0x02    0x00    ITM
0x03    0x02    0x00    ITM
0x04    0x02    0x00    ITM
0x05    0x02    0x00    ITM
0x06    0x05    0x00    STM
0x07    0x05    0x00    STM
0x08    0x05    0x00    STM
0x09    0x05    0x00    STM
0x0a    0x07    0x00    MAM
0x0b    0x07    0x00    MAM
0x0c    0x07    0x00    MAM
```

[1]See `http://www.ztex.de`

```
0x0d    0x07    0x00    MAM
```

The modules are the *Instruction Trace Module (ITM)*, *Software Trace Module (STM)* and *Memory Access Module (MAM)* for each of the four cores.

Before debugging now, you will need to build the software as described before in the `sw` subfolder. Once you have build `hello_simplemp` you can execute it in the simulation.

First you enter interactive mode:

```
$> optimsoc_cli -bdbgnoc -oconn=tcp,host=localhost,port=23000
```

After enumeration you will get an `OpTiMSoC>` shell. First you can initialize the memories:

```
OpTiMSoC> mem_init hello_simplemp.bin 0-3
```

Next you need to enable logging of the software trace events to a file:

```
OpTiMSoC> log_stm_trace strace
```

Then start the system:

```
OpTiMSoC> start
```

Let it run for a while (1 minute) and then leave the command line interface:

```
OpTiMSoC> quit
```

After that you will find the expected output of the trace events in `strace`.

3.6.2 Synthesis

Once you have checked the correct functionality of the system (or alter your extensions) you can go over to system synthesis for the FPGA. At the moment we support the Synopsys FPGA flow (Synplify).

You can find the system synthesis in the folder `syn/dm/system_2x2_cccc_ztex`. A Makefile is used to build the systems.

To generate the system first create the project file:

```
$> make synplify.prj
```

Now the Synplify project file has been generated and you're ready to start the synthesis.

If you want to have the output of the synthesis in a folder different from your source folder (the one where you just ran `make` in), you can set the environment variable `OPTIMSOC_SYN_OUTDIR` to any path you like, e.g. put

```
export OPTIMSOC_SYN_OUTDIR=$HOME/syn
```

in your profile script (e.g. your `~/.bashrc` file) and reload it.

Run the synthesis afterwards (for the ZTEX 1.15b or d board):

```
$> make synplify_115b_ddr
```

Once the synthesis is finished you can generate the bitstream:

```
$> make bitgen_115d_ddr
```

3.6.3 Testing on the FPGA

Now that you have generated a bitstream we're ready to upload it to the FPGA. Connect the ZTEX 1.15 board to your PC via USB.

If you run `lsusb` the board identifies itself as:

```
Bus 001 Device 004: ID 221a:0100
```

There is no manufacturer or further information displayed. The reason is, that OpTiMSoC otherwise may require to buy a set of USB identifiers. Instead, all ZTEX boards share the same identifier and the following command is used to find out details on the Firmware, Board and Capabilities:

```
$> FWLoader -c -ii
```

To use the ZTEX boards as a user, it is recommended to add the following udev rule

```
SUBSYSTEM=="usb", ATTR{idVendor}=="221a", ATTR{idProduct}=="0100", MODE
   ↪ ="0666"
```

for example in `/etc/udev/rules.d/60-usb.rules`.

If you are running OpTiMSoC on the board for the first time you need to update the firmware on the board. To do that, switch to the folder `src/sw/firmware/ztex_usbfpga_1_15_fx2_fw` in your OpTiMSoC source tree. Follow the instructions inside the provided `README` file to build and flash the board with the required firmware. All of this only needs to be done once for each board (until the firmware changes).

Now the board will identify itself using `FWLoader -c -ii`:

```
bus=001  device=4 (`004')  ID=221a:100
   Manufacturer="TUM LIS"  Product="OpTiMSoC - ZTEX USB 1.15"
      ↪ SerialNumber="04A32DBCFA"
   productID=10.13.0.0  fwVer=0  ifVer=1
   FPGA configured
   Capabilities:
      EEPROM read/write
      FPGA configuration
      Flash memory support
      High speed FPGA configuration
      MAC EEPROM read/write
```

Everything ready to go? Then upload the bitstream to the FPGA by running

```
$> make flash_115d_ddr
```

in the same folder where you have been running `make bitstream_...` etc. in the previous section. The output should be something like

```
FWLoader -v 0x221a 0x100 -f -uf /[somepath]/system_2x2_cccc_ztex.bit
FPGA configuration time: 194 ms
```

As the FPGA is now ready you can use the same method to connect to the FPGA and load software on it as you've done in the Section 3.6.1, just this time the connection paramters used in `optimsoc_cli` are a bit different.

Run

```
$> optimsoc_cli -i -bdbgnoc -oconn=usb
```

to connect to the FPGA board over USB. You should again be presented with a listing of all available debug modules. Now you can continue just as you did before by calling `mem_init` to load some software onto the FPGA, etc.

Congratulations, you've run OpTiMSoC on real hardware for the first time! You can now develop software and explore OpTiMSoC. A handy utility is the python interface to the command line interface. Instead of running the interactive mode you can run the script interface like:

```
$> optimsoc_cli -s <script.py> -bdbgnoc -oconn=usb
```

An example python script:

```
mem_init(2,"hello_simple.bin")
log_stm_trace("strace")
start()
```

You can also connect to the USB now using the GUI. Now you're ready to explore and customize OpTiMSoC for yourself. Have fun!

4 Develop OpTiMSoC

After you have worked through some, or even all, of the tutorials in the previous chapter, you're now ready to bring your own ideas to live using OpTiMSoC. This chapter gives you a quick introduction on how to setup your development environment, like editors and the revision control system, and how to contribute back to the OpTiMSoC project.

We assumed in this whole tutorial that you are working on Linux. While it is certainly possible to use Windows or OS X for development, we cannot provide help for those systems and you're on your own.

4.1 Choosing an Editor/IDE

When editing code, an editor or IDE usually comes handy. While there is clearly no "best" or even "recommended" editor or IDE, we will present two or our choices here, together with some settings that make working on OpTiMSoC a pleasant experience. Feel free to adapt these recommendations to your personal preferences!

4.1.1 Eclipse

Eclipse gives you a nice and integrated development across the different parts of the code base by using a couple of plugins. But be aware, Eclipse likes memory and is not exactly "lightweight", but if you have enough memory available (in the area of 500 MB for Eclipse) it can be a very powerful and productive choice.

Installation and Basic Setup

First of all, get Eclipse itself. Go to `http://www.eclipse.org/downloads/` and get the "Eclipse IDE for C/C++ Developers" package or install it from your distribution's package manager. All the following steps were tested with Eclipse Kepler (4.3).

Now start Eclipse and first go to *Help* $\rightarrow$ *Check for Updates*. Install all available updates.

For Verilog syntax highlighting we use a plugin called "VEditor". Go to *Help* $\rightarrow$ *Install New Software...* In the field *Work with* enter the URL of the installation site, `http://veditor.sourceforge.net/update`. Now press the return key and after a couple of seconds, the entry *VEditor Plugin* appears below. Select it and click on the *Next* button until the installation is finished. To complete the process you need to restart Eclipse.

Every project has different preferences regarding the styling of the code. Therefore every editor can be configured to some extend. First, we'll set the general settings for Eclipse, and then for Verilog and C.

Start by clicking on *Window* → *Preferences* inside Eclipse. There, choose *Editors* → *Text Editors*. You should set the following settings:

- Check *Insert spaces for tabs*
- Check *Show print margin*
- Set the *Print margin column* to 80
- Check *Show line numbers*

Just leave the other settings as they are, or change them to your liking.

For the Verilog settings, go to *Verilog/VHDL Editor* → *Code Style*. There, select *Space as Indent Character* and set the *Indent Size* to 3.

For the C style used in liboptimsochost and other libraries we have prepared a settings file. Go to *C/C++* → *Code Style* → *Formatter* and click on *Import...*, choose the settings file `doc/resources/optimsoc-eclipse-cdt-codestyle.xml` inside the OpTiMSoC source tree. Now you should have a new profile "OpTiMSoC" in the list of profiles. Choose this one for all work on the C code.

Creating the OpTiMSoC HDL Project

Now that all the basic settings are in place, we can create the projects inside Eclipse. First, we will create a project for editing the HDL (Verilog) code.

In the *Project Explorer* (on the left side), right click and select *New* → *Project...*. A new dialog window shows. In this window, select *Verilog/VHDL* → *Verilog/VHDL project* and click *Next*. Now enter a project name, e.g. "OpTiMSoC". Uncheck the option *Use default location* and click on *Browse* to choose your OpTiMSoC source directory (the location where you cloned the Git repository to).

The OpTiMSoC source tree not only contains RTL code, but also the necessary software components like liboptimsochost and the OpTiMSoC GUI. This code is better edited in a separate project and should be excluded from the project you just created. To do that, choose *Project* → *Properties* from the main menu. Navigate to *Resource* → *Resource Filters* and click on the *Add...* button. There, choose the following settings:

- Choose *Exclude all* in the group *Filter type*
- Choose *Folders* in the group *Applies to*
- Check *All children (recursive)*
- In the group *File and Folder Attributes* select *Project Relative Path matches* `src/sw`.

Now click on *OK* to finish editing the filter.

Then repeat the steps above to create a new resource filter but as path use `src/sysc` this time.

After you're done with the second filter, click *OK* again to close the dialog.

Creating a C Project

Eclipse is also a great choice for editing C code. As an example, we'll setup Eclipse for the OpTiMSoC host library, liboptimsochost.

In the main menu, click on *File → New → Project*. A dialog window is shown. There, nagivate to *C/C++ → Makefile Project with Existing Code* and click on the *Next* button. Type `liboptimsochost` as *Project Name* and click on *Browse...* to select the source code location of the library. It should be inside your OpTiMSoC code in the folder `src/sw/host/liboptimsochost`. Uncheck *C++* in the *Languages* group and select *GNU Autotools Toolchain* in the box below. Now click on *Finish* to close the dialog.

Before you start coding, double-check if the code style settings are correct. Select the newly created liboptimsochost project from the *Project Navigator* on the left, right-click and choose *Properties*. Nagivate to *C/C++ General → Formatter* and check if `OpTiMSoC` is selected as style. If not, click on *Enable project specific settings* and choose `OpTiMSoC` from the list. (If there is no such entry, go back to the basic Eclipse setup and import the style file properly.)

You can use the Eclipse GUI to build and run liboptimsochost, but this feature is currently not used by any of the developers. Instead, we only use Eclipse for code editing, and build and debug the software using the regular commands on the console.

4.1.2 Emacs

This section will be added shortly.

4.1.3 Verilog-mode

Sometimes, writing Verilog means writing the same information twice in different places of a source file, one example being the port of a module. To save you as developer some typing time, a tool called "Verilog-mode" has been invented. It allows you to specify comments inside your code where information should be placed, and this information is then calculated and inserted automatically. For more information about what it is and how it works, see `http://www.veripool.org/wiki/verilog-mode/`.

Verilog-mode is used extensively throughout the project. Even though using it is not required (the sources can be edited and compiled without it just fine), it will save you a lot of time during development and is highly recommended.

Installation is rather easy, as it comes bundled with GNU Emacs. Simply install Emacs as described above and you're ready to go. To support our coding style, you will need to adjust the Emacs configuration (even though it is the Emacs configuration, it also configures Verilog-mode).

Open the file `~/.emacs` and add the following lines at the end:

```
(add-hook 'verilog-mode-hook '(lambda ()
  ;; Don't auto-insert spaces after ";"
  (setq verilog-auto-newline nil)
  ;; Don't indent with tabs!
  (setq indent-tabs-mode nil)))
(add-hook 'verilog-mode-hook '(lambda ()
  ;; Remove any tabs from file when saving
  (add-hook 'write-file-functions (lambda()
    (untabify (point-min) (point-max))
    nil))))
```

If you also use Emacs as your code editor, Verilog-mode is already enabled and you can read through the documentation to learn how to use it.

Verilog-mode in Eclipse

Even if you use Eclipse, you do not need to switch editors to get the benefits of Verilog-mode; you can run Verilog-mode in batch mode to resolve all the AUTO comments. This will require some manual setup, but afterwards it can be used quite easily.

First, you need to figure out where your `verilog-mode.el` or `verilog-mode.elc` file is located. If you want to use the Verilog-mode which is part of your Emacs installation, it is probably located somewhere in `/usr/share/emacs`, e.g. `/usr/share/emacs/24.3/lisp/progmodes/verilog-mode.elc` on Ubuntu 14.04. You can run

```
$> find /usr/share/emacs -name 'verilog-mode.el*'
```

to search for it. If you found it, write down the path as we'll need it later. If you installed Verilog-mode from source, just note the path where you put your `verilog-mode.el` file (e.g. somewhere in your home directory).

In Eclipse, we will setup Verilog-mode as "Builder". To do so, click in the main menu on *Project* → *Properties* and nagivate to *Builders*. There, click on the *New...* button and select *Program* as configuration type in the shown dialog. After pressing *OK*, enter "verilog-mode" into the field *Name*. In the *Main* tab, write `/usr/bin/emacs` into the field *Location*. Leave the field *Working Directory* empty and enter the following string into the field *Arguments*:

```
--batch --no-site-file -u ${env_var:USER}
-l /usr/share/emacs/24.3/lisp/progmodes/verilog-mode.elc
"${selected_resource_loc}" -f verilog-auto -f save-buffer
```

Replace the path to the verilog-mode.el or verilog-mode.elc file with your own path you found out above.

Now, switch to the tab *Refresh*, check the box *Refresh resources upon completion* and select *The selected resource*. Since we don't need to change anything in the last two tabs, you can now close the dialog by clicking on the *OK* button and on *OK* again to close the project properties dialog.

To test if it all works, navigate to `src/rtl/compute_tile_dm/verilog/compute_tile_dm.v` and change the word "Outputs" in the comment right at the beginning of the file to something else. Then press CTRL-B (or go to *Project* → *Build All*) and after a couple of seconds, you should see the word "Outputs" restored and some output messages in the *Console* view at the bottom. Also check if there were no tabs inserted (e.g. at the instantiation of `u_core0`). If there are tabs then you probably did not setup your `~/.emacs` file correctly.

4.1.4 Qt Creator for GUI Development

Developing the OpTiMSoC GUI requires an IDE which understands the used Qt framework. The most popular choice among the developers is Qt Creator.

To start editing, open Qt Creator and click on *File* $\rightarrow$ *Open File or Project*. Now nagivate to `src/sw/host/optimsocgui` inside your OpTiMSoC source directory and open the file `CMakeLists.txt`. In the following dialog you can specify a build directory (or just leave the default). After clicking on *Next*, a dialog with the title *Run CMake* appears. Type `-DCMAKE_BUILD_TYPE=Debug` into the field *Arguments* and click on the *Run CMake* button. CMake is now run and if everything works as expected you can click on *Finish* to close the project creation wizzard and start hacking on the source code.