Gaisler Research IP Core's Manual

Version eval-1.0, May 2005

Jiri Gaisler, Edvin Catovic

Table of contents

1 Introduction

1.1 Scope

This document describes specific IP cores provided by Gaisler Research inside the GRLIB IP library. When applicable, the cores use the GRLIP plug&play configuration method as described in the ‘GRLIB User’s Manual’.

1.2 IP core overview

The tables below lists the provided IP cores and their AMBA plug&play device ID. All cores use vendor ID 0x01 (Gaisler Research).

TABLE 1. Processors and support functions

Name	Function	Device ID	License
LEON3	SPARC V8 32-bit processor	0x003	COM/GPL
DSU3	Multi-processor Debug support unit	0x004	COM/GPL
IRQMP	Multi-processor Interrupt controller	0x00D	COM/GPL
GRFPU	High-performance IEEE-754 Floating-point unit	-	COM
GRFPC	LEON2/3 Controller for GRFPU	-	COM
MFPC	LEON3 Controller for Meiko FPU	-	COM

TABLE 2. Memory controllers

Name	Function	Device ID	License
SRCTRL	8/32-bit PROM/SRAM controller	0x008	COM/GPL
SDCTRL	PC133 SDRAM controller	0x009	COM/GPL
FTSDCTRL	PC133 SDRAM Controller with EDAC	0x009	COM
FTSRCTRL	32-bit PROM/SRAM controller with EDAC	0x051	COM

TABLE 3. AMBA Bus control

Name	Function	Device ID	License
AHBCTRL	AMBA AHB bus controller with plug&play	-	COM/GPL
APBCTRL	AMBA APB Bridge with plug&play	-	COM/GPL
AHBTRACE	AMBA AHB Trace buffer	0x017	COM/GPL
AHBUART	Serial/AHB debug interface	0x007	COM/GPL
AHBJTAG	JTAG/AHB debug interface	0x01C	COM/GPL
EDCL	Ethernet/AHB debug interface	0x019	COM/GPL

TABLE 4. PCI interface

Name	Function	Device ID	License
PCITARGET	32-bit target-only PCI interface	0x012	COM/GPL
PCIMTF	32-bit PCI master/target interface with FIFO	0x014	COM/GPL
PCITRACE	32-bit PCI trace buffer	0x015	COM/GPL
PCIDMA	DMA controller for PCIMTF	0x016	COM/GPL

TABLE 5. Memory functions

Name	Function	Device ID	License
AHBRAM	Single-port RAM with AHB interface	0x00E	COM/GPL
AHBROM	ROM generator with AHB interface	0x01B	COM/GPL
SYNCRAM	Parametrizable 1-port RAM	-	COM/GPL
SYNCRAM_2P	Parametrizable 2-port RAM	-	COM/GPL
SYNCRAM_DP	Parametrizable dual-port RAM	-	COM/GPL
REGFILE_3P	Parametrizable 3-port register file	-	COM/GPL

TABLE 6. Serial communication

Name	Function	Device ID	License
ETHAHB	AHB interface for Opencores 10/100 Mbit Ethernet MAC	0x005	COM/GPL
APBUART	Programmable UART with APB interface	0x00C	COM/GPL
CANAHB	AHB interface for Opencores CAN 2.0 MAC	0x019	COM/GPL

TABLE 7. Misc. peripherals

Name	Function	Device ID	License
GPTIMER	Modular timer unit	0x011	COM/GPL
GPIO	32-bit General purpose I/O port	0x01A	COM/GPL
TAP	Generic TAP controller	-	COM/GPL
NUHOSP3	PROM & I/O interface for Nuhorizons Spartan3 board	0x02B	COM/GPL

TABLE 8. Simulation and debugging

Name	Function	Device ID	License
APBREPORT	APB bus reporting module	-	COM/GPL
AHBREPORT	AHB bus reporting module	-	COM/GPL
SRAM	SRAM simulation model with srecord pre-load	-	COM/GPL
AHBMSTEM	AHB master simulation model with scripting capability	0x040	COM/GPL
AHBSLVEM	AHB slave simulation model with scripting capability	0x041	COM/GPL

TABLE 9. CCSDS Telecommand and telemetry functions

Name	Function	Device ID	License
GRTM	CCSDS Telemetry encoder	0x030	COM/GPL
GRTC	CCSDS Telecommand decoder	0x031	COM/GPL
GRPW	Packetwire receiver with AHB interface	0x032	COM/GPL
GRCTM	CCSDS Time manager	0x033	COM/GPL

NOTE: The CCSDS functions are described in separate manuals

TABLE 10. Fault-tolerant functions

Name	Function	Device ID	License
FTAHBRAM	Single-port RAM with AHB interface and EDAC protection	0x050	COM
FTSDCTRL	PC133 SDRAM Controller with EDAC	0x009	COM
FTSRCTRL	32-bit PROM/SRAM controller with EDAC	0x051	COM
AHBSTAT	AHB failing address register	0x052	COM
LEON3FT	32-bit Fault-tolerant SPARC V8 Processor	0x053	COM
GRFPCFT	Fault-tolerant LEON2/3 controller for GRFPU	-	COM

2 AHBJTAG - JTAG Debug Link with AHB Master Interface

2.1 Overview

The JTAG Debug link provides access to on-chip AHB bus through JTAG. The JTAG Debug Link module implements a simple protocol which translates JTAG instructions to AHB transfers. Through this link, a read or write transfer can be generated to any address on the AHB bus.

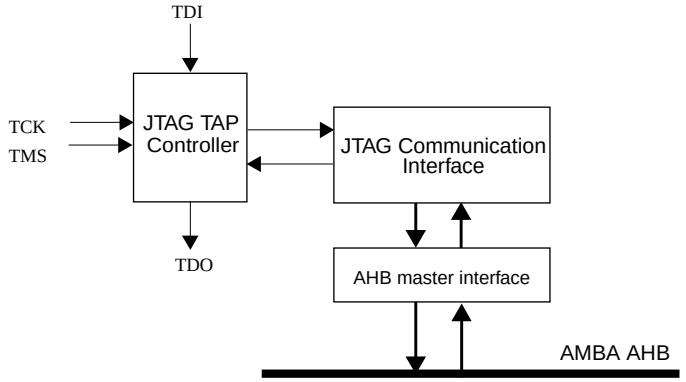

Figure 1. JTAG Debug link block diagram

2.2 Operation

2.2.1 Transmission protocol

The JTAG Debug link decodes two JTAG instructions and implements two JTAG data registers: the command/address register and data register. A read access is initiated by shifting in a command consisting of read/write bit, AHB access size and AHB address into the command/address register. The AHB read access is performed and data is ready to be shifted out of the data register. Write access is performed by shifting in command, AHB size and AHB address into the command/data register followed by shifting in write data into the data register. Sequential transfers can be performed by shifting in command and address for the transfer start address and shifting in SEQ bit in data register for following accesses. The SEQ bit will increment the AHB address for the subsequent access. Sequential transfers should not cross a 1 kB boundary. Sequential transfers are always word based.

Figure 2. JTAG Debug link Command/Address register

[34]: Write (W) - ‘0’ - read transfer, ‘1’ - write transfer

[33:32]: AHB transfer size - “00” - byte, “01” - half-word, “10” - word, “11”- reserved

[31:0]: AHB address

Figure 3. JTAG Debug link Data register

[32]: Sequential transfer (SEQ) - If ‘1’ is shifted in this bit position when read data is shifted out or write data shifted in, the subsequent transfer will be to next word address.

[31:0]: AHB Data - AHB write/read data. For byte and half-word transfers data is aligned according to big-endian order where data with address offset 0 data is placed in MSB bits.

2.3 Configuration options

JTAG Debug link module has the following configuration options (VHDL generics):

TABLE 11. JTAG Debug link configuration options (VHDL generics)

Generic	Function	Allowed range	Default
tech	Target technology	0 - NTECH	0
hindex	AHB master index	0 - NAHBMST-1	0
nsync	Number of synchronization registers between clock regions	1 - 2	1
idcode	JTAG IDCODE instruction code (generic tech only)	0 - 255	9
id_msb	JTAG Device indentification code MSB bits (generic tech only)	0 - 65536	0
id_lsb	JTAG Device indentification code LSB bits (generic tech only)	0 - 65536	0
idcode	JTAG IDCODE instruction (generic tech only)	0 - 255	9
ainst	Code of the JTAG instruction used to access JTAG Debug link command/address register	0 - 255	2
dinst	Code of the JTAG instruction used to access JTAG Debug link data register	0 - 255	3

2.4 Vendor and device id

The module has vendor id 0x01 (Gaisler Research) and device id 0x01C. For description of vendor and device ids see GRLIB IP Library User’s Manual.

2.5 Registers

The JTAG Debug link module does not implement any registers mapped in AHB address space. JTAG registers are described in section 2.2.

2.6 Signal description

JTAG Debug link signals are described in table 12.

TABLE 12. JTAG Debug link signals

Signal name	Field	Type	Function	Active
RST	N/A	Input	Reset	Low
CLK	N/A	Input	System clock (AHB clock domain)	-
TCK	N/A	Input	JTAG clock*	-
TCKN	N/A	Input	Inverted JTAG clock*	-
TMS	N/A	Input	JTAG TMS signal*	High
TDI	N/A	Input	JTAG TDI signal*	High
TDO	N/A	Output	JTAG TDO signal*	High
AHBI	*	Input	AHB Master interface input	-
AHBO	*	Output	AHB Master interface output	-
TAPO_TCK	N/A	Output	TAP Controller User interface TCK signal**	High
TAPO_TDI	N/A	Output	TAP Controller User interface TDI signal**	High
TAPO_INST[7:0]	N/A	Output	TAP Controller User interface INSTsignal**	High
TAPO_RST	N/A	Output	TAP Controller User interface RST signal**	High
TAPO_CAPT	N/A	Output	TAP Controller User interface CAPT signal**	High
TAPO_SHFT	N/A	Output	TAP Controller User interface SHFT signal**	High
TAPO_UPD	N/A	Output	TAP Controller User interface UPD signal**	High
TAPI_TDO	N/A	Input	TAP Controller User interface TDO signal**	High

*) If the target technology is Xilinx Virtex-II or Spartan3 the modules JTAG signals TCK, TCKN, TMS, TDI and TDO are not used. Instead the dedicated FPGA JTAG pins are used. These pins are implicitly made visible to the module through Xilinx TAP controller instantiation.

**) User interface signals from the JTAG TAP controller. These signals are used to interface additional user defined JTAG data registers such as boundary-scan register. For more information on the JTAG TAP controller user interface see JTAG TAP Controller IP-core documentation. If not used tie TAPI_TDO to ground and leave TAPO_* outputs unconnected.

2.7 Library dependencies

Table 13 shows libraries that should be used when instantiating a JTAG Debug link.

TABLE 13. Library dependencies

Library	Package	Imported unit(s)	Description
GRLIB	AMBA	Signals	AMBA signal definitions
GAISLER	JTAG	Signals, component	JTAG signals and component declaration

2.8 JTAG Debug link instantiation

This examples shows how a JTAG Debug link can be instantiated.

```
library ieee;
use ieee.std_logic_1164.all;

library grlib;
use grlib.amba.all;
library gaisler;
use gaisler.jtag.all;

entity ahbjtag_ex is
  port (
    clk : in std_ulogic;
    rstn : in std_ulogic;

    -- JTAG signals
    tck  : in std_ulogic;
    tms  : in std_ulogic;
    tdi  : in std_ulogic;
    tdo  : out std_ulogic
);
end;

architecture rtl of ahbjtag_ex is

  -- AMBA signals
  signal ahbmi : ahb_mst_in_type;
  signal ahbmo : ahb_mst_out_vector := (others => ahbm_none);

signal gnd : std_ulogic;

begin

gnd <= '0';

  -- AMBA Components are instantiated here
  ...

-- AHB JTAG
  ahbjtag0 : ahbjtag generic map(tech => 0, hindex => 1)
 port map(rstn, clkm, tck, tckn, tms, tdi, tdo, ahbmi, ahbmo(1),
               open, open, open, open, open, open, open, gnd);

end;
```

3 AHBCTRL - AMBA AHB controller with plug&play support

3.1 Overview

The AHB controller is a combined AHB arbiter, bus multiplexer and slave decoder according to the AMBA-2.0 standard. The controller supports up to 16 AHB masters, and any number of slaves. The maximum number of masters and slaves are defined in the GRLIB.AMBA package, in the constants NAHBSLV and NAHBMST. It can also be set with the *nahbm* and *nahbs* generics.

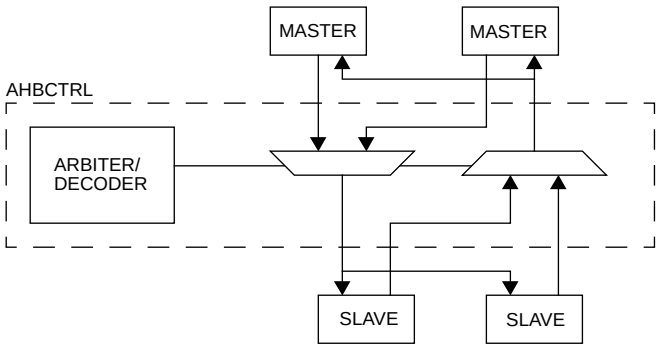

Figure 4. AHB Controller block diagram

3.2 Operation

3.2.1 Arbitration

The AHB controller supports two arbitration algorithms: fixed-priority and round-robin. The selection is done by the generic *rrobin*. In fixed-priority mode (*rrobin* = 0), the bus request priority is equal to the master's bus index, with index 0 being the lowest priority. If no master requests the bus, the master with bus index equal to the generic *defmast* will be granted.

In round-robin mode, priority is rotated one step after each AHB transfer. If no master requests the bus, the last owner will be granted (bus parking).

3.2.2 Decoding

Decoding (generation of HSEL) of AHB slaves is done using the plug&play method explained in the GRLIB User's Manual. A slave can occupy any binary aligned address space with a size of 1 - 4096 Mbyte. A specific I/O area is also decoded, where slaves can occupy 256 byte - 1 Mbyte. The default address of the I/O area is 0xFFF00000, but can be changed with the *cfgaddr* and *cfgmask* generics. Access to unused addresses will cause an AHB error response.

3.2.3 Plug&play information

GRLIB devices contain a number of plug&play information words which are included in the AHB records they drive on the bus (see the GRLIB user's manual for more information). These records are combined into an array which is connected to the AHB controller unit. The plug&play information is mapped on a read-only address area defined by the *cfgaddr* and *cfgmask* generics. by default, the area is mapped on address 0xFFFFF000 - 0xFFFFFFFF. The master information is placed on the first 2Kbyte of the block (0xFFFFF000 - 0xFFFFF800), while the slave information id placed on the second 2Kbyte block. Each unit occupies 32 bytes, which means that the area has place for 64 masters and 64 slaves. The address of the plug&play information for a certain unit is defined by its bus index. The address for masters is thus 0xFFFFF000 + n*32, and 0xFFFFF800 + n*32 for slaves.

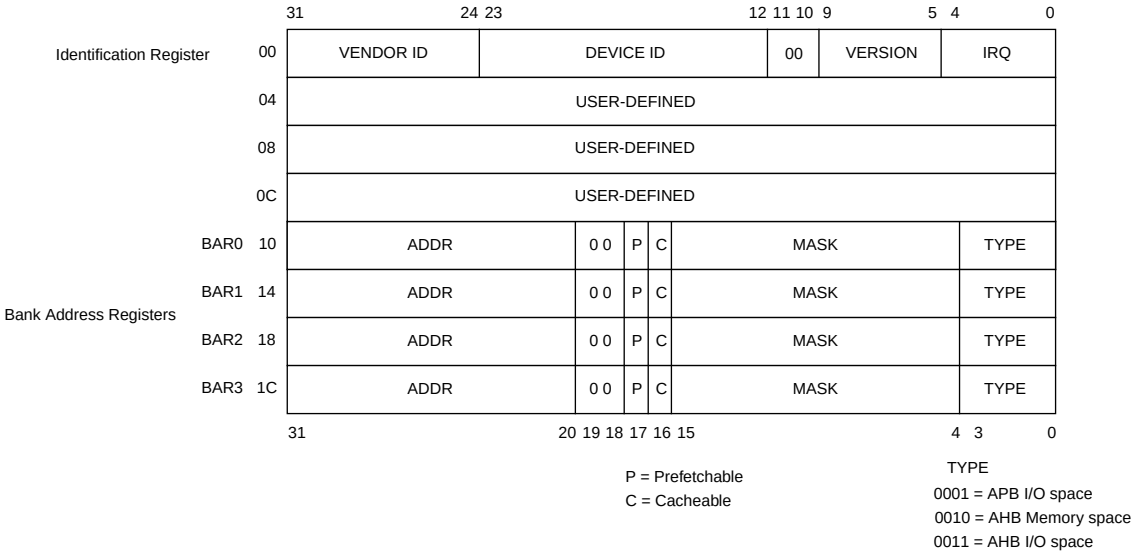

Figure 5. AHB plug&play information record

3.3 AHB split support

AHB SPLIT functionality is supported if the *split* generic is set to 1. In this case, all slaves must driver the AHB SPLIT signal.

3.4 Component declaration

```
library grlib;
use grlib.amba.all;

component ahbctrl
  generic (
    defmast : integer := 0;-- default master
    split   : integer := 0;-- split support
    rrobin  : integer := 0;-- round-robin arbitration
    timeout : integer range 0 to 255 := 0;  -- HREADY timeout
    ioaddr  : ahb_addr_type := 16#fff#;  -- I/O area MSB address
    iomask  : ahb_addr_type := 16#fff#;   -- I/O area address mask
    cfgaddr : ahb_addr_type := 16#ff0#;  -- config area MSB address
    cfgmask : ahb_addr_type := 16#ff0#; -- config area address maskk
    nahbm   : integer range 1 to NAHBMST := NAHBMST; -- number of masters
    nahbs   : integer range 1 to NAHBSLV := NAHBSLV; -- number of slaves
    ioen    : integer range 0 to 15 := 1;   -- enable I/O area
    disirq  : integer range 0 to 1 := 0   -- disable interrupt routing
  );
  port (
    rst     : in  std_ulogic;
    clk     : in  std_ulogic;
    msti    : out ahb_mst_in_type;
    msto    : in  ahb_mst_out_vector;
    slvi    : out ahb_slv_in_type;
    slvo    : in  ahb_slv_out_vector
  );
  end component;
```

3.5 Configuration options

The AHB controller has the following configuration options (VHDL generics):

TABLE 14. AHB controller options (VHDL generics)

Generic	Function	Allowed range	Default
ioaddr	The MSB address of the I/O area. Sets the 12 most significant bits in the 32-bit AHB address.	0 - 16#FFF#	16#FFF#
iomask	The I/O area address mask. Sets the size of the I/O area and the start address together with ioaddr.	0 - 16#FFF#	16#FFF#
cfgaddr	The MSB address of the configuration area.	0 - 16#FFF#	16#FF0#
cfgmask	The address mask of the configuration area. Sets the size of the configuration area and the start address together with cfgaddr. If set to 0, the configuration will be disabled.	0 - 16#FFF#	16#FF0#
rrobin	Selects between round-robin (1) or fixed-priority (0) bus arbitration algorithm.	0 - 1	0
split	Enable support for AHB SPLIT response	0 - 1	0
defmast	Default AHB master	0 - NAHBMST-1	0
ioen	AHB I/O area enable. Set ot 0 to disable the I/O area	0 - 1	1
nahbm	Number of AHB masters	1 - NAHBMST	NAHBMST
nahbs	Number of AHB slaves	1 - NAHBSLV	NAHBSLV
timeout	Perform bus timeout checks (NOT IMPLEMENTED YET).	0 - 1	0

3.6 Signal descriptions

The AHB controller signals are described in Table 15.

TABLE 15. AHB controller signals

Signal name	Type	Function	Active
RST	Input	AHB reset	Low
CLK	Input	AHB clock	-
MSTI*	Output		
MSTO*	Input	AMBA AHB master interface record array	
SLVI*	Output	AMBA AHB master interface record array	-
SLVO*	Input	AMBA AHB slave interface record array	-

*1) see AMBA specification

3.7 Library dependencies

Table 16 shows libraries that should be used when instantiating the AHB controller.

TABLE 16. Library dependencies

Library	Package	Imported unit(s)	Description
GRLIB	AMBA	Types	AMBA signal type definitions

3.8 AHB controller instantiation

This examples shows how an AHB report module can be instantiated.

```
library ieee;
use ieee.std_logic_1164.all;

library grlib;
use grlib.amba.all;

.
.

  -- AMBA signals
  signal ahbsi : ahb_slv_in_type;
  signal ahbso : ahb_slv_out_vector := (others => ahbs_none);
  signal ahbmi : ahb_mst_in_type;
  signal ahbmo : ahb_mst_out_vector := (others => ahbm_none);


begin



-- ARBITER

ahb0 : ahbctrl -- AHB arbiter/multiplexer
  generic map (defmast => CFG_DEFMST, split => CFG_SPLIT,
rrobin => CFG_RROBIN, ioaddr => CFG_AHBIO, nahbm => 8, nahbs => 8)
  port map (rstn, clkm, ahbmi, ahbmo, ahbsi, ahbso);


-- AHB slave

sr0 : srctrl generic map (hindex => 3)
       port map (rstn, clkm, ahbsi, ahbso(3), memi, memo, sdo3);

-- AHB master

e1 : eth_oc
      generic map (mstndx => 2, slvndx => 5, ioaddr => CFG_ETHIO, irq => 12,
                   memtech => memtech)
      port map (rstn, clkm, ahbsi, ahbso(5), ahbmi  => ahbmi,
                   ahbmo => ahbmo(2), ethi1, etho1);

...
end;
```

4 APBCTRL - AMBA AHB/APB bridge with plug&play support

4.1 Overview

The APB bridge is a APB bus master according the AMBA-2.0 standard. The controller supports up to 16 slaves. The actual maximum number of slaves is defined in the GRLIB.AMBA package, in the constant NAPBSLV. The number of slaves can also be set using the *nslaves* generic.

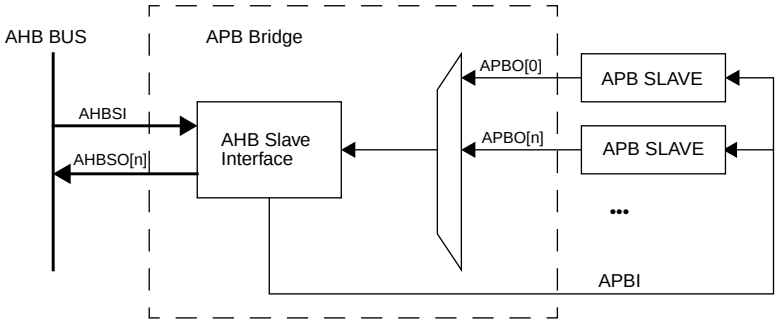

Figure 6. APB Bridge block diagram

4.2 Operation

4.2.1 Decoding

Decoding (generation of PSEL) of APB slaves is done using the plug&play method explained in the GRLIB User's Manual. A slave can occupy any binary aligned address space with a size of 256 bytes - 1 Mbyte.

4.2.2 Plug&play information

GRLIB APB slaves contain two plug&play information words which are included in the APB records they drive on the bus (see the GRLIB User's manual for more information). These records are combined into an array which is connected to the APB bridge. The plug&play information is mapped on a read-only address area at the top 4 kbytes of the bridge address space. Each plug&play block occupies 8 bytes. The address of the plug&play information for a certain unit is defined by its bus index. If the bridge is mapped on AHB address 0x80000000, the address for the plug&play records is thus 0x800FF000 + n*8.

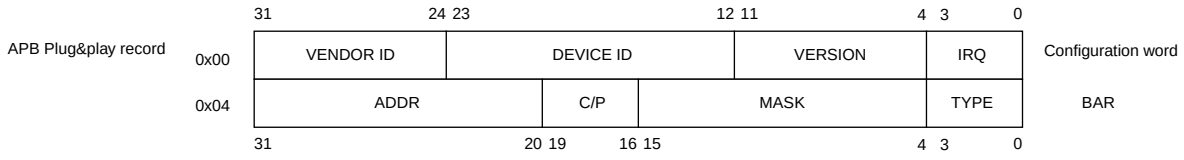

Figure 7. APB plug&play information

4.3 Component declaration

```
library grlib;
use grlib.amba.all;

component apbctrl
  generic (
    hindex  : integer := 0;
    haddr   : integer := 0;
    hmask   : integer := 16#fff#;
    nslaves : integer range 1 to NAPBSLV := NAPBSLV
  );
  port (
    rst     : in  std_ulogic;
    clk     : in  std_ulogic;
    ahbi    : in  ahb_slv_in_type;
    ahbo    : out ahb_slv_out_type;
    apbi    : out apb_slv_in_type;
    apbo    : in  apb_slv_out_vector
  );
  end component;
```

4.4 Configuration options

The APB bridge has the following configuration options (VHDL generics):

TABLE 17. APB bridge options (VHDL generics)

Generic	Function	Allowed range	Default
hindex	AHB slave index	0 - NAHBSLV-1	0
haddr	The MSB address of the AHB area. Sets the 12 most significant bits in the 32-bit AHB address.	0 - 16#FFF#	16#FFF#
hmask	The AHB area address mask. Sets the size of the AHB area and the start address together with haddr.	0 - 16#FFF#	16#FFF#
nslaves	The maximum number of slaves	1 - NAPBSLV	NAPBSLV

4.5 Signal descriptions

The APB bridge signals are described in Table 18.

TABLE 18. APB bridge signals

Signal name	Type	Function	Active
RST	Input	AHB reset	Low
CLK	Input	AHB clock	-
AHBI*	Output	AHB slave inputs	
AHBO*	Input	AHB slave outputs	
APBI*	Output	APB slave inputs	-
APBO*	Input	APB slave outputs	-

*1) see AMBA specification

4.6 Library dependencies

Table 19 shows libraries that should be used when instantiating the APB bridge.

TABLE 19. Library dependencies

Library	Package	Imported unit(s)	Description
GRLIB	AMBA	Types	AMBA signal type definitions

4.7 APB bridge instantiation

This examples shows how an APB bridge can be instantiated.

```
library ieee;
use ieee.std_logic_1164.all;

library grlib;
use grlib.amba.all;
use work.debug.all;

.
.

  -- AMBA signals

  signal ahbsi : ahb_slv_in_type;
  signal ahbso : ahb_slv_out_vector := (others => ahbs_none);

 signal apbi  : apb_slv_in_type;
 signal apbo  : apb_slv_out_vector := (others => apb_none);


begin


-- APB bridge

apb0 : apbctrl-- AHB/APB bridge
  generic map (hindex => 1, haddr => CFG_APBADDR)
  port map (rstn, clk, ahbsi, ahbso(1), apbi, apbo );


-- APB slaves

uart1 : apbuart
    generic map (pindex => 1, paddr => 1,  pirq => 2)
    port map (rstn, clk, apbi, apbo(1), u1i, u1o);

irqctrl0 : irqmp
    generic map (pindex => 2, paddr => 2)
    port map (rstn, clk, apbi, apbo(2), irqo, irqi);


...
end;
```

5 AHBRAM - Single-port RAM with AHB interface

5.1 Overview

AHBRAM implements a 32-bit wide on-chip RAM with an AHB slave interface. Memory size is configurable in binary steps through a VHDL generic. Minimum size is 1kB and maximum size is dependent on target technology and physical resources. Read accesses are zero-waitstate, write access have one waitstate. The RAM supports byte- and half-word accesses, as well as all types of AHB burst accesses. Internally, the AHBRAM instantiates four 8-bit wide SYNCRAM blocks.

5.2 Configuration options

The AHBRAM has the following configuration options (VHDL generics):

TABLE 20. VHDL Generics

Generic	Function	Allowed range	Default
hindex	AHB slave bus index	0 - NAHBSLV-1	0
haddr	The MSB address of the AHB area. Sets the 12 most significant bits in the 32-bit AHB address.	0 - 16#FFF#	16#FFF#
hmask	The AHB area address mask. Sets the size of the AHB area and the start address together with *haddr*.	0 - 16#FFF#	16#FF0#
tech	Technology to implement on-chip RAM	0 - NTECH	0
kbytes	RAM size in Kbytes	target-dependent	1

The AHBRAM has vendor id 0x01 (Gaisler Research) and device id 0x00E. For description of vendor and device ids see GRLIB IP Library User's Manual.

5.3 Library dependencies

Table 21 shows libraries that should be used when instantiating the AHBRAM module.

TABLE 21. Library dependencies

Library	Package	Imported unit(s)	Description
GRLIB	AMBA	Types	AMBA signal type definitions
GAISLER	MISC	Component	Component declaration

5.4 Component declaration

```
library grlib;
use grlib.amba.all;
library gaisler;
use gaisler.misc.all;

component ahbram
  generic ( hindex : integer := 0; haddr : integer := 0; hmask : integer := 16#fff#;
          tech : integer := 0; kbytes : integer := 1);
  port (
      rst : in std_ulogic;
      clk : in std_ulogic;
      ahbsi : in ahb_slv_in_type;
      ahbso : out ahb_slv_out_type
  );
end component;
```

5.5 Component instantiation example

```
library grlib;
use grlib.amba.all;
library gaisler;
use gaisler.misc.all;

.
.

ahbram0 : ahbram generic map (hindex => 7, haddr => CFG_AHBRADDR,
tech => CFG_MEMTECH, kbytes => 8)
    port map ( rstn, clkm, ahbsi, ahbso(7));
```

6 AHBREPORT - AMBA Plug&Play AHB Report Module

6.1 Overview

The AHB report module is provided for printing plug&play information of GRLIB devices on standard output during simulation. It prints vendor-ids and device-ids of all AHB devices and the address ranges occupied by the slave devices. It can also assist in debugging by checking that the index numbers returned by the devices match the signal record they are driving. There is a separate APB report unit which prints information of APB devices.

6.2 Operation

GRLIB devices contain a number of plug&play information words which are included in the AHB records they drive on the bus (see the GRLIB user's manual for more information). These records are combined into an array which is connected to the AHB controller unit. There is one array for all master interfaces and one for all slave interfaces. The AHB report module is also connected to both of these arrays and prints out all the plug & play information found.

The report module starts by scanning the master interface array from 0 to NAHBMST - 1 (defined in the grlib.amba package). It checks each entry in the array for a valid vendor-id (all nonzero ids are considered valid) and if one is found, it also retrieves the device-id. The descriptions for these ids are obtained from the iptable constant in the WORK.DEVLIB package, and are then printed on standard out together with the master number. If the index check is enabled (done with a VHDL generic), the report module also checks if the hindex number returned in the record matches the array number of the record currently checked (the array index). If they do not match, the simulation is aborted and an error message is printed.

This procedure is repeated for slave interfaces found in the slave interface array. It is scanned from 0 to NAHBSLV - 1 and the same information is printed and the same checks are done as for the master interfaces. In addition, the address range and memory type is checked and printed. The address information includes type, address, mask, cacheable and pre-fetchable fields. From this information, the report module calculates the start address of the device and the size of the range. The information finally printed is type, start address, size, cacheability and pre-fetchability. The address ranges currently defined are AHB memory, AHB I/O and APB I/O. APB I/O ranges are ignored by this module.

6.3 Configuration options

The AHB report module has the following configuration options (VHDL generics):

TABLE 22. AHB report module options (generics)

Generic	Function	Allowed range	Default
ioaddr	The MSB address of the I/O area. Sets the 12 most significant bits in the 32-bit AHB address.	0 - 16#FFF#	16#FFF#
iomask	The I/O area address mask. Sets the size of the I/O area and the start address together with ioaddr.	0 - 16#FFF#	16#FFF#
cfgaddr	The MSB address of the configuration area.	0 - 16#FFF#	16#FF0#
cfgmask	The address mask of the configuration area. Sets the size of the configuration area and the start address together with cfgaddr.	0 - 16#FFF#	16#FF0#
icheck	If asserted (1), the hindex and bus index of each device are checked for equality.	0 - 1	1
timeout	Perform bus timeout checks (NOT IMPLEMENTED YET).	0 - 1	0
nmasters	Maximum number of AHB masters	1 - NAHBMST	NAHBMST

TABLE 22. AHB report module options (generics)

Generic	Function	Allowed range	Default
nslaves	Maximum number of AHB slaves	1 - NAHBSLV	NAHBSLV
ioen	AHB I/O area enable	0 - 1	1

6.4 Signal descriptions

The AHB report module signals are described in Table 23.

TABLE 23. AHB report module signals

Signal name	Field	Type	Function	Active
MSTO*	-	Input	AMBA AHB master interface record array	-
SLVO*	-	Input	AMBA AHB slave interface record array	-

*1) see AMBA specification

6.5 Library dependencies

Table 24 shows libraries that should be used when instantiating the AHB report module.

TABLE 24. Library dependencies

Library	Package	Imported unit(s)	Description
GRLIB	AMBA	Types	AMBA signal type definitions
WORK	DEBUG	Component	Component declaration

6.6 Component declaration

```
library ieee;
use ieee.std_logic_1164.all;

library grlib;
use grlib.amba.all;
use work.debug.all;

component ahbreport
  generic (
    ioaddr  : ahb_addr_type := 16#fff#;  -- I/O area MSB address
    iomask  : ahb_addr_type := 16#fff#;  -- I/O area address mask
    cfgaddr : ahb_addr_type := 16#ff0#;  -- config area MSB address
    cfgmask : ahb_addr_type := 16#ff0#;  -- config area address mask
    icheck : integer := 1;-- check bus indexes
    timeout : integer := 0;   -- check bus timeout
    nmasters: integer := NAHBMST;   -- Number of masters
    nslaves : integer := NAHBSLV;   -- Number of slaves
    ioen   : integer := 1-- enable I/O area
  );
  port (
    msto    : in  ahb_mst_out_vector;
    slvo    : in  ahb_slv_out_vector
  );
end component;
```

6.7 AHB report module instantiation

This examples shows how an AHB report module can be instantiated.

```
library ieee;
use ieee.std_logic_1164.all;

library grlib;
use grlib.amba.all;
use work.debug.all;

entity ahbreport_ex is
  port (
    rst : std_ulogic;
    clk : std_ulogic;
    );
end;

architecture rtl of ahbreport_ex is

  -- AMBA signals
  signal ahbsi : ahb_slv_in_type;
  signal ahbso : ahb_slv_out_vector := (others => ahbs_none);
  signal ahbmi : ahb_mst_in_type;
  signal ahbmo : ahb_mst_out_vector := (others => ahbm_none);


begin

  -- AMBA Components are instantiated here
  ...

--pragma translate off
  -- AHB Report
  ahbrep : ahbreport
  generic map (icheck => 1)
  port map(msto => ahbmo, slvo => ahbso);
--pragma translate on
end;
```

7 AHBROM - Single-port ROM with AHB interface

7.1 Overview

AHBROM implements a 32-bit wide on-chip ROM with an AHB slave interface. Read accesses take zero waitstates, or one waitstate if the pipeline option is enabled. The ROM supports byte- and half-word accesses, as well as all types of AHB burst accesses.

7.2 PROM generation

The AHBPROM is automatically generated by the make utility in GRLIB. The input format is a sparc-elf binary file, produced by the BCC cross-compiler (sparc-elf-gcc). To create a PROM, first compile a suitable binary and the run the make utility:

```
bash$ sparc-elf-gcc prom.S -o prom.exe
bash$ make ahbrom.vhd

   Creating ahbrom.vhd : file size 272 bytes, address bits 9
```

The default binary file for creating a PROM is prom.exe. To use a different file, run make with the FILE parameter set to the input file:

```
bash$ make ahbrom.vhd FILE=myfile.exe
```

The created PROM is realized in synthesizable VHDL code, using a CASE statement. For FPGA targets, most synthesis tools will map the CASE statement on a block RAM/ROM if available. For ASIC implementations, the ROM will be synthesized as gates. It is then recommended to use the *pipe* option to improve the timing.

7.3 Configuration options

The AHBROM has the following configuration options (VHDL generics):

TABLE 25. VHDL Generics

Generic	Function	Allowed range	Default
hindex	AHB slave bus index	0 - NAHBSLV-1	0
haddr	The MSB address of the AHB area. Sets the 12 most significant bits in the 32-bit AHB address.	0 - 16#FFF#	16#FFF#
hmask	The AHB area address mask. Sets the size of the AHB area and the start address together with *haddr*.	0 - 16#FFF#	16#FF0#
tech	Not used		
pipe	Add a pipeline stage on read data	0	0
kbytes	Not used		

The AHBRAM has vendor id 0x01 (Gaisler Research) and device id 0x01B. For description of vendor and device ids see GRLIB IP Library User's Manual.

7.4 Library dependencies

Table 26 shows libraries that should be used when instantiating the AHBRAM module.

TABLE 26. Library dependencies

Library	Package	Imported unit(s)	Description
GRLIB	AMBA	Types	AMBA signal type definitions
GAISLER	MISC	Component	Component declaration

7.5 Component declaration

```
library grlib;
use grlib.amba.all;
library gaisler;
use gaisler.misc.all;

component ahbrom
  generic ( hindex : integer := 0; haddr : integer := 0; hmask : integer := 16#fff#;
          pipe : integer := 0; tech : integer := 0);
  port (
      rst : in std_ulogic;
      clk : in std_ulogic;
      ahbsi : in ahb_slv_in_type;
      ahbso : out ahb_slv_out_type
  );
end component;
```

8 AHBSTAT - AHB Status Registers

8.1 Overview

The AHB status registers store information about AHB accesses triggering an error response. There is a status register and a failing address register. Both are contained in a module accessed from the APB bus. Figure 8 shows the registers' contents.

8.2 Operation

The AHB status module monitors AHB bus transactions and stores the current HADDR, HWRITE, HMASTER and HSIZE internally. It is always active after startup and reset until it detects an error response (HRESP = "01"). When the error is detected it freezes the status and address register contents and sets New Error (NE) to one. At the same time it also generates an interrupt on the line selected by the pirq generic. The interrupt is usually connected to the IRQ controller so that the CPU is informed about the condition. The normal procedure is that an interrupt routine handles the error with the aid of the information in the status registers. When it is finished it resets NE and the status module becomes active again.

Not only error responses on the bus can be detected. Many of the fault tolerant units containing EDAC have a correctable error signal which is asserted each time a single error is detected. These should be connected to the stati.cerror input signal of the status module which is ored internally and if the resulting signal is asserted, it will have the same effect as an AHB error response. The only difference is that the CE bit in the status register is set when a single error is detected. When the ce bit is set the interrupt routine can acquire the address containing the single error from the failing address register and correct it.

8.3 Configuration options

The AHB status register has the following configuration options (VHDL generics):

TABLE 27. AHB status register configuration options (VHDL generics)

Generic	Function	Allowed range	Default
pindex	APB slave index	0 - NAHBSLV-1	0
paddr	APB address	0 - 16#FFF#	0
pmask	APB address mask	0 - 16#FFF#	16#FFF#
pirq	Interrupt line driven by the status module	0 - 16#FFF#	0
nftslv	Number of FT slaves connected to the cerror vector	1 - NAHBSLV-1	3

8.4 Vendor and device id

The module has vendor id 0x01 (Gaisler Research) and device id 0x052. For description of vendor and device ids see GRLIB IP Library User's Manual.

8.5 Registers

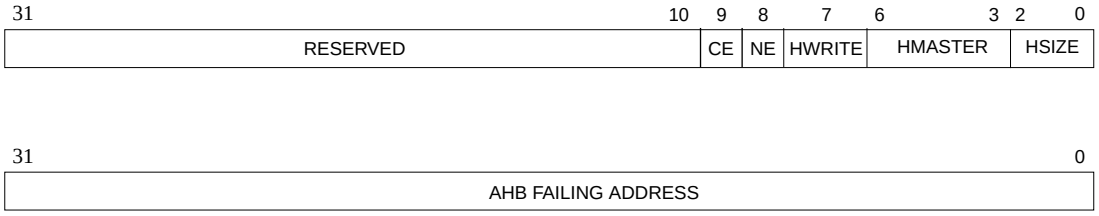

Figure 8. AHB status and failing address register.

Figure 8 shows the status register and failing address register. The registers are accessed from the APB bus. The status register has offset 0x0 and the failing address register has offset 0x4 to the base address of the module.

Status register:

[2:0]: The HSIZE signal of the AHB transaction that caused the error.

[6:3]: The HMASTER signal of the AHB transaction that caused the error.

[7]: The HWRITE signal of the AHB transaction that caused the error.

[8] NE: New Error. Deasserted at start-up and after reset. Asserted when an error is detected. Reset by writing a zero to it.

[9] CE: Correctable Error. Set if the detected error was caused by a single error and zero otherwise.

[31:10] Reserved.

Failing address register:

[31:0]: The HADDR signal of the AHB transaction that caused the error.

8.6 Signal description

The AHB status register signals are described in table 28.

TABLE 28. AHB status register signal description.

Signal name	Field	Type	Function	Active
RST	N/A	Input	Reset	Low
CLK	N/A	Input	Clock	-
AHBMI	*	Input	AHB slave input signals	-
AHBSI	*	Input	AHB slave output signals	-
STATI	CERROR	Input	Correctable Error Signals	High
APBI	*	Input	APB slave input signals	-
APBO	*	Output	APB slave output signals	-

* see GRLIB IP Library User's Manual

8.7 Library dependencies

Table 29 shows libraries that the AHB status register module depends on.

TABLE 29. Library dependencies

Library	Package	Imported unit(s)	Description
GRLIB	AMBA	Signals	AHB signal definitions
GAISLER	MISC	Component	Component declaration

8.8 AHB status register module instantiation

This examples shows how an AHB status register module can be instantiated. The example design contains an AMBA bus with a number of AHB components connected to it including the status register. There are three Fault Tolerant units with EDAC connected to the status register cerror vector. The connection of the different memory controllers to external memory is not shown.

```
library ieee;
use ieee.std_logic_1164.all;

library grlib;
use grlib.amba.all;
use grlib.tech.all;
library gaisler;
use gaisler.memctrl.all;
use gaisler.misc.all;

entity mctrl_ex is
  port (
    clk : in std_ulogic;
    rstn : in std_ulogic;
  --other signals
  ....
    );
end;

architecture rtl of mctrl_ex is

  -- AMBA bus (AHB and APB)
  signal apbi  : apb_slv_in_type;
  signal apbo  : apb_slv_out_vector := (others => apb_none);
  signal ahbsi : ahb_slv_in_type;
  signal ahbso : ahb_slv_out_vector := (others => ahbs_none);
  signal ahbmi : ahb_mst_in_type;
  signal ahbmo : ahb_mst_out_vector := (others => ahbm_none);

  -- signals used to connect memory controller and memory bus
  signal memi : memory_in_type;
  signal memo : memory_out_type;

  signal sdo, sdo2: sdctrl_out_type;

  signal sdi : sdctrl_in_type;

-- correctable error vector
  signal stati : ahbstat_in_type;
  signal aramo : ahbram_out_type;

begin

  -- AMBA Components are defined here ...

-- AHB Status Register
  astat0 : ahbstat generic map(pindex => 13, paddr => 13, pirq => 11,
    nftslv => 3)
    port map(rstn, clkm, ahbmi, ahbsi, stati, apbi, apbo(13));
    stati.cerror(3 to NAHBSLV-1) <= (others => '0');

--FT AHB RAM
  a0 : ftahbram generic map(hindex => 1, haddr => 1, tech => inferred,
    kbytes => 64, pindex => 4, paddr => 4, edacen => 1, autoscrub => 0,
    errcnt => 1, cntbits => 4)
    port map(rst, clk, ahbsi, ahbso, apbi, apbo(4), aramo);
    stati.cerror(0) <= aramo.ce;
```

```
-- SDRAM controller
  sdc : ftsdctrl generic map (hindex => 3, haddr => 16#600#, hmask => 16#F00#,
    ioaddr => 1, fast => 0, pwron => 1, invclk => 0, edacen => 1, errcnt => 1,
    cntbits => 4)
    port map (rstn, clk, ahbsi, ahbso(3), sdi, sdo);
    stati.cerror(1) <= sdo.ce;

-- Memory controller
  mctrl0 : ftsrctrl generic map (rmw => 1, pindex => 10, paddr => 10,
    edacen => 1, errcnt => 1, cntbits => 4)
    port map (rstn, clk, ahbsi, ahbso(0), apbi, apbo(10), memi, memo, sdo2);
    stati.cerror(2) <= memo.ce;
end;
```

9 AHBTRACE - AHB Trace Buffer

9.1 Overview

The AHB trace buffer consists of a circular buffer that stores AHB data transfers. The address, data and various control signals of the AHB bus are stored and can be read out for later analysis.

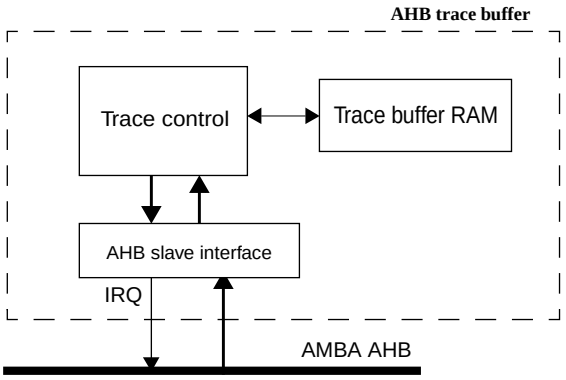

Figure 9. AHBTRACE block diagram

The trace buffer is 128 bits wide, the information stored is indicated in the table below:

TABLE 30. AHB Trace buffer data allocation

Bits	Name	Definition
127:96	Time tag	The value of the time tag counter
95	AHB breakpoint hit	Set to ‘1’ if a DSU AHB breakpoint hit occurred.
94:80	Hirq	AHB HIRQ[15:1]
79	Hwrite	AHB HWRITE
78:77	Htrans	AHB HTRANS
76:74	Hsize	AHB HSIZE
73:71	Hburst	AHB HBURST
70:67	Hmaster	AHB HMASTER
66	Hmastlock	AHB HMASTLOCK
65:64	Hresp	AHB HRESP
63:32	Load/Store data	AHB HRDATA or HWDATA
31:0	Load/Store address	AHB HADDR

In addition to the AHB signals, a 32-bit counter is also stored in the trace as time tag.

9.2 Operation

The trace buffer is enabled by setting the enable bit (EN) in the trace control register. Each AHB transfer is then stored in the buffer in a circular manner. The address to which the next transfer is written is held in the trace buffer index register, and is automatically incremented after each transfer. Tracing is stopped when the EN bit is reset, or when a AHB breakpoint is hit. An interrupt is generated when a breakpoint is hit.

Note: the LEON3 Debug support unit (DSU3) also includes an AHB trace buffer. The AHBTRACE module is intended to be used in system without the LEON3 processor or when the DSU3 is not present.

9.3 Registers

9.3.1 Register address map

The trace buffer occupies 128 Kbyte address space in the AHB I/O area. The following register address are decoded:

TABLE 31. Trace buffer address space

Address	Register
0x000000	Trace buffer control register
0x000004	Trace buffer index register
0x000008	Time tag counter
0x000010	AHB break address 1
0x000014	AHB mask 1
0x000018	AHB break address 2
0x00001C	AHB mask 2
0x010000 - 0x020000	Trace buffer
..0	Trace bits 127 - 96
...4	Trace bits 95 - 64
...8	Trace bits 63 - 32
...C	Trace bits 31 - 0

9.3.2 Trace buffer control register

The trace buffer is controlled by the trace buffer control register:

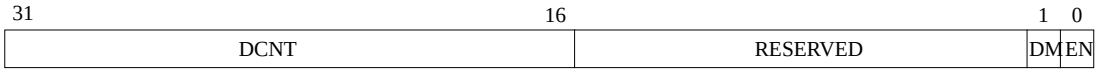

Figure 10. Trace buffer control register

0: Trace enable (EN). Enables the trace buffer.

1: Delay counter mode (DM). Indicates that the trace buffer is in delay counter mode.

31:16 Trace buffer delay counter (DCNT). Note that the number of bits actually implemented depends on the size of the trace buffer.

9.3.3 Trace buffer index register

The trace buffer index register indicates the address of the next 128-bit line to be written.

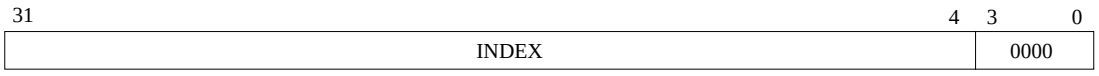

Figure 11. Trace buffer index register

31:4 Trace buffer index counter (INDEX). Note that the number of bits actually implemented depends on the size of the trace buffer.

9.3.4 Trace buffer time tag register

The time tag register contains a 32-bit counter that increments each clock when the trace buffer is enabled. The value of the counter is stored in the trace to provide a time tag.

Figure 12. Trace buffer time tag counter

9.3.5 Trace buffer breakpoint registers

The DSU contains two breakpoint registers for matching AHB addresses. A breakpoint hit is used to freeze the trace buffer by clearing the enable bit. Freezing can be delayed by programming the DCNT field in the trace buffer control register to a non-zero value. In this case, the DCNT value will be decremented for each additional trace until it reaches zero, after which the trace buffer is frozen. A mask register is associated with each breakpoint, allowing breaking on a block of addresses. Only address bits with the corresponding mask bit set to '1' are compared during breakpoint detection. To break on AHB load or store accesses, the LD and/or ST bits should be set.

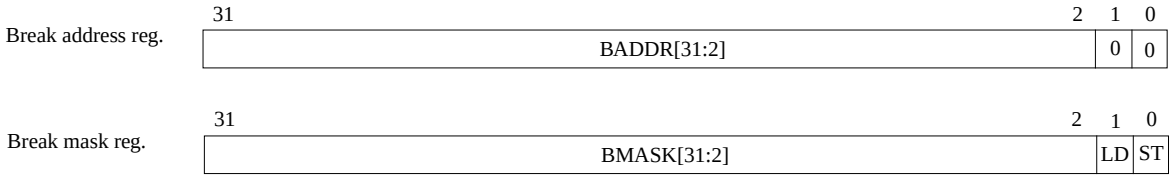

Figure 13. Trace buffer breakpoint registers

BADDR : breakpoint address (bits 31:2)
BMASK : Breakpoint mask (see text)
LD : break on data load address
ST : beak on data store address

9.4 Configuration options

The AHBTRACE has the following configuration options (VHDL generics):

TABLE 32. VHDL Generics

Generic	Function	Allowed range	Default
hindex	AHB slave bus index	0 - NAHBSLV-1	0
ioaddr	The MSB address of the I/O area. Sets the 12 most significant bits in the 20-bit I/O address.	0 - 16#FFF#	16#000#
iomask	The I/O area address mask. Sets the size of the I/O area and the start address together with ioaddr.	0 - 16#FFF#	16#E00#
irq	Interrupt number	0 - NAHBIRQ-1	0
tech	Technology to implement on-chip RAM	0 - NTECH	0
kbytes	Trace buffer size in Kbytes	1 - 64	1

9.5 Vendor and device id

The module has vendor id 0x01 (Gaisler Research) and device id 0x017. For description of vendor and device ids see GRLIB IP Library User's Manual.

9.6 Library dependencies

Table 33 shows libraries that should be used when instantiating the AHBTRACE module.

TABLE 33. Library dependencies

Library	Package	Imported unit(s)	Description
GRLIB	AMBA	Types	AMBA signal type definitions
GAISLER	MISC	Component	Component declaration

9.7 Component declaration

```
library grlib;
use grlib.amba.all;
library gaisler;
use gaisler.misc.all;

component ahbtrace is
  generic (
    hindex  : integer := 0;
    ioaddr  : integer := 16#000#;
    iomask  : integer := 16#E00#;
    tech    : integer := 0;
    irq     : integer := 0;
    kbytes  : integer := 1);
  port (
    rst    : in  std_ulogic;
    clk    : in  std_ulogic;
    ahbmi  : in  ahb_mst_in_type;
    ahbsi  : in  ahb_slv_in_type;
    ahbso  : out ahb_slv_out_type
  );
end component;
```

10 AHBUART - Serial debug interface for AHB

10.1 Overview

The AHBUART consists of a UART connected to the AHB bus as a master (figure 14). A simple communication protocol is supported to transmit access parameters and data. Through the communication link, a read or write transfer can be generated to any address on the AHB bus.

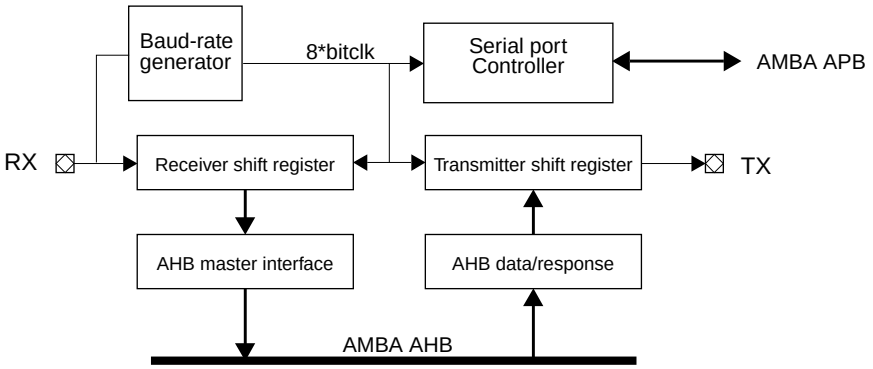

Figure 14. AHB UART block diagram

10.2 Operation

10.2.1 Transmission protocol

The AHB UART supports simple protocol where commands consist of a control byte, followed by a 32-bit address, followed by optional write data. Write access does not return any response, while a read access only returns the read data. Data is sent on 8-bit basis as shown below.

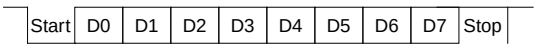

Figure 15. AHB UART data frame

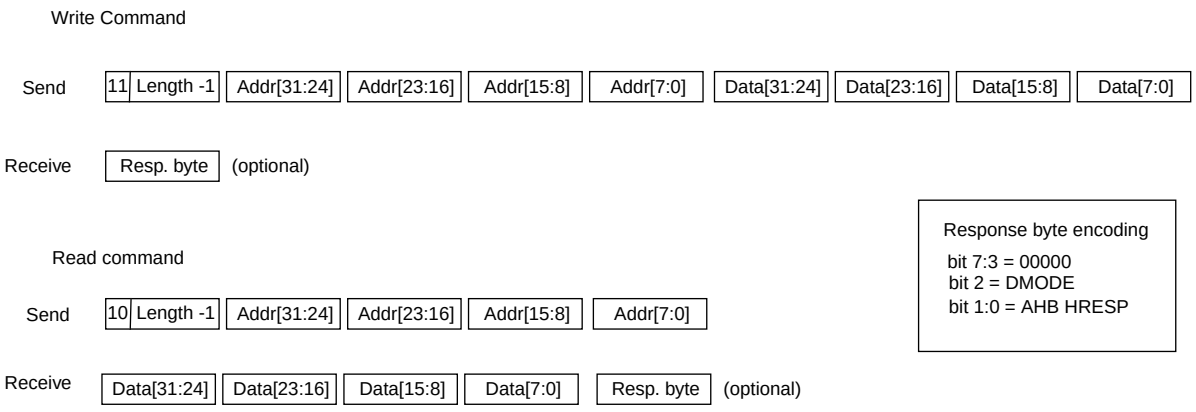

Figure 16. AHB UART commands

Block transfers can be performed be setting the length field to *n*-1, where *n* denotes the number of transferred words. For write accesses, the control byte and address is sent once, followed by the

number of data words to be written. The address is automatically incremented after each data word. For read accesses, the control byte and address is sent once and the corresponding number of data words is returned.

The UART receiver is implemented with same glitch filtering as the GR APB UART.

10.2.2 Baud rate generation

The AHB UART contains a 18-bit down-counting scaler to generate the desired baud-rate. The scaler is clocked by the system clock and generates a UART tick each time it underflows. The scaler is reloaded with the value of the UART scaler reload register after each underflow. The resulting UART tick frequency should be 8 times the desired baud-rate.

If not programmed by software, the baud rate will be automatically discovered. This is done by searching for the shortest period between two falling edges of the received data (corresponding to two bit periods). When three identical two-bit periods has been found, the corresponding scaler reload value is latched into the reload register, and the BL bit is set in the UART control register. If the BL bit is reset by software, the baud rate discovery process is restarted. The baud-rate discovery is also restarted when a 'break' or framing error is detected by the receiver, allowing to change to baudrate from the external transmitter. For proper baudrate detection, the value 0x55 should be transmitted to the receiver after reset or after sending break.

The best scaler value for manually programming the baudrate can be calculated as follows:

```
scaler = (((system_clk*10)/(baudrate*8))-5)/10
```

10.3 Configuration options

AHB UART has the following configuration options (VHDL generics):

TABLE 34. AHB UART configuration options (VHDL generics)

Generic	Function	Allowed range	Default
hindex	AHB master index	0 - NAHBMST-1	0
pindex	APB slave index	0 - NAPBSLV-1	0
paddr	ADDR filed of the APB BAR.	0 - 16#FFF#	0
pmask	MASK filed of the APB BAR.	0 - 16#FFF#	16#FFF#

10.4 Vendor and device id

The module has vendor id 0x01 (Gaisler Research) and device id 0x007. For description of vendor and device ids see GRLIB IP Library User's Manual.

10.5 Registers

The AHB UART is programmed through tree registers mapped into APB address space.

TABLE 35. AHB Uart registers

Register	APB Address offset
AHB UART status register	0x4
AHB UART control register	0x8
AHB UART scaler register	0xC

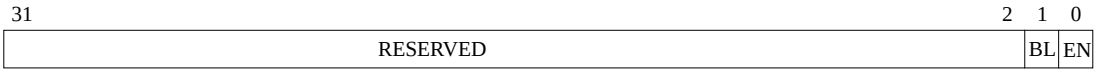

Figure 17. AHB UART control register

0: Receiver enable (RE) - if set, enables both the transmitter and receiver.
1: Baud rate locked (BL) - is automatically set when the baud rate is locked.

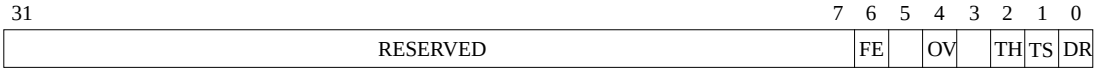

Figure 18. AHB UART status register

0: Data ready (DR) - indicates that new data has been received by the AHB master interface.
1: Transmitter shift register empty (TS) - indicates that the transmitter shift register is empty.

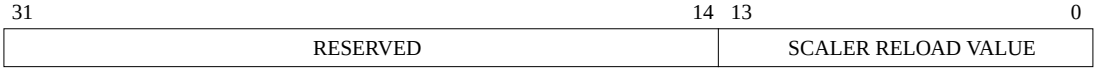

Figure 19. AHB UART scaler reload register

10.6 Signal description

AHB UART signals are described in table 36.

TABLE 36. AHB UART signals

Signal name	Field	Type	Function	Active
RST	N/A	Input	Reset	Low
CLK	N/A	Input	Clock	-
UARTI	RXD	Input	UART receiver data	High
	CTSN	Input	UART clear-to-send	High
	EXTCLK	Input	Use as alternative UART clock	-
UARTO	RTSN	Output	UART request-to-send	High
	TXD	Output	UART transmit data	High
APBI	*	Input	APB slave input signals	-
APBO	*	Output	APB slave output signals	-
AHBI	*	Input	AMB master input signals	-
AHBO	*	Output	AHB master output signals	-

* see GRLIB IP Library User's Manual

10.7 Library dependencies

Table 37 shows libraries that should be used when instantiating an AHB UART.

TABLE 37. Library dependencies

Library	Package	Imported unit(s)	Description
GRLIB	AMBA	Signals	AMBA signal definitions
GAISLER	UART	Signals, component	AHB UART component declaration

10.8 AHB UART instantiation

This examples shows how an AHB UART can be instantiated.

```
library ieee;
use ieee.std_logic_1164.all;

library grlib;
use grlib.amba.all;
library gaisler;
use gaisler.uart.all;

entity ahbuart_ex is
  port (
    clk : in std_ulogic;
    rstn : in std_ulogic;

    -- UART signals
    ahbrxd   : in  std_ulogic;
    ahbtxd   : out std_ulogic
    );
end;

architecture rtl of ahbuart_ex is

  -- AMBA signals
  signal apbi  : apb_slv_in_type;
  signal apbo  : apb_slv_out_vector := (others => apb_none);
  signal ahbmi : ahb_mst_in_type;
  signal ahbmo : ahb_mst_out_vector := (others => ahbm_none);

  -- UART signals
  signal ahbuarti : uart_in_type;
  signal ahbuarto : uart_out_type;

begin

  -- AMBA Components are instantiated here
  ...


  -- AHB UART
  ahbuart0 : ahbuart
  generic map (hindex => 5, pindex => 7, paddr => 7)
  port map (rstn, clk, ahbuarti, ahbuarto, apbi, apbo(7), ahbmi, ahbmo(5));

  -- AHB UART input data
  ahbuarti.rxd <= ahbrxd;

  -- connect AHB UART output to entity output signal
  ahbtxd <= ahbuarto.txd;

end;
```

11 APBREPORT - AMBA Plug&Play APB Report Module

11.1 Overview

The APB report module is provided for printing plug&play information of GRLIB devices on standard output during simulation. It prints vendor-ids and device-ids of all APB devices and their respective address ranges. APB bridge information is also printed. It can also assist in debugging by checking that the index numbers returned by the devices match the signal record they are driving. There is a separate AHB report unit which prints information of AHB devices.

11.2 Operation

GRLIB APB devices contain a number of plug&play information words which are included in the APB records they drive on the bus (see the GRLIB user's manual for more information). These records are combined into an array which is connected to the APB controller unit. The APB report module is also connected to this array and prints out all the plug & play information found.

The report module starts by scanning the array from 0 to NAPBSLV - 1 (defined in the grlib.amba package). It checks each entry in the array for a valid vendor-id (all nonzero ids are considered valid) and if one is found, it also retrieves the device-id. The descriptions for these ids are obtained from the iptable in the global devices file and are then printed on standard out together with the slave number. If the index check is enabled (done with a VHDL generic), the report module also checks if the pindex number returned in the record matches the array number of the record currently checked (the array index). If they do not match, the simulation is aborted and an error message is printed.

The address range and memory type is also checked and printed. The address information includes type, address and mask. The address ranges currently defined are AHB memory, AHB I/O and APB I/O. All APB devices are in the APB I/O range so the type does not have to be checked. From this information, the report module calculates the start address of the device and the size of the range. The information finally printed is start address and size.

11.3 Configuration options

The APB report module has the following configuration options (VHDL generics):

TABLE 38. APB report module options (generics)

Generic	Function	Allowed range	Default
haddr	The MSB address of the I/O area. Sets the 12 most significant bits in the 32-bit AHB address.	0 - 16#FFF#	0
hmask	The I/O area address mask. Sets the size of the I/O area and the start address together with the ioaddr.	0 - 16#FFF#	16#FFF#
icheck	If asserted (1), the pindex and bus index of each device are checked for equality.	0 - 1	1
nslaves	Maximum number of APB slaves	1 - NAPBSLV	NAPBSLV

11.4 Signal descriptions

The APB report module signals are described in Table 39.

TABLE 39. APB report module signals

Signal name	Field	Type	Function	Active
APBO*	-	Input	AMBA APB slave interface record array	-

*1) see AMBA specification

11.5 Library dependencies

Table 40 shows libraries that should be used when instantiating the APB report module.

TABLE 40. Library dependencies

Library	Package	Imported unit(s)	Description
GRLIB	AMBA	Types	AMBA signal type definitions
WORK	DEBUG	Component	Component definition

11.6 Component declaration

```
library ieee;
use ieee.std_logic_1164.all;

library grlib;
use grlib.amba.all;
use work.debug.all;

component apbreport
  generic (
    haddr : integer := 0;
    hmask : integer := 16#fff#;
    icheck  : integer := 1;
    nslaves : integer := NAPBSLV   -- Number of slaves
  );
  port (
    apbo    : in  apb_slv_out_vector
  );
end component;
```

11.7 APB report module instantiation

This examples shows how an APB report module can be instantiated.

```
library ieee;
use ieee.std_logic_1164.all;

library grlib;
use grlib.amba.all;
use work.debug.all;

entity apbreport_ex is
  port (
    rst : std_ulogic;
    clk : std_ulogic;
    );
end;

architecture rtl of apbreport_ex is

  -- AMBA signals
  signal apbi : apb_slv_in_type;
  signal apbo : apb_slv_out_vector := (others => apb_none);


begin
```

```
  -- AMBA Components are instantiated here
  ...

--pragma translate off
  -- APB Report
  apbrep : apbreport
  generic map (haddr => 16#FFF#, icheck => 1)
  port map(apbo => apbo);
--pragma translate on
end;
```

12 [1]APBUART - UART with APB interface

12.1 Overview

The APBUART is provided for serial communications. The UART supports data frames with 8 data bits, one optional parity bit and one stop bit. To generate the bit-rate, each UART has a programmable 12-bit clock divider. Hardware flow-control is supported through the RTSN/CTSN hand-shake signals. Two configurable FIFOs are used for the data transfers between the bus and UART. Figure 20 shows a block diagram of the APB UART.

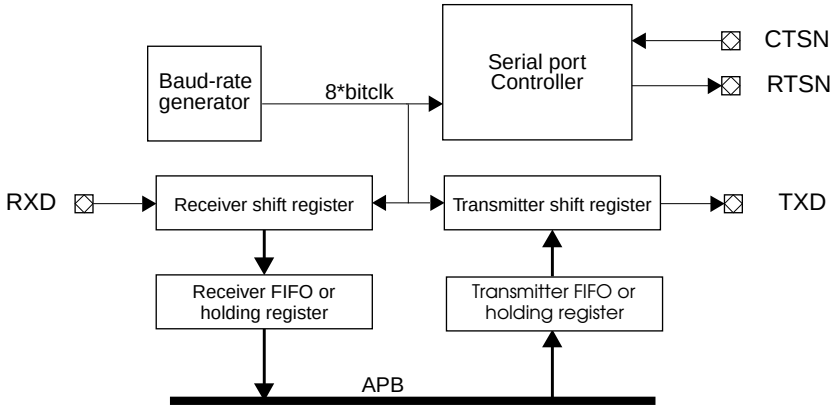

Figure 20. APB UART block diagram

12.2 Operation

12.2.1 Transmitter operation

The transmitter is enabled through the TE bit in the UART *control* register. Data that is to be transferred is stored in the FIFO by writing to the data register (see section 5). This FIFO is configurable to different sizes (see table 1). When the size is 1, only a single holding register is used but in the following discussion both will be referred to as FIFOs. When ready to transmit, data is transferred from the transmitter FIFO to the transmitter shift register and converted to a serial stream on the transmitter serial output pin (TXD). It automatically sends a start bit followed by eight data bits, an optional parity bit, and one stop bit (figure 21). The least significant bit of the data is sent first.

1.

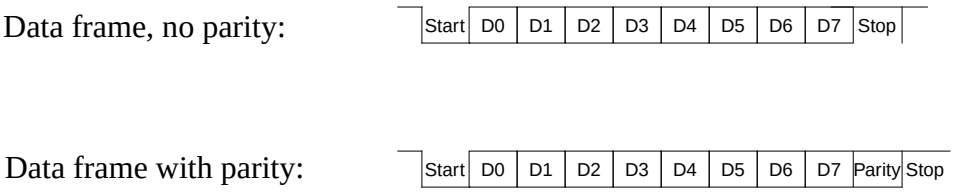

Figure 21. UART data frames

Following the transmission of the stop bit, if a new character is not available in the transmitter FIFO, the transmitter serial data output remains high and the transmitter shift register empty bit (TS) will be set in the UART status register (see section 5). Transmission resumes and the TS is cleared when a new character is loaded into the transmitter FIFO. When the FIFO is empty the TE bit is set in the *status* register. If the transmitter is disabled, it will immediately stop any active transmissions including the character currently being shifted out from the transmitter shift register. The transmitter holding register may not be loaded when the transmitter is disabled or when the FIFO (or holding register) is full. If this is done, data might be overwritten and one or more frames are lost.

The discussion above applies to any FIFO configurations including the special case with a holding register (fifosize = 1). If FIFOs are used (fifosize > 1) some additional status and control bits are available. The TF status bit (not to be confused with the TF control bit) is set if the transmitter FIFO is currently full and the TH bit is set as long as the FIFO is *less* than half-full (less than half of entries in the FIFO contain data). The TF control bit enables FIFO interrupts when set. The status register also contains a counter (TCNT) showing the current number of data entries in the FIFO.

If flow control is enabled, the CTSN input must be low in order for the character to be transmitted. If it is deasserted in the middle of a transmission, the character in the shift register is transmitted and the transmitter serial output then remains inactive until CTSN is asserted again. If the CTSN is connected to a receivers RTSN, overrun can effectively be prevented.

12.2.2 Receiver operation

The receiver is enabled for data reception through the receiver enable (RE) bit in the UART control register. The receiver looks for a high to low transition of a start bit on the receiver serial data input pin. If a transition is detected, the state of the serial input is sampled a half bit clocks later. If the serial input is sampled high the start bit is invalid and the search for a valid start bit continues. If the serial input is still low, a valid start bit is assumed and the receiver continues to sample the serial input at one bit time intervals (at the theoretical centre of the bit) until the proper number of data bits and the parity bit have been assembled and one stop bit has been detected. The serial input is shifted through an 8-bit shift register where all bits have to have the same value before the new value is taken into account, effectively forming a low-pass filter with a cut-off frequency of 1/8 system clock.

The receiver also has a configurable FIFO which is identical to the one in the transmitter. As mentioned in the transmitter part, both the holding register and FIFO will be referred to as FIFO.

During reception, the least significant bit is received first. The data is then transferred to the receiver FIFO and the data ready (DR) bit is set in the UART status register as soon as the FIFO contains at least one data frame. The parity, framing and overrun error bits are set at the received byte boundary, at the same time as the receiver ready bit is set. The data frame is not stored in the FIFO if an error is detected. Also, the new error status bits are or:ed with the old values before they are stored into the status register. Thus, they are not cleared until written to with zeros from the APB bus. If both the receiver FIFO and shift registers are full when a new start bit is detected, then the character held in the receiver shift register will be lost and the overrun bit will be set in the UART status register. If flow control is enabled, then the RTSN

will be negated (high) when a valid start bit is detected and the receiver FIFO is full. When the holding register is read, the RTSN will automatically be reasserted again.

When fifosize > 1, which means that holding registers are not considered here, some additional status and control bits are available. The RF status bit (not to be confused with the RF control bit) is set when the receiver FIFO is full. The RH status bit is set when the receiver FIFO is half-full (at least half of the entries in the FIFO contain data frames). The RF control bit enables receiver FIFO interrupts when set. A RCNT field is also available showing the current number of data frames in the FIFO.

12.3 Baud-rate generation

Each UART contains a 12-bit down-counting scaler to generate the desired baud-rate. The scaler is clocked by the system clock and generates a UART tick each time it underflows. It is reloaded with the value of the UART scaler reload register after each underflow. The resulting UART tick frequency should be 8 times the desired baud-rate. If the EC bit is set, the scaler will be clocked by the UARTI.EXTCLK input rather than the system clock. In this case, the frequency of UARTI.EXTCL must be less than half the frequency of the system clock.

12.3.1 Loop back mode

If the LB bit in the UART control register is set, the UART will be in loop back mode. In this mode, the transmitter output is internally connected to the receiver input and the RTSN is connected to the CTSN. It is then possible to perform loop back tests to verify operation of receiver, transmitter and associated software routines. In this mode, the outputs remain in the inactive state, in order to avoid sending out data.

12.3.2 Interrupt generation

Interrupts are generated differently when a holding register is used (fifosize = 1) and when FIFOs are used (fifosize > 1). When holding registers are used, the UART will generate an interrupt under the following conditions: when the transmitter is enabled, the transmitter interrupt is enabled and the transmitter holding register moves from full to empty; when the receiver is enabled, the receiver interrupt is enabled and the receiver holding register moves from empty to full; when the receiver is enabled, the receiver interrupt is enabled and a character with either parity, framing or overrun error is received.

For FIFOs two different kinds of interrupts are available: normal interrupts and FIFO interrupts. For the transmitter, normal interrupts are generated when transmitter interrupts are enabled (TI), the transmitter is enabled and the transmitter FIFO goes from containing data to being empty. FIFO interrupts are generated when the FIFO interrupts are enabled (TF), transmissions are enabled (TE) and the UART is less than half-full (that is, whenever the TH status bit is set). This is a level interrupt and the interrupt signal is continuously driven high as long as the condition prevails. The receiver interrupts work in the same way. Normal interrupts are generated in the same manner as for the holding register. FIFO interrupts are generated when receiver FIFO interrupts are enabled, the receiver is enabled and the FIFO is half-full. The interrupt signal is continuously driven high as long as the receiver FIFO is half-full (at least half of the entries contain data frames).

12.4 Configuration options

The APB UART has the following configuration options (VHDL generics):

TABLE 41. APB UART configuration options (VHDL generics)

Generic	Function	Allowed range	Default
pindex	APB slave index	0 - NAPBSLV-1	0
paddr	ADDR field of the APB BAR.	0 - 16#FFF#	0
pmask	MASK field of the APB BAR.	0 - 16#FFF#	16#FFF#
console	Prints output from the UART on console during VHDL simulation and speeds up simulation by always returning '1' for Data Ready bit of UART Status register. Does not effect synthesis.	0 - 1	0
pirq	Index of the interrupt line.	0 - NAHBIRQ-1	0
parity	Enables parity	0 - 1	1
flow	Enables flow control	0 - 1	1
fifosize	Selects the size of the Receiver and Transmitter FIFOs	1, 2, 4, 8, 16, 32	1

12.5 Vendor and device id

The module has vendor id 0x01 (Gaisler Research) and device id 0x00C. For a description of vendor and device ids see GRLIB IP Library User's Manual.

12.6 UART registers

The APB UART is controlled through four registers mapped into APB address space.

TABLE 42. APB Uart registers

Register	APB Address offset
UART Data register	0x0
UART Status register	0x4
UART Control register	0x8
UART Scaler register	0xC

12.6.1 UART Data Register

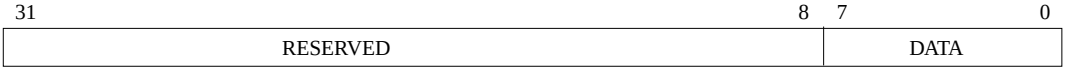

Figure 22. UART data register

[7:0]: Receiver holding register or FIFO (read access)
[7:0]: Transmitter holding register or FIFO (write access)

12.6.2 UART Status Register

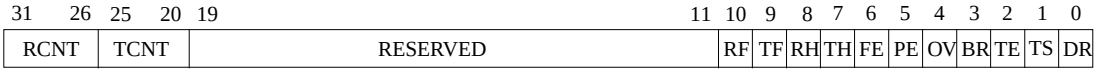

Figure 23. UART status register

0: Data ready (DR) - indicates that new data is available in the receiver holding register.
1: Transmitter shift register empty (TS) - indicates that the transmitter shift register is empty.
2: Transmitter FIFO empty (TE) - indicates that the transmitter FIFO is empty.
3: Break received (BR) - indicates that a BREAK has been received.
4: Overrun (OV) - indicates that one or more character have been lost due to overrun.
5: Parity error (PE) - indicates that a parity error was detected.
6: Framing error (FE) - indicates that a framing error was detected.
7: Transmitter FIFO half-full (TH) - indicates that the FIFO is less than half-full.
8: Receiver FIFO half-full (RH) -indicates that at least half of the FIFO is holding data.
9: Transmitter FIFO full (TF) - indicates that the Transmitter FIFO is full.
10: Receiver FIFO full (RF) - indicates that the Receiver FIFO is full.
[25:20] : Transmitter FIFO count - shows the number of data frames in the transmitter FIFO.
[31:26] : Receiver FIFO count (RCNT) - shows the number of data frames in the receiver FIFO.

12.6.3 UART Control Register

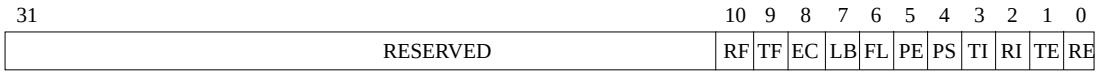

Figure 24. UART control register

0: Receiver enable (RE) - if set, enables the receiver.
1: Transmitter enable (TE) - if set, enables the transmitter.
2: Receiver interrupt enable (RI) - if set, interrupts are generated when a frame is received
3: Transmitter interrupt enable (TI) - if set, interrupts are generated when a frame is transmitted
4: Parity select (PS) - selects parity polarity (0 = even parity, 1 = odd parity)
5: Parity enable (PE) - if set, enables parity generation and checking.
6: Flow control (FL) - if set, enables flow control using CTS/RTS.
7: Loop back (LB) - if set, loop back mode will be enabled.
8: External Clock (EC) - if set, the UART scaler will be clocked by UARTI.EXTCLK
9: Transmitter FIFO interrupt enable (TF) - when set, Transmitter FIFO level interrupts are enabled.
10: Receiver FIFO interrupt enable (RF) - when set, Receiver FIFO level interrupts are enabled.

12.6.4 UART Scaler Register

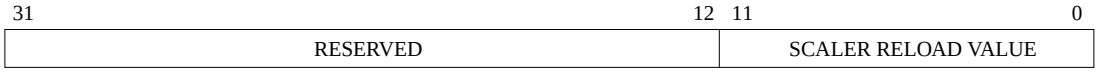

Figure 25. UART scaler reload register

12.6.5 Signal descriptions

APB UART signals are described in table 43.

TABLE 43. APB UART signal descriptions.

Signal name	Field	Type	Function	Active
RST	N/A	Input	Reset	Low
CLK	N/A	Input	Clock	-
APBI	*	Input	APB slave input signals	-
APBO	*	Output	APB slave output signals	-
UARTI	RXD	Input	UART receiver data	-
	CTSN	Input	UART clear-to-send	Low
	EXTCLK	Input	Use as alternative UART clock	-
UARTO	RTSN	Output	UART request-to-send	Low
	TXD	Output	UART transmit data	-

* see GRLIB IP Library User's Manual

12.7 Library dependencies

Table 44 shows libraries that should be used when instantiating an APB UART.

TABLE 44. Library dependencies

Library	Package	Imported unit(s)	Description
GRLIB	AMBA	Signals	APB signal definitions
GAISLER	UART	Signals, component	UART signal and component declaration

12.8 APB UART instantiation

This examples shows how an APB UART can be instantiated.

```
library ieee;
use ieee.std_logic_1164.all;

library grlib;
use grlib.amba.all;
library gaisler;
use gaisler.uart.all;

entity apbuart_ex is
  port (
    clk : in std_ulogic;
    rstn : in std_ulogic;

    -- UART signals
    rxd   : in  std_ulogic;
    txd   : out std_ulogic
    );
end;
```

```
architecture rtl of apbuart_ex is

  -- APB signals
  signal apbi  : apb_slv_in_type;
  signal apbo  : apb_slv_out_vector := (others => apb_none);

  -- UART signals
  signal uarti : uart_in_type;
  signal uarto : uart_out_type;

begin

  -- AMBA Components are instantiated here
   ...


  -- APB UART
  uart0 : apbuart
  generic map (pindex => 1, paddr => 1,  pirq => 2,
 console => 1, fifosize => 1)
  port map (rstn, clk, apbi, apbo(1), uarti, uarto);

  -- UART input data
  uarti.rxd <= rxd;

  -- APB UART inputs not used in this configuration
  uarti.ctsn <= '0'; uarti.extclk <= '0';

  -- connect APB UART output to entity output signal
  txd <= uarto.txd;

end;
```

13 CAN_OC - GRLIB wrapper for Opencore CAN core

13.1 Overview

CAN_OC is GRLIB wrapper for the CAN core from Opencores. It provides a bridge between AMBA AHB and the Wishbone bus, which is used to access the CAN MAC registers. The AHB slave interface is mapped in the AHB I/O space using the GRLIB plug&play functionality. The CAN core interrupt is routed to the AHB interrupt bus, and the interrupt number is selected through the *irq* generic. The FIFO RAM in the CAN core is implemented using the GRLIB parametrizable SYNCRAM_2P memories, assuring portability to all supported technologies.

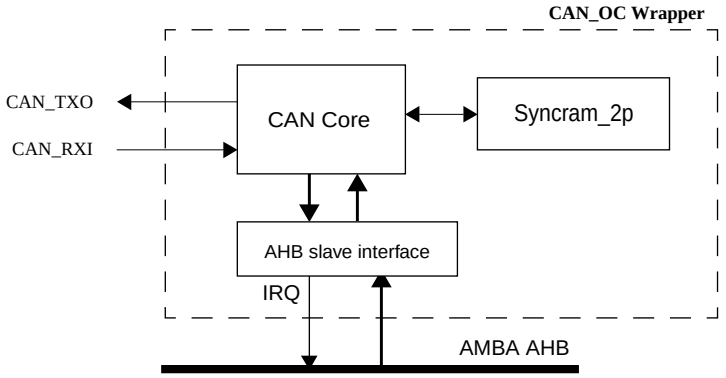

Figure 26. CAN_OC wrapper block diagram

13.2 Configuration options

The CAN_OC wrapper has the following configuration options (VHDL generics):

TABLE 45. VHDL Generics

Generic	Function	Allowed range	Default
slvndx	AHB slave bus index	0 - NAHBSLV-1	0
ioaddr	The MSB address of the I/O area. Sets the 12 most significant bits in the 32-bit AHB address.	0 - 16#FFF#	16#FFF#
iomask	The I/O area address mask. Sets the size of the I/O area and the start address together with ioaddr.	0 - 16#FFF#	16#FF0#
irq	Interrupt number	0 - NAHBIRQ-1	0
memtech	Technology to implement on-chip RAM	0	0 - NTECH

13.3 Vendor and device id

The module has vendor id 0x01 (Gaisler Research) and device id 0x019. For description of vendor and device ids see GRLIB IP Library User’s Manual.

13.4 Signal descriptions

The CAN_OC signals are described in Table 46.

TABLE 46. CAN_OC signals

Signal name	Field	Type	Function	Active
CLK		Input	AHB clock	
RESETN		Input	Reset	Low
AHBSI	*	Input	AMBA AHB slave inputs	-
AHBSO	*	Input	AMBA AHB slave outputs	
CAN_RXI		Input	CAN receiver input	High
CAN_TXO		Output	CAN transmitter output	High

*1) see AMBA specification

13.5 Library dependencies

Table 47 shows libraries that should be used when instantiating the CAN_OC.

TABLE 47. Library dependencies

Library	Package	Imported unit(s)	Description
GRLIB	AMBA	Types	AMBA signal type definitions
WORK	GRCOMP	Component	Component declaration

13.6 Component declaration

```
library grlib;
use grlib.amba.all;
use work.grcomp.all;

 component can_oc
   generic (
    slvndx    : integer := 0;
    ioaddr    : integer := 16#000#;
    iomask    : integer := 16#FF0#;
    irq       : integer := 0;
    memtech   : integer := 0);
   port (
      resetn  : in  std_logic;
      clk     : in  std_logic;
      ahbsi   : in  ahb_slv_in_type;
      ahbso   : out ahb_slv_out_type;
      can_rxi : in  std_logic;
      can_txo : out std_logic
     );
   end component;
```

14 DIV32 - Signed/unsigned 64/32 divider module

14.1 Overview

The divider module performs signed/unsigned 64-bit by 32-bit division. It implements the radix-2 non-restoring iterative division algorithm. The division operation takes 36 clock cycles. The divider leaves no remainder. The result is rounded towards zero. Negative result, zero result and overflow (according to the overflow detection method B of SPARC V8 Architecture manual) are detected.

14.2 Operation

The division is started when ‘1’ is samples on DIVI.START on positive clock edge. Operands are latched externally and provided on inputs DIVI.Y, DIVI.OP1 and DIVI.OP2 during the whole operation. The result appears on the outputs during the clock cycle following the clock cycle after the DIVO.READY was asserted. Asserting the HOLD input at any time will freeze the operation, until HOLDN is de-asserted.

14.3 Signal description

The divider module signals are described in table 48.

TABLE 48. Divider module signals

Signal name	Field	Type	Function	Active
RST	N/A	Input	Reset	Low
CLK	N/A	Input	Clock	-
HOLDN	N/A	Input	Hold	Low
DIVI	Y[32:0]	Input	Dividend - MSB part Y[32] - Sign bit Y[31:0] - Dividend MSB part in 2’s complement format	High
	OP1[32:0]		Dividend - LSB part OP1[32] - Sign bit OP1[31:0] - Dividend LSB part in 2’s complement format	High
	FLUSH		Flush current operation	High
	SIGNED		Signed division	High
	START		Start division	High
DIVO	READY	Output	The result is available one clock after the ready signal is asserted.	High
	NREADY		Not used	-
	ICC[3:0]		Condition codes ICC[3] - Negative result ICC[2] - Zero result ICC[1] - Overflow ICC[0] - Not used. Always ‘0’.	High
	RESULT[31:0]		Result	High

14.4 Library dependencies

Table 49 shows libraries required when instantiating the divider module.

TABLE 49. Library dependencies

Library	Package	Imported unit(s)	Description
GAISLER	ARITH	Signals, component	Divider module signals, component declaration

14.5 Model interface

The divider unit has the following component declaration.

```
component div32
port (
    rst     : in  std_ulogic;
    clk     : in  std_ulogic;
    holdn   : in  std_ulogic;
    divi    : in  div32_in_type;
    divo    : out div32_out_type
);
end component;
```

14.6 Example instantiation

The VHDL-code below shows how the divider module can be instantiated.

```
library ieee;
use ieee.std_logic_1164.all;

library grlib;
use gaisler.arith.all;

.
.
.

signal divi  : div32_in_type;
signal divo  : div32_out_type;

begin

div0 : div32 port map (rst, clk, holdn, divi, divo);

end;
```

15 DSU3 - LEON3 Hardware debug support unit

15.1 Introduction

To simplify debugging on target hardware, the LEON3 processor implements a debug mode during which the pipeline is idle and the processor is controlled through a special debug interface. The LEON3 Debug Support Unit (DSU) is used to control the processor during debug mode. The DSU acts as an AHB slave and can be accessed by any AHB master. An external debug host can therefore access the DSU through several different interfaces, such as a serial UART (RS232), JTAG, PCI or ethernet. The DSU supports multi-processor systems and can handle up to 16 processors.

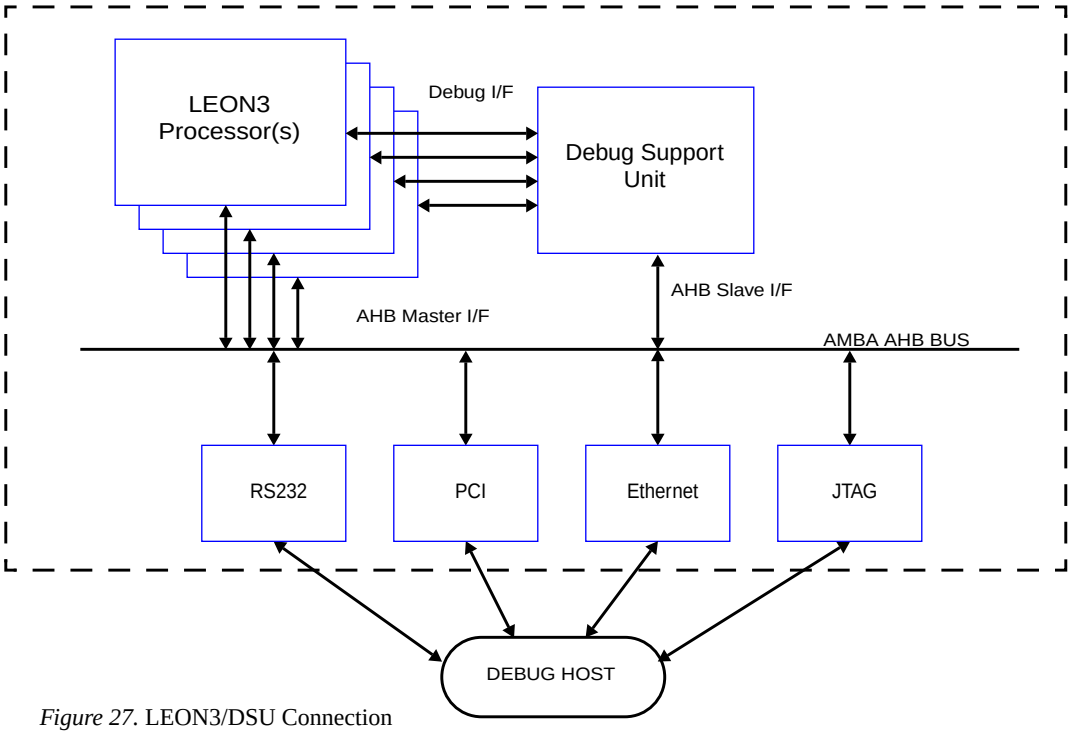

Figure 27. LEON3/DSU Connection

15.2 Operation

Through the DSU AHB slave interface, any AHB master can access the processor registers and the contents of the instruction trace buffer. The DSU control registers can be accessed at any time, while the processor registers, caches and trace buffer can only be accessed when the processor has entered debug mode. In debug mode, the processor pipeline is held and the processor state can be accessed by the DSU. Entering the debug mode can occur on the following events:

- executing a breakpoint instruction (ta 1)
- integer unit hardware breakpoint/watchpoint hit (trap 0xb)
- rising edge of the external break signal (DSUBRE)
- setting the break-now (BN) bit in the DSU control register
- a trap that would cause the processor to enter error mode
- occurrence of any, or a selection of traps as defined in the DSU control register
- after a single-step operation
- one of the processors in a multiprocessor system has entered the debug mode
- DSU breakpoint hit

The debug mode can only be entered when the debug support unit is enabled through an external pin (DSUEN). When the debug mode is entered, the following actions are taken:

- PC and nPC are saved in temporary registers (accessible by the debug unit)
- an output signal (DSUACT) is asserted to indicate the debug state
- the timer unit is (optionally) stopped to freeze the LEON timers and watchdog

The instruction that caused the processor to enter debug mode is not executed, and the processor state is kept unmodified. Execution is resumed by clearing the BN bit in the DSU control register or by de-asserting DSUEN. The timer unit will be re-enabled and execution will continue from the saved PC and nPC. Debug mode can also be entered after the processor has entered error mode, for instance when an application has terminated and halted the processor. The error mode can be reset and the processor restarted at any address.

When a processor is in the debug mode, an access to ASI diagnostic area is forwarded to the IU which performs access with ASI equal to value in the DSU ASI register and address consisting of 20 LSB bits of the original address.

15.3 AHB Trace Buffer

The AHB trace buffer consists of a circular buffer that stores AHB data transfers. The address, data and various control signals of the AHB bus are stored and can be read out for later analysis. The trace buffer is 128 bits wide, the information stored is indicated in the table below:

TABLE 50. AHB Trace buffer data allocation

Bits	Name	Definition
127	AHB breakpoint hit	Set to ‘1’ if a DSU AHB breakpoint hit occurred.
126	-	Not used
125:96	Time tag	DSU time tag counter
95	-	Not used
94:80	Hirq	AHB HIRQ[15:1]
79	Hwrite	AHB HWRITE
78:77	Htrans	AHB HTRANS
76:74	Hsize	AHB HSIZE
73:71	Hburst	AHB HBURST
70:67	Hmaster	AHB HMASTER
66	Hmastlock	AHB HMASTLOCK
65:64	Hresp	AHB HRESP
63:32	Load/Store data	AHB HRDATA or HWDATA
31:0	Load/Store address	AHB HADDR

In addition to the AHB signals, the DSU time tag counter is also stored in the trace.

The trace buffer is enabled by setting the enable bit (EN) in the trace control register. Each AHB transfer is then stored in the buffer in a circular manner. The address to which the next transfer is written is held in the trace buffer index register, and is automatically incremented after each transfer. Tracing is stopped when the EN bit is reset, or when a AHB breakpoint is hit. Tracing is temporarily suspended when the processor enters debug mode. Note that neither the trace buffer memory nor the breakpoint registers (see below) can be read/written by software when the trace buffer is enabled.

15.4 Instruction trace buffer

The instruction trace buffer consists of a circular buffer that stores executed instructions. The instruction trace buffer is located in the processor, and read out via the DSU. The trace buffer is 128 bits wide, the information stored is indicated in the table below:

TABLE 51. Instruction trace buffer data allocation

Bits	Name	Definition
127	-	Unused
126	Multi-cycle instruction	Set to ‘1’ on the second and third instance of a multi-cycle instruction (LDD, ST or FPOP)
125:96	Time tag	The value of the DSU time tag counter
95:64	Load/Store parameters	Instruction result, Store address or Store data
63:34	Program counter	Program counter (2 lsb bits removed since they are always zero)
33	Instruction trap	Set to ‘1’ if traced instruction trapped
32	Processor error mode	Set to ‘1’ if the traced instruction caused processor error mode
31:0	Opcode	Instruction opcode

During tracing, one instruction is stored per line in the trace buffer with the exception of multi-cycle instructions. Multi-cycle instructions are entered two or three times in the trace buffer. For store instructions, bits [63:32] correspond to the store address on the first entry and to the stored data on the second entry (and third in case of STD). Bit 126 is set on the second and third entry to indicate this. A double load (LDD) is entered twice in the trace buffer, with bits [63:32] containing the loaded data. Multiply and divide instructions are entered twice, but only the last entry contains the result. Bit 126 is set for the second entry. For FPU operation producing a double-precision result, the first entry puts the MSB 32 bits of the results in bit [63:32] while the second entry puts the LSB 32 bits in this field.

When the processor enters debug mode, tracing is suspended. The trace buffer and the trace buffer control register can be read and written while the processor is in the debug mode. During the instruction tracing (processor in normal mode) the trace buffer and the trace buffer control register can not be accessed.

15.5 DSU memory map

The DSU memory map can be seen in table 52 below. In a multiprocessor systems, the register map is duplicated and address bits 27 - 24 are used to index the processor.

TABLE 52. DSU memory map

Address offset	Register
0x000000	DSU control register
0x000008	Time tag counter
0x000020	Break and Single Step register
0x000024	Debug Mode Mask register
0x000040	AHB trace buffer control register
0x000044	AHB trace buffer index register
0x000050	AHB breakpoint address 1
0x000054	AHB mask register 1
0x000058	AHB breakpoint address 2
0x00005c	AHB mask register 2
0x100000 - 0x110000	Instruction trace buffer (..0: Trace bits 127 - 96, ..4: Trace bits 95 - 64, ..8: Trace bits 63 - 32, ..C : Trace bits 31 - 0)
0x110000	Intruction Trace buffer control register
0x200000 - 0x210000	AHB trace buffer (..0: Trace bits 127 - 96, ..4: Trace bits 95 - 64, ..8: Trace bits 63 - 32, ..C : Trace bits 31 - 0)
0x300000 - 0x300FFC	IU register file
0x301000 - 0x30107C	FPU register file
0x400000 - 0x4FFFFC	IU special purpose registers
0x400000	Y register
0x400004	PSR register
0x400008	WIM register
0x40000C	TBR register
0x400010	PC register
0x400014	NPC register
0x400018	FSR register
0x40001C	CPSR register
0x400020	DSU trap register
0x400024	DSU ASI register
0x400040 - 0x40007C	ASR16 - ASR31 (when implemented)
0x700000 - 0x7FFFFC	ASI diagnostic access (ASI = value in DSU ASI register, address = address[19:0]) ASI = 0x9 : Local instruction RAM ASI = 0xB : Local data RAM ASI = 0xC : Instruction cache tags ASI = 0xD : Instruction cache data ASI = 0xE : Data cache tags ASI = 0xF : Instruction cache data

The addresses of the IU registers depends on how many register windows has been implemented:

- %o*n* : 0x300000 + (((psr.cwp * 64) + 32 + *n**4) mod (NWINDOWS*64))
- %l*n* : 0x300000 + (((psr.cwp * 64) + 64 + *n**4) mod (NWINDOWS*64))
- %i*n* : 0x300000 + (((psr.cwp * 64) + 96 + *n**4) mod (NWINDOWS*64))
- %g*n* : 0x300000 + (NWINDOWS*64)
- %f*n* : 0x301000 + *n**4

15.6 DSU registers

15.6.1 DSU control register

The DSU is controlled by the DSU control register:

.

Figure 28. DSU control register

[0]: Trace enable (TE). Enables instruction tracing. If set the instructions will be stored in the trace buffer. Remains set when then processor enters debug or error mode.

[1]: Break on error (BE) - if set, will force the processor to debug mode when the processor would have entered error condition (trap in trap).

[2]: Break on IU watchpoint (BW)- if set, debug mode will be forced on a IU watchpoint (trap 0xb).

[3]: Break on S/W breakpoint (BS) - if set, debug mode will be forced when an breakpoint instruction (ta 1) is executed.

[4]: Break on trap (BX) - if set, will force the processor into debug mode when any trap occurs.

[5]: Break on error traps (BZ) - if set, will force the processor into debug mode on all *except* the following traps: priviledged_instruction, fpu_disabled, window_overflow, window_underflow, asynchronous_interrupt, ticc_trap.

[6]: Debug mode (DM). Indicates when the processor has entered debug mode (read-only).

[7]: EB - value of the external DSUBRE signal (read-only)

[8]: EE - value of the external DSUEN signal (read-only)

[9]: Processor error mode (PE) - returns ‘1’ on read when processor is in error mode, else ‘0’. If written with ‘1’, it will clear the error and halt mode.

[10]: Processor halt (HL). Returns ‘1’ on read when processor is halted. If the processor is in debug mode, setting this bit will put the processor in halt mode.

15.6.2 DSU Break and Single Step register

This register is used to break or single step the processor(s). This register controls all processors in a multi-processor system, and is only accessible in the DSU memory map of processor 0.

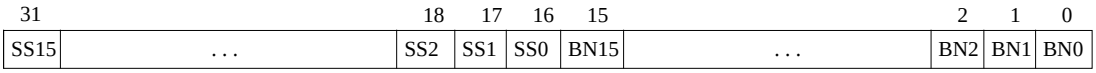

Figure 29. DSU Break and Single Step register

[15:0] : Break now (BNx) -Force processor x into debug mode if the Break on S/W breakpoint (BS) bit in the processors DSU control register is set. If cleared, the processor x will resume execution.

[31:16] : Single step (SSx) - if set, the processor x will execute one instruction and return to debug mode. The bit remains set after the processor goes into the debug mode.

15.6.3 DSU Debug Mode Mask Register

When one of the processors in a multiprocessor LEON3 system enters the debug mode the value of the DSU Debug Mode Mask register determines if the other processors are forced in the debug mode. This register controls all processors in a multi-processor system, and is only accessible in the DSU memory map of processor 0.

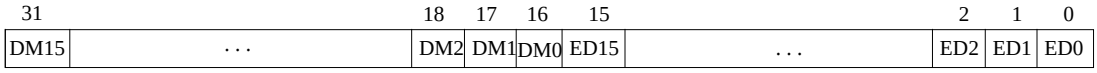

Figure 30. DSU Debug Mode Mask register

[15:0] : Enter debug mode (EDx) - Force processor x into debug mode if any of processors in a multiprocessor system enters the debug mode. If 0, the processor x will not enter the debug mode.

[31:16]: Debug mode mask. If set, the corresponding processor will not be able to force running processors into debug mode even if it enters debug mode.

15.6.4 DSU trap register

The DSU trap register is a read-only register that indicates which SPARC trap type that caused the processor to enter debug mode. When debug mode is force by setting the BN bit in the DSU control register, the trap type will be 0xb (hardware watchpoint trap).

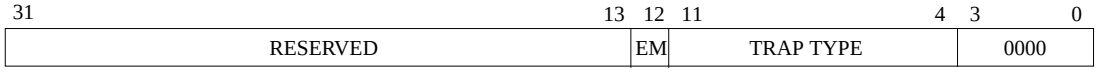

Figure 31. DSU trap register

[11:4]: 8-bit SPARC trap type

[12]: Error mode (EM). Set if the trap would have cause the processor to enter error mode.

15.6.5 Trace buffer time tag counter

The trace buffer time tag counter is incremented each clock as long as the processor is running. The counter is stopped when the processor enters debug mode, and restarted when execution is resumed. The value is used as time tag in the instruction and AHB trace buffer.

Figure 32. Trace buffer time tag counter

The width of the timer (up to 30 bits) is configurable through the DSU generic port.

15.6.6 DSU ASI register

The DSU can perform diagnostic accesses to different ASI areas. The value in the ASI diagnostic access register is used as ASI while the address is supplied from the DSU.

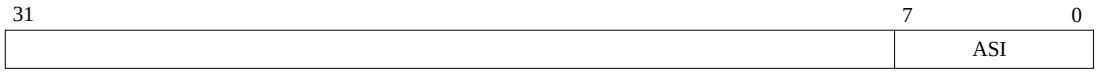

Figure 33. ASI diagnostic access register

[7:0]: ASI to be used on diagnostic ASI access

15.6.7 AHB Trace buffer control register

The AHB trace buffer is controlled by the AHB trace buffer control register:

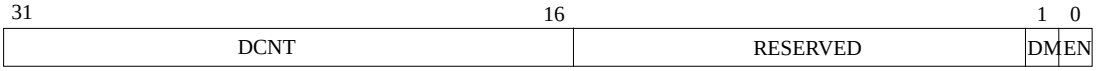

Figure 34. AHB trace buffer control register

[0]: Trace enable (EN). Enables the trace buffer.

[1]: Delay counter mode (DM). Indicates that the trace buffer is in delay counter mode.

[31:16] Trace buffer delay counter (DCNT). Note that the number of bits actually implemented depends on the size of the trace buffer.

15.6.8 AHB trace buffer index register

The AHB trace buffer index register contains the address of the next trace line to be written.

Figure 35. AHB trace buffer index register

31:4 Trace buffer index counter (INDEX). Note that the number of bits actually implemented depends on the size of the trace buffer.

15.6.9 AHB trace buffer breakpoint registers

The DSU contains two breakpoint registers for matching AHB addresses. A breakpoint hit is used to freeze the trace buffer by automatically clearing the enable bit. Freezing can be delayed by programming the DCNT field in the trace buffer control register to a non-zero value. In this case, the DCNT value will be decremented for each additional trace until it reaches zero, after which the trace buffer is frozen. A mask register is associated with each breakpoint, allowing breaking on a block of addresses. Only address bits with the corresponding mask bit set to ‘1’ are compared during breakpoint detection. To break on AHB load or store accesses, the LD and/or ST bits should be set.

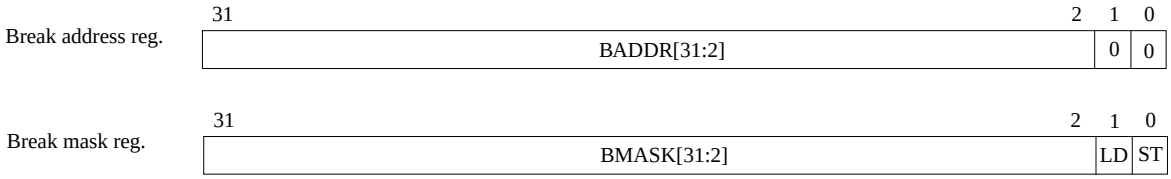

Figure 36. Trace buffer breakpoint registers

[31:2]: Breakpoint address (bits 31:2)

[31:2]: Breakpoint mask (see text)

[1]: LD - break on data load address

[0]: ST - beak on data store address

15.6.10 Instruction trace control register

The instruction trace control register contains a pointer that indicates the next line of the instruction trace buffer to be written.

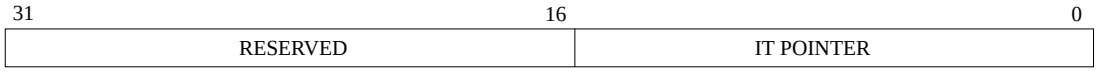

Figure 37. Instruction trace control register

[15:0] Instruction trace pointer. Note that the number of bits actually implemented depends on the size of the trace buffer.

15.7 Configuration and synthesis

15.7.1 Plug&play configuration

The LEON3 DSU is identified with the following vendor and device ID's:

TABLE 53. LEON3 DSU plug&play ID

Vendor ID	Device ID
0x01 (GAISLER)	0x017

15.7.2 Configuration options

The table below describes the generic parameters:

TABLE 54. DSU Generic parameters

Generic	Description	Range	Default
hindex	AHB slave index	0 - AHBSLVMAX-1	0
haddr	AHB slave address (AHB[31:20])	0 - 16#FFF#	16#900#
hmask	AHB slave address mask	0 - 16#FFF#	16#F00#
ncpu	Number of attached processors	1 - 16	1
tbits	Number of bits in the time tag counter	2 - 30	30
tech	Memory technology for trace buffer RAM	0 - TECHMAX-1	0 (inferred)
kbytes	Size of trace buffer memory in Kbytes. A value of 0 will disable the trace buffer function.	0 - 64	0 (disabled)

15.7.3 Signal description

TABLE 55. Signal description

Signal name	Field	Type	Function	Active
RST	N/A	Input	Reset	Low
CLK	N/A	Input	Clock	-
AHBMI	*	Input	AHB master input signals	-
AHBSI	*	Input	AHB slave input signals	-
AHBSO	*	Output	AHB slave output signals	-
DBGI	-	Input	Debug signals from LEON3	-
DBGO	-	Output	Debug signals to LEON3	-
DSUI	ENABLE	Input	DSU enable	High
	BREAK	Input	DSU break	High
DSUO	ACTIVE	Output	Debug mode	High

* see GRLIB IP Library User's Manual

15.7.4 Library dependencies

Table shows libraries that the DSU module depends on.

TABLE 56. Library dependencies

Library	Package	Imported unit(s)	Description
GRLIB	AMBA	Signals	AHB signal definitions
GAISLER	LEON3	Component, signals	DSU3 component declaration, signals declaration

15.7.5 Model interface

The DSU has the following component declaration:

```
component dsu
  generic (
    hindex : integer := 0;
    haddr : integer := 16#900#;
    hmask : integer := 16#f00#;
    ncpu    : integer := 1;
    tbits   : integer := 30;
    tech    : integer := 0;
    irq     : integer := 0;
    kbytes  : integer := 0
  );
  port (
    rst    : in  std_ulogic;
    clk    : in  std_ulogic;
    ahbmi  : in  ahb_mst_in_type;
    ahbsi  : in  ahb_slv_in_type;
    ahbso  : out ahb_slv_out_type;
    dbgi   : in l3_debug_out_vector(0 to NCPU-1);
    dbgo   : out l3_debug_in_vector(0 to NCPU-1);
    dsui   : in dsu_in_type;
    dsuo   : out dsu_out_type
  );
  end component;
```

-

15.7.6 Example instantiation

The DSU is always instantiated with at least one LEON3 processor. It is suitable to use a generate loop for the instantiation of the processors and DSU and showed below:

```
library ieee;
use ieee.std_logic_1164.all;
library grlib;
use grlib.amba.all;
library gaisler;
use gaisler.leon3.all;

.
.
constant NCPU : integer := 1; -- select number of processors

signal leon3i : l3_in_vector(0 to NCPU-1);
signal leon3o : l3_out_vector(0 to NCPU-1);
signal irqi   : irq_in_vector(0 to NCPU-1);
signal irqo   : irq_out_vector(0 to NCPU-1);

signal dbgi : l3_debug_in_vector(0 to NCPU-1);
signal dbgo : l3_debug_out_vector(0 to NCPU-1);

signal dsui   : dsu_in_type;
signal dsuo   : dsu_out_type;

.
.

begin

cpu : for i in 0 to NCPU-1 generate
    u0 : leon3s    -- LEON3 processor
    generic map (ahbndx => i, fabtech => FABTECH, memtech => MEMTECH)
    port map (clkm, rstn, ahbmi, ahbmo(i), ahbsi, ahbsi, ahbso,
    irqi(i), irqo(i), dbgi(i), dbgo(i));
    irqi(i) <= leon3o(i).irq; leon3i(i).irq <= irqo(i);
end generate;

dsu0 : dsu         -- LEON3 Debug Support Unit
    generic map (ahbndx => 2, ncpu => NCPU, tech => memtech, kbytes => 2)
    port map (rstn, clkm, ahbmi, ahbsi, ahbso(2), dbgo, dbgi, dsui, dsuo);

dsui.enable <= dsuen; dsui.break <= dsubre; dsuact <= dsuo.active;
```

16 EDCL - Ethernet Debug communication Link

16.1 Overview

The Ethernet Debug Communication Link provides access to an on-chip AHB bus through Ethernet. It uses the UDP, IP and ARP protocols for the transmissions. A custom application layer protocol is also used, implementing an ARQ algorithm and the AHB instructions. Through this link, a read or write transfer can be generated to any address on the AHB bus. A block diagram of the hardware module is found in figure 38.

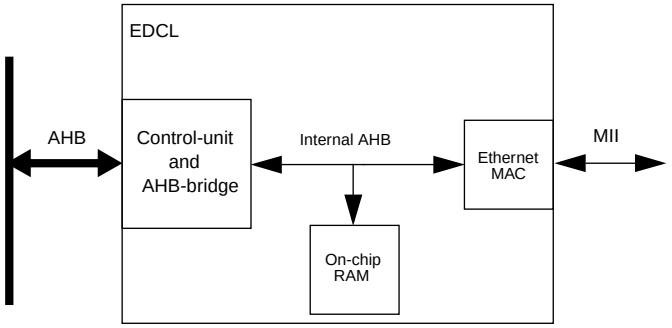

Figure 38. EDCL block diagram

16.2 Operation

16.2.1 Hardware module

The EDCL has external AHB and MII interfaces. The MII interface should be connected to a PHY which handles the physical signaling on the Ethernet medium. Packets arriving on the Ethernet conforming to the EDCL protocol are converted to AHB transfers. The transfers are performed as incremental bursts of unspecified length with word size (32-bit). The EDCL can insert busy cycles in the middle of bursts.

Internally, the EDCL consists of a control-unit, an AHB-bridge, an on-chip RAM and an Ethernet MAC. The Ethernet MAC is from Opencores and handles the Ethernet communication. It reads and writes frames from the on-chip RAM which is used as a temporary buffer. The control-unit initiates transmissions and receptions in the MAC and processes all packets. It checks packet types, IP-addresses, UDP ports and handles the ARQ protocol. Finally, it performs the AHB transfers on the external AHB bus when a correct EDCL packet has been received.

16.2.2 Transmission protocol

The EDCL accepts Ethernet frames containing IP or ARP data. ARP is handled according to the protocol specification with no exceptions.

IP packets carry the actual AHB transfer commands. The EDCL expects an Ethernet frame containing IP, UDP and the EDCL specific application layer parts. Table 57 shows the IP packet required by the EDCL. The contents of the different protocol headers can be found in most text about TCP/IP.

TABLE 57. The IP packet expected by the EDCL.

Ethernet Header	IP Header	UDP Header	2 B Offset	4 B Control word	4 B Address	Data 0 - 242 4B Words	Ethernet CRC

The following is required for successful communication: A correct destination MAC address as set by the generics. This is checked by the Ethernet MAC. The control-unit then checks that either

an ARP or an IP packet is received by checking that the Ethernet type field is 0x0806 or 0x0800. The IP-address is then compared with the value determined by the generics for a match. The IP-header checksum and identification fields are not checked but the transmitted identification field is incremented for each frame. There are a few restrictions on the IP-header fields. The version must be four and the header size must be 5 B (no options). Type-of-service, fragment-offset and flags fields must be set to zero. The protocol field must always be 0x11 indicating a UDP packet.

The EDCL only provides one service at the moment and it is therefore not required to check the UDP port number. It should still be specified (generics) for compatibility reasons. UDP checksums are not used and the checksum field is set to zero in the replies.

The UDP data field contains the EDCL application protocol fields. Table 58 shows the application protocol fields (data field excluded) in packets received by the EDCL. The 16-bit offset is used to align the rest of the application layer data to word boundaries in memory and can thus be set to any value. The R/W field determines whether a read (0) or a write(1) should be

TABLE 58. The EDCL application layer fields in received frames.

16-bit Offset	14-bit Sequence number	1-bit R/W	10-bit Length	7-bit Unused

performed. The length field contains the number of bytes to be read or written. If R/W is one the data field shown in Table 57 contains the data to be written. If R/W is zero the data field is empty in the received packets. Table 59 shows the application layer fields of the replies from the EDCL. The length field is always zero for replies to write requests. For read requests it contains the number of bytes of data contained in the data field.

TABLE 59. The EDCL application layer fields in transmitted frames.

16-bit Offset	14-bit sequence number	1-bit ACK/NAK	10-bit Length	7-bit Unused

The EDCL implements a Go-Back-N algorithm providing reliable transfers. The 14-bit sequence number in received packets are checked against an internal counter for a match. If they do not match, no operation is performed and the ACK/NAK field is set to 1 in the reply frame. The reply frame contains the internal counter value in the sequence number field. If the sequence number matches, the operation is performed, the internal counter is incremented, the internal counter value is stored in the sequence number field and the ACK/NAK field is set to 0 in the reply. The length field is always set to 0 for ACK/NAK=1 frames. The unused field is not checked and is copied to the reply. It can thus be set to hold for example some extra id bits if needed.

16.3 Configuration options

The EDCL module has the following configuration options (VHDL generics):

TABLE 60. EDCL configuration options (VHDL generics)

Generic	Function	Allowed range	Default
mstndx	AHB master index	0 - NAHBMST-1	0
macaddrh	Most significant 3 B of the MAC address	0 - 16#FFFFFF#*	16#00005E#
macaddrl	Least significant 3 B of the MAC address.	0 - 16#FFFFFF#*	16#000000#
ipaddrh	Most significant 2 B of the IP address.	0 - 16#FFFF#*	16#C0A8#
ipaddrl	Least significant 2 B of the IP address.	0 - 16#FFFF#*	16#0035#
udpport	UDP port number.	0 - 16#FFFF#**	8000
extip	Use the value of the edcli.lsbip signal to determine the lowest four bits of the ip address. The upper 28 bits are taken from the generics.	0 - 1	0
fullduplex	Use full-duplex mode for Ethernet transmissions (also affected by mdioenabled and autoneg).	0 - 1	0
memtech	Memory technology for on-chip RAM and FIFOs.	0 - NTECH	0
bufsize	Select the size of the on-chip RAM in kbytes. Only power of 2 values are allowed.	0 - 64	4
autoneg	Let the PHY negotiate the operating mode automatically instead of forcing it to the mode selected by the generics.	0 - 1	0
speed	Enable 100 Mbit mode. Only used when autoneg is set to 0.	0 - 1	0
mdioenabled	Determines whether the fullduplex generic or the value read from the PHY status reg should be used to set the duplex mode. 0 : use generic. 1: use read value. This generic is only used when autoneg is set to 1.	0 - 1	0
phyadr	Address of the PHY chip on the MDIO interface.	0 - 31	0
phyrstcls	Number of clock cycles required for the PHY to become operational after rst.	0 - (2^{32}-1)	100000
sim	Use simulation mode. Skips waitstate for phy reset, ignores interframe gaps, initializes ARQ protocol counter. This simplifies and increases speed in simulations. This generic does not affect synthesis.	0 - 1	0

*) Not all addresses are allowed and most NICs and protocol implementations will discard frames with illegal addresses silently. Consult network literature if unsure about the addresses.

**) Some port numbers (< 5000) are used by well known services and can cause problems when connected from a host PC.

16.4 Vendor and device id

The module has vendor id 0x01 (Gaisler Research) and device id 0x005. For description of vendor and device ids see GRLIB IP Library User's Manual.

16.5 Registers

The EDCL module does not implement any registers mapped in AHB address space.

16.6 Signal description

EDCL signals are described in Table 61.

TABLE 61. EDCL link signals

Signal name	Field	Type	Function	Active
RSTN	N/A	Input	Reset	Low
CLK	N/A	Input	System clock	-
EDCLI	LSBIP[3:0]	Input	Used as the least significant bits in the IP-address if the extip generic is set to 1.	-
AHBMIM	*	Input	AHB master in signals	-
AHBMOM	*	Input	AHB master out signals	-
ETHI	**	Input	Ethernet input signals	-
ETHO	**	Output	Ethernet output signals	-

*) See AMBA-AHB part in the GRLIB manual.

**) See the ETH_AHB section in the GRLIB manual.

16.7 Library dependencies

Table 62 shows libraries that should be used when instantiating an EDCL.

TABLE 62. Library dependencies

Library	Package	Imported unit(s)	Description
GRLIB	AMBA	Signals	AMBA signal definitions
GAISLER	NET	Signals, component	Ethernet signals and EDCL component

16.8 EDCL instantiation

This examples shows how an EDCL can be instantiated.

```
library ieee;
use ieee.std_logic_1164.all;

library grlib;
use grlib.amba.all;

library gaisler;
use gaisler.net.all;

entity edcl_ex is
  port (
    clk : in std_ulogic;
    rstn : in std_ulogic;

    -- Ethernet signals
    ethi : in std_ulogic;
    etho : out std_ulogic
);
```

```
end;

architecture rtl of eth_arb_ex is
  signal edcli : edcl_in_type;

  -- AMBA signals
  signal ahbmi : ahb_mst_in_type;
  signal ahbmo : ahb_mst_out_vector := (others => ahbm_none);
begin

  -- AMBA Components are instantiated here
  ...

  -- EDCL
  e0 : edcl generic map (mstndx => 2, macaddrh => 16#00005E#, macaddrl => #000001#,
    ipaddrh => 16#C0A8#, ipaddrl => 16#0033#, udpport => 8000, memtech => inferred,
    speed => 0)
    port map (rstn, clk, edcli, ahbmi, ahbmo(2), ethi, etho);
   edcli.lsbip <= "0000";

end;
```

17 ETH_ARB - Ethernet PHY arbiter

17.1 Overview

The Ethernet PHY arbiter provides access to a single physical Ethernet medium for two Media Access Controllers (MACs). It handles both half- and full-duplex modes. The MACs are connected to separate MII interfaces which the ETH_ARB the arbitrates to a single MII interface. A block diagram is shown in figure 39.

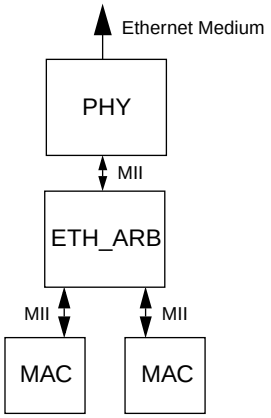

Figure 39. Ethernet PHY arbiter block diagram.

17.2 Operation

17.2.1 Arbitration method

The ETH_ARB provides two MII interfaces to which Ethernet MACs can be connected. These interfaces are identical to the one provided directly from a PHY. There are two different arbitration algorithms: one for full-duplex mode and one for half-duplex. The mode is selected with a VHDL generic and must correspond to the operating mode on the physical medium. In half-duplex mode, both MACs have equal priority and the arbiter uses the normal CSMA-CD protocol for selecting which MAC gains access to the medium.

In full-duplex mode, one MAC has priority (master) over the other (slave). When the master unit wants to transmit, the arbiter stops any ongoing transmissions from the slave MAC and inserts an interframe gap. Since the last 4 B of the interrupted frame will be interpreted as the CRC value, it will be incorrect (with a high probability) and therefore discarded at the receiving end. The master is allowed to transmit after the interframe gap and cannot be interrupted by the slave. The reason for this solution is that there should be no contention of the medium on a full-duplex network link and therefore Ethernet does not provide any signals for delaying a MAC transmission in this mode.

Since the arbiter will cause lost frames due to higher contention of the medium in half-duplex and due to the arbitration protocol in full-duplex, the higher level protocols must provide reliable transmissions (for example ARQ) with long enough time-outs. Normally, this will not be a problem and the standard protocols such as TCP can be used. Half-duplex normally works fine without any special care, but full-duplex can cause problems for the slave since its transmissions can always be interrupted. If the master transmits for a long period without pauses the slave can time-out. This has not yet been seen in normal operation but must be considered.

The MDIO interface is not arbitrated and one MAC has constant access to it. The mdiomaster generic selects which of the MACs has access.

17.3 Configuration options

ETH_ARB has the following configuration options (VHDL generics):

TABLE 63. ETH_ARB configuration options (VHDL generics)

Generic	Function	Allowed range	Default
fullduplex	Select full-duplex mode arbitration.	0 - 1	0
mdiomaster	Select which of the MACs that has access to the MDIO interface. 0 selects the MAC connected to dethi/detho and 1 selects the MAC connected to methi/metho.	0 - 1	0

17.4 Registers

ETH_ARB does not implement any user programmable registers.

17.5 Signal description

ETH_ARB signals are described in table 64.

TABLE 64. ETH_ARB signals

Signal name	Field	Type	Function	Active
RST	N/A	Input	Reset	Low
ETHI	TX_CLK	Input	Transmit clock	-
	RX_CLK	Input	Receive clock	-
	RXD[3:0]	Input	Receive data	-
	RX_DV	Input	Receive data valid	High
	RX_ER	Input	Receive data error	High
	RX_COL	Input	Collision detect	High
	RX_CRS	Input	Carrier sense	High
	MDIO_I	Input	MDIO input data	-
ETHO	RESET	Output	Ethernet Reset	High
	TXD[3:0]	Output	Transmit data	-
	TX_EN	Output	Transmit enable	High
	TX_ER	Output	Transmit error	High
	MDC	Output	MDIO clock	-
	MDIO_O	Output	MDIO output data	-
	MDIO_OE	Output	MDIO output enable	High
METHI	N/A*	Input	MII output signals from master MAC	-
METHO	N/A*	Output	MII input signals to master MAC	-
DETHI	N/A*	Input	MII output signals from slave MAC	-
DETHO	N/A*	Output	MII input signals to slave MAC	-

*) See ETHI and ETHO definitions

17.6 Library dependencies

Table 65 shows libraries that should be used when instantiating an ETH_ARB.

TABLE 65. Library dependencies

Library	Package	Imported unit(s)	Description
GAISLER	NET	Signals, component	Ethernet signals and component declaration

17.7 ETH_ARB instantiation

This examples shows how an Ethernet PHY arbiter can be instantiated.

```
library ieee;
use ieee.std_logic_1164.all;

library grlib;
use grlib.amba.all;

library gaisler;
use gaisler.net.all;

entity eth_arb_ex is
  port (
    clk : in std_ulogic;
    rstn : in std_ulogic;

    -- Ethernet signals
    ethi : in std_ulogic;
    etho : out std_ulogic
);
end;

architecture rtl of eth_arb_ex is
  signal edcli : edcl_in_type;

  -- MII signals
  signal ethi1, ethi2 : eth_in_type;
  signal etho1, etho2 : eth_out_type;

  -- AMBA signals
  signal ahbsi : ahb_slv_in_type;
  signal ahbso : ahb_slv_out_vector := (others => ahbs_none);
  signal ahbmi : ahb_mst_in_type;
  signal ahbmo : ahb_mst_out_vector := (others => ahbm_none);
begin

  -- AMBA Components are instantiated here
  ...

  -- Ethernet MAC
  e0 : eth_oc generic map(mstndx => 1, slvndx => 1, ioaddr => 16#B00#,
    irq => 12, memtech => inferred)
    port map(rstn, clk, ahbsi, ahbso(1), ahbmi, ahbmo(1), ethi1, etho1);

  -- EDCL
  e1 : edcl generic map (mstndx => 2, macaddrh => 16#00005E#, macaddrl => #000000#,
    ipaddrh => 16#C0A8#, ipaddrl => 16#0033#, udpport => 8000, memtech => inferred,
    speed => 0)
```

```
    port map (rstn, clk, edcli, ahbmi, ahbmo(2), ethi2, etho2);
   edcli.lsbip <= "0000";

  -- ETH_ARB
  ea0 : eth_arb generic map(fullduplex => 0, mdiomaster => 1)
    port map(rstn, ethi, etho, etho1, ethi1, etho2, ethi2);

end;
```

18 ETH_OC - GRLIB wrapper for Opencore 10/100 Mbit Ethernet core

18.1 Overview

ETH_OC is GRLIB wrapper for the Ethernet MAC from Opencores. It provides a bridge between AMBA AHB and the Wishbone bus, providing both AHB master and slave interfaces. The AHB slave interface is mapped in the AHB I/O space using the GRLIB plug&play functionality, and used to access the Ethernet MAC registers. The AHB master interface is used by the Ethernet MAC to read and write ethernet packets to buffer memory. The ethernet core interrupt is routed to the AHB interrupt bus, and the interrupt number is selected through the *irq* generic.

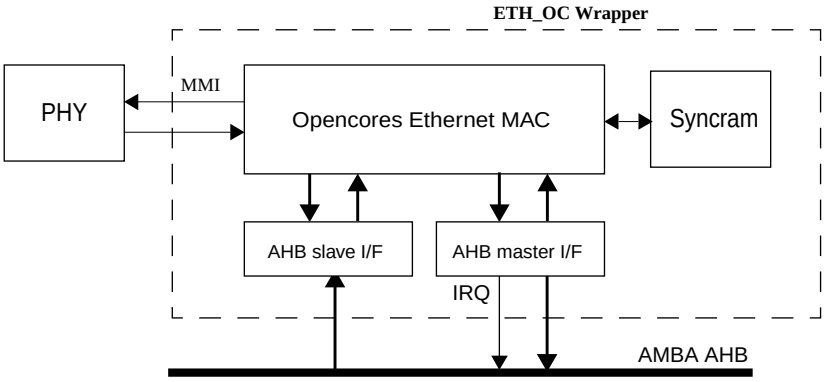

Figure 40. ETH_OC wrapper block diagram

The ETH_OC wrapper uses a parametrizable SYNCRAM core to implement the RAM used for RX/TX descriptors. The implementation of the RAM is set by the *memtech* generic, and assures portability between any supported technology.

Limitations: when using the ETH_OC, be aware of the following limitations in the Opencores ethernet MAC:

- Full-duplex is not working correctly, the core and PHY should be programmed for half-duplex
- The AHB frequency must be larger than twice the PHY frequency. This means that the AHB frequency must be higher than 5 MHz for 10 Mbit operation, and 50 MHz for 100 Mbit operation.

The data sheet for the Ethernet MAC can be found in lib/work/opencores/doc/ethernet.pdf.

18.2 Configuration options

The ETH_OC wrapper has the following configuration options (VHDL generics):

TABLE 66. VHDL Generics

Generic	Function	Allowed range	Default
mstndx	AHB master bus index	0 - NAHBMST-1	0
slvndx	AHB slave bus index	0 - NAHBSLV-1	0
ioaddr	The MSB address of the I/O area. Sets the 12 most significant bits in the 32-bit AHB address.	0 - 16#FFF#	16#FFF#
iomask	The I/O area address mask. Sets the size of the I/O area and the start address together with ioaddr.	0 - 16#FFF#	16#FF0#
irq	Interrupt number	0 - NAHBIRQ-1	0
memtech	Technology to implement on-chip RAM	0	0 - NTECH

18.3 Vendor and device id

The module has vendor id 0x01 (Gaisler Research) and device id 0x005. For description of vendor and device ids, see GRLIB IP Library User's Manual.

18.4 Signal descriptions

TABLE 67. ETH_OC signals

Signal name	Type	Function	Active
CLK	Input	AHB clock	
RST	Input	Reset	Low
AHBSI	Input	AMBA AHB slave inputs	-
AHBSO	Output	AMBA AHB slave outputs	
AHBSI	Input	AMBA AHB master inputs	
AHBSO	Output	AMBA AHB master outputs	
ETHI	Input	Ethernet MMI inputs	
ETHO	Output	Ethernet MMI outputs	

*1) see AMBA specification

18.5 Library dependencies

Table 68 shows libraries that should be used when instantiating the CAN AHB interface.

TABLE 68. Library dependencies

Library	Package	Imported unit(s)	Description
GRLIB	AMBA	Types	AMBA signal type definitions
GAISLER	NET	Types	MMI interface types
WORK	GRCOMP	Component	Component declaration

18.6 Component declaration

```
library grlib;
use grlib.amba.all;
use grlib.tech.all;
library gaisler;
use gaisler.net.all;
use work.grcomp.all;

  component eth_oc
  generic (
    mstndx    : integer := 0;
    slvndx    : integer := 0;
    ioaddr    : integer := 16#000#;
    iomask    : integer := 16#FF0#;
    irq       : integer := 0;
   memtech : integer := inferred);
  port (
    rst  : in  std_logic;
    clk  : in  std_logic;
    ahbsi : in  ahb_slv_in_type;
    ahbso : out ahb_slv_out_type;
    ahbmi : in  ahb_mst_in_type;
    ahbmo : out ahb_mst_out_type;
    ethi : in eth_in_type;
    etho : out eth_out_type);
  end component;
```

```
-- types
type eth_in_type is record
    tx_clk : std_ulogic;
    rx_clk : std_ulogic;
    rxd    : std_logic_vector(3 downto 0);
    rx_dv  : std_ulogic;
    rx_er  : std_ulogic;
    rx_col : std_ulogic;
    rx_crs : std_ulogic;
    mdio_i : std_ulogic;
  end record;

  type eth_out_type is record
    reset   : std_ulogic;
    txd     : std_logic_vector(3 downto 0);
    tx_en   : std_ulogic;
    tx_er   : std_ulogic;
    mdc     : std_ulogic;
    mdio_o  : std_ulogic;
    mdio_oe : std_ulogic;
  end record;
```

18.7 Instantiation example

```
library grlib;
use grlib.amba.all;
use grlib.tech.all;
library gaisler;
use gaisler.net.all;
use work.grcomp.all;
use gaisler.pads.all;

signal ethi : eth_in_type;
signal etho : eth_out_type;
.
.


-- Ethernet core

e1 : eth_oc generic map (mstndx => 1, slvndx => 5,
ioaddr => CFG_ETHIO, irq => 12, memtech => memtech)
      port map ( rst => rstn, clk => clkm, ahbsi => ahbsi, ahbso => ahbso(5),
        ahbmi  => ahbmi, ahbmo => ahbmo(1), ethi => ethi, etho  => etho);

-- PADS for external PHY

emdio_pad : iopad generic map (tech => padtech)
      port map (emdio, etho.mdio_o, etho.mdio_oe, ethi.mdio_i);
      etxc_pad : inpad generic map (tech => padtech)
port map (etx_clk, ethi.tx_clk);
      erxc_pad : inpad generic map (tech => padtech)
port map (erx_clk, ethi.rx_clk);
      erxd_pad : inpadv generic map (tech => padtech, width => 4)
port map (erxd, ethi.rxd);
      erxdv_pad : inpad generic map (tech => padtech)
port map (erx_dv, ethi.rx_dv);
      erxer_pad : inpad generic map (tech => padtech)
port map (erx_er, ethi.rx_er);
      erxco_pad : inpad generic map (tech => padtech)
```

```
        port map (erx_col, ethi.rx_col);
              erxcr_pad : inpad generic map (tech => padtech)
        port map (erx_crs, ethi.rx_crs);

              etxd_pad : outpadv generic map (tech => padtech, width => 4)
        port map (etxd, etho.txd);
              etxen_pad : outpad generic map (tech => padtech)
        port map ( etx_en, etho.tx_en);
              etxer_pad : outpad generic map (tech => padtech)
        port map (etx_er, etho.tx_er);
              emdc_pad : outpad generic map (tech => padtech)
        port map (emdc, etho.mdc);
```

19 FTAHBRAM - On-chip SRAM with EDAC and AHB interface

19.1 Overview

The FTAHBRAM is a version of the normal AHBRAM core with added Error Detection And Correction (EDAC). One error is corrected and two errors are detected, which is done by using a (32, 7) BCH code. Configuration is possible through an APB interface. Some of the features available are: single error counter, diagnostic reads and writes and autoscrubbing (automatic correction of single errors during reads). Figure 1 shows a block diagram of the internals of the RAM.

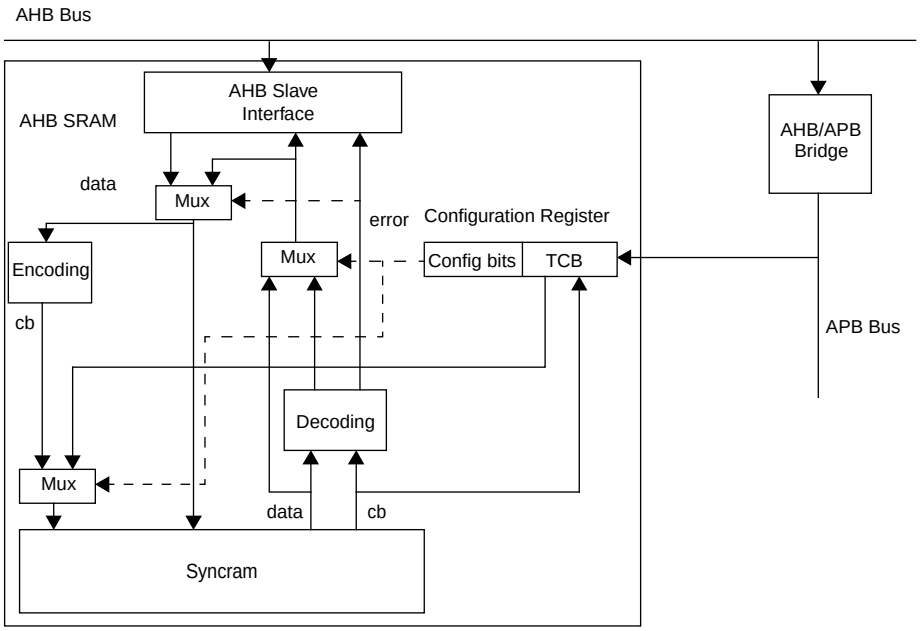

Figure 41. The FT AHB RAM block diagram

19.2 Operation

The on-chip fault tolerant RAM is accessed through an AHB slave interface and the memory address range is configurable with VHDL generics (see section 4). As for the standard AHB RAM, the memory technology and size is configurable through the tech and kbytes generics. The minimum size is 1 kb and the maximum is technology dependent but the values can only be increased in binary steps.

Run-time configuration is done by writing to a configuration register accessed through an APB interface. The address of the interface and the available options are configured with VHDL generics. The fields of the configuration register are shown in detail in section 6. The EDAC functionality can be completely removed by setting the edacen generic to zero during synthesis. The APB interface is also removed since it is redundant without EDAC.

The following can be configured during run-time: EDAC can be enabled and disabled. When it is disabled, reads and writes will behave the same as on the standard AHB RAM. Read and write diagnostics can be controlled through separate bits. The single error counter can be reset.

If EDAC is disabled (EN bit in configuration register set to 0) write data is passed directly to the RAM area and read data will appear on the AHB bus immediately after it arrives from memory. If EDAC is enabled write data is passed to an encoder which outputs a 7 - bit checksum. The checksum is stored together with the data in memory and the whole operation is performed without any added waitstates. This applies to word stores (32 - bit). If a byte or halfword store is performed, the whole word to which the byte or halfword belongs must first be read from memory (read - modify - write). A new checksum is calculated when the new data is placed in the word and both

data and checksum are stored in memory. This is done with 1 - 2 additional waitstates compared to the non EDAC case.

Reads with EDAC disabled are performed with 0 or 1 waitstates while there could also be 2 waitstates when EDAC is enabled. There is no difference between word and subword reads. Table 69 shows a summary of the number of waitstates for the different operations with and without EDAC.

TABLE 69. Summary of the number of waitstates for the different operations for the AHB RAM.

Operation	**Waitstates with EDAC Disabled**	**Waitstates with EDAC Enabled**
Read	0 - 1	0 - 2
Word write	0	0
Subword write	0	1 - 2

When EDAC is used, the data is decoded the first cycle after it arrives from the memory and appears on the bus the next cycle if no uncorrectable error is detected. The decoding is done by comparing the stored checksum with a new one which is calculated from the stored data. This decoding is also done during the read phase for a subword write. A so-called syndrome is generated from the comparison between the checksum and it determines the number of errors that occured. One error is automatically corrected and this situation is not visible on the bus. Two or more detected errors cannot be corrected so the operation is aborted and the required two cycle error response is given on the AHB bus (see the AMBA manual for more details). If no errors are detected data is passed through the decoder unaltered.

As mentioned earlier the AHB RAM provides read and write diagnostics when EDAC is enabled. When write diagnostics are enabled, the calculated checksum is not stored in memory during the write phase. Instead, the TCB field from the configuration register is used. In the same manner, if read diagnostics are enabled, the stored checksum from memory is stored in the TCB field during a read (and also during a subword write). This way, the EDAC functionality can be tested during run-time. Note that checkbits are stored in TCB during reads and subword writes even if a multiple error is detected.

An additional feature is the single error counter which can be enabled with the errcnten generic. If it is asserted a single error counter (SEC) field will be present in the configuration register. It is incremented each time a single databit error is encountered (reads or subword writes). The number of bits of this counter is set with the cntbits generic. It is accessed from bits 14 up to cntbits + 13 in the configuration register. Each bit can be reset to zero by writing a one to it. The counter saturates at the value $2^{cntbits}$ - 1. Each time a single error is detected the aramo.ce signal will be driven high for one cycle. This signal should be connected to an AHB status register which stores information and generates interrupts (see the AHB Status register documentation for more information).

Autoscrubbing is an error handling feature which is enabled with the autoscrub generic and cannot be controlled through the configuration register. If enabled, every single error encountered during a read results in the word being written back with the error corrected and new checkbits generated. It is not visible externally except for that it can generate an extra waitstate. This happens if the read is followed by an odd numbered read in a burst sequence of reads or if a subword write follows. These situations are very rare during normal operation so the total timing impact is negligible. The aramo.ce signal is normally used to generate interrupts which starts an interrupt routine that corrects errors. Since this is not necessary when autoscrubbing is enabled, aramo.ce should not be connected to an AHB status register or the interrupt should be disabled in the interrupt controller.

19.3 Configuration options

Table 70 shows the configuration options (VHDL generics) of the FT AHB SRAM.

TABLE 70. FTAHBRAM Configuration options (generics)

Generic	Function	Allowed range	Default
hindex	Selects which AHB select signal (HSEL) will be used to access the memory.	0 to NAHBMAX-1	0
haddr	ADDR field of the AHB BAR	0 to 16#FFF#	0
hmask	MASK field of the AHB BAR	0 to 16#FFF#	16#FFF#
tech	Memory technology	0 to NTECH	0
kbytes	SRAM size in kbytes	1 to targetdep.	1
pindex	Selects which APB select signal (PSEL) will be used to access the memory configuration registers	0 to NAPBMAX-1	0
paddr	The 12-bit MSB APB address	0 to 16#FFF#	0
pmask	The APB address mask	0 to 16#FFF#	16#FFF#
edacen	Enable (1)/Disable (0) on-chip EDAC	0 to 1	0
autoscrub	Automatically store back corrected data with new checkbits during a read when a single error is detected. Is ignored when edacen is deasserted.	0 to 1	0
errcnten	Enables a single error counter	0 to 1	0
cntbits	number of bits in the single error counter	1 to 8	1

19.4 Vendor and device id

The module has vendor id 0x01 (Gaisler Research) and device id 0x050. For description of vendor and device ids see GRLIB IP Library User's Manual.

19.5 Registers

Table 71 shows the FTAHBRAM registers.

TABLE 71. FT AHB RAM unit registers

Register	APB Address offset
Configuration Register	0x0

Figure 42 below shows the register bit fields. All fields except TCB are initialised at reset. The EDAC is initally disabled (EN = 0), which also applies to diagnostics (RB and WB are zero). Additionally, if the single error counter is enabled, the error counter (SEC) is set to zero and interrupts are disabled (IE = 0).

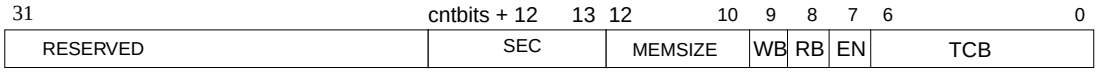

Figure 42. Configuration register bit fields

[31: cntbits + 14] - Reserved.

[cntbits + 12: 13] - Single error counter. Incremented each time a single error is corrected (includes errors on checkbits). Each bit can be set to zero by writing a one to it. This feature is only available if the errcnten generic is set.

[12 : 10] - Log2 of the current memory size

[9] - Write Bypass (WB) : When set, the TCB field is stored as check bits when a write is performed to the memory.

[8] - Read Bypass (RB) : When set during a read or subword write, the check bits loaded from memory are stored in the TCB field.

[7] - EDAC Enable : When set, the EDAC is used otherwise it is bypassed during read and write operations.

[6:0] - Test Check Bits (TCB) : Used as checkbits when the WB bit is set during writes and loaded with the check bits during a read operation when the RB bit is set.

19.6 Signal description

FT AHB RAM signals are described in table 72.

TABLE 72. FT AHB RAM signals

Signal name	Field	Type	Function	Active
RST	N/A	Input	Reset	Low
CLK	N/A	Input	Clock	-
AHBSI	*	Input	AHB slave input signals	-
AHBSO	*	Output	AHB slave output signals	-
APBI	*	Input	APB slave input signals	-
APBO	*	Output	APB slave output signals	-
ARAMO	CE	Output	Single error detected	High

* see GRLIB IP Library User's Manual

19.7 Library dependencies

Table Table 73 shows libraries that should be used when instantiating a FT AHB RAM.

TABLE 73. Library dependencies

Library	Package	Imported unit(s)	Description
GRLIB	AMBA	Signals	AMBA signal definitions
GAISLER	MISC	Component	Component declaration, Signal definition

19.8 FTAHBRAM instantiation

This examples shows how a RAM can be instantiated.

```
library ieee;
use ieee.std_logic_1164.all;

library grlib;
library gaisler;

use grlib.amba.all;
use gaisler.misc.all;

entity ftram_ex is
  port(
    rst : std_ulogic;
    clk : std_ulogic;

    .... --others signals
  );
end;

architecture rtl of ftram_ex is

--AMBA signals
signal ahbsi : ahb_slv_in_type;
signal ahbso : ahb_slv_out_type;
signal apbi  : apb_slv_in_type;
signal apbo  : apb_slv_out_vector;



--other needed signals here
signal stati   : ahbstat_in_type;
signal aramo   : ahbram_out_type;

begin

--other component instantiations here
...

-- AHB Status Register
  astat0 : ahbstat generic map(pindex => 13, paddr => 13, pirq => 11,
    nftslv => 3)
    port map(rstn, clkm, ahbmi, ahbso, stati, apbi, apbo(13));
    stati.cerror(1 to NAHBSLV-1) <= (others => '0');

--FT AHB RAM
a0 : ftahbram generic map(hindex => 1, haddr => 1, tech => inferred,
  kbytes => 64, pindex => 4, paddr => 4, edacen => 1, autoscrub => 0,
  errcnt => 1, cntbits => 4)
  port map(rst, clk, ahbsi, ahbso(1), apbi, apbo(4), aramo);
  stati.cerror(0) <= aramo.ce;

end architecture;
```

20 FTSDCTRL - 32/64-bit PC133 SDRAM Controller with EDAC

20.1 Overview

The FTSDCTRL SDRAM controller handles PC133 SDRAM compatible memory devices attached to 32 or 64 bit wide data bus. The controller acts as a slave on the AHB bus where it occupies configurable amount of address space for SDRAM access. It also contains EDAC logic (only for the 32 - bit bus) which corrects one error and detects two errors. The SDRAM controller function is programmed by writing to a pair of configuration registers mapped into AHB I/O address space.

Chip-select decoding is done for two SDRAM banks.

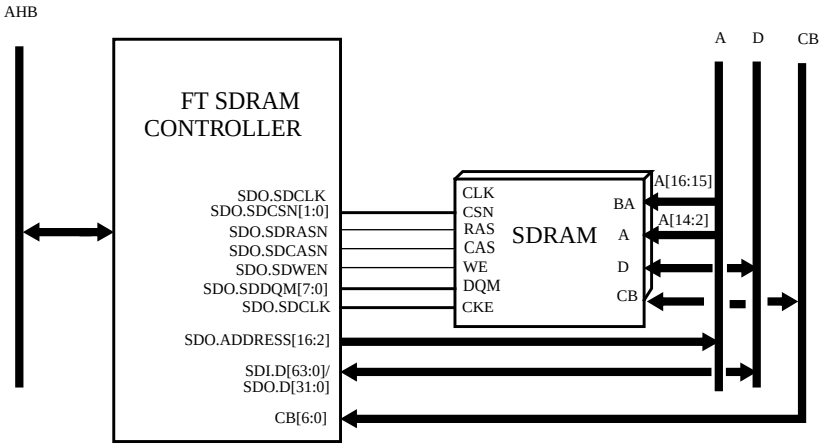

Figure 43. FT SDRAM Memory controller connected to AMBA bus and SDRAM

20.2 Operation

20.2.1 General

Synchronous dynamic RAM (SDRAM) access is supported to two banks of PC100/PC133 compatible devices. The controller supports 64M, 256M and 512M devices with 8 - 12 column-address bits, up to 13 row-address bits, and 4 banks. The size of each of the two banks can be programmed in binary steps between 4 Mbyte and 512 Mbyte. The operation of the SDRAM controller is controlled through the configuration register SDCFG (see section 20.5). A second register, ECFG, is also available for configuring the EDAC functions. SDRAM banks data bus width is configurable between 32 and 64 bits.

20.2.2 Initialisation

When the SDRAM controller is enabled, it automatically performs the SDRAM initialisation sequence of PRECHARGE, 8x AUTO-REFRESH and LOAD-MODE-REG on both banks simultaneously. The controller programs the SDRAM to use page burst on read and single location access on write.

20.2.3 Configurable SDRAM timing parameters

To provide optimum access cycles for different SDRAM devices (and at different frequencies), some SDRAM parameters can be programmed through SDRAM configuration register (SDCFG) The programmable SDRAM parameters can be seen in table below:

TABLE 74. SDRAM programmable timing parameters

Function	Parameter	range	unit
CAS latency, RAS/CAS delay	t_{CAS}, t_{RCD}	2 - 3	clocks
Precharge to activate	t_{RP}	2 - 3	clocks
Auto-refresh command period	t_{RFC}	3 - 11	clocks
Auto-refresh interval		10 - 32768	clocks

Remaining SDRAM timing parameters are according the PC100/PC133 specification.

20.2.4 Refresh

The SDRAM controller contains a refresh function that periodically issues an AUTO-REFRESH command to both SDRAM banks. The period between the commands (in clock periods) is programmed in the refresh counter reload field in the SDCFG register. Depending on SDRAM type, the required period is typically 7.8 or 15.6 µs (corresponding to 780 or 1560 clocks at 100 MHz). The generated refresh period is calculated as (reload value+1)/sysclk. The refresh function is enabled by setting bit 31 in SDCFG register.

20.2.5 SDRAM commands

The controller can issue three SDRAM commands by writing to the SDRAM command field in SDCFG: PRE-CHARGE, AUTO-REFRESH and LOAD-MODE-REG (LMR). If the LMR command is issued, the CAS delay as programmed in SDCFG will be used, remaining fields are fixed: page read burst, single location write, sequential burst. The command field will be cleared after a command has been executed. Note that when changing the value of the CAS delay, a LOAD-MODE-REGISTER command should be generated at the same time.

20.2.6 Read cycles

A read cycle is started by performing an ACTIVATE command to the desired bank and row, followed by a READ command after the programmed CAS delay. A read burst is performed if a burst access has been requested on the AHB bus. The read cycle is terminated with a PRE-CHARGE command, no banks are left open between two accesses. Note that only word bursts are supported by the SDRAM controller. The AHB bus supports bursts of different sizes such as bytes and halfwords but they cannot be used.

20.2.7 Write cycles

Write cycles are performed similarly to read cycles, with the difference that WRITE commands are issued after activation. A write burst on the AHB bus will generate a burst of write commands without idle cycles in-between. As in the read case, only word bursts are supported.

20.2.8 Address bus connection

The SDRAM address bus should be connected to SA[12:0], the bank address to SA[14:13], and the data bus to SD[31:0] or SD[63:0] if 64-bit data bus is used.

20.2.9 Data bus

Data bus width is configurable to 32 or 64 bits. 64-bit data bus allows the 64-bit SDRAM devices to be connected using the full data capacity of the devices. 64-bit SDRAM devices can be connected to 32-bit data bus if 64-bit data bus is not available but in this case only half the full data capacity will be used. SDRAM bus signals are described in chapter 20.6, for configuration options refer to chapter 20.3.

20.2.10 Clocking

The SDRAM clock typically requires special synchronisation at layout level. For Virtex targets, GR Clock Generator can be configured to produce a properly synchronised SDRAM clock. For other FPGA targets, the GR Clock Generator can produce an inverted clock.

20.2.11 EDAC

The controller also contains Error Detection And Correction (EDAC) logic, as mentioned in section 1, using a BCH(32, 7) code. It is capable of correcting one error and detecting two errors. The EDAC logic does not add any additional waitstates during normal operation. Detected errors will cause additional waitstates for correction (single errors) or error reporting (multiple errors). Single errors are automatically corrected and generally not visible externally unless explicitly checked. This checking is done by monitoring the ce signal (section 6) and single error counter (section 5). This counter holds the number of detected single errors. The ce signal is asserted one clock cycle when a single error is detected and should be connected to the AHB status register. This module stores the AHB status of the instruction causing the single error and generates interrupts (see the AHB status register documentation for more information).

The EDAC functionality can be enabled/disabled during run-time from the ECFG register and the logic can also be completely removed during synthesis with VHDL generics. The ECFG register also contains control bits and checkbit fields for diagnostic reads. These diagnostic functions are used for testing the EDAC functions on-chip and allows one to store arbitrary checkbits with each written word. Checkbits read from memory can also be controlled. See section 5 for more information on the available options in the ECFG register.

64-bit bus support is not provided when EDAC is enabled. Thus, the sd64 and edacen generics should never be set to one at the same time.

20.3 Configuration options

The FT SDRAM controller has the following configuration options (VHDL generics):

TABLE 75. FT SDRAM controller configuration options (VHDL generics)

Generic	Function	Allowed range	Default
hindex	AHB slave index	1 - NAHBSLV-1	0
haddr	ADDR filed of the AHB BAR0 defining SDRAM area. Default is 0xF0000000 - 0xFFFFFFFF.	0 - 16#FFF#	16#000#
hmask	MASK filed of the AHB BAR0 defining SDRAM area.	0 - 16#FFF#	16#F00#
ioaddr	ADDR filed of the AHB BAR1 defining I/O address space where SDCFG register is mapped.	0 - 16#FFF#	16#000#
iomask	MASK filed of the AHB BAR1 defining I/O address space.	0 - 16#FFF#	16#FFF#
wprot	Write protection.	0 - 1	0
invclk	Inverted clock is used for the SDRAM.	0 - 1	0
fast	Enable fast SDRAM address decoding.	0 - 1	0
pwron	Enable SDRAM at power-on.	0 - 1	0
sdbits	32 or 64 -bit data bus width.	32, 64	32
edacen	EDAC enable. If set to one, EDAC logic will be included in the synthesized design. An EDAC configuration register will also be available.	0 - 1	0
errcnt	Include an single error counter which is accessible from the EDAC configuration register.	0 - 1	0
cntbits	Number of bits used in the single error counter	1 - 8	1

20.4 Vendor and device id

The module has vendor id 0x01 (Gaisler Research) and device id 0x009. For a description of vendor and device ids see GRLIB IP Library User's Manual.

20.5 Registers

The memory controller is programmed through the FT SDRAM controller configuration register mapped into AHB I/O space defined by the controllers AHB BAR1. If EDAC is enabled, an EDAC configuration register will also be available.

TABLE 76. FT SDRAM controller registers

Register	AHB Address offset
SDRAM Configuration register	0x0
EDAC Configuration register	0x4

0.0.1 SDRAM configuration register (SDCFG)

SDRAM configuration register is used to control the timing of the SDRAM.

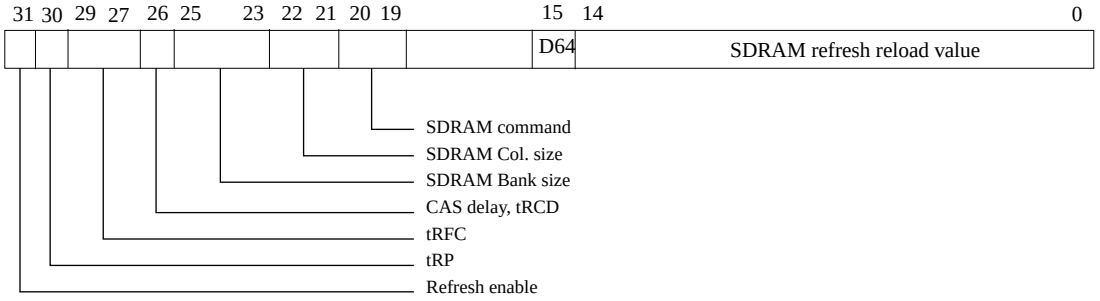

Figure 44. SDRAM configuration register

[14:0]: The period between each AUTO-REFRESH command - Calculated as follows:$t_{REFRESH}$ = ((reload value) + 1) / SYSCLK

[15]: 64-bit data bus (D64) - Reads ‘1’ if memory controller is configured for 64-bit data bus, otherwise ‘0’. Read-only.

[20:19]: SDRAM command. Writing a non-zero value will generate an SDRAM command: “01”=PRECHARGE, “10”=AUTO-REFRESH, “11”=LOAD-COMMAND-REGISTER. The field is reset after command has been executed.

[22:21]: SDRAM column size. “00”=256, “01”=512, “10”=1024, “11”=4096 when bit[25:23]= “111”, 2048 otherwise.

[25:23]: SDRAM banks size. Defines the banks size for SDRAM chip selects: “000”=4 Mbyte, “001”=8 Mbyte, “010”=16 Mbyte .... “111”=512 Mbyte.

[26]: SDRAM CAS delay. Selects 2 or 3 cycle CAS delay (0/1). When changed, a LOAD-COMMAND-REGISTER command must be issued at the same time. Also sets RAS/CAS delay (tRCD).

[29:27]: SDRAM t_{RFC} timing. t_{RFC} will be equal to 3 + field-value system clocks.

[30]: SDRAM t_{RP} timing. t_{RP} will be equal to 2 or 3 system clocks (0/1).

[31]: SDRAM refresh. If set, the SDRAM refresh will be enabled.

20.5.1 EDAC Configuration register (ECFG)

The EDAC configuration register controls the EDAC functions of the SDRAM controller during run time.

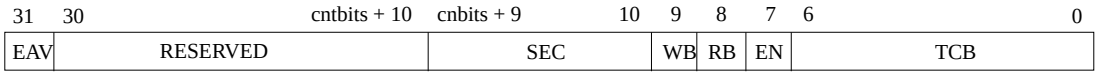

Figure 45. EDAC configuration register

[6:0] TCB : Test checkbits. These bits are written as checkbits into memory during a write operation when the WB bit in the ECFG register is set. Checkbits read from memory during a read operation are written to this field when the RB bit is set.

[7] EN : EDAC enable. Run time enable/disable of the EDAC functions. If EDAC is disabled no error detection will be done during reads and subword writes. Checkbits will still be written to memory during write operations.

[8] RB : Read bypass. Store the checkbits read from memory during a read operation into the TCB field.

[9] WB : Write bypass. Write the TCB field as checkbits into memory for all write operations.

[cntbits + 9:10] SEC : Single error counter. This field is available when the errcnt generic is set to one during synthesis. It increments each time a single error is detected. It saturates when the maximum value is reached. The maximum value is the largest number representable in the number of bits used, which in turn is determined by the cntbits generic. Each bit in the counter can be reset by writing a one to it.

[30:cntbits + 10] Reserved.

[31] EAV : EDAC available. This bit is always one if the SDRAM controller contains EDAC.

20.6 Signal description

FT SDRAM controller signals are described in table 4.

Signal name	Field	Type	Function	Active
CLK	N/A	Input	Clock	-
RST	N/A	Input	Reset	Low
AHBSI	*	Input	AHB slave input signals	-
AHBSO	*	Output	AHB slave output signals	-
SDI	WPROT	Input	Not used	-
	DATA[63:0]	Input	Data	-
	CB[7:0]	Input	Checkbits	-
SDO	SDCKE[1:0]	Output	SDRAM clock enable	High
	SDCSN[1:0]	Output	SDRAM chip select	Low
	SDWEN	Output	SDRAM write enable	Low
	RASN	Output	SDRAM row address strobe	Low
	CASN	Output	SDRAM column address strobe	Low
	DQM[7:0]	Output	SDRAM data mask	Low
	BDRIVE	Output	Drive SDRAM data bus	Low
	ADDRESS[16:2]	Output	SDRAM address	-
	DATA[31:0]	Output	SDRAM data	-
	CB[7:0]	Output	Checkbits	-
CE	N/A	Output	Correctable Error	High

* see GRLIB IP Library User's Manual

TABLE 77. FT SDRAM Controller signals.

20.7 Library dependencies

Table 5 shows libraries that the memory controller module depends on.

TABLE 78. Library dependencies

Library	Package	Imported unit(s)	Description
GRLIB	AMBA	Signals	AHB signal definitions
GAISLER	MEMCTRL	Signals, component	Memory bus signals definitions, component declaration

20.8 Memory controller instantiation

This example shows how the SDRAM controller can be instantiated. The example design contains an AMBA bus with a number of AHB components connected to it including the FT SDRAM controller. The external SDRAM bus is defined in the example designs port map and connected to the SDRAM controller. System clock and reset are generated by GR Clock Generator and Reset Generator. It is also shown how the correctable error (CE) signal is connected to the ahb status register. It is not mandatory to connect this signal. In this example, 3 units can be connected to the status register.

The SDRAM controller decodes SDRAM area: 0x60000000 - 0x6FFFFFFF. SDRAM Configuration and EDAC configuration registers are mapped into AHB I/O space on address (AHB I/O base address + 0x100).

```
library ieee;
use ieee.std_logic_1164.all;

library grlib;
use grlib.amba.all;
use grlib.tech.all;
library gaisler;
use gaisler.memctrl.all;
use gaisler.pads.all;   -- used for I/O pads
use gaisler.misc.all;

entity mctrl_ex is
  port (
    clk : in std_ulogic;
    resetn : in std_ulogic;
    pllref : in  std_ulogic;
    ... -- other signals

-- sdram memory bus
    sdcke    : out std_logic_vector ( 1 downto 0);  -- clk en
    sdcsn    : out std_logic_vector ( 1 downto 0);  -- chip sel
    sdwen    : out std_logic;                       -- write en
    sdrasn   : out std_logic;                       -- row addr stb
    sdcasn   : out std_logic;                       -- col addr stb
    sddqm    : out std_logic_vector (7 downto 0);  -- data i/o mask
    sdclk    : out std_logic;                       -- sdram clk output
    sa       : out std_logic_vector(14 downto 0); -- optional sdram address
    sd       : inout std_logic_vector(63 downto 0); -- optional sdram data
    cb       : inout std_logic_vector(7 downto 0) --EDAC checkbits
    );
end;

architecture rtl of mctrl_ex is

  -- AMBA bus (AHB and APB)
  signal apbi  : apb_slv_in_type;
  signal apbo  : apb_slv_out_vector := (others => apb_none);
  signal ahbsi : ahb_slv_in_type;
  signal ahbso : ahb_slv_out_vector := (others => ahbs_none);
  signal ahbmi : ahb_mst_in_type;
  signal ahbmo : ahb_mst_out_vector := (others => ahbm_none);

  -- signals used to connect SDRAM controller and SDRAM memory bus
  signal sdi   : sdctrl_in_type;
  signal sdo   : sdctrl_out_type;

  signal clkm, rstn : std_ulogic; -- system clock and reset
signal ce : std_logic_vector(0 to 2); --correctable error signal vector

-- signals used by clock and reset generators
```

```
  signal cgi : clkgen_in_type;
  signal cgo : clkgen_out_type;

  signal gnd : std_ulogic;

begin

  -- AMBA Components are defined here ...
  ...

  -- Clock and reset generators
  clkgen0 : clkgen generic map (clk_mul => 2, clk_div => 2, sdramen => 1,
                                tech => virtex2, sdinvclk => 0)
  port map (clk, gnd, clkm, open, open, sdclk, open, cgi, cgo);

  cgi.pllctrl <= "00"; cgi.pllrst <= resetn; cgi.pllref <= pllref;

  rst0 : rstgen
  port map (resetn, clkm, cgo.clklock, rstn);

  -- AHB Status Register
  astat0 : ahbstat generic map(pindex => 13, paddr => 13, pirq => 11,
    nftslv => 3)
    port map(rstn, clkm, ahbmi, ahbsi, ce, apbi, apbo(13));

  -- SDRAM controller
  sdc : ftsdctrl generic map (hindex => 3, haddr => 16#600#, hmask => 16#F00#,
    ioaddr => 1, fast => 0, pwron => 1, invclk => 0, edacen => 1, errcnt => 1,
    cntbits => 4)
    port map (rstn, clkm, ahbsi, ahbso(3), sdi, sdo, ce(0));

  -- input signals
  sdi.data(31 downto 0) <= sd(31 downto 0);

  -- connect SDRAM controller outputs to entity output signals
  sa <= sdo.address; sdcke <= sdo.sdcke; sdwen <= sdo.sdwen;
  sdcsn <= sdo.sdcsn; sdrasn <= sdo.rasn; sdcasn <= sdo.casn;
  sddqm <= sdo.dqm;

  -- I/O pads driving data bus signals
  sd_pad : iopadv generic map (width => 32)
      port map (sd(31 downto 0), sdo.data, sdo.bdrive, sdi.data(31 downto 0));

  -- I/O pads driving checkbit signals
  cb_pad : iopadv generic map (width => 8)
      port map (cb, sdo.cb, sdo.bdrive, sdi.cb);


end;
```

21 FTSRCTL - Fault Tolerant 32-bit PROM/SRAM/IO Controller

21.1 Overview

The Fault Tolerant 32-bit PROM/SRAM Controller uses a common 32-bit memory bus to interface PROM, SRAM and I/O devices. In addition it also provides an Error Detection And Correction Unit (EDAC), correcting one and detecting two errors. Configuration of the memory controller functions is performed through the APB bus interface. Figure 1 shows a block diagram of the controller.

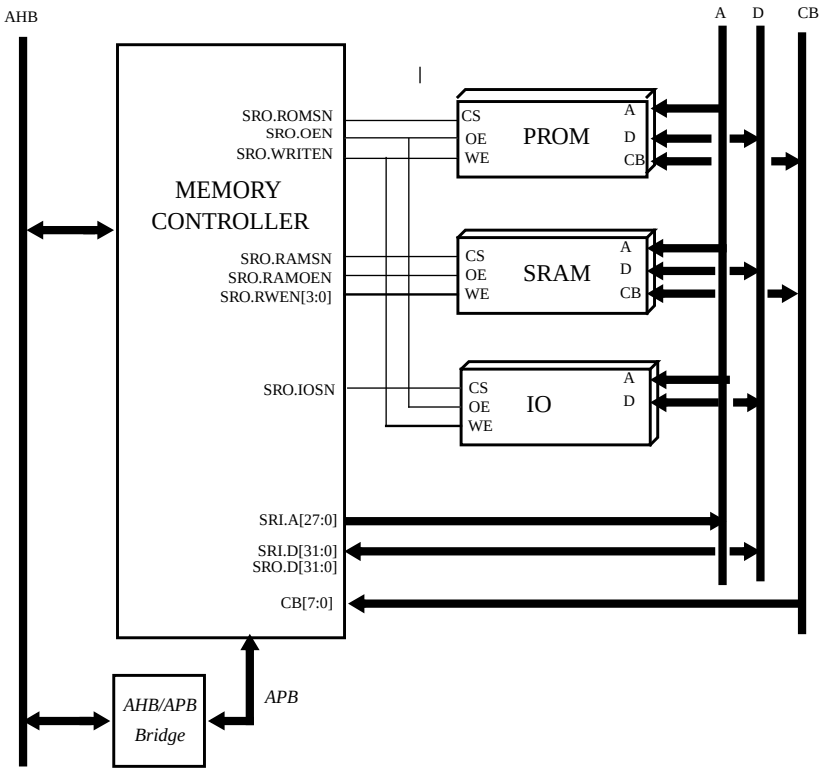

Figure 46. 32-bit FT PROM/SRAM/IO controller

21.2 Operation

The controller is configured through VHDL-generics to decode three address ranges: PROM, SRAM and I/O area. By default PROM area is mapped into address range 0x0 - 0x00FFFFFF, the SRAM area is mapped into address range 0x40000000 - 0x40FFFFFF, and the I/O area is mapped to 0x20000000 - 0x20FFFFFF.

One chip select is decoded for the I/O area, while SRAM and PROM can have up to four and two select signals respectively. The controller generates both a common write-enable signal (WRITEN) as well as four byte-write enable signals (WREN). If the SRAM uses a common write enable signal the controller can be configured to perform read-modify-write cycles for byte and half-word write accesses. Number of waitstates is separately configurable for the three address ranges.

The EDAC function is optional, and can be enabled through the *edacen* generic. The configuration of the EDAC is done through a configuration register accessed from the APB bus (see section 21.7). During nominal operation, the EDAC checksum is generated and checked automatically.

Single errors are corrected without generating any indication of this condition in the bus response. If a multiple error is detected, a two cycle error response is given on the AHB bus.

Single errors can be monitored in two ways:

- by monitoring the CE signal which is asserted for one cycle each time a single error is detected.
- by checking the single error counter which is accessed from the configuration register.

The CE signal can be connected to the AHB status register which stores information of the AHB instruction causing the error and it also generates interrupts. See the AHB status register documentation for more information. When the EDAC is enabled, one extra waitstate is generated during reads and subword writes.

21.3 PROM/SRAM/IO waveforms

Read accesses to 32-bit PROM and RAM with EDAC disabled has the same timing, see figure 47.

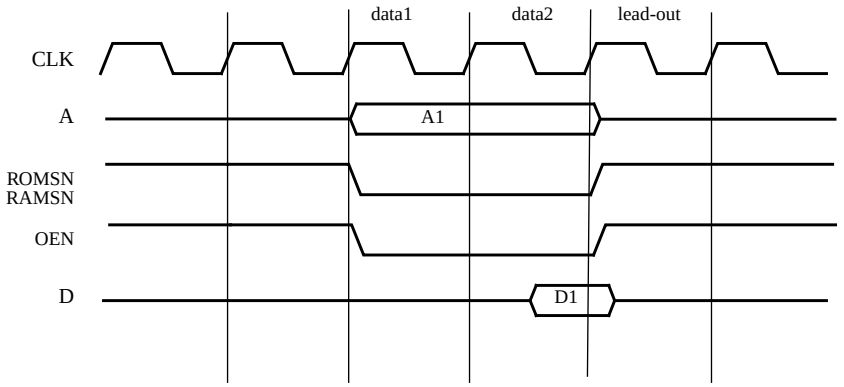

Figure 47. 32-bit PROM/SRAM read cycle

The write access for 32-bit PROM and RAM with EDAC disabled can be seen below.

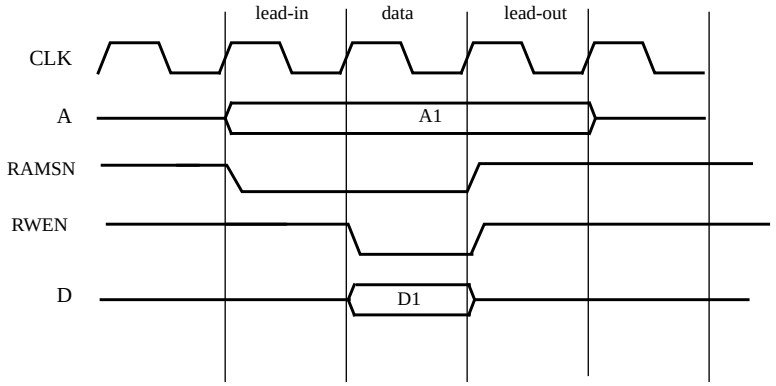

Figure 48. 32-bit PROM/SRAM write cycle

If waitstates are configured through the VHDL generics, one extra data cycle will be inserted for each waitstate in both read and write cycles. The timing for writes is not affected when EDAC is enabled while one extra waitstate is added during reads for decoding.

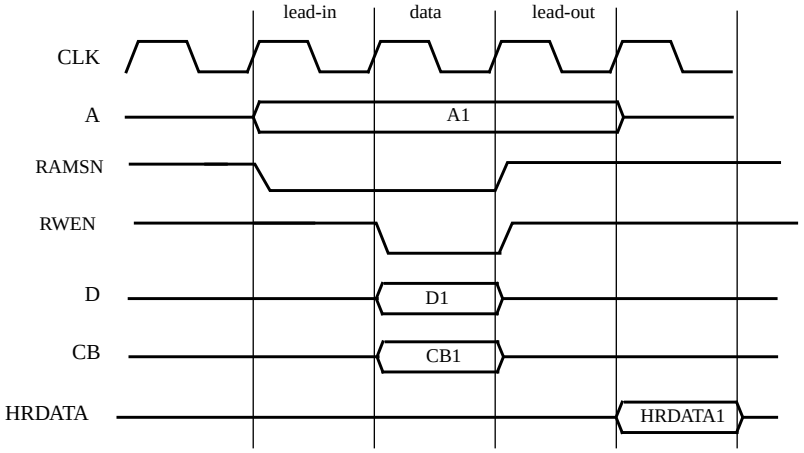

Figure 49. 32-bit PROM/SRAM read cycle with EDAC enabled.

Figure 49 shows read timing when EDAC is enabled. D and CB denotes data and checkbits on the external memory bus. The cycle following the arrival of data from memory is used for decoding and corrected data appears on the AHB-bus the next cycle (HRDATA).

21.4 Component declaration

```
component ftsrctrl is
  generic (
    hindex  : integer := 0;
    romaddr : integer := 0;
    rommask : integer := 16#ff0#;
    ramaddr : integer := 16#400#;
    rammask : integer := 16#ff0#;
    ioaddr  : integer := 16#200#;
    iomask  : integer := 16#ff0#;
    ramws   : integer := 0;
    romws   : integer := 2;
    iows    : integer := 2;
    rmw     : integer := 0;
    srbanks : integer range 1 to 4  := 1;
    banksz  : integer range 0 to 13 := 13;
    romasel : integer range 0 to 28 := 19;
    pindex  : integer := 0;
    paddr   : integer := 0;
    pmask   : integer := 16#fff#;
    edacen  : integer range 0 to 1 := 0;
    errcnt  : integer range 0 to 1 := 0;
    cntbits : integer range 1 to 8 := 1;
    wsreg   : integer := 0
    );
  port (
    rst    : in  std_ulogic;
    clk    : in  std_ulogic;
    ahbsi  : in  ahb_slv_in_type;
    ahbso  : out ahb_slv_out_type;
    apbi   : in  apb_slv_in_type;
    apbo   : out apb_slv_out_type;
    sri    : in  memory_in_type;
    sro    : out memory_out_type;
    sdo    : out sdctrl_out_type
  );
end component;
```

21.5 Configuration options

The Memory controller has the following configuration options (VHDL generics):

TABLE 79. FT 32-bit PROM/SRAM controller configuration options (VHDL generics)

Generic	Function	Allowed range	Default
hindex	AHB slave index.	1 - NAHBSLV-1	0
romaddr	ADDR field of the AHB BAR0 defining PROM address space. Default PROM area is 0x0 - 0xFFFFFF.	0 - 16#FFF#	16#000#
rommask	MASK field of the AHB BAR0 defining PROM address space.	0 - 16#FFF#	16#FF0#
ramaddr	ADDR field of the AHB BAR1 defining RAM address space. Default RAM area is 0x40000000-0x40FFFFFF.	0 - 16#FFF#	16#400#
rammask	MASK field of the AHB BAR1 defining RAM address space.	0 -16#FFF#	16#FF0#
ioaddr	ADDR field of the AHB BAR2 defining IO address space. Default RAM area is 0x20000000-0x20FFFFFF.	0 - 16#FFF#	16#200#
iomask	MASK field of the AHB BAR2 defining IO address space.	0 - 16#FFF#	16#FF0#
ramws	Number of waitstates during access to SRAM area.	0 - 15	0
romws	Number of waitstates during access to PROM area.	0 - 15	2
iows	Number of waitstates during access to IO area.	0 - 15	2
rmw	Enable read-modify-write cycles.	0 - 1	0
srbanks	Set the number of RAM banks.	1 - 4	1
banksz	Set the size of bank 1 - 4. 1 = 16 Kbyte, ... , 13 = 64Mbyte. If set to zero, the bank size is set with the rambsz field in the MCFG2 register.	0 - 13	13
romasel	address bit used for ROM chip select.	0 - 28	19
pindex	APB slave index.	1 - NAPBSLV-1	0
paddr	APB address.	1 - 16#FFF#	0
pmask	APB address mask.	1 - 16#FFF#	16#FFF#
edacen	EDAC enable. If set to one, EDAC logic is synthesized.	0 - 1	0
errcnt	If one, a single error counter is added.	0 - 1	0
cntbits	Number of bits in the single error counter.	1 - 8	1
wsreg	Enable programmable waitstate generation.	0 - 1	0

21.6 Vendor and device id

The module has vendor id 0x01 (Gaisler Research) and device id 0x051. For description of vendor and device ids see GRLIB IP Library User's Manual.

21.7 Registers

The FT PROM/SRAM controller has from one to three configurations registers available through the APB bus. The exact number depends on the values assigned to the VHDL generics. The figures below show the existing configuration options and under which conditions they are available.

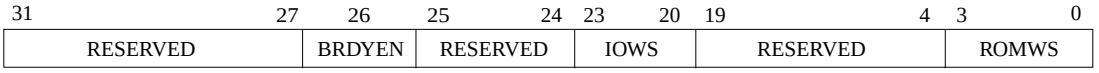

Figure 50. MCFG1 register.

[3:0] ROMWS: Sets the number of waitstates for accesses to the ROM area. Only available if the wsreg generic is set to one.

[23:20] IOWS: Sets the number of waitstates for accesses to the IO area. Only available if the wsreg generic is set to one.

[26] BRDYEN: Enables the BRDYN signal. Always available.

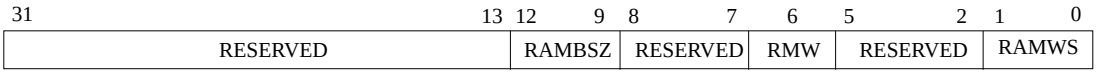

Figure 51. MCFG2 register.

[1:0] RAMWS: Sets the number of waitstates for accesses to the RAM area. Only available if the wsreg generic is set to one.

[6] RMW: If set, read-modify-write cycles are used for write accesses. Only available if the rmw generic is set to one.

[12:9] RAMBSZ: Sets the SRAM bank size. Only available if the banksz generic is set to zero. Otherwise the banksz generic sets the bank size.

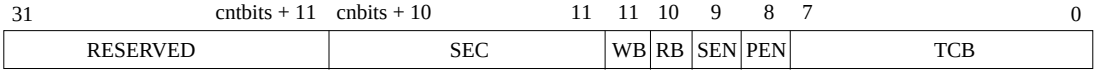

Figure 52. MCFG3 register (EDAC configuration).

[7:0] TCB: Used as checkbits in write operations when WB is one and checkbits from read operations are stored here when RB is one.

[8] PEN: PROM EDAC enable. If set, EDAC will be active for the PROM area.

[9] SEN: SRAM EDAC enable. If set, EDAC will be active for the SRAM area.

[10] RB: Read bypass. If set, checkbits read from memory in all read operations will be stored in the TCB field.

[11] WB: Write bypass. If set, the TCB field will be used as checkbits in all write operations.

[cntbits + 11 :12] SEC. Single error counter. This field increments each time a single error is detected. It saturates at the maximum value that can be stored in this field. Each bit can be reset by writing a one to it.

[31:cntbits + 11] Reserved.

All the fields in MCFG3 register are available if the edacen generic is set to one except SEC field which also requires that the errcnt generic is set to one.

21.8 Signal description

Memory controller signals are described in table 80.

TABLE 80. Memory controller signal description.

Signal name	Field	Type	Function	Active
CLK	N/A	Input	Clock	-
RST	N/A	Input	Reset	Low
MEMI	DATA[31:0]	Input	Memory data	High
	BRDYN	Input	Bus ready. Extends accesses to the IO area.	Low
	BEXCN	Input	Not used	-
	WRN[3:0]	Input	Not used	-
	BWIDTH[1:0]	Input	Not used	-
	SD[31:0]	Input	Not used	-
	CB[7:0]	Input	Checkbits	-
	PROM-DATA[31:0]	Input	Not used	-
MEMO	ADDRESS[27:0]	Output	Memory address	High
	DATA[31:0]	Output	Memory data	High
	RAMSN[4]	Output	Not used. Driven to ‘1’.	Low
	RAMSN[3:0]	Output	SRAM chip-select.	Low
	RAMOEN[4]	Output	Not used. Driven to ‘1’.	Low
	RAMOEN[3:0]	Output	SRAM output enable.	Low
	IOSN	Output	IO area chip select	Low
	ROMSN[1:0]	Output	PROM chip-select	Low
	OEN	Output	Output enable	Low
	WRITEN	Output	Write strobe	Low
	WRN[3:0]	Output	SRAM write enable	Low
	BDRIVE[3:0]	Output	Drive byte lanes on external memory bus. Controls I/O-pads connected to external memory bus.	Low
	READ	Output	Read strobe	High
	SA[14:0]	Output	Not used	-
	CB[7:0]	Output	Checkbits	-
	PSEL	Output	Not used	-
	CE	Output	Single error detected.	High
AHBSI	*	Input	AHB slave input signals	-
AHBSO	*	Output	AHB slave output signals	-
SDO	SDCASN	Output	Not used. All signals are drive to inactive state.	Low

* see GRLIB IP Library User’s Manual

21.9 Library dependencies

Table shows libraries that the memory controller module depends on.

TABLE 81. Library dependencies

Library	Package	Imported unit(s)	Description
GRLIB	AMBA	Signals	AHB signal definitions
GAISLER	MEMCTRL	Signals, component	Memory bus signals definitions, component declaration

21.10 Memory controller instantiation

This example shows how a memory controller can be instantiated. The example design contains an AMBA bus with a number of AHB components connected to it including the memory controller. The external memory bus is defined in the example designs port map and connected to the memory controller. System clock and reset are generated by GR Clock Generator and Reset Generator. The CE signal of the memory controller is also connected to the AHB status register.

Memory controller decodes default memory areas: PROM area is 0x0 - 0xFFFFFF and RAM area is 0x40000000 - 0x40FFFFF.

```
library ieee;
use ieee.std_logic_1164.all;

library grlib;
use grlib.amba.all;
use grlib.tech.all;
library gaisler;
use gaisler.memctrl.all;
use gaisler.pads.all;   -- used for I/O pads
use gaisler.misc.all;
library esa;
use esa.memoryctrl.all;

entity mctrl_ex is
  port (
    clk : in std_ulogic;
    resetn : in std_ulogic;
    pllref : in  std_ulogic;

    -- memory bus
    address  : out   std_logic_vector(27 downto 0); -- memory bus
    data     : inout std_logic_vector(31 downto 0);
    ramsn    : out   std_logic_vector(4 downto 0);
    ramoen   : out   std_logic_vector(4 downto 0);
    rwen     : inout std_logic_vector(3 downto 0);
    romsn    : out   std_logic_vector(1 downto 0);
    iosn     : out   std_logic;
    oen      : out   std_logic;
    read     : out   std_logic;
    writen   : inout std_logic;
    brdyn    : in    std_logic;
    bexcn    : in    std_logic;
-- sdram i/f
    sdcke    : out std_logic_vector ( 1 downto 0);  -- clk en
    sdcsn    : out std_logic_vector ( 1 downto 0);  -- chip sel
    sdwen    : out std_logic;                       -- write en
    sdrasn   : out std_logic;                       -- row addr stb
    sdcasn   : out std_logic;                       -- col addr stb
    sddqm    : out std_logic_vector (7 downto 0);  -- data i/o mask
```

```
    sdclk    : out std_logic;                       -- sdram clk output
    sa       : out std_logic_vector(14 downto 0); -- optional sdram address
    sd       : inout std_logic_vector(63 downto 0); -- optional sdram data
    cb       : inout std_logic_vector(7 downto 0); --checkbits
    );
end;

architecture rtl of mctrl_ex is

  -- AMBA bus (AHB and APB)
  signal apbi  : apb_slv_in_type;
  signal apbo  : apb_slv_out_vector := (others => apb_none);
  signal ahbsi : ahb_slv_in_type;
  signal ahbso : ahb_slv_out_vector := (others => ahbs_none);
  signal ahbmi : ahb_mst_in_type;
  signal ahbmo : ahb_mst_out_vector := (others => ahbm_none);

  -- signals used to connect memory controller and memory bus
  signal memi : memory_in_type;
  signal memo : memory_out_type;

  signal sdo : sdctrl_out_type;

  signal wprot : wprot_out_type;  -- dummy signal, not used
  signal clkm, rstn : std_ulogic; -- system clock and reset

-- signals used by clock and reset generators
  signal cgi : clkgen_in_type;
  signal cgo : clkgen_out_type;

  signal gnd : std_ulogic;

  signal stati : ahbstat_in_type; --correctable error vector

begin

  -- AMBA Components are defined here ...


  -- Clock and reset generators
  clkgen0 : clkgen generic map (clk_mul => 2, clk_div => 2, sdramen => 1,
                                tech => virtex2, sdinvclk => 0)
  port map (clk, gnd, clkm, open, open, sdclk, open, cgi, cgo);

  cgi.pllctrl <= "00"; cgi.pllrst <= resetn; cgi.pllref <= pllref;

  rst0 : rstgen
  port map (resetn, clkm, cgo.clklock, rstn);

  -- AHB Status Register
  astat0 : ahbstat generic map(pindex => 13, paddr => 13, pirq => 11,
    nftslv => 1)
    port map(rstn, clkm, ahbmi, ahbsi, stati, apbi, apbo(13));

  stati.cerror(0) <= memo.ce;

  -- Memory controller
  mctrl0 : ftsrctrl generic map (rmw => 1, pindex => 10, paddr => 10,
    edacen => 1, errcnt => 1, cntbits => 4)
    port map (rstn, clkm, ahbsi, ahbso(0), apbi, apbo(10), memi, memo,
         sdo);


  -- I/O pads driving data memory bus data signals
  datapads : for i in 0 to 3 generate
      data_pad : iopadv generic map (width => 8)
      port map (pad => data(31-i*8 downto 24-i*8),
                o => memi.data(31-i*8 downto 24-i*8),
                en => memo.bdrive(i),
                i => memo.data(31-i*8 downto 24-i*8));
  end generate;
```

```
  --I/O pads driving checkbit signals
  cb_pad : iopadv generic map (width => 8)
      port map (pad => cb,
                o => memi.cb,
                en => memo.bdrive(0),
                i => memo.cb;

  -- connect memory controller outputs to entity output signals
  address <= memo.address; ramsn <= memo.ramsn; romsn <= memo.romsn;
  oen <= memo.oen; rwen <= memo.wrn; ramoen <= memo.ramoen;
  writen <= memo.writen; read <= memo.read; iosn <= memo.iosn;
  sdcke <= sdo.sdcke; sdwen <= sdo.sdwen; sdcsn <= sdo.sdcsn;
  sdrasn <= sdo.rasn; sdcasn <= sdo.casn; sddqm <= sdo.dqm;

end;
```

22 GRGPIO - General Purpose I/O Port

22.1 Overview

The GRGPIO Unit implements a scalable I/O port with interrupt support. The port width can be set to 2 - 32 bits through the *nbits* generic. Each bit in the port can be individually set to input or output, and can optionally generate an interrupt. For interrupt generation, the input can be filtered for polarity and level/edge detection. The figure below shows a diagram for one I/O line.

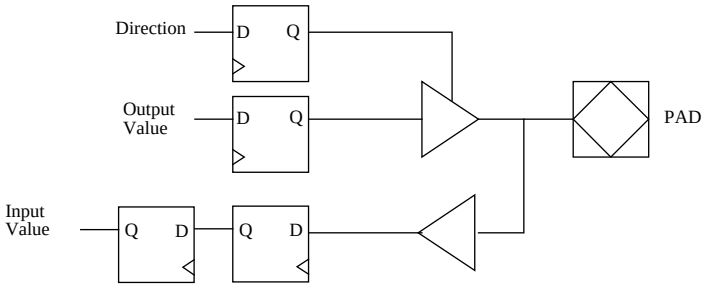

Figure 53. General Purpose I/O port diagram

22.2 Operation

The I/O ports are implemented as bi-directional buffers with programmable output enable. The input from each buffer is synchronized by two flip-flops in series to remove potential meta-stability. The synchronized values can be read-out from the I/O port data register. The output enable is controlled by the I/O port direction register. A ‘1’ in a bit position will enable the output buffer for the corresponding I/O line. The output value driven is taken from the I/O port output register.

Each I/O port can drive a separate interrupt line on the APB interrupt bus. The interrupt number is equal to the I/O line index (PIO[1] = interrupt 1, etc.). The interrupt generation is controlled by three registers: interrupt mask, polarity and edge registers. To enable an interrupt, the corresponding bit in the interrupt mask register must be set. If the edge register is ‘0’, the interrupt is treated as level sensitive. If the polarity register is ‘0’, the interrupt is active low. If the polarity register is ‘1’, the interrupt is active high. If the edge register is ‘1’, the interrupt is edge-triggered. The polarity register then selects between rising edge (‘1’) or falling edge (‘0’).

Interrupt generation and shaping is only available for those I/O lines where the corresponding bit in the *imask* generic has been set to 1.

22.3 Component declaration

```
library gaisler;
use gaisler.misc.all;

entity grgpio is
  generic (
    pindex   : integer := 0;
    paddr    : integer := 0;
    pmask    : integer := 16#fff#;
    imask    : integer := 16#0000#;
    nbits    : integer := 16-- GPIO bits
  );
  port (
    rst    : in  std_ulogic;
    clk    : in  std_ulogic;
    apbi   : in  apb_slv_in_type;
    apbo   : out apb_slv_out_type;
    gpioi  : in  gpio_in_type;
    gpioo  : out gpio_out_type
  );
end;
```

22.4 Configuration options

The I/O port has the following configuration options (VHDL generics):

TABLE 82. General Purpose I/O port options (generics)

Generic	Function	Allowed range	Default
pindex	Selects which APB select signal (PSEL) will be used to access the GPIO unit	0 to NAPBMAX-1	0
paddr	The 12-bit MSB APB address	0 to 16#FFF#	0
pmask	The APB address mask	0 to 16#FFF#	16#FFF#
nbits	Defines the number of bits in the I/O port	1 to 32	8
imask	Defines which I/O lines are provided with interrupt generation and shaping	0 - 16#FFFF#	0
oepol	Select polarity of output enable signals. 0 = active low, 1 = active high.	0 - 1	0

22.5 Vendor and device id

The module has vendor id 0x01 (Gaisler Research) and device id 0x01A. For description of vendor and device ids see GRLIB IP Library User's Manual.

22.6 Registers

Table 83 shows the timer unit registers. The number of implemented registers depend on number of implemented timers.

TABLE 83. I/O port registers

Register	APB Address offset
I/O port data register	0x00
I/O port output register	0x04
I/O port direction register	0x08
Interrupt mask register	0x0C

TABLE 83. I/O port registers

Register	APB Address offset
Interrupt polarity register	0x10
Interrupt edge register	0x14

Figures 54 to 58 shows the layout of the I/O port registers.

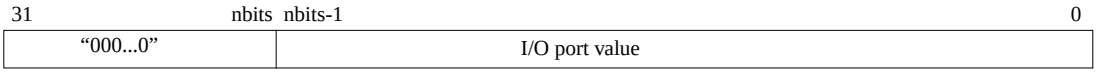

Figure 54. I/O port data register

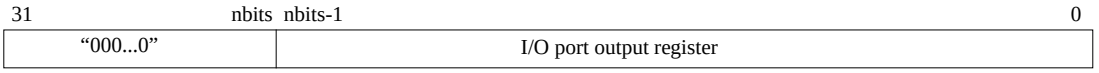

Figure 55. I/O port data register

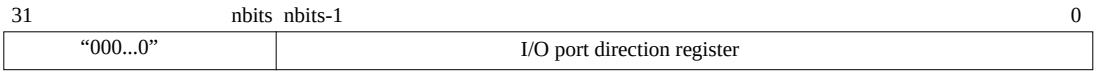

Figure 56. I/O port direction register

Figure 57. Interrupt mask register

Figure 58. Interrupt polarity register

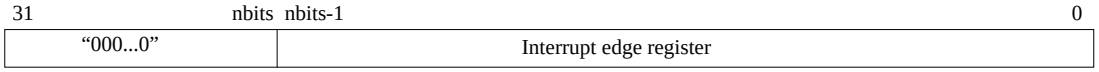

Figure 59. Interrupt edge register

22.7 Signal description

The I/O port signals are described in table 84.

TABLE 84. GP Timer Unit signals

Signal name	Field	Type	Function	Active
RST	N/A	Input	Reset	Low
CLK	N/A	Input	Clock	-
APBI	*	Input	APB slave input signals	-
APBO	*	Output	APB slave output signals	-
GPIOO	OEN[[31:0]	Output	I/O port output enable	High
	DOUT[[31:0]	Output	I/O port outputs	-
GPIOI	DIN[[31:0]	Input	I/O port inputs	-

* see GRLIB IP Library User’s Manual

22.8 Library dependencies

Table 85 shows libraries that should be used when instantiating an I/O port.

TABLE 85. Library dependencies

Library	Package	Imported unit(s)	Description
GRLIB	AMBA	Signals	AMBA signal definitions
GAISLER	MISC	Signals, component	GP Timer Unit component declaration

22.9 I/O port instantiation

This examples shows how an I/O port can be instantiated.

```
library ieee;
use ieee.std_logic_1164.all;

library grlib;
use grlib.amba.all;
library gaisler;
use gaisler.misc.all;


-- AMBA signals
  signal apbi  : apb_slv_in_type;
  signal apbo  : apb_slv_out_vector := (others => apb_none);

  -- GP Timer Unit input signals
  signal gpti : gptimer_in_type;

begin


gpio0 : if CFG_GRGPIO_EN /= 0 generate     -- GR GPIO unit
```

```
    grgpio0: grgpio
      generic map( pindex => 11, paddr => 11, imask => CFG_GRGPIO_IMASK, nbits => 8)
      port map( rstn, clkm, apbi, apbo(11), gpioi, gpioo);

      pio_pads : for i in 0 to 7 generate
        pio_pad : iopad generic map (tech => padtech)
            port map (gpio(i), gpioo.dout(i), gpioo.oen(i), gpioi.din(i));
      end generate;
end generate;
```

23 GPTIMER - General Purpose Timer Unit

23.1 Overview

The Modular Timer Unit implements one prescaler and one to seven decrementing timers. Number of timers is configurable through a VHDL-generic. The timer unit acts a slave on APB bus. The unit is capable of asserting interrupt on when timer(s) underflow. Interrupt is configurable to be common for the whole unit or separate for each timer.

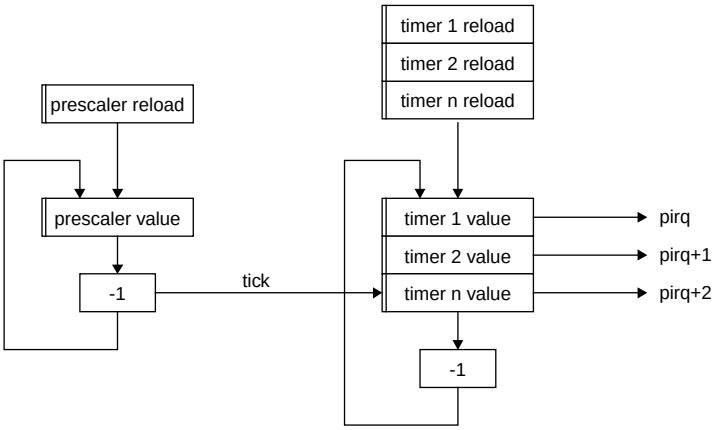

Figure 60. General Purpose Timer Unit block diagram

23.2 Operation

The prescaler is clocked by the system clock and decremented on each clock cycle. When the prescaler underflows, it is reloaded from the prescaler reload register and a timer tick is generated. Timers share the decrementer to save area. On the next timer tick next timer is decremented giving effective division rate equal to (prescaler reload register value + 1).

The operation of each timers is controlled through its control register. A timer is enabled by setting the enable bit in the control register. The timer value is then decremented on each n prescaler tick where n is the number of timers. When a timer underflows, it will automatically be reloaded with the value of the corresponding timer reload register if the restart bit in the control register is set, otherwise it will stop at -1 and reset the enable bit.

The timer unit can be configured to generate common interrupt through a VHDL-generic. The shared interrupt will be signalled when any of the timers with interrupt enable bit underflows. If configured to signal interrupt for each timer the timer unit will signal an interrupt on appropriate line when a timer underflows (if the interrupt enable bit for the current timer is set). The interrupt pending bit in the control register of the underflown timer will be set and remain set until cleared by writing ‘0’.

To minimize complexity, timers share the same decrementer. This means that the minimum allowed prescaler division factor is *ntimers*+1 (reload register = *ntimers*) where *ntimers* is the number of implemented timers.

By setting the chain bit in the control register timer *n* can be chained with preceding timer *n*-1. Decrementing timer *n* will start when timer *n*-1 underflows.

Each timer can be reloaded with the value in its reload register at any time by writing a ‘one’ to the load bit in the control register.

23.3 Configuration options

The timer unit has the following configuration options (VHDL generics):

TABLE 86. General Purpose Timer Unit Configuration options (generics)

Generic	Function	Allowed range	Default
pindex	Selects which APB select signal (PSEL) will be used to access the timer 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
nbits	Defines the number of bits in the timers	1 to 32	32
ntimers	Defines the number of timers in the unit	1 to 7	1
pirq	Defines which APB interrupt the timers will generate	0 to MAXIRQ-1	0
sepirq	If set to 1, each timer will drive an individual interrupt line, starting with interrupt *irq*. If set to 0, all timers will drive the same interrupt line (*irq*).	0 to MAXIRQ-1 (note: *ntimers* + *irq* must be less than MAXIRQ)	0
sbits	Defines the number of bits in the scaler	1 to 32	16

23.4 Vendor and device id

The module has vendor id 0x01 (Gaisler Research) and device id 0x011. For description of vendor and device ids see GRLIB IP Library User's Manual.

23.5 Registers

Table 87 shows the timer unit registers. The number of implemented registers depend on number of implemented timers.

TABLE 87. GP Timer unit registers

Register	APB Address offset
Scaler value	0x00
Scaler reload value	0x04
Configuration register	0x08
Unused	0x0C
Timer 1 counter value register	0x10
Timer 1 reload value register	0x14
Timer 1 control register	0x18
Unused	0x1C
Timer *n* counter value register	0x*n*0
Timer *n* reload value register	0x*n*4
Timer *n* control register	0x*n*8

Figures 61 to 66 shows the layout of the timer unit registers.

Figure 61. Scaler value

Figure 62. Scaler reload value

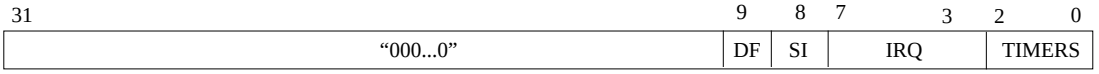

Figure 63. GP Timer Unit Configuration register

[31:10] - Reserved.

[9] - Disable timer freeze (DF). If set the timer unit can not be freezed, otherwise signal GPTI.DHALT freezes the timer unit.

[8] - Separate interrupts (SI). Reads ‘1’ if the timer unit generates separate interrupts for each timer, otherwise ‘0’. Read-only.

[7:3] - APB Interrupt: If configured to use common interrupt all timers will drive APB interrupt nr. IRQ, otherwise timer *n*will drive APB Interrupt IRQ+*n* (has to be less the MAXIRQ). Read-only.

[2:0] - Number of implemented timers. Read-only.

Figure 64. Timer counter value registers

[31:nbits] - Reserved. Always reads as ‘000...0’

[nbits-1:0] - Timer Counter value. Decremented by 1 for each *n* prescaler tick where *n* is number of implemented timers.

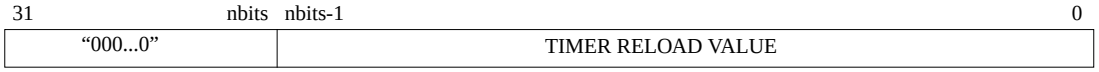

Figure 65. Timer reload value registers

[31:nbits] - Reserved. Always reads as ‘000...0’

[nbits-1:0] - Timer Reload value. This value is loaded into the timer counter value register when ‘1’ is written to load bit in the timers control register.

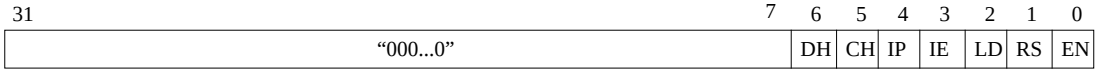

Figure 66. Timer control registers

[31:7] - Reserved. Always reads as ‘000...0’

[6] - Debug Halt (DH): Value of GPTI.DHALT signal which is used to freeze counters (e.g. when a system is in debug mode). Read-only.

[5] - Chain (CH): Chain with preceding timer. If set for timer *n*, decrementing timer *n* begins when timer (*n*-1) underflows.

[4] - Interrupt Pending (IP): Sets when an interrupt is signalled. Remains ‘1’ until cleared by writing ‘0’ to this bit.

[3] - Interrupt Enable (IE): If set the timer signals interrupt when it underflows.

[2] - Load (LD): Load value from the timer reload register to the timer counter value register.

[1] - Restart (RS): If set the value from the timer reload register is loaded to the timer counter value register and decrementing the timer is restarted.

[0] - Enable (EN): Enable the timer.

23.6 Signal description

GP Timer signals are described in table 88.

TABLE 88. GP Timer Unit signals

Signal name	Field	Type	Function	Active
RST	N/A	Input	Reset	Low
CLK	N/A	Input	Clock	-
APBI	*	Input	APB slave input signals	-
APBO	*	Output	APB slave output signals	-
GPTI	DHALT	Input	Freeze timers	High
	EXTCLK	Input	Use as alternative clock	-
GPTO	TICK[1:7]	Output	Timer ticks	High

* see GRLIB IP Library User’s Manual

23.7 Library dependencies

Table 89 shows libraries that should be used when instantiating an GP Timer.

TABLE 89. Library dependencies

Library	Package	Imported unit(s)	Description
GRLIB	AMBA	Signals	AMBA signal definitions
GAISLER	MISC	Signals, component	GP Timer Unit component declaration

23.8 GP Timer instantiation

This examples shows how an GP Timer Unit can be instantiated.

```
library ieee;
use ieee.std_logic_1164.all;

library grlib;
use grlib.amba.all;
library gaisler;
use gaisler.misc.all;

entity gptimer_ex is
  port (
    clk : in std_ulogic;
    rstn : in std_ulogic;

    ... -- other signals
    );
end;

architecture rtl of gptimer_ex is

  -- AMBA signals
  signal apbi  : apb_slv_in_type;
  signal apbo  : apb_slv_out_vector := (others => apb_none);

  -- GP Timer Unit input signals
  signal gpti : gptimer_in_type;

begin

  -- AMBA Components are instantiated here
  ...


  -- General Purpose Timer Unit
  timer0 : gptimer
  generic map (pindex => 3, paddr => 3, pirq => 8, sepirq => 1)
  port map (rstn, clk, apbi, apbo(3), gpti, open);

  gpti.dhalt <= '0'; gpti.extclk <= '0'; -- unused inputs

end;
```

24 GRFPU - High-performance IEEE-754 Floating-point unit

24.1 Overview

GRFPU is a high-performance FPU implementing floating-point operations as defined in IEEE Standard for Binary Floating-Point Arithmetic (IEEE-754) and SPARC V8 standard (IEEE-1754). Supported formats are single and double precision floating-point numbers. The advanced design combines two execution units, a fully pipelined unit for execution of the most common FP operations and a non-blocking unit for execution of divide and square-root operations. The logical view of the GRFPU is shown in figure 67.

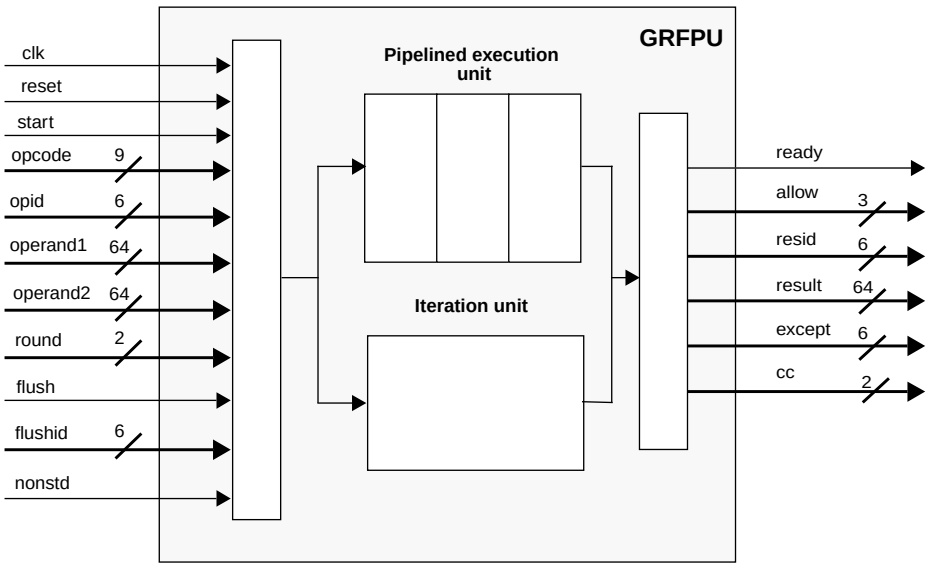

Figure 67. 1: GRFPU Logical View

This document describes GRFPU from functional point of view. Chapter “Functional description” gives details about GRFPU implementation of the IEEE-754 standard including FP formats, operations, opcodes, operation timing, rounding and exceptions. “Signals and timing” describes the GRFPU interface and its signals. “GRFPU Control Unit” describes the software aspects of the GRFPU integration into a LEON processor through the GRFPU Control Unit - GRFPC. For implementation details refer to the white paper, “GRFPU - High Performance IEEE-754 Floating-Point Unit” (available at *www.gaisler.com*).

24.2 Functional Description

24.2.1 Floating-point number formats

GRFPU handles floating-point numbers in single or double precision format as defined in IEEE-754 standard with exception for denormalized numbers. See “Denormalized numbers” on page 126 for more information on denormalized numbers.

24.2.2 FP operations

GRFPU supports four types of floating-point operations: arithmetic, compare, convert and move. The operations implement all FP instructions specified by SPARC V8 instruction set, and most of the operations defined in IEEE-754. All operations are summarized in “: GRFPU operations” on

page 125, with their opcodes, operands, results and exception codes. Throughputs and latencies and are shown in Table 90.

Operation	OpCode[8:0]	Op1	Op2	Result	Exceptions	Description
Arithmetic operations						
FADDS FADDD	001000001 001000010	SP DP	SP DP	SP DP	UNF, NV, OF, UF, NX	Addition
FSUBS FSUBD	001000101 001000110	SP DP	SP DP	SP DP	UNF, NV, OF, UF, NX	Subtraction
FMULS FMULD FSMULD	001001001 001001010 001101001	SP DP SP	SP DP SP	SP DP DP	UNF, NV, OF, UF, NX UNF, NV, OF, UF, NX UNF, NV, OF, UF	Multiplication, FSMULD gives exact double-precision product of two single-precision operands.
FDIVS FDIVD	001001101 001001110	SP DP	SP DP	SP DP	UNF, NV, OF, UF, NX	Division
FSQRTS FSQRTD	000101001 000101010	- -	SP DP	SP DP	UNF, NV, NX	Square-root
Conversion operations						
FITOS FITOD	011000100 011001000	-	INT	SP DP	NX -	Integer to floating-point conversion
FSTOI FDTOI	011010001 011010010	-	SP DP	INT	UNF, NV, NX	Floating-point to integer conversion. The result is rounded in round-to-zero mode.
FSTOI_RND FDTOI_RND	111010001 111010010	-	SP DP	INT	UNF, NV, NX	Floating-point to integer conversion. Rounding according to RND input.
FSTOD FDTOS	011001001 011000110	-	SP DP	DP SP	UNF, NV UNF, NV, OF, UF, NX	Conversion between floating-point formats
Comparison operations						
FCMPS FCMPD	001010001 001010010	SP DP	SP DP	CC	NV	Floating-point compare. Invalid exception is generated if either operand is a signaling NaN.
FCMPES FCMPED	001010101 001010110	SP DP	SP DP	CC	NV	Floating point compare. Invalid exception is generated if either operand is a NaN (quiet or signaling).
Negate, Absolute value and Move						
FABSS	000001001	-	SP	SP	-	Absolute value.
FNEGS	000000101	-	SP	SP	-	Negate.
FMOVS	000000001		SP	SP	-	Move. Copies operand to result output.

SP - single precision floating-point number

DP - double precision floating-point number

INT - 32 bit integer

CC - condition codes, see “GRFPU input and output signals are described in “: Signal description” on page 128. All signals are active high except for RST which is active low.” on page 128

UNF, NV, OF, UF, NX - floating-point exceptions, see “Exceptions” on page 126

TABLE 90. : GRFPU operations

Arithmetic operations include addition, subtraction, multiplication, division and square-root. Each arithmetic operation can be performed in single or double precision formats. Arithmetic operations have one clock cycle throughput and latency of three clock cycles, except for divide and square-root operations, which have a throughput of 14 - 23 clock cycles and latency of 15 - 25 clock cycles (see Table 91). Add, sub and multiply can be started on every clock cycle providing very high throughput for these common operations. Divide and square-root operations have lower throughput and higher latency due to complexity of the algorithms, but are executed parallelly with all other FP operations in a non-blocking iteration unit. Out-of-order execution of operations with different latencies is easily handled through the GRFPU interface by assigning an id to every operation which appears with the result on the output once the operation is completed (see section 3.2).

Operation	Throughput	Latency
FADDS, FADDD, FSUBS, FSUBD, FMULS, FMULD, FSMULD	1	3
FITOS, FITOD, FSTOI, FSTOI_RND, FDTOI, FDTOI_RND, FSTOD, FDTOS	1	3
FCMPS, FCMPD, FCMPES, FCMPED	1	3
FDIVS	15	15
FDIVD	16.5 (15/18)*	16.5 (15/18)*
FSQRTS	23	23
FSQRTD	24.5 (23/26)*	24.5 (23/26)*

* Throughput and latency are data dependant with two possible cases with equal statistical possibility.

TABLE 91. : Throughput and latency

Conversion operations execute in a pipelined execution unit and have throughput of one clock cycle and latency of three clock cycles. Conversion operations provide conversion between different floating-point numbers and between floating-point numbers and integers.

Comparison functions offering two different types of quiet Not-a-numbers (QNaNs) handling are provided. Move, negate and absolute value are also provided. These operations do not ever generate unfinished exception (unfinished exception is never signaled since compare, negate, absolute value and move handle denormalized numbers).

24.2.3 Exceptions

GRFPU detects all exceptions defined by the IEEE-754 standard. This includes detection of Invalid Operation (NV), Overflow (OF), Underflow (UF), Division-by-Zero (DZ) and Inexact (NX) exception conditions. Generation of special results such as NaNs and infinity is also implemented. Overflow (OF) and underflow (UF) are detected before rounding. When an underflow is signaled the result is rounded (flushed) to zero (this variation is allowed by the IEEE-754 standard and is implementation-dependent). A special Unfinished exception (UNF) is signaled when one of the operands is a denormalized number which are not handled by the arithmetic and conversion operations.

24.2.4 Rounding

All four rounding modes defined in the IEEE-754 standard are supported: round-to-nearest, round-to-+inf, round-to--inf and round-to-zero.

24.2.5 Denormalized numbers

Denormalized numbers are not handled by the GRFPU arithmetic and conversion operations. A system (microprocessor) with the GRFPU could emulate rare cases of operations on denormals in software using non-FPU operations. A special Unfinished exception (UNF) is used to signal an arithmetic or conversion operation on the denormalized numbers. Compare, move, negate and absolute value operations can handle denormalized numbers and don’t raise unfinished exception. GRFPU does not generate any denormalized numbers during arithmetic and conversion operations on normalized numbers since the result of an underflowed operation is flushed (rounded) to zero (see “Exceptions” on page 126).

24.2.6 Non-standard Mode

GRFPU can operate in a non-standard mode where all denormalized operands to arithmetic and conversion operations are treated as (correctly signed) zeroes. Calculations are performed on zero operands instead of the denormalized numbers obeying all rules of the floating-point arithmetics including rounding of the results and detecting exceptions.

24.2.7 NaNs

GRFPU supports handling of Not-a-Numbers (NaNs) as defined in the IEEE-754 standard. Operations on signaling NaNs (SNaNs) and invalid operations (e.g. inf/inf) generate Invalid exception and deliver QNaN_GEN as result. Operations on Quiet NaNs (QNaNs), except for FCMPES and FCMPED, do not raise any exceptions and propagate QNaNs through the FP operations by delivering NaN-results according to “: Operations on NaNs” on page 127. QNaN_GEN is 0x7fffe00000000000 for double precision results and 0x7fff0000 for single precision results.

	Operand 2			
Operand 1		**FP**	**QNaN2**	**SNaN2**
	none	FP	QNaN2	QNaN_GEN
	FP	FP	QNaN2	QNaN_GEN
	QNaN1	QNaN1	QNaN2	QNaN_GEN
	SNaN1	QNaN_GEN	QNaN_GEN	QNaN_GEN

TABLE 92. : Operations on NaNs

24.3 Signals and Timing

24.3.1 Signal Description

GRFPU input and output signals are described in “: Signal description” on page 128. All signals are active high except for RST which is active low.

Signal	I/O	Description
CLK	I	Clock
RST	I	Reset
START	I	Start an FP operation on the next rising clock edge
NONSTD	I	Nonstandard mode. Denormalized operands are converted to zero.
OPCODE[8:0]	I	FP operation. For codes see “: GRFPU operations” on page 125.
OPID[5:0]	I	FP operation id. Every operation is associated with an id which will appear on the RESID output when the FP operation is completed. This value shall be incremented by 1 (with wrap-around) for every started FP operation.
OPERAND1[63:0] OPERAND2[63:0]	I	FP operation operands are provided on these one or both of these inputs. All 64 bits are used for IEEE-754 double precision floating-point numbers, bits [63:32] are used for IEEE-754 single precision floating-point numbers and 32-bit integers.
ROUND[1:0]	I	Rounding mode. 00 - rounding-to-nearest, 01 - round-to-zero, 10 - round-to-+inf, 11 - round-to--inf.
FLUSH	I	Flush FP operation with FLUSHID.
FLUSHID[5:0]	I	Id of the FP operation to be flushed.
READY	O	The result of a FP operation will be available at the end of the next clock cycle.
ALLOW[2:0]	O	Indicates allowed FP operations during the next clock cycle. ALLOW[0] - FDIVS, FDIVD, FSQRTS and FSQRTD allowed ALLOW[1] - FMULS, FMULD, FSMULD allowed ALLOW[2] - all other FP operations allowed
RESID[5:0]	O	Id of the FP operation whose result appears at the end of the next clock cycle.
RESULT[63:0]	O	Result of an FP operation. If the result is double precision floating-point number all 64 bits are used, otherwise single precision or integer result appears on RESULT[63:32].
EXCEPT[5:0]	O	Floating-point exceptions generated by an FP operation. EXC[5] - Unfinished FP operation. Generated by an arithmetic or conversion operation with denormalized input(s). EXC[4] - Invalid exception. EXC[3] - Overflow. EXC[2] - Underflow. EXC[1] - Division by zero. EXC[0] - Inexact.
CC[1:0]	O	Result (condition code) of an FP compare operation. 00 - equal, 01 - operand1 < operand2 10 - operand1 > operand2 11 - unordered

TABLE 93. : Signal description

24.3.2 Signal Timing

An FP operation is started by providing the operands, opcode, rounding mode and id before rising edge. The operands need to be provided a small set-up time before a rising edge while all other signals are latched on rising edge.

The FPU is fully pipelined and a new operation can be started every clock cycle. The only exceptions are divide and square-root operations which require 15 to 26 clock cycles to complete, and which are not pipelined. Division and square-root are implemented through iterative series expansion algorithm. Since the algorithms basic step is multiplication the floating-point multiplier is shared between multiplication, division and square-root. Division and square-root do not occupy multiplier during the whole operation and allow multiplication to be interleaved and executed parallelly with division or square-root.

One clock cycle before an operation is completed, the output signal RDY is asserted to indicate that the result of an FPU operation will appear on the output signals at the end of the next cycle. The id of the operation to be completed and allowed operations are reported on signals RESID and ALLOW. During the next clock cycle the result appears on RES, EXCEPT and CC outputs.

“2: Signal timing” on page 129 2 shows signal timing during four arithmetic operations on GRFPU.

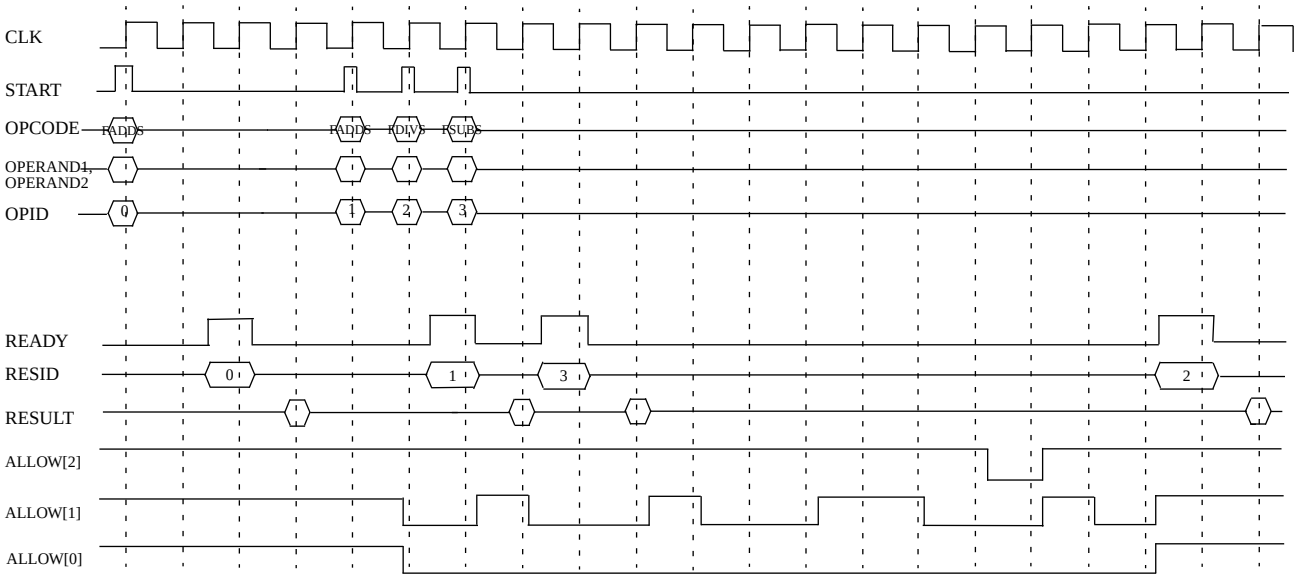

Fig. 2: Signal timing

25 GRFPC - GRFPU Control Unit

GRFPU Control Unit (GRFPC) is used to attach the GRFPU to the LEON integer unit (IU). GRFPC performs scheduling, decoding and dispatching of the FP operations to the GRFPU as well as managing the floating-point register file, the floating-point state register (FSR) and the floating-point deferred-trap queue (FQ). Floating-point operations are executed in parallel with other integer instructions, the LEON integer pipeline is only stalled in case of operand or resource conflicts.

In the FT-version, all registers are protected with TMR and the floating-point register file is protected using (39,7) BCH coding. Correctable errors in the register file are detected and corrected using the instruction restart function in the IU.

25.1 Floating-Point register file

GRFPU floating-point register file contains 32 32-bit floating-point registers (%f0-%f31). The register file is accessed by floating-point load and store instructions (LDF, LDDF, STD, STDF) and floating-point operate instructions (FPop).

25.2 Floating-Point State Register (FSR)

GRFPC manages the floating-point state register (FSR) containing FPU mode and status information. All fields of the FSR register as defined in SPARC V8 specification are implemented and managed by the GRFPU conforming to SPARC V8 specification and IEEE-754 standard. Implementation-specific parts of the FSR managing are the NS (non-standard) bit and *ftt* field.

If the NS (non-standard) bit of the FSR register is set, all floating-point operation will be performed in non-standard mode as described in “Non-standard Mode” on page 127. When NS bit is cleared all operations are performed in standard IEEE-compliant mode.

Following floating-point trap types never occur and are therefore never set in the ftt field:
- unimplemented_FPop: all FPop operations are implemented
- hardware_error: non-resumable hardware error
- invalid_fp_register: no check that double-precision register is 0 mod 2 is performed

GRFPU implements the *qne* bit of the FSR register which reads 0 if the floating-point deferred-queue (FQ) is empty and 1 otherwise.

The FSR is accessed using LDFSR and STFSR instructions.

25.3 Floating-Point Exceptions and Floating-Point Deferred-Queue

GRFPU implements SPARC deferred trap model for floating-point exceptions (fp_exception). A floating-point exception is caused by a floating-point instruction performing an operation resulting in one of following conditions:

- an operation raises IEEE floating-point exception (ftt = IEEE_754_exception) e.g. executing invalid operation such as 0/0 while the NVM bit of the TEM field id set (invalid exception enabled).
- an operation on denormalized floating-point numbers (in standard IEEE-mode) raises unfinished_FPop floating-point exception
- sequence error: abnormal error condition in the FPU due to the erroneous use of the floating-point instructions in the supervisor software.

The trap is deferred to one of the floating-point instruction (FPop, FP load/store, FP branch) following the trap-inducing instruction (note that this may not be next floating-point instruction in the program order due to exception-detecting mechanism and out-of-order instruction execution in the GRFPC). When the trap is taken the floating-point deferred-queue (FQ) contains trap-inducing instruction and up to two FPop instructions that where dispatched in the GRFPC but did not complete.

After the trap is taken the *qne* bit of the FSR is set and remains set until the FQ is emptied. STDFQ instruction reads a double-word from the floating-point deferred queue, the first word is the address of the instruction and the second word is the instruction code. All instructions in the FQ are FPop type instructions. First access to the FQ gives double-word with trap-inducing instruction, following double-words contain pending floating-point instructions. Supervisor software should emulate FPops from the FQ in the same order as they were read from the FQ.

Note that instructions in the FQ may not appear in the same order as the program order since GRFPU executes floating-point instructions out-of-order. A floating-point trap is never deferred past an instruction specifying source registers, destination registers or condition codes that could be modified by the trap-inducing instruction. Execution or emulation of instructions in the FQ by the supervisor software gives therefore the same FPU state as if the instructions where executed in the program order.

26 GRPCI - PCI Target / Master Unit

26.1 Overview

The PCI Target/Master Unit is a bridge between PCI bus and AMB AHB bus. The unit is connected to the PCI bus through two interfaces PCI Target and PCI Master. PCI Master interface is optional and can be disabled in the VHDL model. Two interfaces connect the core to the AHB bus: AHB Slave and AHB Master Interface. PCI Configuration / Status register is attached to AMBA APB bus.

The PCI and AMBA interfaces belong to two different clock domains. Synchronization is performed inside the core through FIFOs with configurable depth.

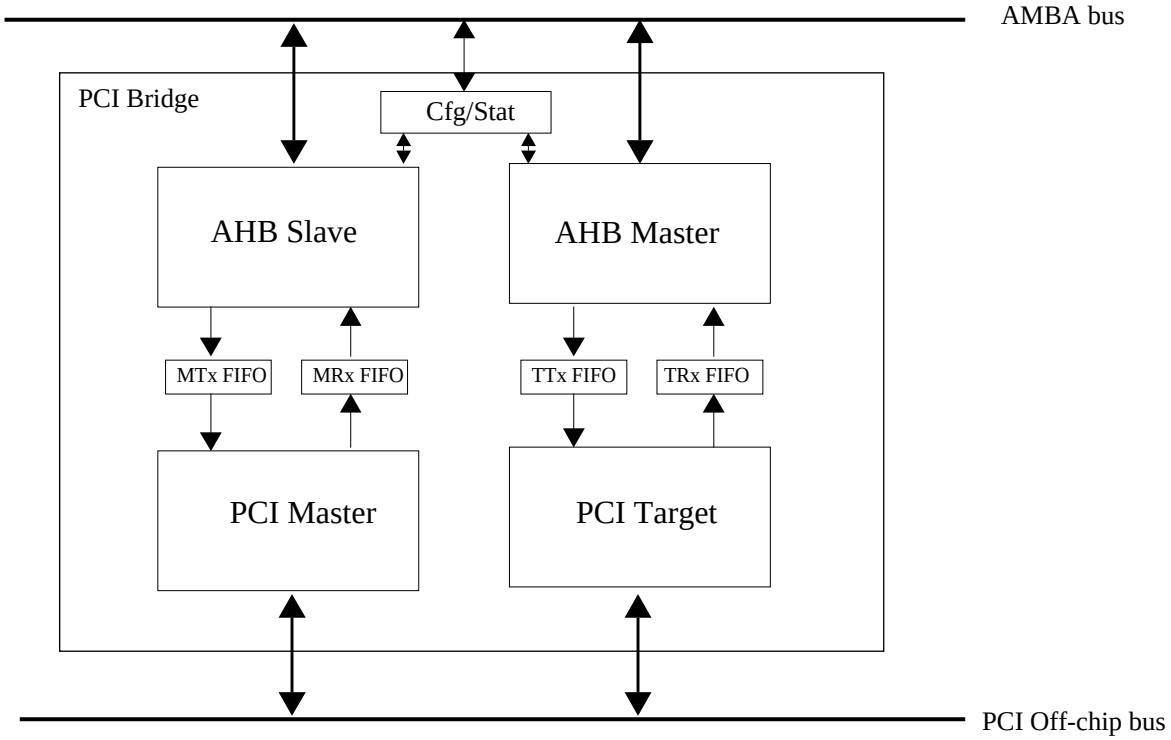

Figure 68. PCI Master/Target Unit

26.2 Operation

A connection between the PCI bus and the AMBA bus is provided by the units PCI target interface and AHB master. The PCI target is capable of handling configuration and single or burst memory cycles on the PCI bus. Configuration cycles are used to access Targets Configuration Space Header while the memory cycles are translated to AHB accesses. The PCI target interface can be programmed to occupy two areas in the PCI address space. Mapping to AHB address space is defined by a pair of map registers accessible from PCI and AHB address space.

The Master interface occupies 256 MB in the AHB address space. An access to this area is translated to PCI configuration, memory or I/O cycles. Generation of PCI cycles and mapping to the PCI address space is controlled through the AMBA Configuration / Space register.

Both target and master interface are capable of burst transactions. Data is buffered internally in FIFOs with configurable size.

26.3 Configuration options

The PCI Target / Master unit has the following configuration options (VHDL generics):

TABLE 94. PCI Target/Master options (generics)

Generic	Function	Allowed range	Default
memtech	The memory technology used for the FIFO instantiation	-	0
mstndx	The AMBA master index for the target backend AHB master interface.	0 - NAHBMST-1	0
dmamst	The AMBA master index for the DMA controller, if present. This value is used by the PCI core to detect when the DMA controller accesses the AHB slave interface.	0 - NAHBMS	NAHBMST (= disabled)
readpref	Prefetch data for the 'memory read' command. If set, the target prefetches a cache line, otherwise the target will give a single word response.	0 -1	0
abits	Least significant implemented bit of BAR0 and PAGE0 registers. Defines PCI address space size.	16 - 28	21
dmaabits	Least significant implemented bit of BAR1 and PAGE1 registers. Defines PCI address space size.	16 - 28	26
fifodepth	Size of each FIFO is 2^fifodepth 32-bit words.	>= 3	5
device_id	PCI device ID number	0 -16#FFE#	0
vendor_id	PCI vendor ID number	0 - 16#FFF#	0
master	Disables/enables PCI master interface.	0 - 1	0
slvndx	The AHB index of the master backend AHB slave interface.	0 - NAHBSLV-1	0
apbndx	The AMBA APB index for the configuration/status APB interface	0 - NAPBMAX-1	0
apbaddr	APB interface base address	0 - 16#FFF#	0
apbmask	APB interface address mask	0 - 16#FFF#	16#FFF#
memaddr	AHB slave base address	0 - 16#FFF#	16#F00#
memmask	AHB address mask	0 - 16#FFF#	16#F00#
ioaddr	AHB I/O area base address	0 - 16#FFF#	0
nsync	The number of clock registers used by each signal that crosses the clock regions.	1 - 2	1
oepol	Polarity of pad output enable signals. 0=active low, 1=active high	0 - 1	0

26.4 Vendor and device id

The module has vendor id 0x01 (Gaisler Research) and device id 0x014. For description of vendor and device ids see GRLIB IP Library User's Manual.

26.5 PCI Target Interface

PCI target interface occupies two memory areas in the PCI address space. Memory mapping is determined by BAR0 and BAR1 registers of the units Configuration Space Header. The size of the PCI memory areas is determined by number of bits actually implemented by BAR registers (configurable through *abits* and *dmaabits* VHDL-generics).

PCI Target interface handles following PCI commands:

- Configuration Read/Write: Single access to Configuration Space Header. No AHB access is performed.
- Memory Read: If prefetching is enabled, the units AHB master interface fetches a cache line, otherwise a single AHB access is performed.
- Memory Read Line: The unit prefetches data according to the value of the cache line size register.
- Memory Read Multiple: The unit performs maximum prefetching. This can cause long response time, depending of the user defined FIFO depth.
- Memory Write, Memory Write and Invalidate: Handled similarly.

The target interface supports incremental bursts for PCI memory cycles.

The target interface can finish a PCI transaction with one of the following abnormal responses:

- Retry: This response indicates that the master should perform the same request later, while the target is temporarily busy. This response is always given at least one time for read accesses, but can also occur for write accesses.
- Disconnect with data: Indicates that the target will accept one more data transaction, but no more. This occurs if the master tries to read more data than the target has prefetched.
- Disconnect without data: Indicates that the target is unable to accept more data. This occurs if the master tries to write more data than the target can buffer.
- Target-Abort: Indicates that the current access caused an internal error, and the target never will be able to finish it.

Targets AHB master interface is capable of burst transactions. Burst transactions are performed on the AHB when supported by the destination unit (AHB slave), otherwise multiple single access are performed. A PCI burst crossing 1 kB address boundary will be performed as multiple AHB bursts by the AHB master interface. The AHB master interface will insert an idle-cycle before requesting a new AHB burst to allow re-arbitration on the AHB. AHB transactions with ‘retry’ response are repeated by the AHB master until ‘okey’ or ‘error’ response is received. The ‘error’ response on AHB bus will result in ‘target abort’ response for the PCI memory read cycle. In case of PCI memory write cycle, AHB access will not be finished with error response since write data is posted to the destination unit. Instead the WE bit will be set in the units AMBA Configuration/Status register.

26.5.1 PCI Target - Configuration Space Header Registers

Following registers are implemented in PCI Configuration Space Header:

TABLE 95. Configuration Space Header registers

Address	Register
0x00	Device ID & Vendor ID
0x04	Status & Command
0x08	Class Code & Revision ID
0x0C	BIST & Header Type & Latency Timer & Cache Line Size
0x10	BAR0
0x14	BAR1

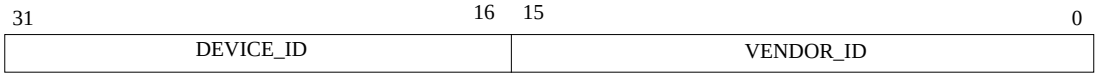

Figure 69. Device ID & Vendor ID register

[31:16]: Device ID (read-only). Returns value of *device_id* VHDL-generic.
[15:0]: Vendor ID (read-only). Returns value of *vendor_id* VHDL-generic.

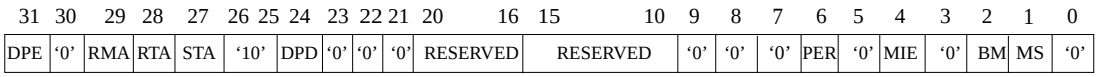

Figure 70. Status & Command register

[31:16]: Status Register - Writing one to a bit 31 - 16 clears the bit. Writes can not set a bit.
[31]: Detected parity Error (DPE).
[30]: Signalled System Error (SSE) - Not implemented. Always reads 0.
[29]: Received Master Abort (RMA) - Set by the PCI Master interface when its transaction is terminated with Master-Abort.
[28]: Received Target Abort (RTA) - Set by the PCI Master interface when its transaction is terminated with Target-Abort.
[27]: Signalled Target Abort (STA) - Set by the PCI Target Interface when the target terminates transaction with Target-Abort.
[26:25]: DEVSEL timing (DST) -Always reads “10” - medium DEVSEL timing.
[24]: Data Parity Error Detected (DPD).
[23]: Fast Back-to-Back Capable - The Target interface is not capable of fast back-to-back transactions. Always reads ‘0’.
[22]: UDF Supported - Not supported. Always reads ‘0’,
[21]: 66 Mhz Capable - Not supported. Always reads ‘0’.
[20:16]: Reserved. Always reads ‘00..0’.
[15:0]: Command Register - Writing one to a bit 15 - 0 sets the bit. Writing zero clears the bit.
[15:10]: Reserved - Always reads as ‘00..0’.
[9]: Fast back-to-Back Enable - Not implemented. Always reads ‘0’.
[8]: SERR# enable - Not implemented. Always reads ‘0’.
[7]: Wait cycle control - Not implemented. Always reads ‘0’.
[6]: Parity Error Response (PER) - Controls units response on parity error.
[5]: VGA Palette Snoop - Not implemented. Always reads ‘0’.
[4]: Memory Write and Invalidate Enable (MIE) - Enables the PCI Master interface to generate Memory Write and Invalidate Command.
[3]: Special Cycles - Not implemented. Always reads ‘0’.
[2]: Bus Master (BM) - Enbales the Master Interface to generate PCI cycles.
[1]: Memory Space (MS) - Allows the unit to respond to Memory space accesses.
[0]: I/O Space (IOS) - The unit never responds to I/O cycles. Always reads as ‘0’.

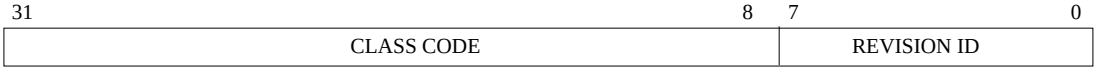

Figure 71. Class Code & revision ID

[31:8]: Class Code - Processor device class code: 0x0B4000 (Read-only).
[7:0]: Revision ID - 0x00 (Read-only).

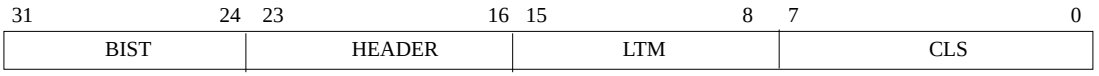

Figure 72. BIST, Header Type, Latency Timer and Cache Line Size register

[31:24]: BIST - Not supported. Reads always as ‘00..0’.
[23:16]: Header Type (HEADER)- Header Type 0. Reads always as ‘00..0’.
[15:8]: Latency Timer (LTIM) - Maximum number of PCI clock cycles that Master can own the bus.
[7:0]: Cache Line Size (CLS) - System cache line size. Defines the prefetch length for ‘Memory Read’ and ‘Memory Read Line’ commands.

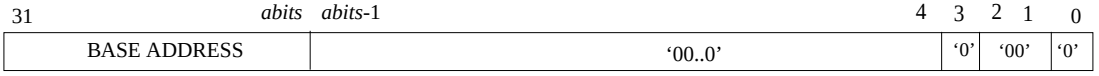

Figure 73. BAR0 register

[31:*abits*]: PCI Base Address - PCI Targets interface Base Address 0. The number of implemented bits depend on the VHDL-generic *abits*. Memory area of size 2^*abits* bytes at Base Address is occupied through this Base Address register. Register PAGE0 is accessed through upper half of this area. PCI memory accesses to the lower half of this area is translated to AHB accesses using PAGE0 map register.
[*abits*-1:4]: These bits are read-only and always read as ‘00..0’. This field can be used to determine devices memory requirement by writing value of all ones to this register and reading the value back. The device will return zeroes in unimplemented bits positions effectively defining memory area requested.
[3]: Prefetchable: Not supported. Always reads ‘0’.
[2:1]: Base Address Type - Mapping can be done anywhere in the 32-bit memory space. Reads always as ‘00’.
[0]: Memory Space Indicator - Register maps into Memory space. Read always as ‘0’.

PAGE0 register is mapped into upper half of the PCI address space defined by BAR0 register.

Figure 74. BAR1 register

[31:*dmaabits*]: PCI Base Address - PCI Targets interface Base Address 1. The number of implemented bits depends on the VHDL-generic *dmaabits*. Memory area of size 2^*dmaabits* bytes at Base Address is occupied through this Base Address register. PCI memory accesses to this memory space are translated to AHB accesses using PAGE1 map register.

[*dmaabits*-1:4]: These bits are read-only and always read as ‘00..0’. This field can be used to determine devices memory requirement by writing value of all ones to this register and reading the value back. The device will return zeroes in unimplemented bits positions effectively defining memory area requested.

[3]: Prefetchable: Not supported. Always reads as ‘0’.

[2:1]: Base Address Type - Mapping can be done anywhere in the 32-bit memory space. Reads always as ‘00’.

[0]: Memory Space Indicator - Register maps in Memory space. Read always as ‘0’.

0.0.2 PCI Target Map Registers

PAGE0 and PAGE1 registers map PCI access to AHB address space.

Address Space	Address	Register
PCI	Upper half of PCI address space defined by BAR0 register	PAGE0
AMBA APB	APB base address + 0x10	PAGE1

Figure 75. PAGE0 register

[31:*dmaabits*-1]: AHB Map Address - Maps PCI accesses to PCI BAR0 address space to AHB address space. AHB address is formed by concatenating AHB MAP with LSB of the PCI address.

[*dmaabits*-2:0]: Reserved. Reads always as ‘00..0’.

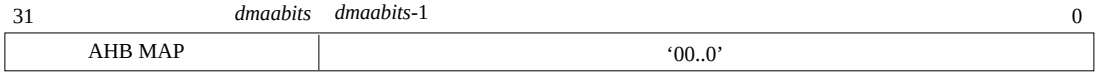

Figure 76. PAGE1 register

[31:*dmaabits*]: AHB Map Address (AHB MAP) - Maps PCI accesses to PCI BAR1 address space to AHB address space. AHB address is formed by concatenating AHB MAP with LSB of the PCI address.

[*dmaabits*-1:0]: Reserved. Reads always as ‘00..0’.

26.6 PCI Master Interface

PCI Master interface occupies 256 MB of AHB memory address space and 128 kB of AHB I/O address space. PCI Master interface handles AHB accesses to its back-end AHB Slave interface and translates them to PCI configuration, memory or I/O cycles.

Mapping of PCI masters AHB address space is configurable through VHDL-generics (see Table 94). PCI cycles performed on the PCI bus are directly dependable on AHB access and value in Configuration/Status register.

The PCI Master interface is capable of performing following PCI cycles:

- *PCI Configuration Cycles*: Single PCI Configuration cycles are performed by accessing upper 64 kB of AHB I/O address space allocated by the PCI Masters AHB Slave. Type 0 Configuration cycles are supported. Figure below shows mapping of LSB of AHB I/O address.

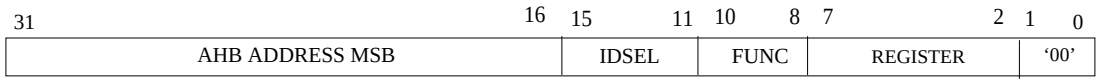

Figure 77. Mapping of AHB I/O addresses to PCI address for PCI Configuration cycles

[31:16]: AHB Address MSB - Not used for Configuration cycle address mapping.

[15:11]: IDSEL - This field is decoded to drive PCI AD[IDSEL+10]. AD[31:11] signal lines are supposed to drive IDSEL lines during Configuration Cycles.

[10:8]: Function Number (FUNC) - Selects function on multi-function device.

[7:2]: Register Number (REGISTER) - Used to index a PCI DWORD in Configuration Space.

[1:0]: Should always be driven to ‘00’ to generate Type 0 Configuration cycle.

- *I/O cycles*: Single PCI I/O cycles are supported. Lower 64 kB of the AHB I/O address space occupied by masters AHB slave interface are translated into PCI I/O cycles. Mapping is determined by value of I/O Map register (see. 26.7).
- *PCI memory cycles* are performed by accessing 256 MB AHB address space occupied by masters AHB slave. Mapping and PCI command generation are determined by value of AMBA Configuration/Status register (see 26.7). Burst operation is supported for PCI memory cycles.

The PCI commands generated by the master are directly dependant of the AMBA transfer type and the value of Configuration/Status register. The Configuration/Status register can be programmed to issue Memory Read, Memory Read Line, Memory Read Multiple, Memory Write or Memory Write and Invalidate.

If a burst AHB access is made to PCI Masters AHB memory space it is translated to burst PCI memory cycle. When the PCI Master interface is busy performing the transaction on the PCI bus, its AHB slave interface will not be able to accept new requests. ‘Retry’ response will be given to all accesses to its AHB slave interface. Requesting AHB Master should repeat its request until ‘OK’ or ‘Error’ response is given by the PCI Masters AHB slave interface.

Note that ‘RETRY’ responses on the PCI bus are not transparent, and will automatically be retried by the master PCI interface until the transfer is either finished or aborted.

For burst accesses, only linear-incremental mode is supported and is directly translated from the AMBA commands. The byte-enables on the PCI bus are translated from the HSIZE control AHB signal. Note that only WORD, HALF-WORD and BYTE values of HSIZE are valid. The data is aligned towards the LSB end of the bus, that is, a request with HZISE = "00" will be forwarded to the PCI bus with valid data on data lines 7:0.

26.7 PCI AMBA Registers

Following registers mapped into AMBA address space:

TABLE 96. AMBA registers

Address offset (from APB base address)	Register	Note
0x00	Configuration/Status register	-
0x04	BAR0 register	Read-only access from AMBA, write/read access from PCI (see 26.5.1).
0x08	PAGE0 register	Read-only access from AMBA, write/read access from PCI (see 0.0.2).
0x0C	BAR1 register	Read-only access from AMBA, write/read access from PCI (see 26.5.1).
0x10	PAGE1 register	-
0x14	IO Map register	-
0x18	Status & Command register (PCI Configuration Space Header)	Read-only access from AMBA, write/read access from PCI (see 26.5.1).

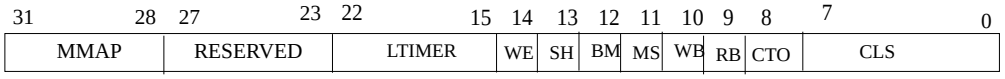

Figure 78. Configuration/Status register

[31:28]: Memory Space Map register - Defines mapping between PCI Masters AHB memory address space and PCI address space when performing PCI memory cycles. Value of this filed is used as 4 MSB of the PCI address. LSB bits are taken from the AHB address.

[27-23]: Reserved

[22-15]: Latency Timer (LTIMER) - Value of Latency Timer Register in Configuration Space Header. (Read-only)

[14]: Write Error (WE) - Target Write Error. Write access to units target interface resulted in error. (Read-only)

[13]: System Host (SH) - Set if the unit is system host. (Read-only)

[12]: Bus Master (BM) - Value of BM field in Command register in Configuration Space Header. (Read-only)

[11]: Memory Space (MS) - Value of MS field in Command register in Configuration Space Header. (Read-only)

[10]: Write Burst Command (WB) - Defines PCI command used for PCI write bursts.
‘0’ - ‘Memory Write’
‘1’ - ‘Memory Write and Invalidate’

[9]: Read Burst Command (RB) - Defines PCI command used for PCI read bursts.
‘0’ - Memory Read Multiple’
‘1’ - Memory Read Line’

[8]: Configuration Timeout (CTO) - Received timeout when performing Configuration cycle. (Read-only)

[7:0]: Cache Line Size (CLS) - Value of Cache Line Size register in Configuration Space Header. (Read-only)

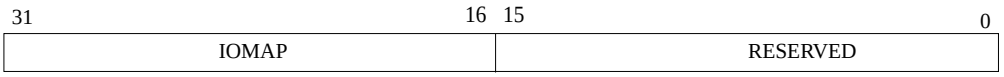

Figure 79. I/O Map register

[31:16]: I/O Map (IOMAP) - Most significant bits of PCI address when performing PCI I/O cycle. Concatenated with low bits of AHB address to from PCI address.

[15:0]: Reserved.

26.8 Signal description

PCI Target/Master unit signals are described in table 97.

TABLE 97. PCI Target/Master unit signals

Signal name	Field	Type	Function	Active
RST	N/A	Input	Reset	Low
CLK	N/A	Input	AMBA system clock	-
PCICLK	N/A	Input	PCI clock	-
PCII	*1	Input	PCI input signals	-
PCIO	*1	Output	PCI output signals	-
APBI	*2	Input	APB slave input signals	-
APBO	*2	Output	APB slave output signals	-
AHBMI	*2	Input	AHB master input signals	-
AHBMO	*2	Output	AHB master output signals	-
AHBSI	*2	Input	AHB slave input signals	-
AHBSO	*2	Output	AHB slave output signals	-

*1) see PCI specification

*2) see GRLIB IP Library User's Manual

The PCIO record contains an additional output enable signal VADEN. It is has the same value as aden at each index but they are all driven from separate registers. A directive is placed on this vector so that the registers will not be removed during synthesis. This output enable vector can be used instead of aden if output delay is an issue in the design.

26.9 Library dependencies

Table 98 shows libraries that should be used when instantiating the PCI Target/Master unit.

TABLE 98. Library dependencies

Library	Package	Imported unit(s)	Description
GRLIB	AMBA	Signals	AMBA signal definitions
GAISLER	PCI	Signals, component	PCI signals and component declaration
GAISLER	PADS	Components	PCI pads

26.10 Example instantiation

```
library ieee;
use ieee.std_logic_1164.all;
library grlib;
use grlib.amba.all;
use grlib.stdlib.all;
use grlib.tech.all;
library gaisler;
use gaisler.pci.all;
use gaisler.pads.all;

.
.
signal apbi  : apb_slv_in_type;
signal apbo  : apb_slv_out_vector := (others => apb_none);
signal ahbsi : ahb_slv_in_type;
signal ahbso : ahb_slv_out_vector := (others => ahbs_none);
signal ahbmi : ahb_mst_in_type;
signal ahbmo : ahb_mst_out_vector := (others => ahbm_none);


signal pcii : pci_in_type;
signal pcio : pci_out_type;

begin

pci0 : pci_mtf generic map (memtech => memtech,
    hmstndx => 1,
    fifodepth => log2(CFG_PCIDEPTH), device_id => CFG_PCIDID, vendor_id => CFG_PCIVID,
    hslvndx => 4, pindex => 4, paddr => 4, haddr => 16#E00#,
    ioaddr => 16#400#, nsync => 2)
port map (rstn, clkm, pciclk, pcii, pcio, apbi, apbo(4), ahbmi,
    ahbmo(1), ahbsi, ahbso(4));

pcipads0 : pcipads generic map (padtech => padtech)-- PCI pads
    port map ( pci_rst, pci_gnt, pci_idsel, pci_lock, pci_ad, pci_cbe,
      pci_frame, pci_irdy, pci_trdy, pci_devsel, pci_stop, pci_perr,
      pci_par, pci_req, pci_serr, pci_host, pci_66, pcii, pcio );
```

27 IRQMP - Multiprocessor Interrupt Controller

27.1 Overview

AMBA system in GRLIB provides an interrupt scheme where interrupt lines are routed together with the remaining AHB/APB bus signals. Interrupts from AHB and APB units are routed through the bus, combined together, and propagated back to all units. The multi-processor interrupt controller core (IRQMP) is attached to AMBA bus as an APB slave, and monitors the combined interrupt signals. The IRQMP core prioritizes, masks and propagates interrupts to one or more LEON3 processors.

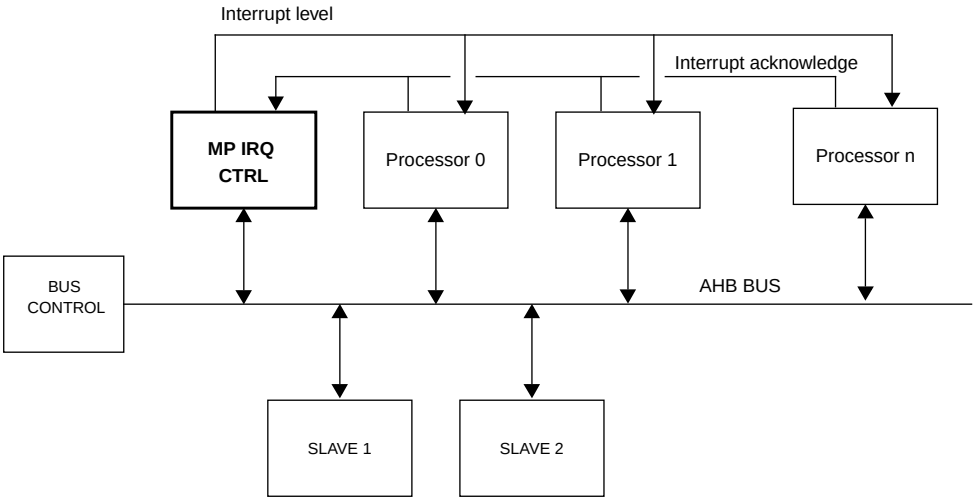

Figure 80. LEON3 Multiprocessor system with MP Interrupt controller

27.2 Operation

27.2.1 Interrupt prioritization

IRQMP monitors interrupt 1 - 15 of the AMBA interrupt bus. Each interrupt can be assigned to one of two levels (0 or 1) as programmed in the interrupt level register. Level 1 has higher priority than level 0. The interrupts are prioritised within each level, with interrupt 15 having the highest priority and interrupt 1 the lowest. The highest interrupt from level 1 will be forwarded to the processor. If no unmasked pending interrupt exists on level 1, then the highest unmasked interrupt from level 0 will be forwarded.

Interrupts are prioritised at system level, while masking and forwarding of interrupts in done for each processor separately. Each processor in an MP system has separate interrupt mask and force registers. When an interrupt is signalled on the AMBA bus, the interrupt controller will prioritize interrupts, perform interrupt masking for each processor according to the mask in the corresponding mask register and forward the interrupts to the processors.

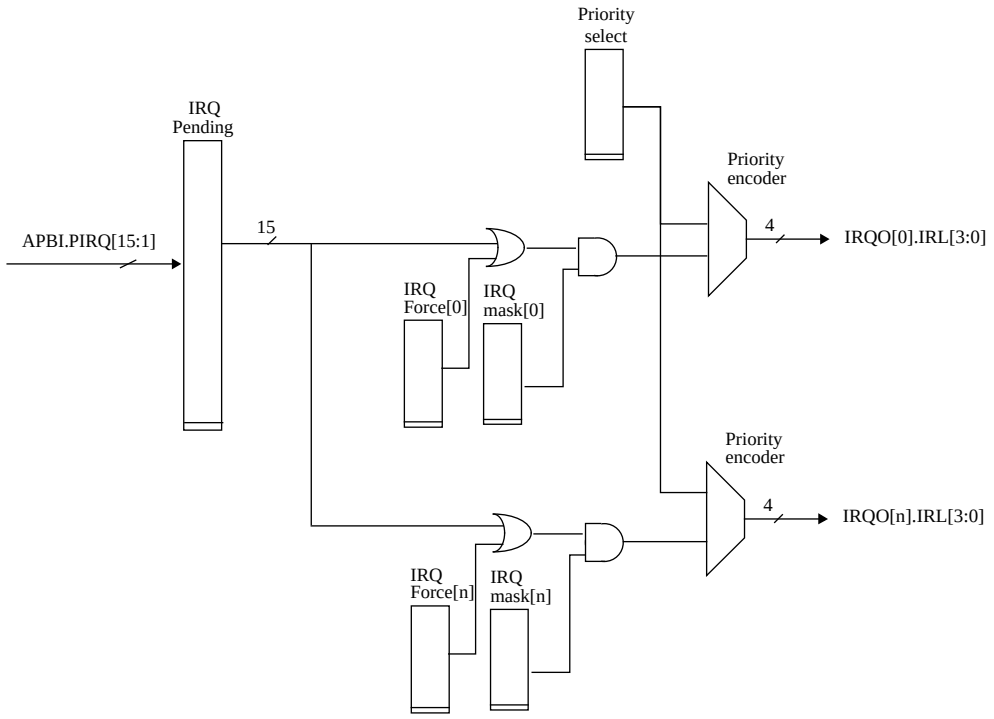

Figure 81. IRQMP Interrupt controller block diagram

When one of the processors acknowledges the interrupt, the corresponding pending bit will automatically be cleared. Interrupt can also be forced by setting a bit in the interrupt force register. In this case, the processor acknowledgement will clear the force bit rather than the pending bit. After reset, the interrupt mask register is set to all zeros while the remaining control registers are undefined. Note that interrupt 15 cannot be maskable by the LEON3 processor and should be used with care - most operating systems do not safely handle this interrupt.

27.2.2 Processor status monitoring

The processor status can be monitored through the MP Status Register. The STATUS field 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 status 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.

27.3 Configuration options

The MP Interrupt Controller has the following configuration options (VHDL generics):

TABLE 99. IRQMP Configuration options (generics)

Generic	Function	Allowed range	Default
pindex	Selects which APB select signal (PSEL) will be used to access the timer 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 MP system.	1 to 16	1

27.4 Vendor and device id

The module has Vendor ID 0x01 (Gaisler Research) and Device ID 0x00D. For description of vendor and device ids see GRLIB IP Library User's Manual.

27.5 Registers

Table 100 shows the MP Interrupt Controller registers. The number of implemented registers depend on number of processor in the MP system.

TABLE 100. MP IRQ Controller registers

Register	APB Address offset
Interrupt level register	0x00
Interrupt pending register	0x04
Interrupt force register (NCPU = 0)	0x08
Interrupt clear register	0x0C
Multi-processor status register	0x10
Processor 0 interrupt mask register	0x40
Processor 1 interrupt mask register	0x44
Processor *n* interrupt mask register	0x40 + 4 * *n*
Processor 0 interrupt force register	0x80
Processor 1 interrupt force register	0x84
Processor *n* interrupt force register	0x80 + 4 * *n*

27.5.1 Interrupt level register

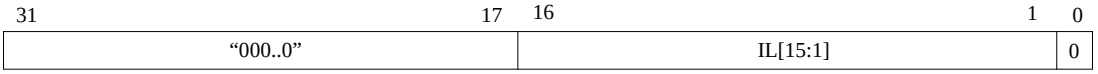

Figure 82. Interrupt level register

[31:16] - Reserved.

[15:1] - Interrupt Level *n* (IL[*n*]): Interrupt level for interrupt *n*.

[0] - Reserved.

27.5.2 Interrupt pending register

Figure 83. Interrupt pending register

[31:17] - Reserved.
[16:1] - Interrupt Pending n (IP[n]): Interrupt pending for interrupt n.
[0] - Reserved

27.5.3 Interrupt force register (NCPU = 0)

Figure 84. Interrupt force register

[31:16] - Reserved.
[15:1] - Interrupt Force *n* (IF[*n*]): Force interrupt nr *n*.
[0] - Reserved.

27.5.4 Interrupt clear register

Figure 85. Interrupt clear register

[31:16] - Reserved.
[15:1] - Interrupt Clear *n* (IC[*n*]): Writing ‘1’ to IC*n* will clear interrupt *n*.
[0] - Reserved.

27.5.5 Interrupt mask register

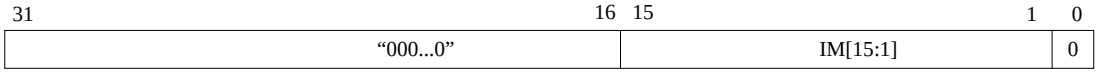

Figure 86. Interrupt mask register

[31:16] - Reserved.
[15:1] - Interrupt Mask *n* (IM[*n*]): If IMn = 0 the interrupt *n* is masked, otherwise it is enabled.
[0] - Reserved.

27.5.6 Multi-processor status register

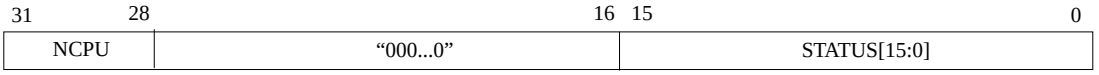

Figure 87. Multi-processor status register

[31:28] - NCPU. Number of CPU’s in the system -1 .

[27:16] - Reserved.

[15:1] - Power-down status of CPU [*n*]: ‘1’ = power-down, ‘0’ = running. Write with ‘1’ to force processor *n* out of power-down.

27.5.7 Interrupt force register (NCPU > 1)

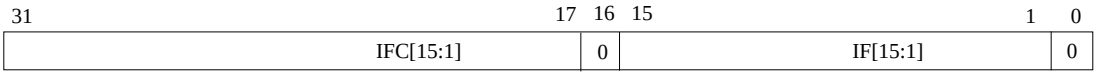

Figure 88. Interrupt force register

[31:17] - Interrupt force clear *n* (IFC[*n*]).

[15:1] - Interrupt Force *n* (IF[*n*]): Force interrupt nr *n*.

[0] - Reserved.

27.6 Signal description

The interrupt controller signals are described in table 101.

TABLE 101. MP Interrupt Controller signals

Signal name	Type	Function	Active
RST	Input	Reset	Low
CLK	Input	Clock	-
APBI	Input	APB slave input signals	-
APBO	Output	APB slave output signals	-
IRQI.INTACK	Input	Interrupt acknowledge	High
IRQI.IRL[3:0]	Input	Processor interrupt level	High
IRQO.IRL[3:0]	Output	Input interrupt level	High

* see GRLIB IP Library User’s Manual

27.7 Library dependencies

Table 102 shows libraries that should be used when instantiating the interrupt controller.

TABLE 102. Library dependencies

Library	Package	Imported unit(s)	Description
GRLIB	AMBA	Signals	AMBA signal definitions
GAISLER	LEON3	Signals, component	Leon3 signals and component declaration

27.8 MP IRQ controller instantiation example

```
library ieee;
use ieee.std_logic_1164.all;

library grlib;
use grlib.amba.all;
library gaisler;
use gaisler.leon3.all;

entity irqmp_ex is
  port (
    clk : in std_ulogic;
    rstn : in std_ulogic;

    ...  -- other signals
    );
end;

architecture rtl of irqmp_ex is
  constant NCPU : integer := 4;

  -- AMBA signals
  signal apbi  : apb_slv_in_type;
  signal apbo  : apb_slv_out_vector := (others => apb_none);
  signal ahbmi : ahb_mst_in_type;
  signal ahbmo : ahb_mst_out_vector := (others => ahbm_none);
  signal ahbsi : ahb_slv_in_type;

  -- GP Timer Unit input signals
  signal irqi   : irq_in_vector(0 to NCPU-1);
  signal irqo   : irq_out_vector(0 to NCPU-1);

  -- LEON3 signals
  signal leon3i : l3_in_vector(0 to NCPU-1);
  signal leon3o : l3_out_vector(0 to NCPU-1);

begin

  -- 4 LEON3 processors are instantiated here
  cpu : for i in 0 to NCPU-1 generate
    u0 : leon3s generic map (hindex => i)
    port map (clk, rstn, ahbmi, ahbmo(i), ahbsi,
irqi(i), irqo(i), dbgi(i), dbgo(i));
  end generate;

  -- MP IRQ controller
  irqctrl0 : irqmp
  generic map (pindex => 2, paddr => 2, ncpu => NCPU)
  port map (rstn, clk, apbi, apbo(2), irqi, irqo);

end;
```

28 LEON3 - High-performance SPARC V8 32-bit Processor

28.1 Overview

LEON3 is a 32-bit processor core conforming to the IEEE-1754 (SPARC V8) architecture. It is designed for embedded applications, combining high performance with low complexity and low power consumption.

The LEON3 core has the following main features: 7-stage pipeline with Harvard architecture, separate instruction and data caches, hardware multiplier and divider, on-chip debug support and multi-processor extensions.

A block diagram of the LEON3 core can be seen below:

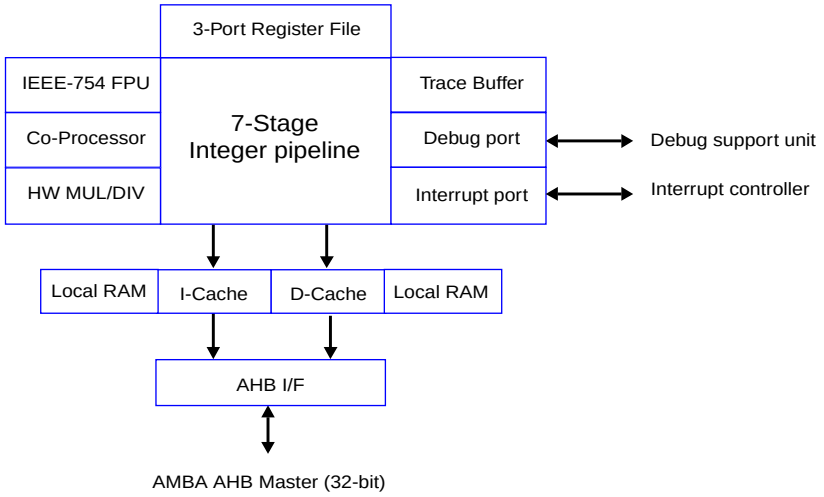

Figure 89. LEON3 processor core block diagram

Note: this manual describes the full functionality of the LEON3 core. Through the use of VHDL generics, parts of the described functionality can be suppressed or modified to generate a smaller or faster implementation.

28.1.1 Integer unit

The LEON3 integer unit implements the full SPARC V8 standard, including hardware multiply and divide instructions. The number of register windows is configurable within the limit of the SPARC standard (2 - 32), with a default setting of 8. The pipeline consists of 7 stages with a separate instruction and data cache interface (Harvard architecture).

28.1.2 Cache sub-system

LEON3 has a highly configurable cache system, consisting of a separate instruction and data cache. Both caches can be configured with 1 - 4 sets, 1 - 256 kbyte/set, 16 or 32 bytes per line. Sub-blocking is implemented with one valid bit per 32-bit word. The instruction cache uses streaming during line-refill to minimize refill latency. The data cache uses write-through policy and implements a double-word write-buffer. The data cache can also perform bus-snooping on the AHB bus. A local scratch pad ram can be added to both the instruction and data cache controllers to allow 0-waitstates access memory without data write back.

28.1.3 Floating-point unit and co-processor

The LEON3 integer unit provides interfaces for a floating-point unit (FPU), and a custom co-processor. Two FPU controllers are available, one for the high-performance GRFPU (available from Gaisler Research) and one for the Meiko FPU core (available from Sun Microsystems). The floating-point processors and co-processor execute in parallel with the integer unit, and does not block the operation unless a data or resource dependency exists.

28.1.4 On-chip debug support

The LEON3 pipeline includes functionality to allow non-intrusive debugging on target hardware. To aid software debugging, up to four watchpoint registers can be enabled. Each register can cause a breakpoint trap on an arbitrary instruction or data address range. When the (optional) debug support unit is attached, the watchpoints can be used to enter debug mode. Through a debug support interface, full access to all processor registers and caches is provided. The debug interfaces also allows single stepping, instruction tracing and hardware breakpoint/watchpoint control. An internal trace buffer can monitor and store executed instructions, which can later be read out over the debug interface.

28.1.5 Interrupt interface

LEON3 supports the SPARC V8 interrupt model with a total of 15 asynchronous interrupts. The interrupt interface provides functionality to both generate and acknowledge interrupts.

28.1.6 AMBA interface

The cache system implements an AMBA AHB master to load and store data to/from the caches. The interface is compliant with the AMBA-2.0 standard. During line refill, incremental burst are generated to optimise the data transfer.

28.1.7 Power-down mode

The LEON3 processor core implements a power-down mode, which halts the pipeline and caches until the next interrupt. This is an efficient way to minimize power-consumption when the application is idle, and does not require tool-specific support in form of clock gating.

28.1.8 Multi-processor support

LEON3 is designed to be use in multi-processor systems. Each processor has a unique index to allow processor enumeration. The write-through caches and snooping mechanism guarantees memory coherency in shared-memory systems.

28.1.9 Performance

Using 8K + 8K caches and a 16x16 multiplier, the dhrystone 2.1 benchmark reports 1,600 iteration/s/MHz using the gcc-2.95.2 compiler (-O2). This translates to 0.91 dhrystone MIPS/MHz using the VAX 11/780 value a reference for one MIPS.

28.2 LEON3 integer unit

28.2.1 Overview

The LEON3 integer unit implements the integer part of the SPARC V8 instruction set. The implementation is focused on high performance and low complexity. The LEON3 integer unit has the following main features:

- 7-stage instruction pipeline
- Separate instruction and data cache interface
- Support for 2 - 32 register windows
- Hardware multiplier with optional 16x16 bit MAC and 40-bit accumulator
- Radix-2 divider (non-restoring)
- Single-vector trapping for reduced code size

Figure 90 shows a block diagram of the integer unit.

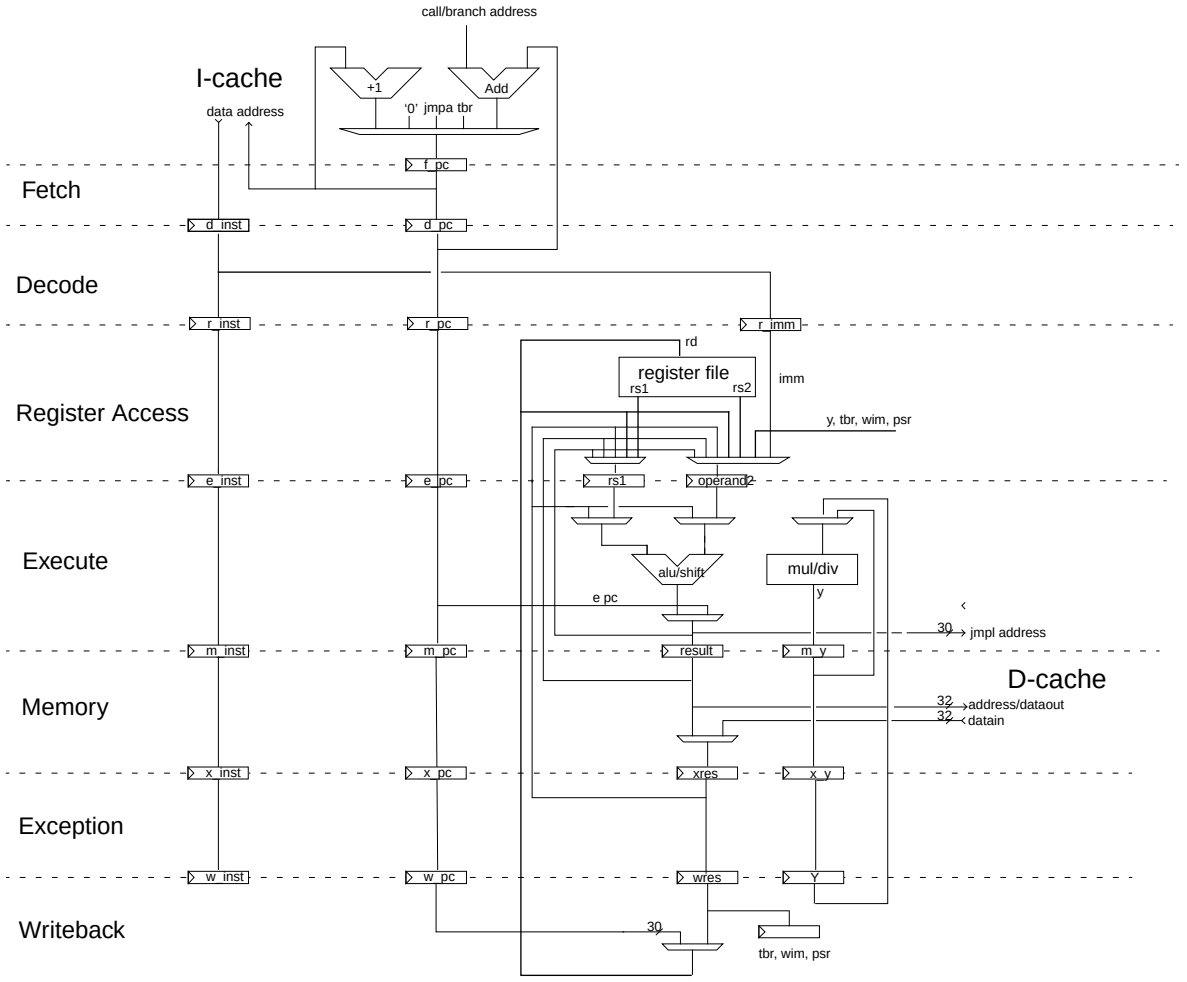

Figure 90. LEON3 integer unit datapath diagram

28.2.2 Instruction pipeline

The LEON integer unit uses a single instruction issue pipeline with 7 stages:

1. FE (Instruction Fetch): If the instruction cache is enabled, the instruction is fetched from the instruction cache. Otherwise, the fetch is forwarded to the memory controller. The instruction is valid at the end of this stage and is latched inside the IU.
2. DE (Decode): The instruction is decoded and the CALL and Branch target addresses are generated.
3. RA (Register access): Operands are read from the register file or from internal data bypasses.
4. EX (Execute): ALU, logical, and shift operations are performed. For memory operations (e.g., LD) and for JMPL/RETT, the address is generated.
5. ME (Memory): Data cache is accessed. Store data read out in the execution stage is written to the data cache at this time.
6. XC (Exception) Traps and interrupts are resolved. For cache reads, the data is aligned as appropriate.
7. WR (Write): The result of any ALU, logical, shift, or cache operations are written back to the register file.

Table 103 lists the cycles per instruction (assuming cache hit and no icc or load interlock):

TABLE 103. Instruction timing

Instruction	Cycles
JMPL, RETT	3
Double load	2
Single store	2
Double store	3
SMUL/UMUL	4*
SDIV/UDIV	35
Taken Trap	5
Atomic load/store	3
All other instructions	**1**

* Multiplication cycle count is 5 clocks when the multiplier is configured to be pipelined.

28.2.3 SPARC Implementor’s ID

Gaisler Research is assigned number 15 (0xF) as SPARC implementor’s identification. This value is hard-coded into bits 31:28 in the %psr register. The version number for LEON3 is 3, which is hard-coded in to bits 27:24 of the %psr.

28.2.4 Multiply instructions

The LEON processor supports the SPARC integer multiply instructions UMUL, SMUL UMULCC and SMULCC. These instructions perform a 32x32-bit integer multiply, producing a 64-bit result. SMUL and SMULCC performs signed multiply while UMUL and UMULCC performs unsigned multiply. UMULCC and SMULCC also set the condition codes to reflect the

result. The multiply instructions are performed using a 16x16 signed hardware multiplier, which is iterated four times. To improve the timing, the 16x16 multiplier can optionally be provided with a pipeline stage.

28.2.5 Multiply and accumulate instructions

To accelerate DSP algorithms, two multiply&accumulate instructions are implemented: UMAC and SMAC. The UMAC performs an unsigned 16-bit multiply, producing a 32-bit result, and adds the result to a 40-bit accumulator made up by the 8 lsb bits from the %y register and the %asr18 register. The least significant 32 bits are also written to the destination register. SMAC works similarly but performs signed multiply and accumulate. The MAC instructions execute in one clock but have two clocks latency, meaning that one pipeline stall cycle will be inserted if the following instruction uses the destination register of the MAC as a source operand.

Assembler syntax:

```
umac            rs1, reg_imm, rd
smac            rs1, reg_imm, rd
```

Operation:

```
prod[31:0] = rs1[15:0] * reg_imm[15:0]
result[39:0] = (Y[7:0] & %asr18[31:0]) + prod[31:0]
(Y[7:0] & %asr18[31:0]) = result[39:0]
rd = result[31:0]
```

%asr18 can be read and written using the RDASR and WRASR instructions.

28.2.6 Divide instructions

Full support for SPARC V8 divide instructions is provided (SDIV, UDIV, SDIVCC & UDIVCC). The divide instructions perform a 64-by-32 bit divide and produce a 32-bit result. Rounding and overflow detection is performed as defined in the SPARC V8 standard.

28.2.7 Hardware breakpoints

The integer unit can be configured to include up to four hardware breakpoints. Each breakpoint consists of a pair of application-specific registers (%asr24/25, %asr26/27, %asr28/30 and %asr30/31) registers; one with the break address and one with a mask:

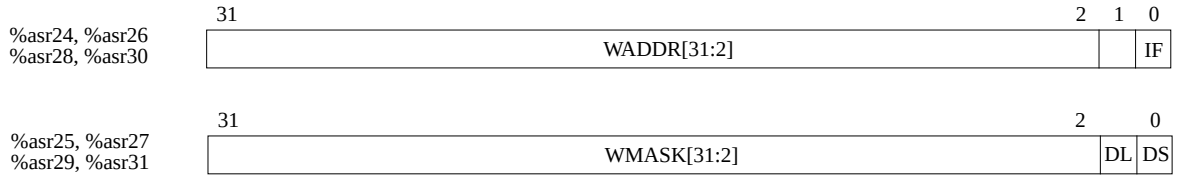

Figure 91. Watch-point registers

Any binary aligned address range can be watched - the range is defined by the WADDR field, masked by the WMASK field (WMASK[x] = 1 enables comparison). On a breakpoint hit, trap 0x0B is generated. By setting the IF, DL and DS bits, a hit can be generated on instruction fetch, data load or data store. Clearing these three bits will effectively disable the breakpoint function.

28.2.8 Instruction trace buffer

The instruction trace buffer consists of a circular buffer that stores executed instructions. The trace buffer operation is controlled through the debug support interface, and does not affect processor operation (see the DSU description). The size of the trace buffer is configurable from 1 to 64 kB through a VHDL generic. The trace buffer is 128 bits wide, and stores the following information:

- Instruction address and opcode
- Instruction result
- Load/store data and address
- Trap information
- 30-bit time tag

The operation and control of the trace buffer is further described in “Instruction trace buffer” on page 66. Note that in multi-processor systems, each processor has its own trace buffer allowing simultaneous tracing of all instruction streams.

28.2.9 Processor configuration register

The application specific register 17 (%asr17) provides information on how various configuration options were set during synthesis. This can be used to enhance the performance of software, or to support enumeration in multi-processor systems. The register can be accessed through the RDASR instruction, and has the following layout:

Figure 92. LEON3 configuration register (%asr17)

Field Definitions:

[31:28]: Processor index. In multi-processor systems, each LEON core gets a unique index to support enumeration. The value in this field is identical to the *hindex* generic parameter in the VHDL model.

[13]: Single-vector trapping (SVT) enable. If set, will enable single-vector trapping. Fixed to zero if SVT is not implemented.

[12]: Load delay. If set, the pipeline uses a 2-cycle load delay. Otherwise, a 1-cycle load delay i s used. Generated from the *lddel* generic parameter in the VHDL model.

[11:10]: FPU option. “00” = no FPU; “01” = GRFPU; “10” = Meiko FPU.

[9]: If set, the optional multiply-accumulate (MAC) instruction is available

[8]: If set, the SPARC V8 multiply and divide instructions are available.

[7:5]: Number of implemented watchpoints (0 - 7)

[4:0]: Number of implemented registers windows corresponds to NWIN+1.

28.2.10 Exceptions

LEON adheres to the general SPARC trap model. The table below shows the implemented traps and their individual priority.

TABLE 104. Trap allocation and priority

Trap	TT	Pri	Description
reset	0x00	1	Power-on reset
write error	0x2b	2	write buffer error
instruction_access_error	0x01	3	Error during instruction fetch
illegal_instruction	0x02	5	UNIMP or other un-implemented instruction
privileged_instruction	0x03	4	Execution of privileged instruction in user mode
fp_disabled	0x04	6	FP instruction while FPU disabled
cp_disabled	0x24	6	CP instruction while Co-processor disabled
watchpoint_detected	0x0B	7	Hardware breakpoint match
window_overflow	0x05	8	SAVE into invalid window
window_underflow	0x06	8	RESTORE into invalid window
register_hadrware_error	0x20	9	register file EDAC error (LEON-FT only)
mem_address_not_aligned	0x07	10	Memory access to un-aligned address
fp_exception	0x08	11	FPU exception
cp_exception	0x28	11	Co-processor exception
data_access_exception	0x09	13	Access error during load or store instruction
tag_overflow	0x0A	14	Tagged arithmetic overflow
divide_exception	0x2A	15	Divide by zero
interrupt_level_1	0x11	31	Asynchronous interrupt 1
interrupt_level_2	0x12	30	Asynchronous interrupt 2
interrupt_level_3	0x13	29	Asynchronous interrupt 3
interrupt_level_4	0x14	28	Asynchronous interrupt 4
interrupt_level_5	0x15	27	Asynchronous interrupt 5
interrupt_level_6	0x16	26	Asynchronous interrupt 6
interrupt_level_7	0x17	25	Asynchronous interrupt 7
interrupt_level_8	0x18	24	Asynchronous interrupt 8
interrupt_level_9	0x19	23	Asynchronous interrupt 9
interrupt_level_10	0x1A	22	Asynchronous interrupt 10
interrupt_level_11	0x1B	21	Asynchronous interrupt 11
interrupt_level_12	0x1C	20	Asynchronous interrupt 12
interrupt_level_13	0x1D	19	Asynchronous interrupt 13
interrupt_level_14	0x1E	18	Asynchronous interrupt 14
interrupt_level_15	0x1F	17	Asynchronous interrupt 15
trap_instruction	0x80 - 0xFF	16	Software trap instruction (TA)

28.2.11 Single vector trapping (SVT)

Single-vector trapping (SVT) is an SPARC V8e option to reduce code size for embedded applications. When enabled, any taken trap will always jump to the reset trap handler (%tbr.tba + 0). The trap type will be indicated in %tbr.tt, and must be decoded by the shared trap handler. SVT is enabled by setting bit 13 in %asr17. The model must also be configured with the SVT generic = 1.

28.2.12 Address space identifiers (ASI)

In addition to the address, a SPARC processor also generates an 8-bit address space identifier (ASI), providing up to 256 separate, 32-bit address spaces. During normal operation, the LEON3 processor accesses instructions and data using ASI 0x8 - 0xB as defined in the SPARC standard. Using the LDA/STA instructions, alternative address spaces can be accessed. The table shows the ASI usage for LEON. Only ASI[5:0] are used for the mapping, ASI[7:6] have no influence on operation.

TABLE 105. ASI usage

ASI	Usage
0x01	Forced cache miss
0x02	System control registers (cache control register)
0x08, 0x09, 0x0A, 0x0B	Normal cached access (replace if cacheable)
0x0C	Instruction cache tags
0x0D	Instruction cache data
0x0E	Data cache tags
0x0F	Data cache data
0x10	Flush instruction cache
0x11	Flush data cache

28.2.13 Power-down

The processor can be configured to include a power-down feature to minimize power consumption during idle periods. The power-down mode is entered by performing a WRASR instruction to %asr19:

```
wr %g0, %asr19
```

During power-down, the pipeline is halted until the next interrupt occurs. Signals inside the processor pipeline and caches are then static, reducing power consumption from dynamic switching.

28.2.14 Processor reset operation

The processor is reset by asserting the RESET input for at least 4 clock cycles. The following table indicates the reset values of the registers which are affected by the reset. All other registers maintain their value (or are undefined).

TABLE 106. Processor reset values

Register	Reset value
PC (program counter)	0x0
nPC (next program counter)	0x4
PSR (processor status register)	ET=0, S=1

By default, the execution will start from address 0. This can be overridden by setting the RSTADDR generic in the model to a non-zero value. The reset address is however always aligned on a 4 kbyte boundary.

28.2.15 Multi-processor support

The LEON3 processor support synchronous multi-processing (SMP) configurations, with up to 16 processors attached to the same AHB bus. In multi-processor systems, only the processor with index 0 will start. All other processors will remain halted in power-down mode. After the system has been initialized, the halted processors can be started by writing to the 'MP status register', located in the multi-processor interrupt controller. The halted processors start executing from the reset address (0 or RSTADDR generic).

28.2.16 Cache sub-system

The LEON3 processor implements a Harvard architecture with separate instruction and data buses, connected to two independent cache controllers. Both instruction and data cache controllers can be separately configured to implement a direct-mapped cache or a multi-set cache with set associativity of 2 - 4. The set size is configurable to 1 - 256 kbyte, divided into cache lines with 16 or 32 bytes of data. In multi-set configurations, one of three replacement policies can be selected: least-recently-used (LRU), least-recently-replaced (LRR) or (pseudo-) random. If the LRR algorithm can only be used when the cache is 2-way associative. A cache line can be locked in the instruction or data cache preventing it from being replaced by the replacement algorithm.

NOTE: The LRR algorithm uses one extra bit in tag rams to store replacement history. The LRU algorithm needs extra flip-flops per cache line to store access history. The random replacement algorithm is implemented through modulo-N counter that selects which line to evict on cache miss.

Cachability for both caches is controlled through the AHB plug&play address information. The memory mapping for each AHB slave indicates whether the area is cachable, and this information is used to (statically) determine which access will be treated as cacheable. This approach means that the cachability mapping is always coherent with the current AHB configuration.

The detailed operation of the instruction and data caches is described in the following sections.

28.3 Instruction cache

28.3.1 Operation

The instruction cache can be configured as a direct-mapped cache or as a multi-set cache with associativity of 2 - 4 implementing either LRU or random replacement policy or as 2-way associative cache implementing LRR algorithm. The set size is configurable to 1 - 64 kbyte and divided into cache lines of 16- 32 bytes. Each line has a cache tag associated with it consisting of a tag field, valid field with one valid bit for each 4-byte sub-block and optional LRR and lock bits. On an instruction cache miss to a cachable location, the instruction is fetched and the corresponding tag and data line updated. In a multi-set configuration a line to be replaced is chosen according to the replacement policy.

If instruction burst fetch is enabled in the cache control register (CCR) the cache line is filled from main memory starting at the missed address and until the end of the line. At the same time, the instructions are forwarded to the IU (streaming). If the IU cannot accept the streamed instructions due to internal dependencies or multi-cycle instruction, the IU is halted until the line fill is completed. If the IU executes a control transfer instruction (branch/CALL/JMPL/RETT/TRAP) during the line fill, the line fill will be terminated on the next fetch. If instruction burst fetch is enabled, instruction streaming is enabled even when the cache is disabled. In this case, the fetched instructions are only forwarded to the IU and the cache is not updated.

If the line fill starts at the first word in the line, 4 or 8 word fixed-length burst is generated on the AHB bus depending on the line size. Otherwise an incremental burst starting at the address of the missed word and ending at line end is generated.

If a memory access error occurs during a line fill with the IU halted, the corresponding valid bit in the cache tag will not be set. If the IU later fetches an instruction from the failed address, a cache miss will occur, triggering a new access to the failed address. If the error remains, an instruction access error trap (tt=0x1) will be generated.

28.3.2 Instruction cache tag

A instruction cache tag entry consists of several fields as shown in figure 93:

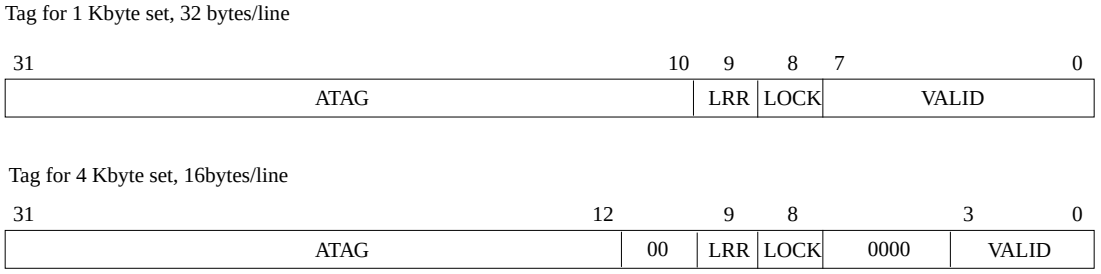

Figure 93. Instruction cache tag layout examples

Field Definitions:

[31:10]: Address Tag (ATAG) - Contains the tag address of the cache line.

[9]: LRR - Used by LRR algorithm to store replacement history, otherwise 0.

[8]: LOCK - Locks a cache line when set. 0 if cache locking not implemented.

[7:0]: Valid (V) - When set, the corresponding sub-block of the cache line contains valid data. These bits is set when a sub-block is filled due to a successful cache miss; a cache fill which results in a memory error will leave the valid bit unset. A FLUSH instruction will clear all valid bits. V[0] corresponds to address 0 in the cache line, V[1] to address 1, V[2] to address 2 and so on.

NOTE: only the necessary bits will be implemented in the cache tag, depending on the cache configuration. As an example, a 4 kbyte cache with 16 bytes per line would only have four valid bits and 20 tag bits. The cache rams are sized automatically by the ram generators in the model.

28.4 Data cache

28.4.1 Operation

The data cache can be configured as a direct-mapped cache or as a multi-set cache with associativity of 2 - 4 implementing either LRU or (pseudo-) random replacement policy or as 2-way associative cache implementing LRR algorithm. The set size is configurable to 1 - 64 kbyte and divided into cache lines of 16 - 32 bytes. Each line has a cache tag associated with it consisting of a tag field, valid field with one valid bit for each 4-byte sub-block and optional lock and LRR bits. On a data cache read-miss to a cachable location 4 bytes of data are loaded into the cache from main memory. The write policy for stores is write-through with no-allocate on write-miss. In a multi-set configuration a line to be replaced on read-miss is chosen according to the replacement policy. If a memory access error occurs during a data load, the corresponding valid bit in the cache tag will not be set. and a data access error trap (tt=0x9) will be generated.

28.4.2 Write buffer

The write buffer (WRB) consists of three 32-bit registers used to temporarily hold store data until it is sent to the destination device. For half-word or byte stores, the stored data replicated into proper byte alignment for writing to a word-addressed device, before being loaded into one of the WRB registers. The WRB is emptied prior to a load-miss cache-fill sequence to avoid any stale data from being read in to the data cache.

Since the processor executes in parallel with the write buffer, a write error will not cause an exception to the store instruction. Depending on memory and cache activity, the write cycle may not occur until several clock cycles after the store instructions has completed. If a write error occurs, the currently executing instruction will take trap 0x2b.

Note: the 0x2b trap handler should flush the data cache, since a write hit would update the cache while the memory would keep the old value due the write error.

28.4.3 Data cache tag

A data cache tag entry consists of several fields as shown in figure 94:

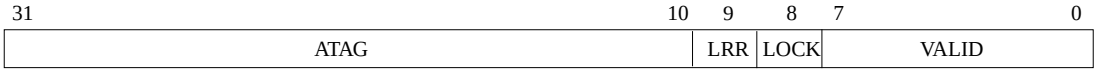

Figure 94. Data cache tag layout

Field Definitions:

[31:10]: Address Tag (ATAG) - Contains the address of the data held in the cache line.

[9]: LRR - Used by LRR algorithm to store replacement history. ‘0’ if LRR is not used.

[8]: LOCK - Locks a cache line when set. ‘0’ if instruction cache locking was not enabled in the configuration.

[3:0]: Valid (V) - When set, the corresponding sub-block of the cache line contains valid data. These bits is set when a sub-block is filled due to a successful cache miss; a cache fill which results in a memory error will leave the valid bit unset. V[0] corresponds to address 0 in the cache line, V[1] to address 1, V[2] to address 2 and V[3] to address 3.

NOTE: only the necessary bits will be implemented in the cache tag, depending on the cache configuration. As an example, a 2 kbyte cache with 32 bytes per line would only have eight valid bits and 21 tag bits. The cache rams are sized automatically by the ram generators in the model.

28.5 Additional cache functionality

28.5.1 Cache flushing

Both instruction and data cache are flushed by executing the FLUSH instruction. The instruction cache is also flushed by setting the FI bit in the cache control register, or by writing to any location with ASI=0x15. The data cache is also flushed by setting the FD bit in the cache control register, or by writing to any location with ASI=0x16. Cache flushing takes one cycle per cache line, during which the IU will not be halted, but during which the caches are disabled. When the flush operation is completed, the cache will resume the state (disabled, enabled or frozen) indicated in the cache control register.

28.5.2 Diagnostic cache access

Tags and data in the instruction and data cache can be accessed through ASI address space 0xC, 0xD, 0xE and 0xF by executing LDA and STA instructions. Address bits making up the cache offset will be used to index the tag to be accessed while the least significant bits of the bits making up the address tag will be used to index the cache set.

Diagnostic read of tags is possible by executing an LDA instruction with ASI=0xC for instruction cache tags and ASI=0xE for data cache tags. A cache line and set are indexed by the address bits making up the cache offset and the least significant bits of the address bits making up the address tag. Similarly, the data sub-blocks may be read by executing an LDA instruction with ASI=0xD for instruction cache data and ASI=0xF for data cache data. The sub-block to be read in the indexed cache line and set is selected by A[4:2].

The tags can be directly written by executing a STA instruction with ASI=0xC for the instruction cache tags and ASI=0xE for the data cache tags. The cache line and set are indexed by the address bits making up the cache offset and the least significant bits of the address bits making up the address tag. D[31:10] is written into the ATAG filed (see above) and the valid bits are written with the D[7:0] of the write data. Bit D[9] is written into the LRR bit (if enabled) and D[8] is written into the lock bit (if enabled). The data sub-blocks can be directly written by executing a STA instruction with ASI=0xD for the instruction cache data and ASI=0xF for the data cache data. The sub-block to be read in the indexed cache line and set is selected by A[4:2].

Note that diagnostic access to the cache is not possible during a FLUSH operation and will cause a data exception (trap=0x09) if attempted.

28.5.3 Cache line locking

In a multi-set configuration the instruction and data cache controllers can be configured with optional lock bit in the cache tag. Setting the lock bit prevents the cache line to be replaced by the replacement algorithm. A cache line is locked by performing a diagnostic write to the instruction tag on the cache offset of the line to be locked setting the Address Tag field to the address tag of the line to be locked, setting the lock bit and clearing the valid bits. The locked cache line will be updated on a read-miss and will remain in the cache until the line is unlocked. The first cache line on certain cache offset is locked in the set 0. If several lines on the same cache offset are to be locked the locking is performed on the same cache offset and in sets in ascending order starting with set 0. The last set can not be locked and is always replaceable. Unlocking is performed in descending set order.

NOTE: Setting the lock bit in a cache tag and reading the same tag will show if the cache line locking was enabled during the LEON3 configuration: the lock bit will be set if the cache line locking was enabled otherwise it will be 0.

28.5.4 Local instruction ram

A local instruction ram can optionally be attached to the instruction cache controller. The size of the local instruction is configurable from 1-64 kB. The local instruction ram can be mapped to any 16 Mbyte block of the address space. When executing in the local instruction ram all instruction

fetches are performed from the local instruction ram and will never cause IU pipeline stall or generate an instruction fetch on the AHB bus. Local instruction ram can be accessed through load/store integer word instructions (LD/ST). Only word accesses are allowed, byte, halfword or double word access to the local instruction ram will generate data exception.

28.5.5 Local data ram

A local data ram can optionally be attached to the data cache controller. Data access (load and store instructions) performed to the local data ram and will not be cached in the normal data cache, nor appear on the AHB bus. The ram can be between 1 - 64 kbyte, and mapped on any 16 Mbyte block in the address space. See “Configuration and synthesis” on page 166 for configuration details.

28.5.6 Cache Control Register

The operation of the instruction and data caches is controlled through a common Cache Control Register (CCR) (figure 95). Each cache can be in one of three modes: disabled, enabled and frozen. If disabled, no cache operation is performed and load and store requests are passed directly to the memory controller. If enabled, the cache operates as described above. In the frozen state, the cache is accessed and kept in sync with the main memory as if it was enabled, but no new lines are allocated on read misses.

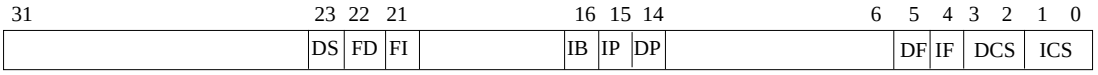

Figure 95. Cache control register

Field Definitions:

[23]: Data cache snoop enable [DS] - if set, will enable data cache snooping.

[22]: Flush data cache (FD). If set, will flush the instruction cache. Always reads as zero.

[21]: Flush Instruction cache (FI). If set, will flush the instruction cache. Always reads as zero.

[16]: Instruction burst fetch (IB). This bit enables burst fill during instruction fetch.

[15]: Instruction cache flush pending (IP). This bit is set when an instruction cache flush operation is in progress.

[14]: Data cache flush pending (DP). This bit is set when an data cache flush operation is in progress.

[5]: Data Cache Freeze on Interrupt (DF) - If set, the data cache will automatically be frozen when an asynchronous interrupt is taken.

[4]: Instruction Cache Freeze on Interrupt (IF) - If set, the instruction cache will automatically be frozen when an asynchronous interrupt is taken.

[3:2]: Data Cache state (DCS) - Indicates the current data cache state according to the following: X0= disabled, 01 = frozen, 11 = enabled.

[1:0]: Instruction Cache state (ICS) - Indicates the current data cache state according to the following: X0= disabled, 01 = frozen, 11 = enabled.

If the DF or IF bit is set, the corresponding cache will be frozen when an asynchronous interrupt is taken. This can be beneficial in real-time system to allow a more accurate calculation of worst-case execution time for a code segment. The execution of the interrupt handler will not evict any cache lines and when control is returned to the interrupted task, the cache state is identical to what it was before the interrupt. If a cache has been frozen by an interrupt, it can only be enabled again by enabling it in the CCR. This is typically done at the end of the interrupt handler before control is returned to the interrupted task.

28.5.7 Cache configuration registers

The configuration of the two caches if defined in two registers: the instruction and data configuration registers. These registers are read-only and indicate the size and configuration of the caches.

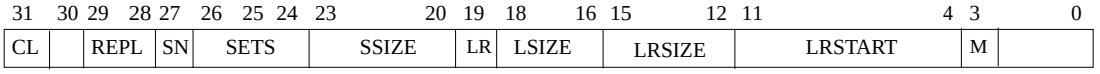

Figure 96. Cache configuration register

[31]: Cache locking (CL). Set if cache locking is implemented.

[29:28]: Cache replacement policy (REPL). 00 - no replacement policy (direct-mapped cache), 01 - least recently used (LRU), 10 - least recently replaced (LRR), 11 - random

[27]: Cache snooping (SN). Set if snooping is implemented.

[26:24]: Cache associativity (SETS). Number of sets in the cache: 000 - direct mapped, 001 - 2-way associative, 010 - 3-way associative, 011 - 4-way associative

[23:20]: Set size (SSIZE). Indicates the size (Kbytes) of each cache set. Size = 2^{SIZE}

[19]: Local ram (LR). Set if local scratch pad ram is implemented.

[18:16]: Line size (LSIZE). Indicated the size (words) of each cache line. Line size = 2^{LSZ}

[15:12]: Local ram size (LRSZ). Indicates the size (Kbytes) of the implemented local scratch pad ram. Local ram size = 2^{LRSZ}

[11:4]: Local ram start address. Indicates the 8 most significant bits of the local ram start address.

[3]: MMU present. This bit is set to ‘1’ if an MMU is present.

All cache registers are accessed through load/store operations to the alternate address space (LDA/STA), using ASI = 2. The table below shows the register addresses:

TABLE 107. ASI 2 (system registers) address map

Address	Register
0x00	Cache control register
0x04	Reserved
0x08	Instruction cache configuration register
0x0C	Data cache configuration register

28.5.8 Software consideration

After reset, the caches are disabled and the cache control register (CCR) is 0. Before the caches may be enabled, a flush operation must be performed to initialized (clear) the tags and valid bits. A suitable assembly sequence could be:

```
flush
set 0x81000f, %g1
st %g1, [%g0] 2
```

28.6 Memory management unit

A memory management unit (MMU) compatible with the SPARC V8 reference MMU can optionally be configured. For details on operation, see the SPARC V8 manual.

28.6.1 ASI mappings

When the MMU is used, the following ASI mappings are added:

ASI	Usage
0x10	Flush page
0x10	MMU flush page
0x13	MMU flush context
0x14	MMU diagnostic dcache context access
0x15	MMU diagnostic icache context access
0x19	MMU registers
0x1C	MMU bypass
0x1D	MMU diagnostic access

TABLE 108. MMU ASI usage

28.6.2 Cache operation

When the MMU is disabled, the caches operate as normal with physical address mapping. When the MMU is enabled, the caches tags store the virtual address and also include an 8-bit context field. AHB cache snooping is not available when the MMU is enabled.

28.6.3 MMU registers

The following MMU registers are implemented:

Address	Register
0x000	MMU control register
0x100	Context pointer register
0x200	Context register
0x300	Fault status register
0x400	Fault address register

TABLE 109. MMU registers (ASI = 0x19)

The definition of the registers can be found in the SPARC V8 manual.

28.6.4 Translation look-aside buffer (TLB)

The MMU can be configured to use a shared TLB, or separate TLB for instructions and data. The number of TLB entries can be set to 2 - 32 in the configuration record. The organisation of the TLB and number of entries is not visible to the software and does thus not require any modification to the operating system.

28.7 Floating-point unit and custom co-processor interface

The SPARC V8 architecture defines two (optional) co-processors: one floating-point unit (FPU) and one user-defined co-processor. The LEON3 pipeline provides an interface port for both of these units. Two different FPU's can be interfaced: Gaisler Research's GRFPU, and the Meiko FPU from Sun. Selection of which FPU to use is done through the VHDL model's generic map. The characteristics of the FPU's are described in the next sections.

28.7.1 Gaisler Research's floating-point unit (GRFPU)

The high-performance GRFPU operates on single- and double-precision operands, and implements all SPARC V8 FPU instructions. The FPU is interfaced to the LEON3 pipeline using a LEON3-specific FPU controller (GRFPC) that allows FPU instructions to be executed simultaneously with integer instructions. Only in case of a data or resource dependency is the integer pipeline held. The GRFPU is fully pipelined and allows the start of one instruction each clock cycle, with the exception is FDIV and FSQRT which can only be executed one at a time. The FDIV and FSQRT are however executed in a separate divide unit and do not block the FPU from performing all other operations in parallel.

All instructions except FDIV and FSQRT has a latency of three cycles, but to improve timing, the LEON3 FPU controller inserts an extra pipeline stage in the result forwarding path. This results in a latency of four clock cycles at instruction level. The table below shows the GRFPU instruction timing when used together with GRFPC:

TABLE 110. GRFPU instruction timing with GRFPC

Instruction	Throughput	Latency
FADDS, FADDD, FSUBS, FSUBD,FMULS, FMULD, FSMULD, FITOS, FITOD, FSTOI, FDTOI, FSTOD, FDTOS, FCMPS, FCMPD, FCMPES. FCMPED	1	4
FDIVS	14	16
FDIVD	15	17
FSQRTS	22	24
FSQRTD	23	25

When the GRFPU is enabled in the model, the version field in %fsr has the value of 1.

The GRFPC controller implements the SPARC deferred trap model, and the FPU trap queue (FQ) can contain up to three queued instructions when an FPU exception is taken.

Note that the GRFPU/GRFPC is not distributed with the open-source LEON model, and must be obtained separately from Gaisler Research.

28.7.2 The Meiko FPU

The Meiko floating-point core operates on both single- and double-precision operands, and implements all SPARC V8 FPU instructions. The Meiko FPU is interfaced through the Meiko FPU controller (MFC), which allows one FPU instruction to execute in parallel with IU operation. The MFC implements the SPARC deferred trap model, and the FPU trap queue (FQ) can contain up to one queued instruction when an FPU exception is taken.

When the Meiko FPU is enabled in the model, the version field in %fsr has the value of 2.

The Meiko FPU is not distributed with the open-source LEON3 model, and must be obtained separately from Sun.

28.7.3 Generic co-processor

LEON can be configured to provide a generic interface to a user-defined co-processor. The interface allows an execution unit to operate in parallel to increase performance. One co-processor instruction can be started each cycle as long as there are no data dependencies. When finished, the result is written back to the co-processor register file.

28.8 Configuration and synthesis

28.8.1 Plug&play configuration

The LEON3 processor is identified with the following vendor and device ID's:

TABLE 111. LEON3 plug&play ID

Vendor ID	Device ID
0x01 (GAISLER)	0x003

28.8.2 Configuration options

The VHDL model of the LEON3 processor can be configured through following VHDL-generics:

TABLE 112. LEON3 VHDL-generics

Generic	Function	Allowed range	Default
hindex	AHB master index	0 - NAHBMST-1	0
fabtech	Target technology	0 - NTECH	0 (inferred)
memtech	Vendor library for regfile and cache RAMs	0 - NTECH	0 (inferred)
nwindows	Number of SPARC register windows. Choose 8 windows to be compatible with Bare-C and RTEMS cross-compilers.	2 - 32	8
dsu	Enable Debug Support Unit interface	0 - 1	0
fpu	Floating-point Unit. 0 - no FPU, 1 - GRFPU, 2 - Meiko.	0 - 2	0
v8	Generate SPARC V8 MUL and DIV instructions	0 - 2	0
cp	Generate co-processor interface	0 -1	0
mac	Generate SPARC V8e SMAC/UMAC instruction	0 - 1	0
pclow	Least significant bit of PC (Program Counter) that is actually generated. PC[1:0] are always zero and are normally not generated. Generating PC[1:0] makes VHDL-debugging easier.	0, 2	2
notag	Currently not used	-	-
nwp	Number of watchpoints	0 - 4	0
irepl	Instruction cache replacement policy. 0 - least recently used (LRU), 1 - least recently replaced (LRR), 2 - random	0 - 1	0
isets	Number of instruction cache sets	1 - 4	1
ilinesize	Instruction cache line size in number of words	4, 8	4
isetsize	Size of each instruction cache set in kByte	1 - 64	1
isetlock	Enable instruction cache line locking	0 - 1	0
drepl	Data cache replacement policy. 0 - least recently used (LRU), 1 - least recently replaced (LRR), 2 - random	0 - 1	0

TABLE 112. LEON3 VHDL-generics

Generic	Function	Allowed range	Default
dsets	Number of data cache sets	1 - 4	1
dlinesize	Data cache line size in number of words	4, 8	4
dsetsize	Size of each data cache set in kByte	1 - 64	1
dsetlock	Enable instruction cache line locking	0 - 1	0
dsnoop	Enable data cache snooping 0: disable, 1: slow, 2: fast (see text)	0 - 2	0
ilram	Enable local instruction RAM	0 - 1	0
ilramsize	Local instruction RAM size in kB	1 - 64	1
ilramstart	8 MSB bits used to decode local instruction RAM area	0 - 255	16#8E#
dlram	Enable local data RAM (scratch-pad RAM)	0 - 1	0
dlramsize	Local data RAM size in kB	1 - 64	1
dlramstart	8 MSB bits used to decode local data RAM area	0 - 255	16#8F#
mmuen	Enable memory management unit (MMU)	0 - 1	0
itlbnum	Number of instruction TLB entries	2 - 64	8
dtlbnum	Number of data TLB entries	2 - 64	8
tlb_type	Separate (0) or shared TLB (1)	0 - 1	1
tlb_rep	Random (0) or LRU (1) TLB replacement	0 - 1	0
lddel	Load delay. One cycle gives best performance, but might create a critical path on targets with slow (data) cache memories. A 2-cycle delay can improve timing but will reduce performance with about 5%.	1 - 2	2
disas	Print instruction disassembly in VHDL simulator console.	0 - 1	0
tbuf	Size of instruction trace buffer in kB (0 - instruction trace disabled)	0 - 64	0
pwd	Power-down. 0 - disabled, 1 - area efficient, 2 - timing efficient.	0 - 2	1
svt	Enable single-vector trapping	0 - 1	0
rstaddr	Default reset start address	0 - (2^{**20}-1)	0

28.8.3 Signal description

TABLE 113. Signal description

Signal name	Field	Type	Function	Active
CLK	N/A	Input	Clock	-
RSTN	N/A	Input	Reset	Low
AHBI	*	Input	AHB master input signals	-
AHBO	*	Output	AHB master output signals	-
AHBSI	*	Input	AHB slave input signals	-
IRQI	IRL[3:0]	Input	Interrupt level	High
IRQO	INTACK	Output	Interrupt acknowledge	High
	IRL[3:0]	Output	Processor interrupt level	High
DBGI	-	Input	Debug inputs from DSU	-
DBGO	-	Output	Debug outputs to DSU	-

* see GRLIB IP Library User's Manual

28.8.4 Library dependencies

Table shows libraries that the LEON3 module depends on.

TABLE 114. Library dependencies

Library	Package	Imported unit(s)	Description
GRLIB	AMBA	Signals	AHB signal definitions
GRLIB	LEON3	Component, signals	LEON3 component declaration, interrupt and debug signals declaration

28.8.5 Model interface

The LEON3 has the following component declaration:

```
entity leon3s is
  generic (
    hindex    : integer               := 0;
    fabtech   : integer range 0 to 7  := 0;
    memtech   : integer range 0 to 7  := 0;
    nwindows  : integer range 2 to 32 := 8;
    dsu       : integer range 0 to 1  := 0;
    fpu       : integer range 0 to 2  := 0;
    v8        : integer range 0 to 2  := 0;
    cp        : integer range 0 to 1  := 0;
    mac       : integer range 0 to 1  := 0;
    pclow     : integer range 0 to 2  := 2;
    notag     : integer range 0 to 1  := 0;
    nwp       : integer range 0 to 4  := 0;
    irepl     : integer range 0 to 2  := 2;
    isets     : integer range 1 to 4  := 1;
    ilinesize : integer range 4 to 8  := 4;
    isetsize  : integer range 1 to 64 := 1;
    isetlock  : integer range 0 to 1  := 0;
    drepl     : integer range 0 to 2  := 2;
    dsets     : integer range 1 to 4  := 1;
    dlinesize : integer range 4 to 8  := 4;
    dsetsize  : integer range 1 to 64 := 1;
    dsetlock  : integer range 0 to 1  := 0;
    dsnoop    : integer range 0 to 2  := 0;
    lram      : integer range 0 to 1  := 0;
    lrambits  : integer range 4 to 14 := 4;
    lramstart : integer range 0 to 255 := 16#40#;
    lddel     : integer range 1 to 2  := 2;
    disas     : integer range 0 to 1  := 0;
    tbuf      : integer range 0 to 64 := 0;
    pwd       : integer range 0 to 2  := 1;
svt       : integer range 0 to 1 := 1;
    rstaddr   : integer := 16#00000#
);
  port (
    clk    : in  std_ulogic;
    rstn   : in  std_ulogic;
    ahbi   : in  ahb_mst_in_type;
    ahbo   : out ahb_mst_out_type;
    ahbsi  : in  ahb_slv_in_type;
    irqi   : in  l3_irq_in_type;
    irqo   : out l3_irq_out_type;
    dbgi   : in  l3_debug_in_type;
    dbgo   : out l3_debug_out_type
  );
end;
```

29 MUL32 - Signed/unsigned 32x32 multiplier module

29.1 Overview

The multiplier module is highly configurable module implementing 32x32 bit multiplier. Multiplier takes two signed or unsigned numbers as input and produces 64-bit result. Multiplication latency and hardware complexity depend on multiplier configuration. Variety of configuration option makes it possible to configure the multiplier to meet wide range of requirements on complexity and performance.

For DSP applications the module can be configured to perform multiply & accumulate (MAC) operation. In this configuration 16x16 multiplication is performed and the 32-bit result is added to 40-bit value accumulator.

29.2 Operation

The multiplication is started when '1' is samples on MULI.START on positive clock edge. Operands are latched externally and provided on inputs MULI.OP1 and MULI.OP2 during the whole operation. The result appears on the outputs during the clock cycle following the clock cycle when MULO.READY is asserted if multiplier if 16x16, 32x8 or 32x16 configuration is used. For 32x32 configuration result appears on the output during the second clock cycle after the MULI.START was asserted.

Signal MULI.MAC shall be asserted to start multiply & accumulate (MAC) operation. This signal is latched on positive clock edge. Multiplication is performed between two 16-bit values on inputs MULI.OP1[15:0] and MULI.OP2[15:0]. The 32-bit result of the multiplication is added to the 40-bit accumulator value on signal MULI.ACC to form a 40-bit value on output MULO.RESULT[39:0]. The result of MAC operation appears during the second clock cycle after the MULI.MAC was asserted.

29.3 Configuration options

The multiplier module is configured through following VHDL-generics:

TABLE 115. Multiplier module configuration options (VHDL-generics)

Generic	Function	Allowed range	Default
infer	If set the multipliers will be inferred by the synthesis tool. Use this option if your synthesis tool i capable of inferring efficient multiplier implementation.	0 to 1	1
multype	Size of the multiplier that is actually implemented. All configuration produce 64-bit result with different latencies. 0 - 16x16 bit multiplier 1 - 32x8 bit multiplier 2 - 32x16 bit multiplier 3 - 32x32 bit multiplier	0 to 3	0
pipe	Used in 16x16 bit multiplier configuration with inferred option enabled. Adds a pipeline register stage to the multiplier. This option gives better timing but adds one clock cycle to latency.	0 to 1	0
mac	Enable multiply & accumulate operation. Use only with 16x16 multiplier option with no pipelining (*pipe* = 0)	0 to 1	0

Table 116 shows hardware complexity in ASIC gates and latency for different multiplier configurations.

TABLE 116. Multiplier latencies and hardware complexity

Multiplier size ***(multype)***	**Pipelined** ***(pipe)***	**Latency (clocks)**	**Approximate area (gates)**
16x16	1	5	6 500
16x16	0	4	6 000
32x8	-	4	5 000
32x16	-	2	9 000
32x32	-	1	15 000

29.4 Signal description

The multiplier module signals are described in table 117.

TABLE 117. Multiplier module signals

Signal name	**Field**	**Type**	**Function**	**Active**
RST	N/A	Input	Reset	Low
CLK	N/A	Input	Clock	-
HOLDN	N/A	Input	Hold	Low
MULI	OP1[32:0]	Input	Operand 1 OP1[32] - Sign bit. OP1[31:0] - Operand 1 in 2's complement format	High
	OP2[32:0]		Operand 2 OP2[32] - Sign bit. OP2[31:0] - Operand 2in 2's complement format	High
	FLUSH		Flush current operation	High
	SIGNED		Signed multiplication	High
	START		Start multiplication	High
	MAC		Multiply & accumulate	High
	ACC[39:0]		Accumulator. Accumulator value is held externally.	High
MULO	READY	Output	Result is ready during the next clock cycle for 16x16, 32x8 and 32x16 configurations. Not used for 32x32 configuration or MAC operation.	High
	NREADY		Not used	-
	ICC[3:0]		Condition codes ICC[3] - Negative result (not used in 32x32 conf) ICC[1] - Zero result (not used in 32x32 conf) ICC[1:0] - Not used	High
	RESULT[63:0]		Result. Available at the end of the clock cycle if MULO.READY was asserted in previous clock cycle. For 32x32 configuration the result is available during second clock cycle after the MULI.START was asserted.	High

29.5 Library dependencies

Table 118 shows libraries required when instantiating the multiplier module.

TABLE 118. Library dependencies

Library	Package	Imported unit(s)	Description
GAISLER	ARITH	Signals, component	Multiplier module signals, component declaration

29.6 Model interface

The multiplier unit has the following component declaration.

```
component mul32
generic (
    infer   : integer := 1;
    multype : integer := 0;
    pipe    : integer := 0;
    mac     : integer := 0
);
port (
    rst     : in  std_ulogic;
    clk     : in  std_ulogic;
    holdn   : in  std_ulogic;
    muli    : in  mul32_in_type;
    mulo    : out mul32_out_type
);
end component;
```

29.7 Example instantiation

The VHDL-code below shows how the multiplier module can be instantiated. The module is configured to implement 16x16 pipelined multiplier with support for MAC operations.

```
library ieee;
use ieee.std_logic_1164.all;

library grlib;
use gaisler.arith.all;

.
.
.

signal muli  : mul32_in_type;
signal mulo  : mul32_out_type;

begin

mul0 : mul32 generic map (infer => 1, multype => 0, pipe => 1, mac => 1)
    port map (rst, clk, holdn, muli, mulo);


end;
```

30 MULTLIB - High-performance multipliers

30.1 Overview

The GRLIB.MULTLIB VHDL-library contains a collection of high-performance multipliers from the Arithmetic Module Generator at Norwegian University of Science and Technology. 32x32, 32x8, 32x16, 16x16 unsigned/signed multipliers are included. 16x16-bit multiplier can be configured to include a pipeline stage. This option improves timing but increases latency with one clock cycle.

30.2 Configuration

16x16 multiplier can be configured to include a pipeline stage through the *mulpipe* VHDL-generic (0 -pipelining disabled, 1 - pipelining enabled).

30.3 Signal description

Multiplier signals are described in table 119.

TABLE 119. Multipliers signals

Signal name	Type	Function	Active
CLK (16x16 multiplier only)	Input	Clock	-
HOLDN (16x16 multiplier only)	Input	Hold. When active, the pipeline register is not updates	Low
X[16:0] (16x16 mult) X[32:0] (32x8 mult) X[32:0] (32x16 mult) X[32:0] (32x32 mult)	Input	Operand 1. MBS bit is sign bit.	High
Y[16:0] (16x16 mult) Y[8:0] (32x8 mult) Y[16:0] (32x16 mult) Y[32:0] (32x32 mult)	Input	Operand 2. MSB bit is sign bit.	High
P[33:0] (16x16 mult) P[41:0] (32x8 mult) P[49:0] (32x16 mult) P[65:0] (32x32 mult)		Result. Two MSB bits are sign bits.	High

30.4 Library dependencies

Table 120 shows libraries required when instantiating the multiplier module.

TABLE 120. Library dependencies

Library	Package	Imported unit	Description
GRLIB	MULTLIB	Component	Multiplier component declarations

30.5 Model interface

Component declaration for 32x32 multiplier is shown below.

```
component mul_33_33
  port (
    x    : in  std_logic_vector(32 downto 0);
    y    : in  std_logic_vector(32 downto 0);
    p    : out std_logic_vector(65 downto 0)
  );
end component;
```

30.6 Example instantiation

The VHDL-code below shows how the multiplier module can be instantiated. The module is configured to implement 16x16 pipelined multiplier with support for MAC operations.

```
library ieee;
use ieee.std_logic_1164.all;

library grlib;
use grlib.multlib.all;

.
.
.

signal op1, op2 : std_logic_vector(32 downto 0);
signal prod : std_logic_vector(65 downto 0);

begin

m0 : mul_33_33
         port map (op1, op2, prod);

end;
```

31 PHY - Ethernet PHY simulation model

31.1 Overview

The Ethernet PHY simulation model is a model of an Ethernet PHY chip which is connected between the physical line and the MAC in Ethernet connections. It receives a bitstream from the MAC and converts it into an analog signal which is driven on the line. This model is loosely based on the Intel LXT971A chip. It does not nearly implement all functionality provided by the real device but merely provides enough functions to make it possible to conduct simple simulations.

31.2 Operation

The PHY simulation model was designed to make it possible to perform simple simulations on the EDCL unit also included in GRLIB. The EDCL uses the Opencores Ethernet MAC for the Ethernet communication and the PHY model provides stimuli for the MAC receiver from a file and also stores output from the MAC in another file. Figure 1 shows a block diagram of a typical connection.

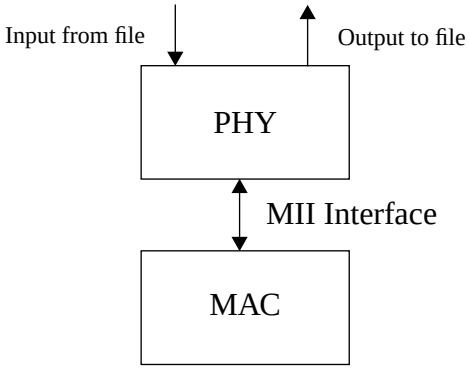

Figure 97. Block diagram of the PHY simulation model connected to a MAC.

The PHY model provides the complete MII interface as defined by the IEEE 802.3 standard with the exception of the management interface. Signals are provided for the management part also but they are currently not used and should be connected to dummy signals or left unconnected were appropriate. Although it was designed to be used with the Opencores MAC in the EDCL, the MII interface should make it possible to connect it to any MAC.

The model can be used in any of the following modes: 10 Mbit half- or full duplex, 100 Mbit half- or full-duplex. The mode is selected with the LEDCFG signals and Table 1 shows the different settings. The rx-clk and tx-clk signals are driven with the correct frequency depending on the selected mode. The other signals are driven in such a way that the follow the 802.3 specification as closely as possible.

The PHY model reads its in-data from a file called ‘indata’ from the current working directory. The data should be stored as ASCII with one nibble in binary format per row. It is read using the VHDL TEXTIO functions. The file should begin with a row containing an invalid bit-vector (anything that cannot be converted to a valid bit-vector by TEXTIO) and all packets should have such a row in-between. This is needed because the PHY inserts a delay between packets and packet boundaries are located by letting the TEXTIO read function set an invalid value parameter to true when an invalid row is found. The delays between packets are currently hard-coded in the design. When a certain number of packets have been sent (set by the win_size generic) a different delay value is used once and then the normal values are used again. This is repeated indefinitely.The output from the MAC is stored in a file called outdata in the current working directory. Data is formatted in the same way as the input file. An example of this formatting is shown in figure 98.

TABLE 1. The led_cfg values used for the different operating modes

LED_CFG	Mode
000	10 Mbit half-duplex
001	10 Mbit full-duplex
010	100 Mbit half-duplex
011	100 Mbit full-duplex

```
start
1101
1001
.
new
1001
0000
.
new
1001
.
```

Figure 98. An example of an indata file layout.

31.3 Configuration options

The PHY model has the following configuration options (VHDL generics):

TABLE 2. PHY model options (generics)

Generic	Function	Allowed range	Default
win_size	Sets the number of packets between each special delay.	all positive integers	3

31.4 Signal descriptions

The PHY model signals are described in table 3.

TABLE 3. PHY model signals

Signal name	Field	Type	Function	Active
RESETN	-	Input	Reset	Low
LED_CFG	-	Input	Configuration signals used to select the operating mode	-
MDIO	-	Input/ Output	Data signal for the management interface (Currently not used)	-
TX_CLK	-	Output	Transmitter clock	-
RX_CLK	-	Output	Receiver clock	-
RXD	-	Output	Receiver data	-
RX_DV	-	Output	Receiver data valid	High
RX_ER	-	Output	Receiver error	High
RX_COL	-	Output	Collision	High
RX_CRS	-	Output	Carrier sense	High
TXD	-	Input	Transmitter data	-
TX_EN	-	Input	Transmitter enable	High
TX_ER	-	Input	Transmitter error	High
MDC	-	Input	Management interface clock (Currently not used)	-

see the IEEE 802.3 standard for a description of how the signals are used.

31.5 Library dependencies

Table 4 shows libraries that should be used when instantiating the PHY model.

TABLE 4. Library dependencies

Library	Package	Imported unit(s)	Description
GAISLER	SIM	Component	Component declaration

31.6 PHY model instantiation

This examples shows how a PHY model module can be instantiated.

```
library ieee;
use ieee.std_logic_1164.all;

library gaisler;
use gaisler.sim.all;

entity phy_ex is
  port (
rst : std_ulogic;
clk : std_ulogic;
    );
end;
```

```
architecture rtl of phy_ex is

  -- Signals

 signal eled_cfg  : std_logic_vector(2 downto 0);
 signal etx_clk   : std_logic;
 signal erx_clk   : std_logic;
 signal erxd      : std_logic_vector(3 downto 0);
 signal erx_dv    : std_logic;
 signal erx_er    : std_logic;
 signal erx_col   : std_logic;
 signal erx_crs   : std_logic;
 signal etxd      : std_logic_vector(3 downto 0);
 signal etx_en    : std_logic;
 signal etx_er    : std_logic;
 signal emdc      : std_logic;

begin

  -- Other components are instantiated here
  ...


  -- PHY model
  phy0 : phy
  generic map (win_size => 8)
  port map(resetn => rst, led_cfg => eled_cfg, mdio => open, tx_clk => etx_clk,
  rx_clk => erx_clk, rxd => erxd, rx_dv => erx_dv, rx_er => erx_er,
  rx_col => erx_col, rx_crs => erx_crs, txd => etxd, tx_en => etx_en,
  tx_er => etx_er, mdc => emdc);
end;
```

32 PCITARGET - Simple 32-bit PCI target with AHB interface

32.1 Overview

This module implements PCI interface with a simple target-only interface. The interface is developed primarily to support DSU communication over the PCI bus. Focus has been put on small area and robust operation, rather than performance. The interface has no FIFOs, limiting the transfer rate to about 5 Mbyte/s. This is however fully sufficient to allow fast download and debugging using the DSU.

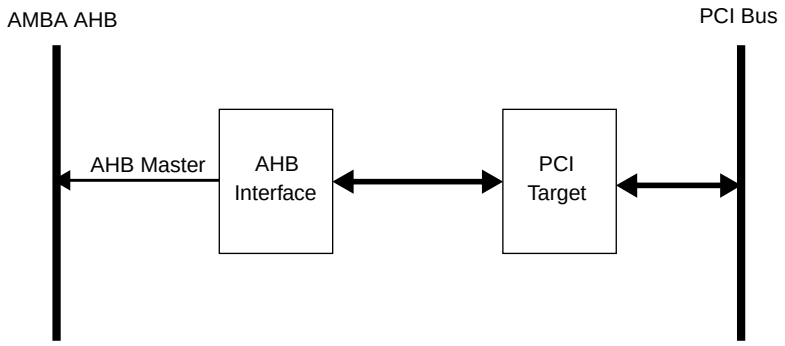

Figure 99. Target-only PCI interface

32.2 Configuration options

The PCI target module has following configuration options (VHDL-generics):

TABLE 5. Simple PCI Target configuration options (VHDL-generics)

Generic	Function	Allowed range	Default
hindex	Selects which AHB select signal (HSEL) will be used to access the PCI target module	0 to NAHBMAX-1	0
abits	Number of bits implemented for PCI memory BAR	0 to 31	21
device_id	PCI device id	0 to 65535	0
vendor_id	PCI vendor id	0 to 65535	0
nsync	One or two synchronization registers between clock regions	1 - 2	1
oepol	Polarity of output enable signals. 0=active low, 1=active high	0 - 1	0

32.3 Vendor and device id

The module has vendor id 0x01 (Gaisler Research) and device id 0x012. For description of vendor and device ids see GRLIB IP Library User's Manual.

32.4 Registers

The module implements one PCI memory BAR.

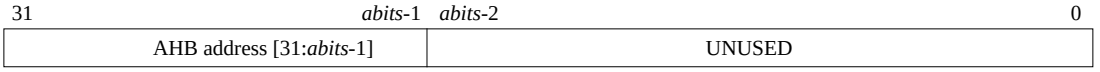

Figure 100. AHB address register (BAR0, 0x100000)

The interface consist of one PCI memory BAR occupying (2^*abits*) bytes (default: 2 Mbyte) of the PCI address space, and an AHB address register. Any access to the lower half of the address space (def.: 0 - 0xFFFFF) will be forwarded to the internal AHB bus. The AHB address will be formed by concatenating the AHB address filed of AHB address register with the LSB bits of the PCI address. An access to the upper half of the address space (default: 1 Mbyte on 0x100000 - 0x1FFFFF) of the BAR will read or write the AHB address register.

32.5 Signal description

PCI Target signals are described in table 6.

TABLE 6. PCI Target signals

Signal name	Field	Type	Function	Active
RST	N/A	Input	Reset	Low
CLK	N/A	Input	AHB system clock	-
PCICLK	N/A	Input	PCI clock	-
PCII	*1	Input	PCI input signals	-
PCIO	*1	Output	PCI output signals	-
APBI	*2	Input	APB slave input signals	-
APBO	*2	Output	APB slave output signals	-

*1) see PCI specification
*2) see GRLIB IP Library User's Manual

The PCIO record contains an additional output enable signal vaden. It is has the same value as aden at each index but they are all driven from separate registers. A directive is placed on this vector so that the registers will not be removed during synthesis. This output enable vector can be used instead of aden if output delay is an issue in the design.

32.6 Library dependencies

Table 7 shows required libraries when instantiating PCI Target module.

TABLE 7. Library dependencies

Library	Package	Imported unit(s)	Description
GRLIB	AMBA	Signals	AMBA signal definitions
GAISLER	PCI	Signals, component	PCI signals and component declaration

33 PCIDMA - DMA Controller for the GRPCI interface

33.1 Introduction

The DMA controller is an add-on interface to the GRPCI interface. This controller perform bursts to or from PCI bus using the master interface of GR PCI Master/target unit. Figure 1 below illustrates how the DMA controller is attached between the AHB bus and the PCI master interface.

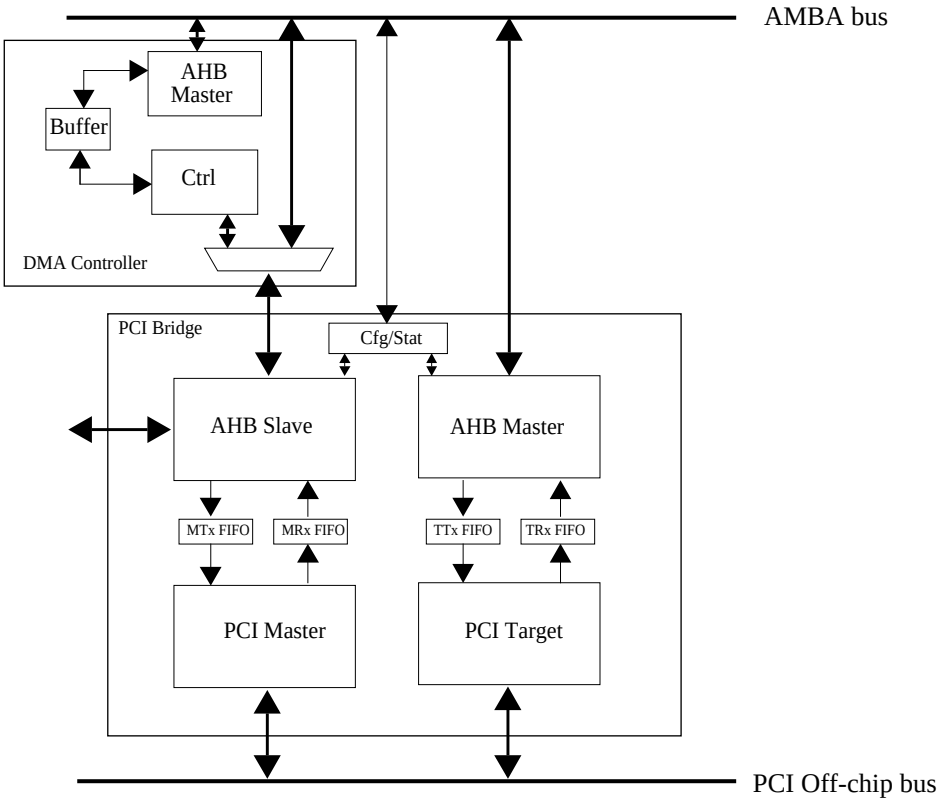

Figure 101. DMA Controller unit

33.2 Operation

The DMA controller is set up by defining the location of memory areas between which the DMA will take place in both PCI and AHB address space as well as direction, length and type of the transfer. Only 32-bit word transfer are supported.

The DMA transfer is automatically aborted when any kind of error is detected during a transfer. The DMA controller does not detect deadlocks in its communication channels. If the system concludes that a deadlock has occurred, it can manually abort the DMA transfer.

When the DMA is not active the AHB slave interface of PCI Master/Target unit will be directly connected to AMBA AHB bus.

33.3 Configuration options

The PCI Target / Master unit has the following configuration options (VHDL generics):

TABLE 8. DMA Controller options (generics)

Generic	Function	Allowed range	Default
mstndx	DMA Controllers AHB Master interface index	0 - NAHBMST-1	0
apbndx	The AMBA APB index for the configuration/status APB interface	0 - NAPBMAX-1	0
apbaddr	APB interface base address	0 - 16#FFF#	0
apbmask	APB interface address mask	0 - 16#FFF#	16#FFF#
blength	Burst length	-	4

33.4 Vendor and device id

The module has vendor id 0x01 (Gaisler Research) and device id 0x016. For description of vendor and device ids see GRLIB IP Library User's Manual.

33.5 Registers

Following registers are mapped into APB address space:

TABLE 9. DMA Controller registers

Address offset	Register
0x00	Command/status register
0x04	AMBA Target Address
0x08	PCI Target Address
0x0C	Burst length

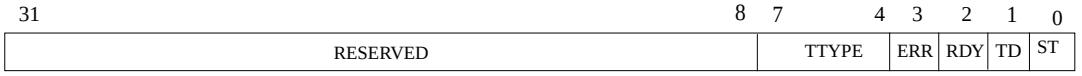

Figure 102. Status/Command register

[31:8]: Reserved.

[7:4]: Transfer Type (TTYPE) - Perform either PCI Memory or I/O cycles. "1000" - memory cycles, "0100" - I/O cycles. This value drives directly HMBSEL signals on PCI Master/Targets units AHB Slave interface.

[3]: Error (ERR) - Last transfer was abnormally terminated. If set by the DMA Controller this bit will remain zero until cleared by writing '1' to it.

[2]: Ready (RDY) - Current transfer is completed. When set by the DMA Controller this bit will remain zero until cleared by writing '1' to it.

[1]: Transfer Direction (TD) - '1' - write to PCI, '0' - read form PCI.

[0]: Start (ST) - Start DMA transfer. Writing '1' will start the DMA transfer. All other registers have to be set up before setting this bit.

Set by the PCI Master interface when its transaction is terminated with Target-Abort. Writing '1'

Figure 103. AMBA Target Address

[31:0]: AMAB Target Address (ATA) - AHB start address for the data on AMBA bus. In case of error, it indicated failing address.

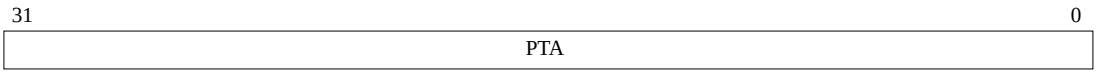

Figure 104. PCI Target Address

[31:0]: PCI Target Address (PTA) - PCI start address on PCI bus. This is a complete 32-bit PCI address and is not further mapped by the PCI Master/Target unit. In case of error, it indicated failing address.

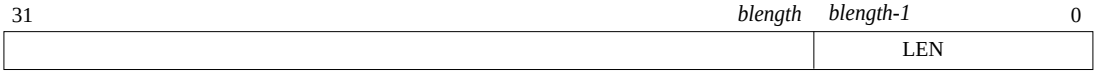

Figure 105. Length register

[*blentgh-1*:0]: DMA Transfer Length (LEN) - Number of 32-bit words to be transferred.

33.6 Signal description

DMA controller unit signals are described in table 10.

TABLE 10. PCI Target/Master unit signals

Signal name	Field	Type	Function	Active
RST	N/A	Input	Reset	Low
CLK	N/A	Input	AMBA system clock	-
PCICLK	N/A	Input	PCI clock	-
APBI	*	Input	APB slave input signals	-
APBO	*	Output	APB slave output signals	-
AHBMI	*	Input	AHB master input signals	-
AHBMO	*	Output	AHB master output signals	-
AHBSI0	*	Input	AHB slave input signals, main AHB bus	-
AHBSO0	*	Output	AHB slave output signals, main AHB bus	-
AHBSI1	*	Input	AHB slave input signals, connected to PCI Target/Master unit	-
AHBSO1	*	Output	AHB slave output signals, connected to PCI Target/Master unit	-

* see GRLIB IP Library User's Manual

33.7 Library dependencies

Table 11 shows libraries that should be used when instantiating the DMA controller.

TABLE 11. Library dependencies

Library	Package	Imported unit(s)	Description
GRLIB	AMBA	Signals	AMBA signal definitions
GAISLER	PCI	Component	Component declaration

33.8 Example instantiation

```
library ieee;
use ieee.std_logic_1164.all;
library grlib;
use grlib.amba.all;
use grlib.tech.all;
library gaisler;
use gaisler.pci.all;
use gaisler.pads.all;

signal pcii : pci_in_type;
signal pcio : pci_out_type;

dma : pcidma generic map (memtech => memtech, dmstndx => 1,
  dapbndx => 5, dapbaddr => 5, blength => blength, mstndx => 0,
  fifodepth => log2(fifodepth), device_id => CFG_PCIDID, vendor_id => CFG_PCIVID,
  slvndx => 4, apbndx => 4, apbaddr => 4, haddr => 16#E00#, ioaddr => 16#800#,
  nsync => 1)
port map (rstn, clkm, pciclk, pcii, pcio, apbo(5), ahbmo(1),
  apbi, apbo(4), ahbmi, ahbmo(0), ahbsi, ahbso(4));


pcipads0 : pcipads generic map (padtech => padtech)
port map ( pci_rst, pci_gnt, pci_idsel, pci_lock, pci_ad, pci_cbe,
      pci_frame, pci_irdy, pci_trdy, pci_devsel, pci_stop, pci_perr,
      pci_par, pci_req, pci_serr, pci_host, pci_66, pcii, pcio );
```

34 REGFILE_3P 3-port RAM generator (2 read, 1 write)

34.1 Operation

The 3-port register file has two read ports and one write port. Each port has a separate address and data bus. All inputs are latched on the rising edge of clk. The read data appears on dataout directly after the clk rising edge. Note: on most technologies, the register file is implemented with two 2-port RAMs with combined write ports. Address width, data width and target technology is parametrizable through generics.

Write-through is supported if the function *syncram_2p_write_through(tech)* returns 1 for the target technology.

34.2 Component declaration

```
library grlib;
use grlib.tech.all;
library gaisler;
use gaisler.memory.all;

component regfile_3p
  generic (tech : integer := 0; abits : integer := 6; dbits : integer := 8;
           wrfst : integer := 0; numregs : integer := 64);
  port (
    wclk   : in  std_ulogic;
    waddr  : in  std_logic_vector((abits -1) downto 0);
    wdata  : in  std_logic_vector((dbits -1) downto 0);
    we     : in  std_ulogic;
    rclk   : in  std_ulogic;
    raddr1 : in  std_logic_vector((abits -1) downto 0);
    re1    : in  std_ulogic;
    rdata1 : out std_logic_vector((dbits -1) downto 0);
    raddr2 : in  std_logic_vector((abits -1) downto 0);
    re2    : in  std_ulogic;
    rdata2 : out std_logic_vector((dbits -1) downto 0)
  );
  end component;
```

34.3 Signals

TABLE 12. REGFILE_3P signals

Signal name	Type	Function	Active
WCLK	Input	Write port clock	
WADDR	Input	Write address	
WDATA	Input	Write data	
WE	Input	Write enable	High
RCLK	Input	Read ports clock	-
RADDR1	Input	Read port1 address	-
RE1	Input	Read port1 enable	High
RDATA1	Output	Read port1 data	-
RADDR2	Input	Read port2 address	-
RE2	Input	Read port2 enable	High
RDATA2	Output	Read port2 data	-

34.4 Parameters (generics)

TABLE 13. VHDL generics

Name	Function	Range	Default
tech	Technology selection	0 - NTECH	0
abits	Address bits. Depth of RAM is $2^{abits-1}$	see table below	-
dbits	Data width	see table below	-
wrfst	Write-first (write-through). Only applicable to inferred technology	0 - 1	0
numregs	Not used		

TABLE 14. Supported technologies

Tech name	Technology	RAM cell	abit range	dbit range
ihp25	IHP 0.25	flip-flops	unlimited	unlimited
inferred	Behavioural description	synthesis tool dependent		
rhumc	Rad-hard UMC 0.18	flip-flops	unlimited	unlimited
virtex	Xilinx Virtex, Virtex-E, Spartan-2	RAMB4_Sn	2 - 10	unlimited
virtex2	Xilinx Virtex2, Spartan3, Virtex4	RAMB16_Sn	2 - 14	unlimited
proasic3	Actel Proasic3	ram4k9	2 - 12	unlimited
memvirage	Virage ASIC RAM	hdss2_64x32cm4sw0 hdss2_128x32cm4sw0 hdss2_256x32cm4sw0 hdss2_512x32cm4sw0	6 - 9	32

35 SDCTRL - 32/64-bit SDRAM PC133 SDRAM Controller

35.1 Overview

The SDRAM controller handles PC133 SDRAM compatible memory devices attached to 32 or 64 bit wide data bus. The controller acts as a slave on the AHB bus where it occupies configurable amount of address space for SDRAM access. The SDRAM controller function is programmed by writing to a configuration register mapped into AHB I/O address space.

Chip-select decoding is provided for two SDRAM banks.

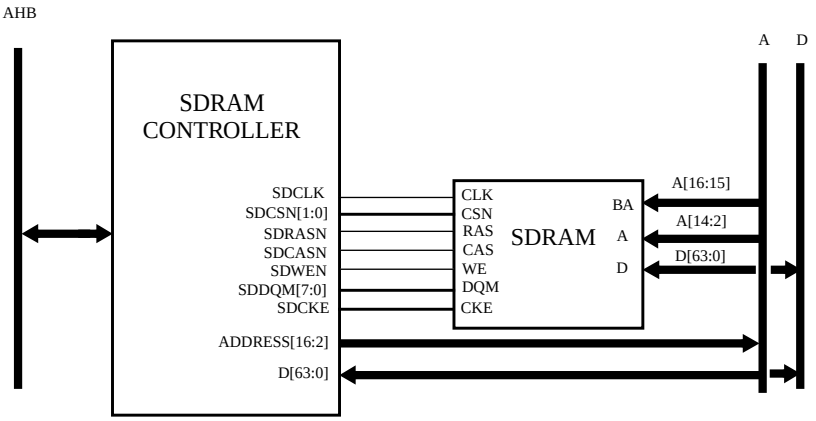

Figure 106. SDRAM Memory controller conected to AMBA bus and SDRAM

35.2 Operation

35.2.1 General

Synchronous dynamic RAM (SDRAM) access is supported to two banks of PC100/PC133 compatible devices. The controller supports 64M, 256M and 512M device with 8 - 12 column-address bits, up to 13 row-address bits, and 4 banks. The size of each of the two banks can be programmed in binary steps between 4 Mbyte and 512 Mbyte. The operation of the SDRAM controller is controlled through the configuration register SDCFG (see chapter 35.4). SDRAM banks data bus width is configurable between 32 and 64 bits.

35.2.2 Initialization

When the SDRAM controller is enabled, it automatically performs the SDRAM initialization sequence of PRECHARGE, 2x AUTO-REFRESH and LOAD-MODE-REG on both banks simultaneously. The controller programs the SDRAM to use page burst on read and single location access on write. If the **pwron** generic is 1, the initialization sequence is also sent automatically when reset is released. Note that some SDRAM require a stable clock of 100 us before any commands might be sent. When using on-chip PLL, this might not always be the case and the **pwron** generic should be set to 0 in such cases.

35.2.3 Configurable SDRAM timing parameters

To provide optimum access cycles for different SDRAM devices (and at different frequencies), some SDRAM parameters can be programmed through SDRAM configuration register (SDCFG) The programmable SDRAM parameters can be seen in table below:

TABLE 15. SDRAM programmable timing parameters

Function	Parameter	range	unit
CAS latency, RAS/CAS delay	t_{CAS}, t_{RCD}	2 - 3	clocks
Precharge to activate	t_{RP}	2 - 3	clocks
Auto-refresh command period	t_{RFC}	3 - 11	clocks
Auto-refresh interval		10 - 32768	clocks

Remaining SDRAM timing parameters are according the PC100/PC133 specification.

35.2.4 Refresh

The SDRAM controller contains a refresh function that periodically issues an AUTO-REFRESH command to both SDRAM banks. The period between the commands (in clock periods) is programmed in the refresh counter reload field in the SDCFG register. Depending on SDRAM type, the required period is typically 7.8 or 15.6 µs (corresponding to 780 or 1560 clocks at 100 MHz). The generated refresh period is calculated as (reload value+1)/sysclk. The refresh function is enabled by setting bit 31 in SDCFG register.

35.2.5 SDRAM commands

The controller can issue three SDRAM commands by writing to the SDRAM command field in SDCFG: PRE-CHARGE, AUTO-REFRESH and LOAD-MODE-REG (LMR). If the LMR command is issued, the CAS delay as programmed in SDCFG will be used, remaining fields are fixed: page read burst, single location write, sequential burst. The command field will be cleared after a command has been executed. Note that when changing the value of the CAS delay, a LOAD-MODE-REGISTER command should be generated at the same time.

35.2.6 Read cycles

A read cycle is started by performing an ACTIVATE command to the desired bank and row, followed by a READ command after the programmed CAS delay. A read burst is performed if a burst access has been requested on the AHB bus. The read cycle is terminated with a PRE-CHARGE command, no banks are left open between two accesses. Note that only word bursts are supported by the SDRAM controller. The AHB bus supports bursts of different sizes such as bytes and half-words but they cannot be used.

35.2.7 Write cycles

Write cycles are performed similarly to read cycles, with the difference that WRITE commands are issued after activation. A write burst on the AHB bus will generate a burst of write commands without idle cycles in-between. As in the read case, only word bursts are supported.

35.2.8 Address bus connection

The SDRAM address bus should be connected to SA[12:0], the bank address to SA[14:13], and the data bus to SD[31:0] or SD[63:0] if 64-bit data bus is used.

35.2.9 Data bus

The external SDRAM data bus is configurable to either 32 or 64 bits width, using the **sdbits** generic. 64-bit data bus allows 64-bit (SO)DIMS to be connected using the full data capacity of the devices. The polarity of the output enable signal to the data pads can be selected with the oepol generic. Sometimes it is difficult to fulfil the output delay requirements of the output enable signal. In this case, the vbdrive signal can be used instead of bdrive. Each index in this

vector is driven by a separate register and a directive is placed on them so that they will not be removed by the synthesis tool.

35.2.10 Clocking

The external SDRAM clock must be in phase with the internal (AHB) clock, and typically requires special synchronization. For Xilinx and Altera FPGA targets, GR Clock Generator can be configured to produce a properly synchronized SDRAM clock. For boards where the SDRAM clock can not be synchronized, the GR Clock Generator can produce an inverted clock. If this scheme is used, the **`invclk`** generic should be set to 1 in order to adjust the SDRAM state machine.

35.2.11 Configuration options

The SDRAM controller has the following configuration options (VHDL generics):

TABLE 16. SDRAM controller configuration options (VHDL generics)

Generic	Function	Allowed range	Default
hindex	AHB slave index	1 - NAHBSLV-1	0
haddr	ADDR filed of the AHB BAR0 defining SDRAM area. Default is 0xF0000000 - 0xFFFFFFFF.	0 - 16#FFF#	16#000#
hmask	MASK filed of the AHB BAR0 defining SDRAM area.	0 - 16#FFF#	16#F00#
ioaddr	ADDR filed of the AHB BAR1 defining I/O address space where SDCFG register is mapped.	0 - 16#FFF#	16#000#
iomask	MASK filed of the AHB BAR1 defining I/O address space.	0 - 16#FFF#	16#FFF#
wprot	Write protection.	0 - 1	0
invclk	Inverted clock is used for the SDRAM.	0 - 1	0
pwron	Enable SDRAM at power-on initialization	0 - 1	0
sdbits	32 or 64-bit data bus width.	32, 64	32
oepol	Polarity of bdrive and vbdrive signals. 0=active low, 1=active high	0 - 1	0

35.3 Vendor and device id

The module has vendor id 0x01 (Gaisler Research) and device id 0x009. For description of vendor and device ids see GRLIB IP Library User's Manual.

35.4 Registers

The memory controller is programmed through the SDRAM controller configuration register mapped into AHB I/O space defined by the controllers AHB BAR1.

35.4.1 SDRAM configuration register (SDCFG)

SDRAM configuration register is used to control the timing of the SDRAM.

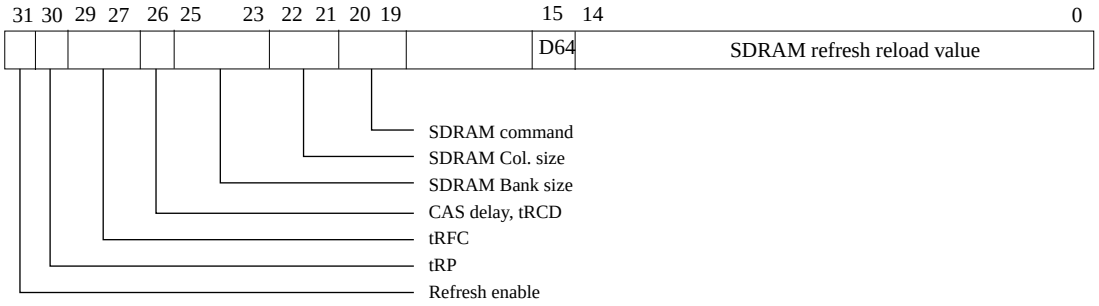

Figure 107. SDRAM configuration register

[14:0]: The period between each AUTO-REFRESH command - Calculated as follows:$t_{REFRESH}$ = ((reload value) + 1) / SYSCLK

[15]: 64-bit data bus (D64) - Reads ‘1’ if memory controller is configured for 64-bit data bus, otherwise ‘0’. Read-only.

[20:19] SDRAM command. Writing a non-zero value will generate an SDRAM command: “01”=PRECHARGE, “10”=AUTO-REFRESH, “11”=LOAD-COMMAND-REGISTER. The field is reset after command has been executed.

[22:21]: SDRAM column size. “00”=256, “01”=512, “10”=1024, “11”=4096 when bit[25:23]= “111”, 2048 otherwise.

[25:23]: SDRAM banks size. Defines the banks size for SDRAM chip selects: “000”=4 Mbyte, “001”=8 Mbyte, “010”=16 Mbyte .... “111”=512 Mbyte.

[26]: SDRAM CAS delay. Selects 2 or 3 cycle CAS delay (0/1). When changed, a LOAD-COMMAND-REGISTER command must be issued at the same time. Also sets RAS/CAS delay (tRCD).

[29:27]: SDRAM t_{RFC} timing. t_{RFC} will be equal to 3 + field-value system clocks.

[30]: SDRAM t_{RP} timing. t_{RP} will be equal to 2 or 3 system clocks (0/1).

[31]: SDRAM refresh. If set, the SDRAM refresh will be enabled.

35.5 Signal description

Signal name	Field	Type	Function	Active
CLK	N/A	Input	Clock	-
RST	N/A	Input	Reset	Low
AHBSI	1)	Input	AHB slave input signals	-
AHBSO	1)	Output	AHB slave output signals	-
SDI	WPROT	Input	Not used	-
	DATA[63:0]	Input	Data	High
SDO	SDCKE[1:0]	Output	SDRAM clock enable	High
	SDCSN[1:0]	Output	SDRAM chip select	Low
	SDWEN	Output	SDRAM write enable	Low
	RASN	Output	SDRAM row address strobe	Low
	CASN	Output	SDRAM column address strobe	Low
	DQM[7:0]	Output	SDRAM data mask	Low
	BDRIVE	Output	Drive SDRAM data bus	Low/High[2]
	VBDRIVE[31:0]	Output	Identical to BDRIVE but has one signal for each data bit. Every index is driven by its own register. This can be used to reduce the output delay.	Low/High[2]
	ADDRESS[16:2]	Output	SDRAM address	Low
	DATA[31:0]	Output	SDRAM data	Low

1) see GRLIB IP Library User's Manual 2) Polarity selected with the oepol generic

35.6 Library dependencies

Table shows libraries that the memory controller module depends on.

TABLE 17. Library dependencies

Library	Package	Imported unit(s)	Description
GRLIB	AMBA	Signals	AHB signal definitions
GAISLER	MEMCTRL	Signals, component	Memory bus signals definitions, component declaration

35.7 Memory controller instantiation

This examples shows how the SDRAM controller can be instantiated. The example design contains an AMBA bus with a number of AHB components connected to it including the SDRAM controller. The external SDRAM bus is defined on the example designs port map and connected to the SDRAM controller. System clock and reset are generated by GR Clock Generator and Reset Generator.

SDRAM controller decodes SDRAM area:0x60000000 - 0x6FFFFFFF. SDRAM Configuration register is mapped into AHB I/O space on address (AHB I/O base address + 0x100).

```
library ieee;
use ieee.std_logic_1164.all;
library grlib;
use grlib.amba.all;
use grlib.tech.all;
library gaisler;
use gaisler.memctrl.all;
use gaisler.pads.all;   -- used for I/O pads
use gaisler.misc.all;

entity mctrl_ex is
  port (
    clk : in std_ulogic;
    resetn : in std_ulogic;
    pllref : in  std_ulogic;
   sdcke    : out std_logic_vector ( 1 downto 0);  -- clk en
    sdcsn    : out std_logic_vector ( 1 downto 0);  -- chip sel
    sdwen    : out std_logic;                       -- write en
    sdrasn   : out std_logic;                       -- row addr stb
    sdcasn   : out std_logic;                       -- col addr stb
    sddqm    : out std_logic_vector (7 downto 0);  -- data i/o mask
    sdclk    : out std_logic;                       -- sdram clk output
    sa       : out std_logic_vector(14 downto 0); -- optional sdram address
    sd       : inout std_logic_vector(63 downto 0) -- optional sdram data
    );
end;

architecture rtl of mctrl_ex is

  -- AMBA bus (AHB and APB)
  signal apbi  : apb_slv_in_type;
  signal apbo  : apb_slv_out_vector := (others => apb_none);
  signal ahbsi : ahb_slv_in_type;
  signal ahbso : ahb_slv_out_vector := (others => ahbs_none);
  signal ahbmi : ahb_mst_in_type;
  signal ahbmo : ahb_mst_out_vector := (others => ahbm_none);

 signal sdi   : sdctrl_in_type;
  signal sdo   : sdctrl_out_type;

  signal clkm, rstn : std_ulogic;
 signal cgi : clkgen_in_type;
  signal cgo : clkgen_out_type;
  signal gnd : std_ulogic;

begin

  -- Clock and reset generators
  clkgen0 : clkgen generic map (clk_mul => 2, clk_div => 2, sdramen => 1,
                                tech => virtex2, sdinvclk => 0)
  port map (clk, gnd, clkm, open, open, sdclk, open, cgi, cgo);

  cgi.pllctrl <= "00"; cgi.pllrst <= resetn; cgi.pllref <= pllref;

  rst0 : rstgen
  port map (resetn, clkm, cgo.clklock, rstn);

  -- SDRAM controller
  sdc : sdctrl generic map (hindex => 3, haddr => 16#600#, hmask => 16#F00#,
    ioaddr => 1, pwron => 0, invclk => 0)
    port map (rstn, clkm, ahbsi, ahbso(3), sdi, sdo);

  -- input signals
  sdi.data(31 downto 0) <= sd(31 downto 0);

  -- connect SDRAM controller outputs to entity output signals
  sa <= sdo.address; sdcke <= sdo.sdcke; sdwen <= sdo.sdwen;
```

```
  sdcsn <= sdo.sdcsn; sdrasn <= sdo.rasn; sdcasn <= sdo.casn;
  sddqm <= sdo.dqm;

--Data pad instantiation with scalar bdrive
  sd_pad : iopadv generic map (width => 32)
    port map (sd(31 downto 0), sdo.data, sdo.bdrive, sdi.data(31 downto 0));
end;

--Alternative data pad instantiation with vectored bdrive
sd_pad : iopadvv generic map (width => 32)
   port map (sd(31 downto 0), sdo.data, sdo.vbdrive, sdi.data(31 downto 0));
end;
```

36 SRCTRL- 8/32-bit PROM/SRAM Controller

36.1 Overview

SRCTRL is an 8/32-bit PROM/SRAM/IO controller that interfaces external asynchronous SRAM, PROM and I/O to the AMBA AHB bus. The controller can handle 32-bit wide SRAM and I/O, and either 8- or 32-bit PROM.

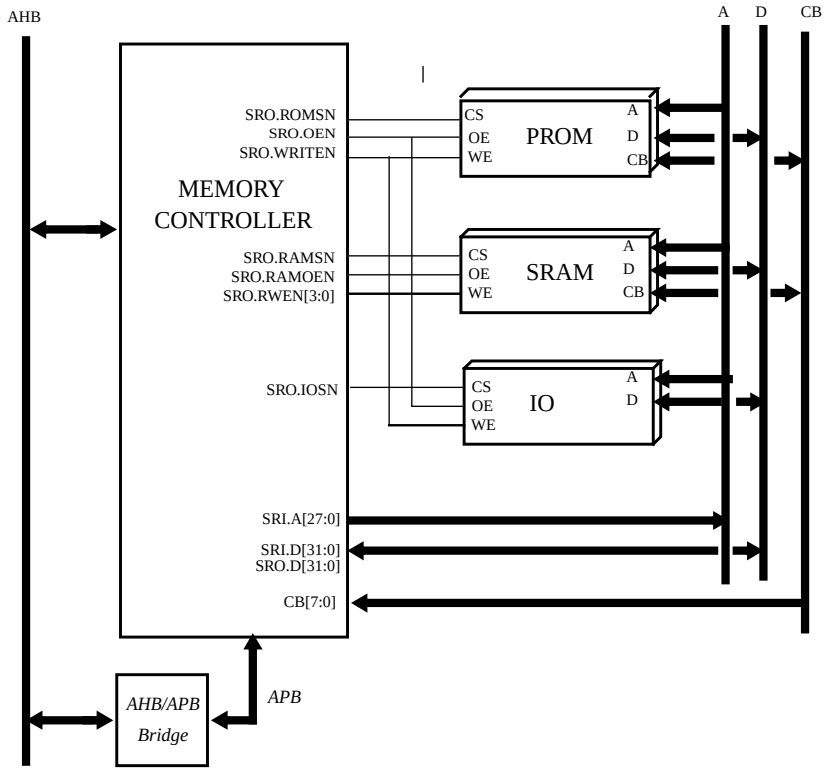

Figure 108. 8/32-bit PROM/SRAM/IO controller

The controller is configured through VHDL-generics to decode three address ranges: PROM, SRAM and I/O area. By default PROM area is mapped into address range 0x0 - 0x00FFFFFF, the SRAM area is mapped into address range 0x40000000 - 0x40FFFFFF, and the I/O area is mapped to 0x20000000 - 0x20FFFFFF.

One chip select is decoded for the I/O area, while SRAM and PROM can have up to four and two select signals respectively. The controller generates both a common write-enable signal (WRITEN) as well as four byte-write enable signals (WREN). If the SRAM uses a common write enable signal the controller can be configured to perform read-modify-write cycles for byte and half-word write accesses. Number of waitstates is separately configurable for the three address ranges.

A single write-enable signal is generated for the PROM area (WRITEN), while four byte-write enable signals (RWEN[3:0]) are provided for the SRAM area. If the external SRAM uses common write enable signal, the controller can be configured to perform read-modify-write cycles for byte and half-word write accesses.

Number of waitstates is configurable through VHDL generics for both PROM and SRAM areas.

A signal (BDRIVE) is provided for enabling the bidirectional pads to which the data signals are connected. The oepol generic is used for selecting the polarity of these enable signals. If output

delay is an issue, a vectored output enable signal (VBDRIVE) can be used instead. In this case, each pad has its own enable signal driven by a separate register. A directive is placed on these registers so that they will not be removed during synthesis (if the output they drive is used in the design).

36.2 8-bit PROM access

The SRCTRL controller can be configured to access a 8-bit wide PROM. The data bus of external PROM should be connected to the upper byte of the 32-bit data bus, i.e. D[31:24]. The 8-bit mode is enabled with the prom8en VHDL generic. When enabled, read accesses to the PROM area will be done in four-byte bursts. The whole 32-bit word is then presented on the AHB data bus. Writes should be done one byte at a time and the byte should always be driven on bit 31-24 on the AHB data bus independent of the byte address.

It is possible to dynamically switch between 8- and 32-bit PROM mode using the BWIDTH[1:0] input signal. When BWIDTH is “00” then 8-bit mode is selected. If BWIDTH is “10” then 32-bit mode is selected. Other BWIDTH values are reserved for future use.

SRAM access is not affected by the 8-bit PROM mode.

36.3 PROM/SRAM waveform

Read accesses to 32-bit PROM and RAM has the same timing, see figure below.

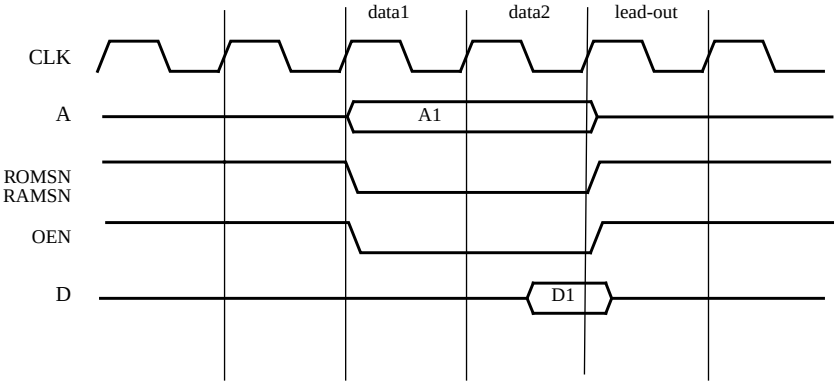

Figure 109. 32-bit PROM/SRAM/IO read cycle

The write access for 32-bit PROM and RAM can be seen below.

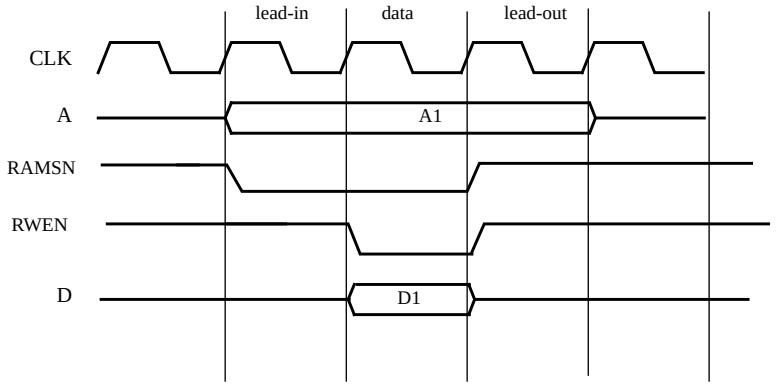

Figure 110. 32-bit PROM/SRAM/IO write cycle

If waitstates are configured through the VHDL generics, one extra data cycle will be inserted for each waitstate in both read and write cycles.

36.4 Burst cycles

To improve the bandwidth of the memory bus, accesses to consecutive addresses can be performed in burst mode. Burst transfers will be generated when the memory controller is accessed using an AHB burst request. These includes instruction cache-line fills and burst from DMA masters. The timing of a burst cycle is identical to the programmed basic cycle with the exception that during read cycles, the lead-out cycle will only occurs after the last transfer.

36.5 Component declaration

```
component srctrl
  generic (
    hindex  : integer := 0;
    romaddr : integer := 0;
    rommask : integer := 16#ff0#;
    ramaddr : integer := 16#400#;
    rammask : integer := 16#ff0#;;
    ioaddr  : integer := 16#200#;
    iomask  : integer := 16#ff0#;
    ramws   : integer := 0;
    romws   : integer := 2;
    iows    : integer := 2;
    rmw     : integer := 0;-- read-modify-write enable
    prom8en : integer := 0;
    oepol   : integer := 0;
    srbanks : integer range 1 to 5 := 1;
    banksz  : integer range 0 to 13:= 13;
    romasel : integer range 0 to 27:= 19
  );
  port (
    rst     : in  std_ulogic;
    clk     : in  std_ulogic;
    ahbsi   : in  ahb_slv_in_type;
    ahbso   : out ahb_slv_out_type;
    sri     : in  memory_in_type;
    sro     : out memory_out_type;
    sdo     : out sdctrl_out_type
  );
end component;
```

36.6 Configuration options

The Memory controller has the following configuration options (VHDL generics):

TABLE 18. Simple 32-bit PROM/SRAM controller configuration options (VHDL generics)

Generic	Function	Allowed range	Default
hindex	AHB slave index	1 - NAHBSLV-1	0
romaddr	ADDR filed of the AHB BAR0 defining PROM address space. Default PROM area is 0x0 - 0xFFFFFF.	0 - 16#FFF#	16#000#
rommask	MASK filed of the AHB BAR0 defining PROM address space.	0 - 16#FFF#	16#FF0#
ramaddr	ADDR filed of the AHB BAR1 defining RAM address space. Default RAM area is 0x40000000-0x40FFFFFF.	0 - 16#FFF#	16#400#
rammask	MASK filed of the AHB BAR1 defining RAM address space.	0 -16#FFF#	16#FF0#
ioaddr	ADDR filed of the AHB BAR2 defining IO address space. Default IO area is 0x20000000-0x20FFFFFF.	0 - 16#FFF#	16#200#
iomask	MASK filed of the AHB BAR2 defining IO address space.	0 -16#FFF#	16#FF0#
ramws	Number of waitstates during access to SRAM area	0 - 15	0
romws	Number of waitstates during access to PROM area	0 - 15	2
iows	Number of waitstates during access to IO area	0 - 15	2
rmw	Enable read-modify-write cycles.	0 - 1	0
prom8en	Enable 8 - bit PROM accesses	0 - 1	0
oepol	Polarity of bdrive and vbdrive signals. 0=active low, 1=active high	0 - 1	0
srbanks	Set the number of RAM banks	1 - 5	1
banksz	Set the size of bank 1 - 4. 0 = 8 Kbyte, 1 = 16 Kbyte, ... , 13 = 64Mbyte.	0 - 13	13
romasel	address bit used for ROM chip select.	0 - 27	19

36.7 Vendor and device id

The module has vendor id 0x01 (Gaisler Research) and device id 0x008. For description of vendor and device ids see GRLIB IP Library User's Manual.

36.8 Registers

The module does not implement any use programmable registers. All configuration is done through the VHDL-generics.

36.9 Signal description

Memory controller signals are described in table 19.

TABLE 19. Memory controller signal description.

Signal name	Field	Type	Function	Polarity
CLK	N/A	Input	Clock	-
RST	N/A	Input	Reset	Low
SRI	DATA[31:0]	Input	Memory data	High
	BRDYN	Input	Not used	-
	BEXCN	Input	Not used	-
	WRN[3:0]	Input	Not used	-
	BWIDTH[1:0]	Input	BWIDTH=”00” => 8-bit PROM mode BWIDTH=”10” => 32-bit PROM mode	-
	SD[31:0]	Input	Not used	-
SRO	ADDRESS[27:0]	Output	Memory address	High
	DATA[31:0]	Output	Memory data	High
	RAMSN[4:0]	Output	SRAM chip-select	Low
	RAMOEN[4:0]	Output	SRAM output enable	Low
	IOSN	Output	Not used. Driven to ‘1’ (inactive)	Low
	ROMSN[1:0]	Output	PROM chip-select	Low
	OEN	Output	Output enable	Low
	WRITEN	Output	Write strobe	Low
	WRN[3:0]	Output	SRAM write enable	Low
	MBEN[3:0]	Output	Byte enable	Low
	BDRIVE[3:0]	Output	Drive byte lanes on external memory bus. Controls I/O-pads connected to external memory bus.	Low/ High[2]
	VBDRIVE[31:0]	Output	Identical to BDRIVE but has one signal for each data bit. Every index is driven by its own register. This can be used to reduce the output delay.	Low/ High[2]
	READ	Output	Read strobe	High
	SA[14:0]	Output	Not used	High
AHBSI	[1)]	Input	AHB slave input signals	-
AHBSO	[1)]	Output	AHB slave output signals	-
SDO	SDCASN	Output	Not used. All signals are driven to inactive state.	Low

1) See GRLIB IP Library User’s Manual

2) Polarity is selected with the oepol generic

36.10 Library dependencies

Table shows libraries that the memory controller module depends on.

TABLE 20. Library dependencies

Library	Package	Imported unit(s)	Description
GRLIB	AMBA	Signals	AHB signal definitions
GAISLER	MEMCTRL	Signals, component	Memory bus signals definitions, component declaration

36.11 Memory controller instantiation

This examples shows how a memory controller can be instantiated. The example design contains an AMBA bus with a number of AHB components connected to it including the memory controller. The external memory bus is defined on the example designs port map and connected to the memory controller. System clock and reset are generated by GR Clock Generator and Reset Generator.

Memory controller decodes default memory areas: PROM area is 0x0 - 0xFFFFFF and RAM area is 0x40000000 - 0x40FFFFF. The 8-bit PROM mode is disabled. Two SRAM banks of size 64 Mbyte are used and the fifth chip select is disabled.

```
library ieee;
use ieee.std_logic_1164.all;

library grlib;
use grlib.amba.all;
use grlib.tech.all;
library gaisler;
use gaisler.memctrl.all;
use gaisler.pads.all;   -- used for I/O pads
use gaisler.misc.all;
library esa;
use esa.memoryctrl.all;

entity srctrl_ex is
  port (
    clk : in std_ulogic;
    resetn : in std_ulogic;
    pllref : in  std_ulogic;

    -- memory bus
    address  : out   std_logic_vector(27 downto 0); -- memory bus
    data     : inout std_logic_vector(31 downto 0);
    ramsn    : out   std_logic_vector(4 downto 0);
    ramoen   : out   std_logic_vector(4 downto 0);
    rwen     : inout std_logic_vector(3 downto 0);
    romsn    : out   std_logic_vector(1 downto 0);
    iosn     : out   std_logic;
    oen      : out   std_logic;
    read     : out   std_logic;
    writen   : inout std_logic;
    brdyn    : in    std_logic;
    bexcn    : in    std_logic;
    modesel  : in    std_logic; --PROM width select
-- sdram i/f
    sdcke    : out std_logic_vector ( 1 downto 0);  -- clk en
    sdcsn    : out std_logic_vector ( 1 downto 0);  -- chip sel
    sdwen    : out std_logic;                       -- write en
    sdrasn   : out std_logic;                       -- row addr stb
    sdcasn   : out std_logic;                       -- col addr stb
    sddqm    : out std_logic_vector (7 downto 0);  -- data i/o mask
    sdclk    : out std_logic;                       -- sdram clk output
    sa       : out std_logic_vector(14 downto 0); -- optional sdram address
    sd       : inout std_logic_vector(63 downto 0) -- optional sdram data
       );
end;
```

```
architecture rtl of srctrl_ex is

  -- AMBA bus (AHB and APB)
  signal apbi  : apb_slv_in_type;
  signal apbo  : apb_slv_out_vector := (others => apb_none);
  signal ahbsi : ahb_slv_in_type;
  signal ahbso : ahb_slv_out_vector := (others => ahbs_none);
  signal ahbmi : ahb_mst_in_type;
  signal ahbmo : ahb_mst_out_vector := (others => ahbm_none);

  -- signals used to connect memory controller and memory bus
  signal memi : memory_in_type;
  signal memo : memory_out_type;

  signal sdo : sdctrl_out_type;

  signal wprot : wprot_out_type;  -- dummy signal, not used
  signal clkm, rstn : std_ulogic; -- system clock and reset

-- signals used by clock and reset generators
  signal cgi : clkgen_in_type;
  signal cgo : clkgen_out_type;

  signal gnd : std_ulogic;

begin

  -- AMBA Components are defined here ...


  -- Clock and reset generators
  clkgen0 : clkgen generic map (clk_mul => 2, clk_div => 2, sdramen => 1,
                                tech => virtex2, sdinvclk => 0)
  port map (clk, gnd, clkm, open, open, sdclk, open, cgi, cgo);

  cgi.pllctrl <= "00"; cgi.pllrst <= resetn; cgi.pllref <= pllref;

  rst0 : rstgen
  port map (resetn, clkm, cgo.clklock, rstn);


  -- Memory controller
  srctrl0 : srctrl generic map (rmw => 1, prom8en => 0, srbanks => 2,
    banksz => 13, ramsel5 => 0)
    port map (rstn, clkm, ahbsi, ahbso(0), memi, memo, sdo);


  -- I/O pads driving data memory bus data signals
  datapads : for i in 0 to 3 generate
      data_pad : iopadv generic map (width => 8)
      port map (pad => data(31-i*8 downto 24-i*8),
                o => memi.data(31-i*8 downto 24-i*8),
                en => memo.bdrive(i),
                i => memo.data(31-i*8 downto 24-i*8));
  end generate;

  -- Alternative I/O pad instantiation with vectored enable instead
  datapads : for i in 0 to 3 generate
      data_pad : iopadvv generic map (width => 8)
      port map (pad => data(31-i*8 downto 24-i*8),
                o => memi.data(31-i*8 downto 24-i*8),
                en => memo.bdrive(31-i*8 downto 24-i*8),
                i => memo.data(31-i*8 downto 24-i*8));
  end generate;
```

```
  -- connect memory controller outputs to entity output signals
  address <= memo.address; ramsn <= memo.ramsn; romsn <= memo.romsn;
  oen <= memo.oen; rwen <= memo.wrn; ramoen <= memo.ramoen;
  writen <= memo.writen; read <= memo.read; iosn <= memo.iosn;
  sdcke <= sdo.sdcke; sdwen <= sdo.sdwen; sdcsn <= sdo.sdcsn;
  sdrasn <= sdo.rasn; sdcasn <= sdo.casn; sddqm <= sdo.dqm;

end;
```

37 SYNCRAM - Single-port RAM generator

37.1 Operation

The single port RAM has a common address bus, and separate data-in and data-out buses. All inputs are latched on the on the rising edge of clk. The read data appears on dataout directly after the clk rising edge.

37.2 Component declaration

```
library grlib;
use grlib.tech.all;

library gaisler;
use gaisler.memory.all;

  component syncram
  generic (tech : integer := 0; abits : integer := 6; dbits : integer := 8);
  port (
    clk      : in std_ulogic;
    address  : in std_logic_vector((abits -1) downto 0);
    datain   : in std_logic_vector((dbits -1) downto 0);
    dataout  : out std_logic_vector((dbits -1) downto 0);
    enable   : in std_ulogic;
    write    : in std_ulogic);
  end component;
```

37.3 Signals

TABLE 21. SYNCRAM signals

Signal name	Type	Function	Active
CLK	Input	Clock. All input signals are latched on the rising edge of the clock.	-
ADDRESS	Input	Address bus. Used for both read and write access.	-
DATAIN	Input	Data inputs for write data	-
DATAOUT	Output	Data outputs for read data	-
ENABLE	Input	Chip select	High
WRITE	Input	Write enable	High

37.4 Parameters and technology support

TABLE 22. SYNCRAM parameters (VHDL generics)

Name	Function	Range	Default
tech	Technology selection	0 - NTECH	0
abits	Address bits. Depth of RAM is $2^{abits-1}$	see table below	-
dbits	Data width	see table below	-

TABLE 23. SYNCRAM supported technologies

Tech name	Technology	RAM cell	abit range	dbit range
ihp15	IHP 0.25	sram2k (512x32)	2 - 9	unlimited
inferred	Behavioral description	Tool dependent	unlimited	unlimited
virtex	Xilinx Virtex, Virtex-E, Spartan-2	RAMB4_Sn	2 - 12	unlimited
virtex2	Xilinx Virtex2, Spartan3, Virtex4	RAMB16_Sn	2 - 14	unlimited
axcel	Actel AX, RTAX	RAM64K36	2 - 12	unlimited
proasic	Actel Proasic	RAM256x9SST	2 - 14	unlimited
proasic3	Actel Proasic3	ram4k9, ram512x18	2 - 12	unlimited
memvirage	Virage ASIC RAM	hdss1_128x32cm4sw0 hdss1_256x32cm4sw0 hdss1_512x32cm4sw0 hdss1_1024x32cm8sw0	7 - 11	32

37.5 Component instantiation

This examples shows how a SYNCRAM can be instantiated.

```
library ieee;
use ieee.std_logic_1164.all;
library gaisler;
use gaisler.memory.all;
.
.
clk      : std_ulogic;
address  : std_logic_vector((abits -1) downto 0);
datain   : std_logic_vector((dbits -1) downto 0);
dataout  : std_logic_vector((dbits -1) downto 0);
enable   : std_ulogic;
write    : std_ulogic);

ram0 : syncram generic map ( tech => tech, abits => addrbits, dbits => dbits)
      port map ( clk, addr, datain, dataout, enable, write);
```

38 SYNCRAM_2P - Two-port RAM generator

38.1 Operation

The two-port RAM generator has a one read port and one write port. Each port has a separate address and data bus. All inputs are registered on the rising edge of clk. The read data appears on dataout directly after the clk rising edge. Address width, data width and target technology is parametrizable through generics.

Write-through is supported if the function *syncram_2p_write_through(tech)* returns 1 for the target technology.

38.2 Component declaration

```
library grlib;
use grlib.tech.all;

library gaisler;
use gaisler.memory.all;

component syncram_2p
  generic (tech : integer := 0; abits : integer := 6; dbits : integer := 8; sepclk :
integer := 0);
  port (
    rclk     : in std_ulogic;
    renable  : in std_ulogic;
    raddress : in std_logic_vector((abits -1) downto 0);
    dataout  : out std_logic_vector((dbits -1) downto 0);
    wclk     : in std_ulogic;
    write    : in std_ulogic;
    waddress : in std_logic_vector((abits -1) downto 0);
    datain   : in std_logic_vector((dbits -1) downto 0));
  end component;
```

38.3 Signals

TABLE 24. SYNCRAM_2P signals

Signal name	Type	Function	Active
RCLK	Input	Read port clock	-
RENABLE	Input	Read enable	High
RADDRESS	Input	Read address bus	-
DATAOUT	Output	Data outputs for read data	-
WCLK	Input	Write port clock	-
WRITE	Input	Write enable	High
WADDRESS	Input	Write address	-
DATAIN	Input	Write data	-

38.4 Parameters and supported technologies

TABLE 25. Parameters (VHDL generics)

Name	Function	Range	Default
tech	Technology selection	0 - NTECH	0
abits	Address bits. Depth of RAM is $2^{abits-1}$	see table below	-
dbits	Data width	see table below	-
sepclk	If 1, separate clocks (rclk/wclk) are used for the two ports. If 0, rclk is used for both ports.	0 - 1	0

TABLE 26. Supported technologies

Tech name	Technology	RAM cell	abit range	dbit range
Inferred	Behavioural description	Tool dependent	unlimited	unlimited
virtex	Xilinx Virtex, Virtex-E, Spartan-2	RAMB4_Sn	2 - 10	unlimited
virtex2	Xilinx Virtex2, Spartan3, Virtex4	RAMB16_Sn	2 - 10	unlimited
axcel	Actel AX, RTAX	RAM64K36	2 - 12	unlimited
proasic	Actel Proasic	RAM256x9SST	2 - 14	unlimited
proasic3	Actel Proasic3	ram4k9, ram512x18	2 - 12	unlimited
memvirage	Virage ASIC RAM	hdss2_64x32cm4sw0 hdss2_128x32cm4sw0 hdss2_256x32cm4sw0 hdss2_512x32cm4sw0	6 - 9	32

38.5 Component instantiation

This examples shows how a SYNCRAM_2P can be instantiated.

```
library ieee;
use ieee.std_logic_1164.all;
library gaisler;
use gaisler.memory.all;

rclk     : in std_ulogic;
renable  : in std_ulogic;
raddress : in std_logic_vector((abits -1) downto 0);
dataout  : out std_logic_vector((dbits -1) downto 0);
wclk     : in std_ulogic;
write    : in std_ulogic;
waddress : in std_logic_vector((abits -1) downto 0);
datain   : in std_logic_vector((dbits -1) downto 0));

ram0 : syncram_2p generic map ( tech => tech, abits => addrbits, dbits => dbits)
      port map ( rclk, renable, raddress, dataout, wclk, write, waddress, datain,
                 enable, write);
```

39 SYNCRAM_DP dual-port RAM generator

39.1 Operation

The dual-port RAM generator has two independent read/write ports. Each port has a separate address and data bus. All inputs are latched on the on the rising edge of clk. The read data appears on dataout directly after the clk rising edge. Address width, data width and target technology is parametrizable through generics. Simultaneous write to the same address is technology dependent, and generally not allowed.

39.2 Component declaration

```
library grlib;
use grlib.tech.all;

library gaisler;
use gaisler.memory.all;

component syncram_dp
  generic (tech : integer := 0; abits : integer := 6; dbits : integer := 8);
  port (
    clk1     : in std_ulogic;
    address1 : in std_logic_vector((abits -1) downto 0);
    datain1  : in std_logic_vector((dbits -1) downto 0);
    dataout1 : out std_logic_vector((dbits -1) downto 0);
    enable1  : in std_ulogic;
    write1   : in std_ulogic;
    clk2     : in std_ulogic;
    address2 : in std_logic_vector((abits -1) downto 0);
    datain2  : in std_logic_vector((dbits -1) downto 0);
    dataout2 : out std_logic_vector((dbits -1) downto 0);
    enable2  : in std_ulogic;
    write2   : in std_ulogic);
  end component;
```

39.3 Signals

TABLE 27. SYNCRAM_DP signals

Signal name	Type	Function	Active
CLK1	Input	Port1 clock	-
ADDRESS1	Input	Port1 address	-
DATAIN1	Input	Port1 write data	-
DATAOUT1	Output	Port1 read data	-
ENABLE1	Input	Port1 chip select	High
WRITE1	Input	Port 1 write enable	High
CLK2	Input	Port2 clock	-
ADDRESS2	Input	Port2 address	-
DATAIN2	Input	Port2 write data	-
DATAOUT2	Output	Port2 read data	-
ENABLE2	Input	Port2 chip select	High
WRITE2	Input	Port 2 write enable	High

39.4 Parameters and supported technologies

TABLE 28. Parameters (VHDL generics)

Name	Function	Range	Default
tech	Technology selection	0 - NTECH	0
abits	Address bits. Depth of RAM is $2^{abits-1}$	see table below	-
dbits	Data width	see table below	-

TABLE 29. Supported technologies

Tech name	Technology	RAM cell	abit range	dbit range
virtex	Xilinx Virtex, Virtex-E, Spartan-2	RAMB4_Sn	2 - 10	unlimited
virtex2	Xilinx Virtex2, Spartan3, Virtex4	RAMB16_Sn	2 - 14	unlimited
proasic3	Actel Proasic3	ram4k9	2 - 12	unlimited
memvirage	Virage ASIC RAM	hdss2_64x32cm4sw0 hdss2_128x32cm4sw0 hdss2_256x32cm4sw0 hdss2_512x32cm4sw0	6 - 9	32

39.5 Component instantiation

This examples shows how a SYNCRAM_DP can be instantiated.

```
library ieee;
use ieee.std_logic_1164.all;
library gaisler;
use gaisler.memory.all;

clk1     : in std_ulogic;
address1 : in std_logic_vector((abits -1) downto 0);
datain1  : in std_logic_vector((dbits -1) downto 0);
dataout1 : out std_logic_vector((dbits -1) downto 0);
enable1  : in std_ulogic;
write1   : in std_ulogic;
clk2     : in std_ulogic;
address2 : in std_logic_vector((abits -1) downto 0);
datain2  : in std_logic_vector((dbits -1) downto 0);
dataout2 : out std_logic_vector((dbits -1) downto 0);
enable2  : in std_ulogic;
write2   : in std_ulogic);

ram0 : syncram_dp generic map ( tech => tech, abits => addrbits, dbits => dbits)
      port map ( clk1, address1, datain1, dataout1, enable1, write1, clk2, address2,
                  datain2, dataout2, enable2, write2);
```

40 TAP - JTAG TAP Controller

40.1 Overview

JTAG TAP Controller provides an Test Access Port according to IEEE-1149 (JTAG) Standard. The module implements the Test Access Port signals, the synchronous TAP state-machine, a number of JTAG data registers (depending on the target technology) and an interface to user-defined JTAG data registers.

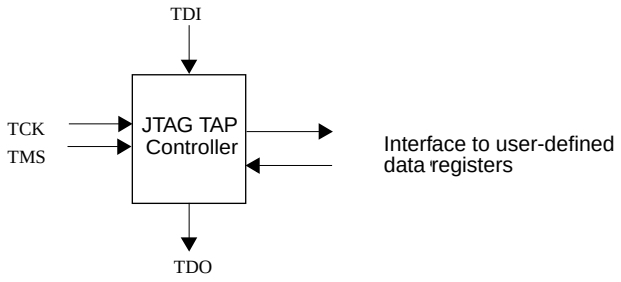

Figure 111. TAP Controller block diagram

40.2 Operation

40.2.1 Generic TAP Controller

The generic TAP Controller implements JTAG Test Access Point interface with signals TCK, TMS, TDI and TDO, a synchronous state-machine compliant to the IEEE-1149 standard, JTAG instruction register and two JTAG data registers: bypass and device identification code register. The module is capable of shifting and updating the JTAG instruction register, putting the device into bypass mode (BYPASS instruction) and shifting out the devices identification number (IDCODE instruction). User-defined JTAG test registers are accessed through user-defined data register interface.

The access to the user-define test data registers is provided through the user-defined data register interface. The instruction in the TAP controller instruction register appears on the interface as well as shift-in data and signals indicating that the TAP controller is in Capture-Data-Register, Shift-Data-Register or Update-Data-Register state. Logic controlling user-defined data registers should observe value in the instruction register and TAP controller state signals in order to capture data, shift data or update data-registers.

JTAG test registers such as boundary-scan register can be interfaced to the TAP controller through the user data register interface.

40.3 Technology specific TAP controllers

The module instantiates technology specific TAP controller if it is available in target technology.

40.4 Configuration options

JTAG TAP Controller module has the following configuration options (VHDL generics):

TABLE 30. TAP Controller configuration options (VHDL generics)

Generic	Function	Allowed range	Default
tech	Target technology	0 - NTECH	0
irlen	Instruction register length (generic tech only)	2 - 8	2
idcode	JTAG IDCODE instruction code(generic tech only)	0 - 255	9
id_msb	JTAG Device indentification code MSB bits (generic tech only)	0 - 65536	0
id_lsb	JTAG Device indentification code LSB bits (generic tech only)	0 - 65536	0
idcode	JTAG IDCODE instruction (generic tech only)	0 - 255	9

40.5 Vendor and device id

The module does not have vendor and device id since it does not have AMBA AHB interface.

40.6 Registers

The module implements three JTAG registers: instruction, bypass and device identification code register.

40.7 Signal description

TAP Controller signals are described in table 31.

TABLE 31. TAP Controller signals

Signal name	Field	Type	Function	Active
RST	N/A	Input	System reset	Low
CLK	N/A	Input	System clock (AHB clock domain)	-
TCK	N/A	Input	JTAG clock*	-
TCKN	N/A	Input	Inverted JTAG clock*	-
TMS	N/A	Input	JTAG TMS signal*	High
TDI	N/A	Input	JTAG TDI signal*	High
TDO	N/A	Output	JTAG TDO signal*	High
User-defined data register interface				
TAPO_TCK	N/A	Output	TCK signal	High
TAPO_TDI	N/A	Output	TDI signal	High
TAPO_INST[7:0]	N/A	Output	Instruction in the TAP Ctrl instruction register	High
TAPO_RST	N/A	Output	TAP Controller in Test-Logic_Reset state	High
TAPO_CAPT	N/A	Output	TAP Controller in Capture-DR state	High
TAPO_SHFT	N/A	Output	TAP Controller in Shift-DR state	High
TAPO_UPD	N/A	Output	TAP Controller in Update-DR state	High
TAPO_XSEL1	N/A	Output	Xilinx User-defined Data Register 1 selected (Xilinx tech only)	High
TAPO_XSEL2	N/A	Output	Xilinx User-defined Data Register 2 selected (Xilinx tech only)	High
TAPI_EN1	N/A	Input	Enable shift-out data port 1 (TAPI_TDO1), when disabled data on port 2 is used	High
TAPI_TDO1	N/A	Input	Shift-out data from user-defined register port 1	High
TAPI_TDO2	N/A	Input	Shift-out data from user-defined register port 2	High

*) If the target technology is Xilinx Virtex-II or Spartan3 the modules JTAG signals TCK, TCKN, TMS, TDI and TDO are not used. Instead the dedicated FPGA JTAG pins are used. These pins are implicitly made visible to the module through Xilinx TAP controller instantiation.

40.8 Library dependencies

Table 32 shows libraries that should be used when instantiating a TAP Controller.

TABLE 32. Library dependencies

Library	Package	Imported unit(s)	Description
GAISLER	JTAG	Component	TAP Controller component declaration

40.9 JTAG TAP Controller instantiation

This examples shows how a TAP Controller can be instantiated.

```
library ieee;
use ieee.std_logic_1164.all;

library gaisler;
use gaisler.jtag.all;

entity tap_ex is
  port (
    clk : in std_ulogic;
    rst : in std_ulogic;

    -- JTAG signals
    tck  : in std_ulogic;
    tms  : in std_ulogic;
    tdi  : in std_ulogic;
    tdo  : out std_ulogic
);
end;

architecture rtl of tap_ex is

signal gnd : std_ulogic;

signal tapo_tck, tapo_tdi, tapo_rst, tapo_capt : std_ulogic;
signal tapo_shft, tapo_upd : std_ulogic;
signal tapi_en1, tapi_tdo : std_ulogic;
signal tapo_inst : std_logic_vector(7 downto 0);


begin

 gnd <= '0';
 tckn <= not tck;


-- TAP Controller

  tap0 : tap (tech => 0)
    port map (rst, tck, tckn, tms, tdi, tdo, tapo_tck, tapo_tdi, tapo_inst,
      tapo_rst, tapo_capt, tapo_shft, tapo_upd, open, open,
       tapi_en1, tapi_tdo, gnd);


-- User-defined JTAG data registers

  ...


end;
```