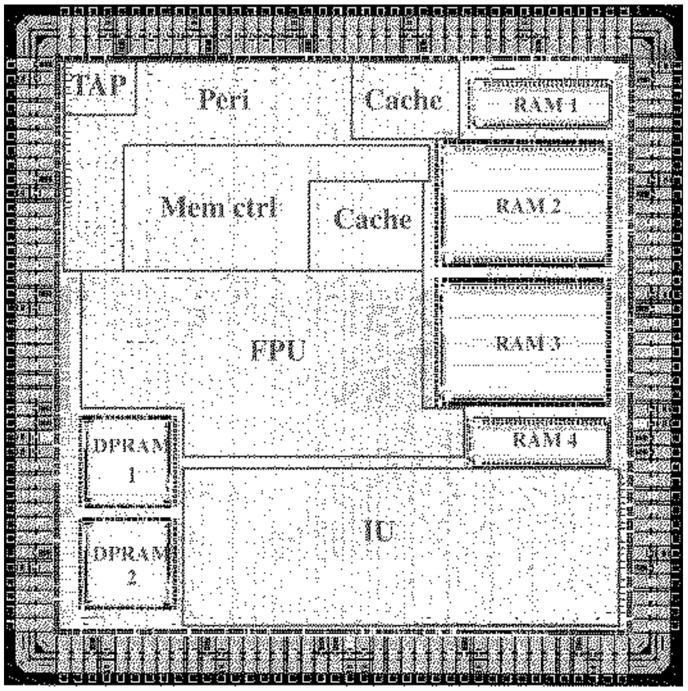
The LEON-2 Processor User's Manual

Version 1.0.2
Issue 1
February 2002



Jiri Gaisler
Gaisler Research

Gaisler Research

jiri@gaisler.com

The LEON-2 Processor User’s Manual

1 Introduction

1.1 Overview

The LEON VHDL model implements a 32-bit processor conforming to the 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 and a flexible memory controller. 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 functionallity of the LEON model. Through the model's configuration record (see "Model Configuration" on page 63), parts of the described functionality can be suppressed or modified to generate a smaller or faster implementation.

1.2 News in LEON-2 version 1.0.2

The following modifications (compared to leon2-1.0.1) has been done:

- Corrected ASI generation during co-processor load/store
- Timing optimisation of data cache hit and HLOCK generation
- Correction of AHB HSIZE during cacheable byte/halfword loads
- Modification of configuration record to work-around Leonardo-2001.x bugs.
- Improve accuracy of DSU timer during stepping
- DSU did not break on load exception (trap 0x09) (fixed)
- Change back to cacheability table (acache.vhd)
- Remove all use of non-standard HCACHE signal (amba.vhd, acache.vhd)
- Add generation of AHB HPROT signals for processor (acache.vhd)
- Modified access with ASI 4 and 7 to not allocate a cache line on cache miss

1.3 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.

1.4 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.

1.5 Functional overview

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

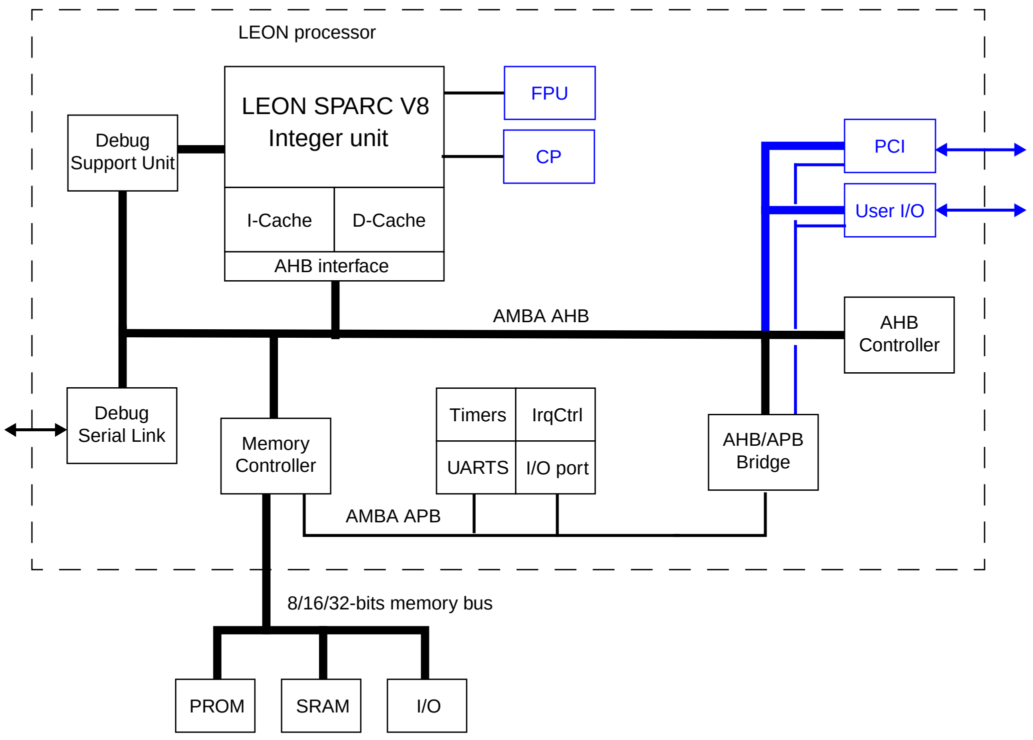


Figure 1: LEON-2 block diagram

1.5.1 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.

1.5.2 Floating-point unit and co-processor

The LEON model does not include an FPU, but provides a direct interface to the Meiko FPU core, and a general interface to connect other floating-point units. A generic co-processor interface is provided to allow interfacing of custom co-processors.

1.5.3 Cache sub-system

Separate instruction and data caches are provided, each configurable to 1 - 64 kbyte, with 8 - 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.

1.5.4 Debug support unit

The (optional) debug support unit (DSU) allows non-intrusive debugging on target hardware. The DSU allows to insert instruction and data 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 peformance and has low area complexity. Communication to an outside debugger (e.g. gdb) is done using a dedicated UART (RS232).

1.5.5 Memory interface

The flexible memory interface provides a direct interface PROM, SRAM and memory mapped I/O devices. The memory areas can be programmed to either 8-, 16- or 32-bit data width.

1.5.6 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.

1.5.7 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.

1.5.8 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.

1.5.9 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.

1.5.10 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.

1.5.11 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.

1.5.12 Boot loader

A on-chip boot loader can optionally be enabled, allowing to boot the processor and download applications without any external boot prom. This feature is mostly suitable for FPGA implementations. In larger FPGAs, a monitor compatible with the GNU debugger (gdb) can also be included.

1.5.13 Watchpoint registers

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.

1.6 Performance

Using 4K + 4K caches and a 16x16 multiplier, the dhrystone 2.1 benchmark reports 1,550 iteration/s/MHz. This translates to 1.0 dhrystone MIPS/MHz using the VAX 11/780 value a reference for one MIPS.

2 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.

2.1 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 (iterative, 16x16, 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.

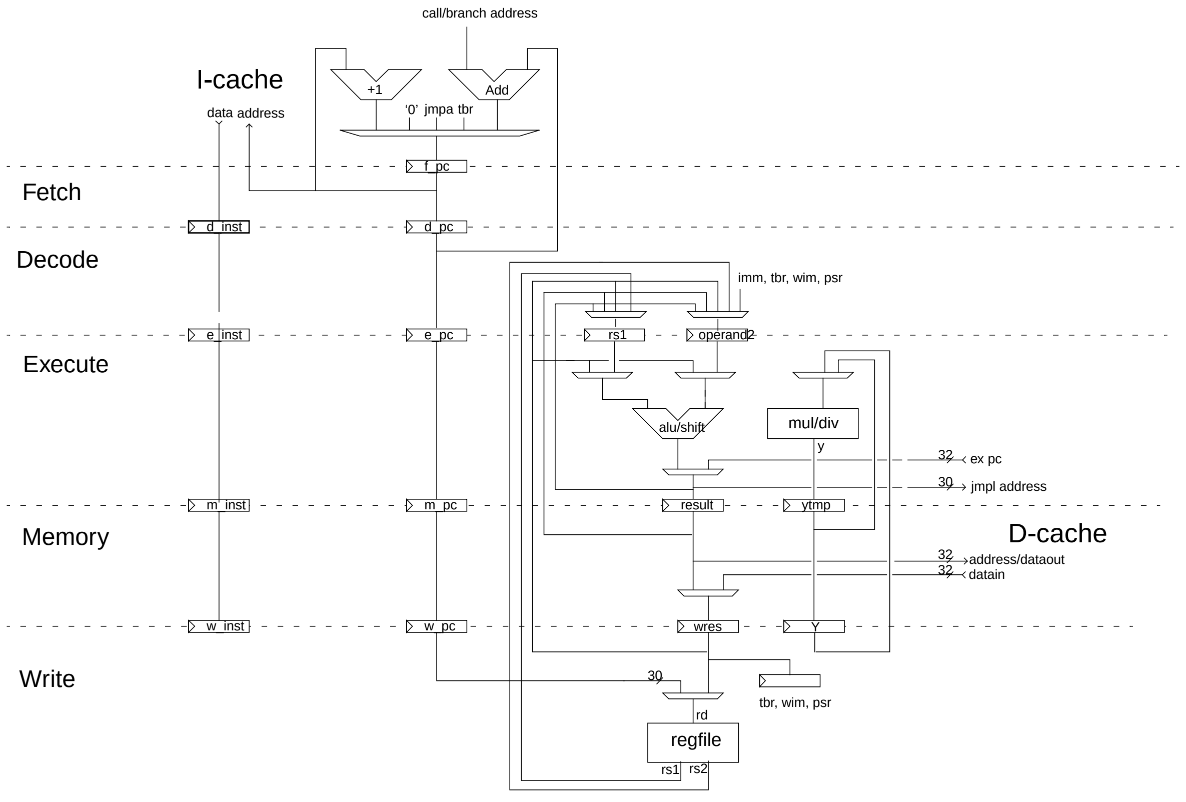


Figure 2: LEON integer unit block diagram

2.2 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 E-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	1/2/4/35*
SDIV/UDIV	35
Taken Trap	4
Atomic load/store	3
All other instructions	1

Table 1: Instruction timing

* depends on multiplier configuration

2.3 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 64 for more details).

2.4 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.

2.5 Divide instructions

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

2.6 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.

2.7 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	Instruction or data watchpoint 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

2.8 Watch-points

The integer unit can be configured to include up to four hardware watch-points. Each watch-point 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:

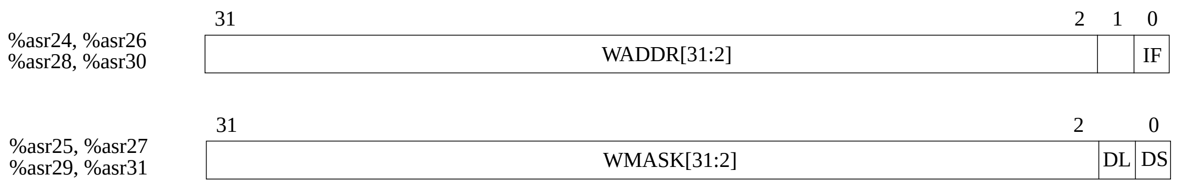


Figure 3: Watch-point registers

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 watch-point 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 watch-point function.

2.9 Floating-point unit and 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 61 for interfacing details.

The LEON model can also be connected to the Meiko floating-point core, thereby providing full floating-point support according to the SPARC-V8 standard. Two interface options are available: either a parallel interface identical to the above described co-processor interface, or an integrated interface where FP instruction do not execute in parallel with IU instruction. The FPU interface is enabled/selected by setting of the FPU element of the configuration record.

The integrated 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.

3 Cache sub-system

3.1 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 - 0xA 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 4: 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 in the model:

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 5: Default cache table

4 Instruction cache

4.0.1 Operation

The LEON instruction cache is a direct-mapped cache, configurable to 1 - 64 kbyte. The instruction cache is divided into cache lines with 8 - 32 bytes of data. Each line has a cache tag associated with it consisting of a tag field and one valid bit for each 4-byte sub-block. On an instruction cache miss to a cachable location, the instruction is fetched and the corresponding tag and data line updated.

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.

4.0.2 Instruction cache flushing

The instruction 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 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.

4.0.3 Diagnostic cache access

Diagnostic reads of the tags is possible by executing an LDA instruction with ASI=0xC. Address bits making up the cache offset will be used to index the tag to be read, all other address bits are ignored. Similarly, the data sub-blocks may be read by executing an LDA with ASI=0xD. The cache offset indexes the line to be read while A[4:2] indexes which of the sub-blocks to be read.

The tags can be directly written by executing a STA with ASI=0xC. The cache offset will index the tag to be written, and D[31:12] is written into the ATAG filed (see below). The valid bits are written with the D[7:0] of the write data. The data sub-blocks can be directly written by executing a STA with ASI=0xD. The cache offset indexes the cache line and A[4:2] selects the sub-block. Note that diagnostic access to the cache is not possible during a FLUSH operation and will cause a data exception (trap=0x09) if attempted.

4.0.4 Instruction cache tag

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

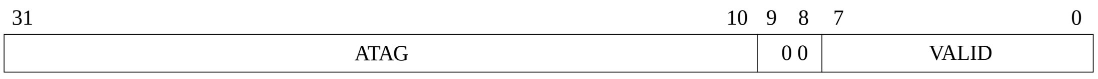


Figure 4: Instruction cache tag layout

Field Definitions:

- [30:10]: Address Tag (ATAG) - Contains the tag address of the cache line.
- [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 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.

4.1 Data cache

4.1.1 Operation

The LEON data cache is a direct-mapped cache, configurable to 1 - 64 kbyte. The write policy for stores is write-through with no-allocate on write-miss. The data cache is divided into cache lines of 8 - 32 bytes. Each line has a cache tag associated with it, containing a tag field and one valid bit per 4-byte sub-block. On a data cache read-miss to a cachable location, 4 bytes of data are loaded into the cache from main memory. 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.

4.1.2 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.

4.1.3 Data cache flushing

The data cache can be flushed by executing the FLUSH instruction, setting the FD bit in the cache control register, or by writing any location with ASI=0x6. The flushing will take one cycle per line during which the IU will not be halted, but during which the data 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.

4.1.4 Diagnostic cache access

Diagnostic software may read the tags directly by executing a single word load alternate space instructions in ASI space 0xE. The cache offset indexes the tag to be read, all other address bits are ignored. Similarly, the data sub-blocks may be read by executing a single word load alternate space instructions in ASI space 0xF. The cache offset indexes the line to be read while A[4:2] index which of the sub-blocks to be read.

The tags can be directly written by executing single word store alternate space instructions in ASI space 0xE. The cache offset indexes the tag to be written, and A[31:10] is written into the ATAG filed (see below). The valid bits are written with the D[7:0] of the write data.

The data sub-blocks can be directly written by executing single word store alternate space instructions in ASI space 0xF. Address bits The cache offset indexes the cache line and A[4:2] selects the sub-block. The sub-block is written with the write data.

Note that diagnostic access to the cache is not possible during a FLUSH operation. An attempt to perform a diagnostic access during an ongoing flush will cause a data exception trap (trap = 0x09).

4.1.5 Data cache tag

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

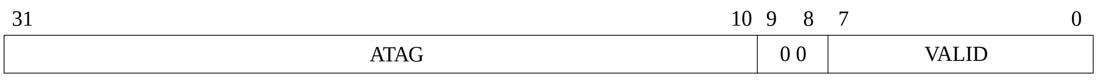


Figure 5: Data cache tag layout

Field Definitions:

- [30:12]: Address Tag (ATAG) - Contains the address of the data held in the cache line.
- [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.

4.2 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	23	22	21		16	15	14	6	5	4	3 2	1 0
RESERVED	DS	FD	FI	RESERVED	IB	IP	DP	RESERVED	DF	IF	DCS	ICS

Figure 6: Cache control register

Field Definitions:

- [31:23]: Reserved
- [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 AMBA on-chip buses

5.1 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.

5.2 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 6 below shows the default address allocation.

Address range	**Size**	**Mapping**	**Module**
0x00000000 - 0x1FFFFFFF 0x20000000 - 0x3FFFFFFF 0x40000000 - 0x7FFFFFFF	512 M 512 M 1G	Prom Memory bus I/O Ram	Memory controller
0x80000000 - 0x8FFFFFFF	256 M	On-chip registers	APB bridge
0x90000000 - 0x9FFFFFFF	256 M	Debug support unit	DSU

Table 6: AHB address allocation

An attempt to access a non-existing device will generate an AHB error response.

5.3 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 7.

5.4 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.

6 On-chip peripherals

6.1 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 pending register
0x80000004	Memory configuration register 2	0x800000B4	Secondary interrupt mask register
0x80000008	Reserved	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	Scaler counter register		
0x80000064	Scaler 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 register		

Table 7: On-chip registers

6.2 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.

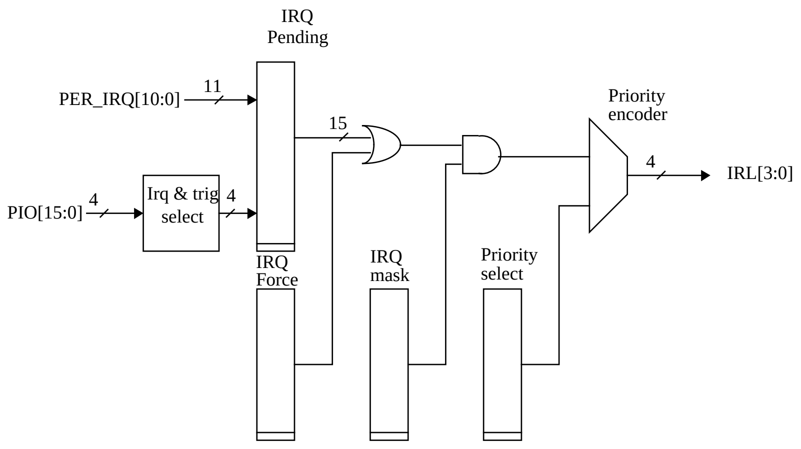


Figure 7: Interrupt controller block diagram

6.2.1 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 safely handle this interrupt.

6.2.2 Interrupt assignment

Table 8 shows the assignment of interrupts.

Interrupt	Source
15	user defined
14	user defined
13	user defined
12	user defined
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 8: Interrupt assignments

6.2.3 Control registers

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

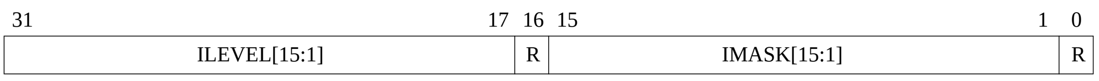


Figure 8: Interrupt mask and priority register

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

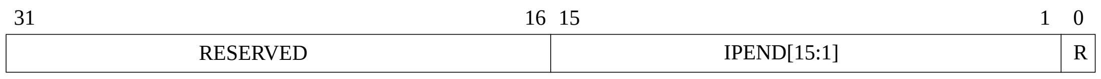


Figure 9: Interrupt pending register

Field Definitions:

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

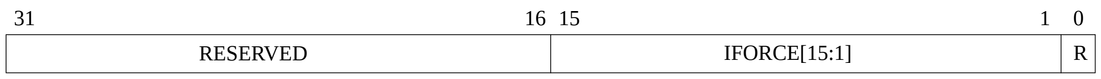


Figure 10: Interrupt force register

Field Definitions:

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

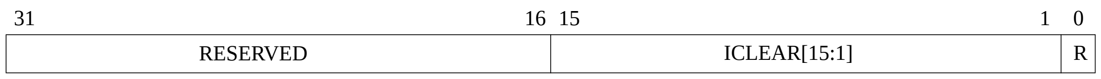


Figure 11: Interrupt clear register

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

6.3 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.

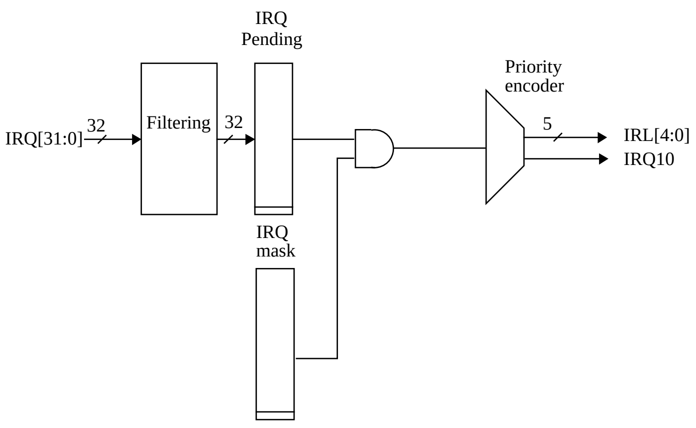


Figure 12: Secondary interrupt controller block diagram

6.3.1 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.

6.3.2 Control registers

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

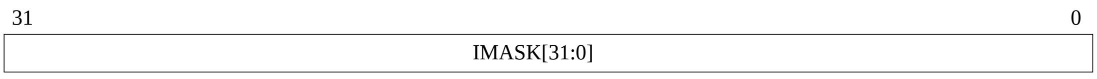


Figure 13: Interrupt mask register

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

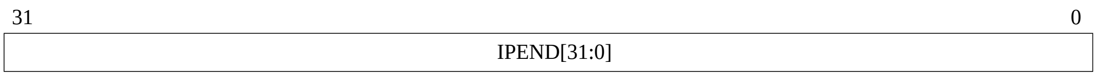


Figure 14: Interrupt pending register

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

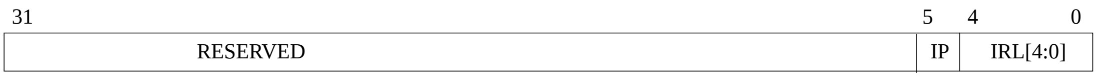


Figure 15: Interrupt status register

- [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.

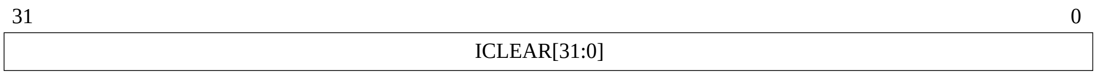


Figure 16: Interrupt clear register

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

6.4 Timer unit

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

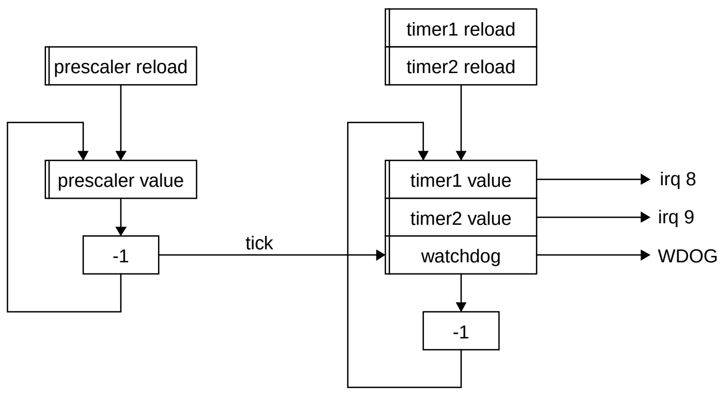


Figure 17: Timer unit block diagram

6.4.1 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).

6.4.2 Registers

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

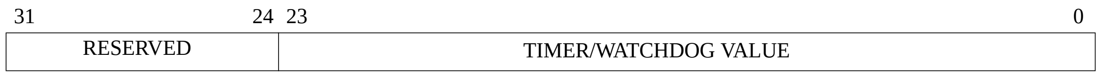


Figure 18: Timer 1/2 and Watchdog counter registers

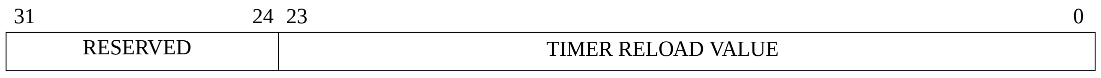


Figure 19: Timer 1/2 reload registers

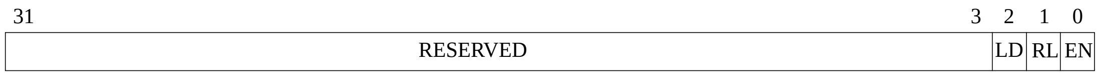


Figure 20: Timer 1/2 control registers

- [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.

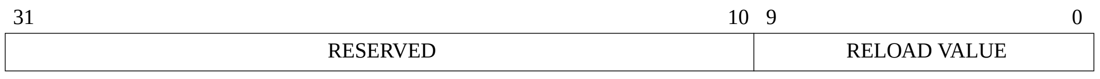


Figure 21: Prescaler reload register

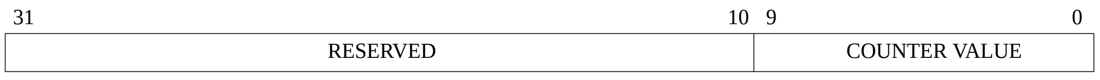


Figure 22: Prescaler counter register

6.5 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.

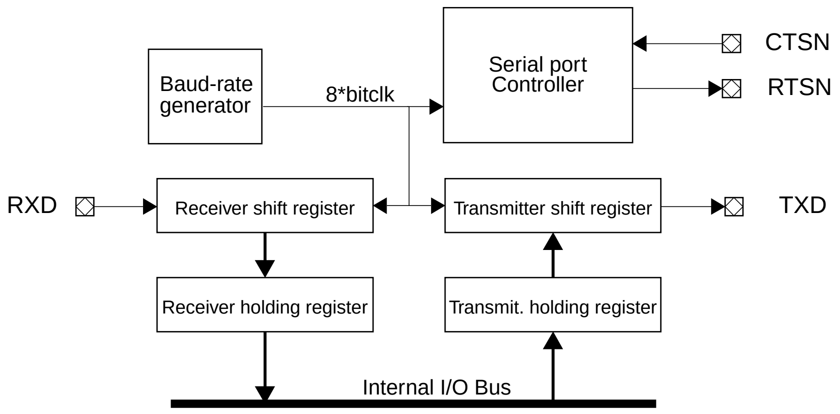


Figure 23: UART block diagram

6.5.1 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

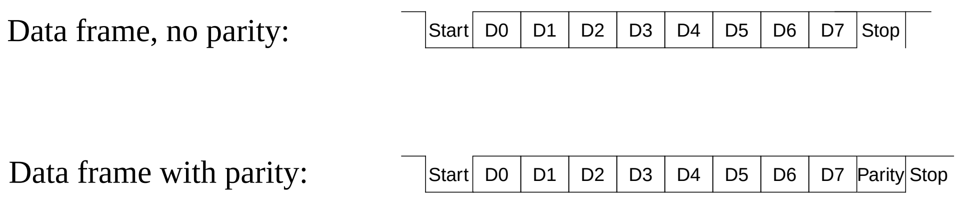

Data frame, no parity:
Start D0 D1 D2 D3 D4 D5 D6 D7 Stop
Data frame with parity:
Start D0 D1 D2 D3 D4 D5 D6 D7 Parity Stop

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.

6.5.2 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 sampled three times for each bit and averaged to filter out noise.

During this process 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.

6.5.3 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.

6.5.4 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.

6.5.5 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.

6.5.6 UART registers

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

Figure 25: 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 = odd parity, 1 = even 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 - 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 26: 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 27: UART scaler reload register

6.6 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 memory configuration" on page 39).

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).

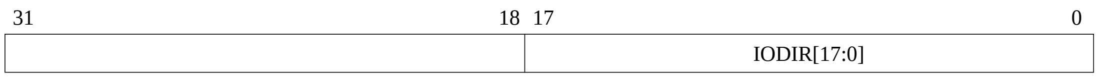


Figure 28: I/O port direction register

- 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 29) defines which port should generate each interrupt and how it should be filtered.

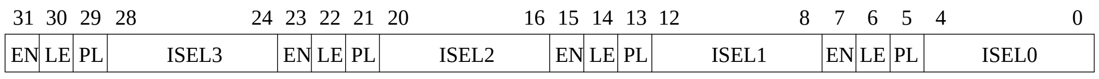


Figure 29: I/O port interrupt configuration register

- 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 9: UART/IO port usage

6.7 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 30 shows the layout of the register.

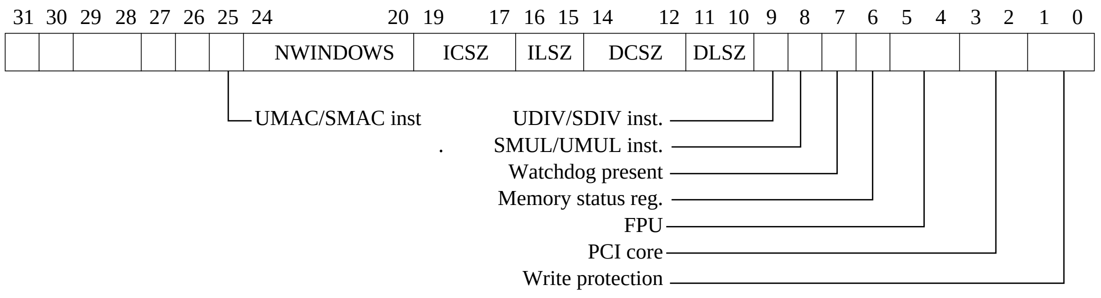


Figure 30: LEON configuration register

- [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)

6.8 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, two consecutive stores to the power-down register should be performed. 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.

6.9 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 memory status register will store the access and error types. The registers are updated when an error occur, and the NE (new error) is set. When the NE bit is set, interrupt 1 is generated to inform the processor about the error. After an error, the NE bit has to be reset by software.

Figure 31 shows the layout of the AHB status register.

31	8	7	6 3	2 0
RESERVED	NE	RW	HMASTER	HSIZE

Figure 31: AHB status register

- [8]: NE - New error. Set when a new 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 External memory access

7.1 Memory interface

The memory bus provides a direct interface to PROM, static RAM and memory mapped I/O devices. Chip-select decoding is done for two PROM banks, one I/O bank and four RAM banks. Figure 32 shows how the connection to the different device types is made.

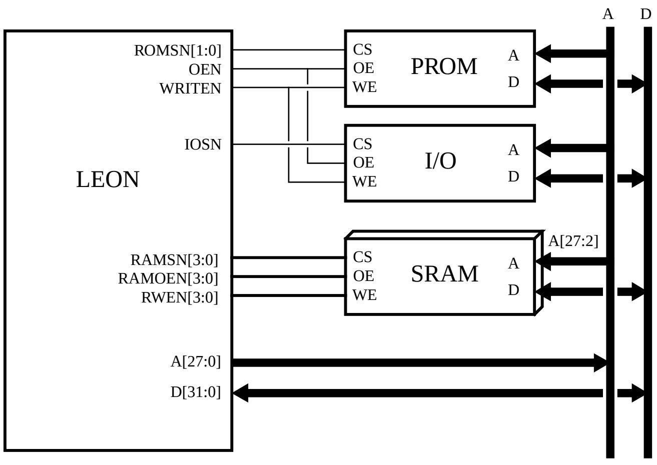


Figure 32: Memory device interface

7.2 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 (MCR1 & MCR2) through the APB bus. The memory bus supports three types of devices: prom, ram and local I/O. The memory bus can also be configured in 8-bit mode for applications with low memory and performance demands. The controller can decode a 2 Gbyte address space, divided according to table table 10:

Address range	**Size**	**Mapping**
0x00000000 - 0x1FFFFFFF	512 M	Prom
0x20000000 - 0x3FFFFFFF	512M	I/O
0x40000000 -0x5FFFFFFF	512 M	RAM (RAMSN[3:0])
0x50000000 -0x7FFFFFFF	512M	RAM (RAMSN[4])

Table 10: Memory controller address map

7.3 RAM access

The RAM 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 static RAM consists of two data cycles and between zero and three waitstates. Acesses to RAMSN[4] can further be stretched by de-asserting BRDYN until the data is availble. 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 33 shows the basic read cycle waveform (zero waitstate).

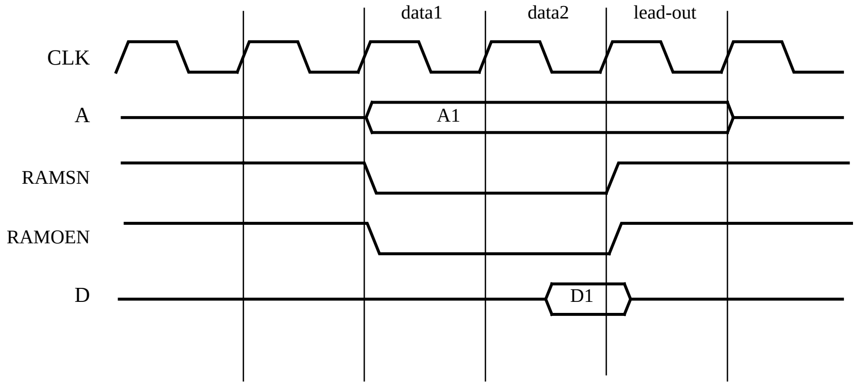


Figure 33: Static ram read cycle

For read accesses to RAMSN[3:0], a separate output enable signal (RAMOEN[n]) is provided for each RAM bank and only asserted when that bank is selected. Memory connected to to RAMSN[4] should use the OEN signal instead which is asserted on any read cycle. A write access is similar to the read access but has takes a minimum of three cycles:

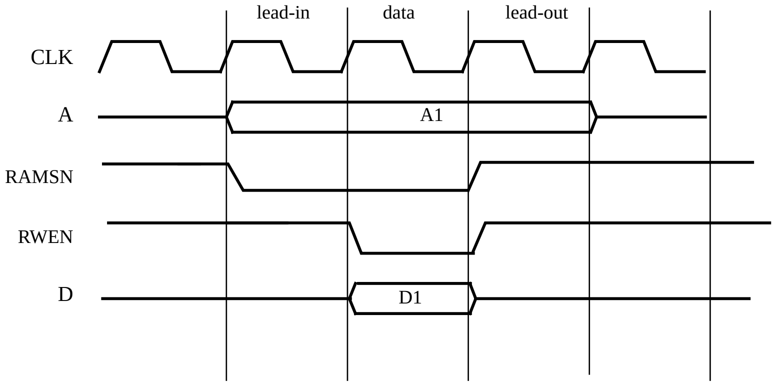


Figure 34: Static ram write cycle

Through a 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 you memory used a common write strobe for the full 16- or 32-bit data, set the read-modify-write bit MCR2 which will enable read-modify-write cycles for sub-word writes.

7.4 PROM access

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

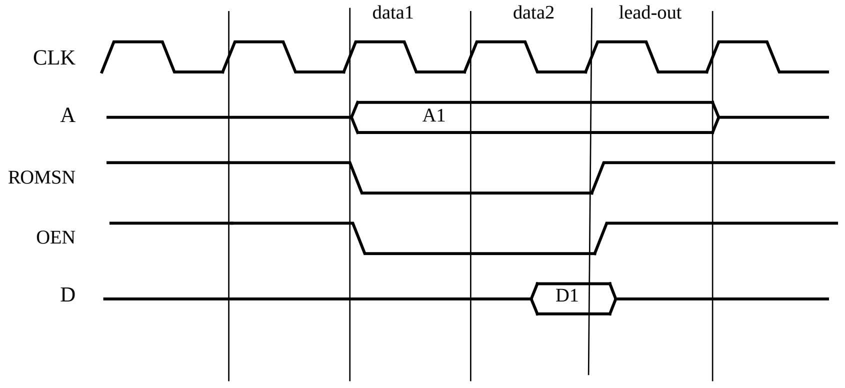


Figure 35: Prom read cycle

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 68), 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.

7.5 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.

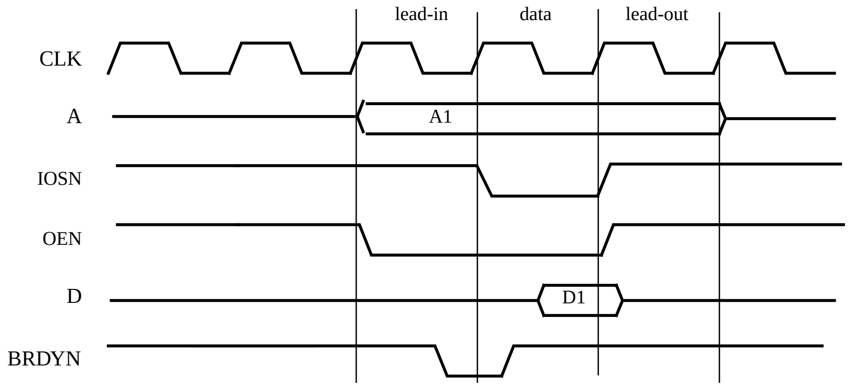


Figure 36: I/O read cycle

7.6 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.

7.7 8-bit and 16-bit memory configuration

To support applications with low memory and performance requirements efficiently, it is not necessary to always have full 32-bit memory banks. The RAM 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 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 37 shows an interface example with 8-bit PROM and 8-bit RAM. Figure 38 shows an example of a 16-bit memory interface.

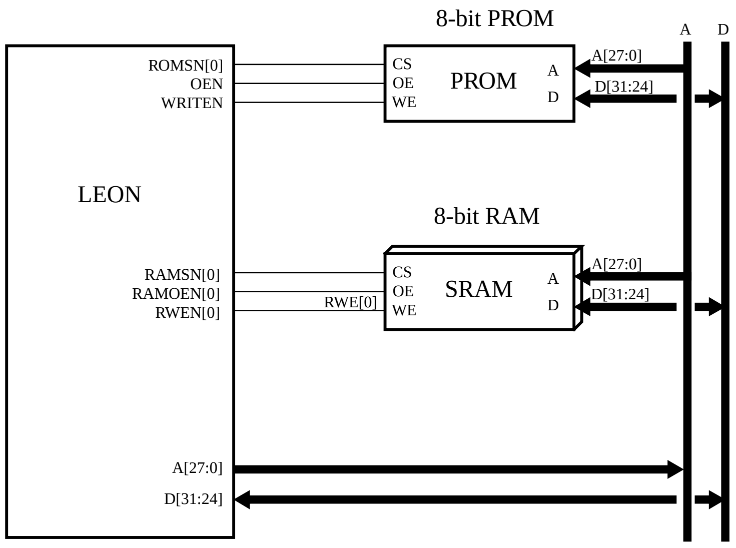


Figure 37: 8-bit memory interface example

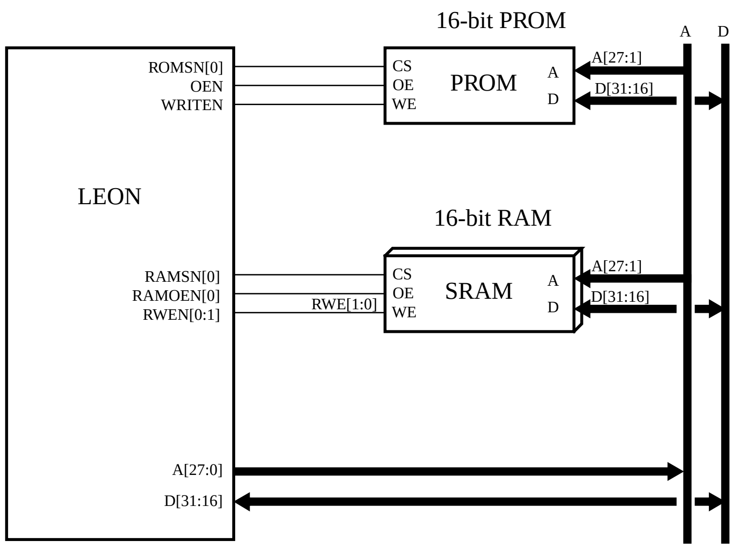


Figure 38: 16-bit memory interface example

7.8 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.

7.9 Memory configuration register 1

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

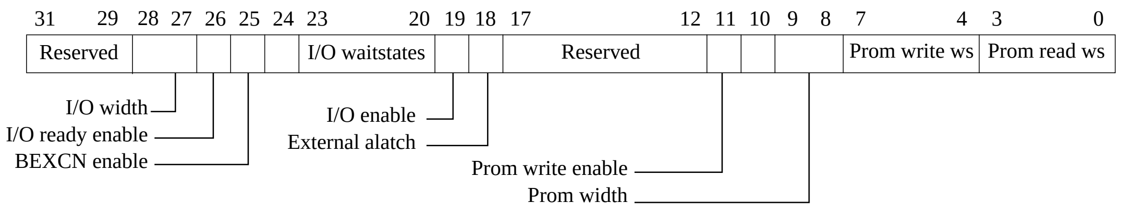


Figure 39: Memory configuration register 1

- [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
- [18]: External address latch enable. If set, the address is sent out unlatched and must be latched by external address latches.
- [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.

7.10 Memory configuration register 2

Memory configuration register 2 is used to control the timing of static ram accesses.

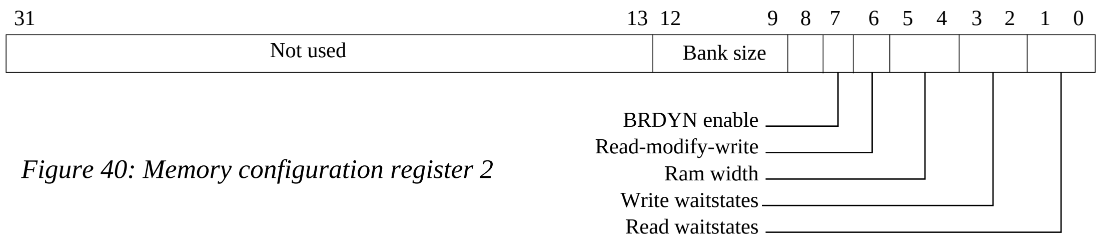


Figure 40: Memory configuration register 2

- [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).

7.11 Write protection

Write protection is provided to protect the memory and I/O areas 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 Mbyte. Each block protect unit is controlled through a control register (figure 41). 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 not 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 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 41: 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.

7.12 Using BRDYN

The BRDYN signal can be used to strecth 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 streched until BRDYN is asserted. BRDYN should be asserted in the cycle preceeding the last one. The use of BRDYN can be enabled separately for the I/O and RAM areas.

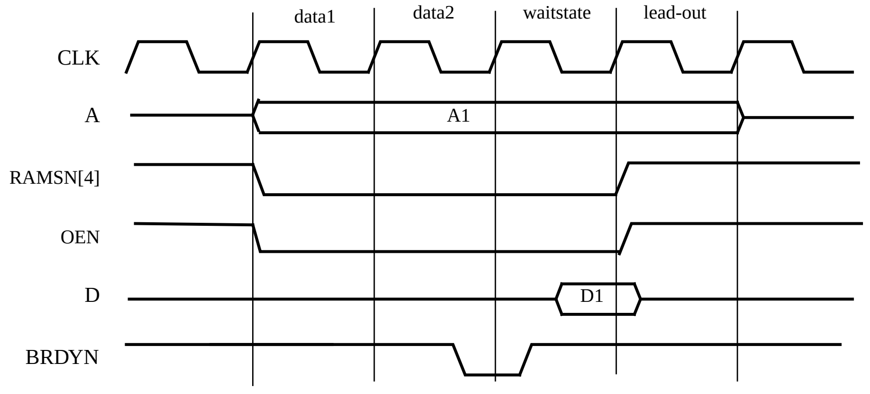


Figure 42: RAM read cycle with one BRDYN controlled waitstate

7.13 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 configurtion register 1, an error respons 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).

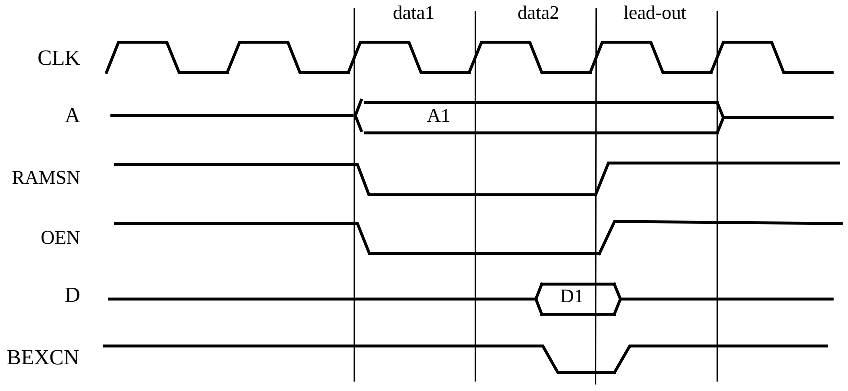


Figure 43: Read cycle with BEXCN

7.14 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.

7.15 Cache configuration

The memory controller marks the PROM and RAM areas as cacheable and the I/O area as non-cacheable.

8 Hardware debug support

8.1 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 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).

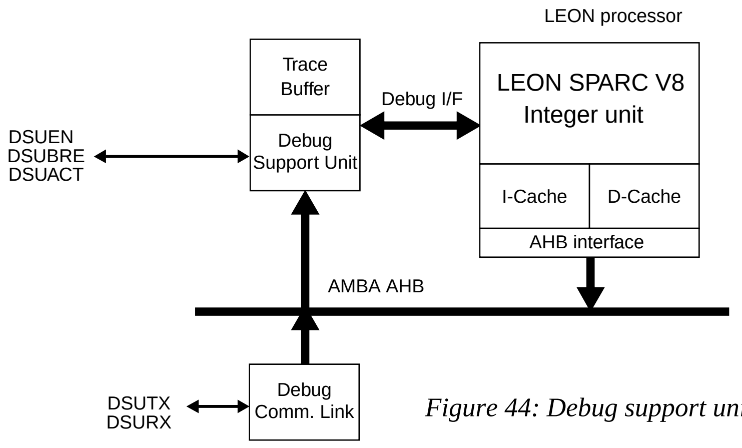


Figure 44: Debug support unit and comm. link

8.2 Debug support unit

8.2.1 Overview

The debug support unit is used 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 is controlled 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.

8.2.2 Trace buffer

The trace buffer consists of a circular buffer that stores executed instructions 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 words (= 2 kbyte), but can be configured to any size through the VHDL model configuration record.

The trace buffer is 128 bits wide, the information stored is indicated in table 11 and table 12 below:

Bits	Name	Definition
127	AHB breakpoint hit	Set to ‘1’ if a DSU AHB breakpoint hit occurred.
126	-	Unused
125:96	DSU counter	The value of the DSU counter
95:92	IRL	Processor interrupt request input
91:88	PIL	Processor interrupt level (psr.pil)
95:80	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 11: 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	DSU counter	The value of the DSU 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 12: 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 onlhy 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 trace buffer can executed instructions, transfers on AHB or both (mixed-mode). The trace buffer control register contains two counter that store the address of which location of the trace buffer will be written on next trace. Since the buffer is circular, it actually points to the oldest entry in the buffer. The indexes are automatically incremented after each stored trace entry.

In mixed mode, the buffer is divided on two halfs, with instructions stored in the lower half and AHB transfers in the upper half. The MSB bit of the AHB index counter is then automatically kept high, while the MSB of the instruction index counter is kept low.

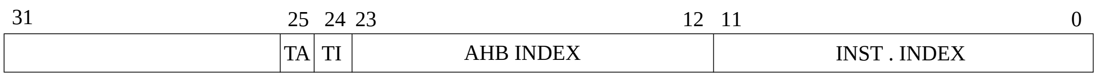


Figure 45: DSU control register

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

Note that the VHDL model configuration allows to disable the mixed-mode. In this case, only the instruction trace index counter is provided, and is used also during AHB tracing.

8.2.3 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 46: 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.

8.2.4 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 TDELAY field in the DSU control register to a non-zero value. In this case, the TDELAY 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.

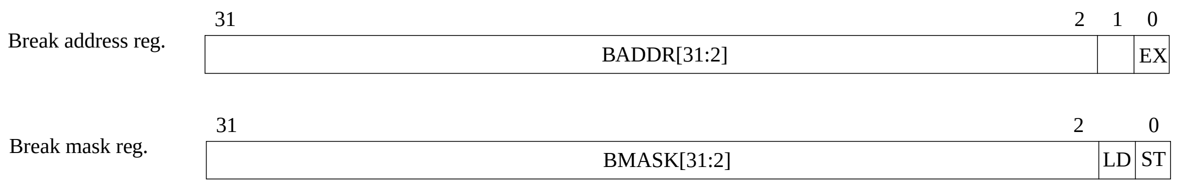


Figure 47: DSU breakpoint registers

8.2.5 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).

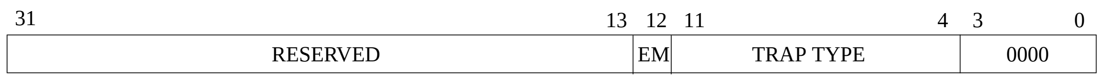


Figure 48: DSU control register

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

8.2.6 DSU memory map

DSU memory map can be seen in table 13 below.

Address	Register
0x800000c4	DSU UART status register
0x800000c8	DSU UART control register
0x800000cc	DSU UART scaler 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

Table 13: DSU address space

Address	Register
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 - 0x90180000	Instruction cache data
0x90180000 - 0x901C0000	Data cache tags
0x901C0000 - 0x90200000	Data cache data

Table 13: DSU address space

The addresses of the IU/FPU registers depends on how many register windows has been implemented and if and 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 (FPU present)
- %f*n* : 0x90020000 + (NWINDOWS*64)

8.3 DSU communication link

8.3.1 Operation

The DSU communication link consists of a UART connected to the AHB bus as a master (figure 49). A simple communication protocol is supported to transmit access parameters and data. A link command consist of a control byte, followed by and 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 51. Through the communication link, a read or write transfer can be generated to any address on the AHB bus.

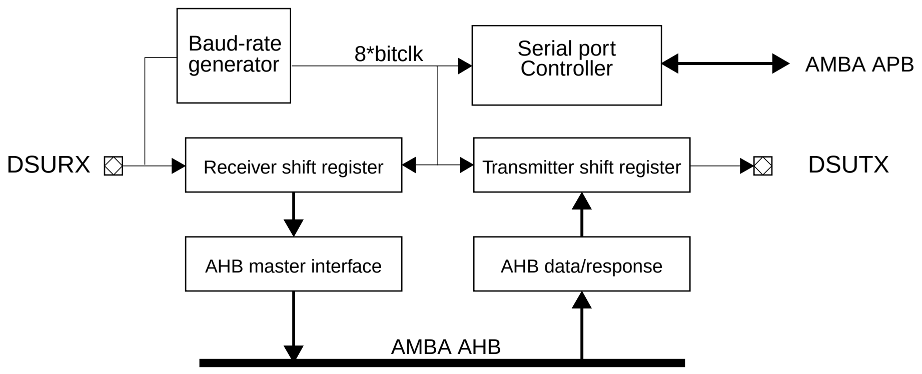


Figure 49: DSU communication link block diagram

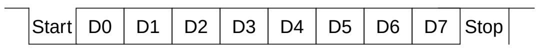


Figure 50: DSU UART data frame

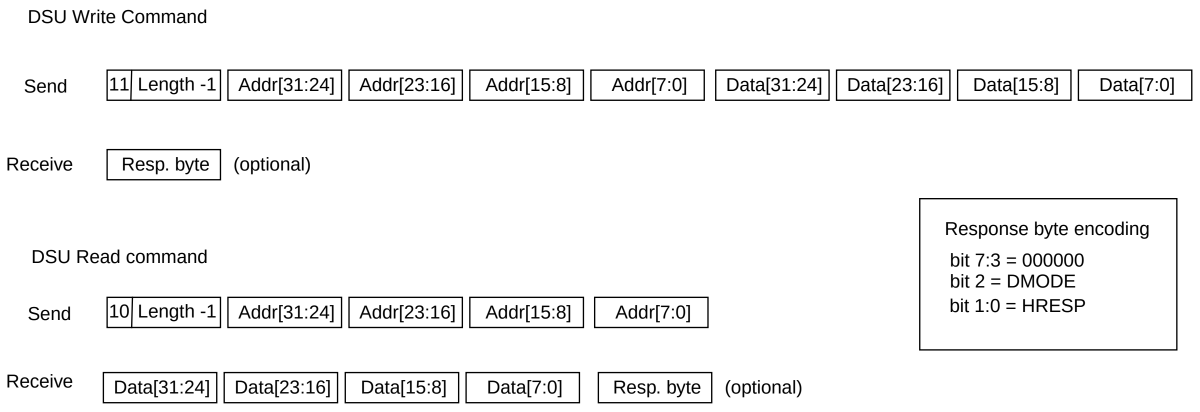


Figure 51: DSU Communication link commands

A response byte is 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.

8.3.2 DSU UART control register

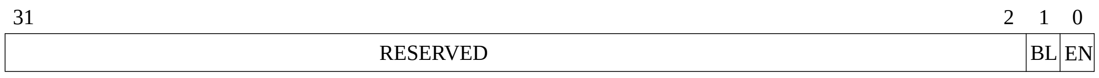


Figure 52: UART control register

- 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.

8.3.3 DSU UART status register

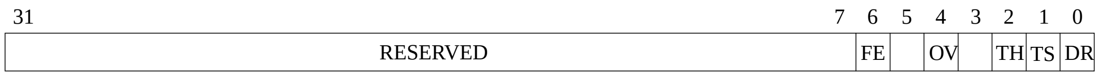


Figure 53: UART status register

- 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.
- 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.

8.3.4 Baud rate generation

The UART contains a 14-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 discovey is also restarted when a ‘break’ is received 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 54: DSU UART scaler reload register

8.4 Common operations

8.4.1 Instruction breakpoints

To insert instruction breakpoints, the breakpoint instruction (ta 1) should be used. This will leave the four IU hardware breakpoints free to be used as data watchpoints. Since cache snooping is only done on the data cache, the instruction cache must be flushed after the insertion or removal of breakpoints. To minimize the influence on execution, it is enough to clear the corresponding instruction cache tag (which is accesible through the DSU).

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

8.4.2 Single stepping

By writing the SS bit and reseting the BN bit in the DSU control register, the processor will resume execution for one instruction and then automatically enter debug mode.

8.4.3 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.

8.4.4 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 standalone booting can be re-programmed.

8.5 Design limitations

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

8.6 DSU monitor

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

8.7 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.

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.

9 Signals

All input signals are latched on the rising edge of CLK. All outputs are clocked on the rising edge of CLK.

9.1 Memory bus signals

Name	Type	Function	Active
A[30:0]	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[3:0]	Output	RAM output enable	Low
RAMSN[3:0]	Output	RAM chip-select	Low
READ	Output	Read strobe	High
ROMSN[1:0]	Output	PROM chip-select	Low
RWEN[3:0]	Output	RAM write enable	Low
WRITEN	Output	Write strobe	Low

Table 14: Memory bus signals

9.2 System interface signals

Name	Type	Function	Active
CLK	Input	System clock	High
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 15: System interface signals

9.3 Signal description

A[30:0] - 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 input provides the output from the DSU communication link transmitter.

ERROR - 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.

RAMOEN[3:0] - RAM output enable (output)

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

RAMSN[3:0] - RAM chip-select (output)

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

READ - Read cycle

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.

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.

10 VHDL model architecture

10.1 Model hierarchy

The LEON VHDL model hierarchy can be seen in table 16 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/CLKGEN	clkgen.vhd	Clock generator
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/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/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

Table 16: LEON model hierarchy

Table 17 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_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_UMC18	tech_umc18.vhd	UMC 0.18 um specific regfile, ram and pad generators
TECH_MAP	tech_map.vhd	Maps mega-cells according to selected target

Table 17: LEON packages

10.2 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.

10.3 Clocking scheme

The model implements two clocking schemes: a continuous clock and the use of multiplexers to enable loading of pipe-line registers, or a gated clock which is stopped during pipe-line stalls. While a continuous clock provides easier timing analysis, gated clocks usually cost less area and power. The selection of clock scheme is done by setting the configuration element GATEDCLK to true or false.

10.4 AMBA buses

10.4.1 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. To be granted the bus, a master must drive the request signal (HREQ). 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;
```

10.4.2 AHB cache aspects

Since no MMU is provided with LEON, the configuration record contains a table which indicates which adresses will be cached by the processor. By default, only the PROM and RAM area of the memory controller are marked as cacheable.

10.4.3 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 priviledged bit is driven when the processor is in supervisor mode, the bufferable and cacheable bits are driven if a cacheable address is accessed.

10.4.4 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 number of APB slaves and their address range is defined through the APB slave table in the TARGET package.

10.5 Floating-point unit and co-processor

10.5.1 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 55

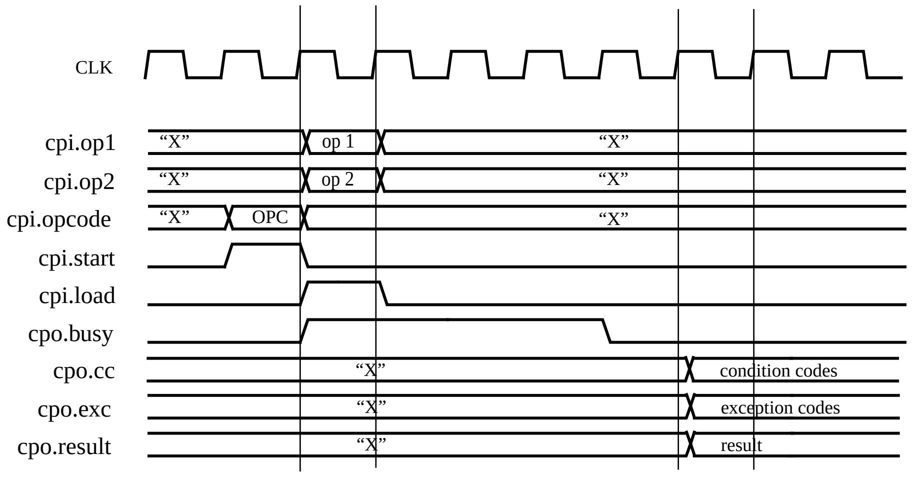


Figure 55: Co-processor execution unit waveform diagram

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.

10.5.2 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 infigure 55 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.

11 Model Configuration

The model is configurable to allow different cache sizes, multiplier performance, clock generation, and target technologies. Several configurations are defined as constant records in the TARGET package while the active configuration record is selected in the DEVICE package. The model is configured from a master configuration record which contains a number of sub-records which each configure a specific module/function:

```
-- complete configuration record type
type config_type is record
  synthesis: syn_config_type;
  iu   : iu_config_type;
  fpu  : fpu_config_type;
  cp  : cp_config_type;
  cache: cache_config_type;
  ahb  : ahb_config_type;
  apb  : apb_config_type;
  mctrl: mctrl_config_type;
  boot : boot_config_type;
  debug: debug_config_type;
  pci  : pci_config_type;
  peri : peri_config_type;
end record;
```

11.1 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, atc35, atc25);
-- synthesis configuration
type syn_config_type is record
  targettech: targettechs;
  infer_ram : boolean;-- infer cache 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
  gatedclk  : boolean;-- select clocking strategy
  rftype : integer;-- register file implementation option
end record;
```

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, three types of target technologies are currently supported: Xilinx Virtex (FPGA), Atmel ATC35 (0.35 um CMOS) and Atmel ATC25 (0.25 um CMOS). 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. Currently, this is only used when **infer_regf** is true and the synthesis tool infers the register file. **Infer_mult** selects how the multiplier is generated, for details see section 11.2 below.

11.2 Integer unit configuration

The integer unit configuration record is used to control the implementation of the integer unit:

```
-- integer unit 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
  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)
  watchpoints: integer range 0 to 4; -- # hardware watchpoints (0-4)
  impl   : integer range 0 to 15; -- IU implementation ID
  version: integer range 0 to 15; -- IU version ID
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	4	6,000
m32x8	4	5,000
m32x16	2	9,000
mx32x32	1	15,000

Table 18: 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 **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. In FPGA 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.

Setting **watchpoint** to a value between 1 - 4 will enable coresponding number of watch-points. Seeting it to 0, will disable all watch-point logic. The **`impl`** and **`version`** fields are used to set the fixed fields in the %psr register.

11.3 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.

11.4 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 proc_cache_config_type is record
  firstaddr: proc_cache_addr_type;
  lastaddr: proc_cache_addr_type;
end record;

type proc_cache_config_vector is array (Natural Range <> ) of
   proc_cache_config_type;
constant proc_cache_config_void : proc_cache_config_type :=
  ((others => '0'), (others => '0'));

type cache_config_type is record
  icachesize: integer;-- size of I-cache in Kbytes
  ilinesize: integer;-- # words per I-cache line
  dcachesize: integer;-- size of D-cache in Kbytes
  dlinesize: integer;-- # words per D-cache line
  dsnoop    : dsnoop_type;-- data-cache snooping
  cachetable : proc_cache_config_vector(0 to PROC_CACHE_MAX-1);
end record;
```

Valid settings for the cache size are 1 - 64 (Kbyte), and must be a power of 2. The line size may be 2 - 4 (words/line). The instruction and data caches may be configured independently.

The default cacheability table marks only prom and ram areas as cacheable:

```
-- standard cacheability config
constant cachetbl_std : proc_cache_config_vector(0 to PROC_CACHE_MAX-1) := (
-- first    last        function   address[31:29]
  ("000", "000"),   -- PROM area    0x0- 0x0
  ("010", "011"),   -- RAM area     0x2- 0x3
   others => proc_cache_config_void);
```

11.5 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
  rawaddr  : boolean;-- enable unlatched address option
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 rawaddr fields enables the raw (unlatched) address output option in the memory controller. If enabled, timing analysis of the address bus might be difficult since the bus outputs can be driven both by registers (synchronous) and combinational logic (asynchronous).

11.6 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 19: DSU trace buffer ram usage

11.7 Peripheral configuration

Enabling of some 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
  irq2cfg      : irq2type;-- secondary interrupt controller config
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.

11.8 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;
```

11.8.1 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 targetted 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.

11.8.2 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
>
```

11.8.3 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.

11.8.4 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 44 for details.

11.9 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;
```

11.9.1 AHB master 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.

11.9.2 AHB slave configuration

The number of AHB slaves and their address range is defined through the AHB slave table. The default table has four pre-defined slaves: the memory controller, APB bridge, DSU and PCI:

```
-- standard slave config
constant ahbslvcfg_dsu : ahb_slv_config_vector(0 to AHB_SLV_MAX-1) := (
-- first    last  index  split  enable  function           HADDR[31:28]
  ("0000", "0111",  0,   false, true), -- memory controller,   0x0- 0x7
  ("1000", "1000",  1,   false, true), -- APB bridge, 128 MB   0x8- 0x8
  ("1001", "1001",  2,   false, true), -- DSU        128 MB   0x9- 0x9
  ("1100", "1111",  3,   false, false),-- PCI initiator        0xC- 0xF
   others => ahb_slv_config_void);
```

The table also indicates if the slave is capable of returning a SPLIT response; if so, the *split* element should be set to true, thereby generating the necessary split support logic in the AHB arbiter. 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.

11.9.3 APB configuration

The number of APB slaves and their address range is defined through the APB slave table in the TARGET package. The default table has 14 slaves.

```
constant APB_SLV_MAX   : integer := 16;  -- maximum APB slaves
constant APB_SLV_ADDR_BITS : integer := 10;  -- address bits to decode APB slaves
subtype apb_range_addr_type is std_logic_vector(APB_SLV_ADDR_BITS-1 downto 0);
type apb_slv_config_type is record
  firstaddr: apb_range_addr_type;
  lastaddr: apb_range_addr_type;
  index   : integer;
  enable: boolean;
end record;
type apb_slv_config_vector is array (Natural Range <> ) of apb_slv_config_type;
constant apb_slv_config_void : apb_slv_config_type :=
  ((others => ’0’), (others => ’0’), 0, false);

-- standard config
constant apbslvcfg_std : apb_slv_config_vector(0 to APB_SLV_MAX-1) := (
--   first         last      index  enable     function          PADDR[9:0]
( "0000000000", "0000001000",  0,   true), -- memory controller, 0x00 - 0x08
( "0000001100", "0000010000",  1,   true), -- AHB status reg.,   0x0C - 0x10
( "0000010100", "0000011000",  2,   true), -- cache controller,  0x14 - 0x18
( "0000011100", "0000100000",  3,   true), -- write protection,  0x1C - 0x20
( "0000100100", "0000100100",  4,   true), -- config register,   0x24 - 0x24
( "0001000000", "0001101100",  5,   true), -- timers,            0x40 - 0x6C
( "0001110000", "0001111100",  6,   true), -- uart1,             0x70 - 0x7C
( "0010000000", "0010001100",  7,   true), -- uart2,             0x80 - 0x8C
( "0010010000", "0010011100",  8,   true), -- interrupt ctrl     0x90 - 0x9C
( "0010100000", "0010101100",  9,   true), -- I/O port           0xA0 - 0xAC
( "0010110000", "0010111100", 10,   false),-- 2nd interrupt ctrl 0xB0 - 0xBC
( "0011000000", "0011001100", 11,   false),-- DSU uart           0xC0 - 0xCC
( "0100000000", "0111111100", 12,   false),-- PCI configuration  0x100- 0x1FC
( "1000000000", "1011111100", 13,   false),-- PCI arbiter        0x200- 0x2FC
  others => apb_slv_config_void);

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

constant apb_std : apb_config_type := (table => apbslvcfg_std);
```

The table is used to automatically configure the AHB/APB bridge. To add APB slaves, edit the slave configuration table and add your modules in MCORE. The APB slaves should be connected to the `apbi/apbo` buses. The `index` field in the table indicates which bus index the slave should connect to.

12 Simulation

12.1 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/modelsim/	Modelsim simulator support files
leon/pmon	Boot-monitor
leon/syn	Synthesis support files
leon/tbench	LEON VHDL test bench
leon/tsource	LEON test bench (C source)

12.2 Compilation of the model

On unix systems (or MS-windows with cygwin installed), the LEON VHDL model and test bench can be built using ‘make’ in the top directory. Doing make without a target (or ‘make all’) will build the model and test benches using the modeltech compiler. Doing a ‘make vss’ will build the model with Synopsys VSS.

To use an other simulator, the makefiles in the leon and tbench sub-directories have to be modified to reflect the simulator commands. On non-unix systems (e.g. windows), the compile.bat file in the leon and tbench directories can be used to compile the model in correct order.

12.3 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 and configuration, by setting the appropriate generics. A default configuration of the test bench, TBDEF, is in tbench/tbdef.vhd. The file tbench/tbleon.vhd contains a number of alternative configuration using the generic test bench.

Once the LEON model have been compiled, one of the test benches (e.g. TBDEF) can be simulated to verify the behaviour of the model. **Simulation should be started in the top directory**. The output from the simulation should be as follows:

```
# # *** 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.

12.4 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 '_d' have disassembly enabled.

12.5 Test suite

The supplied test suite which is run by the test bench and 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 tests' in the top directory or 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 programs 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.

12.6 Simulator specific support

The file 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 directory to provide appropriate library mapping. The complete model can be compiled from within modelsim by executing the modelsim/compile.do file:

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

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.

12.7 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, enable the cache-ram initialisation routines in tsource/leon_test.c and rebuild the test programs (make all).

13 Synthesis

13.1 General

The model is written with synthesis in mind and has been tested with Synopsys DC, Synopsys FPGA-Compiler (FPGA-Express), Exemplar Leonardo and Synplicity Synplify synthesis tools. Technology specific cells are used to implement the IU/FPU register files, cache rams and pads. These cells can be automatically inferred (Synplify and Leonardo only) 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"*.

13.2 Synthesis procedure

Synthesis should be done from the 'syn' directory. It includes scripts/project-files for Synplify, Synopsys-DC, Synopsys-FC2 and Leonardo. The source files are read from the leon directory, so it is essential that the configuration in the TARGET and DEVICE packages is correct. To simplify the synthesis procedure, a number of pre-defined configuration are provided in the TARGET package. The selection of the active configuration is done in the DEVICE package. The following table shows the characteristics of some of the pre-defined configurations:

Configuration	cache	regfile	mul/div	rom	pads	target	syntool
fpga_2k2k	inferred	inferred	none	none	inferred	any	synp, leo
fpga_2k2k_softprom	inferred	inferred	none	inferred	inferred	any	synp, leo
fpga_2k2k_v8_softprom	inferred	inferred	inferred	inferred	inferred	any	synp, leo
virtex_2k2k_blockprom	inferred	instance	none	instance	inferred	virtex	any
virtex_2k2k_v8_blockprom	inferred	instance	inferred	instance	inferred	virtex	any
gen_atc25	instance	instance	instance	none	instance	ATC25	any
gen_atc35	instance	instance	instance	none	instance	ATC35	any
gen_fs90	instance	instance	instance	none	instance	FS90AB	any

Table 20: Some pre-defined synthesis configurations

Note:

- 8/16-bit memory support is optional, make sure that you enable the option(s) if needed.
- Make sure that the selected configuration in the DEVICE package correctly reflects your synthesis tools and target technology!

13.2.1 Synplify

To synthesise LEON using Synplify, start synplify in the syn directory and open leon.prj. A synthesis run takes about 5 minutes on a 1 GHz Pentium-III PC (128 MB ram necessary). The table below shows some obtained synthesis results (post-layout timing):

Icache (Kbyte)	Dcache (Kbyte	Regfile implement.	Device	Freq (MHz)	Area
2	2	EAB	EPF10K200E-1	20	5,800 LC
8	4	blockRam	XCV300E-8	45	5,000 LUT
8	8	RAM16X1	XCV400E-8	48	6,300 LUT

Table 21: Synplify project files

The FSM complier has unfortunately some bugs and **must** be switched off or some modules (including the divider) will not be correctly synthesised. With synplify_pro, re-timing and pipelining **must** also be disabled to guarantee correct result.

13.2.2 Synopsys-DC

To synthesise LEON using Synopsys DC, start synopsys in the syn 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:

```
/*******************************************/
/* Script to compile leon with synopsys DC */
/* Jiri Gaisler, Gaisler Research, 2001    */
/*******************************************/

/* 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

/* don’t touch anything from here unless you know what you are doing */
```

The top-level constraints are used to generate the appropriate synopsys constraints commands.

13.2.3 Synopsys-FC2 and Synopsys-FE

To synthesise LEON using Synopsys-FC2/FE, start fc2_shell (fe2_shell) in the syn directory and execute the script ‘leon.fc2’. The script will analyse all source files and create a ‘leon’ project. Compilation and optimisation is left to the user. Note: FC2/FE-3.4 do NOT support automatic inference of ram cells, rams have to be directly instantiated from the target library. Currently, only the Xilinx VIRTEX technology is supported through the TECH_VIRTEX package.

13.2.4 Leonardo

Use the following steps to synthesise LEON using Exemplar Leonardo:

- Start Leonardo, and select target technology and device
- Read the technology library
- Set working directory to leon/syn
- Run the 'leon.tcl' script which will analyse and elaborate the design

Compilation and optimisation is left to the user. It is essential that the source files are read with the -dont_elaborate switch, or Leonardo will not be able to properly resolve certain generate statements. Note: **only Leonardo version 2001.1a or later can be used**, the earlier 2000.x versions have bugs in type resolution functions and will fail during analysis of the model. Leonardo is capable of automatically inferring the necessary ram cells for register file and caches.

14 Porting to a new technology or synthesis tool

14.1 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	Function	Instantiation method
TECH_GENERIC	Behavioural models	inferred
TECH_VIRTEX	Generators for Xilinx VIRTEX	direct instantiated
TECH_ATC25	Generators for Atmel ATC25	direct instantiated
TECH_ATC35	Generators for Atmel ATC35	direct instantiated
TECH_FS90	Generators for UMC FS90A/B	direct instantiated
TECH_UMC18	Generators for UMC 0.18 um CMOS	direct instantiated
TECH_MAP	Selects mega-cells depending on configuration	-

Table 22: 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.

14.2 Target specific mega-cells

14.2.1 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 56). 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.

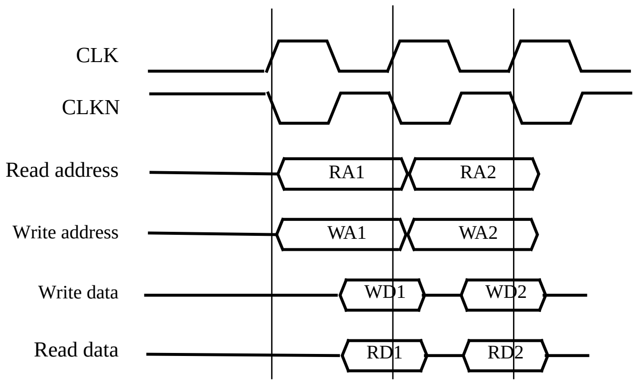


Figure 56: IU register file read/write timing

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.

14.2.2 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.

14.2.3 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 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

Table 23: Cache ram cell sizes

Cache size	Words/line	tag ram	data ram
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 23: 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).

14.2.4 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 and TECH_ATC25 packages include mappings to dual-port rams.

14.2.5 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.

14.2.6 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!