**Title**

HW-SW SystemC Co-Simulation SoC Validation Platform

**Design Flow Report**

Contract No. 4200022968

by

 TU Braunschweig, Germany

October 29, 2012

**Document No.: IDA-SCSV-DF-010**

**Table of Contents**

1 Introduction 3

1.1 Purpose & Scope 3

1.2 Revisions 3

2 Socrocket design Flow 4

2.1 Overview 4

2.2 LEON FFT processor 7

2.2.1 Partitioning the system 7

2.2.2 Setting up a platform template 8

2.2.3 Configuring the template 12

2.2.4 Generating platform simulation and software 16

2.2.5 Running/Debugging simulation 18

2.2.6 Analysis of simulation results 20

**Table of Figures**

Figure 1 - SoCRocket Design Flow 4

Figure 2 - Usage of reconfigurable parameters in Exploration Prototype (EP) 5

Figure 3 - FFT data flow 7

Figure 4 – leon3mp (singlecore) 8

Figure 5 - Configuration Wizard / Welcome screen 12

Figure 6 - Configuration Wizard / Template window 13

Figure 7 - Configuration Wizard / Configuration window 13

Figure 8 – Configuration Wizard / Parameter window 14

Figure 9 - Configuration Wizard / Store configuration 15

Figure 10 - Memory map FFT processor 17

**Table of Tables**

Table 1 - Document Revisions 3

Table 2 - Sections of TPA template descriptor 11

# Introduction

## Purpose & Scope

This document is the Design Flow Report (DFR) of the SystemC Co-Simulation SoC Validation Platform (SoCRocket). It describes the natural way of assembling, building, executing and analyzing a platform simulation using the infrastructure of the library. The document walks the user through all steps of the flow at the hand of an example.

The DFR should be read in conjunction with the Interconnect Methodology Summary (IMS) and the Analysis Capability Report (AR). The IMS describes all the available interconnect options, and the AR the analysis API along the various parameters and performance counter for Design Space Exploration.

## Revisions

The following table will be updated during the course of the project.

|  |  |  |
| --- | --- | --- |
| **Version** | **Date** | **Description** |
| 0.1 | 21/02/12 | Initial draft version |
| 1.0 | 08/09/12 | Changes in infrastructure (leon3mp) |
|  |  |  |
|  |  |  |

Table 1 - Document Revisions

# Socrocket design Flow

## Overview

The SoCRocket SystemC modeling library provides models and tools for designing embedded systems for various applications. Primary use cases are design-space explorations at different levels of abstraction and development stages. Thanks to the built-in reconfiguration mechanism, hardware and software components can be parameterized without superfluous compilations and linking.



Figure 1 - SoCRocket Design Flow

The general design flow is depicted in Figure 1. The designer typically starts from a piece of reference software. In the domain of embedded computing the software is usually written in C/C++ language.

In the first step of the flow, the reference software is segmented into two parts: one that will become the software running on one of the target processors, and one that will become the hardware. Finding the right partitioning for a system is a complex task and usually requires multiple iterations.

In the next step, the structure of the system must be captured in an Exploration Prototype (EP). To establish the EP, the user needs to explicitly instantiate the hardware architecture (using C++ code) to be simulated. Instantiation and configuration of any model subject to exploration can be bound to configuration parameters. These parameters are extracted from the design and stored in XML notation, along with default values, value ranges, and descriptions. The example in Figure 2 demonstrates this procedure. Here only one configuration parameter is specified (**nProcessors**). It will be initialized from the underlying GreenSoCs/GreenControl API at runtime (line 3). The EP creates and binds processors and caches in a loop iterating from 0 to **nProcessors** (lines 5-18). The library provides one predefined EP, the **leon3mp**, which will be used as an example throughout this document. More information about the structure and syntax of Exploration Prototypes is given in section 2.2.1.

All parameters of all components are specified as GreenSocs parameters (**gs\_params**) and, hence, can be easily modified at runtime. The time-consuming step of compiling and linking the platform needs to be carried out only once. The result is a configuration-independent simulator in form of an executable containing the GreenControl API.

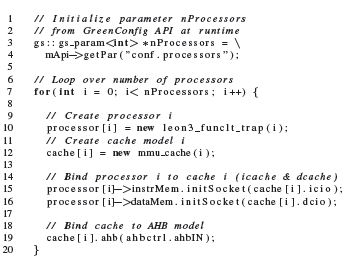


Figure 2 - Usage of reconfigurable parameters in Exploration Prototype (EP)

In order to run the simulation, parameters extracted from the EP must be properly defined. Therefore, the VP provides a small utility called the Configuration Wizard (CW). The CW parses the parameters extracted from the EP (see section 2.2.4). In front-end mode, the user may define all settings in a graphical interface. The interface presents the parameters in a hierarchically organized way. Using front-end mode is helpful for first-time configuration. It is also possible to load and store configurations. For running larger explorations, it is more appropriate to vary the default settings in the XML parameter description and generated configurations by calling the wizard in terminal mode. The outputs of the CW are the SW runtime configuration and the HW runtime configuration. Next to defining parameters, the CW interprets the system’s memory map, generates linker scripts for software mapping, and creates a compilation script for automatic integration of the new system in the platform’s build environment. How to run and debug simulation is explained in 2.2.5.

Running platform simulation still requires mapping the software portions of the design. The predefined EP provides software stacks for bare-metal simulation (with **newlib**) and **RTEMS OS**. To avoid recompilation after each change in the memory map, we follow a two-stage approach: first the application program is compiled into a configuration- and position-independent executable file using a SPARC Compiler. Afterwards, the **mkprom** tool from Aeroflex Gaisler fills in all configuration-specific options, adds the boot code, and compresses the resulting application executable file into a ROM image. At simulation begin, the boot code initializes the registered components, decompresses the application file, and copies all data to the final memory locations. This flow is equivalent to the way hardware simulations are initialized in GRLIB, only that the VP can directly load binaries files.

During simulation, the platform infrastructure gathers execution statistics. For this purpose every simulation model provides a set of performance counters. In the default configuration, the performance counters are written to the terminal after the end of the simulation. It is also possible to trace counters in log or waveform files, and to register handles for certain bounds and events. The analysis of the simulation results is the foundation for the verification of the design goals. The designer usually aims for requirements such as throughput and latency at the lowest possible cost. If the design goals are not met, configuration or partitioning must be optimized. This document only provides an example for using the analysis features of the platform (section 2.2.6). All available options are described in the Analysis Capability Report.

## LEON FFT processor

The following chapter explains all phases of the SoCRocket design flow on the basis of an example shipped with the library. The example demonstrates the implementation of a Fast Fourier Transformation\*. An on-chip sensor delivers the time-domain data. The sensor produces 1000 data frames per second, each consisting of 64 complex samples (512 bytes). The computed frequency domain data is supposed to be transmitted via a SoCWire link.



Figure 3 - FFT data flow

*\*The example is solely intended to demonstrate the features of the SoCRocket design flow. The presented hardware and software is by no means the optimum solution for the given problem. The foremost aim is to provide a representative system, which may be used as a starting point for other activities, and at the same time keep the different design tasks as simple and understandable as possible.*

### Partitioning the system

The starting point of the design activities is the FFT C-Code, which can be found in the **./software/fft64** directory.

The FFT shall be implemented in software on a LEON3 processor. A respective TL simulation model can be found in the SoCRocket IP library (**./models/extern/LEON3**). To get a first idea about the performance, the code may be executed on the stand-alone processor simulator. The simulator can be compiled by executing following command:

**./waf --target=leon3.funclt.platform**

For compiling the reference software enter:

**./waf –target=fft64.sparc**

The simulator binary and the program (sparc executable) will be dumped into the build directory of the library. To start the simulation run:

**./build/models/extern/LEON3/simulatorSources/leon3.funclt/leon3.funclt.platform -a ./build/software/fft64/fft64.sparc -f 100**

The execution statistic, which is generated at the end of the simulation, shows an instruction count of 20655. Considering a target clock rate of 100 MHz it seems feasible to execute the code 1000 times per second.

The library also provides a SoCWire model, an AMBA bus model and various peripherals. Modeling the Sensor requires a custom component. How to design own/custom components is explained in the SoCRocket User Manual. The resulting initial system is depicted in Figure 3.



Figure 4 – leon3mp (singlecore)

To enable design space exploration and mapping of the hardware dependent software portions, structure and parameters of the system must be captured in a design template.

### Setting up a platform template

In the past years the LEON processor provided by ESA and Aeroflex Gaisler has evolved into a standard target for mapping aerospace applications. Therefore, the template presented in this example has a high potential for re-use.

As previously explained a SoCRocket design template consists of a C++ file implementing **sc\_main** (**./platforms/leon3mp/sc\_main.cpp**) and a template descriptor for characterizing the various configuration parameters (**./templates/leon3mp.tpa**). In the following all of them will be explained in depth.

**C template – sc\_main.cpp**

The C template **sc\_main.cpp** implements the SystemC top-level function **sc\_main**. The method instantiates all the simulation models which are required by the design. The parameters of the different components are represented by **gs\_params** respecting following syntax:

**p\_{name of model}\_{hierarchical name of parameter}**

e.g.

**p\_ahbctrl\_ioaddr – ioaddr** parameter of model **ahbctrl**

**Definition:**

**gs::gs\_param<unsigned int> p\_ahbctrl\_ioaddr("conf.ahbctrl.ioaddr", 0xFFF);**

Components can be considered mandatory or optional. In the current example the **ahbctrl** is the central interconnect component and mandatory. Therefore, **ahbctrl** is instantiated directly:

120 // CREATE AHBCTRL unit

121 // ===================

122 // Always enabled.

123 // Needed for basic platform

124 AHBCtrl ahbctrl(**"ahbctrl"**,   
125 p\_ahbctrl\_ioaddr, // The MSB address of the I/O area

126 p\_ahbctrl\_iomask, // The I/O area address mask

127 p\_ahbctrl\_cfgaddr, // The MSB address of the configuration area

128 p\_ahbctrl\_cfgmask, // The address mask for the configuration area

129 p\_ahbctrl\_rrobin, // 1 - round robin, 0 - fixed priority

130 p\_ahbctrl\_split, // Enable support for AHB SPLIT response

131 p\_ahbctrl\_defmast, // Default AHB master

132 p\_ahbctrl\_ioen, // AHB I/O area enable

133 p\_ahbctrl\_fixbrst, // Enable support for fixed-length bursts

134 p\_ahbctrl\_fpnpen, // Enable full decoding of PnP configuration

135 p\_ahbctrl\_mcheck, // Check for intersections in memory map

136 p\_ahbctrl\_powmon, // Enable/disable power monitoring

137 ambaLayer

138 );

The instantiation of optional components depends on configuration parameters. In the **leon3mp** template this accounts for, e.g., the **GPTimer**. Only if the **gs\_param** **p\_gptimer\_en** is defined, the **GPTimer** will be instantiated and connected.

360 if (p\_gptimer\_en) {

361 // CREATE GPTimer

362 // ==============

363 GPTimer gptimer(**"gptimer"**,

364 p\_gptimer\_ntimers,// ntimers

365 p\_gptimer\_index, // index

366 p\_gptimer\_addr, // paddr

367 p\_gptimer\_mask, // pmask

368 p\_gptimer\_pirq, // pirq

369 p\_gptimer\_sepirq, // sepirq

370 p\_gptimer\_sbits, // sbits

371 p\_gptimer\_nbits, // nbits

372 p\_gptimer\_wdog, // wdog

373 p\_gptimer\_powmon // powmon

374 );

375 // Connecting APB Slave

376 apbctrl.apb(gptimer.bus);

377 // Connecting Interrupts

378 **for**(**int** i=0; i < 8; i++) {

379 signalkit::connect(irqmp.irq\_in, gptimer.irq, p\_gptimer\_pirq + i);

380 }

381 // Set clock

382 gptimer.set\_clk(LOCAL\_CLOCK,SC\_NS);

383 }

It is also possible to generate multiple instances of components in a loop. This technique is used for instantiating multiple processors.

It has already been shown above that the binding of sockets can also be hooked to parameters. The presented method has been proven to be very flexible and straightforward. The amount of flexibility in the template can be easily tuned.

**XML template descriptor – leon3mp.tpa**

The XML template descriptor is used to export and describe the parameters in the C++ template file. The purpose is to automate the definition of parameters and the generation of hardware dependent scripts for simulation (e.g. linking, WAF integration). The code abstract below shows the essential parts of the **leon3mp.tpa** template descriptor:

|  |
| --- |
| 1 <template name=**"Leon3mp Platform"**>  2 <description>  3 A standard LEON3 system: The template provides a LEON3 instruction set   simulator, models for caches and memory management unit (MMU\_CACHE),  4 a memory controller with PROM, SRAM and SDRAM simulation memories, an  interrupt controller and an AMBA AHB/APB interconnect.  5 The AHB2SoCWire bridge and the general purpose timer can be optionally  enabled.  6 <br/>  7 <h2>Features:</h2>  8 <ul>  9 <li>Direct elf file loading</li>  10 <li>GDB Stubs build in</li>  11 <li>Power Estimation</li>  12 <li>Timing Report</li>  13 <li>RAM Application - Initial PROM is included</li>  14 </ul>  15 </description>  16 <option name=**"Basic Configuration"** var=**"conf"**>  17 <option name=**"System Options"** var=**"system"**>  18 <option **var**=**"at"** name=**"Execute the simulation in AT mode"** type=**"bool"**   **default**=**"false"** hint=**"True - LT abstraction, False - AT**  **abstraction."**/>  19 <option **var**=**"clock"** name=**"Clock period in NS"** type=**"int"** **default**=**"10"**  hit=**"Assumes a single clock domain for all components"**/>  20 </option>  21 <option name=**"Reports"** **var**=**"report"** >  22 <option **var**=**"timing"** name=**"Enable timing report"** type=**"bool"**  **default**=**"true"** hint=**"If this switch is set to true the simulation**  **models will print a timing report at the end of simulation."**/>  23 <option **var**=**"power"** name=**"Enable power monitoring"** type=**"bool"**  **default**=**"false"** hint=**"If this switch is set to true the simulation**  **models will aggregate power data for all components. "**/>  24 </option>  25 <option name=**"Debugging"** **var**=**"gdb"**>  26 <option **var**=**"en"** name=**"Enable GDB support"** type=**"bool"** **default**=**"false"**  hint=**"If this switch is set to true the simulation models will include**  **GDB support"**/>  27 <option **var**=**"port"** name=**"Port to use for GDB connection"** type=**"int"**  **default**=**"1500"**/>  28 <option **var**=**"proc"** name=**"Processor to connect to"** type=**"int"**  **default**=**"0"**/>  29 </option>  31 <option var=**"ahbctrl"** name=**"AHB Controller"**>  32 <option var=**"ioaddr"** name=**"AHB IO Area Address"** type=**"int"**   **default**=**"0xFFF"** range=**"0..0xFFF"**/>  33 <option var=**"iomask"** name=**"AHB IO Area Mask"** type=**"int"**   **default**=**"0xFFF"** range=**"0..0xFFF"**/>  …  50 <option var=**"mcheck"** name=**"Intersection checking"**   type=**"bool"** **default**=**"true"**>  51 <description>  52 Check if there are any intersections between core memory areas.   In case of error issues a failure assertion.  53 </description>  54 </option>  55 </option>  …  78 <option var=**"mmu\_cache"** name=**"MMU Cache"**>  63 <option var=**"addr"** name=**"AHB Base Address"** type=**"int"** **default**=**"0"**  … 99 </option>  123 </option>  …  216 <option var=**"irqmp"** name=**"IRQ Controller"**>  …  222 </option>  ---------------------------------  224 </template> |

Table 2 - Sections of TPA template descriptor

The outermost tag (lines 1 – 124) defines the name of the template. It is only used for being displayed by the Configuration Wizard and has no other effect. The next section (lines 2 – 15) is also solely dedicated to the Configuration Wizard. It contains a general description of the platform and its features. Features are e.g. Direct ELF file loading, GDB support and Power Estimation. The general description should enable the user to understand the capabilities and limits of the template without having to look into the code itself. Line 16 opens the Basic Configuration Tag. Basic Configurations are System Options or IP Options. While System Options apply to the whole design, IP Options refer to a specific simulation model. The example above specifies System Options in lines 18 – 29. The user can switch between LT and AT mode; enable GDB stubs, and Power Monitoring. An example for IP Options is given in lines 31 – 55. The block describes the parameters of the AHBCTRL. Each parameter is assigned a name, a datatyp, a default value and a range. The **ioaddr** parameter in line 32 is of type integer, has a default of **0xfff** and may be modified to any value in the range of **0x000 – 0xfff**. As can be seen in lines 50 – 55 it is also possible to attach additional explanations **(<description>**).

### Configuring the template

The purpose of a configuration is the initialization of the configuration parameters exported by the platform template. A single template can have an arbitrary number of configurations, each representing an independent instance of the design. Configurations can be created manually or with the help of the SoCRocket Configuration Wizard. The SoCRocket Configuration Wizard has already been explained in the Library User Manual. It can be used in Terminal Mode or GUI Mode. GUI Mode is recommended for creating a first or a completely new configuration, which usually requires a large number of parameters to be set. The Terminal Mode, on the other hand, provides a quick way of generating a simulation from a configuration, which already exists and needs to be slightly modified (see 2.2.4).

To create the initial configuration for the FFT processor, we are going to use GUI mode. The tool can be started with following command:

**./waf generate**

This brings up the wizard welcome screen (Figure 5). Click next to proceed to the template window (Figure 6). From the template window select the **leon3mp** platform template as a starting point. The selection must be confirmed by pressing the next button.

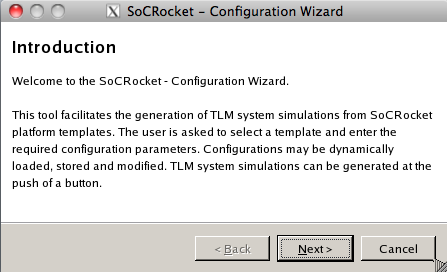


Figure 5 - Configuration Wizard / Welcome screen

In the next step the user can choose between creating a new configuration for the selected template, or for loading a previously stored configuration. The configuration window (Figure 7) shows at least 6 preexisting configurations, related to designs with 1-6 CPUs. These configurations have been used for the example design space exploration, which is described in the SoCRocket HL-DSE report.

For the FFT processor we are creating a new “defaults configuration”. Pressing the next button brings up the main parameter window (Figure 8). Scrolling through the parameter window exposes all the variables and options exported by the template descriptor (see section 2.2.2). System options are shown first. A single click on a parameter will bring up a detailed description. Clicking in the value field of a parameter allows the parameter to be modified.



Figure 6 - Configuration Wizard / Template window



Figure 7 - Configuration Wizard / Configuration window

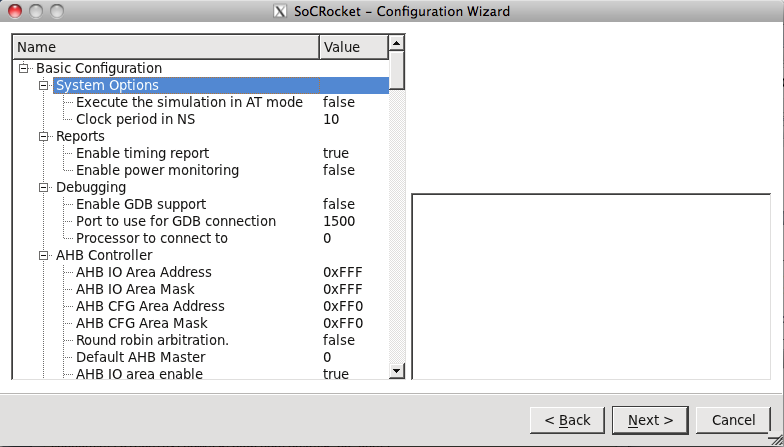


Figure 8 – Configuration Wizard / Parameter window

In order to create the configuration for the FFT processor (Figure 3) and implement the memory map shown in Figure 10 perform following steps:

1. Select AT mode for architecture exploration:**Execute the simulation in AT mode = true**
2. Disable Power Monitoring:  *Enable Power Monitoring = false*
3. Configure APB controller:**base address = 0x800, address mask = 0xfff, bus index = 2**
4. Switch-off caches and mmu:**Enable instruction cache = false, Enable data cache = false,****Enable MMU = false**
5. Switch on AHB Memory:**AHB Memory = true**
6. Configure Memory Controller:**PROM configuration = true, Address Space Base = 0x000,   
   Address Space Mask = 0xE00**
7. Disable IO memory:**IO memory configuration = false**
8. Configure RAM:**Address Space Base = 0x400, Address Space Mask = 0xff0**
9. Make sure SRAM is disabled and SDRAM is enabled:**SRAM configuration = false, SDRAM configuration = true**
10. Enable ahbin (sensor):**set master\_id = 1 and interrupt = 13**
11. Enable SoCWire:**SoCWire = true**

All other settings keep their default values.

Store the configuration under the name **leon\_fft\_initial** (Figure 9). Press the next button and complete the configuration.

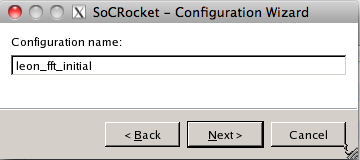


Figure 9 - Configuration Wizard / Store configuration

The configuration file will be written to the **./templates** directory. The code example below shows an abstract of the **leon3mp.leon\_fft\_initial.json** file:

2 **"conf"**: {

3 **"ahbctrl"**: {

4 **"cfgaddr"**: 4080,

5 **"cfgmask"**: 4080,

6 **"defmast"**: 0,

7 **"fixbrst"**: false,

8 **"fpnpen"**: true,

9 **"ioaddr"**: 4095,

10 **"ioen"**: true,

11 **"iomask"**: 4095,

12 **"mcheck"**: true,

13 **"rrobin"**: false,

14 **"split"**: false

15 },

... ...

134 **"report"**: {

135 **"power"**: false,

136 **"timing"**: true

137 },

150 **"system"**: {

151 **"clock"**: 10,

152 **"at"**: false,

152 **"ncpu"**: 1

153 },

Lines 3 – 15 contain the settings for the AHBCTRL. The reporting options are shown in lines 134-137 and the system options in lines 150 – 153. Thanks to the human-readable format of the file (**json** language), the configuration can be easily modified using any text editor without having to pass through the Configuration Wizard again.

***Remark:*** *The system is completely configured at runtime, solely based on the* **json** *config file! No additional build targets (WAF) are required. The configuration can be specified using the* **--jsonconfig** *switch of the* **leon3mp** *executable.*

### Generating platform simulation and software

Generating a platform simulation requires a set of input files. This section explains where these files are located and how, if necessary, to compile them.

**Exploration Prototype:**

The files of the **leon3mp** exploration prototype can be found in the **./platforms/leon3mp** directory.

**sc\_main.cpp** – Top-level of SystemC simulation (instantiates all modules with respect to configuration parameters (from **json** file)

**json.lua** – Configuration parser in LUA language. Responsible for initializing all configuration parameters (**gs\_params**) at begin of the simulation.

**wscript** – WAF build script for the **leon3mp** prototype

For compiling the system simulation enter:

**./waf --target=leon3mp.platform**

**Boot Code:**

The system boot code is located in directory **./software/prom/default.**

**prom.ld** – Linker script for generating default boot code

The linker script is bound to bare-metal systems. The application entrance point is assumed to be at the beginning of the RAM area (SRAM or SDRAM). The boot code itself can be located at any memory location. The processor simulator will start from the address specified in label **\_start**.

**prom.h/S** – Default system boot code (depends on memory map)

The boot code delivered with the example is intended for bare-metal systems. It is supposed to be compiled using the Aeroflex Gaisler BCC compiler and bare-metal runtime library.

**wscript** – WAF build script for compiling boot code with BCC compiler

To compile the boot code (**prom.S**) use following command:

**./waf --target=default.prom**

**Application Software:**

Application code and build script can be found in the **./software/fft64** directory. In the previous section was explained how to execute the code on the stand-alone simulator. Mapping the code to the newly generated platform requires some modifications. As already mentioned, the input data is supposed to be delivered by a sensor device and shall be sent via a SoCWire link after computation. The modified/mapped version of the code is given in the **radix4\_input\_gen.c** file. The code demonstrates how to catch an interrupt from the sensor device (**ahbin**) and how to set-up/control the SoCWire IP. The sensor delivers a total of 10 data frames in intervals of 10 ms. The SoCWire model operates in loop-back mode. Both, RX and TX descriptors are provided.

In order to compile the code using the SPARC bare-metal compiler type:

**./waf –target=fft64\_input\_gen.sparc**

Figure 10 shows the memory map of the design. The ROM is located at address **0x0**; the SDRAM starts from address **0x40000000**, and the AHBMEM from address **0xa0000000**. In order to boot correctly, ROM and SDRAM must be initialized at begin of simulation. Name and location of the image files for loading the memories must be defined in the system configuration (**json** file). AHBMEM is only used for buffering data at runtime.



Figure 10 - Memory map FFT processor

### Running/Debugging simulation

As previously mentioned all simulations based on the **leon3mp** template expect two SPARC executable to be defined in the system configuration (**json** file). One of them initializes the PROM and the other one the SDRAM.

In order to start the GDB Server set **conf.gdb.en = true** (**json** file or wizard)

If the json file contains all required settings simulation can be started by:

**./build/platforms/leon3mp/leon3mp.platform**

Alternatively, missing parameters can be provided on the command line:

**./build/platforms/leon3mp/leon3mp.platform**

**-j ./templates/leon3mp.leon\_fft\_initial.json   
 -o conf.mctrl.prom.elf=build/software/prom/default/default.prom   
 -o conf.mctrl.ram.sdram.elf=build/software/fft64/  
 fft64\_input\_gen.sparc   
 -o conf.system.osemu=build/software/fft64/fft64\_input\_gen.sparc**

After starting the simulation all models print configuration reports. The configuration reports contain all relevant constructor parameters and can be used to verify the settings applied in the configuration step. The example below shows the configuration report of the MCTRL:

**@0 s /0 (mctrl): Info: \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\***

**@0 s /0 (mctrl): Info: \* Created MCTRL with following params:**

**@0 s /0 (mctrl): Info: \* ---------------------------------------**

**@0 s /0 (mctrl): Info: \* romasel: 28**

**@0 s /0 (mctrl): Info: \* sdrasel: 29**

**@0 s /0 (mctrl): Info: \* romaddr/rommask: 0/e00**

**@0 s /0 (mctrl): Info: \* ioaddr/iomask: 200/e00**

**@0 s /0 (mctrl): Info: \* ramaddr/rammask: 400/c00**

**@0 s /0 (mctrl): Info: \* paddr/pmask: 0/fff**

**@0 s /0 (mctrl): Info: \* wprot: 0**

**@0 s /0 (mctrl): Info: \* srbanks: 0**

**@0 s /0 (mctrl): Info: \* ram8: 1**

**@0 s /0 (mctrl): Info: \* ram16: 1**

**@0 s /0 (mctrl): Info: \* sepbus: 0**

**@0 s /0 (mctrl): Info: \* sdbits: 32**

**@0 s /0 (mctrl): Info: \* mobile: 0**

**@0 s /0 (mctrl): Info: \* sden: 1**

**@0 s /0 (mctrl): Info: \* hindex: 0**

**@0 s /0 (mctrl): Info: \* pindex: 0**

**@0 s /0 (mctrl): Info: \* pow\_mon: 0**

**@0 s /0 (mctrl): Info: \* abstractionLayer (LT = 8 / AT = 4): 4**

It can be seen that the AHB address/mask pairs for ROM (**romaddr/rommask**) and RAM (**ramaddr/rammask**) have been properly set and the APB interface populates the correct memory region (**paddr/pmask**).

If the **gdb** switch was set, the simulation will block after printing the configuration reports:

**GDB: waiting for connections on port 1500**

The user can now connect the debugger of its choice and control the simulation by stepping through the code running on the LEON processor.

For more detailed information on how to use GDB together with the LEON processor, please check the TRAP user manual.

If the system contains a serial interface (UART), it will also ask for a terminal to be connected:

**UART: waiting for connections on port 2000**

You may connect e.g. via telnet in the following way:  
  
**telnet localhost 2000**

In the next step the simulation will produce reports for all decoders in the design. Decoders as implemented in the AMBA bus or the MCTRL are initialized using the SystemC **start\_of\_simulation** callback. All internal decoding tables are built up at runtime depending on the information provided by the connected master/slave sockets:

**@0 s /0 (ahbctrl): Info: \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\***

**@0 s /0 (ahbctrl): Info: \* AHB DECODER INITIALIZATION**

**@0 s /0 (ahbctrl): Info: \* --------------------------**

**@0 s /0 (ahbctrl): Info: \* SLAVE name: apbctrl**

**@0 s /0 (ahbctrl): Info: \* BAR0 with MSB addr: 0x800 and mask: 0xfff**

**@0 s /0 (ahbctrl): Info: \* BAR1 not used.**

**@0 s /0 (ahbctrl): Info: \* BAR2 not used.**

**@0 s /0 (ahbctrl): Info: \* BAR3 not used.**

**@0 s /0 (ahbctrl): Info: \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\***

**@0 s /0 (ahbctrl): Info: \* SLAVE name: mctrl**

**@0 s /0 (ahbctrl): Info: \* BAR0 with MSB addr: 0x0 and mask: 0xe00**

**@0 s /0 (ahbctrl): Info: \* BAR1 with MSB addr: 0x200 and mask: 0xe00**

**@0 s /0 (ahbctrl): Info: \* BAR2 with MSB addr: 0x400 and mask: 0xc00**

**@0 s /0 (ahbctrl): Info: \* BAR3 not used.**

**@0 s /0 (ahbctrl): Info: \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\***

**@0 s /0 (ahbctrl): Info: \* Master name: mmu\_cache**

**@0 s /0 (ahbctrl): Info: \* BAR0 not used.**

**@0 s /0 (ahbctrl): Info: \* BAR1 not used.**

**@0 s /0 (ahbctrl): Info: \* BAR2 not used.**

**@0 s /0 (ahbctrl): Info: \* BAR3 not used.**

**@0 s /0 (ahbctrl): Info: \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\***

**@0 s /0 (apbctrl): Info: \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\***

**@0 s /0 (apbctrl): Info: \* APB DECODER INITIALIZATION**

**@0 s /0 (apbctrl): Info: \* --------------------------**

**@0 s /0 (apbctrl): Info: \* Slave name: mctrl**

**@0 s /0 (apbctrl): Info: \* BAR with MSB addr: 0 and mask: fff**

**@0 s /0 (apbctrl): Info: \* Slave name: irqmp**

**@0 s /0 (apbctrl): Info: \* BAR with MSB addr: 1f0 and mask: fff**

**@0 s /0 (apbctrl): Info: \* Slave name: gptimer**

**@0 s /0 (apbctrl): Info: \* BAR with MSB addr: f0 and mask: fff**

**@0 s /0 (apbctrl): Info: \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\***

The report confirms structure and address map of the design. It can be seen that there is only one Master in the system: **mmu\_cache**. In the presented example the **mmu\_cache** IP contains neither caches nor mmu. It solely provides the AHB interface for the LEON processor (comparable to HW **acache** component).

There are two slaves connected to the AHBCTRL: **apbctrl** and **mctrl**. The **mctrl** has three sub-components relating to **prom**, **io** and **sdram**. The APB decoder connects the register interfaces of **mctrl**, interrupt controller (**irqmp**) and timer (**gptimer**).

Without user interventions the simulator will now process the boot code, jump to the application program, which is located at the beginning of the RAM area, and run the application until the **\_exit** label (from C-library) is hit.

Depending on the verbosity level of the simulation more or less output will be generated. For information on setting verbosity levels and configuring the library, please see the SoCRocket User Manual.

### Analysis of simulation results

The simulation sequence can be verified by observing the terminal output. The sensor generates debug output on **info** level. The program contains **printf** statements providing status information, which will be send to a user terminal via the UART serial interface. In case of correct execution the following messages are written to **stdout** in intervals of 10 ms simulation time:

**@1 ms /40213 (sensor): Info: Start sending new data frame!**

**@1012780 ns /40981 (sensor): Info: Transmission of frame completed**

**CPU: Received IRQ**

**CPU: Completed FFT**

**CPU: Start SoCWire DMA action**

The simulation terminates after 10 iterations.

At the end of the simulation all simulation models automatically print execution statistics. By these statistics conclusions may be drawn regarding efficiency and correctness of hardware and software. The example below shows the execution report of the SoCWire model. In total 10 TX and 10 RX descriptors have been consumed.

@91995310ns Report: \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  
@91995310ns Report: \* AHB2SoCWire Statistic:  
@91995310ns Report: \* --------------------------------------------------------------------------------  
@91995310ns Report: \* SoCWire packets received   
 (total, with error RE\_ReceiverError, without error): 11, 0, 10  
@91995310ns Report: \* Bytes received successfully on SoCWire link: 5120  
@91995310ns Report: \* SoCWire packets sent (without error): 10  
@91995310ns Report: \* Bytes sent on SoCWire link: 5120  
@91995310ns Report: \* Packets received without available RX descriptor (SS\_SocWireStall): 0  
@91995310ns Report: \* RX descriptors consumed: 10  
@91995310nsReport: \* TX descriptors consumed: 10  
@91995310ns Report: \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

A detailed explanation of all available execution reports can be found in the SoCRocket Analysis Capability Report. The document also contains a complete list of all performance counters in the systems and instructions on how to use the build-in analysis API of the library. Performance counters are implemented using the GS\_PARAM mechanism provided by GreenSoCs/GreenControl. Respectively, every platform modeled with the SoCRocket library can be extended using GreenControl surveillance tools (e.g. GreenAV). This enables the user to:

* Access all counters at every time from everywhere in the simulation
* Directly log changes in trace-files
* Directly log changes in waveforms (vcd)
* Register/execute callbacks on changes
* Overwrite counters