**Title**

HW-SW SystemC Co-Simulation SoC Validation Platform

**IP User Manual**

Contract No. XX

by

TU Braunschweig, Germany

Braunschweig, 22.03.10

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

**Table of Contents**

1 Introduction 3

1.1 Purpose and Scope 3

1.2 Referenced Documents 3

2 AMBA Kit 3

3 Aeroflex Gaisler GRLIB MCTRL Memory Controller 3

3.1 Functionality and Features 4

3.2 Internal Structure 4

3.3 Parametrization Options 4

3.4 Interface 4

3.5 Compilation Instructions 4

3.6 Example Instantiation 4

4 Memory Model 5

4.1 Functionality and Features 5

4.2 Internal Structure 5

4.3 Parametrization Options 5

4.4 Interface 5

4.5 Compilation Instructions 5

4.6 Example Instantiation 5

5 Harvard L1 Cache 5

5.1 Functionality and Features 5

5.2 Internal Structure 5

5.3 Parametrization Options 5

5.4 Interface 5

5.5 Library Dependencies 5

5.6 Example Instantiation 5

6 SparcV8 Reference MMU 5

6.1 Functionality and Features 5

6.2 Internal Structure 5

6.3 Parametrization Options 5

6.4 Interface 5

6.5 Compilation Instructions 6

6.6 Example Instantiation 6

7 Aeroflex Gaisler GPTIMER General Purpose Timer 6

7.1 Functionality and Features 6

7.2 Internal Structure 6

7.3 Parametrization Options 6

7.4 Interface 6

7.5 Compilation Instructions 6

7.6 Example Instantiation 6

8 Aeroflex Gaisler IRQMP Interrupt Controller 6

8.1 Functionality and Features 6

8.2 Internal Structure 9

8.3 Parametrization Options 11

8.4 Interface 12

8.5 Compilation Instructions 13

8.6 Example Instantiation 13

9 SocWire 14

9.1 Functionality and Features 14

9.2 Internal Structure 14

9.3 Parametrization Options 14

9.4 Interface 14

9.5 Compilation Instructions 14

9.6 Example Instantiation 14

# 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 |
| … | … | … |

Table - Referenced Documents

# AMBA Kit

In chapter 2, the basic features and usage of the AMBA Kit and the AMBA bus interface will be explained and accompanied by a short tutorial. The official documentation of the AMBA kit will be provided and referenced in this chapter. Eventual manual modifications or extensions to the kit will be illustrated.

In case the AMBA kit cannot be used for licensing reasons, IDA will design an AMBA bus model as a backup solution. This model would then be precisely documented in chapter 2 of the UM.

# Aeroflex Gaisler GRLIB MCTRL Memory Controller

Chapters 3 to 11 will describe the IP models. Each of these chapters will be divided into five sections:

## Functionality and Features

This section describes the functionality of the according IP block, summarizing or adopting the specification of the model (e.g. given in the ‘gaisler\_grip’ document). If the IP block can be instantiated in different flavors adding or removing features, these opportunities will be described here. A full list and description of features will be provided.

## Internal Structure

This section provides an overview of the structure of the model source code. It is meant as a guide for developers who need to modify the models themselves.

## Parametrization Options

Most of the models will be parametrizable in some form (e.g. scalability of size, timing annotation, etc.). The according opportunities and options will be described in this section.

## Interface

In this section, the interface of the IP model will be described. The interface comprises the provided sockets and the underlying callback functions. The usage of the sockets, their bindability rules according to the management of extensions to the TLM2.0 generic payload, and the mechanism of how to address the callbacks will be explained.

## Compilation Instructions

A section on compilation instructions will guide the user through the process of organizing, configuring, and compiling the source code. In addition, all the libraries required for compilation of the model will be listed in this section.

## Example Instantiation

An example instantiation in a standard use case will be given and explained in this section. In case specific features require any tricks or offer pitfalls, these will be covered in the example.

# Memory Model

## Functionality and Features

## Internal Structure

## Parametrization Options

## Interface

## Compilation Instructions

## Example Instantiation

# Harvard L1 Cache

## Functionality and Features

## Internal Structure

## Parametrization Options

## Interface

## Library Dependencies

## Example Instantiation

# SparcV8 Reference MMU

## Functionality and Features

## Internal Structure

## Parametrization Options

## Interface

## Compilation Instructions

## Example Instantiation

# Aeroflex Gaisler GPTIMER General Purpose Timer

## Functionality and Features

## Internal Structure

## Parametrization Options

## Interface

## Compilation Instructions

## Example Instantiation

# Aeroflex Gaisler IRQMP Interrupt Controller

## 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 datapath 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 2. 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 – 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 1, the use of the IR Pending or IR Force Registers is determined by the Broadcast Register.

Figure – 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 irqmpreg.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

In the VHDL implementation the bus interface of the IRQMP unit is realized as a set of VHDL records. These records have been automatically translated to C++ structs, which partially implement the interface in the SystemC implementation of the module (see section 8.4). These structs are defined in the ‘irqmp.h’ module header 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 template 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 tp-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, callback functions, and SC\_METHOD processes. SC\_THREAD processes are not used in the module.

The 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().

As the Irqmp class is a template class, the implementation of its member functions needs to be located in the header file. To enable the separation of class definition and class implementation, the implementation file has been renamed from irqmp.cpp to irqmp.tpp and is included at the bottom of the irqmp.h header file.

### 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 2. 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.

r.create\_register( "pending", "Interrupt Pending Register",

0x04,

STANDARD\_REG | SINGLE\_IO | SINGLE\_BUFFER | FULL\_WIDTH,

0x00000000,

IRQMP\_IR\_PENDING\_EIP | IRQMP\_IR\_PENDING\_IP,

32,

0x00

);

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.

GR\_FUNCTION(Irqmp, launch\_irq);

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 template parameters in the SystemC module. The parameters are summarized in

|  |  |  |  |
| --- | --- | --- | --- |
| **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 - 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 2. In addition, the reset signal and the following GRLIB PnP signals can be regarded as bus signals:

* apbi\_pirq is the input vector of the interrupt lines
* apbo\_pconfig\_0 is the APB PnP Identification Register
* apbo\_pconfig\_1 is the APB PnP Bank Address Register
* apbo\_pindex is a control signal indicating to the bus arbiter that it has

addressed the correct component

[The implementation and documentation of the PnP signals may change during the course of the project.]

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 an input vector and an output vector, each containing a struct of signals for each core. The input / output naming convention of these signals is defined from a processor’s point of view. The data type of IRQMP input vector ‘irqi[n]’ is:

struct l3\_irq\_out\_type {

bool intack;

sc\_uint<4> irl;

bool pwd;

bool fpen;

};

An active signal ‘intack’ from the core indicates that the IR on the line specified by the signal ‘irl’ is acknowledged by the core. The according IR is then cleared from the Interrupt Pending Register or the Interrupt Force Register of this core.

The ‘pwd’ signal indicates the power-down status of the core. In the VHDL implementation it is used in the register read access of the Multiprocessor Status Register only. The register itself is not read here, but the contents it should have are calculated from the generics and read from the ‘irqi.pwd’ signals. Hence, the ‘pwd’ signals determine the NCPU least significant bits of the Multiprocessor Status Register.

In the SystemC implementation the ‘pwd’ signals are directly mapped into the Multiprocessor Status Register.

The ‘fpen’ signal is neither documented in RD04 or RD05 nor implemented in the VHDL version of IRQMP. It was therefore left out of the SystemC implementation, too.

The data type of the IRQMP output vector ‘irqo[n]’ is:

struct l3\_irq\_in\_type {

sc\_uint<4> irl;

bool rst;

bool run;

sc\_uint<20> rstvec;

};

The ‘irl’ signal contains the pending interrupt line number.

The ‘rst’ and ‘run’ signals are used to suspend and wake up the core.

The ‘rstvec’ signal contains the reset address of the program count. In the VHDL implementation this signal is constantly pulled down to the default of ‘0x00000’. A similar behavior is implemented in the SystemC model.

## Compilation Instructions

The compilation instructions depend on the build system, which is not fully set up fort he IRQMP unit yet. This section will be written in due course.

## 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.

#include "amba.h"

#include "irqmp.h"

#include "irqmpreg.h"

#include "irqmp\_tb.h"

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

//set generics

const int buswidth = 32;

const int pindex = 0;

const int paddr = 0;

const int pmask = 0xFFF;

const int ncpu = 2;

const int eirq = 1;

//define irqmp signals

sc\_core::sc\_signal<bool> rst("rst");

sc\_core::sc\_signal<l3\_irq\_out\_type> irqi[ncpu];

sc\_core::sc\_signal<l3\_irq\_in\_type> irqo[ncpu];

sc\_core::sc\_signal<sc\_dt::sc\_uint<32> > apbi\_pirq("apbi\_pirq"),

apbo\_pconfig\_0("apbo\_pconfig\_0"),

apbo\_pconfig\_1("apbo\_pconfig\_1");

sc\_core::sc\_signal<sc\_dt::sc\_uint<16> > apbo\_pindex("pindex");

//instantiate testbench and irqmp

irqmp\_tb<buswidth, pindex, paddr, pmask, ncpu, eirq> irqmp\_tb("irqmp\_tb");

Irqmp<pindex, paddr, pmask, ncpu, eirq> irqmp\_inst0("irqmp");

//connect testbench with IRQMP: AMBA bus communication via sockets (TLM)

irqmp\_tb.master\_sock(irqmp\_inst0.bus);

irqmp\_tb.rst(rst);

irqmp\_tb.apbi\_pirq(apbi\_pirq);

for (int i\_cpu=0; i\_cpu<ncpu; i\_cpu++) {

irqmp\_tb.irqi[i\_cpu](irqi[i\_cpu]);

}

//direct connection of all other signals

irqmp\_inst0.rst(rst);

irqmp\_inst0.apbi\_pirq(apbi\_pirq);

irqmp\_inst0.apbo\_pindex(apbo\_pindex);

irqmp\_inst0.apbo\_pconfig\_0(apbo\_pconfig\_0);

irqmp\_inst0.apbo\_pconfig\_1(apbo\_pconfig\_1);

for (int i\_cpu=0; i\_cpu<ncpu; i\_cpu++) {

irqmp\_inst0.irqi[i\_cpu](irqi[i\_cpu]);

irqmp\_inst0.irqo[i\_cpu](irqo[i\_cpu]);

}

sc\_core::sc\_start();

return 0;

}

# SocWire

## Functionality and Features

## Internal Structure

## Parametrization Options

## Interface

## Compilation Instructions

## Example Instantiation