**Title**

HW-SW SystemC Co-Simulation SoC Validation Platform

**Design Flow Report**

Contract No. 4200022968

by

 TU Braunschweig, Germany

March 23, 2012

**Document No.: IDA-SCSV-DF-001**

**Table of Contents**

1 Introduction 2

1.1 Purpose & Scope 2

1.2 Revisions 2

2 Socrocket design Flow 3

2.1 Overview 3

2.2 LEON FFT processor 5

2.2.1 Partitioning the system 5

2.2.2 Setting up a platform template 6

2.2.3 Configuring the template 10

2.2.4 Generating platform simulation and software 15

2.2.5 Running/Debugging simulation 16

2.2.6 Analysis of simulation results 18

**Table of Figures**

Figure 1 - SoCRocket Design Flow 4

Figure 2 - FFT data flow 6

Figure 3 - Initial System 7

Figure 4 - Configuration Wizard / Welcome screen 12

Figure 5 - Configuration Wizard / License agreement 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 14

Figure 10 - Memory map FFT processor 17

# 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 |
|  |  |  |
|  |  |  |
|  |  |  |

# Socrocket design Flow

## Overview

The SoCRocket Design Flow (DF) (Figure 1) provides a method for designing TL system models of embedded hardware for various applications. Primary use cases are early software development and design space exploration. Flow and tools have been particularly developed for the aerospace domain, but are by no means restricted to it.

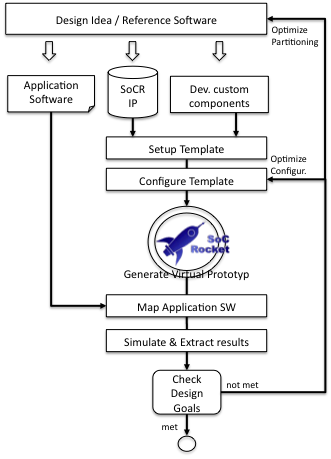


Figure - SoCRocket Design Flow

The flow typically starts from a piece of reference software implementing a design idea. In the domain of embedded computing the software is usually written in C/C++ language. Although, this is not a requirement, and any other language may be selected, it is recommended to use C/C++, because all simulation models of the SoCRocket library are written in SystemC, which is an extension to the C++ standard.

In the first step of the flow the reference software is segmented to create an initial partitioning. The partitioning represents the decision, which parts of the reference software will become software running on one of the target processors and which will become dedicated hardware. Finding the right partitioning for a system is a complex task and usually requires multiple iterations.

For assembling the initial system the SoCRocket library provides a set of simulation IPs representing core components from the Aeroflex Gaisler GRLIB hardware library. This includes a LEON instruction set simulator (ISS), AHB/APB interconnect and peripherals. The library contains SpaceWire and SoCWire models, and is constantly extended. The designer may introduce own custom components, e.g. hardware accelerators for crucial parts of the code. Guidelines for creating and integrating own simulation models can be found in the SoCRocket User Manual.

In the next step the structure of the system must be captured in a template. In the given context a template consists of a SystemC top-level file containing an implementation of sc\_main, a template descriptor file (.tpa), a build script generator and a linker template. To facilitate design space explorations the SystemC top-level file may contain an arbitrary number of variables representing constructor parameters or design options (e.g. cache size or number of CPUs). To hook up with infrastructure and tools, these variables must be exported. This is done in the descriptor. The descriptor is an XML file specifying all parameter, their default value and range. The library provides two predefined platform templates (see ./templates dir). More information about the structure and syntax of templates is given in section 2.2.1.

To compile and run a platform simulation the parameters exposed by the template must be properly initialized. For very simple systems this can be manually done. More comfortable means are provided by the SoCRocket Platform Wizard. The Wizard parses the template descriptor (tpa) and displays all available parameters in a graphical user interface. The parameters are shown along their descriptions and default values. The user can adapt the settings, store the configuration and generate a system simulation at the touch of a button. Configurations and Platform Wizard are explained in section 0.

Running platform simulation still requires mapping the software portions of the design. Generating the respective memory images depends on the system memory map. The Platform Wizard interprets the memory map, generates appropriate scripts for linking and integrates the newly generated platform in the build environment (section 2.2.4). The build system compiles the platform into an executable simulator. It is recommended to pass name and location of the memory images (ELF files) as command line parameters. How to run and debug simulation is explained in section 2.2.5.

During simulation the platform infrastructure records execution statistics. For this purpose every simulation model of the library provides sets 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. 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.

The results of the analysis step are supposed to support the verification of the design goals. The designer usually aims for requirements such as throughput and latency at the lowest possible cost. Thereby costs typically are silicon area or power consumption. Correct prediction of implementation costs is not completely possible at transaction level, because target technology and synthesis strategy have a big impact on the final result. Nevertheless, TL models are capable of delivering a strong indicator (see SoCRocket Power Modeling Report). In terms of timing accuracy the AT-mode of the provided models has proven to be very close to results from RTL simulation (see IP Verification and Performance Report). If the benchmarks are positive and further optimizations do not appear promizing the actual hardware implementation may start. Otherwise the designer can decide to modify parameters (optimize configuration), or alter the partitioning, and try again.

Next to the Approximately Timed AT-mode all models of the SoCRocket library provide a Loosely-Timed LT-mode for fast register/address accurate simulation. This mode is intended to speed-up the software mapping. It can also be used to optimize the software while hardware implementation is in progress, or to develop additional software after hardware is completed.

The remainder of this documents guides the user through all phases of the design flow at the example of a single-core LEON design. All discussed code can be found in the **./templates** and the **./software** directories of the library.

## 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\*. The time-domain data is delivered by an on-chip sensor. 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 - 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 C-Code for a Radix4 FFT, 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**

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/funcLT/LEON3\_funcLT -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 feasable 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 SoCWire User Manual. The resulting initial system is depicted in Figure 3.

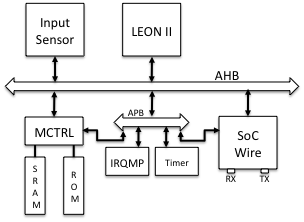


Figure - Initial System

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**, a template descriptor for exporting the various configuration parameters, a build script generator and a linker template. All template files for this example are located in the **./templates/singlecore** directory. In the following all of them will be explained in depth.

**C template - main.cpp**

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

**conf\_{name of model}\_{name of constructor parameter}**

e.g. **conf\_ahbctrl\_ioaddr – ioaddr parameter of model ahbctrl**

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

120 // CREATE AHBCTRL unit

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

122 // Always enabled.

123 // Needed for basic platform

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

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

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

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

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

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

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

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

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

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

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

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

137 ambaLayer

138 );

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

360 #if conf\_gptimer != 0

361 // CREATE GPTimer

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

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

364 conf\_gptimer\_ntimers,// ntimers

365 conf\_gptimer\_index, // index

366 conf\_gptimer\_addr, // paddr

367 conf\_gptimer\_mask, // pmask

368 conf\_gptimer\_pirq, // pirq

369 conf\_gptimer\_sepirq, // sepirq

370 conf\_gptimer\_sbits, // sbits

371 conf\_gptimer\_nbits, // nbits

372 conf\_gptimer\_wdog, // wdog

373 conf\_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, conf\_gptimer\_pirq + i);

380 }

381 // Set clock

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

383 #endif

It is also possible to generate multiple instances of components in a loop. This technique is used in an alternative architecture template, which is also delivered with the library (multicore). In the multicore template the number of CPUs in the system depends on a configuration parameter.

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 straight forward. The amount of flexibility in the template can be easily tuned. Setting up a system for design space exploration requires neither complicated tools nor proprietary techniques.

**XML template descriptor – singlecore.tpa**

The XML template descriptor is used to export and describe the parameters in the C template file. The purpose is to automize 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 **singlecore.tpa** template descriptor:

1 <template name=**"Single-Core Platform"**>

2 <description>

3 Default platform template containing one LEON3 processor, with MMU and Cache, AHB/APB bus system, interrupt controller and a fully   
 configurable memory controller.   
 4 GPTimer, SoCWire and a custom Sensor Device   
 can be optionally enabled.

5 <br/>

6 <h2>Features:</h2>

7 <ul>

8 <li>Direct ELF file loading</li>

9 <li>GDB stubs build-in</li>

10 <li>Power Monitor</li>

11 <li>Timing Report</li>

12 <li>Simulation Statistics</li>

13 <li>Automatic boot-code and memmap alignment</li>

14 </ul>

15 </description>

16 <option name=**"Basic Configuration"** var=**"conf"**>

17 <option name=**"System Options"** var=**"sys"**>

18 <option var=**"lt\_at"** name=**"LT or AT simulation"** type=**"bool"**   
 **default**=**"true"** hint=**"True – LT abstraction, False – AT abstr."**/>

19 <option var=**"gdb"** name=**"Enable GDB support"** type=**"bool"**   
 **default**=**"true"** hint=**"Ture - Simulation models will include GDB   
 support. Just add ' gdb' to the end of the comandline to wait for GDB connection."**/>

20 <option var=**"timing"** name=**"Enable Timing Report"** type=**"bool"**   
 **default**=**"true"** hint=**"True - The simulation models will print a   
 timing report at the end of simulation."**/>

21 <option var=**"power"** name=**"Enable Global Power Report"** type=**"bool"**   
 **default**=**"true"** hint=**"True - Aggregate power data for all components. This may slow down the simulation and take up a lot of RAM and HDD space (Not recommended for default)."**/>

22 </option>

23 <option var=**"ahbctrl"** name=**"AHB Controller"**>

24 <option var=**"ioaddr"** name=**"AHB IO Area Address"** type=**"int"**   
 **default**=**"0xFFF"** range=**"0..0xFFF"**/>

25 <option var=**"iomask"** name=**"AHB IO Area Mask"** type=**"int"**   
 **default**=**"0xFFF"** range=**"0..0xFFF"**/>

…

42 <option var=**"mcheck"** name=**"Check the consistency of the memory map"**   
 type=**"bool"** **default**=**"true"**>

43 <description>

44 Check if there are any intersections between core memory areas.   
 In case of error issues a failure assertion.

45 </description>

46 </option>

47 </option>

…

62 <option var=**"mmu\_cache"** name=**"MMU Cache"**>

63 <option var=**"addr"** name=**"AHB Base Address"** type=**"int"** **default**=**"0"**

…  
 99 </option>

100 </option>

…

183 <option var=**"irqmp"** name=**"IRQ Controller"**>

…

190 </option>

191 <generator type=**"systemc"** name=**"config.h"**/>

192 <generator type=**"template"** path=**"systemc"** name=**"wscript"**   
 src=**"singlecore/scwscript.py"** />

193 <file type=**"systemc"** name=**"sc\_main.cpp"** src=**"singlecore/main.cpp"** />

194 <file type=**"systemc"** name=**"prom.S"** src=**"singlecore/prom.S"** />

195 <file type=**"systemc"** name=**"prom.ld"** src=**"singlecore/prom.ld"** />

196 <file name=**"wscript"** src=**"singlecore/wscript.py"** />

197 </template>

The outermost tag (lines 1 – 197) 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 – 17) is also solely used by 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 Monitoring. 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 17 – 22. The user can switch between LT and AT mode, enable GDB stubs and Power Monitoring. An example for IP Options is given in lines 23 – 47. 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 24 is of type integer, has a default of **0xfff** and may be modified to any value in the range of **0x000 – 0xfff**. As shown in lines 43 – 45 it is also possible to attach additional explanations **(<description>**).

**Build script generator –** **scwscript.py**

The build script generator is implemented in Python and ought to integrate the design in the build system of the library. It is used by the Configuration Wizard in order to generate WAF wscripts for each particular instance of the design. Depending on the names of template and configuration two WAF target are provided.

1. For generating the platform simulator:

**./waf --target=  
 {name\_of\_template}.{name\_of\_configuration}.platform**

2. For generating the boot-code:

**./waf --target=  
 {name of template}.{name of configuration}.prom**

Enabling this targets requires a configuration. Usually multiple configurations are produced for one template during design space exploration. How to create a configuration is described in section 2.2.3.

**Link script template – prom.ld**

The linker script template can be more or less complex depending on the planned memory layout and software architecture. The presented example targets a bare-metal design containing ROM and RAM regions. At system generation time the memory settings from the configuration are copied into the linker script. Mainly start and end addresses of memories, but also the location of system registers are replaced/filled in.

### Configuring the template

The purpose of a configuration are the initialization of the configuration parameters exported by the platform template, and the provision of architecture dependent compilation and simulation scripts. 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 introduced by 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 or has been 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 window (Figure 4). Click next to proceed to the license window (Figure 5). All code and utilities provided with the SoCRocket library are subject to the GNU General Public License (GPL). Accepting the license opens the template window (Figure 6). From the template window select the singlecore platform template as a starting point. The selection must be confirmed by pressing the next button.

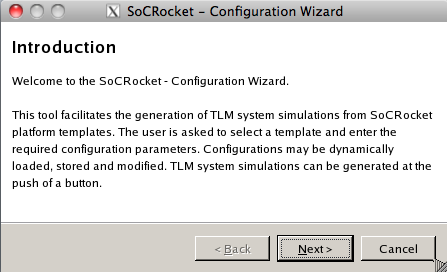


Figure - Configuration Wizard / Welcome screen

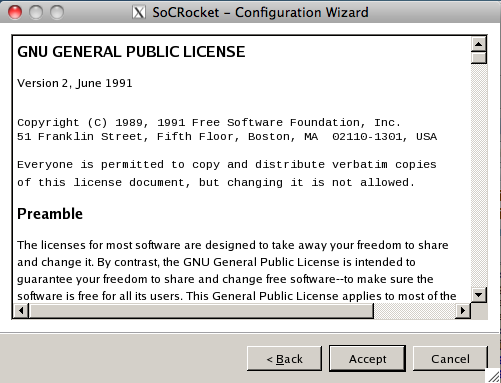


Figure - Configuration Wizard / License agreement

In the next step the user can choose between creating a new configuration for the selected template or loading a previously stored configuration. The configuration window (Figure 7) shows two preexisting configurations: **lt** and **at**. Both configurations instantiate the singlecore template using the default parameters provided by the template descriptor (tpa). The only difference is in the TLM abstraction level. The lt configuration initializes all models for loosely-timed simulation (software development), while the at configuration initializes approximately-timed simulation (architecture exploration).

For the FFT processor we are creating a new “default 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 brings up a description. Clicking in the value field of a parameter allows the parameter to be modified.

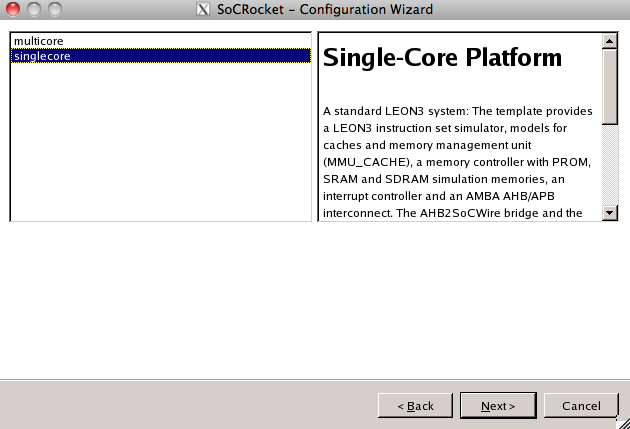


Figure - Configuration Wizard / Template window

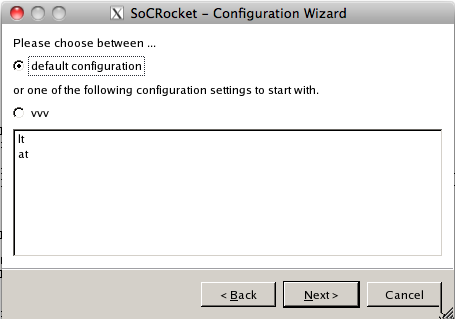


Figure - Configuration Wizard / Configuration window

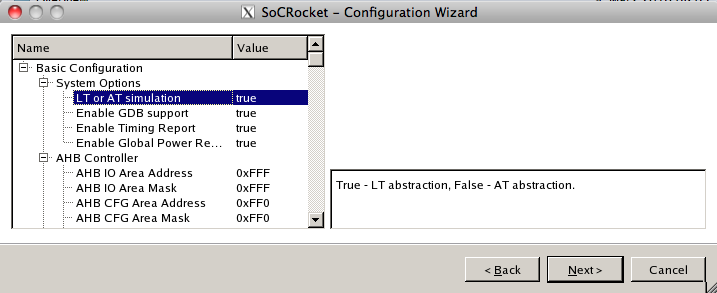


Figure – Configuration Wizard / Parameter window

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

1. *Select AT mode for architecture exploration: LT or AT simulation = false*
2. *Disable Power Monitoring: Enable Power Monitoring = false*
3. *Configure APB controller: base address = 0x800, address mask = 0xfff, bus index = 2*
4. *For initial system 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 SoCWire: SoCWire = true*
11. *Enable input sensor: Sensor = 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.

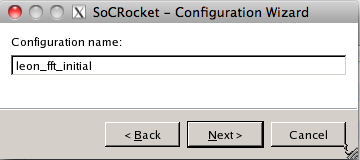


Figure - Configuration Wizard / Store configuration

The configuration data has been written to the **./templates** directory. The code example below shows an abstract of the **singlecore-leon\_fft\_initial.cfg** file:

1 {

2 **"Basic Configuration"**: {

3 **"AHB Controller"**: {

4 **"AHB CFG Area Address"**: 4080,

5 **"AHB CFG Area Mask"**: 4080,

6 **"AHB IO Area Address"**: 4095,

7 **"AHB IO Area Mask"**: 4095,

8 **"AHB IO area enable"**: true,

9 **"Default AHB Master"**: 0,

10 **"Enable full decoding of the PnP configuration records"**: true,

11 **"Enable support for fixed burst length."**: false,

12 **"Enable support for split transactions in AT models"**: false,

13 **"Intersection checking"**: true,

14 **"Round robin arbitration."**: false

15 },

... ...

133 **"System Options"**: {

134 **"Enable GDB support"**: true,

135 **"Enable Global Power Report"**: true,

136 **"Enable Timing Report"**: true,

137 **"LT or AT simulation"**: false

138 }

139 }

140 }

Lines 3 – 15 contain the settings for the AHBCTRL. The system option are shown in lines 133 – 138. Thanks to the human-readable format of the file (JSON language), the configuration can be easily modified using any text editor (without having to load into the Configuration Wizard).

### Generating platform simulation and software

Generating a configuration with the Configuration Wizard does not only generate the configuration file (cfg), but also the system itself. All files required for platform simulation are being automatically copied into the **./platforms/singlecore-leon\_fft\_initial** directory. The directory contains the following files:

**config.h** - Header with definitions of configuration parameters (to be included by platform simulation.

**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 bootcode itself can be located at any memory location. The processor simulator will start from the address specified in label* **\_start***.*

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

*The bootcode 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.*

**sc\_main.cpp** – Top-level of SystemC simulation

**wscript** – WAF build script

For generating a system simulation from an existing configuration, the Configuration Wizard can be used in terminal mode. The following command generates all files for template **singlecore** and configuration **leon\_fft\_initial**:

**./waf generate –t singlecore –l leon\_fft\_initial**

This method allows to rapidly create a large number of system simulations with alternative parameters covering a wide exploration space.

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

**./waf –target=singlecore.leon\_fft\_initial.prom**

For compiling the system simulation enter:

**./waf –target=singlecore.leon\_fft\_initial.platform**

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. Amongst others, the code demonstates how to catch an interupt from the sensor device 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**. Both memories must be initialized at begin of simulation. Therefore, platforms based on the **singlecore** template expect two ELF files as command line parameters.

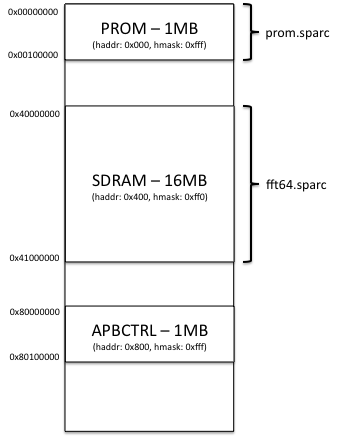


Figure - Memory map FFT processor

### Running/Debugging simulation

As previously mentined all simulations based on the **singlecore** template expect two SPARC executables as command line parameters. The first executable is loaded in the ROM, the second in the SDRAM. Optionally, a third parameter can be provided for starting the GDB server:

**singlecore.leon\_fft\_initial.platform** prom\_image ram\_image [gdb]

Example with full paths:

**./build/platforms/singlecore-leon\_fft\_initial** *\ /systemc/singlecore.leon\_fft\_initial.platform   
./build/platforms/singlecore-leon\_fft\_initial/ \  
systemc/singlecore.test.prom ./build/software/fft64/fft64\_input\_gen.sparc gdb*

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. Also 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.

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 inititialized using the SystemC **start\_of\_simulation** callback. All internal decoding tables are build 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 bootcode, 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. For 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. With the help of the statistics conclucions 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 analysis API of the library. Performance counters are implemented using the GS\_PARAM mechanism provided by the GreenSoCs package 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