**Title**

HW-SW SystemC Co-Simulation SoC Validation Platform

**IP User Manual**

Contract No. 4200022968

by

TU Braunschweig, Germany

May 9, 2011

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

**Table of Contents**

1 Introduction 6

1.1 Purpose and Scope 6

1.2 Referenced Documents 6

1.3 Revisions 6

2 Installation & Dependencies 7

2.1 Required Software Packages 7

2.2 Installing GreenSocs Software 7

2.2.1 GreenSocs 8

2.2.2 GreenSocs/Carbon AMBA Sockets 9

2.3 Building IP Model Library and Tests 9

3 Modeling Concepts 11

3.1 Concepts 11

3.2 Coding AMBA Masters and Slaves 11

3.3 GreenReg 13

3.4 TLM Signal Communication Kit 13

3.5 The Simple\_System Example 13

3.6 TLM Signal modeling kit (SignalKit) 15

3.7 Power Modeling 15

4 AHBCTRL AHB Controller SystemC Model 16

4.1 Functionality and Features 16

4.1.1 Overview 16

4.2 Internal Structure 17

5 AHB2APB bridge systemc model 17

5.1 Functionality and Features 17

5.2 Internal Structure 17

6 MCTRL Memory Controller SystemC Model 18

6.1 Functionality and Features 18

6.1.1 Overview 18

6.1.2 Address Space 18

6.1.3 SDRAM Modes of Operation 22

6.2 Internal Structure 24

6.2.1 The mctrl.h File 25

6.2.2 The mctrl.tpp File 27

6.3 Parametrization Options 29

6.4 Interface 30

6.4.1 APB Bus Communication 30

6.4.2 AHB Bus Communication 31

6.4.3 Memory Device Interface 31

6.5 Compilation Instructions 31

6.6 Example Instantiation 31

7 GENERIC Memory SystemC Model 32

7.1 Functionality and Features 32

7.2 Internal Structure 32

7.2.1 The generic\_memory.h File 33

7.2.2 The generic\_memory.tpp File 33

7.3 Parametrization Options 33

7.4 Interface 33

7.5 Compilation Instructions 34

7.6 Example Instantiation 34

8 MMU\_CACHE Cache sub-system systemC module 35

8.1 Functionality and Features 35

8.1.1 Overview 35

8.1.2 Address Space Identifiers (ASI) 36

8.1.3 System and Control Registers 37

8.1.4 Diagnostic Access 39

8.1.5 Payload Extensions 39

8.1.6 Debug Mechanism 40

8.2 Internal Structure 41

8.2.1 The defines.h file 41

8.2.2 The payload\_extension files 41

8.2.3 The mem\_if.h file 42

8.2.4 The mmu\_cache\_if.h file 43

8.2.5 The mmu\_cache.h/cpp files 43

8.2.6 The vectorcache.h/cpp files 44

8.2.7 The ivectorcache.h/cpp files 45

8.2.8 The dvectorcache.h/cpp files 46

8.2.9 The localram.h/cpp files 46

8.2.10 The mmu.h/cpp files 46

8.2.11 The tlb\_adaptor.h file 47

8.3 Parametrization Options 47

8.4 Interface 49

8.5 Compilation Instructions 49

8.6 Example Instantiation 50

9 GPTIMER General Purpose Timer SystemC model 52

9.1 Functionality and Features 52

9.2 Internal Structure 53

9.2.1 The gptimer.h file 53

9.3 Parametrization Options 55

9.4 Interface 55

9.5 Compilation Instructions 55

9.6 Example Instantiation 56

10 IRQMP Interrupt Controller SystemC model 56

10.1 Functionality and Features 56

10.1.1 Overview 56

10.1.2 Interrupt Prioritization and Forwarding 57

10.1.3 Extended Interrupt Handling 59

10.1.4 Processor Status Monitoring 59

10.2 Internal Structure 59

10.2.1 The irqmpregisters.h File 59

10.2.2 The irqmp.h File 59

10.2.3 The irqmp.tpp file 60

10.3 Parametrization Options 61

10.4 Interface 61

10.4.1 APB Bus Communication 61

10.4.2 Direct Processor Communication 62

10.5 Compilation Instructions 62

10.6 Example Instantiation 63

11 SocWire Systemc model 64

11.1 Functionality and Features 64

11.2 Internal Structure 64

11.3 Parametrization Options 64

11.4 Interfaces 65

11.4.1 APB Slave Interface 65

11.4.2 Interrupt Request 65

11.4.3 AHB Master Interface 65

11.4.4 Socwire Interface 66

11.5 Socwire Payload Object 66

11.6 SocWire Protocol Implementation 67

11.6.1 LT-Level 67

11.6.2 AT-Level 68

11.7 Compilation Instructions 68

11.8 Example Instantiation 68

12 GRLib Plug & Play mechanism 69

12.1 Example Instantiation 69

Annex a – Inconsistencies in the GRIP manual 70

A.1 – MCTRL Memory Controller 70

**Table of Figures**

Figure 1 - AMBAKit / Initiate Transaction 11

Figure 2 - AMBA Slave / blocking transport 13

Figure 3 - Simple\_System Test 14

Figure 4 – RAM address space 19

Figure 5 – Structure of the MCTRL TLM 24

Figure 6 - Structure of Cache Sub-System 35

Figure 7 - Generic Memory Interface / Dependencies 42

Figure 8 – Interrupt Distribution Scheme 57

**Table of Tables**

Table 1 - Referenced Documents 5

Table 2 - Revisions of this document 5

Table 3 - Software Dependencies 6

Table 4 – MCTRL Registers **Fehler! Textmarke nicht definiert.**

Table 5 – MCTRL Template Parameters **Fehler! Textmarke nicht definiert.**

Table 6 - Supported ASIs 33

Table 7 - CACHE CONTROL REGISTER 33

Table 8 - ICACHE & DCACHE Configuration Register 34

Table 9 - MMU Control Registers 35

Table 10 - Debug Extension 36

Table 11 - TLM Sockets Cache Sub-System 40

Table 12 - Page size / index combinations 43

Table 13 - Constructor Configuration Parameters 44

Table 14 - Constructor Simulation Parameters 45

Table 15 - Tests Cache Sub-System 46

Table 16 – GPTimer Registers 48

Table 17 - GPTimer Parameters 51

Table 18 - Timer SignalKit sockets 52

Table 19 – IRQMP Registers 53

Table 20 - Template Parameters 57

# Introduction

## Purpose and Scope

This document is a user manual (UM) of the SystemC transaction level models developed in the HW-SW SystemC Co-Simulation SoC Validation Platform project.

In compliance with the SoW, the „UM describes the IP interface and functions and its use from the perspective of the system architect and the programmer, including examples.“

## Referenced Documents

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

|  |  |  |
| --- | --- | --- |
| **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 |
| RD05 |  | GRLIB IP Library User’s Manual |
| RD06 |  | GreenSocs AMBA LT/AT concepts |
| RD07 |  | GreenSocs AMBA TLM 2.0 Extensions |
| RD08 |  | SPARC V8 Reference Manual |
|  |  |  |

Table 1 - Referenced Documents

## Revisions

|  |  |  |
| --- | --- | --- |
| **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 |  |

Table 2 - Revisions of this document

# Installation & Dependencies

## Required Software Packages

The Model 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 models following external dependencies are needed (Table 3).

| Tool / Lib | Version | Vendor | Installation Path Variables |
| --- | --- | --- | --- |
| Python | >2.3 | Python team | On **$PATH** |
| GCC (x86) | 4.1.0 | GCC team | On **$PATH** |
| GMP | 5.0.0 | GCC team | On **$PATH** |
| MPFR | 2.4.2 | GCC team | On **$PATH** |
| binutils | 2.19 | 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 | 4.0.0 | GreenSocs Ltd. | **$GREENSOCS\_HOME** – installation root |
| AMBAKit | trunk | GreenSocs Ltd. | **$AMBA\_HOME** – installation root |

Table 3 - Software Dependencies

Please make sure that all the software packages mentioned above are properly installed before the library ist build.

## Installing GreenSocs Software

### GreenSocs

The GreenSocs Library can be downloaded from the following location:

<http://www.greensocs.com/files/greensocs-4.0.0.tar.gz>

Extract the tarball to a folder on your build system:

$ tar -xvzf greensocs-4.0.0.tar.gz

Make sure the following shell variables hold paths to the corresponding libraries:

$ 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

Change to the greensocs-4.0.0/greenreg folder and make a small change in the **Makefile.conf**:

Comment out the 2nd to 4th lines with #:

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

3

4 GREENREG\_DIR=$(TOP)

to:

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

3 **#**

4 **#**GREENREG\_DIR=$(TOP)

Compile the GreenReg library:

$ make

A static library is build in the GreenReg directory. This is needed by the model library for the APB modules.

Now the GreenSocs library is ready to use with our model library.

Simply export the GreenSocs root directory as GREENSOCS\_HOME:

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

Or use the waf configuration parameter –greensocs with the greensocs root directory of the model library:

$ ./waf configure –greensocs=<THE FOLDER YOU **EXTRACTED** GREENSOCS>

It is recommended to apply the AMBAKit Sockets patch to the GreenSocs library. This enables you to build your own models with GreenReg and AMBA Sockets outside the model library. Anyway the model library arrives with a workaround.

### GreenSocs/Carbon AMBA Sockets

The GreenSocs/Carbon AMBA Sockets are available from the Website of Carbon Design Systems …

Extract the tarball to a folder on your build system:

$ tar -xvzf AMBAKit-trunk.tar.gz

The AMBAKit is like the TLM Library only a collection of headers. Just export the location to the build system of de model library:

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

Or use the waf configuration parameter –amba with the greensocs root directory of the model library:

$ ./waf configure –amba=<THE FOLDER YOU **EXTRACTED** THE AMBAKIT>

## Building IP Model Library and Tests

The build system is written in **waf**. All dependencies will be checked before the compilation of the project begins. The **waf** binary is located in the project root. It requires at least python 2.3 to run.

Do a “**./waf –h**” to get help on all available commands and options.

waf [command] [options]

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 by one of the following options:

Common Configuration Options:

-b BLDDIR, --blddir=BLDDIR

build dir for the project (configuration)

-s SRCDIR, --srcdir=SRCDIR

src dir for the project (configuration)

--prefix=PREFIX installation prefix (configuration) [default: '/usr/local/']

--download try to download the tools if missing

--cxxflags=CXXFLAGS

C++ compiler flags

C++ Compiler Configuration Options:

--check-cxx-compiler=CHECK\_CXX\_COMPILER

On this platform (linux) the following C++ Compiler will be checked by default: "g++ icpc sunc++"

Boost Configuration Options:

--boost=BOOST\_HOME Basedir of your Boost installation

--boost\_inc=BOOST\_INC

Include dir of your Boost installation

--boost\_lib=BOOST\_LIB

Library dir of your Boost installation

SystemC configuration Options:

--systemc=SYSTEMC\_HOME

Basedir of your SystemC installation

--tlm2=TLM2\_HOME Basedir of your TLM-2.0 distribution

--tlm2tests=TLM2\_TESTS

Example of your TLM-2.0 distribution

--scv=SCV\_HOME Basedir of your SCV distribution

GreenSoCs Configuration Options:

--greensocs=GREENSOCS\_HOME

Basedir of your GreenSoCs instalation

--amba=AMBA\_HOME Basedir of your AMBAKit distribution

--grlib=GRLIB\_HOME Basedir of your grlib distribution

--ambaexamples=AMBA\_EXAMPLES

Basedir of your AMBAKit Examples

1. Execute “**./waf**” – to build all models and tests.  
     
   As an alternative you may select a specific target (test or library) using   
   “**./waf --targets=”list,of,targets”**”.   
     
   A list of targets can be generated with “**./waf list**”.
2. Execute “**./waf docs**” – to generate the source doxygen documentation.

# Modeling Concepts

## Concepts

The IP components described in this project are build respecting the OSCI TLM2.0 transaction level modeling standard. Respectively, all models are available in a loosely timed flavor with blocking transport interfaces and in approximately timed non-blocking flavor.

The models in this library depend on the GreenSocs GreenReg framework and the GreenSocs/Carbon TLM AMBA socket implementation. GreenReg registers can be directly hooked onto TLM sockets facilitating the modeling of memory mapped registers. The mentioned combination of sockets and GreenReg registers is a powerful tool. We advocat this modeling style for potential extensions of the library, although it is not mandatory.

## Coding AMBA Masters and Slaves

The GreenSocs/Carbon AMBA TLM sockets provide a TLM phase protocol and a set of TLM payload extensions that has been adopted by ARM. It can therefore be expected to become a defacto standard for TLM AMBA bus modeling. Reuse of this preexisting solution will also allow users to be compatible with and benefit from the cycle timed verification IP offered by ARM and Carbon.

The respective software pack is available from the Carbon Website free of charge.

In the following the usage of the AHB sockets is demonstrated on an example.

1 class amba\_master : public amba\_master\_socket<32> {  
 2     public:  
 3         amba\_master( sc\_module\_name \_name, amba\_bus\_type \_type,   
 4         amba\_layer\_ids \_layer, bool \_arbiter)  
 5         : amba\_master\_socket<32>(\_name, \_type, \_layer, \_arbiter) {}  
 6   
 7         void write(uint32\_t addr, unsigned char \*data, uint32\_t length) {  
 8             sc\_core::sc\_time t;  
 9             tlm::tlm\_generic\_payload \*gp = amba\_master.get\_transaction();  
10             gp->set\_command(tlm::TLM\_WRITE\_COMMAND);  
11             gp->set\_address(addr);  
12             gp->set\_data\_length(length);  
13             gp->set\_streaming\_width(length);  
14             gp->set\_byte\_enable\_ptr(NULL);  
15             gp->set\_data\_ptr(data);  
16             amba\_master->b\_transport(\*gp,t);  
17             wait(t);  
18             amba\_master.release\_transaction(gp);  
19         }  
20 };

Figure 1 - AMBAKit / Initiate Transaction

Figure 1 shows the setup of a transaction within an AMBA master module. First an object of type *tlm\_generic\_payload* is retrieved from the transaction pool of the AMBA master socket. The payload object can also be newly created. Although, for performance reasons it is recommended to reuse existing objects. Afterwards, the various attributes of the payload object are initialized by calling the appropriate payload member functions. In the most simple case, which is shown here, the payload receives information about the target address, the number of bytes to be transferred and the streaming width. The request may be a TLM\_WRITE\_COMMAND or a TLM\_READ\_COMMAND. In case of a write the *set\_data\_ptr* function receives a pointer to the data to be written as an argument. Read requests also require a data pointer. In that case, the slave copies the data directly to the given target address. The transaction is started by calling the transport function of the master socket (*master\_sock->b\_transport*). The transport function requires two arguments: the address of the payload object and a delay pointer. In a blocking communication the delay is incremented by the target component and by all transfer components (busses, bridges) the transaction passes through. After return of the transport function the master advances the system time by calling the SystemC wait function. Finally, the payload object can be released (*amba\_master.release\_transaction*). That means it goes back to the transaction pool, waiting to be allocated again. For non-blocking communication the structure of the master is more complicated, because the slaves are allowed to independently advance the simulation time (call wait). Respectively, the state of a transaction has to pass through multiple phases along a well-defined protocol.

The structure of an AMBA slave component for blocking communication is shown in Figure 2. To be able to react on incoming transactions, each slave has to implement a transport function. First the latter is ought to be registered at the slave socket (amba\_slave::register\_b\_transport). The transport function expects a payload pointer and a delay pointer as arguments. In the function body the payload object is extracted. First the information about target address and access type are evaluated (*gp.get\_address(), gp.is\_write(), gp.is\_read()*). Depending on the functionality of the component, the data is now directly read/written or other transactions are started (e.g. router). Afterwards the function returns. At AT-level each phase of the bus protocol must be executed one by one. This is mostly implemented within the AMBA sockets and not in all aspects visible to the user. Nevertheless, master and slave thread rotationally receive control in order to prepare the following protocol step, similar to real hardware. For more detailed information please see RD06 and RD07.

 1 class amba\_slave : public amba\_slave\_socket<32> {  
 2     public:  
 3         amba\_slave( sc\_module\_name \_name, gr\_uint\_t \_base\_address,   
 4                     gr\_uint\_t \_decode\_size, amba\_bus\_type \_type,   
 5                     amba\_layer\_ids \_layer, bool \_arbiter)  
 6         : amba\_slave\_socket<32>(\_name, \_type, \_layer, \_arbiter)  
 7         , m\_base( \_base\_address), m\_high( \_base\_address + \_decode\_size) {  
 8             // Bind amba blocking ...  
 9             amba\_slave::register\_b\_transport(this, &amba\_slave::b\_transport);  
10         }  
11         void b\_transport(tlm::tlm\_generic\_payload& gp, sc\_core::sc\_time&) {  
12             unsigned int addr = gp.get\_address() - m\_base;  
13             unsigned int length, msk, data;  
14             transaction\_type trans(this, &gp);  
15             if(gp.is\_write()) {  
16                 length = gp.get\_data\_length();  
17                 msk = len2mask(length);  
18                 memcpy(&data, &(gp.get\_data\_ptr()[0]), length);  
19                 m\_registers->bus\_write(data, addr, msk,   
20                                        &trans, m\_delay\_enabled);  
21             } else if(gp.is\_read()) {  
22                 msk = len2mask(length);  
23                 m\_registers->bus\_read(m\_rdata, addr, msk,   
24                                       &trans, m\_delay\_enabled);  
25                 gs::MData mdata(gs::GSDataType::dtype((uint8\_t \*)&m\_rdata,  
26                                 trans->getMBurstLength()));  
27                 trans->setSData(mdata);  
28             }  
29             gp.set\_response\_status(tlm::TLM\_OK\_RESPONSE);  
30         }  
31   
32         // tlm\_slave\_if implementation  
33         virtual void setAddress(uint64 base, uint64 high) {  
34             m\_base = base; m\_high = high;  
35         }  
36   
37         inline uint64 get\_base\_addr() { return m\_base; }  
38         inline uint64 get\_size() { return m\_high - m\_base;}  
39         inline void set\_delay() {}  
40   
41     protected:  
42         uint64 m\_base, m\_high;  
43         unsigned int m\_rdata;  
44 };

Figure 2 - AMBA Slave / blocking transport

## GreenReg

## TLM Signal Communication Kit

## The Simple\_System Example

A good starting point for working with the library is the Simple\_System example. It is located in the library-level test directory (tests/simple\_system). The structure of the system is depicted in Figure 3.

Figure 3 - Simple\_System Test

The library components are drawn in blue, while the AMBAKit components are drawn in red. The system contains a testbench, which acts as a placeholder for the processor and provides stimuli to the system. The testbench connects to the Cache-Subsystem (mmu\_cache) through two TLM 2.0 sockets. The payload transported over these sockets contains optional extensions for address space selection (ASIs) and debugging. The mmu\_cache IP features an AHB master socket. The master socket is bound to the slave socket of a LT\_CT adapter. The system bus (ahb\_simple\_bus) and the AHB/APB bridge (ahb\_apb\_bridge) are part of the AMBAKit and model the communication with cycle timed accuracy. The memory controller (mctrl) is equipped with one AHB slave socket, one APB slave socket and TLM2 master sockets. The AMBA slaves are connected to the bus through CT\_LT adapters. The TLM2 masters are connected to four instances of the generic memory IP.

[Example will be extended throughout the course of the project!]

## TLM Signal modeling kit (SignalKit)

## Power Modeling

# AHBCTRL AHB Controller 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 level of abstractions (LT and AT). For modeling the AHBCTRL we mostly follow the recommendations given in RD06.

**AHB AT protocol adaptation**

At AT abstraction the AHB protocol is abstracted in a way to map to the phases of the TLM 2.0 standard protocol + one additional but ignorable phase for DATA\_SPLIT. It is assumed that each transaction consists of exactly one request and one response. Also, other then the RTL model, we make no distinction between master-bus and bus-slave communication. Arbitration is modeled implicitly. A BEGIN\_REQ from a master implies a preceeding bus request. The AHBCTRL delays the BEGIN\_REQ until the master wins arbitration. Only then the transaction is forwarded to the slave.

In order to model split transfers we introduce a phase called DATA\_SPLIT. The following rules make DATA\_SPLIT ignorable:

* Masters will never react upon reception of DATA\_SPLIT.
* Slaves may send DATA\_SPLIT on the return path in response to BEGIN\_REQ.
* Slaves may send DATA\_SPLIT on the backward path after having returned TLM\_ACCEPTED to BEGIN\_REQ
* Slaves may send DATA\_SPLIT after having returned TLM\_UPDATED with END\_REQ in response to BEGIN\_REQ
* Slaves may send DATA\_SPLIT after having sent END\_REQ on the backward path.

In any case, a split transfer is considered completed whenever a slave returns BEGIN\_RESP. If the AHBCTRL receives a DATA\_SPLIT it will rearbitrate.

As soon the bus receives BEGIN\_RESP for the transaction for which the slave sent DATA\_SPLIT it will add the respective master to the internal arbitration and continue.

The BEGIN\_REQ at AT approximates the point in time the RTL model starts requesting the bus. END\_REQ marks the end of the address phase. The END\_REQ may be sent via the backward or the return path. The slave may as well return TLM\_COMPLETED. In this case the timing annotation should mark the best possible guess for the end of the data phase.

The phase BEGIN\_RESP shall be sent by a slave at the best possible guess for the begin of the last data burst. The master should return TLM\_UPDATED with END\_RESP or TLM\_COMPLETED, as there is now way to delay the end of the response in AHB, but may as well return TLM\_ACCEPTED and send an explicit END\_RESP later.

**AHB LT protocol adaptation**

At LT the transaction will be transmitted using b\_transport. It will complete within a single call. Compared to AT abstraction features such as arbitration and data split are abstracted away. This makes the model very fast, by providing all the functionality expected from programmers view.

**AHB Payload extensions**

For a detailed description of the AHB payload extensions check out the documentation of the Carbon/GreenSoCs AHB Sockets (RD07 chapter).

### Plug & Play Support

All PnP components have to inherit the GrlibDevice class located in ‘models/utils’. This class builds up the PnP register set identifying the device. The contents of the registers have to be passed to the GrlibDevice in the constructor of the child module.

The GrlibPnP module located in ‘models/pnp’ exports all configuration registers and all bank address registers of the GrlibDevices to the memory area specified for the PnP table. The default for this area is 0xFFFFF000 – 0xFFFFFFFF.

To register the GrlibDevices with the GrlibPnP unit, the register\_master (GrlibDevice master) or register\_slave (GrlibDevice slave) function of the GrlibPnP module have to be called in the main function of the system. Their argument is the instance name of the GrlibDevice within the instance of the TLM hardware component.

## Internal Structure

# AHB2APB bridge systemc model

The AHB2APB TLM model can be used to simulate behavior and timing of the GRLIB AHB-APB Bridge VHDL IP. The model is available at two level of abstractions (LT and AT). For modeling the AHB2APB we mostly follow the recommendations given in RD06.

## Functionality and Features

**AT APB protocol adaptation**

At AT abstraction the AHB protocol is abstracted in a way to map to the phases of the TLM 2.0 standard protocol. Compared to the RTL model the selection signal assertion phase will be abstracted away. All transactions start with phase BEGIN\_REQ. Since the APB protocol is very simple, it is acceptable for slaves to directly return TLM\_COMPLETED (at the time estimated for the end of the access). Nethertheless, the AT version of the AHB2APB supports all legal TLM base protocol sequences.

**LT APB protocol adaptation**

The LT version of the AHB2APB matches the default TLM2.0 scheme for blocking transport (b\_transport). All transactions are assumed to complete in a single call.

**APB Payload extensions**

APB does not require any extensions to the TLM Generic Payload.

## Internal Structure

# MCTRL Memory Controller SystemC Model

## Functionality and Features

### Overview

The TLM implementation of the MCTRL unit reproduces the functionality of the Gaisler GRLIB MCTRL VHDL implementation described in RD04. The timing is approximated at two different levels of abstraction (LT and AT). The MCTRL is a slave on the AHB bus and on the APB bus. It controls a memory subsystem comprising up to two memory busses and up to four different types of memory: PROM, memory mapped I/O, SRAM, and SDRAM. The MCTRL TLM model provides one exclusive TLM master socket for each of the mentioned types of devices. Addressing is done using three distinct address spaces (ROM, I/O, and RAM). Read and write transactions from the AHB slave interface are forwarded to the appropriate memory master socket. Incoming APB transactions are directed towards the register control interface.

The register control interface consists of four configuration registers (Table 4). 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 4 – MCTRL Registers

### 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 GRLIB PnP BAR0 register of the MCTRL. The *rommask* is written to the 12 bit-wide MASK field of the GRLIB PnP BAR0 register. The bit mask represents the 12 most significant bits of the memory address. As the address space is byte-addressable and the address width is 32 bit, the 12 MSBs can mask the address space with a resolution of 2(32-12) bytes, i.e. 1 MByte.

The size of the PROM address space is:

(212 – rommask) 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 a subdivision into memory banks is not supported for local I/O.

The RAM 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* and *SRAM disable* indicating the presence of SRAM, SDRAM, or SRAM & SDRAM. If the *SDRAM enable* bit is low, *SRAM disable* has no effect.

For details and information on the organization of the RAM 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 GRIP user manual (RD04). Examples for possible partitionings of the RAM address space (default size of 1 GByte) are given in Figure 4.

Figure 4 – 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 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 bank5 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 a *rammask* equal 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 4 by overwriting the respective bits in the MCFG2 register. In such an event the MCTRL will recalculate the start and end addresses of all SRAM and SDRAM memory banks. As only the address decoding will change, the contents of the memories remain unchanged. 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.

*Remark:*

*In addition to rommask and rammask, the VHDL model provides the generics romasel and sdrasel to reflect the partitioning of the PROM and RAM address space. Both parameter sets must be properaly aligned to ensure correct operation of the MCTRL. To reduce complexity and redundandency the TLM model does not contain romasel/sdrasel constructor parameters.*

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. With respect to the result, the *streaming width* of the payload is adjusted. Afterwards the transaction is forwarded to the responsible memory socket.

The delay of a transaction is always 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. All delay values are calculated as multiples of the bus clock period. For each memory access, the MCTRL adds a decoding delay of one bus cycle.

#### 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 return a TLM\_COMMAND\_ERROR\_RESPONSE.

A read access to PROM memory takes 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.

A PROM write access can have a streaming width of 32, 16, or 8 bits. The access mode is set in the PROM WIDTH field of the MCFG1 register. If the PROM WIDTH field is set to 16 or 8 bits, a read access to PROM will still result in retrieving a full 32 bit word, which 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 VHDL implementation provides a dynamic *bus-ready-signaling* mechanism, which can induce an arbitrary number of wait states. As these arbitrary wait states model the latency of the attached I/O device, 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 bank5 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

The 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.

In the RTL model, the SDRAM device is controlled by SDRAM commands. An ACTIVATE command opens a row, while a PRECHARGE command closes a row. As long the row is open READ or WRITE commands are issued to access data.

A read access is always performed as a page burst access. Because a page burst 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, the memory device needs to take notice of this change. It will be reconfigured by MCTRL sending a LOAD MODE REGISTER (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 different modes. The availability of operation modes depends on the *mobile* generic, which determines the support of mobile memory. SDRAM memories dedicated to mobile devices support several power saving options.

In case mobile memory is not supported (mobile = ‘0’) the MS field of MCFG2 is set to zero in the initialization phase. This disables the power saving options held in the MCFG4 register of the MCTRL.

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 power saving modes, power down, self-refresh, partial array self refresh, and deep power down. The mode of operation is determined by the MCTRL unit and can be set in the PMODE field of MCFG4.

#### Normal Operation Mode

On system startup, the SDRAM device is initialized for normal operation mode. As the software memory model does not need any initialization sequence, the latter is modeled by just adding the according delay.

In normal operation mode, the memory access works as described in section 4.1.3.4. In case of a change of the operation mode, the memory has to be configured for this mode by issuing a LOAD\_EXTENDED\_MODE\_REGISTER (EMR) command. Like the LMR command, EMR does not have any functional effect, but only introduces the delay of one cycle plus tRP (as defined in the TRP field of MCFG2).

#### 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, i.e. each memory access takes one additional bus clock cycle in power down mode.

As the memory device wakes up on every access, it will also wake up on AUTO REFRESH. Hence, the power down mode is left for at least 16 bus clock cycles after any refresh cycle.

Entirely leaving power down mode by changing the PMODE field of the MCFG4 register induces the delay of an EMR command.

#### Self-Refresh Mode

If the system is powered down, the data in mobile SDRAM can still be retained through sending the SDRAM device into self-refresh mode. Entering self-refresh mode is induced by setting the PMODE field of the MCFG4 register to “010” and induces the delay of an EMR command.

In self-refresh mode, the system is expected to be powered down, i.e. memory access is expected not to be requested. Therefore the MCTRL unit will issue a TLM\_ADDRESS\_ERROR\_RESPONSE, if access to an SDRAM device in self-refresh mode is attempted.

Leaving self-refresh mode will induce a delay of an EMR command plus TXSR as defined in the MCFG4 register plus an auto-refresh cycle as defined by the TRFC field in the MCFG2 register.

#### Partial Array Self-Refresh Mode

In self-refresh mode, it is possible to retain only parts of the data in memory by activating the partial array self-refresh mode. This mode is entered setting the three-bit-wide PASR field in the MCFG4 register to a value not equal to zero. The partial array can be defined as half, quarter, eighth, or sixteenth by setting the PASR field to 001, 010, 101, or 110 respectively. The partial array always refers to the lower fragment of the SDRAM address space.

In the TLM model, entering partial array self-refresh mode will immediately erase all parts of SDRAM, which are not refreshed.

#### 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”. In deep power down mode, the contents of the SDRAM are deleted immediately. Any access to an SDRAM device in deep power down mode will result in a TLM\_ADDRESS\_ERROR\_RESPONSE by the MCTRL unit.

Deep power down mode can be exited changing the PMODE field of MCFG4. Leaving this mode will launch an initialization sequence.

## Internal Structure

The MCTRL module is implemented as a single class. Next to the internal functionality it comprises an AHB slave socket, an APB slave socket, four memory master sockets and a reset input signal. An overview of the module, including its interface, reset mechanism, dynamic configuration, and operation is given in Figure 5.

Figure 5 – Structure of the MCTRL TLM

According to the BSSC2000(1) coding standard, the definition and the implementation of the module are stored in two separate files, mctrl.h and mctrl.cpp, respectively. The subsequent sections describe the contents of these files and explain the components shown in Figure 5.

### The mctrl.h File

The ‘mctrl.h’ file contains the module class definition.

#### Parameterization of the module

The parameterization options, implemented as generics in the VHDL model, are realized as constructor parameters of the class. This makes the module parametrizable during instantiation. Details on the parameters are given in section 4.3.

#### Configuration of the module

The MCTRL unit is configurable through its configuration registers. The configuration registers, which are accessible through the APB bus, are modeled and accessed through the comfortable mechanisms provided by GreenReg. To ensure GreenReg compatibility, the MCTRL module needs to be a child module of a GreenReg 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. Thus, the MCTRL class inherits the gr\_device class.

The ‘mctrl.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.

The default masks are written to the registers at system initialization and in the system-reset function.

#### Communication with the module

A bus master can access the MCRTL unit through the AHB bus. For correct attachment to the AHB bus, the MCTRL unit also needs to inherit the amba\_slave\_base class. The get\_base\_addr and get\_size functions of this class are overloaded in the MCTRL class definition. These functions specify the address space dedicated to the MCTRL unit. Thus, the get\_base\_addr function returns the start address of the lowest memory and the get\_size function returns the size of the entire address space managed by the MCTRL unit.

#### Operation of the module

The MCTRL class definition contains the module interface and the function prototypes of constructor, destructor, callback functions, and pure C++ software routines. For reasons of simulation performance it is always good to avoid the usage of SystemC processes. Hence, the functionality of the MCTRL module has been modeled without any processes.

SystemC processes have to be registered with the SystemC simulation kernel using the SystemC macro, SC\_HAS\_PROCESS(). In a similar fashion, the callback functions, which are hooked to the registers built with GreenReg, are registered with the SystemC simulation kernel using the GreenControl macro, GC\_HAS\_CALLBACKS().

In addition, some class attributes are defined to keep track of the overall configuration and operation of the module:

* The **address space variables** define the borders of each memory bank attached to the device. These variables are required for the address decoding mechanism and to identify, which type of memory is accessed in the current transaction. The type of memory must be known, because the timing is modeled in the MCTRL unit and differs for each type of memory. The timing is modeled in the MCTRL unit and not in the memory itself, because all timing information is known in the MCTRL model and the attached memory model shall be kept generic.
* A **pmode variable** is used to indicate the current operation mode of the SDRAM device. The operation mode will affect the timing of a transaction and therefore needs to be checked for each transaction. Hence, the pmode variable can be interpreted to represent a part of a state machine controlling the SDRAM device.
* A **callback\_delay variable** saves any delay that occurs during the execution of callback functions. To save implementing an SC\_THREAD to model this delay, it is added to the delay of the next transaction.
* A **start\_idle variable** stores the sc\_time\_stamp at which SDRAM enters idle state. If, in power down mode, the start\_idle time lies more than 16 clock cycles in the past, an SDRAM access will take an additional bus clock cycle for waking up from power down state.
* A **next\_refresh variable** stores the point of time at which the next SDRAM refresh cycle is scheduled to start. The variable is updated dynamically. The refresh mechanism does not have any impact on the functionality, but it may influence the timing of a transaction. This may happen in case the transaction starts, while a refresh is active. In that case the transaction will be stalled and started after the end of the refresh cycle.
* A **trfc variable** stores the number of cycles a refresh will take. It can dynamically change by writing to the SDRAM TRFC field of the MCFG2 register. The value is stored to a variable, because it has to be checked for each SDRAM transaction. A variable is much faster than reading the value from the configuration register container each time.
* A **refresh\_stall variable** stores the amount of simulation time for which a refresh has to be stalled. A stall can be necessary if an SDRAM access is currently active, while the start of a refresh cycle is scheduled. The refresh will then be scheduled right after this transaction, i.e. the refresh\_stall will be added to the next\_refresh variable. After a stalled refresh, the next\_refresh variable will be updated adding a refresh period and subtracting the value of the refresh\_stall variable. This keeps the refresh period constant on average.

### The mctrl.tpp File

The ‘mctrl.tpp’ file implements all the member functions of the Mctrl class, including constructor, destructor, and the TLM transport functions.

#### Construction and Initialization of the module

The construction and the initial configuration of the MCTRL unit is carried out in three places:

1. The **constructor** sets the generics and configures the PnP settings, the gr\_device and the bus interface. In addition, the constructor builds a GreenReg register container ‘r’, in which it implements all the registers listed in Table 4. The register container is a C++ class implemented in the GreenReg libraries that provides memory management and interface functions. Within this register container, the GreenReg registers are instantiated and initialized with their default values during instantiation of the MCTRL unit. The default values of the registers are stored as constants in the class definition in the mctrl.h file.
2. The **built-in SystemC function end\_of\_elaboration()** is used for hooking the callback functions to the according registers. This is done after elaboration, when all registers and functions are known to the compiler. The callback functions need to be triggered, when certain bits of the control registers change. To register the callbacks with these bits, specific bit accessors for these bits must be created before registering the sensitivity. This is done with the member function br.create() of the register container. When the bit accessors are defined, the callbacks can be registered by subsequently calling the GreenReg macros GR\_FUNCTION and GR\_SENSITIVE.
3. The **reset\_mctrl()** function is called at the end of the end\_of\_elaboration() function to update and finalize the initial configuration. Although hard-coded default values are already present in the configuration registers, these are overrule by the generics and might therefore be updated. Especially the permission of SDRAM and mobile SDRAM has to be granted or denied. In addition all delay variables are reset to zero, the length of the refresh cycle is read from the registers, and the address space variables are calculated from the generics.

These tasks are performed in a separate reset function, not in the end\_of\_elaboration() function, because the reset\_mctrl() function is also registered as a callback to the reset input signal of the module. That way, the common practice of initialization through an initial reset is applied for the MCTRL unit.

#### Configuration of the module

After initialization, some of the settings in the configuration of the MCTRL unit can be changed dynamically. The dynamic configuration is generally performed by writing to the configuration registers. When the configuration registers change, callback functions react to these changes and perform the necessary configuration operations. The following callback functions are implemented:

* The **configure\_sdram callback function** reacts to the TCAS field of the MCFG2 register and the DS, TCSR, and PASR fields of the MCFG4 register, all of which require an immediate LMR or EMR command. The LMR and EMR commands are assumed to be issued at the same time as the change of the above-mentioned fields. The operation of the MCTRL unit is adapted to the changed register values and the memory model is assumed to operate accordingly after the delay induced by the LMR or EMR command. This delay is added to the callback\_delay variable.
* The **launch\_sdram\_command callback function** reacts to the SDRAM\_COMMAND field of the MCGF2 register. According to the field value being set to 01, 10, or 11, a PRECHARGE, AUTO-REFRESH, or LMR/EMR command can be forced. As LMR/EMR are assumed to be issued when required and only when required, these commands are ignored. AUTO-REFRESH and PRECHARGE are modeled by adding their delay to the callback\_delay variable in the LT model. In any case the SDRAM\_COMMAND field is cleared by this callback.
* The **erase\_sdram callback function** reacts to a change of the mode of operation of the SDRAM device. If SDRAM is sent to partial array self-refresh or deep power down mode, an according erase command is sent to the SDRAM device. As this command is not provided by the generic payload, the ext\_erase extension is appended to the transaction payload. The memory model checks for this extension and calls a software function to erase the according parts of the SDRAM memory area. The memory area to be erased is deduced from the address and data fields of the generic payload, where the address field contains the start address and the data field contains the end address.
* The **sram\_disable, sdram\_enable, sram\_change\_bank\_size and sdram\_change\_bank\_size callback functions** react to changes of the MCFG2 register fields SI, SE, RAM\_BANK\_SIZE, and SDRAM\_BANKSZ respectively. All the functions perform a complete recalculation of the ram address space variables. Several of these functions need to recalculate the SRAM bank address variables. To prevent the multiplication of this lengthy code, it has been outsourced into the sram\_calculate\_bank\_addresses function.

#### Operation of the module

According to the configuration and operating mode, the b\_transport function reacts to incoming transactions from the bus master. First of all, the address field of the generic payload is analyzed and compared to the bank address variables to determine the memory type that has to be accessed. If the according memory is configured to operate in any access mode other than 32 bit, the streaming width of the generic payload is adapted to this setting. The streaming width must be reset to 4 Byte by the memory device.

In the next step, the command field is analyzed to calculate the correct delay for a read or write transaction. Finally, the delay is added and the transaction payload is forwarded to the memory using the according socket. In case of access failure, e.g. write access to read-only PROM, the transaction payload is not forwarded, an error response is given, and only the decoding delay is added to the delay variable.

## Parametrization Options

In the VHDL implementation, the parameterization is fully controlled by generics, which are supported as constructor parameters in the SystemC module. The parameters are summarized in Table 5. The shadowed generics have been removed in the TLM implementation.

|  |  |  |  |
| --- | --- | --- | --- |
| **Parameter** | **Function** | **Allowed Range** | **Default** |
| hindex | AHB slave index | 0 to  NAHBSLV–1 | 0 |
| pindex | APB slave index | 0 to  NAPBSLV–1 | 0 |
| 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 |
| invclk | Inverted clock is used for SDRAM | 0 – 1 | 0 |
| fast | Enable fast SDRAM address decoding | 0 – 1 | 0 |
| romasel | log2(PROM address space size) – 1 | 0 – 31 | 28 |
| sdrasel | log2(RAM address space size) – 1 | 0 – 31 | 29 |
| 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 |
| oepol | Select polarity of drive signals for data pads (0 = active low, 1 = active high) | 0 – 1 | 0 |
| 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 |
|  |  |  |  |

Table 5 – MCTRL Template Parameters

## Interface

The interface of the MCTRL unit comprises the means to APB bus communication, AHB bus communication, and communication with the memory devices.

### APB Bus Communication

The MCTRL configuration registers MCFG1 – MCFG4 listed in Table 4 can be accessed via a TLM socket. As the callbacks described in section 4.2.2 are hooked to these registers, they must be part of a GR\_DEVICE and are addressed via a GreenReg compatible socket. The base address and the size of the register file can be calculated from the paddr and pmask generics.

### AHB Bus Communication

All requests of memory accesses are received from the AHB. An AHB slave socket of the MCTRL unit listens at the entire memory address space of the AHB. A generic payload received by this socket is prepared for further use and forwarded to the memory device indicated by the address field of the payload.

### Memory Device Interface

The MCTRL unit provides four TLM simple initiator sockets, one for each memory device. The sockets are named mctrl\_rom, mctrl\_io, mctrl\_sram, and mctrl\_sdram. According to the address field of a generic payload received on the AHB socket, the transaction is forwarded over the related initiator socket.

For the SDRAM communication, a payload extension is defined in the MCTRL unit. The extension indicated that a specific command, namely erase\_sdram, has to be executed.

## Compilation Instructions

For the compilation of the MCTRL unit, a WAF wscript file is provided and integrated in the superordinate build mechanism of the TLM model library of the Hardware-Software SystemC Co-Simulation SoC Validation Platform project.

The libraries required for the compilation are:

 1 #include **<algorithm>**  
 2 #include **<iostream>**  
 3 #include **<boost/config.hpp>**  
 4 #include **<systemc.h>**  
 5 #include **<tlm.h>**  
 6 #include **<greenreg.h>**  
 7 #include **<greenreg\_ambasocket.h>**  
 8 #include **"greencontrol/all.h"**  
 9 #include **"tlm\_utils/simple\_initiator\_socket.h"**  
10 #include **"mctrlreg.h"**  
11 #include **"generic\_memory.h"**  
12 #include **"grlibdevice.h"**

## Example Instantiation

The following example shows how to instantiate the MCTRL TLM. The AHB and APB sockets are connected to a testbench that behaves like an AHB and APB master. The simple initiator sockets are connected to instances of the Generic\_memory module described in chapter 5.

 1 **int** sc\_main(**int** argc, **char**\*\* argv) {  
 2   //set generics  
 3   **const** **int** hindex = 0;  
 4   **const** **int** pindex = 0;  
 5   **const** **int** romaddr = 0;  
 6   **const** **int** rommask = 3584;  
 7   **const** **int** ioaddr = 512;  
 8   **const** **int** iomask = 3584;  
 9   **const** **int** ramaddr = 1024;  
10   **const** **int** rammask = 3072;  
11   **const** **int** paddr = 0;  
12   **const** **int** pmask = 4095;  
13   **const** **int** wprot = 0;  
14   **const** **int** srbanks = 4;  
15   **const** **int** ram8 = 0;  
16   **const** **int** ram16 = 0;  
17   **const** **int** sepbus = 0;  
18   **const** **int** sdbits = 32;  
19   **const** **int** mobile = 0;  
20   
21   //instantiate mctrl, generic memory, and testbench  
22   Mctrl mctrl\_inst0(**"mctrl\_inst0"**, hindex,  pindex,  romaddr, rommask, ioaddr, iomask,  
23                                    ramaddr, rammask, paddr,   pmask,   wprot,  
24                                    srbanks, ram8,    ram16,   sepbus, sdbits, mobile);  
25   Generic\_memory <**uint8\_t**>  generic\_memory\_rom(**"generic\_memory\_rom"**);  
26   Generic\_memory <**uint32\_t**> generic\_memory\_io(**"generic\_memory\_io"**);  
27   Generic\_memory <**uint8\_t**>  generic\_memory\_sram(**"generic\_memory\_sram"**);  
28   Generic\_memory <**uint32\_t**> generic\_memory\_sdram(**"generic\_memory\_sdram"**);  
29   Mctrl\_tb mctrl\_tb(**"mctrl\_tb"**, hindex,  pindex,  romaddr, rommask, ioaddr, iomask,  
30                                 ramaddr, rammask,  paddr,   pmask,   wprot,  
31                                 srbanks, ram8,     ram16, sepbus, sdbits, mobile);  
32   
33   //bus communication via amba sockets (TLM)  
34   mctrl\_tb.apb\_master\_sock(mctrl\_inst0.apb);  //config registers  
35   mctrl\_tb.ahb\_master\_sock(mctrl\_inst0.ahb);  //memory access  
36   
37   //memory communication via simple TLM sockets  
38   mctrl\_inst0.mctrl\_rom(generic\_memory\_rom.slave\_socket);  
39   mctrl\_inst0.mctrl\_io(generic\_memory\_io.slave\_socket);  
40   mctrl\_inst0.mctrl\_sram(generic\_memory\_sram.slave\_socket);  
41   mctrl\_inst0.mctrl\_sdram(generic\_memory\_sdram.slave\_socket);  
42   
43   sc\_core::sc\_start();  
44   **return** 0;  
45 }

# GENERIC Memory SystemC Model

## Functionality and Features

The Generic\_memory model is not based on any reference design from the Gaisler GRLIB, but it was developed from scratch to suit the MCTRL unit described in chapter 4.

The model is generic in a sense that it can be any type of memory and does not know what kind of memory it is. The model is merely used to store data and does not model any delay. 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 universally applicable.

The memory model supports two different word lengths of 32 bit or 8 bit, which can be determined by a template parameter. For each of the word lengths, a read function and a write function is provided. The read and write functions take a data pointer and a variable indicating the data length as arguments, so any amount of data can be read or written at any time.

As an additional feature, the memory model provides a function that can erase a specific region of the memory. This function can only be accessed through appending the ext\_erase extension to the transaction payload. The ext\_erase extension is implemented in the MCTRL unit.

## Internal Structure

The source code of the Generic memory model is split into two file, generic\_memory.h and generic\_memory.tpp.

### The generic\_memory.h File

The header file defines the class template of the memory model. It implements the simple target socket for communication with the MCTRL unit and defines the constructor and read, write, and erase function prototypes. According to the template parameter given at the time of instantiation, a map representing the memory with values of uint8\_t or uint32\_t is instantiated. (The map data type has initially been chosen for addressing reasons, but might be replaced by an array after a performance evaluation that is still pending.) In addition, a prototype of the TLM transport function is defined.

### The generic\_memory.tpp File

The generic\_memory.tpp is technically a header file that is included at the bottom of the generic\_memory.h file. This mechanism is chosen for enabling the separation of module definition and module implementation although the module is a template class.

The generic\_memory.tpp file implements the functions defined in the header file, including the constructor. The constructor names the socket and registers the b\_transport function with this socket.

The b\_transport function then receives any transaction object that is received by the simple target socket. The generic payload is analyzed and one of the read or write functions is called. As a special case, the erase extension is checked first.

The read or write function to be called depends on the template parameter given at instantiation. For an 8 bit map, the 8 bit read and write functions need to be used, for a 32 bit map the 32 bit function are required. Which function to use can partially be deduced from the generic payload and definitely determined by additionally checking the size of a map element. If the streaming width is 8 Byte, SDRAM is accessed and a 32 bit function has to be used. If the streaming width is 2 Byte or 1 Byte, an 8 Bit function is required. If the streaming width is 4 Byte, the size of a map element needs to be checked for definite determination. Read or write access can be distinguished checking the command field of the generic payload.

The read and write functions are trivial and do not need any further explanation.

## Parametrization Options

The element size of the map implementing the memory can be parameterized via a template parameter. Allowed element types are uint8\_t and uint32\_t. For any other data type, the error free operation of the memory is neither precluded nor guaranteed.

## Interface

The Generic\_memory model instantiates a TLM simple target socket that receives any transaction sent from the MCTRL unit. It understands the ext\_erase extension that is defined in the MCTRL unit.

## Compilation Instructions

The compilation of the Generic\_memory integrated in the compilation of the MCTRL test structure and the superordinate build mechanism of the TLM model library of the Hardware-Software SystemC Co-Simulation SoC Validation Platform project.

The libraries required for the compilation are:

1 #include **<map>**  
2 #include **<boost/config.hpp>**  
3 #include **<systemc.h>**  
4 #include **<tlm.h>**  
5 #include **<greenreg.h>**  
6 #include **<greenreg\_ambasocket.h>**  
7 #include **"greencontrol/all.h"**  
8 #include **"tlm\_utils/simple\_target\_socket.h"**

## Example Instantiation

For an example instantiation, please refer to the MCTRL example in section 4.6.

# MMU\_CACHE Cache sub-system systemC module

## Functionality and Features

### Overview

The Cache Sub-System implements the functionality of the Gaisler GRLIB Harvard L1 Cache and the Sparc V8 Reference 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 6. The top-level class *mmu\_cache* provides two TLM2 *simple\_target\_sockets* (icio, dcio) for communication with the CPU and one TLM2 GreenSocs *amba\_master\_socket* for the connection of an AMBA AHB bus. All the sub-components, such as the mmu, the caches and the localrams are implemented in plain C++ and can be accessed through their member functions (APIs).

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 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 6 - Structure of Cache Sub-System

### Address Space Identifiers (ASI)

SPARC processors can generate an 8-bit address space identifier (ASI), to provide access to up to 256 separate 32-bit address spaces. These ASIs are heavily used to control the cache sub-system. A list of the ASIs supported by the TLM model is given in Table 6.

|  |  |  |
| --- | --- | --- |
| 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 6.1.4 | Access instruction cache tags |
| 0x0d | - “ - | Access instruction cache data |
| 0x0e | - “ - | Access data cache tags |
| 0x0f | - “ - | Access data cache data |
| 0x10 | - “ - | Flush instruction cache |
| 0x11 | - “ - | 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 6 - Supported ASIs

ASIs are emitted by the data interface of the processor. For this purpose an optional payload extension has been linked to the data cache payload (*dcio\_payload\_extensions*). For more information about payload extensions see section 6.1.5.

The ASIs are decoded in the transport function of the dcache in/out socket (dcio), which is located in the top-level module (*mmu\_cache)*. The decoder maps the ASIs to API functions of the corresponding sub-components (caches, mmu). The API functions are described in section 6.4.

### 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 7) 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 8). The CRs are read-only. At system boot they can be accessed to determine structure and size of the caches.

|  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 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 7 - 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 (todo).

[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 8 - 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 through ASI 0x19 (Table 9). All those registers are implemented within the 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 9 - 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.

The full set of options concerning the MMU Control Registers is described in the Sparc V8 Reference Manual [RD06].

### Diagnostic Access

Most of the internal data structures of Cache and MMU can be accessed for diagnostic purpose through dedicated ASIs.

The Tag and Data RAM of instruction and data cache can be read and written using ASI 0x0C – 0x0F (see section 6.1.2). 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 TLM2 generic payload. The extensions are modelled in two classes:

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

dcio\_payload\_extension.h/cpp optional 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 6.1.6.

The dcio extensions additionally contain an ASI attribute, which is also modeled as a 32 bit unsigned integer. Purpose and usage of address space identifiers has been explained in 6.1.2.

### 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. Moreover, a set of macros has been defined in order to ease their handling. The encoding of the debug extension is shown in Table 10. The macro definitions can be found in the defines.h files 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 10 - 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)

The Debug Extension is extensively used in all testbenches of the Cache Sub-System. The mechanism is straight forward as shown below:

dwrite(addr, data, length, asi, flush, flushl, lock, debug);

assert(CACHEWRITEMISS\_CHECK(\*debug));

The dwrite method is a testbench function, which sends a write transaction to the data socket (dcio) of the cache. The last argument of the function is an unsigned int pointer (debug). During execution of the transaction the components of the Cache Sub-System update the pointed debug field. Depending on the state of the system different bits are set. With this information the expected behavior can be checked after return of the transaction.

In the example above the CACHEWRITEMISS\_CHECK macro is connected with an assertion to make sure that the transaction was a write miss.

## Internal Structure

This section briefly describes the internal structure of the Cache-Subsystem SystemC IP. Detailed descriptions as well as inheritance and collaboration diagrams can be found in the doxygen documentation.

The Cache-Subsystem IP is formed by multiple source files, which can be found in the models/mmu\_cache/lib directory.

### The defines.h file

This file contains all globally relevant macro and data type definitions. It defines the structure of the cacheline (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 tag (t\_VAT).

Moreover, the file contains macros for unified formatting of debug output (DUMP), as well as SET and CHECK methods for handling of the debug payload extension described in section 6.1.6.

### The payload\_extension files

The *mmu\_cache* model features two *tlm::simple\_target\_sockets* for connection of the cpu instruction and data ports. These connections implement a simple point-to-point communication, which can be realized, widely relying on TLM2 generic payload. Only a few optional payload extensions are required.

The payload extensions for the instruction cache input/output socket (icio) are implemented by 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 by 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 Cache Sub-System (Figure 7).

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) {};

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

The same interface is used to e.g. access the ahb master interface in class *mmu\_cache* or the *tlb\_adapters* of the *mmu*. This way cache misses can be easily redirected depending on configuration options (late binding).



Figure 7 - Generic Memory Interface / Dependencies

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

The *mmu\_cache\_if* class extends the *mem\_if* class by member functions that provide access to the Cache Control Register. The Cache Control Register is implemented at the top-level of class *mmu\_cache.* Many sub-components, such as the caches and the mmu need to read/write the ahb master interface of *mmu\_cache* or execute checks on the Cache Control Register. These components expect a pointer of type *mmu\_cache\_if* as a constructor argument.

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

The files declare and implement the top-level class of the Cache Sub-System. The class *mmu\_cache* inherits the abstract *mmu\_cache\_if* interface and holds a set of pointers to various sub-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*

The sub-components are dynamically created in the constructor of the class. The instantiation depends on parametrization options (see 6.3). In case a certain module is not required, a NULL pointer will be assigned. If the mmu is enabled, the caches will use the memory interfaces (mem\_if) of the instruction and data tlb adapters for miss processing. Otherwise they are directly connect to the memory interface of *mmu\_cache*.

Next to the AHB master, class mmu\_cache contains two TLM2 slave sockets for connection to the processor (Table 12).

|  |  |  |
| --- | --- | --- |
| 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 11 - TLM Sockets Cache Sub-System

The class implements blocking transport functions for both TLM2 slave sockets. The transport functions decode the payload and translate the request into function calls to the sub-component APIs.

The transport function of the icio instruction cache socket is very compact. After extracting the payload, the *tlm\_command* attribute is checked. TLM read requests are translated into calls to the *read* API function of the icache or ilocalram respectively. TLM write requests are ignored.

The dcio data cache socket transport function is more deeply structured. In contrast to the icio socket, the dcio payload supports address space identifiers (ASIs – see 6.1.2). The ASIs are implemented as an optional extension to the TLM2 generic payload. Depending on the ASI the dcio transport function calls functions from different sub-component APIs. Default cache access is performed for ASI 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. Next to TLM read request, TLM write requests are supported.

The class also contains the Cache Control Register, which is implemented as a 32bit unsigned integer (see Table 7).

### The vectorcache.h/cpp files

The files vectorcache.h and vectorcache.cpp define a common base class for the implementation of instruction and data caches. Almost all the functionality required by both cache types is implemented here.

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

The API consists of a set of protected member functions, which are ougth to be published by child classes. In the following these functions are briefly explained:

To be called for ordinary load operations. The address parameter will be split into a cache tag and a cache index portion. Afterwards, the cache line with the respecting index is loaded and checked. If the tag in one of the sets matches 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.

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

The write function is supposed to be called for ordinary store operations. The address parameter will be split into a cache tag and a cache index portion. Afterwards, the cache line with the respecting tag is loaded and checked. If the tag in one of the sets matches 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 with the write data. 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 6.1.4).

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 must be defined 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. All interface functions are made public. The write function is overwritten, because the instruction cache is not writable. A call to the write function will emit an error message and stop the simulation.

The class implements the virtual check\_mode function. 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. All interface functions are made public.

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. The actual memory is implemented as a c++ std::vector. The class inherits the generic memory interface (mem\_if) and provides implementations for reading and writing data.

### The mmu.h/cpp files

The files implement the memory management unit of the Cache Sub-System. 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 STL std::map containers. The key for a TLB lookup is a virtual address tag (t\_VAT). The caches connect to the mmu through tlb\_adapter objects (section 6.2.11). 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 arguments. In the function body, 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 14):

|  |  |  |  |
| --- | --- | --- | --- |
| 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 12 - 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 the ASI 0x19 (Table 9). Respective 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 *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*.

## Parametrization Options

The Cache-Subsystem can be parametrized through the constructor parameters of the top-level class (*mmu\_cache*). The parameters shown in Table 15 directly refer to VHDL generics with the same name.

|  |  |
| --- | --- |
| Parameter | Description |
| dsu | Enable debug support unit interface (not implemented) |
| 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 |

Table 13 - Constructor Configuration Parameters

The constructor parameter shown in Table 16 are timing or simulation related.

|  |  |
| --- | --- |
| Parameter | Description |
| name | Name of the SystemC component |
| id | ID of the AHB master interface |
| icache\_hit\_read\_response\_delay | Delay of an icache read-hit |
| icache\_miss\_read\_response\_delay | Delay of an icache read-miss |
| dcache\_hit\_read\_response\_delay | Delay of a dcache read-hit |
| dcache\_miss\_read\_response\_delay | Delay of a dcache read-miss  (without AHB delay) |
| dcache\_write\_response\_delay | Delay of a dcache write-hit and write-miss  (without AHB delay) |
| itlb\_hit\_response\_delay | Delay of an ITLB hit |
| itlb\_miss\_response\_delay | Delay of an ITLB miss  (without AHB delay for table walk) |
| dtlb\_hit\_response\_delay | Delay of a DTLB hit |
| dtlb\_miss\_response\_delay | Delay of a DTLB miss  (without AHB delay for table walk) |

Table 14 - Constructor Simulation Parameters

## Interface

The interface of the module comprises three TLM2 sockets.

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

## Compilation Instructions

For the compilation of the Cache-Subsystem IP, a WAF wscript is provided and integrated in the superordinate build mechanism of the TLM model library of the Hardware-Software SystemC Co-Simulation SoC Validation Platform project.

The compilation flow creates a static library of the IP model, which can be linked against testbenches or platform simulations.

Currently the following testbenches are available:

|  |  |
| --- | --- |
| Name / Target | Description |
| lt\_ct\_default\_test  *waf –target lt\_ct\_default.test* | Tests most of the fundamental functionality of the cache e.g. instruction/data read/write hit/miss, diagnostic read/write of cache tags and entries, cache flushing, sub-word access, cache bypass mode and cache freeze.  Configuration summary: caches enabled, mmu disabled, localrams disabled, 4 instruction and data cache sets, 1kb instruction and data cache set size, 1 word per cacheline\*, caches random replacement |
| lt\_ct\_localram\_test  *waf –target lt\_ct\_localram.test* | Tests the cache in a different configuration (2 large sets) and the localrams at maximum size (512kb each).  Configuration summary: caches enabled, mmu disabled, localrams enabled, 2 instruction and data cache sets, 256kb instruction and data cache set size, 512kb instruction and data localram size, 1 word per cacheline\*,  caches random replacement |
| lt\_ct\_lock\_test  *waf –target lt\_ct\_lock.test* | Test for caches with 3 sets. Main purpose is the test of different replacement strategies (LRU, LRR). Todo: add vectors for cache line locking  Configuration summary: caches enabled, mmu disabled, localrams disabled, 3 instruction and data cache sets, 8kb instruction and data cache set size, 4 words per cacheline,  icache LRR replacmenet, dcache LRU replacement |
| lt\_ct\_mmu\_test  *waf –target*  *lt\_ct\_mmu.test* | Test for mmu and tlbs. The test initializes a 3-level page table in main memory. Afterward instruction and data tlb hits/misses are simulated and verified.  Configuration summary: caches enabled, mmu enabled, localrams disabled, 4 instruction and data cache sets, 1kb instruction and data cache size, 1 word per cacheline\*, 8 instruction and data tlbs, 4kB mmu page size |
| lt\_ct\_cacheline\_test  *waf –target*  *lt\_ct\_cacheline.test* | Tests random replacement and instruction burst fetch on cache configuration with 8 words per line (max).  Configuration summary: caches enabled, mmu disabled, localrams disabled, 2 instruction and data cache sets, 64kb instruction and data cache size, 8 words per instruction and data cache line, caches random replacement |

Table 15 - Tests Cache Sub-System

\*Configuration not supported by hardware model.

All tests are located in the *models/mmu\_cache* directory.

## Example Instantiation

The following example demonstrates the instantiation of the *mmu\_cache* module (e.g. within a sc\_main()) and the correct binding of the TLM sockets.

*Instantiation of mmu\_cache with example configuration:*

 1 mmu\_cache m\_mmu\_cache(  
 2         1,             // icen: instruction cache enabled  
 3         3,             // irepl: instruction cache random replacement  
 4         1,             // isets: 2 instruction cache sets  
 5         3,             // ilinesize: 8 words per instruction cache line  
 6         6,             // isetsize:       64 kb per instruction cache set  
 7         0,             // isetlock: no instruction cache locking  
 8         1,             // dcen: data cache enabled  
 9         3,             // drepl: data cache random replacement  
10         1,             // dsets: 2 data cache sets  
11         3,             // dlinesize: 8 words per data cache line  
12         6,             // dsetsize: 64 kb per data cache set  
13         0,             // dsetlock: no data cache locking  
14         0,             // dsnoop: no data cache snooping  
15         0,             // ilram: disabled  
16         0,             // ilramsize: 1kb  
17  0x8e,    // ilramstart: 0x8e000000 start address of ilram (disabled)  
18         0,             // dlram: disabled  
19         0,             // dlramsize: 1kb  
20  0x8f,            // dlramstart: 0x8f000000 start address of dlram (disabled)  
21         0,             // cached: no cachability mask  
22         0,             // mmu\_en: no mmu  
23         3,             // itlb\_num: 8 ITLBs (disabled)  
24         3,             // dtlb\_num: 8 DTLBs (disabled)  
25         0,             // tlb\_type: separate instruction and data TLBs (disabled)  
26         1,             // tlb\_repl:       random TLB replacement  
27         0,             // mmupgsz: 4kb  
28         **"mmu\_cache"**,   // name: SystemC module name  
29         2,             // id: AHB master ID  
30         sc\_core::sc\_time(0, sc\_core::SC\_NS),   // icache\_read\_hit\_response\_delay  
31         sc\_core::sc\_time(0, sc\_core::SC\_NS),   // icache\_read\_miss\_response\_delay  
32         sc\_core::sc\_time(0, sc\_core::SC\_NS),   // dcache\_read\_hit\_response\_delay  
33         sc\_core::sc\_time(0, sc\_core::SC\_NS),   // dcache\_read\_miss\_response\_delay  
34         sc\_core::sc\_time(0, sc\_core::SC\_NS),   // dcache\_write\_response\_delay  
35         sc\_core::sc\_time(0, sc\_core::SC\_NS),   // itlb\_hit\_response\_delay  
36         sc\_core::sc\_time(0, sc\_core::SC\_NS),   // itlb\_miss\_response\_delay  
37         sc\_core::sc\_time(0, sc\_core::SC\_NS),   // dtlb\_hit\_response\_delay  
38         sc\_core::sc\_time(0, sc\_core::SC\_NS));  // dtlb\_miss\_response\_delay  
   
*Bind AHB master socket to e.g. AMBA\_LT\_CT\_Adapter:*   
 1 mmu\_cache.ahb\_master(ahb\_lt\_ct.slave\_sock);  
   
*Bind icache and dcache interfaces to testbench/processor:*   
 1 tb.instruction\_initiator\_socket(mmu\_cache.icio);  
 2 tb.data\_initiator\_socket(mmu\_cache.dcio);

# GPTIMER General Purpose Timer SystemC model

## Functionality and Features

The GPTimer unit acts as a slave at the APB bus. Its basic functionality is a countdown mechanism that asserts an interrupt on underflow. The GPTimer unit 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 GPTimer unit can be configured and operated through its registers addressed through the APB interface. All registers have a width of 32 bits and are summarized in Table 18.

|  |  |
| --- | --- |
| 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 16 – GPTimer Registers

The prescaler and all the timers 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 on 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. The clock period is stored to the ‘clock\_cycle’ variable, which can be set using one of the overloaded ‘clk’ functions. The triggers for decrementing the counters are ticks issued by prescaler underflow. The prescaler is automatically reset on underflow and cannot be halted. Due to a specific characteristic of the VHDL implementation of the GPTimer unit, the prescaler reload value must be greater than or equal to the number of counters implemented in the GPTimer instance.

The configuration register located at address 0x08 can be used to configure the GPTimer unit. The counter n configuration registers located at addresses 0xn8 can be used to configure the individual counters.

The configuration register consists of four fields, DF, SI, IRQ, and TIMERS. The DF field is the only field that can by modified dynamically, all other fields are read only, i.e. their values are determined by VHDL generics and written to the registers at system startup.

The **DF (disable freeze) field** disables the sensitivity to the dhalt input signal. This signal can be used to freeze the timer value registers, if DF is disabled.

The **SI (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 line is specified in the **IRQ field**. Else, counter 1 asserts the interrupt specified in the IRQ field and all other counters are distributed to the subsequent lines. The highest line must not exceed the maximum number of interrupts in the system. For more information on the interrupt scheme, please refer to chapter 8.

The **TIMERS field** specifies the number of counters in the system.

The counter configuration registers are used to configure and control the counters. The counters are controlled by the enable, load, and debug halt fields. Debug halt freezes the counter value register, load immediately reloads the value register with the contents of the reload value register, and enable can be used to enable or disable the counter.

To increase the counting delay, chaining can be activated for individual counters. If counter n is chaining mode, it does not decrement on prescaler ticks, but on ticks generated by an underflow of the previous counter (n-1). For this operating mode, counter (n-1) must be in restart mode, i.e. its value register is automatically reloaded from the reload value register on underflow. In addition, the interrupt assertion of a counter can be disabled, which would be reasonable for counter (n-1) in the described example. It is possible to enable chaining for multiple counters to wait for very long periods.

In addition, it is possible to configure the last counter as a watchdog using the wdog generic. This generic can be set to an alternative reload value, which will be used to set and reload the counter. The watchdog counter will be started on timer reset and cause an assertion of the wdog output on underflow.

## Internal Structure

The TLM implementation of the GPTimer comprises two classes, CGPTimer and CGPCounter. Implementing the counter unit in a class of its own enables the GPTimer unit to be instantiated with a variable number of counters, which are dynamically instantiated in the constructor of the CGPTimer class. For both classes, the definition is put into header files (gptimer.h, gpcounter.h) and the implementation is put into c++ source files (gptimer.cpp, gpcounter.cpp). The contents of these files are described in the subsequent sections.

### The gptimer.h file

The ‘gptimer.h’ file contains the module class definition. Any communication with the environment is performed through the CGPTimer class defined in this file. The Counters are fully encapsulated in the Timer module.

#### Parameterization of the module

The parameterization options, implemented as generics in the VHDL model, are realized as constructor parameters of the CGPTimer class. This makes the module parametrizable during instantiation. Details on the parameters are given in section 7.3.

#### Configuration of the module

The GPTimer unit is configurable through its Timer configuration register and its Counter configuration registers. The configuration registers, which are accessible through the APB bus, are modeled and accessed through the comfortable mechanisms provided by GreenReg. To ensure GreenReg compatibility, the CGPTimer class needs to be a child module of a GreenReg 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. Thus, the CGPTimer class inherits the gr\_device class.

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.

#### Communication with the module

Apart from the APB communication directed to the registers of the GPTimer, the module is equipped with five signals for direct communication with the master devices.

* The **rst input** signal triggers the reset function do\_reset of the module….
* The **dhalt input** signal is a debug signal that can be used to freeze the counters. This signal can be deactivated through setting the DF bit in the configuration register.
* The **tick output** signal is used for debugging purposes only. Whenever a tick is generated by the prescaler or an underflow of any counter occurs, the tick signal will be set to a number assigned to this instance (1 – prescaler; 2..n+1 – Counters).
* The **IRQ output** signal is used to launch an interrupt on counter underflow. The according interrupt line will be provided as the value of this uint32\_t type signal.
* The **wdog output** signal is required if the timer is used as a watchdog. The signal will then be asserted on underflow of Counter 1.

#### Operation of the module

The CGPTimer class definition contains the module interface and the function prototypes of constructor, destructor, SystemC proesses, callback functions, and pure C++ software routines. The GPTimer unit needs to assert interrupt signals at the correct points of time and therefore needs an SC\_THREAD process to keep track of time. A second SC\_THREAD is used for debug only and is disabled by default.

The SystemC processes have to be registered with the SystemC simulation kernel using the SystemC macro, SC\_HAS\_PROCESS(). In a similar fashion, the callback functions, which are hooked to the registers built with GreenReg, are registered with the SystemC simulation kernel using the GreenControl macro, GC\_HAS\_CALLBACKS().

In addition, some class attributes are defined to keep track of the overall state of operation of the module:

* A lasttime variable stores the last timestamp at which the value of the prescaler has been know. This time is required as a reference for any calculation of ticks.
* A lastvalue variable stores the contents of the prescaler value register at the time stored in lasttime. The prescaler value is known when it is calculated With the information given in lasttime and lastvalue it is possible to calculate the next tick.

## Parametrization Options

The model can be parametrized through the constructor arguments of class timer. All available options are listed in Table 19.

|  |  |
| --- | --- |
| Parameter | Description |
| name | The name of the SystemC instance |
| ntimers | 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. |

Table 17 - GPTimer Parameters

## Interface

The control registers of the module can be accessed through a GreenSocs APB slave socket. In addition, the module provides a set of SignalKit sockets. All socket are implemented in the Timer top-level class.

|  |  |  |  |
| --- | --- | --- | --- |
| Name | Type | In/Out | Description |
| rst | bool | in | reset prescaler and all counters |
| dhalt | bool | in | debug halt |
| tick | uint8\_t | out | muxed ticks of prescaler (bit 0) and all counters (bit 1-7) |
| irq | uint32\_t | out | interrupt lines |

Table 18 - Timer SignalKit sockets

## Compilation Instructions

For the compilation of the Timer IP, a WAF wscript is provided and integrated in the superordinate build mechanism of the TLM model library of the Hardware-Software SystemC Co-Simulation SoC Validation Platform project.

## Example Instantiation

*Instantiation of Timer with 4 Counters:*

 1 Timer dut(**"timer"**, 4);  
 2   
 3 Bind APB socket:  
 4   
 5 tb.master\_sock(dut.bus);  
 6   
 7 Bind SignalKit ports:  
 8   
 9 dut.rst(tb.rst);  
10 dut.dhalt(tb.dhalt);  
11 tb.tick(dut.tick);  
12 tb.irq(dut.irq);

# IRQMP Interrupt Controller SystemC model

## Functionality and Features

### Overview

The functionality of the TLM implementation of the IRQMP unit is equivalent to that of the Gaisler GRLIB VHDL implementation described in RD04. It is compliant with the GRLIB interrupt (IR) scheme described in RD05.

The GRLIB IR scheme comprises a 32-bit IR bus, which is routed in parallel to the AMBA bus signals. The 16 LSBs of the IR bus form the vector of regular IRs, which can be handled by the LEON III core. The 16 MSBs of the IR bus form the extended IR (EIR) vector applicable for systems that require more than 16 IRs. The EIR handling will be explained later in this section.

The AHB and APB units may assert IR lines. The assertions on each line are disjunctively combined by the bus controller and monitored by the IRQMP unit. After prioritization and masking, the IRQMP unit forwards the IRs to the according processors.

The IRQMP unit supports multi-processor systems with up to 16 LEON3 cores. Two different modes of interrupt forwarding are provided:

1. 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).
2. 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 of the IRQMP unit is not pipelined, i.e. all operations can be performed within one clock cycle. The behavior can be configured setting the registers summarized in Table 21. All registers have a width of 32 bits.

|  |  |
| --- | --- |
| 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 19 – IRQMP Registers

### 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 1 or level 0. 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. The determination of which cores will receive the interrupt request (IRQ) depends on the Broadcast Register and the Interrupt Mask Registers of the individual cores. In the Scheme of interrupt distribution, as shown in Figure 8, the use of the IR Pending or IR Force Registers is determined by the Broadcast Register.

Figure 8 – 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 has to be acknowledged (i.e. the ISR has to be processed) by each of the cores. This is realized by setting the Interrupt Force Registers for all cores. Each core has to clear its Interrupt Force register separately.

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 realized by setting the Interrupt Pending Register, which can be cleared by any of the cores. In uniprocessor 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 maskable by the LEON3 core and should be used with care. Most operating systems do not safely handle this IR.

### Extended Interrupt Handling

Extended interrupts are implemented in a cascaded fashion, 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

[This section is copied from RD04]

The processor status can be monitored through the Multiprocessor Status Register. The STATUS field [15..0] in this register indicates if a processor is halted (‘1’) or running (‘0’). A halted processor can be reset and restarted by writing a ‘1’ to its staus field. After reset, all processors except processor 0 are halted. When the system is properly initialized, processor 0 can start the remaining processors by writing to their STATUS bits.

## Internal Structure

The source code is split into three files, ‘irqmpreg.h’, ‘irqmp.h’, and ‘irqmp.tpp’.

### The irqmpregisters.h File

The ‘irqmpreg.h’ file contains preprocessor definitions of register addresses and bit masks only. 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.

The default masks are written to the registers in the system-reset function.

### 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. Details are given in section 8.3.

To ensure GreenReg compatibility, the Irqmp module needs to be a child module of a GreenReg 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. Thus, the Irqmp class inherits the gr\_device class.

The Irqmp class definition contains the module interface and the function prototypes of constructor, destructor, and callback functions. Processes are not used in the module.

SystemC processes are registered with the SystemC simulation kernel using the SystemC macro, SC\_HAS\_PROCESS(). In a similar fashion, the callback functions, which are hooked to the registers built with GreenReg, are registered with the SystemC simulation kernel using the GreenControl macro, GC\_HAS\_CALLBACKS().

### The irqmp.tpp file

The ‘irqmp.tpp’ file is technically a header file, which is included at the bottom of the ‘irqmp.h’ file. It implements all the member functions of the Irqmp template class, including constructor and destructor.

The destructor is explicitly defined to unregister the callback functions.

The constructor configures the gr\_device and the bus interface. It constructs a GreenReg register container ‘r’, in which it implements all the registers listed in Table 21. 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 sensitivity of the three SC\_METHOD processes with the simulation kernel. These processes are sensitive to signals that are not part of the standard AMBA interface and therefore cannot be implemented as callback functions hooked to the registers.

The constructor only takes care of building the registers. For hooking the callback functions to the registers, the use of the built-in SystemC function end\_of\_elaboration() is required. It is called at the end of the elaboration process, in which all instances of all models are built and all functions and processes are compiled. So after elaboration, the SystemC simulation kernel is made aware of the callback functions being hooked to the specific registers. This is achieved by subsequently using the GreenReg macros GR\_FUNCTION and GR\_SENSITIVE as shown exemplarily in the following code snippet.

1 GR\_FUNCTION(Irqmp, launch\_irq);  
2 GR\_SENSITIVE(r[PENDING].add\_rule(POST\_WRITE, **"launch\_irq"**, NOTIFY));

This code hooks the ‘launch\_irq’ callback function of the gr\_device ‘Irqmp’ to the register administrated by the register container ‘r’ at the address ‘PENDING’, which has been defined to be 0x04 in a preprocessor directive. The ‘POST\_WRITE’ argument indicates that the callback function is to be called after a write access to the register. The ‘NOTIFY’ argument simply indicates that the function is to be called at every write access without any conditions or parameters. ‘POST\_WRITE’ and ‘NOTIFY’ are the only options used with the ‘add\_rule()’ function within the IRQMP module.

The member functions of the module merely implement its functionality as described in Section 8.1 and therefore do not need to be explained explicitly. There are no interdependencies between the functions. The function names have been chosen to be self-explanatory.

As the IRQMP unit consists of combinational logic only (except for the interface registers), the implementation of the temporal behavior is rather simple. We assume that each operation will be completed within one clock cycle. Hence, there is no use of different implementations for LT and AT modeling.

For the callback functions, the delay of one clock cycle can simply be achieved by using ‘GR\_DELAYED\_SENSITIVE’ instead of ‘GR\_SENSITIVE’ during the callback registration within the ‘end\_of\_elaboration()’ function.

For the SC\_METHOD processes, the delay is modeled by temporarily overriding the static sensitivity of the methods by using the SystemC ‘next\_trigger()’ function with a constant delay. That way the SC\_METHOD is always called twice – once to model the delay and once to model the functionality. As no wait statements are required, this way of modeling allows maximum simulation performance.

## Parametrization Options

In the VHDL implementation, the parameterization is fully controlled by generics, which are supported as constructor parameters in the SystemC module. The parameters are summarized in Table 22.

|  |  |  |  |
| --- | --- | --- | --- |
| **Parameter** | **Function** | **Allowed Range** | **Default** |
| 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 |

Table 20 - Template Parameters

## Interface

The interface of the IRQMP unit can be divided in two parts, APB bus communication and direct processor communication.

### APB Bus Communication

The APB bus communication mainly consists of the registers listed in Table 21. In addition, the reset signal and the irq\_in signal is implemented to model the input vector of the interrupt lines from the bus.

All registers can be written to configure or operate the IRQMP unit. As the only exception, the Extended Interrupt Identification Register is a read-only register. 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. Using this way of clearance, an EIR can also be cleared by software. The SystemC implementation follows the VHDL implementation rather than the manual.

To enable the communication with the registers, the module contains an APB slave socket. The socket can be bound to any compatible APB master socket that may initiate TLM transactions calling the according b\_transport or nb\_transport functions. The register address would then be part of the TLM2.0 generic payload.

### Direct Processor Communication

For immediate access to the reset signals and the IRQ registers of the cores, the IRQMP unit provides an interface for direct processor communication. The interface consists of the signals *cpu\_rst*, *irq\_req*, and *irq\_ack*.

An active signal *irq\_ack* from the core indicates that the IR on the line specified by the signal is acknowledged by the core. The core is recognized automatically by means of the signal kit. The according IR is then cleared from the Interrupt Pending Register or the Interrupt Force Register of this core.

The *irq\_req* signal sends an interrupt request to the cores and contains the pending interrupt line number.

The *cpu\_rst* signal is used to suspend and wake up the core.

## Compilation Instructions

For the compilation of the IRQMP unit, a WAF wscript file is provided and integrated in the superordinate build mechanism of the TLM model library of the Hardware-Software SystemC Co-Simulation SoC Validation Platform project.

The libraries required for the compilation are:

1 #include **<boost/config.hpp>**  
2 #include **<systemc.h>**  
3 #include **<greenreg.h>**  
4 #include **<greenreg\_ambasocket.h>**  
5 #include **"greencontrol/all.h"**  
6 #include **"irqmpregisters.h"**  
7 #include **"signalkit.h"**

## Example Instantiation

The following example shows how to instantiate the IRQMP TLM and connect it to a testbench. To enable the simulation of the system, all signals of the IRQMP module need to be connected to the sc\_main signals.

1 #include **"amba.h"**  
 2 #include **"irqmp.h"**  
 3 #include **"irqmpreg.h"**  
 4 #include **"irqmp\_tb.h"**  
 5   
 6 **int** sc\_main(**int** argc, **char**\*\* argv) {  
 7   //set generics  
 8   **const** **int** buswidth = 32;  
 9   **const** **int** pindex = 0;  
10   **const** **int** paddr = 0;  
11   **const** **int** pmask = 0xFFF;  
12   **const** **int** ncpu = 2;  
13   **const** **int** eirq = 1;  
14   
15   //define irqmp signals  
16   sc\_core::sc\_signal<**bool**>                      rst(**"rst"**);  
17   sc\_core::sc\_signal<l3\_irq\_out\_type>           irqi[ncpu];  
18   sc\_core::sc\_signal<l3\_irq\_in\_type>            irqo[ncpu];  
19   sc\_core::sc\_signal<sc\_dt::sc\_uint<32> >       apbi\_pirq(**"apbi\_pirq"**),  
20                                                 apbo\_pconfig\_0(**"apbo\_pconfig\_0"**),  
21                                                 apbo\_pconfig\_1(**"apbo\_pconfig\_1"**);  
22   sc\_core::sc\_signal<sc\_dt::sc\_uint<16> >       apbo\_pindex(**"pindex"**);  
23   
24   //instantiate testbench and irqmp  
25   irqmp\_tb<buswidth, pindex, paddr, pmask, ncpu, eirq> irqmp\_tb(**"irqmp\_tb"**);  
26   Irqmp<pindex, paddr, pmask, ncpu, eirq> irqmp\_inst0(**"irqmp"**);  
27   
28   //connect testbench with IRQMP: AMBA bus communication via sockets (TLM)  
29   irqmp\_tb.master\_sock(irqmp\_inst0.bus);  
30   irqmp\_tb.rst(rst);  
31   irqmp\_tb.apbi\_pirq(apbi\_pirq);  
32   **for** (**int** i\_cpu=0; i\_cpu<ncpu; i\_cpu++) {  
33     irqmp\_tb.irqi[i\_cpu](irqi[i\_cpu]);  
34   }  
35   
36   //direct connection of all other signals  
37   irqmp\_inst0.rst(rst);  
38   irqmp\_inst0.apbi\_pirq(apbi\_pirq);  
39   irqmp\_inst0.apbo\_pindex(apbo\_pindex);  
40   irqmp\_inst0.apbo\_pconfig\_0(apbo\_pconfig\_0);  
41   irqmp\_inst0.apbo\_pconfig\_1(apbo\_pconfig\_1);  
42   **for** (**int** i\_cpu=0; i\_cpu<ncpu; i\_cpu++) {  
43     irqmp\_inst0.irqi[i\_cpu](irqi[i\_cpu]);  
44     irqmp\_inst0.irqo[i\_cpu](irqo[i\_cpu]);  
45   }  
46   sc\_core::sc\_start();  
47   **return** 0;  
48 }

# SocWire Systemc model

## Functionality and Features

The SocWire IP provides an AHB/SocWire-bridge based on the AHB2SOCW VHDL module developed at IDA.

The AHB2SocWire module is configured via four registers, which are accessible via an APB bus. Moreover, the module works similar to a DMA controller. Transactions are described in transaction descriptors located in a memory at an AHB bus. Data is send over/received from the SocWire link according to these descriptors.

## Internal Structure

The SocWire IP is delivered in four source files, namely: AHB2Socwire.h, AHB2Socwire.cpp, socw\_defs.h and socw\_gp.h.

The AHB2Socwire.h file provides the class declaration of the actual AHB/Socwire-bridge, while AHB2Socwire.cpp holds its implementation.

The header file socw\_defs.h supplies SocWire related data types, methods and constants that might also be useful for applications using the AHB2Socwire module.

The SocWire link utilizes its own custom payload object, which is defined in socw\_gp.h.

## Parametrization Options

The original IP provides several generics, which are, as far as reasonable, also modeled in the TLM model as constructor parameters. These parameters are illustrated in Table 21.

|  |  |  |  |
| --- | --- | --- | --- |
| **Parameter** | **Function** | **Allowed Range** | **Default** |
| nm | SystemC module name | n/a | none |
| paddr | The 12-bit MSB APB address | 0 to 4095 | none |
| pmask | The 12-bit APB address mask | 0 to 4095 | 0 |
| pirq | Interrupt identifier | 0 to 232-1 | 0 |
| socw\_datawidth | Width of the SocWire link | 16, 32 | 32 |
| socw\_speed | SocWire clock period in nano seconds | 1 to 100 | 10 |
| socw\_after64 | FIXME 6.4 us timeout in ns | 1 to6400 | 64 |
| socw\_after128 | FIXME 12.8 us timeout in ns | 1 to 12800 | 128 |
| disconnect\_detection | FIXME Disconnect detection timeout in ns | 1 to 850 | 85 |

Table - SocWire Constructor Parameters

## Interfaces

The module provides several interfaces to communicate with the different parts oft he system. These include the obvious SocWire interface, an AHB master and an APB slave interface.

Furthermore an IRQ signal is provided.

From application perspective the APB interface and the IRQ are the only interfaces of interest.

### APB Slave Interface

An APB slave interface is implemented using the *greenreg* framework in conjunction with *ambasockets*. 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.

FIXME: add register descriptions

|  |  |
| --- | --- |
| **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 - SocWire Control Register Interface

### Interrupt Request

The IRQ signal is a *signalkit* signal of the type *uint32\_t* called *irq*. If an interrupt is generated this signal is driven with the value given as *pirq* to the constructor (default 0) for one clock cycle.

### AHB Master Interface

For reading transaction descriptors and fetching/receiving SocWire payload data the AHB2SOCW module contains an AHB master interface. The respective TLM AHB master socket is named *ahb*.

AHB transactions executed by the AHB2Socwire module are only indirectly influenced from application side by manipulating the module’s control registers.

### Socwire Interface

The SocWire interface is realized with a pair of a *green target socket* and a *green initiator socket*. Since transactions over the socwire link are only controlled indirectly using the mentioned descriptors. Applications should never directly access these sockets.

The TX part is realized with the initiator socket called *master\_socket* and has to be bound to a target socket called *slave\_socket*.

The link utilizes the *socw::Cpayload* objects for transmissions:

## Socwire Payload Object

The custom payload object holds the following attributes:

* **charType:** Type of SocWire character.
  + SOCW\_NULL\_CHAR
  + SOCW\_FCT\_CHAR
  + SOCW\_DATA\_CHAR
* **data:** Pointer to the SocWire payload data.   
  Only valid if *charType*==SOCW\_DATA\_CHAR
* **length:** Length oft he payload data array.
* **eopMarker:** Attached EOP marker.
  + SOCW\_NONE\_EOP
  + SOCW\_EOP
  + SOCW\_EEP
* **hopc:** Contains the number of routing hops the transaction travelled along. Hast o be incremented by every routing element.
* **bytec:** Contains the number of bytes the receiver was actually able to receive. Has to be initialized with 0 by sender.
* **response\_status:** Transactions response status
  + SOCW\_OK\_RESP
  + SOCW\_INCOMPLETE\_RESP
  + SOCW\_GENERIC\_ERROR\_RESP
  + SOCW\_CREDIT\_ERROR\_RESP

The attributes can be modified using the following API methods:

* bool is\_FCT(): Returns true if charType==SOCW\_FCT\_CHAR
* void set\_FCT(): Sets charType to SOCW\_FCT\_CHAR
* bool is\_NULL(): Returns true if charType==SOCW\_NULL\_CHAR
* void set\_NULL(): Sets charType to SOCW\_NULL\_CHAR
* bool is\_dataChar(): Returns true if charType==SOCW\_DATA\_CHAR
* void set\_dataChar(): Sets charType to SOCW\_DATA\_CHAR
* charTypes\_t get\_charType(): Returns charType.
* void set\_charType(charTypes\_t type): Set charType to type.
* unsigned char\* get\_data\_ptr(): Returns data.
* void set\_data\_ptr(unsigned char\* ptr): Sets data to ptr.
* unsigned int get\_data\_length(): Returns length.
* void set\_data\_length(unsigned int le): Sets length to le.
* bool is\_EOP(): Returns true if eopMarker==SOCW\_EOP.
* bool is\_EEP(): Returns true if eopMarker==SOCW\_EEP.
* eopMarkter\_t get\_eopMarker(): Returns eopMarker.
* void set\_eopMarker(eopMarker\_t marker): Sets eopMarker to marker.
* void inc\_hopc(): Increments hopc by one.
* unsigned int get\_hopc(): Returns hopc.
* oid set\_hopc(unsigned int hops): Sets hopc to hops.
* void inc\_bytec(unsigned int inc): Increments bytec by inc.
* unsigned int get\_bytec(): Returns bytec.
* void set\_bytec(unsigned int count): Sets bytec to count.
* bool is\_response\_ok(): Retunrs true if response\_status>0.
* bool is\_response\_error(): Returns true true if response<=0.
* response\_status\_t get\_response\_status(): Returns response\_status.
* void set\_response\_status(response\_status\_t status): Sets response\_status to status.
* std::string get\_response\_string(): Returns response\_status as printable string.

## SocWire Protocol Implementation

### LT-Level

In the LT implementation one transaction over the link corresponds to one TX descriptor. I.E. the data abstraction level is between the *packet* and *exchange* levels described in the socwire standard.

**Routing:** The data array may start with several bytes of routing information. Socwire uses header deletion, i.e. the used routing information is not forwarded to the next node. To avoid copying of data in this implementation the number of hops, which performed header deletion is counted in the *hopc* payload attribute. Every node receiving a transaction shall ignore the first *hopc* bytes of the payload array, because they are considered deleted.

Every node performing routing is required to increment the *hopc* attribute.

**Sending Data:** Upon activating the transmitter the core starts to fetch the first TX descriptor via the AHB interface. If no descriptor could be read or its enable bit is not set, the TA bit in the status register is set and the TX enable bit in the control reg is cleared.

If a valid descriptor is read the intended payload is copied via AHB to the AHB2Socwire core. Due to the data abstraction level this AHB transfer and the following socwire transfer cannot be processed in parallel, as it should be.

Then the data is transmitted over the socwire link. Afterwards the bytec attribute of the payload object is checked in order to recognize credit errors, which are reported by setting the TE bit in the status register if needed.

Furthermore, the TX descriptor is updated to reflect the status of the transaction.

Eventually the send operation is concluded by checking if a socwire packet was completed by the transaction, which would be indicated by setting the TI bit in the status register. Moreover an IRQ is generated if applicable.

If the transmitter or codec are not disabled, this process is repeated until a non-enabled descriptor is read from memory or an error occurs.

**Receiving Data:** Upon calling of b\_transport first the state of the codec FSM is evaluated. In the *ErrorReset* state nothing actually happens, except the time to transfer the intended data over the link is added to the delay argument.

In all other states the payload object is inspected to hold valid data and the transfer delay is added to the delay argument.

Receiving data via the socwire link works as follows (given the link properly initialized and is in Run state): If no already fetched RX descriptor is available an RX descriptor is fetched via the AHB interface. If this operation fails or the descriptor is disabled the RA bit is set in the status register.

Then as many bytes as offered by the RX descriptor or the transaction object requests – whatever number is lower – is transferred via the AHB bus to an address given by the RX descriptor. The number of successfully received bytes is added to the *bytec* payload object attribute.

If the space offered by the RX descriptor is completely used and/or an end of packet marker is detected the RX descriptor is updated in host memory in order to reflect the updated status of the transaction. If updating the RX descriptor fails the RA bit in the status register is set.

Dependent on the type of the end of packet marker the RI, RE or no bit is set in the status register.

The delay of the AHB transfer might be fully added to the delay which is calculated for the socwire transfer, or completely ignored depending on the slave supporting temporal decoupling or not. Again parallelizing AHB and socwire transfers is not possible due to the coarse data abstraction level.

### AT-Level

What is different at AT-Level ?

## Compilation Instructions

For the compilation of the IRQMP unit, a WAF wscript file is provided and integrated in the superordinate build mechanism of the TLM model library of the Hardware-Software SystemC Co-Simulation SoC Validation Platform project.

The libraries required for the compilation are:

FIXME: add lib dependencies

## Example Instantiation

If interrupts are used it has to be made sure a valid value is provided to *pirq* since it defaults to zero, resulting in a IRQ line constantly bound to zero.

FIXME: format source code

#include "AHB2Socwire.h"

// Class instantiaton

CAHB2Socwire u\_ahb2socwire("DUT0", // Name

0xfff, // paddr

0xfff, // pmask

0x4711, // IRQ

32, // Data width

10, // Clock period

64, // after64 timeout

128, // after128 timeout

85); // Disconnect timeout

// Connect sockets

// Connect the APB slave socket to some APB master socket

APB.master\_socket(u\_ahb2socwire.apb);

// Connect the socwire master socket to a socwire slave socket

// (the configuration chosen here shows w.l.o.g. a loopback example)

u\_ahb2socwire.master\_socket(u\_ahb2socwire.slave\_socket);

// Connect the AHB master socket to some AHB slave socket

u\_ahb2socwire.ahb(AHB.slave\_socket);

// Connect the IRQ signal

u\_ahb2socwire.irq(i\_socw\_irq);

# 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.