**Title**

HW-SW SystemC Co-Simulation SoC Validation Platform

**IP User Manual   
&   
Development Document**

Contract No. 4200022968

by

TU Braunschweig, Germany

March 21, 2012

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

**Table of Contents**

1 Introduction 8

1.1 Purpose and Scope 8

1.2 Referenced Documents 8

1.3 Revisions 8

2 The socrocket library 10

2.1 Required Software Packages / Dependencies 10

2.1.1 GreenSocs 11

2.1.2 GreenSocs/Carbon AMBA Sockets 12

2.2 Installation 12

2.2.1 WAF Command Overview 12

2.2.2 Building the library 13

2.2.3 Models separate compilation 14

2.3 Library Structure 14

2.4 Configuration Wizard 16

2.4.1 Terminal Mode 16

2.4.2 GUI Mode 16

3 Modeling Concepts 18

3.1 Coding Style 18

3.1.1 Loosely Timed (LT) 18

3.1.2 Approximately Timed (AT) 18

3.2 Library base classes 19

3.3 TLM Signal Communication Kit 21

3.4 Memory mapped registers 23

3.5 Power Modeling 24

3.6 Extending the library 26

4 AHBCTRL SystemC Model 27

4.1 Functionality and Features 27

4.1.1 Overview 27

4.1.2 Address Decoding 27

4.1.3 Arbitration 27

4.1.4 Data Split 27

4.1.5 Plug & Play Support 27

4.1.6 Snooping 28

4.1.7 Power Monitoring 28

4.2 Interface 28

4.3 Internal Structure 29

4.3.1 Decoder initialization 29

4.3.2 LT behaviour 30

4.3.3 AT behaviour 30

4.4 Compilation 31

4.5 Example Instantiation 31

5 APBCTRL systemc model 33

5.1 Functionality and Features 33

5.1.1 Overview 33

5.1.2 Address Decoding 33

5.1.3 Plug & Play Support 33

5.1.4 Power Monitoring 33

5.2 Interface 33

5.3 Internal Structure 34

5.3.1 Decoder initialization 34

5.3.2 LT behaviour 34

5.3.3 AT behaviour 35

5.4 Compilation 36

5.5 Example Instantiation 36

6 MCTRL Memory Controller SystemC Model 38

6.1 Functionality and Features 38

6.1.1 Overview 38

6.1.2 Control Registers 38

6.1.3 Address Space 39

6.1.4 Local I/O Access 42

6.1.5 SRAM Access 43

6.1.6 SDRAM Access 43

6.1.7 SDRAM Modes of Operation 44

6.1.8 Power Modeling 45

6.2 Interface 46

6.3 Internal Structure 47

6.3.1 Decoder initialization 48

6.3.2 LT behaviour 48

6.3.3 AT behaviour 49

6.4 Compilation 50

6.5 Example Instantiation 50

7 GENERIC Memory SystemC Model 51

7.1 Functionality and Features 51

7.1.1 Overview 51

7.1.2 Power Modeling 51

7.2 Interface 51

7.3 Internal Structure 52

7.3.1 Interface MemDevice 52

7.3.2 Functional Memory 52

7.4 Compilation 52

7.5 Example Instantiation 53

8 MMU\_CACHE Cache sub-system systemC module 55

8.1 Functionality and Features 55

8.1.1 Overview 55

8.1.2 Address Space Identifiers (ASI) 56

8.1.3 System and Control Registers 57

8.1.4 Data Cache Snooping 59

8.1.5 Instruction burst fetch 59

8.1.6 Cache Flushing 60

8.1.7 Freezing and Locking 60

8.1.8 Diagnostic Access 60

8.1.9 Payload Extensions 60

8.1.10 Debug Mechanism 61

8.1.11 Power Monitoring 62

8.2 Interface 62

8.3 Internal Structure 64

8.3.1 Files of the mmu\_cache library 64

8.3.2 LT Behaviour 72

8.3.3 AT Behaviour 72

8.4 The AHB master 73

8.5 Compilation 75

8.6 Example Instantiation 75

9 GPTIMER General Purpose Timer SystemC model 77

9.1 Functionality and Features 77

9.1.1 Overview 77

9.1.2 Control Registers 77

9.1.3 Chaining 78

9.1.4 Watchdog 79

9.1.5 Power Modeling 79

9.2 Interface 79

9.3 Internal Structure 80

9.3.1 Configuration of the module 80

9.3.2 Operation of the module 80

9.4 Compilation 81

9.5 Example Instantiation 81

10 IRQMP Interrupt Controller SystemC model 81

10.1 Functionality and Features 81

10.1.1 Overview 81

10.1.2 Control Registers 82

10.1.3 Interrupt Prioritization and Forwarding 83

10.1.4 Extended Interrupt Handling 85

10.1.5 Processor Status Monitoring 85

10.1.6 Power Modeling 85

10.2 Interface 85

10.3 Internal Structure 86

10.3.1 The irqmp.h file 86

10.3.2 The irqmp.cpp file 87

10.4 Compilation 88

10.5 Example Instantiation 88

11 AHB2Socwire Systemc model 89

11.1 Functionality and Features 89

11.1.1 Overview 89

11.2 Interface 89

11.2.1 Clock, AHB, APB Device Interfaces 90

11.2.2 APB Slave Interface 90

11.3 Internal Structure 90

11.3.1 Interrupts 91

11.3.2 Device Attributes 92

11.3.3 AHB Master Interface 92

11.3.4 TX descriptor handling (LT) 92

11.3.5 RX descriptor handling (LT) 92

11.3.6 SoCWire Socket 93

11.4 Compilation 96

11.5 Example Instantiation 96

Annex a – Inconsistencies in the GRIP manual 98

A.1 – MCTRL Memory Controller 98

**Table of Figures**

Figure 1 - Configuration Wizard 16

Figure 2 – RAM address space 40

Figure 3 – Structure of the TLM MCTRL 47

Figure 4 - Structure of Cache Sub-System 55

Figure 5 - Generic Memory Interface / Dependencies 64

Figure 6 – Interrupt Distribution Scheme 82

Figure 7 – AHB2Socwire Structure 89

**Table of Tables**

Table 1 - Referenced Documents 8

Table 2 - Revisions of this document 9

Table 3 - Software Dependencies (\* optional) 10

Table 4 - AT Phase AHB Protocol 18

Table 5 - AHBCTRL Constructor Parameters 28

Table 6 - APBCTRL Constructor Parameters 33

Table 7 – MCTRL Registers 37

Table 8 - MCFG - Memory Configuration Register 1 (MCFG1) 37

Table 9 - MCTRL - Memory configuration register 2 (MCFG2) 38

Table 10 – MCTRL Constructor Parameters 46

Table 11 - Generic Memory Constructor Parameters 51

Table 12 - Supported ASIs 56

Table 13 - CACHE CONTROL REGISTER 56

Table 14 - ICACHE & DCACHE Configuration Register 57

Table 15 - MMU Control Registers 58

Table 16 - Debug Extension 60

Table 17 - Constructor Configuration Parameters 62

Table 18 - TLM Sockets Cache Sub-System 66

Table 19 - Page size / index combinations 69

Table 20 – GPTimer Registers 75

Table 21 - GPTimer Configuration Register 76

Table 22 - GPTimer Counter Configuration Register 76

Table 23 - GPTimer Parameters 78

Table 24 - Timer SignalKit sockets 78

Table 25 – IRQMP Registers 81

Table 26 - Template Parameters 84

Table 27 - IRQMP SignalKit sockets 84

Table 28 – AHB2Socwire Constructor Parameters 87

Table 29 – AHB2Socwire Registers 88

Table 30 – SoCWire Socket Constructor Parameters 92

Table 31 – SoCWire TLM Protocol Attributes and Extensions 93

# Introduction

## Purpose and Scope

This document is the joint IP User Manual (UM) and Development Document (DD) of the SystemC Co-Simulation SoC Validation Platform (SoCRocket) and its contained simulation models. It describes the interfaces and functions of all IPs from the perspectives of the system architect and the programmer.

## Referenced Documents

|  |  |  |
| --- | --- | --- |
| **Reference** | **Document Number** | **Document Title, Author** |
| RD01 | TEC-EDM/2008.27/BG | Statement of Work to ITT- AO/1-6025/09/NL/JK, ESA |
| RD02 | IDA-PPS-0309-2 | HW-SW Co-Simulation SystemC SoC Validation Platform – Technical Proposal |
| RD03 | IDA-PPS 0309-3 | HW-SW Co-Simulation SystemC SoC Validation Platform – Management Proposal |
| RD04 |  | GRLIB IP Core User’s Manual V1.0.22 |
| RD05 |  | GRLIB IP Library User’s Manual V1.1.0 |
| RD06 |  | GreenSocs AMBA LT/AT concepts |
| RD07 |  | GreenSocs AMBA TLM 2.0 Extensions |
| RD08 |  | SPARC V8 Reference Manual |
| RD09 | IDA-SCSV-IMS-001 | Interconnection Methodology Summary |
| RD10 | IDA-SCSV-DF-001 | Design Flow Report |
| RD11 | IDA-SCSV-PMR-001 | Power Modeling Report |

Table 1 - Referenced Documents

## Revisions

The following table will be updated during the development of the UM.

|  |  |  |
| --- | --- | --- |
| **Version** | **Date** | **Description** |
| 1.0 | 01/09/10 | Initial submission |
| 1.1 | 17/09/10 | Version prior to the MDR meeting |
| 1.2 | 03/05/11 | Update for compliance with library  (e.g. AMBAKit removed) |
| 1.3 | 17/01/12 | Update for compliance with library (new features for models and platform) |
| 1.4 | 14/02/12 | Updated with feedback from DFR meeting |

Table 2 - Revisions of this document

# The socrocket library

## Required Software Packages / Dependencies

The SoCRocket Library can be checked out from our SVN repository at following location:

<https://ntserv1.ida.ing.tu-bs.de/svn/hwswcosim/trunk>

To compile and simulate the comprised models, software and example platforms the following tools are required (Table 3):

| Tool / Lib | Version | Vendor | Installation Path Variables |
| --- | --- | --- | --- |
| Python | >2.4 | Python team | On **$PATH** |
| GCC (x86) | >4.1.0 | GCC team | On **$PATH** |
| GCC/BCC (Sparc) | >4.3.4 | GCC team | On **$PATH** |
| binutils | >2.19 | GNU team | On **$PATH** |
| Doxygen\* | >1.7.5.1 | Doxygen | On **$PATH** |
| GCOV/LCOV\* | >4.1.0 | GNU team | On **$PATH** |
| Boost | >1\_37\_0 | Boost team | **$BOOST\_DIR** - header path  **$BOOST\_LIB** - library path |
| SystemC | >=2.2.0 | OSCI | **$SYSTEMC\_HOME** – installation root |
| SCV |  | OSCI | **$SCV\_HOME** – installation root |
| TLM 2.0 | 2009-07-15 | OSCI | **$TLM2\_HOME** – installation root |
| GreenSocs | 4.2.0 | GreenSocs Ltd. | **$GREENSOCS\_HOME** – inst. root |
| AMBASockets | 1.0 | Carbon Design Systems Inc | **$AMBA\_HOME** – installation root |
| Modelsim\* | >6.0 | Mentor Graphics | On **$PATH** |
| GRLIB\* | 1.0.21 | Aeroflex Gaisler | **$GRLIB\_HOME** – installation root  **$GRLIB\_TECH** – Path to compiled demo design: /designs/leon3-gr-xc3s-1500/modelsim |

Table - Software Dependencies (\* optional)

Please make sure that all the software packages mentioned above are properly installed, before proceeding with building the library.

Compiling software for the LEON ISS requires a SPARC compiler. We recommend to use the GCC/BCC provided by Aeroflex Gaisler. It can be downloaded in different preconfigured packages depending on host system and software layout (e.g. bare-metal, rtems).

<http://www.gaisler.com/doc/libio/bcc.html>

The Mentor Modelsim simulator and the Aeroflex Gaisler GRLIB are required for SystemC/VHDL co-simulation. This feature can be optionally disabled (see 2.2.2).

Gcov/Lcov and Doxygen are also optional components. The build system will not check for them. If the packages are not present test coverage calculation and the generation of additional documentation are not possible.

For the setup of the GreenSocs Software and the Carbon AMBA Sockets some additional instructions are given below (2.1.1, 2.1.2). Those are only intended to complement the documentation of the tools not to replace them.

### GreenSocs

Before installing GreenSocs SytemC, TLM2.0 and BOOST should be available on the target machine.

*1. Download GreenSocs*

The actual GreenSocs software can be downloaded from the following location:

<http://www.greensocs.com/files/greensocs-4.2.0.tar.gz>

*2. Extract the tarball:*

$ tar -xvzf greensocs-4.2.0.tar.gz

*3. Set the following environment variables:*

$ export SYSTEMC\_HOME=<THE **ROOT** OF YOUR SYSTEMC INSTALLATION>

$ export TLM\_HOME=<THE **ROOT** OF YOUR TLM2.0 INSTALLATION>

$ export BOOST\_DIR=<THE **INCLUDE** DIR OF YOUR BOOST INSTALLATION>

$ export GREENSOCS\_HOME=<THE FOLDER YOU **EXTRACTED** GREENSOCS>

$ export GSGPSOCKET\_DIR=$GREENSOCS\_HOME/gsgpsocket

$ export GREENSOCKET\_DIR=$GREENSOCS\_HOME/greensocket

$ export GREENREG\_DIR=$GREENSOCS\_HOME/greenreg

$ export GREENCONTROL\_DIR=$GREENSOCS\_HOME

*4. Fix the GreenSocs Makefile*

Change to the greensocs-4.2.0/greenreg folder and apply a small change to Makefile.conf:

Comment out lines 2-4 (with #):

2 TOP := $(dir $(lastword $(MAKEFILE\_LIST)))

3

4 GREENREG\_DIR=$(TOP)

to:

2 **#**TOP := $(dir $(lastword $(MAKEFILE\_LIST)))

3 **#**

4 **#**GREENREG\_DIR=$(TOP)

This will make the GreenSocs build system compatible to a wider range of Make versions and might save some trouble.

*5. Compile the library:*

The biggest part of the GreenSocs software consists of C language header files, which do not need a compile. The only exception is GreenReg. Call make to compile everything that has to:

$ make

*6. Apply Patches:*

The contrib folder of the SoCRocket library contains two patches which need to be applied to GreenSocs.

greenreg-4.0.0-writemask.patch – Fixes a bug concerning correct setting of writemasks in GreenReg

greensocs-4.0.0.patch – Adds an additional socket variant to the system (Enables GreenReg registers to be bound to Carbon AMBA Sockets).

*GreenSocs is now ready to be used.*

### GreenSocs/Carbon AMBA Sockets

Based on the GreenSocket TLM2.0 sockets (installed with 2.1.1), GreenSocs Ltd. also developed a dedicated AMBA socket. The latter contains payload and protocol extensions for modeling AHB and APB transfer. The socket has been approved by ARM and can be downloaded from the website of Carbon Design Systems without charge (Carbon License):

<https://portal.carbondesignsystems.com/ALLIp.aspx?Category=Free%20Downloads>

After downloading extract the software to a location of your choice:

$ tar -xvzf AMBAKit-trunk.tar.gz

Like the biggest share of GreenSocs, the Carbon AMBA Sockets are header-only and do not need to be compiled. After exporting the location of the source files the software is ready to be used:

$ export AMBA\_HOME=<THE FOLDER YOU **EXTRACTED** THE AMBAKIT>

## Installation

The build system shipped with the SoCRocket library is written in **waf**. It requires at least Python 2.3 to run. The **waf** executable is located in the root directory of the library.

### WAF Command Overview

Do a **./waf –h** to get an overview of and help on all available commands and options.

waf [command] [options]

The following commands are supported:

build : executes the build

clean : cleans the project

configure: configures the project

coverage : calculates the test coverage

dist : makes a tarball for redistributing the sources

distcheck: checks if the project compiles (tarball from 'dist')

distclean: removes the build directory

install : installs the targets on the system

list : lists the targets to execute

step : executes tasks in a step-by-step fashion, for debugging

uninstall: removes the targets installed

update : updates the plugins from the \*waflib/extras\* directory

generate : opens the configuration wizard and/or generates platforms from templates and configurations

-t TEMPLATE, --template=TEMPLATE

Defines a template to generate a new platform

-l CONFIGURATION, --load=CONFIGURATION

Load given configuration of the template

### Building the library

Building the project requires following steps:

*1. Execute* **./waf configure** *to configure the build environment*

The configuration step succeeds in case all the required software packages are available. Otherwise, it fails and shows the broken dependency. If so, the install path variable must be corrected. It is also possible to specify the location of a missing package. Use **./waf –h** to see all the different options (e.g. --systemc, --tlm).

As mentioned in 2.1, SystemC/VHDL co-simulation can be optionally disabled, in order to be independent of commercial tools or, eventually, save compilation time (**./waf configure –nomodelsim**).

Another important switch controls the verbosity of the output that is directed to stdout during simulation (**--verbosity=1..5)**. Verbosity level 1 will only display error messages. Level 2 includes warnings that are issued during simulation. Level 3 prints execution statistics and analysis reports. The recommended setting (default) is Level 4. It additionally shows configuration reports and information about the progress of tests. They highest verbosity is bound to level 5. It e.g. displays a message for each state-change of a transaction, which tremendously slows down simulation and, therefore, is only recommended for debugging.

*2. Compile library and run unit tests*

Execute **./waf** to compile all targets. Optionally, the –jN flag can be used to define to maximum number of parallel threads. If co-simulation is configured make sure you have enough licenses to execute N instances of Modelsim.  
As an alternative, you may select a specific target (test or library) for compilation.   
A list of targets can be generated with **./waf list**. Selective compile is done using **./waf –targets=”comma separated list of targets”**.

After successful compilation the system automatically starts the respective unit test(s) and displays the result on the screen.

*3. Optional Doxygen and Gcov*

In the final step you may generate additional documentation using Doxygen (**./waf docs**) or perform a test coverage calculation using Gcov/Lcov (**./waf coverage**). The configuration step does not check the presens of these tools. In case they are not present, the commands have no effect.

### Models separate compilation

The models of the library can be compiled separately and independently of each other. The following minimum set of source files is required for successful compilation:

*./common  
./contrib  
./Doxyfile  
./models/<****selected\_model****>  
./models/utils  
./models/wscript  
./signalkit  
./waf  
./waftools  
./wscript*

## Library Structure

Next to the actual TLM simulation models, the library contains an extensive set of unit tests and support-functions for building and analyzing simulations. The top-level of the library is structured as follows:

**adapters**

This folder contains a set of SystemC/VHDL adapters/transactors, which are used for the unit tests of the library. For more interformation on transactors, please see the Interconnection Methodology Summary [RD9].

**common**

Contains utility classes and functions, which are jointly used by all models of the library.

This mainly comprises timing monitor, power monitor, endianess conversion and verbosity control. More detailed explanantions on the seperate files in this directory are given in 3.2.

**contrib**

Contains a set of patches for GreenSocs 4.2.0 and a newly develped TLM Sockets that enables GreenReg registers to be bound to Carbon AHB sockets.

**doc**

The documentation to this library.

**generator**

The implementation of the Configuration Wizard. The tool uses templates and configurations to generate platform instances. Example templates can be found in the template folder. The platform sources are written to ./platform. More detailed information is given in 2.4.

**models**

The central folder containing all simulation models and unit tests. Each model is located in a separate sub-directory.

**platforms**

The platforms directory is intitially empty. It will be filled with source code and build scripts for platform simulations by the Configuration Wizard (generator).

**signalkit**

Home of the SoCRocket SignalKit. The SignalKit comprises a set of functions/templates that allow signal communication in TLM-Style, without the overhead of maintaining payload objects. Within the library it is mainly used to model interupts and reset distribution. More detailed information can be found in 3.3.

**templates**

This directory contains example templates and configurations, which can be used with the SoCRocket Configuration Wizard (generator). User defined templates should also be stored in here.

***Doxyfile*** DoxyGen configuration file.

***waf*** WAF executable

***wscript*** Top-level build script

## Configuration Wizard

The SoCRocket configuration wizard can be used to generate and reconfigure system simulations. It may be used in terminal-mode or gui-mode. Both modes are integrated in the WAF build system.

### Terminal Mode

The following command creates a system simulation from template **singlecore** and configuration **lt**:

**./waf generate –t singlecore –l lt**

All input files required for this example are located in the ./templates directory. The sources of the platform simulation will be written to ./platforms/singlecore-lt.

Copying and renaming the **singlecore.lt.cfg** configuration file allows to quickly create a large number systems with various parameters.

### GUI Mode

The gui mode represents a more comfortable way of generating systems from templates and configurations. The Wizard can be started with following command:

**./waf generate**

It provides a dialog, which allows the user to select a template and a configuration, or create a new configuration for an existing template. All parameters of the template will be displayed in an input mask. After adjusting all the required settings, the tool behaves exactly as in terminal mode.



Figure - Configuration Wizard

More detailed information about the SoCRocket configuration wizard can be found in RD10.

# Modeling Concepts

In this chapter we describe the underlying modeling concepts of the library. This comprises coding style/abstraction, as well as common base classes and modeling techniques. The goal is to enable the user to extend the library by developing and integrating his own modules.

## Coding Style

The simulation models of the library are developed in SystemC language and build on the OSCI TLM2.0 standard. Like any TL model they abstract from cycle-timed accuracy by modeling communication in form of function calls. Depending on the use case this can be done in many different ways. The general aim is to save simulation time by sacrificing a certain amount of timing accuracy. Moreover, TL simulations usually give the user a bigger amount of leeway, compared to RTL.

Two major use cases are covered: software development and architecture exploration. Consequently, all IPs of the library support loosely timed (LT) and approximately timed (AT) abstraction. The abstraction layer is selected using constructor parameters.

### Loosely Timed (LT)

The LT configuration of the simulation models is intendend for fast address-accurate simulation (SW development). Communication is modeled using blocking function calls and as little synchronization with the SystemC kernel as possible. Respectively, independent of the protocol, each data transmission completes in a single call to the TLM transport interface. The master stalls/blocks during the call. All involved transfer or target components increment the delay counter, which is carried along the payload. After return of control from the slave, the master may or may not consume the annotated delay. To reduce context switching the master is allowed to run ahead of time and to accumulate the delay of multiple transactions. LT models are supposed to provide ‘just enough’ timing accuracy to allow an operating system to boot.

### Approximately Timed (AT)

The AT configuration of the simulation models is intendend for architecture exploration. To provide the therefore required accuracy it models selected features of the involved communication protocols. This mainly relates to the pipelined nature of AHB. In the AHB protocol the data phase of a request and the address phase of a succeeding request may overlap. Ignoring this fact in simulations, leads to large timing deviations. Nevertheless, the AT abstraction of the IPs in this library do not model the AHB protocol in a cycle-accurate way. The AT mode is supposed to provide a reasonable speedup over RTL simulation, while still being accurate enough to facilitate architectural decisions. This balancing act is approached by describing all kinds of AHB transfers using four phases (Table 4):

|  |  |  |
| --- | --- | --- |
|  | Read Operation | Write Operation |
| Begin of AHB address phase | tlm::BEGIN\_REQ (Master -> Slave) | tlm::BEGIN\_REQ (Master -> Slave) |
| End of AHB address phase (incl. arbitration) | tlm::END\_REQ (Slave -> Master) | tlm::END\_REQ (Slave -> Master) |
| Begin of AHB data phase | tlm::BEGIN\_RESP (Slave -> Master) | amba::BEGIN\_DATA (Master -> Slave) |
| End of AHB data phase | tlm::END\_REQ (Master->Slave) | amba::END\_DATA (Slave -> Master) |

Table - AT Phase AHB Protocol

More information about the AT abstraction can be found in the respective section of the model descriptions (e.g. 4.3.3, 6.3.3, 8.3.3) and in RD09.

## Library base classes

**Clock Device:**

class CLKDevice

models/utils/clkdevice.\*

lib utils

The class **CLKDevice** is used to consistently distribute clock/timing and reset amongst all IPs of the library. Devices that inherit from **CLKDevice** receive two SignalKit inputs: **clk** and **rst**. If the child requires reset behavior, it may implement the virtual function **dorst()**, which is triggered by the **rst** input. Moreover, **CLKDevice** provides a data member **„clock\_cycle“**, which can be used by the child to determine the clock period for delay calculations. The value of **clock\_cycle** is set by connecting a **sc\_time** SignalKit signal to the **clk** input or by calling one of the various **set\_clk** functions of the class.

**AHB Device:**

class AHBDevice

models/utils/ahbdevice.\*

lib utils

All simulation models that are supposed to be connected to the TLM AHBCTRL must be derived from class AHBDevice. The Aeroflex Gaisler AHBCTRL implements a Plug & Play mechanism, which relies on configuration information that is collected from the attached masters and slaves. AHBDevice models the respective configuration data records. The structure of these records is described in RD04. At start\_of\_simulation the TLM AHBCTRL iterates through all connected modules to retrieve AHB bar & mask and build up its internal routing table.

**APB Device:**

class APBDevice

models/utils/apbdevice.\*

lib utils

All simulation models that are supposed to be connected to the TLM APBCTRL must be derived from class APBDevice. Similar to the concept of AHBDevice, the child inherits Plug & Play configuration records representing its device type and address. At start\_of\_simulation the APBCTRL iterates through the connected slaves collecting all APB bar & mask settings for building up its routing table.

Modules, like the MCTRL, which posses an AHB as well as an APB interface must be derived from AHBDevice and APBDevice.

**Memory Device:**

class MEMDevice

models/utils/memdevice.\*

lib utils

The class MEMDevice is the base class of all memories to be connected to the MCTRL. The library provides a Generic Memory, which implements the given interface. The included functions are required to determine the features of the attached component for correct access and delay calculation.

**Timing Monitor:**

class timingmonitor

common/timingmonitor.\*

lib common

Timingmonitor is a support class for timing verification. Within the library it is used in almost all testbench classes. During simulation it records the SystemC simulation time and the real execution time of test phases. For this purpose it provides a set of static control functions. A test phase starts with a call to **phase\_start\_timing**. The function expects a phase ID and a phase description as inputs. This will create a new entry in the internal timing map. After completion of the test phase, the testbench calls **phase\_end\_timing** to close the record. At the end of the test, the testbench may now call **report\_timing** to generate a report showing the timing of all test phases. This is especially useful for comparing simulations at different levels of abstraction.

**Verbosity:**

class color, number, msgstream

common/verbose.\*

The operators defined in **verbose.h** can be used to filter output messages respecting their severity. As explained in 2.2.2 the verbosity level of the simulations must be defined during configuration of the library (**./waf configure –verbosity=1..5**). Five levels may be chosen: error, warning, report, info and debug. The operators are used in a similar way to C++ stdout:

**std::cout << value << std::endl; // Regular C++ stdout**

**v::error << value << v::endl; // Verbosity error stream**

**v::warn << value << v::endl; // Verbosity warn stream**

**v::report << value << v::endl; // Verbosity report stream**

**v::info << value << v::endl; // Verbosity info stream**

**v::debug << value << v::endl; // Verbosity debug stream**

Defining the verbosity at configuration time has the advantage that undesired output is optimized way (compared to runtime switching).

**Endianess:**

class none

common/vendian.h

lib common

The header **vendian.h** provides endianess conversion functions for data types of different lengths. If the host system is little endian, CPU and unit tests must swap byte order. The latter is defined by the macro **LITTLE\_ENDIAN\_BO**.

It has to be kept in mind that the LEON processor is a big endian CPU. Hence, memory images generated with the SPARC compiler (e.g. BCC) are also big endian. If the host system simulates the CPU, or testbench, in little endian byte order, all data items going to/from memory must turn!

## TLM Signal Communication Kit

Signal communication in TLM platforms is usually modeled using SystemC signals (**sc\_signals**). SystemC signals are applied very similar to RTL signals and, more-or-less, represent hardware wires. To achieve the required level of accuracy, all reads and writes of **sc\_signals** need to be scheduled by the SystemC kernel. For modeling at a higher level of abstraction this involves an unwanted overhead. One would prefer a fast function-call based (TLM-style) communication with a preference of retaining the natural, close to hardware, modeling style of **sc\_signals**.

For this purpose this library provides an extra set of functions. The SoCRocket SignalKit can be found in the root directory of the project (./signalkit). Within the library it is mainly used to model the interupt and reset distribution, but also for special purposes like dbus snooping. Syntax and application of SignalKit ports are very close to sc\_signals. Though, signal transmission is performed by directed function calls, similar to TLM blocking transport. However, in contrast to TLM no payload handling is required. The general handling is very simple.

A module that is supposed to utilize SignalKit signals must include the **signalkit.h** header file and must call the **SK\_HAS\_SIGNALS** macro in its class definition. The following code example shows a SignalKit module with an outgoing port of type **int**:

1 #include **"signalkit.h"**

2

3 **class** source : **public** sc\_module {

4

5 SK\_HAS\_SIGNALS(source);

6 SC\_HAS\_PROCESS(source);

7

8 signal<**int**>::out out;

9

10 // Constructor

11 source(sc\_module\_name nm) : sc\_module(nm), out(**"out"**) {

12

13 SC\_THREAD(run);

14

15 }

16

17 **void** run() {

18

19 // ...

20 out = i;

21 // ...

22 }

23 }

24

The actual signal output is defined in line 8. The output is written in line 20. Alternatively to to the shown direct data assignment, the write method of the port may be used (**out.write(i)**).

The next code block shows a signal receiver:

1 #include **"signalkit.h"**

2

3 **class** dest : **public** sc\_module {

4

5 SK\_HAS\_SIGNALS(dest);

6

7 signal<**int**>::in in;

8

9 // Constructor

10 dest(sc\_module\_name nm) : sc\_module(nm), in(&dest::onsignal, **"in"**) {

11

12 }

13

14 // Signal handler for input in

15 **void** onsignal(**const** **int** &value, **const** sc\_time &time) {

16

17 // do something

18

19 }

20

21 }

22

In line 10 the handler function **onsignal** is registered at SignalKit input **in**. If any call is received, this function will be triggered. Int **value** represents the data transmitted.

Sender and receiver can be connected using the SignalKit **connect** method. An example is given below:

1 #include **"source.h"**

2 #include **"dest.h"**

3

4 **int** sc\_main(**int** argc, **char** \*argv[]) {

5

6 source src;

7 dest dst;

8

9 connect(src.out, dst.in);

10

11 ...

12

13 **return** 0;

14

15 }

Next to that trivial direct connection, the **connect** method is capable of handling broadcasting and muxing, and for converting between Signalkit and SystemC signals. For a broadcast the **out** signal may be directly connected to multiple **in**s. In the mux case, multiple transmitters are combined into one receiver. If required the transmitter may be identified by a channel number.

## Memory mapped registers

GreenReg is used to model memory mapped registers throughout the library. Since almost every model requires a set of control registers, this unified scheme yields a high productivity gain. The following steps are required to define a register:

1. Include the GreenReg AMBA Socket header file.

**#include greenreg\_ambasockets.h**

2. Derive your class from GreenReg device

**class foo : public gs::reg::gr\_device**

3. Tell the system that your registers will require callback function (using the build-in macro).

**GC\_HAS\_CALLBACKS()**

4. Create the socket the register is going to be connected to.

**gs::reg::greenreg\_socket<gs::amba::amba\_slave<32> > my\_sock;**

5. In the constructor of the module – initialize gr\_device and socket.

**// This will create a register bank of size bank\_size bytes**

**gr\_device(name, gs::reg::ALIGNED\_ADDRESS, bank\_size, NULL)**

6. The initialization of gr\_device (5) delivers a default pointer (**r**) to the newly generated memory bank. The initialization of the socket requires this pointer along the address settings for the bank as input arguments.

**// Initialize socket and hook up register bank r**

**my\_sock(„sock“, r, start address, end address, protocol (e.g. amba::amba\_APB), abstraction (e.g. amba::amba\_LT), false)**

7. Create a register within the new memory bank.

**// New register in bank r at bank address + offset**

**r.create\_register(name, description, offset, type (e.g. gs::reg::STANDARD\_REG), initial value, write mask, bit width, lock mask (not used))**

8. Register a handler function for the register and make the handler sensitive to a register event.

**GR\_FUNCTION(foo, my\_handler);**

**GR\_SENSITIVE(r[offset].add\_rule(gs::reg::POST\_WRITE, „scaler\_write“, gs::reg::NOTIFY))**

9. In this example the handler will be called after completion of a write operation (POST\_WRITE). The signature of the handler function is void:

**void my\_handler()**

10. A module that uses GreenReg registers needs to call following macro in the destructor:

**GC\_UNREGISTER\_CALLBACKS**

## Power Modeling

The SoCRocket power monitor (PM) provides a database and a set of functions for tracing simulation events, which are relevant for the power consumption of the system. The PM is defined in the files **power\_monitor.h/cpp**, which are located in library **common**.

The PM is implemented as a static C++ class, which provides an API for all simulation models in the library. Most SoCRocket simulation models already implement this API. In that case, power monitoring can be enabled by setting the **pow\_mon** constructor parameter.

IPs that support power monitoring include the **power\_monitor.h** file. They register with the PM using an unique name. This is done in the constructor of the top-level class:

**PM::registerIP(this, “mctrl”, pow\_mon);**

Next to the name of the device, **registerIP** receives a module-pointer and a boolean switch as input arguments. The pointer is used to identify the module in the class hierarchy of the simulation. This also allows to monitor multiple instances of the same device. If the boolean (**pow\_mon**) is false, the body of the function is removed at compile time. If the model has multiple sub-components with relevant contribution to the overall power consumption, each of them can be registered as a separate IP. The MMU\_CACHE IP, for instance, registers their caches, localrams and mmu depending on the configurations.

The overall power consumption of a device consists of a static and a dynamic portion. The static power (leakage) is transmitted using the **send\_idle** function:

**PM::send\_idle(this, “idle”, sc\_time\_stamp(), pow\_mon);**

In the most trivial case the static power of a component is constant during the complete simulation time. This accounts for models such as the GPTimer or the IRQMP. Other IPs (e.g. MCTRL) provide multiple modes of operation, which have direct impact on the static power consumption. Therefore, the **send\_idle** function receives an event name (power mode) and a time stamp as input parameters. For each call to **send\_idle** both parameters are entered in the internal data base. In case **pow\_mon** is disabled (false), the body of the function will be removed by the compiler.

Dynamic power occurs if the internal state of a device changes. This happens for all sorts of events. However, the major share of the dynamic power is usually contributed by embedded register banks and memories. The costs for reading and writing these devices is known from the respective datasheets. Dynamic power events can be switched on and off using the **send** function. The example relates to a read operation from a cache set:

**PM::send(this, “set\_read0”, sc\_time\_stamp(), 0, pow\_mon);**

Similar to static power modes each dynamic power event is identified by a unique name. Other input parameters are the current timestamp, the on/off identifier and the enable switch (**pow\_mon**). It is important that every dynamic event that is switched on is also switched off again. The timestamp of the ‘switch-off’ operation must be higher/later then the timestamp of the ‘switch-on’ operation. In case of a mismatch the **analyze** function of the PM will issue a warning.

At simulation runtime the PM simply records all incoming power events. In order to minimize the impact on the simulation performance all analysis work is done offline. The analysis engine verifies the recorded raw data by running various consistency checks. It also sorts all power data with respect to their timestamp and hierarchically accumulates the information. In the last step all power events are annotated using technology dependent power information. An example for a generic power-tech file is provided with **./models/main\_power.dat**. The structure of this file is self-explanatory. It is grouped in multiple sections describing the IPs of the library. Each IP is identified by a name and provides a list of power events – value pairs:

**!IP!**

**name**

**idle power value  
action 1 power value  
action 2 power value  
action 3 power value**

**!ENDIP!**

The power analyzer can be called from the top-level of the simulation, right after **sc\_stop():**

**// techpath – Path to technology file  
// techfile – The technology specific power file  
// outfolder - Name of the output folder (for results)**

**PM::analyze(string techpath, string techfile, string outfolder);**

Alternatively, the power analyzer can be started in stand alone mode:

**// rawfolder - Path to raw-data  
// rawfile - Name of the raw-data file**

**PM::analyze\_offline(string techpath, string techfile,   
string outfolder, string rawfolder, string rawfile);**

In case the specified **techfile** can not be found the PM checks the POWERMONITORDAT environment variable. If this also fails, the PM generates an error message.

The output of the power analyzer is stored to the **logfiles** sub-directory of the path specified in **outfolder**. The sub-directory plotfiles contains data files and scripts for post-processing with GNUPLOT. To create plots for the power consumption of a certain IP or group of IPs use the following command:

**./plot outfolder/main.IP.gnu**

Depending on the extend and length of a simulation, recording power information can generate huge data files. Respectively, the computation effort for the analyze function may become very high. To reduce computation time it is recommended to limit the observation window of the PM to a meaningful timespan. This can be done using the **limit\_region** function:

**PM::limit\_region(sc\_time start, sc\_time end)**

An example for using the SoCRocket power tools is given in RD11, along additional explanations.

## Extending the library

The SoCRocket library can be easily extended by creating own components. The existing simulation models provide examples for almost all possible combinations of bus interfaces:

|  |  |
| --- | --- |
| **Model** | **Bus interfaces** |
| leon iss | CPU instruction master, CPU data master, Interrupt slave |
| mmu\_cache | AHB master, CPU instruction slave, CPU data slave, Interrupt master, Snooping input |
| ahbctrl | AHB master (multi-socket), AHB slave (multi-socket), Snooping output |
| apbctrl | AHB slave, APB master |
| ahbmem | AHB slave |
| mctrl | AHB slave, APB slave |
| socwire | AHB master, Interrupt master |
| gptimer | APB slave, Interrupt master |
| irqmp | APB slave, Interrupt master (multi-socket), Interrupt slave (multi-socket) |

Table - Overview models/interfaces

The following section illustrates the best practice for creating a new model on the example of an input generator.

The input generator is a component with an AHB master interface and a SignalKit interrupt output. The AHB master interface is used to transfer the data into main memory. After the transmission an interrupt is triggered to notify the CPU.

The input device can be seen as a sensor delivering frames of data at regular intervals. The model is utilized in the example accompanying the Design Flow Report [RD10].

Candidates to be used as a template are the MMU\_CACHE and the SoCWire IP ().

**1. Copy template model**

For each new model it is recommended to create a folder in the **./models** or **./models/extern** directory and, if possible, to copy the core files of an existing model with similar specification (especially bus interfaces).

For the input generator, we are going to start from the top-level files of the **mmu\_cache.**

**mkdir ./models/input\_generator  
  
cp ./models/mmu\_cache/lib/mmu\_cache.h ./models/input\_generator/input\_generator.h**

**cp ./models/mmu\_cache/lib/mmu\_cache.cpp ./models/input\_generator/input\_generator.cpp**

**cp ./models/mmu\_cache/wscript ./models/input\_generator/wscript**

**2. Prune the template**

In the next step all code concerning the behaviour of the template model should be removed. During this procedure all namespace and class name related expressions should be adapted.

For the input\_generator this means:

* Remove CPU instruction and data slave sockets, including transport functions, event queues and protocol threads.
* Remove snooping input and callback
* Remove sub-components and functions containing behaviour of the template component
* Change class name, change namespace, clear constructor. The new constructor should at least contain following parameters:
  + name – The SystemC component name
  + id – The master ID of the AHB interface
  + powmon – Flag for enabling/disabling power monitoring
  + ambaLayer – Abstraction layer of model (LT/AT)
* Change names and dependencies in the copied build script (**wscript**).
* Optionally/Recommended:
  + The input generator is not interested in the cacheability of memory regions. Hence, remove the cacheability check from the **mem\_read** function.
  + Make the **mem\_read** and **mem\_write** functions return void and non-virtual and remove the inheritance from **mmu\_cache\_if**.

*Copies of the pruned files can be found in the ./models/input\_generator directory* (**input\_generator\_pruned.h**/**input\_generator\_pruned.cpp**).

**3. Fill in new behaviour**

The planned behaviour of the input generator can be described as follows:

* Generate frames of random data of size X in intervals of Y ns.
* Transfer the data frame to a reserved memory region starting at address Z.
* Notify the CPU by sending an interrupt

To implement this behaviour three additional constructor parameters are added to the input\_generator device: **framesize** (X), **frameaddr** (Y), **interval** (Z).

The data frame is generated in a new SC\_THREAD (**gen\_frame**), which is activated in intervals of Z by SC\_THREAD (**frame\_trigger**). The notification is done by sending the SC\_EVENT **new\_frame**. A integer pointer is used to address the frame buffer **(\*frame**). The memory for the frame buffer is dynamically allocated in the constructor.

The code of function **gen\_frame** is shown below:

1 // Generate and transmit a frame of random data

2 **void** input\_device::gen\_frame() {

3

4 // Locals

5 **uint32\_t** tmp;

6 **uint32\_t** debug;

7 sc\_core::sc\_time delay = SC\_ZERO\_TIME;

8

9 // Wait for system becoming ready

10 wait(1, SC\_MS);

11

12 **while**(1) {

13

14 v::info << name() << **"Start sending new data frame!"** << v::endl;

15

16 // Generate data frame and send to memory

17 **for** (**uint32\_t** i=0;i<m\_framesize;i++) {

18

19 // Random number

20 tmp = rand();

21

22 // AHB write

23 mem\_write(m\_frameaddr + (i<<2), 0x8, (**unsigned** **char**\*)&tmp, 4, &delay, &debug, false);

24

25 }

26

27 v::info << name() << **"Transmission of frame completed"** << v::endl;

28

29 // Notify CPU by sending interrupt

30 irq.write(true);

31 wait(clock\_cycle);

32 irq.write(false);

33

34 // Wait for next frame trigger

35 wait();

36

37 }

38 }

The loop spanning lines 17 to 25 is responsible for generating the data. It also calls the **mem\_write** function, which will take care of the data being transferred to main memory. After completing the transmission an IRQ will be sent to the processor (lines 30 – 32).

The example is intentionally kept very simple. A natural extension would be an APB slave interface, for making the device software programmable and exposing status information.

**4. Compiling/Using the model**

The new model can be compiled using following command:

**./waf –target=input\_device**

To utilize the model in simulations add **input\_device** to the ‘use’ line of your **wscript**.

All code related to this example can be found in the **./models/input\_device** directory.

# AHBCTRL SystemC Model

## Functionality and Features

### Overview

The AHBCTRL TLM model can be used to simulate behavior and timing of the GRLIB AHB Controller VHDL IP. The model is available at two levels of abstractions (LT and AT). For modeling the AHBCTRL we mostly follow the recommendations given in RD06.

### Address Decoding

For address decoding the TLM AHBCTRL uses the same arithmetic as the GRLIB VHDL model. Each slave in the system provides a configuration record identifying its address range. This is done using two parameters: **haddr** and **hmask**. The **haddr** parameter represents the 12bit MSB base address of the device. The **hmask** parameter indicates the size of the address range. If **addr** is the 12 bit MSB address of a transaction following logic equation must be solved:

**select = (addr ^ haddr) & hmask**

Address **addr** falls in the address range of the slave if **select** equals zero.

### Arbitration

At AT abstraction the AHBCTRL supports two modes of arbitration: round robin and priority based. Arbitration mode can be selected by setting the **rrobin** constructor parameter. In fixed priority mode (**rrobin =** **0**), the bus request priority is equal to the masters’s bus index. The lower the index, the higher the priority. In round robin mode, priority is rotated one step after each AHB transfer. This is implemented in form of a modulo counter, which can be found in function **arbitrate\_me**.

### Data Split

According to the AHB protocol a slave may send a DATA SPLIT signal after receiving the address of a transfer. This usually happens, if the slave expects to need a long time before being able to provide the requested data.

In the SoCRocket library DATA SPLIT is modeled as an ignorable phase. Slaves can decide to send the phase **amba::DATA\_SPLIT** on the backward path, after having received **BEGIN\_REQ**. The bus can now decide to rearbitrate or to ignore the split. The current version of the AHBCTRL always ignores **DATA\_SPLIT**. After **DATA\_SPLIT** the slave must send **END\_REQ** to continue the transfer.

### Plug & Play Support

The TLM AHBCTRL supports the Plug & Play (PNP) mechanism described in RD04. AHB configuration records and access functions are implemented in class **AHBDevice**. Each master and slave to be connected to the bus model must be derived from this class. The PNP information of the slaves is collected at **start\_of\_simulation** (4.3.1). The combined information is mapped to the address range defined by the constructor parameters **cfgaddr** and **cfgmask**. By default, this relates to addresses 0xfffff000 – 0xffffffff. The master information is placed in the first 2kb block and the slave information in the second 2kb block of the device. For internal use all master information is aligned in integer array **mMasters**, while slave information can be found in **mSlaves**.

### Snooping

The TLM AHBCTRL supports dbus snooping. Address, length and master id of any write access will be broadcasted through the SignalKit output **snoop**.

In LT mode this is done in the blocking transport function (**b\_transport**), which is registered at socket **ahbIN**. The AT mode implements snooping within the SC\_THREAD **DataThread**. The **DataThread** is triggered by the non-blocking transport forward function (**nb\_transport\_fw**) on reception of phase **BEGIN\_DATA**.

The snooping information can be broadcasted to all relevant bus masters, by binding the **snoop** output of the **ahbctrl** to multiple snooping inputs (like multi-socket). The example below shows how to connect the snooping signal to four instances of type **mmu\_cache**:

1 // Connect snooping (broadcast)   
2 ahbctrl.snoop(cache0.snoop);   
3 ahbctrl.snoop(cache1.snoop);

4 ahbctrl.snoop(cache2.snoop);

5 ahbctrl.snoop(cache3.snoop);

6

### Power Monitoring

Power monitoring can be enabled by setting constructor parameter **pow\_mon**. The IP registers with the power monitor under the name **ahbctrl**. In the current version of the library two events are traced:

*idle* – Active during the completed runtime of the simulation (represents static power).

*ahb\_trans* – Indicates that a transfer is in progress (dynamic power).

## Interface

The GRLIB VHDL model of the AHBCTRL is configured using Generics. For the implementation of the TLM model most of these Generics were refactored to constructor parameters of class **ahbctrl**. An overview about the available parameters is given in Table 5.

|  |  |
| --- | --- |
| Parameter | Description |
| nm | SystemC name |
| ioaddr | The 12bit MSB address of the AHB I/O area |
| iomask | The 12bit address mask of the AHB I/O area |
| cfgaddr | The 12bit MSB address of the AHB configuration area (PNP) |
| cfgmask | The 12bit address mask of the AHB configuration area (PNP) |
| rrobin | Arbitration mode: 1 – round robin, 0 – priorities (AT only) |
| split | Enables AHB SPLIT response (AT only) |
| defmast | ID of the default master |
| ioen | Enable AHB I/O area |
| fixbrst | Enable support for fixed-length bursts |
| fpnpen | Enable full decoding of PNP configuration records |
| mcheck | Check if there are any intersections between core memory regions. |
| pow\_mon | Enable power monitoring |
| ambaLayer | Coding style/abstraction of model (LT or AT) |

Table - AHBCTRL Constructor Parameters

The system-level interface of the TLM AHBCTRL comprises an AHB master (**ahbOUT**) and an AHB slave socket (**ahbIN**). Both of them enable the connection to multiple masters and slaves (multi-sockets). Depending on the constructor parameter **ambaLayer** the sockets are configured for blocking (LT) or non-blocking (AT) communication. In the LT case the module registers a TLM blocking transport function at **ahbIN**. For the AT abstraction the model provides a TLM non-blocking forward transport function for the **ahbIN** socket and a TLM non-blocking backward transport function for the **ahbOUT** socket. In any case the model registers a TLM debug transport function. Within the current release of the library, debug transport is mainly used for non-intrusive code execution from the LEON-ISS. The signatures of all transport functions are compliant with the TLM2.0 standard.

Next to the TLM sockets the model comes with SignalKit inputs for clock cycle time (**clk**) and reset (**rst**), as well as an SignalKit output for snooping (**snoop**). The **clk** and **rst** inputs are inherited from class **CLKDevice**, while **snoop** is directly defined in **ahbctrl**.

## Internal Structure

This section describes the internal structure of the AHBCTRL. The class hierarchy of the model is flat. All functionality is comprised in class **AHBCtrl**, which is described in the files **ahbctrl.h** and **ahbctrl.cpp.**

### Decoder initialization

The address decoder of the TLM AHBCTRL is based on a routing table implemented as a **std::map**. The **std::map** **slave\_map** contains the index and address information of all the slaves connected to the AHBCTRL. It is initialized in function **start\_of\_simulation**. The function iterates through all slaves bound to socket **ahbOUT**. If the slave is a valid AHB Device (must be derived from class AHBDevice) the module creates one address entry in **slave\_map** per base address register (BAR). There can be at most four sub-devices/BARs for any slave. If the constructor parameter **fpenen** is enabled, the **start\_of\_simulation** function also copies the PNP information of any connected module (masters and slaves) into two 32bit wide arrays (**mSlaves** / **mMasters**). These arrays are mapped into the configuration area of the AHBCTRL (as described in RD04), where they can be accessed by any bus master.

### LT behaviour

In LT mode the AHBCTRL is a simple address decoder. All incoming transactions will be directly forwarded to their targets, without any arbitration done. The decoder is located in the **b\_transport** function. Transactions may be directed to the internal configuration area (PNP) or to one of the connected slaves. The configuration area is read-only. For access to the slave memory range, **b\_transport** calls **get\_index**. The **get\_index** function receives the address of the transaction as an input argument and returns the id of the slave binding (**index**). For this reason **get\_index** iterates through the previously described **slave\_map**. In case no slave can be found the function returns -1. This produces a TLM\_ADDRESS\_ERROR\_RESPONSE and an error message will be written to stdout. In case of success, the transaction is send to the identified slave by calling its **b\_transport** function:

**ahbOUT[index]->b\_transport(trans, delay);**

The LT AHBCTRL adds one cycle of delay to the transaction in order to approximate the delay of the AHB address phase. The delay may be consumed by the slave device or added to the latency of the target. The LT AHBCTRL does not synchronize with the SystemC kernel. The transaction delay is returned to the master, who is responsible for consuming the passed time.

### AT behaviour

The AT mode is intended to more accurately approximate the timing of the GRLIB AHBCTRL hardware model. To facilitate architecture exploration features like arbitration and pipelining are taken into account. Therefore, the AT mode of the AHBCTRL is more complex. It e.g. requires multiple parallel SC\_THREADs. The operation of the module can be best understood by following the control flow of a transaction.

A new transaction arrives in **nb\_transport\_fw** with phase BEGIN\_REQ. The function will first create a new connection record. A connection record consists of the **master\_id** (bus id of master), the **slave\_id** (bus id of slave) and a connection **state**. While the **master\_id** is known, the **slave\_id** still needs to be determined during decode. Hence, at this point in time, **slave\_id** is set to zero. The initial connection state is PENDING. The AHBCTRL keeps track of all transactions using the data structure **pending\_map**. New entries are created by function **addPendingTransaction**.

In the next step the thread **arbitrate\_me** decides which master will receive the bus in the current cycle. This will be done at intervals of **clock\_cycle** ns. The default **clock\_cycle** time is 10 ns. This setting can be overwritten by connecting a clock to input **clk** or by one of the **set\_clk** functions of class CLKDevice. Depending on constructor parameter **rrobin** the transaction with the highest priority (lowest index) or the one pointed by the **robin** counter is selected. All other transactions have to wait. If there is a winner, the respective transaction is entered in the **mRequestPEQ** payload event queue. Their transaction state is set to BUSY.

Now the transaction is ready for address decoding. This is done in thread **RequestThread**. The same mechanisms are used as for LT operation (**get\_index**). The connection record is updated with the index of the slave device. If the transaction is not directed towards the configuration area and a valid slave could be found, it is forwarded to socket **ahbOUT**:

**status = ahbOUT[index]->nb\_transport\_fw(\*trans, pase, delay)**

The slave may now respond in multiple different ways. The moduls of this library either return TLM\_UPDATED with phase END\_REQ or TLM\_ACCEPTED with phase BEGIN\_REQ. In the first case the **RequestThread** sends END\_REQ to the master. In the second it waits for event **mEndRequest**, which will be triggered as soon **nb\_transport\_bw** receives END\_REQ from the slave. This completes the address phase of the protocol.

In case of read transaction the slave is expected to continue by sending BEGIN\_RESP. If BEGIN\_RESP is received by **nb\_transport\_bw**, the transaction unblocks the ResponseThread via the **mResponsePEQ** payload event queue. The ResponseThread uses the pending\_map to find back the respective connection record including the index of the master. Afterwards, BEGIN\_RESP is send to the master. The master can now copy the data and reply with either TLM\_ACCEPTED and BEGIN\_RESP, TLM\_UPDATE and END\_RESP or TLM\_COMPLETED. In the first case the thread will wait for END\_RESP to be send on the forward path. This is indicated by event **mEndResponseEvent**. In all other cases the transaction is considered completed and removed from the **pending\_map**.

For more information on the AHB AT implementation please see RD09.

## Compilation

For the compilation of the AHBCTRL unit, a WAF wscript file is provided and integrated in the superordinate build mechanism of the library.

All required objects for simulating the AHBCTRL on platform level are compiled in a sub-library named **ahbctrl** using following build command:

**./waf –target=ahbctrl**

To utilize **ahbctrl** in simulations with other components, add **ahbctrl** to the use list of your wscript.

## Example Instantiation

The example below demonstrates the instantiation of the AHBCTRL inside an **sc\_main** method or an arbitrary top-level class. The instantiating module needs to include at least **ahbctrl.h** and **amba.h**. The AHBCTRL is created in line 19-32. In line 40 the slave port (**ahbIN**) of the bus is bound to a testbench master. Line 43 shows how to bind a slave to the master socket (**ahbOUT**). Both, bus master and slave socket support multiple bindings.

All additional components are to be connected in equal way. How to bind the snoop Signalkit output is shown in line 46. Since the AHBCTRL has some internal storage (config area), it needs a notion of time. In this example the clock cycle time is set in line 49. For the **set\_clk** function multiple prototypes exist. Have a look at class CLKDevice to learn more (3.2).

1 #include **"tlm.h"**

2 #include **"amba.h"**

3 #include **"socrocket.h"**

4 #include **"power\_monitor.h"**

5

6 #include **"signalkit.h"**

7 #include **"testbench.h"**

8 #include **"ahbctrl.h"**

9 #include **"ahbmem.h"**

10

11 **int** sc\_main(**int** argc, **char**\*\* argv) {

12

13 // \*\*\* CREATE MODULES

14

15 // Create testbench

16 testbench tbm(**"Master"**, 0x400, 0xfff, 0, sc\_core::sc\_time(10, SC\_NS), amba::amba\_LT);

17

18 // Create ahbctrl

19 AHBCtrl ahbctrl(**"ahbctrl"**,

20 0xfff, // ioaddr

21 0xfff, // iomask

22 0xff0, // cfgaddr

23 0xff0, // cfgmask

24 0, // rrobin (no effect at LT)

25 0, // split (no effect at Lt)

26 0, // defmask

27 0, // ioen

28 0, // fixbrst

29 1, // fpnpen

30 1, // mcheck

31 1, // pow\_mon

32 amba::amba\_LT);

33

34 // Create simulation memory

35 AHBMem ahbmem(**"ahbmem"**, 0x400, 0xfff, amba::amba\_LT, 0);

36

37 // \*\*\* BIND SOCKETS

38

39 // Connect testbench master to ahbctrl

40 tbm.ahb(ahbctrl.ahbIN);

41

42 // Connect ahbctrl to simulation memory

43 ahbctrl.ahbOUT(ahbmem.ahb);

44

45 // Connect snooping ports

46 ahbctrl.snoop(tbm.snoop);

47

48 // Set ahbctrl cycle-delay

49 ahbctrl.set\_clk(10, SC\_NS);

50

51 // Start of simulation

52 // -------------------

53 sc\_core::sc\_start();

54

55 // Call power analyzer

56 PM::analyze(**"../../../models/"**,**"main-power.dat"**,**"ahbctrl.1.lt.power"**);

57

58 **return** 0;

59

60 }

# APBCTRL systemc model

## Functionality and Features

### Overview

The APBCTRL TLM model can be used to simulate behavior and timing of the GRLIB APBCTRL AHB-to-APB Bridge VHDL IP. The model is available at two level of abstractions (LT and AT). For modeling the APBCTRL we mostly follow the recommendations given in RD06.

### Address Decoding

For address decoding the TLM APBCTRL uses the same arithmetic as the GRLIB VHDL model. Each APB slave provides a configuration record identifying its address range. This is done using two parameters: **paddr** and **pmask**. The **paddr** represents the 12bit APB base address of the device. The **pmask** parameter indicates the size of the address range. If **addr** is the 12 bit APB address (bits 20 – 8 of absolute address) of a transaction following logic equation must be solved:

**select = (addr ^ paddr) & pmask**

Address **addr** falls in the range of the slave, if **select** equals zero.

### Plug & Play Support

The TLM APBCTRL supports the Plug & Play (PNP) mechanism described in RD04. APB configuration records and access functions are implemented in class **APBDevice**. Each slave connected to the APBCTRL must be derived from this class. The PNP information of the slaves is collected at **start\_of\_simulation** (). The combined information is mapped on a read-only area at the top 4kbytes of the bridge address space.

### Power Monitoring

Power monitoring can be enabled by setting constructor parameter **pow\_mon**. The IP registers with the power monitor under the name **apbctrl**. In the current version of the library two events are traced:

*idle* – Active during the completed runtime of the simulation (represents static power)

*apb\_trans* – Indicates that a transfer is in progress (dynamic power)

## Interface

The GRLIB VHDL model of the APBCTRL is configured using Generics. For the implementation of the TLM model most of these Generics were refactored to constructor parameters of class apbctrl. An overview about the available parameters is given in Table 6.

|  |  |
| --- | --- |
| Parameter | Description |
| nm | SystemC name of the module |
| haddr | The 12bit MSB address at the AHB bus |
| hmask | The 12bit address mask for the AHB bus |
| mcheck | Check if there are any intersections between APB slave memory regions |
| hindex | The AHB bus index |
| pow\_mon | Enable power monitoring |
| ambaLayer | Coding style/abstraction of the model (LT or AT) |

Table - APBCTRL Constructor Parameters

The system-level interface of the APBCTRL comprises an AHB slave socket (**ahb**) and an APB master socket (**apb**). The APB socket can be bound to multiple slaves (multi-socket), while the AHB socket may be bound to only one master. Depending on the constructor parameter **ambaLayer** the **ahb** socket is configured for blocking (LT) or non-blocking (AT) communication. The **ambaLayer** parameter has no effect on the **apb** socket. For the sake of performance the APB communication is modeled using blocking transport only. In case of LT configuration a TLM blocking transport function is registered at the **ahb** socket. For the AT abstraction the model provides a TLM non-blocking forward transport function. Additionally, the model contains a debug transport function for non-intrusive code execution (TRAP) and checking. The signatures of all transport functions are compliant with the TLM2.0 standard. Moreover, the module inherits SignalKit inputs for clock cycle time (**clk**) and reset (**rst**) from class CLKDevice. APBCTRL is also derived from class AHBDevice. Hence, it exposes a PNP configuration record, which is mapped into the configuration area of AHBCTRL.

## Internal Structure

This section describes the internal structure of the APBCTRL. The class hierarchy of the model is flat. All functionality is comprised in class APBCtrl, which is described in the files **apbctrl.h** and **apbctrl.cpp**.

### Decoder initialization

Similar to the AHBCTRL, the address decoder of the APBCTRL is based on a routing table implemented in form of a **std::map**. The **std::map slave\_map** is initialized in function **start\_of\_simulation**. The function iterates through all slaves bound to socket **apb**. If the slave is a valid APB Device (must be derived from class **APBDevice**) the module creates a new address entry in **slave\_map**. The function also copies the configurartion information of the attached slaves into a 32bit wide array (mSlaves). This array is mapped in the configuration area of the APBCTRL (as described in RD04), where it can be accessed by any bus master.

### LT behaviour

Compared to AHB, APB is a rather simple protocol. From the perspective of an AHB bus master the APBCTRL is an ordinary slave device. The APBCTRL does not do any arbitration. Moreover, APB communication is not pipelined. Therefore, the ambaLayer constructor parameter only effects the AHB slave interface of the APBCTRL. The APB socket uses blocking communication.

Most of the behaviour of the APBCTRL is encapsulated in a single function (**exec\_decoder**). In LT mode this function is directly called from **b\_transport**. The **exec\_decoder** function first checks whether the incoming transaction is directed toward the configuration area or not. In the first case the **getPNPReg** function is used to access the APB configuration records (**mSlaves**). The APB configuration area is read-only. Write operations cause a TLM\_COMMAND\_ERROR\_RESPONSE. In the second case **exec\_decoder** calls **get\_index**. The **get\_index** function receives the address of the transaction as an input argument and returns the id of the slave binding (**index**). For this reason **get\_index** iterates through the previously described **slave\_map**. In case no slave can be found the function returns -1. This produces a TLM\_ADDRESS\_ERROR\_RESPONSE and an error message will be written to stdout. In case of success the transaction is send to the identified slave by calling its **b\_transport** function:

**apb[index]->b\_transport(\*trans, delay);**

Since APBCTRL is a bus bridge, the payload event needs to be copied. In this process the segment address of the bridge (**haddr**) is removed from address field of the transaction.

The LT APBCTRL adds one cycle of delay to the transaction in order to approximate the delay of the APB setup phase. The delay may be consumed by the slave or added to the latency of the target. The LT APBCTRL does not synchronize with the SystemC kernel. The transaction delay is returned to the master, who is responsible for consuming the passed time.

### AT behaviour

The AT mode is intended to more accurately approximate the timing of the GRLIB APBCTRL hardware model. This is achieved by respecting the pipelined nature of the AHB protocol. In AT mode the APBCTRL contains two SystemC threads. A routing table is not required, because the communication on the APB side is always blocking. Hence, no more than one transaction can be active on the APB at any time.

A new transaction arrives in **nb\_transport\_fw** with phase BEGIN\_REQ. The function enters the transaction in the **mAcceptPEQ** payload event queue. After consumption of the component accept delay, **mAcceptPEQ** triggers the acceptTXN thread. The latter is responsible for sending END\_REQ to the AHBCTRL. This is the signal for the AHBCTRL that the AHB address phase is completed.

In case of a read transaction, **acceptTXN** forwards the transaction to the **processTXN** thread (via the **mTransactionPEQ** payload event queue). **ProcessTXN** calls the **exec\_decoder** function, which has already been described above (see LT behaviour). After the control has returned from the slave device, **processTXN** sends BEGIN\_RESP on the backward path. Afterwards, the transaction is considered complete. An eventual END\_REQ from the master will be ignored.

If the transaction indicates a write operation, the **mTransactionPEQ** is written from the **nb\_transport\_fw** function, after reception of BEGIN\_DATA. This also triggers the **processTXN** thread and a call to **exec\_decoder**. After return from **exec\_decoder** END\_DATA is send on the backward path. This completes the AHB data phase.

For more informationon about the AHB AT implementation please see RD09.

## Compilation

For the compilation of the APBCTRL unit, a WAF wscript file is provided and integrated in the superordinate build mechanism of the library.

All required objects for simulating the APBCTRL on platform level are compiled in a sub-library name **apbctrl** using following build command:

**./waf –target=apbctrl**

To utilize **apbctrl** in simulations with other components, add **apbctrl** the **use** list of your wscript.

## Example Instantiation

This example shows how to instantiate the module APBCTRL. The APBCTRL is a bridge between the AHB and the APB portion of the AMBA bus system. The component is created in lines 36-41. In line 46 the module is bound to the master socket of the AHBCTRL. Line 49 binds a slave, here the control interface of the MCTRL, to the master socket of the APBCTRL. Similar to the AHBCTRL the APBCTRL needs a notion of time. Hence, it inherits the clock interface of class **CLKDevice**. In this examples the clock cycle time is set in line 55.

1 #include **"amba.h"**

2

3 #include **"ahbctrl.h"**

4 #include **"apbctrl.h"**

5 #include **"genericmemory.h"**

6 #include **"mctrl.h"**

7 #include **"testbench.h"**

8

9 #include **<systemc.h>**

10

11 **using** **namespace** std;

12 **using** **namespace** sc\_core;

13

14 **class** Top : **public** sc\_module {

15 **public**:

16

17 // \*\*\* DECLARE MODULES

18

19 // Testbench master

20 Testbench testbench;

21

22 // AHB bus model

23 AHBCtrl ahbctrl;

24

25 // APB Bridge

26 APBCtrl apbctrl;

27

28 ...

29

30 // Constructor

31 Top(sc\_module\_name nm) : sc\_module(nm),

32

33 ...

34 ahbctrl(**"ahbctrl"**, 0xfff, 0xfff, 0xff0, 0xff0, 0, 0, 0, 0, 0, 1, 0, 0, amba::amba\_LT),

35

36 apbctrl(**"apbctrl"**, // SystemC name

37 0x800, // AHB base address

38 0xfff, // AHB address maks

39 true, // mcheck - Check consistency of address map

40 1, // hindex - AHB bus index

41 amba::amba\_LT) {

42

43 ...

44

45 // APB bridge to AHB bus

46 ahbctrl.ahbOUT(apbctrl.ahb);

47

48 // Memory controller to APB bus

49 apbctrl.apb(mctrl.apb);

50

51 ..

52

53 // Set clock

54 ahbctrl.set\_clk(10,SC\_NS);

55 apbctrl.set\_clk(10,SC\_NS);

56

57 }

58

59 **virtual** ~Top() {}

60 };

# MCTRL Memory Controller SystemC Model

## Functionality and Features

### Overview

The TLM model of the MCTRL unit models behaviour and timing of the GRLIB MCTRL VHDL implementation described in RD04. It controls a memory subsystem comprising four different types of memory: PROM, I/O, SRAM, and SDRAM. All these memories can be accessed through an AHB slave socket, using an internal address decoder. The control register interface of the device is modeled as a GreenReg register bank, which is attached to an APB slave socket. Hence, like any other device containing GreenReg registers, class **mctrl** is derived from class **gr\_device** (3.4).

The MCTRL is a slave on the AHB bus and on the APB bus. Respectively, it inherits PNP configuration records from classes **AHBDevice** and **APBDevice**. The timing of the model is approximated at two different levels of abstraction (LT and AT).

### Control Registers

The register control interface consists of four configuration registers (Table 7). All of them are 32 bits wide.

|  |  |
| --- | --- |
| APB Address Offset | Register |
| 0x00 | MCFG1 (PROM and I/O) |
| 0x04 | MCFG2 (RAM) |
| 0x08 | MCFG3 (SDRAM Refresh Period) |
| 0x0C | MCFG4 (Power Saving Configuration) |

Table 8 – MCTRL Registers

Memory configuration register 1 is used to program the timing of rom and local I/O accesses (Table 8).

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 31 29 | 28 27 | 26 | 23 20 | 19 | 18 12 | 11 | 10 | 9 8 | 7 4 | 3 0 |
| RESERVED | IO BUSW | IBRDY | IO WS | IOEN | Reserved | PWEN |  | PROM WIDTH | PROM WWS | PROM RWS |

Table - MCFG - Memory Configuration Register 1 (MCFG1)

[IOBUSW] I/O bus width – Sets the data width of the I/O area (00–8bit, 01–16bit, 10–32bit)

[IBRDY] I/O bus ready enable – Enables bus ready signalling for the I/O area.

[BEXCN] Bus error enable – Enables bus error signalling.

[IOWS] I/O waitstates – Sets the number of waitstates during I/O access (0-15)

[PWEN] PROM write enable – Enables write cycles to the PROM area

[PROMWIDTH] Data width of the PROM area (00-8bit, 0-16bit, 10-32bit)

[PROM WWS] PROM write waitstates (0-15)

[PROM RWS] PROM read waitstates (0-15)

Memory configuration register 2 is used to control the timing of the SRAM and SDRAM (Table 9).

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 31 | 30 | 29 27 | 26 | 25 23 | 22 21 | 20 19 | 18 | 17 | 16 |
| SDRF | TRP | SDRAM TRFC | TCAS | SDRAM BANKSZ | SDRAM COLSZ | SDRAM CMD | D64 | RES | MS |

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 15 | 14 | 13 | 12 9 | 8 | 7 | 6 | 5 4 | 3 2 | 1 0 |
| RES | SE | SI | RAM BANK SIZE |  | RBRDY | RMW | RAM WIDTH | RAM WRITE WS | RAM READ WS |

Table - MCTRL - Memory configuration register 2 (MCFG2)

[SDRF] SDRAM refresh – Enables SDRAM refresh

[TRP] SDRAM tRP will be equal to 2 or 3 system clocks (0/1)

[TRFC] SDRAM tRFC will be equal to 3+field\_value system clocks

[TCAS] SDRAM CAS delay 2 or 3 cycles (0/1). Also sets tRCD

[BANKSZ] SDRAM bank size – Sets the bank size for SDRAM chip selects (000 – 4 MB,  
 001 – 8 MB, … 111 – 512 MB)

[COLSZ] SDRAM column size (00 – 256, 01 – 512, 10 – 1024, 11 – 4096)

[SD CMD] SDRAM command – (01 – PRECHARGE, 10 – AUTO-REFRESH, 11 – LOAD-CMD)

[D64] SDRAM data bit is 64 bit wide (0/1)

[MS] Mobile SDR support enabled (0/1)

[SE] SDRAM enable – Enables SDRAM (0/1)

[SI] SRAM disable – Disables SRAM if SE is set to 1.

[RAM BS] Sets the size of each SRAM bank (0000 – 8kb, 0001 – 16kb, 1111 – 256 MB)

[RBRDY] Enables bus ready signalling for the SRAM area

[RMW] Enables read-modify-write cycles for sub-word writes to 16bit or 32bit areas

[RAM WIDTH] Sets the data width of the SRAM area (00 – 8, 01 – 16, 1X – 32)

[RAM WWS] Sets the number of wait states for SRAM read cycles (0 – 3)

[RAM RWS] Sets the number of wait states for SRAM write cycles (0 – 3)

MCFG3 is dedicated to SDRAM control and MCFG4 to power saving options (see RD04).

### Address Space

The address space is divided in the three partitions: PROM, I/O, and RAM. The division of the address space is static and cannot be modified after initialization of the MCTRL unit. In the VHDL implementation, the different parts of the address space are calculated from generics, which are implemented as constructor parameters in the TLM module.

The PROM address space is derived from the parameters **romaddr** and **rommask**, which define the start address and the size of the PROM address space. The **romaddr** is written to the 12 bit-wide **ADDR** field of the AHBDevice **BAR0** register of the MCTRL. The **rommask** is written to the 12 bit-wide **MASK** field of the AHBDevice BAR0 register. The PROM address space is byte-addressable and has an address width of 32 bit.

The size of the PROM address space is:

(212 – **MASK**) MByte

The address space is divided into two PROM banks of equal size.

The local I/O address space is calculated in the same way as the PROM address space. All calculations are based on the **ioaddr** and **iomask** parameters. The only difference to PROM is that no memory banks are defined.

The SRAM address space is derived from the **ramaddr** and **rammask** parameters. Again calculations are very similar to PROM and IO. Although, the partitioning of the resulting address space depends on the settings in the **MCFG2** register. The register provides the fields SDRAM enable (**SE**) and SRAM disable(**SI**) indicating the presence of SRAM, SDRAM, or SRAM & SDRAM. If the **SE** bit is low, **SI** has no effect.

For details and information on the organization of the SRAM address space, regarding the number of banks, bank locations, bank sizes, and – in case of SDRAM – number of row and column address bits, see the GRLIB user manual (RD04). Examples for possible partitionings of the RAM address space (default size of 1 GByte) are given in Figure 2.

Figure 3 – RAM address space

The default configuration is the SRAM only configuration (Config 1). The entire RAM address space can be split into up to five SRAM banks. The number of SRAM banks is defined by the constructor parameter **srbanks**. By default four SRAM banks are configured. Banks 1-4 are always located in the lower half of the RAM address space. Their size is variable between 8 KByte – 256 Mbyte. It can be set using the RAM BANK SIZE field (**RAM BS**) of the **MCFG2** register. If the bank size exceeds 128 MByte, the number of banks must be reduced or the size of the address space must be increased. In the SRAM only configuration, a fifth bank can be attached to take up the upper half of the RAM address space.

In the second configuration (Config 2) both SRAM and SDRAM are enabled. In this case, the lower half of the RAM address space is populated by up to 4 SRAM banks. SRAM bank 5 cannot be present, because two SDRAM banks are mapped to the upper half of the RAM address space. The size of the SDRAM banks is scalable between 4 MByte – 512 MByte, according to the **SDRAM****BANKSZ** field of the **MCFG2** register. If the SDRAM bank size exceeds 256 MByte, i.e. if the SDRAM bank size is set to 512MByte, the RAM address space needs to be extended to the size of 2GByte to fit the SDRAM in the upper half of the RAM address space. As this will also extend the SRAM address space to 1GByte giving room for the maximum number of four SRAM banks of the maximum supported size of 256 MByte, a size of 2GByte represents the maximum sensible RAM address space. Such a configuration would be reflected by **rammask** 0x800.

In the SDRAM only configuration (Config 3), the SDRAM banks are mapped into the lower half of the RAM address space. If the SDRAM bank size is set to 512MByte, the RAM address space needs to be extended to the size of 2GByte. The upper half of the address space remains reserved and unused.

In any configuration, the initial bank sizes are calculated to be the maximum possible size, which can be deduced from address space size and number of banks.

It is possible to switch between the three configurations shown in Figure 2 by overwriting the respective bits in the **MCFG2** register. In such an event the MCTRL recalculates the start and end addresses of all SRAM and SDRAM memory banks. Because only the address decoding changes, the content of the memories is not affected. However, the bus master must take care to read from the correct memory banks after having caused a reorganization of the RAM address space. The bus master also has to take care of not exceeding the RAM address space when changing the SRAM or SDRAM bank size. If, for example, the RAM address space is 1GByte and the size of four SRAM banks is dynamically switched from 128 MByte to 256 GByte, the SRAM banks will take up the SDRAM address space, causing an overlap of SRAM and SDRAM device addresses. Due to the SystemC code structure, any access to SDRAM would then be redirected to SRAM causing system malfunction.

The way the MCTRL TLM model handels memory access depends on the typ of memory addressed. This information is extracted by evaluating the target address. In the default case the delay of a transaction is fully modeled in the MCTRL unit, which holds information about all timing parameters involved. The timing parameters are given in the configuration registers. Additional delay information can be deduced from the streaming width and data length, in case of burst transactions. Optionally, the **IORBY** (**MCFG1**) and **RBRBY** (**MCFG2**) may be used to obtain additional timing information from the attached memory models and simulate bus-ready-signalling. All delay values are calculated as multiples of the bus clock period. PROM Access

In case of PROM, write access needs to be explicitly allowed by setting the **PWEN** bit of the **MCFG1** register. Forbidden write operations, will be cancelled and produce a TLM\_COMMAND\_ERROR\_RESPONSE as well as an error message printed to std::out.

A read access to PROM memory requires 4 bus cycles plus 0 – 15 wait states. A write access to PROM memory takes 3 bus cycles plus 0 – 15 wait states. The wait states can be configured via the **PROM READ WS** and **PROM WRITE WS** fields of the **MCFG1** register. The attached PROM memory may have a data width of 8, 16 or 32 bits. This must be reflected by the settings in the **PROM** **WIDTH** field of **MCFG1**. If the **PROM** **WIDTH** field is set to 16 or 8 bits, a read access to PROM will still result in loading a full 32 bit word from memory. The word will be transmitted in a burst of two half-words or four single bytes, adding a delay of two bus cycles (data1 and data2) in 16 bit mode or six bus cycles (3x data1 and 3x data2) in 8 bit mode.

### Local I/O Access

The local I/O area supports access to 32 bit words, 16 bit half-words, and single bytes. A read access takes 4 bus cycles (lead-in, data1, data2, lead-out) and a write access takes 3 bus cycles (lead-in, data, lead-out). For both, read and write operations, the model provides a mechanism for dynamic *bus-ready-signaling*, which can induce an arbitrary number of wait states. It is the task of the attached I/O device to model this delay and add it to the delay parameter of the TLM transport function. The MCTRL unit will observe the delay parameter and add its value to the overall transaction delay.

### SRAM Access

The access to SRAM is similar to the PROM access, the difference being the number of wait states (0 – 3). For a read access, the number of wait states can be set via the **RAM** **READ** **WS** field of the **MCFG2** register. Read accesses to SRAM bank 5 and write accesses to SRAM support dynamic wait states in the VHDL model. Similar to the dynamic *bus-ready-signaling* in the I/O area, it is the task of the TLM memory device to add this delay to the delay variable of the TLM transaction.

### SDRAM Access

In the VHDL model of the MCTRL SDRAM is accessed over a separate bus, if the **sepbus** parameter is set to one. This bus can have a width of 32 or 64 bit, as indicated by the**D64** field of the **MCFG2** register. For TLM communication this architectural detail (separate bus) is not relevant. However, the **D64** bit is taken into account for delay calculation, because it affects the streaming width to and from memory.

At TLM level it is also not necessary to model the SDRAM commands, which are emitted by the SDRAM controller. It is only important to estimate the impact of the different command sequences on the memory access time. This especially accounts for opening and closing memory rows for read and write access. The delay of an ACTIVATE command is added to any operation that needs to buffer a new SDRAM row. Closing a row comes at the cost of the delay contributed by a PRECHARGE command.

A read access to SDRAM is always performed as a page burst access. Because a page bursts can be interrupted by a PRECHARGE command, it is possible to read an arbitrary number of data words. In the TLM model, the data length field of the generic payload can hence be set to any multiple of the SDRAM word length. The delay will be calculated for opening the row, sending one data word each clock cycle, and closing the row again. If the requested sequence of words starts at the end of a row and ends in the next row, the time for opening and closing the second row will be added.

The time required for opening a row is determined by the **TCAS** field of the **MCFG2** register. If the **TCAS** field is changed, a real hardware memory device would require a notification. In the RTL model this is done by sending LMR command. In the TLM model, the MCTRL unit models the timing of each transaction and expects the memory model to behave correctly, i.e. an LMR command would not have any functional effect. Hence, the LMR command is not issued, but its delay is modeled by adding it to the next transaction.

A write access to SDRAM is always performed as a single word write, i.e. burst mode is not supported. A requested write burst from the bus will be transformed into a burst of writes.

To retain data in memory, refresh cycles are required. The MCTRL unit only supports devices capable of AUTO REFRESH, i.e. MCTRL only needs to periodically trigger the refresh, which is then organized by the memory internally. In the TLM implementation, the refresh has no functional effect, but influences the overall operational speed of the memory device. The model keeps track of the refresh period and locks the SDRAM for the duration of one refresh cycle after each refresh period. If an access to the SDRAM device is requested while SDRAM is locked, the transaction will be stalled for the rest of the refresh cycle. The other way round, a refresh can also be stalled by a transaction.

### SDRAM Modes of Operation

The MCTRL unit can configure the SDRAM device to operate in several modes. The various operation modes relate to different power saving options. Operation modes can be enabled using the **mobile** constructor parameter.

On system start, SDRAM is always initialized for “normal operation” (6.1.7.1). The initialization sequence of the hardware model is emulated by adding delay to the first transaction.

In case mobile memory is not supported (**mobile = ‘0’**) the **MS** field of **MCFG2** is set to zero. This disables the power saving features exposed by the **MCFG4** register of the device. If the **mobile** parameter is set to one, mobile memory is supported, but disabled by default. The **MS** field of the **MCFG2** register is set to one, but the **ME** field of the **MCFG4** register is set to zero. Non of the settings in **MCFG4** has any effect as long **ME** is disabled. For **mobile = ‘2’**, mobile memory is supported and enabled by default. For **mobile = ‘3’**, mobile memory cannot be disabled, i.e. the **ME** field of **MCFG4** becomes read only.

If mobile memory is enabled, the SDRAM device supports the following power saving modes: Power Down (6.1.7.2), Self-Refresh (6.1.7.3), Partial Array Self Refresh (6.1.7.4), and Deep Power Down (6.1.7.5). The actual mode can be selected by writing the **PMODE** field of **MCFG4**.

#### Normal Operation Mode

In normal operation mode, the memory access functions as described in section 6.1.6. In case of a change in the operation mode, a hardware SDRAM memory would require to be reconfigured. This would be done using the LOAD\_EXTENDED\_MODE\_REGISTER (EMR) command. Like the LMR command, EMR does not have any functional effect. Moreover, the impact of EMR on the timing is neglectable. Therefore, at TLM-level EMR is ignored.

#### Power Down Mode

To enter power down mode, mobile memory must be enabled and the **PMODE** field of the **MCFG4** register must be changed to **“001”**. In power down mode, the input and output buffers of the SDRAM device are deactivated after an idle period of 16 clock cycles. The buffers can be woken up within one clock cycle at any time. Respectively, each TLM memory access requires an additional delay of one bus clock cycle.

#### Self-Refresh Mode

If the system is powered down, mobile SDRAM can retain its content by switching into self-refresh mode. Entering self-refresh mode is induced by setting the **PMODE** field of the **MCFG4** register to **“010”.**

In self-refresh mode the system is supposed to be shut down. Therefore, no accesses to memory are expected (despite theoretically possible). For TLM requests in self-refresh mode a warning will be send to std::out.

#### Partial Array Self-Refresh Mode

In partial array self-refresh mode parts of the memory can be retained during power down. The mode is entered by setting the three-bit-wide **PASR** field of the **MCFG4** register to a value not equal to zero.

The partial array can be defined as the half, quarter, eighth, or sixteenth part of the memory (**PASR** = 001, 010, 101, or 110). lt is always associated to the lower bound of the SDRAM address space.

Entering partial array self-refresh mode immediately erases all parts of the TLM SDRAM memory, which are outside of the refresh array.

#### Deep Power Down Mode

To enter deep power down mode, mobile memory must be enabled and the **PMODE** field of the **MCFG4** register must be changed to **“101”**. Thereafter, the content of the SDRAM memory is immediately deleted.

During deep powe down mode all accesses to SDRAM produce a TLM\_ADDRESS\_ERROR\_RESPONSE.

Deep power down mode can be left by changing the **PMODE** field of **MCFG4** to any other mode of operation.

### Power Modeling

Power modeling can be enabled by setting constructor parameter **powermon**. The IP registers with the power monitor under the name **mctrl**. The power events produced by the MCTRL relate to the different operation modes of the device, which have direct impact on the power consumed by the attached memories:

*idle –* Static power in Nominal Operation Mode

*idle\_powerdown –* Static power in Power Down Mode

*idle\_partpowerdown2 –* Partial Array Self-Refresh Mode for half the memory

*idle\_partpowerdown4 –* Partial Array Self-Refresh Mode for a quarter of the memory

*idle\_partpowerdown8 –* Partial Array Self-Refresh Mode for the eight part of the memory

*idle\_partpowerdown16 –* Partial Array Self-Refresh Mode for the sixteenth part of the memory

*idle\_selfrefresh –* Static power in Self-Refresh Mode

*idle\_deeppowerdown –* Static power in Deep-Power-Down Mode

## Interface

The GRLIB VHDL model of the MCTRL is configured using Generics. For the implementation of the TLM model most of these Generics were refactored to constructor parameters of class **mctrl**. An overview about the available parameters is given in Table 10.

|  |  |  |  |
| --- | --- | --- | --- |
| **Parameter** | **Function** | **Allowed Range** | **Default** |
| name | SystemC name of the module |  |  |
| romasel | log2(PROM address space size) - 1.  E.g. If size of the PROM area is 0x20000000 romasel is:  log2(2^29)-1 = 28. | 0 – 31 | 28 |
| sdrasel | log2(RAM address space size) - 1.    E.g If size of the RAM area is 0x40000000 sdrasel is:  log2(2^30)-1= 29. | 0 – 31 | 29 |
| romaddr | ADDR field of BAR0 defining PROM address space. | 0 – 0xFFF | 0x000 |
| rommask | MASK field of BAR0 defining PROM address space size. rommask = PROM address space size in MByte | 0 – 0xFFF | 0xE00 |
| ioaddr | similar to romaddr | 0 – 0xFFF | 0x200 |
| iomask | similar to rommask | 0 – 0xFFF | 0xE00 |
| ramaddr | similar to romaddr | 0 – 0xFFF | 0x400 |
| rammask | similar to rommask | 0 – 0xFFF | 0xC00 |
| paddr | ADDR field of the APB BAR configuration registers address space | 0 – 0xFFF | 0x000 |
| pmask | MASK field of the APB BAR configuration registers address space | 0 – 0xFFF | 0xFFF |
| wprot | RAM write protection | 0 – 1 | 0 |
| srbanks | Number of SRAM banks | 0 – 5 | 4 |
| ram8 | Enable 8 bit PROM and SRAM access | 0 – 1 | 0 |
| ram16 | Enable 16 bit PROM and SRAM access | 0 – 1 | 0 |
| sden | Enable SDRAM controller | 0 – 1 | 0 |
| sepbus | SDRAM is located on separate bus | 0 – 1 | 1 |
| sdbits | 32 or 64 bit SDRAM data bus | 24, 64 | 32 |
| mobile | Enable Mobile SDRAM support  0: Mobile SDR is not supported  1: Mobile SDR is supported but disabled  2: Mobile SDR is supported and default  3: Mobile SDR support only | 0 – 3 | 0 |
| hindex | AHB slave index | 0 - NAHBSLV–1 | 0 |
| pindex | APB slave index | 0 - NAPBSLV–1 | 0 |
| pow\_mon | Enable power monitoring | 0 – 1 | 0 |
| ambaLayer | TLM abstraction/coding style | LT/AT | LT |

Table – MCTRL Constructor Parameters

The system-level interface of the TLM MCTRL comprises an AHB slave socket (**ahb**), an APB slave socket (**apb**) and a GreenSocks initiator multi-socket (**mem**). The AHB socket is intended to be bound to the TLM model of the AHBCTRL. The APB socket must be connected to the TLM model of the APBCTRL. The **mem** socket may be connected to any device implementing the **memdevice** memory interface (3.2). The SoCRocket library provides a generic memory complying with this condition (7). It can be used as a ROM, I/O, SRAM or SDRAM device.

Depending on the constructor parameter **abstrLayer** the AHB socket is configured for blocking (LT) or non-blocking (AT) communication. The APB socket is blocking transport (LT) only - indepent of **abstrLayer**. In the LT case the MCTRL registers two TLM blocking transport functions: one for **ahb** and one for **apb**. For the AT abstraction the model provides a TLM non-blocking forward transport function for the AHB socket and a blocking transport function for the APB socket. Additionally, the model contains debug transport functions for both sockets. The signature of all transport functions are compliant with the TLM2.0 standard.

Next to the TLM sockets the model comes with SignalKit inputs for clock cycle time (**clk**) and reset (**rst**). Both of them are inherited from class **CLKDevice**, which is shared amongst most of the models in the library.

## Internal Structure

This section describes the internal structure of the MCTRL. A basic overview is given in Figure 3. The class hierarchy of the model is flat. All functionality is comprised in class **mctrl**, which is described in the files **mctrl.h** and **mctrl.cpp**.

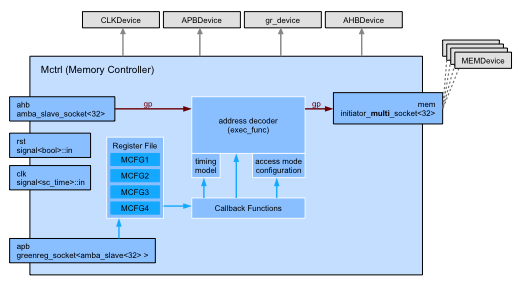


Figure 4 – Structure of the TLM MCTRL

### Decoder initialization

The central behaviour of the MCTRL is represented by its internal address decoder. The decoder decides how and under which conditions a transaction from the AHB socket is forwarded to one of the connected slave memories.

Before the begin of a simulation the decoder must be initialized. This is done in two steps using SystemC callbacks. First, in the **end\_of\_elaboration** function, the memory mapped registers are created, bound and initialized. Afterwards, the **start\_of\_simulation** function iterates through all memories bound to the **mem** socket extracting their configuration information. This is done using the member functions of the **memorydevice** interface. The **start\_of\_simulation** function updates the settings of the MCTRL control registers and creates one PNP base address register entry per memory device (**BAR0-3**). At simulation time the address information in the **BAR** records is the base for routing the TLM transactions from the **ahb** socket to the memories.

### LT behaviour

In LT mode the constructor of the MCTRL registers a blocking transport function (**b\_transport**) at the **ahb** slave socket. All payload objects arriving in **b\_transport** are directly forwarded to the functional part of the model, which is encapsulated in function **exec\_func**.

The **exec\_func** function receives the payload object and the transaction delay pointer as input parameters. At the beginning of the function the payload is extracted. Address, length and data pointer are locally copied. In the next step the address is decoded using the **get\_ports** function. **Get\_ports** compares the address of the transaction with the settings in the decoder registers (**BAR0-3**). It returns an object of type **MEMPort**. In case no slave could be found the MCTRL generates a **TLM\_ADDRESS\_ERROR\_RESPONSE** and an error message. In the following a set of checks is performed to ensure the characteristics of the access are compatible with the addressed memory device. It is checked whether e.g. the transfer length of the transaction is compatible with the memory width, read-modify-write cycles must be inserted or if the targeted memory region is writeable (e.g. PROM). Please see the source code documentation for more detailed information. If one of the constraints is not met, the MCTRL generates a **TLM\_GENERIC\_ERROR\_RESPONSE**.

Afterwards, the MCTRL calculates the base delay for transfering one word of data to the selected memory (**word\_delay**) and the delay offset, which might be involved in the transaction (**trans\_delay** – e.g. for opening a SDRAM row). The calculations have various dependencies. For PROM, I/O and SRAM the number of wait-states are encoded in **MCFG1** and **MCFG2**. The parameters for the SDRAM timing can be found in **MCFG2** and **MCFG3** (**TRP, TRFC, TCAS, …**). For mobile SDRAM, additional delay is accumulated for Power Down and Partial Array Self Refresh Mode. Operations directed to memory in Deep Power Down Mode create a **TLM\_GENERIC\_ERROR\_RESPONSE**.

In the following **exec\_func** creates and initializes a new generic payload. Thereby, the global target address is transformed into an absolute address for the selected memory. The communication between MCTRL and memory is always blocking:

**mem[port\_id]->b\_transport(memgp, mem\_delay);**

If bus-ready signalling is enabled (IBRDY, RBRDY) the calculation of the actual transfer delay is left to the memory (**mem\_delay**). Otherwise, **mem\_delay** is ignored and the final delay is calculated using the transfer base delays (**word\_delay**, **trans\_delay**), the transfer length, the memory width and the clock cycle time.

After return from **exec\_func** the model calls **wait** to consume the component delay. Optionally, this task can be shifted to the bus master.

### AT behaviour

The AT mode is intended to more accurately approximate the timing of the GRLIB MCTRL. This is achieved by modeling the pipelined nature of the AHB protocol, which allows address phase and data phase of consequtive transactions to overlap. The MCTRL registers two **SC\_THREAD**s to support that feature: **acceptTXN** and **processTXN**. Moreover, it provides a TLM non-blocking forward transport function (**nb\_transport\_fw**). The operation of the module in AT-mode can be best understood by following the control flow of a transaction.

A new transaction arrives in **nb\_transport\_fw** with phase **BEGIN\_REQ**. The function enters the transaction in the **mAcceptPEQ** payload event queue and returns to the caller with **TLM\_ACCEPTED**. The **mAcceptPEQ** triggers the **acceptTXN** thread. In case of a read transaction **acceptTXN** directly copies the payload in the **mTransactionPEQ** - for notification of thread **processTXN**. Write transaction are not send to **processTXN**, before their data pointer becomes valid. This is ackknowledged by the master via a **BEGIN\_DATA** on the forward path.

The **processTXN** thread calls the functional interface of the MCTRL, which is encapsulated in function **exec\_func**. The behaviour of **exec\_func,** has already been described in 6.3.2. After return from **exec\_func** the **processTXN** thread actives the **busy** flag, before consuming the accumulated component delay. The **busy** flag is used by thread **acceptTXN** in order to check whether a transaction is in progress. As long as the **busy** flag is true, new transactions are blocked. This means, they are accepted by **nb\_transport\_fw**, but will not receive **END\_REQ**.

After unblocking **acceptTXN** (**busy=false, unlock\_event**) the **processTXN** thread sends **BEGIN\_RESP** for read transactions and **END\_DATA** for write transactions. A final **END\_RESP** on the forward path will be ignored.

## Compilation

For the compilation of the MCTRL unit, a WAF wscript is provided and integrated in the superordinate build mechanism of the libary.

All required objects for simulating the MCTRL on platform level are compiled in a sub-library name mctrl using following command:

**./waf –target=mctrl**

For using the MCTRL in simulations with other component add **mctrl** to the use list of your WAF **wscript**.

## Example Instantiation

An example is given in section 7.5, which jointly demonstrates the instantiation of MCTRL and GenericMemory.

# GENERIC Memory SystemC Model

## Functionality and Features

### Overview

The Generic Memory (GM) model is not based on any reference design from the Gaisler GRLIB. It was developed from scratch to complement the SoCRocket MCTRL unit (Chapter 6).

The GM is generic in a sense that it can act as one of four supported memory types: PROM, IO, SRAM or SDRAM. All memories to be connected to the MCTRL must be derived from class **MemDevice** (3.2), which encapsulates all configuration options. The MCTRL uses this interface to determine the features of the attached components.

The GM models default devices, which means its behaviour is plain functional. All timing related features are provided and controlled by the MCTRL. As any memory controller needs to know all the timing information of the attached memory device anyway, the delay can be added in the memory controller to keep the memory itself universally applicable. Respectively, the GM is merely used to store data and to identify the device on system level.

### Power Modeling

Power monitoring can be enabled by setting the constructor parameter **pow\_mon**. Depending on the memory type the IP registers with the power monitor using one of the following names: prom, io, sram, sdram. Per memory type three events are known:

*idle* – Active during the complete runtime of the simulation (represents static power).

*{prom, io, sram, sdram}\_read* – Read access to memory

*{prom, io, sram, sdram}\_write* – Write access to memory

## Interface

The Generic Memory model can be configured using a set of constructor parameters (Table 11).

|  |  |
| --- | --- |
| Parameter | Description |
| name | SystemC name of the module |
| type | MEMDevice::device\_type  0 – ROM, 1 - IO, 2 – SRAM, 3 – SDRAM |
| banks | Number of parallel banks to be modeled |
| bsize | Size of one memory bank (All banks always considered to have equal size) |
| bits | Bit width of memory |
| cols | Number of SDRAM cols |
| pow\_mon | Enable power monitoring |

Table - Generic Memory Constructor Parameters

The system-level interface consists of a TLM 2.0 target socket (GreenSocket). The TLM payload comprises an extension for clearing memory regions (**ext\_erase**). In the current version of the model this feature is only used by the SDRAM controller.

Since the GM is a plain functional model the communication with the MCTRL is based on blocking transport (LT).

## Internal Structure

This section describes the internal structure of the Generic Memory. The class hierarchy of the model is flat. All functionality is comprised in class **GenericMemory**, which is described in the files **genericmemory.h** and **genericmemory.cpp**. File **ext\_erase.h** provides an additional payload extension.

### Interface MemDevice

The Generic Memory implements the interface **MEMDevice**. The **MEMDevice** interface enables the MCTRL to identify the type of the memory (PROM, IO, SRAM, SDRAM) and its main configuration parameters. For access to this parameters every **MEMDevice** provides a set of virtual functions, which can be overwritten by the child class. The GM uses the default implementation of the access functions.

**get\_type** – Returns the memory type (MEMDevice::device\_type)

**get\_banks** – Returns the number of parallel memory banks

**get\_bsize** – Returns the size of one memory banks in bytes

**get\_bits** – Returns the width of the memory

**get\_cols** – Returns the number of SDRAM cols

### Functional Memory

The storage of the GM is implemented as a **std::map** (memory) with 32bit wide keys (addresses) and 8bit data entries. Byte access to memory is performed using two access functions: **read** and **write**. The access functions are directly called from the **b\_transport** method.

In case the **ext\_erase** payload extension is set, the respective memory region (**start** – **end**) is cleared using the erase function. This happens when switching SDRAM to Deep-Power-Down-Mode or Partial-Self-Refresh.

## Compilation

The compilation of the GM is integrated in the compilation of the MCTRL. An appropriate WAF wscript is provided and integrated in the superordinate build mechanism of the library.

All required objects for simulating the GM and the MCTRL are compiled in a sub-library named **mctrl** using following command:

**./waf –target=mctrl**

To utilize the GM in simulations with other components, add **mctrl** to the use list of your wscript.

## Example Instantiation

The example below demonstrates the instantiation of the GM as PROM, IO, SRAM and SDRAM. The modules are declared in lines 13-16 and created, within the constructor, in lines 30-36. Lines 44-47 show how to bind the **bus** target socket of the GM to the **mem** initiator socket of the MCTRL.

1 #include **<systemc.h>**

2

3 #include **"genericmemory.h"**

4 #include **"mctrl.h"**

5

6 **class** Top : **public** sc\_module {

7 **public**:

8

9 // Memory controller

10 Mctrl mctrl;

11

12 // Generic memories

13 GenericMemory rom;

14 GenericMemory io;

15 GenericMemory sram;

16 GenericMemory sdram;

17

18 ...

19

20 // Constructor

21 Top(sc\_module\_name mn) : sc\_module(mn),

22

23 ...

24

25 // Initialize MCTRL

26 mctrl(**"mctrl"**, romasel, sdrasel, romaddr, rommask, ioaddr, iomask,

27 ramaddr, rammask, paddr, pmask, wprot, srbanks,

28 ram8, ram16, sepbus, sdbits, mobile, sden,

0, 0, 0, amba::amba\_LT),

29 // Initialize PROM

30 rom(**"rom"**, MEMDevice::ROM, 2, 512 \* 1024 \* 1024 / 2, 32, 0, false),

31 // Initialize IO

32 io(**"io"**, MEMDevice::IO, 1, 512 \* 1024 \* 1024, 32, 0, false),

33 // Initialize SRAM

34 sram(**"sram"**, MEMDevice::SRAM, 4, 512 \* 1024 \* 1024 / 4, 32, 16, false),

35 // Initialize SDRAM

36 sdram(**"sdram"**, MEMDevice::SDRAM, 2, 512 \* 1024 \* 1024 / 2, 32, 16, false) {

37

38 ...

39

40 // Set MCTRL timing for delay calculation

41 mctrl.set\_clk(10, SC\_NS);

42

43 // Connect MCTRL to Generic Memories

44 mctrl.mem(rom.bus);

45 mctrl.mem(io.bus);

46 mctrl.mem(sram.bus);

47 mctrl.mem(sdram.bus);

48

49 }

# MMU\_CACHE Cache sub-system systemC module

## Functionality and Features

### Overview

The **MMU\_CACHE** SystemC IP models behaviour and timing of the Gaisler GRLIB Harvard L1 Cache and GRLIB Memory Management Unit (MMU). It is therefore related to the **mmu\_cache** entity of the GRLIB hardware library.

The structure of the Cache Sub-System is depicted in Figure 4. The top-level class **mmu\_cache** provides two TLM 2.0 **simple\_target\_sockets** (**icio**, **dcio**) for communication with the LEON ISS and one Carbon/GreenSocs **amba\_master\_socket**for the connection to the AHBCTRL. All the sub-components, such as the mmu, the caches and the localrams are implemented in plain C++.

Equivalent to the hardware model the caches can be direct mapped, 2-way, 3-way or 4-way set associative. For multi-set configurations LRU, LRR and pseudo-random replacement are supported. The size of the cache sets can be beween 1 and 64 kbytes, with up to 32 bytes per line. The caches can be flushed, frozen or locked on a line-by-line basis. The write policy of the data cache is write through with no-allocate on write miss. The caches can be separately disabled. In that case requests from the ISS are directly forwarded to the AHB master or the MMU (if enabled).

The localrams can be optionally enabled. They provide 0-waitstate access to up to 512 kbyte of memory, starting from a segment address, which can be freely chosen.

The MMU can also be optionally enabled. The MMU page size is 4, 8, 16 or 32 kbyte. The TLBs can hold between 2 and 32 page descriptors. In case of a page miss a 3-level table walk is carried out on main memory. Similarily to the localrams, instantiation of the mmu is done by late binding depending on configuration parameters. The caches connect to the mmu through **tlb\_adaptor** objects. The **tlb\_adaptors** present a unified memory interface towards the caches (**mem\_if**). The same memory interface is used to provide access to the AHB master socket on top-level. This way it can be dynamically decided whether a request from one of the caches shall be forwarded to a shared or common TLB (virtual addressing), or directly go to the AHB interface (physical addressing).

Figure 5 - Structure of Cache Sub-System

### Address Space Identifiers (ASI)

SPARC processors generate an 8-bit address space identifier (ASI), to provide access to up to 256 separate 32-bit address spaces. A big share of the ASIs is used for control of the cache sub-system. A list of the ASIs supported by the TLM model is given in Table 12.

|  |  |  |
| --- | --- | --- |
| ASI | Address | Usage |
| 0x01 | any | Forced cache miss |
| 0x02 | 0x00 | Cache control register |
|  | 0x04 | Reserved |
|  | 0x08 | Instruction cache configuration register |
|  | 0x0c | Data cache configuration register |
|  | *0xff* | *Trigger debug output\** |
| 0x08,0x09,0x0A,0x0B | any | Normal cache access |
| 0x0c | see 8.1.6 | Access instruction cache tags |
| 0x0d | - “ - | Access instruction cache data |
| 0x0e | - “ - | Access data cache tags |
| 0x0f | - “ - | Access data cache data |
| 0x15 | - “ - | Flush instruction cache |
| 0x16 | - “ - | Flush data cache |
| 0x19 | 0x000 | MMU control register |
|  | 0x100 | MMU Context pointer register |
|  | 0x200 | MMU Context register |
|  | 0x300 | MMU Fault status register |
|  | 0x400 | MMU Fault address register |

Table 13 - Supported ASIs

ASIs are emitted by the data interface of the processor. For this purpose an extension has been linked to the data cache payload object (**dcio\_payload\_extensions**). For more information about payload extensions see section 8.1.7.

The ASIs are decoded in the **exec\_data** function of class **mmu\_cache**. The decoder maps the ASIs to API functions of the corresponding sub-components (caches, mmu). The API functions are described in section 8.2.

### System and Control Registers

The cache sub-system is controlled by a set of system registers, which can be accessed using ASIs.

Three of the mentioned registers are dedicated to the caches (ASI 0x02). The Cache Control Register (CCR - Table 13) effects both, data and instruction cache. Therefore, it is implemented on top-level (**mmu\_cache**). Moreover, each of the caches has its own private Configuration Register (CR - Table 14). The CRs describe structure and size of the caches and are read-only.

|  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 31 24 | 23 | 22 | 21 | 20 17 | 16 | 15 | 14 | 13 6 | 5 | 4 | 3 2 | 1 0 |
|  | DS | FD | FI |  | IB | IP | DP |  | DF | IF | DCS | ICS |

Table 14 - CACHE CONTROL REGISTER

[DS] Data cache snoop enable

If set, will enable data cache snooping (todo).

[FD] Flush data cache

If set, will flush the instruction cache. Always reads zero.

[FI] Flush instruction cache

If set, will flush the instruction cache. Always reads zero.

[IB] Instruction burst fetch

This bit enables burst fill during instruction fetch.

[IP] Instruction cache flush pending (not supported)

[DP] Data cache freeze on interrupt (not supported)

[IF] Instruction cache freeze on interrupt (not supported)

[DCS] Data cache state

Indicates the current data cache state according to the following:

X0 = disable, 01 = frozen, 11 = enabled.

[ICS] Instruction cache state

Indicates the current instruction cache state according to the following:

X0 = disabled, 01 = frozen, 11 = enabled.

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 31 | 30 | 29 28 | 27 | 26 24 | 23 20 | 19 | 18 16 | 15 12 | 11 4 | 3 | 2 0 |
| CL |  | REPL | SN | SETS | SSIZE | LR | LSIZE | LRSIZE | LRSTART | M |  |

Table 15 - ICACHE & DCACHE Configuration Register

[CL] Cache looking

If set, cache locking is implemented

[REPL] Cache replacement policy

00 = non (direct mapped), 01 = least recently used (LRU), 10 = least recently used (LRR), 11 = random

[SN] Data cache snooping

Set if snooping is implemented

[SETS] Number of sets in the cache

000 = direct mapped, 001 = 2-way associative, 010 = 3-way associative, 011 = 4-way associative

[SSIZE] Set size

Indicates the size (Kbytes) of each cache set (Size = 2^SSIZE).

[LR] Local RAM

Set if local scratchpad is present.

[LSIZE] Line size

Indicates the size (words) of each cache line (Line size = 2^LSIZE).

[LRSIZE] Local RAM size

Indicates the size (Kbytes) of the implemented scratchpad RAM   
 (Size = 2^LRSIZE).

[LRSTART] Local RAM start address

Indicates the 8 most significant bits of the local RAM start address.

[M] MMU present

Set if MMU is present

The MMU is controlled by five 32-bit registers, which can be accessed with ASI 0x19 (Table 15). All of them are implemented in class **mmu**.

|  |  |  |  |
| --- | --- | --- | --- |
| *MMU Control Register* | 31 1 | | 0 |
| not used | | E |
| *Context Pointer Register* | 31 2 |  | |
| Context Table Pointer |  | |
| *Context Register* | 31 0 | | |
| Context Number | | |
| *Fault Status Register* | 31 0 | | |
| Not implemented yet | | |
| *Fault Address Register* | 31 0 | | |
| Fault address (not implemented yet) | | |

Table 16 - MMU Control Registers

From the MMU Control Register only one bit is implemented in the TLM model. It is used to enable and disable the MMU.

The Context Pointer Register points to the Context Table in main memory. It forms bit 35 – 6 of the physical address. The table is indexed by the contents of the Context Register.

The Context Register contains the number of the current context and defines which of the possible address spaces is used for address translation.

The Fault Status Register provides information on exceptions (faults) issued by the MMU. It is currently not implemented and reads as 0.

The Fault Address Register contains the virtual memory address of the fault recorded in the Fault Status Register. It is currently not implemented and reads as 0.

### Data Cache Snooping

The **mmu\_cache** IP supports data cache snooping. Snooping can be enabled by setting bit 23 of the Cache Control Register. The model provides a SignalKit input **snoop**. The constructor of **mmu\_cache** registers a callback function (**snoopingCallBack**). The AHBCTRL triggers this function on every write operation. Via the **snoop** input the callback receives the bus id of the responsible master, the target address and the length of the write operation. If the master id does not equal the own id and dcache as well as dcache snooping are enabled, **mmu\_cache** calls the **snoop\_invalidate** function of dcache. The latter checks, whether the access is directed to a locally cached address. If the address is cached, the affected entries are invalidated.

### Instruction burst fetch

Instruction burst fetch can be enabled by setting the **IB** bit of the Cache Control Register (Table 13). In burst fetch mode the respective cache line is filled from main memory starting at the missed address until the end of the line. For this purpose the AHB master executes a burst transfer. The RTL reference model forwards the incoming instructions directly to the processor (streaming). In case of internal dependencies or multi-cycle instructions, the processor stalls until the whole line is cached. The TLM model has a slightly simplified behavour. It will always completes the burst, before sending instructions to the ISS. Burst fetch with disabled instruction cache is not supported. If the instruction cache is disabled, instructions are always fetch using single transfers (NONSEQ).

### Cache Flushing

The instruction cache and the data cache can be flushed in multiple ways. If the processor sends a **flush** instruction, both caches are flushed simultaneously. The **mmu\_cache** recognizes a flush via the **flush** payload extension. The hardware model of the **mmu\_cache** additionally provides two more input fields **flushl** and **fline**. They seems to be intended to flush certain cache lines, but are currently not used. The TLM model provides respective payload extension, to be future-proof.

A flush of only the instruction cache can be triggered by setting bit 21 (FI) of the Cache Control Register or by any write operation with ASI 0x15. Equally, the data cache may be flushed by setting bit 22 (FD) of the Cache Control Register or by any write operation with ASI 0x16.

### Freezing and Locking

The instruction cache and/or the data cache can be frozen by setting the **ICS** and**/**or the **DCS** field of the Cache Control Register to “01” (Table 13). A cache in frozen state is accessed and kept in sync with the main memory as if it was enabled, but no new lines are allocated on read misses.

Bit 31 of the two Cache Configuration Registers configures cache line locking. A cache line can be locked by setting the lock bit of a line. This can be done by a diagnostic write to the cache tag (8.1.7). Locked cache lines will be updated on read-miss and will remain in the cache until the line is unlocked.

### Diagnostic Access

Most of the internal data structures of cache and mmu can be accessed for diagnostic purpose via dedicated ASIs.

The tag and data RAMs of instruction and data cache can be read and written using ASI 0x0C – 0x0F (see section 8.1.2). Addressing and alignment of data are equivalent to the mechanism described in section 55.5.2. of RD05.

ADDRESS = WAY & LINE & DATA & “00”

### Payload Extensions

The communication between the processor and the Cache Sub-System requires additional information to be attached to the TLM 2.0 generic payload. The extensions are modeled in two classes:

*icio\_payload\_extension.h/cpp*  extensions for instruction cache socket

*dcio\_payload\_extension.h/cpp* extensions for data cache socket

Both classes declare a debug extension, which is modeled as a 32bit unsigned integer. The usage of the debug extension is explained in 8.1.8.

The **dcio** extension additionally contains fields for cache flushing, bus locking and the address space identifier (**asi**). All are represented by 32 bit unsigned integers.

The **mmu\_cache** checks all incoming transactions for the presence of the payload extensions. For data transactions this is done in function **exec\_data**, for instruction transactions in function **exec\_instr**. An error message is generated, if the extensions are not available.

### Debug Mechanism

The cache sub-system is a rather complex model. Hence, for assertion based verification, it is not sufficient to simply check whether the data response on a request is correct. It is also important to know in which way the result was produced (e.g. cache hit/miss).

For this purpose a 32bit unsigned integer extension has been attached to the generic payload of the **icio**/**dcio** sockets. A set of macros is provided for handling the debug extension. The encoding the **debug** field is shown in Table 16. The macro definitions can be found in the file **defines.h** of the **mmu\_cache** library.

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 31 22 | 21 | 20 16 | 15 | 14 | 13 | 12 | 11 4 | 3 2 | 1 0 |
| Reserved | MMUS | TLBN | Reserved | FM | CB | SP | Reserved | CST | CS |

Table 17 - Debug Extension

[MMUS| MMU state

0 – TLB hit; 1- TLB miss

[TLBN] TLB number

for TLB hit – number of the TLB that delivered the hit

for TLB miss – number of the TLB that was refilled by the miss

[FM] Frozen miss

If the cache is frozen, no new lines are allocated on a read miss. However, unvalid data will be replaced as long the tag of the line does not change. In case the results of a read miss is not cached, the FM bit is switched on.

[CB] Cache bypass

Is set to 1, if cache bypass was used (cache disabled in CCR)

[SP] Scratchpad

Is set to 1, if the request was answered by the local scratchpad RAM

[CST] Cache State

00 – read hit, 01 – read miss, 10 – write hit, 11 – write miss

[CS] Cache Set

for read/write hit – number of set which produced the hit

for read miss – number of set refilled by miss processing

for write miss – 0b00 (no allocate on write miss)

### Power Monitoring

Power monitoring can be enabled by setting constructor parameter **pow\_mon**. The IP registers with the power monitor under the name **mmu\_cache**. For the top-level of the design only one event is traced:

**idle** – Models static power consumption of the top-level. Active during the complete runtime of the simulation.

However, the largest part of the power consumption is contributed by the various sub-components: **vectorcache**, **mmu** or **localram**.

The **vectorcache** is typically instantiated twice: as a instruction cache and as a data cache. It provides an **idle** event, for modeling the static power of the whole cache and two dynamic power events per set for modeling read and write operations:

**set\_readSET** - Read access to cache set SET

**set\_writeSET** - Write access to cache set SET

All read and write transactions emit **set\_read** events for all cache sets, as long as the cache is enabled and the ASI is above 0x3 (0x0 – 0x3 mark bypass operations). This behaviour is independent of the transaction being a cache hit or cache miss. In contrast to that, the **set\_write** event is always only sent for the one particular set, which is used to bring new data to cache (cache read miss).

The **mmu** model also provides an **idle** event for modeling the static power. The dynamic power is approximated using two events:

**tlb\_lookup** – Represents the fully associative read & compare operation across the PDC. The TLBs are also read at onces and in parallel.

**tlb\_write** – Dynamic power for writing a TLB entry (after refill or miss)

Next to an idle event, for modeling the static power, the localram contributes two events to the total power of the **mmu\_cache**:

**lram\_read** – Read operation from localram

**lram\_write** – Write operation to localram

## Interface

The GRLIB VHDL model of the MMU\_CACHE is configured using Generics. For the implementation of the TLM model most of these Generics were refactored to constructor parameters of class **mmu\_cache** (Table 17). The parameters of the top-level class are used for the configuration of all sub-components (caches, localrams, mmu).

|  |  |
| --- | --- |
| Parameter | Description |
| icen | Enable instruction cache |
| irepl | Icache replacement strategy  00 = non, 01 = LRU, 10 = LRR, 11 = random |
| isets | Number of instruction cache sets (1-4) |
| ilinesize | Indicates size of instruction cache line in words (line size = 2^ilinesize, ilinesize <= 3) |
| isetsize | Indicates size (kbytes) of instruction cache set  (set size = 2^isetsize, isetsize <= 6 (max 64 kbytes)) |
| isetlock | Enable instruction cache locking |
| dcen | Enable data cache |
| drepl | Dcache replacement strategy  00 = non, 01 = LRU, 10 = LRR, 11 = random |
| dsets | Number of data cache sets (1-4) |
| dlinesize | Indicates size of data cache line in words  (line size = 2^dlinesize, dlinesize <= 3 ) |
| dsetsize | Indicates size (kbytes) of data cache set  (set size = 2^dsetsize, dsetsize <= 6 (max. 64 kbytes)) |
| dsetlock | Enable data cache locking |
| dsnoop | Enable data cache snooping |
| ilram | Enable instruction scratchpad |
| ilramsize | Indicates size of instruction scratchpad in kbytes  (size = 2^ilramsize, ilramsize <= 9 (max. 512 kbytes)) |
| ilramstart | 8 MSB bits used to decode local instruction RAM area (16 MB segm.) |
| dlram | Enable data scratchpad |
| dlramsize | Indicates size of data scratchpad in kbytes  (size = 2^dlramsize, dlramsize <= 9 (max. 512 kbytes)) |
| dlramstart | 8 MSB bits used to decode local data RAM area (16 MB segment) |
| cached | Fixed cacheability mask (overrides AMBA Plug & Play settings) |
| mmu\_en | Enable MMU |
| itlb\_num | Indicates number of instruction TLBs  (tlb number = 2^itlb\_num, itlb\_num <= 5 (max. 32)) |
| dtlb\_num | Indicates number of data TLBs  (tlb number = 2^dtlb\_num, dtlb\_num <= 5 (max. 32)) |
| tlb\_type | TLB implementation type  0 = separate, 1 = shared instruction and data TLB |
| tlb\_rep | TLB replacement policy  0 = LRU, 1 = random |
| mmupgsz | MMU page size  0, 2 = 4kbytes, 3 = 8kbytes, 4 = 16kbytes, 5 = 32kbytes |
| name | SystemC name of module |
| id | ID of the AHB bus master |
| pow\_mon | Enable power monitoring |
| abstractionLayer | Abstraction/Coding style of the model (LT or AT) |

Table - Constructor Configuration Parameters

The system-level interface of the model comprises two TLM 2.0 **simple\_target\_sockets** (**icio**, **dcio**) and one GreenSocs/Carbon AHB master socket (**ahb\_master**).

tlm\_utils::simple\_target\_socket<mmu\_cache> icio / bind to CPU instruction socket

tlm\_utils::simple\_target\_socket<mmu\_cache> dcio / bind to CPU data socket

amba::amba\_master\_socket<32> ahb\_master / bind to AMBA system bus

Depending on the constructor parameter **abstractionLayer** the sockets are configured for blocking (LT) or non-blocking (AT) communication. In the LT case the module registers one TLM blocking transport function for the **dcio** and one for the **icio** socket. In the AT case the model registers one TLM non-blocking forward transport function for the **dcio** and one for the **icio** socket, and one TLM non-blocking backward transport function for the **ahb\_master** socket. Additionally, the model comes with debug transport functions, for non-intrusive code execution (TRAP) and checking. The signatures of all transport functions are compliant with the TLM2.0 standard. Next to the TLM sockets the model contains SignalKit inputs for data bus snooping (**snoop**), clock cycle time (**clk**) and reset (**rst**). The clk and **rst** inputs are inherited from class **CLKDevice**, while **snoop** is directly defined in **mmu\_cache**.

## Internal Structure

This section describes the internal structure and the behavior of the MMU\_CACHE SystemC IP. The model consists of multiple classes, which are spread over a number of source files, all of which can be found in the **models/mmu\_cache/lib** directory.

### Files of the mmu\_cache library

#### The defines.h file

This file contains data type definitions and macros, and is included by almost all the other files of the library.

It defines the structure of the cachelines (**t\_cache\_line**), the data cache entries (**t\_cache\_data**), the cache tags (**t\_cache\_tag**), the mmu page table entries (**t\_PTE\_context**) and the virtual address tags (**t\_VAT**). Moreover, the file contains macros for handling the debug payload extension (section 8.1.8).

#### The payload\_extension files

The TLM MMU\_CACHE owns two **tlm::simple\_target\_sockets** for connection to the instruction and data sockets of the processor simulator. These connections implement a simple point-to-point communication, which can be widely realized relying on TLM 2.0 generic payload. Only a few optional payload extensions are required.

The payload extensions for the instruction cache input/output socket (**icio**) are implemented in the files **icio\_payload\_extensions.h/cpp**:

// extensions

// ----------

/// flush instruction cache

unsigned int flush;

/// flush instruction cache line

unsigned int flushl;

/// line offset in cache flush

unsigned int fline;

/// debug information  
unsigned int \* debug;

The payload extensions for the data cache input/output socket (**dcio**) are implemented in the files **dcio\_payload\_extensions.h/cpp**:

// extensions

// ----------

/// address space identifier

unsigned int asi;

/// flush data cache

unsigned int flush;

/// flush data cache line

unsigned int flushl;

/// lock cache line

unsigned int lock;

/// debug information

unsigned int \* debug;

#### The mem\_if.h file

The **mem\_if.h** file defines a generic memory interface that is directly or indirectly implemented by almost all the classes of the MMU\_CACHE (Figure 5).

The *class* **mem\_if**is an abstract class with two virtual member functions:

virtual void mem\_write(unsigned int addr, unsigned char \* data, unsigned int length, sc\_core::sc\_time \* t, unsigned int \* debug, bool is\_dbg) {};

virtual void mem\_read(unsigned int addr, unsigned char \* data, unsigned int length, sc\_core::sc\_time \* t, unsigned int \* debug, bool is\_dbg) {};

The interface is implemented by the top-level class **mmu\_cache**, the caches, the localrams and the mmu (**tlb\_adapters**). As a consequence the modules of the **mmu\_cache** library can be bound to each other like building blocks. For example, depending on the **dcen** and **mmu\_en** constructor parameters, transactions from the data socket (**dcio**) can be directed to the cache, to the mmu or to the ahb master. Transactions from the dcache or icache are directly forwarded to the ahb master or to the mmu.



Figure 6 - Generic Memory Interface / Dependencies

#### The mmu\_cache\_if.h file

The **mmu\_cache\_if** class extends the **mem\_if** class by two functions for reading and writing the Cache Control Register (CCR).

virtual unsigned int read\_ccr();

virtual void write\_ccr(unsigned char \* data, unsigned int len,   
 sc\_time \*delay, bool is\_dbg);

The CCR is implemented at the top-level of class **mmu\_cache***.* The caches and the mmu require access to the CCR at runtime. Therefore, they receive a pointer of type **mmu\_cache\_if** as a constructor argument.

#### The cache\_if.h file

The **cache\_if** class is another extension of class **mem\_if**. It describes the interface of all cache models in the system. Next to reading or writing data (**mem\_if**), caches must allow to flush data, to read/write cache tags/entries, to access configuration registers and to handle snooping:

/// flush cache

virtual void flush(sc\_core::sc\_time \* t, unsigned int \* debug, bool is\_dbg) = 0;

/// read data cache tags (ASI 0xe)

virtual void read\_cache\_tag(unsigned int address, unsigned int \* data,

sc\_core::sc\_time \*t) = 0;

/// write data cache tags (ASI 0xe)

virtual void write\_cache\_tag(unsigned int address, unsigned int \* data,

sc\_core::sc\_time \*t) = 0;

/// read data cache entries/data (ASI 0xf)

virtual void read\_cache\_entry(unsigned int address, unsigned int \* data,

sc\_core::sc\_time \*t) = 0;

/// write data cache entries/data (ASI 0xf)

virtual void write\_cache\_entry(unsigned int address, unsigned int \* data,

sc\_core::sc\_time \*t) = 0;

/// read cache configuration register (ASI 0x2)

virtual unsigned int read\_config\_reg(sc\_core::sc\_time \*t) = 0;

/// returns the mode bits of the cache

virtual unsigned int check\_mode() = 0;

/// Snooping function (invalidates cache line(s))

virtual void snoop\_invalidate(const t\_snoop &snoop,

const sc\_core::sc\_time& delay) = 0;

/// Helper functions for definition of clock cycle

virtual void clkcng(sc\_core::sc\_time &clk) = 0;

/// Display of cache lines for debug

virtual void dbg\_out(unsigned int line) = 0;

Next to the data cache (**dvectorcache**) and the instruction cache (**ivectorcache**), the interface is implemented by the plain structural module **nocache**. In case one of the caches is not present in the system (disabled via **i/dcen** constructor parameters), the **mmu\_cache** binds one or two instances of **nocache**. The nocache class implements stubs for all **cache\_if** functions. Forbidden operations generate an error message.

#### The mmu\_cache.h/cpp files

The files declare and implement the top-level class of the MMU\_CACHE. The class **mmu\_cache** implements the **mmu\_cache\_if** interface and instantiates all sub-modules depending on the selected configuration. All sub-components are dynamically created in the constructor of the class. The instantiation depends on parametrization options (see **Fehler! Verweisquelle konnte nicht gefunden werden.**). In case a certain module is not required, a NULL pointer will be assigned. If the mmu is enabled, the caches use the memory interfaces (**mem\_if**) of the instruction **tlb\_adapters** and data **tlb\_adapters** for miss processing, otherwise they are directly connect to the ahb master.

The following pointers provide access to the APIs of all subordinate components:

**ivectorcache\* icache**- instruction cache pointer

**dvectorcache\* dcache**- data cache pointer

**mmu\* m\_mmu**- memory management unit

**localram\* ilocalram** - instruction scratchpad

**localram\* dlocalram**- data scratchpad

All sub-components are implemented in plain C++, for highest possible simulation speed.

MMU\_CACHE also inherits from class **AHBDevice** and class **CLKDevice**. From **AHBDevice mmu\_cache** receives a PNP configuration record for identification as an AHB master. Class **CLKDevice** provides an unified interface for clock and reset distribution, that is shared with most of the components in the SoCRocket library. The timing information received via the **clk** SignalKit input is distributed to all sub-components by function **clkcng**.

As the top-level class, **mmu\_cache** implements the interface to the outside TLM world. Next to a GreenSocs/Carbon AHB master socket (**ahb\_master**), the class contains two TLM 2.0 **simple\_target\_sockets** for connection to the instruction and data ports of the processor simulator (Table 18).

|  |  |  |
| --- | --- | --- |
| Name | Type | Description |
| icio | TLM2 / simple\_target\_socket (LT) | Instruction cache in/out |
| dcio | TLM2 / simple\_target\_socket (LT) | Data cache in/out |
| ahb\_master | GreenSocs / amba\_master\_socket (LT) | AHB bus master |

Table 19 - TLM Sockets Cache Sub-System

With respect to the TLM 2.0 standard the TLM interface of the **mmu\_cache** supports two levels of accuracy: loosely timed (LT) and approximately timed (AT). The abstraction level can be selected via the **abstractionLayer** constructor parameter. The LT interface is described in 8.3.2 and the AT interface in 8.3.3. Both interfaces map the incoming transactions to functions that encapsulate the behaviour of the model. Instruction transactions invoke **exec\_instr** and data transactions **exec\_data**.

The **exec\_instr** function is very compact. After extracting the payload object and verifying the extension, the **tlm\_command** attribute is checked. TLM read commands are translated into calls to the **read** (**mem\_if)** function of the **icache** or **ilocalram**. Because the instruction cache is read only, TLM write requests cause a TLM\_COMMAND\_ERROR\_RESPONSE and an error message to be printed on the screen.

The **exec\_data** function is more deeply structured. The main reason is the decoder for the address space identifiers (ASIs – see 8.1.2). The ASI is implemented as a mandatory extension to the TLM 2.0 generic payload. Depending on the ASI the **exec\_data** function maps the incoming transactions to the APIs of the different sub-components. Default cache access is performed for ASIs 0x8, 0x9, 0xa and 0xb. Other modes are used to access system registers (0x2), tag rams (0xc, 0xe), cache data blocks (0xd, 0xf), mmu internal registers (ASI 0x19) and more. For every transaction the payload extensions are checked. TLM read commands are translated into calls to the **read** (**mem\_if**) function of the **dcache** or the **dlocalram**. If the **dcache** is disabled the transactions are forwarded to the **mmu** or to the **ahb\_master** socket.

The class **mmu\_cache** also contains the Cache Control Register (CCR) and its access functions **read\_ccr** and **write\_ccr** (see Table 13).

#### The vectorcache.h/cpp files

The files **vectorcache.h** and **vectorcache.cpp** form the base class for the implementation of the instruction cache (ivectorcache) and the data cache (dvectorcache). Class vectorcache implements the **cache\_if** API and provides almost all the functionality required by both caches. In the following these functions are briefly described:

/// read from cache  
void read(unsigned int address, unsigned char \* data, unsigned int len, sc\_core::sc\_time \* t, unsigned int \* debug, bool is\_dbg);

The **read** (**mem\_if**) function is called for any type of load operation (byte, short, word, dword). The length of the access in bytes is given by the **len** parameter. The **address** is split into a cache tag and a cache index portion. The respective line is loaded from all sets and compared against the index. If one of the tags equals the index and the valid bit is set, the cache entry is copied to the **\*data** pointer (read hit). In case the tags do not match or the valid bit is not set, the request is forwarded to the ahb interface or to the mmu (read miss). After miss processing, the fresh data is filled into the cache and copied to the **\*data** pointer.

The **is\_dbg** flag signals that the read function was called in a TLM debug transport.

/// write through cache  
void write(unsigned int address, unsigned char \* data, unsigned int len, sc\_core::sc\_time \* t, unsigned int \* debug, bool is\_dbg);

The **write** (**mem\_if**) function is called for any type of store operation (byte, short, word, dword). The length of the access in bytes is given by the **len** parameter. The address is split into a cache tag and a cache index portion. The respective line is loaded from all sets and compared against the index. If one of the tags equals the index and the valid bit is set, the respective data entry is updated and the request is forwarded to the mmu or the ahb interface (write hit). If the tag does not match or the valid bit is not set the request directly goes to mmu or ahb interface (write miss). The cache will not be updated on a write miss. The write policy is write-through with no-allocate on write miss.

/// flush cache  
void flush(sc\_core::sc\_time \* t, unsigned int \* debug);

Flushes the cache*.* During a cache flush all valid data in the cache is transferred to main memory for synchronization.

/// read data cache tags (ASI 0xe)

void read\_cache\_tag(unsigned int address, unsigned int \* data, sc\_time \*t);

/// write data cache tags (ASI 0xe)

void write\_cache\_tag(unsigned int address, unsigned int \* data, sc\_time \*t);

/// read data cache entries/data (ASI 0xf)

void read\_cache\_entry(unsigned int address, unsigned int \* data, sc\_core::sc\_time \*t);

/// write data cache entries/data (ASI 0xf)

void write\_cache\_entry(unsigned int address, unsigned int \* data, sc\_time \*t);

These functions are used for diagnostic access to cache tags and cache entries (see 8.1.6).

unsigned int read\_config\_reg(sc\_core::sc\_time \*t);

Returns the configuration register of the cache. The Cache Configuration Register is initialized in the constructor of class **mmu\_cache**. The register is read only.

virtual unsigned int check\_mode() = 0;

A cache can be in one of three different modes of operation: enabled, disabled or frozen. The current mode can be deterimend by checking the Cache Control Register, which is implemented in the top-level class **mmu\_cache**. Depending on the type of cache (instruction or data) the DCS or ICS bits of the CCR must be checked. Therefore, the **check\_mode** function is plain virtual. The function must be overwritten by the actual **icache** or **dcache** implementation.

#### The ivectorcache.h/cpp files

The class **ivectorcache** contains the actual implementation of the instruction cache. The class inherits from class **vectorcache**. The write function is overwritten, because the instruction cache is not writable. A call to the write function produces an error message and stops the simulation.

The class implements the virtual function **check\_mode**. For checking the mode of operation the ICS bits of the Cache Control Register are used.

#### The dvectorcache.h/cpp files

The class **dvectorcache** contains the actual implementation of the data cache. The class inherits from class **vectorcache**.

The virtual **check\_mode** function is implemented. For checking the mode of operation the DCS bits of the Cache Control Register are used.

#### The localram.h/cpp files

The class **localram** models a fast scratchpad memory that can be attached to both instruction and data cache controllers. It implements the generic memory interface **mem\_if**. The actual memory is implemented as a character array (**scratchpad**).

#### The mmu.h/cpp files

The files implement the memory management unit of the MMU\_CACHE. The component was modeled following the recommendations for the SparcV8 reference MMU given in [RD08]. The class **mmu** receives the number of instruction tlbs, the number of data tlbs, the tlb type, the tlb replacement policy and the mmu page size as constructor arguments. Depending on the tlb type two split TLBs or one shared TLB is generated for instructions and data. The TLBs are implemented as a **std::map**. The key for a TLB lookup is a virtual address tag (**t\_VAT**). The caches connect to the mmu through **tlb\_adapter** objects (section 8.3.1.12). In shared TLB mode only one adapter is generated. Next to the adapter objects the class **mmu** offers a set of API functions. The most important of these functions is:

unsigned int tlb\_lookup(unsigned int addr, std::map<t\_VAT, t\_PTE\_context> \* tlb, unsigned int tlb\_size, sc\_core::sc\_time \* t, unsigned int \* debug);

The **tlb\_lookup** function is responsible for translating virtual addresses into physical addresses. It receives the virtual address and a TLB pointer as input arguments. In the body of the function the virtual address is split into three indices. The bit width of these indices depend on the virtual page size. The following page sizes and index combinations are supported (Table 19):

|  |  |  |  |
| --- | --- | --- | --- |
| virt. page size | idx 1 | idx 2 | idx 3 |
| 4kb | 8 bit | 6 bit | 6 bit |
| 8kb | 7 bit | 6 bit | 6 bit |
| 16kb | 6 bit | 6 bit | 6 bit |
| 32kb | 4 bit | 7 bit | 6 bit |

Table 20 - Page size / index combinations

In case of a TLB miss the indices are used for addressing the page tables in main memory. A successful read of a page table returns either a page table descriptor (PTD) or a page table entry (PTE). A PDC is a pointer to the next-level page table, while a PTE corresponds to an actual TLB entry. Up to three page table levels are supported.

The **mmu** contains a set of internal control registers. These registers can be accessed through ASI 0x19 (Table 15). Respectivly, read and write requests are translated into calls to following functions:

// read mmu internal registers (ASI 0x19)

unsigned int read\_mcr();

unsigned int read\_mctpr();

unsigned int read\_mctxr();

unsigned int read\_mfsr();

unsigned int read\_mfar();

// write mmu internal registers (ASI 0x19)

void write\_mcr(unsigned int \* data);

void write\_mctpr(unsigned int \* data);

void write\_mctxr(unsigned int \* data);

Another group of member functions is dedicated to diagnostic TLB access. The addressing of the different bit fields can be taken from [RD08].

// diagnostic read/write of instruction PDC (ASI 0x5)

void diag\_read\_itlb(unsigned int addr, unsigned int \* data);

void diag\_write\_itlb(unsigned int addr, unsigned int \* data);

// diagno. read/write of data PDC or shared instruction and data PDC (ASI 0x6)

void diag\_read\_dctlb(unsigned int addr, unsigned int \* data);  
void diag\_write\_dctlb(unsigned int addr, unsigned int \* data);

#### The tlb\_adaptor.h file

The class **tlb\_adapter** implements the generic memory interface **mem\_if**. Depending on the configuration the **mmu** creates one or two objects of type **tlb\_adapter**, which provide access to the instruction and/or data tlb. Pointers to these objects can be obtained by calling the mmu API functions **get\_itlb\_if** and **get\_dtlb\_if**.

### LT Behaviour

The LT mode of the MMU\_CACHE is intended for fast register accurate simulation (programmers view).

#### Instruction transactions

For instruction fetch the model provides the **icio** **simple\_target\_socket**. In LT mode this socket is bound to the **icio\_b\_transport** blocking transport function. Incoming transactions are directly forwarded to the **exec\_instr** function (8.3.1.6), which models the functional interface of the **mmu\_cache** IP. Depending on the configuration **exec\_instr** performs a lookup of the instruction cache or loads data from the instruction scratchpad. Cache misses or bypass operations create a transaction on the **ahb\_master** socket. If **mmu\_en** is set all addresses are considered virtual and will be translated to physical addresses by the mmu. In the meantime the processor is blocked. The **exec\_instr** function returns the accumulated delay of all involved sub-components. Before unblocking the master the **icio\_b\_transport** function calls wait to consume the component delay.

#### Data transactions

For data load/store the model provides the **dcio simple\_target\_socket**. In LT mode this socket is bound to the **dcio\_b\_transport** blocking transport function. Similar to instruction fetch, incoming transactions are directly forwarded to a function encapsulating the behaviour of the data cache. Depending on the configuration and the settings contained in the payload extensions the **exec\_data** function performs a lookup of the data cache, loads/store of the instruction or data scratchpad or read/writes of internal registers. Cache misses or bypass operations create a transaction on the **ahb\_master** socket. If **mmu\_en** is set all addresses are considered virtual and will be translated to physical addresses by the mmu. In the meantime the processor is blocked. The **exec\_data** function returns the accumulated delay of all involved sub-components. Before unblocking the master the **dcio\_b\_transport** function calls wait to consume the component delay.

### AT Behaviour

The AT mode of the MMU\_CACHE is intended for architecture exploration and RTL co-simulation. It contains multiple parallel threads, which are not present in LT mode.

#### Instruction transactions

Instruction transactions arrive in the **icio\_nb\_transport\_fw** function with phase **BEGIN\_REQ**. The function enters the transaction in the **icio\_PEQ** payload event queue and returns to the master with **END\_REQ** and **TLM\_UPDATED**. The SC\_THREAD **icio\_service\_thread** is sensitive to the default event of **icio\_PEQ**. It invokes the **exec\_instr** function for every transaction from the queue. As already mentioned, **exec\_instr** encapsulates the functional part of the model. Within **exec\_instr** the payload is processed in the same way as described for LT mode. After return from **exec\_instr** the **icio\_service\_thread** consumes the accumulated delay of all involved mmu\_cache sub-components. Afterwards, the master is notified by sending **BEGIN\_RESP** on the backward path. The master may reply with **TLM\_COMPLETED** or **TLM\_ACCEPTED**. A final **END\_RESP** from the master will be accepted, but is not required.

#### Data transactions

In AT mode the constructor of **mmu\_cache** registers a non-blocking transport function at the **dcio** **simple\_target\_socket** (**dcio\_nb\_transport\_fw**). The **dcio\_nb\_transport\_fw** function is called at every phase change of a data transaction. New transactions arrive with phase **BEGIN\_REQ**. The transport function enters the transaction in the **dcio\_PEQ** payload event queue and returns to the master with **END\_REQ** and **TLM\_UPDATED**. The **dcio\_PEQ** is used to forward the transaction to the SC\_THREAD **dcio\_service\_thread**. It invokes the **exec\_data** function for every transaction from the queue. Within **exec\_data** the payload is processed in the same way as described for LT mode. After return from **exec\_data** the **dcio\_service\_thread** consumes the accumulated delay of all involved **mmu\_cache** sub-components. Afterwards, the master is notified by sending **BEGIN\_RESP** on the backward path. The master may reply with **TLM\_COMPLETED** or **TLM\_ACCEPTED**. Similar to instruction transactions, a final **END\_RESP** from the master will be accepted, but is not required.

## The AHB master

Class **mmu\_cache** implements the **mem\_if** memory interface, to provide access to the **ahb\_master** socket, for all modules of the library.

For read transaction this invokes function **mem\_read**. Every call to mem\_read creates a new payload object. The payload is taken from the transaction pool provided by the GreenSocs/Carbon ahb socket. Target address and payload pointer of the original transaction are copied. Next to the default payload attributes, the function initializes a set of ahb specific extensions:

**amba::amba\_burst\_size** - Relates to the streaming width of the AHB bus.  
 The actual size of a burst (in bytes) is given by the length parameter.

**amba::amba\_id** - The AHB master id of the module.

**amba::amba\_trans\_type** - AHB transfer type extension. Since all transfers are modeled in a single transaction trans\_type is always NON\_SEQUENTIAL.

After setting up the payload the **mem\_read** function checks the **is\_dbg** flag and the **abstractionLayer** parameter. In debug mode the transaction is send using the untimed TLM debug transport interface:

**ahb->transport\_dbg(\*trans)**

In LT mode mem\_read invokes a blocking transport:

**ahb->b\_transport(\*trans, delay)**

After returning from **b\_transport** the model synchronizes with the SystemC scheduler by calling **wait**. This consumes the accumulated delay of the AHB transfer.

In AT mode the bus transfer is modeled using multiple phases. This requires a non-blocking backward transport function (**ahb\_nb\_transport\_bw**) to be bound to the **ahb\_master** socket and a number of SC\_THREADs. If **mem\_read** is called in AT mode the AHB transfer is initialized by sending **BEGIN\_REQ** on the forward path:

**ahb->nb\_transport\_fw(\*trans, phase, delay);**

In the AHBCTRL this causes the transaction to be scheduled for arbitration. The bus model will reply with TLM\_ACCEPTED. The signal for successful arbitration is END\_REQ being received on the backward path. At the time of END\_REQ the ahb\_nb\_transport\_bw function notifies the mEndRequestEvent, which unblocks the mem\_read function. For read operations END\_REQ is directly followed by BEGIN\_RESP. A BEGIN\_RESP from the AHBCTRL triggers the ResponseThread (via mResponsePEQ). The ResponseThread is responsible for sending END\_RESP to the AHBCTRL. Moreover, it forwards the transaction to the cleanUP thread. The latter returns the transaction to the memory pool with a delay of 100 **clock\_cycles**. The additional lifetime of the transaction guarantees that the data pointer can be savely copied by the master.

Write transactions are processed in a very similar way. Modules writing to the **ahb\_master** socket use the **mem\_write** (**mem\_if**) interface function. The **mem\_write** function obtains a payload object from the memory pool and initializes all data members including the mentioned ahb specific extensions. The **mem\_write** function also distinguishes between debug, blocking (LT) and non-blocking (AT) communication. While debug and blocking communication are trivial, the non-blocking communication differs from the standard TLM protocol. This is due to the pipelined nature of AHB.

AHB communication is split into two phases: address and data. RTL slaves sample the address at the first clock edge and the data at the second. The data phase of the first transaction equals the address phase of the second (succeeding) one. Especially for write transactions from RTL masters to TLM slaves, the TLM standard protocol is insufficient. The slave can never know, when the data pointer of a transaction becomes valid. Therefore, the **BEGIN\_RESP** phase of the standard protocol has been replaced by phase **BEGIN\_DATA**, which is directed from the master to the slave. The **END\_RESP** phase is replaced by phase **END\_DATA**. **END\_DATA** is send by the slave and indicates the end of a write operation.

The **mem\_write** function initiates a bus transfer by sending BEGIN\_REQ on the forward path. Equal to read operations the AHBCTRL will reply with END\_REQ (backward path), as soon the master has won arbitration. After receiving END\_REQ the **ahb\_nb\_transport\_bw** function notifies the **mEndRequestEven**t. Moreover, the transaction is forwarded to SC\_THREAD **DataThread** (via **mDataPEQ**). The **DataThread** sends **BEGIN\_DATA** to the AHBCTRL. As soon as the bus has sent **END\_DATA** (via backward or return path), the transaction is considered complete. To make sure all pointers can been properly saved, the payload is returned to the memory pool with a delay of 100 **clock\_cycles** (**mEndTransactionPEQ**).

## Compilation

For the compilation of the MMU\_CACHE IP, a WAF wscript is provided and integrated in the superordinate build mechanism of the library.

All required objects for simulating the MMU\_CACHE on platform level are compiled in a sub-library named **mmu\_cache** using following command:

**./waf –target=mmu\_cache**

To utilize **mmu\_cache** in simulations with other components, add **mmu\_cache** to the use list of your wscript.

## Example Instantiation

The example below demonstrates the instantiation of the MMU\_CACHE inside an **sc\_main** or an arbitrary top-level class. The instantiating module needs to include at least **mmu\_cache.h** and **amba.h**. The MMU\_CACHE is created in line 3 – 32. In lines 39 – 40 the **icio** and **dcio** slave sockets are bound to the master sockets of the testbench (or processor). The ahb master port is bound to the AHBCTRL in line 43. Line 46 shows how to connect the snooping. Since MMU\_CACHE has internal storage, it needs a notion of time. In this example the clock cycle time is set in line 49.

1 // CREATE MMU Cache

2 // ----------------

3 mmu\_cache mmu\_cache(1, // int icen = 1 (icache enabled)

4 2, // int irepl = 2 (icache random replacement)

5 4, // int isets = 4 (4 instruction cache sets)

6 4, // int ilinesize = 4 (4 words per icache line)

7 1, // int isetsize = 1 (1kB per icache set)

8 0, // int isetlock = 0 (no icache locking)

9 1, // int dcen = 1 (dcache enabled)

10 2, // int drepl = 2 (dcache random replacement)

11 4, // int dsets = 4 (4 data cache sets)

12 4, // int dlinesize = 4 (4 words per dcache line)

13 1, // int dsetsize = 1 (1kB per dcache set)

14 0, // int dsetlock = 0 (no dcache locking)

15 0, // int dsnoop = 0 (no cache snooping)

16 0, // int ilram = 0 (instr. localram disable)

17 1, // int ilramsize = 1 (1kB ilram size - disabled)

18 0x0000008e, // int ilramstart = 8e (0x8e000000 default ilram start address)

19 0, // int dlram = 0 (data localram disable)

20 1, // int dlramsize = 1 (1kB dlram size - disabled)

21 0x0000008f, // int dlramstart = 8f (0x8f000000 default dlram start address)

22 0xffff, // int cached = 0 (fixed cacheability mask)

23 0, // int mmu\_en = 0 (mmu not present)

24 8, // int itlb\_num = 8 (8 itlbs - not present)

25 8, // int dtlb\_num = 8 (8 dtlbs - not present)

26 0, // int tlb\_type = 0 (split tlb mode - not present)

27 1, // int tlb\_rep = 1 (random replacement)

28 0, // int mmupgsz = 0 (4kB mmu page size)>

29 **"mmu\_cache"**, // name of sysc module

30 0, // id of the AHB master

31 0, // bool pow\_mon = 1 (disable power monitoring)

32 amba::amba\_LT); // select LT or AT abstraction

33

34 ...

35

36 // \*\*\* BIND SOCKETS

37

38 // Connect testbench (cpu) to mmu-cache

39 tbm.icio(mmu\_cache.icio);

40 tbm.dcio(mmu\_cache.dcio);

41

42 // Connect mmu\_cache to TLM bus

43 mmu\_cache.ahb(ahbctrl.ahbIN);

44

45 // Connect snooping

46 ahbctrl.snoop(mmu\_cache.snoop);

47

48 // Set timing (clock cycle)

49 mmu\_cache.set\_clk(10, SC\_NS);

50

# GPTIMER General Purpose Timer SystemC model

## Functionality and Features

### Overview

The **GPTimer** unit acts as a slave at the APB bus. The module inherits from class **APBDevice** and exports its configuration in from of a GRLIB PNP record. Its basic functionality is a countdown mechanism that asserts an interrupt on underflow. The GPTimer consists of a prescaler unit that is generating ticks and up to seven counter units that are decrementing on prescaler ticks. In the VHDL model, the counter units are named ‘timers’ just like the entire IP model. As this is a potential source of confusion, the name has been changed to ‘counters’ in the TLM implementation.

The prescaler and all the counters are equipped with a value register and a reload value register. The value register is decremented on each trigger and can be reset to the reload value on underflow or by a reset command. In the VHDL model, the trigger for decrementing the prescaler is the bus clock input of the GPTimer unit. In the SystemC model the prescaler ticks are calculated by multiplying the clock period with the prescaler reload register. To provide a notion of time class **GPTimer** inherits the timing interface of class **CLKDevice**, which is shared amongst all components of the SoCRocket library.

### Control Registers

The **GPTimer** unit can be configured and operated through a set of internal registers, which are bound to the APB interface. The registers are implemented as a GreenReg register bank. Respectively, class **GPTimer** is derived from class **gr\_device** (3.4). All registers have a width of 32 bits and are summarized in Table 20.

|  |  |
| --- | --- |
| APB Address Offset | Register |
| 0x00 | Scaler Value |
| 0x04 | Scaler Reload Value |
| 0x08 | Configuration Register |
| 0x0C | Unused |
| 0xn0 | Counter n Value Register |
| 0cn4 | Counter n Reload Register |
| 0xn8 | Counter n Configuration Register |
| 0xnC | Unused |

Table 21 – GPTimer Registers

The timer configuration register located at address 0x08 can be used to configure the **GPTimer** unit. It consists of four fields: **DF**, **SI**, **IRQ**, and **TIMERS** (Table 21). The **DF** field is the only field that can be modified dynamically, all other fields are read only, i.e. their values are determined by constructor parameters and written to the registers at system startup.

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 31 |  |  |  |  | 10 | 9 | 8 | 7 3 | 2 0 |
| Reserved | | | | | | DF | SI | IRQ | TIMERS |

Table - GPTimer Configuration Register

[DF] In the VHDL model the Disable Freeze field disables the sensitivity to the dhalt input signal. If DF is disabled the dhalt signal can be used to freeze the timer value registers. At TLM-level the dhalt input is needless. Therefore, also DF has no practical effect.

[SI] The Separate Interrupt field specifies whether each counter asserts an individual interrupt line or all counters assert the same interrupt line. If all counters assert the same interrupt line, this irq number is defined by the IRQ field. Otherwise, only the first counter uses the interrupt IRQ. The interrupts of the remaining counters are asserted to the subsequent lines (ascending order).

[IRQ] Default IRQ number as decribed above (see [SI]).

[TIMERS] The TIMERS field specifies the number of counters in the system.

The counter configuration registers are used to configure and control the up to 7 counters of the timer (Table 22). They can be found at address **0xn8**, with **n** being the index of the counter.

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 31 |  |  | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| Reserved | | | | DH | CH | IP | IE | LD | RS | EN |

Table - GPTimer Counter Configuration Register

[DH] Debug Halt: Used by the VHDL model to freeze the counters in debug mode. At TLM level the system can be halted anytime. Therfore, the DH signals is ignored.

[CH] Chain with preceeding timer (see 9.1.3)

[IP] Interrupt Pending: Set for signalling an interrupt. Remains ‘1’ until cleared.

[IE] Interrupt Enable: If set the timer sends an interrupt for each underflow.

[LD] Load Value from the timer reload register to the timer counter value register

[RS] ReStart: If set, the timer counter value register is reloaded with the value of the reload register when the timer underflows.

[EN] Enables the timer

### Chaining

Chaining can be activated for one or multiple counters to increase the counting delay. This can be done by setting the **CH** bit of the appropriate counter control registers.

If counter **n** is in chaining mode, it does not decrement on prescaler ticks, but on ticks generated by an underflow of the previous counter (**n-1**). For this mode of operation, the **RS** bit of counter (**n-1**) must be set. This causes the value register of counter (n-1) to be automatically reloaded from the reload value register (on underflow).

Chained counters are still producing interrupts on underflow. Usually this behaviour is not desired. To disable interrupt generation of chained counter reset the **IE** bits of all affected counter control registers.

### Watchdog

The last counter of the timer can be used as a watchdog. This feature can be enabled by setting the **wdog** constructor parameter to an alternative reload value, which will be used to set and reload the counter.

The watchdog is started directly after the reset of the timer. On underflow it asserts a interrupt to the wdog output.

### Power Modeling

Power monitoring can be enabled by setting the constructor parameter **pow\_mon**. The IP registers with the power monitor with name **gptimer**. Moreover, each counter of the timer is registered as an independent sub-module (name: **gpcounter**). Timer and counters provide an **idle** event for modeling static power:

*idle* – Active during the complete runtime of the simulation (static power)

Moreover, the counter contains two dynamic events:

*active* – Signals that the counter is enabled and counting

*underflow* – Counter underflow. An interrupt has been emitted.

## Interface

The GRLIB VHDL model of the GPTIMER is configured using Generics. For the implementation of the TLM model most of these Generics were refactored to constructor parameters of class **GPtimer**. An overview about the available parameters is given in Table 23.

|  |  |
| --- | --- |
| Parameter | Description |
| name | The name of the SystemC instance |
| ncounters | Number of counters (1-7) |
| pirq | Defines which APB interrupt the timers will generate. |
| sepirq | If set to 1, each timer drives an individual interrupt line, starting with interrupt pirq. If set to 0, all timers will drive the same interrupt line. |
| nbits | Bitwidth of the counters |
| sbits | Bitwidth of prescaler |
| wdog | Watchdog reset value. |
| pow\_mon | Enable power monitoring |

Table - GPTimer Parameters

The system-level interface of the module comprises an GreenSocs/Carbon APB slave socket and multiple SocRocket SignalKit ports.

|  |  |  |  |
| --- | --- | --- | --- |
| Name | Type | In/Out | Description |
| rst | bool | in | Reset prescaler and all counters |
| clk | sc\_time | in | Annotates clock period |
| irq | uint32\_t | out | Muxed interrupt lines |
| wdog | bool | out | Watchdog |

Table - Timer SignalKit sockets

## Internal Structure

The TLM implementation of the GPTimer comprises two classes, **GPTimer** and **GPCounter**. They are described in the files **gptimer.h/cpp** and **gpcounter.h/cpp**. Implementing the counter unit in a class of its own enables GPTimer to dynamically create up to seven counters.

### Configuration of the module

The **GPTimer** unit is configurable through its timer configuration register (Table 21) and its counter configuration registers (Table 22). Both are implemented using GreenReg, along the other memory mapped registers described in section 9.1.2.. For this reason the module is derived from class **gr\_device**. A **gr\_device** is a top-level encapsulation for a complete functional unit and provides containment structures for other GreenReg elements, e.g. registers.

The **gptimer.h** file contains const variables defining register addresses and bit masks. These definitions are made for programming convenience.

The write masks of the registers can be used to ensure that only permitted bits are set when writing to a register. They can also be applied for reading specific fields of a register masking all other bits.

### Operation of the module

The **GPTimer** class definition contains the module interface and the function prototypes of constructor, destructor, SystemC processes, callback functions, and pure C++ software routines. **GPTimer** requires a notion of time to assert its interrupts with correct order and latency. Therefore, each **GPCounter** sub-component contains an SC\_THREAD **ticking**. The **ticking** thread calculates the timespan till the next timer tick using the function **calculate**. Afterwards, the thread suspends and does not wake up before underflow.

The class **GPTimer** comprises one additional thread SC\_THREAD for debugging. It provides detailed information about the state of the timer for every time step (**clock\_cycle**). By default this thread is disabled (macro DEBUG).

For keeping track of the overall state of operation the following variables may be used:

**lasttime** - Holds the timestamp for the current prescaler value. Is required as a reference for calculating ticks.

**lastvalue** - Holds the content of the prescaler value register at the time stored in lasttime. The prescaler value is known at the time it is calculated.

The remaining behaviour of the model is straight forward. Details about registration and handling of GreenReg register and there respective callback functions can be found in 3.4.

## Compilation

For the compilation of the **GPTimer** IP, a WAF wscript is provided and integrated in the superordinate build mechanism of the libarary.

All required objects for simulating the GPTimer are compiled in a sub-library named gptimer using following command:

**./waf –target=gptimer**

To utilize the **GPTimer** in simulations with other components, add gptimer to the use list of your wscript.

## Example Instantiation

The example below demonstrates the instantiation of the **GPTimer** in an **sc\_main** or an arbitraty top-level class. The module is created in line 2. Line 5 shows how to connect the **GPTimer** to the APBCTRL. Lines 8-10 connect the SignalKit ports of the **GPTimer** to a **testbench** representing processor and interrupt controller. The timing is annotated in line 13, by setting the clock cycle-time to 10 ns.

 1 // Instantiate GPTimer with 4 counters and IRQ 2

 2 GPTimer timer(**"timer"**, 4, 0, 0, 0xfff, 2, 0, 16, 32, 0);  
 3

4 // Bind APB port (bus) to APBCTRL  
 5 apbctrl.apb(timer.bus)

6

 7 // Connect SignalKit ports to testbench

 8 connect(testbench.rst, timer.rst);  
 9 connect(testbench.irq, timer.irq);

10 connect(testbench.wdog, timer.wdog);

11

12 // Set clock period to 10 ns  
13 timer.set\_clk(10, SC\_NS)

# IRQMP Interrupt Controller SystemC model

## Functionality and Features

### Overview

The SystemC IRQMP unit models behaviour and timing of the IRQMP VHDL model from the Aeroflex/Gaisler GRLIB (RD04). Purpose of the IP is the priorization and masking of the interrupts from all AHB and APB devices in the system. The interrupt with the highest priority is propagated to one or multiple processors. Up to 16 LEON3 cores are supported. Two different modes of IR distribution are implemented:

* The IR is forwarded to all cores and cleared by the first core that acknowledges the IR (i.e. the ISR is processed only once).
* The IR is broadcasted and has to be acknowledged (and processed) by each of the cores.

Interrupts can be masked for each core separately. The data path or the IRQMP unit is not pipelined, i.e. all operations can be performed within one clock cycle.

The GRLIB interrupt scheme comprises a 32-bit interrupt (IR) bus, which is routed in parallel to the AMBA bus signals. The 16 LSBs of the IR bus are associated to regular IRs and the 16 MSBs to extended IRs (EIR). In the SoCRocket library IRs are modeled using the TLM SignalKit, which has been described in chapter 3.3 of this document. Interrupts from any simulation model can be bound to the IRQMP using the connect method.

The following command connects the GPTimer interrupt output number 3 (**SignalKit::selector**) with the IRQMP interrupt input number 5.:

**connect(gptimer.irq, irqmp.irq\_in, 3, 5);**

In case the sending device has only one interrupt output (**SignalKit::out**) only one channel number (for the IRQMP) must be defined:

**connect(socwire.irq, irqmp.irq\_in, 6);**

The connection of the IRQMP towards the processors is implemented in very similar way.

### Control Registers

The IRQMP can be configured and controlled by a set of memory mapped registers (Table 25). All control registers are 32 bit wide and implemented in form of a GreenReg register bank. Therefore, as described in 3.4, class **irqmp** is a chield of class **gr\_device**. An overview about the available registers is given in Table 25.

|  |  |
| --- | --- |
| APB Address Offset | Register |
| 0x00 | Interrupt Level Register |
| 0x04 | Interrupt Pending Register |
| 0x08 | Interrupt Force Register (NCPU = 0) |
| 0x0C | Interrupt Clear Register |
| 0x10 | Multiprocessor Status Register |
| 0c14 | Broadcast Register |
| 0x40 + 4 \* n | Processor n Interrupt Mask Register |
| 0x80 + 4 \* n | Processor n Interrupt Force Register |
| 0xC0 + 4 \* n | Processor n Extended Interrupt Identification Register |

Table 26 – IRQMP Registers

All registers can be written to configure or operate the IRQMP unit. Only the **Extended Interrupt Identification Register** is read-only. The function and configuration options of the registers are described in full detail in section 54.3 of RD04. However, two differences between RD04 and the SystemC implementation have to be noted:

1. The **Interrupt Force Register** for NCPU = 0 has been left out in the SystemC implementation. In a single-processor system the function of the **Interrupt Force Register** is identical to that of the **Interrupt Pending Register**.
2. In RD04 it is stated that the bits [31..17] of the **Interrupt Clear Register** are all constantly pulled down to ‘0’. This differs from the VHDL implementation, in which these bits are used for extended interrupt clearance. Respectively, an EIR can also be cleared by software. The SystemC implementation follows the VHDL implementation rather than the manual.

### Interrupt Prioritization and Forwarding

The IRs are prioritized in a two-dimensional prioritization scheme. Both dimensions are referred to as “interrupt level” in RD04. For clarification purposes, terms will be redefined for this document.

The first dimension of prioritization is determined by the Interrupt Level Register. For each IR line, the according bit in the IR Level Register can be set to level 0 or level 1. Each level 1 IR has got a higher priority than any level 0 IR. The first dimension of prioritization will be referred to as “interrupt level” throughout this document.

The 16 regular IR lines are modeled with a 16-bit vector. The most significant bit (IR15) has got the highest priority and IR1 has got the lowest priority. IR0 is reserved. This second dimension of prioritization will be referred to as “interrupt line” throughout this document.

When several IRs are pending, the highest priority IR will be calculated according to the scheme described above. Which cores receives the interrupt request (IRQ) depends on the settings in the **Broadcast Register** and the **Interrupt Mask Registers** of the individual cores. As shown in Figure 6, the use of the **IR Pending** or **IR Force Registers** is determined by the **Broadcast Register**.

Figure 7 – Interrupt Distribution Scheme

The **Interrupt Broadcast Register** can be set for each IR line individually. If the broadcast bit of an interrupt line is set, the IRQ is sent to all cores and also has to be acknowledged (i.e. the ISR has to be processed) all of them. This is accomplished by setting the **Interrupt Force Registers** of all the cores. Each core has to individually clear its **Interrupt Force Register**!

If the broadcast bit is not set, the IRQ is sent to all cores and has to be acknowledged only once, i.e. only the first core that acknowledges the IR has to process the ISR. This is done by setting the **Interrupt Pending Register**, which can be cleared by any of the cores. In uni-processor systems the **Broadcast Register** is disabled.

Interrupts can be masked for each core individually. If bit **n** of the **Interrupt Mask Register** of core **m** is set to 0, then interrupt **n** is masked for this core, i.e. core **m** will never receive IRQ **n**. As a matter of fact, the VHDL implementation does not prevent an interrupt **n** clearance by core **m** in this case. For now, the SystemC module has been aligned to this behavior.

Interrupt masking takes place before prioritization, so the highest priority unmasked IR is always forwarded to the processors.

Interrupt 15 cannot be masked by the LEON3 core and should be used with care. Most operating systems do not safely handle this IR.

### Extended Interrupt Handling

In the IRQMP extended interrupts are cascaded, i.e. one of the regular IR lines may be defined as a cascade for the 16 EIR lines. The cascade is defined in bits 19..16 of the **Multiprocessor Status Register**.

If EIRs are asserted and the cascade is the highest priority active regular IR, the cascade is forwarded to the cores. After receiving the interrupt acknowledge signal from a core, the IRQMP unit writes the number of the asserted EIR line into the **Extended Interrupt Identification Register**. Thus, the ISR of the cascade has to send the acknowledge signal and afterwards read the EIR ID Register to call the correct ISR of the asserted EIR.

### Processor Status Monitoring

The processor status can be monitored through the **Multiprocessor Status Register**. The STATUS field [15..0] in this register indicates whether a processor is halted (‘1’) or running (‘0’). A halted processor can be reset and restarted by writing a ‘1’ to its STATUS field.

After reset, all processors except processor 0 are on halt. Once the system is properly initialized, processor 0 may start all other processors by switching on the respective STATUS bits.

To support this mechanism the LEON ISS, which is shipped with this library, has been modified. It provides a SignalKit input **run** and a SignalKit output **status**. Both signals are of type bool and must be routed to the **cpu\_rst** and **cpu\_stat** ports of the IRQMP. The **Multiprocessor Status Register** is kept in sync with the status information provided by the various processors.

### Power Modeling

Power monitoring can be enabled by setting the constructor parameter **pow\_mon**. The IP registers with the power monitor under the name **irqmp**. The model provides one event for modeling the static power consumption (idle) and 16 events representing interrupt request directed to the various CPUs in the system (**irq{1-16}**).

*idle* – Active during the complete runtime of the simulation (static power)

*irqX* – Dynamic power of driving an interrupt to processor X.

## Interface

The GRLIB VHDL model of the IRQMP is configured using Generics. For the implementation of the TLM model most of these Generics were refactored to constructor parameters of class **irqmp**. An overview about the available parameters is given in Table 26.

|  |  |  |  |
| --- | --- | --- | --- |
| Parameter | Function | Allowed Range | Default |
| name | SystemC name of the module |  |  |
| pindex | Selects which APB select signal (PSEL) will be used to access the IRQMP unit | 0 to  NAPBMAX – 1 | 0 |
| paddr | The 12-bit MSB APB address | 0 to 4095 | 0 |
| pmask | The APB address mask | 0 to 4095 | 4095 |
| ncpu | Number of processors in multicore systems | 1 to 16 | 1 |
| eirq | The cascade line of EIRs | 0 to 15 | 0 |
| pow\_mon | Enable power monitoring | 0 to 1 | 0 |

Table - Template Parameters

The system-level interface of the module comprises an GreenSocs/Carbon APB slave socket and multiple SoCRocket SignalKit ports (Table 27).

|  |  |  |  |
| --- | --- | --- | --- |
| Name | Type | In/Out | Description |
| rst | bool | in | Reset prescaler and all counters |
| clk | sc\_time | in | Annotates clock period |
| cpu\_rst | bool | selector | Generate reset for the processor(s) |
| cpu\_stat | bool | infield | Receive status inf. (halt/running) from processor(s) |
| irq\_req | uint32\_t | selector | Interrupt requests for the processors(s) |
| irq\_ack | uint32\_t | infield | Interrupt ackknowledge signals from processors(s) |
| irq\_in | uint32\_t | infield | Muxed interrupts from IRQ sources |

Table - IRQMP SignalKit sockets

## Internal Structure

This section describes the internal structure of the IRQMP. The class hierarchy of the model is flat. All functionality is comprised in class **Irpmp**, which is described in the files **irqmp.h** and **irqmp.cpp.**

### The irqmp.h file

The **IRQMP** unit consists of only one class. The **irqmp.h** file contains the module class definition. The parameterization options, implemented as generics in the VHDL model, are realized as constructor parameters of the class.

Class **Irqmp** is a child of **gr\_device**. A GreenReg device is a top-level encapsulation for a complete functional unit and provides containment structures for other GreenReg elements, e.g. registers. Moreover, **Irqmp** inherits the PNP configuration record of class **APBDevice**, and the clock and reset interface defined in **CLKDevice**.

The Irqmp class definition contains the module interface and the function prototypes of constructor, destructor, and callback functions. Next, to the well-known **SC\_HAS\_PROCESS** macro, the model call **GC\_HAS\_CALLBACKs** for registration with GreenReg and **SK\_HAS\_SIGNALS** for registration with the SignalKit.

### The irqmp.cpp file

The constructor of **Irqmp** configures the **apb\_device**, the **gr\_device** and the bus interface. It constructs a GreenReg register container **r**, in which it implements all the registers listed in Table 25. The register container is a C++ class implemented in the GreenReg libraries that provides memory management and interface functions. Within this register container, a GreenReg register may be instantiated like in the following code snippet:

1 r.create\_register(**"pending"**, **"Interrupt Pending Register"**,  
2                   0x04,  
3                   STANDARD\_REG | SINGLE\_IO | SINGLE\_BUFFER | FULL\_WIDTH,  
4                   0x00000000,  
5                   IRQMP\_IR\_PENDING\_EIP | IRQMP\_IR\_PENDING\_IP,  
6                   32,  
7                   0x00  
8 );

The arguments to the **create\_register()** function are **name**, **description**, **offset**, **configuration**, **init value**, **write mask**, **register width**, and **lock mask**. For a detailed description of these options, please refer to the GreenReg documentation.

In addition to building the interface, the constructor registers the **SC\_THREAD** **launch\_irq**. The **launch\_irq** thread is sensitive to the SystemC event **e\_signals** and contains the behavioral core of the model. The **e\_signals** event is triggered from three locations:

**incoming\_irq** - Handler bound to **irq\_in** socket, receiving the interrupts from all interrupt sources in the system.

**ackknowledge\_irq** - Handler bound to **irq\_ack** socket, receiving the ackknowledge signals from the processors.

**clear\_write** - Callback bound to Interrupt Clear register

**force\_write** - Callback bound to Interrupt Force register

**pending\_write** - Callback bound to Interrupt Pending register

For every state change in one of the observed registers or sockets, the **launch\_irq** function recalculates the IR lines for all connected processors. This is done in a loop starting from the processor with the highest ID. For each processor **launch\_irq** combines the pending register with the processor interrupt mask, to check whether there is an IR pending. It also checks for extended and forced IRs. From the resulting mask of IRs, the thread selects the level 1 IR with the highest priority for submission. Level 0 IRs are only considered, if there is no level 1 IR waiting. The selected IR is written to the processor as follows:

**irq\_req.write(1 << cpu, std::pair<uint32\_t, bool>(number, true));**

The first argument of the expression selects the processor, the second is a std::pair consisting of the interrupt number and a boolean value. The latter defines whether the interrupt line is switched on or off. This feature is especially important for RTL co-simulation. For plain TLM simulation transmission of the IR number would be sufficient.

## Compilation

For the compilation of the IRQMP unit, a WAF wscript file is provided and integrated in the superordinate build mechanism of the library.

All required objects for simulating the IRQMP are compiled in a sub-library named **irqmp** using following command:

**./waf –target=irqmp**

To utilize the IRQMP in simulations with other components, add **irqmp** to the use list of your **wscript**.

## Example Instantiation

The example below demonstrates the instantiation of the IRQMP TL model is a sc\_main or an arbitraty top-level class. the module is created in line 12. Line 15 connects the APB slave socket to the testbench (or APBCTRL). Lines 18 – 21 show how to bind the SignalKit sockets directed to the processor side. The Interrupt sources (**irq\_in**) are connected in line 26. The timing is annotated in line 30.

1 // Define Testbench

2 Testbench testbench;

3

4 // Define IRQMP

5 Irqmp irqmp;

6

7 // Constructor

8 Top(sc\_core::sc\_module\_name mn) : sc\_module(mn),

9

10 // Create Testbench & IRQMP

11 testbench(**"testbench"**, pindex, paddr, pmask, ncpu, eirq),

12 irqmp(**"irqmp"**, paddr, pmask, ncpu, eirq, 0) {

13

14 // Bind IRQMP APB socket to testbench

15 testbench.apb\_mst(irqmp.apb\_slv);

16

17 // Connect multiple virtual CPUs (testbenches) to IRQMP

18 **for**(**int** i=0; i<ncpu; ++i) {

19 connect(testbench.cpu\_rst, irqmp.cpu\_rst, i);

20 connect(testbench.irq\_req, irqmp.irq\_req, i);

21 connect(testbench.irq\_ack, irqmp.irq\_ack, i);

22 }

23

24 // Connect multiple interrupt sources to IRQMP

25 **for**(**int** i=0; i<32; ++i) {

26 connect(testbench.irq\_out, irqmp.irq\_in, i);

27 }

28

29 // Annotate timing

30 irqmp.set\_clk(10.0, SC\_NS);

# AHB2Socwire Systemc model

## Functionality and Features

### Overview

The AHB2SoCWire TLM IP models behaviour and timing of the AHB2SoCWire hardware model developed at IDA, TU-Braunschweig.

The implementation of the model is based on the “Technical specification of the AHB/SoCWire Bridge (Draft 2011-11-04)” document.

The model provides an AHB master interface and operates similar to a DMA controller. It is capable of independently transferring data from main memory over a SoCWire link (up-link) and from a SoCWire link to main memory (down-link). The operation of the device is controlled using a set of registers, which are accessible via a APB slave interface (11.2.2). During operation the module consumes memory descriptors indicating location, length and state of the data to be moved.

The SoCWire protocol is strongly related to the wide-spread SpaceWire protocol. Although, in contrast to SpaceWire, it is dedicated to on-chip communication. The SoCWire TLM IP contains a special SoCWire socket for network access. Communication on the network (SoCWire) side is packet based, not memory mapped. LT and AT mode of the model mainly differ in the abstraction of the packet flow-control.

## Interface

The original IP provides several generics, which some are modeled in the TLM model as constructor parameters, shown in Table 28.

|  |  |  |  |
| --- | --- | --- | --- |
| **Parameter** | **Function** | **Allowed Range** | **Default** |
| nm | SystemC module name | n/a | none |
| paddr | APB device address (12 bit MSB) | 0 to 4095 | none |
| pmask | APB address mask (12 bit) | 0 to 4095 | 0 |
| pindex | APB device bus ID |  | 0 |
| apb\_irq\_id | APB and AHB IRQ ID |  | 0 |
| hindex | AHB device bus ID |  | 0 |
| ambaLayer | Abstraction level (amba\_layer) | LT or AT | LT |

Table – AHB2Socwire Constructor Parameters

From application perspective the APB interface, the IRQs and the Clock Device are the interfaces of interest.

### Clock, AHB, APB Device Interfaces

The AHB2Socwire derives from device classes (compare section 3.2) and inherits their interfaces:

* Clock Device (class **CLKDevice**) allows manipulating the clock speed.
* AHB Device (class **AHBDevice**) manages the AHB device plug and play information.
* APB Device (class **APBDevice**) manages the APB device plug and play information.

### APB Slave Interface

The APB slave interface is implemented using the GreenReg framework. Respectively, the **AHB2SoCWire** class inherits from class **gr\_device**. The APB address space is configured via the constructor parameters **paddr** and **pmask**. The instance name of the GreenReg-Amba-socket is **apb**.

The AHB2Socwire module provides access to four registers within the assigned APB memory region.

|  |  |
| --- | --- |
| **APB Address Offset** | **Register** |
| 0x00 | Control Register |
| 0x04 | Status Register |
| 0x14 | Transmit descriptor table base address register |
| 0x18 | Receiver descriptor table base address register |

Table – AHB2Socwire Registers

The registers are described in the spec of the “AHB/SOCWire Bridge”. The bits and bit ranges can be accessed using named GreenReg bit ranges.

## Internal Structure

The block diagram in Figure 7 shows the structure of the AHB/SoCWire-Bridge illustrating its interfaces – classes the AHB2Socwire class is derived from and public members to be connected.

The module provides socket to communicate with the different parts of the system. These include the SoCWire socket, an AHB master and an APB slave socket.

Main parts of the SystemC module AHB2Socwire are

* an AHB master socket,
* several GreenReg registers, connected to
* a GreenSocket APB slave socket,
* a bidirectional SoCWire socket and
* an interrupt signal.



Figure – AHB2Socwire Structure

The AHB/SoCWire-Bridge IP is delivered in the following source files:

* AHB2Socwire.h : provides the class declaration of the actual AHB/SoCWire-Bridge, while AHB2Socwire.cpp holds its implementation.
* AHB2Socwire\_defs.h: defines some data types (e.g. rx/tx\_descriptor\_t).

The SoCWire socket (path socw\_socket) consists of the following files:

* socw\_socket.h: class declaration for the SoCWire socket template class
* socw\_socket.cpp: template implementation being included at the bottom of the header file (*not* to be compiled stand-alone!)
* socw\_defs.h: supplies SoCWire related data types, methods and constants that might also be useful for applications using the AHB2Socwire module.
* socw\_gp.h: defines TLM protocol extensions.
* socw\_debug\_functions.h: provides functions for debug purpose, socw\_debug\_functions.cpp the according implementations.

### Interrupts

The IRQ is a **signalkit** signal (**irq**) of type **bool** which represents several types of interrupts that can be checked by software by reading the status register (e.g. Transmitter Interrupt, Receiver Interrupt, SocWire CODEC status change Interrupt). The according interrupt line will be provided as the value of this **uint32\_t** type signal.

If an interrupt is generated its signal is set to true. Note: The interrupt is not resetted to false in simulation!

Examles:

The interrupt is thrown after a TX descriptor had been processed and the SoCWire packet had been sent (TI or TE status set).

The interrupt is thrown after data received on the SoCWire link has been written to the AHB memory and the RX descriptor had been updated (RI or RE status set).

The interrupt is thrown on reception of a codec state transition callback from the **socwire** socket (no status set).

### Device Attributes

AHB and APB vendor ID is 0x04.

AHB and APB device ID is TBD, currently 0.

SoCWire data width is 32 bit.

### AHB Master Interface

For reading transaction descriptors and fetching/receiving SoCWire payload data the AHB2Socwire module consists of an AHB master interface. The amba master socket is called **ahb**.

AHB transactions executed by the AHB2Socwire module are only indirectly influenced from application side by manipulating the module’s control registers.

### TX descriptor handling (LT)

The following describes the usual way without errors occurring.

The SystemC thread **handle\_TX\_descriptors\_LT** automatically walks through all the enabled TX descriptors in the AHB memory until all are processed. The thread is triggered by the event **ev\_handle\_TX\_descriptors\_LT** which is notified when Transmit Enable (TE) becomes activated within the control register **r[0x00].br["txEnable"]**.

The following steps are processed and repeated infinitely when the thread is triggered:

* read next TX descriptor from AHB memory, stop if disabled
* read data to be transferred from AHB memory
* send data as SoCWire packet
* update (e.g. disable) TX descriptor by writing it to AHB memory
* throw transmitter interrupt irq (\_TI)
* increment descriptor pointer and continue on top

### RX descriptor handling (LT)

The following describes the usual way without errors occurring.

The SystemC thread **handle\_RX\_descriptors\_LT** reads the next RX descriptor from AHB memory. The thread is triggered by the event **ev\_handle\_RX\_descriptors\_LT** which is notified when Receiver Enable (RE) becomes activated within the control register **r[0x00].br["rxEnable"]**.

When a SoCWire packet arrives, the **sockwire** socket gives it to the callback function **receive\_packet\_from\_socw\_cb** where the following steps are done:

* write data to the AHB memory (address according to current RX descriptor),
* update RX descriptor by writing it to AHB memory (resetting some flags and setting the length),
* throw receiver interrupt irq (\_RI)
* increment descriptor pointer
* read next RX descriptor to be used on next incoming SoCWire packet

### SoCWire Socket

#### Overview

The SoCWire link is encapsulated within the **socw\_socket** class. It hides standard actions like opening the link and keeping it running. Since transactions over the SoCWire link are only controlled indirectly using the descriptors, applications should never directly access the SoCWire socket.

#### Interface

The SoCWire socket is built of a pair of a GreenSocket **simple\_target\_socket** and a **simple\_initiator\_socket**. The sending (TX) initiator socket is called **master\_socket**. The receiving (RX) target socket is called **slave\_socket**.

The SoCWire socket uses **tlm\_generic\_payload** objects for transmissions, and adds some extensions due to the specific protocol, that are shown in Table 31.

SoCWire transactions are always writes so the command field is set to **tlm::TLM\_WRITE\_COMMAND** and can be ignored on receiver side.

|  |  |  |  |
| --- | --- | --- | --- |
| **Parameter** | **Function** | **Allowed Range** | **Default** |
| nm | SystemC module name | n/a | none |
| abstractionLevel | Abstraction level (amba\_layer) | LT or AT | LT |
| socw\_after64 | Timeout (equiv. to 6.4 us) in ns | 1 to 6400 | 64 |
| socw\_after128 | Timeout (equiv. to 12.8 us) in ns | 1 to 12800 | 128 |
| socw\_disconnect\_detection | Disconnection timeout in ns | 1 to 850 | 85 |
| socw\_automatic\_open\_link | If the socket shall open the link automatically. | true / false | true |
|  |  |  |  |

Table – SoCWire Socket Constructor Parameters

On reception of a wrong command the returned phase is **tlm::TLM\_COMMAND\_ERROR\_RESPONSE**.

**Data width:** The data width (template parameter DATA\_WIDTH) specifies the size of one word. Default is width 32 bit which means one word consists of 4 bytes (= 4 data characters in SoCWire terms).

|  |  |  |  |
| --- | --- | --- | --- |
| **Attribute** | **TLM Generic Payload or GreenSocket Extension (and type)** | **Usage** | **data  lifespan** |
| command | GP attribute | Only allowed TLM\_WRITE\_COMMAND | e2e |
| address\_offset\_ext | GS Extension (data, unsigned int) | Position of the next address within the data array to be used by router or device  Valid only if data length > 0, e.g.: 0: first 32 bit (Byte 0 to 3) to be used as address,  1: bytes 4 to 7 to be used as address, ... | p2p |
| data pointer | GP attribute | Pointer to data, details see below | e2e |
| data length | GP attribute | Size (in bytes) of the data array the data pointer points to – including all addresses, even already used ones.  Shall always be a multiple of bytes per word (DATA\_WIDTH/8). | e2e |
| EOP\_ext | GS Extension (guard only) | If packet ended with EOP  (not included in data) | e2e |
| EEP\_ext | GS Extension (guard only) | If packet ended with EEP  (not included in data) | e2e |
| FCT\_ext | GS Extension (data, unsigned int) | How many 0…x FCTs are contained in the packet (data might be empty or not, FCTs are not included in data attribute) | e2e |
| NULL\_ext | GS Extension (data, unsigned int) | How many 0…x NULLs are contained in the packet (data usually will be empty, NULLS are not included in data attribute) | e2e |
| parity\_err\_ext | GS Extension (guard only) | If there is a parity error in this packet – this abstracts from the position within the packet thus adding some inaccuracy here. | p2p |
| disconnect\_err \_ext | GS Extension (guard only) | When set the link shall disconnect and a disconnect error shall be reported (LT only ?) | p2p |
| response status | GP attribute | Debug information (no response provided by SoCWire protocol) | p2p |

Table – SoCWire TLM Protocol Attributes and Extensions

Note: Other TLM GP attributes like address are unused.

Note: To provide the error position within the data character array, the extension **parity\_err\_ext** could be changed to a data extension alternatively to being a guard only extension

Note: Data lifespan – end-to-end (e2e) means only the sender is allowed to set (Remains unchanged thought the life of a transaction). point-to-point (p2p) means that data can be modified by any component in the path.

Note: FTC\_ext is allowed to break protocol rules in LT mode by exceeding the maximum.

**Data pointer**: Pointer to packet data. The data contains the SoCWire packet’s data characters, One data character is represented by 1 byte while omitting the first 2 bits (parity + ‘0’) that are defined in the original protocol. The data includes all addresses but not control characters EOP, EEP, FCTs and NULLs which are transferred using other attributes. The data stays untouched by routers, so the receiving user has to strip the consumed addresses using **address\_offset\_ext** before processing the data. The data length attribute gives the (data character = byte) length including the addresses already being consumed. To get the real data length on receiver side the address offset (4 bytes = 1 word per offset) needs to be subtracted (e.g. **received\_data\_length = data\_length – (bytes\_per\_word\* data\_offset)**) and the pointer needs to be calculated accordingly.

#### LT Behaviour

The **disconnect\_err\_ext** is used to explicitly cause a disconnect error which would have been caused by a timeout in an AT simulation. This allows omitting the NULL transmissions.

The flow control is not modeled in LT mode. During link initialization MAX\_PACKET\_SIZE/8 (max characters divided by 8 characters because 1 FCT = 8 characters) is used as value for FCT\_ext.

## Compilation

For the compilation a WAF wscript file is provided and integrated in the superordinate build mechanism of the library.

All required objects for simulating the AHB2SoCWire bridge on platform level are compiled in a sub-library named **socwire** using following command:

**./waf –target=socwire**

To utilize SoCWire in simulations with other components, add **socwire** to the use list of your **wscript**.

## Example Instantiation

The example below demonstrates the instantiation of the SoCWire IP inside an sc\_main or an arbitraty top-level class. The AHB2Socwire bridge is create in lines 11 – 17. Line 25 connects the apb slave socket, line 25 the ahb master socket. The socwire socket is bound in line 31. Line 34 shows how to connect the SignalKit interrupt port and line 37 annotates the timing.

1 #include **"demo\_testbench.h"**

2 #include **"AHB2Socwire.h"**

3 #include **"ahbmem.h"**

4 #include **"ahbctrl.h"**

5 #include **"verbose.h"**

6

7

8 **int** sc\_main(**int** argc, **char**\* argv[]) {

9

10 // Instantiate AHB2SoCWire brdige (DUT)

11 socw::AHB2Socwire DUT(**"DUT"**, // Name

12 0xfff, // haddr

13 0xfff, // hmask

14 amba::amba\_LT,

15 10, // speed

16 0, // master\_id

17 0); // pindex

18

19 ...

20

21 // Instantiate test bench

22 Testbench tb(**"tb"**);

23

24 // APB Connection

25 tb.apb(DUT.apb);

26

27 // AHB Connections

28 DUT.ahb(ahbctrl.ahbIN);

29

30 // SoCWire Connections

31 DUT.socwire(tb.socw);

32

33 // IRQ Connection

34 connect(tb.irq\_in, DUT.irq);

35

36 // Clock

37 DUT.set\_clk(10, SC\_NS);

38

39 ...

# Annex a – Inconsistencies in the GRIP manual

Some details of the intended behavior of the IP cores are not stated clearly in the GRIP manual (LF04). In this annex, a list of such inconsistencies is given to indicate where the TLM IP cores had to be implemented in a consistent way within the scope allowed by the GRIP manual.

## A.1 – MCTRL Memory Controller

1. The GRIP document states (Sect. 58.4, p. 573):

*The SRAM area can be up to 1GB […]. The fifth bank decodes the upper 512 MB (controlled by means of the sdrasel VHDL generic) […].*

Conflict:

In contradiction to that, the allowed range for the *rommask*, *iomask*, and *rammmask* generics is 0 – 0xFFF, representing up to 4GB for each of the address ranges. The allowed range of the *romasel* and *sdrasel* generics is 0 – 31, also representing up to 4GB. *(Sect. 58.15, p. 584)*

Solution:

In the TLM model, the **default** SRAM area is set to 1 GB, but it is parameterizable according to the allowed ranges given in section 58.15 of the GRIP manual (up to 4GB). As – according to the GRIP manual – the size of the 5th SRAM bank is controlled by the *sdrasel* generic, which determines the size of the entire RAM area, the size of the 5th SRAM bank will always be half of the RAM address space (hence filling the entire SDRAM area if no SDRAM is present).

1. The GRIP document states (Sect 58.8, p.576):

*The SDRAM controller supports […] 512MB devices with 8-12 column address bits and up to 13 row address bits. The size of the two banks can be […] 512 MB.*

The size of a single SDRAM bank can be configured to be up to 512MB, resulting in an SDRAM device of 1GB capacity *(Sect. 58.13.2, p. 581)*.

Conflict:

With the given number of address bits (12 x 13), we do not see a possibility to address 512MB of memory (not even in 64 bit mode). The maximum would be:

212 \* 213 \* 64 bit = 212+13+3 Bytes = 228 Bytes = 256 MB

Solution:

In the TLM implementation, no address lines are required. This potential conflict is therefore ignored. The size of the SDRAM banks is configurable in binary steps from 4MB to 512MB.

1. The GRIP document defines (Sect. 58.13.2, p. 581)

*SDRAM COLSZ – “00”=256 […] “11”=4096 when SDRAM BANKSZ = 512MB, 2048 otherwise*

Conflict:

256 – 4096 would refer to 8 – 12 column address bits (then indicating the number of columns, i.e. the row length, not the column size. Again, addressing a 512MB bank would not be possible.

Note: For banks <512MB, COLSZ is fixed to 11 address bits, which would reduce the size of addressable SDRAM banks to 128MB.

Problem statement:

In the TLM implementation the row length can have an impact on the timing of burst accesses that span over two rows. In such a case, both rows would need to be opened and closed again implying an additional amount of delay.

Solution:

Regardless of the potential issue of a lack of address bits, the SDRAM COLSZ field is interpreted to define the row length and is therefore used in the unlikely case of an attempted burst access spanning over two rows.