OpTiMSoC User Guide

S. Wallentowitz et. al

January 28, 2013

Contents

Document Changes

January 28, 2013

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

1 Introduction

2 Getting started

2.1 Requirements

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.

2.1.1 Cross Compiler

You will need the `or32-elf-gcc` cross compiler to compile your code for the OpenRISC processor. Precompiled libraries can currently be found here: `http://opencores.org/or1k/OpenRISC_GNU_tool_chain`. This release has been tested with the current snapshot:

`ftp://ocuser:ocuser@openrisc.opencores.org/toolchain/openrisc-toolchain-ocsvn-rev789.tar.bz2`

Alternatively, you will need to build the cross compiler manually (can take some hours).

You will need an additional script `bin2vmem` to initialize the memory during simulation. It is distributed with ORPSoC and also available in the OpTiMSoC repository under `tools/utils`. Just compile it

```
> gcc -o bin2vmem bin2vmem.c
```

and install it somewhere in your PATH.

2.1.2 Simulation Tools

- Mentor Modelsim (tested with 10.1 and later)

2.1.3 Synthesis Tools

- Synopsys FPGA Synthesis (Synplify), tested with F-2012.03 and later
- Xilinx ISE, tested with 13.4 and later

2.2 LISNoC

LISNoC is the Network-on-Chip implementation OpTiMSoC builds on. You can find LISNoC at `http://www.lisnoc.org`. The source code is available from a git repository.

This document is written to match the tagged commit available in the repository. You need to create a local copy of the repository (line 1). It is recommended that you switch to the tag this document is tested with (line 2):

```
> git clone http://lis.ei.tum.de/git/lisnoc.git
> cd lisnoc; git checkout rel20130128
```

Two environment variables are needed in the following: `LISNOC` and `LISNOC_RTL`. You should either set them in your environment or start your session by sourcing a script in the LISNoC toplevel:

```
> source source.sh
```

2.3 OpTiMSoC

OpTiMSoC itself is a repository that contains all modules and software required to work with OpTiMSoC except LISNoC. The current version and updated information on OpTiMSoC can be found at the OpTiMSoC website: `http://www.optimsoc.org`.

The sources are again available as git repository. Again, you need to create a local copy of the repository (line 1) and it is recommended that you switch to the tag this document is tested with (line 2):

```
> git clone http://lis.ei.tum.de/git/optimsoc.git
> cd lisnoc; git checkout rel20130128
```

Again, two environment variables are needed: `OPTIMSOC` and `OPTIMSOC_RTL`. As for LISNoC, you should either set them in your environment or start your session by sourcing a script in the OpTiMSoC toplevel folder:

```
> source source.sh
```

3 Tutorials

3.1 Compute Tile and Software

It is a good starting point to simulate a single compute tile of a distributed memory system. Therefore a simple testbench 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 `tbench/rtl/dm/compute_tile`. You need to have Modelsim installed before you run:

```
> make build
```

The output should be as follows:

```
Model Technology ModelSim SE-64 vlog 10.1b Compiler 2012.04 Apr 26 2012
-- Compiling module tb_compute_tile
-- Compiling module trace_monitor
[..]
-- Compiling module lisnoc_mp_simple
-- Compiling module lisnoc_mp_simple_wb

Top level modules:
tb_compute_tile
```

In case you see errors check that the environment variables are set correctly.

The testbench can now be started in the Modelsim user interface or can be executed in command line mode using `make sim`, but there is no software loaded to the memory, what results in a warning:

```
# ** Warning: (vsim-7) Failed to open readmem file "ct.vmem" in read mode.
```

The simulations always expect vmem files that initialize the memories. This needs to be generated from the compiled source code. Before you build your own software you will need the support libraries. You can find all system software and example codes in `src/sw/system/dm/`.

In the system software folder you will find the `libbaremetal` library. Go to the `build` directory of the library and `make` it. The code should compile without errors and warnings.

The library contains many necessary functions, the boot code and helper macros. Two versions are build: `libbaremetal.a` is the normal distributed memory baremetal software and `libbaremetal-paging.a` is required when the local tile memory needs to be initialized from global memory, e.g., on an FPGA. This will later be extended to support smaller tile memories and do page swapping etc. This partially works, but an alternative approach to initialization and paging is described below with the PGAS system.

When you link the library to your own code, you need to link some external symbols as described in `sysconfig.h`. Those are definitions specific to a certain system instance, such as the dimensions, the presence of I/O, the distribution of compute ressources, etc.

Therefore you will always need to link your system-specific `sysconfig.o` to the library and your application objects, what we will describe in the following. The `sysconfig.c` can be found as part of the compute tile testbench (`${OPTIMSOC}/tbench/rtl/dm/compute\_tile`) under `sw`.

To build a simple "Hello World" example, simply switch to `sw`. The example application can be found in the software path. You should copy the whole path

```
> cp -r ${OPTIMSOC}/src/sw/system/dm/apps/baremetal/hello_simple sw/
```

Inside the `hello_simple` folder you can find the `hello_simple.c` and the `Makefile`. Now `make` the example. This will automatically also build the sysconfig.o and link all together to the elf file. Furthermore some other files are build:

- `hello_world.dis` is the disassembly of the file
- `hello_world.bin` is the elf file als loaded binary file
- `hello_world.vmem` is a textual copy of the binary file

(Note: If the latter is not build, ensure you have `bin2vmem` in your `PATH`, see 2.1.1).

You can now run the example. First go back to the compute tile testbench main folder. Before the simulation warned that `ct.vmem` cannot be found. Therefore we simply link the software to this filename

```
> ln -s sw/hello_simple/hello_simple.vmem ct.vmem
```

If you now run the software (`make sim`), the simulation should terminate with:

```
# Terminate @00001140
```

`0x1140` is the program counter (can vary depending on your cross compiler version) the simulation terminated. This is correct behavior and will be explained below. Furthermore you will find a file called `stdout` which shows the actual output:

```
[            42900000] Hello World!
```

The `[..]` part is the time stamp. But how does the actual printf-output get there when there is no UART or similar?

OpTiMSoC software (especially in future releases) makes excessive use of a tricky part of the OpenRISC ISA. The "no operation" instruction `l.nop` has a paramter `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`. If you have a look at the disassembly `hello_simple.dis` at instruction `0x1140` (see above) you exactly find this instruction. The actual action is then done with the trace monitor.

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 `utils.c` in `libbaremetal`):

```
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 "reports" usind a macro defined in `utils.h`:

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

This future is used extensively by future OpTiMSoC software.

3.2 Simulate Small 2x2 Distributed Memory System

Next you might want to build an actual multicore system. You can find such a system in `tbench/rtl/dm` directory as `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.

When you switch to this directory, you can build the system using `make`. In the following build the `hello_simple` software identical to the description above. After running `make sim` you will find the files `stdout.0` to `stdout.3`, each containing "Hello World".

There is a second "hello world" example available: `hello_simplemp`. This program uses the simple message passing facilities of the network adapter to send messages. In

that example all cores send a message to core 0, that prints a message when receiving the packets. When all cores sent their messags, the core acknowledges this by printing its own “Hello World” message.

Finally, a real world example is given by `heat`. Copy this example similar as above (do not forget to link the correct ct.vmem at all time!). This example calculates the heat distribution in a distributed manner. The cores coordinate their boundary regions by sending messages around.