**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

## Functionality and Features

### Overview

The functionality of the TLM implementation of the MCTRL unit reproduces that of the Gaisler GRLIB VHDL implementation described in RD04. The controller reacts as a slave on the AHB bus and controls a memory subsystem comprising up to two memory buses and up to four types of memory. These are PROM, memory mapped I/O devices, SRAM, and SDRAM. The memory devices are addressed within three address spaces for ROM, local I/O, and RAM. The MCTRL unit forwards transaction objects between the AHB bus master and the attached memory devices.

The behavior of the MCTRL unit may be configured through setting its four configuration registers, which are attached to the APB bus and are summarized in Table 2. All registers have a width of 32 bits.

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

Table – MCTRL Registers

### Address Space

The address space is divided in the three parts of PROM, local 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. These constructor parameters will be called generics throughout this document.

The PROM address space is deduced from the generics romaddr and rommask, which define the start address and the size of the PROM address space. The rommask generic is written to the 12 bit-wide Mask field of the GRLIB PnP BAR0 register of the MCTRL unit. The bit mask represents the 12 most significant bits of the memory address allowing to mask the address space with a resolution of 2^(32-12) Bytes, i.e. 1 MByte. As address bits are masked at high level (logic 1), a higher value of the rommask generic will reduce the size of the address space.

The PROM address space covers

(2^12 – rommask) MByte = 4096 – rommask MByte,

which are divided into two PROM banks of equal size.

The local I/O address space is calculated similarly from the ioaddr and iomask generics. However, any subdivision into memory banks is not supported for local I/O.

The RAM address space calculation operates similar to the PROM address space calculation, but is based on the ramaddr and rammask generics. The subdivision of the RAM address space depends on the settings made in the MCFG2 register. The register provides the fields ‘SDRAM enable’ and ‘SRAM disable’ defining the presence of SRAM only, SDRAM only, or both. If the SDRAM enable bit is low, the SRAM disable bit takes no effect.

According to the organization of the RAM address space with respect to the number of banks, bank locations, bank sizes, and – in case of SDRAM – row and column address bits the GRIP user manual (RD04) is full of contradictions. Hence, the address space organization has been implemented as a consistent scheme closest possible to the structure probably intended to be described in RD04. For the default size of 1 GByte of RAM address space, an example of the bank sizes for each configuration is given in Figure 1. The default configuration is Config 2.

Figure – RAM address space

In the SRAM only configuration, the entire RAM address space can be taken up by up to five SRAM banks. The number of SRAM banks is determined by the generic ‘srbanks’, the default value being 4. Banks 1-4 are always located in the lower half of the RAM address space. Their size is scalable between 8 KByte – 256 MByte and is determined by 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 default configuration, SRAM and SDRAM are attached to the MCTRL device. In this case, the lower half of the RAM address space is taken up by up to 4 SRAM banks. SRAM bank5 cannot be present, because two SDRAM banks take up 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. This would be represented by rammask = 0x800.

In the SDRAM only configuration, 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 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 1 dynamically by changing the according bits of the MCFG2 register. The MCTRL unit will react by recalculating the start and end addresses of the RAM 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 code structure, any access to SDRAM would then be redirected to SRAM causing system malfunction.

Remark:

Although dynamic reorganization of the address space regarding attached devices and bank sizes is supported, a practical use for a hardware system seems to be hard to identify. The functionality has been implemented, because the according fields in the configuration register aren’t flagged ‘read only’. However, it is recommended to keep the address ranges static.

Remark:

In addition to rommask and rammask, two generics, romasel and sdrasel, can be used to determine the PROM and RAM address space sizes in the VHDL implementation. Both generics must display an address space size equal to the according address mask generics to guarantee correct operation of the MCTRL unit. For the purpose of a reduction of error-proneness, the romasel and sdrasel generics have been removed from the TLM implementation.

### Memory Access

The memory access of the TLM MCTRL unit is handled in accordance with the type of memory that has to be accessed. The type of memory is determined by decoding the address given by the bus master, i.e. by analyzing the transaction payload. The payload is then eventually modified and forwarded to the memory device.

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 unit 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. In case of disallowed write accesses, the MCTRL unit will abort the transaction and issue a TLM\_COMMAND\_ERROR\_RESPONSE.

A read access to PROM memory takes 2 bus cycles plus 0 – 16 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.

PROM access is possible in 32 bit, 16 bit, or 8 bit mode. The access mode is set in the PROM WIDTH field of the MCFG1 register. While write access will only modify the sub-words given by the address, a sub-word read access will always result in a burst read of a 32 bit word adding a delay of one bus cycle for each additional transmission.

#### Local I/O Access

The local I/O area supports 32 bit access only. A read access takes 2 bus cycles and a write access takes 3 bus cycles. 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. Such a mechanism cannot be implemented in the LT model.

#### SRAM Access

The access to SRAM is similar to the PROM access, the difference being the number of wait states. 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, which are missing in the LT implementation.

#### SDRAM Access

The SDRAM banks can be accessed over a separate bus, if the sepbus generic is set to 1. The separate bus can have a width of 32 bit or 64 bit as determined by the D64 field of the MCFG2 register.

In the RTL model, the SDRAM device is controlled by SDRAM commands. An ACTIVATE command is used to open a row, while a PRECHARGE command closes the row. While a row is open, READ or WRITE access to this row is possible.

A read access is always performed as a page burst access. As a burst access can be interrupted issuing 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. A periodic delay would require an SC\_THREAD of its own, which would substantially slow down the simulation. As the refresh period is at least 780 clock cycles long, while the refresh cycle only takes a maximum of 10 cycles, the overall slow-down of the memory operation caused by refresh will always be less than 1.3%. As an inaccuracy of 10% is tolerated, the refresh mechanism is omitted in the TLM implementation for the benefit of a reduced simulation time.

### 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. Mobile memory is designed for mobile devices and supports several power saving options.

For mobile = 0, mobile memory is not supported. In this case the MS field of MCFG2 will be set to 0 during the initialization phase of the MCTRL unit. This disables any effect of the MCFG4 register, which defines the power saving options and states of mobile memory.

For mobile = 1, mobile memory is supported, but disabled by default. The MS field of the MCFG2 register is set and the ME field of the MCFG4 register is set to 0. All other fields of the MCFG4 register cannot take effect, as long as the ME field 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 3.1.3.4. 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. Despite the fact that refresh is ignored in the TLM model, the power down mode is nevertheless left for at least 16 bus clock cycles after an idle time of the length of a refresh period. The refresh period is defined in the MCFG3 register.

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 source code is split into three files, ‘mctrlreg.h’, ‘mctrl.h’, and ‘mctrl.tpp’.

### The mctrlreg.h File

The ‘mctrlreg.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. 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.

### The mctrl.h File

The MCTRL unit consists of only one class. The ‘mctrl.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 3.3.

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.

For correct attachment to the AHB bus, the MCTRL unit also needs to inherit the amba\_base\_slave 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.

The Mctrl class definition contains the module interface and the function prototypes of constructor, destructor, callback functions, and pure C++ software routines. An SC\_THREAD process for initializing the module is also present.

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().

In addition, some global variables 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.
* A pmode variable is used to indicate the current operation mode of 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 saving 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.

In case of deep power down mode or PASR mode, large parts of the SDRAM memory need to be erased. As an ‘erase’ command is not provided by the generic payload, the sdram\_erase function of the Generic\_memory is triggered by an ignorable extension. The erase extension is also defined in the mctrl.h header file.

### 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.

The destructor is explicitly defined to unregister the callback functions.

The constructor sets the generics and configures the PnP setting, the gr\_device and the bus interface. The SC\_THREAD for initialization is registered with the simulation kernel and does not have a sensitivity list, i.e. it is only processed once after elaboration. In addition, the constructor builds 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, the GreenReg registers are instantiated.

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. The callback are registered by subsequently calling the GreenReg macros GR\_FUNCTION and GR\_SENSITIVE. As the callback functions only have to react to specific fields of the registers, bit accessors for these fields must be created before registering the sensitivity.

In general, the callback functions react to changes in the configuration registers and therefore perform configuration operations. The initial configuration is determined by the generics and register default values and is calculated in the initialize\_mctrl process.

The configure\_sdram callback function reacts to the TCAS, DS, TSCR, and PASR fields, 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 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 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 functions react to changes of the 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.

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 is expected to 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 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 3. 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 – 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 2 can be accessed via a TLM socket. As the callbacks described in section 3.2.3 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:

#include <algorithm>

#include <iostream>

#include <boost/config.hpp>

#include <systemc.h>

#include <tlm.h>

#include <greenreg.h>

#include <greenreg\_ambasocket.h>

#include "greencontrol/all.h"

#include "tlm\_utils/simple\_initiator\_socket.h"

#include "mctrlreg.h"

#include "generic\_memory.h"

#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 4.

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

//set generics

const int hindex = 0;

const int pindex = 0;

const int romaddr = 0;

const int rommask = 3584;

const int ioaddr = 512;

const int iomask = 3584;

const int ramaddr = 1024;

const int rammask = 3072;

const int paddr = 0;

const int pmask = 4095;

const int wprot = 0;

const int srbanks = 4;

const int ram8 = 0;

const int ram16 = 0;

const int sepbus = 0;

const int sdbits = 32;

const int mobile = 0;

//instantiate mctrl, generic memory, and testbench

Mctrl mctrl\_inst0("mctrl\_inst0", hindex, pindex, romaddr, rommask, ioaddr, iomask,

ramaddr, rammask, paddr, pmask, wprot,

srbanks, ram8, ram16, sepbus, sdbits, mobile);

Generic\_memory <uint8\_t> generic\_memory\_rom("generic\_memory\_rom");

Generic\_memory <uint32\_t> generic\_memory\_io("generic\_memory\_io");

Generic\_memory <uint8\_t> generic\_memory\_sram("generic\_memory\_sram");

Generic\_memory <uint32\_t> generic\_memory\_sdram("generic\_memory\_sdram");

Mctrl\_tb mctrl\_tb("mctrl\_tb", hindex, pindex, romaddr, rommask, ioaddr, iomask,

ramaddr, rammask, paddr, pmask, wprot,

srbanks, ram8, ram16, sepbus, sdbits, mobile);

//bus communication via amba sockets (TLM)

mctrl\_tb.apb\_master\_sock(mctrl\_inst0.apb); //config registers

mctrl\_tb.ahb\_master\_sock(mctrl\_inst0.ahb); //memory access

//memory communication via simple TLM sockets

mctrl\_inst0.mctrl\_rom(generic\_memory\_rom.slave\_socket);

mctrl\_inst0.mctrl\_io(generic\_memory\_io.slave\_socket);

mctrl\_inst0.mctrl\_sram(generic\_memory\_sram.slave\_socket);

mctrl\_inst0.mctrl\_sdram(generic\_memory\_sdram.slave\_socket);

sc\_core::sc\_start();

return 0;

}

# Memory 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 desribed in chapter 3.

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 memory map 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.

## Parameterization 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:

#include <map>

#include <boost/config.hpp>

#include <systemc.h>

#include <tlm.h>

#include <greenreg.h>

#include <greenreg\_ambasocket.h>

#include "greencontrol/all.h"

#include "tlm\_utils/simple\_target\_socket.h"

## Example Instantiation

For an example instantiation, please refer to the MCTRL example in section 3.6.

# 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 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, 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 constructor parameters in the SystemC module. The parameters are summarized in Table 4.

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

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.