



RISC-V®

# The NEORV32 RISC-V Processor

## *Datasheet*

Version v1.9.9



## Documentation

The online documentation of the project (a.k.a. the **data sheet**) is available on GitHub-pages: <https://stnolting.github.io/neorv32/>

The online documentation of the **software framework** is also available on GitHub-pages: <https://stnolting.github.io/neorv32/sw/files.html>

# Table of Contents

|   |    |
|---|----|
| 1. Overview .....   | 6  |
| 1.1. Rationale .....  | 7  |
| 1.2. Project Key Features .....   | 9  |
| 1.3. Project Folder Structure .....   | 12 |
| 1.4. VHDL File Hierarchy .....  | 13 |
| 1.5. FPGA Implementation Results .....  | 15 |
| 1.6. CPU Performance .....  | 18 |
| 2. NEORV32 Processor (SoC) .....  | 19 |
| 2.1. Processor Top Entity - Signals .....                                     | 21 |
| 2.2. Processor Top Entity - Generics .....                                    | 25 |
| 2.3. Processor Clocking .....   | 31 |
| 2.3.1. Clock Gating .....   | 31 |
| 2.3.2. Peripheral Clocks .....  | 31 |
| 2.4. Processor Reset .....  | 33 |
| 2.5. Processor Interrupts .....   | 34 |
| 2.5.1. RISC-V Standard Interrupts .....                                       | 34 |
| 2.5.2. NEORV32-Specific Fast Interrupt Requests .....                         | 34 |
| 2.6. Address Space .....  | 36 |
| 2.6.1. Bus System .....   | 37 |
| 2.6.2. Bus Gateway .....  | 38 |
| 2.6.3. Reservation Set Controller .....                                       | 39 |
| 2.6.4. IO Switch .....  | 41 |
| 2.6.5. Boot Configuration .....   | 41 |
| 2.7. Processor-Internal Modules .....   | 44 |
| 2.7.1. Instruction Memory (IMEM) .....  | 45 |
| 2.7.2. Data Memory (DMEM) .....   | 47 |
| 2.7.3. Bootloader ROM (BOOTROM) .....   | 48 |
| 2.7.4. Processor-Internal Instruction Cache (iCACHE) .....                    | 49 |
| 2.7.5. Processor-Internal Data Cache (dCACHE) .....                           | 50 |
| 2.7.6. Direct Memory Access Controller (DMA) .....                            | 52 |
| 2.7.7. Processor-External Bus Interface (XBUS) .....                          | 56 |
| 2.7.8. Stream Link Interface (SLINK) .....                                    | 60 |
| 2.7.9. General Purpose Input and Output Port (GPIO) .....                     | 64 |
| 2.7.10. Cyclic Redundancy Check (CRC) .....                                   | 65 |
| 2.7.11. Watchdog Timer (WDT) .....  | 67 |
| 2.7.12. Machine System Timer (MTIME) .....                                    | 70 |
| 2.7.13. Primary Universal Asynchronous Receiver and Transmitter (UART0) ..... | 71 |

|   |     |
|---|-----|
| 2.7.14. Secondary Universal Asynchronous Receiver and Transmitter (UART1) . . . . . | 76  |
| 2.7.15. Serial Peripheral Interface Controller (SPI) . . . . .                      | 77  |
| 2.7.16. Serial Data Interface Controller (SDI) . . . . .                            | 81  |
| 2.7.17. Two-Wire Serial Interface Controller (TWI) . . . . .                        | 84  |
| 2.7.18. One-Wire Serial Interface Controller (ONEWIRE) . . . . .                    | 88  |
| 2.7.19. Pulse-Width Modulation Controller (PWM) . . . . .                           | 93  |
| 2.7.20. True Random-Number Generator (TRNG) . . . . .                               | 95  |
| 2.7.21. Custom Functions Subsystem (CFS) . . . . .                                  | 97  |
| 2.7.22. Smart LED Interface (NEOLED) . . . . .                                      | 99  |
| 2.7.23. External Interrupt Controller (XIRQ) . . . . .                              | 104 |
| 2.7.24. General Purpose Timer (GPTMR) . . . . .                                     | 106 |
| 2.7.25. Execute In Place Module (XIP) . . . . .                                     | 108 |
| 2.7.26. System Configuration Information Memory (SYSINFO) . . . . .                 | 113 |
| 3. NEORV32 Central Processing Unit (CPU) . . . . .                                  | 117 |
| 3.1. RISC-V Compatibility . . . . .   | 117 |
| 3.2. CPU Top Entity - Signals . . . . .   | 119 |
| 3.3. CPU Top Entity - Generics . . . . .  | 120 |
| 3.4. Architecture . . . . .   | 121 |
| 3.4.1. CPU Register File . . . . .  | 121 |
| 3.4.2. CPU Arithmetic Logic Unit . . . . .  | 122 |
| 3.4.3. CPU Bus Unit . . . . .   | 123 |
| 3.4.4. CPU Control Unit . . . . .   | 123 |
| 3.4.5. Sleep Mode . . . . .   | 124 |
| 3.4.6. Full Virtualization . . . . .  | 124 |
| 3.5. Bus Interface . . . . .  | 126 |
| 3.5.1. Bus Interface Protocol . . . . .   | 126 |
| 3.5.2. Atomic Accesses . . . . .  | 127 |
| 3.6. Instruction Sets and Extensions . . . . .                                      | 130 |
| 3.6.1. A ISA Extension . . . . .  | 131 |
| 3.6.2. B ISA Extension . . . . .  | 132 |
| 3.6.3. C ISA Extension . . . . .  | 132 |
| 3.6.4. E ISA Extension . . . . .  | 133 |
| 3.6.5. I ISA Extension . . . . .  | 133 |
| 3.6.6. M ISA Extension . . . . .  | 134 |
| 3.6.7. U ISA Extension . . . . .  | 134 |
| 3.6.8. X ISA Extension . . . . .  | 135 |
| 3.6.9. Zifencei ISA Extension . . . . .   | 135 |
| 3.6.10. Zfinx ISA Extension . . . . .   | 135 |
| 3.6.11. Zicntr ISA Extension . . . . .  | 136 |

|  |     |
|--|-----|
| 3.6.12. <i>Zicond</i> ISA Extension . . . . .                | 136 |
| 3.6.13. <i>Zicsr</i> ISA Extension . . . . .                 | 136 |
| 3.6.14. <i>Zihpm</i> ISA Extension . . . . .                 | 137 |
| 3.6.15. <i>Zmmul</i> - ISA Extension . . . . .               | 137 |
| 3.6.16. <i>Zxfcuh</i> ISA Extension . . . . .                | 137 |
| 3.6.17. <i>Smpmp</i> ISA Extension . . . . .                 | 137 |
| 3.6.18. <i>Sdext</i> ISA Extension . . . . .                 | 138 |
| 3.6.19. <i>Sdtrig</i> ISA Extension . . . . .                | 138 |
| 3.7. Custom Functions Unit (CFU) . . . . .                   | 139 |
| 3.7.1. CFU Instruction Formats . . . . .                     | 139 |
| 3.7.2. Using Custom Instructions in Software . . . . .       | 142 |
| 3.7.3. CFU Control and Status Registers (CFU-CSRs) . . . . . | 143 |
| 3.7.4. Custom Instructions Hardware . . . . .                | 143 |
| 3.8. Control and Status Registers (CSRs) . . . . .           | 145 |
| 3.8.1. Floating-Point CSRs . . . . .                         | 150 |
| 3.8.2. Machine Trap Setup CSRs . . . . .                     | 152 |
| 3.8.3. Machine Trap Handling CSRs . . . . .                  | 156 |
| 3.8.4. Machine Configuration CSRs . . . . .                  | 159 |
| 3.8.5. Machine Physical Memory Protection CSRs . . . . .     | 160 |
| 3.8.6. Custom Functions Unit (CFU) CSRs . . . . .            | 162 |
| 3.8.7. (Machine) Counter and Timer CSRs . . . . .            | 163 |
| 3.8.8. Hardware Performance Monitors (HPM) CSRs . . . . .    | 165 |
| 3.8.9. Machine Counter Setup CSRs . . . . .                  | 169 |
| 3.8.10. Machine Information CSRs . . . . .                   | 170 |
| 3.8.11. NEORV32-Specific CSRs . . . . .                      | 172 |
| 3.8.12. Traps, Exceptions and Interrupts . . . . .           | 174 |
| 4. Software Framework . . . . .                              | 178 |
| 4.1. Compiler Toolchain . . . . .                            | 179 |
| 4.2. Core Libraries . . . . .                                | 180 |
| 4.3. Application Makefile . . . . .                          | 182 |
| 4.3.1. Makefile Targets . . . . .                            | 182 |
| 4.3.2. Makefile Configuration . . . . .                      | 183 |
| 4.3.3. Default Compiler Flags . . . . .                      | 185 |
| 4.3.4. Custom (Compiler) Flags . . . . .                     | 185 |
| 4.4. Executable Image Format . . . . .                       | 187 |
| 4.4.1. Linker Script . . . . .                               | 187 |
| 4.4.2. RAM Layout . . . . .                                  | 188 |
| 4.4.3. C Standard Library . . . . .                          | 190 |
| 4.4.4. Executable Image Generator . . . . .                  | 190 |

|  |     |
|--|-----|
| 4.4.5. Start-Up Code (crt0) . . . . .                | 191 |
| 4.5. Bootloader . . . . .                            | 193 |
| 4.5.1. Bootloader SoC/CPU Requirements . . . . .     | 193 |
| 4.5.2. Bootloader Flash Requirements . . . . .       | 194 |
| 4.5.3. Bootloader Console. . . . .                   | 194 |
| 4.5.4. Auto Boot Sequence . . . . .                  | 197 |
| 4.5.5. Bootloader Error Codes . . . . .              | 197 |
| 4.6. NEORV32 Runtime Environment . . . . .           | 199 |
| 4.6.1. RTE Operation . . . . .                       | 199 |
| 4.6.2. Using the RTE . . . . .                       | 199 |
| 4.6.3. Default RTE Trap Handlers . . . . .           | 202 |
| 4.6.4. Application Context Handling . . . . .        | 204 |
| 5. On-Chip Debugger (OCD) . . . . .                  | 205 |
| 5.1. Debug Transport Module (DTM) . . . . .          | 207 |
| 5.2. Debug Module (DM) . . . . .                     | 209 |
| 5.2.1. DM Registers . . . . .                        | 209 |
| 5.2.2. DM CPU Access . . . . .                       | 214 |
| 5.3. CPU Debug Mode . . . . .                        | 217 |
| 5.3.1. CPU Debug Mode CSRs . . . . .                 | 218 |
| 5.4. Trigger Module . . . . .                        | 221 |
| 5.4.1. Trigger Module CSRs . . . . .                 | 221 |
| 6. Legal . . . . .                                   | 224 |
| About . . . . .                                      | 224 |
| License . . . . .                                    | 224 |
| Proprietary Notice . . . . .                         | 225 |
| Disclaimer . . . . .                                 | 225 |
| Limitation of Liability for External Links . . . . . | 225 |
| Citing . . . . .                                     | 225 |
| Acknowledgments . . . . .                            | 226 |

# Chapter 1. Overview

The NEORV32 RISC-V Processor is an open-source RISC-V compatible processor system that is intended as **ready-to-go** auxiliary processor within a larger SoC designs or as stand-alone custom / customizable microcontroller.

The system is highly configurable and provides optional common peripherals like embedded memories, timers, serial interfaces, general purpose IO ports and an external bus interface to connect custom IP like memories, NoCs and other peripherals. On-line and in-system debugging is supported by an OpenOCD/gdb compatible on-chip debugger accessible via JTAG.

Special focus is paid on **execution safety** to provide defined and predictable behavior at any time. Therefore, the CPU ensures that all memory access are acknowledged and no invalid/malformed instructions are executed. Whenever an unexpected situation occurs, the application code is informed via hardware exceptions.

The software framework of the processor comes with application makefiles, software libraries for all CPU and processor features, a bootloader, a runtime environment and several example programs - including a port of the CoreMark MCU benchmark and the official RISC-V architecture test suite. RISC-V GCC is used as default toolchain ([prebuilt toolchains are also provided](#)).

Check out the processor's [online User Guide](#) that provides hands-on tutorials to get you started.

## Structure

2. [NEORV32 Processor \(SoC\)](#)
3. [NEORV32 Central Processing Unit \(CPU\)](#)
4. [Software Framework](#)
5. [On-Chip Debugger \(OCD\)](#)
6. [Legal](#)

## Annotations



Warning



Important



Note



Tip

## 1.1. Rationale

### Why did you make this?

Processor and CPU architecture designs are fascinating things: they are the magic frontier where software meets hardware. This project started as something like a *journey* into this magic realm to understand how things actually work down on this very low level and evolved over time to a capable system on chip.

But there is more: when I started to dive into the emerging RISC-V ecosystem I felt overwhelmed by the complexity. As a beginner it is hard to get an overview - especially when you want to setup a minimal platform to tinker with... Which core to use? How to get the right toolchain? What features do I need? How does booting work? How do I create an actual executable? How to get that into the hardware? How to customize things? **Where to start???**

This project aims to provide a *simple to understand* and *easy to use* yet *powerful* and *flexible* platform that targets FPGA and RISC-V beginners as well as advanced users.

### Why a *soft-core* processor?

As a matter of fact soft-core processors *cannot* compete with discrete (like FPGA hard-macro) processors in terms of performance, energy efficiency and size. But they do fill a niche in FPGA design space: for example, soft-core processors allow to implement the *control flow part* of certain applications (e.g. communication protocol handling) using software like plain C. This provides high flexibility as software can be easily changed, re-compiled and re-uploaded again.

Furthermore, the concept of flexibility applies to all aspects of a soft-core processor. The user can add *exactly* the features that are required by the application: additional memories, custom interfaces, specialized co-processors and even user-defined instructions.

### Why RISC-V?



RISC-V is a free and open ISA enabling a new era of processor innovation through open standard collaboration.

— RISC-V International, <https://riscv.org/about/>

Open-source is a great thing! While open-source has already become quite popular in *software*, hardware-focused projects still need to catch up. Admittedly, there has been quite a development, but mainly in terms of *platforms* and *applications* (so schematics, PCBs, etc.). Although processors and CPUs are the heart of almost every digital system, having a true open-source silicon is still a rarity. RISC-V aims to change that - and even it is *just one approach*, it helps paving the road for future development.

Furthermore, I highly appreciate the community aspect of RISC-V. The ISA and everything beyond is developed in direct contact with the community: this includes businesses and professionals but also hobbyist, amateurs and people that are just curious. Everyone can join discussions and contribute to RISC-V in their very own way.

Finally, I really like the RISC-V ISA itself. It aims to be a clean, orthogonal and "intuitive" ISA that resembles with the basic concepts of *RISC*: simple yet effective.

## Yet another RISC-V core? What makes it special?

The NEORV32 is not based on another RISC-V core. It was build entirely from ground up (just following the official ISA specs). The project does not intend to replace certain RISC-V cores or just beat existing ones like [VexRISC](#) in terms of performance or [SERV](#) in terms of size. It was build having a different design goal in mind.

The project aims to provide *another option* in the RISC-V / soft-core design space with a different performance vs. size trade-off and a different focus: *embrace* concepts like documentation, platform-independence / portability, RISC-V compatibility, *extensibility & customization* and *ease of use* (see the [Project Key Features](#) below).

Furthermore, the NEORV32 pays special focus on *execution safety* using [Full Virtualization](#). The CPU aims to provide fall-backs for *everything that could go wrong*. This includes malformed instruction words, privilege escalations and even memory accesses that are checked for address space holes and deterministic response times of memory-mapped devices. Precise exceptions allow a defined and fully-synchronized state of the CPU at every time and in every situation.

## A multi-cycle architecture?!?!

Most mainstream CPUs out there are pipelined architectures to increase throughput. In contrast, most CPUs used for teaching are single-cycle designs since they are probably the most easiest to understand. But what about the multi-cycle architectures?

In terms of energy, throughput, area and maximal clock frequency multi-cycle architectures are somewhere in between single-cycle and fully-pipelined designs: they provide higher throughput and clock speed when compared to their single-cycle counterparts while having less hardware complexity (= area) than a fully-pipelined designs. I decided to use the multi-cycle approach because of the following reasons:

- Multi-cycle architecture are quite small! There is no need for pipeline hazard detection and resolution logic (e.g. forwarding). Furthermore, you can "re-use" parts of the core to do several tasks (e.g. the ALU is used for the actual data processing, but also for address generation, branch condition check and branch target computation).
- Single-cycle architectures require memories that can be read asynchronously - a thing that is not feasible to implement in real world applications (i.e. FPGA block RAM is entirely synchronous). Furthermore, such design usually have a very long critical path tremendously reducing maximal operating frequency.
- Pipelined designs increase performance by having several instruction "in fly" at the same time.

But this also means there is some kind of "out-of-order" behavior: if an instruction at the end of the pipeline causes an exception all the instructions in earlier stages have to be invalidated. Potential architecture state changes have to be made *undone* requiring additional (exception-handling) logic. In a multi-cycle architecture this situation cannot occur because only a single instruction is "in fly" at a time.

- Having only a single instruction in fly does not only reduce hardware costs, it also simplifies simulation/verification/debugging, state preservation/restoring during exceptions and extensibility (no need to care about pipeline hazards) - but of course at the cost of reduced throughput.

To counteract the loss of performance implied by a *pure* multi-cycle architecture, the NEORV32 CPU uses a *mixed* approach: instruction fetch (front-end) and instruction execution (back-end) are decoupled to operate independently of each other. Data is interchanged via a queue building a simple 2-stage pipeline. Each "pipeline" stage in terms is implemented as multi-cycle architecture to simplify the hardware and to provide *precise* state control (e.g. during exceptions).

## 1.2. Project Key Features

### Project

- all-in-one package: **CPU + SoC + Software Framework & Tooling**
- completely described in behavioral, platform-independent VHDL - no vendor- or technology-specific primitives, attributes, macros, libraries, etc. are used at all
- all-Verilog "version" available (auto-generated netlist)
- extensive configuration options for adapting the processor to the requirements of the application
- highly extensible hardware - on CPU, SoC and system level
- aims to be as small as possible while being as RISC-V-compliant as possible - with a reasonable area-vs-performance trade-off
- FPGA friendly (e.g. all internal memories can be mapped to block RAM - including the register file)
- optimized for high clock frequencies to ease timing closure and integration
- from zero to "*hello world!*" - completely open source and documented
- easy to use even for FPGA/RISC-V starters – intended to *work out of the box*

### NEORV32 CPU (the core)

- 32-bit RISC-V CPU
- fully compatible to the RISC-V ISA specs. - checked by the [official RISCOF architecture tests](#)
- base ISA + privileged ISA + several optional standard and custom ISA extensions
- option to add user-defined RISC-V instructions as custom ISA extension

- rich set of customization options (ISA extensions, design goal: performance / area / energy, tuning options, ...)
- [Full Virtualization](#) capabilities to increase execution safety
- official RISC-V open source architecture ID

## NEORV32 Processor (the SoC)

- highly-configurable full-scale microcontroller-like processor system
- based on the NEORV32 CPU
- optional standard serial interfaces (UART, TWI, SPI (host and device), 1-Wire)
- optional timers and counters (watchdog, system timer)
- optional general purpose IO and PWM; a native NeoPixel(c)-compatible smart LED interface
- optional embedded memories and caches for data, instructions and bootloader
- optional external memory interface for custom connectivity
- optional execute in-place (XIP) module to execute code directly from an external SPI flash
- optional DMA controller for CPU-independent data transfers
- optional CRC module to check data integrity
- on-chip debugger compatible with OpenOCD and gdb including hardware trigger module

## Software framework

- GCC-based toolchain - [prebuilt toolchains available](#); application compilation based on GNU makefiles
- internal bootloader with serial user interface (via UART)
- core libraries and HAL for high-level usage of the provided functions and peripherals
- processor-specific runtime environment and several example programs
- doxygen-based documentation of the software framework; a deployed version is available at <https://stnolting.github.io/neorv32/sw/files.html>
- FreeRTOS port + demos available

## Extensibility and Customization

The NEORV32 processor is designed to ease customization and extensibility and provides several options for adding application-specific custom hardware modules and accelerators. The three most common options for adding custom on-chip modules are listed below.

- [Processor-External Bus Interface \(XBUS\)](#) to attach processor-external IP modules (memories and peripherals)
- [Custom Functions Subsystem \(CFS\)](#) for tightly-coupled processor-internal co-processors
- [Custom Functions Unit \(CFU\)](#) for custom RISC-V instructions



A more detailed comparison of the extension/customization options can be found in section [Adding Custom Hardware Modules](#) of the user guide.

## 1.3. Project Folder Structure

```
neorv32          - Project home folder  
|  
|-docs           - Project documentation  
| |-datasheet    - AsciiDoc sources for the NEORV32 data sheet  
| |-figures       - Figures and logos  
| |-references    - Data sheets and RISC-V specs  
| |-sources        - Sources for the images in 'figures/'  
| |-userguide     - AsciiDoc sources for the NEORV32 user guide  
|  
|-rtl            - VHDL sources  
| |-core          - Core sources of the CPU & SoC  
| | |-mem          - SoC-internal memories (default architectures)  
| |-processor_templates - Pre-configured SoC wrappers  
| |-system_integration - System wrappers for advanced connectivity  
| |-test_setups     - Minimal test setup "SoCs" used in the User Guide  
|  
|-sim            - Simulation files (see User Guide)  
|  
|-sw              - Software framework  
| |-bootloader     - Sources of the processor-internal bootloader  
| |-common         - Linker script, crt0.S start-up code and central makefile  
| |-example        - Example programs for the core and the SoC modules  
| |-lib             - Processor core library  
| | |-include       - Header files (*.h)  
| | |-source        - Source files (*.c)  
| |-image_gen      - Helper program to generate NEORV32 executables  
| |-ocd_firmware   - Firmware for the on-chip debugger's "park loop"  
| |-openocd        - OpenOCD configuration files  
| |-svd             - Processor system view description file (CMSIS-SVD)
```

## 1.4. VHDL File Hierarchy

All necessary VHDL hardware description files are located in the project's `rtl/core` folder. The top entity of the entire processor including all the required configuration generics is `neorv32_top.vhd`.

### *Compile Order*



Most of the RTL sources use **entity instantiation**. Hence, the RTL compile order might be relevant. The list below shows the hierarchical compile order starting at the top.

### *VHDL Library*



All core VHDL files from the list below have to be assigned to a **new library** named `neorv32`.

|  |  |
|--|--|
| <code>neorv32_package.vhd</code>           | - Processor/CPU main VHDL package file                 |
| <code>neorv32_clockgate.vhd</code>         | - Generic clock gating switch                          |
| <code>neorv32_fifo.vhd</code>              | - Generic FIFO component                               |
|  |  |
| <code>neorv32_cpu_cp_bitmanip.vhd</code>   | - Bit-manipulation co-processor (B ext.)               |
| <code>neorv32_cpu_cp_cfu.vhd</code>        | - Custom instructions co-processor (Zxfcuh ext.)       |
| <code>neorv32_cpu_cp_cond.vhd</code>       | - Integer conditional operations (Zicond ext.)         |
| <code>neorv32_cpu_cp_fpu.vhd</code>        | - Floating-point co-processor (Zfinx ext.)             |
| <code>neorv32_cpu_cp_shifter.vhd</code>    | - Bit-shift co-processor (base ISA)                    |
| <code>neorv32_cpu_cp_muldiv.vhd</code>     | - Mul/Div co-processor (M ext.)                        |
|  |  |
| <code>neorv32_cpu_alu.vhd</code>           | - Arithmetic/logic unit                                |
| <code>neorv32_cpu_pmp.vhd</code>           | - Physical memory protection unit (Smpmp ext.)         |
| <code>neorv32_cpu_lsu.vhd</code>           | - Load/store unit                                      |
| <code>neorv32_cpu_decompressor.vhd</code>  | - Compressed instructions decoder (C ext.)             |
| <code>neorv32_cpu_control.vhd</code>       | - CPU control, exception system and CSRs               |
| <code>neorv32_cpu_regfile.vhd</code>       | - Data register file                                   |
|  |  |
| <code>neorv32_cpu.vhd</code>               | - NEORV32 CPU TOP ENTITY                               |
|  |  |
| <code>mem/neorv32_dmem.default.vhd</code>  | - *Default* data memory (architecture-only)            |
| <code>mem/neorv32_imem.default.vhd</code>  | - *Default* instruction memory (architecture-only)     |
|  |  |
| <code>neorv32_bootloader_image.vhd</code>  | - Bootloader ROM memory image                          |
| <code>neorv32_boot_rom.vhd</code>          | - Bootloader ROM                                       |
|  |  |
| <code>neorv32_application_image.vhd</code> | - IMEM application initialization image                |
| <code>neorv32_imem.entity.vhd</code>       | - Processor-internal instruction memory (entity-only!) |
|  |  |
| <code>neorv32_cfs.vhd</code>               | - Custom functions subsystem                           |
| <code>neorv32_crc.vhd</code>               | - Cyclic redundancy check unit                         |

|                         |   |
|-------------------------|---|
| neorv32_cache.vhd       | - Generic cache module                              |
| neorv32_debug_dm.vhd    | - on-chip debugger: debug module                    |
| neorv32_debug_dtm.vhd   | - on-chip debugger: debug transfer module           |
| neorv32_dma.vhd         | - Direct memory access controller                   |
| neorv32_dmem.entity.vhd | - Processor-internal data memory (entity-only!)     |
| neorv32_gpio.vhd        | - General purpose input/output port unit            |
| neorv32_gptmr.vhd       | - General purpose 32-bit timer                      |
| neorv32_intercon.vhd    | - SoC bus infrastructure                            |
| neorv32_mtime.vhd       | - Machine system timer                              |
| neorv32_neoled.vhd      | - NeoPixel (TM) compatible smart LED interface      |
| neorv32_onewire.vhd     | - One-Wire serial interface controller              |
| neorv32_pwm.vhd         | - Pulse-width modulation controller                 |
| neorv32_sdi.vhd         | - Serial data interface controller (SPI device)     |
| neorv32_slink.vhd       | - Stream link interface                             |
| neorv32_spi.vhd         | - Serial peripheral interface controller (SPI host) |
| neorv32_sysinfo.vhd     | - System configuration information memory           |
| neorv32_trng.vhd        | - True random number generator                      |
| neorv32_twi.vhd         | - Two wire serial interface controller              |
| neorv32_uart.vhd        | - Universal async. receiver/transmitter             |
| neorv32_wdt.vhd         | - Watchdog timer                                    |
| neorv32_xbus.vhd        | - External (Wishbone) bus interface gateways        |
| neorv32_xip.vhd         | - Execute in place module                           |
| neorv32_xirq.vhd        | - External interrupt controller                     |
|                         |   |
| neorv32_top.vhd         | - NEORV32 PROCESSOR TOP ENTITY                      |



The processor-internal instruction and data memories (IMEM and DMEM) are split into two design files each: a plain entity definition (`neorv32_*mem.entity.vhd`) and the actual architecture definition (`mem/neorv32_*mem.default.vhd`). The `*.default.vhd` architecture definitions from `rtl/core/mem` provide a *generic* and *platform independent* memory design (inferring embedded memory blocks). You can replace/modify the architecture source file in order to use platform-specific features (like advanced memory resources) or to improve technology mapping and/or timing.

## 1.5. FPGA Implementation Results

This section shows **exemplary** FPGA implementation results for the NEORV32 CPU and NEORV32 Processor modules.



The results are generated by manual synthesis runs. Hence, they might not represent the latest version of the processor.

### CPU

|              |  |
|--------------|--|
| HW version:  | <a href="#">1.7.8.5</a>  |
| Top entity:  | <a href="#">rtl/core/neorv32_cpu.vhd</a>   |
| FPGA:        | Intel Cyclone IV E <a href="#">EP4CE22F17C6</a>  |
| Toolchain:   | Quartus Prime Lite 21.1  |
| Constraints: | <b>no timing constraints</b> , "balanced optimization", $f_{max}$ from " <i>Slow 1200mV OC Model</i> " |

| CPU ISA Configuration   | LEs  | FFs  | MEM bits | DSPs | $f_{max}$ |
|---|------|------|----------|------|-----------|
| <a href="#">rv32i_Zicsr</a>                                     | 1223 | 607  | 1024     | 0    | 130 MHz   |
| <a href="#">rv32i_Zicsr_Zicntr</a>                              | 1578 | 773  | 1024     | 0    | 130 MHz   |
| <a href="#">rv32im_Zicsr_Zicntr</a>                             | 2087 | 983  | 1024     | 0    | 130 MHz   |
| <a href="#">rv32imc_Zicsr_Zicntr</a>                            | 2338 | 992  | 1024     | 0    | 130 MHz   |
| <a href="#">rv32imcb_Zicsr_Zicntr</a>                           | 3175 | 1247 | 1024     | 0    | 130 MHz   |
| <a href="#">rv32imcbu_Zicsr_Zicntr</a>                          | 3186 | 1254 | 1024     | 0    | 130 MHz   |
| <a href="#">rv32imcbu_Zicsr_Zicntr_Zifencei</a>                 | 3187 | 1254 | 1024     | 0    | 130 MHz   |
| <a href="#">rv32imcbu_Zicsr_Zicntr_Zifencei_Zfinx</a>           | 4450 | 1906 | 1024     | 7    | 123 MHz   |
| <a href="#">rv32imcbu_Zicsr_Zicntr_Zifencei_Zfinx_DebugMode</a> | 4825 | 2018 | 1024     | 7    | 123 MHz   |



*Goal-Driven Optimization*

The CPU provides further options to reduce the area footprint or to increase

performance. See section [Processor Top Entity - Generics](#) for more information. Also, take a look at the User Guide section [Application-Specific Processor Configuration](#).

## Processor - Modules

|              |  |
|--------------|--|
| HW version:  | <a href="#">1.8.6.7</a>                                |
| Top entity:  | <a href="#">rtl/core/neorv32_top.vhd</a>               |
| FPGA:        | Intel Cyclone IV E <a href="#">EP4CE22F17C6</a>        |
| Toolchain:   | Quartus Prime Lite 21.1                                |
| Constraints: | <b>no timing constraints</b> , "balanced optimization" |

*Table 1. Hardware utilization by processor module*

| Module            | Description   | LEs | FFs | MEM bits | DSPs |
|-------------------|---|-----|-----|----------|------|
| BOOT ROM          | Bootloader ROM (4kB)  | 2   | 2   | 32768    | 0    |
| Bus switch (core) | <i>SoC bus infrastructure</i>                               | 28  | 15  | 0        | 0    |
| Bus switch (DMA)  | <i>SoC bus infrastructure</i>                               | 159 | 9   | 0        | 0    |
| CFS               | Custom functions subsystem (depends on custom design logic) | -   | -   | -        | -    |
| CRC               | Cyclic redundancy check unit                                | 130 | 117 | 0        | 0    |
| dCACHE            | Data cache (4 blocks, 64 bytes per block)                   | 300 | 167 | 2112     | 0    |
| DM                | On-chip debugger - debug module                             | 377 | 241 | 0        | 0    |
| DTM               | On-chip debugger - debug transfer module (JTAG)             | 262 | 220 | 0        | 0    |
| DMA               | Direct memory access controller                             | 365 | 291 | 0        | 0    |
| DMEM              | Processor-internal data memory (8kB)                        | 6   | 2   | 65536    | 0    |
| Gateway           | <i>SoC bus infrastructure</i>                               | 215 | 91  | 0        | 0    |
| GPIO              | General purpose input/output ports                          | 102 | 98  | 0        | 0    |
| GPTMR             | General Purpose Timer                                       | 150 | 105 | 0        | 0    |
| IO Switch         | <i>SoC bus infrastructure</i>                               | 217 | 0   | 0        | 0    |
| iCACHE            | Instruction cache (2x4 blocks, 64 bytes per block)          | 458 | 296 | 4096     | 0    |
| IMEM              | Processor-internal instruction memory (16kB)                | 7   | 2   | 131072   | 0    |
| MTIME             | Machine system timer  | 307 | 166 | 0        | 0    |
| NEOLED            | Smart LED Interface (NeoPixel/WS28128) (FIFO_depth=1)       | 171 | 129 | 0        | 0    |

| Module          | Description   | LEs | FFs | MEM bits | DSPs |
|-----------------|---|-----|-----|----------|------|
| ONEWIRE         | 1-wire interface  | 105 | 77  | 0        | 0    |
| PWM             | Pulse_width modulation controller (4 channels)                    | 91  | 81  | 0        | 0    |
| Reservation Set | Reservation set controller for LR/SC instructions                 | 52  | 33  | 0        | 0    |
| SDI             | Serial data interface   | 103 | 77  | 512      | 0    |
| SLINK           | Stream link interface (RX/TX FIFO depth=32)                       | 96  | 73  | 2048     | 0    |
| SPI             | Serial peripheral interface                                       | 137 | 97  | 1024     | 0    |
| SYSINFO         | System configuration information memory                           | 11  | 11  | 0        | 0    |
| TRNG            | True random number generator                                      | 140 | 108 | 512      | 0    |
| TWI             | Two-wire interface  | 93  | 64  | 0        | 0    |
| UART0,<br>UART1 | Universal asynchronous receiver/transmitter 0/1<br>(FIFO_depth=1) | 222 | 142 | 1024     | 0    |
| WDT             | Watchdog timer  | 107 | 89  | 0        | 0    |
| WISHBONE        | External memory interface   | 122 | 112 | 0        | 0    |
| XIP             | Execute in place module   | 369 | 276 | 0        | 0    |
| XIRQ            | External interrupt controller (4 channels)                        | 35  | 29  | 0        | 0    |

## 1.6. CPU Performance

The performance of the NEORV32 was tested and evaluated using the [Core Mark CPU benchmark](#). The according sources can be found in the `sw/example/coremark` folder. The resulting CoreMark score is defined as CoreMark iterations per second per MHz.

*Table 2. Configuration*

|                 |   |
|-----------------|---|
| HW version:     | 1.5.7.10  |
| Hardware:       | 32kB int. IMEM, 16kB int. DMEM, no caches, 100MHz clock                 |
| CoreMark:       | 2000 iterations, MEM_METHOD is MEM_STACK                                |
| Compiler:       | RISCV32-GCC 10.2.0 (compiled with <code>march=rv32i mabi=ilp32</code> ) |
| Compiler flags: | default but with <code>-O3</code> , see makefile                        |

*Table 3. CoreMark results*

| CPU   | CoreMark Score | CoreMark s/MHz | Average CPI |
|---|----------------|----------------|-------------|
| <i>small</i> ( <code>rv32i_Zicsr_Zifencei</code> )                        | 33.89          | 0.3389         | 4.04        |
| <i>medium</i> ( <code>rv32imc_Zicsr_Zifencei</code> )                     | 62.50          | 0.6250         | 5.34        |
| <i>performance</i> ( <code>rv32imc_Zicsr_Zifencei</code> + perf. options) | 95.23          | 0.9523         | 3.54        |

The NEORV32 CPU is based on a multi-cycle architecture. Each instruction is executed in a sequence of several consecutive micro operations. The average CPI (cycles per instruction) depends on the instruction mix of a specific applications and also on the available CPU extensions. More information regarding the execution time of each implemented instruction can be found in section [Instruction Sets and Extensions](#).

# Chapter 2. NEORV32 Processor (SoC)

The NEORV32 Processor is based on the NEORV32 CPU. Together with common peripheral interfaces and embedded memories it provides a RISC-V-based full-scale microcontroller-like SoC platform.

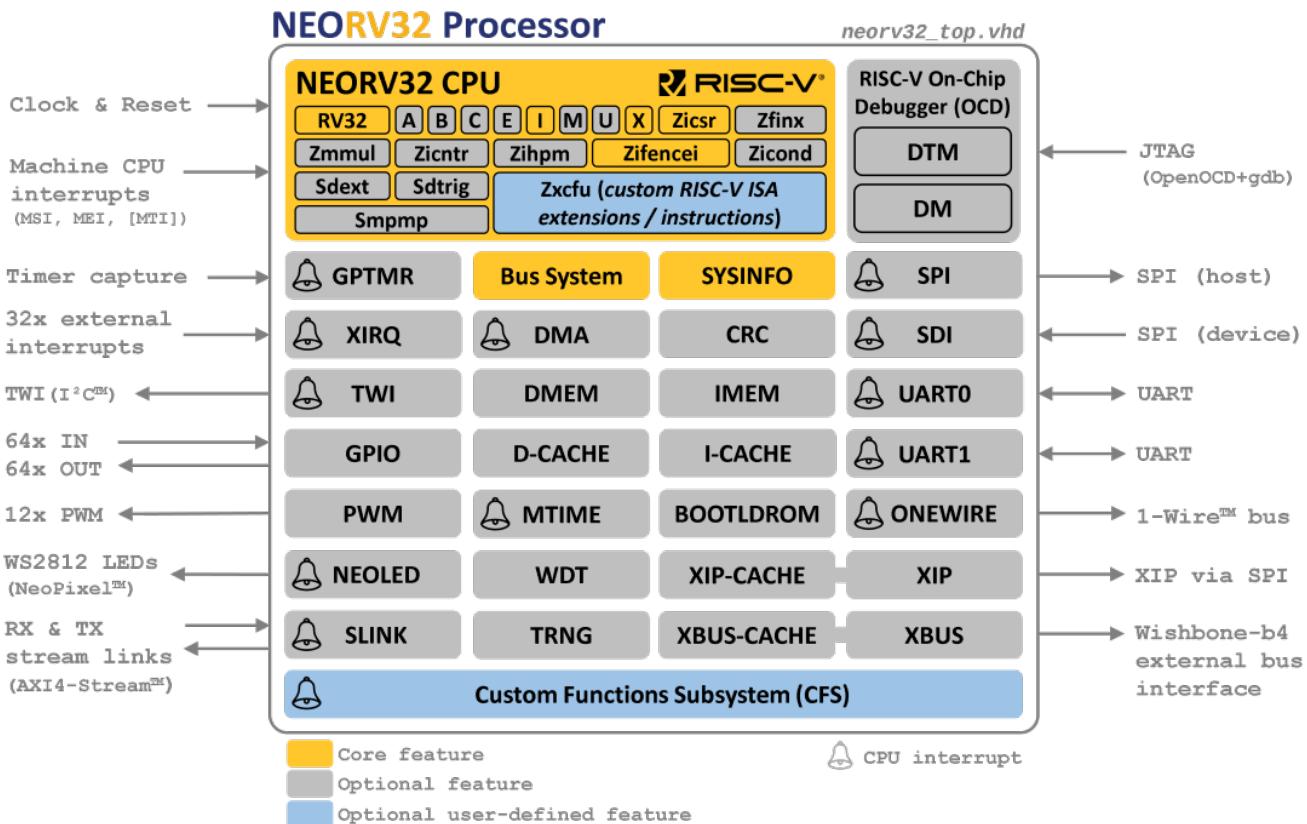


Figure 1. The NEORV32 Processor (Block Diagram)

## Section Structure

- Processor Top Entity - Signals and Processor Top Entity - Generics
- Processor Clocking and Processor Reset
- Processor Interrupts
- Address Space and Boot Configuration
- Processor-Internal Modules

## Key Features

- optional* processor-internal data and instruction memories (**DMEM/IMEM**)
- optional* caches (**iCACHE**, **dCACHE**, `<<_execute_in_place_module_xip,xipCACHE>>`, `<<_processor_external_bus_interface_xbus,xCACHE>>`)
- optional* internal bootloader (**BOOTROM**) with UART console & SPI flash boot option
- optional* machine system timer (**MTIME**), RISC-V-compatible

- *optional* two independent universal asynchronous receivers and transmitters (**UART0**, **UART1**) with optional hardware flow control (RTS/CTS)
- *optional* serial peripheral interface host controller (**SPI**) with 8 dedicated CS lines
- *optional* 8-bit serial data device interface (**SDI**)
- *optional* two wire serial interface controller (**TWI**), compatible to the I<sup>2</sup>C standard
- *optional* general purpose parallel IO port (**GPIO**), 64xOut, 64xIn
- *optional* 32-bit external bus interface, Wishbone b4 / AXI4-Lite compatible (**XBUS**)
- *optional* watchdog timer (**WDT**)
- *optional* PWM controller with up to 12 channels & 8-bit duty cycle resolution (**PWM**)
- *optional* ring-oscillator-based true random number generator (**TRNG**)
- *optional* custom functions subsystem for custom co-processor extensions (**CFS**)
- *optional* NeoPixel™/WS2812-compatible smart LED interface (**NEOLED**)
- *optional* external interrupt controller with up to 32 channels (**XIRQ**)
- *optional* general purpose 32-bit timer (**GPTMR**) with capture input
- *optional* execute in-place module (**XIP**)
- *optional* 1-wire serial interface controller (**ONEWIRE**), compatible to the 1-wire standard
- *optional* autonomous direct memory access controller (**DMA**)
- *optional* stream link interface (**SLINK**), AXI4-Stream compatible
- *optional* cyclic redundancy check unit (**CRC**)
- *optional* on-chip debugger with JTAG TAP (**OCD**)
- system configuration information memory to check HW configuration via software (**SYSINFO**)

## 2.1. Processor Top Entity - Signals

The following table shows all interface signals of the processor top entity ([rtl/core/neorv32\\_top.vhd](#)). All signals are of type `std_ulegic` or `std_ulegic_vector`, respectively.

### *Default Values of Inputs*



All *optional* input signals provide default values in case they are not explicitly assigned during instantiation. The weak driver strengths of VHDL ('L' and 'H') are used to model a pull-down or pull-up resistor.

### *Configurable Amount of Channels*



Some peripherals allow to configure the number of channels to-be-implemented by a generic (for example the number of PWM channels). The according input/output signals have a fixed sized regardless of the actually configured amount of channels. If less than the maximum number of channels is configured, only the LSB-aligned channels are used: in case of an *input port* the remaining bits/channels are left unconnected; in case of an *output port* the remaining bits/channels are hardwired to zero.

### *Tri-State Interfaces*



Some interfaces (like the TWI and the 1-Wire bus) require tri-state drivers in the designs top module.

Table 4. NEORV32 Processor Signal List

| Name   | Width | Direction | Default | Description  |
|--|-------|-----------|---------|--|
|  | h     | tion      | ult     |  |
| <b>Global Control (Processor Clocking and Processor Reset)</b> |       |           |         |  |
| <code>clk_i</code>   | 1     | in        | none    | global clock line, all registers triggering on rising edge |
| <code>rstn_i</code>  | 1     | in        | none    | global reset, asynchronous, <b>low-active</b>              |
| <b>JTAG Access Port for On-Chip Debugger (OCD)</b>             |       |           |         |  |
| <code>jtag_trst_i</code>                                       | 1     | in        | 'H'     | TAP reset, low-active (optional)                           |
| <code>jtag_tck_i</code>  | 1     | in        | 'L'     | serial clock   |
| <code>jtag_tdi_i</code>  | 1     | in        | 'L'     | serial data input  |
| <code>jtag_tdo_o</code>  | 1     | out       | -       | serial data output   |
| <code>jtag_tms_i</code>  | 1     | in        | 'L'     | mode select  |
| <b>Processor-External Bus Interface (XBUS)</b>                 |       |           |         |  |
| <code>xbus_adr_o</code>  | 32    | out       | -       | destination address  |
| <code>xbus_dat_o</code>  | 32    | out       | -       | read data  |
| <code>xbus_we_o</code>   | 1     | out       | -       | write enable ('0' = read transfer)                         |

| Name   | Width | Direction | Default | Description                             |
|--|-------|-----------|---------|---|
| <code>xbus_sel_o</code>  | 4     | out       | -       | byte enable                             |
| <code>xbus_stb_o</code>  | 1     | out       | -       | strobe                                  |
| <code>xbus_cyc_o</code>  | 1     | out       | -       | valid cycle                             |
| <code>xbus_dat_i</code>  | 32    | in        | 'L'     | write data                              |
| <code>xbus_ack_i</code>  | 1     | in        | 'L'     | transfer acknowledge                    |
| <code>xbus_err_i</code>  | 1     | in        | 'L'     | transfer error                          |
| <b>Stream Link Interface (SLINK)</b>                                     |       |           |         |   |
| <code>slink_rx_dat_i</code>  | 32    | in        | 'L'     | RX data                                 |
| <code>slink_rx_val_i</code>  | 1     | in        | 'L'     | RX data valid                           |
| <code>slink_rx_lst_i</code>  | 1     | in        | 'L'     | RX last element of stream               |
| <code>slink_rx_rdy_o</code>  | 1     | out       | -       | RX ready to receive                     |
| <code>slink_tx_dat_o</code>  | 32    | out       | -       | TX data                                 |
| <code>slink_tx_val_o</code>  | 1     | out       | -       | TX data valid                           |
| <code>slink_tx_lst_o</code>  | 1     | out       | -       | TX last element of stream               |
| <code>slink_tx_rdy_i</code>  | 1     | in        | 'L'     | TX allowed to send                      |
| <b>Execute In Place Module (XIP)</b>                                     |       |           |         |   |
| <code>xip_csn_o</code>   | 1     | out       | -       | chip select, low-active                 |
| <code>xip_clk_o</code>   | 1     | out       | -       | serial clock                            |
| <code>xip_dat_i</code>   | 1     | in        | 'L'     | serial data input                       |
| <code>xip_dat_o</code>   | 1     | out       | -       | serial data output                      |
| <b>General Purpose Input and Output Port (GPIO)</b>                      |       |           |         |   |
| <code>gpio_o</code>  | 64    | out       | -       | general purpose parallel output         |
| <code>gpio_i</code>  | 64    | in        | 'L'     | general purpose parallel input          |
| <b>Primary Universal Asynchronous Receiver and Transmitter (UART0)</b>   |       |           |         |   |
| <code>uart0_txd_o</code>   | 1     | out       | -       | serial transmitter                      |
| <code>uart0_rxd_i</code>   | 1     | in        | 'L'     | serial receiver                         |
| <code>uart0_rts_o</code>   | 1     | out       | -       | RX ready to receive new char            |
| <code>uart0_cts_i</code>   | 1     | in        | 'L'     | TX allowed to start sending, low-active |
| <b>Secondary Universal Asynchronous Receiver and Transmitter (UART1)</b> |       |           |         |   |
| <code>uart1_txd_o</code>   | 1     | out       | -       | serial transmitter                      |
| <code>uart1_rxd_i</code>   | 1     | in        | 'L'     | serial receiver                         |

| Name  | Width | Direction | Default | Description                              |
|---|-------|-----------|---------|--|
| uart1_rts_o   | 1     | out       | -       | RX ready to receive new char             |
| uart1_cts_i   | 1     | in        | 'L'     | TX allowed to start sending, low-active  |
| <b>Serial Peripheral Interface Controller (SPI)</b>   |       |           |         |  |
| spi_clk_o   | 1     | out       | -       | controller clock line                    |
| spi_dat_o   | 1     | out       | -       | serial data output                       |
| spi_dat_i   | 1     | in        | 'L'     | serial data input                        |
| spi_csn_o   | 8     | out       | -       | select (low-active)                      |
| <b>Serial Data Interface Controller (SDI)</b>         |       |           |         |  |
| sdi_clk_i   | 1     | in        | 'L'     | controller clock line                    |
| sdi_dat_o   | 1     | out       | -       | serial data output                       |
| sdi_dat_i   | 1     | in        | 'L'     | serial data input                        |
| sdi_csn_i   | 1     | in        | 'H'     | chip select, low-active                  |
| <b>Two-Wire Serial Interface Controller (TWI)</b>     |       |           |         |  |
| twi_sda_i   | 1     | in        | 'H'     | serial data line sense input             |
| twi_sda_o   | 1     | out       | -       | serial data line output (pull low only)  |
| twi_scl_i   | 1     | in        | 'H'     | serial clock line sense input            |
| twi_scl_o   | 1     | out       | -       | serial clock line output (pull low only) |
| <b>One-Wire Serial Interface Controller (ONEWIRE)</b> |       |           |         |  |
| onewire_i   | 1     | in        | 'H'     | 1-wire bus sense input                   |
| onewire_o   | 1     | out       | -       | 1-wire bus output (pull low only)        |
| <b>Pulse-Width Modulation Controller (PWM)</b>        |       |           |         |  |
| pwm_o   | 12    | out       | -       | pulse-width modulated channels           |
| <b>Custom Functions Subsystem (CFS)</b>               |       |           |         |  |
| cfs_in_i  | 32    | in        | 'L'     | custom CFS input signal conduit          |
| cfs_out_o   | 32    | out       | -       | custom CFS output signal conduit         |
| <b>Smart LED Interface (NEOLED)</b>                   |       |           |         |  |
| neoled_o  | 1     | out       | -       | asynchronous serial data output          |
| <b>Machine System Timer (MTIME)</b>                   |       |           |         |  |
| mtime_time_o  | 64    | out       | -       | MTIME system time output                 |
| <b>General Purpose Timer (GPTMR)</b>                  |       |           |         |  |
| gptmr_trig_i  | 1     | in        | 'L'     | timer capture input                      |

| Name  | Width | Direction | Default | Description  |
|---|-------|-----------|---------|--|
|   | h     | tion      | ult     |  |
| <b>External Interrupt Controller (XIRQ)</b>     |       |           |         |  |
| xirq_i  | 32    | in        | 'L'     | external interrupt requests                            |
| <b>RISC-V Machine-Mode Processor Interrupts</b> |       |           |         |  |
| mtime_irq_i                                     | 1     | in        | 'L'     | machine timer interrupt (RISC-V), high-level-active    |
| msw_irq_i                                       | 1     | in        | 'L'     | machine software interrupt (RISC-V), high-level-active |
| mext_irq_i                                      | 1     | in        | 'L'     | machine external interrupt (RISC-V), high-level-active |

## 2.2. Processor Top Entity - Generics

This section lists all configuration generics of the NEORV32 processor top entity ([rtl/neorv32\\_top.vhd](#)).

### *Customization*



The NEORV32 generics allow to configure the system according to your needs. The generics are used to control implementation of certain CPU extensions and peripheral modules and even allow to optimize the system for certain design goals like minimal area or maximum performance.

### *Default Values*



All *optional* configuration generics provide default values in case they are not explicitly assigned during instantiation.

### *Software Discovery of Configuration*



Software can determine the actual CPU configuration via the `misa` and `mxisa` CSRs. The Soc/Processor and can be determined via the `SYSINFO` memory-mapped registers.

### *Excluded Modules and Extensions*



If optional modules (like CPU extensions or peripheral devices) are not enabled the according hardware will not be synthesized at all. Hence, the disabled modules do not increase area and power requirements and do not impact timing.

### *Table Abbreviations*



The generic type “`suv(x:y)`” is an abbreviation for “`std_ulegic_vector(x downto y)`”.

Table 5. NEORV32 Processor Generic List

| Name                           | Type                   | Default    | Description  |
|--------------------------------|------------------------|------------|--|
| <b>General</b>                 |                        |            |  |
| <code>CLOCK_FREQUENCY</code>   | natural                | -          | The clock frequency of the processor's <code>clk_i</code> input port in Hertz (Hz).  |
| <code>CLOCK_GATING_EN</code>   | boolean                | false      | Enable clock gating when CPU is in sleep mode (see sections <a href="#">Sleep Mode</a> and <a href="#">Processor Clocking</a> ).               |
| <code>INT_BOOTLOADER_EN</code> | boolean                | false      | Implement the processor-internal <a href="#">Bootloader ROM (BOOTROM)</a> , pre-initialized with the default <a href="#">Bootloader</a> image. |
| <code>HART_ID</code>           | <code>suv(31:0)</code> | 0x00000000 | The hart thread ID of the CPU (passed to <code>mhartid</code> CSR).  |

| Name                                       | Type      | Default         | Description   |
|--|-----------|-----------------|---|
| <code>JEDEC_ID</code>                      | suv(10:0) | 0000000000<br>0 | JEDEC ID; continuation codes plus vendor ID (passed to <code>mvendorid</code> CSR and to the <b>Debug Transport Module (DTM)</b> ). |
| <b>On-Chip Debugger (OCD)</b>              |           |                 |   |
| <code>ON_CHIP_DEBUGGER_EN</code>           | boolean   | false           | Implement the on-chip debugger and the CPU debug mode.  |
| <code>DM_LEGACY_MODE</code>                | boolean   | false           | Debug module spec. version: <code>false</code> = v1.0, <code>true</code> = v0.13 (legacy mode).                                     |
| <b>CPU Instruction Sets and Extensions</b> |           |                 |   |
| <code>CPU_EXTENSION_RISC_V_A</code>        | boolean   | false           | Enable <b>A ISA Extension</b> (atomic memory accesses).   |
| <code>CPU_EXTENSION_RISC_V_B</code>        | boolean   | false           | Enable <b>B ISA Extension</b> (bit-manipulation).   |
| <code>CPU_EXTENSION_RISC_V_C</code>        | boolean   | false           | Enable <b>C ISA Extension</b> (compressed instructions).  |
| <code>CPU_EXTENSION_RISC_V_E</code>        | boolean   | false           | Enable <b>E ISA Extension</b> (reduced register file size).   |
| <code>CPU_EXTENSION_RISC_V_M</code>        | boolean   | false           | Enable <b>M ISA Extension</b> (hardware-based integer multiplication and division).   |
| <code>CPU_EXTENSION_RISC_V_U</code>        | boolean   | false           | Enable <b>U ISA Extension</b> (less-privileged user mode).  |
| <code>CPU_EXTENSION_RISC_V_Zfinx</code>    | boolean   | false           | Enable <b>Zfinx ISA Extension</b> (single-precision floating-point unit).   |
| <code>CPU_EXTENSION_RISC_V_Zicntr</code>   | boolean   | true            | Enable <b>Zicntr ISA Extension</b> (CPU base counters).   |
| <code>CPU_EXTENSION_RISC_V_Zicond</code>   | boolean   | false           | Enable <b>Zicond ISA Extension</b> (integer conditional operations).  |
| <code>CPU_EXTENSION_RISC_V_Zihpm</code>    | boolean   | false           | Enable <b>Zihpm ISA Extension</b> (hardware performance monitors).  |
| <code>CPU_EXTENSION_RISC_V_Zmmul</code>    | boolean   | false           | Enable <b>Zmmul - ISA Extension</b> (hardware-based integer multiplication).  |
| <code>CPU_EXTENSION_RISC_V_Zxcfu</code>    | boolean   | false           | Enable NEORV32-specific <b>Zxcfu ISA Extension</b> (custom RISC-V instructions).  |
| <b>CPU Architecture Tuning Options</b>     |           |                 |   |
| <code>FAST_MUL_EN</code>                   | boolean   | false           | Implement fast but large full-parallel multipliers (trying to infer DSP blocks); see section <b>CPU Arithmetic Logic Unit</b> .     |
| <code>FAST_SHIFT_EN</code>                 | boolean   | false           | Implement fast but large full-parallel barrel shifters; see section <b>CPU Arithmetic Logic Unit</b> .                              |

| Name  | Type    | Default | Description  |
|---|---------|---------|--|
| REGFILE_HW_RST  | boolean | false   | Implement full hardware reset for register file (prevent inferring of BRAM); see section <a href="#">CPU Register File</a> . |
| <b>Physical Memory Protection (<a href="#">Smpmp ISA Extension</a>)</b>                   |         |         |  |
| PMP_NUM_REGIONS   | natural | 0       | Number of implemented PMP regions (0..16).   |
| PMP_MIN_GRANULARITY   | natural | 4       | Minimal region granularity in bytes. Has to be a power of two, min 4.  |
| PMP_TOR_MODE_EN   | boolean | true    | Implement support for top-of-region (TOR) mode.  |
| PMP_NAP_MODE_EN   | boolean | true    | Implement support for naturally-aligned power-of-two (NAPOT & NA4) modes.  |
| <b>Hardware Performance Monitors (<a href="#">Zihpm ISA Extension</a>)</b>                |         |         |  |
| HPM_NUM_CNTS  | natural | 0       | Number of implemented hardware performance monitor counters (0..13).   |
| HPM_CNT_WIDTH   | natural | 40      | Total LSB-aligned size of each HPM counter. Min 0, max 64.   |
| <b>Atomic Memory Access Reservation Set Granularity (<a href="#">A ISA Extension</a>)</b> |         |         |  |
| AMO_RVS_GRANULARITY   | natural | 4       | Size in bytes, has to be a power of 2, min 4.  |
| <b>Internal Instruction Memory (IMEM)</b>   |         |         |  |
| MEM_INT_IMEM_EN   | boolean | false   | Implement the processor-internal instruction memory.   |
| MEM_INT_IMEM_SIZE   | natural | 16*1024 | Size in bytes of the processor internal instruction memory (use a power of 2).   |
| <b>Internal Data Memory (DMEM)</b>  |         |         |  |
| MEM_INT_DMEM_EN   | boolean | false   | Implement the processor-internal data memory.  |
| MEM_INT_DMEM_SIZE   | natural | 8*1024  | Size in bytes of the processor-internal data memory (use a power of 2).  |
| <b>Processor-Internal Instruction Cache (iCACHE)</b>                                      |         |         |  |
| ICACHE_EN   | boolean | false   | Implement the instruction cache.   |
| ICACHE_NUM_BLOCKS   | natural | 4       | Number of blocks ("lines") Has to be a power of two.   |
| ICACHE_BLOCK_SIZE   | natural | 64      | Size in bytes of each block. Has to be a power of two.   |
| <b>Processor-Internal Data Cache (dCACHE)</b>   |         |         |  |
| DCACHE_EN   | boolean | false   | Implement the data cache.  |
| DCACHE_NUM_BLOCKS   | natural | 4       | Number of blocks ("lines"). Has to be a power of two.  |

| Name  | Type      | Default    | Description  |
|---|-----------|------------|--|
| DCACHE_BLOCK_SIZE   | natural   | 64         | Size in bytes of each block. Has to be a power of two.   |
| <b>Processor-External Bus Interface (XBUS) (Wishbone b4 protocol)</b> |           |            |  |
| XBUS_EN   | boolean   | false      | Implement the external bus interface.  |
| XBUS_TIMEOUT  | natural   | 255        | Clock cycles after which a pending external bus access will auto-terminate and raise a bus fault exception.        |
| XBUS_REGSTAGE_EN  | boolean   | false      | Implement XBUS register stages to ease timing closure.   |
| XBUS_CACHE_EN   | boolean   | false      | Implement the external bus cache.  |
| XBUS_CACHE_NUM_BLOCKS   | natural   | 64         | Number of blocks ("lines"). Has to be a power of two.  |
| XBUS_CACHE_BLOCK_SIZE   | natural   | 32         | Size in bytes of each block. Has to be a power of two.   |
| <b>Execute In Place Module (XIP)</b>                                  |           |            |  |
| XIP_EN  | boolean   | false      | Implement the execute in-place module.   |
| XIP_CACHE_EN  | boolean   | false      | Implement XIP cache.   |
| XIP_CACHE_NUM_BLOCKS  | natural   | 8          | Number of blocks in XIP cache. Has to be a power of two.   |
| XIP_CACHE_BLOCK_SIZE  | natural   | 256        | Number of bytes per XIP cache block. Has to be a power of two, min 4.  |
| <b>External Interrupt Controller (XIRQ)</b>                           |           |            |  |
| XIRQ_NUM_CH   | natural   | 0          | Number of channels of the external interrupt controller. Valid values are 0..32.                                   |
| XIRQ_TRIGGER_TYPE   | suv(31:0) | 0xFFFFFFFF | Trigger type (one bit per channel): 0 = level-triggered, '1' = edge triggered.                                     |
| XIRQ_TRIGGER_POLARITY   | suv(31:0) | 0xFFFFFFFF | Trigger polarity (one bit per channel): 0 = low-level/falling-edge, '1' = high-level/rising-edge.                  |
| <b>Peripheral/IO Modules</b>  |           |            |  |
| IO_GPIO_NUM   | natural   | 0          | Number of general purpose input/output pairs of the <a href="#">General Purpose Input and Output Port (GPIO)</a> . |
| IO_MTIME_EN   | boolean   | false      | Implement the <a href="#">Machine System Timer (MTIME)</a> .   |
| IO_UART0_EN   | boolean   | false      | Implement the <a href="#">Primary Universal Asynchronous Receiver and Transmitter (UART0)</a> .                    |
| IO_UART0_RX_FIFO  | natural   | 1          | UART0 RX FIFO depth, has to be a power of two, minimum value is 1, max 32768.                                      |

| Name             | Type      | Default    | Description   |
|------------------|-----------|------------|---|
| IO_UART0_TX_FIFO | natural   | 1          | UART0 TX FIFO depth, has to be a power of two, minimum value is 1, max 32768.                               |
| IO_UART1_EN      | boolean   | false      | Implement the Secondary Universal Asynchronous Receiver and Transmitter (UART1).                            |
| IO_UART1_RX_FIFO | natural   | 1          | UART1 RX FIFO depth, has to be a power of two, minimum value is 1, max 32768.                               |
| IO_UART1_TX_FIFO | natural   | 1          | UART1 TX FIFO depth, has to be a power of two, minimum value is 1, max 32768.                               |
| IO_SPI_EN        | boolean   | false      | Implement the Serial Peripheral Interface Controller (SPI).   |
| IO_SPI_FIFO      | natural   | 1          | Depth of the Serial Peripheral Interface Controller (SPI) FIFO. Has to be a power of two, min 1, max 32768. |
| IO_SDI_EN        | boolean   | false      | Implement the Serial Data Interface Controller (SDI).   |
| IO_SDI_FIFO      | natural   | 1          | Depth of the Serial Data Interface Controller (SDI) FIFO. Has to be a power of two, min 1, max 32768.       |
| IO_TWI_EN        | boolean   | false      | Implement the Two-Wire Serial Interface Controller (TWI).   |
| IO_TWI_FIFO      | natural   | 1          | Depth of the Two-Wire Serial Interface Controller (TWI) FIFO. Has to be a power of two, min 1, max 32768.   |
| IO_PWM_NUM_CH    | natural   | 0          | Number of channels of the Pulse-Width Modulation Controller (PWM) to implement (0..12).                     |
| IO_WDT_EN        | boolean   | false      | Implement the Watchdog Timer (WDT).   |
| IO_TRNG_EN       | boolean   | false      | Implement the True Random-Number Generator (TRNG).  |
| IO_TRNG_FIFO     | natural   | 1          | Depth of the TRNG data FIFO. Has to be a power of two, min 1, max 32768.                                    |
| IO_CFS_EN        | boolean   | false      | Implement the Custom Functions Subsystem (CFS).   |
| IO_CFS_CONFIG    | suv(31:0) | 0x00000000 | "Conduit" generic to pass user-defined flags to the Custom Functions Subsystem (CFS).                       |
| IO_CFS_IN_SIZE   | natural   | 32         | Size of the Custom Functions Subsystem (CFS) input signal conduit ( <code>cfs_in_i</code> ).                |
| IO_CFS_OUT_SIZE  | natural   | 32         | Size of the Custom Functions Subsystem (CFS) output signal conduit ( <code>cfs_out_o</code> ).              |
| IO_NEOLED_EN     | boolean   | false      | Implement the Smart LED Interface (NEOLED).   |

| Name              | Type    | Default | Description  |
|-------------------|---------|---------|--|
| IO_NEOLED_TX_FIFO | natural | 1       | TX FIFO depth of the the Smart LED Interface (NEOLED). Has to be a power of two, min 1, max 32768. |
| IO_GPTMR_EN       | boolean | false   | Implement the General Purpose Timer (GPTMR).   |
| IO_ONEWIRE_EN     | boolean | false   | Implement the One-Wire Serial Interface Controller (ONEWIRE).                                      |
| IO_DMA_EN         | boolean | false   | Implement the Direct Memory Access Controller (DMA).   |
| IO_SLINK_EN       | boolean | false   | Implement the Stream Link Interface (SLINK).   |
| IO_SLINK_RX_FIFO  | natural | 1       | SLINK RX FIFO depth, has to be a power of two, minimum value is 1, max 32768.                      |
| IO_SLINK_TX_FIFO  | natural | 1       | SLINK TX FIFO depth, has to be a power of two, minimum value is 1, max 32768.                      |
| IO_CRC_EN         | boolean | false   | Implement the Cyclic Redundancy Check (CRC) unit.  |

## 2.3. Processor Clocking

The processor is implemented as fully-synchronous logic design using a single clock domain that is driven entirely by the top's `clk_i` signal. This clock signal is used by all internal registers and memories, which trigger on the rising edge of this clock signal - except for the **Processor Reset** and the clock switching gate that trigger on a falling edge. External "clocks" like the OCD's JTAG clock or the SDI's serial clock are synchronized into the processor's clock domain before being further processed.

### 2.3.1. Clock Gating

The single clock domain of the processor can be split into an always-on clock domain and a switchable clock domain. The switchable clock domain is used to clock the CPU core, the CPU's bus switch and - if implemented - the caches. This domain can be deactivated to reduce power consumption. The always-on clock domain is used to clock all other processor modules like peripherals, memories and IO devices. Hence, these modules can continue operation (e.g. a timer keeps running) even if the CPU is shut down.

The splitting into two clock domain is enabled by the `CLOCK_GATING_EN` generic ([Processor Top Entity - Generics](#)). When enabled, a generic clock switching gate is added to decouple the switchable clock from the always-on clock domain (VHDL file `neorv32_clockgate.vhd`). Whenever the CPU enters **Sleep Mode** the CPU clock domain ist shut down.

#### *Clock Switch Hardware*



By default, a generic clock gate is used (`rtl/core/neorv32_clockgate.vhd`) to shut down the CPU clock. Especially for FPGA setups it is highly recommended to replace this default version by a technology-specific primitive or macro wrapper to improve efficiency (clock skew, global clock tree usage, etc.).

### 2.3.2. Peripheral Clocks

Many processor modules like the UARTs or the timers provide a programmable time base for operations. In order to simplify the hardware, the processor implements a global "clock generator" that provides *clock enables* for certain frequencies that are derived from the main clock. Hence, these clock enable signals are synchronous to the system's main clock and will be high for only a single cycle. The processor modules can use these enables for sub-main-clock operations while still providing a single clock domain only.

In total, 8 sub-main-clock signals are available. All processor modules, which feature a time-based configuration, provide a programmable three-bit prescaler select in their control register to select one of the 8 available clocks. The mapping of the prescaler select bits to the according clock source is shown in the table below. Here,  $f$  represents the processor main clock from the top entity's `clk_i` signal.

|                 |       |       |       |       |       |       |       |       |
|-----------------|-------|-------|-------|-------|-------|-------|-------|-------|
| Prescaler bits: | 0b000 | 0b001 | 0b010 | 0b011 | 0b100 | 0b101 | 0b110 | 0b111 |
|-----------------|-------|-------|-------|-------|-------|-------|-------|-------|

Resulting clock:

[f/2](#)    [f/4](#)    [f/8](#)    [f/64](#)    [f/128](#)    [f/1024](#)    [f/2048](#)    [f/4096](#)*Power Saving*

If no peripheral modules require a clock signal from the internal clock generator (all according modules are disabled by clearing the enable bit in the according module's control register) the generator is automatically deactivated to reduce dynamic power consumption.

## 2.4. Processor Reset



### Processor Reset Signal

Always make sure to connect the processor's reset signal `rstn_i` to a valid reset source (a button, the "locked" signal of a PLL, a dedicated reset controller, etc.).

The processor-wide reset can be triggered by any of the following sources:

- the asynchronous low-active `rstn_i` top entity input signal
- the [On-Chip Debugger \(OCD\)](#)
- the [Watchdog Timer \(WDT\)](#)



### Reset Cause

The actual reset cause can be determined via the [Watchdog Timer \(WDT\)](#).

If any of these sources trigger a reset, the internal reset will be triggered for at least 4 clock cycles ensuring a valid reset of the entire processor. The internal global reset is asserted *asynchronously* if triggered by the external `rstn_i` signal. For internal reset sources, the global reset is asserted *synchronously*. If the reset cause gets inactive the internal reset is de-asserted *synchronously* at a falling clock edge.

Internally, **all registers** that are not meant for mapping to blockRAM (like the register file) do provide a dedicated and low-active **asynchronous hardware reset**. This asynchronous reset ensures that the entire processor logic is reset to a defined state even if the main clock is not operational yet.



The system reset will only reset the control registers of each implemented IO/peripheral module. This control register reset will also reset the according "module enable flag" to zero, which - in turn - will cause a *synchronous* module-internal reset of the remaining logic.

## 2.5. Processor Interrupts

The NEORV32 Processor provides several interrupt request signals (IRQs) for custom platform use.



### Trigger Type

All interrupt request lines are **level-triggered and high-active**. Once set, the signal should remain high until the interrupt request is explicitly acknowledged (e.g. writing to a memory-mapped register).

### 2.5.1. RISC-V Standard Interrupts

The processor setup features the standard machine-level RISC-V interrupt lines for "machine timer interrupt", "machine software interrupt" and "machine external interrupt". Their usage is defined by the RISC-V privileged architecture specifications. However, bare-metal system can also repurpose these interrupts. See CPU section [Traps, Exceptions and Interrupts](#) for more information.

| Top signal               | Description   |
|--------------------------|---|
| <code>mtime_irq_i</code> | Machine timer interrupt from <i>processor-external</i> MTIME unit ( <a href="#">MTI</a> ). This IRQ is only available if the processor-internal <a href="#">Machine System Timer (MTIME)</a> unit is not implemented. |
| <code>msw_irq_i</code>   | Machine software interrupt ( <a href="#">MSI</a> ). This interrupt is used for inter-processor interrupts in multi-core systems. However, it can also be used for any custom purpose.                                 |
| <code>mext_irq_i</code>  | Machine external interrupt ( <a href="#">MEI</a> ). This interrupt is used for any processor-external interrupt source (like a platform interrupt controller).  |

### 2.5.2. NEORV32-Specific Fast Interrupt Requests

As part of the NEORV32-specific CPU extensions, the processor core features 16 fast interrupt request signals ([FIRQ0](#) to [FIRQ15](#)) providing dedicated bits in the `mip` and `mie` CSRs and custom `mcause` trap codes. The FIRQ signals are reserved for *processor-internal* modules only (for example for the communication interfaces to signal "available incoming data" or "ready to send new data").

The mapping of the 16 FIRQ channels to the according processor-internal modules is shown in the following table (the channel number also corresponds to the according FIRQ priority: 0 = highest, 15 = lowest):

Table 6. NEORV32 Fast Interrupt Request (FIRQ) Mapping

| Channel | Source                | Description   |
|---------|-----------------------|---|
| 0       | -                     | <i>reserved</i>   |
| 1       | <a href="#">CFS</a>   | custom functions subsystem (CFS) interrupt (user-defined) |
| 2       | <a href="#">UART0</a> | UART0 RX FIFO level interrupt                             |

| Channel | Source  | Description                             |
|---------|---------|---|
| 3       | UART0   | UART0 TX FIFO level interrupt           |
| 4       | UART1   | UART1 RX FIFO level interrupt           |
| 5       | UART1   | UART1 TX FIFO level interrupt           |
| 6       | SPI     | SPI FIFO level interrupt                |
| 7       | TWI     | TWI FIFO level interrupt                |
| 8       | XIRQ    | External interrupt controller interrupt |
| 9       | NEOLED  | NEOLED TX FIFO level interrupt          |
| 10      | DMA     | DMA transfer done interrupt             |
| 11      | SDI     | SDI FIFO level interrupt                |
| 12      | GPTMR   | General purpose timer interrupt         |
| 13      | ONEWIRE | 1-wire idle interrupt                   |
| 14      | SLINK   | SLINK RX FIFO level interrupt           |
| 15      | SLINK   | SLINK TX FIFO level interrupt           |

## 2.6. Address Space

As a 32-bit architecture the NEORV32 can access a 4GB physical address space. By default, this address space is split into six main regions. Each region provides specific *physical memory attributes* ("PMAs") that define the access capabilities (`rwxac`; `r` = read permission, `w` = write permission, `x` - execute permission, `a` = atomic access support, `c` = cached CPU access).

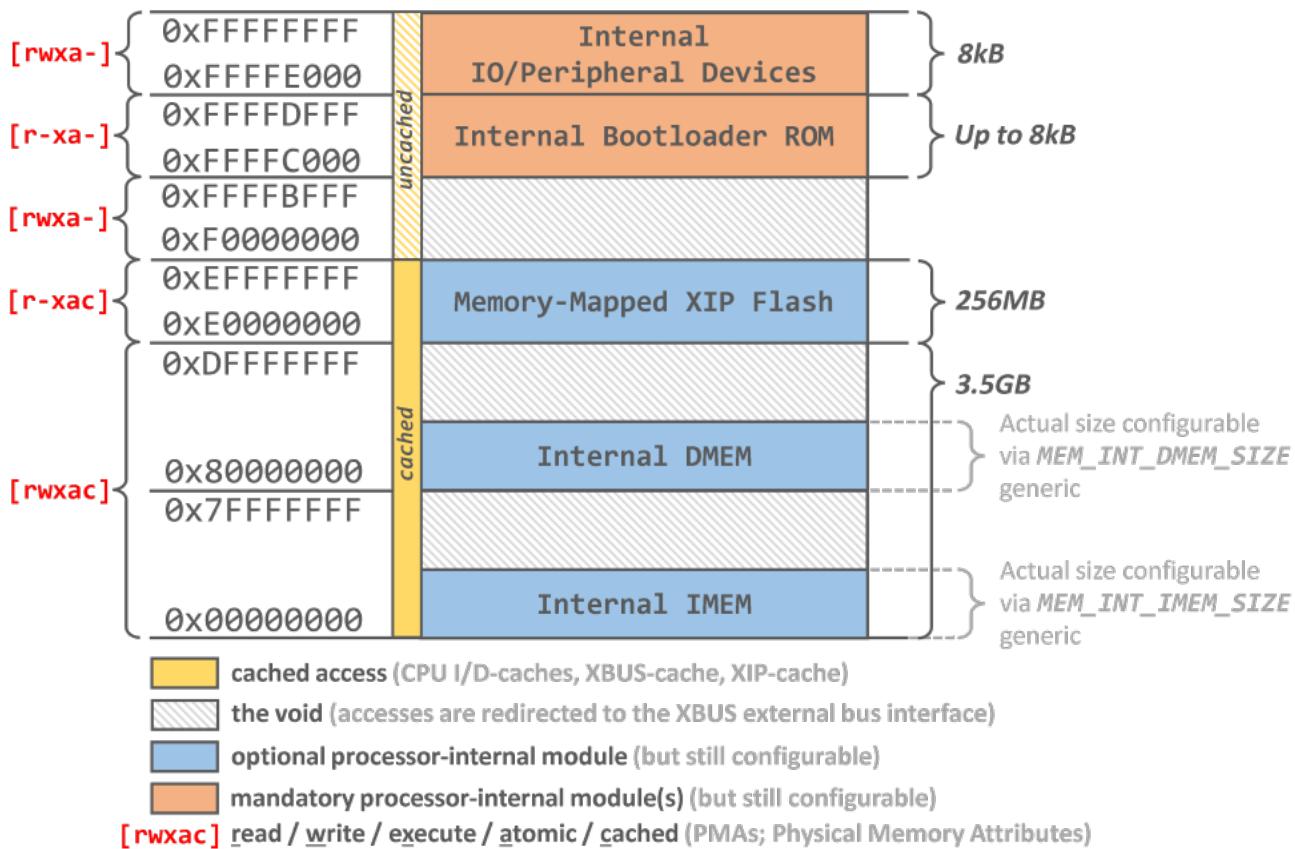


Figure 2. NEORV32 Processor Address Space (Default Configuration)

Table 7. Main Address Regions

| # | Region                      | PMAs               | Description  |
|---|-----------------------------|--------------------|--|
| 1 | Internal IMEM address space | <code>rwxac</code> | For instructions (=code) and constants; mapped to the internal <b>Instruction Memory (IMEM)</b> .            |
| 2 | Internal DMEM address space | <code>rwxac</code> | For application runtime data (heap, stack, etc.); mapped to the internal <b>Data Memory (DMEM)</b> .         |
| 3 | Memory-mapped XIP flash     | <code>r-xac</code> | Memory-mapped access to the <b>Execute In Place Module (XIP)</b> SPI flash.                                  |
| 4 | Bootloader address space    | <code>r-xa-</code> | Read-only memory for the internal <b>Bootloader ROM (BOOTROM)</b> containing the default <b>Bootloader</b> . |
| 5 | IO/peripheral address space | <code>rwxac</code> | Processor-internal peripherals / IO devices.   |

| # | Region     | PMAs               | Description  |
|---|------------|--------------------|--|
| 6 | The "void" | <code>RWXAC</code> | Unmapped address space. All accesses to this region(s) are redirected to the <a href="#">Processor-External Bus Interface (XBUS)</a> (if implemented). |



#### *Custom PMAs*

Physical memory attributes can be customized (constrained) using the CPU's [Smpmp ISA Extension](#).

### 2.6.1. Bus System

The CPU can access all of the 32-bit address space from the instruction fetch interface and also from the data access interface. Both CPU interfaces can be equipped with optional caches ([Processor-Internal Data Cache \(dCACHE\)](#) and [Processor-Internal Instruction Cache \(iCACHE\)](#)). The two CPU interfaces are multiplexed by a simple bus switch into a *single processor-internal bus*. Optionally, this bus is further switched by another instance of the bus switch so the [Direct Memory Access Controller \(DMA\)](#) controller can also access the entire address space. Accesses via the resulting SoC bus are split by the [Bus Gateway](#) that redirects accesses to the according main address regions (see table above). Accesses to the processor-internal IO/peripheral devices are further redirected via a dedicated [IO Switch](#).

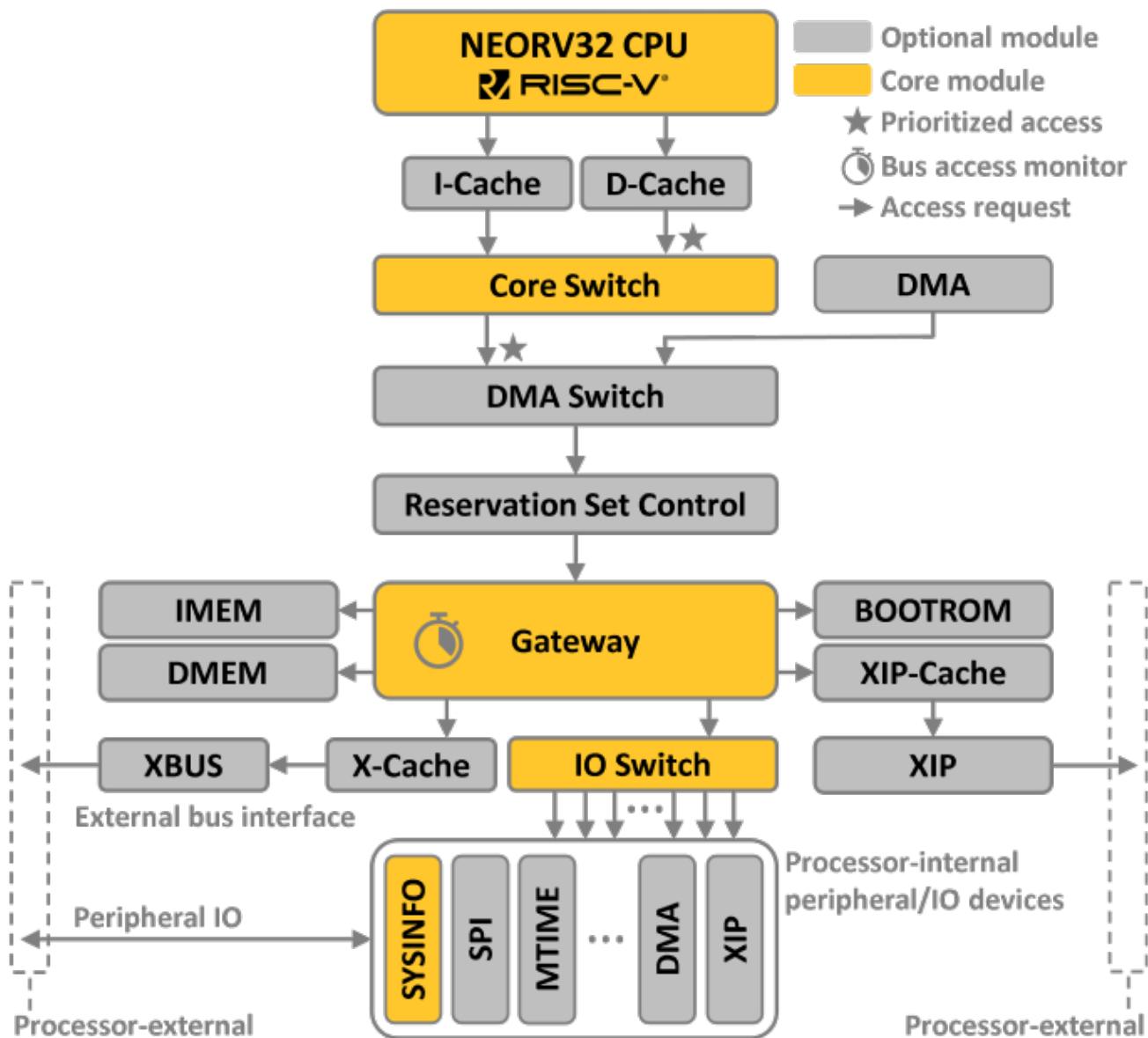


Figure 3. Processor-Internal Bus Architecture

*Bus Interface*

See sections [CPU Architecture](#) and [Bus Interface](#) for more information regarding the CPU bus accesses.

## 2.6.2. Bus Gateway

The central bus gateway serves two purposes: **redirect** core accesses to the according modules (e.g. memory accesses vs. memory-mapped IO accesses) and **monitor** all bus transactions. The redirection of access request is based on a customizable memory map implemented via VHDL constants in the main package file ([rtl/core/neorv323\\_package.vhd](#)):

Listing 1. Main Address Regions Configuration in the VHDL Package File

```
-- Main Address Regions --
constant mem_imem_base_c : std_ulogic_vector(31 downto 0) := x"00000000";
```

```
constant mem_dmem_base_c : std_ulegic_vector(31 downto 0) := x"80000000";
constant mem_xip_base_c : std_ulegic_vector(31 downto 0) := x"e0000000";
constant mem_xip_size_c : natural := 256*1024*1024;
constant mem_boot_base_c : std_ulegic_vector(31 downto 0) := x"fffffc000";
constant mem_boot_size_c : natural := 8*1024;
constant mem_io_base_c : std_ulegic_vector(31 downto 0) := x"fffffe000";
constant mem_io_size_c : natural := 8*1024;
```

Besides the delegation of bus requests the gateway also implements a bus monitor (aka "the bus keeper") that tracks all active bus transactions to ensure *safe* and *deterministic* operations.

Whenever a memory-mapped device is accessed (a real memory, a memory-mapped IO or some processor-external module) the bus monitor starts an internal timer. The accessed module has to respond ("ACK") to the bus request within a specific **time window**. This time window is defined by a global constant in the processor's VHDL package file ([rtl/core/neorv323\\_package.vhd](#)).

*Listing 2. Internal Bus Timeout Configuration*

```
constant bus_timeout_c : natural := 15;
```

This constant defines the *maximum* number of cycles after which a non-responding bus request (i.e. no **ack** and no **err** signal) will time out raising a bus access fault exception. For example this can happen when accessing "address space holes" - addresses that are not mapped to any physical module. The resulting exception type corresponds to the according access type, i.e. instruction fetch access exception, load access exception or store access exception.

#### XIP Timeout



Accesses to the memory-mapped XIP flash (via the [Execute In Place Module \(XIP\)](#)) will *never* time out.

#### External Bus Interface Timeout



Accesses that are delegated to the external bus interface have a different maximum timeout value that is defined by an explicit specific processor generic. See section [Processor-External Bus Interface \(XBUS\)](#) for more information.

### 2.6.3. Reservation Set Controller

The reservation set controller is responsible for handling the load-reservate and store-conditional bus transaction that are triggered by the **lr.w** (LR) and **sc.w** (SC) instructions from the CPU's [A ISA Extension](#).

A "reservation" defines an address or address range that provides a guarding mechanism to support atomic accesses. A new reservation is registered by the LR instruction. The address provided by this instruction defines the memory location that is now monitored for atomic accesses. The according SC instruction evaluates the state of this reservation. If the reservation is

still valid the write access triggered by the SC instruction is finally executed and the instruction return a "success" state ( $rd = 0$ ). If the reservation has been invalidated the SC instruction will not write to memory and will return a "failed" state ( $rd = 1$ ).

The reservation is invalidated if...

- an SC instruction is executed that accesses an address **outside** of the reservation set of the previous LR instruction. This SC instruction will **fail** (not writing to memory).
- an SC instruction is executed that accesses an address **inside** of the reservation set of the previous LR instruction. This SC instruction will **succeed** (finally writing to memory).
- a normal store operation accesses an address **inside** of the current reservation set (by the CPU or by the DMA).
- a hardware reset is triggered.

#### *Consecutive LR Instructions*



If an LR instruction is followed by another LR instruction the reservation set of the former one is overridden by the reservation set of the latter one.

#### *Bus Access Errors*



If the LR operation causes a bus access error (raising a load access exception) the reservation **is registered anyway**. If the SC operation causes a bus access error (raising a store access exception) an already registered reservation set **is invalidated anyway**.

#### *Strong Semantic*



The LR/SC mechanism follows the *strong semantic* approach: the LR/SC instruction pair fails only if there is a write access to the referenced memory location between the LR and SC instructions (by the CPU itself or by the DMA). Context changes, interrupts, traps, etc. do not effect nor invalidate the reservation state at all.

The controller supports only a single global reservation set. By default this reservation set "monitors" a word-aligned 4-byte granule. However, the granularity can be customized via the `AMO_RVS_GRANULARITY` top entity generic (see [Processor Top Entity - Generics](#)) to cover an arbitrarily large naturally aligned address region. The only constraint is that the size of the address region has to be a power of two. The configured granularity can be determined by software via the [System Configuration Information Memory \(SYSINFO\)](#) module.

#### *Physical Memory Attributes*



The reservation set can be set for *any* address (only constrained by the configured granularity). This also includes cached memory, memory-mapped IO devices and processor-external address spaces.

Bus transactions triggered by the LR instruction register a new reservation set and are delegated to the addressed memory/device. Bus transactions triggered by the SC remove a reservation set and are

forwarded to the addressed memory/device only if the SC operations succeeds. Otherwise, the access request is not forwarded and a local ACK is generated to terminate the bus transaction.

#### *LR/SC Bus Protocol*



More information regarding the LR/SC bus transactions and the the according protocol can be found in section [Bus Interface / Atomic Accesses](#).

#### *Cache Coherency*



Atomic operations **always bypass** the cache using direct/uncached accesses. Care must be taken to maintain data cache coherency (e.g. by using the `fence` instruction).

### 2.6.4. IO Switch

The IO switch further decodes the address when accessing the processor-internal IO/peripheral devices and forwards the access request to the according module. Note that a total address space size of 256 bytes is assigned to each IO module in order to simplify address decoding. The IO-specific address map is also defined in the main VHDL package file ([rtl/core/neorv323\\_package.vhd](#)).

*Listing 3. Exemplary Cut-Out from the IO Address Map*

```
-- IO Address Map --
constant iodev_size_c    : natural := 256; -- size of a single IO device (bytes)
constant base_io_cfs_c   : std_ulogic_vector(31 downto 0) := x"fffffeb00";
constant base_io_slink_c : std_ulogic_vector(31 downto 0) := x"fffffec00";
constant base_io_dma_c   : std_ulogic_vector(31 downto 0) := x"fffffed00";
```

### 2.6.5. Boot Configuration

Due to the flexible memory configuration, the NEORV32 Processor provides several different boot scenarios. The following section illustrates the two most common boot scenarios.

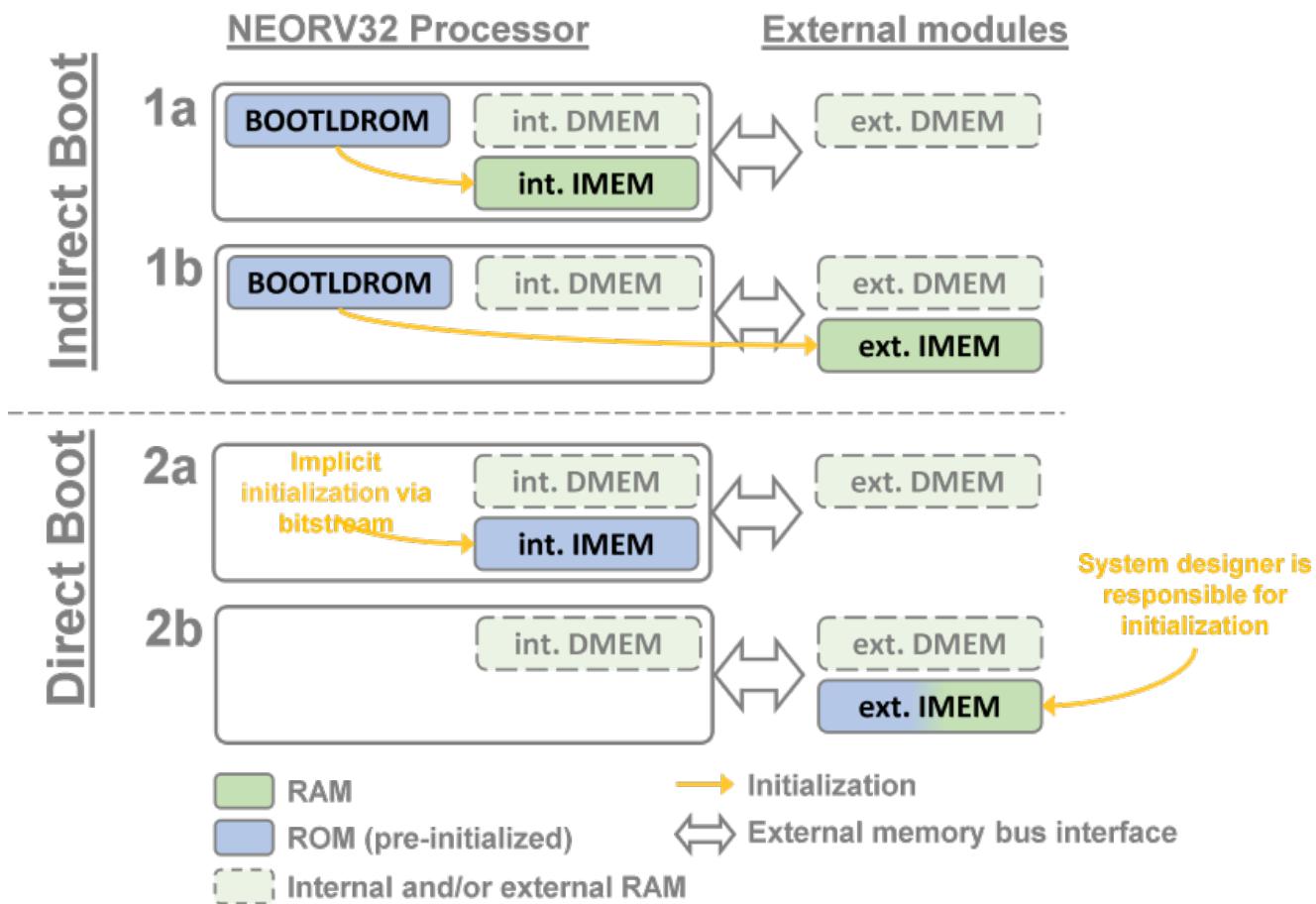


Figure 4. NEORV32 Boot Configurations

There are two general boot scenarios: *Indirect Boot* (1a and 1b) and *Direct Boot* (2a and 2b) configured via the `INT_BOOTLOADER_EN` generic. If this generic is `true` the *indirect boot scenario* is used. This is also the default boot configuration of the processor. If `INT_BOOTLOADER_EN` is `*false` the *direct boot scenario* is used.

### Indirect Boot

The indirect\_boot scenarios **1a** and **1b** are based on the processor-internal **Bootloader**. This boot setup is enabled by setting the `INT_BOOTLOADER_EN` generic to `true`, which will implement the processor-internal **Bootloader ROM (BOOTROM)**. This read-only memory is pre-initialized during synthesis with the default bootloader firmware. The bootloader provides several options to upload an executable copying it to the beginning of the *instruction address space* so the CPU can execute it.

Boot scenario **1a** uses the processor-internal IMEM. This scenario implements the internal **Instruction Memory (IMEM)** as non-initialized RAM so the bootloader can copy the actual executable to it.

Boot scenario **1b** uses a processor-external IMEM that is connected via the processor's bus interface. In this scenario the internal **Instruction Memory (IMEM)** is not implemented at all and the bootloader will copy the executable to the processor-external memory. Hence, the external memory has to be implemented as RAM.

## Direct Boot

The direct boot scenarios **2a** and **2b** do not use the processor-internal bootloader since the `INT_BOOTLOADER_EN` generic is set `false`. In this configuration the **Bootloader ROM (BOOTROM)** is not implemented at all and the CPU will directly begin executing code from the beginning of the instruction address space after reset. An application-specific "pre-initialization" mechanism is required in order to provide an executable inside the memory.

Boot scenario **2a** uses the processor-internal IMEM implemented as *read-only memory* in this scenario. It is pre-initialized (by the bitstream) with the actual application executable during synthesis.

In contrast, boot scenario **2b** uses a processor-external IMEM. In this scenario the system designer is responsible for providing an initialized external memory that contains the actual application to be executed.

## 2.7. Processor-Internal Modules

### *Module Address Space Mapping*



The base address of each component/module has to be aligned to the total size of the module's occupied address space. The occupied address space has to be a power of two (minimum 4 bytes). Addresses of peripheral modules must not overlap.



### *Full-Word Write Accesses Only*

All peripheral/IO devices should only be written in full-word mode (i.e. 32-bit). Byte or half-word (8/16-bit) write accesses might cause undefined behavior.



### *IO Module's Address Space*

Each peripheral/IO module occupies an address space of 256 bytes (64 words). Most devices do not fully utilize this address space and will simply *mirror* the available interface registers across the entire 256 bytes of address space.



### *Unimplemented Modules / Address Holes*

When accessing an IO device that has not been implemented (disabled via the according generic) or when accessing an address that is actually unused, a load/store access fault exception is raised.



### *Module Interrupts*

Most peripheral/IO devices provide some kind of interrupt (for example to signal available incoming data). These interrupts are entirely mapped to the CPU's **Custom Fast Interrupt Request Lines**. See section **Processor Interrupts** for more information.



### *CMSIS System Description View (SVD)*

A CMSIS-SVD-compatible **System View Description (SVD)** file including all peripherals is available in [sw/svd](#).

## 2.7.1. Instruction Memory (IMEM)

|                          |  |   |
|--------------------------|--|---|
| Hardware source file(s): | neorv32_imem.entity.vhd<br>mem/neorv32_imem.default.vhd<br>mem/neorv32_imem.legacy.vhd | entity-only definition<br>default <i>platform-agnostic</i> memory architecture<br>alternative legacy-style memory architecture  |
| Software driver file(s): | none   | <i>implicitly used</i>  |
| Top entity port:         | none   |   |
| Configuration generics:  | <b>MEM_INT_IMEM_EN</b><br><b>MEM_INT_IMEM_SIZE</b><br><b>INT_BOOTLOADER_EN</b>         | implement processor-internal IMEM when <b>true</b><br>IMEM size in bytes (use a power of 2)<br>use internal bootloader when <b>true</b><br>(implements IMEM as <i>uninitialized</i> RAM, otherwise the IMEM is implemented as <i>pre-initialized</i> ROM) |
| CPU interrupts:          | none   |   |

Implementation of the processor-internal instruction memory is enabled by the processor's **MEM\_INT\_IMEM\_EN** generic. The total memory size in bytes is defined via the **MEM\_INT\_IMEM\_SIZE** generic. Note that this size should be a power of two to optimize physical implementation. If the IMEM is implemented, it is mapped to base address **0x00000000** by default (see section [Address Space](#)).

By default the IMEM is implemented as true RAM so the content can be modified during run time. This is required when using the bootloader (or the on-chip debugger) so it can update the content of the IMEM at any time. If this feature is not required the IMEM can be implemented as *pre-initialized* ROM so that the application code permanently resides in memory. This is automatically implemented when the processor-internal bootloader is disabled (**INT\_BOOTLOADER\_EN = false**).

When the IMEM is implemented as ROM, it will be initialized during synthesis with the actual application program image. The compiler toolchain provides an option to generate and override the default VHDL initialization file **rtl/core/neorv32\_application\_image.vhd**, which is automatically inserted into the IMEM. If the IMEM is implemented as RAM (default), the memory block will **not be initialized at all**.

### Memory Size



If the configured memory size (via the **MEM\_INT\_IMEM\_SIZE** generic) is **not** a power of two the actual memory size will be auto-adjusted to the next power of two (e.g. configuring a memory size of 60kB will result in a physical memory size of 64kB).



### VHDL Source File

The actual IMEM is split into two design files: a plain entity definition

(`neorv32_imem.entity.vhd`) and the actual architecture definition `mem/neorv32_imem.default.vhd`. This **default architecture** provides a *generic* and *platform independent* memory design that infers embedded memory blocks (blockRAM). The default architecture can be replaced by platform-specific modules in order to use platform-specific features or to improve technology mapping and/or timing. A "legacy-style" memory architecture is provided in `rtl/mem` that can be used if the synthesis does not correctly infer blockRAMs.

#### *Read-Only Access*



If the IMEM is implemented as true ROM any write attempt to it will raise a *store access fault* exception.

## 2.7.2. Data Memory (DMEM)

|                          |  |  |
|--------------------------|--|--|
| Hardware source file(s): | neorv32_dmem.entity.vhd<br>mem/neorv32_dmem.default.vhd<br>mem/neorv32_dmem.legacy.vhd | entity-only definition<br><i>default platform-agnostic memory architecture</i><br>alternative legacy-style memory architecture |
| Software driver file(s): | none   | <i>implicitly used</i>   |
| Top entity port:         | none   |  |
| Configuration generics:  | <b>MEM_INT_DMEM_EN</b><br><b>MEM_INT_DMEM_SIZE</b>                                     | implement processor-internal DMEM when <b>true</b><br>DMEM size in bytes (use a power of 2)                                    |
| CPU interrupts:          | none   |  |

Implementation of the processor-internal data memory is enabled by the processor's **MEM\_INT\_DMEM\_EN** generic. The total memory size in bytes is defined via the **MEM\_INT\_DMEM\_SIZE** generic. Note that this size should be a power of two to optimize physical implementation. If the DMEM is implemented, it is mapped to base address **0x80000000** by default (see section [Address Space](#)). The DMEM is always implemented as true RAM.



### Memory Size

If the configured memory size (via the **MEM\_INT\_IMEM\_SIZE** generic) is **not** a power of two the actual memory size will be auto-adjusted to the next power of two (e.g. configuring a memory size of 60kB will result in a physical memory size of 64kB).



### VHDL Source File

The actual DMEM is split into two design files: a plain entity definition **neorv32\_dmem.entity.vhd** and the actual architecture definition **mem/neorv32\_dmem.default.vhd**. This **default architecture** provides a *generic* and *platform independent* memory design that infers embedded memory blocks (blockRAM). The default architecture can be replaced by platform-specific modules in order to use platform-specific features or to improve technology mapping and/or timing. A "legacy-style" memory architecture is provided in **rtl/mem** that can be used if the synthesis does not correctly infer blockRAMs.



### Execute from RAM

The CPU is capable of executing code also from arbitrary data memory.

### 2.7.3. Bootloader ROM (BOOTROM)

|                          |   |
|--------------------------|---|
| Hardware source file(s): | neorv32_boot_rom.vhd  |
| Software driver file(s): | none  |
| Top entity port:         | none  |
| Configuration generics:  | <code>INT_BOOTLOADER_EN</code> implement processor-internal bootloader when <code>true</code> |
| CPU interrupts:          | none  |

This boot ROM module provides a read-only memory that contain the executable image of the default NEORV32 [Bootloader](#). If the internal bootloader is enabled via the `INT_BOOTLOADER_EN` generic the CPU's boot address is automatically set to the beginning of the bootloader ROM. See sections [Address Space](#) and [Boot Configuration](#) for more information regarding the processor's different boot scenarios.

#### *Memory Size*



If the configured boot ROM size is **not** a power of two the actual memory size will be auto-adjusted to the next power of two (e.g. configuring a memory size of 6kB will result in a physical memory size of 8kB).

#### *Bootloader Image*



The boot ROM is initialized during synthesis with the default bootloader image (`rtl/core/neorv32_bootloader_image.vhd`).

## 2.7.4. Processor-Internal Instruction Cache (iCACHE)

|                          |                          |   |
|--------------------------|--------------------------|---|
| Hardware source file(s): | neorv32_cache.vhd        | Generic cache module  |
| Software driver file(s): | none                     | <i>implicitly used</i>  |
| Top entity port:         | none                     |   |
| Configuration generics:  | <b>ICACHE_EN</b>         | implement processor-internal instruction cache when <b>true</b> |
|                          | <b>ICACHE_NUM_BLOCKS</b> | number of cache blocks (pages/lines)                            |
|                          | <b>ICACHE_BLOCK_SIZE</b> | size of a cache block in bytes                                  |
| CPU interrupts:          | none                     |   |

The processor features an optional instruction cache to improve performance when using memories with high access latencies. The cache is connected directly to the CPU's instruction fetch interface and provides full-transparent accesses. The cache is direct-mapped and read-only.

### *Cached/Uncached Accesses*



The data cache provides direct accesses (= uncached) to memory in order to access memory-mapped IO (like the processor-internal IO/peripheral modules). All accesses that target the address range from **0xF0000000** to **0xFFFFFFFF** will not be cached at all (see section [Address Space](#)).

### *Caching Internal Memories*



The data cache is intended to accelerate data access to **processor-external** memories. The CPU cache(s) should not be implemented when using only processor-internal data and instruction memories.

### *Manual Cache Clear/Reload*



By executing the **fence(.i)** instruction the cache is cleared and a reload from main memory is triggered.

### *Retrieve Cache Configuration from Software*



Software can retrieve the cache configuration/layout from the [SYSINFO - Cache Configuration](#) register.

### *Bus Access Fault Handling*



The cache always loads a complete cache block (aligned to the block size) every time a cache miss is detected. Each cached word from this block provides a single status bit that indicates if the according bus access was successful or caused a bus error. Hence, the whole cache block remains valid even if certain addresses inside caused a bus error. If the CPU accesses any of the faulty cache words, an instruction bus error exception is raised.

## 2.7.5. Processor-Internal Data Cache (dCACHE)

|                          |  |  |
|--------------------------|--|--|
| Hardware source file(s): | neorv32_cache.vhd  | Generic cache module   |
| Software driver file(s): | none   | <i>implicitly used</i>   |
| Top entity port:         | none   |  |
| Configuration generics:  | <b>DCACHE_EN</b><br><b>DCACHE_NUM_BLOCKS</b><br><b>DCACHE_BLOCK_SIZE</b> | implement processor-internal data cache when <b>true</b><br>number of cache blocks (pages/lines)<br>size of a cache block in bytes |
| CPU interrupts:          | none   |  |

The processor features an optional data cache to improve performance when using memories with high access latencies. The cache is connected directly to the CPU's data access interface and provides full-transparent accesses. The cache is direct-mapped and uses "write-allocate" and "write-back" strategies.

### *Cached/Uncached Accesses*



The data cache provides direct accesses (= uncached) to memory in order to access memory-mapped IO (like the processor-internal IO/peripheral modules). All accesses that target the address range from **0xF0000000** to **0xFFFFFFFF** will not be cached at all (see section [Address Space](#)).

### *Caching Internal Memories*



The data cache is intended to accelerate data access to **processor-external** memories. The CPU cache(s) should not be implemented when using only processor-internal data and instruction memories.

### *Manual Cache Flush/Clear/Reload*



By executing the **fence(.i)** instruction the cache is flushed, cleared and a reload from main memory is triggered.

### *Retrieve Cache Configuration from Software*



Software can retrieve the cache configuration/layout from the [SYSINFO - Cache Configuration](#) register.

### *Bus Access Fault Handling*



The cache always loads a complete cache block (aligned to the block size) every time a cache miss is detected. Each cached word from this block provides a single status bit that indicates if the according bus access was successful or caused a bus error. Hence, the whole cache block remains valid even if certain addresses inside caused a bus error. If the CPU accesses any of the faulty cache words, a data bus

error exception is raised.

## 2.7.6. Direct Memory Access Controller (DMA)

|                         |                                |   |
|-------------------------|--------------------------------|---|
| Hardware source files:  | neorv32_dma.vhd                |   |
| Software driver files:  | neorv32_dma.c<br>neorv32_dma.h |   |
| Top entity ports:       | none                           |   |
| Configuration generics: | <code>IO_DMA_EN</code>         | implement DMA when <code>true</code>                          |
| CPU interrupts:         | fast IRQ channel 10            | DMA transfer done (see <a href="#">Processor Interrupts</a> ) |

### Overview

The NEORV32 DMA provides a small-scale scatter/gather direct memory access controller that allows to transfer and modify data independently of the CPU. A single read/write transfer channel is implemented that is configured via memory-mapped registers. A configured transfer can either be triggered manually or by a programmable CPU FIRQ interrupt (see [NEORV32-Specific Fast Interrupt Requests](#)).

The DMA is connected to the central processor-internal bus system (see section [Address Space](#)) and can access the same address space as the CPU core. It uses *interleaving mode* accessing the central processor bus only if the CPU does not currently request and bus access.

The controller can handle different data quantities (e.g. read bytes and write them back as sign-extend words) and can also change the Endianness of data while transferring.



#### DMA Demo Program

A DMA example program can be found in [sw/example/demo\\_dma](#).

### Theory of Operation

The DMA provides four memory-mapped interface registers: A status and control register `CTRL` and three registers for configuring the actual DMA transfer. The base address of the source data is programmed via the `SRC_BASE` register. Vice versa, the base address of the destination data is programmed via the `DST_BASE`. The third configuration register `TTYPE` is used to configure the actual transfer type and the number of elements to transfer.

The DMA is enabled by setting the `DMA_CTRL_EN` bit of the control register. Manual trigger mode (i.e. the DMA transfer is triggered by writing to the `TTYPE` register) is selected if `DMA_CTRL_AUTO` is cleared. Alternatively, the DMA transfer can be triggered by a processor internal FIRQ signal if `DMA_CTRL_AUTO` is set (see section below).

The DMA uses a load-modify-write data transfer process. Data is read from the bus system, internally modified and then written back to the bus system. This combination is implemented as an atomic progress, so canceling the current transfer by clearing the `DMA_CTRL_EN` bit will stop the DMA right after the current load-modify-write operation.

If the DMA controller detects a bus error during operation, it will set either the `DMA_CTRL_ERROR_RD` (error during last read access) or `DMA_CTRL_ERROR_WR` (error during last write access) and will terminate the current transfer. Software can read the `SRC_BASE` or `DST_BASE` register to retrieve the address that caused the according error. Alternatively, software can read back the `NUM` bits of the control register to determine the index of the element that caused the error. The error bits are automatically cleared when starting a new transfer.

When the `DMA_CTRL_DONE` flag is set the DMA has actually executed a transfer. However, the `DMA_CTRL_ERROR_*` flags should also be checked to verify that the executed transfer completed without errors. The `DMA_CTRL_DONE` flag is automatically cleared when writing the `CTRL` register.

#### *DMA Access Privilege Level*



Transactions performed by the DMA are executed as bus transactions with elevated **machine-mode** privilege level. Additionally, all physical memory protection rules ([Smpmp ISA Extension](#)) defined by the CPU are **bypassed**.

## Transfer Configuration

If the DMA is set to **manual trigger mode** (`DMA_CTRL_AUTO` = 0) writing the `TTRIG` register will start the programmed DMA transfer. Once started, the DMA will read one data quantity from the source address, processes it internally and then will write it back to the destination address. The `DMA_TTYPE_NUM` bits of the `TTYPE` register define how many times this process is repeated by specifying the number of elements to transfer.

Optionally, the source and/or destination addresses can be increments according to the data quantities automatically by setting the according `DMA_TTYPE_SRC_INC` and/or `DMA_TTYPE_DST_INC` bit.

Four different transfer quantities are available, which are configured via the `DMA_TTYPE_QSEL` bits:

- **00**: Read source data as byte, write destination data as byte
- **01**: Read source data as byte, write destination data as zero-extended word
- **10**: Read source data as byte, write destination data as sign-extended word
- **11**: Read source data as word, write destination data as word

Optionally, the DMA controller can automatically convert Endianness of the transferred data if the `DMA_TTYPE_ENDIAN` bit is set.

#### *Address Alignment*



Make sure to align the source and destination base addresses to the according transfer data quantities. For instance, word-to-word transfers require that the two LSB of `SRC_BASE` and `DST_BASE` are cleared.

#### *Writing to IO Device*



When writing data to IO / peripheral devices (for example to the [Cyclic Redundancy Check \(CRC\)](#)) the destination data quantity has to be set to **word** (32-

bit) since all IO registers can only be written in full 32-bit word mode.

## Automatic Trigger

As an alternative to the manual trigger mode, the DMA can be configured to **automatic trigger mode** starting a pre-configured transfer if a specific processor-internal peripheral issues a FIRQ interrupt request. The automatic trigger mode is enabled by setting the `CTRL` register's `DMA_CTRL_AUTO` bit. In this configuration *no* transfer is started when writing to the DMA's `TTYPE` register.

The actually triggering FIRQ channel is configured via the control register's `DMA_CTRL_FIRQ_SEL` bits. Writing a 0 will select FIRQ channel 0, writing a 1 will select FIRQ channel 1, and so on. See section [Processor Interrupts](#) for a list of all FIRQ channels and their according sources.

### FIRQ Trigger



The DMA transfer will start if a **rising edge** is detected on the configured FIRQ channel. Hence, the DMA is triggered only once even if the selected FIRQ channel keeps pending.

## Memory Barrier / Fence Operation

Optionally, the DMA can issue a FENCE request to the downstream memory system when a transfer has been completed without errors. This can be used to re-sync caches (flush and reload) and buffers to maintain data coherency. This automatic fencing is enabled by the setting the control register's `DMA_CTRL_FENCE` bit.

## DMA Interrupt

The DMA features a single CPU interrupt that is triggered when the programmed transfer has completed. This interrupt is also triggered if the DMA encounters a bus error during operation. The interrupt will remain pending until the control register's `DMA_CTRL_DONE` is cleared (this will happen upon any write access to the control register).

## Register Map

Table 8. DMA Register Map (`struct NEORV32_DMA`)

| Address     | Name     | Bit(s), Name [C]                                    | R/W | Function  |
|-------------|----------|---|-----|---|
| 0xfffffed00 | CTRL     | 0 DMA_CTRL_EN                                       | r/w | DMA module enable   |
|             |          | 1 DMA_CTRL_AUTO                                     | r/w | Enable automatic mode (FIRQ-triggered)  |
|             |          | 2 DMA_CTRL_FENCE                                    | r/w | Issue a downstream FENCE operation when DMA transfer completes (without errors)     |
|             |          | 7:3 reserved  | r/- | reserved, read as zero  |
|             |          | 8 DMA_CTRL_ERROR_RD                                 | r/- | Error during read access, clears when starting a new transfer                       |
|             |          | 9 DMA_CTRL_ERROR_WR                                 | r/- | Error during write access, clears when starting a new transfer                      |
|             |          | 10 DMA_CTRL_BUSY                                    | r/- | DMA transfer in progress  |
|             |          | 11 DMA_CTRL_DONE                                    | r/c | Set if a transfer was executed; auto-clears on write-access                         |
|             |          | 15:12 reserved                                      | r/- | reserved, read as zero  |
|             |          | 19:16 DMA_CTRL_FIRQ_SEL_MSB : DMA_CTRL_FIRQ_SEL_LSB | r/w | FIRQ trigger select (FIRQ0=0 ... FIRQ15=15)   |
|             |          | 31:20 reserved                                      | r/- | reserved, read as zero  |
| 0xfffffed04 | SRC_BASE | 31:0  | r/w | Source base address (shows the last-accessed source address when read)              |
| 0xfffffed08 | DST_BASE | 31:0  | r/w | Destination base address (shows the last-accessed destination address when read)    |
| 0xfffffed0c | TTYPE    | 23:0 DMA_TTYPE_NUM_MSB : DMA_TTYPE_NUM_LSB          | r/w | Number of elements to transfer (shows the last-transferred element index when read) |
|             |          | 26:24 reserved                                      | r/- | reserved, read as zero  |
|             |          | 28:27 DMA_TTYPE_QSEL_MSB : DMA_TTYPE_QSEL_LSB       | r/w | Source data quantity select (00 = byte, 01 = half-word, 10 = word)                  |
|             |          | 29 DMA_TTYPE_SRC_INC                                | r/w | Constant (0) or incrementing (1) source address                                     |
|             |          | 30 DMA_TTYPE_DST_INC                                | r/w | Constant (0) or incrementing (1) destination address                                |
|             |          | 31 DMA_TTYPE_ENDIAN                                 | r/w | Swap Endianness when set  |

## 2.7.7. Processor-External Bus Interface (XBUS)

|                         |                                    |   |
|-------------------------|------------------------------------|---|
| Hardware source files:  | <code>neorv32_xbus.vhd</code>      | External bus gateway  |
|                         | <code>neorv32_cache.vhd</code>     | Generic cache module  |
| Software driver files:  | <code>none</code>                  | <i>implicitly used</i>  |
| Top entity ports:       | <code>xbus_adr_o</code>            | address output (32-bit)   |
|                         | <code>xbus_dat_i</code>            | data input (32-bit)   |
|                         | <code>xbus_dat_o</code>            | data output (32-bit)  |
|                         | <code>xbus_we_o</code>             | write enable (1-bit)  |
|                         | <code>xbus_sel_o</code>            | byte enable (4-bit)   |
|                         | <code>xbus_stb_o</code>            | bus strobe (1-bit)  |
|                         | <code>xbus_cyc_o</code>            | valid cycle (1-bit)   |
|                         | <code>xbus_ack_i</code>            | acknowledge (1-bit)   |
|                         | <code>xbus_err_i</code>            | bus error (1-bit)   |
| Configuration generics: | <code>XBUS_EN</code>               | enable external bus interface when <code>true</code>  |
|                         | <code>XBUS_TIMEOUT</code>          | number of clock cycles after which an unacknowledged external bus access will auto-terminate (0 = disabled) |
|                         | <code>XBUS_REGSTAGE_EN</code>      | implement XBUS register stages  |
|                         | <code>XBUS_CACHE_EN</code>         | implement the external bus cache  |
|                         | <code>XBUS_CACHE_NUM_BLOCKS</code> | number of blocks ("lines"), has to be a power of two.   |
|                         | <code>XBUS_CACHE_BLOCK_SIZE</code> | size in bytes of each block, has to be a power of two.  |
| CPU interrupts:         | <code>none</code>                  |   |

### Overview

The external bus interface provides a **Wishbone b4**-compatible on-chip bus interface that is implemented if the `XBUS_EN` generic is `true`. This bus interface can be used to attach processor-external modules like memories, custom hardware accelerators or additional peripheral devices. An optional cache module ("XCACHE") can be enabled to improve memory access latency.

#### *Address Mapping*



The external interface is **not** mapped to a specific address space. Instead, all CPU memory accesses that do not target a specific (and actually implemented) processor-internal address region (hence, accessing the "void"; see section [Address Space](#)) are **redirected** to the external bus interface.

## Wishbone Bus Protocol

The external bus interface complies to the **pipelined Wishbone b4** protocol. Even though this protocol was explicitly designed to support pipelined transfers, only a single transfer will be "in fly" at once. Hence, just two types of bus transactions are generated by the XBUS controller (see images below).

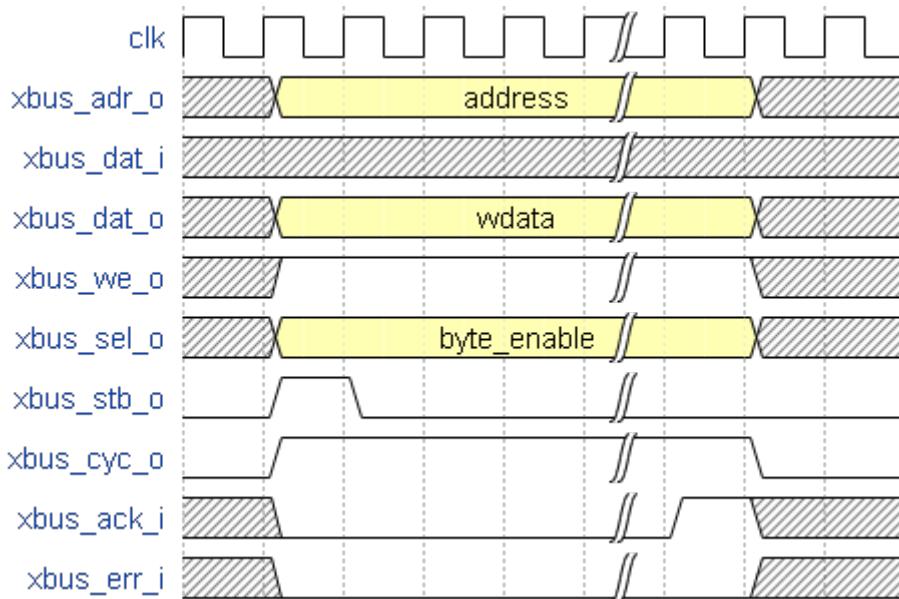


Figure 5. XBUS/Wishbone Write Transaction

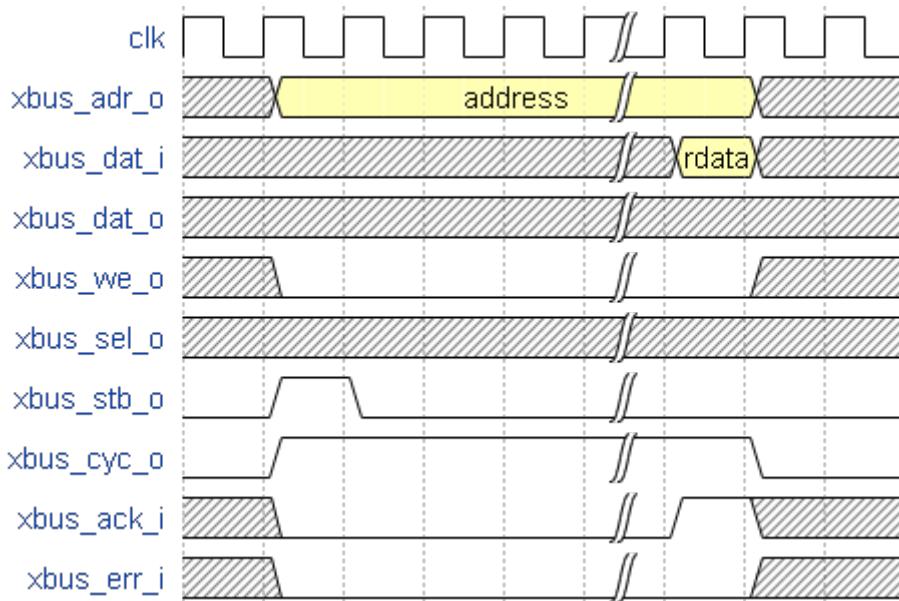


Figure 6. XBUS/Wishbone Read Transaction

### Wishbone "Classic" Protocol



Native support for the "classic" Wishbone protocol has been deprecated. However, classic mode can still be *emulated* by connecting the processor's `xbus_cyc_o` directly to the device's / bus system's `cyc` and `stb` signals (omitting the processor's `xbus_stb_o` signal).

*Endianness*

Just like the processor itself the XBUS interface uses **little-endian** byte order.

*Wishbone Specs.*

A detailed description of the implemented Wishbone bus protocol and the according interface signals can be found in the data sheet "Wishbone B4 - WISHBONE System-on-Chip (SoC) Interconnection Architecture for Portable IP Cores". A copy of this document can be found in the [docs](#) folder of this project.

An accessed XBUS/Wishbone device does not have to respond immediately to a bus request by sending an [ACK](#). Instead, there is a **time window** where the device has to acknowledge the transfer. This time window is configured by the [XBUS\\_TIMEOUT](#) generic and it defines the maximum time (in clock cycles) a bus access can be pending before it is automatically terminated raising an bus fault exception. If [XBUS\\_TIMEOUT](#) is set to zero, the timeout is disabled and a bus access can take an arbitrary number of cycles to complete. Note that this is not recommended as a missing ACK will permanently stall the entire processor!

Furthermore, an accessed XBUS/Wishbone device can signal an error condition at any time by setting the [ERR](#) signal high for one cycle. This will also terminate the current bus transaction before raising a CPU bus fault exception.

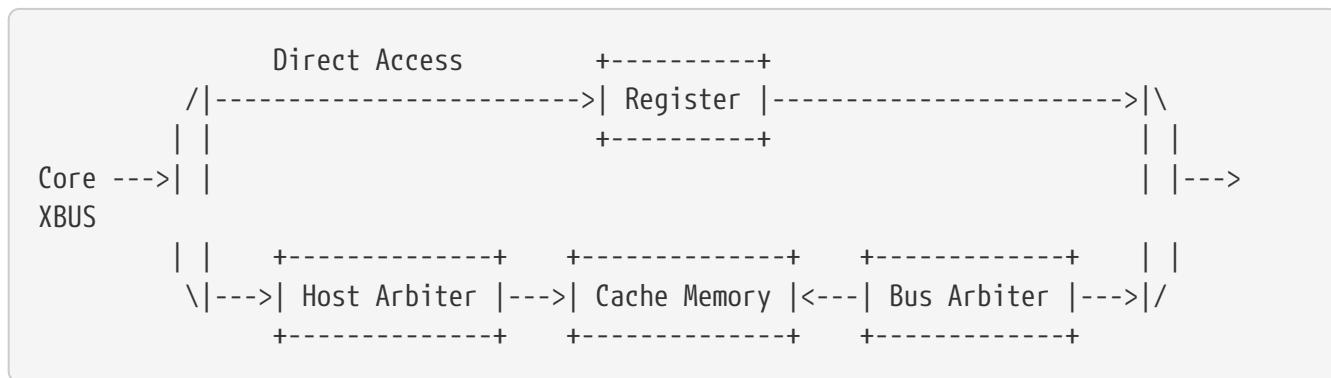
*Register Stage*

An optional register stage can be added to the XBUS gateway to break up the critical path easing timing closure. When [XBUS\\_REGSTAGE\\_EN](#) is *true* all outgoing and incoming XBUS signals are registered increasing access latency by two cycles. Furthermore, all outgoing signals (like the address) will be kept stable if there is no bus access being initiated.

## External Bus Cache (X-CACHE)

The XBUS interface provides an optional internal cache that can be used to buffer processor-external accesses. The x-cache is enabled via the [XBUS\\_CACHE\\_EN](#) generic. The total size of the cache is split into the number of cache lines or cache blocks ([XBUS\\_CACHE\\_NUM\\_BLOCKS](#) generic) and the line or block size in bytes ([XBUS\\_CACHE\\_BLOCK\\_SIZE](#) generic).

*Listing 4. Simplified X-Cache Architecture*



The cache uses a direct-mapped architecture that implements "write-allocate" and "write-back" strategies. The **write-allocate** strategy will fetch the entire referenced block from main memory when encountering a cache write-miss. The **write-back** strategy will gather all writes locally inside the cache until the according cache block is about to be replaced. In this case, the entire modified cache block is written back to main memory.

The x-cache also provides "direct accesses" that bypass the cache. For example, this can be used to access processor-external memory-mapped IO. All accesses that target the address range from `0xF0000000` to `0xFFFFFFFF` will always bypass the cache (see section [Address Space](#)). Furthermore, load-reservate and store conditional [Atomic Accesses](#) will also always bypass the cache **regardless of the accessed address**.

## 2.7.8. Stream Link Interface (SLINK)

|                         |                               |  |
|-------------------------|-------------------------------|--|
| Hardware source files:  | neorv32_slink.vhd             |  |
| Software driver files:  | neorv32_slink.c               |  |
|                         | neorv32_slink.h               |  |
| Top entity ports:       | <code>slink_rx_dat_i</code>   | RX link data (32-bit)                                    |
|                         | <code>slink_rx_val_i</code>   | RX link data valid (1-bit)                               |
|                         | <code>slink_rx_lst_i</code>   | RX link last element of stream (1-bit)                   |
|                         | <code>slink_rx_rdy_o</code>   | RX link ready to receive (1-bit)                         |
|                         | <code>slink_tx_dat_o</code>   | TX link data (32-bit)                                    |
|                         | <code>slink_tx_val_o</code>   | TX link data valid (1-bit)                               |
|                         | <code>slink_tx_lst_o</code>   | TX link last element of stream (1-bit)                   |
|                         | <code>slink_tx_rdy_i</code>   | TX link allowed to send (1-bit)                          |
| Configuration generics: | <code>IO_SLINK_EN</code>      | implement SLINK when <i>true</i>                         |
|                         | <code>IO_SLINK_RX_FIFO</code> | RX FIFO depth (1..32k), has to be a power of two, min 1  |
|                         | <code>IO_SLINK_TX_FIFO</code> | TX FIFO depth (1..32k), has to be a power of two, min 1  |
| CPU interrupts:         | fast IRQ channel 14           | RX SLINK IRQ (see <a href="#">Processor Interrupts</a> ) |
|                         | fast IRQ channel 15           | TX SLINK IRQ (see <a href="#">Processor Interrupts</a> ) |

### Overview

The stream link interface provides independent RX and TX channels for sending and receiving stream data. Each channel features a configurable internal FIFO to buffer stream data (`IO_SLINK_RX_FIFO` for the RX FIFO, `IO_SLINK_TX_FIFO` for the TX FIFO). The SLINK interface provides higher bandwidth and less latency than the external bus interface making it ideally suited for coupling custom stream processors or streaming peripherals.



#### Example Program

An example program for the SLINK module is available in [sw/example/demo\\_slink](#).

### Interface & Protocol

The SLINK interface consists of four signals for each channel:

- `dat` contains the actual data word
- `val` marks the current transmission cycle as valid

- `lst` marks the current transmission cycle as the last element of a stream
- `rdy` indicates that the receiver is ready to receive

#### *AXI4-Stream Compatibility*



The interface names and the underlying protocol is compatible to the AXI4-Stream protocol standard.

## Theory of Operation

The SLINK provides four interface registers. The control register (`CTRL`) is used to configure the module and to check its status. Two individual data registers (`DATA` and `DATA_LAST`) are used to send and receive the link's actual data stream.

The `DATA` register provides direct access to the RX/TX FIFO buffers. Read accesses return data from the RX FIFO. After reading data from this register the control register's `SLINK_CTRL_RX_LAST` flag can be checked to determine if the according data word has been marked as "end of stream" via the `slink_rx_lst_i` signal (this signal is also buffered by the link's FIFO). Writing to the `DATA` register will immediately write to the TX link FIFO. When writing to the `TX_DATA_LAST` the according data word will also be marked as "end of stream" via the `slink_tx_lst_o` signal (this signal is also buffered by the link's FIFO).

The configured FIFO sizes can be retrieved by software via the control register's `SLINK_CTRL_RX_FIFO_*` and `SLINK_CTRL_TX_FIFO_*` bits.

The SLINK is globally activated by setting the control register's enable bit `SLINK_CTRL_EN`. Clearing this bit will reset all internal logic and will also clear both FIFOs. The FIFOs can also be cleared manually at any time by setting the `SLINK_CTRL_RX_CLR` and/or `SLINK_CTRL_TX_CLR` bits (these bits will auto-clear).

#### *FIFO Overflow*



Writing to the TX channel's FIFO while it is *full* will have no effect. Reading from the RX channel's FIFO while it is *empty* will also have no effect and will return the last received data word. There is no overflow indicator implemented yet.

The current status of the RX and TX FIFOs can be determined via the control register's `SLINK_CTRL_RX_*` and `SLINK_CTRL_TX_*` flags.

## Interrupts

The SLINK module provides two independent interrupt channels: one for RX events and one for TX events. The interrupt conditions are based on the according link's FIFO status flags and are configured via the control register's `SLINK_CTRL_IRQ_*` flags. The according interrupt will fire when the module is enabled (`SLINK_CTRL_EN`) and the selected interrupt conditions are met. Note that all enabled interrupt conditions are logically OR-ed per channel. If any enable interrupt conditions becomes active the interrupt will become pending until the interrupt-causing condition is resolved (e.g. by reading from the RX FIFO).

## Register Map

Table 9. SLINK register map (`struct NEORV32_SLINK`)

| Address     | Name [C]           | Bit(s)  | R/W | Function   |
|-------------|--------------------|---|-----|--|
| 0xfffffec00 | NEORV32_SLINK.CTRL | 0 SLINK_CTRL_EN                                       | r/w | SLINK global enable                                      |
|             |                    | 1 SLINK_CTRL_RX_CLR                                   | -w  | Clear RX FIFO when set (bit auto-clears)                 |
|             |                    | 2 SLINK_CTRL_TX_CLR                                   | -w  | Clear TX FIFO when set (bit auto-clears)                 |
|             |                    | 3 reserved  | r-  | <i>reserved</i> , read as zero                           |
|             |                    | 4 SLINK_CTRL_RX_LAST                                  | r-  | Last word read from RX_DATA is marked as "end of stream" |
|             |                    | 7:5 reserved  | r-  | <i>reserved</i> , read as zero                           |
|             |                    | 8 SLINK_CTRL_RX_EMPTY                                 | r-  | RX FIFO empty  |
|             |                    | 9 SLINK_CTRL_RX_HALF                                  | r-  | RX FIFO at least half full                               |
|             |                    | 10 SLINK_CTRL_RX_FULL                                 | r-  | RX FIFO full   |
|             |                    | 11 SLINK_CTRL_TX_EMPTY                                | r-  | TX FIFO empty  |
|             |                    | 12 SLINK_CTRL_TX_HALF                                 | r-  | TX FIFO at least half full                               |
|             |                    | 13 SLINK_CTRL_TX_FULL                                 | r-  | TX FIFO full   |
|             |                    | 15:14 reserved  | r-  | <i>reserved</i> , read as zero                           |
|             |                    | 16 SLINK_CTRL_IRQ_RX_NEMPTY                           | r/w | RX interrupt if RX FIFO not empty                        |
|             |                    | 17 SLINK_CTRL_IRQ_RX_HALF                             | r/w | RX interrupt if RX FIFO at least half full               |
|             |                    | 18 SLINK_CTRL_IRQ_RX_FULL                             | r/w | RX interrupt if RX FIFO full                             |
|             |                    | 19 SLINK_CTRL_IRQ_TX_EMPTY                            | r/w | TX interrupt if TX FIFO empty                            |
|             |                    | 20 SLINK_CTRL_IRQ_TX_NHALF                            | r/w | TX interrupt if TX FIFO not at least half full           |
|             |                    | 21 SLINK_CTRL_IRQ_TX_NFULL                            | r/w | TX interrupt if TX FIFO not full                         |
|             |                    | 23:22 reserved  | r-  | <i>reserved</i> , read as zero                           |
|             |                    | 27:24 SLINK_CTRL_RX_FIFO_MSB : SLINK_CTRL_RX_FIFO_LSB | r-  | $\log_2(\text{RX FIFO size})$                            |
|             |                    | 31:28 SLINK_CTRL_TX_FIFO_MSB : SLINK_CTRL_TX_FIFO_LSB | r-  | $\log_2(\text{TX FIFO size})$                            |
| 0xfffffec04 | -                  | 31:0  | -/- | <i>reserved</i>  |

| Address     | Name [C]                    | Bit(s) | R/W | Function   |
|-------------|-----------------------------|--------|-----|--|
| 0xfffffec08 | NEORV32_SLINK<br>.DATA      | 31:0   | r/w | Write data to TX FIFO; read data from RX FIFO                              |
| 0xfffffec0c | NEORV32_SLINK<br>.DATA_LAST | 31:0   | r/w | Write data to TX FIFO (and also set "last" signal); read data from RX FIFO |

## 2.7.9. General Purpose Input and Output Port (GPIO)

|                          |                                  |   |
|--------------------------|----------------------------------|---|
| Hardware source file(s): | neorv32_gpio.vhd                 |   |
| Software driver file(s): | neorv32_gpio.c<br>neorv32_gpio.h |   |
| Top entity port:         | <code>gpio_o</code>              | 64-bit parallel output port                       |
|                          | <code>gpio_i</code>              | 64-bit parallel input port                        |
| Configuration generics:  | <code>IO_GPIO_NUM</code>         | number of input/output pairs to implement (0..64) |
| CPU interrupts:          | none                             |   |

The general purpose parallel IO unit provides a simple parallel input and output port. These ports can be used chip-externally (for example to drive status LEDs, connect buttons, etc.) or chip-internally to provide control signals for other IP modules.

The actual number of input/output pairs is defined by the `IO_GPIO_NUM` generic. When set to zero, the GPIO module is excluded from synthesis and the output port `gpio_o` is tied to all-zero. If `IO_GPIO_NUM` is less than the maximum value of 64, only the LSB-aligned bits in `gpio_o` and `gpio_i` are actually connected while the remaining bits are tied to zero or are left unconnected, respectively.

### Access Atomicity



The GPIO modules uses two memory-mapped registers (each 32-bit) each for accessing the input and output signals. Since the CPU can only process 32-bit "at once" updating the entire output cannot be performed within a single clock cycle.

## Register Map

Table 10. GPIO unit register map (`struct NEORV32_GPIO`)

| Address                 | Name [C]               | Bit(s) | R/W | Function                        |
|-------------------------|------------------------|--------|-----|---------------------------------|
| <code>0xfffffc00</code> | <code>INPUT_LO</code>  | 31:0   | r/- | parallel input port pins 31:0   |
| <code>0xfffffc04</code> | <code>INPUT_HI</code>  | 31:0   | r/- | parallel input port pins 63:32  |
| <code>0xfffffc08</code> | <code>OUTPUT_LO</code> | 31:0   | r/w | parallel output port pins 31:0  |
| <code>0xfffffc0c</code> | <code>OUTPUT_HI</code> | 31:0   | r/w | parallel output port pins 63:32 |

## 2.7.10. Cyclic Redundancy Check (CRC)

|                          |  |
|--------------------------|--|
| Hardware source file(s): | neorv32_crc.vhd  |
| Software driver file(s): | neorv32_crc.c<br>neorv32_crc.h                         |
| Top entity port:         | none   |
| Configuration generics:  | <b>IO_CRC_EN</b> implement CRC module when <b>true</b> |
| CPU interrupts:          | none   |

### Overview

The cyclic redundancy check unit provides a programmable checksum computation module. The unit operates on single bytes and can either compute CRC8, CRC16 or CRC32 checksums based on an arbitrary polynomial and start value.

#### DMA Demo Program



A CRC example program (also using CPU-independent DMA transfers) can be found in [sw/example/crc\\_dma](#).

#### CPU-Independent Operation



The CRC unit can compute a checksum for an arbitrary memory array without any CPU overhead by using the processor's **Direct Memory Access Controller (DMA)**.

### Theory of Operation

The module provides four interface registers:

- **MODE**: selects either CRC8-, CRC16- or CRC32-mode
- **POLY**: programmable polynomial
- **DATA**: data input register (single bytes only)
- **SREG**: the CRC shift register; this register is used to define the start value and to obtain the final processing result

The **MODE**, **POLY** and **SREG** registers need to be programmed before the actual processing can be started. Writing a byte to **DATA** will update the current checksum in **SREG**.

#### Access Latency



Write access to the CRC module have an increased latency of 8 clock cycles. This additional latency ensures that the internal bit-serial processing of the current data byte has also been completed when the transfer is completed.

#### Data Size



For CRC8-mode only bits **7:0** of **POLY** and **SREG** are relevant; for CRC16-mode only bits **15:0** are used and for CRC32-mode the entire 32-bit of **POLY** and **SREG** are used.

## Register Map

Table 11. CRC Register Map (`struct NEORV32_CRC`)

| Address            | Name [C]    | Bit(s), Name [C] | R/W | Function  |
|--------------------|-------------|------------------|-----|---|
| <b>0xfffffee00</b> | <b>CTRL</b> | <b>1:0</b>       | r/w | CRC mode select ( <b>00</b> CRC8, <b>01</b> : CRC16, <b>10</b> : CRC32) |
|                    |             | <b>31:2</b>      | r/- | <i>reserved</i> , read as zero  |
| <b>0xfffffee04</b> | <b>POLY</b> | <b>31:0</b>      | r/w | CRC polynomial  |
| <b>0xfffffee08</b> | <b>DATA</b> | <b>7:0</b>       | r/w | data input (single byte)  |
|                    |             | <b>31:8</b>      | r/- | <i>reserved</i> , read as zero, writes are ignored                      |
| <b>0xfffffee0c</b> | <b>SREG</b> | <b>32:0</b>      | r/w | current CRC shift register value (set start value on write)             |

## 2.7.11. Watchdog Timer (WDT)

|                          |                                |   |
|--------------------------|--------------------------------|---|
| Hardware source file(s): | neorv32_wdt.vhd                |   |
| Software driver file(s): | neorv32_wdt.c<br>neorv32_wdt.h |   |
| Top entity port:         | none                           |   |
| Configuration generics:  | <code>IO_WDT_EN</code>         | implement watchdog when <code>true</code> |
| CPU interrupts:          | -                              | none                                      |

### Theory of Operation

The watchdog (WDT) provides a last resort for safety-critical applications. The WDT provides a "bark and bite" concept. The timeout counter first triggers an optional CPU interrupt ("bark") when reaching half of the programmed interval to inform the application of the imminent timeout. When the full timeout value is reached a system-wide hardware reset is generated ("bite"). The internal counter has to be reset explicitly by the application program every now and then to prevent a timeout.

### Configuration

The watchdog is enabled by setting the control register's `WDT_CTRL_EN` bit. When this bit is cleared, the internal timeout counter is reset to zero and no system reset can be triggered by this module.

The internal 32-bit timeout counter is clocked at 1/4096th of the processor's main clock ( $f_{WDT}[\text{Hz}] = f_{\text{main}}[\text{Hz}] / 4096$ ). Whenever this counter reaches the programmed timeout value (`WDT_CTRL_TIMEOUT` bits in the control register) a hardware reset is triggered.

The watchdog's timeout counter is reset ("feeding the watchdog") by writing the reset **PASSWORD** to the `RESET` register. The password is hardwired to hexadecimal `0x709D1AB3`.

#### *Watchdog Operation during Debugging*

 By default, the watchdog stops operation when the CPU enters debug mode and will resume normal operation after the CPU has left debug mode again. This will prevent an unintended watchdog timeout during a debug session. However, the watchdog can also be configured to keep operating even when the CPU is in debug mode by setting the control register's `WDT_CTRL_DBEN` bit.

#### *Watchdog Operation during CPU Sleep*

 By default, the watchdog stops operating when the CPU enters sleep mode. However, the watchdog can also be configured to keep operating even when the CPU is in sleep mode by setting the control register's `WDT_CTRL_SEN` bit.

### Configuration Lock

The watchdog control register can be *locked* to protect the current configuration from being

modified. The lock is activated by setting the `WDT_CTRL_LOCK` bit. In the locked state any write access to the control register is entirely ignored (see table below, "writable if locked"). However, read accesses to the control register as well as watchdog resets are further possible.

The lock bit can only be set if the WDT is already enabled (`WDT_CTRL_EN` is set). Furthermore, the lock bit can only be cleared again by a system-wide hardware reset.

## Strict Mode

The *strict operation mode* provides additional safety functions. If the strict mode is enabled by the `WDT_CTRL_STRICT` control register bit an **immediate hardware** reset is enforced if

- the `RESET` register is written with an incorrect password or
- the `CTRL` register is written and the `WDT_CTRL_LOCK` bit is set.

## Cause of last Hardware Reset

The cause of the last system hardware reset can be determined via the `WDT_CTRL_RCAUSE_*` bits:

- `0b00`: Reset caused by external reset signal/pin
- `0b01`: Reset caused by on-chip debugger
- `0b10`: Reset caused by watchdog

## Register Map

Table 12. WDT register map (`struct NEORV32_WDT`)

| Address    | Name [C] | Bit(s), Name [C]                                 | R/W | Reset value | Writable if locked | Function   |
|------------|----------|--|-----|-------------|--------------------|--|
| 0xfffffb00 | CTRL     | 0 WDT_CTRL_EN                                    | r/w | 0           | no                 | watchdog enable  |
|            |          | 1 WDT_CTRL_LOCK                                  | r/w | 0           | no                 | lock configuration when set, clears only on system reset, can only be set if enable bit is set already                     |
|            |          | 2 WDT_CTRL_DBEN                                  | r/w | 0           | no                 | set to allow WDT to continue operation even when CPU is in debug mode  |
|            |          | 3 WDT_CTRL_SEN                                   | r/w | 0           | no                 | set to allow WDT to continue operation even when CPU is in sleep mode  |
|            |          | 4 WDT_CTRL_STRICT                                | r/w | 0           | no                 | set to enable strict mode (force hardware reset if reset password is incorrect or if write access to locked CTRL register) |
|            |          | 6:5 WDT_CTRL_RCAUSE_HI : WDT_CTRL_RCAUSE_LO      | r/- | 0           | -                  | cause of last system reset; 0=external reset, 1=ocd-reset, 2=watchdog reset  |
|            |          | 7 -  | r/- | -           | -                  | <i>reserved</i> , reads as zero  |
|            |          | 31:8 WDT_CTRL_TIMEOUT_MSB : WDT_CTRL_TIMEOUT_LSB | r/w | 0           | no                 | 24-bit watchdog timeout value  |
| 0xfffffb04 | RESET    |  | -/w | -           | yes                | Write PASSWORD to reset WDT timeout counter ("feed the watchdog")  |

## 2.7.12. Machine System Timer (MTIME)

|                          |                           |   |
|--------------------------|---------------------------|---|
| Hardware source file(s): | neorv32_mtime.vhd         |   |
| Software driver file(s): | neorv32_mtime.c           |   |
|                          | neorv32_mtime.h           |   |
| Top entity port:         | <code>mtime_irq_i</code>  | RISC-V machine timer IRQ if internal one is <b>not</b> implemented  |
|                          | <code>mtime_time_o</code> | Current system time ( <code>TIME</code> register)                   |
| Configuration generics:  | <code>IO_MTIME_EN</code>  | implement machine timer when <code>true</code>                      |
| CPU interrupts:          | <code>MTI</code>          | machine timer interrupt (see <a href="#">Processor Interrupts</a> ) |

The MTIME module implements a memory-mapped machine system timer that is compatible to the RISC-V privileged specifications. The 64-bit system time is accessed via individual `TIME_LO` and `TIME_HI` registers. A 64-bit time compare register, which is accessible via individual `TIMECMP_LO` and `TIMECMP_HI` registers, can be used to configure the CPU's machine timer interrupt (`MTI`). The interrupt is triggered whenever `TIME` (high & low part) is greater than or equal to `TIMECMP` (high & low part). The interrupt remains active (=pending) until `TIME` becomes less than `TIMECMP` again (either by modifying `TIME` or `TIMECMP`). The current system time is available for other SoC modules via the top's `mtime_time_o` signal.



### Hardware Reset

After a hardware reset the `TIME` and `TIMECMP` register are reset to all-zero.



### External MTIME Interrupt

If the internal MTIME module is disabled (`IO_MTIME_EN = false`) the machine timer interrupt becomes available as external signal. The `mtime_irq_i` signal is level-triggered and high-active. Once set the signal has to stay high until the interrupt request is explicitly acknowledged (e.g. writing to a user-defined memory-mapped register).

## Register Map

Table 13. MTIME register map (`struct NEORV32_MTIME`)

| Address                   | Name [C]                | Bits | R/W | Function                |
|---------------------------|-------------------------|------|-----|-------------------------|
| <code>0xfffffff400</code> | <code>TIME_LO</code>    | 31:0 | r/w | system time, low word   |
| <code>0xfffffff404</code> | <code>TIME_HI</code>    | 31:0 | r/w | system time, high word  |
| <code>0xfffffff408</code> | <code>TIMECMP_LO</code> | 31:0 | r/w | time compare, low word  |
| <code>0xfffffff40c</code> | <code>TIMECMP_HI</code> | 31:0 | r/w | time compare, high word |

## 2.7.13. Primary Universal Asynchronous Receiver and Transmitter (UART0)

|                          |                                  |  |
|--------------------------|----------------------------------|--|
| Hardware source file(s): | neorv32_uart.vhd                 |  |
| Software driver file(s): | neorv32_uart.c<br>neorv32_uart.h |  |
| Top entity port:         | <code>uart0_txd_o</code>         | serial transmitter output                                |
|                          | <code>uart0_rxd_i</code>         | serial receiver input                                    |
|                          | <code>uart0_rts_o</code>         | flow control: RX ready to receive, low-active            |
|                          | <code>uart0_cts_i</code>         | flow control: RX ready to receive, low-active            |
| Configuration generics:  | <code>IO_UART0_EN</code>         | implement UART0 when <code>true</code>                   |
|                          | <code>UART0_RX_FIFO</code>       | RX FIFO depth (power of 2, min 1)                        |
|                          | <code>UART0_TX_FIFO</code>       | TX FIFO depth (power of 2, min 1)                        |
| CPU interrupts:          | fast IRQ channel 2               | RX interrupt   |
|                          | fast IRQ channel 3               | TX interrupt (see <a href="#">Processor Interrupts</a> ) |

### Overview

The NEORV32 UART provides a standard serial interface with independent transmitter and receiver channels, each equipped with a configurable FIFO. The transmission frame is fixed to **8N1**: 8 data bits, no parity bit, 1 stop bit. The actual transmission rate (Baud rate) is programmable via software. The module features two memory-mapped registers: `CTRL` and `DATA`. These are used for configuration, status check and data transfer.

#### Standard Console



All default example programs and software libraries of the NEORV32 software framework (including the bootloader and the runtime environment) use the primary UART (*UART0*) as default user console interface. Furthermore, *UART0* is used to implement the "standard consoles" (`STDIN`, `STDOUT` and `STDERR`).

### Theory of Operation

The module is enabled by setting the `UART_CTRL_EN` bit in the *UART0* control register `CTRL`. The Baud rate is configured via a 10-bit `UART_CTRL_BAUDx` baud divisor (`baud_div`) and a 3-bit `UART_CTRL_PRSCx` clock prescaler (`clock_prescaler`).

Table 14. *UART0* Clock Configuration

| <code>UART_CTRL_PRSCx</code>           | <code>0b000</code> | <code>0b001</code> | <code>0b010</code> | <code>0b011</code> | <code>0b100</code> | <code>0b101</code> | <code>0b110</code> | <code>0b111</code> |
|--|--------------------|--------------------|--------------------|--------------------|--------------------|--------------------|--------------------|--------------------|
| Resulting <code>clock_prescaler</code> | 2                  | 4                  | 8                  | 64                 | 128                | 1024               | 2048               | 4096               |

$$\text{Baud rate} = (f_{\text{main}}[\text{Hz}] / \text{clock_prescaler}) / (\text{baud\_div} + 1)$$

The control register's `UART_CTRL_RX_*` and `UART_CTRL_TX_*` flags provide information about the RX and TX FIFO fill level. Disabling the module via the `UART_CTRL_EN` bit will also clear these FIFOs.

A new TX transmission is started by writing to the `DATA` register. The transfer is completed when the `UART_CTRL_TX_BUSY` control register flag returns to zero. RX data is available when the `UART_CTRL_RX_NEMPTY` flag becomes set. The `UART_CTRL_RX_OVER` will be set if the RX FIFO overflows. This flag is cleared only by disabling the module via `UART_CTRL_EN`.

## UART Interrupts

The UART module provides independent interrupt channels for RX and TX. These interrupts are triggered by certain RX and TX FIFO levels. The actual configuration is programmed independently for the RX and TX interrupt channel via the control register's `UART_CTRL_IRQ_RX_*` and `UART_CTRL_IRQ_TX_*` bits:

1. **RX IRQ** The RX interrupt can be triggered by three different RX FIFO level states: If `UART_CTRL_IRQ_RX_NEMPTY` is set the interrupt fires if the RX FIFO is *not* empty (e.g. when incoming data is available). If `UART_CTRL_IRQ_RX_HALF` is set the RX IRQ fires if the RX FIFO is at least half-full. If `UART_CTRL_IRQ_RX_FULL` the interrupt fires if the RX FIFO is full. Note that all these programmable conditions are logically OR-ed (interrupt fires if any enabled conditions is true).
2. **TX IRQ** The TX interrupt can be triggered by two different TX FIFO level states: If `UART_CTRL_IRQ_TX_EMPTY` is set the interrupt fires if the TX FIFO is empty. If `UART_CTRL_IRQ_TX_NHALF` is set the interrupt fires if the TX FIFO is *not* at least half full. Note that all these programmable conditions are logically OR-ed (interrupt fires if any enabled conditions is true).

Once an UART interrupt has fired it remains pending until the actual cause of the interrupt is resolved; for example if just the `UART_CTRL_IRQ_RX_NEMPTY` bit is set, the RX interrupt will keep firing until the RX FIFO is empty again.



### RX/TX FIFO Size

Software can retrieve the configured sizes of the RX and TX FIFO via the according `UART_DATA_RX_FIFO_SIZE` and `UART_DATA_TX_FIFO_SIZE` bits from the `DATA` register.

## RTS/CTS Hardware Flow Control

The NEORV32 UART supports optional hardware flow control using the standard CTS `uart0_cts_i` ("clear to send") and RTS `uart0_rts_o` ("ready to send" / "ready to receive (RTR)") signals. Both signals are low-active. Hardware flow control is enabled by setting the `UART_CTRL_HWFC_EN` bit in the modules control register `CTRL`.

When hardware flow control is enabled:

1. The UART's transmitter will not start a new transmission until the `uart0_cts_i` signal goes low. During this time, the UART busy flag `UART_CTRL_TX_BUSY` remains set.
2. The UART will set `uart0_rts_o` signal low if the RX FIFO is **less than half full** (to have a wide safety margin). As long as this signal is low, the connected device can send new data.

`uart0_rts_o` is always low if the hardware flow-control is disabled. Disabling the UART (setting `UART_CTRL_EN` low) while having hardware flow-control enabled, will set `uart0_rts_o` high to signal that the UARt is not capable of receiving new data.



Note that RTS and CTS signaling can only be activated together. If the RTS handshake is not required the signal can be left unconnected. If the CTS handshake is not required it has to be tied to zero.

## Simulation Mode

The UART provides a *simulation-only* mode to dump console data as well as raw data directly to a file. When the simulation mode is enabled (by setting the `UART_CTRL_SIM_MODE` bit) there will be **no** physical transaction on the `uart0_txd_o` signal. Instead, all data written to the `DATA` register is immediately dumped to a file. Data written to `DATA[7:0]` will be dumped as ASCII chars to a file named `nerv32 uart0.sim_mode.text.out`. Additionally, the ASCII data is printed to the simulator console.

Both file are created in the simulation's home folder.

## Register Map

Table 15. *UART0 register map (struct NEORV32\_UART0)*

| Address    | Name [C] | Bit(s), Name [C]                          | R/W | Function  |
|------------|----------|---|-----|---|
| 0xfffff500 | CTRL     | 0 UART_CTRL_EN                            | r/w | UART enable                                       |
|            |          | 1 UART_CTRL_SIM_MODE                      | r/w | enable <b>simulation mode</b>                     |
|            |          | 2 UART_CTRL_HWFC_EN                       | r/w | enable RTS/CTS hardware flow-control              |
|            |          | 5:3 UART_CTRL_PRSC2 :<br>UART_CTRL_PRSC0  | r/w | Baud rate clock prescaler select                  |
|            |          | 15:6 UART_CTRL_BAUD9 :<br>UART_CTRL_BAUD0 | r/w | 12-bit Baud value configuration value             |
|            |          | 16 UART_CTRL_RX_NEMPTY                    | r/- | RX FIFO not empty                                 |
|            |          | 17 UART_CTRL_RX_HALF                      | r/- | RX FIFO at least half-full                        |
|            |          | 18 UART_CTRL_RX_FULL                      | r/- | RX FIFO full                                      |
|            |          | 19 UART_CTRL_TX_EMPTY                     | r/- | TX FIFO empty                                     |
|            |          | 20 UART_CTRL_TX_NHALF                     | r/- | TX FIFO not at least half-full                    |
|            |          | 21 UART_CTRL_TX_FULL                      | r/- | TX FIFO full                                      |
|            |          | 22<br>UART_CTRL_IRQ_RX_NEMPTY             | r/w | fire IRQ if RX FIFO not empty                     |
|            |          | 23 UART_CTRL_IRQ_RX_HALF                  | r/w | fire IRQ if RX FIFO at least half-full            |
|            |          | 24 UART_CTRL_IRQ_RX_FULL                  | r/w | fire IRQ if RX FIFO full                          |
|            |          | 25 UART_CTRL_IRQ_TX_EMPTY                 | r/w | fire IRQ if TX FIFO empty                         |
|            |          | 26 UART_CTRL_IRQ_TX_NHALF                 | r/w | fire IRQ if TX not at least half full             |
|            |          | 29:27 -                                   | r/- | <i>reserved</i> read as zero                      |
|            |          | 30 UART_CTRL_RX_OVER                      | r/- | RX FIFO overflow; cleared by disabling the module |
|            |          | 31 UART_CTRL_TX_BUSY                      | r/- | TX busy or TX FIFO not empty                      |

| Address    | Name [C] | Bit(s), Name [C]  | R/W | Function                       |
|------------|----------|---|-----|--------------------------------|
| 0xfffff504 | DATA     | 7:0 UART_DATA_RTX_MSB :<br>UART_DATA_RTX_LSB                                | r/w | receive/transmit data          |
|            |          | 11:8<br>UART_DATA_RX_FIFO_SIZE_MS<br>B :<br>UART_DATA_RX_FIFO_SIZE_LS<br>B  | r/- | log2(RX FIFO size)             |
|            |          | 15:12<br>UART_DATA_TX_FIFO_SIZE_MS<br>B :<br>UART_DATA_TX_FIFO_SIZE_LS<br>B | r/- | log2(TX FIFO size)             |
|            |          | 31:16   | r/- | <i>reserved</i> , read as zero |

## 2.7.14. Secondary Universal Asynchronous Receiver and Transmitter (UART1)

|                          |                                  |  |
|--------------------------|----------------------------------|--|
| Hardware source file(s): | neorv32_uart.vhd                 |  |
| Software driver file(s): | neorv32_uart.c<br>neorv32_uart.h |  |
| Top entity port:         | <code>uart1_txd_o</code>         | serial transmitter output                                |
|                          | <code>uart1_rxd_i</code>         | serial receiver input                                    |
|                          | <code>uart1_rts_o</code>         | flow control: RX ready to receive, low-active            |
|                          | <code>uart1_cts_i</code>         | flow control: TX ready to receive, low-active            |
| Configuration generics:  | <code>IO_UART1_EN</code>         | implement UART1 when <code>true</code>                   |
|                          | <code>UART1_RX_FIFO</code>       | RX FIFO depth (power of 2, min 1)                        |
|                          | <code>UART1_TX_FIFO</code>       | TX FIFO depth (power of 2, min 1)                        |
| CPU interrupts:          | fast IRQ channel 4               | RX interrupt   |
|                          | fast IRQ channel 5               | TX interrupt (see <a href="#">Processor Interrupts</a> ) |

### Overview

The secondary UART (UART1) is functionally identical to the primary UART ([Primary Universal Asynchronous Receiver and Transmitter \(UART0\)](#)). Obviously, UART1 uses different addresses for the control register (`CTRL`) and the data register (`DATA`). The register's bits/flags use the same bit positions and naming as for the primary UART. The RX and TX interrupts of UART1 are mapped to different CPU fast interrupt (FIRQ) channels.

### Simulation Mode

The secondary UART (UART1) provides the same simulation options as the primary UART (UART0). However, output data is written to UART1-specific file `neorv32_uart1.sim_mode.text.out`. This data is also printed to the simulator console.

### Register Map

Table 16. UART1 register map (`struct NEORV32_UART1`)

| Address                 | Name [C]          | Bit(s), Name [C] | R/W | Function      |
|-------------------------|-------------------|------------------|-----|---------------|
| <code>0xfffff600</code> | <code>CTRL</code> | ...              | ... | Same as UART0 |
| <code>0xfffff604</code> | <code>DATA</code> | ...              | ... | Same as UART0 |

## 2.7.15. Serial Peripheral Interface Controller (SPI)

|                         |                          |  |
|-------------------------|--------------------------|--|
| Hardware source files:  | neorv32_spi.vhd          |  |
| Software driver files:  | neorv32_spi.c            |  |
|                         | neorv32_spi.h            |  |
| Top entity ports:       | <code>spi_clk_o</code>   | 1-bit serial clock output  |
|                         | <code>spi_dat_o</code>   | 1-bit serial data output   |
|                         | <code>spi_dat_i</code>   | 1-bit serial data input  |
|                         | <code>spi_csn_o</code>   | 8-bit dedicated chip select output (low-active)                        |
| Configuration generics: | <code>IO_SPI_EN</code>   | implement SPI controller when <code>true</code>                        |
|                         | <code>IO_SPI_FIFO</code> | FIFO depth, has to be a power of two, min 1                            |
| CPU interrupts:         | fast IRQ channel 6       | configurable SPI interrupt (see <a href="#">Processor Interrupts</a> ) |

### Overview

The NEORV32 SPI transceiver module operates on 8-bit base, supports all 4 standard clock modes and provides up to 8 dedicated chip select signals via the top entity's `spi_csn_o` signal. An receive/transmit FIFO can be configured via the `IO_SPI_FIFO` generic to support block-based transmissions without CPU interaction.

The SPI module provides a single control register `CTRL` to configure the module and to check it's status and a single data register `DATA` for receiving/transmitting data.

#### *Host-Mode Only*



The NEORV32 SPI module only supports *host mode*. Transmission are initiated only by the processor's SPI module and not by an external SPI module. If you are looking for a *device-mode* serial peripheral interface (transactions initiated by an external host) check out the [Serial Data Interface Controller \(SDI\)](#).

### Theory of Operation

The SPI module is enabled by setting the `SPI_CTRL_EN` bit in the `CTRL` control register. No transfer can be initiated and no interrupt request will be triggered if this bit is cleared. Clearing this bit will reset the module, clear the FIFO and terminate any transfer being in process.

The data quantity to be transferred within a single data transmission is fixed to 8 bits. However, the total transmission length is left to the user: after asserting chip-select an arbitrary amount of 8-bit transmission can be made before de-asserting chip-select again.

A transmission is started when writing data to the transmitter FIFO via the `DATA` register. Note that

data always transferred MSB-first. The SPI operation is completed as soon as the `SPI_CTRL_BUSY` flag clears. Received data can be retrieved by reading the RX FIFO also via the `DATA` register. The control register's `SPI_CTRL_RX_AVAIL`, `SPI_CTRL_TX_EMPTY`, `SPI_CTRL_TX_NHALF` and `SPI_CTRL_TX_FULL` flags provide information regarding the RX/TX FIFO levels.

The SPI controller features 8 dedicated chip-select lines. These lines are controlled via the control register's `SPI_CTRL_CS_SELx` and `SPI_CTRL_CS_EN` bits. The 3-bit `SPI_CTRL_CS_SELx` bits are used to select one out of the eight dedicated chip select lines. As soon as `SPI_CTRL_CS_EN` is set the selected chip select line is activated (driven low). Note that disabling the SPI module via the `SPI_CTRL_EN` bit will also deactivate any currently activated chip select line.

### SPI Clock Configuration

The SPI module supports all standard SPI clock modes (0, 1, 2, 3), which are configured via the two control register bits `SPI_CTRL_CPHA` and `SPI_CTRL_CPOL`. The `SPI_CTRL_CPHA` bit defines the *clock phase* and the `SPI_CTRL_CPOL` bit defines the *clock polarity*.

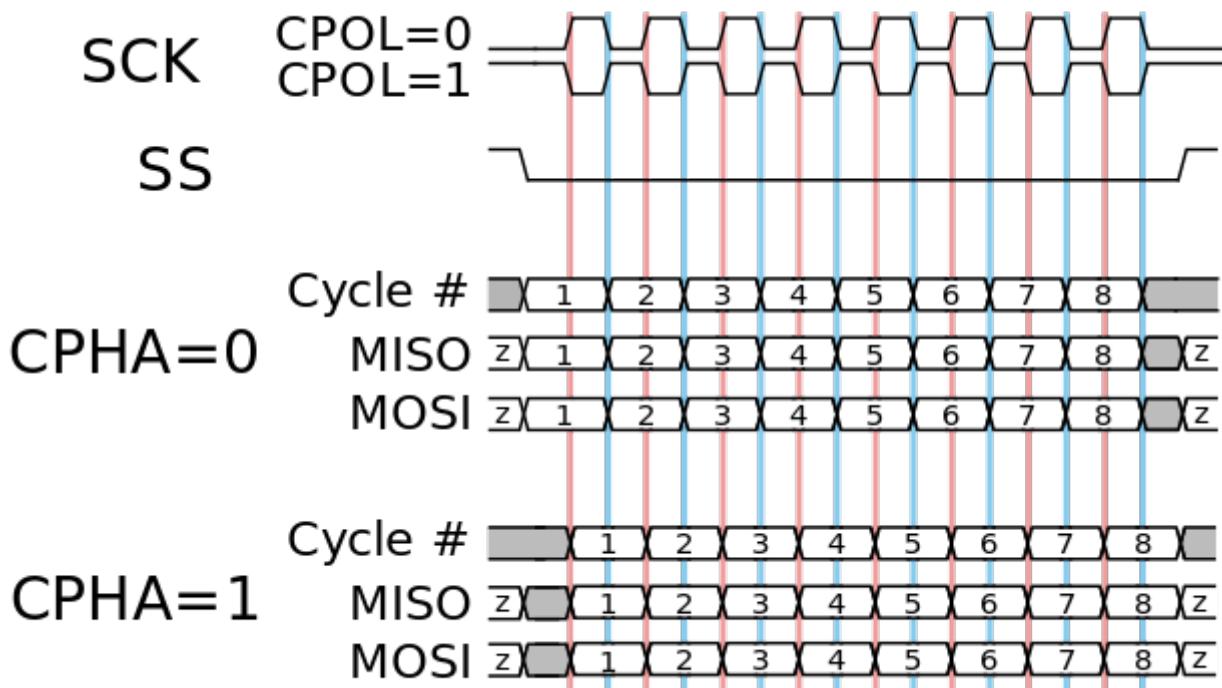


Figure 7. SPI clock modes; image from [https://en.wikipedia.org/wiki/File:SPI\\_timing\\_diagram2.svg](https://en.wikipedia.org/wiki/File:SPI_timing_diagram2.svg) (license: (Wikimedia) Creative Commons Attribution-Share Alike 3.0 Unported)

The SPI clock frequency (`spi_clk_o`) is programmed by the 3-bit `SPI_CTRL_PRSCx` clock prescaler for a coarse clock selection and a 4-bit clock divider `SPI_CTRL_CDIVx` for a fine clock configuration. The following clock prescalers (`SPI_CTRL_PRSCx`) are available:

Table 17. SPI prescaler configuration

| <code>SPI_CTRL_PRSCx</code>            | 0b000 | 0b001 | 0b010 | 0b011 | 0b100 | 0b101 | 0b110 | 0b111 |
|--|-------|-------|-------|-------|-------|-------|-------|-------|
| Resulting <code>clock_prescaler</code> | 2     | 4     | 8     | 64    | 128   | 1024  | 2048  | 4096  |

Based on the programmed clock configuration, the actual SPI clock frequency  $f_{\text{SPI}}$  is derived from

the processor's main clock  $f_{\text{main}}$  according to the following equation:

$$f_{\text{SPI}} = f_{\text{main}}[\text{Hz}] / (2 * \text{clock_prescaler} * (1 + \text{SPI_CTRL_CDIVx}))$$

Hence, the maximum SPI clock is  $f_{\text{main}} / 4$  and the lowest SPI clock is  $f_{\text{main}} / 131072$ . The SPI clock is always symmetric having a duty cycle of 50%.

## High-Speed Mode

The SPI provides a high-speed mode to further boost the maximum SPI clock frequency. When enabled via the control register's `SPI_CTRL_HIGHSPEED` bit the clock prescaler configuration (`SPI_CTRL_PRSCx` bits) is overridden setting it to a minimal factor of 1. However, the clock speed can still be fine-tuned using the `SPI_CTRL_CDIVx` bits.

$$f_{\text{SPI}} = f_{\text{main}}[\text{Hz}] / (2 * 1 * (1 + \text{SPI_CTRL_CDIVx}))$$

Hence, the maximum SPI clock when in high-speed mode is  $f_{\text{main}} / 2$ .

## SPI Interrupt

The SPI module provides a set of programmable interrupt conditions based on the level of the RX/TX FIFO. The different interrupt sources are enabled by setting the according control register's `SPI_CTRL_IRQ_*` bits. All enabled interrupt conditions are logically OR-ed so any enabled interrupt source will trigger the module's interrupt signal.

Once the SPI interrupt has fired it remains pending until the actual cause of the interrupt is resolved; for example if just the `SPI_CTRL_IRQ_RX_AVAIL` bit is set, the interrupt will keep firing until the RX FIFO is empty again.

## Register Map

Table 18. SPI register map (`struct NEORV32_SPI`)

| Address    | Name [C] | Bit(s), Name [C]                               | R/W | Function   |
|------------|----------|--|-----|--|
| 0xfffff800 | CTRL     | 0 SPI_CTRL_EN                                  | r/w | SPI module enable  |
|            |          | 1 SPI_CTRL_CPHA                                | r/w | clock phase  |
|            |          | 2 SPI_CTRL_CPOL                                | r/w | clock polarity   |
|            |          | 5:3 SPI_CTRL_CS_SEL2 :<br>SPI_CTRL_CS_SEL0     | r/w | Direct chip-select 0..7  |
|            |          | 6 SPI_CTRL_CS_EN                               | r/w | Direct chip-select enable: setting<br><code>spi_csn_o(SPI_CTRL_CS_SEL)</code> low when set     |
|            |          | 9:7 SPI_CTRL_PRSC2 :<br>SPI_CTRL_PRSC0         | r/w | 3-bit clock prescaler select   |
|            |          | 13:10 SPI_CTRL_CDIV2 :<br>SPI_CTRL_CDIV0       | r/w | 4-bit clock divider for fine-tuning  |
|            |          | 14 SPI_CTRL_HIGHSPEED                          | r/w | high-speed mode enable (overriding<br><code>SPI_CTRL_PRSC</code> )                             |
|            |          | 15 reserved                                    | r/- | reserved, read as zero   |
|            |          | 16 SPI_CTRL_RX_AVAIL                           | r/- | RX FIFO data available (RX FIFO not empty)   |
|            |          | 17 SPI_CTRL_TX_EMPTY                           | r/- | TX FIFO empty  |
|            |          | 18 SPI_CTRL_TX_NHALF                           | r/- | TX FIFO <i>not</i> at least half full  |
|            |          | 19 SPI_CTRL_TX_FULL                            | r/- | TX FIFO full   |
|            |          | 20 SPI_CTRL_IRQ_RX_AVAIL                       | r/w | Trigger IRQ if RX FIFO not empty   |
|            |          | 21 SPI_CTRL_IRQ_TX_EMPTY                       | r/w | Trigger IRQ if TX FIFO empty   |
|            |          | 22 SPI_CTRL_IRQ_TX_NHALF                       | r/w | Trigger IRQ if TX FIFO <i>not</i> at least half full   |
|            |          | 23 SPI_CTRL_IRQ_IDLE                           | r/w | Trigger IRQ if TX FIFO is empty and SPI bus<br>engine is idle                                  |
|            |          | 27:24 SPI_CTRL_FIFO_MSB :<br>SPI_CTRL_FIFO_LSB | r/- | FIFO depth; $\log_2(\text{IO\_SPI\_FIFO})$   |
|            |          | 30:28 reserved                                 | r/- | reserved, read as zero   |
|            |          | 31 SPI_CTRL_BUSY                               | r/- | SPI module busy when set (serial engine<br>operation in progress and TX FIFO not empty<br>yet) |
| 0xfffff804 | DATA     | 7:0  | r/w | receive/transmit data (FIFO)   |

## 2.7.16. Serial Data Interface Controller (SDI)

|                         |                          |  |
|-------------------------|--------------------------|--|
| Hardware source files:  | neorv32_sdi.vhd          |  |
| Software driver files:  | neorv32_sdi.c            |  |
|                         | neorv32_sdi.h            |  |
| Top entity ports:       | <code>sdi_clk_i</code>   | 1-bit serial clock input   |
|                         | <code>sdi_dat_o</code>   | 1-bit serial data output   |
|                         | <code>sdi_dat_i</code>   | 1-bit serial data input  |
|                         | <code>sdi_csn_i</code>   | 1-bit chip-select input (low-active)                                   |
| Configuration generics: | <code>IO_SDI_EN</code>   | implement SDI controller when <code>true</code>                        |
|                         | <code>IO_SDI_FIFO</code> | data FIFO size, has to a power of two, min 1                           |
| CPU interrupts:         | fast IRQ channel 11      | configurable SDI interrupt (see <a href="#">Processor Interrupts</a> ) |

### Overview

The serial data interface module provides a **device-class** SPI interface and allows to connect the processor to an external SPI *host*, which is responsible for triggering (clocking) the actual transmission - the SDI is entirely passive. An optional receive/transmit FIFO can be configured via the `IO_SDI_FIFO` generic to support block-based transmissions without CPU interaction.

#### *Device-Mode Only*



The NEORV32 SDI module only supports *device mode*. Transmission are initiated by an external host and not by the the processor itself. If you are looking for a *host-mode* serial peripheral interface (transactions initiated by the NEORV32) check out the [Serial Peripheral Interface Controller \(SPI\)](#).

The SDI module provides a single control register `CTRL` to configure the module and to check it's status and a single data register `DATA` for receiving/transmitting data.

### Theory of Operation

The SDI module is enabled by setting the `SDI_CTRL_EN` bit in the `CTRL` control register. Clearing this bit resets the entire module including the RX and TX FIFOs.

The SDI operates on byte-level only. Data written to the `DATA` register will be pushed to the TX FIFO. Received data can be retrieved by reading the RX FIFO via the `DATA` register. The current state of these FIFOs is available via the control register's `SDI_CTRL_RX_*` and `SDI_CTRL_TX_*` flags. The RX FIFO can be manually cleared at any time by setting the `SDI_CTRL_CLR_RX` bit.

If no data is available in the TX FIFO while an external device performs a transmission the external device will read all-zero from the SDI controller.

*MSB-first Only*

The NEORV32 SDI module only supports MSB-first mode.

*Transmission Abort*

If the external SPI controller aborts a transmission (by setting the chip-select signal high again) *before* 8 data bits have been transferred, no data is written to the RX FIFO.

## SDI Clocking

The SDI module supports both SPI clock polarity modes ("CPOL") but regarding the clock phase only "CPHA=0" is supported yet. All SDI operations are clocked by the external `sdi_clk_i` signal. This signal is synchronized to the processor's clock domain to simplify timing behavior. However, the clock synchronization requires that the external SDI clock (`sdi_clk_i`) does **not exceed 1/4 of the processor's main clock**.

## SDI Interrupt

The SDI module provides a set of programmable interrupt conditions based on the level of the RX & TX FIFOs. The different interrupt sources are enabled by setting the according control register's `SDI_CTRL_IRQ_*` bits. All enabled interrupt conditions are logically OR-ed so any enabled interrupt source will trigger the module's interrupt signal.

Once the SDI interrupt has fired it will remain active until the actual cause of the interrupt is resolved; for example if just the `SDI_CTRL_IRQ_RX_AVAIL` bit is set, the interrupt will keep firing until the RX FIFO is empty again.

## Register Map

Table 19. SDI register map (`struct NEORV32_SDI`)

| Address    | Name [C] | Bit(s), Name [C]                             | R/W | Function  |
|------------|----------|--|-----|---|
| 0xfffff700 | CTRL     | 0 SDI_CTRL_EN                                | r/w | SDI module enable                                   |
|            |          | 1 SDI_CTRL_CLR_RX                            | -/w | clear RX FIFO when set, bit auto-clears             |
|            |          | 3:2 reserved                                 | r/- | reserved, read as zero                              |
|            |          | 7:4 SDI_CTRL_FIFO_MSB :<br>SDI_CTRL_FIFO_LSB | r/- | FIFO depth; $\log_2(IO\_SDI\_FIFO)$                 |
|            |          | 14:8 reserved                                | r/- | reserved, read as zero                              |
|            |          | 15 SDI_CTRL_IRQ_RX_AVAIL                     | r/w | fire interrupt if RX FIFO is not empty              |
|            |          | 16 SDI_CTRL_IRQ_RX_HALF                      | r/w | fire interrupt if RX FIFO is at least half full     |
|            |          | 17 SDI_CTRL_IRQ_RX_FULL                      | r/w | fire interrupt if RX FIFO is full                   |
|            |          | 18 SDI_CTRL_IRQ_TX_EMPTY                     | r/w | fire interrupt if TX FIFO is empty                  |
|            |          | 19 SDI_CTRL_IRQ_TX_NHALF                     | r/w | fire interrupt if TX FIFO is not at least half full |
|            |          | 22:20 reserved                               | r/- | reserved, read as zero                              |
|            |          | 23 SDI_CTRL_RX_AVAIL                         | r/- | RX FIFO data available (RX FIFO not empty)          |
|            |          | 24 SDI_CTRL_RX_HALF                          | r/- | RX FIFO at least half full                          |
|            |          | 25 SDI_CTRL_RX_FULL                          | r/- | RX FIFO full  |
|            |          | 26 SDI_CTRL_TX_EMPTY                         | r/- | TX FIFO empty                                       |
|            |          | 27 SDI_CTRL_TX_NHALF                         | r/- | TX FIFO not at least half full                      |
|            |          | 28 SDI_CTRL_TX_FULL                          | r/- | TX FIFO full  |
|            |          | 31:29 reserved                               | r/- | reserved, read as zero                              |
| 0xfffff704 | DATA     | 7:0  | r/w | receive/transmit data (FIFO)                        |

## 2.7.17. Two-Wire Serial Interface Controller (TWI)

|                         |                    |  |
|-------------------------|--------------------|--|
| Hardware source files:  | neorv32_twi.vhd    |  |
| Software driver files:  | neorv32_twi.c      |  |
|                         | neorv32_twi.h      |  |
| Top entity ports:       | <b>twi_sda_i</b>   | 1-bit serial data line sense input   |
|                         | <b>twi_sda_o</b>   | 1-bit serial data line output (pull low only)                                    |
|                         | <b>twi_scl_i</b>   | 1-bit serial clock line sense input  |
|                         | <b>twi_scl_o</b>   | 1-bit serial clock line output (pull low only)                                   |
| Configuration generics: | <b>IO_TWI_EN</b>   | implement TWI controller when <b>true</b>  |
|                         | <b>IO_TWI_FIFO</b> | FIFO depth, has to be a power of two, min 1                                      |
| CPU interrupts:         | fast IRQ channel 7 | FIFO empty and module idle interrupt (see <a href="#">Processor Interrupts</a> ) |

### Overview

The NEORV32 TWI implements an I2C-compatible host controller to communicate with arbitrary I2C-devices. Note that peripheral-mode (controller acts as a device) and multi-controller mode are not supported yet.

The TWI controller provides two memory-mapped registers that are used for configuring the module and for triggering operation: **CTRL** is the control and status register, **DCMD** is the command and data register.

### Key features:

- Programmable clock speed
- Optional clock stretching
- Generate START / repeated-START and STOP conditions
- Sending & receiving 8 data bits including ACK/NACK
- Generating a host-ACK (ACK send by the TWI controller)
- Configurable data/command FIFO to "program" large TWI sequences without further involvement of the CPU

### Tristate Drivers

The TWI module requires two tristate drivers (actually: open-drain drivers; signals can only be actively driven low) for the SDA and SCL lines, which have to be implemented by the user in the setup's top module / IO ring. A generic VHDL example is shown below (here, **sda\_io** and **scl\_io** are

the actual TWI bus lines, which are of type `std_logic`).

*Listing 5. TWI VHDL Tristate Driver Example*

```
sda_io    <= '0' when (twi_sda_o = '0') else 'Z'; -- drive
scl_io    <= '0' when (twi_scl_o = '0') else 'Z'; -- drive
twi_sda_i <= std_ulogic(sda_io); -- sense
twi_scl_i <= std_ulogic(scl_io); -- sense
```

## TWI Clock Speed

The TWI clock frequency is programmed by two bit-fields in the device's control register `CTRL`: a 3-bit `TWI_CTRL_PRSCx` clock prescaler is used for a coarse clock configuration and a 4-bit clock divider `TWI_CTRL_CDIVx` is used for a fine clock configuration.

*Table 20. TWI prescaler configuration*

| <code>TWI_CTRL_PRSCx</code>            | 0b000 | 0b001 | 0b010 | 0b011 | 0b100 | 0b101 | 0b110 | 0b111 |
|--|-------|-------|-------|-------|-------|-------|-------|-------|
| Resulting <code>clock_prescaler</code> | 2     | 4     | 8     | 64    | 128   | 1024  | 2048  | 4096  |

Based on the clock configuration, the actual TWI clock frequency  $f_{SCL}$  is derived from the processor's main clock  $f_{main}$  according to the following equation:

$$f_{SCL} = f_{main}[Hz] / (4 * \text{clock\_prescaler} * (1 + \text{TWI\_CTRL\_CDIV}))$$

Hence, the maximum TWI clock is  $f_{main} / 8$  and the lowest TWI clock is  $f_{main} / 262144$ . The generated TWI clock is always symmetric having a duty cycle of exactly 50%.

### Clock Stretching



An accessed peripheral can slow down/halt the controller's bus clock by using clock stretching (= actively keeping the SCL line low). The controller will halt operation in this case. Clock stretching is enabled by setting the `TWI_CTRL_CLKSTR` bit in the module's control register `CTRL`.

## TWI Transfers

The TWI is enabled via the `TWI_CTRL_EN` bit in the `CTRL` control register. All TWI operations are controlled by the `DCMD` register. The actual operation is selected by a 2-bit value that is written to the register's `TWI_DCMD_CMD_*` bit-field:

- **00**: NOP (no-operation); all further bit-fields in `DCMD` are ignored
- **01**: Generate a (repeated) START conditions; all further bit-fields in `DCMD` are ignored
- **10**: Generate a STOP conditions; all further bit-fields in `DCMD` are ignored
- **11**: Trigger a data transmission; the data to be send has to be written to the register's `TWI_DCMD_MSB : TWI_DCMD_LSB` bit-field; if `TWI_DCMD_ACK` is set the controller will send a host-ACK in the ACK/NACK time slot; after the transmission is completed `TWI_DCMD_MSB : TWI_DCMD_LSB`

contains the RX data and `TWI_DCMD_ACK` the device's response if no host-ACK was configured (`0` = ACK, `1` = ACK)

All operations/data written to the `DCMD` register are buffered by a configurable data/command FIFO. The depth of the FIFO is configured by the `IO_TWI_FIFO` top generic. Software can retrieve this size by reading the control register's `TWI_CTRL_FIFO` bits.

The command/data FIFO is internally split into a TX FIFO and a RX FIFO. Writing to `DCMD` will write to the TX FIFO while reading from `DCMD` will read from the RX FIFO. The TX FIFO is full when the `TWI_CTRL_TX_FULL` flag is set. Accordingly, the RX FIFO contains valid data when the `TWI_CTRL_RX_AVAIL` flag is set.

The control register's busy flag `TWI_CTRL_BUSY` is set as long as the TX FIFO contains valid data (i.e. programmed TWI operations that have not been executed yet) or of the TWI bus engine is still processing an operation.



An active transmission can be terminated at any time by disabling the TWI module. This will also clear the data/command FIFO.



When reading data from a device, an all-one byte (`0xFF`) has to be written to TWI data register `NEORV32_TWI.DATA` so the accessed device can actively pull-down SDA when required.

## TWI Interrupt

The TWI module provides a single interrupt to signal "idle condition" to the CPU. The interrupt becomes active when the TWI module is enabled (`TWI_CTRL_EN = 1`) and the TX FIFO is empty and the TWI bus engine is idle.

## Register Map

Table 21. TWI register map (struct `NEORV32_TWI`)

| Address    | Name [C] | Bit(s), Name [C]                            | R/W | Function   |
|------------|----------|---|-----|--|
| 0xfffff900 | CTRL     | 0 TWI_CTRL_EN                               | r/w | TWI enable, reset if cleared   |
|            |          | 3:1 TWI_CTRL_PRSC2 : TWI_CTRL_PRSC0         | r/w | 3-bit clock prescaler select   |
|            |          | 7:4 TWI_CTRL_CDIV3 : TWI_CTRL_CDIV0         | r/w | 4-bit clock divider  |
|            |          | 8 TWI_CTRL_CLKSTR                           | r/w | Enable (allow) clock stretching  |
|            |          | 14:9 -                                      | r/- | <i>reserved</i> , read as zero   |
|            |          | 18:15 TWI_CTRL_FIFO_MSB : TWI_CTRL_FIFO_LSB | r/- | FIFO depth; log2( <a href="#">IO_TWI_FIFO</a> )  |
|            |          | 28:12 -                                     | r/- | <i>reserved</i> , read as zero   |
|            |          | 29 TWI_CTRL_TX_FULL                         | r/- | set if the TWI bus is claimed by any controller  |
|            |          | 30 TWI_CTRL_RX_AVAIL                        | r/- | RX FIFO data available   |
|            |          | 31 TWI_CTRL_BUSY                            | r/- | TWI bus engine busy or TX FIFO not empty   |
| 0xfffff904 | DCMD     | 7:0 TWI_DCMD_MSB : TWI_DCMD_LSB             | r/w | RX/TX data byte  |
|            |          | 8 TWI_DCMD_ACK                              | r/w | write: ACK bit sent by controller; read: 1 = device NACK, 0 = device ACK                     |
|            |          | 10:9 TWI_DCMD_CMD_HI : TWI_DCMD_CMD_LO      | r/w | TWI operation (00 = NOP, 01 = START conditions, 10 = STOP condition, 11 = data transmission) |

## 2.7.18. One-Wire Serial Interface Controller (ONEWIRE)

|                          |   |
|--------------------------|---|
| Hardware source file(s): | neorv32_onewire.vhd   |
| Software driver file(s): | neorv32_onewire.c<br>neorv32_onewire.h  |
| Top entity port:         | <b>onewire_i</b> 1-bit 1-wire bus sense input<br><b>onewire_o</b> 1-bit 1-wire bus output (pull low only) |
| Configuration generics:  | <b>IO_ONEWIRE_EN</b> implement ONEWIRE interface controller when <b>true</b>                              |
| CPU interrupts:          | fast IRQ channel 13      operation done interrupt (see <a href="#">Processor Interrupts</a> )             |

### Overview

The NEORV32 ONEWIRE module implements a single-wire interface controller that is compatible to the *Dallas/Maxim 1-Wire* protocol, which is an asynchronous half-duplex bus requiring only a single signal wire connected to **onewire\_io** (plus ground).

The bus is based on a single open-drain signal. The controller and all the devices can only pull-down the bus actively. Hence, an external pull-up resistor is required. Recommended values are between  $1\text{k}\Omega$  and  $4\text{k}\Omega$  depending on the bus characteristics (wire length, number of devices, etc.). Furthermore, a series resistor ( $\sim 100\Omega$ ) at the controller side is recommended to control the slew rate and to reduce signal reflections. Also, additional external ESD protection clamp diodes should be added to the bus line.

### Tri-State Drivers

The ONEWIRE module requires a tri-state driver (actually, open-drain) for the 1-wire bus line, which has to be implemented in the top module of the setup. A generic VHDL example is given below (**onewire** is the actual 1-wire bus signal, which is of type **std\_logic**).

*Listing 6. ONEWIRE VHDL tri-state driver example*

```
onewire  <= '0' when (onewire_o = '0') else 'Z'; -- drive
onewire_i <= std_ulogic(onewire); -- sense
```

### Theory of Operation

The ONEWIRE controller provides two interface registers: **CTRL** and **DATA**. The control registers (**CTRL**) is used to configure the module, to trigger bus transactions and to monitor the current state of the module. The **DATA** register is used to read/write data from/to the bus.

The module is enabled by setting the **ONEWIRE\_CTRL\_EN** bit in the control register. If this bit is cleared, the module is automatically reset and the bus is brought to high-level (due to the external pull-up resistor). The basic timing configuration is programmed via the clock prescaler bits

`ONEWIRE_CTRL_PRSCx` and the clock divider bits `ONEWIRE_CTRL_CLKDIVx` (see next section).

The controller can execute three basic bus operations, which are triggered by setting one out of three specific control register bits (the bits auto-clear):

1. generate reset pulse and check for device presence; triggered when setting `ONEWIRE_CTRL_TRIG_RST`
2. transfer a single-bit (read-while-write); triggered when setting `ONEWIRE_CTRL_TRIG_BIT`
3. transfer a full-byte (read-while-write); triggered when setting `ONEWIRE_CTRL_TRIG_BYTE`



Only one trigger bit may be set at once, otherwise undefined behavior might occur.

When a single-bit operation has been triggered, the data previously written to `DATA[0]` will be send to the bus and `DATA[7]` will be sampled from the bus. Accordingly, a full-byte transmission will send the previously byte written to `DATA[7:0]` to the bus and will update `DATA[7:0]` with the data read from the bus (LSB-first). The triggered operation has completed when the module's busy flag `ONEWIRE_CTRL_BUSY` has cleared again.

#### *Read from Bus*



In order to read a single bit from the bus `DATA[0]` has to be set to `1` before triggering the bit transmission operation to allow the accessed device to pull-down the bus. Accordingly, `DATA` has to be set to `0xFF` before triggering the byte transmission operation when the controller shall read a byte from the bus.

The `ONEWIRE_CTRL_PRESENCE` bit gets set if at least one device has send a "presence" signal right after the reset pulse.

## Bus Timing

The control register provides a 2-bit clock prescaler select (`ONEWIRE_CTRL_PRSCx`) and a 8-bit clock divider (`ONEWIRE_CTRL_CLKDIVx`) for timing configuration. Both are used to define the elementary **base time**  $T_{base}$ . All bus operations are timed using *multiples* of this elementary base time.

Table 22. ONEWIRE Clock Prescaler Configurations

| <code>ONEWIRE_CTRL_PRSCx</code>        | <code>0b00</code> | <code>0b01</code> | <code>0b10</code> | <code>0b11</code> |
|--|-------------------|-------------------|-------------------|-------------------|
| Resulting <code>clock_prescaler</code> | 2                 | 4                 | 8                 | 64                |

Together with the clock divider value (`ONEWIRE_CTRL_PRSCx` bits = `clock_divider`) the base time is defined by the following formula:

$$T_{base} = (1 / f_{main}[Hz]) * \text{clock\_prescaler} * (\text{clock\_divider} + 1)$$

Example:

- $f_{main} = 100\text{MHz}$
- clock prescaler select = `0b01` → `clock_prescaler` = 4

- clock divider `clock_divider` = 249

$$T_{base} = (1 / 100000000Hz) * 4 * (249 + 1) = 10000ns = 10\mu s$$

The base time is used to coordinate all bus interactions. Hence, all delays, time slots and points in time are quantized as multiples of the base time. The following images show the two basic operations of the ONEWIRE controller: single-bit (0 or 1) transaction and reset with presence detect. The relevant points in time are shown as *absolute* time (in multiples of the time base) with the bus' falling edge as reference point.

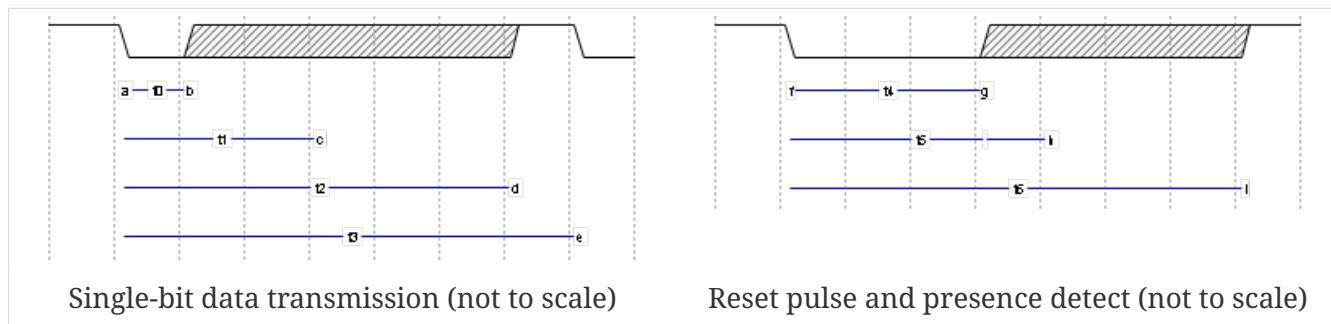


Table 23. Data Transmission Timing

| Symbol                                 | Description   | Multiples of $T_{base}$ | Time when $T_{base} = 10\mu s$ |
|--|---|-------------------------|--------------------------------|
| <b>Single-bit data transmission</b>    |   |                         |                                |
| $t_0$ (a → b)                          | Time until end of active low-phase when writing a '1' or when reading | 1                       | 10μs                           |
| $t_1$ (a → c)                          | Time until controller samples bus state (read operation)              | 2                       | 20μs                           |
| $t_2$ (a → d)                          | Time until end of bit time slot (when writing a '0' or when reading)  | 7                       | 70μs                           |
| $t_3$ (a → e)                          | Time until end of inter-slot pause (= total duration of one bit)      | 9                       | 90μs                           |
| <b>Reset pulse and presence detect</b> |   |                         |                                |
| $t_4$ (f → g)                          | Time until end of active reset pulse                                  | 48                      | 480μs                          |
| $t_5$ (f → h)                          | Time until controller samples bus presence                            | 55                      | 550μs                          |
| $t_6$ (f → i)                          | Time until end of presence phase                                      | 96                      | 960μs                          |



The default values for base time multiples were chosen to for stable and reliable bus operation (not for maximum throughput).

The absolute points in time are hardwired by the VHDL code and cannot be changed during runtime. However, the timing parameter can be customized by editing the ONEWIRE's VHDL source file:

*Listing 7. Hardwired time configuration in neorv32\_onewire.vhd*

```
-- timing configuration (absolute time in multiples of the base tick time t_base) --
constant t_write_one_c      : unsigned(6 downto 0) := to_unsigned( 1, 7); -- t0
constant t_read_sample_c    : unsigned(6 downto 0) := to_unsigned( 2, 7); -- t1
constant t_slot_end_c       : unsigned(6 downto 0) := to_unsigned( 7, 7); -- t2
constant t_pause_end_c      : unsigned(6 downto 0) := to_unsigned( 9, 7); -- t3
constant t_reset_end_c      : unsigned(6 downto 0) := to_unsigned(48, 7); -- t4
constant t_presence_sample_c : unsigned(6 downto 0) := to_unsigned(55, 7); -- t5
constant t_presence_end_c    : unsigned(6 downto 0) := to_unsigned(96, 7); -- t6
```

### *Overdrive*



The ONEWIRE controller does not support the *overdrive* mode. However, it can be implemented by reducing the base time  $T_{base}$  (and by eventually changing the hardwired timing configuration in the VHDL source file).

## Interrupt

A single interrupt is provided by the ONEWIRE module to signal "idle" condition to the CPU. Whenever the controller is idle (again) the interrupt becomes active.

## Register Map

*Table 24. ONEWIRE register map ([struct NEORV32\\_ONEWIRE](#))*

| Address    | Name [C] | Bit(s), Name [C]                                 | R/W | Function                                     |
|------------|----------|--|-----|--|
| 0xfffff200 | CTRL     | 0 ONEWIRE_CTRL_EN                                | r/w | ONEWIRE enable, reset if cleared             |
|            |          | 2:1 ONEWIRE_CTRL_PRSC1 : ONEWIRE_CTRL_PRSC0      | r/w | 2-bit clock prescaler select                 |
|            |          | 10:3 ONEWIRE_CTRL_CLKDIV7 : ONEWIRE_CTRL_CLKDIV0 | r/w | 8-bit clock divider value                    |
|            |          | 11 ONEWIRE_CTRL_TRIG_RST                         | -/w | trigger reset pulse, auto-clears             |
|            |          | 12 ONEWIRE_CTRL_TRIG_BIT                         | -/w | trigger single bit transmission, auto-clears |
|            |          | 13 ONEWIRE_CTRL_TRIG_BYTE                        | -/w | trigger full-byte transmission, auto-clears  |
|            |          | 28:14 -  | r/- | <i>reserved</i> , read as zero               |
|            |          | 29 ONEWIRE_CTRL_SENSE                            | r/- | current state of the bus line                |
|            |          | 30 ONEWIRE_CTRL_PRESENCE                         | r/- | device presence detected after reset pulse   |
|            |          | 31 ONEWIRE_CTRL_BUSY                             | r/- | operation in progress when set               |
| 0xfffff204 | DATA     | 7:0 ONEWIRE_DATA_MSB : ONEWIRE_DATA_LSB          | r/w | receive/transmit data (8-bit)                |

## 2.7.19. Pulse-Width Modulation Controller (PWM)

|                          |                                |
|--------------------------|--------------------------------|
| Hardware source file(s): | neorv32_pwm.vhd                |
| Software driver file(s): | neorv32_pwm.c<br>neorv32_pwm.h |
| Top entity port:         | <code>pwm_o</code>             |
| Configuration generics:  | <code>IO_PWM_NUM_CH</code>     |
| CPU interrupts:          | none                           |

### Overview

The PWM module implements a pulse-width modulation controller with up to 12 independent channels providing 8-bit resolution per channel. The actual number of implemented channels is defined by the `IO_PWM_NUM_CH` generic. Setting this generic to zero will completely remove the PWM controller from the design.



The `pwm_o` has a static size of 12-bit. If less than 12 PWM channels are configured, only the LSB-aligned channel bits are used while the remaining bits are hardwired to zero.

### Theory of Operation

The PWM controller is activated by setting the `PWM_CTRL_EN` bit in the module's control register `CTRL`. When this bit is cleared, the unit is reset and all PWM output channels are set to zero. The module provides three duty cycle registers `DC[0]` to `DC[2]`. Each register contains the duty cycle configuration for four consecutive channels. For example, the duty cycle of channel 0 is defined via bits 7:0 in `DC[0]`. The duty cycle of channel 2 is defined via bits 15:0 in `DC[0]` and so on.



Regardless of the configuration of `IO_PWM_NUM_CH` all module registers can be accessed without raising an exception. Software can discover the number of available channels by writing 0xff to all duty cycle configuration bytes and reading those values back. The duty-cycle of channels that were not implemented always reads as zero.

Based on the configured duty cycle the according intensity of the channel can be computed by the following formula:

$$\text{Intensity}_x = \text{DC}[y](i*8+7 \text{ downto } i*8) / (2^8)$$

The base frequency of the generated PWM signals is defined by the PWM core clock. This clock is derived from the main processor clock and divided by a prescaler via the 3-bit `PWM_CTRL_PRSCx` in the unit's control register.

Table 25. PWM prescaler configuration

|  |       |       |       |       |       |       |       |       |
|--|-------|-------|-------|-------|-------|-------|-------|-------|
| PWM_CTRL_PRSCx                         | 0b000 | 0b001 | 0b010 | 0b011 | 0b100 | 0b101 | 0b110 | 0b111 |
| Resulting <code>clock_prescaler</code> | 2     | 4     | 8     | 64    | 128   | 1024  | 2048  | 4096  |

The resulting PWM carrier frequency is defined by:

$$f_{PWM} = f_{main}[Hz] / (2^8 * \text{clock_prescaler})$$

## Register Map

Table 26. PWM register map (`struct nerv32_pwm_t`)

| Address    | Name  | Bit(s), Name [C]                    | R/W | Function                        |
|------------|-------|-------------------------------------|-----|---------------------------------|
| 0xfffff000 | CTRL  | 0 PWM_CTRL_EN                       | r/w | PWM enable                      |
|            |       | 3:1 PWM_CTRL_PRSC2 : PWM_CTRL_PRSC0 | r/w | 3-bit clock prescaler select    |
|            |       | 31:4 -                              | r/- | <i>reserved</i> , read as zero  |
| 0xfffff004 | DC[0] | 7:0                                 | r/w | 8-bit duty cycle for channel 0  |
|            |       | 15:8                                | r/w | 8-bit duty cycle for channel 1  |
|            |       | 23:16                               | r/w | 8-bit duty cycle for channel 2  |
|            |       | 31:24                               | r/w | 8-bit duty cycle for channel 3  |
| 0xfffff008 | DC[1] | 7:0                                 | r/w | 8-bit duty cycle for channel 4  |
|            |       | 15:8                                | r/w | 8-bit duty cycle for channel 5  |
|            |       | 23:16                               | r/w | 8-bit duty cycle for channel 6  |
|            |       | 31:24                               | r/w | 8-bit duty cycle for channel 7  |
| 0xfffff00c | DC[2] | 7:0                                 | r/w | 8-bit duty cycle for channel 8  |
|            |       | 15:8                                | r/w | 8-bit duty cycle for channel 9  |
|            |       | 23:16                               | r/w | 8-bit duty cycle for channel 10 |
|            |       | 31:24                               | r/w | 8-bit duty cycle for channel 11 |

## 2.7.20. True Random-Number Generator (TRNG)

|                          |                           |  |
|--------------------------|---------------------------|--|
| Hardware source file(s): | neorv32_trng.vhd          |  |
| Software driver file(s): | neorv32_trng.c            |  |
|                          | neorv32_trng.h            |  |
| Top entity port:         | none                      |  |
| Configuration generics:  | <code>IO_TRNG_EN</code>   | implement TRNG when <code>true</code>            |
|                          | <code>IO_TRNG_FIFO</code> | data FIFO depth, min 1, has to be a power of two |
| CPU interrupts:          | -                         | none   |

### Overview

The NEORV32 true random number generator provides *physically* true random numbers. It is based on free-running ring-oscillators that generate **phase noise** when being sampled by a constant clock. This phase noise is used as physical entropy source. The TRNG features a platform independent architecture without FPGA-specific primitives, macros or attributes so it can be synthesized for *any* FPGA.

#### *In-Depth Documentation*



For more information about the neoTRNG architecture and an analysis of its random quality check out the neoTRNG repository: <https://github.com/stnolting/neoTRNG>

#### *Inferring Latches*



The synthesis tool might emit warnings regarding **inferred latches** or **combinatorial loops**. However, this is not design flaw as this is exactly what we want. ;)

#### *Simulation*



When simulating the processor the TRNG is automatically set to "simulation mode". In this mode the physical entropy sources (the ring oscillators) are replaced by a simple **pseudo RNG** based on a LFSR providing only **deterministic pseudo-random** data. The `TRNG_CTRL_SIM_MODE` flag of the control register is set if simulation mode is active.

### Theory of Operation

The TRNG features a single control register `CTRL` for control, status check and data access. When the `TRNG_CTRL_EN` bit is set, the TRNG is enabled and starts operation. As soon as the `TRNG_CTRL_VALID` bit is set a new random data byte is available and can be obtained from the lowest 8 bits of the `CTRL` register. If this bit is cleared, there is no valid data available and the lowest 8 bit of the `CTRL` register are set to all-zero.

An internal entropy FIFO can be configured using the `IO_TRNG_FIFO` generic. This FIFO automatically samples new random data from the TRNG to provide some kind of *random data pool* for applications, which require a large number of random data in a short time. The random data FIFO can be cleared at any time either by disabling the TRNG or by setting the `TRNG_CTRL_FIFO_CLR` flag. The FIFO depth can be retrieved by software via the `TRNG_CTRL_FIFO_*` bits.

## Register Map

Table 27. TRNG register map (`struct NEORV32_TRNG`)

| Address    | Name<br>e [C] | Bit(s), Name [C]  | R/W | Function                                     |
|------------|---------------|---|-----|--|
| 0xfffffa00 | CTRL          | 7:0 <code>TRNG_CTRL_DATA_MSB</code> : <code>TRNG_CTRL_DATA_MSB</code>   | r/- | 8-bit random data                            |
|            |               | 15:8 -  | r/- | reserved, read as zero                       |
|            |               | 19:16 <code>TRNG_CTRL_FIFO_MSB</code> : <code>TRNG_CTRL_FIFO_MSB</code> | r/- | FIFO depth, $\log_2(\text{IO\_TRNG\_FIFO})$  |
|            |               | 27:20 -   | r/- | reserved, read as zero                       |
|            |               | 28 <code>TRNG_CTRL_FIFO_CLR</code>                                      | -/w | flush random data FIFO when set; auto-clears |
|            |               | 29 <code>TRNG_CTRL_SIM_MODE</code>                                      | r/- | simulation mode (PRNG!)                      |
|            |               | 30 <code>TRNG_CTRL_EN</code>  | r/w | TRNG enable                                  |
|            |               | 31 <code>TRNG_CTRL_VALID</code>   | r/- | random data is valid when set                |

## 2.7.21. Custom Functions Subsystem (CFS)

|                          |   |   |
|--------------------------|---|---|
| Hardware source file(s): | neorv32_cfs.vhd   |   |
| Software driver file(s): | neorv32_cfs.c<br>neorv32_cfs.h  |   |
| Top entity port:         | <code>cfs_in_i</code><br><code>cfs_out_o</code>   | custom input conduit<br>custom output conduit   |
| Configuration generics:  | <code>IO_CFS_EN</code><br><code>IO_CFS_CONFIG</code><br><code>IO_CFS_IN_SIZE</code><br><code>IO_CFS_OUT_SIZE</code> | implement CFS when <code>true</code><br>custom generic conduit<br>size of <code>cfs_in_i</code><br>size of <code>cfs_out_o</code> |
| CPU interrupts:          | fast IRQ channel 1  | CFS interrupt (see <a href="#">Processor Interrupts</a> )   |

### Theory of Operation

The custom functions subsystem is meant for implementing custom tightly-coupled co-processors or interfaces. It provides up to 64 32-bit memory-mapped read/write registers (`REG`, see register map below) that can be accessed by the CPU via normal load/store operations. The actual functionality of these register has to be defined by the hardware designer. Furthermore, the CFS provides two IO conduits to implement custom on-chip or off-chip interfaces.

Just like any other externally-connected IP, logic implemented within the custom functions subsystem can operate *independently* of the CPU providing true parallel processing capabilities. Potential use cases might include dedicated hardware accelerators for en-/decryption (AES), signal processing (FFT) or AI applications (CNNs) as well as custom IO systems like fast memory interfaces (DDR) and mass storage (SDIO), networking (CAN) or real-time data transport (I2S).



If you like to implement *custom instructions* that are executed right within the CPU's ALU see the [Zxcfu ISA Extension](#) and the according [Custom Functions Unit \(CFU\)](#).



Take a look at the template CFS VHDL source file ([rtl/core/neorv32\\_cfs.vhd](#)). The file is highly commented to illustrate all aspects that are relevant for implementing custom CFS-based co-processor designs.



The CFS can also be used to *replicate* existing NEORV32 modules - for example to implement several TWI controllers.

### CFS Software Access

The CFS memory-mapped registers can be accessed by software using the provided C-language aliases (see register map table below). Note that all interface registers are defined as 32-bit words of

type `uint32_t`.

*Listing 8. CFS Software Access Example*

```
// C-code CFS usage example
NEORV32_CFS->REG[0] = (uint32_t)some_data_array(i); // write to CFS register 0
int temp = (int)NEORV32_CFS->REG[20]; // read from CFS register 20
```

## CFS Interrupt

The CFS provides a single high-level-triggered interrupt request signal mapped to the CPU's fast interrupt channel 1.

## CFS Configuration Generic

By default, the CFS provides a single 32-bit `std::vector<std::logic>` configuration generic `IO_CFS_CONFIG` that is available in the processor's top entity. This generic can be used to pass custom configuration options from the top entity directly down to the CFS. The actual definition of the generic and its usage inside the CFS is left to the hardware designer.

## CFS Custom IOs

By default, the CFS also provides two unidirectional input and output conduits `cfs_in_i` and `cfs_out_o`. These signals are directly propagated to the processor's top entity. These conduits can be used to implement application-specific interfaces like memory or peripheral connections. The actual use case of these signals has to be defined by the hardware designer.

The size of the input signal conduit `cfs_in_i` is defined via the top's `IO_CFS_IN_SIZE` configuration generic (default = 32-bit). The size of the output signal conduit `cfs_out_o` is defined via the top's `IO_CFS_OUT_SIZE` configuration generic (default = 32-bit). If the custom function subsystem is not implemented (`IO_CFS_EN` = false) the `cfs_out_o` signal is tied to all-zero.

## Register Map

*Table 28. CFS register map (struct `NEORV32_CFS`)*

| Address                  | Name [C]             | Bit(s)            | R/W     | Function               |
|--------------------------|----------------------|-------------------|---------|------------------------|
| <code>0xfffffeb00</code> | <code>REG[0]</code>  | <code>31:0</code> | (r)/(w) | custom CFS register 0  |
| <code>0xfffffeb04</code> | <code>REG[1]</code>  | <code>31:0</code> | (r)/(w) | custom CFS register 1  |
| ...                      | ...                  | <code>31:0</code> | (r)/(w) | ...                    |
| <code>0xfffffeb8</code>  | <code>REG[62]</code> | <code>31:0</code> | (r)/(w) | custom CFS register 62 |
| <code>0xfffffebfc</code> | <code>REG[63]</code> | <code>31:0</code> | (r)/(w) | custom CFS register 63 |

## 2.7.22. Smart LED Interface (NEOLED)

|                          |                                |   |
|--------------------------|--------------------------------|---|
| Hardware source file(s): | neorv32_neoled.vhd             |   |
| Software driver file(s): | neorv32_neoled.c               |   |
|                          | neorv32_neoled.h               |   |
| Top entity port:         | <code>neoled_o</code>          | 1-bit serial data output  |
| Configuration generics:  | <code>IO_NEOLED_EN</code>      | implement NEOLED controller when <code>true</code>                                  |
|                          | <code>IO_NEOLED_TX_FIFO</code> | TX FIFO depth, has to be a power of 2, min 1  |
| CPU interrupts:          | fast IRQ channel 9             | configurable NEOLED data FIFO interrupt (see <a href="#">Processor Interrupts</a> ) |

### Overview

The NEOLED module provides a dedicated interface for "smart RGB LEDs" like WS2812, WS2811 or any other compatible LEDs. These LEDs provide a single-wire interface that uses an asynchronous serial protocol for transmitting color data. Using the NEOLED module allows CPU-independent operation of an arbitrary number of smart LEDs. A configurable data buffer (FIFO) allows to utilize block transfer operation without requiring the CPU.



The NEOLED interface is compatible to the "Adafruit Industries NeoPixel™" products, which feature WS2812 (or older WS2811) smart LEDs. Other LEDs might be compatible as well when adjusting the controller's programmable timing configuration.

The interface provides a single 1-bit output `neoled_o` to drive an arbitrary number of cascaded LEDs. Since the NEOLED module provides 24-bit and 32-bit operating modes, a mixed setup with RGB LEDs (24-bit color) and RGBW LEDs (32-bit color including a dedicated white LED chip) is possible.

### Theory of Operation

The NEOLED modules provides two accessible interface registers: the control register `CTRL` and the write-only TX data register `DATA`. The NEOLED module is globally enabled via the control register's `NEOLED_CTRL_EN` bit. Clearing this bit will terminate any current operation, clear the TX buffer, reset the module and set the `neoled_o` output to zero. The precise timing (e.g. implementing the **WS2812** protocol) and transmission mode are fully programmable via the `CTRL` register to provide maximum flexibility.

### RGB / RGBW Configuration

NeoPixel™ LEDs are available in two "color" version: LEDs with three chips providing RGB color and LEDs with four chips providing RGB color plus a dedicated white LED chip (= RGBW). Since the intensity of every LED chip is defined via an 8-bit value the RGB LEDs require a frame of 24-bit per

module and the RGBW LEDs require a frame of 32-bit per module.

The data transfer quantity of the NEOLED module can be programmed via the `NEOLED_MODE_EN` control register bit. If this bit is cleared, the NEOLED interface operates in 24-bit mode and will transmit bits `23:0` of the data written to `DATA` to the LEDs. If `NEOLED_MODE_EN` is set, the NEOLED interface operates in 32-bit mode and will transmit bits `31:0` of the data written to `DATA` to the LEDs.

The mode bit can be reconfigured before writing a new data word to `DATA` in order to support an arbitrary setup/mixture of RGB and RGBW LEDs.

## Protocol

The interface of the WS2812 LEDs uses an 800kHz carrier signal. Data is transmitted in a serial manner starting with LSB-first. The intensity for each R, G & B (& W) LED chip (= color code) is defined via an 8-bit value. The actual data bits are transferred by modifying the duty cycle of the signal (the timings for the WS2812 are shown below). A RESET command is "send" by pulling the data line LOW for at least 50μs.

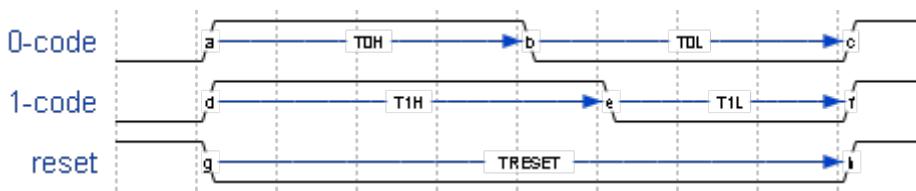


Figure 8. WS2812 bit-level timing (timing does not scale)

Table 29. WS2812 interface timing

|   |                                     |                                      |
|---|-------------------------------------|--------------------------------------|
| $T_{\text{total}} (T_{\text{carrier}})$ | $1.25\mu\text{s} +/- 300\text{ns}$  | period for a single bit              |
| $T_{0H}$                                | $0.4\mu\text{s} +/- 150\text{ns}$   | high-time for sending a 1            |
| $T_{0L}$                                | $0.8\mu\text{s} +/- 150\text{ns}$   | low-time for sending a 1             |
| $T_{1H}$                                | $0.85\mu\text{s} +/- 150\text{ns}$  | high-time for sending a 0            |
| $T_{1L}$                                | $0.45\mu\text{s} +/- 150\text{ ns}$ | low-time for sending a 0             |
| RESET                                   | Above $50\mu\text{s}$               | low-time for sending a RESET command |

## Timing Configuration

The basic carrier frequency (800kHz for the WS2812 LEDs) is configured via a 3-bit main clock prescaler (`NEOLED_CTRL_PRSC*`, see table below) that scales the main processor clock  $f_{\text{main}}$  and a 5-bit cycle multiplier `NEOLED_CTRL_T_TOT_*`.

Table 30. NEOLED Prescaler Configuration

| <code>NEOLED_CTRL_PRSCx</code>         | <code>0b000</code> | <code>0b001</code> | <code>0b010</code> | <code>0b011</code> | <code>0b100</code> | <code>0b101</code> | <code>0b110</code> | <code>0b111</code> |
|--|--------------------|--------------------|--------------------|--------------------|--------------------|--------------------|--------------------|--------------------|
| Resulting <code>clock_prescaler</code> | 2                  | 4                  | 8                  | 64                 | 128                | 1024               | 2048               | 4096               |

The duty-cycles (or more precisely: the high- and low-times for sending either a '1' bit or a '0' bit) are defined via the 5-bit `NEOLED_CTRL_T_ONE_H_*` and `NEOLED_CTRL_T_ZERO_H_*` values, respectively.

These programmable timing constants allow to adapt the interface for a wide variety of smart LED protocol (for example WS2812 vs. WS2811).

### Timing Configuration - Example (WS2812)

Generate the base clock  $f_{TX}$  for the NEOLED TX engine:

- processor clock  $f_{main} = 100$  MHz
- $\text{NEOLED\_CTRL\_PRSCx} = \text{0b001} = f_{main} / 4$

$$f_{TX} = f_{main}[\text{Hz}] / \text{clock_prescaler} = 100\text{MHz} / 4 = 25\text{MHz}$$

$$T_{TX} = 1 / f_{TX} = 40\text{ns}$$

Generate carrier period ( $T_{carrier}$ ) and **high-times** (duty cycle) for sending **0** ( $T_{0H}$ ) and **1** ( $T_{1H}$ ) bits:

- $\text{NEOLED\_CTRL\_T\_TOT} = \text{0b11110}$  (= decimal 30)
- $\text{NEOLED\_CTRL\_T\_ZERO\_H} = \text{0b01010}$  (= decimal 10)
- $\text{NEOLED\_CTRL\_T\_ONE\_H} = \text{0b10100}$  (= decimal 20)

$$T_{carrier} = T_{TX} * \text{NEOLED\_CTRL\_T\_TOT} = 40\text{ns} * 30 = 1.4\mu\text{s}$$

$$T_{0H} = T_{TX} * \text{NEOLED\_CTRL\_T\_ZERO\_H} = 40\text{ns} * 10 = 0.4\mu\text{s}$$

$$T_{1H} = T_{TX} * \text{NEOLED\_CTRL\_T\_ONE\_H} = 40\text{ns} * 20 = 0.8\mu\text{s}$$



The NEOLED SW driver library ([neorv32\\_neoled.h](#)) provides a simplified configuration function that configures all timing parameters for driving WS2812 LEDs based on the processor clock frequency.

### TX Data FIFO

The interface features a configurable TX data buffer (a FIFO) to allow more CPU-independent operation. The buffer depth is configured via the `IO_NEOLED_TX_FIFO` top generic (default = 1 entry). The FIFO size configuration can be read via the `NEOLED_CTRL_BUFS_x` control register bits, which result  $\log_2(\text{IO_NEOLED\_TX\_FIFO})$ .

When writing data to the `DATA` register the data is automatically written to the TX buffer. Whenever data is available in the buffer the serial transmission engine will take and transmit it to the LEDs. The data transfer size (`NEOLED_MODE_EN`) can be modified at any time since this control register bit is also buffered in the FIFO. This allows an arbitrary mix of RGB and RGBW LEDs in the chain.

Software can check the FIFO fill level via the control register's `NEOLED_CTRL_TX_EMPTY`, `NEOLED_CTRL_TX_HALF` and `NEOLED_CTRL_TX_FULL` flags. The `NEOLED_CTRL_TX_BUSY` flag provides additional information if the the serial transmit engine is still busy sending data.



Please note that the timing configurations (`NEOLED_CTRL_PRSCx`, `NEOLED_CTRL_T_TOT_x`, `NEOLED_CTRL_T_ONE_H_x` and `NEOLED_CTRL_T_ZERO_H_x`) are NOT stored to the buffer.

Changing these value while the buffer is not empty or the TX engine is still busy will cause data corruption.

### Strobe Command ("RESET")

According to the WS2812 specs the data written to the LED's shift registers is strobed to the actual PWM driver registers when the data line is low for 50µs ("RESET" command, see table above). This can be implemented using busy-wait for at least 50µs. Obviously, this concept wastes a lot of processing power.

To circumvent this, the NEOLED module provides an option to automatically issue an idle time for creating the RESET command. If the `NEOLED_CTRL_STROBE` control register bit is set, *all* data written to the data FIFO (via `DATA`, the actually written data is irrelevant) will trigger an idle phase (`neoled_o` = zero) of 127 periods (=  $T_{carrier}$ ). This idle time will cause the LEDs to strobe the color data into the PWM driver registers.

Since the `NEOLED_CTRL_STROBE` flag is also buffered in the TX buffer, the RESET command is treated just as another data word being written to the TX buffer making busy wait concepts obsolete and allowing maximum refresh rates.

### NEOLED Interrupt

The NEOLED modules features a single interrupt that triggers based on the current TX buffer fill level. The interrupt can only become pending if the NEOLED module is enabled. The specific interrupt condition is configured via the `NEOLED_CTRL_IRQ_CONF` bit in the unit's control register.

If `NEOLED_CTRL_IRQ_CONF` is set, the module's interrupt is generated whenever the TX FIFO is less than half-full. In this case software can write up to `IO_NEOLED_TX_FIFO/2` new data words to `DATA` without checking the FIFO status flags. If `NEOLED_CTRL_IRQ_CONF` is cleared, an interrupt is generated when the TX FIFO is empty.

Once the NEOLED interrupt has fired it remains pending until the actual cause of the interrupt is resolved.

### Register Map

Table 31. NEOLED register map (`struct NEORV32_NEOLED`)

| Address    | Name [C] | Bit(s), Name [C]                                      | R/W | Function  |
|------------|----------|---|-----|---|
| 0xfffffd00 | CTRL     | 0 NEOLED_CTRL_EN                                      | r/w | NEOLED enable   |
|            |          | 1 NEOLED_CTRL_MODE                                    | r/w | data transfer size; 0=24-bit; 1=32-bit  |
|            |          | 2 NEOLED_CTRL_STROBE                                  | r/w | 0=send normal color data; 1=send RESET command on data write access                           |
|            |          | 5:3 NEOLED_CTRL_PRSC2 : NEOLED_CTRL_PRSC0             | r/w | 3-bit clock prescaler, bit 0  |
|            |          | 9:6 NEOLED_CTRL_BUFS3 : NEOLED_CTRL_BUFS0             | r/- | 4-bit log2( $IO\_NEOLED\_TX\_FIFO$ )  |
|            |          | 14:10 NEOLED_CTRL_T_TOT_4 : NEOLED_CTRL_T_TOT_0       | r/w | 5-bit pulse clock ticks per total single-bit period ( $T_{total}$ )                           |
|            |          | 19:15 NEOLED_CTRL_T_ZERO_H_4 : NEOLED_CTRL_T_ZERO_H_0 | r/w | 5-bit pulse clock ticks per high-time for sending a zero-bit ( $T_{0H}$ )                     |
|            |          | 24:20 NEOLED_CTRL_T_ONE_H_4 : NEOLED_CTRL_T_ONE_H_0   | r/w | 5-bit pulse clock ticks per high-time for sending a one-bit ( $T_{1H}$ )                      |
|            |          | 27 NEOLED_CTRL_IRQ_CONF                               | r/w | TX FIFO interrupt configuration: 0=IRQ if FIFO is empty, 1=IRQ if FIFO is less than half-full |
|            |          | 28 NEOLED_CTRL_TX_EMPTY                               | r/- | TX FIFO is empty  |
|            |          | 29 NEOLED_CTRL_TX_HALF                                | r/- | TX FIFO is <i>at least</i> half full  |
|            |          | 30 NEOLED_CTRL_TX_FULL                                | r/- | TX FIFO is full   |
|            |          | 31 NEOLED_CTRL_TX_BUSY                                | r/- | TX serial engine is busy when set   |
| 0xfffffd04 | DATA     | 31:0 / 23:0   | -/w | TX data (32- or 24-bit, depending on <i>NEOLED_CTRL_MODE</i> bit)                             |

## 2.7.23. External Interrupt Controller (XIRQ)

|                          |                                    |  |
|--------------------------|------------------------------------|--|
| Hardware source file(s): | neorv32_xirq.vhd                   |  |
| Software driver file(s): | neorv32_xirq.c                     |  |
|                          | neorv32_xirq.h                     |  |
| Top entity port:         | <code>xirq_i</code>                | External interrupts input (32-bit)                   |
| Configuration generics:  | <code>XIRQ_NUM_CH</code>           | Number of external IRQ channels to implement (0..32) |
|                          | <code>XIRQ_TRIGGER_TYPE</code>     | IRQ trigger type configuration                       |
|                          | <code>XIRQ_TRIGGER_POLARITY</code> | IRQ trigger polarity configuration                   |
| CPU interrupts:          | fast IRQ channel 8                 | XIRQ (see <a href="#">Processor Interrupts</a> )     |

### Overview

The external interrupt controller provides a simple mechanism to implement up to 32 processor-external interrupt request signals. The external IRQ requests are prioritized, queued and signaled to the CPU via a *single* CPU fast interrupt request.

### Theory of Operation

The XIRQ provides up to 32 external interrupt channels configured via the `XIRQ_NUM_CH` generic. Each bit in the `xirq_i` input signal vector represents one interrupt channel. If less than 32 channels are configured, only the LSB-aligned channels are used while the remaining ones are left unconnected internally. The actual interrupt trigger type is configured before synthesis using the `XIRQ_TRIGGER_TYPE` and `XIRQ_TRIGGER_POLARITY` generics (see table below).

Table 32. XIRQ Trigger Configuration

| <code>XIRQ_TRIGGER_TYPE(i)</code> | <code>XIRQ_TRIGGER_POLARITY(i)</code> | Resulting Trigger of <code>xirq_i(i)</code> |
|-----------------------------------|---------------------------------------|---|
| 0                                 | 0                                     | low-level                                   |
| 0                                 | 1                                     | high-level                                  |
| 1                                 | 0                                     | falling-edge                                |
| 1                                 | 1                                     | rising-edge                                 |

The interrupt controller features three interface registers: external interrupt channel enable (`EIE`), external interrupt channel pending (`EIP`) and external interrupt source (`ESC`). From a functional point of view, the functionality of these registers follow the one of the RISC-V `mie`, `mip` and `mcause` CSRs.

If the configured trigger of an interrupt channel fires (e.g. a rising edge) the according interrupt channel becomes *pending*, which is indicated by the according channel bit being set in the `EIP` register. This pending interrupt can be cleared at any time by writing zero to the according `EIP` bit.

A pending interrupt can only trigger a CPU interrupt if the according is enabled via the `EIE` register.

Once triggered, disabled channels that were triggered remain pending until explicitly cleared. The channels are prioritized in a static order, i.e. channel 0 (`xirq_i(0)`) has the highest priority and channel 31 (`xirq_i(31)`) has the lowest priority. If any pending interrupt channel is actually enabled, an interrupt request is sent to the CPU.

The CPU can determine the most prioritized external interrupt request either by checking the bits in the `IPR` register or by reading the interrupt source register `ESC`. This register provides a 5-bit wide ID (0..31) identifying the currently firing external interrupt. Writing *any* value to this register will acknowledge the *current XIRQ* interrupt (so the XIRQ controller can issue a new CPU interrupt).

In order to acknowledge an XIRQ interrupt, the interrupt handler has to... \* clear the pending XIRQ channel by clearing the according `EIP` bit \* writing *any* value to `ESC` to acknowledge the XIRQ interrupt

## Register Map

Table 33. XIRQ register map (`struct NEORV32_XIRQ`)

| Address                 | Name<br>[C]      | Bit(s)            | R/W | Description   |
|-------------------------|------------------|-------------------|-----|---|
| <code>0xfffff300</code> | <code>EIE</code> | <code>31:0</code> | r/w | External interrupt enable register (one bit per channel, LSB-aligned)   |
| <code>0xfffff304</code> | <code>EIP</code> | <code>31:0</code> | r/w | External interrupt pending register (one bit per channel, LSB-aligned); writing 0 to a bit clears the according pending interrupt |
| <code>0xfffff308</code> | <code>ESC</code> | <code>4:0</code>  | r/w | Interrupt source ID (0..31) of firing IRQ (prioritized!); writing <i>any</i> value will acknowledge the current XIRQ interrupt    |
| <code>0xfffff30c</code> | -                | <code>31:0</code> | r/- | <i>reserved</i> , read as zero  |

## 2.7.24. General Purpose Timer (GPTMR)

|                          |                           |   |
|--------------------------|---------------------------|---|
| Hardware source file(s): | neorv32_gptmr.vhd         |   |
| Software driver file(s): | neorv32_gptmr.c           |   |
|                          | neorv32_gptmr.h           |   |
| Top entity port:         | <code>gptmr_trig_i</code> | timer capture input   |
| Configuration generics:  | <code>IO_GPTMR_EN</code>  | implement general purpose timer when <code>true</code>      |
| CPU interrupts:          | fast IRQ channel 12       | timer interrupt (see <a href="#">Processor Interrupts</a> ) |

### Overview

The general purpose timer module implements a simple yet universal 32-bit timer. It is implemented if the processor's `IO_GPTMR_EN` top generic is set `true`. The timer provides a pre-scaled counter register that can trigger an interrupt when reaching a programmable threshold value. Additionally, a timer-capture feature is implemented that copies the current counter value to a dedicated register if a programmable edge occurs at the `gptmr_trig_i` input signal.

Four interface registers are available: a control register (`CTRL`), a 32-bit counter register (`COUNT`), a 32-bit threshold register (`THRES`) and a 32-bit read-only capture register (`CAPTURE`). The timer is globally enabled by setting the `GPTMR_CTRL_EN` bit in the device's control register `CTRL`. When the timer is enable the `COUNT` register will start incrementing at a programmable rate, which scales the main processor clock. The pre-scaler value is configured via the three `GPTMR_CTRL_PRSCx` control register bits:

Table 34. GPTMR prescaler configuration

| <code>GPTMR_CTRL_PRSCx</code>          | <code>0b000</code> | <code>0b001</code> | <code>0b010</code> | <code>0b011</code> | <code>0b100</code> | <code>0b101</code> | <code>0b110</code> | <code>0b111</code> |
|--|--------------------|--------------------|--------------------|--------------------|--------------------|--------------------|--------------------|--------------------|
| Resulting <code>clock_prescaler</code> | 2                  | 4                  | 8                  | 64                 | 128                | 1024               | 2048               | 4096               |



Disabling the timer will not clear the `COUNT` register. However, it can be manually reset at any time by writing zero to it.

### Interval Timer

Whenever the counter register `COUNT` reaches the programmable threshold value `THRES` the counter register is reset to zero and the *timer-match* flag `GPTMR_CTRL_TRIGM` gets set. This flag has to be cleared manually by writing zero to it. Optionally, an interrupt can be triggered if the `GPTMR_CTRL_IRQM` bit is set.

### Timer Capture

In addition to the the internal timer, the GPTMR provides a timer-capture feature. Whenever an edge is detected at the `gptmr_trig_i` input signal the current `COUNT` value is copied to the read-only `CAPTURE` register and the *capture-trigger* flag `GPTMR_CTRL_TRIGC` gets set. This flag has to be cleared

manually by writing zero to it. Optionally, an interrupt can be triggered if the `GPTMR_CTRL_IRQC` bit is set.

The triggering edge can be a rising-edge (if `GPTMR_CTRL_RISE` is set), a falling-edge (if `GPTMR_CTRL_FALL` is set) or even both. By default, the `gptmr_trig_i` is sampled two times at the processor clock for checking for edges. This simple edge detection is sufficient for trigger signals that are generated by (on-chip) digital logic.

For sampling chip-external signals an optional filtering mode is available that can be enabled by the `GPTMR_CTRL_FILTER` bit. If this bit is set, the `gptmr_trig_i` is sampled at a reduced clock speed (1/4 of the processor clock) and the signal has to be stable for at least 4 sample clock in order to be considered high or low. This stabilized signal is then fed to the edge detection logic.

## Interrupt

The GPTRM provides a single interrupt line that can be triggered by a timer-match event and/or by a timer-compare event. Once triggered, the interrupt will stay active until explicitly cleared by writing zero to the according interrupt flag (`GPTMR_CTRL_TRIGM` or `GPTMR_CTRL_TRIGC`).

## Register Map

*Table 35. GPTMR register map (struct NEORV32\_GPTMR)*

| Address                 | Name [C]             | Bit(s), Name [C]                                     | R/W | Function  |
|-------------------------|----------------------|--|-----|---|
| <code>0xfffff100</code> | <code>CTRL</code>    | <code>0 GPTMR_CTRL_EN</code>                         | r/w | Timer enable flag                               |
|                         |                      | <code>3:1 GPTMR_CTRL_PRSC2 : GPTMR_CTRL_PRSC0</code> | r/w | 3-bit clock prescaler select                    |
|                         |                      | <code>4 GPTMR_CTRL_IRQM</code>                       | r/w | Enable interrupt on timer-match                 |
|                         |                      | <code>5 GPTMR_CTRL_IRQC</code>                       | r/w | Enable interrupt on capture-trigger             |
|                         |                      | <code>6 GPTMR_CTRL_RISE</code>                       | r/w | Capture on rising edge                          |
|                         |                      | <code>7 GPTMR_CTRL_FALL</code>                       | r/w | Capture on falling edge                         |
|                         |                      | <code>8 GPTMR_CTRL_FILTER</code>                     | r/w | Filter capture input                            |
|                         |                      | <code>29:9 -</code>                                  | r/- | <i>reserved</i> , read as zero                  |
|                         |                      | <code>30 GPTMR_CTRL_TRIGM</code>                     | r/c | Timer-match has fired, cleared by writing 0     |
|                         |                      | <code>31 GPTMR_CTRL_TRIGC</code>                     | r/c | Capture-trigger has fired, cleared by writing 0 |
| <code>0xfffff104</code> | <code>THRES</code>   | <code>31:0</code>                                    | r/w | Threshold value register                        |
| <code>0xfffff108</code> | <code>COUNT</code>   | <code>31:0</code>                                    | r/w | Counter register                                |
| <code>0xfffff10C</code> | <code>CAPTURE</code> | <code>31:0</code>                                    | r/- | Capture register                                |

## 2.7.25. Execute In Place Module (XIP)

|                          |  |  |
|--------------------------|--|--|
| Hardware source file(s): | neorv32_xip.vhd<br>neorv32_cache.vhd   | XIP module<br>Generic cache module   |
| Software driver file(s): | neorv32_xip.c<br>neorv32_xip.h   |  |
| Top entity port:         | <code>xip_csn_o</code><br><code>xip_clk_o</code><br><code>xip_dat_i</code><br><code>xip_dat_o</code>                           | 1-bit chip select, low-active<br>1-bit serial clock output<br>1-bit serial data input<br>1-bit serial data output  |
| Configuration generics:  | <code>XIP_EN</code><br><code>XIP_CACHE_EN</code><br><code>XIP_CACHE_NUM_BLOCKS</code><br><br><code>XIP_CACHE_BLOCK_SIZE</code> | implement XIP module when <code>true</code><br>implement XIP cache when <code>true</code><br>number of blocks in XIP cache; has to be a power of two<br>number of bytes per XIP cache block; has to be a power of two, min 4 |
| CPU interrupts:          | none   |  |

### Overview

The execute in-place (XIP) module allows to execute code (and read constant data) directly from an external SPI flash memory. The standard serial peripheral interface (SPI) is used as transfer protocol. All bus requests issued by the CPU are converted transparently into SPI flash access commands. Hence, the external XIP flash behaves like a simple on-chip ROM.

From the CPU side, the module provides two independent interfaces: one for transparently accessing the XIP flash and another one for accessing the module's control and status registers. The first interface provides the *transparent* gateway to the SPI flash, so the CPU can directly fetch and execute instructions and/or read constant data. Note that this interface is read-only. Any write access will raise a bus error exception. The second interface is mapped to the processor's IO space and allows accesses to the XIP module's configuration registers as well as conducting individual SPI transfers.

The XIP module provides an optional configurable cache to accelerate SPI flash accesses.

#### XIP Address Mapping



When XIP mode is enabled the flash is mapped to fixed address space region starting at address `0xE0000000` (see section [Address Space](#)) supporting a maximum flash size of 256MB.



#### XIP Example Program

An example program is provided in [sw/example/demo\\_xip](#) that illustrate how to program and configure an external SPI flash to run a program from it.

## SPI Configuration

The XIP module accesses external flash using the standard SPI protocol. The module always sends data MSB-first and provides all of the standard four clock modes (0..3), which are configured via the [XIP\\_CTRL\\_CPOL](#) (clock polarity) and [XIP\\_CTRL\\_CPHA](#) (clock phase) control register bits, respectively. The flash's "read command", which initiates a read access, is defined by the [XIP\\_CTRL\\_RD\\_CMD](#) control register bits. For most SPI flash memories this is [0x03](#) for *normal* SPI mode.

The SPI clock ([xip\\_clk\\_o](#)) frequency is programmed by the 3-bit [XIP\\_CTRL\\_PRSCx](#) clock prescaler for a coarse clock selection and a 4-bit clock divider [XPI\\_CTRL\\_CDIVx](#) for a fine clock selection. The following clock prescalers ([XIP\\_CTRL\\_PRSCx](#)) are available:

Table 36. XIP clock prescaler configuration

| <a href="#">XIP_CTRL_PRSCx</a>            | <a href="#">0b000</a> | <a href="#">0b001</a> | <a href="#">0b010</a> | <a href="#">0b011</a> | <a href="#">0b100</a> | <a href="#">0b101</a> | <a href="#">0b110</a> | <a href="#">0b111</a> |
|---|-----------------------|-----------------------|-----------------------|-----------------------|-----------------------|-----------------------|-----------------------|-----------------------|
| Resulting <a href="#">clock_prescaler</a> | 2                     | 4                     | 8                     | 64                    | 128                   | 1024                  | 2048                  | 4096                  |

Based on the programmed clock configuration, the actual SPI clock frequency  $f_{SPI}$  is derived from the processor's main clock  $f_{main}$  according to the following equation:

$$f_{SPI} = f_{main}[\text{Hz}] / (2 * \text{clock_prescaler} * (1 + XPI_CTRL_CDIVx))$$

Hence, the maximum SPI clock is  $f_{main} / 4$  and the lowest SPI clock is  $f_{main} / 131072$ . The SPI clock is always symmetric having a duty cycle of 50%.

## High-Speed Mode

The XIP module provides a high-speed mode to further boost the maximum SPI clock frequency. When enabled via the control register's [XIP\\_CTRL\\_HIGHSPEED](#) bit the clock prescaler configuration ([XIP\\_CTRL\\_PRSCx](#) bits) is overridden setting it to a minimal factor of 1. However, the clock speed can still be fine-tuned using the [XPI\\_CTRL\\_CDIVx](#) bits.

$$f_{SPI} = f_{main}[\text{Hz}] / (2 * 1 * (1 + XPI_CTRL_CDIVx))$$

Hence, the maximum SPI clock when in high-speed mode is  $f_{main} / 2$ .

## Direct SPI Access

The XIP module allows to initiate *direct* SPI transactions. This feature can be used to configure the attached SPI flash or to perform direct read and write accesses to the flash memory. Two data registers [DATA\\_LO](#) and [DATA\\_HI](#) are provided to send up to 64-bit of SPI data. The [DATA\\_HI](#) register is write-only, so a total of just 32-bits of receive data is provided. Note that the module handles the chip-select line ([xip\\_csn\\_o](#)) by itself so it is not possible to construct larger consecutive transfers.

The actual data transmission size in bytes is defined by the control register's [XIP\\_CTRL\\_SPI\\_NBYTES](#) bits. Any configuration from 1 byte to 8 bytes is valid. Other value will result in unpredictable

behavior.

Since data is always transferred MSB-first, the data in `DATA_HI:DATA_LO` also has to be MSB-aligned. Receive data is available in `DATA_LO` only since `DATA_HI` is write-only. Writing to `DATA_HI` triggers the actual SPI transmission. The `XIP_CTRL_PHY_BUSY` control register flag indicates a transmission being in progress.

The chip-select line of the XIP module (`xip_csn_o`) will only become asserted (enabled, pulled low) if the `XIP_CTRL_SPI_CSEN` control register bit is set. If this bit is cleared, `xip_csn_o` is always disabled (pulled high).



Direct SPI mode is only possible when the module is enabled (setting `XIP_CTRL_EN`) but **before** the actual XIP mode is enabled via `XIP_CTRL_XIP_EN`.



When the XIP mode is not enabled, the XIP module can also be used as additional general purpose SPI controller with a transfer size of up to 64 bits per transmission.

## Using the XIP Mode

The XIP module is globally enabled by setting the `XIP_CTRL_EN` bit in the device's `CTRL` control register. Clearing this bit will reset the whole module and will also terminate any pending SPI transfer.

Since there is a wide variety of SPI flash components with different sizes, the XIP module allows to specify the address width of the flash: the number of address bytes used for addressing flash memory content has to be configured using the control register's `XIP_CTRL_XIP_ABYTES` bits. These two bits contain the number of SPI address bytes (**minus one**). For example for a SPI flash with 24-bit addresses these bits have to be set to `0b10`.

The transparent XIP accesses are transformed into SPI transmissions with the following format (starting with the MSB):

- 8-bit command: configured by the `XIP_CTRL_RD_CMD` control register bits ("SPI read command")
- 8 to 32 bits address: defined by the `XIP_CTRL_XIP_ABYTES` control register bits ("number of address bytes")
- 32-bit data: sending zeros and receiving the according flash word (32-bit)

Hence, the maximum XIP transmission size is 72-bit, which has to be configured via the `XIP_CTRL_SPI_NBYTES` control register bits. Note that the 72-bit transmission size is only available in XIP mode. The transmission size of the direct SPI accesses is limited to 64-bit.



When using four SPI flash address bytes, the most significant 4 bits of the address are always hardwired to zero allowing a maximum **accessible** flash size of 256MB.



The XIP module always fetches a full naturally aligned 32-bit word from the SPI

flash. Any sub-word data masking or alignment will be performed by the CPU core logic.



The XIP mode requires the 4-byte data words in the flash to be ordered in **little-endian** byte order.

After the SPI properties (including the amount of address bytes **and** the total amount of SPI transfer bytes) and XIP address mapping are configured, the actual XIP mode can be enabled by setting the control register's `XIP_CTRL_XIP_EN` bit. This will enable the "transparent SPI access port" of the module and thus, the *transparent* conversion of access requests into proper SPI flash transmissions. Hence, any access to the processor's memory-mapped XIP region (`0xE0000000` to `0xFFFFFFFF`) will be converted into SPI flash accesses. Make sure `XIP_CTRL_SPI_CSEN` is also set so the module can actually select/enable the attached SPI flash. No more direct SPI accesses via `DATA_HI:DATA_LO` are possible when the XIP mode is enabled. However, the XIP mode can be disabled at any time.



If the XIP module is disabled (`XIP_CTRL_EN = 0`) any accesses to the memory-mapped XIP flash address region will raise a bus access exception. If the XIP module is enabled (`XIP_CTRL_EN = 1`) but XIP mode is not enabled yet (`XIP_CTRL_XIP_EN = '0'`) any access to the programmed XIP memory segment will also raise a bus access exception.



It is highly recommended to enable the [Processor-Internal Instruction Cache \(iCACHE\)](#) to cover some of the SPI access latency.

## XIP Cache

Since every single instruction fetch request from the CPU is translated into serial SPI transmissions the access latency is very high resulting in a low throughput. In order to improve performance, the XIP module provides an optional cache that allows to buffer recently-accessed data. The cache is implemented as a simple direct-mapped read-only cache with a configurable cache layout:

- `XIP_CACHE_EN`: when set to `true` the CIP cache is implemented
- `XIP_CACHE_NUM_BLOCKS` defines the number of cache blocks (or lines)
- `XIP_CACHE_BLOCK_SIZE` defines the size in bytes of each cache block

When the cache is implemented, the XIP module operates in **burst mode** utilizing the flash's *incremental read* capabilities. Thus, several bytes (= `XIP_CACHE_BLOCK_SIZE`) are read consecutively from the flash using a single read command.

The XIP cache is cleared when the XIP module is disabled (`XIP_CTRL_EN = 0`), when XIP mode is disabled (`XIP_CTRL_XIP_EN = 0`) or when the CPU issues a `fence(.i)` instruction.

## Register Map

*Table 37. XIP Register Map (struct NEORV32\_XIP)*

| Address      | Name [C] | Bit(s), Name [C]  | R/W | Function   |
|--------------|----------|---|-----|--|
| 0xffffffff40 | CTRL     | 0 XIP_CTRL_EN   | r/w | XIP module enable  |
|              |          | 3:1 XIP_CTRL_PRSC2 : XIP_CTRL_PRSC0                     | r/w | 3-bit SPI clock prescaler select   |
|              |          | 4 XIP_CTRL_CPOL   | r/w | SPI clock polarity   |
|              |          | 5 XIP_CTRL_CPHA   | r/w | SPI clock phase  |
|              |          | 9:6 XIP_CTRL_SPI_NBYTES_MSB : XIP_CTRL_SPI_NBYTES_LSB   | r/w | Number of bytes in SPI transaction (1..9)                                  |
|              |          | 10 XIP_CTRL_XIP_EN                                      | r/w | XIP mode enable  |
|              |          | 12:11 XIP_CTRL_XIP_ABYTES_MSB : XIP_CTRL_XIP_ABYTES_LSB | r/w | Number of address bytes for XIP flash (minus 1)                            |
|              |          | 20:13 XIP_CTRL_RD_CMD_MSB : XIP_CTRL_RD_CMD_LSB         | r/w | Flash read command   |
|              |          | 21 XIP_CTRL_SPI_CSEN                                    | r/w | Allow SPI chip-select to be actually asserted when set                     |
|              |          | 22 XIP_CTRL_HIGHSPEED                                   | r/w | enable SPI high-speed mode (ignoring XIP_CTRL_PRSCx)                       |
|              |          | 26:23 XIP_CTRL_CDIV3 : XIP_CTRL_CDIV0                   | r/- | 4-bit clock divider for fine-tuning  |
|              |          | 29:27 -   | r/- | <i>reserved</i> , read as zero   |
|              |          | 30 XIP_CTRL_PHY_BUSY                                    | r/- | SPI PHY busy when set  |
|              |          | 31 XIP_CTRL_XIP_BUSY                                    | r/- | XIP access in progress when set  |
| 0xffffffff44 | reserved | 31:0  | r/- | <i>reserved</i> , read as zero   |
| 0xffffffff48 | DATA_LO  | 31:0  | r/w | Direct SPI access - data register low                                      |
| 0xffffffff4C | DATA_HI  | 31:0  | -/w | Direct SPI access - data register high; write access triggers SPI transfer |

## 2.7.26. System Configuration Information Memory (SYSINFO)

|                          |                     |  |
|--------------------------|---------------------|--|
| Hardware source file(s): | neorv32_sysinfo.vhd |  |
| Software driver file(s): | neorv32_sysinfo.h   |  |
| Top entity port:         | none                |  |
| Configuration generics:  | *                   | most of the top's configuration generics |
| CPU interrupts:          | none                |  |

### Overview

The SYSINFO allows the application software to determine the setting of most of the [Processor Top Entity - Generics](#) that are related to processor/SoC configuration. All registers of this unit are read-only. This device is always implemented - regardless of the actual hardware configuration. The bootloader as well as the NEORV32 software runtime environment require information from this device (like memory layout and default clock frequency) for correct operation.

### Register Map

Table 38. SYSINFO register map (`struct NEORV32_SYSINFO`)

| Address     | Name [C] | Function   |
|-------------|----------|--|
| 0xffffffe00 | CLK      | clock frequency in Hz (via top's <code>CLOCK_FREQUENCY</code> generic)               |
| 0xffffffe04 | MEM[4]   | internal memory configuration (see <a href="#">SYSINFO - Memory Configuration</a> )  |
| 0xffffffe08 | SOC      | specific SoC configuration (see <a href="#">SYSINFO - SoC Configuration</a> )        |
| 0xffffffe0c | CACHE    | cache configuration information (see <a href="#">SYSINFO - Cache Configuration</a> ) |

### SYSINFO - Memory Configuration



Bit fields in this register are set to all-zero if the according cache is not implemented.

Table 39. SYSINFO MEM Bytes

| Byte | Name [C]         | Function   |
|------|------------------|--|
| 0    | SYSINFO_MEM_IMEM | $\log_2$ (internal IMEM size in bytes), via top's <code>MEM_INT_IMEM_SIZE</code> generic |
| 1    | SYSINFO_MEM_DMEM | $\log_2$ (internal DMEM size in bytes), via top's <code>MEM_INT_DMEM_SIZE</code> generic |
| 2    | -                | <i>reserved</i> , read as zero   |

| Byte | Name [C]         | Function   |
|------|------------------|--|
| 3    | SYSINFO_MEM_RVSG | $\log_2$ (reservation set size granularity in bytes), via top's <code>AMO_RVS_GRANULARITY</code> generic |

## SYSINFO - SoC Configuration

Table 40. SYSINFO SOC Bits

| Bit   | Name [C]                 | Function   |
|-------|--------------------------|--|
| 0     | SYSINFO_SOC_BOOTLOADER   | set if processor-internal bootloader is implemented (via top's <code>INT_BOOTLOADER_EN</code> generic) |
| 1     | SYSINFO_SOC_XBUS         | set if external Wishbone bus interface is implemented (via top's <code>XBUS_EN</code> generic)         |
| 2     | SYSINFO_SOC_MEM_INT_IMEM | set if processor-internal DMEM implemented (via top's <code>MEM_INT_DMEM_EN</code> generic)            |
| 3     | SYSINFO_SOC_MEM_INT_DMEM | set if processor-internal IMEM is implemented (via top's <code>MEM_INT_IMEM_EN</code> generic)         |
| 4     | SYSINFO_SOC_OCD          | set if on-chip debugger is implemented (via top's <code>ON_CHIP_DEBUGGER_EN</code> generic)            |
| 5     | SYSINFO_SOC_ICACHE       | set if processor-internal instruction cache is implemented (via top's <code>ICACHE_EN</code> generic)  |
| 6     | SYSINFO_SOC_DCACHE       | set if processor-internal data cache is implemented (via top's <code>DCACHE_EN</code> generic)         |
| 7     | SYSINFO_SOC_CLOCK_GATING | set if CPU clock gating is implemented (via top's <code>CLOCK_GATING_EN</code> generic)                |
| 8     | SYSINFO_SOC_XBUS_CACHE   | set if external bus interface cache is implemented (via top's <code>XBUS_CACHE_EN</code> generic)      |
| 9     | SYSINFO_SOC_XIP          | set if XIP module is implemented (via top's <code>XIP_EN</code> generic)                               |
| 10    | SYSINFO_SOC_XIP_CACHE    | set if XIP cache is implemented (via top's <code>XIP_CACHE_EN</code> generic)                          |
| 13:11 | -                        | <i>reserved</i> , read as zero   |
| 14    | SYSINFO_SOC_IO_DMA       | set if direct memory access controller is implemented (via top's <code>IO_DMA_EN</code> generic)       |
| 15    | SYSINFO_SOC_IO_GPIO      | set if GPIO is implemented (via top's <code>IO_GPIO_EN</code> generic)                                 |
| 16    | SYSINFO_SOC_IO_MTIME     | set if MTIME is implemented (via top's <code>IO_MTIME_EN</code> generic)                               |
| 17    | SYSINFO_SOC_IO_UART0     | set if primary UART0 is implemented (via top's <code>IO_UART0_EN</code> generic)                       |

| Bit | Name [C]               | Function  |
|-----|------------------------|---|
| 18  | SYSINFO_SOC_IO_SPI     | set if SPI is implemented (via top's <code>IO_SPI_EN</code> generic)                          |
| 19  | SYSINFO_SOC_IO_TWI     | set if TWI is implemented (via top's <code>IO_TWI_EN</code> generic)                          |
| 20  | SYSINFO_SOC_IO_PWM     | set if PWM is implemented (via top's <code>IO_PWM_NUM_CH</code> generic)                      |
| 21  | SYSINFO_SOC_IO_WDT     | set if WDT is implemented (via top's <code>IO_WDT_EN</code> generic)                          |
| 22  | SYSINFO_SOC_IO_CFS     | set if custom functions subsystem is implemented (via top's <code>IO_CFS_EN</code> generic)   |
| 23  | SYSINFO_SOC_IO_TRNG    | set if TRNG is implemented (via top's <code>IO_TRNG_EN</code> generic)                        |
| 24  | SYSINFO_SOC_IO_SDI     | set if SDI is implemented (via top's <code>IO_SDI_EN</code> generic)                          |
| 25  | SYSINFO_SOC_IO_UART1   | set if secondary UART1 is implemented (via top's <code>IO_UART1_EN</code> generic)            |
| 26  | SYSINFO_SOC_IO_NEOLED  | set if NEOLED is implemented (via top's <code>IO_NEOLED_EN</code> generic)                    |
| 27  | SYSINFO_SOC_IO_XIRQ    | set if XIRQ is implemented (via top's <code>XIRQ_NUM_CH</code> generic)                       |
| 28  | SYSINFO_SOC_IO_GPTMR   | set if GPTMR is implemented (via top's <code>IO_GPTMR_EN</code> generic)                      |
| 29  | SYSINFO_SOC_IO_SLINK   | set if stream link interface is implemented (via top's <code>IO_SLINK_EN</code> generic)      |
| 30  | SYSINFO_SOC_IO_ONewire | set if ONEWIRE interface is implemented (via top's <code>IO_ONewire_EN</code> generic)        |
| 31  | SYSINFO_SOC_IO_CRC     | set if cyclic redundancy check unit is implemented (via top's <code>IO_CRC_EN</code> generic) |

## SYSINFO - Cache Configuration



Bit fields in this register are set to all-zero if the according cache is not implemented.

Table 41. SYSINFO CACHE Bits

| Bit | Name [C]   | Function  |
|-----|--|---|
| 3:0 | SYSINFO_CACHE_INST_BLOCK_SIZE_3 :<br>SYSINFO_CACHE_INST_BLOCK_SIZE_0 | $\log_2$ (i-cache block size in bytes), via top's <code>ICACHE_BLOCK_SIZE</code> generic    |
| 7:4 | SYSINFO_CACHE_INST_NUM_BLOCKS_3 :<br>SYSINFO_CACHE_INST_NUM_BLOCKS_0 | $\log_2$ (i-cache number of cache blocks), via top's <code>ICACHE_NUM_BLOCKS</code> generic |

| Bit       | Name [C]   | Function   |
|-----------|--|--|
| 11:<br>8  | SYSINFO_CACHE_DATA_BLOCK_SIZE_3 :<br>SYSINFO_CACHE_DATA_BLOCK_SIZE_0 | $\log_2$ (d-cache block size in bytes), via top's <b>DCACHE_BLOCK_SIZE</b> generic           |
| 15:<br>12 | SYSINFO_CACHE_DATA_NUM_BLOCKS_3 :<br>SYSINFO_CACHE_DATA_NUM_BLOCKS_0 | $\log_2$ (d-cache number of cache blocks), via top's <b>DCACHE_NUM_BLOCKS</b> generic        |
| 19:<br>16 | SYSINFO_CACHE_XIP_BLOCK_SIZE_3 :<br>SYSINFO_CACHE_XIP_BLOCK_SIZE_0   | $\log_2$ (xip-cache block size in bytes), via top's <b>XIP_CACHE_BLOCK_SIZE</b> generic      |
| 23:<br>20 | SYSINFO_CACHE_XIP_NUM_BLOCKS_3 :<br>SYSINFO_CACHE_XIP_NUM_BLOCKS_0   | $\log_2$ (xip-cache number of cache blocks), via top's <b>XIP_CACHE_NUM_BLOCKS</b> generic   |
| 27:<br>24 | SYSINFO_CACHE_XBUS_BLOCK_SIZE_3 :<br>SYSINFO_CACHE_XBUS_BLOCK_SIZE_0 | $\log_2$ (xbus-cache block size in bytes), via top's <b>XBUS_CACHE_BLOCK_SIZE</b> generic    |
| 31:<br>28 | SYSINFO_CACHE_XBUS_NUM_BLOCKS_3 :<br>SYSINFO_CACHE_XBUS_NUM_BLOCKS_0 | $\log_2$ (xbus-cache number of cache blocks), via top's <b>XBUS_CACHE_NUM_BLOCKS</b> generic |

# Chapter 3. NEORV32 Central Processing Unit (CPU)

The NEORV32 CPU is an area-optimized RISC-V core implementing the `rv32i_zicsr_zifencei` base (privileged) ISA and supporting several additional/optional ISA extensions. The CPU's micro architecture is based on a von-Neumann machine build upon a mixture of multi-cycle and pipelined execution schemes.



This chapter assumes that the reader is familiar with the official RISC-V *User* and *Privileged Architecture* specifications.

## Section Structure

- [RISC-V Compatibility](#)
- [CPU Top Entity - Signals](#) and [CPU Top Entity - Generics](#)
- [Architecture and Full Virtualization](#)
- [Instruction Sets and Extensions](#) and [Custom Functions Unit \(CFU\)](#)
- [Control and Status Registers \(CSRs\)](#)
- [Traps, Exceptions and Interrupts](#)
- [Bus Interface](#)

## 3.1. RISC-V Compatibility

The NEORV32 CPU passes the tests of the **official RISCOF RISC-V Architecture Test Framework**. This framework is used to check RISC-V implementations for compatibility to the official RISC-V user/privileged ISA specifications. The NEORV32 port of this test framework is available in a separate repository at GitHub: <https://github.com/stnolting/neorv32-riscof>



### *Unsupported ISA Extensions*

Executing instructions or accessing CSRs from yet unsupported ISA extensions will raise an illegal instruction exception (see section [Full Virtualization](#)).

## Incompatibility Issues and Limitations



### *time[h] CSRs (Wall Clock Time)*

The NEORV32 does not implement the `time[h]` registers. Any access to these registers will trap. It is recommended that the trap handler software provides a means of accessing the platform-defined [Machine System Timer \(MTIME\)](#).



### *No Hardware Support of Misaligned Memory Accesses*

The CPU does not support resolving unaligned memory access by the hardware

(this is not a RISC-V-incompatibility issue but an important thing to know!). Any kind of unaligned memory access will raise an exception to allow a *software-based* emulation provided by the application. However, unaligned memory access can be **emulated** using the NEORV32 runtime environment. See section [Application Context Handling](#) for more information.

#### *No Atomic Read-Modify-Write Operations*

The NEORV32 [A ISA Extension](#) only supports the load-reservate (LR) and store-conditional (SR) instructions. The remaining read-modify-write operations are not supported. However, these missing instructions can be emulated. The NEORV32 [Core Libraries](#) provide an emulation wrapper for the missing AMO/read-modify-write instructions that is based on LR/SC pairs. A demo/program can be found in [sw/example/atomic\\_test](#).



## 3.2. CPU Top Entity - Signals

The following table shows all interface signals of the CPU top entity `rtl/core/neorv32_cpu.vhd`. The type of all signals is `std_ulogic` or `std_ulogic_vector`, respectively. The "Dir." column shows the signal direction as seen from the CPU.

*Table 42. NEORV32 CPU Signal List*

| Signal   | Width/Type             | Dir | Description   |
|--|------------------------|-----|---|
| <b>Global Signals</b>                                |                        |     |   |
| <code>clk_i</code>                                   | 1                      | in  | Global clock line, all registers triggering on rising edge, this clock can be switched off during <b>Sleep Mode</b> |
| <code>clk_aux_i</code>                               | 1                      | in  | Always-on clock, used to keep the sleep control active when <code>clk_i</code> is switched off                      |
| <code>rstn_i</code>                                  | 1                      | in  | Global reset, low-active  |
| <code>sleep_o</code>                                 | 1                      | out | CPU is in <b>Sleep Mode</b> when set  |
| <code>debug_o</code>                                 | 1                      | out | CPU is in <b>debug mode</b> when set  |
| <b>Interrupts (Traps, Exceptions and Interrupts)</b> |                        |     |   |
| <code>msi_i</code>                                   | 1                      | in  | RISC-V machine software interrupt   |
| <code>mei_i</code>                                   | 1                      | in  | RISC-V machine external interrupt   |
| <code>mti_i</code>                                   | 1                      | in  | RISC-V machine timer interrupt  |
| <code>firq_i</code>                                  | 16                     | in  | Custom fast interrupt request signals   |
| <code>dbi_i</code>                                   | 1                      | in  | Request CPU to halt and enter debug mode (RISC-V <b>On-Chip Debugger (OCD)</b> )                                    |
| <b>Instruction Bus Interface</b>                     |                        |     |   |
| <code>ibus_req_o</code>                              | <code>bus_req_t</code> | out | Instruction fetch bus request   |
| <code>ibus_rsp_i</code>                              | <code>bus_rsp_t</code> | in  | Instruction fetch bus response  |
| <b>Data Bus Interface</b>                            |                        |     |   |
| <code>dbus_req_o</code>                              | <code>bus_req_t</code> | out | Data access (load/store) bus request  |
| <code>dbus_rsp_i</code>                              | <code>bus_rsp_t</code> | in  | Data access (load/store) bus response   |

### Bus Interface Protocol



See section **Bus Interface** for the instruction fetch and data access interface protocol and the according interface types (`bus_req_t` and `bus_rsp_t`).

### 3.3. CPU Top Entity - Generics

Most of the CPU configuration generics are a subset of the actual Processor configuration generics (see section [Processor Top Entity - Generics](#)). and are not listed here. However, the CPU provides some *specific* generics that are used to configure the CPU for the NEORV32 processor setup. These generics are assigned by the processor setup only and are not available for user defined configuration. The specific generics are listed below.



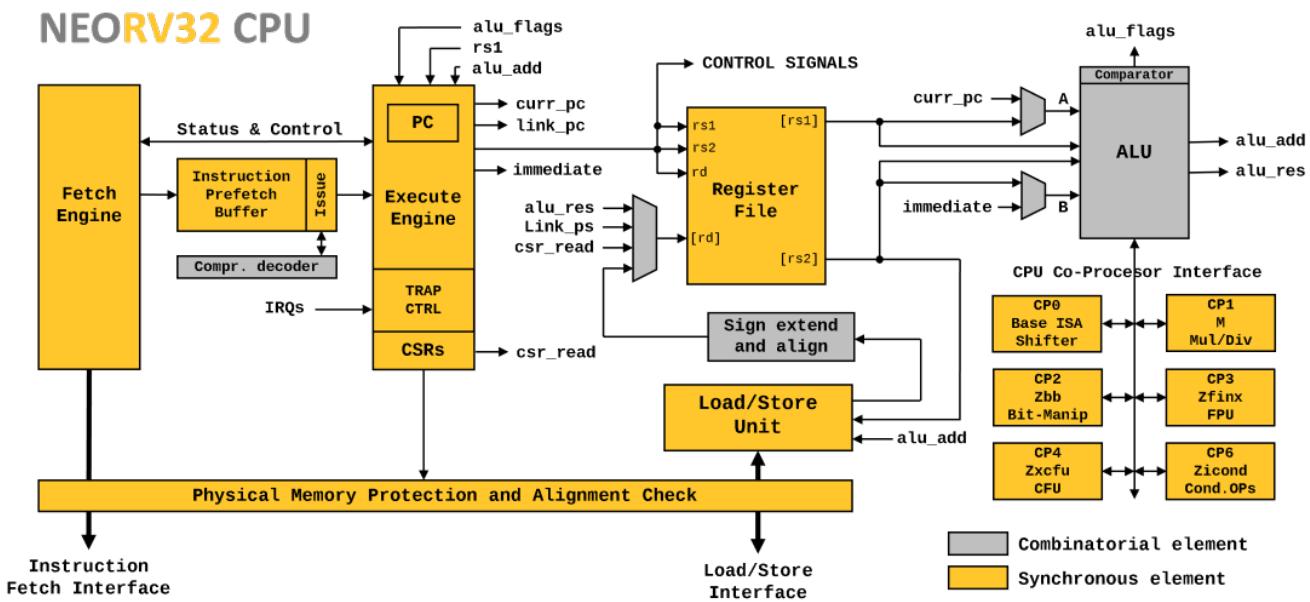
*Table Abbreviations*

The generic type "suv(x:y)" defines a `std_ulogic_vector(x downto y)`.

*Table 43. NEORV32 CPU-Exclusive Generic List*

| Name                                    | Type      | Description  |
|---|-----------|--|
| <code>VENDOR_ID</code>                  | suv(31:0) | Value for the <a href="#">mvendorid</a> CSR.   |
| <code>CPU_BOOT_ADDR</code>              | suv(31:0) | CPU reset address. See section <a href="#">Address Space</a> .   |
| <code>CPU_DEBUG_PARK_ADDR</code>        | suv(31:0) | "Park loop" entry address for the <a href="#">On-Chip Debugger (OCD)</a> , has to be 4-byte aligned.             |
| <code>CPU_DEBUG_EXC_ADDR</code>         | suv(31:0) | "Exception" entry address for the <a href="#">On-Chip Debugger (OCD)</a> , has to be 4-byte aligned.             |
| <code>CPU_EXTENSION_RISCV_Sdext</code>  | boolean   | Implement RISC-V-compatible "debug" CPU operation mode required for the <a href="#">On-Chip Debugger (OCD)</a> . |
| <code>CPU_EXTENSION_RISCV_Sdtrig</code> | boolean   | Implement RISC-V-compatible trigger module. See section <a href="#">On-Chip Debugger (OCD)</a> .                 |
| <code>CPU_EXTENSION_RISCV_Smpmp</code>  | boolean   | Implement RISC-V-compatible physical memory protection (PMP). See section <a href="#">Smpmp ISA Extension</a> .  |

## 3.4. Architecture



The CPU implements a pipelined multi-cycle architecture: each instruction is executed as a series of consecutive micro-operations. In order to increase performance, the CPU's front-end (instruction fetch) and back-end (instruction execution) are de-coupled via a FIFO (the instruction prefetch buffer). Thus, the front-end can already fetch new instructions while the back-end is still processing the previously-fetched instructions.

Basically, the CPU's micro architecture is somewhere between a classical pipelined architecture, where each stage requires exactly one processing cycle (if not stalled) and a classical multi-cycle architecture, which executes every single instruction (*including* fetch) in a series of consecutive micro-operations. The combination of these two design paradigms allows an increased instruction execution in contrast to a pure multi-cycle approach (due to overlapping operation of fetch and execute) at a reduced hardware footprint (due to the multi-cycle concept).

As a Von-Neumann machine, the CPU provides independent interfaces for instruction fetch and data access. However, these two bus interfaces are merged into a single processor-internal bus via a prioritizing bus switch (data accesses have higher priority). Hence, *all* memory addresses including peripheral devices are mapped to a single unified 32-bit [Address Space](#).



The CPU does not perform any speculative/out-of-order operations at all. Hence, it is not vulnerable to security issues caused by speculative execution (like Spectre or Meltdown).

### 3.4.1. CPU Register File

The data register file contains the general purpose architecture registers  $x_0$  to  $x_{31}$ . For the [rv32e](#) ISA only the lower 16 registers are implemented. Register zero ( $x_0/zero$ ) always read as zero and any write access to it has no effect. Up to four individual synchronous read ports allow to fetch up to 4 register operands at once. The write and read accesses are mutually exclusive as they happen in

separate cycles. Hence, there is no need to consider things like "read-during-write" behavior.

The register file provides two different implementation options configured via the top's `REGFILE_HW_RST` generic.

- `REGFILE_HW_RST = false` (default): In this configuration the register file is implemented as plain memory array without a dictated hardware reset. This architecture allows to infer FPGA block RAM for the entire register file resulting in minimal logic utilization and optimal timing.
- `REGFILE_HW_RST = true`: This configuration is based on individual FFs that do provide a dedicated hardware reset. Hence, the register cannot be mapped to FPGA block RAM. This optional should only be selected if the application requires a reset of the register file (e.g. for security reasons) or if the design shall be synthesized for an **ASIC** implementation.

The state of this configuration generic can be checked by software via the `mxisa` CSR.



#### FPGA Implementation

Enabling the `REGFILE_HW_RST` option for FPGA implementation is not recommended as this will massively increase the amount of required logic resources.



#### Implementation of the `zero` Register within FPGA Block RAM

Register `zero` is also mapped to a *physical memory location* within the register file's block RAM. By this, there is no need to add a further multiplexer to "insert" zero if reading from register `zero` reducing logic requirements and shortening the critical path. However, this also requires that the physical storage bits of register `zero` are explicitly initialized (set to zero) by the hardware. This is done transparently by the CPU control requiring no additional processing overhead.

#### Block RAM Ports



The default register file configuration uses two access ports: a read-only port for reading register `rs2` (second source operand) and a read/write port for reading register `rs1` (first source operand) and for writing processing results to register `rd` (destination register). Hence, a simple dual-port RAM can be used to implement the entire register file. From a functional point of view, read and write accesses to the register file do never occur in the same clock cycle, so no bypass logic is required at all.

### 3.4.2. CPU Arithmetic Logic Unit

The arithmetic/logic unit (ALU) is used for actual data processing as well as generating memory and branch addresses. All "simple" **I ISA Extension** computational instructions (like `add` and `or`) are implemented as plain combinatorial logic requiring only a single cycle to complete. More sophisticated instructions like shift operations or multiplications are processed by so-called "ALU co-processors".

The co-processors are implemented as iterative units that require several cycles to complete

processing. Besides the base ISA's shift instructions, the co-processors are used to implement all further processing-based ISA extensions (e.g. [M ISA Extension](#) and [B ISA Extension](#)).

#### *Multi-Cycle Execution Monitor*



The CPU control will raise an illegal instruction exception if a multi-cycle functional unit (like the [Custom Functions Unit \(CFU\)](#)) does not complete processing in a bound amount of time (configured via the package's `monitor_mc_tmo_c` constant; default = 512 clock cycles).

#### *Tuning Options*



The ALU architecture can be tuned for an application-specific area-vs-performance trade-off. The `FAST_MUL_EN` and `FAST_SHIFT_EN` generics can be used to implement performance-optimized barrel shifters and DSP blocks, respectively. See sections [I ISA Extension](#), [B ISA Extension](#) and [M ISA Extension](#) for specific examples.

### 3.4.3. CPU Bus Unit

The bus unit takes care of handling data memory accesses via load and store instructions. It handles data adjustment when accessing sub-word data quantities (16-bit or 8-bit) and performs sign-extension for singed load operations. The bus unit also includes the optional [Smpmp ISA Extension](#) that performs permission checks for all data and instruction accesses.

A list of the bus interface signals and a detailed description of the protocol can be found in section [Bus Interface](#). All bus interface signals are driven/buffered by registers; so even a complex SoC interconnection bus network will not effect maximal operation frequency.

#### *Unaligned Accesses*



The CPU does not support a hardware-based handling of unaligned memory accesses! Any unaligned access will raise a bus load/store unaligned address exception. The exception handler can be used to *emulate* unaligned memory accesses in software. See the NEORV32 Runtime Environment's [Application Context Handling](#) section for more information.

### 3.4.4. CPU Control Unit

The CPU control unit is responsible for generating all the control signals for the different CPU modules. The control unit is split into a "front-end" and a "back-end".

#### Front-End

The front-end is responsible for fetching instructions in chunks of 32-bits. This can be a single aligned 32-bit instruction, two aligned 16-bit instructions or a mixture of those. The instructions including control and exception information are stored to a FIFO queue - the instruction prefetch buffer (IPB). This FIFO has a depth of two entries by default but can be customized via the

`ipb_depth_c` VHDL package constant.

The FIFO allows the front-end to do "speculative" instruction fetches, as it keeps fetching the next consecutive instruction all the time. This also allows to decouple front-end (instruction fetch) and back-end (instruction execution) so both modules can operate in parallel to increase performance. However, all potential side effects that are caused by this "speculative" instruction fetch are already handled by the CPU front-end ensuring a defined execution stage while preventing security side attacks.

## Back-End

Instruction data from the instruction prefetch buffer is decompressed (if the `C` ISA extension is enabled) and sent to the CPU back-end for actual execution. Execution is conducted by a state-machine that controls all of the CPU modules. The back-end also includes the [Control and Status Registers \(CSRs\)](#) as well as the trap controller.

### 3.4.5. Sleep Mode

The NEORV32 CPU provides a single sleep mode that can be entered to power-down the core reducing dynamic power consumption. Sleep mode is entered by executing the `wfi` ("wait for interrupt") instruction.



The `wfi` instruction will raise an illegal instruction exception when executed in user-mode if `TW` in `mstatus` is set. When executed in debug-mode or during single-stepping `wfi` will behave as simple `nop` without entering sleep mode.

After executing the `wfi` instruction the CPU's `sleep_o` signal ([CPU Top Entity - Signals](#)) will become set as soon as the CPU has fully halted ("CPU is sleeping"):

*There is no enabled interrupt being pending.*

CPU-external modules like memories, timers and peripheral interfaces are not affected by this. Furthermore, the CPU will continue to buffer/enqueue incoming interrupt. The CPU will leave sleep mode as soon as any *enabled (via `mie`) interrupt source becomes \_pending* or if a debug session is started.

## Power-Down Mode

Optionally, the sleep mode can also be used to shut down the CPU's main clock to further reduce power consumption by halting the core's clock tree. This clock gating mode is enabled by the `CLOCK_GATING_EN` generic ([Processor Top Entity - Generics](#)). See section [Processor Clocking](#) for more information.

### 3.4.6. Full Virtualization

Just like the RISC-V ISA, the NEORV32 aims to provide *maximum virtualization* capabilities on CPU and SoC level to allow a high standard of **execution safety**. The CPU supports **all** traps specified by the official RISC-V specifications. Thus, the CPU provides defined hardware fall-backs via traps for

any expected and unexpected situations (e.g. executing a malformed or not supported instruction or accessing a non-allocated memory address). For any kind of trap the core is always in a defined and fully synchronized state throughout the whole system (i.e. there are no out-of-order operations that might have to be reverted). This allows a defined and predictable execution behavior at any time improving overall execution safety.

## 3.5. Bus Interface

The NEORV32 CPU provides separated instruction fetch and data access interfaces making it a **Harvard Architecture**: the instruction fetch interface (`i_bus_*` signals) is used for fetching instructions and the data access interface (`d_bus_*` signals) is used to access data via load and store operations. Each of these interfaces can access an address space of up to  $2^{32}$  bytes (4GB).

The bus interface uses two custom interface types: `bus_req_t` is used to propagate the bus access **requests**. These signals are driven by the *accessing* device (i.e. the CPU core). `bus_rsp_t` is used to return the bus **response** and is driven by the *accessed* device or bus system (i.e. a processor-internal memory or IO device).

*Table 44. Bus Interface - Request Bus (`bus_req_t`)*

| Signal             | Width | Description  |
|--------------------|-------|--|
| <code>addr</code>  | 32    | Access address (byte addressing)   |
| <code>data</code>  | 32    | Write data   |
| <code>ben</code>   | 4     | Byte-enable for each byte in <code>data</code>   |
| <code>stb</code>   | 1     | Request trigger ("strobe", single-shot)  |
| <code>rw</code>    | 1     | Access direction (0 = read, 1 = write)   |
| <code>src</code>   | 1     | Access source (0 = instruction fetch, 1 = load/store)  |
| <code>priv</code>  | 1     | Set if privileged (M-mode) access  |
| <code>rvso</code>  | 1     | Set if current access is a reservation-set operation (atomic <code>lr</code> or <code>sc</code> instruction) |
| <code>fence</code> | 1     | Data/instruction fence operation; valid without <code>stb</code> being set                                   |

*Table 45. Bus Interface - Response Bus (`bus_rsp_t`)*

| Signal            | Width | Description                                  |
|-------------------|-------|--|
| <code>data</code> | 32    | Read data (single-shot)                      |
| <code>ack</code>  | 1     | Transfer acknowledge / success (single-shot) |
| <code>err</code>  | 1     | Transfer error / fail (single-shot)          |

### 3.5.1. Bus Interface Protocol

Transactions are triggered entirely by the request bus. A new bus request is initiated by setting the *strobe* signal `stb` high for exactly one cycle. All remaining signals of the bus are set together with `stb` and will remain unchanged until the transaction is completed.

The transaction is completed when the accessed device returns a response via the response interface: `ack` is high for exactly one cycle if the transaction was completed successfully. `err` is high for exactly one cycle if the transaction failed to complete. These two signals are mutually exclusive. In case of a read access the read data is returned together with the `ack` signal. Otherwise, the return

data signal is kept at all-zero allowing wired-or interconnection of all response buses.

The figure below shows three exemplary bus accesses:

1. A read access to address `A_addr` returning `rdata` after several cycles (slow response; `ACK` arrives after several cycles).
2. A write access to address `B_addr` writing `wdata` (fastest response; `ACK` arrives right in the next cycle).
3. A failing read access to address `C_addr` (slow response; `ERR` arrives after several cycles).

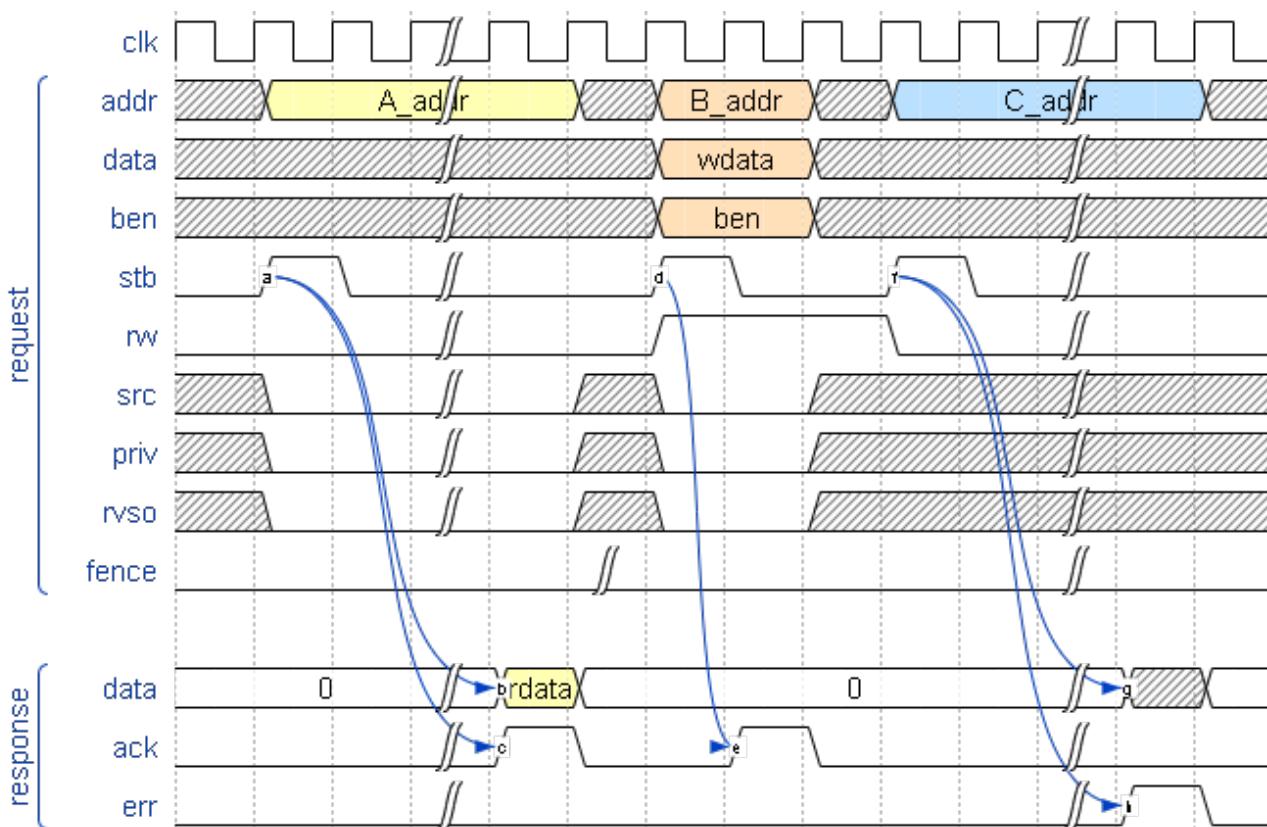


Figure 9. Three Exemplary Bus Transactions

#### Adding Register Stages



Arbitrary pipeline stages can be added to the request and response buses at any point to relax timing (at the cost of additional latency). However, *all* bus signals (request and response) need to be registered.

### 3.5.2. Atomic Accesses

The load-reservate (`lr.w`) and store-conditional (`sc.w`) instructions from the [A ISA Extension](#) execute as standard load/store bus transactions but with the `rvso` ("reservation set operation") signal being set. It is the task of the [Reservation Set Controller](#) to handle these LR/SC bus transactions accordingly. Note that these reservation set operations are intended for processor-internal usage only (i.e. the reservation state is not available for processor-external modules yet).



#### Reservation Set Controller

See section [Address Space / Reservation Set Controller](#) for more information.

### *Read-Modify-Write Operations*



Read-modify-write operations (like an atomic swap / `amoswap.w`) are **not** supported yet. However, the NEORV32 [Core Libraries](#) provide an emulation wrapper for those unsupported instructions that is based on LR/SC pairs. A demo/program can be found in [sw/example/atomic\\_test](#).

The figure below shows three exemplary bus accesses (1 to 3 from left to right). The `req` signal record represents the CPU-side of the bus interface. For easier understanding the current state of the reservation set is added as `rsv_valid` signal.

1. A load-reservate (LR) instruction using `addr` as address. This instruction returns the loaded data `rdata` via `rsp.data` and also registers a reservation for the address `addr` (`rsv_valid` becomes set).
2. A store-conditional (SC) instruction attempts to write `wdata1` to address `addr`. This SC operation **succeeds**, so `wdata1` is actually written to address `addr`. The successful operation is indicated by a **0** being returned via `rsp.data` together with `ack`. As the LR/SC is completed the registered reservation is invalidated (`rsv_valid` becomes cleared).
3. Another store-conditional (SC) instruction attempts to write `wdata2` to address `addr`. As the reservation set is already invalidated (`rsv_valid` is **0**) the store access fails, so `wdata2` is **not** written to address `addr` at all. The failed operation is indicated by a **1** being returned via `rsp.data` together with `ack`.

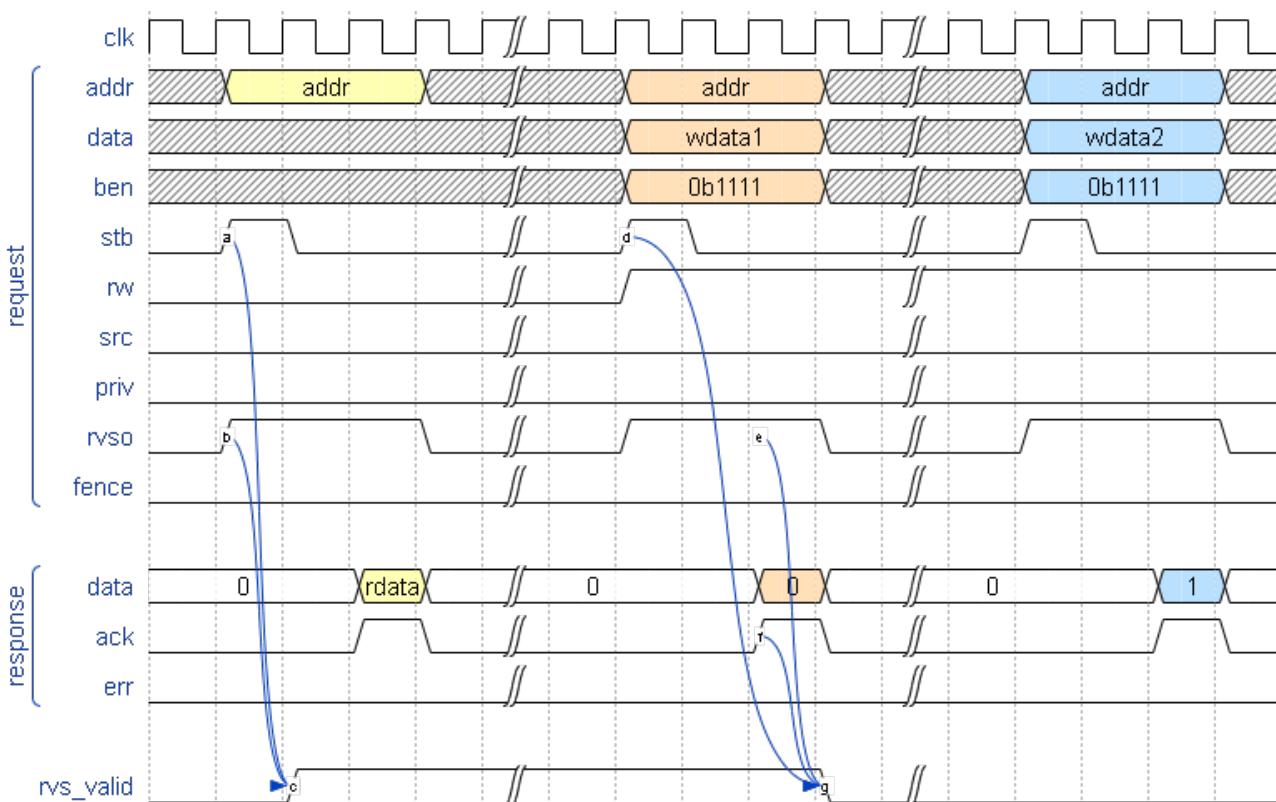


Figure 10. Three Exemplary LR/SC Bus Transactions



SC Status

The "normal" load data mechanism is used to return success/failure of the `sc.w` instruction to the CPU (via the LSB of `rsp.data`).

## 3.6. Instruction Sets and Extensions

The NEORV32 CPU provides several optional RISC-V and custom ISA extensions. The extensions can be enabled/configured via the according [Processor Top Entity - Generics](#). This chapter gives a brief overview of the different ISA extensions.

*Table 46. NEORV32 Instruction Set Extensions*

| Name     | Description   | Enabled by Generic   |
|----------|---|--|
| A        | Atomic memory access instructions                   | CPU_EXTENSION_RISCV_A  |
| B        | Bit-manipulation instructions                       | CPU_EXTENSION_RISCV_B  |
| C        | Compressed (16-bit) instructions                    | CPU_EXTENSION_RISCV_C  |
| E        | Embedded CPU extension (reduced register file size) | CPU_EXTENSION_RISCV_E  |
| I        | Integer base ISA                                    | Enabled if<br>CPU_EXTENSION_RISCV_E is <b>not</b><br>enabled |
| M        | Integer multiplication and division instructions    | CPU_EXTENSION_RISCV_M  |
| U        | Less-privileged <i>user</i> mode extension          | CPU_EXTENSION_RISCV_U  |
| X        | Platform-specific / NEORV32-specific extension      | Always enabled   |
| Zifencei | Instruction stream synchronization instruction      | Always enabled   |
| Zfinx    | Floating-point instructions using integer registers | CPU_EXTENSION_RISCV_Zfinx                                    |
| Zicntr   | Base counters extension                             | CPU_EXTENSION_RISCV_Zicntr                                   |
| Zicond   | Integer conditional operations                      | CPU_EXTENSION_RISCV_Zicond                                   |
| Zicsr    | Control and status register access instructions     | Always enabled   |
| Zihpm    | Hardware performance monitors extension             | CPU_EXTENSION_RISCV_Zihpm                                    |
| Zmmul    | Integer multiplication-only instruction             | CPU_EXTENSION_RISCV_Zmmul                                    |
| Zcfu     | Custom / user-defined instructions                  | CPU_EXTENSION_RISCV_Zcfu                                     |
| Smpmp    | Physical memory protection (PMP) extension          | CPU_EXTENSION_RISCV_Smpmp                                    |
| Sdext    | External debug support extension                    | ON_CHIP_DEBUGGER_EN  |
| Sdtrig   | Trigger module extension                            | ON_CHIP_DEBUGGER_EN  |

### RISC-V ISA Specifications



For more information regarding the RISC-V ISA extensions please refer to the "RISC-V Instruction Set Manual - Volume I: Unprivileged ISA" and "The RISC-V Instruction Set Manual Volume II: Privileged Architecture". A copy of all currently implemented ISA extensions can be found in the projects [docs/references](#) folder.

*Discovering ISA Extensions*

Software can discover available ISA extensions via the `misa` and `mxisa` CSRs or by executing an instruction and checking for an illegal instruction exception (i.e. [Full Virtualization](#)).

*Instruction Cycles*

This chapter shows the CPI values (cycles per instruction) for each individual instruction/type. Note that values reflect *optimal conditions* (i.e. not additional memory delay, no cache misses, no pipeline waits, etc.). To benchmark a certain processor configuration for its setup-specific CPI value please refer to the [sw/example/performance\\_tests](#) test programs.

### 3.6.1. A ISA Extension

The **A** ISA extension adds instructions and mechanisms for atomic memory access operations. Note that the NEORV32 **A** only includes the *load-reservate* (`lr.w`) and *store-conditional* (`sc.w`) instructions - the remaining read-modify-write instructions (like `amoswap`) are **not supported**. However, these missing instructions can be emulated using the LR and SC operations.

*AMO Emulation*

The NEORV32 [Core Libraries](#) provide an emulation wrapper for the missing AMO/read-modify-write instructions that is based on LR/SC pairs. A demo/program can be found in [sw/example/atomic\\_test](#).

Atomic instructions allow to notify an application if a certain memory location has been altered by another instance (like another process running on the same CPU or a DMA access). Hence, they can be used to implement synchronization mechanisms like mutexes and semaphores).

The NEORV32 **A** extension is enabled via the `CPU_EXTENSION_RISCV_A` generic (see [Processor Top Entity - Generics](#)). When enabled the following additional instructions are available.

*Table 47. Instructions and Timing*

| Class                  | Instructions      | Execution cycles |
|------------------------|-------------------|------------------|
| Load-reservate word    | <code>lr.w</code> | 5                |
| Store-conditional word | <code>sc.w</code> | 5                |

The `lr.w` instructions stores one word to a word-aligned address and registers a *reservation set*. The `sc.w` instruction stores a word to a word-aligned address only if the reservation set is still valid. Furthermore, the `sc.w` operations returns the state of the reservation set (0 = reservation set still valid, data has been written; 1 = reservation set was broken, no data has been written). The reservation set is invalidated if another `lr.w` instruction is executed or if any write access to the *reserved* address takes place. Traps and/or CPU privilege level changes do not modify current reservation sets.

**aq and lr Bits**

The instruction word's **aq** and **lr** memory ordering bits are not evaluated by the hardware at all.

**Atomic Memory Access on Hardware Level**

More information regarding the atomic memory accesses and the according reservation sets can be found in section [Reservation Set Controller](#).

**Cache Coherency**

Atomic operations **always bypass** the CPU caches using direct/uncached accesses. Care must be taken to maintain data cache coherency (e.g. by using the **fence** instruction).

### 3.6.2. B ISA Extension

The **B** ISA extension adds instructions for bit-manipulation operations. This ISA extension is implemented as multi-cycle ALU co-process ([rtl/core/neorv32\\_cpu\\_cp\\_bitmanip.vhd](#)). The NEORV32 **B** ISA extension includes the following sub-extensions:

- **Zba** - Address-generation instructions
- **Zbb** - Basic bit-manipulation instructions
- **Zbs** - Single-bit instructions

*Table 48. Instructions and Timing*

| Class            | Instructions  | Execution cycles                    |
|------------------|---|-------------------------------------|
| Arithmetic/logic | <code>min[u] max[u]</code> <code>sext.b sext.h andn orn xnor</code><br><code>zext(pack) rev8(grevi) orc.b(gorci)</code> | 4                                   |
| Shifts           | <code>clz ctz</code>  | $3 + 1..32$ ; FAST_SHIFT: 4         |
| Shifts           | <code>cpop</code>   | 36; FAST_SHIFT: 4                   |
| Shifts           | <code>rol ror[i]</code>   | $4 + shift\_amount$ ; FAST_SHIFT: 4 |
| Shifted-add      | <code>sh1add sh2add sh3add</code>   | 4                                   |
| Single-bit       | <code>sbset[i] sbclr[i] sbinv[i] sbext[i]</code>  | 4                                   |

**Shift Operations**

Shift operations can be accelerated (at the cost of additional logic resources) by enabling the **FAST\_SHIFT\_EN** configuration option that will replace the (time-variant) bit-serial shifter by a (time-constant) barrel shifter.

### 3.6.3. C ISA Extension

The "compressed" ISA extension provides 16-bit encodings of commonly used instructions to reduce

code space size.

*Table 49. Instructions and Timing*

| Class         | Instructions  | Execution cycles         |
|---------------|---|--------------------------|
| ALU           | c.addi4spn c.nop c.add[i] c.li c.addi16sp<br>c.lui c.and[i] c.sub c.xor c.or c.mv | 2                        |
| ALU           | c.srl c.sra i c.slli  | 3 + 1..32; FAST_SHIFT: 4 |
| Branches      | c.beqz c.bnez   | taken: 6; not taken: 3   |
| Jumps / calls | c.jal[r] c.j c.jr   | 6                        |
| Memory access | c.lw c.sw c.lwsp c.swsp   | 4                        |
| System        | c.break   | 3                        |

### 3.6.4. E ISA Extension

The "embedded" ISA extensions reduces the size of the general purpose register file from 32 entries to 16 entries to shrink hardware size. It provides the same instructions as the the base I ISA extensions.



Due to the reduced register file size an alternate toolchain ABI ([ilp32e\\*](#)) is required.

### 3.6.5. I ISA Extension

The I ISA extensions is the base RISC-V integer ISA that is always enabled.

*Table 50. Instructions and Timing*

| Class         | Instructions   | Execution cycles         |
|---------------|--|--------------------------|
| ALU           | add[i] slt[i] slt[i]u xor[i] or[i] and[i]<br>sub lui auipc | 2                        |
| ALU shifts    | sll[i] srl[i] sra[i]                                       | 3 + 1..32; FAST_SHIFT: 4 |
| Branches      | beq bne blt bge bltu bgeu                                  | taken: 6; not taken: 3   |
| Jump/call     | jal[r]   | 6                        |
| Load/store    | lb lh lw lbu lhu sb sh sw                                  | 5                        |
| System        | ecall ebreak   | 3                        |
| Data fence    | fence  | 5                        |
| System        | wfi  | 3                        |
| System        | mret   | 5                        |
| Illegal inst. | -  | 3                        |

**fence** Instruction

 The **fence** instruction word's *predecessor* and *successor* bits (used for memory ordering) are not evaluated by the hardware at all. For the NEORV32 the **fence** instruction behaves exactly like the **fence.i** instruction (see [Zifencei ISA Extension](#)). However, software should still use distinct **fence** and **fence.i** to provide platform-compatibility and to indicate the actual intention of the according fence instruction(s).

**wfi** Instruction

 The **wfi** instruction is used to enter [Sleep Mode](#). Executing the **wfi** instruction in user-mode will raise an illegal instruction exception if the **TW** bit of **mstatus** is set.

## Barrel Shifter

 The shift operations are implemented as multi-cycle ALU co-process ([rtl/core/neorv32\\_cpu\\_cp\\_shifter.vhd](#)). These operations can be accelerated (at the cost of additional logic resources) by enabling the **FAST\_SHIFT\_EN** configuration option that will replace the (time-variant) bit-serial shifter by a (time-constant) barrel shifter.

### 3.6.6. M ISA Extension

Hardware-accelerated integer multiplication and division operations are available via the RISC-V **M** ISA extension. This ISA extension is implemented as multi-cycle ALU co-process ([rtl/core/neorv32\\_cpu\\_cp\\_muldiv.vhd](#)).

Table 51. Instructions and Timing

| Class          | Instructions                 | Execution cycles |
|----------------|------------------------------|------------------|
| Multiplication | <b>mul mulh mulhsu mulhu</b> | 36; FAST_MUL: 4  |
| Division       | <b>div divu rem remu</b>     | 36               |

## DSP Blocks

 Multiplication operations can be accelerated (at the cost of additional logic resources) by enabling the **FAST\_MUL\_EN** configuration option that will replace the (time-variant) bit-serial multiplier by (time-constant) FPGA DSP blocks.

### 3.6.7. U ISA Extension

In addition to the highest-privileged machine-mode, the user-mode ISA extensions adds a second **less-privileged** operation mode. Code executed in user-mode has reduced CSR access rights. Furthermore, user-mode accesses to the address space (like peripheral/IO devices) can be constrained via the physical memory protection. Any kind of privilege rights violation will raise an exception to allow [Full Virtualization](#).

### 3.6.8. X ISA Extension

The NEORV32-specific ISA extensions **X** is always enabled. The most important points of the NEORV32-specific extensions are:

- \* The CPU provides 16 *fast interrupt* interrupts (**FIRQ**), which are controlled via custom bits in the **mie** and **mip** CSRs. These extensions are mapped to CSR bits, that are available for custom use according to the RISC-V specs. Also, custom trap codes for **mcause** are implemented.
- \* All undefined/unimplemented/malformed/illegal instructions do raise an illegal instruction exception (see [Full Virtualization](#)).
- \* There are [NEORV32-Specific CSRs](#).

### 3.6.9. Zifencei ISA Extension

The **Zifencei** CPU extension allows manual synchronization of the instruction stream. This extension is always enabled.

#### NEORV32 Fence Instructions



The NEORV32 treats both fence instructions (**fence** = data fence, **fence.i** = instruction fence) in exactly the same way. Both instructions cause a flush of the CPU's instruction prefetch buffer and also send a fence request via the system bus (see [Bus Interface](#)). This system bus fence operation will, for example, clear/flush all downstream caches.

*Table 52. Instructions and Timing*

| Class             | Instructions   | Execution cycles |
|-------------------|----------------|------------------|
| Instruction fence | <b>fence.i</b> | 5                |

### 3.6.10. Zfinx ISA Extension

The **Zfinx** floating-point extension is an *alternative* of the standard **F** floating-point ISA extension. It also uses the integer register file **x** to store and operate on floating-point data instead of a dedicated floating-point register file. Thus, the **Zfinx** extension requires less hardware resources and features faster context changes. This also implies that there are NO dedicated **f** register file-related load/store or move instructions. The **Zfinx** extension's floating-point unit is controlled via dedicated [Floating-Point CSRs](#). This ISA extension is implemented as multi-cycle ALU co-process ([rtl/core/neorv32\\_cpu\\_cp\\_fpu.vhd](#)).

#### Fused Multiply-Add and Division Instructions



Fused multiply-add instructions **f[n]m[add/sub].s** are not supported! Division **fdiv.s** and square root **fsqrt.s** instructions are not supported yet!

#### Subnormal Number



Subnormal numbers ("de-normalized" numbers, i.e. exponent = 0) are not supported by the NEORV32 FPU. Subnormal numbers are *flushed to zero* setting them to +/- 0 before being processed by **any** FPU operation. If a computational instruction generates a subnormal result it is also flushed to zero during

normalization.

Table 53. Instructions and Timing

| Class      | Instructions                          | Execution cycles |
|------------|---------------------------------------|------------------|
| Arithmetic | fadd.s                                | 110              |
| Arithmetic | fsub.s                                | 112              |
| Arithmetic | fmul.s                                | 22               |
| Compare    | fmin.s fmax.s feq.s flt.s fle.s       | 13               |
| Conversion | fcvt.w.s fcvt.wu.s fcvt.s.w fcvt.s.wu | 48               |
| Misc       | fsgnj.s fsgnjn.s fsgnjx.s fcclass.s   | 12               |

### 3.6.11. Zicntr ISA Extension

The **Zicntr** ISA extension adds the basic `cycle[h]`, `mcycle[h]`, `instret[h]` and `minstret[h]` counter CSRs. Section [\(Machine\) Counter and Timer CSRs](#) shows a list of all **Zicntr**-related CSRs.



The user-mode `time[h]` CSRs are **not implemented**. Any access will trap allowing the trap handler to retrieve system time from the [Machine System Timer \(MTIME\)](#).



This extensions is stated as *mandatory* by the RISC-V spec. However, area-constrained setups may remove support for these counters.

### 3.6.12. Zicond ISA Extension

The **Zicond** ISA extension adds integer conditional move primitives that allow to implement branch-less control flows. It is enabled by the top's `CPU_EXTENSION_RISCV_Zicond` generic. This ISA extension is implemented as multi-cycle ALU co-process (`rtl/core/neorv32_cpu_cp_cond.vhd`).

Table 54. Instructions and Timing

| Class       | Instructions        | Execution cycles |
|-------------|---------------------|------------------|
| Conditional | czero.eqz czero.nez | 3                |

### 3.6.13. Zicsr ISA Extension

This ISA extensions provides instructions for accessing the [Control and Status Registers \(CSRs\)](#) as well as further privileged-architecture extensions. This extension is mandatory and cannot be disabled. Hence, there is no generic for enabling/disabling this ISA extension.



If `rd=x0` for the `csrrw[i]` instructions there will be no actual read access to the according CSR. However, access privileges are still enforced so these instruction variants *do* cause side-effects (the RISC-V spec. state that these combinations "shall" not cause any side-effects).

Table 55. Instructions and Timing

| Class  | Instructions                            | Execution cycles |
|--------|---|------------------|
| System | <code>csrrw[i] csrrs[i] csrrc[i]</code> | 3                |

### 3.6.14. **Zihpm** ISA Extension

In addition to the base counters the NEORV32 CPU provides up to 13 hardware performance monitors (HPM 3..15), which can be used to benchmark applications. Each HPM consists of an N-bit wide counter (split in a high-word 32-bit CSR and a low-word 32-bit CSR), where N is defined via the top's `HPM_CNT_WIDTH` generic and a corresponding event configuration CSR. The event configuration CSR defines the architectural events that lead to an increment of the associated HPM counter. See section [Hardware Performance Monitors \(HPM\) CSRs](#) for a list of all HPM-related CSRs and event configurations.



Auto-increment of the HPMs can be deactivated individually via the `mcountinhibit` CSR.

### 3.6.15. **Zmmul** - ISA Extension

This is a sub-extension of the [M ISA Extension](#) ISA extension. It implements only the multiplication operations of the [M](#) extensions and is intended for size-constrained setups that require hardware-based integer multiplications but not hardware-based divisions, which will be computed entirely in software.

### 3.6.16. **Zxfc** ISA Extension

The [Zxfc](#) presents a NEORV32-specific ISA extension. It adds the [Custom Functions Unit \(CFU\)](#) to the CPU core, which allows to add custom RISC-V instructions to the processor core. For detailed information regarding the CFU, its hardware and the according software interface see section [Custom Functions Unit \(CFU\)](#).

Software can utilize the custom instructions by using *intrinsics*, which are basically inline assembly functions that behave like regular C functions but that evaluate to a single custom instruction word (no calling overhead at all).

### 3.6.17. **Smpmp** ISA Extension

The NEORV32 physical memory protection (PMP) provides an elementary memory protection mechanism that can be used to constrain read, write and execute rights of arbitrary memory regions. The NEORV32 PMP is fully compatible to the RISC-V Privileged Architecture Specifications. In general, the PMP can **grant permissions to user mode**, which by default has none, and can **revoke permissions from M-mode**, which by default has full permissions. The PMP is configured via the [Machine Physical Memory Protection CSRs](#).

Several [Processor Top Entity - Generics](#) are provided to fine-tune the CPU's PMP capabilities: \* `PMP_NUM_REGIONS` defines the number of implemented PMP region \* `PMP_MIN_GRANULARITY` defines the

minimal granularity of each region \* `PMP_TOR_MODE_EN` controls the implementation of the top-of-region (TOR) mode \* `PMP_NAP_MODE_EN` controls the implementation of the naturally-aligned-power-of-two (NA4 and NAPOT) modes

*PMP Rules when in Debug Mode*



When in debug-mode all PMP rules are ignored making the debugger have maximum access rights.



Instruction fetches are also triggered when denied by a certain PMP rule. However, the fetched instruction(s) will not be executed and will not change CPU core state.

### 3.6.18. `Sdext` ISA Extension

This ISA extension enables the RISC-V-compatible "external debug support" by implementing the CPU "debug mode", which is required for the on-chip debugger. See section [On-Chip Debugger \(OCD\) / CPU Debug Mode](#) for more information.

*Table 56. Instructions and Timing*

| Class  | Instructions      | Execution cycles |
|--------|-------------------|------------------|
| System | <code>dret</code> | 5                |

### 3.6.19. `Sdtrig` ISA Extension

This ISA extension implements the RISC-V-compatible "trigger module". See section [On-Chip Debugger \(OCD\) / Trigger Module](#) for more information.

## 3.7. Custom Functions Unit (CFU)

The Custom Functions Unit (CFU) is the central part of the NEORV32-specific [Zxfcuh ISA Extension](#) and represents the actual hardware module that can be used to implement **custom RISC-V instructions**.

The CFU is intended for operations that are inefficient in terms of performance, latency, energy consumption or program memory requirements when implemented entirely in software. Some potential application fields and exemplary use-cases might include:

- **AI:** sub-word / vector / SIMD operations like processing all four bytes of a 32-bit data word in parallel
- **Cryptographic:** bit substitution and permutation
- **Communication:** conversions like binary to gray-code; multiply-add operations
- **Image processing:** look-up-tables for color space transformations
- implementing instructions from **other RISC-V ISA extensions** that are not yet supported by NEORV32

The CFU is not intended for complex and **CPU-independent** functional units that implement complete accelerators (like block-based AES encryption). These kind of accelerators should be implemented as memory-mapped [Custom Functions Subsystem \(CFS\)](#). A comparison of all NEORV32-specific chip-internal hardware extension options is provided in the user guide section [Adding Custom Hardware Modules](#).



### Default CFU Hardware Example

The default CFU module ([rtl/core/neorv32\\_cpu\\_cp\\_cfu.vhd](#)) implements the *Extended Tiny Encryption Algorithm (XTEA)* as "real world" application example.

### 3.7.1. CFU Instruction Formats

The custom instructions executed by the CFU utilize a specific opcode space in the **rv32** 32-bit instruction encoding space that has been explicitly reserved for user-defined extensions by the RISC-V specifications ("Guaranteed Non-Standard Encoding Space"). The NEORV32 CFU uses the **custom-\*** opcodes to identify the instructions implemented by the CFU and to differentiate between the predefined instruction formats. The according binary encoding of these opcodes is shown below:

- **custom-0: 0001011** RISC-V standard, used for [CFU R3-Type Instructions](#)
- **custom-1: 0101011** RISC-V standard, used for [CFU R4-Type Instructions](#)
- **custom-2: 1011011** NEORV32-specific, used for [CFU R5-Type Instructions type A](#)
- **custom-3: 1111011** NEORV32-specific, used for [CFU R5-Type Instructions type B](#)



The four presented instructions types/formats are predefined to allow an easy integration framework. However, system designers are free to ignore those and

use their own instruction types and formats.

## CFU R3-Type Instructions

The R3-type CFU instructions operate on two source registers **rs1** and **rs2** and return the processing result to the destination register **rd**. The actual operation can be defined by using the **funct7** and **funct3** bit fields. These immediates can also be used to pass additional data to the CFU like offsets, look-up-tables addresses or shift-amounts. However, the actual functionality is entirely user-defined. Note that all immediate values are always compile-time-static.

Example operation:  $rd \leftarrow rs1 \text{ xnor } rs2$  (bit-wise XNOR)

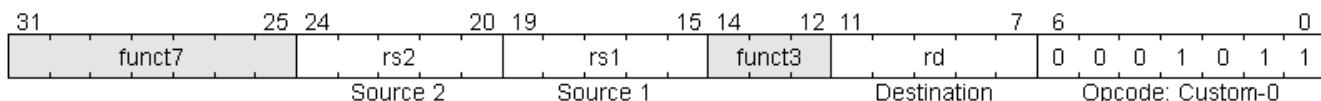


Figure 11. CFU R3-type instruction format

- **funct7**: 7-bit immediate (further operand data or function select)
- **rs2**: address of second source register (32-bit source data)
- **rs1**: address of first source register (32-bit source data)
- **funct3**: 3-bit immediate (further operand data or function select)
- **rd**: address of destination register (for the 32-bit processing result)
- **opcode**: **0001011** (RISC-V "custom-0" opcode)



### RISC-V compatibility

The CFU R3-type instruction format is compliant to the RISC-V ISA specification.

### Instruction encoding space



By using the **funct7** and **funct3** bit fields entirely for selecting the actual operation a total of 1024 custom R3-type instructions can be implemented ( $7\text{-bit} + 3\text{-bit} = 10$  bit  $\rightarrow$  1024 different values).

## CFU R4-Type Instructions

The R4-type CFU instructions operate on three source registers **rs1**, **rs2** and **rs3** and return the processing result to the destination register **rd**. The actual operation can be defined by using the **funct3** bit field. Alternatively, this immediate can also be used to pass additional data to the CFU like offsets, look-up-tables addresses or shift-amounts. However, the actual functionality is entirely user-defined. Note that all immediate values are always compile-time-static.

Example operation:  $rd \leftarrow (rs1 * rs2 + rs3)[31:0]$  (multiply-and-accumulate; "MAC")

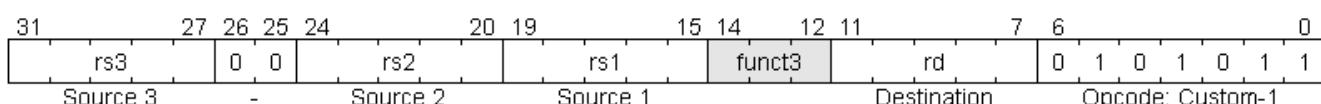


Figure 12. CFU R4-type instruction format

- **rs3**: address of third source register (32-bit source data)
- **rs2**: address of second source register (32-bit source data)
- **rs1**: address of first source register (32-bit source data)
- **funct3**: 3-bit immediate (further operand data or function select)
- **rd**: address of destination register (for the 32-bit processing result)
- **opcode**: **0101011** (RISC-V "custom-1" opcode)



#### *RISC-V compatibility*

The CFU R4-type instruction format is compliant to the RISC-V ISA specification.



#### *Unused instruction bits*

The RISC-V ISA specification defines bits [26:25] of the R4-type instruction word to be all-zero. These bits are ignored by the hardware (CFU and illegal instruction check logic) and should be set to all-zero to preserve compatibility with future ISA spec. versions.



#### *Instruction encoding space*

By using the **funct3** bit field entirely for selecting the actual operation a total of 8 custom R4-type instructions can be implemented (3-bit → 8 different values).

## CFU R5-Type Instructions

The R5-type CFU instructions operate on four source registers **rs1**, **rs2**, **rs3** and **rs4** and return the processing result to the destination register **rd**. As all bits of the instruction word are used to encode the five registers and the opcode, no further immediate bits are available to specify the actual operation. There are two different R5-type instruction with two different opcodes available. Hence, only two R5-type operations can be implemented by default.

Example operation:  $rd \leftarrow rs1 \& rs2 \& rs3 \& rs4$  (bit-wise AND of 4 operands)

|          |          |          |          |          |    |             |                  |    |    |    |   |   |   |
|----------|----------|----------|----------|----------|----|-------------|------------------|----|----|----|---|---|---|
| 31       | 27       | 26       | 25       | 24       | 20 | 19          | 15               | 14 | 12 | 11 | 7 | 6 | 0 |
| rs3      | rs4.hi   | rs2      | rs1      | rs4.lo   | rd | Destination | Opcode: Custom-2 |    |    |    |   |   |   |
| Source 3 | Source 4 | Source 2 | Source 1 | Source 4 |    |             |                  |    |    |    |   |   |   |

Figure 13. CFU R5-type instruction A format

|          |          |          |          |          |    |             |                  |    |    |    |   |   |   |
|----------|----------|----------|----------|----------|----|-------------|------------------|----|----|----|---|---|---|
| 31       | 27       | 26       | 25       | 24       | 20 | 19          | 15               | 14 | 12 | 11 | 7 | 6 | 0 |
| rs3      | rs4.hi   | rs2      | rs1      | rs4.lo   | rd | Destination | Opcode: Custom-3 |    |    |    |   |   |   |
| Source 3 | Source 4 | Source 2 | Source 1 | Source 4 |    |             |                  |    |    |    |   |   |   |

Figure 14. CFU R5-type instruction B format

- **rs4.hi** & **rs4.lo**: address of fourth source register (32-bit source data)
- **rs3**: address of third source register (32-bit source data)
- **rs2**: address of second source register (32-bit source data)
- **rs1**: address of first source register (32-bit source data)

- `rd`: address of destination register (for the 32-bit processing result)
- `opcode`: `1011011` (RISC-V "custom-2" opcode) and/or `1111011` (RISC-V "custom-3" opcode)

#### *RISC-V compatibility*



The RISC-V ISA specifications does not specify a R5-type instruction format. Hence, this instruction format is NEORV32-specific.

#### *Instruction encoding space*



There are no immediate fields in the CFU R5-type instruction so the actual operation is specified entirely by the opcode resulting in just two different operations out of the box. However, another CFU instruction (like a R3-type instruction) can be used to "program" the actual operation of a R5-type instruction by writing operation information to a CFU-internal "command" register.

### 3.7.2. Using Custom Instructions in Software

The custom instructions provided by the CFU can be used in plain C code by using **intrinsics**. Intrinsics behave like "normal" C functions but under the hood they are a set of macros that hide the complexity of inline assembly. Using intrinsics removes the need to modify the compiler, built-in libraries or the assembler when using custom instructions. Each intrinsic will be compiled into a single 32-bit instruction word providing maximum code efficiency.

#### *CFU Example Program*



There is an example program for the CFU, which shows how to use the *default* CFU hardware module. This example program is located in [sw/example/demo\\_cfu](#).

The NEORV32 software framework provides four pre-defined prototypes for custom instructions, which are defined in [sw/lib/include/neorv32\\_cpu\\_cfu.h](#):

#### *Listing 9. CFU instruction prototypes*

```
neorv32_cfu_r3_instr(funct7, funct3, rs1, rs2) // R3-type instructions
neorv32_cfu_r4_instr(funct3, rs1, rs2, rs3)    // R4-type instructions
neorv32_cfu_r5_instr_a(rs1, rs2, rs3, rs4)     // R5-type instruction A
neorv32_cfu_r5_instr_b(rs1, rs2, rs3, rs4)     // R5-type instruction B
```

The intrinsic functions always return a 32-bit value of type `uint32_t` (the processing result), which can be discarded if not needed. Each intrinsic function requires several arguments depending on the instruction type/format:

- `funct7` - 7-bit immediate (R3-type only)
- `funct3` - 3-bit immediate (R3-type, R4-type)
- `rs1` - source operand 1, 32-bit (R3-type, R4-type)
- `rs2` - source operand 2, 32-bit (R3-type, R4-type)

- `rs3` - source operand 3, 32-bit (R3-type, R4-type, R5-type)
- `rs4` - source operand 4, 32-bit (R4-type, R4-type, R5-type)

The `funct3` and `funct7` bit-fields are used to pass 3-bit or 7-bit literals to the CFU. The `rs1`, `rs2`, `rs3` and `r4` arguments pass the actual data to the CFU. These register arguments can be populated with variables or literals. The following example shows how to pass arguments:

*Listing 10. CFU instruction usage example*

```
uint32_t tmp = some_function();
...
uint32_t res = neorv32_cfu_r3_instr(0b0000000, 0b101, tmp, 123);
uint32_t foo = neorv32_cfu_r4_instr(0b011, tmp, res, (uint32_t)some_array[i]);
uint32_t bar = neorv32_cfu_r5_instr_a(tmp, res, foo, tmp);
```

### 3.7.3. CFU Control and Status Registers (CFU-CSRs)

The CPU provides up to four control and status registers (`cfureg*`) to be used within the CFU. These CSRs are mapped to the "custom user-mode read/write" CSR address space, which is explicitly reserved for platform-specific application by the RISC-V spec. For example, these CSRs can be used to pass additional operands to the CFU, to obtain additional results, to check processing status or to program operation modes.

*Listing 11. CFU CSR Access Example*

```
neorv32_cpu_csr_write(CSR_CFUREG0, 0xabcdabcd); // write data to CFU CSR 0
uint32_t tmp = neorv32_cpu_csr_read(CSR_CFUREG3); // read data from CFU CSR 3
```

#### Additional CFU-internal CSRs



If more than four CFU-internal CSRs are required the designer can implement an "indirect access mechanism" based on just two of the default CSRs: one CSR is used to configure the index while the other is used as alias to exchange data with the indexed CFU-internal CSR - this concept is similar to the RISC-V Indirect CSR Access Extension Specification ([Smcsrind](#)).

#### Security Considerations



The CFU CSRs are mapped to the user-mode CSR space so software running at *any privilege level* can access these CSRs. However, accesses can be constrained to certain privilege level (see [Custom Instructions Hardware](#)).

### 3.7.4. Custom Instructions Hardware

The actual functionality of the CFU's custom instructions is defined by the user-defined logic inside the CFU hardware module `rtl/core/neorv32_cpu_cp_cfu.vhd`. This file is highly commented to illustrate the hardware design considerations.

CFU operations can be entirely combinatorial (like bit-reversal) so the result is available at the end of the current clock cycle. Operations can also take several clock cycles to complete (like multiplications) and may also include internal states and memories. The CFU's internal control unit takes care of interfacing the custom user logic to the CPU pipeline.

#### *CFU Hardware Resource Requirements*



Enabling the CFU and actually implementing R4-type and/or R5-type instructions (or more precisely, using the according operands for the CFU hardware) will add one or two, respectively, additional read ports to the core's register file significantly increasing resource requirements.

#### *CFU Access Privilege Levels*



The CFU is accessible from all privilege modes (including CFU-internal registers accessed via the indirects CSR access mechanism). It is the task of the CFU designers to add according access-constraining logic if certain CFU states shall not be exposed to all privilege levels (i.e. encryption keys).

#### *CFU Execution Time*



The CFU has to complete computation within a **bound time window**. Otherwise, the CFU operation is terminated by the hardware and an illegal instruction exception is raised. See section [CPU Arithmetic Logic Unit](#) for more information.

#### *CFU Exception*



The CFU can intentionally raise an illegal instruction exception by not asserting the `done` at all causing an execution timeout. For example this can be used to signal invalid configurations/operations to the runtime environment. See the CFU's VHDL file for more information.

## 3.8. Control and Status Registers (CSRs)

The following table shows a summary of all available NEORV32 CSRs. The address field defines the CSR address for the CSR access instructions. The "Name [ASM]" column provides the CSR name aliases that can be used in (inline) assembly. The "Name [C]" column lists the name aliases that are defined by the NEORV32 core library. These can be used in plain C code. The "Access" column shows the minimal required privilege mode required for accessing the according CSR (**M** = machine-mode, **U** = user-mode, **D** = debug-mode) and the read/write capabilities (**RW** = read-write, **RO** = read-only)

*Unused, Reserved, Unimplemented and Disabled CSRs*



All CSRs and CSR bits that are not listed in the table below are *unimplemented* and are *hardwired to zero*. Additionally, CSRs that are unavailable ("disabled") because the according ISA extension is not enabled are also considered *unimplemented* and are also hardwired to zero. Any access to such a CSR will raise an illegal instruction exception. All writable CSRs provide **WARL** behavior (write all values; read only legal values). Application software should always read back a CSR after writing to check if the targeted bits can actually be modified.

Table 57. NEORV32 Control and Status Registers (CSRs)

| Address                        | Name [ASM]     | Name [C]           | Access      | Description                          |
|--------------------------------|----------------|--------------------|-------------|--------------------------------------|
| <b>Floating-Point CSRs</b>     |                |                    |             |                                      |
| 0x001                          | <b>fflags</b>  | <b>CSR_FFLAGS</b>  | UR<br>W     | Floating-point accrued exceptions    |
| 0x002                          | <b>frm</b>     | <b>CSR_FRM</b>     | UR<br>W     | Floating-point dynamic rounding mode |
| 0x003                          | <b>fcsr</b>    | <b>CSR_FCSR</b>    | UR<br>W     | Floating-point control and status    |
| <b>Machine Trap Setup CSRs</b> |                |                    |             |                                      |
| 0x300                          | <b>mstatus</b> | <b>CSR_MSTATUS</b> | M<br>R<br>W | Machine status register - low word   |
| 0x301                          | <b>misa</b>    | <b>CSR_MISA</b>    | M<br>R<br>W | Machine CPU ISA and extensions       |
| 0x304                          | <b>mie</b>     | <b>CSR_MIE</b>     | M<br>R<br>W | Machine interrupt enable register    |

| Address | Name [ASM]              | Name [C]                    | Access      | Description                                     |
|---------|-------------------------|-----------------------------|-------------|---|
| 0x305   | <code>mtvec</code>      | <code>CSR_MTVEC</code>      | M<br>R<br>W | Machine trap-handler base address for ALL traps |
| 0x306   | <code>mcounteren</code> | <code>CSR_MCOUNTEREN</code> | M<br>R<br>W | Machine counter-enable register                 |
| 0x310   | <code>mstatush</code>   | <code>CSR_MSTATUSH</code>   | M<br>R<br>W | Machine status register - high word             |

### Machine Configuration CSRs

|       |                       |                           |             |  |
|-------|-----------------------|---------------------------|-------------|--|
| 0x30a | <code>menvcfg</code>  | <code>CSR_MENVCFG</code>  | M<br>R<br>W | Machine environment configuration register - low word  |
| 0x31a | <code>menvcfgh</code> | <code>CSR_MENVCFGH</code> | M<br>R<br>W | Machine environment configuration register - high word |

### Machine Counter Setup CSRs

|       |                            |                                |             |                                  |
|-------|----------------------------|--------------------------------|-------------|----------------------------------|
| 0x320 | <code>mcountinhibit</code> | <code>CSR_MCOUNTINHIBIT</code> | M<br>R<br>W | Machine counter-inhibit register |
|-------|----------------------------|--------------------------------|-------------|----------------------------------|

### Machine Trap Handling CSRs

|       |                       |                           |             |                                    |
|-------|-----------------------|---------------------------|-------------|------------------------------------|
| 0x340 | <code>mscratch</code> | <code>CSR_MSCRATCH</code> | M<br>R<br>W | Machine scratch register           |
| 0x341 | <code>mepc</code>     | <code>CSR_MEPC</code>     | M<br>R<br>W | Machine exception program counter  |
| 0x342 | <code>mcause</code>   | <code>CSR_MCAUSE</code>   | M<br>R<br>W | Machine trap cause                 |
| 0x343 | <code>mtval</code>    | <code>CSR_MTVAL</code>    | M<br>R<br>W | Machine trap value                 |
| 0x344 | <code>mip</code>      | <code>CSR_MIP</code>      | M<br>R<br>W | Machine interrupt pending register |

| Address  | Name [ASM]                             | Name [C]                                       | Access      | Description  |
|--|--|--|-------------|--|
| 0x34a  | <code>mtinst</code>                    | <code>CSR_MTINST</code>                        | M<br>R<br>W | Machine trap instruction                           |
| <b>Machine Physical Memory Protection CSRs</b> |  |  |             |  |
| 0x3a0 ..<br>0x303                              | <code>pmpcfg0 ..<br/>pmpcfg3</code>    | <code>CSR_PMPCFG0 ..<br/>CSR_PMPCFG3</code>    | M<br>R<br>W | Physical memory protection configuration registers |
| 0x3b0 ..<br>0x3bf                              | <code>pmpaddr0 ..<br/>pmpaddr15</code> | <code>CSR_PMPADDR0 ..<br/>CSR_PMPADDR15</code> | M<br>R<br>W | Physical memory protection address registers       |
| <b>Trigger Module CSRs</b>                     |  |  |             |  |
| 0x7a0  | <code>tselect</code>                   | <code>CSR_TSELECT</code>                       | M<br>R<br>W | Trigger select register                            |
| 0x7a1  | <code>tdata1</code>                    | <code>CSR_TDATA1</code>                        | M<br>R<br>W | Trigger data register 1                            |
| 0x7a2  | <code>tdata2</code>                    | <code>CSR_TDATA2</code>                        | M<br>R<br>W | Trigger data register 2                            |
| 0x7a4  | <code>tinfo</code>                     | <code>CSR_TINFO</code>                         | M<br>R<br>W | Trigger information register                       |
| <b>CPU Debug Mode CSRs</b>                     |  |  |             |  |
| 0x7b0  | <code>dcsr</code>                      | -  | D<br>R<br>W | Debug control and status register                  |
| 0x7b1  | <code>dpc</code>                       | -  | D<br>R<br>W | Debug program counter                              |
| 0x7b2  | <code>dscratch0</code>                 | -  | D<br>R<br>W | Debug scratch register 0                           |
| <b>Custom Functions Unit (CFU) CSRs</b>        |  |  |             |  |
| 0x800 ..<br>0x803                              | <code>cfureg0 ..<br/>cfureg3</code>    | <code>CSR_CFUCREG0 ..<br/>CSR_CFUCREG3</code>  | U<br>R<br>W | Custom CFU registers 0 to 3                        |
| <b>(Machine) Counter and Timer CSRs</b>        |  |  |             |  |

| Address | Name [ASM]             | Name [C]                   | Access      | Description                                   |
|---------|------------------------|----------------------------|-------------|---|
| 0xb00   | <code>mcycle</code>    | <code>CSR_MCYCLE</code>    | M<br>R<br>W | Machine cycle counter low word                |
| 0xb02   | <code>minstret</code>  | <code>CSR_MINSTRET</code>  | M<br>R<br>W | Machine instruction-retired counter low word  |
| 0xb80   | <code>mcycleh</code>   | <code>CSR_MCYCLEH</code>   | M<br>R<br>W | Machine cycle counter high word               |
| 0xb82   | <code>minstreth</code> | <code>CSR_MINSTRETH</code> | M<br>R<br>W | Machine instruction-retired counter high word |
| 0xc00   | <code>cycle</code>     | <code>CSR_CYCLE</code>     | UR<br>O     | Cycle counter low word                        |
| 0xc02   | <code>instret</code>   | <code>CSR_INSTRET</code>   | UR<br>O     | Instruction-retired counter low word          |
| 0xc80   | <code>cycleh</code>    | <code>CSR_CYCLEH</code>    | UR<br>O     | Cycle counter high word                       |
| 0xc82   | <code>instreth</code>  | <code>CSR_INSTRETH</code>  | UR<br>O     | Instruction-retired counter high word         |

**Hardware Performance Monitors (HPM) CSRs**

|                   |  |  |             |   |
|-------------------|--|--|-------------|---|
| 0x323 ..<br>0x32f | <code>mhpmevent3 ..</code><br><code>mhpmevent15</code>       | <code>CSR_MHPMEVENT3 ..</code><br><code>CSR_MHPMEVENT15</code>       | M<br>R<br>W | Machine performance-monitoring event select for counter 3..15 |
| 0xb03 ..<br>0xb0f | <code>mhpmcOUNTER3 ..</code><br><code>mhpmcOUNTER15</code>   | <code>CSR_MHPMCOUNTER3 ..</code><br><code>CSR_MHPMCOUNTER15</code>   | M<br>R<br>W | Machine performance-monitoring counter 3..15 low word         |
| 0xb83 ..<br>0xb8f | <code>mhpmcOUNTER3h ..</code><br><code>mhpmcOUNTER15h</code> | <code>CSR_MHPMCOUNTER3H ..</code><br><code>CSR_MHPMCOUNTER15H</code> | M<br>R<br>W | Machine performance-monitoring counter 3..15 high word        |
| 0xc03 ..<br>0xc0f | <code>hpmcounter3 ..</code><br><code>hpmcounter15</code>     | <code>CSR_HPMCOUNTER3 ..</code><br><code>CSR_HPMCOUNTER15</code>     | UR<br>O     | User performance-monitoring counter 3..15 low word            |
| 0xc83 ..<br>0xc8f | <code>hpmcounter3h ..</code><br><code>hpmcounter15h</code>   | <code>CSR_HPMCOUNTER3H ..</code><br><code>CSR_HPMCOUNTER15H</code>   | UR<br>O     | User performance-monitoring counter 3..15 high word           |

**Machine Information CSRs**

| Address | Name [ASM]              | Name [C]                    | Access  | Description                            |
|---------|-------------------------|-----------------------------|---------|--|
| 0xf11   | <code>mvendorid</code>  | <code>CSR_MVENDORID</code>  | M<br>RO | Machine vendor ID                      |
| 0xf12   | <code>marchid</code>    | <code>CSR_MARCHID</code>    | M<br>RO | Machine architecture ID                |
| 0xf13   | <code>mimpid</code>     | <code>CSR_MIMPID</code>     | M<br>RO | Machine implementation ID / version    |
| 0xf14   | <code>mhartid</code>    | <code>CSR_MHARTID</code>    | M<br>RO | Machine hardware thread ID             |
| 0xf15   | <code>mconfigptr</code> | <code>CSR_MCONFIGPTR</code> | M<br>RO | Machine configuration pointer register |

### NEORV32-Specific CSRs

|       |                    |                        |         |  |
|-------|--------------------|------------------------|---------|--|
| 0xfc0 | <code>mxisa</code> | <code>CSR_MXISA</code> | M<br>RO | NEORV32-specific "eXtended" machine CPU ISA and extensions |
|-------|--------------------|------------------------|---------|--|

### 3.8.1. Floating-Point CSRs

#### **fflags**

Name Floating-point accrued exceptions

Address **0x001**

Reset **0x00000000**

value

ISA **Zicsr & Zfinx**

Description FPU status flags.  
on

Table 58. **fflags** CSR bits

| Bit | R/W | Function                     |
|-----|-----|------------------------------|
| 0   | r/w | <b>NX:</b> inexact           |
| 1   | r/w | <b>UF:</b> underflow         |
| 2   | r/w | <b>OF:</b> overflow          |
| 3   | r/w | <b>DZ:</b> division by zero  |
| 4   | r/w | <b>NV:</b> invalid operation |

#### **frm**

Name Floating-point dynamic rounding mode

Address **0x002**

Reset **0x00000000**

value

ISA **Zicsr & Zfinx**

Description The **frm** CSR is used to configure the rounding mode of the FPU.  
on

Table 59. **frm** CSR bits

| Bit | R/W | Function      |
|-----|-----|---------------|
| 2:0 | r/w | Rounding mode |

#### **fcsr**

Name Floating-point control and status register

Address **0x003**

Reset **0x00000000**  
value

ISA **Zicsr & Zfinx**

Description The **fcsr** provides combined access to the **fflags** and **frm** flags.  
on

Table 60. **fcsr** CSR bits

| Bit | R/W | Function                                  |
|-----|-----|---|
| 4:0 | r/w | Accrued exception flags ( <b>fflags</b> ) |
| 7:5 | r/w | Rounding mode ( <b>frm</b> )              |

### 3.8.2. Machine Trap Setup CSRs

#### `mstatus`

Name Machine status register - low word

Address `0x300`

Reset `0x00001800`

value

ISA `Zicsr`

Description The `mstatus` CSR is used to configure general machine environment parameters.

on

Table 61. `mstatus` CSR bits

| Bit   | Name [C]   | R/W | Function  |
|-------|--|-----|---|
| 3     | <code>CSR_MSTATUS_MIE</code>                                       | r/w | <b>MIE:</b> Machine-mode interrupt enable flag  |
| 7     | <code>CSR_MSTATUS_MPIE</code>                                      | r/w | <b>MPIE:</b> Previous machine-mode interrupt enable flag state  |
| 12:11 | <code>CSR_MSTATUS_MPP_H</code> :<br><code>CSR_MSTATUS_MPP_L</code> | r/w | <b>MPP:</b> Previous machine privilege mode, <code>11</code> = machine-mode "M", <code>00</code> = user-mode "U"; other values will fall-back to machine-mode |
| 17    | <code>CSR_MSTATUS_MPRV</code>                                      | r/w | <b>MPRV:</b> Effective privilege mode for load/stores; use <b>MPP</b> as effective privilege mode when set; hardwired to zero if user-mode not implemented    |
| 21    | <code>CSR_MSTATUS_TW</code>  | r/w | <b>TW:</b> Trap on execution of <code>wfi</code> instruction in user mode when set; hardwired to zero if user-mode not implemented                            |



If the core is in user-mode, machine-mode interrupts are globally **enabled** even if `mstatus.mie` is cleared: "Interrupts for higher-privilege modes,  $y > x$ , are always globally enabled regardless of the setting of the global `yIE` bit for the higher-privilege mode." - RISC-V ISA Spec.

#### `misa`

Name ISA and extensions

Address `0x301`

Reset **DEFINED**, according to enabled ISA extensions

value

ISA `Zicsr`

Description The `misa` CSR provides information regarding the availability of basic RISC-V ISA extensions.



The NEORV32 `misa` CSR is read-only. Hence, active CPU extensions are entirely defined by pre-synthesis configurations and cannot be switched on/off during runtime. For compatibility reasons any write access to this CSR is simply ignored and will *not* cause an illegal instruction exception.

Table 62. `misa` CSR bits

| Bit   | Name [C]                              | R/W | Function   |
|-------|---------------------------------------|-----|--|
| 0     | <code>CSR_MISA_A_EXT</code>           | r/- | A: CPU extension (atomic memory access) available, set when <b>A ISA Extension</b> enabled   |
| 1     | <code>CSR_MISA_B_EXT</code>           | r/- | B: CPU extension (bit-manipulation) available, set when <b>B ISA Extension</b> enabled       |
| 2     | <code>CSR_MISA_C_EXT</code>           | r/- | C: CPU extension (compressed instruction) available, set when <b>C ISA Extension</b> enabled |
| 4     | <code>CSR_MISA_E_EXT</code>           | r/- | E: CPU extension (embedded) available, set when <b>E ISA Extension</b> enabled               |
| 8     | <code>CSR_MISA_I_EXT</code>           | r/- | I: CPU base ISA, cleared when <b>E ISA Extension</b> enabled                                 |
| 12    | <code>CSR_MISA_M_EXT</code>           | r/- | M: CPU extension (mul/div) available, set when <b>M ISA Extension</b> enabled                |
| 20    | <code>CSR_MISA_U_EXT</code>           | r/- | U: CPU extension (user mode) available, set when <b>U ISA Extension</b> enabled              |
| 23    | <code>CSR_MISA_X_EXT</code>           | r/- | X: bit is always set to indicate non-standard / NEORV32-specific extensions                  |
| 31:30 | <code>CSR_MISA_MXL_HI_EXT</code><br>: | r/- | MXL: 32-bit architecture indicator (always <b>01</b> )                                       |
|       | <code>CSR_MISA_MXL_LO_EXT</code>      |     |  |



Machine-mode software can discover available **Z\*** *sub-extensions* (like `Zicsr` or `Zfinx`) by checking the NEORV32-specific `mxisa` CSR.

## `mie`

|             |  |
|-------------|--|
| Name        | Machine interrupt-enable register  |
| Address     | <code>0x304</code>   |
| Reset value | <code>0x00000000</code>  |
| ISA         | <code>Zicsr</code>   |
| Description | The <code>mie</code> CSR is used to enable/disable individual interrupt sources. |
| on          |  |

Table 63. `mie` CSR bits

| Bit   | Name [C]  | R/W | Function   |
|-------|---|-----|--|
| 3     | <code>CSR_MIE_MSIE</code>                                     | r/w | <b>MSIE</b> : Machine <i>software</i> interrupt enable   |
| 7     | <code>CSR_MIE_MTIE</code>                                     | r/w | <b>MTIE</b> : Machine <i>timer</i> interrupt enable (from <a href="#">Machine System Timer (MTIME)</a> ) |
| 11    | <code>CSR_MIE_MEIE</code>                                     | r/w | <b>MEIE</b> : Machine <i>external</i> interrupt enable   |
| 31:16 | <code>CSR_MIE_FIRQ15E</code> :<br><code>CSR_MIE_FIRQ0E</code> | r/w | Fast interrupt channel 15..0 enable  |

**mtvec**

|             |   |
|-------------|---|
| Name        | Machine trap-handler base address   |
| Address     | <code>0x305</code>  |
| Reset value | <a href="#">CPU_BOOT_ADDR</a> , CPU boot address, 4-byte aligned (see <a href="#">CPU Top Entity - Generics and Address Space</a> ) |
| ISA         | <a href="#">Zicsr</a>   |
| Description | The <code>mtvec</code> CSR holds the trap vector configuration.   |
| on          |   |

Table 64. `mtvec` CSR bits

| Bit  | R/W | Function  |
|------|-----|---|
| 1:0  | r/w | <b>MODE</b> : mode configuration, <b>00</b> = DIRECT, <b>01</b> = VECTORED. (Others will fall back to DIRECT mode.)   |
| 31:2 | r/w | <b>BASE</b> : in DIRECT mode = 4-byte aligned base address of trap base handler, <i>all</i> traps set <code>pc</code> = <b>BASE</b> ; in VECTORED mode = 128-byte aligned base address of trap vector table, interrupts cause a jump to <code>pc</code> = <b>BASE</b> + 4 * <code>mcause</code> and exceptions to <code>pc</code> = <b>BASE</b> . |

*Interrupt Latency*

The vectored `mtvec` mode is useful for reducing the time between interrupt request (IRQ) and servicing it (ISR). As software does not need to determine the interrupt cause the reduction in latency can be 5 to 10 times and as low as 26 cycles.

**mcounteren**

|         |                        |
|---------|------------------------|
| Name    | Machine counter enable |
| Address | <code>0x306</code>     |

Reset **0x00000000**  
value

ISA **Zicsr & U**

Description The **mcounteren** CSR is used to constrain user-mode access to the CPU's counter CSRs.  
on

Table 65. **mcounteren** CSR bits

| Bit  | R/W     | Function   |
|------|---------|--|
| 0    | r/w (!) | <b>CY</b> : User-mode is allowed to read <b>cycle[h]</b> CSRs when set       |
| 1    | r/-     | <b>TM</b> : not implemented, hardwired to zero                               |
| 2    | r/w (!) | <b>IR</b> : User-mode is allowed to read <b>instret[h]</b> CSRs when set     |
| 15:3 | r/w (!) | <b>HPM</b> : user-mode is allowed to read <b>hpmcounter[h]</b> CSRs when set |



Physically, the NEORV32's **mcounteren** CSR is implemented as a **single 1-bit register**. Setting *any* bit of the CSR will result in all bits being set. Hence, user-mode access can either be granted for **all** counter CSRs or entirely denied allowing access to **none** counter CSRs.

## **mstatush**

Name Machine status register - high word

Address **0x310**

Reset **0x00000000**  
value

ISA **Zicsr**

Description The features of this CSR are not implemented yet. The register is read-only and always returns zero.

### 3.8.3. Machine Trap Handling CSRs

#### `mscratch`

|             |  |
|-------------|--|
| Name        | Scratch register for machine trap handlers |
| Address     | <code>0x340</code>                         |
| Reset value | <code>0x00000000</code>                    |
| ISA         | <code>Zicsr</code>                         |

Description The `mscratch` is a general-purpose machine-mode scratch register.

#### `mepc`

|             |  |
|-------------|--|
| Name        | Machine exception program counter  |
| Address     | <code>0x341</code>   |
| Reset value | <code>CPU_BOOT_ADDR</code> , CPU boot address, 4-byte aligned (see <a href="#">CPU Top Entity - Generics</a> and <a href="#">Address Space</a> )   |
| ISA         | <code>Zicsr</code>   |
| Description | The <code>mepc</code> CSR provides the instruction address where execution has stopped/failed when an instruction is triggered / an exception is raised. See section <a href="#">Traps, Exceptions and Interrupts</a> for a list of all legal values. The <code>mret</code> instruction will return to the address stored in <code>mepc</code> by automatically moving <code>mepc</code> to the program counter. |



`mepc[0]` is hardwired to zero. If IALIGN = 32 (i.e. `C ISA Extension` is disabled) then `mepc[1]` is also hardwired to zero.

#### `mcause`

|             |                         |
|-------------|-------------------------|
| Name        | Machine trap cause      |
| Address     | <code>0x342</code>      |
| Reset value | <code>0x00000000</code> |
| ISA         | <code>Zicsr</code>      |

Description The `mcause` CSRs shows the exact cause of a trap. See section [Traps, Exceptions and Interrupts](#) for a list of all legal values.

Table 66. `mcause` CSR bits

| Bit | R/W | Function  |
|-----|-----|---|
| 4:0 | r/w | <b>Exception code:</b> see <a href="#">NEORV32 Trap Listing</a>                                     |
| 31  | r/w | <b>Interrupt:</b> 1 if the trap is caused by an interrupt (0 if the trap is caused by an exception) |

## mtval

Name Machine trap value

Address [0x343](#)

Reset [0x00000000](#)  
value

ISA [Zicsr](#)

Description The [mtval](#) CSR provides additional information why a trap was entered. See section on [Traps, Exceptions and Interrupts](#) for more information.

*Read-Only*



Note that the NEORV32 [mtval](#) CSR is updated by the hardware only and cannot be written from software. However, any write-access will be ignored and will not cause an exception to maintain RISC-V compatibility.

## mip

Name Machine interrupt pending

Address [0x344](#)

Reset [0x00000000](#)  
value

ISA [Zicsr](#)

Description The [mip](#) CSR shows currently *pending* machine-mode interrupt requests. Any write access to this register is ignored.

Table 67. [mip](#) CSR bits

| Bit | Name [C]                     | R/W | Function   |
|-----|------------------------------|-----|--|
| 3   | <a href="#">CSR_MIP_MSIP</a> | r/- | <b>MSIP:</b> Machine <i>software</i> interrupt pending; <i>cleared by platform-defined mechanism</i> |
| 7   | <a href="#">CSR_MIP_MTIP</a> | r/- | <b>MTIP:</b> Machine <i>timer</i> interrupt pending; <i>cleared by platform-defined mechanism</i>    |

| Bit   | Name [C]  | R/W | Function  |
|-------|---|-----|---|
| 11    | <code>CSR_MIP_MEIP</code>                                     | r/- | <b>MEIP</b> : Machine <i>external</i> interrupt pending; <i>cleared by platform-defined mechanism</i> |
| 31:16 | <code>CSR_MIP_FIRQ15P</code> :<br><code>CSR_MIP_FIRQ0P</code> | r/- | <b>FIRQxP</b> : Fast interrupt channel 15..0 pending; <i>cleared by platform-defined mechanism</i>    |



#### *FIRQ Channel Mapping*

See section [NEORV32-Specific Fast Interrupt Requests](#) for the mapping of the FIRQ channels and the according interrupt-triggering processor module.

## `mtinst`

Name Machine trap instruction

Address `0x34a`

Reset `0x00000000`  
value

ISA `Zicsr`

Description The `mtinst` CSR provides additional information why a trap was entered. See section on [Traps, Exceptions and Interrupts](#) for more information.

#### *Read-Only*



Note that the NEORV32 `mtinst` CSR is updated by the hardware only and cannot be written from software. However, any write-access will be ignored and will not cause an exception to maintain RISC-V compatibility.

#### *Instruction Transformation*



The RISC-V priv. spec. suggests that the instruction word written to `mtinst` by the hardware should be "transformed". However, the NEORV32 `mtinst` CSR uses a simplified transformation scheme: if the trap-causing instruction is a standard 32-bit instruction, `mtinst` contains the exact instruction word that caused the trap. If the trap-causing instruction is a compressed instruction, `mtinst` contains the de-compressed 32-bit equivalent with bit 1 being cleared.

### 3.8.4. Machine Configuration CSRs

#### menvcfg

|             |   |
|-------------|---|
| Name        | Machine environment configuration register - low word   |
| Address     | <code>0x30a</code>  |
| Reset value | <code>0x00000000</code>   |
| ISA         | <code>Zicsr &amp; U</code>  |
| Description | Currently, the features of this CSR are not supported. Hence, the entire register is hardwired to all-zero. |

#### menvcfg

|             |   |
|-------------|---|
| Name        | Machine environment configuration register - high word  |
| Address     | <code>0x31a</code>  |
| Reset value | <code>0x00000000</code>   |
| ISA         | <code>Zicsr &amp; U</code>  |
| Description | Currently, the features of this CSR are not supported. Hence, the entire register is hardwired to all-zero. |

### 3.8.5. Machine Physical Memory Protection CSRs

The physical memory protection system is configured via the `PMP_NUM_REGIONS` and `PMP_MIN_GRANULARITY` top entity generics. `PMP_NUM_REGIONS` defines the total number of implemented regions. Note that the maximum number of regions is constrained to 16. If trying to access a PMP-related CSR beyond `PMP_NUM_REGIONS` **no illegal instruction exception** is triggered. The according CSRs are read-only (writes are ignored) and always return zero. See section [Smpmp ISA Extension](#) for more information.

#### pmpcfg

|             |  |
|-------------|--|
| Name        | PMP region configuration registers   |
| Address     | <code>0x3a0</code> ( <code>pmpcfg0</code> )<br><code>0x3a1</code> ( <code>pmpcfg1</code> )<br><code>0x3a2</code> ( <code>pmpcfg2</code> )<br><code>0x3a3</code> ( <code>pmpcfg3</code> ) |
| Reset value | <code>0x00000000</code>  |
| ISA         | <a href="#">Zicsr</a> & <a href="#">PMP</a>  |
| Description | Configuration of physical memory protection regions. Each region provides an individual 8-bit array in these CSRs.   |

Table 68. `pmpcfg0` CSR Bits

| Bit | Name [C]                 | R/W | Function  |
|-----|--------------------------|-----|---|
| 0   | <code>PMPCFG_R</code>    | r/w | R: Read permission  |
| 1   | <code>PMPCFG_W</code>    | r/w | W: Write permission   |
| 2   | <code>PMPCFG_X</code>    | r/w | X: Execute permission   |
| 4:3 | <code>PMPCFG_A_MS</code> | r/w | A: Mode configuration (00 = OFF, 01 = TOR, 10 = NA4, 11 = NAPOT)<br>B :<br><code>PMPCFG_A_LS</code><br>B                    |
| 7   | <code>PMPCFG_L</code>    | r/w | L: Lock bit, prevents further write accesses, also enforces access rights in machine-mode, can only be cleared by CPU reset |

#### Implemented Modes



In order to reduce the CPU size certain PMP modes (A bits) can be excluded from synthesis. Use the `PMP_TOR_MODE_EN` and `PMP_NAP_MODE_EN` [Processor Top Entity - Generics](#) to control implementation of the according modes.

## pmpaddr

The `pmpaddr*` CSRs are used to configure the region's address boundaries.

|             |  |
|-------------|--|
| Name        | Physical memory protection address registers   |
| Address     | <code>0x3b0 (pmpaddr1)</code><br><code>0x3b1 (pmpaddr2)</code><br><code>0x3b2 (pmpaddr3)</code><br><code>0x3b3 (pmpaddr4)</code><br><code>0x3b4 (pmpaddr5)</code><br><code>0x3b5 (pmpaddr6)</code><br><code>0x3b6 (pmpaddr6)</code><br><code>0x3b7 (pmpaddr7)</code><br><code>0x3b8 (pmpaddr8)</code><br><code>0x3b9 (pmpaddr9)</code><br><code>0x3ba (pmpaddr10)</code><br><code>0x3bb (pmpaddr11)</code><br><code>0x3bc (pmpaddr12)</code><br><code>0x3bd (pmpaddr13)</code><br><code>0x3be (pmpaddr14)</code><br><code>0x3bf (pmpaddr15)</code> |
| Reset value | <code>0x00000000</code>  |
| ISA         | <code>Zicsr</code> & <code>PMP</code>  |
| Description | Region address configuration. The two MSBs of each CSR are hardwired to zero (= bits 33:32 of the physical address).   |

### 3.8.6. Custom Functions Unit (CFU) CSRs

#### cfureg

Name      Custom (user-defined) CFU CSRs

Address    `0x800 (cfureg0)`

`0x801 (cfureg1)`

`0x802 (cfureg2)`

`0x803 (cfureg3)`

Reset     `0x00000000`

value

ISA        `Zicsr & Zxcfu`

Description User-defined CSRs to be used within the [Custom Functions Unit \(CFU\)](#).  
on

### 3.8.7. (Machine) Counter and Timer CSRs

#### `time[h]` CSRs (Wall Clock Time)



The NEORV32 does not implement the user-mode `time[h]` registers. Any access to these registers will trap. It is recommended that the trap handler software provides a means of accessing the platform-defined **Machine System Timer (MTIME)**.



#### *Instruction Retired Counter Increment*

The `[m]instret[h]` counter always increments when a instruction enters the pipeline's execute stage no matter if this instruction is actually going to retire or if it causes an exception.

#### `cycle[h]`

|             |   |
|-------------|---|
| Name        | Cycle counter   |
| Address     | <code>0xc00 (cycle)</code><br><code>0xc80 (cycleh)</code>   |
| Reset value | <code>0x00000000</code>   |
| ISA         | <code>Zicsr &amp; Zicntr</code>   |
| Description | The <code>cycle[h]</code> CSRs are user-mode shadow copies of the according <code>mcycle[h]</code> CSRs. The user-mode counter are read-only. Any write access will raise an illegal instruction exception. |

#### `instret[h]`

|             |   |
|-------------|---|
| Name        | Instructions-retired counter  |
| Address     | <code>0xc02 (instret)</code><br><code>0xc82 (instreth)</code>   |
| Reset value | <code>0x00000000</code>   |
| ISA         | <code>Zicsr &amp; Zicntr</code>   |
| Description | The <code>instret[h]</code> CSRs are user-mode shadow copies of the according <code>minstret[h]</code> CSRs. The user-mode counter are read-only. Any write access will raise an illegal instruction exception. |

## **mcycle[h]**

|             |  |
|-------------|--|
| Name        | Machine cycle counter  |
| Address     | <code>0xb00</code> ( <code>mcycle</code> )<br><code>0xb80</code> ( <code>mcycleh</code> )  |
| Reset value | <code>0x00000000</code>  |
| ISA         | <code>Zicsr</code> & <code>Zicntr</code>   |
| Description | If not halted via the <code>mcountinhibit</code> CSR the <code>cycle[h]</code> CSRs will increment with every active CPU clock cycle (CPU not in sleep mode). These registers are read/write only for machine-mode software. |

## **minstret[h]**

|             |   |
|-------------|---|
| Name        | Machine instructions-retired counter  |
| Address     | <code>0xb02</code> ( <code>minstret</code> )<br><code>0xb82</code> ( <code>minstreth</code> )   |
| Reset value | <code>0x00000000</code>   |
| ISA         | <code>Zicsr</code> & <code>Zicntr</code>  |
| Description | If not halted via the <code>mcountinhibit</code> CSR the <code>minstret[h]</code> CSRs will increment with every retired instruction. These registers are read/write only for machine-mode software |



### *Instruction Retiring*

Note that **all** executed instruction do increment the `[m]instret[h]` counters even if they do not retire (e.g. if the instruction causes an exception).

### 3.8.8. Hardware Performance Monitors (HPM) CSRs

The actual number of implemented hardware performance monitors is configured via the `HPM_NUM_CNTS` top entity generic. Note that always all 13 HPM counter and configuration registers (`mhpmcnter*[h]` and `mhpmevent*`) are implemented, but only the actually configured ones are implemented as "real" physical registers - the remaining ones will be hardwired to zero.

If trying to access an HPM-related CSR beyond `HPM_NUM_CNTS` **no illegal instruction exception is triggered**. These CSRs are read-only (writes are ignored) and always return zero.

The total counter width of the HPMs can be configured before synthesis via the `HPM_CNT_WIDTH` generic (0..64-bit). If `HPM_NUM_CNTS` is less than 64, all remaining MSB-aligned bits are hardwired to zero.

#### `mhpmevent`

|             |  |
|-------------|--|
| Name        | Machine hardware performance monitor event select  |
| Address     | <code>0x233</code> ( <code>mhpmevent3</code> )<br><code>0x234</code> ( <code>mhpmevent4</code> )<br><code>0x235</code> ( <code>mhpmevent5</code> )<br><code>0x236</code> ( <code>mhpmevent6</code> )<br><code>0x237</code> ( <code>mhpmevent7</code> )<br><code>0x238</code> ( <code>mhpmevent8</code> )<br><code>0x239</code> ( <code>mhpmevent9</code> )<br><code>0x23a</code> ( <code>mhpmevent10</code> )<br><code>0x23b</code> ( <code>mhpmevent11</code> )<br><code>0x23c</code> ( <code>mhpmevent12</code> )<br><code>0x23d</code> ( <code>mhpmevent13</code> )<br><code>0x23e</code> ( <code>mhpmevent14</code> )<br><code>0x23f</code> ( <code>mhpmevent15</code> ) |
| Reset value | <code>0x00000000</code>  |
| ISA         | <code>Zicsr</code> & <code>Zihpm</code>  |
| Description | <p>The value in these CSRs define the architectural events that cause an increment of the according <code>mhpmcnter*[h]</code> counter(s). All available events are listed in the table below.</p> <p>If more than one event is selected, the according counter will increment if <i>any</i> of the enabled events is observed (logical OR). Note that the counter will only increment by 1 step per clock cycle even if more than one trigger event is observed.</p>  |

Table 69. `mhpmevent*` CSR Bits

| Bit                      | Name [C]               | R/W | Event Description   |
|--------------------------|------------------------|-----|---|
| <b>RISC-V-compatible</b> |                        |     |   |
| 0                        | HPMCNT_EVENT_CY        | r/w | active clock cycle (CPU not in <a href="#">Sleep Mode</a> )   |
| 1                        | HPMCNT_EVENT_TM        | r/- | <i>not implemented</i> , hardwired to zero  |
| 2                        | HPMCNT_EVENT_IR        | r/w | any executed instruction (16-bit/compressed or 32-bit/uncompressed)   |
| <b>NEORV32-specific</b>  |                        |     |   |
| 3                        | HPMCNT_EVENT_COMPR     | r/w | any executed 16-bit/compressed ( <a href="#">C ISA Extension</a> ) instruction  |
| 4                        | HPMCNT_EVENT_WAIT_IS   | r/w | instruction dispatch wait cycle (wait for instruction prefetch-buffer refill ( <a href="#">CPU Control Unit</a> IPB); caused by a fence instruction, a control flow transfer or a instruction fetch bus wait cycle) |
| 5                        | HPMCNT_EVENT_WAIT_LU   | r/w | any delay/wait cycle caused by a <i>multi-cycle CPU Arithmetic Logic Unit</i> operation   |
| 6                        | HPMCNT_EVENT_BRANCH    | r/w | any executed branch instruction (unconditional, conditional-taken or conditional-not-taken)   |
| 7                        | HPMCNT_EVENT_BRANCH_ED | r/w | any control transfer operation (unconditional jump, taken conditional branch or trap entry/exit)  |
| 8                        | HPMCNT_EVENT_LOAD      | r/w | any executed load operation (including atomic memory operations, <a href="#">A ISA Extension</a> )  |
| 9                        | HPMCNT_EVENT_STORE     | r/w | any executed store operation (including atomic memory operations, <a href="#">A ISA Extension</a> )   |
| 10                       | HPMCNT_EVENT_WAIT_SU   | r/w | any memory/bus/cache/etc. delay/wait cycle while executing any load or store operation (caused by a data bus wait cycle))   |
| 11                       | HPMCNT_EVENT_TRAP      | r/w | starting processing of any trap ( <a href="#">Traps, Exceptions and Interrupts</a> )  |

*Instruction Retiring ("Retired == Executed")*



The CPU HPM/counter logic treats all executed instruction as "retired" even if they raise an exception, cause an interrupt, trigger a privilege mode change or were not meant to retire (by the RISC-V spec.).

## mhpmcOUNTER[h]

|         |  |
|---------|--|
| Name    | Machine hardware performance monitor (HPM) counter                     |
| Address | <code>0xb03, 0xb83</code> ( <code>mhpmcOUNTER3, mhpmcOUNTER3h</code> ) |
|         | <code>0xb04, 0xb84</code> ( <code>mhpmcOUNTER4, mhpmcOUNTER4h</code> ) |

|              |   |
|--------------|---|
| 0xb05, 0xb85 | ( <a href="#">mhpmcOUNTER5</a> , <a href="#">mhpmcOUNTER5h</a> )  |
| 0xb06, 0xb86 | ( <a href="#">mhpmcOUNTER6</a> , <a href="#">mhpmcOUNTER6h</a> )  |
| 0xb07, 0xb87 | ( <a href="#">mhpmcOUNTER7</a> , <a href="#">mhpmcOUNTER7h</a> )  |
| 0xb08, 0xb88 | ( <a href="#">mhpmcOUNTER8</a> , <a href="#">mhpmcOUNTER8h</a> )  |
| 0xb09, 0xb89 | ( <a href="#">mhpmcOUNTER9</a> , <a href="#">mhpmcOUNTER9h</a> )  |
| 0xb0a, 0xb8a | ( <a href="#">mhpmcOUNTER10</a> , <a href="#">mhpmcOUNTER10h</a> )  |
| 0xb0b, 0xb8b | ( <a href="#">mhpmcOUNTER11</a> , <a href="#">mhpmcOUNTER11h</a> )  |
| 0xb0c, 0xb8c | ( <a href="#">mhpmcOUNTER12</a> , <a href="#">mhpmcOUNTER12h</a> )  |
| 0xb0d, 0xb8d | ( <a href="#">mhpmcOUNTER13</a> , <a href="#">mhpmcOUNTER13h</a> )  |
| 0xb0e, 0xb8e | ( <a href="#">mhpmcOUNTER14</a> , <a href="#">mhpmcOUNTER14h</a> )  |
| 0xb0f, 0xb8f | ( <a href="#">mhpmcOUNTER15</a> , <a href="#">mhpmcOUNTER15h</a> )  |
| Reset value  | 0x00000000  |
| ISA          | <a href="#">Zicsr</a> & <a href="#">Zihpm</a>   |
| Description  | If not halted via the <a href="#">mCOUNTINHIBIT</a> CSR the HPM counter CSR(s) increment whenever a configured event from the according <a href="#">mHPMEVENT</a> CSR occurs. The counter registers are read/write for machine mode and are not accessible for lower-privileged software. |

## hpmcounter[h]

|         |   |
|---------|---|
| Name    | User hardware performance monitor (HPM) counter   |
| Address | 0xc03, 0xc83 ( <a href="#">hpmcounter3</a> , <a href="#">hpmcounter3h</a> )<br>0xc04, 0xc84 ( <a href="#">hpmcounter4</a> , <a href="#">hpmcounter4h</a> )<br>0xc05, 0xc85 ( <a href="#">hpmcounter5</a> , <a href="#">hpmcounter5h</a> )<br>0xc06, 0xc86 ( <a href="#">hpmcounter6</a> , <a href="#">hpmcounter6h</a> )<br>0xc07, 0xc87 ( <a href="#">hpmcounter7</a> , <a href="#">hpmcounter7h</a> )<br>0xc08, 0xc88 ( <a href="#">hpmcounter8</a> , <a href="#">hpmcounter8h</a> )<br>0xc09, 0xc89 ( <a href="#">hpmcounter9</a> , <a href="#">hpmcounter9h</a> )<br>0xc0a, 0xc8a ( <a href="#">hpmcounter10</a> , <a href="#">hpmcounter10h</a> )<br>0xc0b, 0xc8b ( <a href="#">hpmcounter11</a> , <a href="#">hpmcounter11h</a> )<br>0xc0c, 0xc8c ( <a href="#">hpmcounter12</a> , <a href="#">hpmcounter12h</a> )<br>0xc0d, 0xc8d ( <a href="#">hpmcounter13</a> , <a href="#">hpmcounter13h</a> )<br>0xc0e, 0xc8e ( <a href="#">hpmcounter14</a> , <a href="#">hpmcounter14h</a> )<br>0xc0f, 0xc8f ( <a href="#">hpmcounter15</a> , <a href="#">hpmcounter15h</a> ) |

Reset      `0x00000000`  
value

ISA      `Zicsr & Zihpm`

Description The `hpmcounter*[h]` are user-mode shadow copies of the according `mhpmcOUNTER[h]` CSRs.  
on      The user mode counter CSRs are read-only. Any write access will raise an illegal  
instruction exception.

---

### 3.8.9. Machine Counter Setup CSRs

#### mcountinhibit

|             |  |
|-------------|--|
| Name        | Machine counter-inhibit register                 |
| Address     | 0x320  |
| Reset value | 0x00000000                                       |
| ISA         | Zicsr  |
| Description | Set bit to halt the according counter CSR.<br>on |

Table 70. mcountinhibit CSR Bits

| Bit  | Name [C]  | R/W | Description   |
|------|---|-----|---|
| 0    | CSR_MCOUNTINHIBIT_I<br>R                                    | r/w | <b>IR:</b> Set to 1 to halt [ $m$ ]instret[h]; hardwired to zero if Zicntr ISA extension is disabled    |
| 1    | -   | r/- | <b>TM:</b> Hardwired to zero as time[h] CSRs are not implemented  |
| 2    | CSR_MCOUNTINHIBIT_C<br>Y                                    | r/w | <b>CY:</b> Set to 1 to halt [ $m$ ]cycle[h]; hardwired to zero if Zicntr ISA extension is disabled      |
| 15:3 | CSR_MCOUNTINHIBIT_H<br>PM3 :<br>CSR_MCOUNTINHIBIT_H<br>PM15 | r/w | <b>HPMx:</b> Set to 1 to halt [ $m$ ]hpmcount*[h]; hardwired to zero if Zihpm ISA extension is disabled |

### 3.8.10. Machine Information CSRs

#### **mvendorid**

|             |   |
|-------------|---|
| Name        | Machine vendor ID   |
| Address     | <code>0xf11</code>  |
| Reset value | <code>DEFINED</code>  |
| ISA         | <code>Zicsr</code>  |
| Description | Vendor ID (JEDEC identifier, lowest 11 bits), assigned via the <code>JEDEC_ID</code> top generic on <a href="#">(Processor Top Entity - Generics)</a> . |

#### **marchid**

|             |   |
|-------------|---|
| Name        | Machine architecture ID   |
| Address     | <code>0xf12</code>  |
| Reset value | <code>0x00000013</code>   |
| ISA         | <code>Zicsr</code>  |
| Description | The <code>marchid</code> CSR is read-only and provides the NEORV32 official RISC-V open-source architecture ID (decimal: 19, 32-bit hexadecimal: 0x00000013). |

#### **mimpid**

|             |  |
|-------------|--|
| Name        | Machine implementation ID  |
| Address     | <code>0xf13</code>   |
| Reset value | <code>DEFINED</code>   |
| ISA         | <code>Zicsr</code>   |
| Description | The <code>mimpid</code> CSR is read-only and provides the version of the NEORV32 as BCD-coded number (example: <code>mimpid = 0x01020312</code> → 01.02.03.12 → version 1.2.3.12). |

#### **mhartid**

|         |                            |
|---------|----------------------------|
| Name    | Machine hardware thread ID |
| Address | <code>0xf14</code>         |

Reset **DEFINED**  
value

ISA **Zicsr**

Description The **mhartid** CSR is read-only and provides the core's hart ID, which is assigned via the **HW\_THREAD\_ID** top generic ([Processor Top Entity - Generics](#)).

## **mconfigptr**

Name Machine configuration pointer register

Address **0xf15**

Reset **0x00000000**  
value

ISA **Zicsr**

Description The features of this CSR are not implemented yet. The register is read-only and always returns zero.

### 3.8.11. NEORV32-Specific CSRs



All NEORV32-specific CSRs are mapped to addresses that are explicitly reserved for custom **Machine-Mode**, **read-only** CSRs (assured by the RISC-V privileged specifications). Hence, these CSRs can only be accessed when in machine-mode. Any access outside of machine-mode will raise an illegal instruction exception.

#### `mxisa`

|             |   |
|-------------|---|
| Name        | Machine extended isa and extensions register  |
| Address     | <code>0xfc0</code>  |
| Reset value | <code>DEFINED</code>  |
| ISA         | <code>Zicsr &amp; X</code>  |
| Description | The <code>mxisa</code> CSRs is a NEORV32-specific read-only CSR that helps machine-mode software to discover ISA sub-extensions and CPU configuration options |

Table 71. `mxisa` CSR Bits

| Bit   | Name [C]                        | R/W | Description   |
|-------|---------------------------------|-----|---|
| 0     | <code>CSR_MXISA_ZICSR</code>    | r/- | <code>Zicsr</code> ISA Extension available  |
| 1     | <code>CSR_MXISA_ZIFENCEI</code> | r/- | <code>Zifencei</code> ISA Extension available   |
| 2     | <code>CSR_MXISA_ZMMUL</code>    | r/- | <code>Zmmul</code> - ISA Extension available  |
| 3     | <code>CSR_MXISA_ZXCFU</code>    | r/- | <code>Zxcfу</code> ISA Extension available  |
| 4     | -                               | r/- | <i>reserved</i> , hardwired to zero   |
| 5     | <code>CSR_MXISA_ZFINX</code>    | r/- | <code>Zfinx</code> ISA Extension available  |
| 6     | <code>CSR_MXISA_ZICOND</code>   | r/- | <code>Zicond</code> ISA Extension available   |
| 7     | <code>CSR_MXISA_ZICNTR</code>   | r/- | <code>Zicntr</code> ISA Extension available   |
| 8     | <code>CSR_MXISA_SMPMP</code>    | r/- | <code>Smpmp</code> ISA Extension available  |
| 9     | <code>CSR_MXISA_ZIHPM</code>    | r/- | <code>Zihpm</code> ISA Extension available  |
| 10    | <code>CSR_MXISA_SDEXT</code>    | r/- | <code>Sdext</code> ISA Extension available  |
| 11    | <code>CSR_MXISA_SDTRIG</code>   | r/- | <code>Sdtrig</code> ISA Extension available   |
| 19:12 | -                               | r/- | hardwired to zero   |
| 20    | <code>CSR_MXISA_IS_SIM</code>   | r/- | set if CPU is being <b>simulated</b> (not guaranteed)                                   |
| 28:21 | -                               | r/- | hardwired to zero   |
| 29    | <code>CSR_MXISA_RFHWRST</code>  | r/- | full hardware reset of register file available when set ( <code>REGFILE_HW_RST</code> ) |

| Bit | Name [C]            | R/W | Description   |
|-----|---------------------|-----|---|
| 30  | CSR_MXISA_FASTMUL   | r/- | fast multiplication available when set<br>( <a href="#">FAST_MUL_EN</a> ) |
| 31  | CSR_MXISA_FASTSHIFT | r/- | fast shifts available when set ( <a href="#">FAST_SHIFT_EN</a> )          |

### 3.8.12. Traps, Exceptions and Interrupts

In this document the following terminology is used (derived from the RISC-V trace specification available at <https://github.com/riscv-non-isa/riscv-trace-spec>):

- **exception**: an unusual condition occurring at run time associated (i.e. *synchronous*) with an instruction in a RISC-V hart
- **interrupt**: an external *asynchronous* event that may cause a RISC-V hart to experience an unexpected transfer of control
- **trap**: the transfer of control to a trap handler caused by either an *exception* or an *interrupt*

Whenever an exception or interrupt is triggered, the CPU switches to machine-mode (if not already in machine-mode) and continues operation at the address being stored in the **mtvec** CSR. The cause of the trap can be determined via the **mcause** CSR. A list of all implemented **mcause** values and the according description can be found below in section [NEORV32 Trap Listing](#). The address that reflects the current program counter when a trap was taken is stored to **mepc** CSR. Additional information regarding the cause of the trap can be retrieved from the **mtval** and **mtinst** CSRs.

The traps are prioritized. If several *exceptions* occur at once only the one with highest priority is triggered while all remaining exceptions are ignored and discarded. If several *interrupts* trigger at once, the one with highest priority is serviced first while the remaining ones stay *pending*. After completing the interrupt handler the interrupt with the second highest priority will get serviced and so on until no further interrupts are pending.

#### *Interrupts when in User-Mode*



If the core is currently operating in less privileged user-mode, interrupts are globally enabled even if **mstatus.mie** is cleared.

#### *Interrupt Signal Requirements - Standard RISC-V Interrupts*



All interrupt request signals are **high-active**. Once triggered, a interrupt request line should stay high until it is explicitly acknowledged by a source-specific mechanism (for example by writing to a specific memory-mapped register).

#### *Instruction Atomicity and Forward-Progress*



All instructions execute as atomic operations - interrupts can only trigger *between* consecutive instructions. Additionally, if there is a permanent interrupt request, exactly one instruction from the interrupted program will be executed before another interrupt handler can start. This allows program progress even if there are permanent interrupt requests.

### Memory Access Exceptions

If a load operation causes any exception, the instruction's destination register is **not written** at all. Furthermore, exceptions caused by a misaligned memory address a physical memory protection fault do not trigger a memory access request at all.

For 32-bit-only instructions (= no `C` extension) the misaligned instruction exception is raised if bit 1 of the fetch address is set (i.e. not on a 32-bit boundary). If the `C` extension is implemented there will **never** be a misaligned instruction exception at all.

## Custom Fast Interrupt Request Lines

As a custom extension, the NEORV32 CPU features 16 fast interrupt request (FIRQ) lines via the `firq_i` CPU top entity signals. These interrupts have custom configuration and status flags in the `mie` and `mip` CSRs and also provide custom trap codes in `mcause`. These FIRQs are reserved for NEORV32 processor-internal usage only.

## NEORV32 Trap Listing

The following tables show all traps that are currently supported by the NEORV32 CPU. It also shows the prioritization and the CSR side-effects.

### Table Annotations

The "Prio." column shows the priority of each trap with the highest priority being 1. The "RTE Trap ID" aliases are defined by the NEORV32 core library (the runtime environment *RTE*) and can be used in plain C code when interacting with the pre-defined RTE function. The `mcause`, `mepc`, `mtval` and `mtinst` columns show the value being written to the according CSRs when a trap is triggered:

- **I-PC** - address of intercepted instruction (instruction has *not* been executed yet)
- **PC** - address of instruction that caused the trap (instruction has been executed)
- **ADR** - bad data memory access address that caused the trap
- **INS** - the transformed/decompressed instruction word that caused the trap
- **0** - zero

Table 72. NEORV32 Trap Listing

| Pr   | <code>mcause</code>     | RTE Trap ID                         | Cause                                | <code>mepc</code> | <code>mtval</code> | <code>mtinst</code> |
|--|-------------------------|-------------------------------------|--------------------------------------|-------------------|--------------------|---------------------|
| io   |                         |                                     |                                      |                   |                    |                     |
| .  |                         |                                     |                                      |                   |                    |                     |
| <b>Exceptions (synchronous to instruction execution)</b> |                         |                                     |                                      |                   |                    |                     |
| 1  | <code>0x00000001</code> | <code>TRAP_CODE_I_ACCESS</code>     | instruction access fault             | I-PC              | 0                  | INS                 |
| 2  | <code>0x00000002</code> | <code>TRAP_CODE_I_ILLEGAL</code>    | illegal instruction                  | PC                | 0                  | INS                 |
| 3  | <code>0x00000000</code> | <code>TRAP_CODE_I_MISALIGNED</code> | instruction address misaligned       | PC                | 0                  | INS                 |
| 4  | <code>0x0000000b</code> | <code>TRAP_CODE_MENV_CALL</code>    | environment call from M-mode         | PC                | 0                  | INS                 |
| 5  | <code>0x00000008</code> | <code>TRAP_CODE_UENV_CALL</code>    | environment call from U-mode         | PC                | 0                  | INS                 |
| 6  | <code>0x00000003</code> | <code>TRAP_CODE_BREAKPOINT</code>   | software breakpoint / trigger firing | PC                | 0                  | INS                 |
| 7  | <code>0x00000006</code> | <code>TRAP_CODE_S_MISALIGNED</code> | store address misaligned             | PC                | ADR                | INS                 |

| Pr<br>io  | mcause     | RTE Trap ID            | Cause                               | mepc | mtval | mtins<br>t |
|---|------------|------------------------|-------------------------------------|------|-------|------------|
| .   |            |                        |                                     |      |       |            |
| 8   | 0x00000004 | TRAP_CODE_L_MISALIGNED | load address misaligned             | PC   | ADR   | INS        |
| 9   | 0x00000007 | TRAP_CODE_S_ACCESS     | store access fault                  | PC   | ADR   | INS        |
| 10  | 0x00000005 | TRAP_CODE_L_ACCESS     | load access fault                   | PC   | ADR   | INS        |
| <b>Interrupts (asynchronous to instruction execution)</b> |            |                        |                                     |      |       |            |
| 11  | 0x80000010 | TRAP_CODE_FIRQ_0       | fast interrupt request channel 0    | I-PC | 0     | 0          |
| 12  | 0x80000011 | TRAP_CODE_FIRQ_1       | fast interrupt request channel 1    | I-PC | 0     | 0          |
| 13  | 0x80000012 | TRAP_CODE_FIRQ_2       | fast interrupt request channel 2    | I-PC | 0     | 0          |
| 14  | 0x80000013 | TRAP_CODE_FIRQ_3       | fast interrupt request channel 3    | I-PC | 0     | 0          |
| 15  | 0x80000014 | TRAP_CODE_FIRQ_4       | fast interrupt request channel 4    | I-PC | 0     | 0          |
| 16  | 0x80000015 | TRAP_CODE_FIRQ_5       | fast interrupt request channel 5    | I-PC | 0     | 0          |
| 17  | 0x80000016 | TRAP_CODE_FIRQ_6       | fast interrupt request channel 6    | I-PC | 0     | 0          |
| 18  | 0x80000017 | TRAP_CODE_FIRQ_7       | fast interrupt request channel 7    | I-PC | 0     | 0          |
| 19  | 0x80000018 | TRAP_CODE_FIRQ_8       | fast interrupt request channel 8    | I-PC | 0     | 0          |
| 20  | 0x80000019 | TRAP_CODE_FIRQ_9       | fast interrupt request channel 9    | I-PC | 0     | 0          |
| 21  | 0x8000001a | TRAP_CODE_FIRQ_10      | fast interrupt request channel 10   | I-PC | 0     | 0          |
| 22  | 0x8000001b | TRAP_CODE_FIRQ_11      | fast interrupt request channel 11   | I-PC | 0     | 0          |
| 23  | 0x8000001c | TRAP_CODE_FIRQ_12      | fast interrupt request channel 12   | I-PC | 0     | 0          |
| 24  | 0x8000001d | TRAP_CODE_FIRQ_13      | fast interrupt request channel 13   | I-PC | 0     | 0          |
| 25  | 0x8000001e | TRAP_CODE_FIRQ_14      | fast interrupt request channel 14   | I-PC | 0     | 0          |
| 26  | 0x8000001f | TRAP_CODE_FIRQ_15      | fast interrupt request channel 15   | I-PC | 0     | 0          |
| 27  | 0x8000000B | TRAP_CODE_MEI          | machine external interrupt<br>(MEI) | I-PC | 0     | 0          |
| 28  | 0x80000003 | TRAP_CODE_MSI          | machine software interrupt<br>(MSI) | I-PC | 0     | 0          |
| 29  | 0x80000007 | TRAP_CODE_MTI          | machine timer interrupt (MTI)       | I-PC | 0     | 0          |

Table 73. NEORV32 Trap Description

| Trap ID [C]         | Triggered when ...   |
|---------------------|--|
| TRAP_CODE_I_ACCESS  | bus timeout, bus access error or PMP rule violation during instruction fetch                           |
| TRAP_CODE_I_ILLEGAL | trying to execute an invalid instruction word (malformed or not supported) or on a privilege violation |

| Trap ID [C]            | Triggered when ...   |
|------------------------|--|
| TRAP_CODE_I_MISALIGNED | fetching a 32-bit instruction word that is not 32-bit-aligned (see note below)   |
| TRAP_CODE_MENV_CALL    | executing <code>ecall</code> instruction in machine-mode   |
| TRAP_CODE_UENV_CALL    | executing <code>ecall</code> instruction in user-mode  |
| TRAP_CODE_BREAKPOINT   | executing <code>ebreak</code> instruction or if <a href="#">Trigger Module</a> fires   |
| TRAP_CODE_S_MISALIGNED | storing data to an address that is not naturally aligned to the data size (half/word)  |
| TRAP_CODE_L_MISALIGNED | loading data from an address that is not naturally aligned to the data size (half/word)  |
| TRAP_CODE_S_ACCESS     | bus timeout, bus access error or <a href="#">PMP</a> rule violation during load data operation   |
| TRAP_CODE_L_ACCESS     | bus timeout, bus access error or <a href="#">PMP</a> rule violation during store data operation  |
| TRAP_CODE_FIRQ_*       | caused by interrupt-condition of <b>processor-internal modules</b> , see <a href="#">NEORV32-Specific Fast Interrupt Requests</a>                |
| TRAP_CODE_MEI          | machine external interrupt (via dedicated <a href="#">Processor Top Entity - Signals</a> )   |
| TRAP_CODE_MSI          | machine software interrupt (via dedicated <a href="#">Processor Top Entity - Signals</a> )   |
| TRAP_CODE_MTI          | machine timer interrupt (internal <a href="#">Machine System Timer (MTIME)</a> or via dedicated <a href="#">Processor Top Entity - Signals</a> ) |

#### *Resumable Exceptions*



Note that not all exceptions are resumable. For example, the "instruction access fault" exception or the "instruction address misaligned" exception are not resumable in most cases. These exception might indicate a fatal memory hardware failure.

# Chapter 4. Software Framework

The NEORV32 project comes with a complete software ecosystem called the "software framework", which is based on the C-language RISC-V GCC port and consists of the following parts:

- Compiler Toolchain
- Core Libraries
- Application Makefile
- Executable Image Format
  - Linker Script
  - RAM Layout
  - C Standard Library
  - Start-Up Code (crt0)
- Bootloader
- NEORV32 Runtime Environment

A summarizing list of the most important elements of the software framework and their according files and folders is shown below:

|   |   |
|---|---|
| Application start-up code               | <code>sw/common/crt0.S</code>                     |
| Application linker script               | <code>sw/common/nerv32.ld</code>                  |
| Core hardware driver libraries ("HAL")  | <code>sw/lib/include/ &amp; sw/lib/source/</code> |
| Central application makefile            | <code>sw/common/common.mk</code>                  |
| Tool for generating NEORV32 executables | <code>sw/image_gen/</code>                        |
| Default bootloader                      | <code>sw/bootloader</code>                        |
| Example programs                        | <code>sw/example</code>                           |

## *Software Documentation*



All core libraries and example programs are documented "in-code" using **Doxygen**. The documentation is automatically built and deployed to GitHub pages and is available online at <https://stnolting.github.io/nerv32/sw/files.html>.

## *Example Programs*



A collection of annotated example programs, which show how to use certain CPU functions and peripheral/IO modules, can be found in `sw/example`.

## 4.1. Compiler Toolchain

The toolchain for this project is based on the free and open RISC-V GCC-port. You can find the compiler sources and build instructions on the official RISC-V GNU toolchain GitHub page: <https://github.com/riscv/riscv-gnutoolchain>.

The NEORV32 implements a 32-bit RISC-V architecture and uses a 32-bit integer and soft-float ABI by default. Make sure the toolchain / toolchain build is configured accordingly.

- `MARCH=rv32i`
- `MABI=ilp32`
- `RISCV_PREFIX=riscv32-unknown-elf-`

These default configurations can be overridden at any times using [Application Makefile](#) variables.



More information regarding the toolchain (building from scratch or downloading prebuilt ones) can be found in the user guide section [Software Toolchain Setup](#).

## 4.2. Core Libraries

The NEORV32 project provides a set of pre-defined C libraries that allow an easy integration of the processor/CPU features (also called "HAL" - hardware abstraction layer). All driver and runtime-related files are located in `sw/lib`. These library files are automatically included and linked by adding the following include statement:

```
#include <neorv32.h> // NEORV32 HAL, core and runtime libraries
```

*Table 74. NEORV32 HAL File List*

| C source file     | C header file        | Description  |
|-------------------|----------------------|--|
| -                 | neorv32.h            | Main NEORV32 library file  |
| neorv32_cfs.c     | neorv32_cfs.h        | Custom Functions Subsystem (CFS) HAL                                       |
| neorv32_crc.c     | neorv32_crc.h        | Cyclic Redundancy Check (CRC) HAL  |
| neorv32_cpu.c     | neorv32_cpu.h        | NEORV32 Central Processing Unit (CPU) HAL                                  |
| neorv32_cpu_amo.c | neorv32_cpu_amo.h    | Emulation functions for the read-modify-write A ISA Extension instructions |
|                   | neorv32_cpu_csr.h    | Control and Status Registers (CSRs) definitions                            |
| neorv32_cpu_cfu.c | neorv32_cpu_cfu.h    | Custom Functions Unit (CFU) HAL  |
| -                 | neorv32_dm.h         | Debug Module (DM) HAL  |
| neorv32_dma.c     | neorv32_dma.h        | Direct Memory Access Controller (DMA) HAL                                  |
| neorv32_gpio.c    | neorv32_gpio.h       | General Purpose Input and Output Port (GPIO) HAL                           |
| neorv32_gptmr.c   | neorv32_gptmr.h      | General Purpose Timer (GPTMR) HAL  |
| -                 | neorv32_intrinsics.h | Macros for intrinsics & custom instructions                                |
| neorv32_mtime.c   | neorv32_mtime.h      | Machine System Timer (MTIME) HAL   |
| neorv32_neoled.c  | neorv32_neoled.h     | Smart LED Interface (NEOLED) HAL   |
| neorv32_onewire.c | neorv32_onewire.h    | One-Wire Serial Interface Controller (ONEWIRE) HAL                         |
| neorv32_pwm.c     | neorv32_pwm.h        | Pulse-Width Modulation Controller (PWM) HAL                                |
| neorv32_rte.c     | neorv32_rte.h        | NEORV32 Runtime Environment  |
| neorv32_sdi.c     | neorv32_sdi.h        | Serial Data Interface Controller (SDI) HAL                                 |
| neorv32_slink.c   | neorv32_slink.h      | Stream Link Interface (SLINK) HAL  |
| neorv32_spi.c     | neorv32_spi.h        | Serial Peripheral Interface Controller (SPI) HAL                           |
| -                 | neorv32_sysinfo.h    | System Configuration Information Memory (SYSINFO) HAL                      |

| C source file  | C header file  | Description   |
|----------------|----------------|---|
| neorv32_trng.c | neorv32_trng.h | True Random-Number Generator (TRNG) HAL                                       |
| neorv32_twi.c  | neorv32_twi.h  | Two-Wire Serial Interface Controller (TWI) HAL                                |
| neorv32_uart.c | neorv32_uart.h | Primary Universal Asynchronous Receiver and Transmitter (UART0) and UART1 HAL |
| neorv32_wdt.c  | neorv32_wdt.h  | Watchdog Timer (WDT) HAL  |
| neorv32_xip.c  | neorv32_xip.h  | Execute In Place Module (XIP) HAL   |
| neorv32_xirq.c | neorv32_xirq.h | External Interrupt Controller (XIRQ) HAL                                      |
| syscalls.c     | -              | Newlib "system calls" (stubs)   |
| -              | legacy.h       | Backwards compatibility wrappers and functions (do not use for new designs)   |

*Core Library Documentation*



The *doxygen*-based documentation of the software framework including all core libraries is available online at <https://stnolting.github.io/neorv32/sw/files.html>.

*CMSIS System View Description File (SVD)*



A CMSIS-SVD-compatible **System View Description (SVD)** file including all peripherals is available in `sw/svd`. Together with a third-party plugin the processor's SVD file can be imported right into GDB to allow comfortable debugging of peripheral/IO devices (see <https://github.com/stnolting/neorv32/discussions/656>).

## 4.3. Application Makefile

Application compilation is based on a single, centralized GNU makefile ([sw/common/common.mk](#)). Each project in the [sw/example](#) folder provides a makefile that just *includes* this central makefile.



When creating a new project, copy an existing project folder or at least the makefile to the new project folder. It is recommended to create new projects also in [sw/example](#) to keep the file dependencies. However, these dependencies can be manually configured via makefile variables if the new project is located somewhere else.



Before the makefile can be used to compile applications, the RISC-V GCC toolchain needs to be installed and the compiler's `bin` folder has to be added to the system's `PATH` environment variable. More information can be found in [User Guide: Software Toolchain Setup](#).

### 4.3.1. Makefile Targets

Just executing `make` (or executing `make help`) will show the help menu listing all available targets.

```
$ make
NEORV32 Software Application Makefile
Find more information at https://github.com/stnolting/neorv32

Targets:
help      - show this text
check     - check toolchain
info      - show makefile/toolchain configuration
gdb       - run GNU debugging session
asm        - compile and generate <main.asm> assembly listing file for manual
debugging
elf        - compile and generate <main.elf> ELF file
bin        - compile and generate <neorv32_raw_exe.bin> RAW executable file (binary
file, no header)
hex        - compile and generate <neorv32_raw_exe.hex> RAW executable file (hex char
file, no header)
image      - compile and generate VHDL IMEM boot image (for application, no header)
in local folder
install    - compile, generate and install VHDL IMEM boot image (for application, no
header)
sim        - in-console simulation using default/simple testbench and GHDL
all        - exe + install + hex + bin + asm
elf_info   - show ELF layout info
clean      - clean up project home folder
clean_all  - clean up whole project, core libraries and image generator
bl_image   - compile and generate VHDL BOOTROM boot image (for bootloader only, no
header) in local folder
```

bootloader - compile, generate and install VHDL BOOTROM boot image (for bootloader only, no header)

#### Variables:

|                |   |
|----------------|---|
| USER_FLAGS     | - Custom toolchain flags [append only]: ""                                |
| USER_LIBS      | - Custom libraries [append only]: ""                                      |
| EFFORT         | - Optimization level: "-Os"   |
| MARCH          | - Machine architecture: "rv32i_zicsr_zifencei"                            |
| MABI           | - Machine binary interface: "ilp32"                                       |
| APP_INC        | - C include folder(s) [append only]: "-I ."                               |
| ASM_INC        | - ASM include folder(s) [append only]: "-I ."                             |
| RISCV_PREFIX   | - Toolchain prefix: "riscv32-unknown-elf-"                                |
| NEORV32_HOME   | - NEORV32 home folder: "../../.."   |
| GDB_ARGS       | - GDB (connection) arguments: "-ex target extended-remote localhost:3333" |
| GHDL_RUN_FLAGS | - GHDL simulation run arguments: ""                                       |

### 4.3.2. Makefile Configuration

The compilation flow is configured via variables right at the beginning of the central makefile ([sw/common/common.mk](#)):

#### *Customizing Makefile Variables*



The makefile configuration variables can be overridden or extended directly when invoking the makefile. For example `$ make MARCH=rv32ic_zicsr_zifencei clean_all exe` overrides the default `MARCH` variable definitions.

*Listing 12. Default Makefile Configuration*

```
# ****
# USER CONFIGURATION
# ****
# User's application sources (*.c, *.cpp, *.s, *.S); add additional files here
APP_SRC ?= $(wildcard ./*.c) $(wildcard ./*.s) $(wildcard ./*.cpp) $(wildcard ./*.S)
# User's application include folders (don't forget the '-I' before each entry)
APP_INC ?= -I .
# User's application include folders - for assembly files only (don't forget the '-I' before each entry)
ASM_INC ?= -I .
# Optimization
EFFORT ?= -Os
# Compiler toolchain
RISCV_PREFIX ?= riscv32-unknown-elf-
# CPU architecture and ABI
MARCH ?= rv32i_zicsr_zifencei
MABI ?= ilp32
```

```
# User flags for additional configuration (will be added to compiler flags)
USER_FLAGS ?=
# User libraries (will be included by linker)
USER_LIBS ?=
# Language specific compiler flags: C
CFLAGS ?=
# C++
CXXFLAGS ?=
# Assembly
ASFLAGS ?=
# Flags passed only to the linker
LDFLAGS ?=
# Relative or absolute path to the NEORV32 home folder
NEORV32_HOME ?= ../../..
# GDB arguments
GDB_ARGS ?= -ex "target extended-remote localhost:3333"
# ****
```

Table 75. Variables Description

|                       |  |
|-----------------------|--|
| <b>APP_SRC</b>        | The source files of the application ( <b>.c</b> , <b>.cpp</b> , <b>.S</b> and <b>.s</b> files are allowed; files of these types in the project folder are automatically added via wild cards). Additional files can be added separated by white spaces |
| <b>APP_INC</b>        | Include file folders; separated by white spaces; must be defined with <b>-I</b> prefix   |
| <b>ASM_INC</b>        | Include file folders that are used only for the assembly source files ( <b>.S/.s</b> ).  |
| <b>EFFORT</b>         | Optimization level, optimize for size ( <b>-Os</b> ) is default; legal values: <b>-O0</b> , <b>-O1</b> , <b>-O2</b> , <b>-O3</b> , <b>-Os</b> , <b>-Ofast</b> , ...  |
| <b>RISCV_PREFIX</b>   | The toolchain prefix to be used; follows the triplet naming convention <b>[architecture]-[host_system]-[output]-...</b>  |
| <b>MARCH</b>          | The targeted RISC-V architecture/ISA   |
| <b>MABI</b>           | Application binary interface (default: 32-bit integer ABI <b>ilp32</b> )   |
| <b>USER_FLAGS</b>     | Additional flags that will be forwarded to the compiler tools  |
| <b>USER_LIBS</b>      | Additional libraries to include during linking ( <b>*.a</b> )  |
| <b>CFLAGS</b>         | Additional flags that will be forwarded to the C compiler  |
| <b>CXXFLAGS</b>       | Additional flags that will be forwarded to the C++ compiler  |
| <b>ASFLAGS</b>        | Additional flags that will be forwarded to the assembler   |
| <b>LDFLAGS</b>        | Additional flags that will be forwarded to the linker  |
| <b>NEORV32_HOME</b>   | Relative or absolute path to the NEORV32 project home folder; adapt this if the makefile/project is not in the project's default <b>sw/example</b> folder  |
| <b>GDB_ARGS</b>       | Default GDB arguments when running the <b>gdb</b> target   |
| <b>GHDL_RUN_FLAGS</b> | GHDL run arguments (e.g. <b>--stop-time=1ms</b> )  |

### 4.3.3. Default Compiler Flags

The following default compiler flags are used for compiling an application. These flags are defined via the `CC_OPTS` variable.



The makefile's `CC_OPTS` is exported as `define` to be available within a C program; for example `neorv32_uart0_printf("%s\n", CC_OPTS);`.

|                                  |  |
|----------------------------------|--|
| <code>-Wall</code>               | Enable all compiler warnings.  |
| <code>-ffunction-sections</code> | Put functions and data segment in independent sections. This allows a code optimization as dead code and unused data can be easily removed.                |
| <code>-nostartfiles</code>       | Do not use the default start code. Instead, the NEORV32-specific start-up code ( <code>sw/common/crt0.S</code> ) is used (pulled-in by the linker script). |
| <code>-Wl,--gc-sections</code>   | Make the linker perform dead code elimination.   |
| <code>-lm</code>                 | Include/link with <code>math.h</code> .  |
| <code>-lc</code>                 | Search for the standard C library when linking.  |
| <code>-lgcc</code>               | Make sure we have no unresolved references to internal GCC library subroutines.  |
| <code>-mno-fdiv</code>           | Use built-in software functions for floating-point divisions and square roots (since the according instructions are not supported yet).                    |
| <code>-g</code>                  | Include debugging information/symbols in ELF.  |
| <code>-mstrict-align</code>      | Unaligned memory accesses cannot be resolved by the hardware and require emulation.  |
| <code>-mbbranch-cost=10</code>   | Branching costs a lot of cycles.   |

### 4.3.4. Custom (Compiler) Flags

Custom flags can be *appended* to the `USER_FLAGS` variable. This allows to customize the entire software framework while calling `make` without the need to change the makefile(s) or the linker script. The following example will add debug symbols to the executable (`-g`) and will also re-define the linker script's `_neorv32_heap_size` variable setting the maximal heap size to 4096 bytes (see sections [Linker Script](#) and [RAM Layout](#)):

*Listing 13. Using the `USER_FLAGS` Variable for Customization*

```
$ make USER_FLAGS+=" -g -Wl,--_neorv32_heap_size,_heap_size=4096" clean_all exe
```

The configuration can also be made "permanent" by adapting the application's makefile (make sure to use the `override` command here):

*Listing 14. Using the `USER_FLAGS` Variable for Permanent Customization*

```
override USER_FLAGS += "-g -Wl,--__neorv32_heap_size,__heap_size=4096"
```

## 4.4. Executable Image Format

In order to generate an executable for the processors all source files have to be compiled, linked and packed into a final executable.

### 4.4.1. Linker Script

After all the application sources have been compiled, they need to be *linked*. For this purpose the makefile uses the NEORV32-specific linker script. The linker script defines several sections for the final executable (compare with [Address Space](#)). However, only the `ram` and `rom` sections are relevant for the executable itself; the raining sections are just listed for completeness.

*Table 76. Linker script - memory sections*

| Memory section    | Description   |
|-------------------|---|
| <code>ram</code>  | Data memory address space (processor-internal <a href="#">Instruction Memory (IMEM)</a> and/or external memory) |
| <code>rom</code>  | Instruction memory address space (processor-internal <a href="#">Data Memory (DMEM)</a> and/or external memory) |
| <code>xip</code>  | Address space for the <a href="#">Execute In Place Module (XIP)</a> (accessing an external SPI flash)           |
| <code>boot</code> | Address space for the processor-internal <a href="#">Bootloader ROM (BOOTROM)</a>                               |
| <code>io</code>   | Address space for the processor-internal IO/peripheral devices  |



The `rom` section is automatically re-mapped to the processor-internal [Bootloader ROM \(BOOTROM\)](#) when compiling the bootloader sources.

Each section has two main attributes: `ORIGIN` and `LENGTH`. `ORIGIN` defines the base address of the according section while `LENGTH` defines its size in bytes. For the `ram` and `rom` sections these attributes are configured indirectly via variables that provide default values.

*Listing 15. Linker script - section configuration*

```
/* Default rom/ram (IMEM/DMEM) sizes */
__neorv32_rom_size = DEFINED(__neorv32_rom_size) ? __neorv32_rom_size : 2048M;
__neorv32_ram_size = DEFINED(__neorv32_ram_size) ? __neorv32_ram_size : 8K;

/* Default section base addresses */
__neorv32_rom_base = DEFINED(__neorv32_rom_base) ? __neorv32_rom_base : 0x00000000;
__neorv32_ram_base = DEFINED(__neorv32_ram_base) ? __neorv32_ram_base : 0x80000000;
```

The region size and base address configuration can be edited by the user - either by explicitly changing the default values in the linker script or by overriding them when invoking `make`:

Listing 16. Overriding default `rom` size configuration (configuring 4096 bytes)

```
$ make USER_FLAGS+=“-Wl,--defsym,__neorv32_rom_size=4096” clean_all exe
```



`neorv32_rom_base` (= `ORIGIN` of the `ram` section) and `neorv32_ram_base` (= `ORIGIN` of the `rom` section) have to be sync to the actual memory layout configuration of the processor (see section [Address Space](#)).



The default configuration for the `rom` section assumes a maximum of 2GB *logical* memory address space. This size does not have to reflect the *actual* physical size of the entire instruction memory. It just provides a maximum limit. When uploading a new executable via the bootloader, the bootloader itself checks if sufficient *physical* instruction memory is available. If a new executable is embedded right into the internal-IMEM the synthesis tool will check, if the configured instruction memory size is sufficient.

The linker maps all the regions from the compiled object files into five final sections: `.text`, `.rodata`, `.data`, `.bss` and `.heap`:

Table 77. Linker script - memory regions

| Region               | Description   |
|----------------------|---|
| <code>.text</code>   | Executable instructions generated from the start-up code and all application sources.           |
| <code>.rodata</code> | Constants (like strings) from the application; also the initial data for initialized variables. |
| <code>.data</code>   | This section is required for the address generation of fixed (= global) variables only.         |
| <code>.bss</code>    | This section is required for the address generation of dynamic memory constructs only.          |
| <code>.heap</code>   | This section is required for the address generation of dynamic memory constructs only.          |

The `.text` and `.rodata` sections are mapped to processor's instruction memory space and the `.data`, `.bss` and `heap` sections are mapped to the processor's data memory space. Finally, the `.text`, `.rodata` and `.data` sections are extracted and concatenated into a single file `main.bin`.

#### Section Alignment



The default NEORV32 linker script aligns *all* regions so they start and end on a 32-bit (word) boundaries. The default NEORV32 start-up code (`crt0`) makes use of this alignment by using word-level memory instructions to initialize the `.data` section and to clear the `.bss` section (faster!).

## 4.4.2. RAM Layout

The default NEORV32 linker script uses all of the defined RAM (linker script memory section `ram`) to several sections. Note that depending on the application some sections might have zero size.

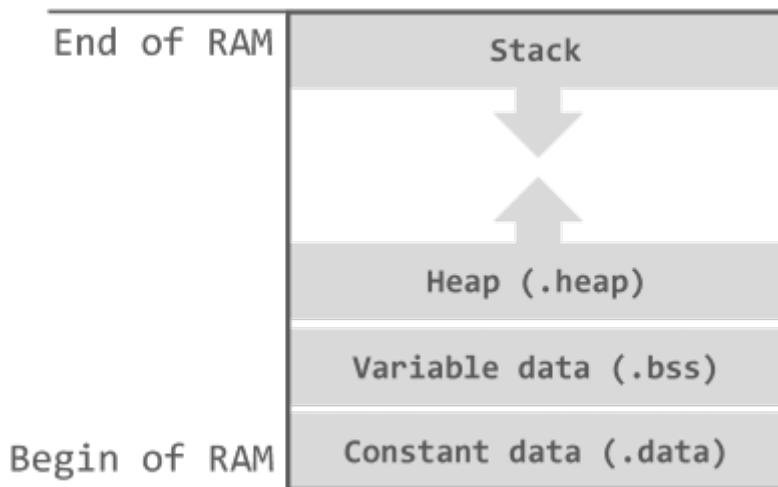


Figure 15. Default RAM Layout

- Constant data (.data):** The constant data section is placed right at the beginning of the RAM. For example, this section contains *explicitly initialized* global variables. This section is initialized by the executable.
- Dynamic data (.bss):** The constant data section is followed by the dynamic data section, which contains *uninitialized* data like global variables without explicit initialization. This section is cleared by the start-up code `crt0.S`.
- Heap (.heap):** The heap is used for dynamic memory that is managed by functions like `malloc()` and `free()`. The heap grows upwards. This section is not initialized at all.
- Stack:** The stack starts at the very end of the RAM at address `ORIGIN(ram) + LENGTH(ram) - 4`. The stack grows downwards.

There is *no explicit limit* for the maximum stack size as this is hard to check. However, a physical memory protection rule could be used to configure a maximum size by adding a "protection area" between stack and heap (a PMP region without any access rights).

#### Heap Size

The maximum size of the heap is defined by the linker script's `neorv32_heap_size` variable. This variable has to be explicitly defined in order to define a heap size (and to use dynamic memory allocation at all) other than zero. The user can define the heap size while invoking the application makefile: `$USER_FLAGS+=-Wl,--defsym,neorv32_heap_size=4k` make clean\_all exe (defines a heap size of 4\*1024 bytes).



#### Heap-Stack Collisions

Take care when using dynamic memory to avoid collision of the heap and stack memory areas. There is no compile-time protection mechanism available as the actual heap and stack size are defined by *runtime* data. Also beware of fragmentation when using dynamic memory allocation.



### 4.4.3. C Standard Library

The default software framework relies on **newlib** as default C standard library.

#### *RTOS Support*



The NEORV32 CPU and processor **do support** embedded RTOS like FreeRTOS and Zephyr. See the User guide section [Zephyr RTOS Support](#) and [FreeRTOS Support](#) for more information.

- + The FreeRTOS port and demo is available in a separate repository: <https://github.com/stnolting/neorv32-freertos>

Newlib provides stubs for common "system calls" (like file handling and standard input/output) that are used by other C libraries like **stdio**. These stubs are available in [sw/source/source/syscalls.c](#) and were adapted for the NEORV32 processor.

#### *Standard Consoles*



The **UART0** is used to implement all the standard input, output and error consoles (**STDIN**, **STDOUT** and **STDERR**).

#### *Constructors and Destructors*



Constructors and destructors for plain C code or for C++ applications are supported by the software framework. See [sw/example/hello\\_cpp](#) for a minimal example.

#### *Newlib Test/Demo Program*



A simple test and demo program, which uses some of newlib's core functions (like **malloc/free** and **read/write**) is available in [sw/example/demo\\_newlib](#)

### 4.4.4. Executable Image Generator

The **main.bin** file is packed by the NEORV32 image generator ([sw/image\\_gen](#)) to generate the final executable file. The image generator can generate several types of executables selected by a flag when calling the generator:

|                 |   |
|-----------------|---|
| <b>-app_bin</b> | Generates an executable binary file <b>neorv32_exe.bin</b> (including header) for UART uploading via the bootloader.  |
| <b>-app_img</b> | Generates an executable VHDL memory initialization image (no header) for the processor-internal IMEM. This option generates the <b>rtl/core/neorv32_application_image.vhd</b> file. |
| <b>-raw_hex</b> | Generates a plain ASCII hex-char file <b>neorv32_raw_exe.hex</b> (no header) for custom purpose.  |

|                       |  |
|-----------------------|--|
| <code>-raw_bin</code> | Generates a plain binary file <code>neorv32_raw_exe.bin</code> (no header) for custom purpose.   |
| <code>-bld_img</code> | Generates an executable VHDL memory initialization image (no header) for the processor-internal BOOT ROM. This option generates the <code>rtl/core/neorv32_bootloader_image.vhd</code> file. |

All these options are managed by the makefile. The normal application compilation flow will generate the `neorv32_exe.bin` executable designated for uploading via the default NEORV32 bootloader.

#### *Image Generator Compilation*



The sources of the image generator are automatically compiled when invoking the makefile (requiring a native GCC installation).

#### *Executable Header*



The image generator adds a small header to the `neorv32_exe.bin` executable, which consists of three 32-bit words located right at the beginning of the file. The first word of the executable is the signature word and is always `0x4788cafe`. Based on this word the bootloader can identify a valid image file. The next word represents the size in bytes of the actual program image in bytes. A simple "complement" checksum of the actual program image is given by the third word. This provides a simple protection against data transmission or storage errors. **Note that this executable format cannot be used for direct execution (e.g. via XIP or direct memory access).**

#### 4.4.5. Start-Up Code (`crt0`)

The CPU and also the processor require a minimal start-up and initialization code to bring the CPU (and the SoC) into a stable and initialized state and to initialize the C runtime environment before the actual application can be executed. This start-up code is located in `sw/common/crt0.S` and is automatically linked every application program and placed right before the actual application code so it gets executed right after reset.

The `crt0.S` start-up performs the following operations:

1. Clear `mstatus`.
2. Clear `mie` disabling all interrupt sources.
3. Install an **Early Trap Handler** to `mtvec`.
4. Initialize the global pointer `gp` and the stack pointer `sp` according to the **RAM Layout** provided by the linker script.
5. Initialize all integer register `x1 - x31` (only `x1 - x15` if the **E** CPU extension is enabled).
6. Setup `.data` section to configure initialized variables.
7. Clear the `.bss` section.

8. Call all *constructors* (if there are any).
9. Call the application's `main` function (with no arguments: `argc = argv = 0`).
10. If `main` returns:
  - All interrupt sources are disabled by clearing `mie`.
  - The return value of `main` is copied to the `mscratch` CSR to allow inspection by the debugger.
  - Call all *destructors* (if there are any).
  - The CPU enters sleep mode executing the `wfi` instruction in an endless loop.

#### *Bootloader Start-Up Code*



The bootloader uses the same start-up code as any "usual" application. However, certain parts are omitted when compiling `crt0` for the bootloader (like calling constructors and destructors). See the `crt0` source code for more information.

### Early Trap Handler

The start-up code provides a very basic trap handler for the early boot stage. This handler does nothing but trying to move on to the next linear instruction whenever an interrupt or synchronous exception is encountered.

This simple trap handler does not interact with the stack at all as it just uses a single register that is backup-ed using the `mscratch` CSR. Furthermore, the information if the trap-causing instruction is compressed or uncompressed is **not** determined by loading the instruction from memory. Instead, the transformed instruction word is read from the `mtinst` CSRs. These two features allow the trap handler to execute with minimal latency and high robustness.



The early-trap handler should be replaced by a more capable / informative one as soon as the application software is started (for example by using the [NEORV32 Runtime Environment](#)).

## 4.5. Bootloader

### Pre-Built Bootloader Image



This section refers to the **default** NEORV32 bootloader. A pre-compiled memory image for the processor-internal **Bootloader ROM (BOOTROM)** is available in the project's `rtl` folder: `rtl/core/nerv32_bootloader_image.vhd`. This image is automatically inserted into the boot ROM when synthesizing the processor with the bootloader being enabled. **Note that the default bootloader image was compiled for a minimal rv32i + priv. ISA!**

The NEORV32 bootloader (`sw/bootloader/bootloader.c`) provides an optional built-in firmware that allows to upload new application executables at *any time* without the need to re-synthesize the FPGA's bitstream. A UART connection is used to provide a simple text-based user interface that allows to upload executables.

Furthermore, the bootloader provides options to store an executable to a processor-external SPI flash. An "auto boot" feature can optionally fetch this executable right after reset if there is no user interaction via UART. This allows to build processor setups with *non-volatile application storage* while maintaining the option to update the application software at any timer.

### 4.5.1. Bootloader SoC/CPU Requirements

The bootloader requires certain CPU and SoC extensions and modules to be enabled in order to operate correctly.

|                    |   |
|--------------------|---|
| <b>REQUIRED</b>    | The bootloader is implemented only if the <code>INT_BOOTLOADER_EN</code> top generic is <code>true</code> . This will automatically select the CPU's <b>Indirect Boot</b> boot configuration. |
| <b>REQUIRED</b>    | The bootloader requires the privileged architecture CPU extension ( <b>Zicsr ISA Extension</b> ) to be enabled.   |
| <b>REQUIRED</b>    | At least 512 bytes of data memory (processor-internal DMEM or processor-external DMEM) are required for the bootloader's stack and global variables.  |
| <b>RECOMMENDED</b> | For user interaction via the <b>Bootloader Console</b> (like uploading executables) the primary UART ( <b>Primary Universal Asynchronous Receiver and Transmitter (UART0)</b> ) is required.  |
| <b>RECOMMENDED</b> | The default bootloader uses bit 0 of the <b>General Purpose Input and Output Port (GPIO)</b> output port to drive a high-active "heart beat" status LED.                                      |
| <b>RECOMMENDED</b> | The <b>Machine System Timer (MTIME)</b> is used to control blinking of the status LED and also to automatically trigger the <b>Auto Boot Sequence</b> .                                       |
| <b>OPTIONAL</b>    | The SPI controller ( <b>Serial Peripheral Interface Controller (SPI)</b> ) is needed to store/load executable from external flash using the <b>Auto Boot Sequence</b> .                       |
| <b>OPTIONAL</b>    | The XIP controller ( <b>Execute In Place Module (XIP)</b> ) is needed to boot/execute code directly from a pre-programmed SPI flash.  |

## 4.5.2. Bootloader Flash Requirements

The bootloader can access an SPI-compatible flash via the processor's top entity SPI port. By default, the flash chip-select line is driven by `spi_csn_o(0)` and the SPI clock uses 1/8 of the processor's main clock as clock frequency. The SPI flash has to support single-byte read and write operations, 24-bit addresses and at least the following standard commands:

- `0x02`: Program page (write byte)
- `0x03`: Read data (byte)
- `0x04`: Write disable (for volatile status register)
- `0x05`: Read (first) status register
- `0x06`: Write enable (for volatile status register)
- `0xAB`: Wake-up from sleep mode (optional)
- `0xD8`: Block erase (64kB)

### *Custom Configuration*



Most properties (like chip select line, flash address width, SPI clock frequency, ...) of the default bootloader can be reconfigured without the need to change the source code. Custom configuration can be made using command line switches (defines) when recompiling the bootloader. See the User Guide [https://stnolting.github.io/neorv32/ug/#\\_customizing\\_the\\_internal\\_bootloader](https://stnolting.github.io/neorv32/ug/#_customizing_the_internal_bootloader) for more information.

## 4.5.3. Bootloader Console

To interact with the bootloader, connect the primary UART (UART0) signals (`uart0_txd_o` and `uart0_rxd_o`) of the processor's top entity via a serial port (-adapter) to your computer (hardware flow control is not used so the according interface signals can be ignored), configure your terminal program using the following settings and perform a reset of the processor.

Terminal console settings ([19200-8-N-1](#)):

- 19200 Baud
- 8 data bits
- no parity bit
- 1 stop bit
- newline on `\r\n` (carriage return, newline)
- no transfer protocol / control flow protocol - just raw bytes

### *Terminal Program*



Any terminal program that can connect to a serial port should work. However, make sure the program can transfer data in *raw* byte mode without any protocol

overhead (e.g. XMODEM). Some terminal programs struggle with transmitting files larger than 4kB (see <https://github.com/stnolting/neorv32/pull/215>). Try a different terminal program if uploading of a binary does not work.

The bootloader uses the LSB of the top entity's `gpio_o` output port as high-active status LED. All other output pins are set to low level and won't be altered. After reset, the status LED will start blinking at 2Hz and the following intro screen shows up:

```
<< NEORV32 Bootloader >>
```

```
BLDV: Mar 7 2023
HWV: 0x01080107
CLK: 0x05f5e100
MISA: 0x40901106
XISA: 0xc0000fab
SOC: 0xfffff402f
IMEM: 0x00008000
DMEM: 0x00002000
```

```
Autoboot in 8s. Press any key to abort.
```

The start-up screen gives some brief information about the bootloader and several system configuration parameters:

|             |  |
|-------------|--|
| <b>BLDV</b> | Bootloader version (built date).   |
| <b>HWV</b>  | Processor hardware version (the <code>mimpid</code> CSR); in BCD format; example: <code>0x01040606</code> = v1.4.6.6).                       |
| <b>CLK</b>  | Processor clock speed in Hz (via the <code>CLK</code> register from the <a href="#">System Configuration Information Memory (SYSINFO)</a> ). |
| <b>MISA</b> | RISC-V CPU extensions ( <code>misa</code> CSR).  |
| <b>XISA</b> | NEORV32-specific CPU extensions ( <code>mxisa</code> CSR).   |
| <b>SOC</b>  | Processor configuration (via the <code>SOC</code> register from the <a href="#">System Configuration Information Memory (SYSINFO)</a> ).     |
| <b>IMEM</b> | Internal IMEM size in byte (via the <code>MEM</code> register from the <a href="#">System Configuration Information Memory (SYSINFO)</a> ).  |
| <b>DMEM</b> | Internal DMEM size in byte (via the <code>MEM</code> register from the <a href="#">System Configuration Information Memory (SYSINFO)</a> ).  |

Now you have 8 seconds to press *any* key. Otherwise, the bootloader starts the [Auto Boot Sequence](#). When you press any key within the 8 seconds, the actual bootloader user console starts:

```
<< NEORV32 Bootloader >>
```

```
BLDV: Mar 7 2023
HWV: 0x01080107
CLK: 0x05f5e100
MISA: 0x40901106
XISA: 0xc0000fab
SOC: 0xfffff402f
IMEM: 0x00008000
DMEM: 0x00002000
```

Autoboot in 8s. Press any key to abort. ①  
Aborted.

Available CMDs:

```
h: Help
r: Restart
u: Upload
s: Store to flash
l: Load from flash
x: Boot from flash (XIP)
e: Execute
CMD:>
```

① Auto boot sequence aborted due to user console input.

The auto boot countdown is stopped and the bootloader's user console is ready to receive one of the following commands:

- **h**: Show the help text (again)
- **r**: Restart the bootloader and the auto-boot sequence
- **u**: Upload new program executable (`neorv32_exe.bin`) via UART into the instruction memory
- **s**: Store executable to SPI flash at `spi_csn_o(0)` (little-endian byte order)
- **l**: Load executable from SPI flash at `spi_csn_o(0)` (little-endian byte order)
- **x**: Boot program directly from flash via XIP (requires a pre-programmed image)
- **e**: Start the application, which is currently stored in the instruction memory (IMEM)

A new executable can be uploaded via UART by executing the **u** command. After that, the executable can be directly executed via the **e** command. To store the recently uploaded executable to an attached SPI flash press **s**. To directly load an executable from the SPI flash press **l**. The bootloader and the auto-boot sequence can be manually restarted via the **r** command.

#### *Executable Upload*



Make sure to upload the NEORV32 executable `neorv32_exe.bin`. Uploading any other file (like `main.bin`) will cause an `ERR_EXE` bootloader error (see [Bootloader Error Codes](#)).

*Booting via XIP*

The bootloader allows to execute an application right from flash using the **Execute In Place Module (XIP)** module. This requires a pre-programmed flash. The bootloader's "store" option can **not** be used to program an XIP image.

*SPI Flash Power Down Mode*

The bootloader will issue a "wake-up" command prior to using the SPI flash to ensure it is not in sleep mode / power-down mode (see <https://github.com/stnolting/neorv32/pull/552>).

*Default Configuration*

More information regarding the default SPI, GPIO, XIP, etc. configuration can be found in the User Guide section [https://stnolting.github.io/neorv32/ug/#\\_customizing\\_the\\_internal\\_bootloader](https://stnolting.github.io/neorv32/ug/#_customizing_the_internal_bootloader).

*SPI Flash Programming*

For detailed information on using an SPI flash for application storage see User Guide section **Programming an External SPI Flash via the Bootloader**.

#### 4.5.4. Auto Boot Sequence

When you reset the NEORV32 processor, the bootloader waits 8 seconds for a UART console input before it starts the automatic boot sequence. This sequence tries to fetch a valid boot image from the external SPI flash, connected to SPI chip select `spi_csn_o(0)`. If a valid boot image is found that can be successfully transferred into the instruction memory, it is automatically started. If no SPI flash is detected or if there is no valid boot image found, an error code will be shown.

#### 4.5.5. Bootloader Error Codes

If something goes wrong during bootloader operation an error code and a short message is shown. In this case the processor is halted, the bootloader status LED is permanently activated and the processor has to be reset manually.



In many cases the error source is just *temporary* (like some HF spike during an UART upload). Just try again.

|                 |   |
|-----------------|---|
| <b>ERR_EXE</b>  | If you try to transfer an invalid executable (via UART or from the external SPI flash), this error message shows up. There might be a transfer protocol configuration error in the terminal program or maybe just the wrong file was selected. Also, if no SPI flash was found during an auto-boot attempt, this message will be displayed. |
| <b>ERR_SIZE</b> | Your program is way too big for the internal processor's instructions memory. Increase the memory size or reduce your application code.   |

|                 |  |
|-----------------|--|
| <b>ERR_CHKS</b> | This indicates a checksum error. Something went wrong during the transfer of the program image (upload via UART or loading from the external SPI flash). If the error was caused by a UART upload, just try it again. When the error was generated during a flash access, the stored image might be corrupted. |
| <b>ERR_FLSH</b> | This error occurs if the attached SPI flash cannot be accessed. Make sure you have the right type of flash and that it is properly connected to the NEORV32 SPI port using chip select #0.   |
| <b>ERR_EXC</b>  | The bootloader encountered an unexpected exception during operation. This might be caused when it tries to access peripherals that were not implemented during synthesis. Example: executing commands <b>l</b> or <b>s</b> (SPI flash operations) without the SPI module being implemented.                    |



If an unexpected exception has been raised the bootloader prints hexadecimal debug information showing the **mcause**, **mepc** and **mtval** CSR values.

## 4.6. NEORV32 Runtime Environment

The NEORV32 software framework provides a minimal **runtime environment** (abbreviated "RTE") that takes care of a stable and *safe* execution environment by handling *all* traps (exceptions & interrupts). The RTE simplifies trap handling by wrapping the CPU's privileged architecture (i.e. trap-related CSRs) into a unified software API.

Once initialized, the RTE provides **Default RTE Trap Handlers** that catch all possible traps. These default handlers just output a message via UART to inform the user when a certain trap has been triggered. The default handlers can be overridden by the application code to install application-specific handler functions for each trap.

 Using the RTE is **optional but highly recommended**. The RTE provides a simple and comfortable way of delegating traps to application-specific handlers while making sure that all traps (even though they are not explicitly used by the application) are handled correctly. Performance-optimized applications or embedded operating systems may not use the RTE at all in order to increase response time.

### 4.6.1. RTE Operation

The RTE manages the trap-related CSRs of the CPU's privileged architecture (**Machine Trap Handling CSRs**). It initializes the **mtvec** CSR in DIRECT mode, which then provides the base entry point for *all* traps. The address stored to this register defines the address of the **first-level trap handler**, which is provided by the NEORV32 RTE. Whenever an exception or interrupt is triggered this first-level trap handler is executed.

The first-level handler performs a complete context save, analyzes the source of the trap and calls the according **second-level trap handler**, which takes care of the actual exception/interrupt handling. The RTE manages a private look-up table to store the addresses of the according second-level trap handlers.

After the initial RTE setup, each entry in the RTE's trap handler look-up table is initialized with a **Default RTE Trap Handlers**. These default handler do not execute any trap-related operations - they just output a message via the **primary UART (UART0)** to inform the user that a trap has occurred, which is not (yet) handled by the actual application. After sending this message, the RTE tries to continue executing the actual program by resolving the trap cause.

### 4.6.2. Using the RTE



All provided RTE functions can be called only from machine-mode code.

The NEORV32 is part of the default NEORV32 software framework. However, it has to explicitly enabled by calling the RTE's setup function:

*Listing 17. RTE Setup (Function Prototype)*

```
void neorv32_rte_setup(void);
```



The RTE should be enabled right at the beginning of the application's `main` function.



It is recommended to not use the `mscratch` CSR when using the RTE as this register is used to provide services for [Application Context Handling](#) (i.e. modifying the registers of application code that caused a trap).

As mentioned above, all traps will just trigger execution of the RTE's [Default RTE Trap Handlers](#) at first. To use application-specific handlers, which actually "handle" a trap, the default handlers can be overridden by installing user-defined ones:

*Listing 18. Installing an Application-Specific Trap Handler (Function Prototype)*

```
int neorv32_rte_handler_install(uint8_t id, void (*handler)(void));
```

The first argument `id` defines the "trap ID" (for example a certain interrupt request) that shall be handled by the user-defined handler. These IDs are defined in [sw/lib/include/neorv32\\_rte.h](#):

*Listing 19. RTE Trap Identifiers (cut-out)*

```
enum NEORV32_RTE_TRAP_enum {
    RTE_TRAP_I_MISALIGNED = 0, /*< Instruction address misaligned */
    RTE_TRAP_I_ACCESS = 1, /*< Instruction (bus) access fault */
    RTE_TRAP_I_ILLEGAL = 2, /*< Illegal instruction */
    RTE_TRAP_BREAKPOINT = 3, /*< Breakpoint (EBREAK instruction) */
    RTE_TRAP_L_MISALIGNED = 4, /*< Load address misaligned */
    RTE_TRAP_L_ACCESS = 5, /*< Load (bus) access fault */
    RTE_TRAP_S_MISALIGNED = 6, /*< Store address misaligned */
    RTE_TRAP_S_ACCESS = 7, /*< Store (bus) access fault */
    RTE_TRAP_UENV_CALL = 8, /*< Environment call from user mode (ECALL instruction)*/
}
    RTE_TRAP_MENV_CALL = 9, /*< Environment call from machine mode (ECALL instruction)*/
    RTE_TRAP_MSI = 10, /*< Machine software interrupt */
    RTE_TRAP_MTI = 11, /*< Machine timer interrupt */
    RTE_TRAP_MEI = 12, /*< Machine external interrupt */
    RTE_TRAP_FIRQ_0 = 13, /*< Fast interrupt channel 0 */
    RTE_TRAP_FIRQ_1 = 14, /*< Fast interrupt channel 1 */
    RTE_TRAP_FIRQ_2 = 15, /*< Fast interrupt channel 2 */
    RTE_TRAP_FIRQ_3 = 16, /*< Fast interrupt channel 3 */
    RTE_TRAP_FIRQ_4 = 17, /*< Fast interrupt channel 4 */
    RTE_TRAP_FIRQ_5 = 18, /*< Fast interrupt channel 5 */
    RTE_TRAP_FIRQ_6 = 19, /*< Fast interrupt channel 6 */
```

```
RTE_TRAP_FIRQ_7      = 20, /*< Fast interrupt channel 7 */
RTE_TRAP_FIRQ_8      = 21, /*< Fast interrupt channel 8 */
RTE_TRAP_FIRQ_9      = 22, /*< Fast interrupt channel 9 */
RTE_TRAP_FIRQ_10     = 23, /*< Fast interrupt channel 10 */
RTE_TRAP_FIRQ_11     = 24, /*< Fast interrupt channel 11 */
RTE_TRAP_FIRQ_12     = 25, /*< Fast interrupt channel 12 */
RTE_TRAP_FIRQ_13     = 26, /*< Fast interrupt channel 13 */
RTE_TRAP_FIRQ_14     = 27, /*< Fast interrupt channel 14 */
RTE_TRAP_FIRQ_15     = 28  /*< Fast interrupt channel 15 */
```

The second argument **\*handler** is the actual function that implements the user-defined trap handler. The custom handler functions need to have a specific format without any arguments and with no return value:

*Listing 20. Custom Trap Handler (Function Prototype)*

```
void custom_trap_handler_xyz(void) {
    // handle trap...
}
```

#### Custom Trap Handler Attributes



Do NOT use the **interrupt** attribute for the application trap handler functions! This will place a **mret** instruction to the end of it making it impossible to return to the first-level trap handler of the RTE core, which will cause stack corruption.

The following example shows how to install a custom handler (**custom\_mtime\_irq\_handler**) for handling the RISC-V machine timer (MTIME) interrupt:

*Listing 21. Installing a MTIME IRQ Handler*

```
neorv32_rte_handler_install(RTE_TRAP_MTI, custom_mtime_irq_handler);
```

User-defined trap handlers can also be un-installed. This will remove the users trap handler from the RTE core and will re-install the **Default RTE Trap Handlers** for the specific trap.

*Listing 22. Function Prototype: Installing an Application-Specific Trap Handler*

```
int neorv32_rte_handler_uninstall(uint8_t id);
```

The argument **id** defines the identifier of the according trap that shall be un-installed. The following example shows how to un-install the custom handler **custom\_mtime\_irq\_handler** from the RISC-V machine timer (MTIME) interrupt:

*Listing 23. Example: Removing the Custom MTIME IRQ Handler*

```
neorv32_rte_handler_uninstall(RTE_TRAP_MTI);
```



The current RTE configuration can be printed via UART0 via the `neorv32_rte_info` function.

#### 4.6.3. Default RTE Trap Handlers

The default RTE trap handlers are executed when a certain trap is triggered that is not (yet) handled by an application-defined trap handler. The default handler will output a message giving additional debug information via the **Primary Universal Asynchronous Receiver and Transmitter (UART0)** to inform the user and it will also try to resume normal program execution. Some exemplary RTE outputs are shown below.

##### *Continuing Execution*



In most cases the RTE can successfully continue operation - for example if it catches an **interrupt** request that is not handled by the actual application program. However, if the RTE catches an un-handled **trap** like a bus access fault exception continuing execution will most likely fail making the CPU crash. Some exceptions cannot be resolved by the default debug trap handlers and will halt the CPU (see example below).

*Listing 24. RTE Default Trap Handler Output Examples*

```
<NEORV32-RTE> [M] Illegal instruction @ PC=0x000002d6, MTINST=0x000000FF,  
MTVAL=0x00000000 </NEORV32-RTE> ①  
<NEORV32-RTE> [U] Illegal instruction @ PC=0x00000302, MTINST=0x00000000,  
MTVAL=0x00000000 </NEORV32-RTE> ②  
<NEORV32-RTE> [U] Load address misaligned @ PC=0x00000440, MTINST=0x01052603,  
MTVAL=0x80000101 </NEORV32-RTE> ③  
<NEORV32-RTE> [M] Fast IRQ 0x00000003 @ PC=0x00000820, MTINST=0x00000000,  
MTVAL=0x00000000 </NEORV32-RTE> ④  
<NEORV32-RTE> [M] Instruction access fault @ PC=0x90000000, MTINST=0x42078b63,  
MTVAL=0x00000000 !!FATAL EXCEPTION!! Halting CPU. </NEORV32-RTE>\n ⑤
```

- ① Illegal 32-bit instruction `MTINST=0x000000FF` at address `PC=0x000002d6` while the CPU was in machine-mode (`[M]`).
- ② Illegal 16-bit instruction `MTINST=0x00000000` at address `PC=0x00000302` while the CPU was in user-mode (`[U]`).
- ③ Misaligned load access at address `PC=0x00000440` caused by instruction `MTINST=0x01052603` (trying to load a full 32-bit word from address `MTVAL=0x80000101`) while the CPU was in machine-mode (`[U]`).
- ④ Fast interrupt request from channel 3 before executing instruction at address `PC=0x00000820`

while the CPU was in machine-mode ([M]).

- ⑤ Instruction bus access fault at address `PC=0x90000000` while executing instruction `MTINST=0x42078b63` - this is fatal for the default debug trap handler while the CPU was in machine-mode ([M]).

The specific message right at the beginning of the debug trap handler message corresponds to the trap code obtained from the `mcause` CSR (see [NEORV32 Trap Listing](#)). A full list of all messages and the according `mcause` trap codes is shown below.

*Table 78. RTE Default Trap Handler Messages and According mcause Values*

| Trap identifier                  | According <code>mcause</code> CSR value |
|----------------------------------|---|
| "Instruction address misaligned" | <code>0x00000000</code>                 |
| "Instruction access fault"       | <code>0x00000001</code>                 |
| "Illegal instruction"            | <code>0x00000002</code>                 |
| "Breakpoint"                     | <code>0x00000003</code>                 |
| "Load address misaligned"        | <code>0x00000004</code>                 |
| "Load access fault"              | <code>0x00000005</code>                 |
| "Store address misaligned"       | <code>0x00000006</code>                 |
| "Store access fault"             | <code>0x00000007</code>                 |
| "Environment call from U-mode"   | <code>0x00000008</code>                 |
| "Environment call from M-mode"   | <code>0x0000000b</code>                 |
| "Machine software IRQ"           | <code>0x80000003</code>                 |
| "Machine timer IRQ"              | <code>0x80000007</code>                 |
| "Machine external IRQ"           | <code>0x8000000b</code>                 |
| "Fast IRQ 0x00000000"            | <code>0x80000010</code>                 |
| "Fast IRQ 0x00000001"            | <code>0x80000011</code>                 |
| "Fast IRQ 0x00000002"            | <code>0x80000012</code>                 |
| "Fast IRQ 0x00000003"            | <code>0x80000013</code>                 |
| "Fast IRQ 0x00000004"            | <code>0x80000014</code>                 |
| "Fast IRQ 0x00000005"            | <code>0x80000015</code>                 |
| "Fast IRQ 0x00000006"            | <code>0x80000016</code>                 |
| "Fast IRQ 0x00000007"            | <code>0x80000017</code>                 |
| "Fast IRQ 0x00000008"            | <code>0x80000018</code>                 |
| "Fast IRQ 0x00000009"            | <code>0x80000019</code>                 |
| "Fast IRQ 0x0000000a"            | <code>0x8000001a</code>                 |
| "Fast IRQ 0x0000000b"            | <code>0x8000001b</code>                 |

| Trap identifier       | According <code>mcause</code> CSR value |
|-----------------------|---|
| "Fast IRQ 0x0000000c" | 0x8000001c                              |
| "Fast IRQ 0x0000000d" | 0x8000001d                              |
| "Fast IRQ 0x0000000e" | 0x8000001e                              |
| "Fast IRQ 0x0000000f" | 0x8000001f                              |
| "Unknown trap cause"  | undefined                               |

#### 4.6.4. Application Context Handling

Upon trap entry the RTE backups the *entire* application context (i.e. all `x` general purpose registers) to the stack. The context is restored automatically after trap completion. The base address of the according stack frame is copied to the `mscratch` CSR. By having this information available, the RTE provides dedicated functions for accessing and *altering* the application context:

*Listing 25. Context Access Functions*

```
// Prototypes
uint32_t neorv32_rte_context_get(int x); // read register x
void    neorv32_rte_context_put(int x, uint32_t data); write data to register x

// Examples
uint32_t tmp = neorv32_rte_context_get(9); // read register 'x9'
neorv32_rte_context_put(28, tmp); // write 'tmp' to register 'x28'
```



##### RISC-V E Extension

Registers `x16..x31` are not available if the RISC-V E ISA Extension is enabled.

The context access functions can be used by application-specific trap handlers to emulate unsupported CPU / SoC features like unimplemented IO modules, unsupported instructions and even unaligned memory accesses.

*Demo Program: Emulate Unaligned Memory Access*



A demo program, which showcases how to emulate unaligned memory accesses using the NEORV32 runtime environment can be found in `sw/example/demo_emulate_unaligned`.

# Chapter 5. On-Chip Debugger (OCD)

The NEORV32 Processor features an *on-chip debugger* (OCD) implementing the **execution-based debugging** scheme compatible to the **Minimal RISC-V Debug Specification**. A copy of the specification is available in [docs/references](#).

## Key Features

- standard JTAG access port
- full control of the CPU: halting, single-stepping and resuming
- indirect access to all core registers (via program buffer)
- indirect access to the whole processor address space (via program buffer)
- trigger module for hardware breakpoints
- compatible with upstream OpenOCD and GDB

## Section Structure

This chapter is separated into four sections:

- [Debug Transport Module \(DTM\)](#)
- [Debug Module \(DM\)](#)
- [CPU Debug Mode](#)
- [Trigger Module](#)

### *GDB + SVD*



Together with a third-party plugin the processor's SVD file can be imported right into GDB to allow comfortable debugging of peripheral/IO devices (see <https://github.com/stnolting/nerv32/discussions/656>).

### *Hands-On Tutorial*



A simple example on how to use NEORV32 on-chip debugger in combination with OpenOCD and the GNU debugger is shown in section [Debugging using the On-Chip Debugger](#) of the User Guide.

The NEORV32 on-chip debugger is based on four hardware modules:

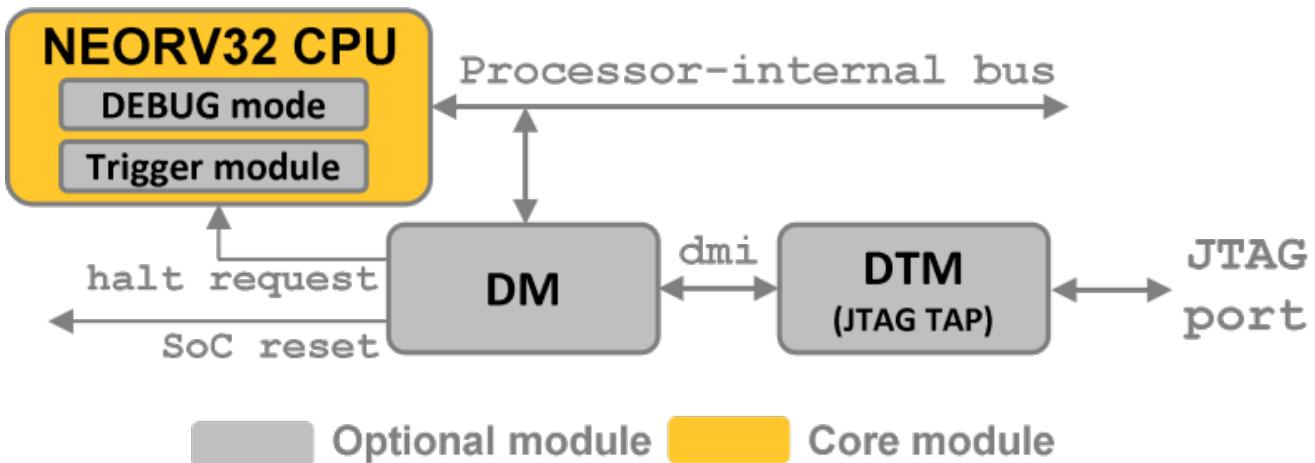


Figure 16. NEORV32 on-chip debugger complex

1. **Debug Transport Module (DTM)** ([rtl/core/neorv32\\_debug\\_dtm.vhd](#)): JTAG access tap to allow an external adapter to interface with the *debug module (DM)* using the *debug module interface (dmi)*.
2. **Debug Module (DM)** ([rtl/core/neorv32\\_debug\\_tm.vhd](#)): RISC-V debug module that is configured by the DTM via the dmi. From the CPU's "point of view" this module behaves as another memory-mapped peripheral that can be accessed via the processor-internal bus. The memory-mapped registers provide an internal *data buffer* for data transfer from/to the DM, a *code ROM* containing the "park loop" code, a *program buffer* to allow the debugger to execute small programs defined by the DM and a *status register* that is used to communicate *exception*, *halt*, *resume* and *execute* requests/acknowledges from/to the DM.
3. **CPU Debug Mode** extension (part of [rtl/core/neorv32\\_cpu\\_control.vhd](#)): This extension provides the "debug execution mode" as another operation mode, which is used to execute the park loop code from the DM. This mode also provides additional CSRs and instructions.
4. **CPU Trigger Module** (also part of [rtl/core/neorv32\\_cpu\\_control.vhd](#)): This module provides a single *hardware breakpoint*.

## Theory of Operation

When debugging the system using the OCD, the debugger issues a halt request to the CPU (via the CPU's `db_halt_req_i` signal) to make the CPU enter *debug mode*. In this mode, the application-defined architectural state of the system/CPU is "frozen" so the debugger can monitor it without interfering with the actual application. However, the OCD can also modify the entire architectural state at any time. While in debug mode, the debugger has full control over the entire CPU and processor operating at highest-privileged mode.

While in debug mode, the CPU executes the "park loop" code from the code ROM of the debug module (DM). This park loop implements an endless loop, where the CPU polls a memory-mapped **Status Register** that is controlled by the DM. The flags in this register are used to communicate requests from the DM and to acknowledge them by the CPU: trigger execution of the program buffer or resume the halted application. Furthermore, the CPU uses this register to signal that the CPU has halted after a halt request and to signal that an exception has been triggered while being in debug mode.

## 5.1. Debug Transport Module (DTM)

The debug transport module "DTM" (VHDL module: [rtl/core/neorv32\\_debug\\_dtm.vhd](#)) provides a JTAG test access port ("tap"). External JTAG access is provided by the following top-level ports:

*Table 79. JTAG top level signals*

| Name                     | Width | Direction | Description   |
|--------------------------|-------|-----------|---|
| <code>jtag_trst_i</code> | 1     | in        | TAP reset (low-active); this signal is optional, make sure to pull it <b>high</b> if not used |
| <code>jtag_tck_i</code>  | 1     | in        | serial clock  |
| <code>jtag_tdi_i</code>  | 1     | in        | serial data input   |
| <code>jtag_tdo_o</code>  | 1     | out       | serial data output  |
| <code>jtag_tms_i</code>  | 1     | in        | mode select   |

*Maximum JTAG Clock*



All JTAG signals are synchronized to the processor's clock domain. Hence, no additional clock domain is required for the DTM. However, this constraints the maximal JTAG clock frequency (`jtag_tck_i`) to be less than or equal to **1/5** of the processor clock frequency (`clk_i`).

*Maintaining JTAG Chain*



If the on-chip debugger is disabled the JTAG serial input `jtag_tdi_i` is directly connected to the JTAG serial output `jtag_tdo_o` to maintain the JTAG chain.

JTAG accesses are based on a single *instruction register IR*, which is 5 bit wide, and several *data registers DR* with different sizes. The individual data registers are accessed by writing the according address to the instruction register. The following table shows the available data registers and their addresses:

*Table 80. JTAG TAP registers*

| Address (via IR)   | Name                | Size (bits) | Description  |
|--------------------|---------------------|-------------|--|
| <code>00001</code> | <code>IDCODE</code> | 32          | identifier, version and part ID fields are hardwired to zero, manufacturer ID is assigned via the <a href="#">JEDEC_ID</a> top generic ( <a href="#">Processor Top Entity - Generics</a> ) |
| <code>10000</code> | <code>DTMCS</code>  | 32          | debug transport module control and status register   |
| <code>10001</code> | <code>DMI</code>    | 41          | debug module interface ( <i>dmi</i> ); 7-bit address, 32-bit read/write data, 2-bit operation ( <code>00</code> = NOP; <code>10</code> = write; <code>01</code> = read)                    |
| others             | <code>BYPASS</code> | 1           | default JTAG bypass register   |

Table 81. DTMCS - DTM Control and Status Register

| Bit(s) | Name         | R/W | Description   |
|--------|--------------|-----|---|
| 31:18  | -            | r/- | <i>reserved</i> , hardwired to zero   |
| 17     | dmihardreset | r/w | setting this bit will reset the debug module interface; this bit auto-clears  |
| 16     | dmireset     | r/w | setting this bit will clear the sticky error state; this bit auto-clears  |
| 15     | -            | r/- | <i>reserved</i> , hardwired to zero   |
| 14:12  | idle         | r/- | recommended idle states (= 0, no idle states required)  |
| 11:10  | dmistat      | r/- | DMI status: <b>00</b> = no error, <b>01</b> = reserved, <b>10</b> = operation failed, <b>11</b> = failed operation during pending DMI operation |
| 9:4    | abits        | r/- | number of address bits in DMI register (= 6)  |
| 3:0    | version      | r/- | <b>0001</b> = DTM is compatible to spec. versions v0.13 and v1.0  |

## 5.2. Debug Module (DM)

The debug module "DM" (VHDL module: [rtl/core/neorv32\\_debug\\_dm.vhd](#)) acts as a translation interface between abstract operations issued by the debugger application (like GDB) and the platform-specific debugger hardware. It supports the following features:

- Gives the debugger necessary information about the implementation.
- Allows the hart to be halted/resumed/reset and provides the current status.
- Provides abstract read and write access to the halted hart's general purpose registers.
- Provides access to a reset signal that allows debugging from the very first instruction after reset.
- Provides a *Program Buffer* to force the hart to execute arbitrary instructions.
- Allows memory access from a hart's point of view.

The NEORV32 DM follows the "Minimal RISC-V External Debug Specification" to provide full debugging capabilities while keeping resource/area requirements at a minimum. It implements the **execution based debugging scheme** for a single hart and provides the following core features:

- program buffer with 2 entries and an implicit `ebreak` instruction
- indirect bus access via the CPU using the program buffer
- abstract commands: "access register" plus auto-execution
- halt-on-reset capability

### *DM Spec. Version*



By default, the OCD's debug module supports version 1.0 of the RISC-V debug spec. For backwards compatibility, the DM can be "downgraded" back to version 0.13 via the `DM_LEGACY_MODE` generic (see [Processor Top Entity - Generics](#)).

The DM provides two access "point of views": accesses from the DTM via the *debug module interface* (`dmi`) and accesses from the CPU via the processor-internal bus system. From the DTM's point of view, the DM implements a set of **DM Registers** that are used to control and monitor the debugging session. From the CPU's point of view, the DM implements several memory-mapped registers (within the *normal* address space) that are used for communicating debugging control and status ([DM CPU Access](#)).

### 5.2.1. DM Registers

The DM is controlled via a set of registers that are accessed via the DTM's *debug module interface* (`dmi`). The following registers are implemented:



Write accesses to registers that are not implemented are simply ignored and read accesses to these registers will always return zero.

*Table 82. Available DM registers*

| Address | Name                | Description   |
|---------|---------------------|---|
| 0x04    | <b>data0</b>        | Abstract data 0, used for data transfer between debugger and processor                                      |
| 0x10    | <b>dmcontrol</b>    | Debug module control  |
| 0x11    | <b>dmstatus</b>     | Debug module status   |
| 0x12    | <b>hartinfo</b>     | Hart information  |
| 0x16    | <b>abstracts</b>    | Abstract control and status   |
| 0x17    | <b>command</b>      | Abstract command  |
| 0x18    | <b>abstractauto</b> | Abstract command auto-execution   |
| 0x1d    | <b>nextdm</b>       | Base address of next DM; reads as zero to indicate there is only one DM                                     |
| 0x20    | <b>progbuf0</b>     | Program buffer 0  |
| 0x21    | <b>progbuf1</b>     | Program buffer 1  |
| 0x38    | <b>sbc</b>          | System bus access control and status; reads as zero to indicate there is <b>no</b> direct system bus access |
| 0x40    | <b>haltsum0</b>     | Halted harts  |

**data0**0x04      **Abstract data 0****data0**Reset value: **0x00000000**

Basic read/write data exchange register to be used with abstract commands (for example to read/write data from/to CPU GPRs).

**dmcontrol**0x10      **Debug module control register****dmcontrol**Reset value: **0x00000000**

Control of the overall debug module and the hart. The following table shows all implemented bits. All remaining bits/bit-fields are configured as "zero" and are read-only. Writing '1' to these bits/fields will be ignored.

Table 83. **dmcontrol** Register Bits

| Bit | Name [RISC-V]       | R/W | Description   |
|-----|---------------------|-----|---|
| 31  | <b>haltreq</b>      | -/w | set/clear hart halt request                           |
| 30  | <b>resumereq</b>    | -/w | request hart to resume                                |
| 28  | <b>ackhavereset</b> | -/w | write 1 to clear *havereset flags                     |
| 1   | <b>ndmreset</b>     | r/w | put whole system (except OCD) into reset state when 1 |

| Bit | Name [RISC-V] | R/W | Description                              |
|-----|---------------|-----|--|
| 0   | dmactive      | r/w | DM enable; writing 0-1 will reset the DM |

## dmstatus

|  |                                     |          |
|--|-------------------------------------|----------|
| 0x11   | <b>Debug module status register</b> | dmstatus |
|  | Reset value: 0x00400083             |          |
| Current status of the overall debug module and the hart. The entire register is read-only. |                                     |          |

*Table 84. dmstatus Register Bits*

| Bit   | Name [RISC-V]           | Description  |
|-------|-------------------------|--|
| 31:23 | <i>reserved</i>         | reserved; always zero  |
| 22    | <b>impebreak</b>        | always <b>1</b> ; indicates an implicit <b>ebreak</b> instruction after the last program buffer entry                                    |
| 21:20 | <i>reserved</i>         | reserved; always zero  |
| 19    | <b>allhavereset</b>     | 1 when the hart is in reset  |
| 18    | <b>anyhavereset</b>     |  |
| 17    | <b>allresumeack</b>     | 1 when the hart has acknowledged a resume request  |
| 16    | <b>anyresumeack</b>     |  |
| 15    | <b>allnonexistent</b>   | always zero to indicate the hart is always existent  |
| 14    | <b>anynonexistent</b>   |  |
| 13    | <b>allunavail</b>       | 1 when the DM is disabled to indicate the hart is unavailable  |
| 12    | <b>anyunavail</b>       |  |
| 11    | <b>allrunning</b>       | 1 when the hart is running   |
| 10    | <b>anyrunning</b>       |  |
| 9     | <b>allhalted</b>        | 1 when the hart is halted  |
| 8     | <b>anyhalted</b>        |  |
| 7     | <b>authenticated</b>    | always <b>1</b> ; there is no authentication   |
| 6     | <b>authbusy</b>         | always <b>0</b> ; there is no authentication   |
| 5     | <b>hasresethaltereq</b> | always <b>0</b> ; halt-on-reset is not supported (directly)  |
| 4     | <b>confstrptrvalid</b>  | always <b>0</b> ; no configuration string available  |
| 3:0   | <b>version</b>          | debug spec. version; <b>0011</b> (v1.0) or <b>0010</b> (v0.13); configured via the <b>DM_LEGACY_MODE</b> Processor Top Entity - Generics |

*OCD Security*

JTAG access via the OCD is **always authenticated** (`dmstatus.authenticated = 1`). Hence, the entire system can always be accessed via the on-chip debugger.

**hartinfo****0x12 Hart information****hartinfo**

Reset value: *see below*

This register gives information about the hart. The entire register is read-only.

*Table 85. hartinfo Register Bits*

| Bit   | Name [RISC-V]           | Description  |
|-------|-------------------------|--|
| 31:24 | <i>reserved</i>         | reserved; always zero  |
| 23:20 | <code>nscratch</code>   | <code>0001</code> , number of <code>dscratch</code> * CPU registers = 1  |
| 19:17 | <i>reserved</i>         | reserved; always zero  |
| 16    | <code>dataaccess</code> | <code>0</code> , the <code>data</code> registers are shadowed in the hart's address space  |
| 15:12 | <code>datasize</code>   | <code>0001</code> , number of 32-bit words in the address space dedicated to shadowing the <code>data</code> registers (1 register)      |
| 11:0  | <code>dataaddr</code>   | = <code>dm_data_base_c(11:0)</code> , signed base address of <code>data</code> words (see address map in <a href="#">DM CPU Access</a> ) |

**abstracts****0x16 Abstract control and status****abstracts**

Reset value: `0x02000801`

Command execution info and status.

*Table 86. abstracts Register Bits*

| Bit   | Name [RISC-V]            | R/W | Description   |
|-------|--------------------------|-----|---|
| 31:29 | <i>reserved</i>          | r/- | reserved; always zero   |
| 28:24 | <code>progbufsize</code> | r/- | always <code>0010</code> : size of the program buffer ( <code>probuf</code> ) = 2 entries |
| 23:11 | <i>reserved</i>          | r/- | reserved; always zero   |
| 12    | <code>busy</code>        | r/- | <code>1</code> when a command is being executed   |
| 11    | <code>relaxedpriv</code> | r/- | always <code>1</code> : PMP rules are ignored when in debug mode                          |
| 10:8  | <code>cmderr</code>      | r/w | error during command execution (see below); has to be cleared by writing <code>111</code> |
| 7:4   | <i>reserved</i>          | r/- | reserved; always zero   |

| Bit | Name [RISC-V] | R/W | Description  |
|-----|---------------|-----|--|
| 3:0 | datacount     | r/- | always <b>0001</b> : number of implemented <b>data</b> registers for abstract commands = 1 |

Error codes in **cmderr** (highest priority first):

- **000** - no error
- **100** - command cannot be executed since hart is not in expected state
- **011** - exception during command execution
- **010** - unsupported command
- **001** - invalid DM register read/write while command is/was executing

#### command

##### 0x17 Abstract command

command

Reset value: **0x00000000**

Writing this register will trigger the execution of an abstract command. New command can only be executed if **cmderr** is zero. The entire register is write-only (reads will return zero).



The NEORV32 DM only supports **Access Register** abstract commands. These commands can only access the hart's GPRs x0 - x15/31 (abstract command register index **0x1000 - 0x101f**).

Table 87. **command** Register Bits

| Bit   | Name [RISC-V]     | R/W | Description / required value  |
|-------|-------------------|-----|---|
| 31:24 | cmdtype           | -/w | <b>00000000</b> to indicate "access register" command   |
| 23    | reserved          | -/w | reserved, has to be <b>0</b> when writing   |
| 22:20 | aarsize           | -/w | <b>010</b> to indicate 32-bit accesses  |
| 21    | aarpostincreme nt | -/w | <b>0</b> , post-increment is not supported  |
| 18    | postexec          | -/w | if set the program buffer is executed <i>after</i> the command                                  |
| 17    | transfer          | -/w | if set the operation in <b>write</b> is conducted   |
| 16    | write             | -/w | <b>1</b> : copy <b>data0</b> to <b>[regno]</b> , <b>0</b> : copy <b>[regno]</b> to <b>data0</b> |
| 15:0  | regno             | -/w | GPR-access only; has to be <b>0x1000 - 0x101f</b>   |

#### abstractauto

##### 0x18 Abstract command auto-execution

abstractauto

Reset value: **0x00000000**

Register to configure if a read/write access to a DM register re-triggers execution of the last abstract command.

Table 88. `abstractauto` Register Bits

| Bit | Name [RISC-V]                        | R/W | Description  |
|-----|--------------------------------------|-----|--|
| 17  | <code>autoexecprogbu<br/>f[1]</code> | r/w | when set reading/writing from/to <code>progbuf1</code> will execute <code>command</code> again |
| 16  | <code>autoexecprogbu<br/>f[0]</code> | r/w | when set reading/writing from/to <code>progbuf0</code> will execute <code>command</code> again |
| 0   | <code>autoexecdata[0<br/>]</code>    | r/w | when set reading/writing from/to <code>data0</code> will execute <code>command</code> again    |

### `progbuf`

|  |                         |                       |
|--|-------------------------|-----------------------|
| 0x20   | <b>Program buffer 0</b> | <code>progbuf0</code> |
| 0x21   | <b>Program buffer 1</b> | <code>progbuf1</code> |
| Reset value: <code>0x00000013</code> ("NOP") |                         |                       |
| Program buffer (two entries) for the DM.     |                         |                       |

### `haltsum0`

|  |                            |                       |
|--|----------------------------|-----------------------|
| 0x408                                      | <b>Halted harts status</b> | <code>haltsum0</code> |
| Reset value: <code>0x00000000</code>       |                            |                       |
| Hart has halted when according bit is set. |                            |                       |

Table 89. `haltsum0` Register Bits

| Bit | Name [RISC-V]            | R/W | Description              |
|-----|--------------------------|-----|--------------------------|
| 0   | <code>haltsum0[0]</code> | r/- | Hart is halted when set. |

## 5.2.2. DM CPU Access

From the CPU's perspective, the DM acts like another memory-mapped peripheral. It occupies 256 bytes of the CPU's address space starting at address `dm_base_c` (see table below). This address space is divided into four sections of 64 bytes each to provide access to the *park loop code ROM*, the *program buffer*, the *data buffer* and the *status register*. The program buffer, the data buffer and the status register do not fully occupy the 64-byte-wide sections and are mirrored several times to fill the entire section.

Table 90. DM CPU Access - Address Map

| Base address              | Actual size | Description                  |
|---------------------------|-------------|------------------------------|
| <code>0xffffffff00</code> | 64 bytes    | ROM for the "park loop" code |

| Base address | Actual size | Description                                |
|--------------|-------------|--|
| 0xffffffff40 | 16 bytes    | Program buffer ( <a href="#">proobuf</a> ) |
| 0xffffffff80 | 4 bytes     | Data buffer ( <a href="#">data0</a> )      |
| 0xfffffffcc0 | 4 bytes     | Control and Status Register                |

### DM Register Access



All memory-mapped registers of the DM can only be accessed by the CPU if it is actually in debug mode. Hence, the DM registers are not "visible" for normal CPU operations. Any CPU access outside of debug mode will raise a bus access fault exception.



### Park Loop Code Sources ("OCD Firmware")

The assembly sources of the **park loop code** are available in [sw/ocd-firmware/park\\_loop.S](#).

## Code ROM Entry Points

The park loop code provides two entry points where the actual code execution can start. These are used to enter the park loop either when an explicit debug-entry request has been issued (for example a halt request) or when an exception has occurred while executing code *inside* debug mode.

Table 91. Park Loop Entry Points

| Address   | Description             |
|---|-------------------------|
| <code>dm_exc_entry_c</code> ( <code>dm_base_c + 0</code> )  | Exception entry address |
| <code>dm_park_entry_c</code> ( <code>dm_base_c + 8</code> ) | Normal entry address    |

When the CPU enters or re-enters debug mode (for example via an `ebreak` in the DM's program buffer), it jumps to the *normal entry point* that is configured via the `CPU_DEBUG_PARK_ADDR` (= `dm_base_c`) generic ([CPU Top Entity - Generics](#)). By default, this generic is set to `dm_park_entry_c`, which is defined in main package file. If an exception is encountered during debug mode, the CPU jumps to the address of the *exception entry point* configured via the `CPU_DEBUG_EXC_ADDR` generic ([CPU Top Entity - Generics](#)). By default, this generic is set to `dm_exc_entry_c`, which is also defined in main package file.

## Status Register

The status register provides a direct communication channel between the CPU's debug mode executing the park loop and the debugger-controlled debug module. This register is used to communicate *requests*, which are issued by the DM and the according *acknowledges*, which are generated by the CPU.

There are only 4 bits in this register that are used to implement the requests/acknowledges. Each

bit is left-aligned in one sub-byte of the entire 32-bit register. Thus, the CPU can access each bit individually using *store-byte* and *load-byte* instructions. This eliminates the need to perform bit-masking in the park loop code leading to less code size and faster execution.

Table 92. DM Status Register - CPU Access

| Bit | Name                          | CPU access | Description  |
|-----|-------------------------------|------------|--|
| 0   | <code>sreg_halt_ack</code>    | read       | -  |
|     | -                             | write      | Set by the CPU while it is halted (and executing the park loop). |
| 8   | <code>sreg_resume_req</code>  | read       | Set by the DM to request the CPU to resume normal operation.     |
|     | <code>sreg_resume_ack</code>  | write      | Set by the CPU before it starts resuming.                        |
| 16  | <code>sreg_execute_req</code> | read       | Set by the DM to request execution of the program buffer.        |
|     | <code>sreg_execute_ack</code> | write      | Set by the CPU before it starts executing the program buffer.    |
| 24  | -                             | read       | -  |
|     | <code>sreg_execute_ack</code> | write      | Set by the CPU if an exception occurs while being in debug mode. |

## 5.3. CPU Debug Mode

The NEORV32 CPU Debug Mode is compatible to the [Minimal RISC-V Debug Specification 1.0 Sdext](#) (external debug) ISA extension. When enabled via the CPU's [Sdext ISA Extension](#) generic and/or the processor's [ON\\_CHIP\\_DEBUGGER\\_EN](#) it adds a new CPU operation mode ("debug mode"), three additional [CPU Debug Mode CSRs](#) and one additional instruction ([dret](#)) to the core.

Debug-mode is entered on any of the following events:

1. The CPU executes an [ebreak](#) instruction (when in machine-mode and [ebreakm](#) in [dcsr](#) is set OR when in user-mode and [ebreaku](#) in [dcsr](#) is set).
2. A debug halt request is issued by the DM (via CPU signal [db\\_halt\\_req\\_i](#), high-active).
3. The CPU completes executing of a single instruction while being in single-step debugging mode ([step](#) in [dcsr](#) is set).
4. A hardware trigger from the [Trigger Module](#) fires ([exe](#) and [action](#) in [tdata1](#) / [mcontrol](#) are set).



From a hardware point of view these debug-mode-entry conditions are special traps (synchronous exceptions or asynchronous interrupts) that are handled transparently by the control logic.

**Whenever the CPU enters debug-mode it performs the following operations:**

- wake-up CPU if it was send to sleep mode by the [wfi](#) instruction
- switch to debug-mode privilege level
- move the current program counter to [dpc](#)
- copy the hart's current privilege level to the [prv](#) flags in [dcsr](#)
- set [cause](#) in [\[\\_dcrs\]](#) according to the cause why debug mode is entered
- **no update** of [mtval](#), [mcause](#), [mtval](#) and [mstatus](#) CSRs
- load the address configured via the CPU's [CPU\\_DEBUG\\_PARK\\_ADDR](#) ([CPU Top Entity - Generics](#)) generic to the program counter jumping to the "debugger park loop" code stored in the debug module (DM)

**When the CPU is in debug-mode the following things are important:**

- while in debug mode, the CPU executes the parking loop and - if requested by the DM - the program buffer
- effective CPU privilege level is [machine](#) mode; any active physical memory protection (PMP) configuration is bypassed
- the [wfi](#) instruction acts as a [nop](#) (also during single-stepping)
- if an exception occurs while being in debug mode:
  - if the exception was caused by any debug-mode entry action the CPU jumps to the normal entry point (defined by [CPU\\_DEBUG\\_PARK\\_ADDR](#) generic of the [CPU Top Entity - Generics](#)) of the

park loop again (for example when executing `ebreak` while in debug-mode)

- for all other exception sources the CPU jumps to the exception entry point (defined by `CPU_DEBUG_EXC_ADDR` generic of the [CPU Top Entity - Generics](#)) to signal an exception to the DM; the CPU restarts the park loop again afterwards
- interrupts are disabled; however, they will remain pending and will get executed after the CPU has left debug mode and is not being single-stepped
- if the DM makes a resume request, the park loop exits and the CPU leaves debug mode (executing `dret`)
- the standard counters ([Machine](#)) Counter and Timer CSRs `[m]cycle[h]` and `[m]instret[h]` are stopped
- all [Hardware Performance Monitors \(HPM\) CSRs](#) are stopped

Debug mode is left either by executing the `dret` instruction or by performing a hardware reset of the CPU. Executing `dret` outside of debug mode will raise an illegal instruction exception.

#### Whenever the CPU leaves debug mode it performs the following operations:

- set the hart's current privilege level according to the `prv` flags of `dcsr`
- restore the original program counter from `[_dpcs]` resuming normal operation

### 5.3.1. CPU Debug Mode CSRs

Two additional CSRs are required by the *Minimal RISC-V Debug Specification*: the debug mode control and status register `dcsr` and the debug program counter `dpc`. An additional general purpose scratch register for debug-mode-only (`dscratch0`) allows faster execution by having a fast-accessible backup register. These CSRs are only accessible when the CPU is *in* debug mode. If these CSRs are accessed outside of debug mode an illegal instruction exception is raised.

#### `dcsr`

|             |   |
|-------------|---|
| Name        | Debug control and status register   |
| Address     | <code>0x7b0</code>  |
| Reset value | <code>0x40000413</code>   |
| ISA         | <code>Zicsr &amp; Sdext</code>  |
| Description | This register is used to configure the debug mode environment and provides additional status information. |

Table 93. Debug control and status register `dcsr` bits

| Bit   | Name [RISC-V]          | R/W | Description   |
|-------|------------------------|-----|---|
| 31:28 | <code>xdebugver</code> | r/- | <code>0100</code> - CPU debug mode is compatible to spec. version 1.0 |
| 27:16 | -                      | r/- | <code>000000000000</code> - reserved                                  |

| Bit | Name [RISC-V]          | R/W | Description   |
|-----|------------------------|-----|---|
| 15  | <code>ebreakm</code>   | r/w | <code>ebreak</code> instructions in <code>machine</code> mode will <i>enter</i> debug mode when set |
| 14  | <code>ebreakh</code>   | r/- | 0 - hypervisor mode not supported   |
| 13  | <code>ebreaks</code>   | r/- | 0 - supervisor mode not supported   |
| 12  | <code>ebreaku</code>   | r/w | <code>ebreak</code> instructions in <code>user</code> mode will <i>enter</i> debug mode when set    |
| 11  | <code>stepie</code>    | r/- | 0 - IRQs are disabled during single-stepping  |
| 10  | <code>stopcount</code> | r/- | 1 - standard counters and HPMs are stopped when in debug mode                                       |
| 9   | <code>stoptime</code>  | r/- | 0 - timers increment as usual   |
| 8:6 | <code>cause</code>     | r/- | cause identifier - why debug mode was entered (see below)   |
| 5   | -                      | r/- | 0 - reserved  |
| 4   | <code>mprvn</code>     | r/- | 1 - <code>mprv</code> in <code>mstatus</code> is also evaluated when in debug mode                  |
| 3   | <code>nmiip</code>     | r/- | 0 - non-maskable interrupt is pending   |
| 2   | <code>step</code>      | r/w | enable single-stepping when set   |
| 1:0 | <code>prv</code>       | r/w | CPU privilege level before/after debug mode   |

Cause codes in `dcsr.cause` (highest priority first):

- 010 - triggered by hardware [Trigger Module](#)
- 001 - executed [EBREAK](#) instruction
- 011 - external halt request (from DM)
- 100 - return from single-stepping

### dpc

|              |  |
|--------------|--|
| Name         | Debug program counter  |
| Address      | 0x7b1  |
| Reset value  | <code>CPU_BOOT_ADDR</code> , CPU boot address, 4-byte aligned (see <a href="#">CPU Top Entity - Generics</a> and <a href="#">Address Space</a> )   |
| ISA          | <code>Zicsr &amp; Sdext</code>   |
| Descripti on | The register is used to store the current program counter when debug mode is entered. The <code>dret</code> instruction will return to the address stored in <code>dpc</code> by automatically moving <code>dpc</code> to the program counter. |



`dpc[0]` is hardwired to zero. If `IALIGN` = 32 (i.e. [C ISA Extension](#) is disabled) then `dpc[1]` is also hardwired to zero.

**dscratch0**

---

|             |   |
|-------------|---|
| Name        | Debug scratch register 0  |
| Address     | <code>0x7b2</code>  |
| Reset value | <code>0x00000000</code>   |
| ISA         | <code>Zicsr &amp; Sdext</code>  |
| Description | The register provides a general purpose debug mode-only scratch register. |
| on          |   |

---

## 5.4. Trigger Module

"Normal" *software* breakpoints (using GDB's `b/break` command) are implemented by temporarily replacing the according instruction word by an `[c.]ebreak` instruction. However, this is not possible when debugging code that is executed from read-only memory (for example when debugging programs that are executed via the [Execute In Place Module \(XIP\)](#)). To circumvent this limitation a hardware trigger logic allows to (re-)enter debug-mode when instruction execution reaches a programmable address. These "hardware-assisted breakpoints" are used by GDB's `hb/hbreak` commands.

The RISC-V [Sdtrig](#) ISA extension adds a programmable *trigger module* to the CPU core that is enabled via the [Sdtrig ISA Extension](#) generic. The trigger module implements a subset of the features described in the "RISC-V Debug Specification / Trigger Module" and complies to version v1.0 of the [Sdtrig](#) spec.

The NEORV32 trigger module features only a *single* trigger implementing a "type 6 - instruction address match" trigger. This limitation is granted by the RISC-V debug spec and is sufficient to **debug code executed from read-only memory (ROM)**. The trigger module can also be used independently of the CPU debug-mode / [Sdext](#) ISA extension. Machine-mode software can use the trigger module to raise a breakpoint exception when instruction execution reaches a programmed address.



### Trigger Timing

When enabled the address match trigger will fire **BEFORE** the instruction at the programmed address gets executed.



### MEPC & DPC CSRs

The breakpoint exception when raised by the trigger module behaves different than the "normal" trapping (see [NEORV32 Trap Listing](#)): `mepc` / `dpc` is set to the address of the next instruction that needs to be executed to preserve the program flow. A "normal" breakpoint exception would set `mepc` / `dpc` to the address of the actual `ebreak` instruction itself.

### 5.4.1. Trigger Module CSRs

The [Sdtrig](#) ISA extension adds 4 additional CSRs that are accessible from debug-mode and also from machine-mode. Machine-mode write accesses can be ignored by setting 'dmode' in `tdata1`. This is automatically done by the debugger if it uses the trigger module for implementing a "hardware breakpoint"

#### `tselect`

Name Trigger select register

Address `0x7a0`

|             |   |
|-------------|---|
| Reset value | <code>0x00000000</code>   |
| ISA         | <code>Zicsr &amp; Sdtrig</code>   |
| Description | This CSR is hardwired to zero indicating there is only one trigger available. Any write on access is ignored. |

**tdata1**

|             |   |
|-------------|---|
| Name        | Trigger data register 1, visible as trigger "type 6 match control" ( <code>mcontrol6</code> ) |
| Address     | <code>0x7a1</code>  |
| Reset value | <code>0x60000048</code>   |
| ISA         | <code>Zicsr &amp; Sdtrig</code>   |
| Description | This CSR is used to configure the address match trigger using the "type 6" format.            |
| on          |   |

Table 94. Match Control CSR (`tdata1`) Bits

| Bit   | Name [RISC-V]            | R/W | Description   |
|-------|--------------------------|-----|---|
| 31:28 | <code>type</code>        | r/- | <code>0100</code> - address match trigger type 6  |
| 27    | <code>dmode</code>       | r/w | set to ignore write accesses to <code>tdata1</code> and <code>tdata2</code> from machine-mode; writable from debug-mode only                      |
| 26    | <code>uncertain</code>   | r/- | <code>0</code> - trigger satisfies the configured conditions  |
| 25    | <code>hit1</code>        | r/- | <code>0</code> - hardwired to zero, only <code>hit0</code> is used  |
| 24    | <code>vs</code>          | r/- | <code>0</code> - VS-mode not supported  |
| 23    | <code>vu</code>          | r/- | <code>0</code> - VU-mode not supported  |
| 22    | <code>hit0</code>        | r/c | set when trigger has fired ( <b>BEFORE</b> executing the triggering address); must be explicitly cleared by writing zero; writing 1 has no effect |
| 21    | <code>select</code>      | r/- | <code>0</code> - only address matching is supported   |
| 20:19 | reserved                 | r/- | <code>00</code> - hardwired to zero   |
| 18:16 | <code>size</code>        | r/- | <code>000</code> - match accesses of any size   |
| 15:12 | <code>action</code>      | r/w | <code>0000</code> = breakpoint exception on trigger match, <code>0001</code> = enter debug-mode on trigger match                                  |
| 11    | <code>chain</code>       | r/- | <code>0</code> - chaining is not supported as there is only one trigger   |
| 10:6  | <code>match</code>       | r/- | <code>0000</code> - equal-match only  |
| 6     | <code>m</code>           | r/- | <code>1</code> - trigger enabled when in machine-mode   |
| 5     | <code>uncertainen</code> | r/- | <code>0</code> - feature not supported, hardwired to zero   |

| Bit | Name [RISC-V] | R/W | Description  |
|-----|---------------|-----|--|
| 4   | s             | r/- | 0 - supervisor-mode not supported  |
| 3   | u             | r/- | 0/1 - trigger enabled when in user-mode, set if U ISA extension is enabled |
| 2   | execute       | r/w | set to enable trigger matching on instruction address                      |
| 1   | store         | r/- | 0 - store address/data matching not supported                              |
| 0   | load          | r/- | 0 - load address/data matching not supported                               |

## tdata2

|              |  |
|--------------|--|
| Name         | Trigger data register 2  |
| Address      | 0x7a2  |
| Reset value  | 0x00000000   |
| ISA          | Zicsr & Sdtrig   |
| Descripti on | Since only the "address match trigger" type is supported, this r/w CSR is used to configure the address of the triggering instruction. Note that the trigger module will fire <b>before</b> the instruction at the programmed address gets executed. |

## tinfo

|              |  |
|--------------|--|
| Name         | Trigger information register   |
| Address      | 0x7a4  |
| Reset value  | 0x01000006   |
| ISA          | Zicsr & Sdtrig   |
| Descripti on | The CSR shows global trigger information (see below). Any write access is ignored. |

Table 95. Trigger Info CSR (tinfo) Bits

| Bit   | Name [RISC-V] | R/W | Description                                     |
|-------|---------------|-----|---|
| 31:24 | version       | r/- | 0x01 - compatible to spec. version v1.0         |
| 23:15 | reserved      | r/- | 0x00 - hardwired to zero                        |
| 15:0  | info          | r/- | 0x0006 - only the "type 6 trigger" is supported |

# Chapter 6. Legal

## About

### The NEORV32 RISC-V Processor

<https://github.com/stnolting/neorv32>

Dipl.-Ing. (M.Sc.) Stephan Nolting

European Union, Germany

[stnolting@gmail.com](mailto:stnolting@gmail.com)

## License

### BSD 3-Clause License

Copyright (c) NEORV32 contributors. Copyright (c) 2020 - 2024, Stephan Nolting. All rights reserved.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

1. Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
3. Neither the name of the copyright holder nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT HOLDER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.



*SPDX Identifier*

SPDX-License-Identifier: BSD-3-Clause

# Proprietary Notice

- "GitHub" is a Subsidiary of Microsoft Corporation.
- "Vivado" and "Artix" are trademarks of AMD Inc.
- "AXI", "AXI4-Lite" and "AXI4-Stream" are trademarks of Arm Holdings plc.
- "ModelSim" is a trademark of Mentor Graphics – A Siemens Business.
- "Quartus Prime" and "Cyclone" are trademarks of Intel Corporation.
- "iCE40", "UltraPlus" and "Radiant" are trademarks of Lattice Semiconductor Corporation.
- "Windows" is a trademark of Microsoft Corporation.
- "Tera Term" copyright by T. Teranishi.
- "NeoPixel" is a trademark of Adafruit Industries.
- Images/figures made with *Microsoft Power Point*.
- Timing diagrams made with *WaveDrom Editor*.
- Documentation made with [asciidoc](#).
- "Segger Embedded Studio" and "J-Link" are trademarks of Segger Microcontroller Systems GmbH.

All further/unreferenced projects/products/brands belong to their according copyright holders. No copyright infringement intended.

# Disclaimer

This project is released under the BSD 3-Clause license. No copyright infringement intended. Other implied or used projects/sources might have different licensing – see their documentation to get more information.

# Limitation of Liability for External Links

This document contains links to the websites of third parties ("external links"). As the content of these websites is not under our control, we cannot assume any liability for such external content. In all cases, the provider of information of the linked websites is liable for the content and accuracy of the information provided. At the point in time when the links were placed, no infringements of the law were recognizable to us. As soon as an infringement of the law becomes known to us, we will immediately remove the link in question.

# Citing



This is an open-source project that is free of charge. Use this project in any way you like (as long as it complies to the permissive license). Please cite it appropriately. ☺

*Contributors & Community* ⓘ

Please add as many [contributors](#) as possible to the [author](#) field.  
This project would not be where it is without them.

*DOI*

This project provides a *digital object identifier* provided by [zenodo](#):  
DOI 10.5281/zenodo.5018888

## Acknowledgments

A big shout-out to the community and all the [contributors](#), who helped improving this project! This project would not be where it is without them. ⓘ

[RISC-V](#) - instruction sets want to be free!

Continuous integration provided by [GitHub Actions](#) and powered by [GHDL](#).