Synthesizable System-on-Chip with 64-bits RISC-V processor core

Technical Reference Manual

GENERATED: JANUARY 15, 2020
AUTHOR: SERGEY KHABAROV

	Name	Position	Signature	Date
Prepared by	Sergey Khabarov	Technical Lead		
Reviewed by	Denis Nefedov			
Approved by				
Approved by				

Chapter 1

Introduction

The rapidly growing area of embedded control applications is representing one of the most time-critical operating environments for today's microcontrollers. Specially developed System-on-Chip design is one of the most effective way to provide maximum performance of the algorithms combining advantages of the hardware and software. Especially it is beneficial for small, highly specialized tasks.

Key elements of such system are Central Processor Unit (CPU) and system bus controller that are defining performance and capabilities of the full system and peripheries.

This document describes implementation of such system based on open-source synthesizable 64-bits RISC-V processor River. One or several processors connected to open-source System Bus Controller implementing AXI4 cross-bar. Processor, bus controller and others elements of a SoC distributed with license Apache 2.0 that shown below.

The tool environment for the River 64-bit processor includes the following tools:

/li Compilers (Asm/C/C++) /li Macro-assemblers, linkers, locators, format-converters /li Full system functional and Cycle-true Simulators /li Debuggers /li Real-time operating system and bare-metal examples /li VHDL chip models /li Project files for different FPGA and EDA tools /li Netlists (by contract) for a specific technology process.

User Manual information

This document was generated using the following program tools:

- **doxygen** (https://www.stack.nl/∼dimitri/doxygen/download.html).
- **MikTex** (https://miktex.org/download)

Documentation build system uses text files in Unicode format and automatically insert build date and other elements of the formatting. Documentation could be shared in ∗**.pdf**, ∗**.html** or ∗**.rtf** formats.

To build documentation in pdf-format use the following sequence of commands:

```
# cd <git_repository>/ergo/docs
# doxygen.exe doxygen/config/Doxyfile
# cd latex
# make.bat
```

Default output file name is *refman.pdf*

1.1 License

9. Accepting Warranty or Additional Liability. While redistributing the Work or Derivative Works thereof, You may choose to offer, and charge a fee for, acceptance of support, warranty, indemnity, or other liability obligations and/or rights consistent with this License. However, in accepting such obligations, You may act only on Your own behalf and on Your sole responsibility, not on behalf of any other Contributor, and only if You agree to indemnify, defend, and hold each Contributor harmless for any liability incurred by, or claims asserted against, such Contributor by reason of your accepting any such warranty or additional liability.

1.2 Open source

Open source repository with VHLD libraries, Debugger and SW examples is available at:

```
https://github.com/sergeykhbr/riscv_vhdl
```

Examples folder includes:

- RISC-V compatibility tests modified to run on current SoC. Original sources are available at: https://github.com/riscv/riscv-tests
- Interrupts initialization and usage
- Interaction with the peripheries (UART, Timer etc)
- Benchmark for the integer performance evaluation Dhrystone 2.1.

This repository is organized around VHDL libraries, where each major IP is assigned to a unique library name. Using separate libraries avoids name clashes between IP cores and hides unnecessary implementation details from the end user.

Implemented Plug&Play system represents the customized system implemented in the open source library GRLIB (CPU Leon3). Customization was done to provide better integration into AXI4 RISC-V SoC. Device memory mapping and adding/removing new AXI4 modules into the system is implemented on project top-level in a similar to GRLIB way.

Information about GNSS (*Satellite Navigation Engine*) you can find at www.gnss-sensor.com

SERGEYKHBR
RISCV_VHDL

Chapter 2

Architectural Overview

The diagram below shows the functional blocks and their basic connectivity within an AMBA AXI4 System Bus.

The system components are built around a **River** CPU which is an open source version of RISC-V compatible processors. Current generation of the 64-bits CPU is compatible to the RISC-V Unprivileged ISA version 2.2 and RISC-V Privileged ISA version 1.11 instruction sets. Default compilers toolchain used with the shared examples `GCC 7.2.0` may be provided as a part of Ubuntu image.

Note

Check the github repository to get the information about the latest supported ISA extensions and instructions.

Functional abstracts of the components contained in a River generic System-on-Chip with enabled commercial IP blocks are provided in the following subchapters.

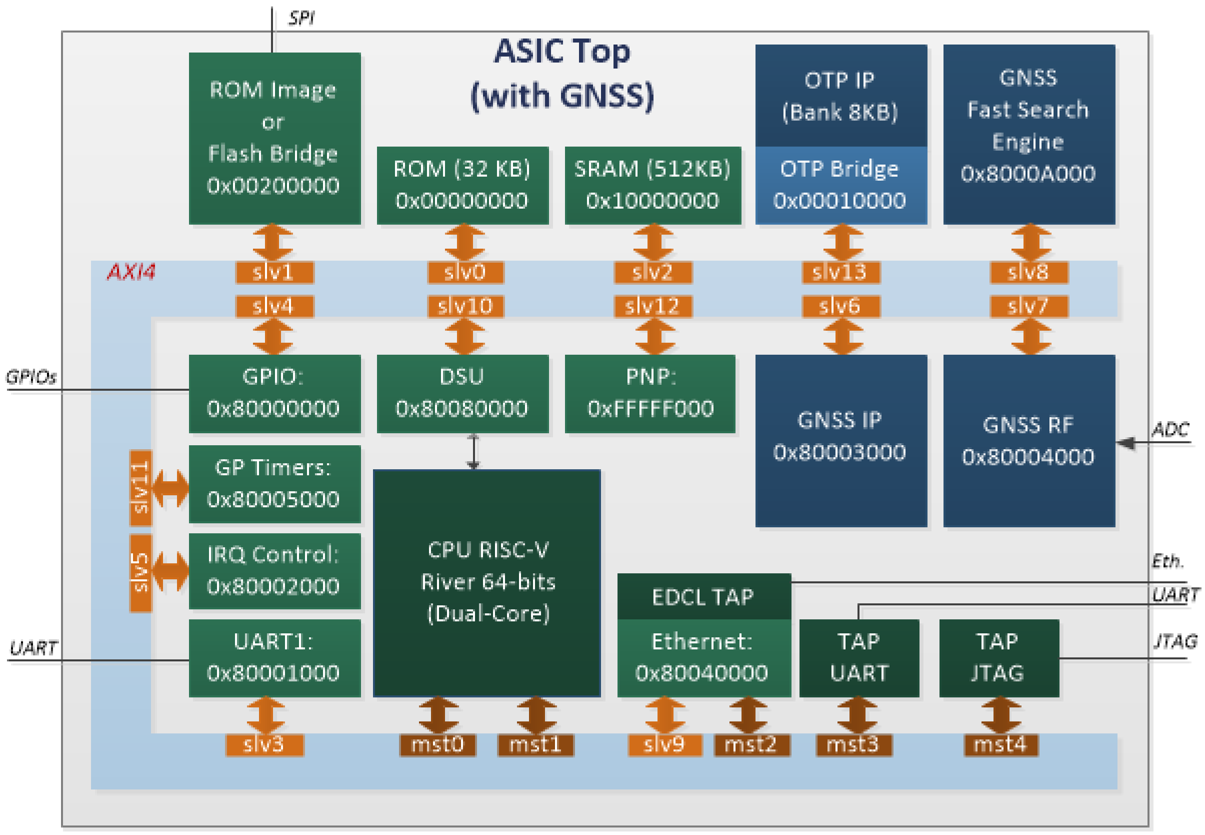

Figure 2-1. SoC Functional Block Diagram.

2.1 Summary of Features

Generic System-on-Chip top-level (`asic_top.vhd`) is built using only blocks available as open source IPs. This project can be used as reference for the further system customization or for porting on any modern FPGA board.

Open source repository also contains customized top-level files (`asic_soc_gnss.vhd`) that contains IP blocks specially developed for a specific technology process, like OTP memory, or provided as a commercial products (GNSS).

Single-Core and Dual-Core configurations either are available in the open-source repository.

Integrated On-Chip Memory

- 32 KB on-chip Boot ROM with pre-built `examples/boot.hex` image
- 512 KB on-chip high-speed SRAM for code and data
- 64 KB Instruction Cache with DP-SRAM blocks
- 64 KB Data Cache with DP-SRAM blocks
- 8 KB on-chip One-Time Programmable Memory (OTP) special type of non-volatile memory
- 256 KB on-chip ROM with pre-built application image (or SPI Flash Controller)

Note

Sizes of the on-chip memory banks are represent the default configuration and may be freely changed during tuning and customization process.

Intelligent On-Chip Peripheral Subsystems

- General peripheries set
  - Interrupt Controller
  - Serial UART port
  - 12 bi-directional IOs
  - 2 General Purpose Timers with RTC
- Debug Support Unit (DSU) for the `River` CPU
  - CPU context switching
  - Run Controller: run/halt or instruction stepping
  - Hardware Breakpoints
  - Access to CPU Integer/FPU Registers banks
  - Access to CPU CSR Registers

- Access to CPU I/D Caches
- Access to CPU Stack Trace Buffer
- Access to CPU Instructions Trace Buffer

- System Watchdog Timer with programmable time intervals
- OTP IP Blocks
- Satellite Navigation (GNSS) IP Blocks

Note

These peripheries parameters represents default system configuration and may be freely changed during tuning and customization process.

High Performance 64-bit CPU with Out-of-Order Memory access

- Single clock cycle instruction execution for most instructions
- Configurable Instruction and Data Caches
- Memory Protection Unit
- Queued Memory access with depth 4 requests in Data path.
- Registers Forwarding with Tagging that allows to continue instruction execution while LOAD instructions are in progress without risk to overwrite new data by loaded from memory.
- 4 clock cycles integer multiplication (64-bit x 64-bit)
- 34 clock cycles integer division (64-bit / 64-bit)
- Floating Point Unit (FPU):
  - 19 clock cycles double precision multiplication
  - 19 clock cycles double precision division
  - 2 clock cycles long or int to double conversion
  - 3 clock cycles double to long or int conversion
- Automatic rounding included
- System stack cache support with automatic stack overflow/underflow detection
- High performance branch, call, and loop processing
- Zero-cycle jump execution

On-Chip Debug Support

- Communication through Ethernet (UDP) or JTAG (5-wire) or UART (2-wire) debug interfaces
- Dual-Core configuration native support
- Hardware and software breakpoints
- Access to any internal register or memory location via debug interfaces

2.2 System Clock

Default configuration implements internal PLL module wrapped into "virtual" component and connected to the `riscv_soc.vhd` level of system. Such architecture allows to re-define PLL instance depending of the target (RTL simulation, certain FPGA or ASIC).

Default clock frequency is equal to 40 MHz. This frequency value is used by Boot ROM firmware and defines UART scale rate to form 115200 Baud output.

Warning

Changing PLL frequency will affect UART output and should be properly handled in boot firmware.

It is possible to use external clock instead of PLL as a system clock directly but it requires the implementation of additional control IO pin. That's a part of system customization process.

2.3 System Reset

Internal System-on-Chip reset signal connected to external reset pin `i_rst` and `PLL Lock` status via special module `reset_global.vhd`. This module provides delayed on 8 cycles output.

Input pin `i_rst` has active level HIGH and usually connected to the reset button. Internal reset signal has active LOW level and becomes HIGH (no reset) only after button unpressed and PLL shows status LOCK.

It is possible to reset system via DSU register (software reset) see DSU registers description.

Note

Optionally General Purpose timer may implement additional reset input `i_rst_ts` used to clear RTC timer and synchronize multi-chips hardware platform.

Chapter 3

Memory Organization

The memory space of the System-on-Chip is configured in a "Von Neumann" architecture. This means that code and data are accessed within the same linear address space. All of the physically separated memory areas, including internal ROM and Flash, internal RAM, the processors Control and Status Registers (CSRs), the internal IO area, and external memory are mapped into one common address space.

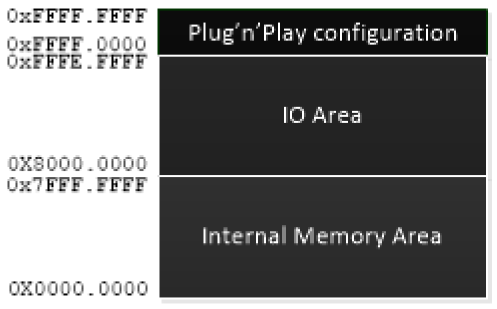

Figure 3-1. Address Space Overview.

The following VHDL configuration parameters defines address and data space sizes.

Parameter Name	Value	Description
CFG_SYSBUS_ADDR_BITS	32	Address Bus bit width
CFG_SYSBUS_DATA_BITS	64	Data Bus bit width

Table 3-1. General System Bus parameters.

Totally available 4 GB of memory space in the default implementation.

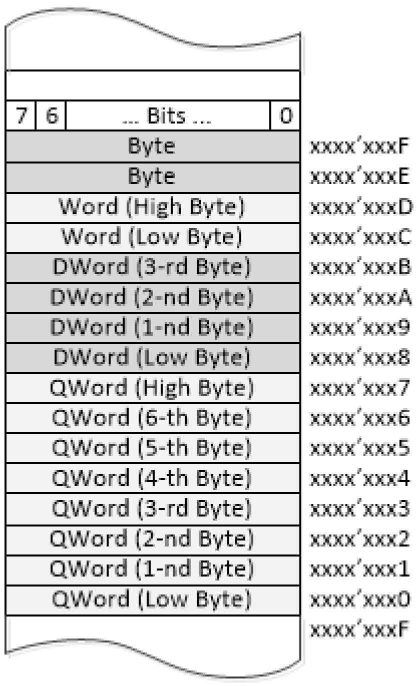

Figure 3-2. Storage of Words, Bytes and Bits in a Byte Organized Memory.

RISC-V architecture supposes memory is addressed as 8-bit bytes, with words being in `little-endian` order. That's similar to x86 computers. Words, up to the register size, can be accessed with the load and store instructions.

Accessed memory addresses need to be aligned to their word-width, otherwise "`unaligned store`" or "`unaligned load`" exception will be raised.

Note

Bus controller with the full unaligned memory access support was implemented in the earlier versions of River. It can be reused in future version by complicating procedures implementing AXI4 accesses.

Despite the default 64-bit data bus width (8-bytes) there's support of 4-byte aligned address transactions (including the burst operations). This was made to support the potential compatibility with the 32-bits data bus.

Example:

```
DMA read burst transaction request
       address = 0x00000004
       size = 0x2 (4 Bytes)
       len = 1 (2 transactions)

This operation will returns 64-bit data:
       read cycle 0: [data32[0x08], data32[0x04]]
       read cycle 1: [data32[0x0C], data32[0x08]]
```

CPU `River` always uses 8-byte aligned memory access with the appropriate strobes on a write operations.

3.1 Address Mapping

All the various memory areas and peripheral registers are mapped into one contiguous address space. All sections can be accessed in the same way. The memory map of the SoC contains some reserved areas, so future derivatives can be enhanced in an upward-compatible fashion.

Note

Table 3-2 shows the available memory areas in the default configuration. The actual available memory areas depend on the certain implementation.

Address Area	Start Loc.	End Loc.	Area Size	Notes
Boot ROM	00000000h	00007FFFh	32 Kbytes	Universal Bootloader image
OTP	00010000h	00011FFFh	8 Kbytes	One-Time Programmable secure code
User's ROM Image	00100000h	0013FFFFh	256 Kbytes	Internal ROM application
External Flash IC	00200000h	0023FFFFh	256 Kbytes	Micron Flash IC (25xx1024)
SRAM	10000000h	1007FFFFh	512 Kbytes	Internal code or data SRAM
GPIO Controller	80000000h	80000FFFh	4 Kbytes	User specific IOs
UART1	80001000h	80001FFFh	4 Kbytes	Default serial port
IRQ Controller	80002000h	80002FFFh	4 Kbytes	Interrupt Controller
GP Timers	80005000h	80005FFFh	4 Kbytes	Two general purpose timers with RTC
GNSS Sub-System	80008000h	8000FFFFh	32 Kbytes	Customizable Satellite Navigation Sub-↩ System (GPS/Glonass/Galileo)
Ethernet MAC	80040000h	8007FFFFh	256 Kbytes	100 Mb Ethernet MAC with the Debug support
DSU	80080000h	8009FFFFh	128 Kbytes	Debug Support Unit with the multiprocessors support
I/D Caches	800A0000h	800BFFFFh	128 Kbytes	Reserved
PNP	FFFFF000h	FFFFFFFFh	4 Kbytes	System status and configuration registers

Table 3-2. Default SoC Memory Map.

3.2 Boot ROM

Generic internal ROM module initialized with the default boot loader image (HEX-file) available in the folder *examples/boot/*.

Default bootrom image allows to implement the following functionality:

- Default exception handlers printing exception code into UART port.

- Implement initial system initialization, like: registers reset, stack initialization.
- Implement GPIO input reading to select boot options.
- Copy User image into SRAM as is and jump into SRAM entry point in User Mode.

VHDL IP generic parameters

Name	Value	Description
memtech	CFG_MEMTECH	**Memory technology**. This integer parameter defines memory banks instantiated in system: • 0 (inferred) = generic memory blocks (RTL Simulation) • not 0 = FPGA or ASIC technology specific memory banks
async_reset	FALSE	**Reset Type**. Internal registers reset type: • FALSE syncrhonous reset (FPGA) • TRUE asynchronous reset (ASIC)
xaddr	16#00000#	**Base address**. Base Address value defines bits [31:12] of the allocated memory space
xmask	16#FFFF8#	**Address Mask**. Address Mask is used by system controller to defines allocated memory size
sim_hexfile	CFG_SIM_BOOTROM_HEX	**Image file**. HEX-file used for ROM initialization

Table 3-3. Boot ROM generic parameters.

These generic parameters directly define the Boot ROM location in the system memory map 0x00000000. *xmask* value used to compute allocated memory size (in this default case 32 KB).

3.3 User ROM

Additional internal ROM memory bank initialized with the custom user's program image *fwimage.hex*. This image is coping as is into internal SRAM by Boot Loader and the Base Address of internal SRAM (0x10000000) used as entry point of the program.

This ROM allows to write simple user application without need to implement processor initialization code that executed in privileged Machine Mode (see *examples/helloworld/* or *examples/riscv-tests/*).

Additionally, no needs to program external flash IC or other external memory storage. That's significantly simplify system debugging on FPGA. But in common case this ROM may be removed from the system using only external Flash Memory storage.

VHDL IP generic parameters

Name	Value	Description
memtech	CFG_MEMTECH	**Memory technology**. This integer parameter defines memory banks instantiated in system: • 0 (inferred) = generic memory blocks (RTL Simulation) • not 0 = FPGA or ASIC technology specific memory banks
async_reset	FALSE	**Reset Type**. Internal registers reset type: • FALSE syncrhonous reset (FPGA) • TRUE asynchronous reset (ASIC)
xaddr	16#00100#	**Base address**. Base Address value defines bits [31:12] of the allocated memory space
xmask	16#FFFC0#	**Address Mask**. Address Mask is used by system controller to defines allocated memory size
sim_hexfile	CFG_SIM_FWIMAGE_HEX	**Image file**. HEX-file used for ROM initialization

Table 3-4. User's ROM generic parameters.

These generic parameters directly define the User's ROM location in the system memory map 0x00100000. *xmask* value used to compute allocated memory size (in this default case 256 KB).

3.4 External Flash Memory

External flash memory is connecting to the system-on-chip via SPI interface. SPI Controller with th AXI4 interface was specially developed to support IC `Micron M25AA1024` and other compatible ICs. Flash memory is directly mapped into memory map i.e. it visible to CPU and other master devices as a simple memory region of 256 KB. Read and Write accesses to this region automatically transformed into SPI flash commands by SPI controller. Access latency to flash depends of SPI command length and configured clock scaler (by default it forms 10 MHz SPI clock frequency).

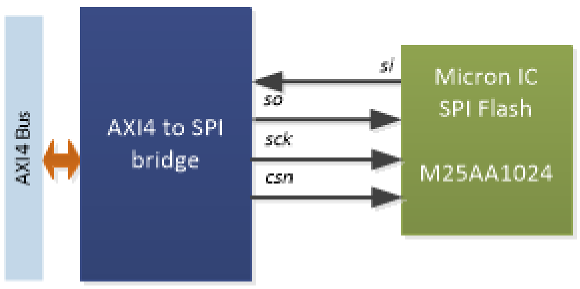

Figure 3-3. SPI Flash and SoC connection.

External Flash memory is fully accessible for reading as a simple memory bank without any limitations. System Bus controller is always wait response from Flash region while in a read transaction and stop other activities. But write transactions must be done via page buffer (256 Bytes) with the `Busy` bit polling in the `STATUS register`.

3.4.1 AXI4 SPI Bridge

This VHDL module was specially developed to interact with the following ICs:

- Microchip `25AA1024` and `25LC1024`
- Milandr `1636PP52Y`

The full list of the serial commands supported by AXI4 bridge and the external flash IC is shown below.

Name	Instruction Format	Description
READ	0000 0011	Read data from memory array beginning at selected address
WRITE	0000 0010	Write data to memory array (256 Bytes) beginning at selected address
WRENA	0000 0110	Set the write enable latch (enable write operations)
WRDI	0000 0100	Reset the write enable latch (disable write operations)
RDSR	0000 0101	Read STATUS register
WRSR	0000 0001	Write STATUS register
PE	0100 0010	Page Erase - erase one page in memory array
SE	1101 1000	Sector Erase - erase one sector in memory array
CE	1100 0111	Chip Erase - erase all sectors in memory array
RDID	1010 1011	Release from Deep Power-down mode and Read Electronic Signature
DPD	1011 1001	Deep Power-Down mode

Table 3-5. External Flash IC Instruction Set.

AXI4 read request into SPI controller address space automatically generates the signal sequence on SoC outputs (nCS, SDO, SCK) that directly connected to the external IC.

AXI4 transaction is in the wait state while no response from external IC so that CPU (or DMA, TAP) can't execute next request to the System Bus.

3.4.2 AXI4 SPI Registers

AXI4 SPI Bridge connected to the system bus as a slave device and available for reading and writing for any master device in a system.

Device Base Address define as 0x00200000. Full allocated memory range is 256 KB splitted on two equals parts. The first half is the direct representation of the external flash memory array. The second half (starting from address 0x00220000) contains the managing registers of the AXI4 bridge

VHDL IP generic parameters

AXI4 SPI Bridge configured with the following generic parameters by default:

Name	Value	Description
async_reset	FALSE	**Reset Type**. Internal registers reset type: • FALSE syncrhonous reset (FPGA) • TRUE asynchronous reset (ASIC)
xaddr	16#00200#	**Base address**. Base Address value defines bits [31:12] of the allocated memory space
xmask	16#FFFC0#	**Address Mask**. Address Mask is used by system controller to defines allocated memory size

Table 3-6. SPI Bridge generic parameters.

These generic parameters directly define the UART device memory location in the system memory map 0x00200000. Allocated memory size is 256 KB.

The full list of Registers relative Device Base Address offset is shown in the following table.

Device Registers list

Offset	Name	Reset Val.	Description
0x00000	FLASH_DATA	****:****h	128 KB of external flash IC directly accessible via this address range
0x20000	FLASH_SCALER	0000:0000h	System Bus clock divider threshold forming half period of the SPI frequency
0x20010	FLASH_STATUS	0000:0000h	Direct reading STATUS register from external IC
0x20018	FLASH_ID	0000:0029h	Direct reading of Flash ID
0x20020	FLASH_WRITE_ENA	0000:0000h	Write enable
0x20028	FLASH_WRITE_PAGE	0000:0000h	Write internal page buffer (256 B) into external memory array
0x20030	FLASH_WRITE_DIS	0000:0000h	Write disable
0x20038	FLASH_ERASE_PAGE	0000:0000h	Clear memory page
0x20040	FLASH_ERASE_SECTOR	0000:0000h	Clear memory sector
0x20048	FLASH_ERASE_CHIP	0000:0000h	Clear all memory sectors
0x20050	FLASH_POWER_DOWN	0000:0000h	Enable Deep Power-down mode

Table 3-7. AXI4 SPI Bridge Registers.

FLASH_DATA 128 KB (0x00000..0x1FFFF)

AXI4 read-transactions from this region is directly converting into signal sequence of the following format:

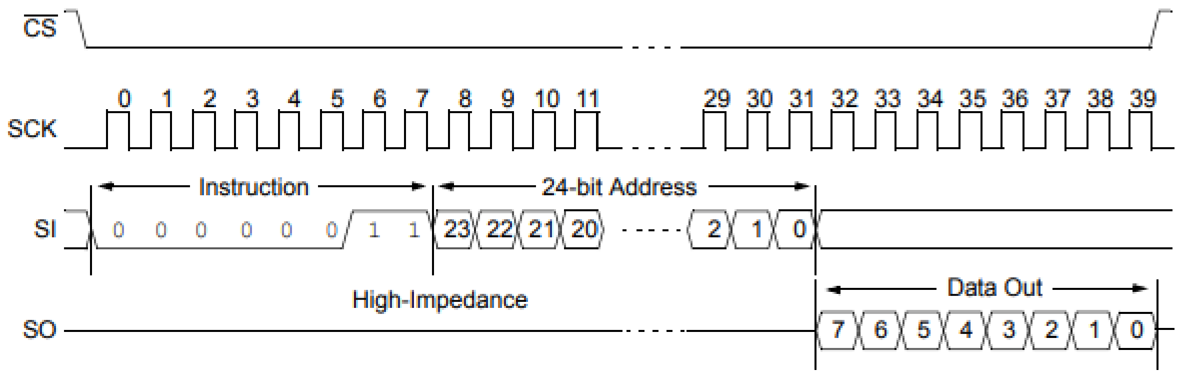

Figure 3-4. READ sequence format.

The bridge supports 4 and 8-bytes read operations.

Write access to this region **doesn't activate** any activities on SPI bus. All data is writing into the internal page buffer (256 Byte) using only address bits [7:2], address bits [31:8] are ignored. It is possible to write into page buffer using 4- and 8-bytes transactions. Page address is defined by register `FLASH_WRITE_PAGE`.

FLASH_SCALER Register (0x20000)

Field	Bits	Type	Description
scaler	[31:0]	RW	**Scaler Threshold**. This registers specifies the overflow threshold that is used to form `posedge` and `negedge` events forming SPI clock `SCLK`.

FLASH_STATUS Register (0x20010)

The `STATUS` register may be read at any time, even during a write cycle.

Field	Bits	Type	Description
rsrv	[31:4]	RZ	reserved
BP	[3:2]	RW	**Block Protection**. These bits control Write Protection for a specific segments (see IC description)
WEL	[1]	RO	**Write Enable Latch**: • '1' Enable write to flash • '0' Disable write to flash
WIP	[0]	RO	**Write In Progress**. External Flash IC is in writing state. • '1' Writing in progress • '0' Ready to accept the next command

Read request to this register forms the following signals sequence on SPI Bus:

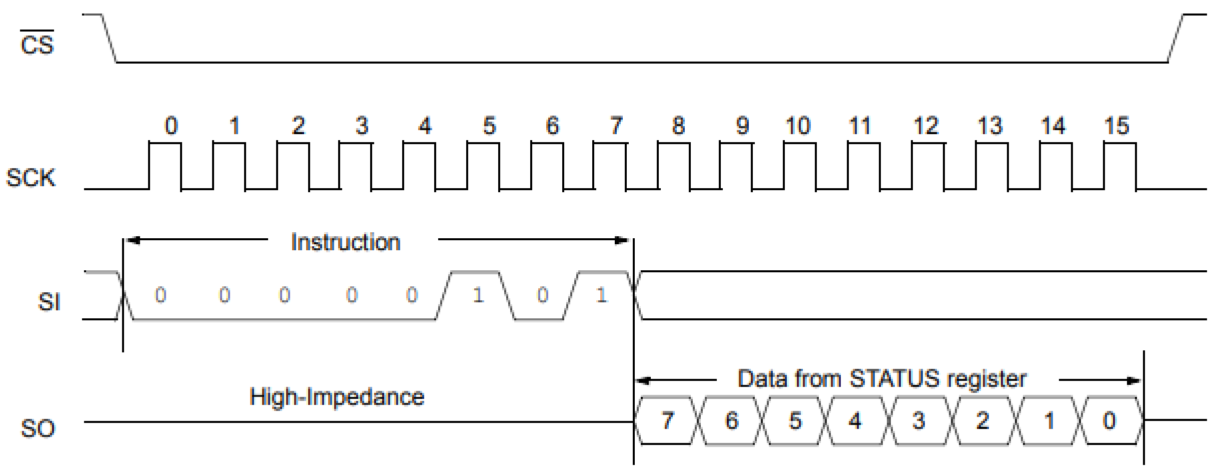

Figure 3-5. Read STATUS register SPI sequence.

Write request into this register forms the following signals sequence on SPI Bus:

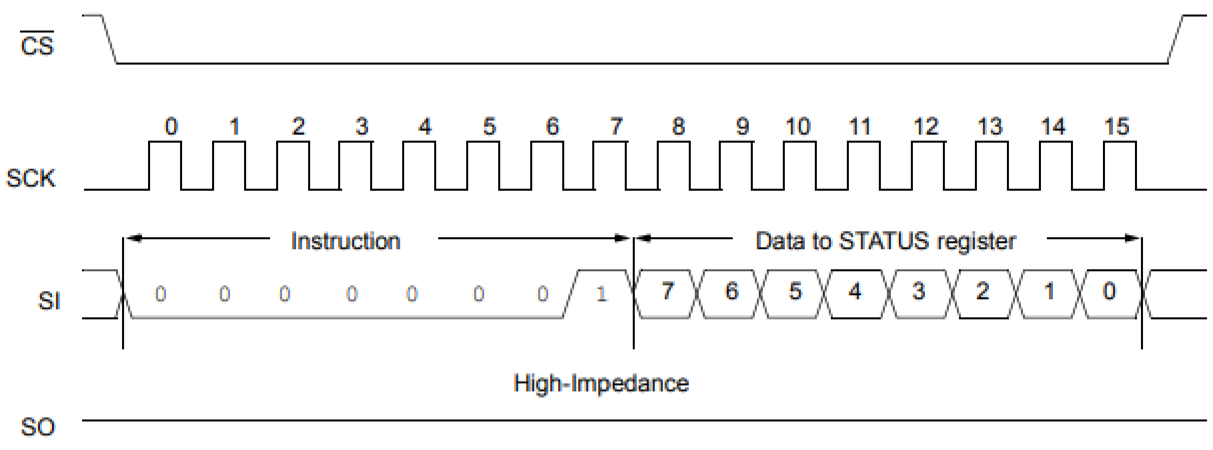

Figure 3-6. Write STATUS register SPI sequence.

FLASH_ID Register (0x20018)

Field	Bits	Type	Description
rsrv	[31:8]	RZ	reserved
ID	[7:0]	RO	**Manufacturer ID**. MicronChip ID 0x29

Read request to this register forms the following signals sequence on SPI Bus:

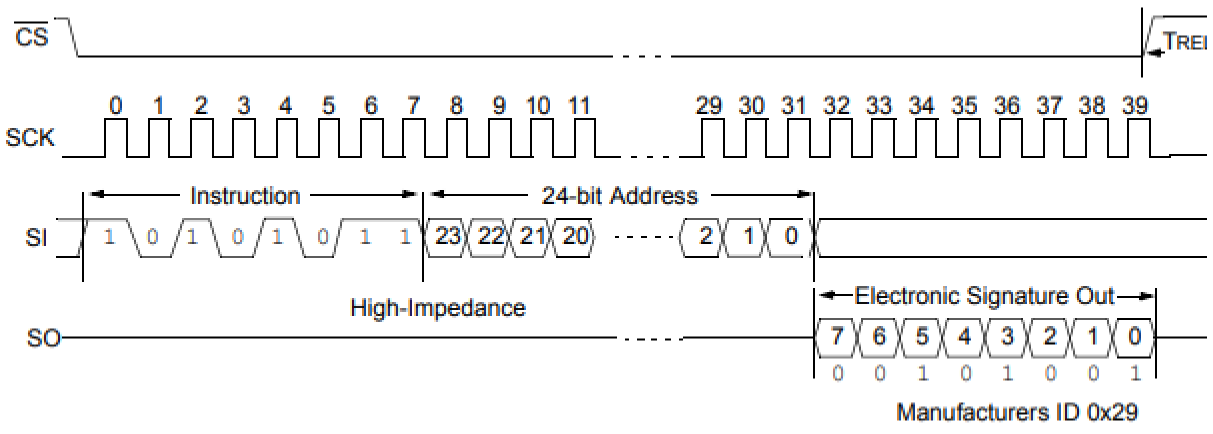

Figure 3-7. Read ID register SPI sequence.

Read request from this register will release the external flash device from Deep Power-down mode and outputs the electronic signature, and then returns the device to Standby mode.

FLASH_WRITE_ENA Register (0x20020)

Field	Bits	Type	Description
WE	[31:0]	WO	**Write Enable**. Any value written to this register generates SPI sequence enabling write access in External Flash

Write request into this register forms the following signals sequence on SPI Bus:

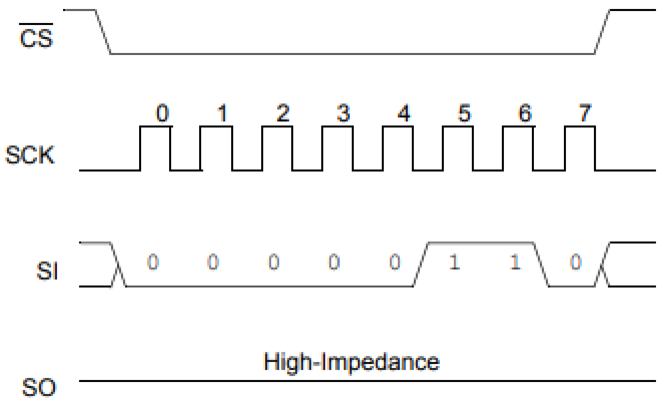

Figure 3-8. Write Enable SPI sequence.

FLASH_WRITE_PAGE Register (0x20028)

Field	Bits	Type	Description
rsrv	[31:17]	WO	reserved
PA	[16:8]	WO	**Page Address**. This value automatically inserted into SPI sequence for the page writing into External Flash
rsrv	[7:0]	WO	reserved

Write request into this register forms the following signals sequence on SPI Bus:

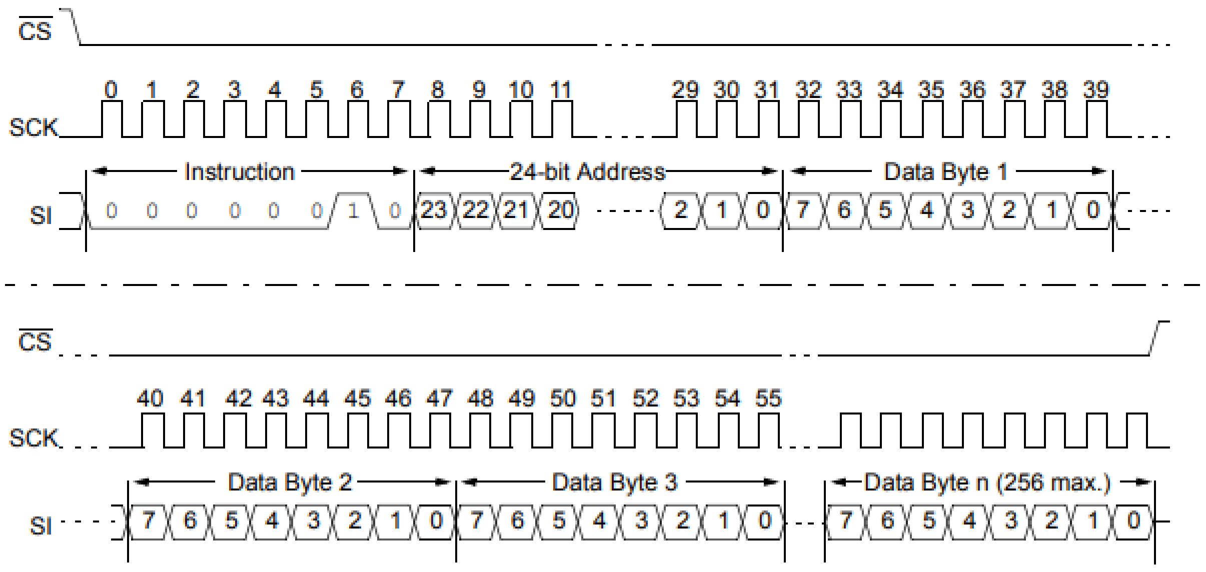

Figure 3-9. PAGE write sequence.

It is necessary polling bit WIP in the STATUS register and wait while it become Low. It is not allowed to call other SPI transactions while write process isn't finished otherwise data corruption may happen.

FLASH_WRITE_DIS Register (0x20030)

Field	Bits	Type	Description
WD	[16:8]	WO	**Write Disable**. Any value written to this register will generate SPI sequence disabling write access

Write request into this register forms the following signals sequence on SPI Bus:

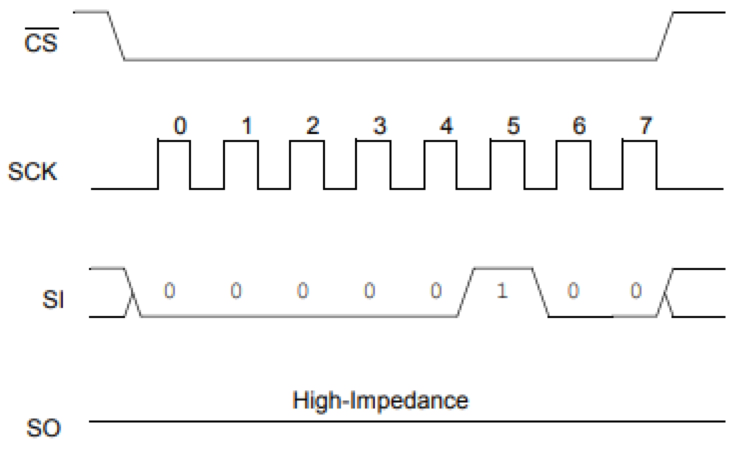

Figure 3-10. Write Disable sequence.

FLASH_ERASE_PAGE Register (0x20038)

Field	Bits	Type	Description
rsrv	[31:24]	WO	reserved
EPA	[23:0]	WO	**Erase Page Address**. This value automatically inserted into SPI sequence for the page erasing in External Flash

Write request into this register forms the following signals sequence on SPI Bus:

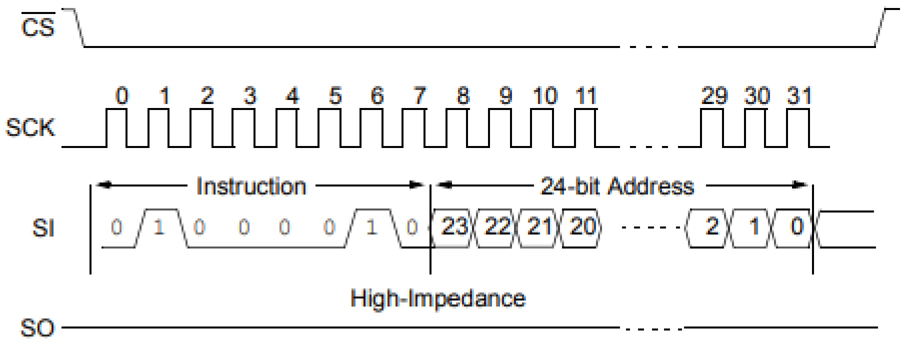

Figure 3-11. Page Erase sequence.

FLASH_ERASE_SECTOR Register (0x20040)

Field	Bits	Type	Description
rsrv	[31:24]	WO	reserved
ESA	[23:0]	WO	**Erase Sector Address**. This value automatically inserted into SPI sequence for the sector erasing in External Flash

Write request into this register forms the following signals sequence on SPI Bus:

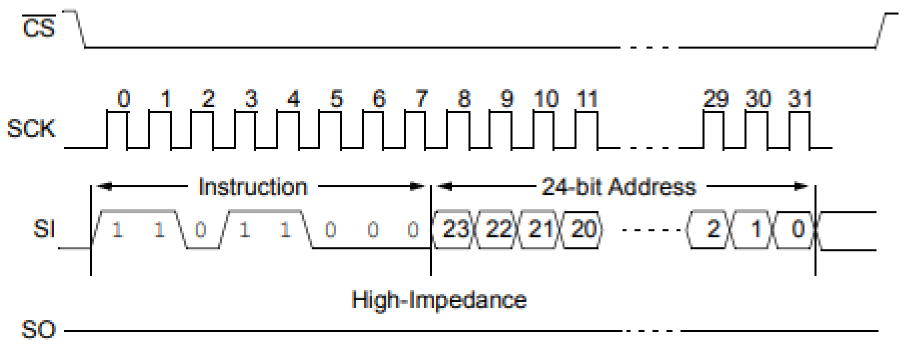

Figure 3-12. Sector Erase sequence.

FLASH_ERASE_CHIP Register (0x20048)

Field	Bits	Type	Description
CE	[31:0]	WO	**Chip Erase**. Any written value generates SPI sequence

Write request into this register forms the following signals sequence on SPI Bus:

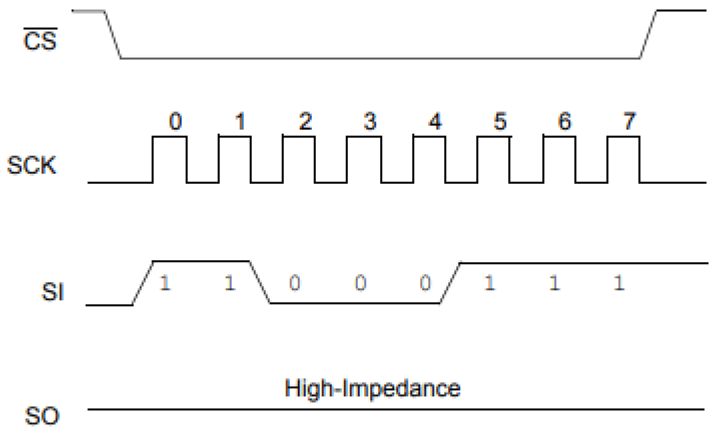

Figure 3-13. Chip Erase sequence.

FLASH_POWER_DOWN Register (0x20050)

Field	Bits	Type	Description
DPD	[31:0]	WO	**Deep Power-down mode**. Any written value generates SPI sequence

Write request into this register forms the following signals sequence on SPI Bus:

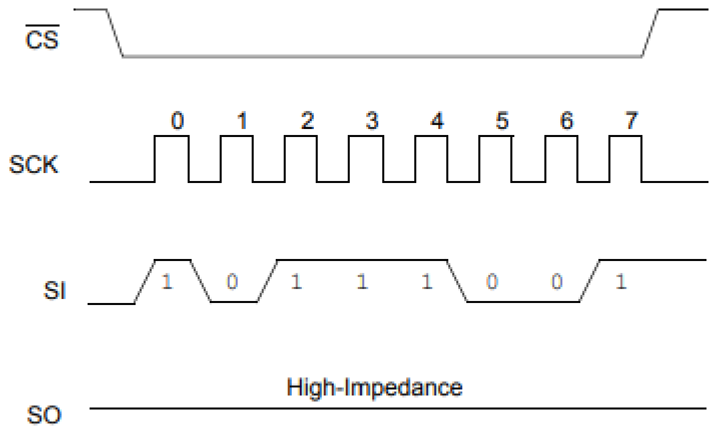

Figure 3-14. Enable Deep Power-down mode sequence.

3.4.3 C-code example

The following C-example demonstrates procedure of preparing and writing the internal page buffer (256 Bytes) into the external Flash memory array.

```
// Constant address definitions:
#define FLASH_DATA          0x200000
#define FLASH_SCALER        0x220000
#define FLASH_STATUS        0x220010
#define FLASH_WRITE_ENA     0x220020
#define FLASH_WRITE_PAGE    0x220028
#define FLASH_WRITE_DIS     0x220030
#define FLASH_ERASE_CHIP    0x220048


// Check status and wait
void flash_wait_ready() {
    volatile uint32_t *status = (volatile uint32_t *)FLASH_STATUS;
    while (status[0] & 0x1) {}
}

// Test example
void flash_write_test() {
    // Enable SCK clock 10 MHz on SPI interface
    *((uint32_t *)FLASH_SCALER) = 4;

    // Erase flash and check it readiness
    *((uint32_t *)FLASH_ERASE_CHIP) = 4;
    flash_wait_ready();

    // Enable Writing
    *((uint32_t *)FLASH_WRITE_ENA) = 1;

    // Prepare page with data (256 bytes)
    for (int i = 0; i < 64; i++) {
        ((uint32_t *)FLASH_DATA)[i] = 0x03020100 + i;
    }

    // Write Page 0
    *((uint32_t *)FLASH_WRITE_PAGE) = (0 << 8);
    flash_wait_ready();
    // Write Page 1
    *((uint32_t *)FLASH_WRITE_PAGE) = (1 << 8);
    flash_wait_ready();

    // Disable Writing
    *((uint32_t *)FLASH_WRITE_DIS) = 1;
}
```

SERGEYKHBR
RISCV_VHDL

Chapter 4

RISC-V Processor Core

Basic tasks of Central Processor Unit (CPU) are:

- Fetch and decode instructions
- Supply operands for the Arithmetic and Logic unit (ALU), the Integer Multiply and Divide block and for the Floating Pointing Unit (FPU) if enabled.
- Load and Store data operands of the instructions

River CPU implements five-stage pipeline with the possibility to `Halt` execution at any time without neccessity of pipeline flushing. `Halt` signal itself can be generated by User via Debug Support Unit (DSU) or by modules inside of pipeline. This architecture allows to proccess several instructions in parallel and provides functionality of the Debug Mode at any time.

Moreover, processor River provides **non-invasive debugging** access that allows to read/modify processors registers without halting the pipeline.

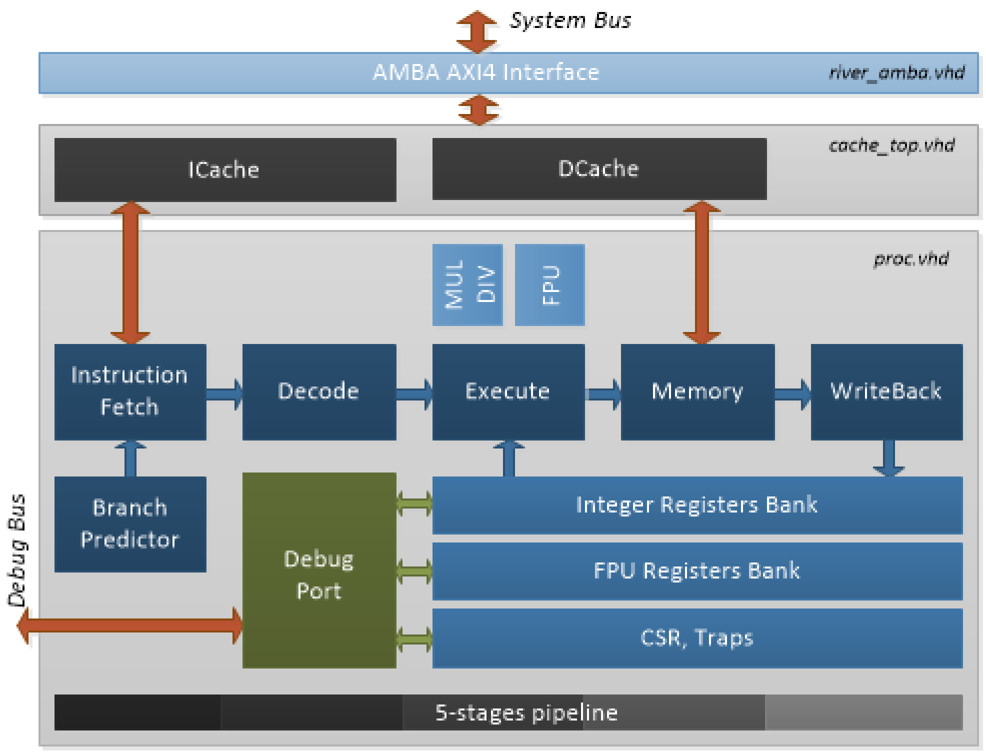

Figure 4-1. Functional diagram of the River CPU.

RISC-V Core `River` continues developing to support new instructions sets and improve performance. The following table contains brief information about already implemented features and features that are now in the development.

Function	River, the latest ver.	Description
RISC-V User Level spec	2.2	See `riscv.org.`
RISC-V Priviledged Level spec	1.10	See `riscv.org.`
GCC version	7.2.2 or newer	Used as a default toolchain in the open-source repository
GCC for Windows.	Yes	Cross-platform toolchain is available for a different OS
C-extension	Yes	Compressed (16-bit) instructions set support
D-extension	Yes	Double precision (64-bits) floating point instructions set
F-extension	Partially	Single precision (32-bits) floating point instructions set not used in the current DSP algoritms. Full support will be added later or by request.
Instruction Cache	Yes	Configurable IP. Default configurations: 4-ways, 16 KB, LRU
Data Cache	Yes	Configurable IP. Default configurations: 4-ways, 16 KB, LRU.
Memory Protection Unit	Yes	Configurable number of MPU entries. Programmable: region size, caching enable/disable, read-write-execute access rights.
Brach Predictor	Yes	Improve CPI performance index on 15 %
Stack Protection	Yes	Generate exceptions on stack overflow or underflow events. This extension implements custom Control and Status Registers (CSR) in CPU.
A-extension	Planned	Atomic instructions set support.
Watchdog	Planned	Generate reset signal if there's no CPU activity specified number of clock cycles.

Table 4-1. RIVER CPU features list.

4.1 Generic Configuration

River CPU implementation shared as IP block writen on VHDL. All source files should be placed in `riverlib` VHDL library before compilation. The following main generic parameters are avaiable for the user configuration.

Name	Value	Description
async_reset	FALSE	**Reset Type**. Internal registers reset type: • FALSE: syncrhonous reset (FPGA) • TRUE: asynchronous reset (ASIC)
CFG_VENDOR_ID	X"000000F1"	**Vendor ID**. Hardcoded constant available for reading in standard CSR register described in RISC-V ISA
CFG_IMPLEMENTATION_ID	X"20190512"	**Implementation ID**. Hardcoded constant available for reading in standard CSR register described in RISC-V I↩ SA.
CFG_HW_FPU_ENABLE	true	**Enable FPU**. Enable/disable FPU IP: • FALSE: Disable FPU • TRUE: Enable FPU
RISCV_ARCH	64	**Architecture Size**. General CPU registers bit width: • 32: RISC-V 32-bit not supproted • 64: RISC-V 64-bit • 128: RISC-V 128-bit not supported
BUS_ADDR_WIDTH	32	**System Bus Width**. Typicall value (other values acceptable but not tested): • 32: Use 32-bits address bus (default) • 64: Use 64-bits address bus
CFG_LOG2_DATA_BYTES	3	**Data Bus Bytes** parameter. Logarithmic value of System Data Bus in Bytes. • 3: Number of bytes 2∗∗3 = 8 (64 bits)
CFG_ILOG2_BYTES_PER_LINE	5	**Bytes per Lane**. Logarithmic value of Bytes per one I-cache line. This value and Data Bus width define length of one burst transaction when cached memory access is used. • 4: 32 Bytes per I-cache line. Burst length 4 with 64-bits data bus • 5: 64 Bytes per I-Cache line. Burst length 8 with 64-bits data bus

Name	Value	Description
CFG_ILOG2_LINES_PER_WAY	7	**Lines Per Way**. Logarithmic value defines I-cache size or number of lines instantiated for the each way: • 1: Analog of Cache disabled • 7: 128 lines per Way. (16 KB in default configuration) • 8: 256 lines Way. • 9: 512 lines Way.
CFG_ILOG2_NWAYS	2	**Number of Way**. Logarithmic value defines I-cache associativity: • 2: 4 Ways. (default) • 3: 8 Ways.
CFG_DLOG2_BYTES_PER_LINE	5	**Bytes per Lane**. Logarithmic value of Bytes per one D-cache line. This value and Data Bus width define length of one burst transaction when cached memory access is used. • 4: 32 Bytes per D-cache line. Burst length 4 with 64-bits data bus • 5: 64 Bytes per D-Cache line. Burst length 8 with 64-bits data bus
CFG_DLOG2_LINES_PER_WAY	7	**Lines Per Way**. Logarithmic value defines D-cache size or number of lines instantiated for the each way: • 1: Analog of Cache disabled • 7: 128 lines per Way. (16 KB in default configuration) • 8: 256 lines Way. • 9: 512 lines Way.
CFG_DLOG2_NWAYS	2	**Number of Way**. Logarithmic value defines D-cache associativity: • 2: 4 Ways. (default) • 3: 8 Ways.
CFG_MPU_TBL_WIDTH	2	**MPU table length**. Logarithmic value defines number of entries (memory regions) supported by Memory Protection Unit: • 2: 4 regions (default) • 3: 8 regions. • 4: 16 regions.

Table 4-2. RIVER generic parameters.

4.2 Memory Protection Unit

The MPU based *Protected Memory System Architecture* provides a considerably simpler memory protection scheme than the MMU based model providing the *Virtual Memory System Architecture*. The simplification applies to both the hardware and the software.

Main simplification is that the MPU does not use translation tables. Instead, Control and Status Registers (C↩ SRs) are used to fully define *protection regions*, eliminating the need for hardware to do translation table walks, and for software to set up and maintain the translation tables. This has the benefit of making memory checking fully determenistic. However, the level of control is now region based rater than page based, that is, the control is considerably less fine-grained.

A second simplification is that virtual-to-physical address translation is not supported. Processor works only with the physical addresses. The following feature are common to all MPU based designs:

- The memory is divided into regions. Dedicated Control registers are used to to define the region size, base address, and memory attributes, for example, cachability and access permissions of a region.
- Memory region control (read and write access) is permitted only from priviledged modes.
- If an address is defined in multiple regions, a fixed priority scheme (highest region number) is used to define the properties of the address being accessed.
- All addresses are physical addresses, address translation is not supported.

4.2.1 Memory access sequence

When River CPU generated a memory access, the I/D cache systems check availability of data inside of it. If the *miss* signal generated MPU compares the memory address with the programmed memory regions:

- If the memory region marked as uncached cache subsystem generates memory request directly into system bus interface without storing response into cache memory.
- If the memory region marked as cachable the cache subsystem generates memory burst request to read full cache line (on read operation) or read-modify-write to cache memory on write operation.
- If the memory region is enabled and access rights do not allow to complete operation then cache subsystem returns error response without access to system bus.
- Current implementation of MPU allows to read, write and execute data for all disabled regions.

4.2.2 Overlapping regions

The Protection Unit can be programmed with two or more overlapping regions. When overlapping regions are programmed, a fixed priority scheme is applied to determine the region whose attributes are applied to memory access.

For example, if the `CFG_MPU_TBL_WIDTH` = 3 enabling 8 programmable regions then region 7 take the highest priority and region 0 takes lowest priority. Lets suppose:

- Data region 1 is programmed to be 16 KB in size, starting from address 0x0 with the caching flag = 1.
- Data region 2 is programmed 64 Bytes in size, starting from address 0x3000 with the caching flag = 0.

Then this 64 B of memory can be used for the inter-processor communication without performance penalty on cache flushing operations because read and write access will be re-directed on system bus without L1 caching.

Address	Enabled	Cachable	Read	Write	Execute
0x00000000-0x7FFFFFFF	TRUE	TRUE	TRUE	TRUE	TRUE
0x80000000-0xFFFFFFFF	TRUE	FALSE	TRUE	TRUE	FALSE

Table 4-3. Default memory map.

4.2.3 Memory access control

The following RTL /i River Configuration parameter specify the addresses of exceptions handlers for MPU:

Parameter	Value	Description
CFG_NMI_INSTR_PAGE_FAULT_ADDR	00000068h	Execute instruction from the enabled region with the attribute executable = FALSE
CFG_NMI_LOAD_PAGE_FAULT_ADDR	00000070h	Load data from the enabled region with the attribute readable = FALSE
CFG_NMI_STORE_PAGE_FAULT_ADDR	00000078h	Store data into the enabled region with the attribute writable = FALSE

Access to a memory region is controlled by River CPU specific Control and Status Registers (CSRs).

CSR_MPU_ADDR Register (0x352)

Field	Bits	Type	Description
reserved	[63:BUS_ADDR_WIDTH]	WO	.

Field	Bits	Type	Description
ADDR	[BUS_ADDR_WIDTH-1:0]	WO	**MPU Address**. Base address of programming region. This value applied to the MPU region on write access into the register `CSR_MPU_CTRL`

CSR_MPU_MASK Register (0x353)

Field	Bits	Type	Description
reserved	[63:BUS_ADDR_WIDTH]	WO	.
MASK	[BUS_ADDR_WIDTH-1:0]	WO	**MPU Address Mask**. Address mask defining region size. This value applied to the MPU region on write access into the register `CSR_MPU_CTRL`.

CSR_MPU_CTRL Register (0x354)

Field	Bits	Type	Description
IDX	[15:8]	RW	**Region Index**. This field specify region index that will be modified on write access to this register on write access. This value contains total number of available regions `CFG_MPU_TBL_SIZE` on read access
ENA	[4]	WO	**Region Enable**. Enable region: • 0 = region disabled (not used) • 1 = region enabled
CACHABLE	[3]	WO	**Caching Enable**. Enable region for caching: • 0 = caching disabled • 1 = caching enabled
EXEC	[2]	WO	**Execute Enable**. Enable region for the instruction execution: • 0 = execution disabled • 1 = execution enabled
RD	[1]	WO	**Read Enable**. Enable region for reading: • 0 = read disabled • 1 = read enabled
WR	[0]	WO	**Write Enable**. Enable region for writing: • 0 = write disabled • 1 = write enabled

4.2.4 C-code example

The following C-example shows how to read total available memory regions provided by MPU:

```
// 0x354 = CSR_MPU_CTRL:
int get_region_total() {
    int ret;
    asm("csrr %0, 0x354" : "=r" (ret));
    return (ret >> 8) 0xFF;
}
```

Example of methods that controlling specific memory region via CSRs registers:

```
// CSR register CSR_MPU_CTRL
typedef union mpu_ctrl_type {
    struct bits_type {
        uint64_t WR     : 1;     // [0]
        uint64_t RD     : 1;     // [1]
        uint64_t EXEC   : 1;     // [2]
        uint64_t CACHABLE : 1;   // [3]
        uint64_t ENA     : 1;    // [4]
        uint64_t rsrv1   : 3;    // [7:5]
        uint64_t IDX     : 8;    // [15:8]
        uint64_t rsrv2   : 48;   //
    } bits;
    uint64_t value;
} mpu_ctrl_type;


// Enable MPU region with the specifc rights:
void mpu_enable_region(int idx,             // MPU entry index
                       uint64_t bar,        // Memory region base address
                       uint64_t KB,         // Memory region size in KB
                       int cached,          // 1=cached; 0=uncached
                       const char *rwx) {   // Region access rights string
    uint64_t mask = (~0ull) << 10;
    const char *p = rwx;
    mpu_ctrl_type ctrl;

    // Write base address
    asm("csrw 0x352, %0" : :"r"(bar));

    // Computes region mask using KB value
    KB >>= 1;
    while (KB) {
        mask <<= 1;
        KB >>= 1;
    }
    asm("csrw 0x353, %0" : :"r"(mask));

    // Write Control word
    ctrl.value = 0;
    ctrl.bits.IDX = idx;
    ctrl.bits.ENA = 1;
    ctrl.bits.CACHABLE = cached;
    while (*p) {
        if (*p == 'r') {
            ctrl.bits.RD = 1;
        }
        if (*p == 'w') {
            ctrl.bits.WR = 1;
        }
        if (*p == 'x') {
            ctrl.bits.EXEC = 1;
        }
        p++;
    }
    asm("csrw 0x354, %0" : :"r"(ctrl.value));
}

// Disable MPU region
void mpu_disable_region(int idx) {
    mpu_ctrl_type ctrl;
    ctrl.value = 0;
    ctrl.bits.IDX = idx;
    asm("csrw 0x354, %0" : :"r"(ctrl.value));
}
```

The following C-example demonstrates procedures of interaction with the MPU module that allows to configure, enable and disable certain memory regions access rights using previously defined methods.

```
// Test function
void test_mpu() {
    int reg_total = get_region_total();

    // Make 0x20000..0x20FFF region:
    //       uncached
    //       read, write, not executable
    mpu_enable_region(reg_total-1, 0x20000, 4, 0, "rw")
    ..
    mpu_disable_region(reg_total-1);
}
```

4.3 Program Flow Control

The Instruction Fetch Unit prefetches and preprocesses instructions to provide a continuous instruction flow. Fetch module continuously requests 32-bits instruction words (1-standard or 2-compressed instructions) on each clock cycle.

If Instruction Cache module cannot return data on the next clock cycle Fetch module generates `hold pipeline` signal. `Hold pipeline` signal released when `ICache` module returns valid data. Typically it will take 6 clock cycles to request and save 4-clock burst transaction.

Preprocessing of branch instructions enables the instruction flow to be predicted. While the CPU is in the process of decoding and executing an instruction the fetcher starts to request a new instruction at a predicted target address. Even for a non-sequential instruction execution, Branch Predictor and Fetcher can generally provide a continuous instruction flow.

During the prefetch stage, the The Branch Predictor logic analyzes up to three prefetched instructions stored in history buffer. If a branch is detected, then the fetcher starts to request the next instructions from the instruction cache accordingly to the predicted rules.

4.3.1 Branch Prediction Rules

The Branch Predictor preprocesses instructions and classifies detected branches. Depending on the branch class, the Branch Predictor predicts the program flow using the following rules:

Branch Instruction Classes	Instructions	Prediction Rule
Unconditional Jump	JAL, C_J	The branch is always taken
Return instructions	RET	The branch is always taken
Conditional Branches	BEQ, BNE, BLT, BGE, BLTU, BG↩ EU	Offset direction: • Unconditional or backward↩ : branch 'taken' • Conditional forward: branch 'not taken'
Relative Jump	JALR	The branch is always not taken
Standard 4-bytes instruction	*	npc = pc + 4
Compressed 2-bytes instruction	*	npc = pc + 2

Table 4-4. Branch Classes and Prediction Rules.

4.4 General Purpose Registers

CPU RIVER provides one bank of 32 regsiters x0, x1, ... x31, called General Purpose Registers (GPR), which can be accessed in one CPU cycle. Each register is a 64-bits width and can be used as a 32-bits operand in the RV32 (W) instructions. The GPRs are the working registers of the arithmetic and logic units and also serve as address pointers for indirect addressing modes.

The register bank are access via 3-port register file providing the high access speed required for the CPU's performance. At any time there's available 2 registers for reading and 1 for writing. Additional Debug Port provides debuging registers access via `DSU`.

Register	ABI Name	Description
x0	r0	Hard-wired zero
x1	ra	Return address
x2	sp	Stack pointer
x3	gp	Global pointer
x4	tp	Thread pointer
x5	t0	Temporary/alternate link register
x6	t1	Temporary register 1
x7	t2	Temporary register 2
x8	s0/fp	Saved register 0 / frame pointer
x9	s1	Saved register 1
x10	a0	Function argument 0 / return value 0
x11	a1	Function argument 1 / return value 1
x12	a2	Function argument 2
x13	a3	Function argument 3
x14	a4	Function argument 4
x15	a5	Function argument 5
x16	a6	Function argument 6
x17	a7	Function argument 7
x18	s2	Saved register 2
x19	s3	Saved register 3
x20	s4	Saved register 4
x21	s5	Saved register 5
x22	s6	Saved register 6
x23	s7	Saved register 7
x24	s8	Saved register 8
x25	s9	Saved register 9
x26	s10	Saved register 10
x27	s11	Saved register 11
x28	t3	Temporary register 3
x29	t4	Temporary register 4
x30	t5	Temporary register 5
x31	t6	Temporary register 6

Table 4-5. Integer Registers bank.

Register `x0` (zero), `x1` (return address) usage cannot be changed by software and hardcoded on hardware level. But usage of all others registers is specified for the compilers to provide Application Binary Compatibility (ABI).

Writing into register `x0` doesn't lead to any modification and can be interpreted as an empty operation.

4.5 FPU Registers

River CPU implements additional registers bank with the 32 64-bit registers. Configuration parameter `CFG_HW_`↩ `FPU_ENABLE` allows to disable instantiation of the bank if the FPU was disabled.

Register	ABI Name	Description
f0	ft0	FP temporary register 0
f1	ft1	FP temporary register 1
f2	ft2	FP temporary register 2
f3	ft3	FP temporary register 3
f4	ft4	FP temporary register 4
f5	ft5	FP temporary register 5
f6	ft6	FP temporary register 6
f7	ft7	FP temporary register 27
f8	fs0	FP saved register 0
f9	fs1	FP saved register 1
f10	fa0	FP argument 0 / return value 0
f11	fa1	FP argument 1 / return value 1
f12	fa2	FP argument 2
f13	fa3	FP argument 3
f14	fa4	FP argument 4
f15	fa5	FP argument 5
f16	fa6	FP argument 6
f17	fa7	FP argument 7
f18	fs2	FP saved register 2
f19	fs3	FP saved register 3
f20	fs4	FP saved register 4
f21	fs5	FP saved register 5
f22	fs6	FP saved register 6
f23	fs7	FP saved register 7
f24	fs8	FP saved register 8
f25	fs9	FP saved register 9
f26	fs10	FP saved register 10
f27	fs11	FP saved register 11
f28	ft3	FP temporary register 3
f29	ft4	FP temporary register 4
f30	ft5	FP temporary register 5
f31	ft6	FP temporary register 6

Table 4-6. Floating-Point Unit Registers bank.

Note

The future version of River is planned with the additional `generic` parameter enabling and disabing FPU that will allow to setup configuration for each core separetely.

4.6 Control and Status Registers

Special privileged instructions provides access to the internal CPU Control and Status Registers (CSR) set. In a common case, accordingly with the RISC-V specification, read/write/modify access rights depends of the current processor mode. If processor tries to execute privileged access to CSR while in lower than neccessary privileged level, then special trap is generated with the increased rights.

This functionality is used for the hypervisor and others virtual platforms development and either to provide operational system security when running user's applications.

Processor River of the latest revision supports and correctly handles states in two modes: `Machine mode` (maximal privileged level) and `User mode`. But `User mode` have the same set of CSR registers and the same access rights as the `Machine mode`. This is actually RISC-V specification violation that would be fixed in the future release, but for the current moment it raises only security issues of working with operational system but it doesn't affect the possibility to run multitasks environment.

Addr.	Name	Reset Value	Description
0x001	FFLAGS	0x0	FPU accrued Exceptions fields from FCSR
0x002	FRM	0x8	FPU dynamic Rounding Mode fields from FCSR
0x003	FCSR	0x8	FPU Control and Status register (FRM + FFLA↩ GS)
0xf10	MISA	80000000:00101108h	List of supported instruction sets
0xf11	MVENDORID	00000000:000000F1h	Vendor ID: `CFG_VENDOR_ID` parameter
0xf12	MARCHID	0x0	Architecture ID
0xf13	MIMPLEMENTATIONID	00000000:20190521h	Implementation ID: `CFG_IMPLEMENTATION↩ _ID` parameter
0xf14	MHARTID	0x0	Core ID in multi-processors system
0x041	UEPC	0x0	User program counter
0x300	MSTATUS	00000000:00001800h	Machine mode status register
0x302	MEDELEG	0x0	Machine exception delegation
0x303	MIDELEG	0x0	Machine interrupt delegation
0x304	MIE	0x0	Machine interrupt enable bit
0x305	MTVEC	0x0	Machine Interrupts Table pointer
0x340	MSCRATCH	0x0	Machine scratch register
0x341	MEPC	0x0	Machine program counter
0x342	MCAUSE	0x0	Machine trap cause
0x343	MBADADDR	0x0	Machine bad address
0x344	MIP	0x0	Machine Interrupt pending

Addr.	Name	Reset Value	Description
0x350	MSTACKOVR[1]	0x0	Machine Stack Overflow limit
0x351	MSTACKUND[1]	0x0	Machine Stack Underflow limit
0x352	MPU_ADDR[1]	0x0	MPU region Base address
0x353	MPU_MASK[1]	0x0	MPU region size
0x354	MPU_CTRL[1]	0x0	MPU Control Register

Table 4-7. List of implemented CSR registers.

[1] - Non-standard CSR register (River CPU only).

FFLAGS Register (0x001)

Field	Bits	Type	Description
reserved	[63:5]	RO	.
NV	[4]	RO	**Invalid Operation**. Last FPU operation status flag
DZ	[3]	RO	**Divide by Zero**. Last FPU operation status flag
OF	[2]	RO	**Overflow**. Last FPU operation status flag
UF	[1]	RO	**Underflow**. Last FPU operation status flag
NX	[0]	RO	**Inexact**. Last FPU operation status flag

FRM Register (0x002)

Field	Bits	Type	Description
reserved	[63:8]	RO	.
FRM	[7:5]	RO	**Rounding Mode**: • 000: RNE - Round to Nearest, ties to Even • 001: RTZ - Round towards Zero • 010: RDN - Round Down • 011: RUP - Round Up • 100: RMM - Round to Nearest, ties to Max Magnitude • 101: Invalid. Reserved for future use. • 110: Invalid. Reserved for future use. • 111: Dynamic rounding mode
reserved	[4:0]	RO	.

FCSR Register (0x003)

Field	Bits	Type	Description
reserved	[63:8]	RO	.
FRM	[7:5]	RO	**Rounding Mode**: • 000: RNE - Round to Nearest, ties to Even • 001: RTZ - Round towards Zero • 010: RDN - Round Down • 011: RUP - Round Up • 100: RMM - Round to Nearest, ties to Max Magnitude • 101: Invalid. Reserved for future use. • 110: Invalid. Reserved for future use. • 111: Dynamic rounding mode
NV	[4]	RO	**Invalid Operation**. Last FPU operation status flag
DZ	[3]	RO	**Divide by Zero**. Last FPU operation status flag
OF	[2]	RO	**Overflow**. Last FPU operation status flag
UF	[1]	RO	**Underflow**. Last FPU operation status flag
NX	[0]	RO	**Inexact**. Last FPU operation status flag

MISA Register (0xf10)

Field	Bits	Type	Description
MXL	[63:62]	RO	**Machine XLEN**: • 1 - 32-bits • 2 - 64-bits (River CPU) • 3 - 128-bits
reserved	[61:26]	RZ	.
Z	[25]	RZ	Reserved
Y	[24]	RZ	Reserved
X	[23]	RO	Non-standard extension present
W	[22]	RZ	Reserved
V	[21]	RO	Tentatively reserved for Vector extension
U	[20]	RO	User mode implemented
T	[19]	RO	Tentatively reserved for Transactional memory extension
S	[18]	RO	Supervisor mode implemented
R	[17]	RZ	Reserved
Q	[16]	RO	Quad-precision floating-point extension
P	[15]	RO	Tentatively reserved for Packed-SIMD extension
O	[14]	RZ	Reserved
N	[13]	RO	User-level interrupts supported
M	[12]	RO	Integer Multiply/Divide extension
L	[11]	RO	Tentatively reserved for Decimal Floating-Point extension

Field	Bits	Type	Description
K	[10]	RZ	Reserved
J	[9]	RO	Tentatively reserved for Dynamically Translated Languages extension
I	[8]	RO	RV32I/64I/128I base ISA
H	[7]	RZ	Reserved
G	[6]	RO	Additional standard extensions present
F	[5]	RO	Single-precision floating-point extension
E	[4]	RO	RV32E base ISA
D	[3]	RO	Double-precision floating-point extension
C	[2]	RO	Compressed extension
B	[1]	RO	Tentatively reserved for Bit operations extension
A	[0]	RO	Atomic extension

MVENDORID Register (0xf11)

This register should provide the JEDEC manufacturer ID of the provider of the core. River CPU doesn't have a registered JEDEC identification.

Field	Bits	Type	Description
BANK	[63:7]	RO	The number of one-byte continuation JEDEC codes: `CFG_VENDOR_ID`[63:7]
OFFSET	[6:0]	RO	JEDEC final byte: `CFG_VENDOR_ID`[6:0]

MARCHID Register (0xf12)

Field	Bits	Type	Description
AID	[63:0]	RZ	Hard-wired to zero.

MIMPLEMENTATIONID Register (0xf13)

Field	Bits	Type	Description
reserved	[63:32]	RZ	.
IMPL	[31:0]	RO	**Implementation**. This value can be changed via configuration parameter `CFG←_IMPLEMENTATION_ID`

MHARTID Register (0xf14)

Field	Bits	Type	Description
HART	[63:0]	RO	**Hart ID**. This value is defined on SoC top-level via River CPU generic parameter `hartid`. This value should be unique for each Core.

UEPC Register (0x041)

Field	Bits	Type	Description
UEPC	[63:0]	RZ	**User Exception program counter**. Always read zero.

MSTATUS Register (0x300)

Field	Bits	Type	Description
SD	[63]	RO	**Extensions Context Dirty**. Not implemented. Read zero.
reserved	[62:36]	RZ	.
SXL	[35:34]	RO	**Supervisor mode support**: • 0 - not supported (River CPU default) • 1 - 32-bits • 2 - 64-bits • 3 - 128-bits
UXL	[33:32]	RO	**User mode support**: • 0 - not supported • 1 - 32-bits • 2 - 64-bits (River CPU default) • 3 - 128-bits
reserved	[31:23]	RZ	.
TSR	[22]	RO	**Trap SRET**. Not implemented
TW	[21]	RO	**Timeout Wait**. Not implemented
TVM	[20]	RO	**Trap Virtual Memory**. Not implemented
MXR	[19]	RO	**Make executable readable**. Not implemented
SUM	[18]	RO	**Permit Supervisor User Memory**. Not implemented
MPRV	[17]	RO	**Modify Privilege**. Not implemented
XS	[16:15]	RO	Not implemented
FS	[14:13]	RO	Extension context status: • always 0 - when FPU disabled (`CFG_HW_FPU_ENABLE` = false) • always 1 - when FPU enabled (`CFG_HW_FPU_ENABLE` = true)
MPP	[12:11]	RW	Privilege level prior the trap
reserved	[10:9]	RZ	.
SPP	[8]	RO	Not implemented
MPIE	[7]	RW	Interrupt Enable bit prior the trap
reserved	[6]	RZ	.
SPIE	[5]	RO	Not implemented
UPIE	[4]	RO	Not implemented
MIE	[3]	RW	Interrupt Enable for Machine mode: • 0 - Interrupts disabled • 1 - Interrupts enabled
reserved	[2]	RZ	.
SIE	[1]	RO	Not implemented

Field	Bits	Type	Description
UIE	[0]	RW	Interrupt Enable for User mode: • 0 - Interrupts disabled • 1 - Interrupts enabled

MEDELEG Register (0x302)

Field	Bits	Type	Description
SE	[63:0]	RZ	**Synchronous Exceptions**. Not implemented.

MIDELEG Register (0x303)

Field	Bits	Type	Description
IRQS	[63:0]	RZ	**Interrupts**. Not implemented

MIE Register (0x304)

Field	Bits	Type	Description
reserved	[63:12]	RZ	.
MEIE	[11]	RW	**Machine External Interrupt Enable**. This bit is directly controlled via `MSTATUS` register.
reserved	[10]	RZ	.
SEIE	[9]	RW	**Supervisor External Interrupt Enable**. This bit is directly controlled via `MST↩ ATUS` register.
UEIE	[8]	RW	**User External Interrupt Enable**. This bit is directly controlled via `MSTATUS` register.
MTIE	[7]	RW	**Machine Timer IE**. River CPU doesn't implement wallclock timer only external General Purposes Timers
reserved	[6]	RZ	.
STIE	[5]	RW	**Supervisor Timer IE**. Wallclock timer not implemented
UTIE	[4]	RW	**User Timer IE**. Wallclock timer not implemented
MSIE	[3]	RW	**Machine Software Interrupt Enable**. Not implemented
reserved	[2]	RZ	.
SSIE	[1]	RW	**Supervisor Software Interrupt Enable**. Not implemented
USIE	[0]	RW	**Machine Software Interrupt Enable**. Not implemented

MTVEC Register (0x305)

Field	Bits	Type	Description
BASE	[63:1]	RW	**Trap-Vector Base Address**. All Interrupts cause the `pc` to be set to the address in this field

Field	Bits	Type	Description
MODE	[0]	RW	**Trap Mode**: • 0 = Direct mode. All traps cause the `pc` to be set to the address in BASE field. • 1 = Vectored. Not implemented.

MSCRATCH Register (0x340)

Field	Bits	Type	Description
MSCRATCH	[63:0]	RW	**Machine Scratch**. Typically, it is used to hold a pointer to a machine-mode hart-local context space and swapped with a user register upon entry to an M-mode trap handler.

MEPC Register (0x341)

Field	Bits	Type	Description
MEPC	[63:0]	RW	**Machine Exception Program Counter**. Register specifies the jump address after execution of `mret` instruction.

MCAUSE Register (0x342)

Field	Bits	Type	Description
IRQ	[62:0]	R0	**Interrupt flag**. • '0' - Exception trap • '1' - Interrupt trap
CODE	[62:0]	R0	**Exception Code**. See tables 4-6 and 4-7 with the information about trap codes

Exception Codes

Val	Name	Description
0	InstrMisalign	Instruction address misaligned
1	InstrFault	Instruction access fault
2	InstrIllegal	Illegal instruction
3	Breakpoint	Software berakpoint (instruction EBREAK)
4	LoadMisalign	Load address misaligned
5	LoadFault	Load access fault
6	StoreMisalign	Store address misaligned
7	StoreFault	Store address fault
8	CallFromUMode	Environment call from U-mode (instruction ECALL)
9	CallFromSMode	Not used
10	CallFromHMode	Not used
11	CallFromMMode	Environment call from M-mode (instruction ECALL)

Val	Name	Description
12	InstrPageFault	Not used
13	LoadPageFault	Not used
14	reserved	Not used
15	StorePageFault	Not used
16	StackOverflow	Stack pointer (sp) is below than overflow limit
17	StackUnderflow	Stack pointer (sp) is higher than underflow limit

Table 4-8. CODE value in a case of exception.

Interrupt Codes

Val	Name	Description
0	USoftware	Not used
1	SSoftware	Not used
2	HSoftware	Not used
3	MSoftware	Not used
4	UTimer	Not used
5	STimer	Not used
6	HTimer	Not used
7	MTimer	Not used
8	UExternal	Not used
9	SExternal	Not used
10	HExternal	Not used
11	MExternal	Interrupt request from an external IRQ controller

Table 4-9. CODE value in a case of interrupt.

MBADADDR Register (0x343)

Field	Bits	Type	Description
MBADADDR	[63:0]	RO	**Bad Address**. Registers captures bad program counter on an instruction fault or memory access address on an invalid data load/store operations.

MIP Register (0x344)

Field	Bits	Type	Description
reserved	[63:12]	RZ	.
MEIP	[11]	RW	**Machine External Interrupt Pending**. Not implemented.
reserved	[10]	RZ	.
SEIP	[9]	RW	**Supervisor External Interrupt Pending**. Not implemented.

Field	Bits	Type	Description
UEIP	[8]	RW	**User External Interrupt Pending**. Not implemented.
MTIP	[7]	RW	**Machine Timer IP**. River CPU doesn't implement wallclock timer only external General Purposes Timers
reserved	[6]	RZ	.
STIP	[5]	RW	**Supervisor Timer IP**. Wallclock timer not implemented
UTIP	[4]	RW	**User Timer IP**. Wallclock timer not implemented
MSIP	[3]	RW	**Machine Software Interrupt Pending**. Not implemented
reserved	[2]	RZ	.
SSIP	[1]	RW	**Supervisor Software Interrupt Pending**. Not implemented
USIP	[0]	RW	**Machine Software Interrupt Pending**. Not implemented

4.7 Stack Protection

River CPU implements two special non-standard CSR registers `MSTACKOVR` and `MSTACKUND`. These dedicated registers allows to setup the boundaries at which the exception event is generated if the 'sp' comes out of these limits.

MSTACKOVR Register (0x350)

Field	Bits	Type	Description
SOVR	[63:0]	RW	**Stack Overflow border**. Value automatically cleared just after exception generated: • zero - Stack Overflow exception disabled • non-zero - Raise Stack Overflow exception if register value `sp` is less than this value.

MSTACKUND Register (0x351)

Field	Bits	Type	Description
SUND	[63:0]	RW	**Stack Underflow border**. Value automatically cleared just after exception generated: • zero - Stack Overflow exception disabled • non-zero - Raise Stack Underflow exception if register value `sp` is greater than this value.

MPU_ADDR Register (0x352)

Field	Bits	Type	Description
reserved	[63:BUS_ADDR_WIDTH]	WO	.

Field	Bits	Type	Description
ADDR	[BUS_ADDR_WIDTH-1:0]	WO	**MPU Address**. Base address of programming region. This value applied to the MPU region on write access into the register `MPU_CTRL`

MPU_MASK Register (0x353)

Field	Bits	Type	Description
reserved	[63:BUS_ADDR_WIDTH]	WO	.
MASK	[BUS_ADDR_WIDTH-1:0]	WO	**MPU Address Mask**. Address mask defining region size. This value applied to the MPU region on write access into the register `MPU_CTRL`.

MPU_CTRL Register (0x354)

Field	Bits	Type	Description
IDX	[15:8]	RW	**Region Index**. This field specify region index that will be modified on write access to this register on write access. This value contains total number of available regions `CFG_MPU_TBL_SIZE` on read access
ENA	[4]	WO	**Region Enable**. Enable region: • 0 = region disabled (not used) • 1 = region enabled
CACHABLE	[3]	WO	**Caching Enable**. Enable region for caching: • 0 = caching disabled • 1 = caching enabled
EXEC	[2]	WO	**Execute Enable**. Enable region for the instruction execution: • 0 = execution disabled • 1 = execution enabled
RD	[1]	WO	**Read Enable**. Enable region for reading: • 0 = read disabled • 1 = read enabled
WR	[0]	WO	**Write Enable**. Enable region for writing: • 0 = write disabled • 1 = write enabled

4.8 Exceptions and Interrupts

CPU River supports two types of traps:

- Unmaskable Exceptions (Bit `MCAUSE`[64] = '0')
- External Interrupts (Bit `MCAUSE`[64] = '1')

In a case of occured valid trap CPU executes the following steps:

- Store Trap type and Trap index into `MCAUSE` register
- Store next program counter value (npc) into the register `MEPC` or `UEPC` depending of the current processor mode.
- Fetch and execute instruction stored in the address specified by register `MTVEC` if the occured trap is an Interrupt. If the current trap is an Exception then CPU jumps to address accordingly with the configuration **table 4-6**.
- CPU switches into Machine Mode and properly shows its status in the register `MSTATUS`.
- All interrupts are masked (`MIE`).

Parameter	Address	Description
CFG_NMI_RESET_VECTOR	0x00000000	CPU Reset vector
CFG_NMI_INSTR_UNALIGNED_ADDR	0x00000008	Instruction address misaligned handler
CFG_NMI_INSTR_FAULT_ADDR	0x00000010	Instruction address fault handler
CFG_NMI_INSTR_ILLEGAL_ADDR	0x00000018	Illegal or unsupported instruction handler
CFG_NMI_BREAKPOINT_ADDR	0x00000020	Software breakpoint (EBREAK) handler
CFG_NMI_LOAD_UNALIGNED_ADDR	0x00000028	Load address misaligned handler
CFG_NMI_LOAD_FAULT_ADDR	0x00000030	Load access fault handler
CFG_NMI_STORE_UNALIGNED_ADDR	0x00000038	Store address misaligned handler
CFG_NMI_STORE_FAULT_ADDR	0x00000040	Store access fault handler
CFG_NMI_CALL_FROM_UMODE_ADDR	0x00000048	Environment call from U-mode handler
CFG_NMI_CALL_FROM_SMODE_ADDR	0x00000050	Environment call from S-mode handler
CFG_NMI_CALL_FROM_HMODE_ADDR	0x00000058	Environment call from H-mode handler
CFG_NMI_CALL_FROM_MMODE_ADDR	0x00000060	Environment call from M-mode handler
CFG_NMI_INSTR_PAGE_FAULT_ADDR	0x00000068	MPU prohibits instruction execution
CFG_NMI_LOAD_PAGE_FAULT_ADDR	0x00000070	MPU prohibits data loading
CFG_NMI_STORE_PAGE_FAULT_ADDR	0x00000078	MPU prohibits data storing
CFG_NMI_STACK_OVERFLOW_ADDR	0x00000080	Stack Overflow handler
CFG_NMI_STACK_UNDERFLOW_ADDR	0x00000088	Stack Underflow handler

Table 4-10. CPU River Exceptions Table RTL configuration.

Note

To reduce memory usage by Exception Table it is possible to assign all exceptions into one address and manage the occured trap type using the value stored in the `MCAUSE` register.

4.9 C-code example

The following C-example demonstrates how to read current Core ID using inline assembler:

```
int fw_get_cpuid() {
    int ret;
    asm("csrr %0, mhartid" : "=r" (ret));
    return ret;
}
```

Chapter 5

System Debug

River CPU doesn't implement any special Debug Modes. Processor's architecture allows to halt pipeline at any time without performance degradation that allows to support non-invasive debugging. This feature is used by the debugger to continuously read CPU registers, Stack Trace buffer, CPI and etc without the execution interrupting.

Test Access Points (TAP) provides access to a processors registers, peripheries and a mapped memory banks. The latest repository version provides the following TAPs:

- UART TAP - debug via UART Master serial port with the default speed 115200 Baud
- JTAG - debug via JTAG interface with the usual frequency 12 MHz
- Ethernet with EDCL support - debug via 100 Mb Ethernet using UDP protocol and hardcoded MAC and IP.

It is possible to use all these TAP devices at the same time that was confirmed on FPGA board with simultaneous access via UART and Ethernet. Special debugger is a part of `riscv_vhdl` repository that is shared with the Apache 2.0. license too.

Additional hardware feature of the developed debugger:

- Hardware calculation of the Clock Per Instruction (CPI) index for the each core independenetly.
- Per each master device the bus loading statistic for the read and write transactions separately.
- Stack tracer on the hardware level.
- Hardware Breakpoints that a perfectly suited for the application debug entirely placed in a ROM.
- I/D Cache state statuses.

5.1 Debug Support Unit (DSU)

Debug Support Unit (DSU) was developed to interact with "RIVER" CPU via its debug port interace. This bus provides access to all internal CPU registers and states and may be additionally extended by request. Run control functionality like 'run', 'halt', 'step' or 'breakpoints' imlemented using proprietary algorithms and intend to simplify integration with debugger application.

Set of general registers and control registers (CSR) are described in RISC-V privileged ISA specification and also available for read and write access via debug port.

Note

Take into account that CPU can have any number of platform specific CSRs that usually not entirely documented.

5.2 DSU registers mapping

DSU acts like a slave AMBA AXI4 device that is directly mapped into physical memory. Default address location for our implementation is 0x80020000. Allocated for the DSU memory space is splited on several regions. The first 3 regions provides access to CPU's data via the debug port:

- **0x00000..0x08000 (Region 1):** CSR registers.
- **0x08000..0x10000 (Region 2):** General set of registers.
- **0x10000..0x18000 (Region 3):** Run control and debug support registers.
- **0x18000..0x20000 (Region 4):** Local DSU region that doesn't access CPU debug port.

Example:

Bus transaction at address *0x80023C10* will be redirected to Debug port with CSR index *0x782*.

5.2.1 CSR Region (32 KB)

Processor Control and Status Registers (MSR) directly mapped into this region of the DSU. Each CSR register is 64-bit width and mapped into address space accordingly with its index:

$$CSR_OFFSET = 8 * CSR_index;$$

Offset	Name	Reset	Description

Offset	Name	Reset	Description
0x00008	CPU_CSR_FFLAGS	00000000:00000000h	0x001: FPU accrued Exceptions fields from FCSR
0x00010	CPU_CSR_FRM	00000000:00000008h	0x002: FPU dynamic Rounding Mode fields from FCSR
0x00018	CPU_CSR_FCSR	00000000:00000008h	0x003: FPU Control and Status register (FRM + FFLAGS)
0x07880	CPU_CSR_MISA	80000000:00101108h	0xf10: List of supported instruction sets
0x07888	CPU_CSR_MVENDORID	00000000:000000F1h	0xf11: Vendor ID: `CFG_VENDOR_ID` parameter
0x07890	CPU_CSR_MARCHID	00000000:00000000h	0xf12: Architecture ID
0x07898	CPU_CSR_MIMPLEMENTATIONID	00000000:20190521h	0xf13: Implementation ID: `CFG_IMPLEMENTATION_ID` parameter
0x078A0	CPU_CSR_MHARTID	00000000:00000000h	0xf14: Core ID in multi-processors system
0x00208	CPU_CSR_UEPC	00000000:00000000h	0x041: User program counter
0x01800	CPU_CSR_MSTATUS	00000000:00001800h	0x300: Machine mode status register
0x01810	CPU_CSR_MEDELEG	00000000:00000000h	0x302: Machine exception delegation
0x01818	CPU_CSR_MIDELEG	00000000:00000000h	0x303: Machine interrupt delegation
0x01820	CPU_CSR_MIE	00000000:00000000h	0x304: Machine interrupt enable bit
0x01828	CPU_CSR_MTVEC	00000000:00000000h	0x305: Machine Interrupts Table pointer
0x01A00	CPU_CSR_MSCRATCH	00000000:00000000h	0x340: Machine scratch register
0x01A08	CPU_CSR_MEPC	00000000:00000000h	0x341: Machine program counter
0x01A10	CPU_CSR_MCAUSE	00000000:00000000h	0x342: Machine trap cause
0x01A18	CPU_CSR_MBADADDR	00000000:00000000h	0x343: Machine bad address
0x01A20	CPU_CSR_MIP	00000000:00000000h	0x344: Machine Interrupt pending
0x01A80	CPU_CSR_MSTACKOVR	00000000:00000000h	0x350: Machine Stack Overflow limit
0x01A88	CPU_CSR_MSTACKUND	00000000:00000000h	0x351: Machine Stack Underflow limit
0x01A90	CPU_CSR_MPU_ADDR	00000000:00000000h	0x352: MPU Region Base Address
0x01A98	CPU_CSR_MPU_MASK	00000000:00000000h	0x353: MPU Region Base Size
0x01AA0	CPU_CSR_MPU_CTRL	00000000:00000000h	0x354: MPU Region Control register

Table 5-1. List of mapped CPU CSR registers.

5.2.2 CPU General Registers Region (32 KB)

AXI4 memory requests to this DSU region (0x08000) directly routed to the debug interface connected to all CPUs in a system. DSU_CPU_CONTEXT Register selects the exact CPU target in a multi-processor configuration.

Offset	Name	Reset	Description
0x08000	CPU_ZERO	64h'0	**x0**. Hardware zero.
0x08008	CPU_RA	64h'0	**x1**. Return address.
0x08010	CPU_SP	64h'0	**x2**. Stack pointer.
0x08018	CPU_GP	64h'0	**x3**. Global pointer.
0x08020	CPU_TP	64h'0	**x4**. Thread pointer.
0x08028	CPU_T0	64h'0	**x5**. Temporaries 0.
0x08030	CPU_T1	64h'0	**x6**. Temporaries 1.
0x08038	CPU_T2	64h'0	**x7**. Temporaries 2.
0x08040	CPU_S0	64h'0	**x8**. Saved register 0/ Frame pointer.
0x08048	CPU_S1	64h'0	**x9**. Saved register 1.
0x08050	CPU_A0	64h'0	**x10**. Function argument 0. Return value.
0x08058	CPU_A1	64h'0	**x11**. Function argument 1.
0x08060	CPU_A2	64h'0	**x12**. Function argument 2.
0x08068	CPU_A3	64h'0	**x13**. Function argument 3.
0x08070	CPU_A4	64h'0	**x14**. Function argument 4.
0x08078	CPU_A5	64h'0	**x15**. Function argument 5.
0x08080	CPU_A6	64h'0	**x16**. Function argument 6.
0x08088	CPU_A7	64h'0	**x17**. Function argument 7.
0x08090	CPU_S2	64h'0	**x18**. Saved register 2.
0x08098	CPU_S3	64h'0	**x19**. Saved register 3.
0x080a0	CPU_S4	64h'0	**x20**. Saved register 4.
0x080a8	CPU_S5	64h'0	**x21**. Saved register 5.
0x080b0	CPU_S6	64h'0	**x22**. Saved register 6.
0x080b8	CPU_S7	64h'0	**x23**. Saved register 7.
0x080c0	CPU_S8	64h'0	**x24**. Saved register 8.
0x080c8	CPU_S9	64h'0	**x25**. Saved register 9.
0x080d0	CPU_S10	64h'0	**x26**. Saved register 10.
0x080d8	CPU_S11	64h'0	**x27**. Saved register 11.
0x080e0	CPU_T3	64h'0	**x28**. Temporaries 3.
0x080e8	CPU_T4	64h'0	**x29**. Temporaries 4.
0x080f0	CPU_T5	64h'0	**x30**. Temporaries 5.
0x080f8	CPU_T6	64h'0	**x31**. Temporaries 6.
0x08100	CPU_PC	64h'0	The latest executed instruction address
0x08108	CPU_NPC	64h'0	Next Instruction Pointer
0x08110	CPU_STACKTRACE_CNT	64h'0	Hardware stack trace counter
0x08200	CPU_FT0	64h'0	FPU register 0
0x08208	CPU_FT1	64h'0	FPU register 1
0x08210	CPU_FT2	64h'0	FPU register 2
0x08218	CPU_FT3	64h'0	FPU register 3
0x08220	CPU_FT4	64h'0	FPU register 4

Offset	Name	Reset	Description
0x08228	CPU_FT5	64h'0	FPU register 5
0x08230	CPU_FT6	64h'0	FPU register 6
0x08238	CPU_FT7	64h'0	FPU register 7
0x08240	CPU_FS0	64h'0	FPU register 8
0x08248	CPU_FS1	64h'0	FPU register 9
0x08250	CPU_FA0	64h'0	FPU register 10
0x08258	CPU_FA1	64h'0	FPU register 11
0x08260	CPU_FA2	64h'0	FPU register 12
0x08268	CPU_FA3	64h'0	FPU register 13
0x08270	CPU_FA4	64h'0	FPU register 14
0x08278	CPU_FA5	64h'0	FPU register 15
0x08280	CPU_FA6	64h'0	FPU register 16
0x08288	CPU_FA7	64h'0	FPU register 17
0x08290	CPU_FS2	64h'0	FPU register 18
0x08298	CPU_FS3	64h'0	FPU register 19
0x082A0	CPU_FS4	64h'0	FPU register 20
0x082A8	CPU_FS5	64h'0	FPU register 21
0x082B0	CPU_FS6	64h'0	FPU register 22
0x082B8	CPU_FS7	64h'0	FPU register 23
0x082C0	CPU_FS8	64h'0	FPU register 24
0x082C8	CPU_FS9	64h'0	FPU register 25
0x082D0	CPU_FS10	64h'0	FPU register 26
0x082D8	CPU_FS11	64h'0	FPU register 27
0x082E0	CPU_FT8	64h'0	FPU register 28
0x082E8	CPU_FT9	64h'0	FPU register 29
0x082F0	CPU_FT10	64h'0	FPU register 30
0x082F8	CPU_FT11	64h'0	FPU register 31
0x08400	CPU_STACKTRACE_BUF	64h'0	Hardware Stack Trace Buffer

Table 5-2. List of mapped CPU General Purpose Registers.

5.2.3 Run Control Region (32 KB)

AXI4 memory requests to this DSU region (0x10000) directly routed to the debug interface connected to all CPUs in a system. DSU_CPU_CONTEXT Register selects the exact CPU target in a multi-processor configuration.

Offset	Name	Reset Val.	Description
0x10000	CPU_CONTROL	00000000:00000000h	Run Control and Status
0x10008	CPU_STEPS	00000000:00000000h	Steps value while in the Stepping Mode
0x10010	CPU_CLOCK_CNT	00000000:00000000h	Clock cycles counter since hardware reset
0x10018	CPU_EXECUTED_CNT	00000000:00000000h	Total number of the executed instructions

Offset	Name	Reset Val.	Description
0x10020	CPU_BR_CONTROL	00000000:00000000h	Breakpoint mode control
0x10028	CPU_BR_ADD	00000000:00000000h	Add Hardware breakpoint address
0x10030	CPU_BR_REMOVE	00000000:00000000h	Remove Hardware breakpoint address
0x10038	CPU_BR_ADDR_FETCH	00000000:00000000h	Redefine instruction from this address
0x10040	CPU_BR_INSTR_FETCH	00000000:00000000h	Redefined instruction value for the fecthed address
0x10048	CPU_IFLUSH	00000000:00000000h	Flush Instruction Cache

Table 5-3. Run Control registers list.

CPU_CONTROL Register (0x10000)

Field	Bits	Type	Description
rsrv	[63:16]	RZ	reserved.
core_id	[19:4]	RO	**Core Index**. Unique Core index in a multiprocessor system.
hw_br	[3]	RO	**Hardware Breakpoint**. CPU was halted on a hardware breakpoint satus.
sw_br	[2]	RO	**Software Breakpoint**. CPU was halted on the EBREAK instruction.
stepping	[1]	RW	**Stepping Mode**. Enable/disable CPU stepping mode. • '0' - Normal mode • '1' - Stepping mode In Stepping Mode use register `CPU_STEPS` to execute a certain number of the instructions.
halt	[0]	RW	**Halt**. Start/stop CPU execution. • '0' - CPU is running • '1' - CPU is halted Start or Stop processor execution doesn't affect the CPU pipeline or caches. And can be implemented at any time without instructions penalties.

CPU_STEPS Register (0x10008)

Field	Bits	Type	Description
steps	[63:0]	RW	**Step Counter**. Total number of instructions that CPU should execute before switching into the halt state. Bit `'step_mode'` in the Register `CPU_CONTROL` must be set to HIGH to switch CPU into the Stepping Mode.

CPU_CLOCK_CNT Register (0x10010)

Field	Bits	Type	Description
clk_cnt	[63:0]	RO	**Clock Counter**. Clock counter is used in the hardware computation of the CPI rate. Clock counter isn't incrementing if the CPU is in Halt state.

CPU_EXECUTED_CNT Register (0x10018)

Field	Bits	Type	Description
exec_cnt	[63:0]	RO	**Step Counter**. Total number of the executed instructions. Step counter is used in the hardware computation of the CPI rate.

CPU_BR_CONTROL Register (0x10020)

Field	Bits	Type	Description
rsrv	[63:1]	RZ	reserved.
trap_on_break	[0]	RW	**Trap On Breakpoint**. Enable/disable exception `'Breakpoint'` generation on EBREAK instruction. If Trap-On-Breakpoint disabled CPU halts the pipeline without exception. • '0' - Trap-On-Breakpoint disabled (default) • '1' - Trap-On-Breakpoint enabled

CPU_BR_ADD Register (0x10028)

Field	Bits	Type	Description
add_break	[63:0]	WO	**Breakpoint Address**. Add specified address into Hardware breakpoint array. In case of matching Instruction Pointer (`pc`) and any value from this array `Fetch` module in the CPU pipeline injects EBREAK instruction on the hardware level. Type of the breakpoint is shown in the status bits of `CPU_CONTROL` register.

CPU_BR_REMOVE Register (0x10030)

Field	Bits	Type	Description
rm_break	[63:0]	WO	**Breakpoint Address**. Remove specified address from the Hardware breakpoints array.

CPU_BR_ADDR_FETCH Register (0x10038)

Field	Bits	Type	Description
br_addr_fetch	[63:0]	RW	**Breakpoint Address**. Specify address that will be ignored by CPU Fetch module and will be redefined instruction data stored in the register `CPU↩ _BR_INSTR_FETCH`. This logic is used to avoid re-writing EBREAK into memory on each Software breakpoint. See examples chapter for more details.

CPU_BR_INSTR_FECTH Register (0x10040)

Field	Bits	Type	Description
br_instr_fetch	[63:0]	RW	**Breakpoint Instruction Data**. Specify the instruction data that will be executed instead of fetched from memory in a case of matching `CPU_BR_AD↩ DR_FECTH` register and CPU Instruction Pointer (`pc`). See examples chapter for more details.

5.2.4 Local DSU Region (32 KB)

Access to the DSU registers in this region from the system bus doesn't generate transactions on the debug interface connected to CPUs debug port.

Offset	Name	Reset Val.	Description
0x18000	DSU_SOFT_RESET	00000000:00000000h	CPU Software Reset
0x18008	DSU_CPU_CONTEXT	00000000:00000000h	CPU selector
0x18040	DSU_MST0_W_CNT	00000000:00000000h	Bus write statistic for Master[0]
0x18048	DSU_MST0_R_CNT	00000000:00000000h	Bus read statistic for Master[0]
0x18050	DSU_MST1_W_CNT	00000000:00000000h	Bus write statistic for Master[1]
0x18058	DSU_MST1_R_CNT	00000000:00000000h	Bus read statistic for Master[1]
0x18060	DSU_MST2_W_CNT	00000000:00000000h	Bus write statistic for Master[2]
0x18068	DSU_MST2_R_CNT	00000000:00000000h	Bus read statistic for Master[2]
0x18070	DSU_MST3_W_CNT	00000000:00000000h	Bus write statistic for Master[3]
0x18078	DSU_MST3_R_CNT	00000000:00000000h	Bus read statistic for Master[3]

Table 5-4. DSU Local registers list.

DSU_SOFT_RESET Register (0x18000)

Field	Bits	Type	Description
rsrv	[63:1]	RZ	reserved.
soft_rst	[0]	RW	**Software Reset**. Reset the connected processor cores. This reset signal doesn't affect Interrupt Controller, GPIO Controller and other modules so the software has to properly handle software resets.

DSU_CPU_CONTEXT Register (0x18008)

Field	Bits	Type	Description
rsrv	[63:16]	RZ	reserved.
context	[15:0]	RW	**CPU Context**. This value swithes DSU debug interface to a specific CPU debug port. For the single core configuration this value always equals to zero. In a multi-core configuration it can be changed in a range from 0 to (CPU_TOTAL-1).

DSU_MSTx_W_CNT Regsiter (0x18040 + 0x10$*$x)

Field	Bits	Type	Description
w_cnt	[63:0]	RO	**Write transactions counter**. These registers represents the simplify bus tracer and contain information about total number of write requests for a specific master device on a AXI4 system bus.

DSU_MSTx_R_CNT Regsiter (0x18048 + 0x10∗x)

Field	Bits	Type	Description
r_cnt	[63:0]	RO	**Read transactions counter**. These registers represents the simplify bus tracer and contain information about total number of read requests for a specific master device on a AXI4 system bus.

5.3 C-examples

The CPU bundle is shared with enabled DSU module and Debugger application that can be connected through the Ethernet, JTAG or UART interfaces. This section contains some information of how the Debugger application interacts with the Hardware platform.

5.3.1 Breakpoints

Example of the functions adding/removing the Software Breakpoint:

```
// DSU registers:
#define CPU_IFLASH 80030048

union Reg64Type {
    uint8_t v8[8];
    uint16_t v16[4];
    uint32_t v32[2];
    uint64_t val;
};

// Prototypes of the access methods to SoC via the debug interface:
void tap_read_mem64(uint64_t addr, uint64_t *val);
void tap_write_mem64(uint64_t addr, uint64_t val);

void add_sw_breakpoint(uint64_t addr) {
    Reg64Type old;
    Reg64Type br;
    // Read and save somewhere original memory value
    tap_read_mem64(addr, &old.val);

    // Inject EBREAK instruction into memory:
    br = old;
    if ((old.v32[0] & 0x3) == 0x3) {
        // 32-bits instruction:
        br.v32[0] = 0x00100073;     // EBREAK instruction;
    } else {
        // Compressed 16-bits instruction
        br.v16[0] = 0x9002;         // C.EBREAK instruction;
    }
    tap_write_mem64(addr, br.val);

    // Access to DSU to flush ICache lines containing breakpoint address
    tap_write_mem64(CPU_IFLASH, addr);
}


void remove_sw_breakpoint(uint64_t addr) {
    Reg64Type old;
    get_original_value(addr, &old);    // must be implemented somewhere

    // Restore original memory value and flush ICache lines:
    tap_write_mem64(addr, br.val);
    tap_write_mem64(CPU_IFLASH, addr);
}
```

Example of the functions adding/removing the Hardware Breakpoint:

```
// DSU registers:
#define CPU_BR_ADD    80030028
#define CPU_BR_REMOVE 80030030

// No need to flush ICache
void add_hw_breakpoint(uint64_t addr) {
    tap_write_mem64(CPU_BR_ADD, addr);
}

void remove_hw_breakpoint(uint64_t addr) {
    tap_write_mem64(CPU_BR_REMOVE, addr);
}
```

5.3.2 Run Control

Example of run and halt the processor execution:

```
// DSU registers:
#define CPU_CONTROL    80030000
#define CPU_STEPS      80030008

// CPU Control register bit fields:
union cpu_control_type {
    uint64_t val;
    struct bits_type {
        uint64_t halt     : 1;
        uint64_t stepping : 1;
        uint64_t sw_breakpoint : 1;
        uint64_t hw_breakpoint : 1;
        uint64_t core_id  : 16;
        uint64_t rsrv     : 44;
    } bits;
};

// Run processor
void run_cpu() {
    cpu_control_type ctrl;
    ctrl.val = 0;
    tap_write_mem64(CPU_CONTROL, ctrl.val);
}

// Halt processor
void halt_cpu() {
    cpu_control_type ctrl;
    ctrl.val = 0;
    ctrl.bits.halt = 1;
    tap_write_mem64(CPU_CONTROL, ctrl.val);
}

// Run specified number of instructions
void step_cpu(int cnt) {
    cpu_control_type ctrl;

    tap_write_mem64(CPU_STEPS, (uint64_t)cnt);

    ctrl.val = 0;
    ctrl.bits.stepping = 1;
    tap_write_mem64(CPU_CONTROL, ctrl.val);
}
```

Example of the function polling processor status bits and waiting any breakpoint that will halt the execution:

```
// DSU registers
#define CPU_CONTROL         80030000
#define CPU_BR_ADDR_FETCH   80030038
#define CPU_BR_INSTR_FETCH  80030040

void foo1() {
    Reg64Type t1;
    cpu_control_type ctrl;

    // Add breakpoint at address 0x100:
    tap_read_mem64(0x100, &t1.val);     // save original value
    add_sw_breakpoint(0x100);

    // Run processor:
    run_cpu();

    // Wait breakpoint
    tap_read_mem64(CPU_CONTROL, &ctrl.val);
    while (ctrl.bits.halt == 0) {
        tap_read_mem64(CPU_CONTROL, &ctrl.val);
    }

    // To avoid CPU stuck on the breakpoint we should skip current breakpoint
    // but would like to avoid removing it:
    if (ctrl.bits.sw_breakpoint == 1) {
        tap_write_mem64(CPU_BR_ADDR_FETCH, 0x100);
        tap_write_mem64(CPU_BR_INSTR_FETCH, t1.val);
    }

    // Continue
    run_cpu();
}
```

5.3.3 Multi-Cores contexts

Example of function reading all integer registers for each of two available CPU cores:

```
// DSU registers
#define CPU_ZERO            80028000
#define DSU_CPU_CONTEXT     80038008

void read_iregs() {
    Reg64Type ireg0[34];   // x0..x31, pc, npc
    Reg64Type ireg1[34];   // x0..x31, pc, npc

    // Select CPU[0]
    tap_write_mem64(CPDSU_CPU_CONTEXT, 0);
    tap_read_mem(CPU_ZERO, ireg0[0].v8, sizeof(ireg0));

    // Select CPU[1]
    tap_write_mem64(CPDSU_CPU_CONTEXT, 1);
    tap_read_mem(CPU_ZERO, ireg1[0].v8, sizeof(ireg0));
}
```

SERGEYKHBR RISCV_VHDL

Chapter 6

Interrupts Management

All interrupts used in a system connected through the dedicated interrupt controller `irqctrl`, available on AXI4 bus as the slave device.

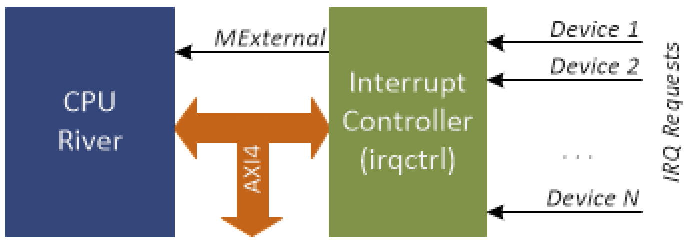

Figure 6-1. Connection of the Interrupt Controller to the System Bus.

Any interrupt request from any device is directly latching into the pending register with the 1 clock cycle delay in a case if the bit `lock` in register IRQS_LOCK equals to 0. Otherwise IRQ request will be latched into `irq_wait↩ _unlock` register and postponed until `lock` bit becomes 0. This logic allows to postpone interrupts even if its duration equals to 1 clock cycle.

The signal `MExternal` connected to a processor and asserted to the level HIGH until register `IRQS_PENDING` not equal to zero. Processor should handle `MExternal` interrupt by calling the top-level handler. Address of this handler function is specified by value of CSR `MTVEC`. The handler has to check and clear all pending bits in `IRQS_PENDING` register and call appropriate handler of lower level corresponding to the each raised bit.

Currrent River CPU revision doesn't implement `Wallclock Timer` which is described in RISC-V specification. This functionality is fully implemented in a separate AXI devices:

- Interrupt Controller (`irqctrl`)
- General Purpose Timers (`gptimers`).

6.1 Interrupts List

IRQ pins configuration is the part of generic constants defined in file *ambalib/types_amba4.vhd*. Number of the interrupts and its indexes can be changed in a future releases.

Pin	Name	Description
0	IRQ_ZERO	Not used. Connected to GND.
1	IRQ_UART1	Serial input/output device interrupt request
2	IRQ_ETHMAC	Ethernet IRQ.
3	IRQ_GPTIMERS	General Purpose Timers interrupt request.
4	reserved	reserved
5	IRQ_GNSSENGINE	Reserved for the GNSS Engine

Lower Interrupt index has a higher priority. Interrupt index 0 cannot be assigned to a specific device and always connected to GND.

6.2 Interrupt Controller Registers

IRQ Controller module is connected as a slave device to the AXI4 Bus Controller and available for reading and writing as a single port memory for any master device in the system.

AXI4 IRQ Controller configured with the following generic parameters by default:

Name	Value	Description
async_reset	FALSE	**Reset Type**. Internal registers reset type: • FALSE syncrhonous reset (FPGA) • TRUE asynchronous reset (ASIC)
xaddr	16#80002#	**Base address**. Base Address value defines bits [31:12] of the allocated memory space
xmask	16#FFFFF#	**Address Mask**. Address Mask is used by system controller to defines allocated memory size

Table 6-1. IRQCTRL generic parameters.

These generic parameters directly define the IRQ Controller device memory location in the system memory map. Base Address is 0x80002000. Allocated memory size is 4 KB.

The full list of Registers relative Device Base Address offset is shown in the following table.

Device Registers list

Offset	Name	Reset Val.	Description
0x000	IRQS_MASK	0000:003Eh	Interrupt mask
0x004	IRQS_PENDING	0000:0000h	Interrupt requests
0x008	IRQS_CLEAR	0000:0000h	Pending bits clear

Offset	Name	Reset Val.	Description
0x00C	IRQS_RAISE	0000:0000h	Manual interrupt requests
0x010	ISR_TBL_LOW	0000:0000h	Software Interrupt Handlers table (lower dword)
0x014	ISR_TBL_HIGH	0000:0000h	Software Interrupt Handlers table (upper dword)
0x018	DBG_CAUSE_LOW	0000:0000h	Debug register to save interrupt cause (lower dword)
0x01C	DBG_CAUSE_HIGH	0000:0000h	Debug register to save interrupt cause (upper dword)
0x020	DBG_EPC_LOW	0000:0000h	Debug register to save instruction pointer (lower dword)
0x024	DBG_EPC_HIGH	0000:0000h	Debug register to save instruction pointer (upper dword)
0x028	IRQ_LOCK	0000:0000h	Software interrupts lock
0x02C	IRQ_CAUSE_IDX	0000:0000h	Debug register with the IRQ index

Table 6-2. AXI4 Interrupt Controller Registers.

IRQS_MASK Register (0x000)

Field	Bits	Type	Description
rsrv	[31:N]	RZ	reserved.
mask	[N-1:0]	RW	**IRQ Mask**. Enable or disable the interrupt request for the specific device defined by bit position: • '0' - Enabled • '1' - Disabled (reset value)

IRQS_PENDING Register (0x004)

Field	Bits	Type	Description
rsrv	[31:N]	RZ	reserved.
pending	[N-1:0]	RO	**Pending Bits**. This register holds the requested interrupts from all unmasked devices: • '0' - No IRQ request • '1' - Was IRQ request If any of these bits not equal to zero Interrupt Controller set the signal to HIGH. Register `IRQS_CLEAR` has to be used to clear specified bit position.

IRQS_CLEAR Register (0x008)

Field	Bits	Type	Description
rsrv	[31:N]	RZ	reserved.

Field	Bits	Type	Description
clear_irq	[N-1:0]	WO	**Clear Pending Bits**. Clear bits in IRQS_PENDING register marked as 1: • '0' - No action • '1' - Clear pending bit

IRQS_RAISE Register (0x00C)

Field	Bits	Type	Description
rsrv	[31:N]	RZ	reserved.
raise_irq	[N-1:0]	WO	**Request IRQ**. This register allows to request IRQ for a specific device manually to call the interrupt handler and verify Software functionality: • '0' - No action • '1' - Request IRQ

ISR_TBL Register (0x010)

Field	Bits	Type	Description
isr_table	[63:0]	RW	**SW Interrupt Table address**. Software may use this dedicated register to store Software defined IRQs handlers table. Upper and lower parts of this register can be accessed separately. For more details see folder with `riscv_vhdl/examples/*`.

DBG_CAUSE Register (0x018)

Field	Bits	Type	Description
dbg_cause	[63:0]	RW	**Cause of the Interrupt**. This register stores the latest `MCAUSE` CSR value. This value is an optional and updates by ROM ISR handler in the current implementation to simplify RTL simulation analysis. Upper and lower parts of this register can be accessed separately.

DBG_EPC Register (0x020)

Field	Bits	Type	Description
dbg_epc	[63:0]	RW	**Instruction Pointer before trap**. This register stores copy of xEPC value. This value is an optional and updates by ROM ISR handler in the current implementation to simplify RTL simulation analysis. Upper and lower parts of this register can be accessed separately.

IRQ_LOCK Register (0x028)

Field	Bits	Type	Description
rsrv	[31:1]	RZ	reserved.
lock	[0]	RW	**Lock Interrupts**. Disabled all interrupts when this bit is 1. All new interrupt requests marked as postponed and will be raised when 'lock' signal becomes cleared: • '0' - interrupts enabled • '1' - interrupts disabled (postponed)

IRQ_CAUSE_IDX Register (0x02C)

Field	Bits	Type	Description
irq_idx	[31:0]	RW	**Interrupt Index**. This register stores current interrupt index while in processor executes ISR handler. This value is optional and updates by ROM ISR handler in current implementation to simplify RTL simulation analysis.

6.3 C-examples

The CPU bundle is shared with the software examples describing how to properly setup and use the interrupts in your application. For more details see folder `riscv_vhdl/examples/`.

The following C-strucutre can be used to describe IRQ Controller memory map:

```
// Interrupt handler prototype
typedef void (*IRQ_TABLE_HANDLER)(int idx, void *arg);

typedef struct irqctrl_map {
    volatile uint32_t irq_mask;      // 0x00: [RW] 1=disable; 0=enable
    volatile uint32_t irq_pending;   // 0x04: [RW]
    volatile uint32_t irq_clear;     // 0x08: [WO]
    volatile uint32_t irq_rise;      // 0x0C: [WO]
    volatile uint64_t isr_table;     // 0x10: [RW]
    volatile uint64_t dbg_cause;     // 0x18:
    volatile uint64_t dbg_epc;       // 0x20:
    volatile uint32_t irq_lock;      // 0x28: lock/unlock all interrupts
    volatile uint32_t irq_cause_idx;// 0x2c:
} irqctrl_map;
```

Pseudo-code of the assembler function that is used as top-level interrupt handler including 'prologue' and 'epilogue' of the function.

```
// Close C-analog of the assembler function running in Machine mode:
void trap_entry() {
    save_cpu_context();         // Save CPU registers into memory
    MSTATUS &= ~MEIP;           // Clear MEIP bit in mstatus register

    handle_trap(MCAUSE,
                MEPC,
                StackPointer); // Call C-handler

    retore_cpu_context();       // Restore CPU registers into memory
    __asm__("mret");            // Return from interrupt restoring previous CPU mode
}
```

Example of top-level handler polling pending bits and calling the interrupt handlers of low level:

```
void handle_trap(cause, pc, sp) {
    irqctrl_map *p_irqctrl = (irqctrl_map *)0x80002000;

    // Software table with the ISR handlers:
    IRQ_HANDLER irq_handler = (IRQ_HANDLER *)p_irqctrl->isr_table;

    // Read into local variable and clear pending bits:
    p_irqctrl->irq_lock = 1;
    pending = p_irqctrl->irq_pending;
    p_irqctrl->irq_clear = pending;
    p_irqctrl->irq_lock = 0;

    // Sequentially call device isr handlers:
    for (int i = 0; i < CFG_IRQ_TOTAL; i++) {
        if (pending & 0x1) {
            p_irqctrl->irq_cause_idx = i;
            irq_handler(i, NULL);
        }
        pending >>= 1;
    }
}
```

Chapter 7

GPIO Controller

System-on-Chip based on River CPU provides additional periphery module GPIO Controller (`axi4_gpio`). This modules instantiates `width` number of bi-directional pins, where `width` is the generic parameter which is equal to 12 for all used fpga boards (4 LEDs + 8 User defined DIPs). ALL GPIOs connected via bi-directional buffer with the Z-state and can be programmed as inputs or outputs.

Default ASIC configuraton supposes that all GPIO will be configures as input pins after hard reset but for the FPGA targets GPIOs are splitted on input connected to User's DIP switch and output connected to LEDs so the hard reset state may differ from default configuration.

7.1 Boot Loader Configuration

GPIOs pins is used to pass some configuration and select desired behaviour of program loaded into Boot ROM. There're plenty number of possible images that can be used as boot image as available in folder `examples` ether as not available:

- `examples/boot/*`. Demonstration ROM image starting user application.
- `examples/bootrom_tests/*`. Self test application running directly from ROM
- `examples/sysboot/*`. Full functional ASIC boot loader.
- Other custom application.

Depending of the used ROM program there're may be the following GPIO pins assignment:

GPIO[1]	GPIO[0]	Boot Description
0	0	Boot from the internal Boot ROM module (default)
0	1	Boot from the external flash IC
1	0	Secured Boot from the internal OTP memory
1	1	Run infinite SRAM test from the Boot ROM

7.2 GPIO Controller Registers

GPIO Controller module is connected as a slave device to the AXI4 Bus Controller and available for reading and writing as a single port memory for any master device in the system.

AXI4 GPIO Controller configured with the following generic parameters by default:

Name	Value	Description
async_reset	FALSE	**Reset Type**. Internal registers reset type: • FALSE syncrhonous reset (FPGA) • TRUE asynchronous reset (ASIC)
xaddr	16#80002#	**Base address**. Base Address value defines bits [31:12] of the allocated memory space
xmask	16#FFFFF#	**Address Mask**. Address Mask is used by system controller to defines allocated memory size
xirq	0	**IRQ index**. Unused in the current configuration.
width	12	**GPIO Width**. Total number of bi-directional pins.

Table 7-1. GPIO generic parameters.

These generic parameters directly define the GPIO Controller device memory location in the system memory map. Base Address is 0x80000000. Allocated memory size is 4 KB.

The full list of Registers relative Device Base Address offset is shown in the following table.

Device Registers list

Offset	Name	Reset Val.	Description
0x000	GPIO_DIRECTION	0000:03FFh	IOs direction
0x004	GPIO_IUSER	0000:0000h	Input IOs values
0x008	GPIO_OUSER	0000:0000h	Output IOs values
0x00C	GPIO_REG32	0000:0000h	Debug register

Table 7-2. AXI4 GPIO Controller Registers.

GPIO_DIRECTION Register (0x000)

Field	Bits	Type	Description
rsrv	[31:width]	RZ	reserved.

Field	Bits	Type	Description
direction	[width-1:0]	RW	**IO Direction**. Data direction register determines which pins should work as an inputs and which one as an outputs: • '0' - Output • '1' - Input (reset value)

GPIO_IUSER Register (0x004)

Field	Bits	Type	Description
rsrv	[31:width]	RZ	reserved.
iuser	[width-1:0]	RO	**Input Data**. This register provides direct read access to the IOs pins. If some pins were configured as an output then corresponding bits in the read value will have unpredictable values and should be ignored.

GPIO_OUSER Register (0x008)

Field	Bits	Type	Description
rsrv	[31:width]	RZ	reserved.
ouser	[width-1:0]	RW	**Output Data**. Output Data register contains a data latch for each pins configured as an output. If pin configured as an input corresponding bit value is ignoring.

GPIO_REG32 Register (0x00C)

Field	Bits	Type	Description
reg32↩_3	[31:0]	RW	**Temporary Data**. Read/Write temporary register used by FW with the debug purposes.

7.3 C-code example

The following C-example demonstrates procedures of interaction with the GPIO module that allows to read User's DIP configuration and switch User's LEDs connected to SoC outputs. Let's suppose the following configuration:

- GPIO[3:0] controlled by user DIPs (inputs)
- GPIO[11:4] connected to LEDs (outputs)

```
// Memory Base address of the device
#define ADDR_BUS0_XSLV_GPIO   0x80002000

typedef struct gpio_map {
    volatile uint32_t direction;
    volatile uint32_t iuser;
    volatile uint32_t ouser;
    volatile uint32_t reg32_3;
} gpio_map;
```

```c
// Set index specific LED turn on/off
void set_led(int onoff, int idx) {
    gpio_map *gpio = (gpio_map *)ADDR_BUS0_XSLV_GPIO;
    // [11:4] LED, [3:0] DIP pins
    uint32_t ouser = gpio->ouser >> 4;
    ouser &= ~(1 << idx);
    ouser |= (onoff & 0x1) << idx;
    gpio->ouser = (ouser << 4);
}

// Read index specific DIP value
int get_dip(int idx) {
    // [3:0] DIP pins
    int dip = ((gpio_map *)ADDR_BUS0_XSLV_GPIO)->iuser >> idx;
    return dip & 1;
}

// Some initialization function
void soc_init() {
    gpio_map *gpio = (gpio_map *)GPIO_BASE_ADDR;
    gpio->direction = 0x00F;   // [11:4] LED, [3:0] DIP

    // Turn on LED[7]
    set_led(7, 0);

    // see examples/boot/main.c
    if (get_dip(0) == 1) {
        print_uart("Coping FLASH\r\n", 14);
        memcpy(sram, flash, FW_IMAGE_SIZE_BYTES);
    } else {
        print_uart("Coping FWIMAGE\r\n", 16);
        memcpy(sram, fwrom, FW_IMAGE_SIZE_BYTES);
    }
}
```

SERGEYKHBR
RISCV_VHDL

Chapter 8

UART Interface

In general UART interface uses two couples of unidirectional signal lines. One line for data transmission and one line to signal readiness for each direction. The input block `DX0` processes input signal `Rx`. Optional wires CTS and RTS are supported by UART IP block but they are unused and pull-down to GND line on the top level of the default SoC implementation (see `asic_top.vhd`)

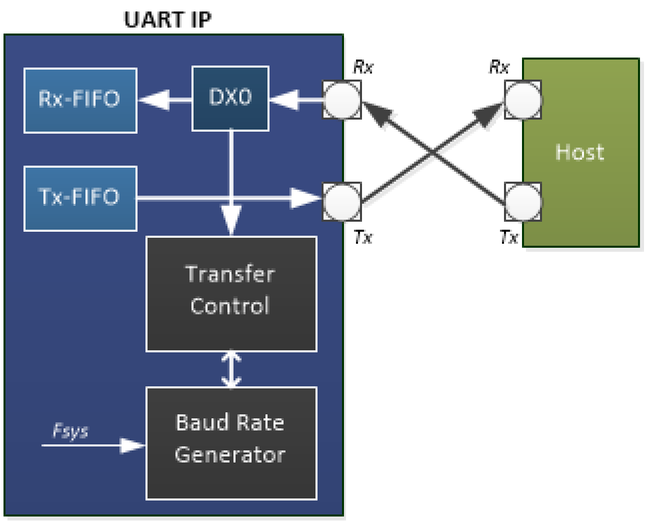

Figure 8-1. Bi-directional serial connection.

UART Frame format

The standard UART frame is shown below and it consists of the following parts:

- Idle state with signal level '1'
- One start bit of frame (SOF) with the signal level '0'

- Data sequence of 8 bits. Number of bits hardcoded on hardware level.
- Parity bit (P) that controlled via bit `parity_bit` in the register `UART_CTRL_STATUS`. This bit is computed as: D(7)^D(6)^D(5)^D(4)^D(3)^D(2)^D(1)^D(0).
- On stop bit

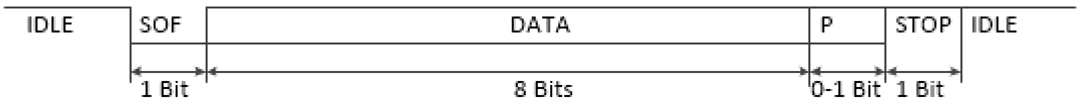

Figure 8-2. Standard UART frame.

Special protocol bits (SOF, P, STOP) automatically formed and handled in the finite state machine of the IP block and do not transmitted via FIFO buffers of the receiver and transmitter.

Baudrate Control

The final baudrate of the UART directly depends of the clock frequency of the IP block. In the default system implementation clock frequency of the IP block is equal to system bus frequency. Special counter `scaler` is used to form `posedge` and `negedge` events of the UART clock period. If the scaler value is equal to zero then UART is disabled and cannot receiving or transmitting information.

$$scaler = \frac{SYS_HZ}{2 * baudrate};$$

where `SYS_HZ` = 40 MHz (default FPGA system frequency); `baudrate` - 115200 (default UART baudrate).

8.1 UART Registers

UART module is connected as a slave device to the AXI4 Bus Controller and available for reading and writing as a single port memory for any master device in the system.

AXI4 UART configured with the following generic parameters by default:

Name	Value	Description
async_reset	FALSE	**Reset Type**. Internal registers reset type: • FALSE syncrhonous reset (FPGA) • TRUE asynchronous reset (ASIC)
xaddr	16#80001#	**Base address**. Base Address value defines bits [31:12] of the allocated memory space
xmask	16#FFFFF#	**Address Mask**. Address Mask is used by system controller to defines allocated memory size
irqx	1	**IRQ Index**. This value is used only as the information data in the Plug'n'Play structure and doesn't actualy define line index.
fifosz	16	**FIFO Size**. Size of the Tx and Rx FIFOs in bytes.

Table 8-1. UART generic parameters.

These generic parameters directly define the UART device memory location in the system memory map. Base Address is 0x80001000. Allocated memory size is 4 KB.

The full list of Registers relative Device Base Address offset is shown in the following table.

Device Registers list

Offset	Name	Reset Val.	Description
0x000	UART_CTRL_STATUS	0000:6022h	Control and Status Register
0x004	UART_SCALER	0000:0000h	Baudrate Control Register
0x008	UART_FWCPUID	0000:0000h	Multiprocessor Marker Register
0x010	UART_DATA	0000:0000h	Data Buffer Register

Table 8-2. AXI4 UART Registers.

UART_CTRL_STATUS Register (0x000)

Field	Bits	Type	Description
rsrv	[31:16]	RZ	reserved.
parity	[15]	RW	**Enable Parity check**. This bit is controlled by Software and allows to enable/disable parity checking: • 0 - parity disabled • 1 - parity enabled
tx_irq	[14]	RW	**Enable Tx Interrupt**. Generate interrupt when number of symbols in the output FIFO is less or equal than defined in Tx Threshold register (default threshold is equal zero). • 0 - interrupt disabled • 1 - interrupt enabled
rx_irq	[13]	RW	**Enable Rx Interrupt**. Generate interrupt when number of available symbols in the input FIFO is greater than defined in Rx Threshold register (default threshold is equal zero). • 0 - interrupt disabled • 1 - interrupt enabled
rsrv	[12:10]	RZ	reserved.
err_stopbit	[9]	RO	**Stop Bit Error**. This bit is set HIGH when the Stop Bit has the wrong polarity.
err_parity	[8]	RO	**Parity Error**. This bit is set HIGH when the Parity error occurs. Will be automatically cleared by the next received symbol if the parity OK.

Field	Bits	Type	Description
rsrv	[7:6]	RZ	reserved.
rx_fifo_empty	[5]	RO	**Receive FIFO Empty**. • 0 - Received data is available • 1 - No data in Rx FIFO
rx_fifo_full	[4]	RO	**Receive FIFO Full**. • 0 - Receiver can accept data • 1 - Next received symbol will be ignored
rsrv	[3:2]	RZ	reserved.
tx_fifo_empty	[1]	RO	**Transmit FIFO Empty**. • 0 - Transmitting in progress • 1 - All symbols were sent
tx_fifo_full	[0]	RO	**Transmit FIFO Full**. • 0 - Transmitter can accept data from Software • 1 - Writen symbol will be ignored

UART_SCALER Register (0x004)

Field	Bits	Type	Description
scaler	[31:0]	RW	**Scaler Threshold**. This registers specifies the overflow threshold that is used to form `posedge` and `negedge` events and form UART Baudrate.

UART_FWCPUID Register (0x008)

Field	Bits	Type	Description
fwcpuid	[31:0]	RW	**CPU ID**. This registers is controlled by software and is used to provide 'lock' mechanism in a multiprocessor configuration. • Zero value - Can be writen at any time • Non-Zero value - Can be writen only if previous value was zero

UART_DATA Register (0x010)

Field	Bits	Type	Description
rsrv	[31:8]	RZ	reserved.
data	[7:0]	RW	**Data**. This registers provides access to the Rx FIFO on read operation and to the Tx FIFO on write operation. FIFOs status should be controlled via UART_CTRL_STA↩ TUS register.

8.2 C-code example

The following C-example demonstrates procedures of interaction with the UART module that allows to send and receive data buffers.

```
// Memory Base address of the device
#define UART0_BASE_ADDR         0x80000000

// Control and Status Register bits:
#define UART_STATUS_TX_FULL     0x00000001
#define UART_STATUS_TX_EMPTY    0x00000002
#define UART_STATUS_RX_FULL     0x00000010
#define UART_STATUS_RX_EMPTY    0x00000020
#define UART_STATUS_ERR_PARITY  0x00000100
#define UART_STATUS_ERR_STOPBIT 0x00000200

#define UART_CONTROL_TXIRQ_ENA  0x00004000

typedef struct uart_map {
    volatile uint32_t status;
    volatile uint32_t scaler;
    volatile uint32_t fwcpuid;
    uint32_t rsrv[1];
    volatile uint32_t data;
} uart_map;

// Setup baudrate depending bus clock frequency.
// Default baudrate = 115200
void uart_set_baudrate(int bus_hz) {
    uart_map *uart = (uart_map *)UART0_BASE_ADDR;
    uart->scaler = bus_hz / 115200 / 2;
}

// Read data from Rx FIFO
int uart_read_data(char *buf) {
    uart_map *uart = (uart_map *)UART0_BASE_ADDR;
    int ret = 0;
    while ((uart->status & UART_STATUS_RX_EMPTY) == 0) {
        buf[ret++] = (char)uart->data;
    }
    return ret;
}

// Write data to Tx FIFO. Return number of bytes were writen.
int uart_write_data(char *buf, int sz) {
    uart_map *uart = (uart_map *)UART0_BASE_ADDR;
    int ret = 0;
    while (sz != 0) {
        uart->data = buf[ret++];
        if ((uart->status & UART_STATUS_TX_FULL) != 0) {
            break;
        }
    }
    return ret;
}

// Lock UART device for a specific CPU Core in multi-processors
// configuration.
uint32_t uart_lock(uint32_t cpuid) {
    uart_map *uart = (uart_map *)UART0_BASE_ADDR;
    while (uart->fwcpuid != cpuid) {
        uart->fwcpuid = cpuid;
    }
    return uart->fwcpuid;
}

// Release UART after CPU Core has finished interaction
void uart_unlock() {
    uart_map *uart = (uart_map *)UART0_BASE_ADDR;
    uart->fwcpuid = 0;
}
```

The following C-example demonstrates how to use all these methods in a simple "Hello world" application:

```
#define SYS_HZ 40000000    // System Clock frequency (default 40 MHz)
const char *message[] = "Hello World\n";

int main(int argc, char *argv[]) {
    uart_set_baudrate(SYS_HZ);
    int msgsz = strlen(message);
    int txcnt = 0;

    // Lock UART device to the current CPU Core #1
    uart_lock(1);

    // Send message buffer
    while (txcnt < msgsz) {
        txcnt += uart_write_data(&message[txcnt], msgsz - txcnt);
    }

    // Release UART
    uart_unlock();
    return 0;
}
```

Chapter 9

General Purpose Timers

9.1 GPTimers overview

This section describes General Purpose Timer interface module (GPTs). The GPTs is a N-channel timer that provides a timing reference with the interrupt generation and pulse-width-modulation functions. The following figure is a block diagram of the GPTs:

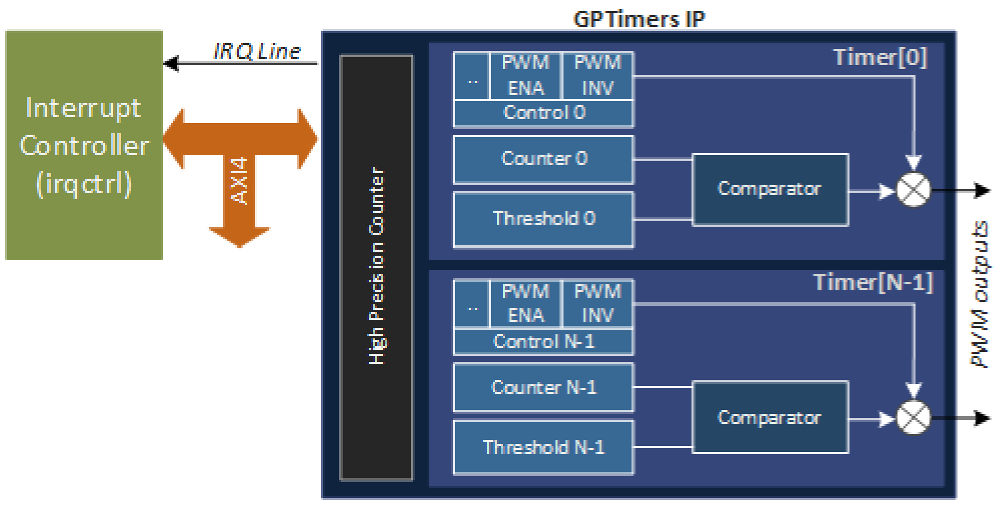

Figure 9-1. GPT connection to the System Bus.

All instantiated timers implement own registers set that includes pair of registers `GPT_INIT_VALUE_n` and G↩ PT_PWM_THRESHOLD_n. These registers allow to form modulated signal as shown on the following diagrams:

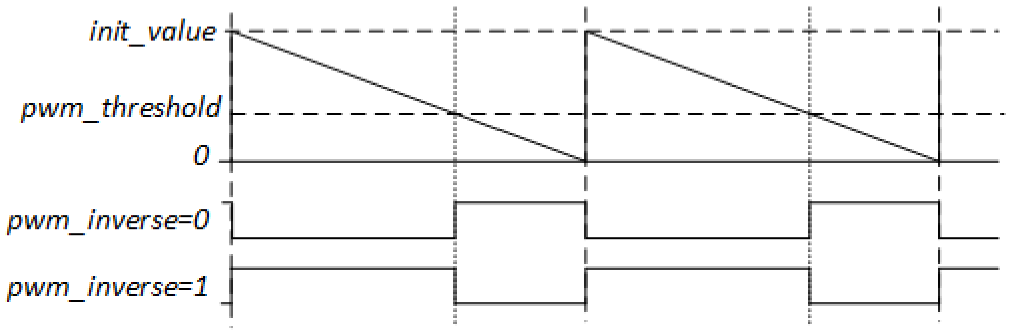

Figure 9-2. PWM signal formed by single timer channel.

Disabling channel timer or PWM modulation via control bits will set output pin into value writen in bit *pwm_polarity* of register `GPT_CONTROL_n`.

9.2 GPTs Registers

GPTs module is connected as a slave device to the AXI4 Bus Controller and available for reading and writing as a single port memory for any master device in the system.

AXI4 GPTs configured with the following generic parameters by default:

Name	Value	Description
async_reset	FALSE	**Reset Type**. Internal registers reset type: • FALSE syncrhonous reset (FPGA) • TRUE asynchronous reset (ASIC)
xaddr	16#80005#	**Base address**. Base Address value defines bits [31:12] of the allocated memory space
xmask	16#FFFFF#	**Address Mask**. Address Mask is used by system controller to defines allocated memory size
irqx	3	**IRQ Index**. This value is used only as the information data in the Plug'n'Play structure and doesn't actualy define line index.
tmr_total	2	**Total Number of Timers**. Total number of timers instantiated in the module. This value also defines total number of PWM outputs (one PWM signal per one timer). High precision counter is always present in GPTs.

Table 9-1. GPTs generic parameters.

These generic parameters directly define the GPTs device memory location in the system memory map. Base Address is 0x80005000. Allocated memory size is 4 KB.

The full list of Registers relative Device Base Address offset is shown in the following table.

Default Device Registers list (tmr_total = 2)

Offset	Name	Reset Val.	Description
0x000	GPT_HIGH_CNT	00000000:00000000h	High-precision 64-bit counter
0x008	GPT_IRQ_PENDING	0000:0000h	Interrupt Pending bits (1 bit per channel)
0x00C	GPT_PWM	0000:0000h	PWM output bits (1 bit per channel)
0x040	GPT_CONTROL_0	0000:0000h	Timer 0: Control Register
0x044	reserved	0000:0000h	Reserved
0x048	GPT_CNT_VALUE_0	00000000:00000000h	Timer 0: Current value of the channel's counter
0x050	GPT_INIT_VALUE_0	00000000:00000000h	Timer 0: Channel's counter init value when zero reached
0x058	GPT_PWM_THRESHOLD↩ _0	00000000:00000000h	Timer 0: Channel's threshold to invert pwm output
0x060	GPT_CONTROL_1	0000:0000h	Timer 1: Control Register
0x064	reserved	0000:0000h	Reserved
0x068	GPT_CNT_VALUE_1	00000000:00000000h	Timer 1: Current value of the channel's counter
0x070	GPT_INIT_VALUE_1	00000000:00000000h	Timer 1: Channel's counter init value when zero reached
0x078	GPT_PWM_THRESHOLD↩ _1	00000000:00000000h	Timer 1: Channel's threshold to invert pwm output

Table 9-2. AXI4 GPTs Registers.

GPT_HIGH_CNT Register (0x000)

Field	Bits	Type	Description
highcnt	[63:0]	RW	**High precision counter**. This counter isn't used as a source of interrupt and cannot be stopped from SW.

GPT_IRQ_PENDING Register (0x008)

Field	Bits	Type	Description
pending	[tmr_total-1:0]	RW	**Interrupts Pending**. Each timer can be configured to generate interrupt. Simultaneously with interrupt is raising pending bit that has to be lowered by Software • 0 - no interrupt • 1 - pending interrupt

GPT_IRQ_PWM Register (0x00C)

Field	Bits	Type	Description
pwm	[tmr_total-1:0]	RO	**PWM outputs**. Each timer can be configured to generate pwm pulses. This registers represents state of the output pin (one bit per channel).

GPT_CONTROL_n Register (0x040 + 0x20∗n)

Field	Bits	Type	Description
rsrv	[31:6]	RZ	reserved.
pwm_polarity	[5]	RW	**PWM polarity**. This bit is controlled by Software and allows to change PWM polarity: • 0 - Set pwm to `LOW` when counter equals 0x0 and `HIGH` when counter value equals to `GPT_PWM_THRESHOLD_n` • 1 - Set pwm to `HIGH` when counter equals 0x0 and `LOW` when counter value equals to `GPT_PWM_THRESHOLD_n`
pwm_ena	[4]	RW	**PWM enable**. This bit enables inversion of the pwm output when counter value reaches value writen in `GPT_PWM_THRESHOLD_n` regsiter. • 0 - Disable PWM • 1 - Enable PWM
rsrv	[3:2]	RZ	reserved
irq_ena	[1]	RW	**Interrupt Enable**. Enable IRQ pulse generation when counter reaches zero value • 0 - IRQ disabled • 1 - IRQ enabled
count_ena	[4]	RW	**Count Enabled**. • 0 - Timer disabled • 1 - Timer enabled

GPT_CNT_VALUE_n Register (0x050 + 0x20∗n)

Field	Bits	Type	Description
value	[63:0]	RW	**Timer Value**. Countdown counter value available for reading and writing. When this value reaches zero it will be re-initialized by value writen in regsiter `GPT_INIT_V`↩ `ALUE_n`

GPT_INIT_VALUE_n Register (0x058 + 0x20∗n)

Field	Bits	Type	Description
init_value	[63:0]	RW	**Timer Init Value**. Read/Write register is used for cycle timer re-initializtion. If init_value = 0 and value != 0 then the timer is used as a 'single shot' timer.

9.3 C-code example

The following C-example demonstrates procedure of the interaction with the High Precision counter to implement delay function:

```
// System clock in Hz (60 MHz)
#define SYS_HZ 60000000

// Memory Base address of the device
#define ADDR_BUS0_XSLV_GPTIMERS    0x80005000
#define GPT_TIMERS_TOTAL           2

// Channel's timer Control Registers bits
#define TIMER_CONTROL_ENA             (1 << 0)
#define TIMER_CONTROL_IRQ_ENA         (1 << 1)
#define TIMER_CONTROL_PWM_ENA         (1 << 4)
#define TIMER_CONTROL_PWM_POLARITY    (1 << 5)

// Channel timer register mapping
typedef struct gptimer_type {
    volatile uint32_t control;
    volatile uint32_t rsv1;
    volatile uint64_t cur_value;
    volatile uint64_t init_value;
    volatile uint64_t pwm_threshold;
} gptimer_type;

// Device registers mapping
typedef struct gptimers_map {
    volatile uint64_t highcnt;
    volatile uint32_t pending;
    volatile uint32_t pwm;
    uint32_t rsvr[12];
    gptimer_type timer[GPT_TIMERS_TOTAL];
} gptimers_map;

// Delay function in microseconds:
void delay(uint64_t usec) {
    gptimers_map *ptmr = (gptimers_map *)ADDR_BUS0_XSLV_GPTIMERS;
    uint64_t dlt_clk = usec * (SYS_HZ / 1000000ull);
    uint64_t starttime = ptmr->highcnt;
    uint64_t endtime = starttime + dlt_clk;

    // Check overlapping:
    if (endtime < starttime) {
        while (starttime < ptmr->highcnt) {}
    }

    while (ptmr->highcnt < endtime) {}
}
```

The following example shows how to enable PWM generation in timer channel 1 with the following parameters:

- Period PWM: 1 kHz
- Duty Cycle: 0.25

```
void enable_pwm() {
    gptimers_map *ptmr = (gptimers_map *)ADDR_BUS0_XSLV_GPTIMERS;
    ptmr->timer[1].init_value = SYS_HZ / 1000;
    ptmr->timer[1].pwm_treshold = SYS_HZ / 1000 / 4;
    ptmr->timer[1].icontrol = TIMER_CONTROL_ENA | TIMER_CONTROL_PWM_ENA;
}
```

Chapter 10

Plug'n'Play support module

10.1 PNP registers mapping

PNP module acts like a slave AMBA AXI4 device that is directly mapped into physical memory. Default address location for our implementation is defined as 0xFFFFF000. Memory size is 4 KB.

HW ID register (0x000).

Bits	Type	Reset	Field Name	Bits	Description
32	RO	CFG_HW_ID	hw_id	31:0	**HW ID**. Read only SoC identificator. Now it contains manually specified date in hex-format. Can be changed via CFG_HW_ID configuration parameter.

FW ID register (0x004).

Bits	Type	Reset	Field Name	Bits	Description
32	RW	32'h0	fw_id	31:0	**Firmware ID**. This value is modified by bootloader or user's firmware. Can be used to simplify firmware version tracking.

AXI Slots Configuration Register (0x008).

Bits	Type	Reset	Field Name	Bits	Description
8	RO	CFG_TECH	tech	7:0	**Technology ID**. Read Only value specifies the target configuration. Possible values: inferred, virtex6, kintex7. Other targets ID could be added in a future.
8	RO	CFG_NASTI_SLAVES_TOTAL	slaves	15:8	**Total number of AXI slave slots**. This value specifies maximum number of slave devices connected to the system bus. If device wasn't connected the dummy signals must be applied to the slave interface otherwise SoC behaviour isn't defined.
8	RO	CFG_NASTI_MASTER_TOTAL	masters	23:16	**Total number of AXI master slots**. This value specifies maximum number of master devices connected to the system bus. Slot signals cannot be unconnected either.
8	RO	8'h0	adc_detect	31:24	**ADC clock detector**. This value is used by GNSS firmware to detect presence of the ADC clock frequency that allows to detect presence of the RF front-end board.

Debug IDT register (0x010).

Bits	Type	Reset	Field Name	Bits	Description
64	RW	64'h0	idt	63:0	**Debug IDT**. This is debug register used by GNSS firmware to store debug information.

Debug Memory Allocation Pointer register (0x018).

Bits	Type	Reset	Field Name	Bits	Description
64	RW	64'h0	malloc_addr	63:0	**Memory Allocation Pointer**. This is debug register used by GNSS firmware to store 'heap' pointer and allows to debug memory management.

Debug Memory Allocation Size register (0x020).

Bits	Type	Reset	Field Name	Bits	Description
64	RW	64'h0	malloc_size	63:0	**Memory Allocation size**. This is debug register used by G↩ NSS firmware to store total allocated memory size.

Debug Firmware1 register (0x028).

Bits	Type	Reset	Field Name	Bits	Description
64	RW	64'h0	fwdbg1	63:0	**Firmware debug1**. This is debug register used by GNSS firmware to store temporary information.

10.2 PNP Device descriptors

Our SoC implementaion provides capability to read in real-time information about mapped devices. Such information is packed into special device descriptors. Now we can provide 3 types of descriptors:

- Master device descriptor
- Slave device descriptor
- Custom device descriptor

All descriptors mapped sequentually starting from 0xFFFFF040. Each descriptor implements field 'size' in Bytes that specifies offset to the next mapped descriptor.

Master device descriptor

Bits	Description
[7:0]	**Descriptor Size.** Read Only value specifies size in Bytes of the current descriptor. This value should be used as offset to the next descriptor. Master descriptor size is hardwired to PNP_CFG_MASTE↩ R_DESCR_BYTES value (8'h08).
[9:8]	**Descriptor Type.** Master descriptor type is hardwired to PNP_CFG_TYPE_MASTER value (2'b01).
[31:10]	**Reserved.**
[47:32]	**Device ID.** Unique Master identificator.
[63:48]	**Vendor ID.** Unique Vendor identificator.

Slave device descriptor

Bits	Description
[7:0]	**Descriptor Size.** Read Only value specifies size in Bytes of the current descriptor. This value should be used as offset to the next descriptor. Slave descriptor size is hardwired to PNP_CFG_↩ SLAVE_DESCR_BYTES value (8'h10).
[9:8]	**Descriptor Type.** Slave descriptor type is hardwired to PNP_CFG_TYPE_SLAVE value (2'b10).
[15:10]	**Reserved.**
[23:16]	**IRQ ID.** Interrupt line index assigned to the device.
[31:24]	**Reserved.**
[47:32]	**Device ID.** Unique Master identificator.
[63:48]	**Vendor ID.** Unique Vendor identificator.
[75:64]	**zero.** Hardwired to X"000".
[95:76]	**Base Address Mask** specifies the memory region allocated for the device.
[107:96]	**zero.** Hardwired to X"000".
[127:108]	**Base Address** value of the device.