OpTiMSoC User Guide

March 24, 2017

Document Changes

Next Release

- Don't ship our own FuseSoC any more
- Temporarily deprecate GUI

2016.1

- Switched to FuseSoC-based build system
- Merged our own debug infrastructure with Open SoC Debug
- Port the basic 2x2 example to the Nexys 4 DDR board
- Adjust the tutorial and the installation instructions.

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

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.

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

Contents

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 for baremetal systems (newlib libc). Additionally, we have 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.

2 Installation & Configuration

Installing an OpTiMSoC release should not take more than 10 minutes (if you have a decent internet connection), so let's get started!

2.1 System Requirements

Please first check the system requirements.

- Supported Linux distribution: Ubuntu 14.04 or 16.04 LTS on x86_64
- 20 GB disk space (mostly for Xilinx Vivado and other tools)
- 4 GB or more of RAM helps greatly (especially during FPGA synthesis)
- We recommend not using a VM, but running directly on the hardware. Using a VM is possible, but will result in significantly slower compilation and synthesis runs.
- (optional, only if you want to do FPGA synthesis) Xilinx Vivado 2016.2 (the free WebPack edition is sufficient for most use cases)

Note

Some external tools, especially EDA tools for synthesis and simulators, 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.

2.2 Dependencies

In OpTiMSoC, we try to use only external dependencies which are known to be stable and readily available. Some can be installed as distribution packages, and others can be downloaded as binaries from us.

First, install all required packages from Ubuntu.

```
sudo apt-get -y install curl git build-essential tcl libusb-1.0-0-dev \
  libboost-dev libelf-dev swig python3 python3-yaml python3-pip

# optional, but highly recommended: a waveform viewer
sudo apt-get -y install gtkwave
```

We build our systems with FuseSoC, which is a package manager for RTL designs. It automatically arranges the sources and writes project files for our systems. You can install it simply from the Python package index:

```
sudo pip3 install "fusesoc>=1.6.1"
```

Additionally, we need two things which are not available as Ubuntu packages right now: a recent version of Verilator, and the or1k-elf-multicore toolchain (compiler, C library, debugger, etc.). Install them with our binary installation script:

```
# if it does not exist yet: prepare the /opt/optimsoc directory
sudo mkdir /opt/optimsoc
sudo chown $USER /opt/optimsoc

# download and install the prebuilt tools
curl -O https://raw.githubusercontent.com/optimsoc/prebuilts/master/
    ↪ optimsoc-prebuilt-deploy.py
sudo python optimsoc-prebuilt-deploy.py -d /opt/optimsoc verilator
    ↪ or1kelf
```

To use the prebuilt tools some environment variables need to be adjusted. This is done by running the following command **in every terminal session that you want to use OpTiMSoC in**:

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

Note

Automatically load the prebuilts in every new terminal session by adding it to your `~/.bashrc` file:

```
echo 'source /opt/optimsoc/setup_prebuilt.sh' >> ~/.bashrc
```

2.3 Install OpTiMSoC

Now that all preparations are done, you are now ready to install the OpTiMSoC framework itself. There are two options: either, you can install a prebuilt release package, or you can build OpTiMSoC yourself from the sources. We recommend starting with a binary release installation, and move to a custom-built version only after you verified that everything works.

2.3.1 Recommended: OpTiMSoC binary releases

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 packages that can be extracted into any directory and used directly

from there. The recommended default is to install OpTiMSoC into `/opt/optimsoc`. There are two packages: the `base` package contains the programs, libraries and tools to get started. The `examples` package contains prebuilt example systems (both in simulation and FPGA bitstreams) for the real quick start.

To install the 2016.1 release into `/opt/optimsoc`, run the following commands:

```
wget https://github.com/optimsoc/sources/releases/download/v2016.1/
    ↪ optimsoc-2016.1-base.tar.gz
wget https://github.com/optimsoc/sources/releases/download/v2016.1/
    ↪ optimsoc-2016.1-examples.tar.gz
tar -xf optimsoc-2016.1-base.tar.gz -C /opt/optimsoc
tar -xf optimsoc-2016.1-examples.tar.gz -C /opt/optimsoc
```

To use OpTiMSoC multiple environment variables need to be set. This is done by running the following command **in every terminal session that you want to use OpTiMSoC in**:

```
cd /opt/optimsoc/2016.1
source optimsoc-environment.sh
```

Note

Automatically load the OpTiMSoC environment in every new terminal session by adding it to your `~/.bashrc` file:

```
echo 'cd /opt/optimsoc/2016.1; source optimsoc-environment.sh' >> ~/.
    ↪ bashrc
```

Installation complete! You are now ready to go to the tutorials in Chapter 3.

2.3.2 Alternative: Build OpTiMSoC from sources

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.

First, you need some additional tools (the "build dependencies"):

```
sudo apt-get -y install texlive texlive-latex-extra texlive-fonts-extra
```

Then get the sources from git:

```
git clone https://github.com/optimsoc/sources.git optimsoc-sources
cd optimsoc-sources
# optional: checkout a release version
git checkout v2016.1
```

Now you're ready to build OpTiMSoC. OpTiMSoC contains a Makefile which controls the whole build process. Building is as simple as calling (inside the top-level source directory that you just got from git)

```
make
```

By default this also builds the documentation, the Verilator examples and the FPGA bitstreams (which requires Xilinx Vivado to be working). You can disable some features by passing variables to the `Makefile`:

```
# only build Verilator examples, but no bitstreams and no docs
make BUILD_EXAMPLES=yes BUILD_EXAMPLES_FPGA=no BUILD_DOCS=no
```

If you need even more fine-grained control over the build process, call the build script `tools/build.py` directly. Running `tools/build.py --help` will give you a list of all available options.

After the build process, all build artifacts are located in `objdir/dist`. You can either use OpTiMSoC directly from there (good during development), or copy it to a more suitable installation location in `/opt/optimsoc/VERSION` by running

```
make install
```

You can also modify the target directory using environment variables passed to `make`. This is especially useful if you don't have enough permissions to write to `/opt/optimsoc`.

- Use `INSTALL_PREFIX` to change the installation prefix from `/opt/optimsoc` to something else. The installation will then go into `INSTALL_PREFIX/VERSION`.
- Use `INSTALL_TARGET` to fully override the installation path. The installation will then go exactly into this directory.

```
# use INSTALL_PREFIX to install into ~/optimsoc/VERSION
make install INSTALL_PREFIX=~/optimsoc

# full control for special cases: use INSTALL_TARGET
# to install into ~/optimsoc-testversion
make install INSTALL_TARGET=~/optimsoc-testversion
```

Independent of which directory you chose, to use OpTiMSoC multiple environment variables need to be set. This is done by running the following command **in every terminal session that you want to use OpTiMSoC in**:

```
cd YOUR_INSTALLATION_DIR
source optimsoc-environment.sh
```

See the binary installation description above for information on how to make this change permanent.

OpTiMSoC is now ready to be used and you can continue with the tutorials in Chapter 3.

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 Embedded 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 `$OPTIMSOC/examples/sim/compute_tile`.

In addition to the SoC "hardware" (which, in this case, is simulated of course), you also need software that runs on the system. 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 built. They are essentially transformed versions of the ELF file, i.e. the software binary.

- `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

Now you have everything you need to run the hello world example on a simulated SoC hardware:

```
$OPTIMSOC/examples/sim/compute_tile/compute_tile_sim_singlecore --meminit
    ↪ =hello.vmem
```

And you'll get roughly this output:

```
TOP.tb_compute_tile.u_compute_tile.gen_cores[0].u_core.u_cpu.bus_gen.
    ↪ ibus_bridge: Wishbone bus IF is B3_REGISTERED_FEEDBACK
TOP.tb_compute_tile.u_compute_tile.gen_cores[0].u_core.u_cpu.bus_gen.
    ↪ dbus_bridge: Wishbone bus IF is B3_REGISTERED_FEEDBACK
[                  22, 0] Software reset
[               63128, 0] Terminated at address 0x0000e958 (status:
    ↪           0)
- ../src/optimsoc_trace_monitor_trace_monitor/verilog/trace_monitor.sv
    ↪ :89: Verilog $finish
```

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

```
# OpTiMSoC trace_monitor stdout file
# [TIME, CORE] MESSAGE
[               39614, 0] Hello World! Core 0 of 1 in tile 0, my absolute
    ↪  core id is: 0
[               48764, 0] There are 1 compute tiles:
[               57162, 0]  rank 0 is tile 0
```

Congratulations, you've ran your first OpTiMSoC system!

Note

If you are already familiar with embedded systems or microcontrollers, you might wonder: how did the `printf()` output from the software get into the `stdout.000` file if there is no UART or anything 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 `$OPTIMSOC/soc/sw/include/baremetal/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 See the Waves

One major benefit of simulating a SoC is the possibility to inspect every signal inside the hardware design quite easily. When running a Verilator simulation, as we did in the previous step, you can simply add the `--vcd` command line option. It instructs Verilator to write all signals into a file. You can then start a waveform viewer, like GTKWave to display it.

```
$OPTIMSOC/examples/sim/compute_tile/compute_tile_sim_singlecore --meminit
    ↪ =hello.vmem --vcd
```

This command will run the hello world example like it did before, but this time Verilator additionally writes a `sim.vcd` waveform file. You can now view this file.

```
gtkwave -o sim.vcd
```

The screenshot in Figure 3.1 is similar to what you should see when running GTKWave.

On the left side you find a hierarchy of all signals in the system. Add them to the wave view and explore all internals of a working SoC at your fingertips! Can you find the program counter? The instruction and data caches? The branch predictor?

3.3 Going Multicore: Simulate a Multicore Compute Tile

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.

Inside `$OPTIMSOC/examples/sim/compute_tile` you'll find a dual-core version and a quad-core version of the system with just one compute tile that you just simulated in the previous step. You can run those examples like you did before.

The first thing you observe: the simulation runs become longer. After each run, inspect the `stdout.*` files.

Welcome to the multicore world!

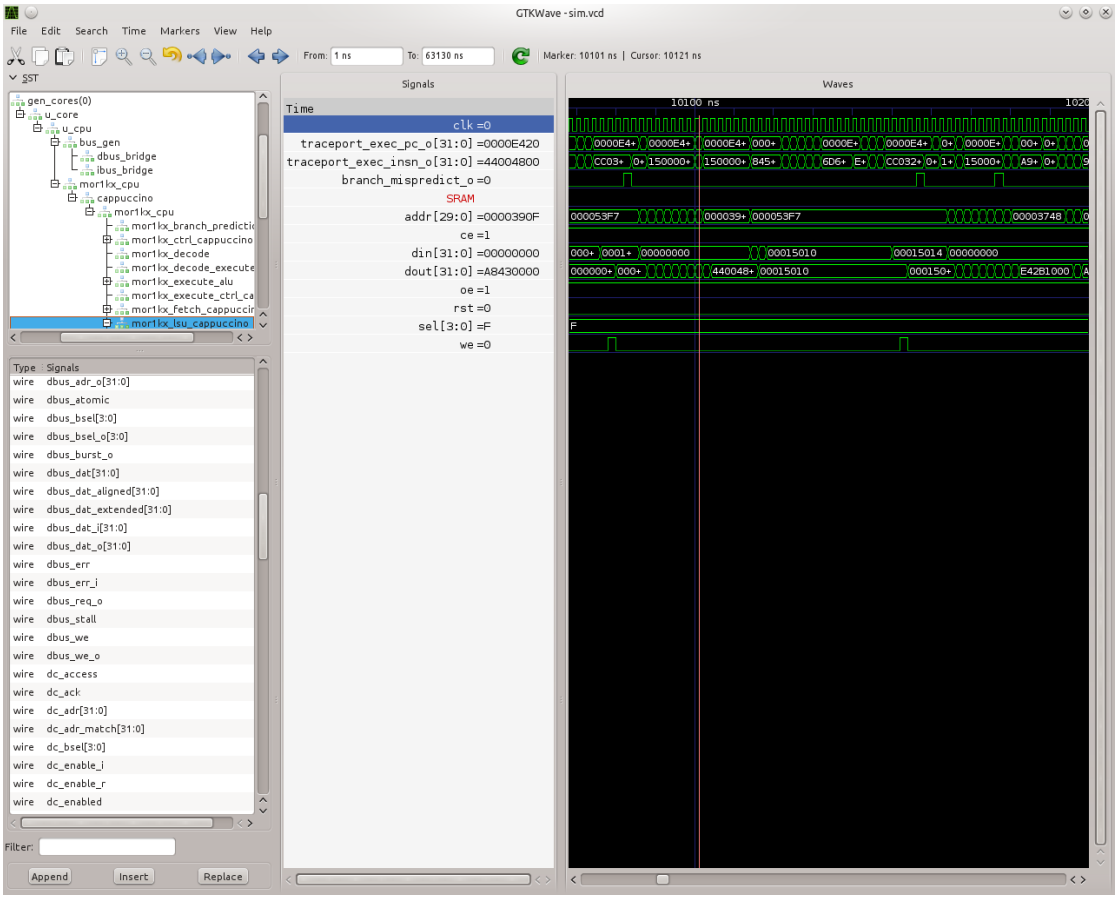

Figure 3.1: Screenshot of GTKWave displaying some signals from a running `compute_tile_dm` SoC.

3.4 Tiled Multicore SoC: Simulate a Small 2x2 Distributed Memory System

Next we want to run an actual NoC-based tiled multicore 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. In our pre-built example, each compute tile has two CPU cores, meaning you have eight CPU cores in total.

Execute it again to get the hello world experience:

```
$OPTIMSOC/examples/sim/system_2x2_cccc/system_2x2_cccc_sim_dualcore --
    ↪ 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 `$OPTIMSOC/soc/sw/include/baremetal/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!".

```
# start from the the baremetal-apps source code directory
cd hello_mpsimple
make
$OPTIMSOC/examples/sim/system_2x2_cccc/system_2x2_cccc_sim_dualcore --
    ↪ meminit=hello_mpsimple.vmem
```

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

```
$> cat stdout.000
# OpTiMSoC trace_monitor stdout file
# [TIME, CORE] MESSAGE
[               42844, 0] Wait for 3 messages
[               48734, 0] 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.

3.5 Observing Software During Execution: The Debug System

Up to now, you have seen the output of the software that runs on your SoC. And you had a look deep into the inner works of the SoC by looking at the waveforms.

In a real-world system, you need something in between: a way to observe the software as it executes on a chip, but without observing or understanding all the signals inside the hardware. This is what the debug system provides: hardware inside the chip which allows you to observe what's going on during software execution.

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 necessary 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 heterogeneous many-core SoC. Instead, the debug support in OpTiMSoC mainly relies on tracing. Tracing does not stop or otherwise influence the SoC itself; it only "records" what's going on during software execution, and transmits this data to the developer.

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.

After this introduction, let's make use of the debug system to obtain various traces. Just like in the previous examples, our SoC hardware is still running in Verilator. This tutorial works best if you have multiple terminal windows open at the same time, as we'll need to have multiple programs running at the same time.

So, open a new terminal (or a new tab inside your terminal), and start the simulation of the SoC hardware.

```
$OPTIMSOC/examples/sim/system_2x2_cccc/system_2x2_cccc_sim_dualcore_debug
```

Now, open a second terminal (leave the first one running!) and type

```
opensocdebugd tcp
```

This starts the "Open SoC Debug daemon." Open SoC Debug (or short, OSD) is the name of the debug infrastructure that's included with OpTiMSoC. The Open SoC Debug daemon is a tool which connects to the debug system inside the SoC hardware and interacts with it. In our case, since the SoC is running inside the Verilator simulation on the same PC, we use TCP to connect the simulated hardware `opensocdebugd`. Later, if we run the hardware on an FPGA, we'll use UART or USB instead of TCP – but all the commands stay the same.

After some seconds, `opensocdebugd` will output something like this.

```
Open SoC Debug Daemon
Backend: tcp
System ID: 0001
22 debug modules found:
 [0]: HOST
   version: 0000
 [1]: SCM
   version: 0000
 [2]: MAM
   version: 0000
   data width: 32, address width: 32
   number of regions: 1
     [0] base address: 0x0000000000000000, memory size: 33554432 Bytes
 [3]: STM
   version: 0000
   xlen: 32
 [4]: CTM
   version: 0000
   addr_width: 32
   data_width: 32
 [5]: STM
   version: 0000
   xlen: 32
 [6]: CTM
   version: 0000
   addr_width: 32
   data_width: 32
 [7]: MAM
   version: 0000
   data width: 32, address width: 32
   number of regions: 1
     [0] base address: 0x0000000000000000, memory size: 33554432 Bytes
 [8]: STM
   version: 0000
   xlen: 32
 [9]: CTM
   version: 0000
   addr_width: 32
   data_width: 32
 [10]: STM
   version: 0000
```

```
    xlen: 32
  [11]: CTM
    version: 0000
    addr_width: 32
    data_width: 32
... (we've skipped some output here) ...
Wait for connection
```

What you see is the output of the "debug system enumeration." Internally, the debug system consists of many modules. When first started, `opensocdebugd` first asks the SoC hardware for all available modules and prints them out. Without going into too much details, the most important ones are the following ones.

- The Memory Access Module (MAM) allows us to write and read memories inside the SoC from the host. We'll make use of this in a bit to load our software into the SoC.
- The System Trace Module (STM) is mainly responsible to transmit the output of all `printf()` calls to the host PC.
- The Core Trace Module (CTM) observes the software execution on the processor. We use it mainly to generate a function trace, i.e. a list of all software functions which have been called.

Why is there not just one of each modules? We're running a system with four tiles, each with two CPU cores. There are so many modules, because some of the debug modules are part of a tile, and some are attached to each CPU core. So this explains why there are four MAM modules, and eight CTM and STM modules each.

Let's go back to our terminals. Up to now we have two terminals open, let's open a third one. In here, we start `osd-cli`, a command line application that allows you to interact with the SoC hardware.

```
osd-cli
```

`osd-cli` supports many commands, and the `help` command is probably a good starting point.

```
osd> help
Available commands:
  help        Print this help
  <cmd> help  Print help for command
  quit        Exit the command line
  reset       Reset the system
  start       Start the processor cores
  mem         Access memory
  ctm         Configure core trace module
  stm         Configure software trace module
  terminal    Start terminal for device emulation module
  wait        Wait for given seconds
osd> mem help
Available subcommands:
```

```
help        Print this help
test        Run memory tests
loadelf     Load an elf to memory
```

Now let's run our hello world software on the SoC.

- First, we reset and then halt all CPUs. This gives us a "silent" system, i.e. nothing is running and we can modify the memory without being disturbed by the CPUs.

```
osd> reset -halt
```

- Next, we load the ELF file of the hello world program into the memory of compute tile 0. To do this, we tell the MAM module with ID 2 to write the file into the memory. (See the output of opensocdebugd for all IDs that are available.) After writing, the `-verify` option instructs `osd-cli` to read back all memory content and check if the read data is equal to the written data. This step is not strictly necessary, but is helpful to check that the memory write was successful indeed.

```
osd> mem loadelf hello.elf 2 -verify
Verify: 1
Load program header 0
Load program header 1
Verify program header 0
Verify program header 1
```

- Before we start the system, we want to observe what's going on when the software is executed. We therefore instruct the STM and CTM modules of core 0 to write log files. To the CTM we also pass the ELF file, i.e. the program that is executed. The CTM can use the information inside this file to record not only the program counter that is executed, but also tell you which function (as written inside the C code) a program counter refers to. This makes the CTM logs much nicer to read (at least for humans).

```
osd> stm log stm000.log 3
osd> ctm log ctm000.log 4 hello.elf
```

- Finally, we are ready to start the system, i.e. lower the reset signal.

```
osd> start
osd> [STM 003] 004616b5 Hello World! Core 0 of 2 in tile 0, my
    ↪ absolute core id is: 0
[STM 003] 0046266e There are 4 compute tiles:
[STM 003] 00463792  rank 0 is tile 0
[STM 003] 0046484d  rank 1 is tile 1
[STM 003] 00465918  rank 2 is tile 2
[STM 003] 004669ea  rank 3 is tile 3
```

Since we have written our hello world program only to core 0, we only get the `printf()` output from this core.

- Now that the software has finished, we can close the connection by typing

```
osd> quit
```

Remember that we instructed the STM and CTM modules to write log files? Have a look at the files `stm000.log` and `ctm000.log` to find all STM and CTM messages that were issued by the system. If possible the modules already assemble them back together to be more useful to the human user. For example, the STM creates the `printf()` output out of the trace messages (and you see both inside the file). The CTM uses the passed ELF file to resolve the function names that you see in the log file.

3.6 Automating System Interaction

In the previous section, you have manually typed commands into `osd-cli` to interact with the debug system. We understand that this is something you don't want to do all day. To make things easier, our debug components come with a Python interface that you can use to automate all the steps. To make it even more easy, you can use an example script that does exactly what you just typed manually: load all memories of a system and start the CPUs. The script then waits for ten seconds before it closes the connection to the `opensocdebugd`. (If your application runs longer than that adjust the script accordingly.)

```
# only Python 2 is supported at the moment
python2 $OPTIMSOC/host/share/opensocdebug/examples/runelf.py hello.elf
```

This ends our experiments with SoCs running as Verilator simulation. In the next sections, we'll move to an FPGA board and see how we can run software on that.

3.7 Our SoC on an FPGA

Welcome to the fun of real hardware! Before we can get started, you need to clarify some prerequisites.

3.7.1 Prerequisites: FPGA board and Vivado

This, of course, first means that you need borrow, buy or otherwise obtain an FPGA board. In this tutorial, we use the Nexys 4 DDR board by Xilinx/Digilent. It's not that expensive (of course, depending on your financial situation) and widely available. If you need help obtaining one, let us know – maybe we can help out in some way.

Additionally you need to download and install the Xilinx Vivado tool (the cost-free WebPack license is sufficient). We used the 2016.2 version when preparing this tutorial; we strongly recommend you also use this exact version.

Once you have obtained the FPGA board, connect it to the PC on the "PROG UART" USB connection. You don't need to connect any additional power supply.

3.7.2 Programming the FPGA

With the board connected, we can program (or "flash") the FPGA with our hardware design, the "bitstream." The OpTiMSoC release contains pre-built bitstreams for the single compute tile system and a 2x2 system with four compute tiles, meaning we can start directly with programming the FPGA.

There are two ways to program the device: using the Vivado GUI, or using the command line.

Programming the FPGA with the Vivado GUI

- Open Vivado (e.g. by typing `vivado` into a terminal window)
- On the welcome screen, click on "Hardware Manager"
- Ensure that your Nexys4 DDR board is plugged into your PC and is turned on.
- Click on "Open Target" in the green bar on the top, and then on "Auto Connect"
- Now click on "Program Device" in the same green bar and select the only option "xc7a100t_0" (that's the FPGA on the board).
- In the dialog window, select the bitstream file. We'll start directly with the larger 2x2 system, you can find the bitstream in `$OPTISMOC/examples/fpga/nexys4ddr/system_2x2_cccc/system_2x2_cccc_nexys4ddr.bit`.
- You can leave the other field "Debug probes file" empty.
- Click on "Program" to flash the bitstream onto the FPGA.

After a couple of seconds, your FPGA contains the SoC hardware and is ready to be used.

Programming the FPGA on the Command Line

```
optimsoc-pgm-fpga $OPTISMOC/examples/fpga/nexys4ddr/system_2x2_cccc/
    ↪ system_2x2_cccc_nexys4ddr.bit xc7a100t_0
```

3.7.3 Connecting

In the previous tutorials, we have already seen the debug infrastructure and connected to it over TCP. We now use the same tools to connect to our SoC, but this time we connect to the FPGA using UART. Fortunately, you don't need to connect any additional cables; the USB cable that you just used to program the FPGA is also the serial connection.

First, check which serial port was assigned to the board. Usually the easiest way is to do a

```
ls /dev/ttyUSB*
```

If you have only the Nexys 4 DDR board connected, you'll see only one device, e.g. `/dev/ttyUSB0`. Make note of this device name, and replace it accordingly in all the following steps in this tutorial.

Just as before, we'll need more than one terminal window. Open a first terminal and start `opensocdebugd` (remember to replace the device with your device name).

```
opensocdebugd uart device=/dev/ttyUSB1 speed=12000000
```

The output you see should be almost identical to what you've seen in Section 3.5, with one change: the system you're now using has just one CPU per compute tile, so only four cores in total. As consequence, you see less CTM and STM modules.

3.7.4 Running Software

Now that you've connected to the system, can you run software on it? Yes, you already know how! Open a new terminal window, and use `osd-cli` or the Python script to flash the memories with an ELF file and run the system.

When you run software, you'll notice two things: first, the output is the same as you've already seen when running the system in simulation. But: it's much faster. The FPGA runs at 50 MHz, which is still quite slow compared to current desktop processors, but still much faster than the simulation.

This concludes our tutorial session, and hands over to you: modify the software as you wish, program it again, analyze the simulations and explore your first multicore SoC.

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 Building Hardware

When building software, engineers have gotten used to tools like `make`, CMake and similar build systems. Such build tools ensure that all dependencies of a software project are met, and then start the various tools (such as the compiler, linker, etc.) to produce the output files, e.g. the program binary. In the hardware world, no standard tool for this job exists. A new, but very promising contestent in this sector is is *FuseSoC*.

FuseSoC allows developers to write *core files*: short declarative files in an INI-like format that describe which components are required to build a hardware design. When you look around in `$OPTIMSOC/soc/hw` you'll find such core files for all components that make up the SoC. But the core files not only describe the modules inside the SoC design, they are also used to describe the toplevel SoC.

For example, let's have a look at the file `$OPTIMSOC_SOURCE/examples/sim/compute_tile/compute_tile_sim.core` inside the OpTiMSoC source tree (it's not installed!). In there you find all dependencies that are needed to build the system with only one compute tile. You also find the toplevel files that are used to simulate the system in Verilator and in XSIM (the Vivado built-in simulator).

The great benefit of using FuseSoC is that you can now simply compile and run the system with one simple command.

Before we start, two notes:

- We set an environment variable (`$FUSESOC_CORES`) during the installation that makes FuseSoC find all OpTiMSoC hardware modules. You do not need to add a special configuration for this. However, the examples inside `$OPTIMSOC_SOURCE` are not part of this search path.
- You can call `fusesoc` from any directory. We recommend **not** calling FuseSoC from inside your source directory. (This allows you to just delete the build folder and retain a clean source folder.)

So let's look at a couple of examples how to build a SoC hardware with fusesoc.

Note

All the examples require an OpTiMSoC source tree to be available at `$OPTIMSOC_SOURCE`.

1. Build and run a Verilator-based simulation of a single compute tile.

```
cd some/directory
fusesoc --cores-root $OPTIMSOC_SOURCE/examples sim optimsoc:examples
    ↪ :compute_tile_sim
```

2. Only build a Verilator simulation of a single compute tile

```
fusesoc --cores-root $OPTIMSOC_SOURCE/examples sim --build-only
    ↪ optimsoc:examples:compute_tile_sim
```

3. Set the parameter `NUM_CORES` to 2 to create a system with two CPU cores inside the compute tile. You can have a look inside the top-level source file `$OPTIMSOC_SOURCE/examples/sim/compute_tile/tb_compute_tile.sv` for other parameters that are available.

```
fusesoc --cores-root $OPTIMSOC_SOURCE/examples sim --build-only
    ↪ optimsoc:examples:compute_tile_sim --NUM_CORES 2
```

4. Synthesize a 2x2 system with four compute tiles for the Nexys 4 DDR board using Xilinx Vivado. This step requires Vivado to be installed and working, and a lot of time (approx. 30 minutes, depending on your machine).

```
fusesoc --cores-root $OPTIMSOC_SOURCE/examples build optimsoc:
    ↪ examples:system_2x2_cccc_nexys4ddr
```

5. Now flash the bitstream that the previous step generated to the FPGA.

```
fusesoc --cores-root $OPTIMSOC_SOURCE/examples pgm optimsoc:examples
    ↪ :system_2x2_cccc_nexys4ddr
```

4.2 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!

4.2.1 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* → *Check for Updates*. Install all available updates.

For Verilog syntax highlighting we use a plugin called "VEditor". Go to *Help* → *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.

4.2.2 Emacs

This section will be added shortly.

4.2.3 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.

4.2.4 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* → *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.