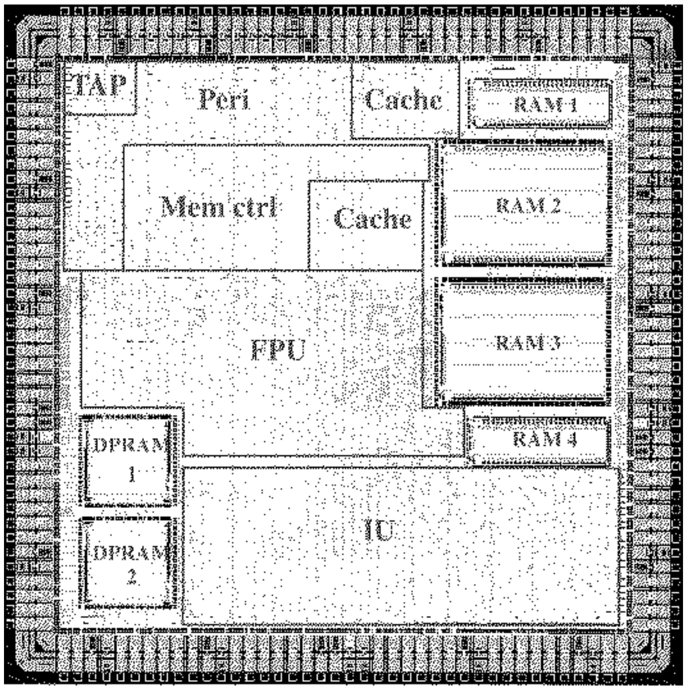
# LEON2 Processor User's Manual

Version 1.0.30

XST Edition

July 2005



GAISLER RESEARCH

1 Introduction .......... 8
1.1 Overview .......... 8
1.2 License .......... 8
1.3 Fault-tolerant LEON (LEON-FT) .......... 8
1.4 Functional overview .......... 9
1.4.1 Integer unit .......... 9
1.4.2 Floating-point unit and co-processor .......... 9
1.4.3 Cache sub-system .......... 9
1.4.4 Memory management unit .......... 10
1.4.5 Debug support unit .......... 10
1.4.6 Memory interface .......... 10
1.4.7 Timers .......... 10
1.4.8 Watchdog .......... 10
1.4.9 UARTs .......... 10
1.4.10 Interrupt controller .......... 10
1.4.11 Parallel I/O port .......... 10
1.4.12 AMBA on-chip buses .......... 11
1.4.13 PCI interface .......... 11
1.4.14 Ethernet MAC .......... 11
1.4.15 On-chip ram .......... 11
1.5 Performance .......... 11

2 Simulation and synthesis .......... 12
2.1 Un-packing the tar-file .......... 12
2.2 Configuration .......... 12
2.3 Simulation .......... 12
2.3.1 Compilation of the model .......... 12
2.3.2 Generic test bench .......... 13
2.3.3 Disassembler .......... 13
2.3.4 Modelsim support .......... 14
2.3.5 Ncsim support .......... 14
2.3.6 GNU VHDL (GHDL) support .......... 14
2.3.7 Synopsys VSS support .......... 14
2.3.8 Post-synthesis simulation .......... 14
2.4 Synthesis .......... 15
2.4.1 General .......... 15
2.4.2 Synplify .......... 15
2.4.3 Synopsys-DC .......... 15
2.4.4 Xilinx XST .......... 16
2.4.5 Synthesis board support packages .......... 16

3 LEON integer unit .......... 18
3.1 Overview .......... 18
3.2 Instruction pipeline .......... 19

3.3 Multiply instructions ........ 19
3.4 Multiply and accumulate instructions ........ 20
3.5 Divide instructions ........ 20
3.6 Processor reset operation ........ 20
3.7 Exceptions ........ 21
3.8 Hardware breakpoints ........ 22
3.9 SPARC Implementor’s ID ........ 22
3.10 Floating-point unit ........ 22
3.10.1 Introduction ........ 22
3.10.2 Gaisler Research’s floating-point unit (GRFPU) ........ 22
3.10.3 The Meiko FPU ........ 23
3.10.4 The LTH FPU ........ 23
3.10.5 Generic co-processor ........ 23

4 Cache sub-system ........ 24
4.1 Overview ........ 24
4.2 Instruction cache ........ 25
4.2.1 Operation ........ 25
4.2.2 Instruction cache tag ........ 25
4.3 Data cache ........ 26
4.3.1 Operation ........ 26
4.3.2 Write buffer ........ 26
4.3.3 Data cache snooping ........ 27
4.3.4 Data cache tag ........ 27
4.4 Cache flushing ........ 27
4.5 Diagnostic cache access ........ 28
4.6 Cache line locking ........ 28
4.7 Local ram ........ 29
4.7.1 Local instruction ram ........ 29
4.7.2 Local data ram ........ 29
4.8 Cache Control Register ........ 29

5 Memory management unit ........ 31
5.1 ASI mappings ........ 31
5.2 Caches ........ 31
5.3 MMU registers ........ 31
5.4 Translation look-aside buffer (TLB) ........ 32

6 AMBA on-chip buses ........ 33
6.1 Overview ........ 33
6.2 AHB bus ........ 33
6.3 APB bus ........ 33
6.4 AHB transfers generated by the processor ........ 33

7 On-chip peripherals ... 34
7.1 On-chip registers ... 34
7.2 Interrupt controller ... 35
7.2.1 Operation ... 35
7.2.2 Interrupt assignment ... 36
7.2.3 Control registers ... 36
7.3 Secondary interrupt controller ... 38
7.3.1 Operation ... 38
7.3.2 Control registers ... 39
7.4 Timer unit ... 40
7.4.1 Operation ... 40
7.4.2 Registers ... 41
7.5 UARTs ... 42
7.5.1 Transmitter operation ... 42
7.5.2 Receiver operation ... 43
7.5.3 Baud-rate generation ... 43
7.5.4 Loop back mode ... 43
7.5.5 Interrupt generation ... 44
7.5.6 UART registers ... 44
7.6 Parallel I/O port ... 45
7.7 LEON configuration register ... 47
7.8 Power-down ... 47
7.9 AHB status register ... 48
7.10 AHB ram ... 48

8 External memory access ... 49
8.1 Memory interface ... 49
8.2 Memory controller ... 49
8.3 PROM access ... 50
8.4 Memory mapped I/O ... 50
8.5 SRAM access ... 51
8.6 Burst cycles ... 52
8.7 8-bit and 16-bit PROM and SRAM access ... 52
8.8 8- and 16-bit I/O access ... 53
8.9 SDRAM access ... 53
8.9.1 General ... 53
8.9.2 Address mapping ... 53
8.9.3 Initialisation ... 54
8.9.4 Configurable SDRAM timing parameters ... 54
8.9.5 Refresh ... 54
8.9.6 SDRAM commands ... 54
8.9.7 Read cycles ... 54
8.9.8 Write cycles ... 55

8.9.9 Address bus connection .................. 55
8.9.10 Clocking .................. 55
8.10 Memory configuration register 1 (MCFG1) .................. 55
8.11 Memory configuration register 2 (MCFG2) .................. 56
8.12 Memory configuration register 3 (MCFG3) .................. 57
8.13 Write protection .................. 57
8.14 Using BRDYN .................. 57
8.15 Access errors .................. 58
8.16 Attaching an external DRAM controller .................. 58

9 PCI interface .................. 59

10 Ethernet interface .................. 60

11 Hardware debug support .................. 61
11.1 Overview .................. 61
11.2 Debug support unit .................. 61
11.2.1 Overview .................. 61
11.2.2 Trace buffer .................. 62
11.2.3 DSU memory map .................. 64
11.2.4 DSU control register .................. 65
11.2.5 DSU breakpoint registers .................. 66
11.2.6 DSU trap register .................. 66
11.3 DSU communication link .................. 67
11.3.1 Operation .................. 67
11.3.2 DSU UART control register .................. 68
11.3.3 DSU UART status register .................. 68
11.3.4 Baud rate generation .................. 69
11.4 Common operations .................. 69
11.4.1 Instruction breakpoints .................. 69
11.4.2 Single stepping .................. 70
11.4.3 Alternative debug sources .................. 70
11.4.4 Booting from DSU .................. 70
11.5 Design limitations .................. 70
11.6 DSU monitor .................. 70
11.7 External DSU signals .................. 70

12 Signals .................. 72
12.1 Memory bus signals .................. 72
12.2 System interface signals .................. 72
12.3 Signal description .................. 73

13 VHDL model architecture .................. 76
13.1 Model hierarchy .................. 76

13.2 Model coding style ........ 77
13.3 AMBA buses ........ 78
13.3.1 AMBA AHB ........ 78
13.3.2 AHB cache aspects ........ 79
13.3.3 AHB protection signals ........ 79
13.3.4 APB bus ........ 79
13.4 Floating-point unit and co-processor ........ 79
13.4.1 Generic CP interface ........ 79
13.4.2 FPU interface ........ 80

14 Model Configuration ........ 81
14.1 Synthesis configuration ........ 81
14.2 Integer unit configuration ........ 81
14.3 FPU configuration ........ 83
14.4 Cache configuration ........ 83
14.5 Memory controller configuration ........ 84
14.6 Debug configuration ........ 84
14.7 Peripheral configuration ........ 85
14.8 Boot configuration ........ 86
14.8.1 Booting from internal prom ........ 86
14.8.2 PMON S-record loader ........ 87
14.8.3 Rdbmon ........ 87
14.8.4 Booting from the debug support unit ........ 87
14.9 AMBA configuration ........ 87
14.9.1 AHB configuration ........ 88
14.9.2 APB configuration ........ 88
14.10 PCI configuration ........ 88

15 Porting to a new technology or synthesis tool ........ 90
15.1 General ........ 90
15.2 Target specific mega-cells ........ 90
15.2.1 Integer unit register-file ........ 90
15.2.2 Parallel FPU & co-processor register file ........ 91
15.2.3 Cache ram memory cells ........ 91
15.2.4 Dual-port rams ........ 92
15.2.5 Pads ........ 92
15.2.6 Adding a new technology or synthesis tool ........ 92

# 1 Introduction

## 1.1 Overview

The LEON VHDL model implements a 32-bit processor conforming to the IEEE-1754 (SPARC V8) architecture. It is designed for embedded applications with the following features on-chip: separate instruction and data caches, hardware multiplier and divider, interrupt controller, debug support unit with trace buffer, two 24-bit timers, two UARTs, power-down function, watchdog, 16-bit I/O port, flexible memory controller, ethernet MAC and PCI interface. New modules can easily be added using the on-chip AMBA AHB/APB buses. The VHDL model is fully synthesisable with most synthesis tools and can be implemented on both FPGAs and ASICs. Simulation can be done with all VHDL-87 compliant simulators.

**Note:** this manual describes the full functionality of the LEON model. Through the model’s configuration record (see “Model Configuration” on page 81), parts of the described functionality can be suppressed or modified to generate a smaller or faster implementation.

## 1.2 License

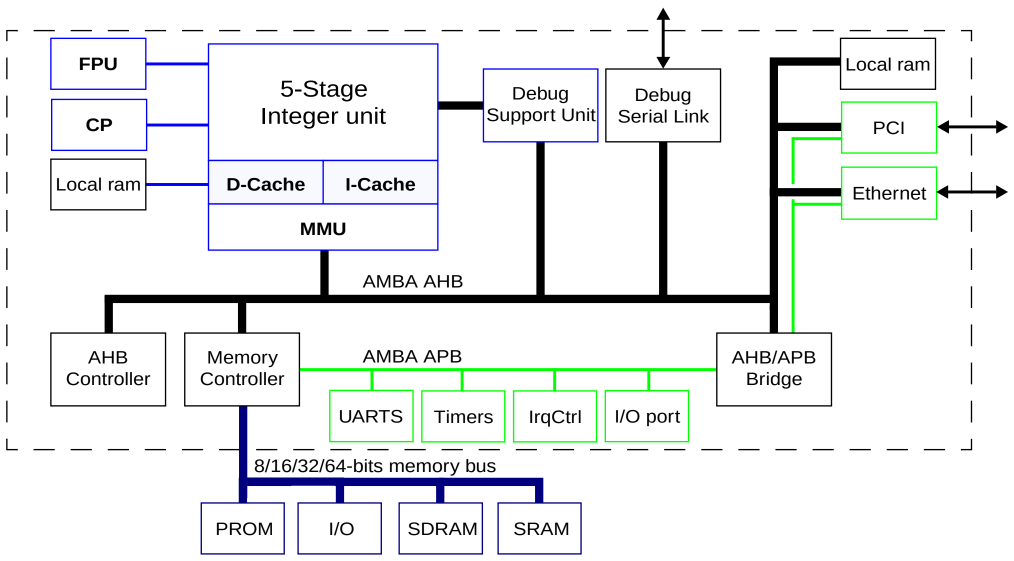
The LEON VHDL model is provided under two licenses: the GNU Public License (GPL) and the Lesser GNU Public License (LGPL). The LGPL applies to the LEON model itself while remaining support files and test benches are provided under GPL. This means that you can use LEON as a core in a system-on-chip design without having to publish the source code of any additional IP-cores you might use. You must however publish any modifications you have made to the LEON core itself, as described in LGPL.

## 1.3 Fault-tolerant LEON (LEON-FT)

The original LEON design includes advanced fault-tolerance features to withstand arbitrary single-event upset (SEU) errors without loss of data. The fault-tolerance is provided at design (VHDL) level, and does not require an SEU-hard semiconductor process, nor a custom cell library or special back-end tools. This document provides some references to LEON-FT functionality, which users of the LGPL version can safely disregard since all FT logic has been removed in the LGPL version.

## 1.4 Functional overview

A block diagram of LEON-2 can be seen in figure 1.



### 1.4.1 Integer unit

The LEON integer unit implements the full SPARC V8 standard, including all 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. To aid software debugging, up to four watchpoint registers can be configured. Each register can cause a trap on an arbitrary instruction or data address range. If the debug support unit is enabled, the watchpoints can be used to enter debug mode.

### 1.4.2 Floating-point unit and co-processor

The LEON processor model provides an interface to the high-performance GRFPU (available from Gaisler Research), the Meiko FPU core (available from Sun Microsystems) and the incomplete LTH FPU. A generic co-processor interface is provided to allow interfacing of custom co-processors.

### 1.4.3 Cache sub-system

Separate, multi-set instruction and data caches are provided, each configurable with 1 - 4 sets, 1 - 64 kbyte/set, 16 - 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 minimise 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 the data cache controller to allow 0-waitstates access without requiring data write back to external memory.

### 1.4.4 Memory management unit

The (optional) memory management unit (MMU) implements a SPARC V8 reference MMU and allows usage of robust operating system such as linux or solaris. The MMU can have separate (I + D) or a common translation look-aside buffer (TLB). The TLB is configurable for 2 - 32 fully associative entries.

### 1.4.5 Debug support unit

The (optional) debug support unit (DSU) allows non-intrusive debugging on target hardware. The DSU allows to insert breakpoints and watchpoints, and access to all on-chip registers from a remote debugger. A trace buffer is provided to trace the executed instruction flow and/or AHB bus traffic. The DSU has no impact on performance and has low area complexity. Communication to an outside debugger (e.g. gdb) is done using a dedicated UART (RS232) or through any AHB master (e.g. PCI).

### 1.4.6 Memory interface

The flexible memory interface provides a direct interface PROM, memory mapped I/O devices, static RAM (SRAM) and synchronous dynamic RAM (SDRAM). The memory areas can be programmed to either 8-, 16-, 32- or 64-bit data width.

### 1.4.7 Timers

Two 24-bit timers are provided on-chip. The timers can work in periodic or one-shot mode. Both timers are clocked by a common 10-bit prescaler.

### 1.4.8 Watchdog

A 24-bit watchdog is provided on-chip. The watchdog is clocked by the timer prescaler. When the watchdog reaches zero, an output signal (WDOG) is asserted. This signal can be used to generate system reset.

### 1.4.9 UARTs

Two 8-bit UARTs are provided on-chip. The baud-rate is individually programmable and data is sent in 8-bits frames with one stop bit. Optionally, one parity bit can be generated and checked.

### 1.4.10 Interrupt controller

The interrupt controller manages a total of 15 interrupts, originating from internal and external sources. Each interrupt can be programmed to one of two priority levels. A chained, secondary controller for up to 32 additional interrupts is also available.

### 1.4.11 Parallel I/O port

A 32-bit parallel I/O port is provided. 16 bits are always available and can be individually programmed by software to be an input or an output. An additional 16 bits are only available when the memory bus is configured for 8- or 16-bit operation. Some of the bits have alternate usage, such as UART inputs/outputs and external interrupts inputs.

#### 1.4.12 AMBA on-chip buses

The processor has a full implementation of AMBA AHB and APB on-chip buses. A flexible configuration scheme makes it simple to add new IP cores. Also, all provided peripheral units implement the AMBA AHB/APB interface making it easy to add more of them, or reuse them on other processors using AMBA.

#### 1.4.13 PCI interface

A PCI interface supporting target-only can be enabled. The interface has low complexity and can be used for debugging over the PCI bus.

#### 1.4.14 Ethernet MAC

An ethernet 10/100 Mbit MAC can be enabled. The MAC is based on the ethernet MAC core from OpenCores, with an additional AHB interface.

#### 1.4.15 On-chip ram

A (small) on-chip ram can be attached to AHB bus, providing a fast local memory. The size can be configured from 1 - 64 kbyte.

## 1.5 Performance

Using 4K + 4K caches and a 16x16 multiplier, the dhrystone 2.1 benchmark reports 1,550 iteration/s/MHz using the gcc-2.95.2 compiler (-O2). This translates to 0.9 dhrystone MIPS/MHz using the VAX 11/780 value a reference for one MIPS.

# 2 Simulation and synthesis

## 2.1 Un-packing the tar-file

The model is distributed as a gzipped tar-file; leon-2.x.x.tar.gz. On unix systems, use the command ‘gunzip -c leon-2.x.x.tar.gz | tar xf -’ to unpack the model in the current directory. The LEON model has the following directory structure:

| | |
|---|---|
| leon | top directory |
| leon/Makefile | top-level makefile |
| leon/leon/ | LEON vhdl model |
| leon/pmon | Simple boot-monitor |
| leon/sim/ | Simulator support files |
| leon/syn | Synthesis support files |
| leon/tbench | LEON VHDL test bench |
| leon/tsource | LEON test bench (C source) |

## 2.2 Configuration

The LEON model is highly configurable, allowing the model to be customised for a certain application or target technology. A graphical configuration tool based on the linux kernel *tkconfig* scripts is used to configure the model. In the leon top-level directory, do a ‘make xconfig’ on unix platforms or a ‘make wconfig’ on Windows/Cygwin platforms. After a configuration has been saved, the corresponding VHDL configuration file (device.vhd) will generated and installed when doing a ‘make dep’. Note that a working installation of gcc and tcl/tk must be installed on the host for tkconfig to work. A configuration can also be made by editing leon/device.vhd directly. The tkconfig tools has help texts for each configuration option - use those to derive a suitable configuration. For a more detailed description of the configuration options and their effects, see “Model Configuration” on page 81. The tkconfig tool allows loading of pre-defined configurations using the ‘Load configuration’ option and a few configuration files (config_xxx) are provided in the ‘boards’ directory.

## 2.3 Simulation

### 2.3.1 Compilation of the model

On unix systems (or MS-windows with cygwin installed), the model and test benches can be built using the ‘make’ utility. Four make targets are currently supported:

| | |
|---|---|
| make vsim: | compile for the modelsim simulator (vcom/vlog) |
| make ncsim: | compile for Cadence ncsim simulator (ncvhdl/ncvlog) |
| make vss: | compile for Synopsys VSS simulator (vhdlan) |
| make ghdl: | compile for GNU VHDL simulator (ghdl, experimental) |

To change the compiler parameters or to use an other simulator, modify the Makefile in the top directory. On non-unix systems, the compile.bat file in the leon and tbench directories can be used to compile the model in correct order.

### 2.3.2 Generic test bench

A generic test bench is provided in tbench/tbgen.vhd. This test bench allows to generate a model of a LEON system with various memory sizes/types by setting the appropriate generics. The file tbench/tbleon.vhd contains a number of alternative configuration using the generic test bench:

- TB_FUNC8, TB_FUNC16, TB_FUNC32, TB_FUNC_SDRAM: Functional tests performing a quick check of most on-chip functions using either 8-, 16- or 32-bit external static ram, or 32-bit SDRAM.
- TB_MEM: Testing all on-chip memory with patterns of 0x55 and 0xAA.
- TB_FULL: Combined memory and functional tests, suitable to generate test vectors for manufacturing.

Once the LEON model have been compiled, use the TB_FUNC32 test bench to verify the behaviour of the model. **Simulation should be started in the top directory**. The output from the simulation should be similar to:

```
# # *** Starting LEON system test ***
# # Memory interface test
# # Cache memory
# # Register file
# # Interrupt controller
# # Timers, watchdog and power-down
# # Parallel I/O port
# # UARTs
# # Test completed OK, halting with failure
# ** Failure: TEST COMPLETED OK, ending with FAILURE
```

Simulation is halted by generating a failure.

The supplied test program which is run by the test benches only tests on-chip peripherals and interfaces, compliance to the SPARC standard has been tested with proprietary test vectors, not supplied with the model. To re-compile the test program, the LEON/ERC32 GNU Cross-Compiler System (LECCS) provided by Gaisler Research (www.gaisler.com) needs to be installed. The test programs are in the tsource directory and are built by executing 'make' in the tsource directory. The makefile will build the program and generate prom and ram images for the test bench. Pre-compiled images are supplied so that the test suite can be run without installing the compiler.

The test program probes the LEON configuration register to determine which options are enabled in the particular LEON configuration, and only tests those. E.g., if no FPU is present, the test program will not attempt to perform FPU testing.

### 2.3.3 Disassembler

A SPARC disassembler is provided in the DEBUG package. It is used by the test bench to disassemble the executed instructions and print them to stdout (if enabled). Test bench configurations with names ending in a '_disas' have disassembly enabled.

### 2.3.4 Modelsim support

The file sim/modelsim/wave.do is a macro file for modelsim to display some useful internal LEON signals. A modelsim init file (modelsim.ini) is present in the top directory and in the leon and tbench directories to provide appropriate library mapping. The complete model can be compiled from within modelsim by executing the modelsim/compile.do file:

```
vsim> do sim/modelsim/compile.do
```

### 2.3.5 Ncsim support

A ncsim library mapping file (cds.lib) is provided in the top directory, and in the leon and tbench directories to provide the appropriate library mappings. A shell script is provided in ncsim/compile.sh to build the whole model:

```
bash$ ./sim/ncsim/compile.sh
```

Note that a test benches first must be elaborated before they can be simulated:

```
ncelab -nocopyright tb_func32
```

### 2.3.6 GNU VHDL (GHDL) support

A shell script is provided in sim/ghdl/compile.sh to build the whole model:

```
bash$ ./sim/ghdl/compile.sh
```

Note that a test benches first must be elaborated before they can be simulated:

```
ghdl -e --workdir=leon/work --ieee=synopsys --std=87 tb_func32
./tb_func32

LEON-2 generic testbench (leon2-1.0.23-xst)
Bug reports to Jiri Gaisler, jiri@gaisler.com

Testbench configuration:
32 kbyte 32-bit rom, 0-ws
2x128 kbyte 32-bit ram, 2x64 Mbyte SDRAM
```

The GHDL simulator is still in development and might not simulate the leon model correctly, see http://ghdl.free.fr/ for the latest version information and bug reports.

### 2.3.7 Synopsys VSS support

A .synopsys_vss.setup file is present in the top directory and in the leon and tbench directory to provide appropriate library mapping for Synopsys VSS.

### 2.3.8 Post-synthesis simulation

The supplied test-benches can be used to simulate the synthesised netlist. Use the following procedure:

- Compile the complete model (i.e. do a ‘make’ at the top level). It is **essential** that you use the same configuration as during synthesis! This step is necessary because the test bench uses the target, config and device packages.
- In the top directory, compile the simulation libraries for you ASIC/FPGA technology, and then your VHDL netlist.

- Cd to tbench, and do ‘make clean all’. This will rebuild the test bench, ‘linking’ it with your netlist.
- Cd back to the top directory and simulate you test bench as usual.
- If you get problem with ‘X’ during simulation, use the TB_FULL test bench to make sure that all on-chip memories are properly initialised. Also make sure that you have loaded the SDF file into the simulator and the you use ps time resolution.

## 2.4 Synthesis

### 2.4.1 General

The model is written with synthesis in mind and has been tested with Synopsys DC, Xilinx XST, and Synplicity Synplify synthesis tools. Some versions of the Synopsys FPGA-Compiler (FPGA-Express) and Exemplar Leonardo can also be used, but these tools are know to be ‘buggy’ do not always succeed. Technology specific cells are used to implement the IU/FPU register files, cache rams, PCI FIFOs and pads. These cells can be automatically inferred (Synplify, XST and Leonardo) or directly instantiated from the target library (Synopsys).

Non-synthesisable code is enclosed in a set of embedded pragmas as shown below:

```
-- pragma translate_off

... non-synthesisable code...

-- pragma translate_on
```

This works with most synthesis tools, although in Synopsys requires the *hdlin_translate_off_skip_text* variable be set to *“true”*.

The ‘syn’ directory includes scripts/project-files for Synplify, Synopsys-DC, Synopsys-FC2, XST and Leonardo. The source files are read from the leon directory, so it is essential that the model has been correctly configured before. There are 4 top-level entities to choose from:

| | | |
|---|---|---|
| • leon.vhd | : | standard top entity |
| • leon_pci.vhd | : | Leon + PCI |
| • leon_eth.vhd | : | Leon + ethernet |
| • leon_eth_pci.vhd | : | Leon + ethernet + PCI |

### 2.4.2 Synplify

To synthesise LEON using Synplify, start synplify in the syn/synplify directory and open leon.prj. Make sure you use Synplify-8.2 or later, some previous versions could generate a incorrect netlist under certain circumstances.

### 2.4.3 Synopsys-DC

To synthesise LEON using Synopsys DC, start synopsys in the syn/synopsys directory and execute the script ‘leon.dcsh’. Before executing the script, edit the beginning of the script to insure that the library search paths reflects your synopsys installation and that the timing constraints are appropriate:

```
/* List paths to your sources, target, and link libraries below. */

include atc35setup.dcsh

/* constraints - tailor to your own technology. */

frequency = 62.5
clock_skew = 0.25
input_setup = 2.0
output_delay = 5.0
```

The top-level constraints are used to generate the appropriate synopsys constraints commands. For best results, use Synopsys DC version 2003.6.SP1. Newer versions with the PRESTO VHDL compiler are known to crash or produce the wrong netlist of certain LEON2 blocks.

### 2.4.4 Xilinx XST

Synthesise with Xilinx ISE/XST is possible, and version 7.1.02i is strongly recommended. The preferred synthesis method with ISE is to use a synthesis board support packages and the 'make' command as described below. However, a sample xst script is in syn/xst/leon.xst and can be used as a template. A template ISE 5.2 project file is provided in syn/xst/leon.npl.

### 2.4.5 Synthesis board support packages

To simplify targeting a particular FPGA board, templates for a few board has been made. The templates contains scripts that synthesises and place&route the complete design for the target board. The scripts detects the usage of PCI or ethernet, and chooses the corresponding top-level entity. The templates are in boards/xxx, where xxx denotes the target board. Using 'make', the VHDL model can be configured and an fpga programming file generated for any of the these boards:

- Gaisler/Pender GR-CPCI-XC2V (Xilinx XC2V6000)
- Gaisler/Pender GR-PCI-XC2V (Xilinx XC2V3000)
- Gaisler/Pender GR-XC3S1500 (Xilinx Spartan3-1500)
- Avnet Virtex-E Development board (XCV1000E)
- Hans Tiggler's XCV800 demo board (XCV800)
- XESS XSV800 prototype board (XCV800)
- Xilinx Virtex-II Multimedia Board (XC2V2000)

Below is the 'make' output from the top directory:

```
bash-2.05a$ make

choose one of following targets:

 make xgates     : compile Xilinx gate level models with modelsim
 make vsim       : compile simulation model with modelsim
 make ncsim      : compile simulation model with ncsim
 make vss        : compile simulation model with synopsys-vss
 make test       : run tb_func32 testbench with modelsim simulator
 make config     : configure VHDL model for the GR-PCI-XC2V fpga board
 make xconfig    : run the graphical configuration tool
 make fpga       : do synthesis and p&r for the GR-PCI-XC2V fpga board
 make clean      : remove all temporary files except config and fpga bitfiles
 make dist-clean : remove all temporary files
```

```
To target other fpga boards than the GR-PCI-XC2V, add BOARD=xxx
where xxx is one of gr-pci-xc2v, gr-cpci-xc2v, avnet-xcv1000e,
gr-xc3s1500, tiggeler-xcv800, hecht-xsv800 or billo-mblaze-xc2v

When synthesising for Virtex fpga’s, Xilinx XST is used by default.
To use Synplify, add parameter SYN=synplify. Place&route effort is
by default 3, but can be set through the EFFORT parameter. The CONFIG
parameter can be used to specify alternative model configurations.

Examples:
          make config fpga EFFORT=high CONFIG=60mhz
          make config fpga BOARD=avnet-xcv1000e SYN=synplify EFFORT=med
```

Pre-requisites are a working installation of either XST or synplify (supporting -batch), ISE 7.1.02i, and cygwin (on windows platforms). Tested platforms are linux, solaris and windows/cygwin.

# 3 LEON integer unit

The LEON integer unit (IU) implements SPARC integer instructions as defined in SPARC Architecture Manual version 8. It is a new implementation, not based on any previous designs. The implementation is focused on portability and low complexity.

## 3.1 Overview

The LEON integer unit has the following features:

- 5-stage instruction pipeline
- Separate instruction and data cache interface
- Support for 2 - 32 register windows
- Configurable multiplier (16x16, 32x1, 32x8, 32x16 & 32x32)
- Optional 16x16 bit MAC with 40-bit accumulator
- Radix-2 divider (non-restoring)

Figure 2 shows a block diagram of the integer unit.

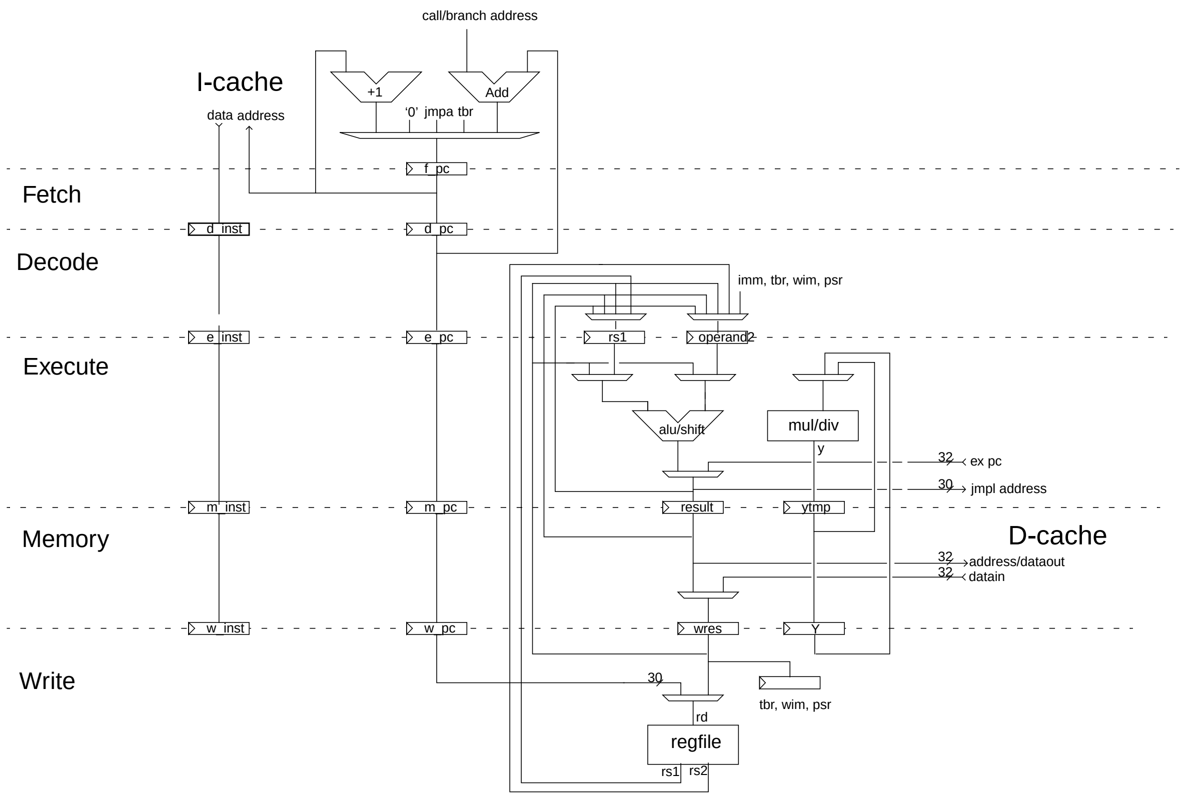


*Figure 2: LEON integer unit block diagram*

## 3.2 Instruction pipeline

The LEON integer unit uses a single instruction issue pipeline with 5 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 operands are read. Operands may come from the register file or from internal data bypasses. CALL and Branch target addresses are generated in this stage.

3. EX (Execute): ALU, logical, and shift operations are performed. For memory operations (e.g., LD) and for JMPL/RETT, the address is generated.

4. ME (Memory): Data cache is accessed. For cache reads, the data will be valid by the end of this stage, at which point it is aligned as appropriate. Store data read out in the execution stage is written to the data cache at this time.

5. WR (Write): The result of any ALU, logical, shift, or cache read operations are written back to the register file.

Table 1 lists the cycles per instruction (assuming cache hit and no load interlock):

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

*Table 1: Instruction timing*

* depends on multiplier configuration

## 3.3 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. Several multiplier implementation are provided, making it possible to choose between area, delay and latency (see “Integer unit configuration” on page 81 for more details).

## 3.4 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.

## 3.5 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.

## 3.6 Processor reset operation

The processor is reset by asserting the RESET input for at least one clock cycle. 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).

| **Register** | **Reset value** |
|---|---|
| PC (program counter) | 0x0 |
| nPC (next program counter) | 0x4 |
| PSR (processor status register) | ET=0, S=1 |
| CCR (cache control register) | 0x0 |

*Table 2: Processor reset values*

Execution will start from address 0.

## 3.7 Exceptions

LEON adheres to the general SPARC trap model. The table below shows the implemented traps and their individual 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) |

*Table 3: Trap allocation and priority*

## 3.8 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:

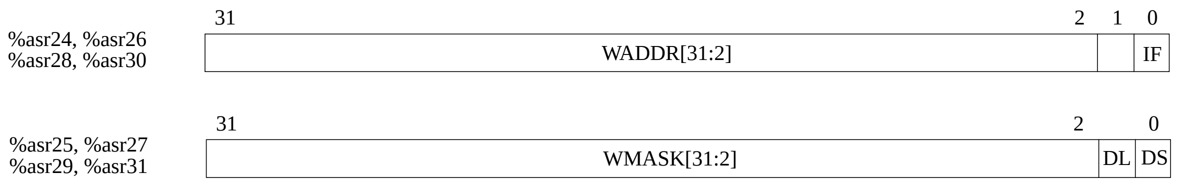


*Figure 3: Watch-point registers*

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.

## 3.9 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 LEON2 is 2, which is hard-coded in to bits 27:24 of the %psr.

## 3.10 Floating-point unit

### 3.10.1 Introduction

The SPARC V8 architecture defines two (optional) co-processors: one floating-point unit (FPU) and one custom-defined co-processor. The LEON2 pipeline provides one interface port for each of these units. Three different FPU's can be interfaced: Gaisler Research's GRFPU, the Meiko FPU from Sun and the LTH FPU. Selection of which FPU to use is done in the model's configuration record.

### 3.10.2 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 LEON2 pipeline using a LEON2-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 a new 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 LEON2 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:

| 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 |

*Table 4: GRFPU instruction timing with GRFPC*

Note that the GRFPU/GRFPC is not distributed with the open-source LEON model, and must be obtained separately from Gaisler Research.

### 3.10.3 The Meiko FPU

The Meiko floating-point core operates on both single- and double-precision operands, and implements all SPARC V8 FPU instructions. The interface operates in serial fashion, where FP instruction do not execute in parallel with IU instruction and the processor is stopped during the execution of floating-point instructions. This means that QNE bit in the %fsr register always is zero, and any attempts of executing the STDFQ instruction will generate a FPU exception trap. The Meiko FPU is not distributed with the open-source LEON model, and must be obtained separately from Sun.

### 3.10.4 The LTH FPU

The LTH FPU is designed by Martin Kasprzyk and operates on single- and double-precision operands. The FPU does not implement all SPARC V8 instruction, nor is it IEEE-754 compliant. It can thus not be used for general purpose programs, and should be seen as work in progress. The FPU uses the same serial interface as the Meiko FPU.

### 3.10.5 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. See “Floating-point unit and co-processor” on page 79 for interfacing details.

# 4 Cache sub-system

## 4.1 Overview

The LEON processor implements a Harvard architecture with separate instruction and data buses, connected to two independent cache controllers. 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 LEON 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[3:0] are used for the mapping, ASI[7:4] have no influence on operation.

| ASI | Usage |
|---|---|
| 0x0, 0x1, 0x2, 0x3 | Forced cache miss (replace if cacheable) |
| 0x4, 0x7 | Forced cache miss (update on hit) |
| 0x5 | Flush instruction cache |
| 0x6 | Flush data cache |
| 0x8, 0x9, 0xA, 0xB | Normal cached access (replace if cacheable) |
| 0xC | Instruction cache tags |
| 0xD | Instruction cache data |
| 0xE | Data cache tags |
| 0xF | Data cache data |

*Table 5: ASI usage*

Access to ASI 4 and 7 will force a cache miss, and update the cache if the data was previously cached. Access with ASI 0 - 3 will force a cache miss, update the cache if the data was previously cached, or allocated a new line if the data was not in the cache and the address refers to a cacheable location. The cacheable areas are by default the prom and ram areas, but are configurable by modifying the is_cacheable() function in macro.vhd:

| Address range | Area | Cached |
|---|---|---|
| 0x00000000 - 0x1FFFFFFF | PROM | Cacheable |
| 0x20000000 - 0x3FFFFFFF | I/O | Non-cacheable |
| 0x40000000 -0x7FFFFFFF | RAM | Cacheable |
| 0x80000000 -0xFFFFFFFF | Internal (AHB) | Non-cacheable |

*Table 6: Default cache table*

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 - 64 kbyte divided into cache lines with 8 - 32 bytes of data. In the multi-set configuration one of three replacement policies can be selected: least-recently-used (LRU), least-recently-replaced (LRR) or (pseudo-) random. If the LRR algorithm is used the cache has to be 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.

## 4.2 Instruction cache

### 4.2.1 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 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.

### 4.2.2 Instruction cache tag

A instruction cache tag entry consists of several fields as shown in figure 4:

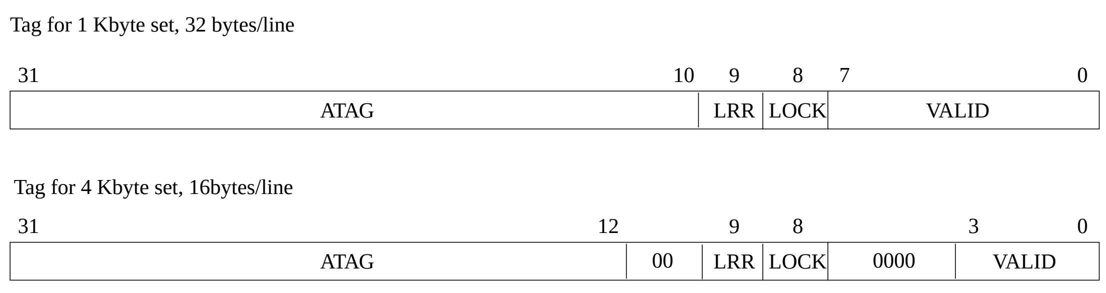

Tag for 1 Kbyte set, 32 bytes/line
31
10
9
8
7
0
ATAG
LRR
LOCK
VALID
Tag for 4 Kbyte set, 16bytes/line
31
12
9
8
3
0
ATAG
00
LRR
LOCK
0000
VALID

*Figure 4: 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. 0 if other replacement policy is used.
- [8]: LOCK - Locks a cache line when set. 0 if instruction cache locking was not enabled in the configuration.
- [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.

## 4.3 Data cache

### 4.3.1 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.

### 4.3.2 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.

### 4.3.3 Data cache snooping

The data cache can optionally perform snooping on the AHB bus. When snooping is enabled, the data cache controller will monitor write accesses on the AHB bus performed by other AHB masters (DMA). When a write access is performed to a cacheable memory location, the corresponding cache line will be invalidated in the data cache if present. Cache snooping has no overhead and does not affect performance. It can be dynamically enabled/disabled through bit 23 in the cache control register. Note that snooping is an optional feature and must be enabled in the VHDL configuration. Cache snooping requires the target technology to implement dual-port memories, which will be used to implement the cache tag RAM. It is not possible to enable snooping when an MMU is present in the system, since the cache addresses are virtual and the AHB addresses are physical.

### 4.3.4 Data cache tag

A data cache tag entry consists of several fields as shown in figure 5:

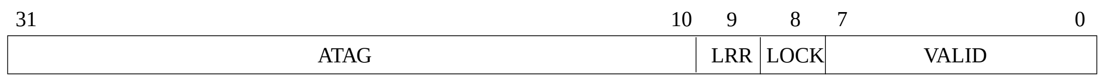


*Figure 5: Data cache tag layout*

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 other replacement policy is 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.

## 4.4 Cache flushing

The instruction and data cache is flushed by executing the FLUSH instruction, setting the FI bit in the cache control register, or by writing to any location with ASI=0x5. The flushing will take one cycle per cache line and set during which the IU will not be halted, but during which the instruction cache will be disabled. When the flush operation is completed, the cache will resume the state (disabled, enabled or frozen) indicated in the cache control register.

## 4.5 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.

## 4.6 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 LEON configuration: the lock bit will be set if the cache line locking was enabled otherwise it will be 0.

## 4.7 Local ram

### 4.7.1 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). When writing to the local ram, only word accesses are allowed. Byte, halfword or double word write will generate an data exception (trap 0x09). Read accesses can have any data size.

### 4.7.2 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 “Cache configuration” on page 83 for configuration details. Note that the ram cannot be used for instructions, only data.

## 4.8 Cache Control Register

The operation of the instruction and data caches is controlled through a common Cache Control Register (CCR) (figure 6). 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.

| 31 30 | 29 28 | 27 26 | 25 24 | 23 | 22 | 21 | | 16 | 15 | 14 | 6 | 5 | 4 | 3 2 | 1 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| DREPL | IREPL | ISETS | DSETS | DS | FD | FI | RESERVED | IB | IP | DP | RESERVED | DF | IF | DCS | ICS |

*Figure 6: Cache control register*

Field Definitions:

- [31:30]: Data cache replacement policy (DREPL) - 00 - no replacement policy (direct-mapped cache), 01 - random, 10 - least-recently replaced (LRR), 11 - least-recently used (LRU)
- [29:28]: Instruction cache replacement policy (IREPL) - 00 - no replacement policy (direct-mapped cache), 01 - random, 10 - least-recently replaced (LRR), 11 - least-recently used (LRU)
- [27:26]: Instruction cache associativity (ISETS) - Number of sets in the instruction cache - 1: 00 - direct mapped, 01 - 2-way associative, 10 - 3-way associative, 11 - 4-way associative
- [25:24]: Data cache associativity (DSETS) - Number of sets in the data cache - 1: 00 - direct mapped, 01 - 2-way associative, 10 - 3-way associative, 11 - 4-way associative
- [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.
- [20:17] Reserved
- [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.
- [13:6]: Reserved
- [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.

# 5 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.

## 5.1 ASI mappings

When the MMU is used, the following ASI mappings are made:

| **ASI** | **Usage** |
|---|---|
| 0x5 | Flush instruction cache |
| 0x6 | Flush data cache |
| 0x8, 0x9, 0xA, 0xB | Normal cached access (replace if cacheable) |
| 0xC | Instruction cache tags |
| 0xD | Instruction cache data |
| 0xE | Data cache tags |
| 0xF | Data cache data |
| 0x10 | Flush page |
| 0x13 | Flush context |
| 0x19 | MMU registers |
| 0x1C | MMU bypass |

*Table 7: MMU ASI usage*

## 5.2 Caches

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.

## 5.3 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 8: MMU registers (ASI = 0x19)*

The definition of the registers can be found in the SPARC V8 manual.

## 5.4 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.

# 6 AMBA on-chip buses

## 6.1 Overview

Two on-chip buses are provided: AMBA AHB and APB. The APB bus is used to access on-chip registers in the peripheral functions, while the AHB bus is used for high-speed data transfers. The specification for the AMBA bus can be downloaded from ARM, at: www.arm.com. The full AHB/APB standard is implemented and the AHB/APB bus controllers can be customised through the TARGET package. Additional (user defined) AHB/APB peripherals should be added in the MCORE module (see “Model hierarchy” on page 76).

## 6.2 AHB bus

LEON uses the AMBA-2.0 AHB bus to connect the processor cache controllers to the memory controller and other (optional) high-speed units. In the default configuration, the processor is the only master on the bus, while two slaves are provided: the memory controller and the APB bridge. Table 9 below shows the default address allocation. An attempt to access a non-existing device will generate an AHB error response.

| Address range | Size | Mapping | Module |
|---|---|---|---|
| 0x00000000 - 0x1FFFFFFF<br>0x20000000 - 0x3FFFFFFF<br>0x40000000 - 0x7FFFFFFF | 512 M<br>512 M<br>1 G | Prom<br>Memory bus I/O<br>SRAM and/or SDRAM | Memory controller |
| 0x80000000 - 0x8FFFFFFF | 256 M | On-chip registers | APB bridge |
| 0x90000000 - 0x9FFFFFFF | 256 M | Debug support unit | DSU |
| | | | |
| 0xB0000000 - 0xB001FFFF | 128 K | Ethernet MAC registers | Ethernet |
| | | | |

*Table 9: Default AHB address allocation*

## 6.3 APB bus

The APB bridge is connected to the AHB bus as a slave and acts as the (only) master on the APB bus. Most on-chip peripherals are accessed through the APB bus. The address mapping of the APB bus can be seen in table 10.

## 6.4 AHB transfers generated by the processor

The processor is connected to the AHB bus through the instruction and data cache controllers. Access conflicts between the two cache controllers are resolved locally and only one AHB master interface is connected to the AHB bus. The processor will perform burst transfers to fetch instruction cache lines or reading/writing data as results of double load/store instructions. Byte, half-word and word load/store instructions will perform single (non-sequential accesses. Locked transfers are only performed on LDST and SWAP instructions. Double load/store transfers are however also guaranteed to be atomic since the arbiter will not re-arbitrate the bus during burst transfers.

# 7 On-chip peripherals

## 7.1 On-chip registers

A number of system support functions are provided directly on-chip. The functions are controlled through registers mapped APB bus according to the following table:

| Address | Register | Address | |
|---|---|---|---|
| 0x80000000 | Memory configuration register 1 | 0x800000B0 | Secondary interrupt mask register |
| 0x80000004 | Memory configuration register 2 | 0x800000B4 | Secondary interrupt pending register |
| 0x80000008 | Memory configuration register 3 | 0x800000B8 | Secondary interrupt status register |
| 0x8000000C | AHB Failing address register | 0x800000B8 | Secondary interrupt clear register |
| 0x80000010 | AHB status register | | |
| 0x80000014 | Cache control register | 0x800000C4 | DSU UART status register |
| 0x80000018 | Power-down register | 0x800000C8 | DSU UART control register |
| 0x8000001C | Write protection register 1 | 0x800000CC | DSU UART scaler register |
| 0x80000020 | Write protection register 2 | | |
| 0x80000024 | LEON configuration register | | |
| 0x80000040 | Timer 1 counter register | | |
| 0x80000044 | Timer 1 reload register | | |
| 0x80000048 | Timer 1 control register | | |
| 0x8000004C | Watchdog register | | |
| 0x80000050 | Timer 2 counter register | | |
| 0x80000054 | Timer 2 reload register | | |
| 0x80000058 | Timer 2 control register | | |
| 0x80000060 | Prescaler counter register | | |
| 0x80000064 | prescaler reload register | | |
| 0x80000070 | Uart 1 data register | | |
| 0x80000074 | Uart 1 status register | | |
| 0x80000078 | Uart 1 control register | | |
| 0x8000007C | Uart 1 scaler register | | |
| 0x80000080 | Uart 2 data register | | |
| 0x80000084 | Uart 2 status register | | |
| 0x80000088 | Uart 2 control register | | |
| 0x8000008C | Uart 2 scaler register | | |
| 0x80000090 | Interrupt mask and priority register | | |
| 0x80000094 | Interrupt pending register | | |
| 0x80000098 | Interrupt force register | | |
| 0x8000009C | Interrupt clear register | | |
| 0x800000A0 | I/O port input/output register | | |
| 0x800000A4 | I/O port direction register | | |
| 0x800000A8 | I/O port interrupt config. register | | |

*Table 10: On-chip registers (APB bus)*

## 7.2 Interrupt controller

The LEON interrupt controller is used to prioritize and propagate interrupt requests from internal or external devices to the integer unit. In total 15 interrupts are handled, divided on two priority levels. Figure 7 shows a block diagram of the interrupt controller.

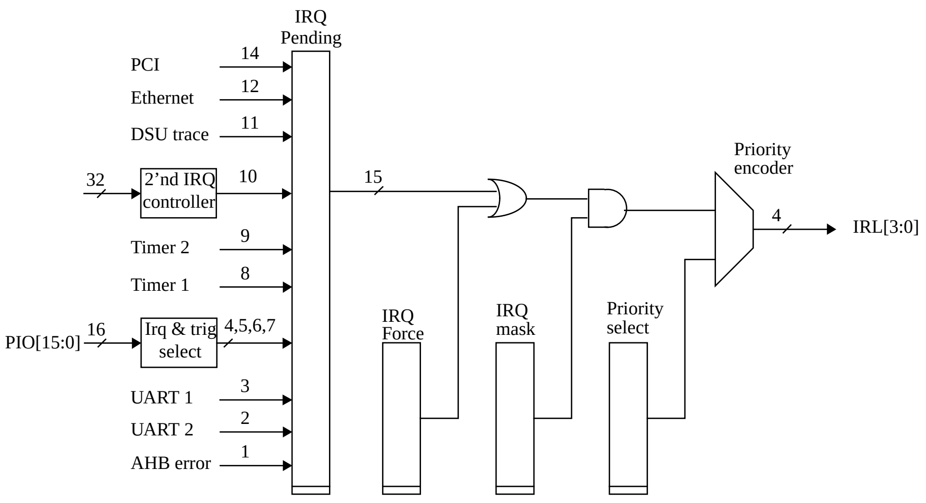


*Figure 7: Interrupt controller block diagram*

### 7.2.1 Operation

When an interrupt is generated, the corresponding bit is set in the interrupt pending register. The pending bits are ANDed with the interrupt mask register and then forwarded to the priority selector. Each interrupt can be assigned to one of two levels 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 IU - if no unmasked pending interrupt exists on level 1, then the highest unmasked interrupt from level 0 will be forwarded. When the IU 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 IU 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.

Interrupts 10 - 15 are unused by the default configuration of LEON and can be use by added IP cores. Note that interrupt 15 cannot be maskable by the integer unit and should be used with care - most operating system do not safely handle this interrupt.

### 7.2.2 Interrupt assignment

Table 11 shows the assignment of interrupts.

| Interrupt | Source |
|---|---|
| 15 | user defined |
| 14 | PCI |
| 13 | user defined |
| 12 | Ethernet MAC |
| 11 | DSU trace buffer |
| 10 | Second interrupt controller |
| 9 | Timer 2 |
| 8 | Timer 1 |
| 7 | Parallel I/O[3] |
| 6 | Parallel I/O[2] |
| 5 | Parallel I/O[1] |
| 4 | Parallel I/O[0] |
| 3 | UART 1 |
| 2 | UART 2 |
| 1 | AHB error |

*Table 11: Interrupt assignments*

### 7.2.3 Control registers

The operation of the interrupt controller is programmed through the following registers:

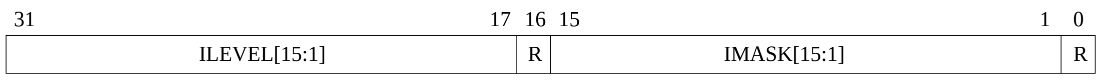


*Figure 8: Interrupt mask and priority register*

Field Definitions:

- [31:17]: Interrupt level (ILEVEL[15:1]) - indicates whether an interrupt belongs to priority level 1 (ILEVEL[n]=1) or level 0 (ILEVEL[n]=0).
- [15:1]: Interrupt mask (IMASK[15:0]) - indicates whether an interrupt is masked (IMASK[n]=0) or enabled (IMASK[n]=1).
- [16], [0]: Reserved

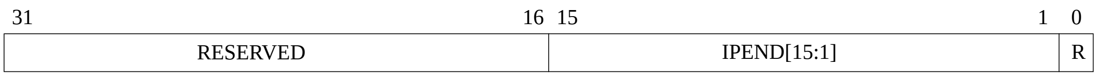


*Figure 9: Interrupt pending register*

Field Definitions:

- [15:1]: Interrupt pending (IPEND[15:1]) - indicates whether an interrupt is pending (IPEND[n]=1).
- [31:16], [0]: Reserved

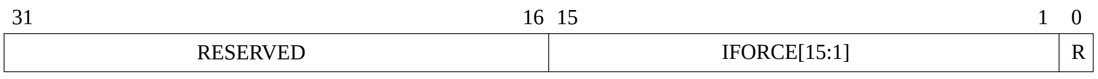


*Figure 10: Interrupt force register*

Field Definitions:

- [15:1]: Interrupt force (IFORCE[15:1]) - indicates whether an interrupt is being forced (IFORCE[n]=1).
- [31:16], [0]: Reserved

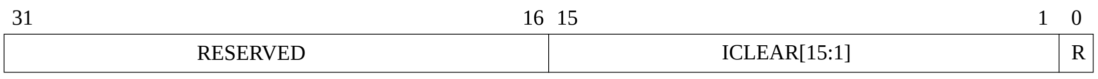


*Figure 11: Interrupt clear register*

Field Definitions:

- [15:1]: Interrupt clear (ICLEAR[15:1]) - if written with a ‘1’, will clear the corresponding bit(s) in the interrupt pending register. A read returns zero.
- [31:16], [0]: Reserved

## 7.3 Secondary interrupt controller

The (optional) secondary interrupt controller is used add up to 32 additional interrupts, to be used by on-chip units in system-on-chip designs. Figure 7 shows a block diagram of the interrupt controller.

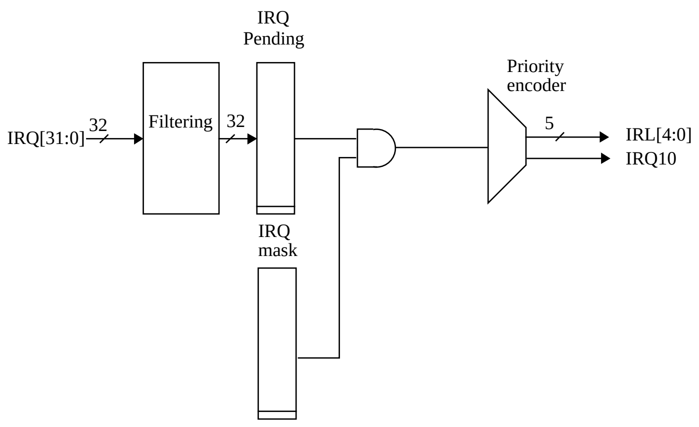


*Figure 12: Secondary interrupt controller block diagram*

### 7.3.1 Operation

The incoming interrupt signals are filtered according to the setting in the configuration record. The filtering condition can be one of four: active low, active high, negative edge-triggered and positive edge-triggered. When the condition is fulfilled, the corresponding bit is set in the interrupt pending register. The pending bits are ANDed with the interrupt mask register and then forwarded to the priority selector. If at least one unmasked pending interrupt exists, the interrupt output will be driven, generating interrupt 10 (by default). The highest unmasked pending interrupt can be read from the interrupt status register (see below).

Interrupts are not cleared automatically upon a taken interrupt - the interrupt handler must reset the pending bit by writing a ‘1’ to the corresponding bit in the interrupt clear register. It must then also clear interrupt 10 in the primary interrupt controller. Testing of interrupts can be done by writing directly to the interrupt pending registers. Bits written with ‘1’ will be set while bits written with ‘0’ will keep their previous value.

Note that not all 32 interrupts have to be implemented, how many are actually used depends on the configuration. Unused interrupts are ignored and the corresponding register bits are not generated. Mapping of interrupts to the secondary interrupt controller is done by editing mcore.vhd. See the configuration section on how to enable the controller and how to configure the interrupt filters.

After reset, the interrupt mask register is set to all zeros while the remaining control registers are undefined.

### 7.3.2 Control registers

The operation of the secondary interrupt controller is programmed through the following registers:

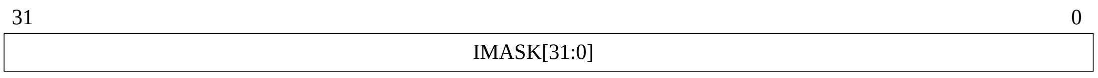


*Figure 13: Secondary interrupt mask register*

- [31:0]: Interrupt mask - indicates whether an interrupt is masked (IMASK[n]=0) or enabled (IMASK[n]=1).

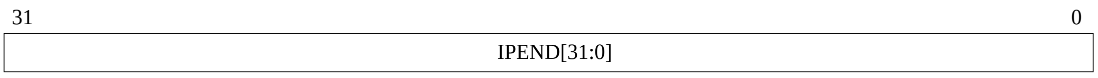


*Figure 14: Secondary interrupt pending register*

- [31:0]: Interrupt pending - indicates whether an interrupt is pending (IPEND[n]=1).

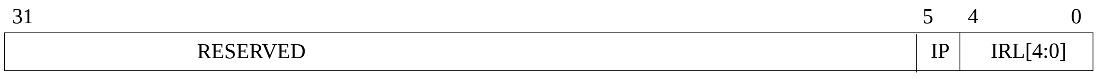


*Figure 15: Secondary interrupt status register*

- [4:0]: Interrupt request level - indicates the highest unmasked pending interrupt.
- [5]: Interrupt pending - if set, then IRL is valid. If cleared, no unmasked interrupt is pending.

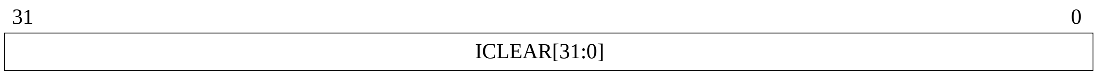


*Figure 16: Secondary interrupt clear register*

- [31:0]: Interrupt clear - if written with a '1', will clear the corresponding bit(s) in the interrupt pending register.

## 7.4 Timer unit

The timer unit implements two 24-bit timers, one 24-bit watchdog and one 10-bit shared prescaler (figure 17).

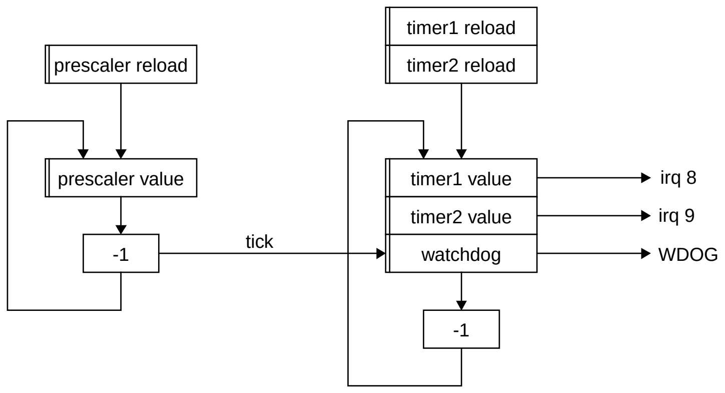


*Figure 17: Timer unit block diagram*

### 7.4.1 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 for the two timers and watchdog. The effective division rate is therefore equal to prescaler reload register value + 1.

The operation of the timers is controlled through the timer control register. A timer is enabled by setting the enable bit in the control register. The timer value is then decremented each time the prescaler generates a timer tick. When a timer underflows, it will automatically be reloaded with the value of the timer reload register if the reload bit is set, otherwise it will stop (at 0xffffff) and reset the enable bit. An interrupt will be generated after each underflow.

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

The watchdog operates similar to the timers, with the difference that it is always enabled and upon underflow asserts the external signal WDOG. This signal can be used to generate a system reset.

To minimise complexity, the two timers and watchdog share the same decrementer. This means that the minimum allowed prescaler division factor is 4 (reload register = 3).

### 7.4.2 Registers

Figures 18 to 22 shows the layout of the timer unit registers.

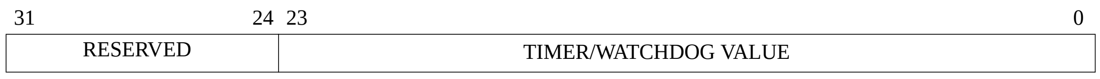


*Figure 18: Timer 1/2 and Watchdog counter registers*

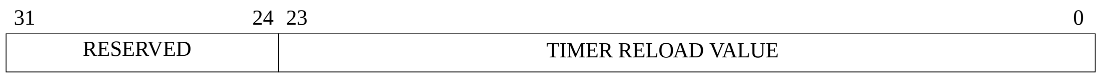


*Figure 19: Timer 1/2 reload registers*

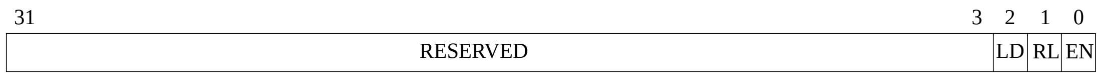


*Figure 20: Timer 1/2 control registers*

- [2]: Load counter (LD) - when written with ‘one’, will load the timer reload register into the timer counter register. Always reads as a ‘zero’.
- [1]: Reload counter (RL) - if RL is set, then the counter will automatically be reloaded with the reload value after each underflow.
- [0]: Enable (EN) - enables the timer when set.

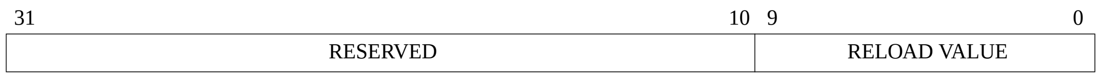


*Figure 21: Prescaler reload register*

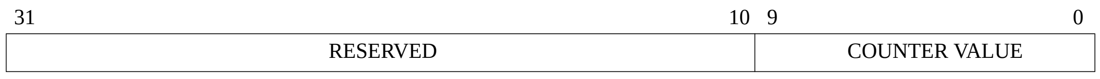


*Figure 22: Prescaler counter register*

## 7.5 UARTs

Two identical UARTs are provided for serial communications. The UARTs support 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-bits clock divider. Hardware flow-control is supported through the RTSN/CTSN hand-shake signals. Figure 23 shows a block diagram of a UART.

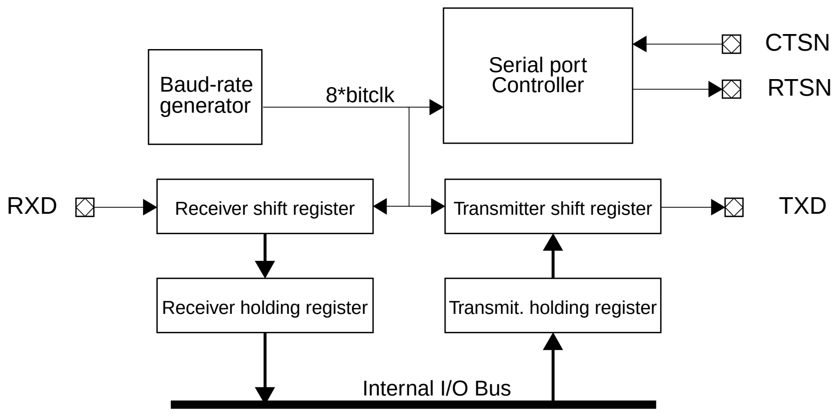


*Figure 23: UART block diagram*

### 7.5.1 Transmitter operation

The transmitter is enabled through the TE bit in the UART control register. When ready to transmit, data is transferred from the transmitter holding register 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 bits (figure 24). The least significant bit of the data is sent first

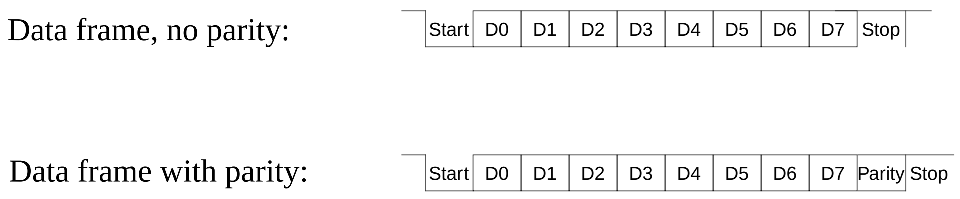

Data frame, no parity:
Start D0 D1 D2 D3 D4 D5 D6 D7 Stop
Data frame with parity:
Start D0 D1 D2 D3 D4 D5 D6 D7 Parity Stop

*Figure 24: UART data frames*

Following the transmission of the stop bit, if a new character is not available in the transmitter holding register, the transmitter serial data output remains high and the transmitter shift register empty bit (TSRE) will be set in the UART control register. Transmission resumes and the TSRE is cleared when a new character is loaded in the transmitter holding register. If the transmitter is disabled, it will continue operating until the character currently being transmitted is completely sent out. The transmitter holding register cannot be loaded when the transmitter is disabled.

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.

### 7.5.2 Receiver operation

The receiver is enabled for data reception through the receiver enable (RE) bit in the USART 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.

During reception, the least significant bit is received first. The data is then transferred to the receiver holding register (RHR) and the data ready (DR) bit is set in the USART status register. 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. If both receiver holding and shift registers contain an un-read character 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 holding register contains an un-read character. When the holding register is read, the RTSN will automatically be reasserted again.

### 7.5.3 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. 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 the EC bit is set, the scaler will be clocked by the PIO[3] input rather than the system clock. In this case, the frequency of PIO[3] must be less than half the frequency of the system clock.

### 7.5.4 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.

### 7.5.5 Interrupt generation

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.

### 7.5.6 UART registers

UART registers

| 31 8 | 7 0 |
|---|---|
| RESERVED | DATA |

*Figure 25: UART data register*

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

| 31 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|
| RESERVED | EC | LB | FL | PE | PS | TI | RI | TE | RE |

*Figure 26: 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, enables generation of receiver interrupt.
- 3: Transmitter interrupt enable (TI) - if set, enables generation of transmitter interrupt.
- 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 PIO[3]

| 31 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|
| RESERVED | | FE | PE | OV | BR | TH | TS | DR |

*Figure 27: 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 hold register empty (TH) - indicates that the transmitter hold register 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.

| 31 … 12 | 11 … 0 |
|---|---|
| RESERVED | SCALER RELOAD VALUE |

*Figure 28: UART scaler reload register*

## 7.6 Parallel I/O port

A partially bit-wise programmable 32-bit I/O port is provided on-chip. The port is split in two parts - the lower 16-bits are accessible via the PIO[15:0] signal while the upper 16-bits uses D[15:0] and can only be used when all areas (rom, ram and I/O) of the memory bus are in 8- or 16-bit mode (see “8-bit and 16-bit PROM and SRAM access” on page 52). If the SDRAM controller is enabled, the upper 16-bits cannot be used.

The lower 16 bits of the I/O port can be individually programmed as output or input, while the high 16 bits of the I/O port only be configures as outputs or inputs on byte basis. Two registers are associated with the operation of the I/O port; the combined I/O input/output register, and I/O direction register. When read, the input/output register will return the current value of the I/O port; when written, the value will be driven on the port signals (if enabled as output). The direction register defines the direction for each individual port bit (0=input, 1=output).

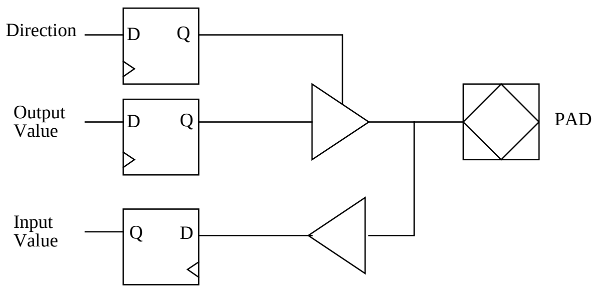


*Figure 29: I/O port block diagram*

| 31 18 | 17 0 |
|---|---|
| | IODIR[17:0] |

*Figure 30: I/O port direction register*

- IODIR*n* - I/O port direction. The value of IODIR[15:0] defines the direction of I/O ports 15 - 0. If bit *n* is set the corresponding I/O port becomes an output, otherwise it is an input. IODIR[16] controls D[15:8] while IODIR[17] controls D[7:0]

The I/O ports can also be used as interrupt inputs from external devices. A total of four interrupts can be generated, corresponding to interrupt levels 4, 5, 6 and 7. The I/O port interrupt configuration register (figure 31) defines which port should generate each interrupt and how it should be filtered.

| 31 | 30 | 29 | 28 24 | 23 | 22 | 21 | 20 16 | 15 | 14 | 13 | 12 8 | 7 | 6 | 5 | 4 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| EN | LE | PL | ISEL3 | EN | LE | PL | ISEL2 | EN | LE | PL | ISEL1 | EN | LE | PL | ISEL0 |

*Figure 31: I/O port interrupt configuration register*

- ISEL*n* - I/O port select. The value of this field defines which I/O port (0 - 31) should generate parallel I/O port interrupt *n*.
- PL - Polarity. If set, the corresponding interrupt will be active high (or edge-triggered on positive edge). Otherwise, it will be active low (or edge-triggered on negative edge).
- LE - Level/edge triggered. If set, the interrupt will be edge-triggered, otherwise level sensitive.
- EN - Enable. If set, the corresponding interrupt will be enabled, otherwise it will be masked.

To save pins, I/O pins are shared with other functions according to the table below:

| **I/O port** | **Function** | **Type** | **Description** | **Enabling condition** |
|---|---|---|---|---|
| PIO[15] | TXD1 | Output | UART1 transmitter data | UART1 transmitter enabled |
| PIO[14] | RXD1 | Input | UART1 receiver data | - |
| PIO[13] | RTS1 | Output | UART1 request-to-send | UART1 flow-control enabled |
| PIO[12] | CTS1 | Input | UART1 clear-to-send | - |
| PIO[11] | TXD2 | Output | UART2 transmitter data | UART2 transmitter enabled |
| PIO[10] | RXD2 | Input | UART2 receiver data | - |
| PIO[9] | RTS2 | Output | UART2 request-to-send | UART2 flow-control enabled |
| PIO[8] | CTS2 | Input | UART2 clear-to-send | - |
| PIO[4] | Boot select | Input | Internal or external boot prom | - |
| PIO[3] | UART clock | Input | Use as alternative UART clock | - |
| PIO[1:0] | Prom width | Input | Defines prom width at boot time | - |

*Table 12: UART/IO port usage*

## 7.7 LEON configuration register

Since LEON is synthesised from a extensively configurable VHDL model, the LEON configuration register (read-only) is used to indicate which options were enabled during synthesis. For each option present, the corresponding register bit is hardwired to ‘1’. Figure 32 shows the layout of the register.

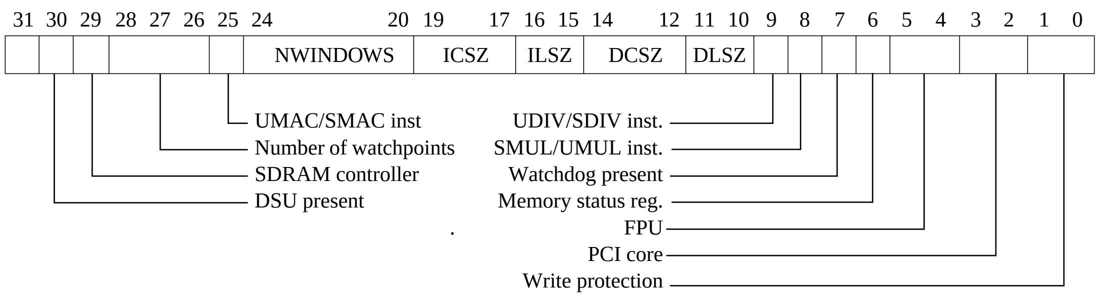


*Figure 32: LEON configuration register*

- [30]: Debug support unit (0=disabled, 1=present)
- [29]: SDRAM controller present (0=disabled, 1=present)
- [28:26]: Number of implemented watchpoints (0 - 4)
- [25]: UMAC/SMAC instruction implemented
- [24:20]: Number of register windows. The implemented number of SPARC register windows -1.
- [19:17]: Instruction cache size. The size (in Kbytes) of the instruction cache. Cache size = $2^{ICSZ}$.
- [16:15]: Instruction cache line size.The line size (in 32-bit words) of each line. Line size = $2^{ILSZ}$.
- [14:12]: Data cache size. The size (in kbytes) of the data cache. Cache size = $2^{DCSZ}$.
- [11:10]: Data cache line size. The line size (in 32-bit words) of each line. Line size = $2^{DLSZ}$.
- [9]: UDIV/SDIV instruction implemented
- [8]: UMUL/SMUL instruction implemented
- [6]: Memory status and failing address register present
- [5:4]: FPU type (00 = none, 01=Meiko)
- [3:2]: PCI core type (00=none, 01=InSilicon, 10=ESA, 11=other)
- [1:0]: Write protection type (00=none, 01=standard)

## 7.8 Power-down

The processor can be powered-down by writing (an arbitrary) value to the power-down register. Power-down mode will be entered on the next load or store instruction. To enter power-down mode immediately, a store to the power-down register should be performed followed a ‘dummy’ load. During power-down mode, the integer unit will effectively be halted. The power-down mode will be terminated (and the integer unit re-enabled) when an unmasked interrupt with higher level than the current processor interrupt level (PIL) becomes pending. All other functions and peripherals operate as nominal during the power-down mode. A suitable power-down routine could be:

```
struct pwd_reg_type { volatile int pwd; };

power_down()
{
   struct pwd_reg_type *lreg = (struct pwd_reg_type *) 0x80000018;
   while (1) lreg->pwd = lreg->pwd;
}
```

In assembly, a suitable sequence could be:

```
power_down:
set   0x80000000, %l3
st    %g0, [%l3 + 0x18]
ba    power_down
ld    [%l3 + 0x18], %g0
```

## 7.9 AHB status register

Any access triggering an error response on the AHB bus will be registered in two registers; AHB failing address register and AHB status register. The failing address register will store the address of the access while the AHB status register will store the access and error types. The registers are updated when an error occur, and the EV (error valid) is not set. When the EV bit is set, interrupt 1 is generated to inform the processor about the error. After an error, the EV bit has to be reset by software.

Figure 33 shows the layout of the AHB status register.

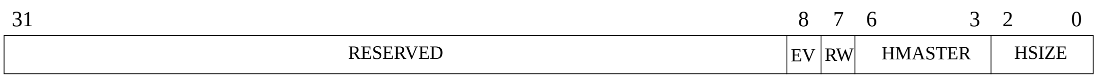


*Figure 33: AHB status register*

- [8]: NE - error valid. Set when an error occurred.
- [7]: RW - Read/Write. This bit is set if the failed access was a read cycle, otherwise it is cleared.
- [6:3]: HMASTER - AHB master. This field contains the HMASTER[3:0] of the failed access.
- [2:0] HSIZE - transfer size. This filed contains the HSIZE[2:0] of the failed transfer.

## 7.10 AHB ram

An optional RAM module can be placed on the AHB bus, providing high-speed on-chip memory. The RAM can be 1 - 64 Kbyte, and will occupy address 0x60000000 - 0x70000000 on the AHB bus. Enabling of the RAM is done in the configuration record.

# 8 External memory access

## 8.1 Memory interface

The memory bus provides a direct interface to PROM, memory mapped I/O devices, asynchronous static ram (SRAM) and synchronous dynamic ram (SDRAM). Chip-select decoding is done for two PROM banks, one I/O bank, five SRAM banks and two SDRAM banks. Figure 34 shows how the connection to the different device types is made.

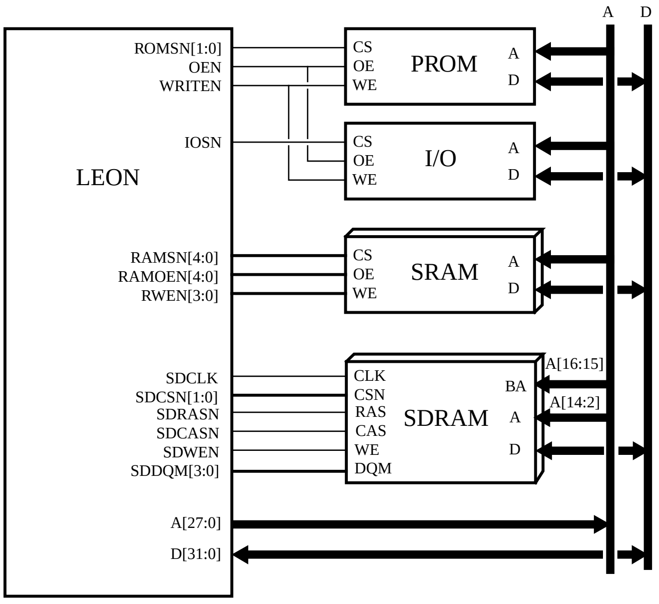


*Figure 34: Memory device interface*

## 8.2 Memory controller

The external memory bus is controlled by a programmable memory controller. The controller acts as a slave on the AHB bus. The function of the memory controller is programmed through memory configuration registers 1, 2 & 3 (MCR1, MCR2 & MCR3) through the APB bus. The memory bus supports four types of devices: prom, sram, sdram and local I/O. The memory bus can also be configured in 8- or 16-bit mode for applications with low memory and performance demands. The controller decodes a 2 Gbyte address space, divided according to table 13:

| Address range | Size | Mapping |
|---|---|---|
| 0x00000000 - 0x1FFFFFFF | 512 M | Prom |
| 0x20000000 - 0x3FFFFFFF | 512M | I/O |
| 0x40000000 -0x7FFFFFFF | 1 G | SRAM/SDRAM |

*Table 13: Memory controller address map*

## 8.3 PROM access

Accesses to prom have the same timing as RAM accesses, the differences being that PROM cycles can have up to 15 waitstates.

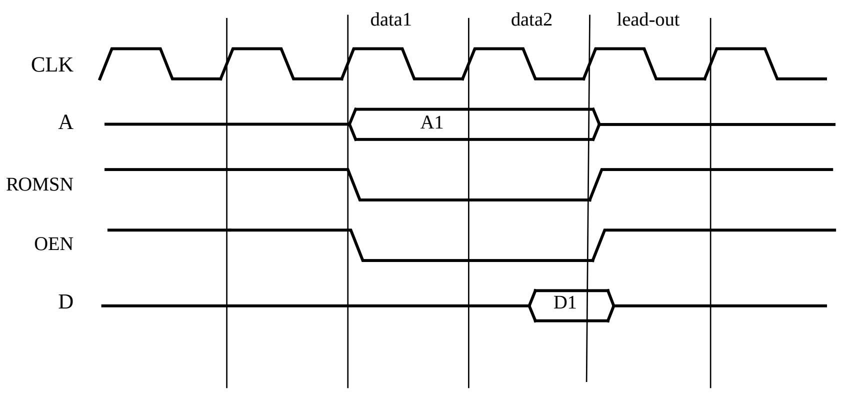


*Figure 35: Prom read cycle*

Two PROM chip-select signals are provided, ROMSN[1:0]. ROMSN[0] is asserted when the lower half (0 - 0x10000000) of the PROM area as addressed while ROMSN[1] is asserted for the upper half (0x10000000 - 0x20000000). When the VHDL model is configured to boot from internal prom (see “Boot configuration” on page 86), ROMSN[0] is never asserted and all accesses between 0 - 0x10000000 are mapped on the internal prom. When the model is configured to support both external and internal boot prom, the PIO[4] input is used to enable the internal prom.

## 8.4 Memory mapped I/O

Accesses to I/O have similar timing to ROM/RAM accesses, the differences being that a additional waitstates can be inserted by de-asserting the BRDYN signal. The I/O select signal (IOSN) is delayed one clock to provide stable address before IOSN is asserted.

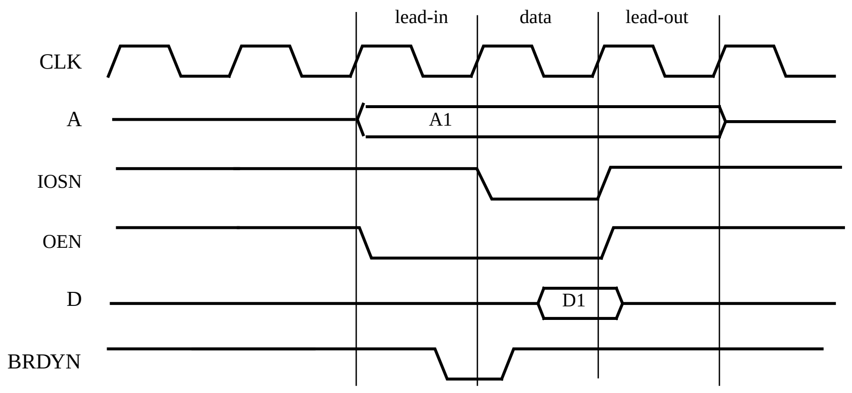


*Figure 36: I/O read cycle*

## 8.5 SRAM access

The SRAM area can be up to 1 Gbyte, divided on up to five RAM banks. The size of banks 1-4 (RAMSN[3:0]is programmed in the RAM bank-size field (MCR2[12:9]) and can be set in binary steps from 8 Kbyte to 256 Mbyte. The fifth bank (RAMSN[4]) decodes the upper 512 Mbyte. A read access to SRAM consists of two data cycles and between zero and three waitstates. Accesses to RAMSN[4] can further be stretched by de-asserting BRDYN until the data is available. On non-consecutive accesses, a lead-out cycle is added after a read cycle to prevent bus contention due to slow turn-off time of memories or I/O devices. Figure 37 shows the basic read cycle waveform (zero waitstate).

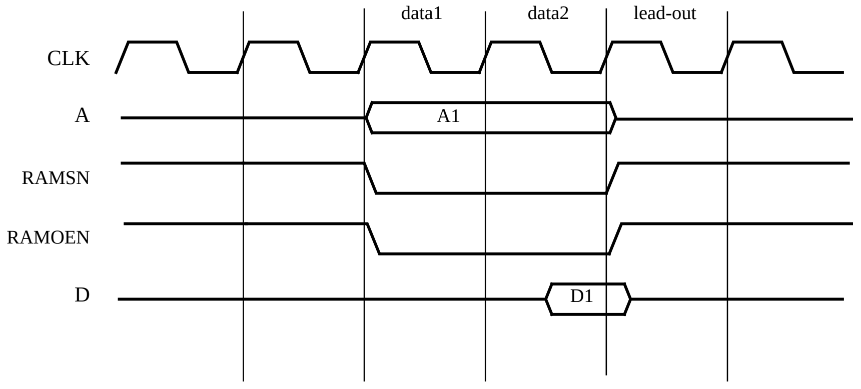


*Figure 37: Static ram read cycle (0-waitstate)*

For read accesses to RAMSN[4:0], a separate output enable signal (RAMOEN[n]) is provided for each RAM bank and only asserted when that bank is selected. A write access is similar to the read access but takes a minimum of three cycles:

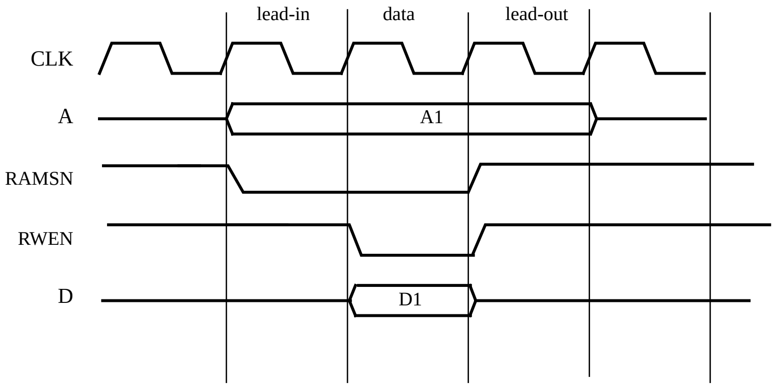


*Figure 38: Static ram write cycle*

Through an (optional) feed-back loop from the write strobes, the data bus is guaranteed to be driven until the write strobes are de-asserted. Each byte lane has an individual write strobe to allow efficient byte and half-word writes. If the memory uses a common write strobe for the full 16- or 32-bit data, the read-modify-write bit MCR2 should be set to enable read-modify-write cycles for sub-word writes.

## 8.6 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, double loads and double stores. 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.

## 8.7 8-bit and 16-bit PROM and SRAM access

To support applications with low memory and performance requirements efficiently, it is not necessary to always have full 32-bit memory banks. The SRAM and PROM areas can be individually configured for 8- or 16-bit operation by programming the ROM and RAM size fields in the memory configuration registers. Since read access to memory is always done on 32-bit word basis, read access to 8-bit memory will be transformed in a burst of four read cycles while access to 16-bit memory will generate a burst of two 16-bits reads. During writes, only the necessary bytes will be writen. Figure 39 shows an interface example with 8-bit PROM and 8-bit SRAM. Figure 40 shows an example of a 16-bit memory interface.

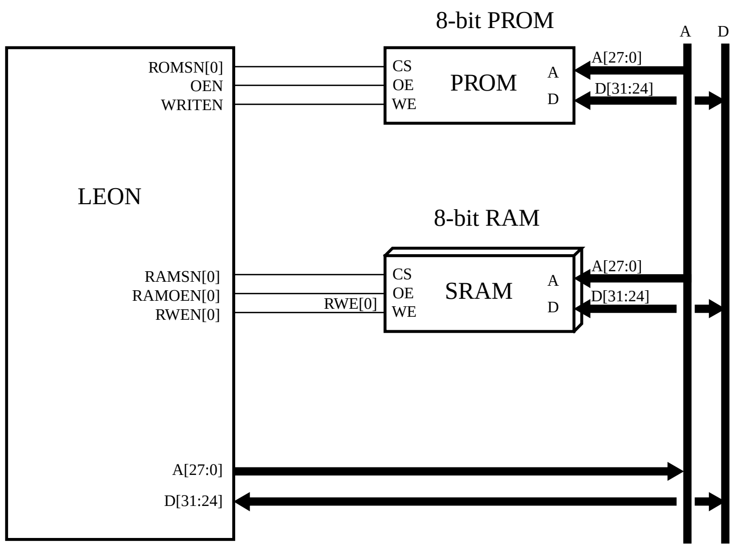


*Figure 39: 8-bit memory interface example*

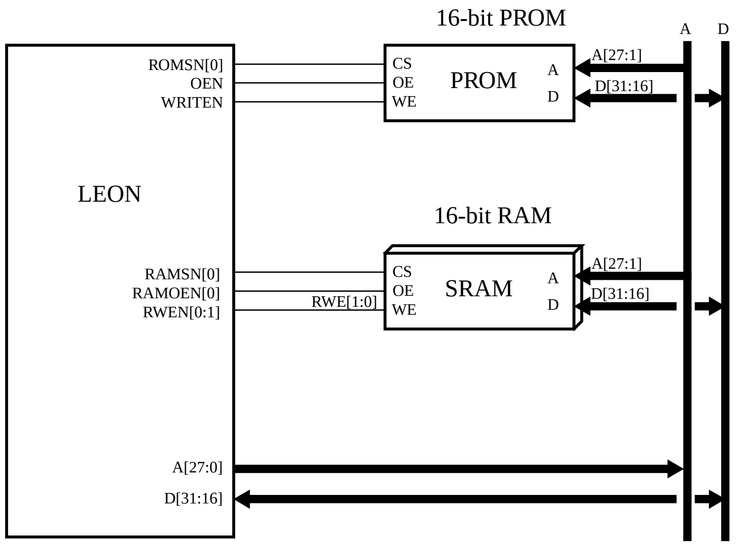


*Figure 40: 16-bit memory interface example*

## 8.8 8- and 16-bit I/O access

Similar to the PROM/RAM areas, the I/O area can also be configured to 8- or 16-bits mode. However, the I/O device will NOT be accessed by multiple 8/16 bits accesses as the memory areas, but only with one single access just as in 32-bit mode. To accesses an I/O device on a 16-bit bus, LDUH/STH instructions should be used while LDUB/STB should be used with an 8-bit bus.

## 8.9 SDRAM access

### 8.9.1 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 MCFG2 and MCFG3 (see below).

The SDRAM has a separate address and control bus. The SDRAM data signals can either be attached using the common data bus (DATA[31:0]), or be attached via a dedicated SDRAM data bus (SD[63:0]). In the later case, the SDRAM can be either 32 or 64 bits wide. The data width is selected during the configuration of the VHDL model, and cannot be change later by software.

### 8.9.2 Address mapping

The two SDRAM banks can be mapped starting at address 0x40000000 or 0x60000000. When the SDRAM enable bit is set in MCFG2, the controller is enabled and mapped at

0x60000000 as long as the SRAM disable bit is not set. If the SRAM disable bit is set, all access to SRAM is disabled and the SDRAM banks are mapped starting at 0x40000000.

### 8.9.3 Initialisation

When the SDRAM controller is enabled, it automatically performs the SDRAM initialisation 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.

### 8.9.4 Configurable SDRAM timing parameters

To provide optimum access cycles for different SDRAM devices (and at different frequencies), some SDRAM parameters can be programmed through memory configuration register 2 (MCFG2) The programmable SDRAM parameters can be seen in table 14:

| 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 |

*Table 14: SDRAM programmable timing parameters*

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

### 8.9.5 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 MCFG3 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 MCFG2.

### 8.9.6 SDRAM commands

The controller can issue three SDRAM commands by writing to the SDRAM command field in MCFG2: PRE-CHARGE, AUTO-REFRESH and LOAD-MODE-REG (LMR). If the LMR command is issued, the CAS delay as programmed in MCFG2 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.

### 8.9.7 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.

### 8.9.8 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.

### 8.9.9 Address bus connection

The memory controller can be configured to either share the address and data buses with the SRAM, or to use separate address and data buses. When the buses are shared, the address bus of the SDRAMs should be connected to A[14:2], the bank address to A[16:15]. The MSB part of A[14:2] can be left unconnected if not used. When separate buses are used, 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].

### 8.9.10 Clocking

The SDRAM clock is generated by LEON, and typically requires special synchronisation at layout level. For Virtex targets, enabling the CLKDLL or DCM will also produce a properly synchronised SDRAM clock. For other FPGA targets, the inverted clock option can be used.

## 8.10 Memory configuration register 1 (MCFG1)

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

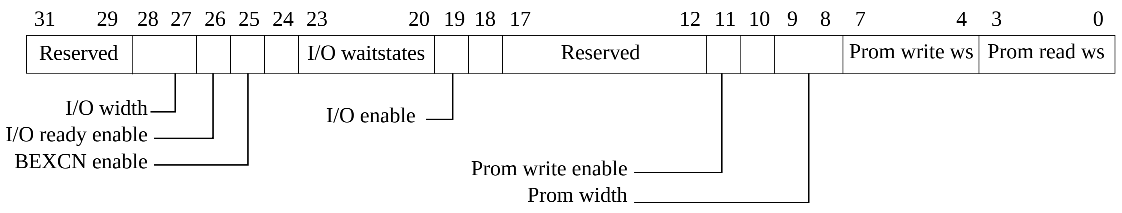


*Figure 41: Memory configuration register 1*

- [3:0]: Prom read waitstates. Defines the number of waitstates during prom read cycles (“0000”=0, “0001”=1,... “1111”=15).
- [7:4]: Prom write waitstates. Defines the number of waitstates during prom write cycles (“0000”=0, “0001”=1,... “1111”=15).
- [9:8]: Prom width. Defines the data with of the prom area (“00”=8, “01”=16, “10”=32).
- [10]: Reserved
- [11]: Prom write enable. If set, enables write cycles to the prom area.
- [17:12]: Reserved
- [19]: I/O enable. If set, the access to the memory bus I/O area are enabled.
- [23:20]: I/O waitstates. Defines the number of waitstates during I/O accesses (“0000”=0, “0001”=1, “0010”=2,..., “1111”=15).
- [25]: Bus error (BEXCN) enable.
- [26]:Bus ready (BRDYN) enable.
- [28:27]: I/O bus width. Defines the data with of the I/O area (“00”=8, “01”=16, “10”=32).

During power-up, the prom width (bits [9:8]) are set with value on PIO[1:0] inputs. The prom waitstates fields are set to 15 (maximum). External bus error and bus ready are disabled. All other fields are undefined.

## 8.11 Memory configuration register 2 (MCFG2)

Memory configuration register 2 is used to control the timing of the SRAM and SDRAM.

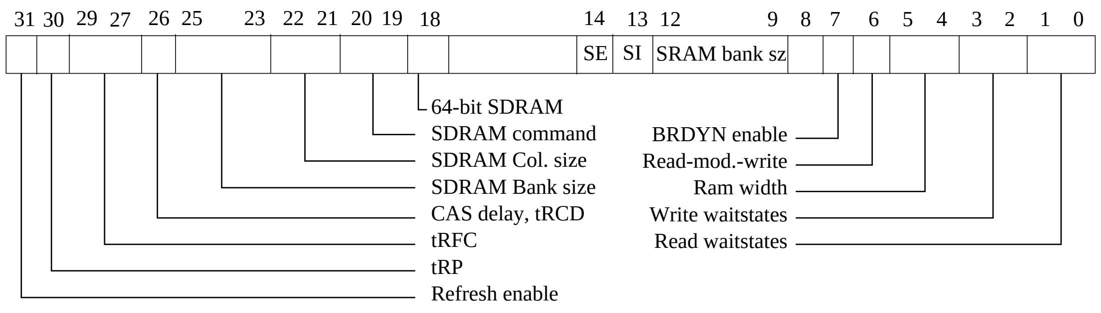


*Figure 42: Memory configuration register 2*

- [1:0]: Ram read waitstates. Defines the number of waitstates during ram read cycles ("00"=0, "01"=1, "10"=2, "11"=3).
- [3:2]: Ram write waitstates. Defines the number of waitstates during ram write cycles ("00"=0, "01"=1, "10"=2, "11"=3).
- [5:4]: Ram with. Defines the data with of the ram area ("00"=8, "01"=16, "1X"= 32).
- [6]: Read-modify-write. Enable read-modify-write cycles on sub-word writes to 16- and 32-bit areas with common write strobe (no byte write strobe).
- [7]: Bus ready enable. If set, will enable BRDYN for ram area
- [12:9]: Ram bank size. Defines the size of each ram bank ("0000"=8 Kbyte, "0001"=16 Kbyte... "1111"=256 Mbyte).
- [13]: SI - SRAM disable. If set together with bit 14 (SDRAM enable), the static ram access will be disabled.
- [14]: SE - SDRAM enable. If set, the SDRAM controller will be enabled.
- [18]: 64-bit SDRAM. If set, the SDRAM controller is configured to use 64-bit wide SDRAM.
- [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.

## 8.12 Memory configuration register 3 (MCFG3)

MCFG3 is contains the reload value for the SDRAM refresh counter.

| 31 27 | 26 12 | 11 0 |
|---|---|---|
| RESERVED | SDRAM refresh reload value | RESERVED |

*Figure 43: Memory configuration register 3*

The period between each AUTO-REFRESH command is calculated as follows:

$t_{REFRESH}$ = ((reload value) + 1) / SYSCLK

## 8.13 Write protection

Write protection is provided to protect the RAM area against accidental over-writing. It is implemented as two block protect units capable of disabling or enabling write access to a binary aligned memory block in the range of 32 Kbyte - 1 Gbyte. Each block protect unit is controlled through a control register (figure 44). The units operate as follows: on each write access to RAM, address bits (29:15) are xored with the tag field in the control register, and anded with the mask field. A write protection error is generated if the result is equal to zero, the corresponding unit is enabled and the block protect bit (BP) is set, or if the BP bit is cleared and the result is not equal to zero. If a write protection error is detected, the write cycle is aborted and a memory access error is generated.

| 31 | 30 | 29 15 | 14 0 |
|---|---|---|---|
| EN | BP | TAG[14:0] | MASK[14:0] |

*Figure 44: Write protection register 1 & 2*

- [14:0] Address mask (MASK) - this field contains the address mask
- [29:15] Address tag (TAG) - this field is compared against address(29:15)
- [30] Block protect (BP) - if set, selects block protect mode
- [31] Enable (EN) - if set, enables the write protect unit

The ROM area can be write protected by clearing the write enable bit MCR1.

## 8.14 Using BRDYN

The BRDYN signal can be used to stretch access cycles to the I/O area and the ram area decoded by RAMSN[4]. The accesses will always have at least the pre-programmed number of waitstates as defined in memory configuration registers 1 & 2, but will be further stretched until BRDYN is asserted. BRDYN should be asserted in the cycle preceding the last one. The use of BRDYN can be enabled separately for the I/O and RAM areas.

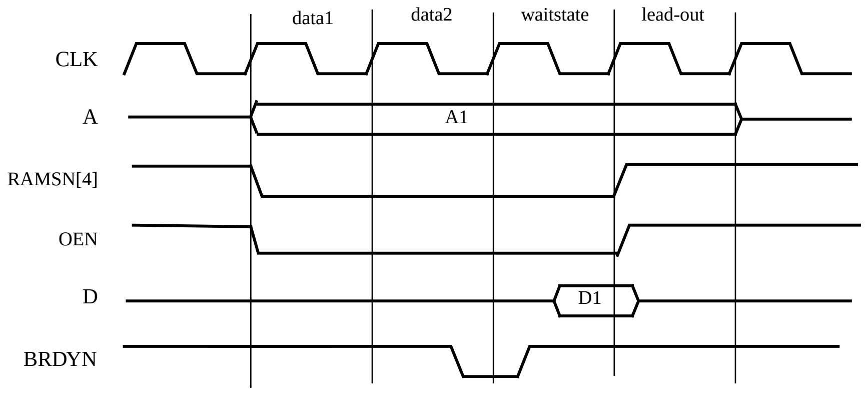


*Figure 45: RAM read cycle with one BRDYN controlled waitstate*

## 8.15 Access errors

An access error can be signalled by asserting the BEXCN signal, which is sampled together with the data. If the usage of BEXCN is enabled in memory configuration register 1, an error response will be generated on the internal AMBA bus. BEXCN can be enabled or disabled through memory configuration register 1, and is active for all areas (PROM, I/O an RAM).

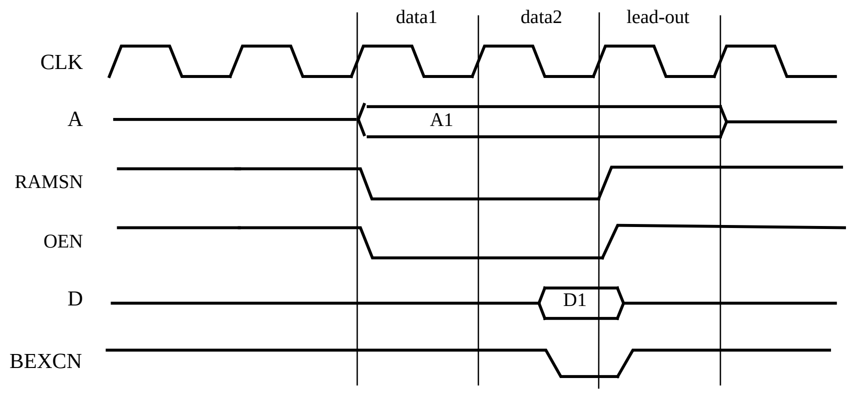


*Figure 46: Read cycle with BEXCN*

## 8.16 Attaching an external DRAM controller

To attach an external DRAM controller, RAMSN[4] should be used since it allows the cycle time to vary through the use of BRDYN. In this way, delays can be inserted as required for opening of banks and refresh.

# 9 PCI interface

The LEON2-XST model includes one optional PCI interface: a simple target-only interface. The interface is developed primarily to support DSU communications 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.

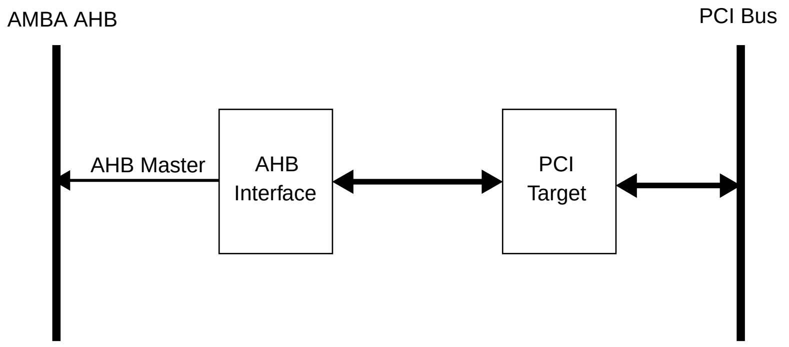


*Figure 47: Target-only PCI interface*

The interface consist of one PCI memory BAR occupying 2 Mbyte of the PCI address space, and an AHB address register. Any access to the first 1 Mbyte of the address space (0 - 0xFFFFF) will be forwarded to the internal AHB bus. The AHB address will be formed by concatenating the 12 MSB bits of the AHB address register with the LSB 20 bits of the PCI address. An access to the upper 1 Mbyte (0x100000 - 0x1FFFFF) of the BAR will read or write the AHB address register.

| 31 - 20 | 19 - 0 |
|---|---|
| AHB address [31:20] | UNUSED |

*Figure 48: AHB address register (BAR0, 0x100000)*

# 10 Ethernet interface

The LEON model includes an optional ethernet interface based on the 10/100 Mbit ethernet MAC from OpenCores. The ethernet MAC has one AHB slave interface and one AHB master interface. The slave interface is mapped at address 0xB0000000 as indicated in table 15 below. The AHB master interface is used by the MAC DMA engine to transfer ethernet packets to and from memory. The interrupt generated by the ethernet MAC is routed to interrupt 12 on the LEON interrupt controller.

| **AHB address** | **Ethernet MAC** |
|---|---|
| 0xB0000000 - 0xB00000FF | Control registers |
| 0xB0000400 - 0xB00007FF | Transmitter & receiver descriptors |
| | |

*Table 15: AHB / Ethernet MAC mapping*

For details on the ethernet MAC itself, see doc/ethernet.pdf or refer to the documentation on www.opencores.org.

# 11 Hardware debug support

## 11.1 Overview

The LEON processor includes hardware debug support to aid software debugging on target hardware. The support is provided through two modules: a debug support unit (DSU) and a debug communication link (DCL). The DSU can put the processor in debug mode, allowing read/write access to all processor registers and cache memories. The DSU also contains a trace buffer which stores executed instructions and/or data transfers on the AMBA AHB bus. The debug communications link implements a simple read/write protocol and uses standard asynchronous UART communications (RS232C).

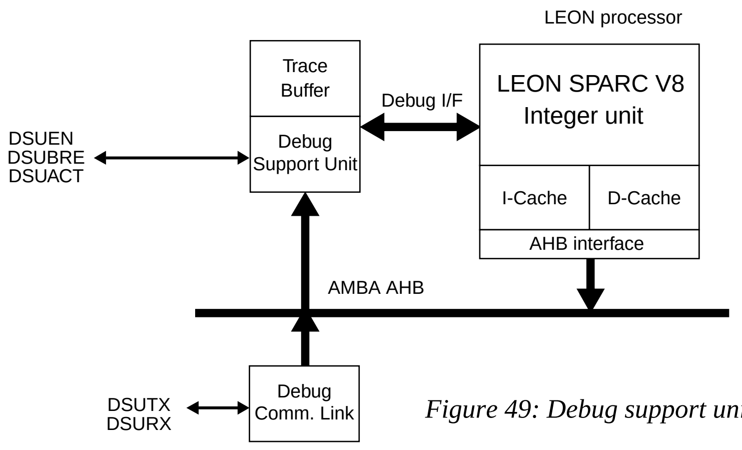


*Figure 49: Debug support unit and comm. link*

## 11.2 Debug support unit

### 11.2.1 Overview

The debug support unit is used to control the trace buffer and the processor debug mode. The DSU is attached to the AHB bus as slave, occupying a 2 Mbyte address space. Through this address space, any AHB master can access the processor registers and the contents of the trace buffer. The DSU control registers can be accessed at any time, while the processor registers and caches can only be accessed when the processor has entered debug mode. The trace buffer can be accessed only when tracing is disabled/completed. 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
- 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.

### 11.2.2 Trace buffer

The trace buffer consists of a circular buffer that stores executed instructions and/or AHB data transfers. A 30-bit counter is also provided and stored in the trace as time tag. The trace buffer operation is controlled through the DSU control register and the Trace buffer control register (see below). When the processor enters debug mode, tracing is suspended. The size of the trace buffer is by default 128 lines ( = 2 kbyte), but can be configured to 8 - 4096 lines the VHDL model configuration record.

Each lihe in the trace buffer is 128 bits wide, and the information stored is indicated in table 16 and table 17 below:

| **Bits** | **Name** | **Definition** |
|---|---|---|
| 127 | AHB breakpoint hit | Set to ‘1’ if a DSU AHB breakpoint hit occurred. |
| 126 | - | Unused |
| 125:96 | Time tag | The value of the time tag counter |
| 95:92 | IRL | Processor interrupt request input |
| 91:88 | PIL | Processor interrupt level (psr.pil) |
| 87:80 | Trap type | Processor trap type (psr.tt) |
| 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 |

*Table 16: Trace buffer data allocation, AHB tracing mode*

| Bits | Name | Definition |
|---|---|---|
| 127 | Instruction breakpoint hit | Set to ‘1’ if a DSU instruction breakpoint hit occurred. |
| 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 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 |

*Table 17: Trace buffer data allocation, Instruction tracing mode*

During instruction 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 a trace is frozen, interrupt 11 is generated.

The DSU 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.

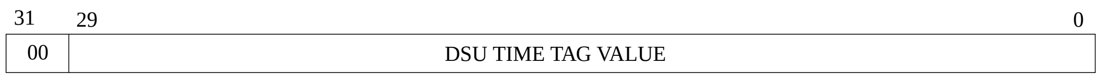


*Figure 50: DSU time tag counter*

The trace buffer control register contains two counters that contain the next address of the trace buffer to be written. Since the buffer is circular, it actually points to the oldest entry in the buffer. The counters are automatically incremented after each stored trace entry.

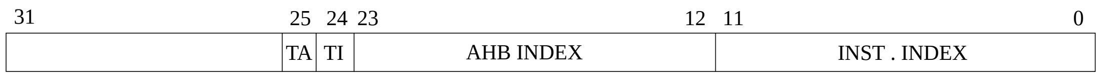


*Figure 51: Trace buffer control register*

- [11:0]: Instruction trace index counter
- [23:12]: AHB trace index counter
- [24]: Trace instruction enable
- [25]: Trace AHB enable

When both instructions and AHB transfers are traced ('mixed mode tracing'), the buffer is divided on two halves. Instructions are stored in the lower half and AHB transfers in the upper half of the buffer. The MSB bit of the AHB index counter is then automatically kept high, while the MSB of the instruction index counter is kept low.

Note that the VHDL model configuration allows to disable the mixed-mode capability. In this case, only the instruction trace index counter is provided, and is used also during AHB tracing. Setting both TA and TI bits in the trace buffer control register is then illegal.

### 11.2.3 DSU memory map

DSU memory map can be seen in table 18 below.

| Address | Register |
|---:|---|
| 0x90000000 | DSU control register |
| 0x90000004 | Trace buffer control register |
| 0x90000008 | Time tag counter |
| 0x90000010 | AHB break address 1 |
| 0x90000014 | AHB mask 1 |
| 0x90000018 | AHB break address 2 |
| 0x9000001C | AHB mask 2 |
| 0x90010000 - 0x90020000 | Trace buffer |
| ..0 | Trace bits 127 - 96 |
| ...4 | Trace bits 95 - 64 |
| ...8 | Trace bits 63 - 32 |
| ...C | Trace bits 31 - 0 |
| 0x90020000 - 0x90040000 | IU/FPU register file |
| 0x90080000 - 0x90100000 | IU special purpose registers |
| 0x90080000 | Y register |
| 0x90080004 | PSR register |
| 0x90080008 | WIM register |
| 0x9008000C | TBR register |
| 0x90080010 | PC register |
| 0x90080014 | NPC register |
| 0x90080018 | FSR register |
| 0x9008001C | DSU trap register |
| 0x90080040 - 0x9008007C | ASR16 - ASR31 (when implemented) |
| 0x90100000 - 0x90140000 | Instruction cache tags |
| 0x90140000 - 0x90160000 | Instruction cache data |
| 0x90160000 - 0x90180000 | Local instruction ram (instruction scratch pad ram) |
| 0x90180000 - 0x901C0000 | Data cache tags |
| 0x901C0000 - 0x901F0000 | Data cache data |
| 0x901F0000 - 0x90200000 | Local data ram (data scratch pad ram) |
| 0x90300000 - 0x90340000 | Instruction cache context field (MMU only) |
| 0x90380000 - 0x903C0000 | Data cache context field (MMU only) |

*Table 18: DSU address space*

The addresses of the IU/FPU registers depends on how many register windows has been implemented and if and what type of FPU is present. The registers can be accessed at the following addresses (NWINDOWS = number of SPARC register windows):

- %o*n* : 0x90020000 + (((psr.cwp * 64) + 32 + *n*) mod (NWINDOWS*64))
- %l*n* : 0x90020000 + (((psr.cwp * 64) + 64 + *n*) mod (NWINDOWS*64))
- %i*n* : 0x90020000 + (((psr.cwp * 64) + 96 + *n*) mod (NWINDOWS*64))
- %g*n* : 0x90020000 + (NWINDOWS*64) (no FPU)
- %g*n* : 0x90020000 + (NWINDOWS*64) + 128 (Meiko/LTH FPU present)
- %f*n* : 0x90020000 + (NWINDOWS*64) (Meiko/LTH FPU)
- %f*n* : 0x90030000 (GRFPU)

When the MMU is present, the following MMU registers can be accessed by the DSU:

| Address | Register |
|---:|---|
| 0x901E0000 | MMU control register |
| 0x901E0004 | MMU context register |
| 0x901E0008 | MMU context table pointer register |
| 0x901E000C | MMU fault status register |
| 0x901E0010 | MMU fault address register |

*Table 19: MMU registers address space*

### 11.2.4 DSU control register

The DSU is controlled by the DSU control register:

| 31 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| DCNT | RE | DR | LR | SS | PE | EE | EB | DM | DE | BZ | BX | BB | BN | BS | BW | BE | FT | BT | DM | TE |

*Figure 52: DSU control register*

- 0: Trace enable (TE). Enables the trace buffer.
- 1: Delay counter mode (DM). In mixed tracing mode, setting this bit will cause the delay counter to decrement on AHB traces. If reset, the delay counter will decrement on instruction traces.
- 2: Break on trace (BT) - if set, will generate a DSU break condition on trace freeze.
- 3: Freeze timers (FT) - if set, the scaler in the LEON timer unit will be stopped during debug mode to preserve the time for the software application.
- 4: Break on error (BE) - if set, will force the processor to debug mode when the processor would have entered error condition (trap in trap).
- 5: Break on IU watchpoint - if set, debug mode will be forced on a IU watchpoint (trap 0xb).
- 6: Break on S/W breakpoint (BS) - if set, debug mode will be forced when an breakpoint instruction (ta 1) is executed.
- 7: Break now (BN) -Force processor into debug mode. If cleared, the processor will resume execution.
- 8: Break on DSU breakpoint (BD) - if set, will force the processor to debug mode when an DSU breakpoint is hit.
- 9: Break on trap (BX) - if set, will force the processor into debug mode when any trap occurs.
- 10: 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.
- 11: Delay counter enable (DE) - if set, the trace buffer delay counter will decrement for each stored trace. This bit is set automatically when an DSU breakpoint is hit and the delay counter is not equal to zero.
- 12: Debug mode (DM). Indicates when the processor has entered debug mode (read-only).
- 13: EB - value of the external DSUBRE signal (read-only)
- 14: EE - value of the external DSUEN signal (read-only)
- 15: Processor error mode (PE) - returns ‘1’ on read when processor is in error mode, else ‘0’.

- 16: Single step (SS) - if set, the processor will execute one instruction and the return to debug mode.
- 17: Link response (LR) - is set, the DSU communication link will send a response word after AHB transfer.
- 18: Debug mode response (DR) - if set, the DSU communication link will send a response word when the processor enters debug mode.
- 19: Reset error mode (RE) - if set, will clear the error mode in the processor.
- 31:20 Trace buffer delay counter (DCNT). Note that the number of bits actually implemented depends on the size of the trace buffer.

### 11.2.5 DSU breakpoint registers

The DSU contains two breakpoint registers for matching either AHB addresses or executed processor instructions. A breakpoint hit is typically used to freeze the trace buffer, but can also put the processor in debug mode. Freezing can be delayed by programming the DCNT field in the DSU 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. If the BT bit in the DSU control register is set, the DSU will force the processor into debug mode when the trace buffer is frozen. Note that due to pipeline delays, up to 4 additional instruction can be executed before the processor is placed in debug mode. 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 executed instructions, the EX bit should be set. To break on AHB load or store accesses, the LD and/or ST bits should be set.

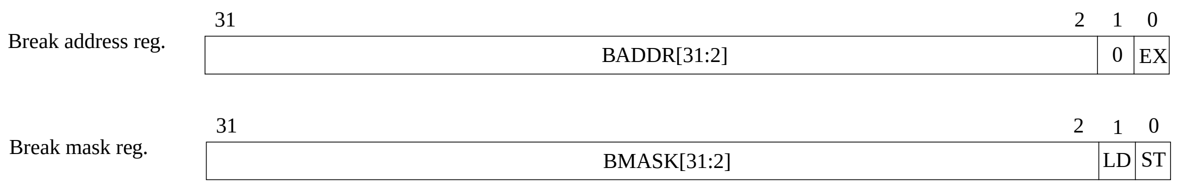


*Figure 53: DSU breakpoint registers*

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

### 11.2.6 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).

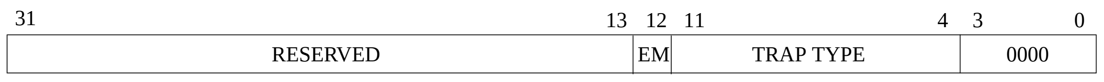


*Figure 54: DSU trap register*

- [11:4]: 8-bit SPARC trap type
- [12]: Error mode (EM). Set if the trap would have cause the processor to enter error mode.

## 11.3 DSU communication link

### 11.3.1 Operation

The DSU communication link consists of a UART connected to the AHB bus as a master (figure 55). A simple communication protocol is supported to transmit access parameters and data. A link command consist of a control byte, followed by a 32-bit address, followed by optional write data. If the LR bit in the DSU control register is set, a response byte will be sent after each AHB transfer. If the LR bit is not set, a 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 in figure 57. Through the communication link, a read or write transfer can be generated to any address on the AHB bus.

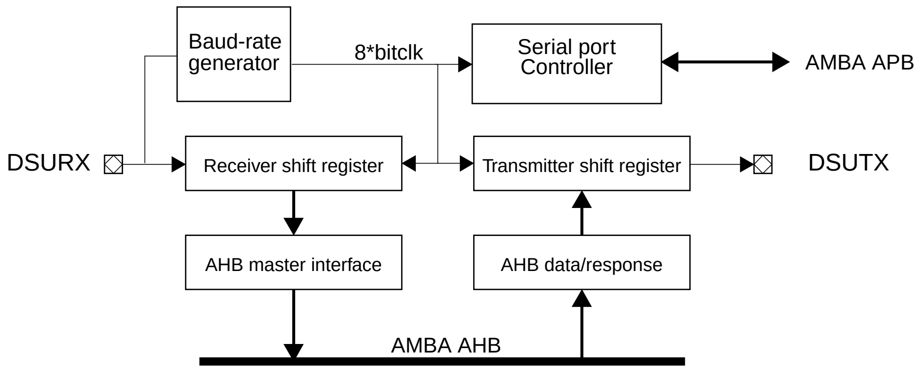


*Figure 55: DSU communication link block diagram*

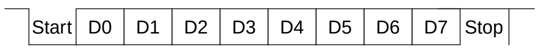


*Figure 56: DSU UART data frame*

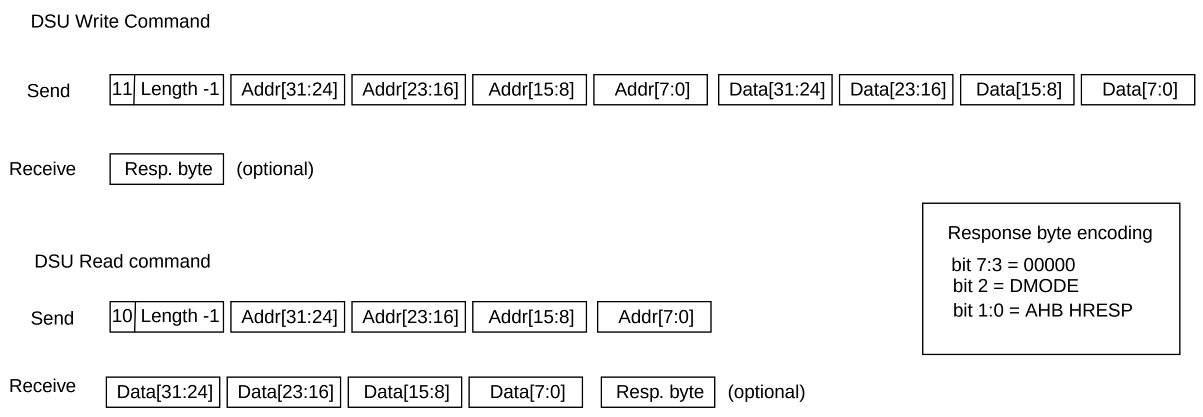

DSU Write Command
Send
11 | Length -1 | Addr[31:24] | Addr[23:16] | Addr[15:8] | Addr[7:0] | Data[31:24] | Data[23:16] | Data[15:8] | Data[7:0]
Receive
Resp. byte (optional)
DSU Read command
Send
10 | Length -1 | Addr[31:24] | Addr[23:16] | Addr[15:8] | Addr[7:0]
Receive
Data[31:24] | Data[23:16] | Data[15:8] | Data[7:0] | Resp. byte (optional)
Response byte encoding
bit 7:3 = 00000
bit 2 = DMODE
bit 1:0 = AHB HRESP

*Figure 57: DSU Communication link commands*

A response byte can optionally be sent when the processor goes from execution mode to debug mode. 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 nominal UARTs.

### 11.3.2 DSU UART control register

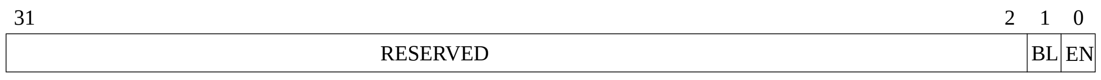


*Figure 58: UART control register*

- 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.

### 11.3.3 DSU UART status register

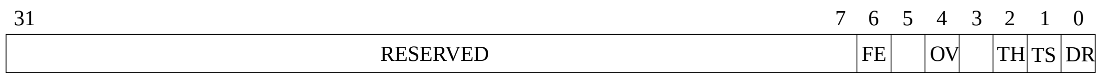


*Figure 59: UART status register*

- 0: Data ready (DR) - indicates that new data has been received and not yet read-out by the AHB master interface.
- 1: Transmitter shift register empty (TS) - indicates that the transmitter shift register is empty.

- 2: Transmitter hold register empty (TH) - indicates that the transmitter hold register is empty.
- 4: Overrun (OV) - indicates that one or more character have been lost due to overrun.
- 6: Framing error (FE) - indicates that a framing error was detected.

### 11.3.4 Baud rate generation

The 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 be 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

| 31 14 | 13 0 |
|---|---|
| RESERVED | SCALER RELOAD VALUE |

*Figure 60: DSU UART scaler reload register*

## 11.4 Common operations

### 11.4.1 Instruction breakpoints

Instruction breakpoints can be inserted by writing the breakpoint instruction (ta 1) to the desired memory address (software breakpoint) or using any of the four integer unit hardware breakpoints. Since cache snooping is only done on the data cache, the instruction cache must be flushed after the insertion or removal of software breakpoints. To minimize the influence on execution, it is enough to clear the corresponding instruction cache tag valid bit (which is accessible through the DSU).

The two DSU hardware breakpoints should only be used to freeze the trace buffer, and not for software debugging since there is a 4-instruction delay from the breakpoint hit before the processor enters the debug mode.

#### 11.4.2 Single stepping

By setting the SS bit and clearing the BN bit in the DSU control register, the processor will resume execution for one instruction and then automatically return to debug mode.

#### 11.4.3 Alternative debug sources

It is possible to debug the processor through any available AHB master since the DSU is a regular AHB slave. For instance, if a PCI interface is available, all debugging features will be available from any other PCI master.

#### 11.4.4 Booting from DSU

By asserting DSUEN and DSUBRE at reset time, the processor will directly enter debug mode without executing any instructions. The system can then be initialised from the communication link, and applications can be downloaded and debugged. Additionally, external (flash) proms for stand-alone booting can be re-programmed.

### 11.5 Design limitations

The registers of a co-processor or FPU in parallel configuration (separate register file) can not be read by the DSU.

### 11.6 DSU monitor

Gaisler Research provides a DSU monitor that allows both stand-alone debugging as well as an interface to gdb. See www.gaisler.com for details.

### 11.7 External DSU signals

The DSU uses five external signals: DSUACT, DSUBRE, DSUEN, DSURX and DSUTX. They are used as follows:

**DSUACT - DSU active (output)**

This active high output is asserted when the processor is in debug mode and controlled by the DSU.

**DSUBRE - DSU break enable**

A low-to-high transition on this active high input will generate break condition and put the processor in debug mode. After a low-to-high transition is detected, up to four instruction will be executed before debug node is entered.

**DSUEN - DSU enable (input)**

The active high input enables the DSU unit. If de-asserted, the DSU trace buffer will continue to operate but the processor will not enter debug mode.

**DSURX - DSU receiver (input)**

This active high input provides the data to the DSU communication link receiver.

**DSUTX - DSU transmitter (output)**

This active high input provides the output from the DSU communication link transmitter.

# 12 Signals

## 12.1 Memory bus signals

| Name | Type | Function | Active |
|---|---|---|---|
| A[27:0] | Output | Memory address | High |
| BEXCN | Input | Bus exception | Low |
| BRDYN | Input | Bus ready strobe | Low |
| D[31:0] | Bidir | Memory data | High |
| IOSN | Output | Local I/O select | Low |
| OEN | Output | Output enable | Low |
| RAMOEN[4:0] | Output | SRAM output enable | Low |
| RAMSN[4:0] | Output | SRAM chip-select | Low |
| READ | Output | Read strobe | High |
| ROMSN[1:0] | Output | PROM chip-select | Low |
| RWEN[3:0] | Output | SRAM write enable | Low |
| SA[14:0] | Output | SDRAM separate address bus | High |
| SDCASN | Output | SDRAM column address strobe | Low |
| SDCLK | Output | SDRAM clock | - |
| SDCKE[1:0] | Output | SDRAM clock enable | High |
| SDCSN[1:0] | Output | SDRAM chip select | Low |
| SD[31:0] | Bidir | SDRAM separate data bus | High |
| SDDQM[3:0] | Output | SDRAM data mask | Low |
| SDRASN | Output | SDRAM row address strobe | Low |
| SDWEN | Output | SDRAM write enable | Low |
| WRITEN | Output | Write strobe | Low |

*Table 20: Memory bus signals*

## 12.2 System interface signals

| Name | Type | Function | Active |
|---|---|---|---|
| CLK | Input | System clock | |
| PLLREF | Input | SDRAM clock PLL/DLL feedback | |
| ERRORN | Open-drain | System error | Low |
| PIO[15:0] | Bidir | Parallel I/O port | High |
| RESETN | Input | System reset | Low |
| WDOGN | Open-drain | Watchdog output | Low |
| DSUACT | Output | DSU active | High |
| DSUBRE | Input | DSU break | High |
| DSUEN | Input | DSU enable | High |
| DSURX | Input | DSU communication link transmission input | High |
| DSUTX | Output | DSU communication link transmission output | High |

*Table 21: System interface signals*

## 12.3 Signal description

All signals are clocked on the rising edge of CLK.

#### A[27:0] - Address bus (output)

These active high outputs carry the address during accesses on the memory bus. When no access is performed, the address of the last access is driven (also internal cycles).

#### BEXCN - Bus exception (input)

This active low input is sampled simultaneously with the data during accesses on the memory bus. If asserted, a memory error will be generated.

#### BRDYN - Bus ready (input)

This active low input indicates that the access to a memory mapped I/O area can be terminated on the next rising clock edge.

#### CLK - Processor clock (input)

This active high input provides the main processor clock.

#### D[31:0] - Data bus (bi-directional)

D[31:0] carries the data during transfers on the memory bus. The processor only drives the bus during write cycles. During accesses to 8-bit areas, only D[31:24] are used.

#### DSUACT - DSU active (output)

This active high output is asserted when the processor is in debug mode and controlled by the DSU.

#### DSUBRE - DSU break enable

A low-to-high transition on this active high input will generate break condition and put the processor in debug mode.

#### DSUEN - DSU enable (input)

The active high input enables the DSU unit. If de-asserted, the DSU trace buffer will continue to operate but the processor will not enter debug mode.

#### DSURX - DSU receiver (input)

This active high input provides the data to the DSU communication link receiver

**DSUTX - DSU transmitter (output)**

This active high output provides the data from the DSU communication link transmitter.

**ERRORN - Processor error (open-drain output)**

This active low output is asserted when the processor has entered error state and is halted. This happens when traps are disabled and an synchronous (un-maskable) trap occurs.

**IOSN - I/O select (output)**

This active low output is the chip-select signal for the memory mapped I/O area.

**OEN - Output enable (output)**

This active low output is asserted during read cycles on the memory bus.

**PIO[15:0] - Parallel I/O port (bi-directional)**

These bi-directional signals can be used as inputs or outputs to control external devices.

**PLLREF- PLL/DLL reference (input)**

Used as reference input for SDRAM clock generation in some implementations. Should be connected to SDCLK.

**RAMOEN[4:0] - RAM output enable (output)**

These active low signals provide an individual output enable for each RAM bank.

**RAMSN[4:0] - RAM chip-select (output)**

These active low outputs provide the chip-select signals for each RAM bank.

**READ - Read cycle (output)**

This active high output is asserted during read cycles on the memory bus.

**RESETN - Processor reset (input)**

When asserted, this active low input will reset the processor and all on-chip peripherals.

**ROMSN[1:0] - PROM chip-select (output)**

These active low outputs provide the chip-select signal for the PROM area. ROMSN[0] is asserted when the lower half of the PROM area is accessed (0 - 0x10000000), while ROMSN[1] is asserted for the upper half.

**RWEN [3:0] - RAM write enable (output)**

These active low outputs provide individual write strobes for each byte lane. RWEN[0] controls D[31:24], RWEN[1] controls D[23:16], etc.

**SA[14:0] - SDRAM separate address bus (output)**

When the model is configured for separate SDRAM address/data buses, the SDRAM address will appear on SA[12:0], while the SDRAM bank address will be on SA[14:13].

**SDCLK - SDRAM clock (output)**

SDRAM clock, can be configured to be identical or inverted in relation to the system clock.

**SDCASN - SDRAM column address strobe (output)**

This active low signal provides a common CAS for all SDRAM devices.

**SDCSN[1:0] - SDRAM chip select (output)**

These active low outputs provide the chip select signals for the two SDRAM banks.

**SD[31:0] - SDRAM separate data bus (bidir)**

Provides the SDRAM data bus when the model is configured for separate SDRAM buses.

**SDDQM[3:0] - SDRAM data mask (output)**

These active low outputs provide the DQM signals for both SDRAM banks.

**SDRASN - SDRAM row address strobe (output)**

This active low signal provides a common RAS for all SDRAM devices.

**SDWEN - SDRAM write strobe (output)**

This active low signal provides a common write strobe for all SDRAM devices.

**WDOGN - Watchdog time-out (open-drain output)**

This active low output is asserted when the watchdog times-out.

**WRITEN - Write enable (output)**

This active low output provides a write strobe during write cycles on the memory bus.

# 13 VHDL model architecture

## 13.1 Model hierarchy

The LEON VHDL model hierarchy can be seen in table 22 below.

| Entity/Package | File name | Function |
|---|---|---|
| LEON | leon.vhd | LEON top level entity |
| LEON_PCI | leon_pci.vhd | LEON/PCI top level entity |
| LEON/MCORE | mcore.vhd | Main core |
| LEON/MCORE/RSTGEN | rstgen.vhd | Reset generator |
| LEON/MCORE/AHBARB | ahbarb.vhd | AMBA/AHB controller |
| LEON/MCORE/APBMST | apbmst.vhd | AMBA/APB controller |
| LEON/MCORE/MCTRL | mctrl.vhd | Memory controller |
| LEON/MCORE/MCTRL/BPROM | bprom.vhd | Internal boot prom |
| LEON/MCORE/MCTRL/SDMCTRL | sdmctrl.vhd | SDRAM controller |
| LEON/MCORE/PROC | proc.vhd | Processor core |
| LEON/MCORE/PROC/CACHE | cache.vhd | Cache module |
| LEON/MCORE/PROC/CACHEMEM | cachemem.vhd | Cache ram |
| LEON/MCORE/PROC/CACHE/DCACHE | dcache.vhd | Data cache controller |
| LEON/MCORE/PROC/CACHE/ICACHE | icache.vhd | Instruction cache controller |
| LEON/MCORE/PROC/CACHE/ACACHE | acache.vhd | AHB/cache interface module |
| LEON/MCORE/PROC/IU | iu.vhd | Processor integer unit |
| LEON/MCORE/PROC/IU/MUL | mul.vhd | Multiplier state machined |
| LEON/MCORE/PROC/IU/DIV | div.vhd | radix-2 divider |
| LEON/MCORE/PROC/REGFILE | regfil.vhd | Integer unit register file |
| LEON/MCORE/PROC/FPU | meiko.vhd | Meiko FPU core (not included) |
| LEON/MCORE/PROC/FPU_LTH | fpu_lth.vhd | FPU core from Lund University |
| LEON/MCORE/PROC/FPU_CORE | fpu_core.vhd | FPU core wrapper |
| LEON/MCORE/PROC/FP1EU | fp1eu.vhd | parallel FPU interface |
| LEON/MCORE/IRQCTRL | irqctrl.vhd | Interrupt controller |
| LEON/MCORE/IOPORT | ioport.vhd | Parallel I/O port |
| LEON/MCORE/TIMERS | timers.vhd | Timers and watchdog |
| LEON/MCORE/UART | uart.vhd | UARTs |
| LEON/MCORE/LCONF | lconf.vhd | LEON configuration register |
| LEON/MCORE/AHBSTAT | ahbstat.vhd | AHB status register |
| LEON/MCORE/AHBMEM | ahbmem.vhd | AHB ram |
| LEON/MCORE/DSU | dsu.vhd | Debug support unit |
| LEON/MCORE/DSU_MEM | dsu_mem.vhd | DSU trace buffer memory |
| LEON/MCORE/DCOM | dcom.vhd | Debug comm. link controller |
| LEON/MCORE/DCOM/DCOM_UART | dcom_uart.vhd | UART for debug comm. link |
| LEON/MCORE/PCI/PCI_GR | pci_gr.vhd | Target-only PCI interface |
| LEON/MCORE/PCI | pci.vhd | Main PCI module |

*Table 22: LEON model hierarchy*

Table 23 shows the packages used in the LEON model.

| Package | File name | Function |
|---|---|---|
| TARGET | target.vhd | Pre-defined configurations for various targets |
| DEVICE | device.vhd | Current configuration |
| CONFIG | config.vhd | Generation of various constants for processor and caches |
| SPARCV8 | sparcv8.vhd | SPARCV8 opcode definitions |
| IFACE | iface.vhd | Type declarations for module interface signals |
| MACRO | macro.vhd | Various utility functions |
| AMBA | amba.vhd | Type definitions for the AMBA buses |
| AMBACOMP | ambacomp.vhd | AMBA component declarations |
| MULTLIB | multlib.vhd | Multiplier modules |
| FPULIB | fpu.vhd | FPU interface package |
| DEBUG | debug.vhd | Debug package with SPARC disassembler |
| TECH_GENERIC | tech_generic.vhd | Generic regfile and pad models |
| TECH_ATC18 | tech_atc18.vhd | Atmel ATC18 specific pads with Virage ram cells |
| TECH_ATC25 | tech_atc25.vhd | Atmel ATC25 specific regfile, ram and pad generators |
| TECH_ATC35 | tech_atc35.vhd | Atmel ATC35 specific regfile, ram and pad generators |
| TECH_FS90 | tech_fs90.vhd | UMC/FS90AB specific regfile, ram and pad generators |
| TECH_TSMC25 | tech_tsmc25.vhd | TSMC 0.25 um specific pads, with Artisan ram cells |
| TECH_UMC18 | tech_umc18.vhd | UMC 0.18 um specific regfile, ram and pad generators |
| TECH_VIRTEX | tech_virtex.vhd | Xilinx Virtex specific regfile and ram generators |
| TECH_VIRTEX2 | tech_virtex2.vhd | Xilinx Virtex2 specific regfile and ram generators |
| TECH_AXCEL | tech_axcel.vhd | Actel Axcellerator specific regfile and ram generators |
| TECH_PROASIC | tech_proasic.vhd | Actel Proasic specific regfile and ram generators |
| TECH_MAP | tech_map.vhd | Maps mega-cells according to selected target |

*Table 23: LEON packages*

## 13.2 Model coding style

The LEON VHDL model is designed to be used for both synthesis and board-level simulation. It is therefore written using rather high-level VHDL constructs, mostly using sequential statements. Typically, each module only contains two processes, one combinational process describing all functionality and one process implementing registers. Records are used extensively to group signals according their functionality. In particular, signals between modules are passed in records.

The model is fully synchronous using a continuous clock and the use of multiplexers to enable loading of pipe-line registers. The rising edge of the clock is used for all registers and most on-chip rams. However, some technology-specific rams use the negative edge to generate write strobes or as enable signal for address and data latches.

## 13.3 AMBA buses

### 13.3.1 AMBA AHB

The AHB bus can connect up to 16 masters and any number of slaves. The LEON processor core is normally connected as master 0, while the memory controller and APB bridge are connected at slaves 0 and 1.

The AHB controller (AHBARB) controls the AHB bus and implements both bus decoder/ multiplexer and the bus arbiter. The arbitration scheme is fixed priority where the bus master with highest index has highest priority. The processor is by default put on the lowest index. Re-arbitration is done after each transfer, but not during burst transfers (HTRANS = SEQ) or locked cycles (HLOCK asserted during arbitration).

Each AHB master is connected to the bus through two records, corresponding to the AHB signals as defined in the AMBA 2.0 standard:

```
-- AHB master inputs (HCLK and HRESETn routed separately)
   type AHB_Mst_In_Type is
      record
         HGRANT:     Std_ULogic;                        -- bus grant
         HREADY:     Std_ULogic;                        -- transfer done
         HRESP:      Std_Logic_Vector(1       downto 0); -- response type
         HRDATA:     Std_Logic_Vector(HDMAX-1 downto 0); -- read data bus
     end record;

   -- AHB master outputs
   type AHB_Mst_Out_Type is
      record
         HBUSREQ:    Std_ULogic;                        -- bus request
         HLOCK:      Std_ULogic;                        -- lock request
         HTRANS:     Std_Logic_Vector(1       downto 0); -- transfer type
         HADDR:      Std_Logic_Vector(HAMAX-1 downto 0); -- address bus (byte)
         HWRITE:     Std_ULogic;                        -- read/write
         HSIZE:      Std_Logic_Vector(2       downto 0); -- transfer size
         HBURST:     Std_Logic_Vector(2       downto 0); -- burst type
         HPROT:      Std_Logic_Vector(3       downto 0); -- protection control
         HWDATA:     Std_Logic_Vector(HDMAX-1 downto 0); -- write data bus
      end record;
```

Each AHB slave is similarly connected through two records:

```
-- AHB slave inputs (HCLK and HRESETn routed separately)
   type AHB_Slv_In_Type is
      record
         HSEL:       Std_ULogic;                        -- slave select
         HADDR:      Std_Logic_Vector(HAMAX-1 downto 0); -- address bus (byte)
         HWRITE:     Std_ULogic;                        -- read/write
         HTRANS:     Std_Logic_Vector(1       downto 0); -- transfer type
         HSIZE:      Std_Logic_Vector(2       downto 0); -- transfer size
         HBURST:     Std_Logic_Vector(2       downto 0); -- burst type
         HWDATA:     Std_Logic_Vector(HDMAX-1 downto 0); -- write data bus
         HPROT:      Std_Logic_Vector(3       downto 0); -- protection control
         HREADY:     Std_ULogic;                        -- transfer done
         HMASTER:    Std_Logic_Vector(3       downto 0); -- current master
         HMASTLOCK:  Std_ULogic;                        -- locked access
      end record;

   -- AHB slave outputs
type AHB_Slv_Out_Type is
      record
         HREADY:     Std_Logic;                        -- transfer done
         HRESP:      Std_Logic_Vector(1       downto 0); -- response type
         HRDATA:     Std_Logic_Vector(HDMAX-1 downto 0); -- read data bus
         HSPLIT:     Std_Logic_Vector(15      downto 0); -- split completion
      end record;
```

### 13.3.2 AHB cache aspects

The function is_cacheable() in macro.vhd defines which addresses will be cached by the processor. By default, only the PROM and RAM area of the memory controller are marked as cacheable.

### 13.3.3 AHB protection signals

The processor drives the AHB protection signals (HPROT) as follows: the opcode/data bit is driven according to if an instruction fetch or a data load/store is performed, the privileged bit is driven when the processor is in supervisor mode, the bufferable and cacheable bits are driven if a cacheable address is accessed.

### 13.3.4 APB bus

The APB bridge is connected to the AHB bus as a slave and acts as the (only) master on the APB bus. The slaves are connected through a pair of records containing the APB signals:

```
type APB_Slv_In_Type is
   record
      PSEL:        Std_ULogic;
      PENABLE:     Std_ULogic;
      PADDR:       Std_Logic_Vector(PAMAX-1 downto 0);
      PWRITE:      Std_ULogic;
      PWDATA:      Std_Logic_Vector(PDMAX-1 downto 0);
   end record;

type APB_Slv_Out_Type is
   record
      PRDATA:      Std_Logic_Vector(PDMAX-1 downto 0);
   end record;
```

The maximum number of APB slaves is defined by the APB_SLV_MAX constant in the TARGET package. The address decoding of each slave is done in apbmst.vhd.

## 13.4 Floating-point unit and co-processor

### 13.4.1 Generic CP interface

LEON can be configured to provide a generic interface to a special-purpose 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. The execution unit is connected to the interface using the following two records:

```
type cp_unit_in_type is record-- coprocessor execution unit input
  op1      : std_logic_vector (63 downto 0); -- operand 1
  op2      : std_logic_vector (63 downto 0); -- operand 2
  opcode   : std_logic_vector (9 downto 0);  -- opcode
  start    : std_logic;             -- start
  load     : std_logic;             -- load operands
  flush    : std_logic;             -- cancel operation
end record;

type cp_unit_out_type is record-- coprocessor execution unit output
  res      : std_logic_vector (63 downto 0); -- result
  cc       : std_logic_vector (1 downto 0);  -- condition codes
  exc      : std_logic_vector (5 downto 0);  -- exception
  busy     : std_logic;             -- eu busy
end record;
```

The waveform diagram for the execution unit interface can be seen in figure 61

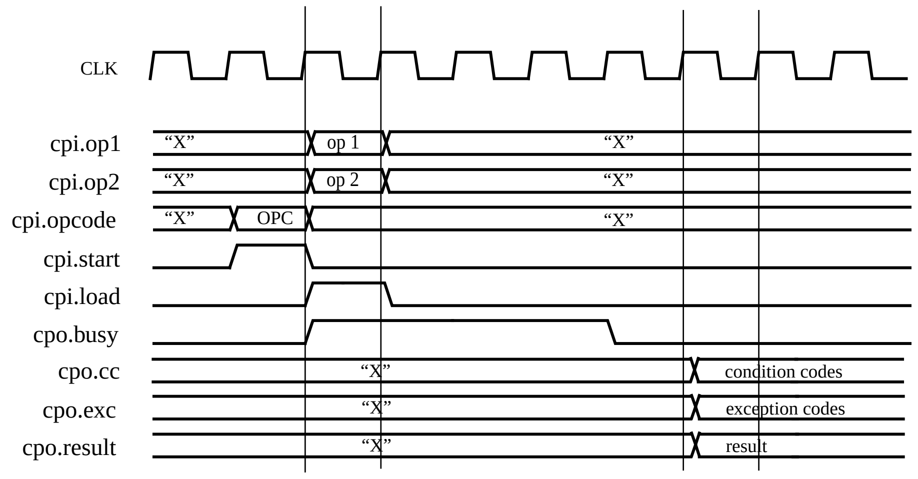


*Figure 61: Co-processor execution unit waveform diagram*

The execution unit is started by asserting the start signal together with a valid opcode. The operands are driven on the following cycle together with the load signal. If the instruction will take more than one cycle to complete, the execution unit must drive busy from the cycle after the start signal was asserted, until the cycle before the result is valid. The result, condition codes and exception information are valid from the cycle after the de-assertion of busy, and until the next assertion of start. The opcode (cpi.opcode[9:0]) is the concatenation of bits [19,13:5] of the instruction. If execution of a co-processor instruction need to be prematurely aborted (due to an IU trap), cpi.flush will be asserted for two clock cycles. The execution unit should then be reset to its idle condition.

### 13.4.2 FPU interface

The LEON model two interface options for a floating-point unit: either a parallel interface or an integrated interface where FP instruction do not execute in parallel with IU instruction. Both interface methods expect an FPU core to have the same interface as described in figure 61 above, and which also is the interface used by the Meiko FPU core.

The direct FPU interface does not implement a floating-point queue, the processor is stopped during the execution of floating-point instructions. This means that QNE bit in the %fsr register always is zero, and any attempts of executing the STDFQ instruction will generate a FPU exception trap. The parallel interface lets FPU instructions execute in parallel with IU instructions and only halts the processor in case of data- or resource dependencies. Refer to the SPARC V8 manual for a more in-depth discussion of the FPU and co-processor characteristics.

As of leon2-1.0.1, a partial implementation of an IEEE-754 compatible FPU is included in the model (fpu_lth.vhd). This FPU is contributed by Martin Kasprzyk, a student at Lund Technical University, and does currently implement single- and double-precision addition, subtraction and compare. All rounding modes are implemented as well as a Meiko compatible interface. To make this FPU useful for LEON, multiplication, divide and square-root must however also be implemented. A document describing this FPU is provided in `doc`.

# 14 Model Configuration

The model is configurable to allow different cache sizes, multiplier performance, clock generation, and target technologies. The model is configured through a number of constant records which each configure a specific module/function. The declaration of the configuration record types is in the TARGET package, while the specification of the configuration records is defined in the DEVICE package.

## 14.1 Synthesis configuration

The synthesis configuration sub-record is used to adapt the model to various synthesis tools and target libraries:

```
type targettechs is
  (gen, virtex, virtex2, atc35, atc25, atc18, fs90, umc18, tsmc25, proasic, axcel);

-- synthesis configuration
type syn_config_type is record
  targettech: targettechs;
  infer_ram : boolean;-- infer cache and dsu ram automatically
  infer_regf : boolean;-- infer regfile automatically
  infer_rom: boolean;-- infer boot prom automatically
  infer_pads: boolean;-- infer pads automatically
  infer_mult: boolean;-- infer multiplier automatically
  rftype    : integer;-- regfile implementation option
  targetclk : targettechs;  -- use technology specific clock generation
  clk_mul   : integer;-- PLL clock multiply factor
  clk_div   : integer;-- PLL clock divide factor
  pci_dll    : boolean;-- Re-synchronize PCI clock using a DLL
end record;type targettechs is (gen, virtex, atc35, atc25);
```

Depending on synthesis tool and target technology, the technology dependant mega-cells (ram, rom, pads) can either be automatically inferred or directly instantiated. Using direct instantiation, 8 types of target technologies are currently supported: Xilinx Virtex (FPGA), Atmel ATC35 and ATC25 (0.35 & 0.25 um CMOS), TSMC 0.25 um CMOS, UMC 0.25 & 0.18 um CMOS, Actel ProAsic (FPGA), and Actel Axcellerator (anti-fuse FPGA). In addition, any technology that is supported by synthesis tools capable of automatic inference of mega-cells (Synplify and Leonardo) is also supported. When using tools with inference capability targeting Xilinx Virtex, a choice can be made to either infer the mega-cells automatically or to use direct instantiation. The choice is done by setting the parameters **infer_ram**, **infer_regf** and **infer_rom** accordingly.

The **rftype** option has impact on target technologies which are capable of providing more than one type of register file. **Infer_mult** selects how the multiplier is generated, for details see section 14.2 below. On Virtex targets, **clk_mul** and **clk_div** are used to configure the frequency synthesizer (DCM or CLKDLL).

## 14.2 Integer unit configuration

The integer unit configuration record defines the implementation of the integer unit:

```
-- processor configuration
type multypes is (none, iterative, m32x8, m16x16, m32x16, m32x32);
type divtypes is (none, radix2);
type iu_config_type is record
  nwindows: integer;-- # register windows (2 - 32)
  multiplier: multypes;-- multiplier type
```

```
 mulpipe: boolean; -- multiplier pipeline registers
 divider   : divtypes;-- divider type
  mac : boolean;   -- multiply/accumulate
  fpuen: integer range 0 to 1;-- FPU enable
  cpen: boolean;   -- co-processor enable
  fastjump   : boolean;-- enable fast jump address generation
  icchold   : boolean;-- enable fast branch logic
  lddelay: integer range 1 to 2; -- # load delay cycles (1-2)
  fastdecode : boolean;-- optimise instruction decoding (FPGA only)
  rflowpow   : boolean;-- disable regfile when not accessed
  watchpoints: integer range 0 to 4; -- # hardware watchpoints (0-4)
end record;
```

**`nwindows`** set the number of register windows; the SPARC standard allows 2 - 32 windows, but to be compatible with the window overflow/underflow handlers in the LECCS compiler, 8 windows should be used.

The **multiplier** option selects how the multiply instructions are implemented The table below show the possible configurations:

| Configuration | latency (clocks) | approx. area (Kgates) |
|---|---|---|
| iterative | 35 | 1000 |
| m16x16 + pipeline reg | 5 | 6,500 |
| m16x16 | 4 | 6,000 |
| m32x8 | 4 | 5,000 |
| m32x16 | 2 | 9,000 |
| mx32x32 | 1 | 15,000 |

*Table 24: Multiplier configurations*

If **infer_mult** in the synthesis configuration record (see above) is false, the multipliers are implemented using the module generators in multlib.vhd. If **infer_mult** is true, the synthesis tool will infer a multiplier. For FPGA implementations, best performance is achieved when **infer_mult** is true and m16x16 is selected. ASIC implementations (using synopsys DC) should set **infer_mult** to false since the provided multiplier macros in MULTLIB are faster than the synopsys generated equivalents. The **mac** option enables the SMAC/UMAC instructions. Requires the **multiplier** to use the m16x16 configuration. The **mulpipe** option can be used to infer pipeline registers in the m16x16 multiplier when **infer_mult** is false. This will improve the timing of the multiplier but increase the latency from 4 to 5 clocks.

The **divider** field select how the UDIV/SDIV instructions are implemented. Currently, only a radix-2 divider is available.

If an FPU will be attached, `fpuen` should be set to 1. If a co-processor will be attached, `cpen` should be set to true.

To speed up branch address generation, **fastjump** can be set to implement a separate branch address adder. The pipeline can be configured to have either one or two load delay cycles using the **lddelay** option. One cycle gives higher performance (lower CPI) but may result in slower timing in ASIC implementations. Setting **`icchold`** will improve timing by adding a pipeline hold cycle if a branch instruction is preceded by an icc-modifying instruction.

Similarly, **`fastdecode`** will improve timing by adding parallel logic for register file address generation. The rflowpow option will enable read-enable signals to the register file write ports, thereby saving power when the register file is not accessed. However, this option might introduce a critical path to the read-enable ports on some register files.

Setting **watchpoint** to a value between 1 - 4 will enable corresponding number of watch-points. Setting it to 0, will disable all watch-point logic.

## 14.3 FPU configuration

The FPU configuration record is used to select FPU interface and core type:

```
type fpucoretype  is (meiko, lth);  -- FPU core type
type fpuiftype is (none, serial, parallel);         -- FPU interface type
type fpu_config_type is record
  core: fpucoretype;-- FPU core type
  interface: fpuiftype;-- FPU inteface type
  fregs: integer;  -- 32 for serial interface, 0 for parallel
  version: integer range 0 to 7; -- FPU version ID
end record;
```

The **core** element can either be **meiko** or **lth**, selecting which of the two cores will be used. The **interface** element defines whether to use a **serial**, **parallel** or **none** (no FPU) interface. The **version** element defines the (constant) FPU version ID in the %fsr register.

## 14.4 Cache configuration

The cache is configured through the cache configuration record:

```
type dsnoop_type is (none, slow, fast); -- snoop implementation type
constant PROC_CACHE_MAX: integer := 4;   -- maximum cacheability ranges
constant PROC_CACHE_ADDR_MSB : integer := 3;
subtype proc_cache_addr_type is std_logic_vector(PROC_CACHE_ADDR_MSB-1 downto 0);

type cache_replace_type is (lru, lrr, rnd);  -- cache replacement algorithm
constant MAXSETS  : integer := 4;

type cache_config_type is record
  isets         : integer range 1 to MAXSETS;   -- # of sets in icache
  isetsize: integer;-- I-cache size per set in Kbytes
  ilinesize: integer;-- # words per I-cache line
  ireplace      : cache_replace_type;         -- icache replacement algorithm
  ilock     : integer;-- icache locking
  ilram      : boolean;    -- local inst ram enable
  ilramsize : integer;-- local inst ram size in kbytes
  ilramaddr : integer;-- local inst ram start address (8 msb)
  dsets         : integer range 1 to MAXSETS;   -- # of sets in dcache
  dsetsize: integer;-- D-cache size per set in Kbytes
  dlinesize: integer;-- # words per D-cache line
  dreplace      : cache_replace_type;         -- icache replacement algorithm
  dlock     : integer;-- dcache locking
  dsnoop    : dsnoop_type;-- data-cache snooping
  drfast    : boolean;-- data-cache fast read-data generation
  dwfast    : boolean;-- data-cache fast write-data generation
  dlram     : boolean;-- local data ram enable
  dlramsize : integer;-- local data ram size in kbytes
  dlramaddr : integer;-- local data ram start address (8 msb)
end record;
```

Valid settings for the cache set size are 1 - 64 (Kbyte), and must be a power of 2. The line size may be 4 - 8 (words/line). Valid settings for the number of sets are 1 - 4 (2 if LRR algorithm is selected). Replacement algorithm may be random, LRR or LRU. The instruction and data caches may be configured independently. The **dlock** and **ilock** fields enable cache locking for the data and instruction cache respectively. The **drfast** field enables parallel logic to improve data cache read timing, while the **dwfast** field improves data cache write timing. Local instruction and/or data ram are enabled by setting **ilram** and/or **dlram** to true. The size of the rams in Kbyte is determined by values of **ilramsize** and **dlramsize**. The 8 MSB bits of the ram start addresses are set in **ilramaddr** and **dlramaddr**. To enable data cache snooping, **dsnoop** should be set to true.

The cacheable areas is defined by the function is_cacheable() in macro.vhd.

## 14.5 Memory controller configuration

The memory controller is configured through the memory controller configuration record:

```
type mctrl_config_type is record
  bus8en      : boolean; -- enable 8-bit bus operation
  bus16en     : boolean;-- enable 16-bit bus operation
  wendfb      : boolean; -- enable wen feed-back to data bus drivers
  ramsel5     : boolean; -- enable 5th ram select
  sdramen     : boolean;-- enable sdram controller
  sdinvclk    : boolean;-- invert sdram clock
   sdsepbus  : boolean;-- enable separate SDRAM buses
end record;
```

The 8- and 16-bit memory interface features are optional; if set to false the associated function will be disabled, resulting in a smaller design. The **ramsel5** fields enables the fifth (RAMSN[4]) chip select signal in the memory controller. The **sdramen** field enables the SDRAM controller, while **sdinvclk** controls the polarity of the SDRAM clock. If **sdinvclk** is true, the SDRAM clock output (SDCLK) will be inverted with respect to the system clock. Setting **sdsepbus** to *true* will configure the SDRAM controller to use separate address and data buses (SA[14:0], SD[31:0]). When set to false, the SDRAM should be connected to the common address and data buses (A[16:2], D[31:0]).

## 14.6 Debug configuration

Various debug features are controlled through the debug configuration record:

```
type debug_config_type is record
  enable     : boolean;-- enable debug port
  uart     : boolean;-- enable fast uart data to console
  iureg    : boolean;-- enable tracing of iu register writes
  fpureg      : boolean;-- enable tracing of fpu register writes
  nohalt      : boolean;-- dont halt on error
  pclow       : integer;-- set to 2 for synthesis, 0 for debug
  dsuenable    : boolean;-- enable DSU
  tracesize: integer;-- # trace buffer size in kbyte
end record;
```

The **`enable`** field has to be true to enable the built-in disassembler (it does not affect synthesis) and to allow DSU operations. Setting **`uart`** to true will tie the UART transmitter ready bit permanently high for simulation (does not affect synthesis) and output any sent characters on the simulator console (line buffered). The UART output (TX) will not simulate properly in this mode. Setting **`iureg`** will trace all IU register writes to the console. Setting **`fpureg`** will trace all FPU register writes to the console.

Setting **nohalt** will cause the processor to take a reset trap and continue execution when error mode (trap in a trap) is encountered. Do NOT set this bit for synthesis since it will violate the SPARC standard and will make it impossible to halt the processor.

Since SPARC instructions are always word-aligned, all internal program counter registers only have 30 bits (A[31:2]), making them difficult to trace in waveforms. If **pclow** is set to 0, the program counters will be made 32-bit to aid debugging. Only use **pclow**=2 for synthesis.

The **dsuenable** field enables the debug support unit. **Dsutrace** enables the trace buffer. The **tracelines** field indicates how many lines the trace buffer should contain. Note that for each line in the trace buffer, 16 bytes will be used by the trace buffer memory. The **dsumixed** field enables the mixed tracing mode (simultaneous instruction and AHB tracing). The **dsudpram** enables the DSU trace buffer to be implemented with dual-port rams, otherwise ordinary single-port rams are used. Ram blocks with 32-bit width will be used for the trace buffer memory; the table below shows the type and number of blocks used as a function of the configuration options.

| dsumixed | dsudpram | single-port | dual-port |
|---|---|---|---|
| false | false | 4 | 0 |
| false | true | 0 | 2 |
| true | false | 8 | 0 |
| true | false | 0 | 4 |

*Table 25: DSU trace buffer ram usage*

## 14.7 Peripheral configuration

Enabling of peripheral function is controlled through the peripheral configuration record:

```
type irq_filter_type is (lvl0, lvl1, edge0, edge1);
type irq_filter_vec is array (0 to 31) of irq_filter_type;

type irq2type is record
  enable   : boolean;-- secondary interrupt controller
  channels : integer;-- number of additional interrupts (1 - 32)
  filter: irq_filter_vec; -- irq filter definitions
end record;

type peri_config_type is record
  cfgreg   : boolean;-- enable LEON configuration register
  ahbstat  : boolean;-- enable AHB status register
  wprot  : boolean;-- enable RAM write-protection unit
  wdog   : boolean;-- enable watchdog
   ahbram : boolean;-- enable AHB RAM
  ahbrambits : integer;-- address bits in AHB ram
end record;
```

If not enabled, the corresponding function will be suppressed resulting in a smaller design.

The secondary interrupt controller is enabled by selecting a configuration record with irq2cfg.enable = true. An example record defining four extra interrupts could look like this:

```
constant irq2chan4 : irq2type := ( enable => true, channels => 4,
  filter => (lvl0, lvl1, edge0, edge1, others => lvl0));
```

Lvl0 mean that the interrupt will be treated as active low, lvl1 as active high, edge0 as negative edge-triggered and edge1 as positive edge-triggered. Since the registers in the secondary interrupt controller are accessed through the APB bus, an APB configuration with the interrupt controller present must be selected.

The on-chip AHB ram is enabled by setting **ahbram** to true. The **ahbrambits** denote the number of address bits used by the ram. Since a 32-bit ram is used, 8 address bits will results in a 1-kbyte ram block.

## 14.8 Boot configuration

Apart from that standard boot procedure of booting from address 0 in the external memory, LEON can be configured to boot from an internal prom or from the debug support unit. The boot options are defined on the boot configuration record as defined in the TARGET package:

```
type boottype is (memory, prom, dual);
type boot_config_type is record
  boot : boottype; -- select boot source
  ramrws   : integer range 0 to 3;-- ram read waitstates
  ramwws   : integer range 0 to 3;-- ram write waitstates
  sysclk   : integer;-- cpu clock
  baud     : positive;-- UART baud rate
  extbaud  : boolean;-- use external baud rate setting
  pabits   : positive;-- internal boot-prom address bits
end record;
```

### 14.8.1 Booting from internal prom

If the boot option is set to ‘prom’, an internal prom will be inferred. When booting from internal prom, the UART baud generator, timer 1 scaler, and memory configuration register 2 are preset to the values calculated from the boot configuration record. The UART scaler is preset to generate the desired baud rate, taking the system clock frequency into account. The timer 1 scaler is preset to generate a 1 MHz tick to the timers. The ram read and write waitstate are set directly from to the boot configuration record. If the extbaud variable is set in the boot configuration record, the UART scalers will instead be initialised with the value on I/O port [7:0] (the top 4 bits of the scalers will be cleared). Using external straps or assigning the port as pull-up/pull-down, the desired baud rate can be set regardless of clock frequency and without having to regenerate the prom or re-synthesise. If a different boot program *is* desired, use the utility in the pmon directory to generate a new prom file. When the **dual** boottype is configured, the boot source is defined by PIO[4]. If PIO[4] is asserted (=1), the internal prom will be enabled, otherwise the external prom will be used.

Which content is placed in the boot-prom is decided by the infer_prom and the pabits settings in the configuration record. If infer_prom is true, the contents is generated from bprom.vhd, which by default contains PMON (see below). If infer_prom is false, only Xilinx Virtex devices can be targeted and the prom is directly instantiated. Depending on the value of pabits, either a prom with 1, 2, 4 or 8 kbyte is instantiated. The xilinx sub-directory contains two templates, virtex_prom256 (1 kbyte) and virtex_prom2048 (8 kbyte). The virtex_prom256 contains PMON, while virtex_prom2048 contains a prom version of rdbmon from LECCS-1.1.1. The pre-defined configuration virtex_2k1k_rdbmon in device.vhd will instantiate the virtex_prom2048 prom.

### 14.8.2 PMON S-record loader

Pmon is a simple monitor that can be placed in an on-chip boot prom, external prom or cache memories (using the boot-cache configuration). Two versions are provided, one to be used for on-chip prom or caches (bprom.c) and one for external proms (eprom.c).

On reset, the monitor scans all ram-banks and configures the memory control register 2 accordingly. The monitor can detect if 8-, 16- or 32-bit memory is attached, if read-modify-write sub-word write cycles are needed and the size of each ram bank. It will also set the stack pointer to the top of ram. The monitor writes a boot message on UART1 transmitter describing the detected memory configuration and then waits for S-records to be downloaded on UART1 receiver. It recognises two types of S-records: memory contents and start address. A memory content S-record is saved to the specified address in memory, while a start address record will cause the monitor to jump to the indicated address. Applications compiled with LECCS can be converted to a suitable S-record stream with:

sparc-rtems-objcopy -O srec app app.srec

See the README files in the pmon directory for more details. After successful boot, the monitor will write a message similar to:

```
LEON-1: 2*2048K 32-bit memory
>
```

### 14.8.3 Rdbmon

A promable version of rdbmon is provided in pmon/lmon.o. It can be put in the boot-prom if infer_prom is false and pabits = 11. Note that rdbmon needs to be re-compiled for each specific target hardware, it does not automatically detect the memory configuration. To do this, change the makefile in the pmon directory so that the mkprom settings will reflect your hardware. Then, do a 'make' which will produce a virtex_prom2048.mif file. Use the Xilinx Coregen to produce a synchronous ram from the .mif file, and put the resulting edif file (virtex_prom2048.edn) in the syn directory so that the Xilinx place&route tools will find it during design expansion. The file virtex_prom2048.xco contains a suitable project file for coregen. LECCS-1.1.1 or higher is needed to build rdbmon for the boot-prom. Rdbmon consumes 8 kbyte (16 Virtex blockrams), so at least an XCV800 device is needed to fit both the boot prom and ram for the caches and register file.

### 14.8.4 Booting from the debug support unit

Booting from the debug support unit can be done regardless of which boot configuration has been made, by asserting both DSUEN and DSUBRE at reset time. See "Hardware debug support" on page 61 for details.

## 14.9 AMBA configuration

The AMBA buses are the main way of adding new functional units. The LEON model provides a flexible configuration method to add and map new AHB/APB compliant modules. The full AMBA configuration is defined through two configuration sub-records, one for the AHB bus and one for APB:

```
type ahb_config_type is record
```

```
  masters: integer range 1 to AHB_MST_MAX;
  defmst : integer range 0 to AHB_MST_MAX-1;
  split  : boolean;-- add support for SPLIT reponse
  slvtable : ahb_slv_config_vector(0 to AHB_SLV_MAX-1);
  cachetable : ahb_cache_config_vector(0 to AHB_CACHE_MAX-1);
end record;


type apb_config_type is record
  table    : apb_slv_config_vector(0 to APB_SLV_MAX-1);
end record;
```

### 14.9.1 AHB configuration

The number of attached masters is defined by the **`masters`** field in the AHB configuration record. The masters are connected to the `ahbmi/ahbmo` buses in the MCORE module. AHB master should be connected to index 0 - (`masters`-1) of the `ahbmi/ahbmo` buses. The defmst field indicates which master is granted by default if no other master is requesting the bus.

The number of AHB slaves and their address range is defined through the AHB slave table. The table is located in device.vhd, and defines which AHB slave will be addressed for each 256 Mbyte block of the address space. Slave number 7 means that the block is unused.

```
-- standard slave config
constant ahbrange_config  : ahbslv_addr_type :=
        (0,0,0,0,0,0,0,0,1,7,7,7,7,7,7,7);

constant ahb_config : ahb_config_type := ( masters => 1, defmst => 0,
    split => false, testmod => false);
```

To add or remove an AHB slave, edit the configuration table and the AHB bus decoder/ multiplexer and will automatically be reconfigured. The AHB slaves should be connected to the `ahbsi/ahbso` buses. The `index` field in the table indicates which bus index the slave should connect to. If **split** is set to true, the AHB arbiter will include split support and each slave must then driver the SPLIT signal.

### 14.9.2 APB configuration

The decoding of APB slaves is defined in the APB bridge (apbmst.vhd). To add APB slaves, edit the corresponding case statement in apbmst.vhd and add your modules in MCORE. The APB slaves should be connected to the `apbi/apbo` buses.

## 14.10 PCI configuration

Configuration of the PCI interface is done through the PCI configuration record:

```
constant pci_config : pci_config_type := (
    pcicore => simple_target, ahbmasters => 1, ahbslaves => 0,
    arbiter => false, fixpri => false, prilevels => 4, pcimasters => 4,
    vendorid => 16#16E3#, deviceid => 16#0210#, subsysid => 16#0000#,
    revisionid => 16#01#, classcode =>16#000000#, pmepads => false,
    p66pad => false, pcirstall => false);
```

The **pcicore** field indicates which PCI core to use. Currently, only the target-only PCI core is provided with model. Set **pcicore** to *none* to disable the PCI interface. The **ahbmasters** and **ahbslaves** fields indicate how many AHB master and slave interfaces the selected core types has. To enable the PCI arbiter, set **arbiter** to true. This should only be done in case the processor acts as a system controller. The PCI vendorid, deviceid and subsystemid can be configured through the corresponding fields.

# 15 Porting to a new technology or synthesis tool

## 15.1 General

LEON uses three types of technology dependant cells; rams for the cache memories, 3-port register file for the IU/FPU registers, and pads. These cells can either be inferred by the synthesis tool or directly instantiated from a target specific library. For each technology or instantiation method, a specific package is provided. The selection of instantiation method and target library is done through the configuration record in the TARGET package. The following technology dependant packages are provided:

| **package** | **technology** | **RAM** | **PADS** |
|---|---|---|---|
| TECH_GENERIC | Behavioural models | inferred | inferred |
| TECH_VIRTEX/VIRTEX2 | Xilinx VIRTEX/2 FPGA | instantiated | inferred |
| TECH_ATC18/25/35 | Atmel ATC18/25/35 | instantiated | instantiated |
| TECH_FS90 | UMC FS90A/B | instantiated | instantiated |
| TECH_UMC18 | UMC 0.18 um CMOS | instantiated | instantiated |
| TECH_TSMC25 | TSMC 0.25 um w. Artisan rams | instantiated | instantiated |
| TECH_PROASIC | Actel Proasic FPGA | instantiated | inferred |
| TECH_AXCEL | Actel AX anti-fuse FPGA | instantiated | inferred |
| TECH_MAP | Selects mega-cells depending on configuration | - | |

*Table 26: Technology mapping packages*

The technology dependant packages can be seen a wrappers around the mega cells provided by the target technology or synthesis tool. The wrappers are then called from TECH_MAP, where the selection is done depending on the configured synthesis method and target technology. To port to a new tool or target library, a technology dependant package should be added, exporting the proper cell generators. In the TARGET package, the *targettechs* type should be updated to include the new technology or synthesis tool, while the TECH_MAP package should be edited to call the exported cell generators for the appropriate configuration.

## 15.2 Target specific mega-cells

### 15.2.1 Integer unit register-file

The IU register-file must have one 32-bits write port and two 32-bits read ports. The number of registers depend on the configured number of register windows. The standard configuration of 8 windows requires 136 registers, numbered 0 - 135. Note that register 128 is not used and will never be written (corresponds to SPARC register %g0).

If the Meiko FPU is enabled using the direct interface, the register file should have 32 extra registers to store the FPU registers (i.e 168 registers for 8 register windows + FPU). For all target technologies (FPGA and ASIC), the register file is currently implemented as two parallel dual-port rams, each one with one read port and one write port.

The register file must provide the read-data at the end of the same cycle as the read address is provided (figure 62). This can be implemented with asynchronous read ports, or by clocking a synchronous read port on the negative clock (CLKN). Read/write collisions in the same cycle (RA1/WA1) does not have to be handled since this will be detected in the IU pipeline and the write data will be bypassed automatically. However, collision between two consecutive cycles (WA1/RA2) is not handled and the register file must provide a bypass in case write-through is not supported.

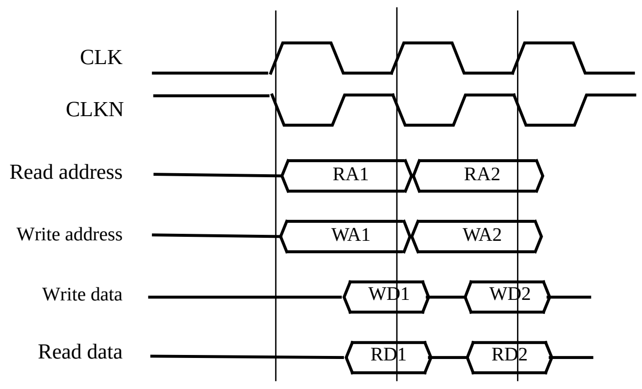


*Figure 62: IU register file read/write timing*

The TECH_ATC35 package provides an example of a synchronous register file clocked on the inverted clock, while TECH_ATC25 shows an example of a fully asynchronous register file. TECH_GENERIC contains an example of WA1/RA2 contention and associated bypass logic.

### 15.2.2 Parallel FPU & co-processor register file

The parallel FPU and co-processor uses a separate register file with 32 32-bit words. The FPU/CP controller (fp1eu.vhd) instantiates two 16x32 register files to make up one 32x32 register file with two 64-bit read ports and one 64-bit write port with individual(32-bits) write enables. To use fp1eu.vhd, the technology file must contain a register file with two 32-bit read ports and one 32-bit write port. All ports should operate synchronously on the rising edge. Read/write contention in the same cycle does not have to be handled, the FPU/CP controller contains contention and bypass logic. See TECH_GENERIC and TECH_ATC25 for examples.

### 15.2.3 Cache ram memory cells

Synchronous single-port ram cells are used for both tag and data in the cache. The width and depth depends on the configuration as defined in the configuration record. The table below shows the ram size for certain cache configurations:

| Cache set size | Words/line | tag ram | data ram |
|---|---|---|---|
| 1 kbyte | 8 | 32x30 | 256x32 |
| 1 kbyte | 4 | 64x26 | 256x32 |
| 2 kbyte | 8 | 64x29 | 512x32 |
| 2 kbyte | 4 | 128x25 | 512x32 |
| 4 kbyte | 8 | 128x28 | 1024x32 |
| 4 kbyte | 4 | 256x24 | 1024x32 |
| 8 kbyte | 8 | 256x27 | 2048x32 |
| 8 kbyte | 4 | 512x23 | 2048x32 |
| 16 kbyte | 8 | 512x26 | 4096x32 |
| 16 kbyte | 4 | 1024x22 | 4096x32 |

*Table 27: Cache ram cell sizes*

The cache controllers are designed such that the used ram cells do NOT have to support write-through (simultaneous read of written data).

### 15.2.4 Dual-port rams

If data cache snooping is enabled, or the DSU trace buffer is set to use dual-port rams, the target technology must contains synchronous dual-port rams. The dual-port rams will be used to implement the data cache tag memory or the trace buffer memory. Currently, only the TECH_VIRTEX, TECH_ATC25 and TECH_TSMC25 packages include mappings to dual-port rams.

### 15.2.5 Pads

Technology specific pads are usually automatically inferred by the synthesis tool targeting FPGA technologies. For ASIC technologies, generate statements are used to instantiate technology dependant pads. The selection of pads is done in TECH_MAP. Output pads has a generic parameter to select driving strength, see TECH_ATC25 for examples.

### 15.2.6 Adding a new technology or synthesis tool

Adding support for a new target library or synthesis tool is done as follows:

1. Create a package similar to tech_*.vhd, containing the specific rams, regfile, and pads.
2. Edit target.vhd to include your technology or synthesis tool in targettechs.
3. Edit tech_map.vhd to instantiate the cells when the technology is selected.
4. Define and select a configuration using the new technology (target.vhd/device.vhd).
5. Submit your changes to jiri@gaisler.com for inclusion in future version of LEON!