



RISC-V®

# The NEORV32 RISC-V Processor

## *Datasheet*

by Stephan Nolting (M.Sc.)

Version v1.8.4-r0-g7aa10cf4



## 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 .....	11
1.4. VHDL File Hierarchy .....	12
1.5. FPGA Implementation Results .....	14
1.6. CPU Performance .....	17
2. NEORV32 Processor (SoC) .....	18
2.1. Processor Top Entity - Signals .....	20
2.2. Processor Top Entity - Generics .....	23
2.3. Processor Clocking .....	28
2.4. Processor Reset .....	29
2.5. Processor Interrupts .....	30
2.5.1. RISC-V Standard Interrupts .....	30
2.5.2. NEORV32-Specific Fast Interrupt Requests .....	30
2.6. Address Space .....	32
2.6.1. Physical Memory Attributes (PMAs) .....	32
2.6.2. CPU Data and Instruction Access .....	33
2.6.3. Address Space Layout .....	34
2.6.4. Memory Configuration .....	34
2.6.5. Boot Configuration .....	36
2.7. Processor-Internal Modules .....	38
2.7.1. Instruction Memory (IMEM) .....	39
2.7.2. Data Memory (DMEM) .....	41
2.7.3. Bootloader ROM (BOOTROM) .....	42
2.7.4. Processor-Internal Instruction Cache (iCACHE) .....	43
2.7.5. Processor-Internal Data Cache (dCACHE) .....	45
2.7.6. Processor-External Memory Interface (WISHBONE) .....	47
2.7.7. Internal Bus Monitor (BUSKEEPER) .....	51
2.7.8. General Purpose Input and Output Port (GPIO) .....	53
2.7.9. Watchdog Timer (WDT) .....	54
2.7.10. Machine System Timer (MTIME) .....	57
2.7.11. Primary Universal Asynchronous Receiver and Transmitter (UART0) .....	58
2.7.12. Secondary Universal Asynchronous Receiver and Transmitter (UART1) .....	63
2.7.13. Serial Peripheral Interface Controller (SPI) .....	64
2.7.14. Serial Data Interface Controller (SDI) .....	68
2.7.15. Two-Wire Serial Interface Controller (TWI) .....	71

2.7.16. One-Wire Serial Interface Controller (ONEWIRE) . . . . .	74
2.7.17. Pulse-Width Modulation Controller (PWM) . . . . .	79
2.7.18. True Random-Number Generator (TRNG) . . . . .	81
2.7.19. Custom Functions Subsystem (CFS) . . . . .	83
2.7.20. Smart LED Interface (NEOLED) . . . . .	85
2.7.21. External Interrupt Controller (XIRQ) . . . . .	90
2.7.22. General Purpose Timer (GPTMR) . . . . .	92
2.7.23. Execute In Place Module (XIP) . . . . .	94
2.7.24. System Configuration Information Memory (SYSINFO) . . . . .	99
3. NEORV32 Central Processing Unit (CPU) . . . . .	103
3.1. RISC-V Compatibility . . . . .	103
3.2. Architecture . . . . .	105
3.2.1. CPU Register File . . . . .	105
3.2.2. CPU Arithmetic Logic Unit . . . . .	106
3.2.3. CPU Bus Unit . . . . .	106
3.2.4. CPU Control Unit . . . . .	107
3.3. Sleep Mode . . . . .	107
3.4. Full Virtualization . . . . .	108
3.5. CPU Top Entity - Signals . . . . .	109
3.6. CPU Top Entity - Generics . . . . .	111
3.7. Instruction Sets and Extensions . . . . .	112
3.7.1. <b>I</b> ISA Extension . . . . .	112
3.7.2. <b>B</b> ISA Extension . . . . .	113
3.7.3. <b>C</b> ISA Extension . . . . .	113
3.7.4. <b>E</b> ISA Extension . . . . .	114
3.7.5. <b>M</b> ISA Extension . . . . .	114
3.7.6. <b>U</b> ISA Extension . . . . .	114
3.7.7. <b>X</b> ISA Extension . . . . .	114
3.7.8. <b>Zifencei</b> ISA Extension . . . . .	114
3.7.9. <b>Zfinx</b> ISA Extension . . . . .	115
3.7.10. <b>Zicntr</b> ISA Extension . . . . .	116
3.7.11. <b>Zicond</b> ISA Extension . . . . .	116
3.7.12. <b>Zicsr</b> ISA Extension . . . . .	116
3.7.13. <b>Zihpm</b> ISA Extension . . . . .	117
3.7.14. <b>Zmmul</b> - ISA Extension . . . . .	117
3.7.15. <b>Zxfcfs</b> ISA Extension . . . . .	117
3.7.16. <b>PMP</b> ISA Extension . . . . .	117
3.7.17. <b>Sdext</b> ISA Extension . . . . .	118
3.7.18. <b>Sdtrig</b> ISA Extension . . . . .	118

3.8. Custom Functions Unit (CFU) . . . . .	119
3.8.1. CFU Instruction Formats . . . . .	119
3.8.2. CFU R3-Type Instructions . . . . .	119
3.8.3. CFU R4-Type Instructions . . . . .	120
3.8.4. CFU R5-Type Instructions . . . . .	121
3.8.5. Using Custom Instructions in Software . . . . .	122
3.8.6. Custom Instructions Hardware . . . . .	123
3.9. Control and Status Registers (CSRs) . . . . .	125
3.9.1. Floating-Point CSRs . . . . .	130
3.9.2. Machine Configuration CSRs . . . . .	132
3.9.3. Machine Trap Setup CSRs . . . . .	133
3.9.4. Machine Trap Handling CSRs . . . . .	137
3.9.5. Machine Physical Memory Protection CSRs . . . . .	140
3.9.6. (Machine) Counter and Timer CSRs . . . . .	142
3.9.7. Hardware Performance Monitors (HPM) CSRs . . . . .	144
3.9.8. Machine Counter Setup CSRs . . . . .	146
3.9.9. Machine Information CSRs . . . . .	147
3.9.10. NEORV32-Specific CSRs . . . . .	149
3.9.11. Traps, Exceptions and Interrupts . . . . .	150
3.9.12. Bus Interface . . . . .	154
4. Software Framework . . . . .	157
4.1. Compiler Toolchain . . . . .	158
4.2. Core Libraries . . . . .	159
4.3. Application Makefile . . . . .	161
4.3.1. Makefile Targets . . . . .	161
4.3.2. Makefile Configuration . . . . .	162
4.3.3. Default Compiler Flags . . . . .	163
4.3.4. Custom (Compiler) Flags . . . . .	164
4.4. Executable Image Format . . . . .	165
4.4.1. Linker Script . . . . .	165
4.4.2. RAM Layout . . . . .	166
4.4.3. C Standard Library . . . . .	168
4.4.4. Executable Image Generator . . . . .	168
4.4.5. Start-Up Code (crt0) . . . . .	169
4.5. Bootloader . . . . .	171
4.5.1. Bootloader SoC/CPU Requirements . . . . .	171
4.5.2. Bootloader Flash Requirements . . . . .	171
4.5.3. Bootloader Console . . . . .	172
4.5.4. Auto Boot Sequence . . . . .	175

4.5.5. Bootloader Error Codes .....	175
4.6. NEORV32 Runtime Environment .....	177
4.6.1. RTE Operation .....	177
4.6.2. Using the RTE .....	177
4.6.3. Default RTE Trap Handlers .....	179
5. On-Chip Debugger (OCD) .....	182
5.1. Debug Transport Module (DTM) .....	184
5.2. Debug Module (DM) .....	186
5.2.1. DM Registers .....	186
5.2.2. DM CPU Access .....	191
5.3. CPU Debug Mode .....	194
5.3.1. CPU Debug Mode CSRs .....	195
5.4. Trigger Module .....	198
5.4.1. Trigger Module CSRs .....	198
6. Legal .....	201
License .....	201
Proprietary Notice .....	202
Disclaimer .....	202
Limitation of Liability for External Links .....	202
Citing .....	202
Acknowledgments .....	203

# 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
- 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 Memory Interface \(WISHBONE\)](#) to attach processor-external IP modules
- [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  
| |-icons         - Misc. symbols  
| |-references    - Data sheets and RISC-V specs.  
| \-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`.



### *NEORV32 VHDL Library*

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

<code>neorv32_top.vhd</code>	- NEORV32 Processor top entity
└ <code>neorv32_fifo.vhd</code>	- General purpose FIFO component
└ <code>neorv32_package.vhd</code>	- Processor/CPU main VHDL package file
└ <code>neorv32_cpu.vhd</code>	- NEORV32 CPU top entity
<code>neorv32_cpu_alu.vhd</code>	- Arithmetic/logic unit
<code>neorv32_cpu_cp_bitmanip.vhd</code>	- Bit-manipulation co-processor (B ext.)
<code>neorv32_cpu_cp_cfu.vhd</code>	- Custom instructions co-processor (Zxcfu ext.)
<code>neorv32_cpu_cp_cond.vhd</code>	- Conditional operations co-processor (Zicond ext.)
<code>neorv32_cpu_cp_fpu.vhd</code>	- Floating-point co-processor (Zfinx ext.)
<code>neorv32_cpu_cp_muldiv.vhd</code>	- Mul/Div co-processor (M ext.)
<code>neorv32_cpu_cp_shifter.vhd</code>	- Bit-shift co-processor (base ISA)
└ <code>neorv32_cpu_bus.vhd</code>	- Load/store unit + physical memory protection
└ <code>neorv32_cpu_control.vhd</code>	- CPU control, exception system and CSRs
<code>neorv32_cpu_decompressor.vhd</code>	- Compressed instructions decoder
└ <code>neorv32_cpu_regfile.vhd</code>	- Data register file
└ <code>neorv32_boot_rom.vhd</code>	- Bootloader ROM
<code>neorv32_bootloader_image.vhd</code>	- Bootloader ROM memory image
└ <code>neorv32_busswitch.vhd</code>	- Processor bus switch for CPU buses (I&D)
└ <code>neorv32_bus_keeper.vhd</code>	- Processor-internal bus monitor
└ <code>neorv32_cfs.vhd</code>	- Custom functions subsystem
└ <code>neorv32_dcache.vhd</code>	- Processor-internal data cache
└ <code>neorv32_debug_dm.vhd</code>	- on-chip debugger: debug module
└ <code>neorv32_debug_dtm.vhd</code>	- on-chip debugger: debug transfer module
└ <code>neorv32_dmem.entity.vhd</code>	- Processor-internal data memory (entity-only!)
└ <code>neorv32_gpio.vhd</code>	- General purpose input/output port unit
└ <code>neorv32_gptmr.vhd</code>	- General purpose 32-bit timer
└ <code>neorv32_icache.vhd</code>	- Processor-internal instruction cache
└ <code>neorv32_imem.entity.vhd</code>	- Processor-internal instruction memory (entity-only!)
<code>neorv32_application_image.vhd</code>	- IMEM application initialization image
└ <code>neorv32_mtime.vhd</code>	- Machine system timer
└ <code>neorv32_neoled.vhd</code>	- NeoPixel (TM) compatible smart LED interface
└ <code>neorv32_onewire.vhd</code>	- One-Wire serial interface controller
└ <code>neorv32_pwm.vhd</code>	- Pulse-width modulation controller

↳ neorv32_sdi.vhd	- Serial data interface controller (SPI device)
↳ 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_wishbone.vhd	- External (Wishbone) bus interface
↳ neorv32_xip.vhd	- Execute in place module
↳ neorv32_xirq.vhd	- External interrupt controller
↳ mem/neorv32_dmem.default.vhd	- <u>_Default_</u> data memory (architecture-only)
↳ mem/neorv32_imem.default.vhd	- <u>_Default_</u> instruction memory (architecture-only)



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.

### 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> , " <i>balanced optimization</i> ", $f_{max}$ 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="#">1.6.8.3++</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> , " <i>balanced optimization</i> "

*Table 1. Hardware utilization by processor module (mandatory modules highlighted in bold)*

Module	Description	LEs	FFs	MEM bits	DSPs
Boot ROM	Bootloader ROM (4kB)	3	2	32768	0
<b>BUSKEEPER</b>	Processor-internal bus monitor	28	15	0	0
<b>BUSSWITCH</b>	Bus multiplexer for CPU instr. and data interface	69	8	0	0
CFS	Custom functions subsystem <sup>[1]</sup>	-	-	-	-
DM	On-chip debugger - debug module	391	220	0	0
DTM	On-chip debugger - debug transfer module (JTAG)	259	221	0	0
DMEM	Processor-internal data memory (8kB)	18	2	65536	0
GPIO	General purpose input/output ports	102	98	0	0
GPTMR	General Purpose Timer	153	105	0	0
iCACHE	Instruction cache (2x4 blocks, 64 bytes per block)	417	297	4096	0
dCACHE	Data cache (8 blocks, 64 bytes per block)	417	297	4096	0
IMEM	Processor-internal instruction memory (16kB)	12	2	131072	0
MTIME	Machine system timer	345	166	0	0
NEOLED	Smart LED Interface (NeoPixel/WS2812B) (FIFO_depth=1)	227	184	0	0
ONEWIRE	1-wire interface	107	77	0	0
PWM	Pulse_width modulation controller (8 channels)	128	117	0	0
SPI	Serial peripheral interface	114	94	0	0
SDI	Serial data interface	72	66	0	0
<b>SYSINFO</b>	System configuration information memory	13	11	0	0
TRNG	True random number generator	89	79	0	0

Module	Description	LEs	FFs	MEM bits	DSPs
TWI	Two-wire interface	77	43	0	0
UART0, UART1 (FIFO_depth=1)	Universal asynchronous receiver/transmitter 0/1	195	143	0	0
WDT	Watchdog timer	61	46	0	0
WISHBONE	External memory interface	120	112	0	0
XIP	Execute in place module	318	244	0	0
XIRQ	External interrupt controller (32 channels)	245	200	0	0

## Processor - Exemplary Setup

HW version:	<a href="#">1.7.7.3</a>
CPU:	<a href="#">rv32imcu_Zicsr_Zicnt_DEBUG + FST_MUL + FAST_SHIFT</a>
Peripherals:	<a href="#">UART0 + MTIME + GPIO</a>
FPGA:	<a href="#">Xilinx Artix-7 xc7a35ticsg324-1L</a>
Toolchain:	Xilinx Vivado 2019.2
Constraints:	clock constrained to 150 MHz, default/standard synthesis & implementation settings

LUTs	FFs	BRAMs	DSPs	Clock
2488	1807	7	4	150 MHz

### *Exemplary Processor Setups*



Check out the [neorv32-setups](#) repository (on GitHub: <https://github.com/stnolting/neorv32-setups>), which provides several demo setups and community projects for various FPGA boards and toolchains.

## 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.



### Dhrystone Benchmark

A very simple port of the Dhrystone benchmark is also available in [sw/example/dhrystone](#).

*Table 2. Configuration*

HW version:	<a href="#">1.5.7.10</a>
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 (with <code>-O3</code> ), see makefile

*Table 3. CoreMark results*

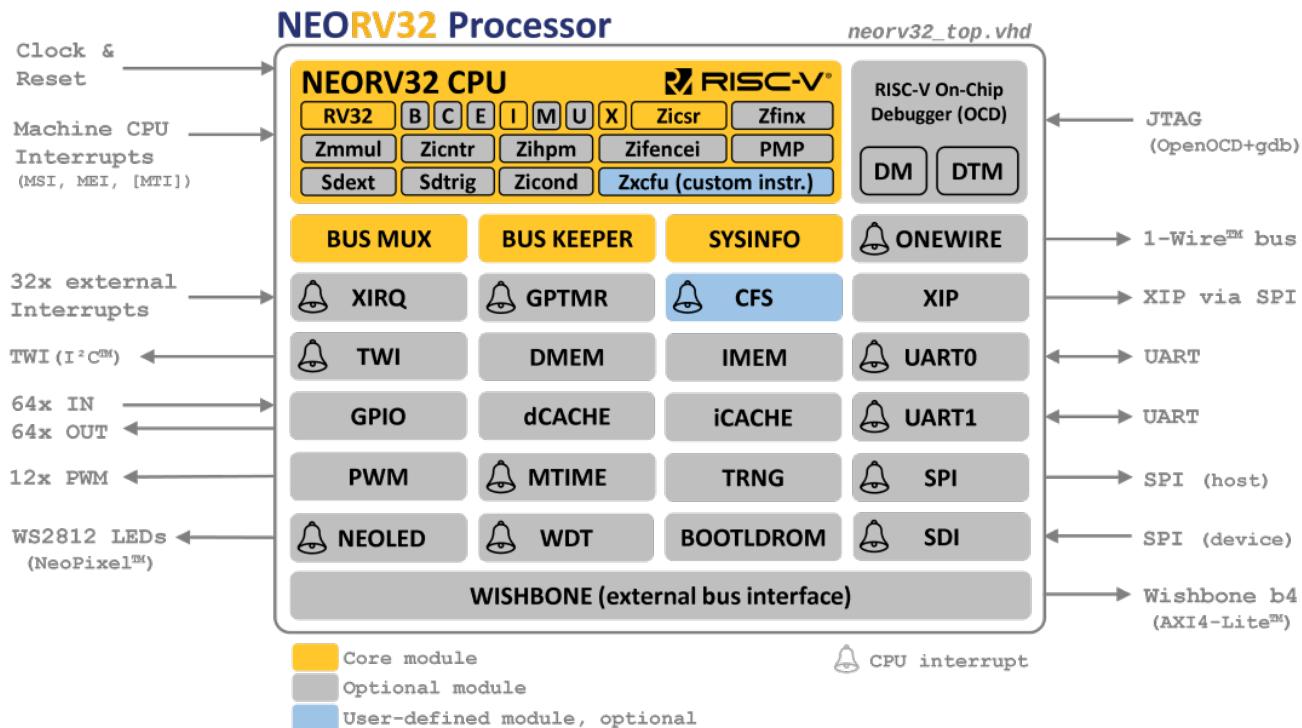
CPU	CoreMark Score	CoreMarks/ MHz	Average CPI
<i>small (rv32i_Zicsr)</i>	33.89	<b>0.3389</b>	<b>4.04</b>
<i>medium (rv32imc_Zicsr)</i>	62.50	<b>0.6250</b>	<b>5.34</b>
<i>performance (rv32imc_Zicsr + perf. options)</i>	95.23	<b>0.9523</b>	<b>3.54</b>

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](#).

[1] Resource utilization depends on custom design logic.

# 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.



## 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**)
- 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 (**WISHBONE**)
- *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**)
- *optional* execute in place module (**XIP**)
- *optional* 1-wire serial interface controller (**ONEWIRE**), compatible to the 1-wire standard
- *optional* on-chip debugger with JTAG TAP (**OCD**)
- bus keeper to monitor processor-internal bus transactions (**BUSKEEPER**)
- 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 input signals provide default values in case they are not explicitly assigned during instantiation.



### *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	Direct Description	ion
<b>Global Control (Processor Clocking and Processor Reset)</b>			
<code>clk_i</code>	1	in	global clock line, all registers triggering on rising edge
<code>rstn_i</code>	1	in	global reset, asynchronous, <b>low-active</b>
<b>JTAG Access Port for On-Chip Debugger (OCD)</b>			
<code>jtag_trst_i</code>	1	in	TAP reset, low-active (optional)
<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
<b>Processor-External Memory Interface (WISHBONE)</b>			
<code>wb_tag_o</code>	3	out	tag (access type identifier)
<code>wb_adr_o</code>	32	out	destination address
<code>wb_dat_i</code>	32	in	write data

Name	Width	Direct	Description
ion			
wb_dat_o	32	out	read data
wb_we_o	1	out	write enable ('0' = read transfer)
wb_sel_o	4	out	byte enable
wb_stb_o	1	out	strobe
wb_cyc_o	1	out	valid cycle
wb_lock_o	1	out	exclusive access request
wb_ack_i	1	in	transfer acknowledge
wb_err_i	1	in	transfer error
Advanced Memory Control Signals			
fence_o	1	out	indicates an executed <i>fence</i> instruction
fencei_o	1	out	indicates an executed <i>fencei</i> instruction
Execute In Place Module (XIP)			
xip_csn_o	1	out	chi select, low-active
xip_clk_o	1	out	serial clock
xip_dat_i	1	in	serial data input
xip_dat_o	1	out	serial data output
General Purpose Input and Output Port (GPIO)			
gpio_o	64	out	general purpose parallel output
gpio_i	64	in	general purpose parallel input
Primary Universal Asynchronous Receiver and Transmitter (UART0)			
uart0_txd_o	1	out	serial transmitter
uart0_rxd_i	1	in	serial receiver
uart0_rts_o	1	out	RX ready to receive new char
uart0_cts_i	1	in	TX allowed to start sending
Secondary Universal Asynchronous Receiver and Transmitter (UART1)			
uart1_txd_o	1	out	serial transmitter
uart1_rxd_i	1	in	serial receiver
uart1_rts_o	1	out	RX ready to receive new char
uart1_cts_i	1	in	TX allowed to start sending
Serial Peripheral Interface Controller (SPI)			
spi_clk_o	1	out	controller clock line

Name	Width	Direct	Description
ion			
spi_dat_o	1	out	serial data output
spi_dat_i	1	in	serial data input
spi_csn_o	8	out	select (low-active)
<b>Serial Data Interface Controller (SDI)</b>			
sdi_clk_i	1	in	controller clock line
sdi_dat_o	1	out	serial data output
sdi_dat_i	1	in	serial data input
sdi_csn_i	1	in	chip select (low-active)
<b>Two-Wire Serial Interface Controller (TWI)</b>			
twi_sda_i	1	in	serial data line sense input
twi_sda_o	1	out	serial data line output (pull low only)
twi_scl_i	1	in	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	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	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>External Interrupt Controller (XIRQ)</b>			
xirq_i	32	in	external interrupt requests
<b>RISC-V Machine-Mode Processor Interrupts</b>			
mtime_irq_i	1	in	machine timer interrupt (RISC-V), high-level-active
msw_irq_i	1	in	machine software interrupt (RISC-V), high-level-active
mext_irq_i	1	in	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.



### *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.



### *Configuration Check*

Not all configuration combinations are valid. The processor RTL code provides sanity checks to inform the user during synthesis/simulation if an invalid combination has been detected. It is recommended to run a quick simulation using the provided simulation/GHDL scripts to verify the configuration of the processor generics.



### *Table Abbreviations*

The generic type "suv(x:y)" defines a `std_ulogic_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>INT_BOOTLOADER_EN</code>	boolean	false	Implement the processor-internal <b>Bootloader ROM (BOOTROM)</b> , pre-initialized with the default <b>Bootloader</b> image.
<code>HART_ID</code>	suv(31:0)	0x00000000	The hart thread ID of the CPU (passed to <code>mhartid</code> CSRs).

Name	Type	Default	Description
VENDOR_ID	suv(31:0)	0x00000000	JEDEC ID (passed to <code>mvendorid</code> CSRs).
CUSTOM_ID	suv(31:0)	0x00000000	User-defined identifier to identify a certain setup or to pass user-defined flags to software (via the <a href="#">System Configuration Information Memory (SYSINFO)</a> ).
ON_CHIP_DEBUGGER_EN	boolean	false	Implement the on-chip debugger <a href="#">On-Chip Debugger (OCD)</a> and the CPU debug mode.
<b>CPU Instruction Sets and Extensions</b>			
CPU_EXTENSION_RISC_V_B	boolean	false	Enable <a href="#">B ISA Extension</a> (bit-manipulation).
CPU_EXTENSION_RISC_V_C	boolean	false	Enable <a href="#">C ISA Extension</a> (compressed instructions).
CPU_EXTENSION_RISC_V_E	boolean	false	Enable <a href="#">E ISA Extension</a> (reduced register file size).
CPU_EXTENSION_RISC_V_M	boolean	false	Enable <a href="#">M ISA Extension</a> (hardware-based integer multiplication and division).
CPU_EXTENSION_RISC_V_U	boolean	false	Enable <a href="#">U ISA Extension</a> (less-privileged user mode).
CPU_EXTENSION_RISC_V_Zfinx	boolean	false	Enable <a href="#">Zfinx ISA Extension</a> (single-precision floating-point unit).
CPU_EXTENSION_RISC_V_Zicntr	boolean	true	Enable <a href="#">Zicntr ISA Extension</a> (CPU base counters).
CPU_EXTENSION_RISC_V_Zicond	boolean	false	Enable <a href="#">Zicond ISA Extension</a> (conditional operations).
CPU_EXTENSION_RISC_V_Zihpm	boolean	false	Enable <a href="#">Zihpm ISA Extension</a> (hardware performance monitors).
CPU_EXTENSION_RISC_V_Zifencei	boolean	false	Enable <a href="#">Zifencei ISA Extension</a> (instruction stream synchronization).
CPU_EXTENSION_RISC_V_Zmmul	boolean	false	Enable <a href="#">Zmmul - ISA Extension</a> (hardware-based integer multiplication).
CPU_EXTENSION_RISC_V_Zxfcu	boolean	false	Enable NEORV32-specific <a href="#">Zxfcu ISA Extension</a> (custom RISC-V instructions).
<b>CPU Tuning Options</b>			
FAST_MUL_EN	boolean	false	Implement fast (but large) full-parallel multipliers (trying to infer DSP blocks).
FAST_SHIFT_EN	boolean	false	Implement fast (but large) pull-parallel barrel shifters.
CPU_IPB_ENTRIES	natural	1	Number of entries in the CPU's instruction prefetch buffer.

Name	Type	Default	Description
<b>Physical Memory Protection (PMP ISA Extension)</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.
<b>Hardware Performance Monitors (Zihpm ISA Extension)</b>			
HPM_NUM_CNTS	natural	0	Number of implemented hardware performance monitor counters (0..29).
HPM_CNT_WIDTH	natural	40	Total LSB-aligned size of each HPM counter. Min 0, max 64.
<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.
<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.
<b>Processor-Internal Instruction Cache (iCACHE)</b>			
ICACHE_EN	boolean	false	Implement the instruction cache.
ICACHE_NUM_BLOCKS	natural	4	Number of blocks ("pages" or "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.
ICACHE_ASSOCIATIVITY	natural	1	Associativity (number of sets). Allowed configurations: 1 = 1 set, direct mapped; 2 = 2-way set-associative.
<b>Processor-Internal Data Cache (dCACHE)</b>			
DCACHE_EN	boolean	false	Implement the data cache.
DCACHE_NUM_BLOCKS	natural	4	Number of blocks ("pages" or "lines") Has to be a power of two.
DCACHE_BLOCK_SIZE	natural	64	Size in bytes of each block. Has to be a power of two.
<b>Processor-External Memory Interface (WISHBONE)</b>			
MEM_EXT_EN	boolean	false	Implement the external bus interface.

Name	Type	Default	Description
MEM_EXT_TIMEOUT	natural	255	Clock cycles after which a pending external bus access will auto-terminate and raise a bus fault exception.
MEM_EXT_PIPE_MODE	boolean	false	Use <i>standard</i> ("classic") Wishbone protocol when false. Use <i>pipelined</i> Wishbone protocol when true.
MEM_EXT_BIG_ENDIAN	boolean	false	Use BIG endian data order interface for external bus.
MEM_EXT_ASYNC_RX	boolean	false	Disable input registers when true.
MEM_EXT_ASYNC_TX	boolean	false	Disable output registers when true.
<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.
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 <a href="#">Secondary Universal Asynchronous Receiver and Transmitter (UART1)</a> .
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 <a href="#">Serial Peripheral Interface Controller (SPI)</a> .
IO_SPI_FIFO	natural	1	Depth of the <a href="#">Serial Peripheral Interface Controller (SPI)</a> FIFO. Has to be a power of two, min 1, max 32768.

Name	Type	Default	Description
IO_SDI_EN	boolean	false	Implement the <a href="#">Serial Data Interface Controller (SDI)</a> .
IO_SDI_FIFO	natural	1	Depth of the <a href="#">Serial Data Interface Controller (SDI)</a> FIFO. Has to be a power of two, min 1, max 32768.
IO_TWI_EN	boolean	false	Implement the <a href="#">Two-Wire Serial Interface Controller (TWI)</a> .
IO_PWM_NUM_CH	natural	0	Number of channels of the <a href="#">Pulse-Width Modulation Controller (PWM)</a> to implement (0..12).
IO_WDT_EN	boolean	false	Implement the <a href="#">Watchdog Timer (WDT)</a> .
IO_TRNG_EN	boolean	false	Implement the <a href="#">True Random-Number Generator (TRNG)</a> .
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 <a href="#">Custom Functions Subsystem (CFS)</a> .
IO_CFS_CONFIG	suv(31:0)	0x00000000	"Conduit" generic to pass user-defined flags to the <a href="#">Custom Functions Subsystem (CFS)</a> .
IO_CFS_IN_SIZE	natural	32	Size of the <a href="#">Custom Functions Subsystem (CFS)</a> input signal conduit ( <a href="#">cfs_in_i</a> ).
IO_CFS_OUT_SIZE	natural	32	Size of the <a href="#">Custom Functions Subsystem (CFS)</a> output signal conduit ( <a href="#">cfs_out_o</a> ).
IO_NEOLED_EN	boolean	false	Implement the <a href="#">Smart LED Interface (NEOLED)</a> .
IO_NEOLED_TX_FIFO	natural	1	TX FIFO depth of the the <a href="#">Smart LED Interface (NEOLED)</a> . Has to be a power of two, min 1, max 32768.
IO_GPTMR_EN	boolean	false	Implement the <a href="#">General Purpose Timer (GPTMR)</a> .
IO_XIP_EN	boolean	false	Implement the <a href="#">Execute In Place Module (XIP)</a> .
IO_ONE_EN	boolean	false	Implement the <a href="#">One-Wire Serial Interface Controller (ONEWIRE)</a> .

## 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. External "clocks" like the OCD's JTAG clock or the TWI's serial clock are synchronized into the processor's clock domain before being further processed.



Only the registers of the [Processor Reset](#) system trigger on a *falling* clock edge.

Many processor modules like the UARTs or the timers require 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. These clock enable signals are synchronous to the system's main clock and will be high for only a single cycle. Hence, processor modules can use these enables for sub-main-clock operations while still having 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 according 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:	$f/2$	$f/4$	$f/8$	$f/64$	$f/128$	$f/1024$	$f/2048$	$f/4096$

The software framework provides pre-defined aliases for the prescaler select bits:

*Listing 1. Prescaler Aliases from nerv32.h*

```
enum NEORV32_CLOCK_PRSC_enum {
    CLK_PRSC_2      = 0, /*< CPU_CLK (from clk_i top signal) / 2 */
    CLK_PRSC_4      = 1, /*< CPU_CLK (from clk_i top signal) / 4 */
    CLK_PRSC_8      = 2, /*< CPU_CLK (from clk_i top signal) / 8 */
    CLK_PRSC_64     = 3, /*< CPU_CLK (from clk_i top signal) / 64 */
    CLK_PRSC_128    = 4, /*< CPU_CLK (from clk_i top signal) / 128 */
    CLK_PRSC_1024   = 5, /*< CPU_CLK (from clk_i top signal) / 1024 */
    CLK_PRSC_2048   = 6, /*< CPU_CLK (from clk_i top signal) / 2048 */
    CLK_PRSC_4096   = 7, /*< CPU_CLK (from clk_i top signal) / 4096 */
};
```



If no peripheral modules requires a clock signal from the internal generator (all available modules disabled by clearing the enable bit in the according module's control register), it 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 \(WDT\)](#)

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 processor registers that actually do provide a hardware reset use an **asynchronous reset**. An asynchronous reset ensures that the entire processor logic is reset to a defined state even if the main clock is not yet operational.

In order to reduce routing constraints (and by this the actual hardware requirements), some *uncritical registers* of the NEORV32 CPU as well as many registers of the entire NEORV32 Processor do not use a dedicated hardware reset. For example there are several pipeline registers and "buffer" registers that do not require a defined initial state to ensure correct operation.



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.

### 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).

#### Trigger Type



The RISC-V standard interrupts are **level-triggered and high-active**. Once set, the signal has to remain high until the interrupt request is explicitly acknowledged (e.g. writing to a memory-mapped register). The RISC-V standard interrupts **CANNOT** be acknowledged/cleared by writing zero to the according [mip](#) CSR bit.

### 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](#) - [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	<a href="#">WDT</a>	watchdog timeout interrupt

Channel	Source	Description
1	CFS	custom functions subsystem (CFS) interrupt (user-defined)
2	UART0	Configurable UART0 RX interrupt
3	UART0	Configurable UART0 TX interrupt
4	UART1	Configurable UART1 RX interrupt
5	UART1	Configurable UART1 TX interrupt
6	SPI	Configurable SPI interrupt
7	TWI	TWI transmission done interrupt
8	XIRQ	External interrupt controller interrupt
9	NEOLED	NEOLED TX buffer interrupt
10	-	<i>reserved</i>
11	SDI	Configurable SDI interrupt
12	GPTMR	General purpose timer interrupt
13	ONEWIRE	1-wire operation done interrupt
14	-	<i>reserved</i>
15	-	<i>reserved</i>

*Trigger Type*

The fast interrupt request channels become pending after being triggering by one-cycle-high signal. A pending FIRQ has to be explicitly cleared by writing zero to the according `mip` CSR bit.

## 2.6. Address Space

As a 32-bit architecture the NEORV32 can access a 4GB physical address space. By default, this address space is divided into five main regions:

1. **Instruction address space** for instructions (=code) and constants. A configurable section of this address space can be used by the instruction memory ([Instruction Memory \(IMEM\)](#)).
2. **Data address space** for application runtime data (heap, stack, etc.). A configurable section of this address space can be used by the data memory ([Data Memory \(DMEM\)](#)).
3. **Bootloader address space** for the internal bootloader memory [Bootloader ROM \(BOOTROM\)](#).
4. **On-Chip Debugger address space** for the processor's [On-Chip Debugger \(OCD\)](#).
5. **IO/peripheral address space**: for the processor-internal memory-mapped [Processor-Internal Modules](#).

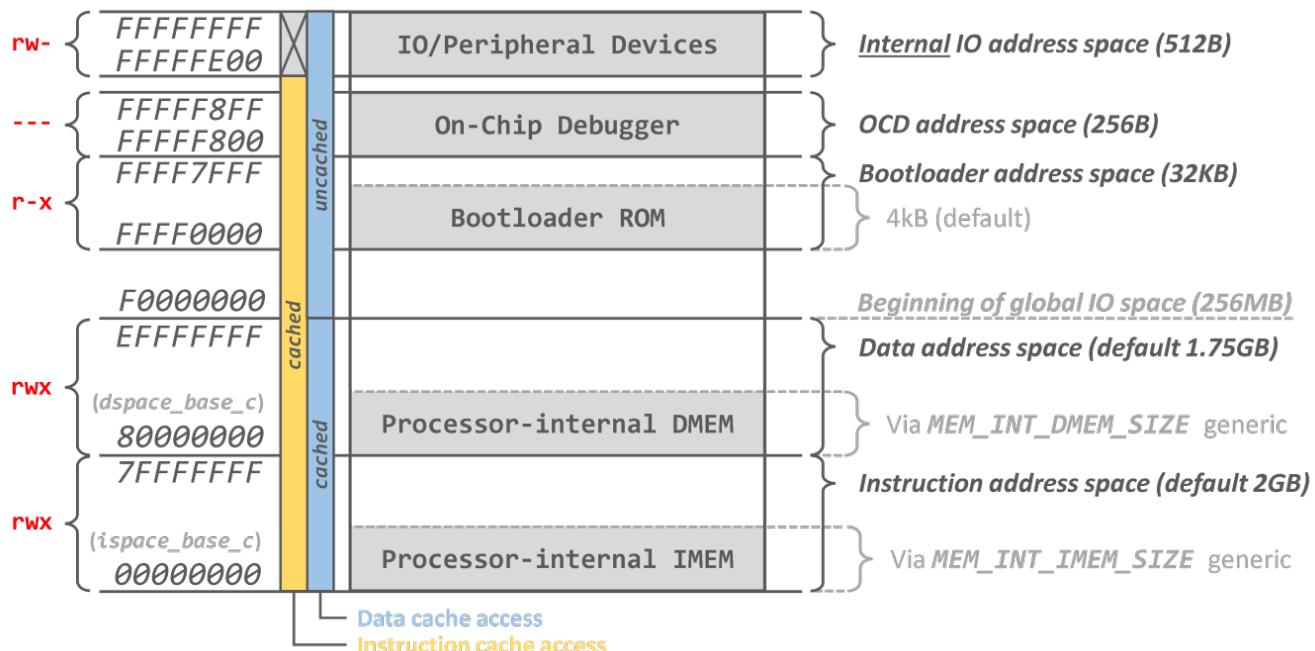


Figure 1. NEORV32 Processor Address Space (Default Configuration)



### RAM Layout

The actual usage of the data address space by the software/executables (stack, heap, ...) is illustrated in section [RAM Layout](#).

### 2.6.1. Physical Memory Attributes (PMAs)

Each default region of the NEORV32 address space provides specific physical memory attributes that define the allowed access types. The according access permission are enforced by the hardware and cannot be changed. If an access violates the PMA's permissions an exception is raised. The access permissions can be further constrained using the CPU's [PMP ISA Extension](#).

- **r** - data read access

- **W** - data write access
- **X** - instruction fetch access ("execute")

#	Region Description	PMAs	Note
1	Instruction address space	<b>R(W)X</b>	Write accesses to the the internal <b>Instruction Memory (IMEM)</b> can be disabled.
2	Data address space	<b>RWX</b>	Code can also be executed from data memory.
3	Bootloader address space	<b>R-X</b>	Read-only memory.
4	On-Chip Debugger address space	<b>---</b>	Not accessible at all by "normal" software - accessible only when the CPU is in <b>CPU Debug Mode</b> .
5	IO/peripheral address space	<b>RW-</b>	Read/write accesses only.

## 2.6.2. CPU Data and Instruction Access

The CPU can access all of the 32-bit address space from the instruction fetch interface (**I**) and also from the data access interface (**D**). These two CPU interfaces are multiplexed by a simple bus switch ([rtl/core/neorv32\\_busswitch.vhd](#)) into a single processor-internal bus. All processor-internal memories, peripherals and also the external memory interface are connected to this bus. Hence, both CPU interfaces (instruction fetch & data access) have access to the same (identical!) address space making the processor a **modified von-Neumann architecture**.

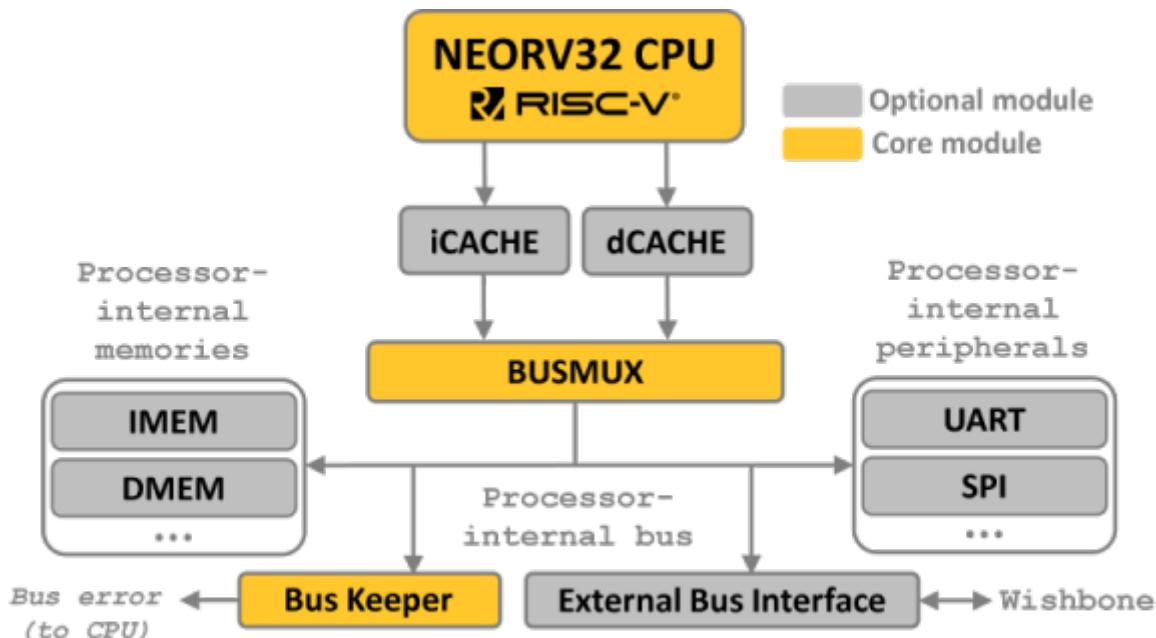


Figure 2. Processor-internal bus architecture



The internal processor bus might appear as bottleneck. In order to reduce traffic jam on this bus (when instruction fetch and data interface access the bus at the same time) the instruction fetch of the CPU is equipped with a prefetch buffer.

Memory accesses can be further buffered using the caches. Furthermore, data accesses (loads and stores) have higher priority than instruction fetch accesses.



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

### 2.6.3. Address Space Layout

The general address space layout consists of two main configuration constants: `ispace_base_c` defining the base address of the *instruction memory address space* and `dspace_base_c` defining the base address of the *data memory address space*. Both constants are defined in the NEORV32 VHDL package file `rtl/core/neorv32_package.vhd`:

```
-- Architecture Configuration -----
```

```
-----  
constant ispace_base_c : std_ulogic_vector(31 downto 0) := x"00000000";  
constant dspace_base_c : std_ulogic_vector(31 downto 0) := x"80000000";
```

The default configuration assumes the *instruction memory address space* starting at address `0x00000000` and the *data memory address space* starting at `0x80000000`. Both values can be modified for a specific setup and the address space may also overlap or can even be identical.

### 2.6.4. Memory Configuration

The NEORV32 Processor was designed to provide maximum flexibility for the actual memory configuration. The processor can populate the *instruction address space* and/or the *data address space* with **internal memories** for instructions (IMEM) and data (DMEM). Processor **external memories** can be used as an alternative or even in combination with the internal ones. The figure below show some exemplary memory configurations.

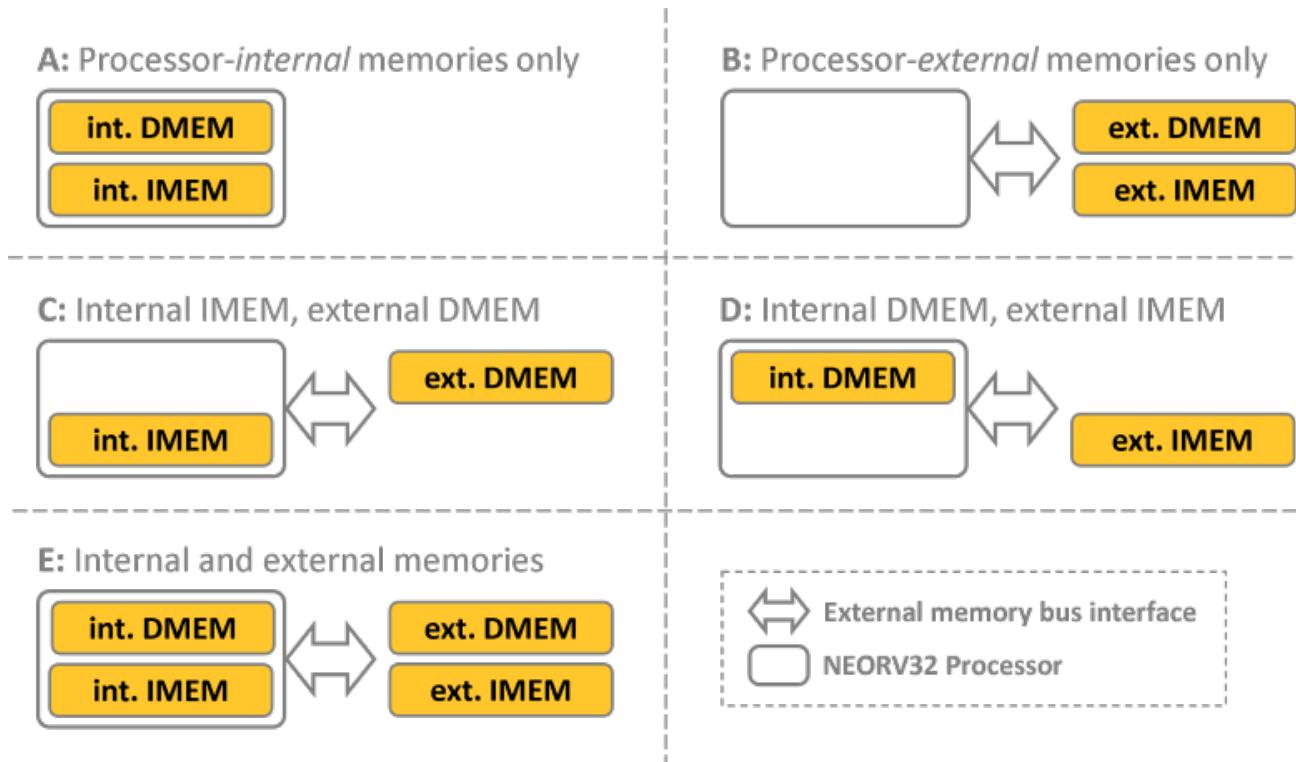


Figure 3. Exemplary Memory Configurations



Any access to "address holes" (addresses that do not belong to a specific module) will raise a bus exception fault due to the missing access acknowledge.

## Internal Memories

If the processor-internal **Instruction Memory (IMEM)** is implemented, it is located right at the base address of the instruction address space (default `ispace_base_c = 0x00000000`). Vice versa, the processor-internal **Data Memory (DMEM)** is located right at the beginning of the data address space (default `dspace_base_c = 0x80000000`) when implemented.

## External Memories

Any CPU access (data or instructions), which does **not** fulfill at least one of the following conditions, is forwarded via the **Processor-External Memory Interface (WISHBONE)** to external components:

- access to the processor-internal **Instruction Memory (IMEM)** and processor-internal IMEM is implemented
- access to the processor-internal **Data Memory (DMEM)** and processor-internal DMEM is implemented
- access to the processor-internal bootloader ROM and beyond



If the **Execute In Place Module (XIP)** is implemented accesses mapped to this module are not forwarded to the external memory interface.



If the **Processor-External Memory Interface (WISHBONE)** is not implemented, any

access exceeding the internal memory address space will raise a bus access fault exception.

## 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.

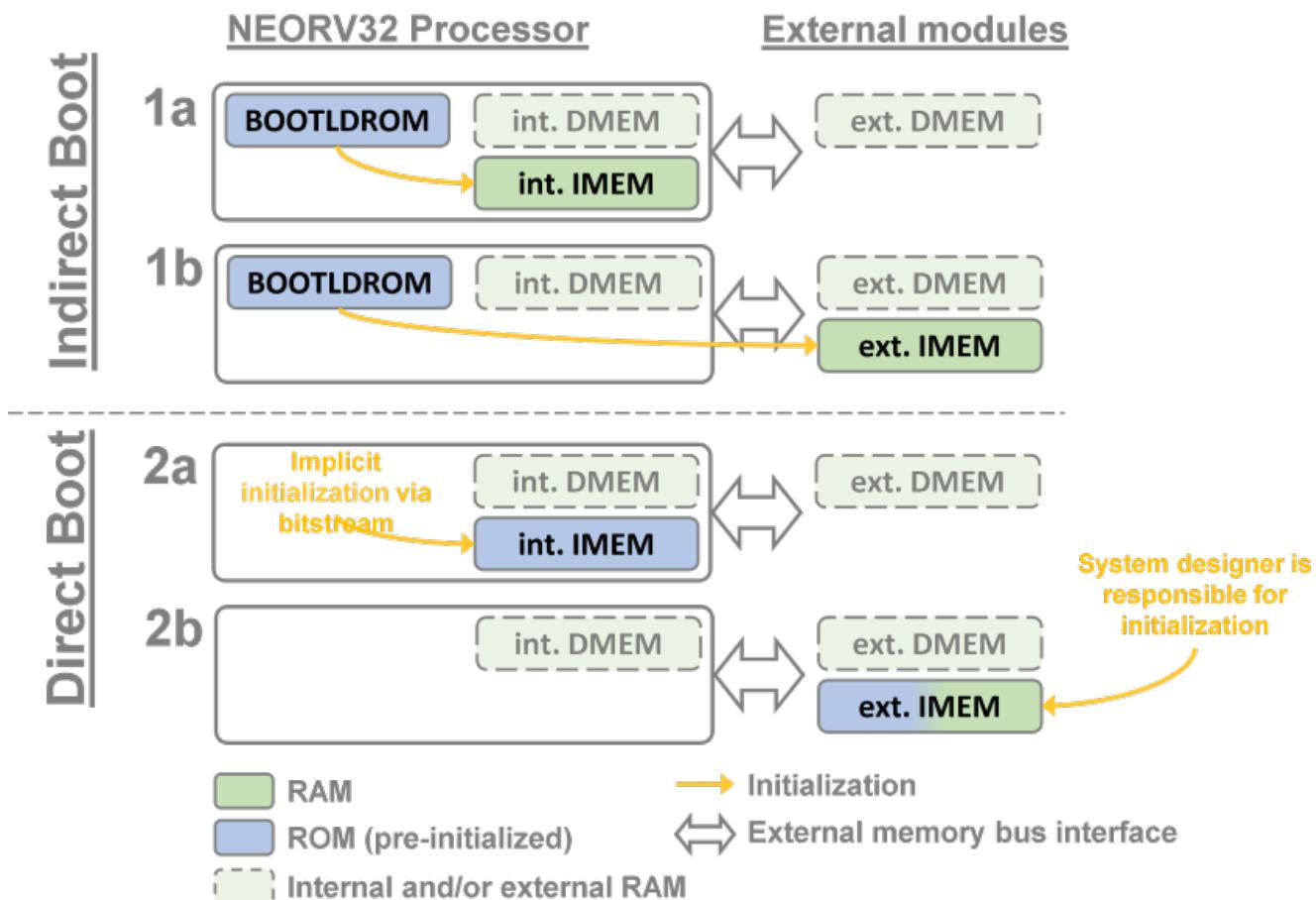


Figure 4. NEORV32 Boot Configurations

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

The NEORV32 processor is a SoC (system-on-chip) consisting of the NEORV32 CPU, peripheral/IO devices, embedded memories, an external memory interface and a bus infrastructure to interconnect all modules.

The processor-internal peripheral/IO devices are located at the end of the 32-bit address space at base address `0xFFFFFE00`. A region of 512 bytes is reserved for these devices. Hence, all peripheral/IO devices are accessed using a memory-mapped scheme.

### *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 can only be written in full-word mode (i.e. 32-bit). Byte or half-word (8/16-bit) write accesses will raise a store access fault exception. Read accesses are not size constrained. Processor-internal memories as well as modules connected to the external memory interface can be written with arbitrary granularity.

### *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 or 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 mem/neorv32_imem.default.vhd	entity-only definition default <i>platform-agnostic</i> memory architecture
Software driver file(s):	none	<i>implicitly used</i>
Top entity port:	none	
Configuration generics:	<code>MEM_INT_IMEM_EN</code> <code>MEM_INT_IMEM_SIZE</code> <code>INT_BOOTLOADER_EN</code>	implement processor-internal IMEM when <code>true</code> IMEM size in bytes use internal bootloader when <code>true</code> (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 via the processor's `MEM_INT_IMEM_EN` generic. The size in bytes is defined via the `MEM_INT_IMEM_SIZE` generic. If the IMEM is implemented, the memory is mapped into the instruction memory space and located right at the beginning of the instruction memory space (default `ispace_base_c` = 0x00000000), see [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 a bootloader that can update the content of the IMEM at any time. If you do not need the bootloader anymore - since your application development has completed and you want the program to permanently reside in the internal instruction memory - the IMEM is automatically implemented as *pre-initialized* ROM 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 will **not be initialized at all**.

### 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 (should) infers embedded memory block. 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.

*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 mem/neorv32_dmem.default.vhd	entity-only definition default <i>platform-agnostic</i> memory architecture
Software driver file(s):	none	<i>implicitly used</i>
Top entity port:	none	
Configuration generics:	<code>MEM_INT_DMEM_EN</code>	implement processor-internal DMEM when <code>true</code>
	<code>MEM_INT_DMEM_SIZE</code>	DMEM size in bytes
CPU interrupts:	none	

Implementation of the processor-internal data memory is enabled via the processor's `MEM_INT_DMEM_EN` generic. The size in bytes is defined via the `MEM_INT_DMEM_SIZE` generic. If the DMEM is implemented, the memory is mapped into the data memory space and located right at the beginning of the data memory space (default `dspace_base_c` = 0x80000000), see [Address Space](#). The DMEM is always implemented as true RAM.

### 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 (should) infers embedded memory block. 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.



### 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.



#### *Bootloader Image*

The boot ROM is initialized during synthesis with the default bootloader image ([rtl/core/neorv32\\_bootloader\\_image.vhd](#)).



#### *Read-Only Access*

Any write access to the BOOTROM will raise a *store access fault* exception.

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

Hardware source file(s):	neorv32_icache.vhd	
Software driver file(s):	none	<i>implicitly used</i>
Top entity port:	none	
Configuration generics:	<b>ICACHE_EN</b> <b>ICACHE_NUM_BLOCKS</b> <b>ICACHE_BLOCK_SIZE</b> <b>ICACHE_ASSOCIATIVITY</b>	implement processor-internal instruction cache when <b>true</b> number of cache blocks (pages/lines) size of a cache block in bytes associativity / number of sets
CPU interrupts:	none	

The processor features an optional instruction cache to improve performance when using memories with high access latencies. The cache is directly connected to the CPU's instruction fetch interface and provides full-transparent buffering of instruction fetch accesses to the **entire address space**.

The cache is implemented if the **ICACHE\_EN** generic is **true**. The size of the cache memory is defined via **ICACHE\_BLOCK\_SIZE** (the size of a single cache block/page/line in bytes; has to be a power of two and greater than or equal to 4 bytes), **ICACHE\_NUM\_BLOCKS** (the total amount of cache blocks; has to be a power of two and greater than or equal to 1) and the actual cache associativity **ICACHE\_ASSOCIATIVITY** (number of sets; 1 = direct-mapped, 2 = 2-way set-associative) generics. If the cache associativity is greater than one the LRU replacement policy (least recently used) is used.



### Caching Internal Memories

The instruction cache is intended to accelerate instruction fetches from **processor-external** memories (via the external bus interface or via the XIP module).



### Manual Cache Clear/Reload

By executing the **fence.i** instruction ([Zifencei ISA Extension](#)) the cache is cleared and a reload from main memory is triggered. This also allows to implement self-modifying code.



### 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_dcache.vhd	
Software driver file(s):	none	<i>implicitly used</i>
Top entity port:	none	
Configuration generics:	<b>DCACHE_EN</b> <b>DCACHE_NUM_BLOCKS</b> <b>DCACHE_BLOCK_SIZE</b>	implement processor-internal data cache when <b>true</b> number of cache blocks (pages/lines) 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 directly connected to the CPU's data access interface and provides full-transparent buffering.

The cache is implemented if the **DCACHE\_EN** generic is **true**. The size of the cache memory is defined via the **DCACHE\_BLOCK\_SIZE** (the size of a single cache block/page/line in bytes; has to be a power of two and greater than or equal to 4 bytes) and **DCACHE\_NUM\_BLOCKS** (the total amount of cache blocks; has to be a power of two and greater than or equal to 1) generics. The data cache provides only a single set, hence it is direct-mapped.

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 **0xF000000** to **0xFFFFFFF** will not be cached at all. This also allows to attach custom IO modules via the processor's external memory interface when they are mapped to upper-most 256 MB address page (see section [Address Space](#)).



### Caching Internal Memories

The data cache is intended to accelerate data access to **processor-external** memories (via the external bus interface or via the XIP module).



### Manual Cache Clear/Reload

By executing the **fence** instruction ([I ISA Extension](#)) 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, a data bus error exception is raised.

## 2.7.6. Processor-External Memory Interface (WISHBONE)

Hardware source file(s):	neorv32_wishbone.vhd	
Software driver file(s):	none	<i>implicitly used</i>
Top entity port:	<code>wb_tag_o</code>	request tag output (3-bit)
	<code>wb_adr_o</code>	address output (32-bit)
	<code>wb_dat_i</code>	data input (32-bit)
	<code>wb_dat_o</code>	data output (32-bit)
	<code>wb_we_o</code>	write enable (1-bit)
	<code>wb_sel_o</code>	byte enable (4-bit)
	<code>wb_stb_o</code>	strobe (1-bit)
	<code>wb_cyc_o</code>	valid cycle (1-bit)
	<code>wb_ack_i</code>	acknowledge (1-bit)
	<code>wb_err_i</code>	bus error (1-bit)
	<code>fence_o</code>	an executed <code>fence</code> instruction
	<code>fencei_o</code>	an executed <code>fence.i</code> instruction
Configuration generics:	<code>MEM_EXT_EN</code>	enable external memory interface when <code>true</code>
	<code>MEM_EXT_TIMEOUT</code>	number of clock cycles after which an unacknowledged external bus access will auto-terminate (0 = disabled)
	<code>MEM_EXT_PIPE_MODE</code>	when <code>false</code> (default): classic/standard Wishbone protocol; when <code>true</code> : pipelined Wishbone protocol
	<code>MEM_EXT_BIG_ENDIAN</code>	byte-order (Endianness) of external memory interface; <code>true</code> =BIG, <code>false</code> =little (default)
	<code>MEM_EXT_ASYNC_RX</code>	use registered RX path when <code>false</code> (default); use async/direct RX path when <code>true</code>
	<code>MEM_EXT_ASYNC_TX_</code>	use registered TX path when <code>false</code> (default); use async/direct TX path when <code>true</code>
CPU interrupts:	none	

The external memory interface provides a Wishbone b4-compatible on-chip bus interface. The bus interface is implemented if the `MEM_EXT_EN` generic is `true`. This interface can be used to attach external memories, custom hardware accelerators, additional IO devices or all other kinds of IP

blocks to the processor.

The external interface is not mapped to a specific address space. Instead, all CPU memory accesses that do not target a processor-internal module are delegated to the external memory interface. In summary, a CPU load/store access is delegated via the external bus interface if...

- the access does not target the internal instruction memory IMEM (if implemented at all)
- **and** the access does not target the internal data memory DMEM (if implemented at all)
- **and** the access does not target the internal bootloader ROM or any of the internal modules - regardless if one or more of these components are actually implemented or not.



#### *Address Space Layout*

See section [Address Space](#) for more information.



#### *Execute-in-Place Module*

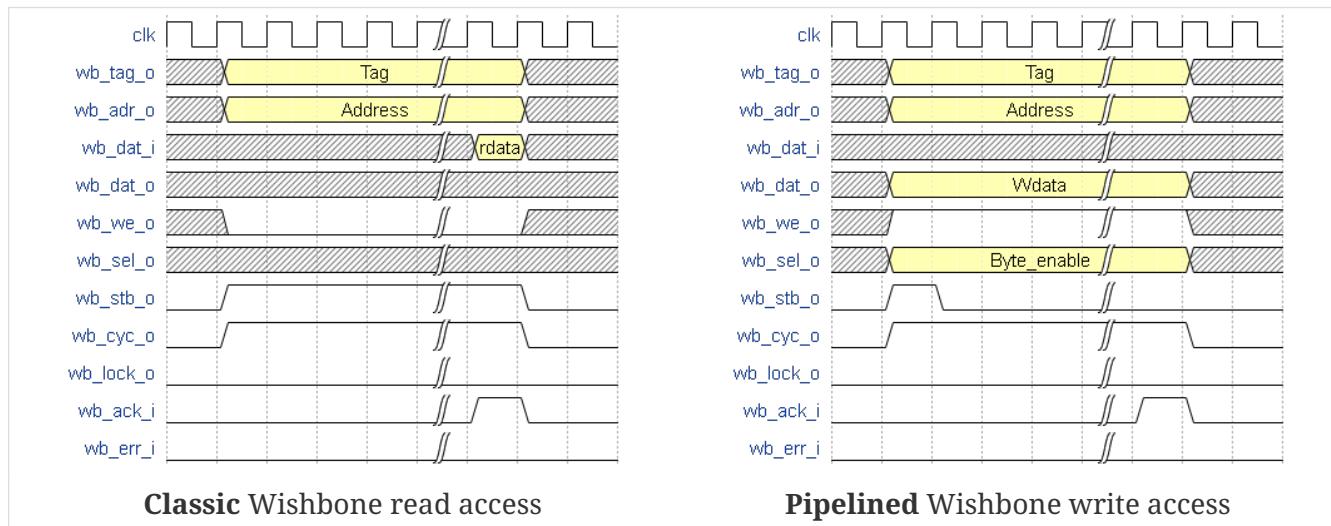
If the Execute In Place module (XIP) is implemented, accesses targeting the XIP memory-mapped-region will not be forwarded to the external memory interface. See section [Execute In Place Module \(XIP\)](#) for more information.

## Wishbone Bus Protocol

The external memory interface either uses the **standard** (also called "classic") Wishbone protocol (default) or **pipelined** Wishbone protocol. The protocol to be used is configured via the `MEM_EXT_PIPE_MODE` generic:

- If `MEM_EXT_PIPE_MODE` is `false`, all bus control signals including `wb_stb_o` are active and remain stable until the transfer is acknowledged/terminated.
- If `MEM_EXT_PIPE_MODE` is `true`, all bus control except `wb_stb_o` are active and remain until the transfer is acknowledged/terminated. In this case, `wb_stb_o` is asserted only during the very first bus clock cycle.

*Table 7. Exemplary Wishbone bus accesses using "classic" and "pipelined" protocol*





### 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.

## Bus Access

The NEORV32 Wishbone gateway does not support burst transfer yet, so there is always just a single transfer "in fly". Hence, the Wishbone **STALL** signal is not implemented. An accessed 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 **MEM\_EXT\_TIMEOUT** generic that defines the maximum time (in clock cycles) a bus access can be pending before it is automatically terminated with an error condition. If **MEM\_EXT\_TIMEOUT** is set to zero, the timeout is disabled and a bus access can take an arbitrary number of cycles to complete (this is not recommended!).

When **MEM\_EXT\_TIMEOUT** is greater than zero, the Wishbone gateway starts an internal countdown whenever the CPU accesses an address via the external memory interface. If the accessed device does not acknowledge (via **wb\_ack\_i**) or terminate (via **wb\_err\_i**) the transfer within **MEM\_EXT\_TIMEOUT** clock cycles, the bus access is automatically canceled setting **wb\_cyc\_o** low again and a CPU load/store/instruction fetch bus access fault exception is raised.

## Wishbone Tag

The 3-bit wishbone **wb\_tag\_o** signal provides additional information regarding the access type:

- **wb\_tag\_o(0)** 1: privileged access (CPU is in machine mode); 0: unprivileged access (CPU is not in machine mode)
- **wb\_tag\_o(1)** always zero (indicating "secure access")
- **wb\_tag\_o(2)** 1: instruction fetch access, 0: data access

## Endianness

The NEORV32 CPU and the Processor setup are **little-endian** architectures. To allow direct connection to a big-endian memory system the external bus interface provides an Endianness configuration. The Endianness of the external memory interface can be configured via the **MEM\_EXT\_BIG\_ENDIAN** generic. By default, the external memory interface uses little-endian byte-order.

Application software can check the Endianness configuration of the external bus interface via the **SYSINFO** module (see section [System Configuration Information Memory \(SYSINFO\)](#) for more information).

## Access Latency

By default, the Wishbone gateway introduces two additional latency cycles: processor-outgoing (**\*\_o**) and processor-incoming (**\*\_i**) signals are fully registered. Thus, any access from the CPU to a

processor-external devices via Wishbone requires 2 additional clock cycles. This can ease timing closure when using large (combinatorial) Wishbone interconnection networks.

Optionally, the latency of the Wishbone gateway can be reduced by removing the input and output register stages. Enabling the `MEM_EXT_ASYNC_RX` option will remove the input register stage; enabling `MEM_EXT_ASYNC_TX` option will remove the output register stages. Each enabled option reduces access latency by 1 cycle.

#### *Output Gating*

All outgoing Wishbone signals use a "gating mechanism" so they only change if there is a actual Wishbone transaction being in progress. This can reduce dynamic switching activity in the external bus system and also simplifies simulation-based inspection of the Wishbone transactions. Note that this output gating is only available if the output register buffer is not disabled (`MEM_EXT_ASYNC_TX = false`).



## 2.7.7. Internal Bus Monitor (BUSKEEPER)

---

Hardware source file(s):	neorv32_buskeeper.vhd
Software driver file(s):	none
Top entity port:	none
Configuration generics:	none
Package constants:	<code>max_proc_int_response_time_c</code> Access time window (maximum number of cycles)
CPU interrupts:	none

---

### Theory of Operation

The Bus Keeper is a fundamental component of the processor's internal bus system that ensures correct bus operations while maintaining execution safety. It monitors every single bus transactions that is initiated by the CPU. If an accessed device responds with an error condition or do not respond at all within a specific *access time window*, an according bus access fault exception is raised. The following exceptions can be raised by the Bus Keeper (see section [Traps, Exceptions and Interrupts](#) for a list of all available bus access-related exceptions):

- `TRAP_CODE_I_ACCESS`: error during instruction fetch bus access
- `TRAP_CODE_S_ACCESS`: error during data store bus access
- `TRAP_CODE_L_ACCESS`: error during data load bus access

The **access time window**, in which an accessed device has to respond, is defined by the `max_proc_int_response_time_c` constant from the processor's VHDL package file ([rtl/neorv32\\_package.vhd](#)). The default value is **15 clock cycles**.

In case of a bus access fault exception application software can evaluate the Bus Keeper's control register `CTRL` to retrieve further details regarding the bus exception. The `BUSKEEPER_ERR_FLAG` bit indicates that an actual bus access fault has occurred. The bit is sticky once set and is automatically cleared when reading or writing the `NEORV32_BUSKEEPER CTRL` register. The `BUSKEEPER_ERR_TYPE` bit defines the type of the bus fault:

- **0** - "Device Error": The bus access exception was cause by the memory-mapped device that has been accessed (the device asserted its `err_o`).
- **1** - "Timeout Error": The bus access exception was caused by the Bus Keeper because the accessed memory-mapped device did not respond within the access time window. Note that this error type can also be raised by the optional timeout feature of the external bus interface.

### Register Map

Table 8. BUSKEEPER register map (`struct NEORV32_BUSKEEPER`)

Address	Name [C]	Bit(s), Name [C]	R/W	Function
0xffffffff78	CTRL	0 BUSKEEPER_ERR_TYPE	r/-	Bus error type, valid if <i>BUSKEEPER_ERR_FLAG</i>
		31 BUSKEEPER_ERR_FLAG	r/c	Sticky error flag, clears after read or write access
0xffffffff7c	-	<i>reserved</i>	r/c	<i>reserved</i> (mirrored access to <a href="#">CTRL</a> )

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

Hardware source file(s):	neorv32_gpio.vhd
Software driver file(s):	neorv32_gpio.c 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 9. GPIO unit register map (`struct NEORV32_GPIO`)

Address	Name [C]	Bit(s)	R/W	Function
<code>0xffffffffc0</code>	<code>INPUT_LO</code>	31:0	r/-	parallel input port pins 31:0
<code>0xffffffffc4</code>	<code>INPUT_HI</code>	31:0	r/-	parallel input port pins 63:32
<code>0xffffffffc8</code>	<code>OUTPUT_LO</code>	31:0	r/w	parallel output port pins 31:0
<code>0xffffffffcc</code>	<code>OUTPUT_HI</code>	31:0	r/w	parallel output port pins 63:32

## 2.7.9. Watchdog Timer (WDT)

Hardware source file(s):	neorv32_wdt.vhd	
Software driver file(s):	neorv32_wdt.c neorv32_wdt.h	
Top entity port:	none	
Configuration generics:	<code>IO_WDT_EN</code>	implement watchdog when <code>true</code>
CPU interrupts:	fast IRQ channel 0	watchdog timeout (see <a href="#">Processor Interrupts</a> )

### 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 interrupt and no system reset can be triggered.

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. In order to inform the application of an imminent timeout, an optional CPU interrupt is triggered when the timeout counter reaches *half* of the programmed timeout value.

The watchdog is "fed" by writing 1 to the `WDT_CTRL_RESET` control register bit, which will reset the internal timeout counter back to zero.



#### Forced Reset

Writing all-zero to the `WDT_CTRL_TIMEOUT` bits will immediately trigger a system-wide reset.



#### Watchdog Interrupt

A watchdog interrupt occurs when the watchdog is enabled and the internal counter reaches *exactly* half of the programmed timeout value. Hence, the interrupt only fires once. However, a triggered WDT interrupt has to be explicitly cleared by writing zero to the according `mip` CSR bit.



#### 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"). Read accesses to the control register as well as watchdog resets (by setting the `WDT_CTRL_RESET` flag) are not affected.

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.

## Cause of last Hardware Reset

The cause of the last system hardware reset can be determined via the `WDT_CTRL_RCAUSE` flag. If this flag is cleared, the processor has been reset via the external reset signal (or the on-chip debugger). If this flag is set, the last system reset was caused by the watchdog itself.

## Register Map

Table 10. WDT register map (`struct NEORV32_WDT`)

Address	Name [C]	Bit(s), Name [C]	R/W	Reset value	Writable if locked	Function
0xffffffffbc	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_RESET	-/w	-	yes	reset watchdog when set, auto-clears
		5 WDT_CTRL_RCAUSE	r/-	0	-	cause of last system reset: 0=caused by external reset signal, 1=caused by watchdog
		7:6 -	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

## 2.7.10. 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
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 MTIME machine system timer that is compatible to the RISC-V privileged specifications. The 64-bit system time is accessed via the memory-mapped `TIME_LO` and `TIME_HI` registers. A 64-bit time compare register, which is accessible via the memory-mapped `TIMECMP_LO` and `TIMECMP_HI` registers, can be used to configure the CPU's `MTI` (machine timer interrupt). 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 `TIMECMP` again (either by modifying `TIME` or `TIMECMP`).



### 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 memory-mapped register).

## Register Map

Table 11. MTIME register map (`struct NEORV32_MTIME`)

Address	Name [C]	Bits	R/W	Function
<code>0xffffffff90</code>	<code>TIME_LO</code>	31:0	r/w	machine system time, low word
<code>0xffffffff94</code>	<code>TIME_HI</code>	31:0	r/w	machine system time, high word
<code>0xffffffff98</code>	<code>TIMECMP_LO</code>	31:0	r/w	time compare, low word
<code>0xffffffff9c</code>	<code>TIMECMP_HI</code>	31:0	r/w	time compare, high word

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

Hardware source file(s):	neorv32_uart.vhd	
Software driver file(s):	neorv32_uart.c 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 12. `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 data to the lowest byte of 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 by reading the `DATA` register or by disabling the module.

## 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. Furthermore, a pending UART interrupt has to be explicitly cleared again by writing zero to the according `mip` CSR bit.



### 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.

1. 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.
2. Data written to `DATA[31:0]` will be dumped as 8-chars ASCII hexadecimal value to a file named `nerv32 uart0 sim mode data.out`.

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

## Register Map

Table 13. *UART0 register map (struct NEORV32\_UART0)*

Address	Name [C]	Bit(s), Name [C]	R/W	Function
0xffffffffa0	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 : UART_CTRL_PRSC0	r/w	Baud rate clock prescaler select
		15:6 UART_CTRL_BAUD9 : 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 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
		31 UART_CTRL_TX_BUSY	r/-	TX busy or TX FIFO not empty

Address	Name [C]	Bit(s), Name [C]	R/W	Function
0xffffffffa4	DATA	7:0 UART_DATA_RTX_MSB : UART_DATA_RTX_LSB	r/w	receive/transmit data
		11:8 UART_DATA_RX_FIFO_SIZE_MS B : UART_DATA_RX_FIFO_SIZE_LS B	r/-	log2(RX FIFO size)
		15:12 UART_DATA_TX_FIFO_SIZE_MS B : UART_DATA_TX_FIFO_SIZE_LS B	r/-	log2(TX FIFO size)
		31:16	r/-	<i>reserved,</i> read as zero
				31:0

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

Hardware source file(s):	neorv32_uart.vhd		
Software driver file(s):	neorv32_uart.c 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 files: `neorv32_uart1.sim_mode.text.out` is used to dump plain ASCII text. This data is also printed to the simulator console. `neorv32_uart1.sim_mode.data.out` is used to dump full 32-bit hexadecimal ASCII-chars data words.

### Register Map

Table 14. UART1 register map (`struct NEORV32_UART1`)

Address	Name [C]	Bit(s), Name [C]	R/W	Function
<code>0xfffffff0</code>	<code>CTRL</code>	...	...	Same as UART0
<code>0xffffffd4</code>	<code>DATA</code>	...	...	Same as UART0

## 2.7.13. Serial Peripheral Interface Controller (SPI)

Hardware source file(s):	neorv32_spi.vhd	
Software driver file(s):	neorv32_spi.c	
	neorv32_spi.h	
Top entity port:	<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. A 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 its 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*.

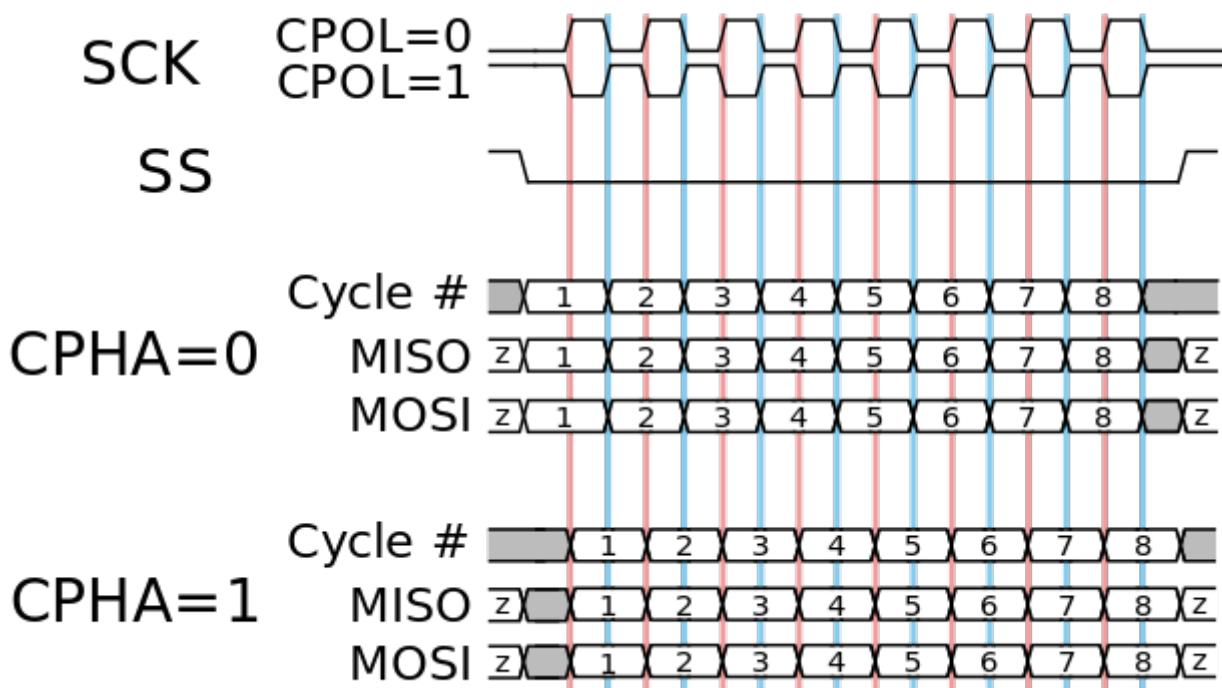


Figure 5. SPI clock modes; image from [https://en.wikipedia.org/wiki/File:SPI\\_timing\\_diagram2.svg](https://en.wikipedia.org/wiki/File:SPI_timing_diagram2.svg) (license: (Wikimedia) Creative Commons Attribution-Share Alike 3.0 Unported)

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 15. 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%.

## 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. Furthermore, an active SPI interrupt has to be explicitly cleared again by writing zero to the according `mip` CSR bit.

## Register Map

Table 16. SPI register map (`struct NEORV32_SPI`)

Address	Name [C]	Bit(s), Name [C]	R/W	Function
<code>0xffffffa8</code>	<code>CTRL</code>	<code>0 SPI_CTRL_EN</code>	r/w	SPI module enable
		<code>1 SPI_CTRL_CPHA</code>	r/w	clock phase
		<code>2 SPI_CTRL_CPOL</code>	r/w	clock polarity
		<code>5:3 SPI_CTRL_CS_SEL2 : SPI_CTRL_CS_SEL0</code>	r/w	Direct chip-select 0..7
		<code>6 SPI_CTRL_CS_EN</code>	r/w	Direct chip-select enable: setting <code>spi_csn_o(SPI_CTRL_CS_SEL)</code> low when set
		<code>9:7 SPI_CTRL_PRSC2 : SPI_CTRL_PRSC0</code>	r/w	3-bit clock prescaler select
		<code>13:10 SPI_CTRL_CDIV2 : SPI_CTRL_CDIV0</code>	r/w	4-bit clock divider
		<code>15:14 reserved</code>	r/-	reserved, read as zero
		<code>16 SPI_CTRL_RX_AVAIL</code>	r/-	RX FIFO data available (RX FIFO not empty)
		<code>17 SPI_CTRL_TX_EMPTY</code>	r/-	TX FIFO empty
		<code>18 SPI_CTRL_TX_NHALF</code>	r/-	TX FIFO <i>not</i> at least half full
		<code>19 SPI_CTRL_TX_FULL</code>	r/-	TX FIFO full
		<code>20 SPI_CTRL_IRQ_RX_AVAIL</code>	r/w	Trigger IRQ if RX FIFO not empty
		<code>21 SPI_CTRL_IRQ_TX_EMPTY</code>	r/w	Trigger IRQ if TX FIFO empty
		<code>22 SPI_CTRL_IRQ_TX_NHALF</code>	r/w	Trigger IRQ if TX FIFO <i>not</i> at least half full
		<code>26:23 SPI_CTRL_FIFO_MSB : SPI_CTRL_FIFO_LSB</code>	r/-	FIFO depth; $\log_2(\text{IO\_SPI\_FIFO})$
		<code>30:27 `reserved_</code>	r/-	reserved, read as zero
		<code>31 SPI_CTRL_BUSY</code>	r/-	SPI module busy when set (serial engine operation in progress and TX FIFO not empty yet)
<code>0xfffffac</code>	<code>DATA</code>	<code>7:0</code>	r/w	receive/transmit data (FIFO)

## 2.7.14. Serial Data Interface Controller (SDI)

Hardware source file(s):	neorv32_sdi.vhd	
Software driver file(s):	neorv32_sdi.c neorv32_sdi.h	
Top entity port:	<code>sdi_clk_i</code> <code>sdi_dat_o</code> <code>sdi_dat_i</code> <code>sdi_csn_i</code>	1-bit serial clock input 1-bit serial data output 1-bit serial data input 1-bit chip-select input (low-active)
Configuration generics:	<code>IO_SDI_EN</code> <code>IO_SDI_FIFO</code>	implement SDI controller when <code>true</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.

#### *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. Furthermore, an active SDI interrupt has to be explicitly cleared again by writing zero to the according `mip` CSR bit.

## Register Map

Table 17. SDI register map (`struct NEORV32_SDI`)

Address	Name [C]	Bit(s), Name [C]	R/W	Function
0xffffffff0	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 : 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
		22:19 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_FULL	r/-	TX FIFO full
		31:28 reserved	r/-	reserved, read as zero
0xffffffff4	DATA	7:0	r/w	receive/transmit data (FIFO)

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

Hardware source file(s):	neorv32_twi.vhd	
Software driver file(s):	neorv32_twi.c	
	neorv32_twi.h	
Top entity port:	<code>twi_sda_i</code>	1-bit serial data line sense input
	<code>twi_sda_o</code>	1-bit serial data line output (pull low only)
	<code>twi_scl_i</code>	1-bit serial clock line sense input
	<code>twi_scl_o</code>	1-bit serial clock line output (pull low only)
Configuration generics:	<code>IO_TWI_EN</code>	implement TWI controller when <code>true</code>
CPU interrupts:	fast IRQ channel 7	transmission done interrupt (see <a href="#">Processor Interrupts</a> )

### Overview

The NEORV32 TWI implements a **TWI controller**. Currently, **no multi-controller support** is available. Furthermore, the NEORV32 TWI unit cannot operate in peripheral mode.



The serial clock (SCL) and the serial data (SDA) lines can only be actively driven low by the controller. Hence, external pull-up resistors are required for these lines.

### Tri-State Drivers

The TWI module requires two tri-state drivers (actually: open-drain) for the SDA and SCL lines, which have to be implemented in the top module of the setup. A generic VHDL example is given below (`sda` and `scl` are the actual TWI bus signal, which are of type `std_logic`).

*Listing 2. TWI VHDL tri-state driver example*

```

sda      <= '0' when (twi_sda_o = '0') else 'Z'; -- drive
scl      <= '0' when (twi_scl_o = '0') else 'Z'; -- drive
twi_sda_i <= std_ulogic(sda); -- sense
twi_scl_i <= std_ulogic(scl); -- sense

```

### TWI Clock Speed

The TWI clock frequency is programmed by the 3-bit `TWI_CTRL_PRSCx` clock prescaler for a coarse selection and a 4-bit clock divider `TWI_CTRL_CDIVx` for a fine selection.

*Table 18. 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 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:

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

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%. However, an accessed peripheral can "slow down" the bus clock by using **clock stretching** (= actively driving the SCL line low). The controller will pause operation in this case if clock stretching is enabled via the `TWI_CTRL_CSEN` bit of the unit's control register `CTRL`.

## TWI Transfers

The TWI is enabled via the `TWI_CTRL_EN` bit in the `CTRL` control register. The user program can start / stop a transmission by issuing a START or STOP condition. These conditions are generated by setting the according bits (`TWI_CTRL_START` or `TWI_CTRL_STOP`) in the control register.

Data is transferred via the TWI bus by writing a byte to the `DATA` register. The written byte is send via the TWI bus and the received byte from the bus is also available in this register after the transmission is completed.

The TWI operation (transmitting data or performing a START or STOP condition) is in progress as long as the control register's `TWI_CTRL_BUSY` bit is set.



A transmission can be terminated at any time by disabling the TWI module by clearing the `TWI_CTRL_EN` control register bit. This will also reset the whole module.



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 ACK/NACK and MACK

An accessed TWI peripheral has to acknowledge each transferred byte. When the `TWI_CTRL_ACK` bit is set after a completed transmission the accessed peripheral has send an ACKNOWLEDGE (ACK). If this bit is cleared after a completed transmission, the peripheral has send a NOT-ACKNOWLEDGE (NACK).

The NEORV32 TWI controller can also send an ACK generated by itself ("controller acknowledge 'MACK'") right after transmitting a byte by driving SDA low during the ACK time slot. Some TWI modules require this MACK to acknowledge certain data movement operations.

The control register's `TWI_CTRL_MACK` bit has to be set to make the TWI module automatically

generate a MACK after the byte transmission has been completed. If this bit is cleared, the ACK/NACK generated by the peripheral is sampled in this time slot instead (normal mode).

## TWI Bus Status

The TWI controller can check if the TWI bus is currently claimed (SCL and SDA both low). The bus can be claimed by the NEORV32 TWI itself or by any other controller. Bit `TWI_CTRL_CLAIME` of the control register will be set if the bus is currently claimed.

## TWI Interrupt

The TWI module provides a single interrupt to signal "transmission done" to the CPU. Whenever the TWI module completes the current transmission of one byte the interrupt is triggered. Note the the interrupt is **not** triggered when completing a START or STOP condition. Once triggered, the interrupt has to be explicitly cleared again by writing zero to the according `mip` CSR bit.

## Register Map

Table 19. TWI register map (`struct NEORV32_TWI`)

Address	Name [C]	Bit(s), Name [C]	R/W	Function
0xffffffffb0	CTRL	0 <code>TWI_CTRL_EN</code>	r/w	TWI enable, reset if cleared
		1 <code>TWI_CTRL_START</code>	-/w	generate START condition, auto-clears
		2 <code>TWI_CTRL_STOP</code>	-/w	generate STOP condition, auto-clears
		3 <code>TWI_CTRL_MACK</code>	r/w	generate controller-ACK for each transmission ("MACK")
		4 <code>TWI_CTRL_CSEN</code>	r/w	allow clock stretching when set
		7:5 <code>TWI_CTRL_PRSC2 : TWI_CTRL_PRSC0</code>	r/w	3-bit clock prescaler select
		11:8 <code>TWI_CTRL_CDIV3 : TWI_CTRL_CDIV0</code>	r/w	4-bit clock divider
		28:12 -	r/-	<i>reserved</i> , read as zero
		29 <code>TWI_CTRL CLAIMED</code>	r/-	set if the TWI bus is claimed by any controller
		30 <code>TWI_CTRL_ACK</code>	r/-	ACK received when set, NACK received when cleared
31 <code>TWI_CTRL_BU SY</code>	r/-	transfer/START/STOP in progress when set	0xffff ffffb4	DATA

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

Hardware source file(s):	neorv32_onewire.vhd
Software driver file(s):	neorv32_onewire.c neorv32_onewire.h
Top entity port:	<b>onewire_i</b> 1-bit 1-wire bus sense input <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 3. 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 20. 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.

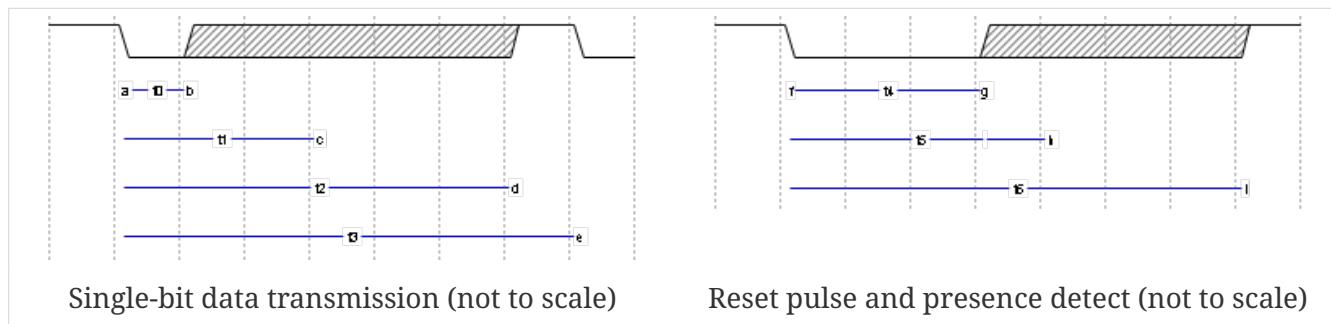


Table 21. Data Transmission Timing

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 4. 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 "operation done" condition to the CPU. Whenever the controller completes a "generate reset pulse", a "transfer single-bit" or a "transfer full-byte" operation the interrupt is triggered. Once triggered, the interrupt has to be *explicitly* cleared again by writing zero to the according `mip` CSR FIRQ bit.

## Register Map

*Table 22. ONEWIRE register map (`struct NEORV32_ONEWIRE`)*

Address	Name [C]	Bit(s), Name [C]	R/W	Function
0xffffffff70	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
0xffffffff74	DATA	7:0 ONEWIRE_DATA_MSB : ONEWIRE_DATA_LSB	r/w	receive/transmit data (8-bit)

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

Hardware source file(s):	neorv32_pwm.vhd
Software driver file(s):	neorv32_pwm.c 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 23. 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 base frequency is defined by:

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

## Register Map

Table 24. PWM register map (`struct nerv32_pwm_t`)

Address	Name	Bit(s), Name [C]	R/W	Function
0xffffffff50	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
0xffffffff54	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
0xffffffff58	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
0xffffffff5c	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

## 2.7.18. 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. Instead of using a pseudo RNG like a LFSR, the TRNG uses a simple, straight-forward ring oscillator concept as physical entropy source. Hence, voltage, thermal and also semiconductor manufacturing fluctuations are used to provide a true 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. It is based on the **neoTRNG V2**, which is a "spin-off project" of the NEORV32 processor. More detailed information about the neoTRNG, its architecture and a detailed evaluation of the random number quality can be found in the neoTRNG repository: <https://github.com/stnolting/neoTRNG>

#### *Inferring Latches*



The synthesis tool might emit a warning like "inferring latches for ... neorv32\_trng ...". This is no problem as this is what we actually want: the TRNG is based on latches, which implement the inverters of the ring oscillators.

#### *Simulation*



When simulating the processor the NEORV32 TRNG is automatically set to "simulation mode". In this mode, the physical entropy sources (= the ring oscillators) are replaced by a simple **pseudo RNG (LFSR)** providing weak pseudo-random data only. 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 RND 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.



#### *Data Gating*

The TRNG data bits `TRNG_CTRL_DATA_MSB : TRNG_CTRL_DATA_MSB` are set to zero if `TRNG_CTRL_VALID` is low. This prevents a random byte being read twice.

## Register Map

Table 25. TRNG register map (`struct NEORV32_TRNG`)

Address	Name e [C]	Bit(s), Name [C]	R/W	Function
<code>0xffffffffb8</code>	<code>CTRL</code>	<code>7:0 TRNG_CTRL_DATA_MSB : TRNG_CTRL_DATA_MSB</code>	r/-	8-bit random data
		<code>28 TRNG_CTRL_FIFO_CLR</code>	-/w	flush random data FIFO when set (auto-clears)
		<code>29 TRNG_CTRL_SIM_MODE</code>	r/-	simulation mode (PRNG!)
		<code>30 TRNG_CTRL_EN</code>	r/w	TRNG enable
		<code>31 TRNG_CTRL_VALID</code>	r/-	random data is valid when set

## 2.7.19. Custom Functions Subsystem (CFS)

Hardware source file(s):	neorv32_cfs.vhd	
Software driver file(s):	neorv32_cfs.c neorv32_cfs.h	
Top entity port:	<code>cfs_in_i</code> <code>cfs_out_o</code>	custom input conduit custom output conduit
Configuration generics:	<code>IO_CFS_EN</code> <code>IO_CFS_CONFIG</code> <code>IO_CFS_IN_SIZE</code> <code>IO_CFS_OUT_SIZE</code>	implement CFS when <code>true</code> custom generic conduit size of <code>cfs_in_i</code> 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 [Zxfc 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 5. 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. Once triggered, the interrupt becomes pending (if enabled in the `mis` CSR) and has to be explicitly cleared again by writing zero to the according `mip` CSR bit. See section [Processor Interrupts](#) for more information.

## CFS Configuration Generic

By default, the CFS provides a single 32-bit `std::vector<uint32_t>` 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 26. CFS register map (struct `NEORV32_CFS`)*

Address	Name [C]	Bit(s)	R/W	Function
<code>0xfffffe00</code>	<code>REG[0]</code>	<code>31:0</code>	(r)/(w)	custom CFS register 0
<code>0xfffffe04</code>	<code>REG[1]</code>	<code>31:0</code>	(r)/(w)	custom CFS register 1
...	...	<code>31:0</code>	(r)/(w)	...
<code>0xfffffef8</code>	<code>REG[62]</code>	<code>31:0</code>	(r)/(w)	custom CFS register 62
<code>0xfffffec</code>	<code>REG[63]</code>	<code>31:0</code>	(r)/(w)	custom CFS register 63

## 2.7.20. 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.

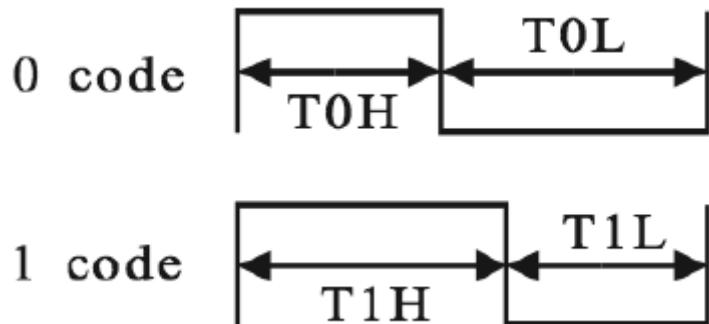


Figure 6. WS2812 bit-level protocol - taken from the "Adafruit NeoPixel™ Überguide"

Table 27. 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 28. NEOLED Prescaler Configuration

<code>NEOLED_CTRL_PRSCx</code>	0b000	0b001	0b010	0b011	0b100	0b101	0b110	0b111
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  $f_{main} = 100 \text{ MHz}$
- `NEOLED_CTRL_PRSCx = 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:

- `NEOLED_CTRL_T_TOT = 0b11110` (= decimal 30)
- `NEOLED_CTRL_T_ZERO_H = 0b01010` (= decimal 10)
- `NEOLED_CTRL_T_ONE_H = 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 (`nerv32_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 been triggered and became pending, it has to explicitly cleared again by writing zero to according `mip` CSR bit.

### Register Map

Table 29. NEOLED register map (`struct NEORV32_NEOLED`)

Address	Name [C]	Bit(s), Name [C]	R/W	Function
0xfffffd8	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
0xfffffdc	DATA	31:0 / 23:0	-/w	TX data (32- or 24-bit, depending on <i>NEOLED_CTRL_MODE</i> bit)

## 2.7.21. External Interrupt Controller (XIRQ)

Hardware source file(s):	neorv32_xirq.vhd	
Software driver file(s):	neorv32_xirq.c	
	neorv32_xirq.h	
Top entity port:	xirq_i	External interrupts input (32-bit)
Configuration generics:	XIRQ_NUM_CH	Number of external IRQ channels to implement (0..32)
	XIRQ_TRIGGER_TYPE	IRQ trigger type configuration
	XIRQ_TRIGGER_POLARITY	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 30. XIRQ Trigger Configuration

XIRQ_TRIGGER_TYPE(i)	XIRQ_TRIGGER_POLARITY(i)	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 `[_macuse]` 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 CPU FIRQ by clearing the according `mip` CSR bit \* clear the pending XIRQ channel by clearing the according `EIP` bit \* writing *any* value to `ESC` to acknowledge the XIRQ interrupt

## Register Map

Table 31. XIRQ register map (`struct NEORV32_XIRQ`)

Address	Name [C]	Bit(s)	R/W	Description
<code>0xffffffff80</code>	<code>EIE</code>	<code>31:0</code>	r/w	External interrupt enable register (one bit per channel, LSB-aligned)
<code>0xffffffff84</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>0xffffffff88</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>0xffffffff8c</code>	-	<code>31:0</code>	r/-	<i>reserved</i> , read as zero

## 2.7.22. General Purpose Timer (GPTMR)

Hardware source file(s):	neorv32_gptmr.vhd	
Software driver file(s):	neorv32_gptmr.c	
	neorv32_gptmr.h	
Top entity port:	none	
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> )

### Theory of Operation

The general purpose timer module provides a simple yet universal 32-bit timer. The timer is implemented if `IO_GPTMR_EN` top generic is set `true`. It provides a 32-bit counter register (`COUNT`) and a 32-bit threshold register (`THRES`). An interrupt is generated whenever the value of the counter registers matches the one from threshold register.

The timer is enabled by setting the `GPTMR_CTRL_EN` bit in the device's control register `CTRL`. 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 32. 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

The timer provides two operation modes that are configured via the `GPTMR_CTRL_MODE` control register bit: .If `GPTMR_CTRL_MODE` is cleared (0) the timer operates in *single-shot mode*. As soon as `COUNT` matches `THRES` an interrupt request is generated and the timer stops operation (i.e. it stops incrementing) .If `GPTMR_CTRL_MODE` is set (1) the timer operates in *continuous mode*. When `COUNT` matches `THRES` an interrupt request is generated and `COUNT` is automatically reset to all-zero before continuing to increment.



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

### Timer Interrupt

The GPTMR interrupt is triggered when the timer is enabled and `COUNT` matches `THRES`. The interrupt remains pending inside the CPU until it explicitly cleared by writing zero to the according `mip` CSR bit.

### Register Map

Table 33. GPTMR register map (`struct NEORV32_GPTMR`)

Address	Name [C]	Bit(s), Name [C]	R/W	Function
0xffffffff60	CTRL	0 GPTMR_CTRL_EN	r/w	Timer enable flag
		3:1 GPTMR_CTRL_PRSC2 : GPTMR_CTRL_PRSC0	r/w	3-bit clock prescaler select
		4 GPTMR_CTRL_MODE	r/w	Counter mode: 0=single-shot, 1=continuous
		31:5 -	r/-	<i>reserved</i> , read as zero
0xffffffff64	THRES	31:0	r/w	Threshold value register
0xffffffff68	COUNT	31:0	r/w	Counter register

## 2.7.23. Execute In Place Module (XIP)

Hardware source file(s):	neorv32_xip.vhd	
Software driver file(s):	neorv32_xip.c neorv32_xip.h	
Top entity port:	<code>xip_csn_o</code>	1-bit chip select, low-active
	<code>xip_clk_o</code>	1-bit serial clock output
	<code>xip_dat_i</code>	1-bit serial data input
	<code>xip_dat_o</code>	1-bit serial data output
Configuration generics:	<code>IO_XIP_EN</code>	implement XIP module when <code>true</code>
CPU interrupts:	none	

### Overview

The execute in-place (XIP) module allows to execute code (and read constant data) directly from a SPI flash memory. Hence, it uses the standard serial peripheral interface (SPI) as transfer protocol under the hood.

The XIP flash is not mapped to a specific region of the processor's address space. Instead, it provides a programmable mapping scheme to allow a flexible user-defined mapping of the flash to *any section* of the address space.

From the CPU side, the module provides two different interfaces: one for transparently accessing the XIP flash and another one for accessing the module's control and status registers. The first interface provides a *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 data accesses to the XIP module's configuration registers.



#### XIP Example Program

An example program is provided in `sw/example/demo_xip` that illustrates how to program and configure an external SPI flash to run a program from it.

### SPI Protocol

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 clock speed of the interface (`xip_clk_o`) is defined by a three-bit clock pre-scaler configured using the `XIP_CTRL_PRSCx` bits:

Table 34. XIP prescaler configuration

XIP_CTRL_PRSCx	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 XIP SPI clock frequency  $f_{XIP}$  is derived from the processor's main clock  $f_{main}$  and is determined by:

$$f_{XIP} = f_{main}[\text{Hz}] / (2 * \text{clock\_prescaler})$$

Hence, the maximum XIP clock speed is  $f_{main} / 4$ .

#### High-Speed SPI mode



The module provides a "high-speed" SPI mode. In this mode the clock prescaler configuration (`XIP_CTRL_PRSCx`) is ignored and the SPI clock operates at  $f_{main} / 2$  (half of the processor's main clock). High speed SPI mode is enabled by setting the control register's `XIP_CTRL_HIGHSPEED` bit.

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.

### 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_HI` and `DATA_LO` 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.

## Address Mapping

The address mapping of the XIP flash is not fixed by design. It can be mapped to *any section* within the processor's address space. A *section* refers to one out of 16 naturally aligned 256MB wide memory segments. This segment is defined by the four most significant bits of the address (31:28) and the XIP's segment is programmed by the four `XIP_CTRL_XIP_PAGE` bits in the unit's control register. All accesses within this page will be mapped to the XIP flash.



Care must be taken when programming the page mapping to prevent access collisions with other modules (like internal memories or modules attached to the external memory interface).

## 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. 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 programmed XIP memory segment are ignored by the module and might be forwarded to the processor's external memory interface (if implemented) or will cause a bus 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 raise a bus exception.



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

## XIP Burst Mode

By default, every XIP access to the flash transmits the read command and the word-aligned address before reading four consecutive data bytes. Obviously, this introduces a certain transmission overhead. To reduce this overhead, the XIP mode allows to utilize the flash's *incremental read* function, which will return consecutive bytes when continuing to send clock cycles after a read command. Hence, the XIP module provides an optional "burst mode" to accelerate consecutive read accesses.

The XIP burst mode is enabled by setting the `XIP_CTRL_BURST_EN` bit in the module's control register. The burst mode only affects the actual XIP mode and *not* the direct SPI mode. Hence, it should be enabled right before enabling XIP mode only. By using the XIP burst mode flash read accesses can be accelerated by up to 50%.

## Register Map

Table 35. 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
		24:21 XIP_CTRL_XIP_PAGE_MSB : XIP_CTRL_XIP_PAGE_LSB	r/w	XIP memory page
		25 XIP_CTRL_SPI_CSEN	r/w	Allow SPI chip-select to be actually asserted when set
		26 XIP_CTRL_HIGHSPEED	r/w	enable SPI high-speed mode (ignoring XIP_CTRL_PRSC)
		27 XIP_CTRL_BURST_EN	r/w	Enable XIP burst mode
		29:28 -	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

## 2.7.24. System Configuration Information Memory (SYSINFO)

Hardware source file(s):	neorv32_sysinfo.vhd
Software driver file(s):	neorv32.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 speed) for correct operation.



Any write access to the SYSINFO module will raise a store bus error exception. The [Internal Bus Monitor \(BUSKEEPER\)](#) will signal a "DEVICE ERROR" in this case.

### Register Map

Table 36. SYSINFO register map (`struct NEORV32_SYSINFO`)

Address	Name [C]	Function
<code>0xffffffe0</code>	<code>CLK</code>	clock speed in Hz (via top's <code>CLOCK_FREQUENCY</code> generic)
<code>0xffffffe4</code>	<code>`CUSTOM_ID</code>	custom user-defined ID (via top's <code>CUSTOM_ID</code> generic)
<code>0xffffffe8</code>	<code>SOC</code>	specific SoC configuration (see <code>`sysinfo_soc_configuration&gt;&gt;</code> )
<code>0xfffffec</code>	<code>CACHE</code>	cache configuration information (see <a href="#">SYSINFO - Cache Configuration</a> )
<code>0xffffffff0</code>	<code>ISPACE_BASE</code>	instruction address space base (via package's <code>ispace_base_c</code> constant)
<code>0xffffffff4</code>	<code>IMEM_SIZE</code>	internal IMEM size in bytes (via top's <code>MEM_INT_IMEM_SIZE</code> generic)
<code>0xffffffff8</code>	<code>DSPACE_BASE</code>	data address space base (via package's <code>sdspace_base_c</code> constant)
<code>0xffffffc</code>	<code>DMEM_SIZE</code>	internal DMEM size in bytes (via top's <code>MEM_INT_DMEM_SIZE</code> generic)

## SYSINFO - SoC Configuration

Table 37. SYSINFO\_SOC bits

Bit	Name [C]	Function
0	SYSINFO_SOC_BOOTLOADER	set if the processor-internal bootloader is implemented (via top's <code>INT_BOOTLOADER_EN</code> generic)
1	SYSINFO_SOC_MEM_EXT	set if the external Wishbone bus interface is implemented (via top's <code>MEM_EXT_EN</code> generic)
2	SYSINFO_SOC_MEM_INT_IMEM	set if the processor-internal DMEM implemented (via top's <code>MEM_INT_DMEM_EN</code> generic)
3	SYSINFO_SOC_MEM_INT_DMEM	set if the processor-internal IMEM is implemented (via top's <code>MEM_INT_IMEM_EN</code> generic)
4	SYSINFO_SOC_MEM_EXT_ENDIAN	set if external bus interface uses BIG-endian byte-order (via top's <code>MEM_EXT_BIG_ENDIAN</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)
12: 7	-	<i>reserved</i> , read as zero
13	SYSINFO_SOC_IS_SIM	set if processor is being <b>simulated</b> (not guaranteed)
14	SYSINFO_SOC_OCD	set if on-chip debugger implemented (via top's <code>ON_CHIP_DEBUGGER_EN</code> generic)
15	-	<i>reserved</i> , read as zero
16	SYSINFO_SOC_IO_GPIO	set if the GPIO is implemented (via top's <code>IO_GPIO_EN</code> generic)
17	SYSINFO_SOC_IO_MTIME	set if the MTIME is implemented (via top's <code>IO_MTIME_EN</code> generic)
18	SYSINFO_SOC_IO_UART0	set if the primary UART0 is implemented (via top's <code>IO_UART0_EN</code> generic)
19	SYSINFO_SOC_IO_SPI	set if the SPI is implemented (via top's <code>IO_SPI_EN</code> generic)
20	SYSINFO_SOC_IO_TWI	set if the TWI is implemented (via top's <code>IO_TWI_EN</code> generic)
21	SYSINFO_SOC_IO_PWM	set if the PWM is implemented (via top's <code>IO_PWM_NUM_CH</code> generic)

Bit	Name [C]	Function
22	SYSINFO_SOC_IO_WDT	set if the WDT is implemented (via top's <code>IO_WDT_EN</code> generic)
23	SYSINFO_SOC_IO_CFS	set if the custom functions subsystem is implemented (via top's <code>IO_CFS_EN</code> generic)
24	SYSINFO_SOC_IO_TRNG	set if the TRNG is implemented (via top's <code>IO_TRNG_EN</code> generic)
25	SYSINFO_SOC_IO_SDI	set if the SDI is implemented (via top's <code>IO_SDI_EN</code> generic)
26	SYSINFO_SOC_IO_UART1	set if the secondary UART1 is implemented (via top's <code>IO_UART1_EN</code> generic)
27	SYSINFO_SOC_IO_NEOLED	set if the NEOLED is implemented (via top's <code>IO_NEOLED_EN</code> generic)
28	SYSINFO_SOC_IO_XIRQ	set if the XIRQ is implemented (via top's <code>XIRQ_NUM_CH</code> generic)
29	SYSINFO_SOC_IO_GPTMR	set if the GPTMR is implemented (via top's <code>IO_GPTMR_EN</code> generic)
30	SYSINFO_SOC_IO_XIP	set if the XIP module is implemented (via top's <code>IO_XIP_EN</code> generic)
31	SYSINFO_SOC_IO_ONewire	set if the ONEWIRE interface is implemented (via top's <code>IO_ONewire_EN</code> generic)

## SYSINFO - Cache Configuration



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

Table 38. SYSINFO\_CACHE bits

Bit	Name [C]	Function
3:0	SYSINFO_CACHE_IC_BLOCK_SIZE_3 : SYSINFO_CACHE_IC_BLOCK_SIZE_0	$\log_2$ (i-cache block size in bytes), via top's <code>ICACHE_BLOCK_SIZE</code> generic
7:4	SYSINFO_CACHE_IC_NUM_BLOCKS_3 : SYSINFO_CACHE_IC_NUM_BLOCKS_0	$\log_2$ (i-cache number of cache blocks), via top's <code>ICACHE_NUM_BLOCKS</code> generic
11: 9	SYSINFO_CACHE_IC_ASSOCIATIVITY_3 : SYSINFO_CACHE_IC_ASSOCIATIVITY_0	$\log_2$ (i-cache associativity), via top's <code>ICACHE_ASSOCIATIVITY</code> generic
15: 12	SYSINFO_CACHE_IC_REPLACEMENT_3 : SYSINFO_CACHE_IC_REPLACEMENT_0	i-cache replacement policy ( <code>0001</code> = LRU if associativity > 0)
19: 16	SYSINFO_CACHE_DC_BLOCK_SIZE_3 : SYSINFO_CACHE_DC_BLOCK_SIZE_0	$\log_2$ (d-cache block size in bytes), via top's <code>DCACHE_BLOCK_SIZE</code> generic

Bit	Name [C]	Function
23: 20	SYSINFO_CACHE_DC_NUM_BLOCKS_3 : SYSINFO_CACHE_DC_NUM_BLOCKS_0	$\log_2(\text{d-cache number of cache blocks})$ , via top's DCACHE_NUM_BLOCKS generic
27: 24	SYSINFO_CACHE_DC_ASSOCIATIVITY_3 : SYSINFO_CACHE_DC_ASSOCIATIVITY_0	always zero
31: 28	SYSINFO_CACHE_DC_REPLACEMENT_3 : SYSINFO_CACHE_DC_REPLACEMENT_0	always zero

# Chapter 3. NEORV32 Central Processing Unit (CPU)

The NEORV32 CPU is an area-optimized RISC-V core implementing the `rv32iZicsr` base ISA and supporting several 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

- [Architecture, Full Virtualization and RISC-V Compatibility](#)
- [CPU Top Entity - Signals and CPU Top Entity - Generics](#)
- [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 (→ [Full Virtualization](#)).

## Incompatibility Issues and Limitations



### *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 \(MTIME\)](#).

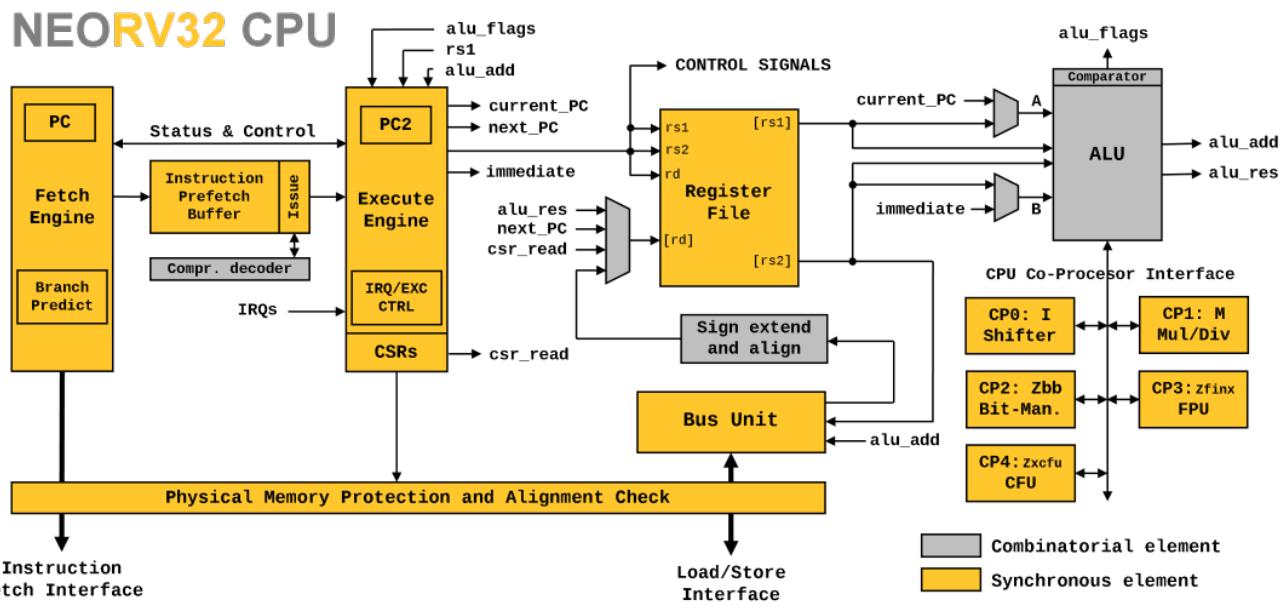


### *No Hardware Support of Misaligned Memory Accesses*

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.

## 3.2. 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 (see section [CPU Data and Instruction Access](#)).



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).

### 3.2.1. CPU Register File

The data register file contains the general purpose “x” architecture registers. For the **rv32i** ISA there are 32 32-bit registers and for the **rv32e** ISA there are 16 32-bit registers. Register zero (**x0/zero**) always read as zero and any write access to it is discarded.

The register file is implemented as synchronous memory with synchronous read and write accesses. Register `zero` is also mapped to a *physical memory location* in the register file. 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. Furthermore, the whole register file can be mapped entirely to FPGA block RAM.

The memory of the register file 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) or 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.

### 3.2.2. CPU Arithmetic Logic Unit

The arithmetic/logic unit (ALU) is used for processing data from the register file and also for memory and branch address computations. All simple [I ISA Extension](#) processing operations (`add`, `and`, ...) are implemented as combinatorial logic requiring only a single cycle to complete. More sophisticated instructions (shift operations from the base ISA and all further ISA extensions) 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](#)).

Once triggered, the selected co-processor is required to complete processing within a bound time window. Otherwise the co-processor operation is terminated by the hardware and an illegal instruction exception is raised. The time window is  $2^T$  clock cycles wide;  $T$  is defined by the `cp_timeout_c` VHDL package constant (default = 7 → 128 cycles).

### 3.2.3. 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 [PMP 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.

### 3.2.4. 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). The depth of this FIFO can be configured by the `CPU_IPB_ENTRIES` top generic.

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.

#### Branch Prediction



The front-end implements a very simple branch prediction that **stops** fetching further instruction while a branch/jump/call operation is in progress.

#### 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.

## 3.3. 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` instruction. When in sleep mode, all CPU-internal operations are stopped (execution, instruction fetch, counter increments, ...). However, this does not affect the operation of any peripheral/IO modules like interfaces and timers. Furthermore, the CPU will continue to buffer/enqueue incoming interrupt requests. The CPU will leave sleep mode as soon as any *enabled* interrupt source becomes *pending*.



If sleep mode is entered without at least one enabled interrupt source the CPU will be *permanently* halted.



The CPU automatically wakes up from sleep mode if a debug session is started via the on-chip debugger. `wfi` behaves as a simple `nop` when the CPU is *in* debug-mode or during single-stepping.

## 3.4. 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. 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 39. NEORV32 CPU Signal List*

Signal	Width	Dir	Description
<b>Global Signals</b>			
<code>clk_i</code>	1	in	Global clock line, all registers triggering on rising edge
<code>rstn_i</code>	1	in	Global reset, low-active
<code>sleep_o</code>	1	out	CPU is in sleep mode when set
<code>debug_o</code>	1	out	CPU is in debug mode when set
<b>Instruction Bus Interface</b>			
<code>i_bus_addr_o</code>	32	out	Access address
<code>i_bus_rdata_i</code>	32	in	Read data
<code>i_bus_re_o</code>	1	out	Read request (one-shot) trigger
<code>i_bus_ack_i</code>	1	in	Bus transfer acknowledge from accessed peripheral
<code>i_bus_err_i</code>	1	in	Bus transfer terminate from accessed peripheral
<code>i_bus_fence_o</code>	1	out	Indicates an executed <code>fence.i</code> instruction
<code>i_bus_priv_o</code>	1	out	Current <i>effective</i> CPU privilege level ( <code>0</code> = user, <code>1</code> = machine)
<b>Data Bus Interface</b>			
<code>d_bus_addr_o</code>	32	out	Access address
<code>d_bus_rdata_i</code>	32	in	Read data
<code>d_bus_wdata_o</code>	32	out	Write data
<code>d_bus_ben_o</code>	4	out	Byte enable
<code>d_bus_we_o</code>	1	out	Write request (one-shot) trigger
<code>d_bus_re_o</code>	1	out	Read request (one-shot) trigger
<code>d_bus_ack_i</code>	1	in	Bus transfer acknowledge from accessed peripheral
<code>d_bus_err_i</code>	1	in	Bus transfer terminate from accessed peripheral
<code>d_bus_fence_o</code>	1	out	Indicates an executed <code>fence</code> instruction
<code>d_bus_priv_o</code>	1	out	Current <i>effective</i> CPU privilege level ( <code>0</code> = user, <code>1</code> = machine)
<b>Interrupts (Traps, Exceptions and Interrupts)</b>			

Signal	Width	Dir	Description
<code>msw_irq_i</code>	1	in	RISC-V machine software interrupt
<code>mext_irq_i</code>	1	in	RISC-V machine external interrupt
<code>mtime_irq_i</code>	1	in	RISC-V machine timer interrupt
<code>firq_i</code>	16	in	Custom fast interrupt request signals
<code>db_halt_req_i</code>	1	in	Request CPU to halt and enter debug mode (RISC-V On-Chip Debugger (OCD))

*Bus Interface Protocol*

See section [Bus Interface](#) for the instruction fetch and data access interface protocol.

## 3.6. 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 40. NEORV32 CPU-Exclusive Generic List*

Name	Type	Description
<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> .
<code>CPU_DEBUG_EXC_ADDR</code>	suv(31:0)	"Exception" entry address for the <a href="#">On-Chip Debugger (OCD)</a> .
<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> .

## 3.7. 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.

### *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", which are also available 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](#)).

### *ISA Extensions-Specific CSRs*



The [Control and Status Registers \(CSRs\)](#) section lists the according ISA extensions for all CSRs.

### 3.7.1. I ISA Extension

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

*Table 41. Instructions and Timing*

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

### *fence Instruction*



CPU-internally, the `fence` instruction does not perform any operation inside the CPU. It only sets the top's `d_bus_fence_o` signal high for one cycle to inform the memory system a `fence` instruction has been executed. Any flags within the `fence`

instruction word are ignore by the hardware. However, the `d_bus_fence_o` signal is connected to the [Processor-Internal Data Cache \(dCACHE\)](#). Hence, executing the `fence` instruction will clear/flush the data cache and resynchronize it with main memory.

### 3.7.2. B ISA Extension

The `B` ISA extension adds instructions for bit-manipulation operations. The NEORV32 `B` ISA extension includes the following sub-extensions:

- `Zba` - Address-generation instructions
- `Zbb` - Basic bit-manipulation instructions
- `Zbc` - Carry-less multiplication instructions
- `Zbs` - Single-bit instructions

*Table 42. Instructions and Timing*

Class	Instructions	Execution cycles
Arithmetic/logic	<code>min[u] max[u] sext.b sext.h andn orn xnor</code> <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 + <i>shift_amount</i> ; FAST_SHIFT: 4
Shifted-add	<code>sh1add sh2add sh3add</code>	4
Single-bit	<code>sbset[i] sbclr[i] sbinv[i] sbext[i]</code>	4
Carry-less multiply	<code>clmul clmulh clmuls</code>	36

### 3.7.3. C ISA Extension

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

*Table 43. Instructions and Timing*

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

Class	Instructions	Execution cycles
System	c.break	3

### 3.7.4. 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.

### 3.7.5. M ISA Extension

Hardware-accelerated integer multiplication and division operations are available via the RISC-V M ISA extension.

*Table 44. Instructions and Timing*

Class	Instructions	Execution cycles
Multiplication	mul mulh mulhsu mulhu	36; FAST_MUL: 4
Division	div divu rem remu	36

### 3.7.6. 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](#).

### 3.7.7. 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](#).

### 3.7.8. Zifencei ISA Extension

The [Zifencei](#) CPU extension allows manual synchronization of the instruction stream. The [fence.i](#) instruction resets the CPU's front-end (instruction fetch) and flushes the prefetch buffer. This allows a clean re-fetch of modified instructions from memory. Also, the top's [i\\_bus\\_fencei\\_o](#) signal is set

high for one cycle to inform the memory system (like the [Processor-Internal Instruction Cache \(iCACHE\)](#)) to perform a flush/reload. Any additional flags within the `fence.i` instruction word are ignored by the hardware.

*Table 45. Instructions and Timing*

Class	Instructions	Execution cycles
Instruction fence	<code>fence.i</code>	5

### 3.7.9. 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](#).



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 numbers ("de-normalized" numbers) are not supported by the NEORV32 FPU. Subnormal numbers (exponent = 0) are *flushed to zero* setting them to +/- 0 before entering the FPU's processing core. If a computational instruction (like `fmul.s`) generates a subnormal result, the result is also flushed to zero during normalization.



The [Zfinx](#) extension is not yet officially ratified, but is expected to stay unchanged. There is no software support for the [Zfinx](#) extension in the upstream GCC RISC-V port yet. However, an intrinsic library is provided to utilize the provided [Zfinx](#) floating-point extension from C-language code (see [sw/example/floating\\_point\\_test](#)).

*Table 46. Instructions and Timing*

Class	Instructions	Execution cycles
Arithmetric	<code>fadd.s</code>	110
Arithmetric	<code>fsub.s</code>	112
Arithmetric	<code>fmul.s</code>	22
Compare	<code>fmin.s fmax.s feq.s flt.s fle.s</code>	13
Conversion	<code>fcvt.w.s fcvt.wu.s fcvt.s.w fcvt.s.wu</code>	48
Misc	<code>fsgnj.s fsgnjn.s fsgnjx.s fclass.s</code>	12

### 3.7.10. 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.

### 3.7.11. Zicond ISA Extension

The **Zicond** ISA extension implements conditional operations ("move or set-to-zero").



The **Zicond** extension is neither ratified nor frozen yet. An intrinsic library is provided to utilize the **Zicond** conditional operations from C-language code (see [sw/example/zicond\\_test](#)).

*Table 47. Instructions and Timing*

Class	Instructions	Execution cycles
Conditional operations	<code>czero.eqz</code> <code>czero.nez</code>	3

### 3.7.12. 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).



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

*Table 48. Instructions and Timing*

Class	Instructions	Execution cycles
System	<code>csrrw[i]</code> <code>csrrs[i]</code> <code>csrrc[i]</code>	3
System	<code>wfi</code>	3
System	<code>mret</code> <code>dret</code>	5

### 3.7.13. *Zihpm* ISA Extension

In addition to the base counters the NEORV32 CPU provides up to 29 hardware performance monitors (HPM 3..31), 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.

### 3.7.14. *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. This extension requires only ~50% of the hardware utilization of the "full" [M](#) extension.

### 3.7.15. *Zxcfu* ISA Extension

The [Zxcfu](#) 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 more 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 *intrinsic*, which are basically inline assembly functions that behave like regular C functions but that evaluate to a single custom instruction word (not calling overhead at all).

### 3.7.16. *PMP* 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 partly 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](#).



#### *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 to preserve memory access protection.

### 3.7.17. **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.

### 3.7.18. **Sdtrig** ISA Extension

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

## 3.8. Custom Functions Unit (CFU)

The Custom Functions Unit is the central part of the [Zxcfu ISA Extension](#) and represents the actual hardware module, which 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 the 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](#).

### 3.8.1. CFU Instruction Formats

The custom instructions executed by the CFU utilize a specific opcode space in the [rv32](#) 32-bit instruction 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 different 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 instruction A
- [custom-3: 1111011](#) NEORV32-specific, used for CFU R5-type instruction B

### 3.8.2. 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.

Example operation: `rd ← rs1 xnor rs2`

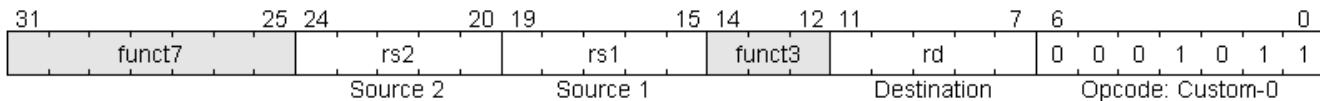


Figure 7. CFU R3-type instruction format

- **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-bit + 3-bit = 10 bit → 1024 different values).

### 3.8.3. 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.

Example operation: `rd ← (rs1 * rs2 + rs3)[31:0]`

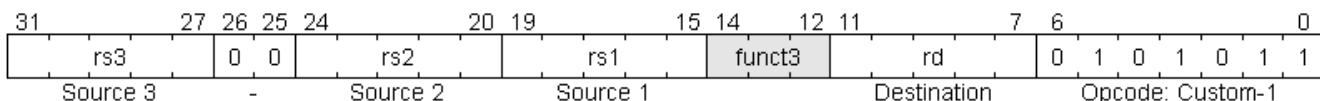


Figure 8. CFU R4-type instruction format

- **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).

## 3.8.4. 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 out of the box.

Example operation: `rd ← rs1 & rs2 & rs3 & rs4`

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

Figure 9. CFU R5-type instruction A format

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

Figure 10. 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)



### RS4 bit field

The `rs4` bit-field is split into two instruction word fields `rs4.hi` and `rs4.lo`. This allows a simple decoding logic as the location of the remaining register fields is identical to other R-type instructions.



### *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.

### 3.8.5. 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.

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

*Listing 6. CFU instruction prototypes*

```
neorv32_cfu_r3_instr(func7, 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 when executing all exemplary CFU instruction types:

*Listing 7. 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);
```

*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](#).

### 3.8.6. 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](#).

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 Example & More Details*

The default CFU hardware module already implements some exemplary instructions that are used for illustration by the CFU example program. See the CFU's VHDL source file ([rtl/core/neorv32\\_cpu\\_cp\\_cfu.vhd](#)), which is highly commented to explain the available signals and the handshake with 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 additional read ports to the core's register file increasing resource requirements.

*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" signal within a bound time window. For example this can be used to signal invalid configurations/operations to the runtime environment. See the CFU's

VHDL file for more information.

## 3.9. 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 "AC" column shows the minimal required privilege level 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 and Excluded CSRs*



All CSR bits that are unused / not implemented / not shown are *hardwired to zero*. All CSRs that are not implemented, not supported or disabled will raise an illegal instruction exception when being accessed.

### *WARL Behavior*



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.

### *Illegal CSR Accesses*



Any illegal read access to a CSR will return zero in the operation's destination register.

Table 49. NEORV32 Control and Status Registers (CSRs)

Address	Name [ASM]	Name [C]	AC	Function
<b>Floating-Point CSRs</b>				
0x001	<b>fflags</b>	<b>CSR_FFLAGS</b>	UR W	Floating-point accrued exceptions
0x002	<b>frm</b>	<b>CSR_FRM</b>	UR W	Floating-point dynamic rounding mode
0x003	<b>fcsr</b>	<b>CSR_FCSR</b>	UR W	Floating-point control and status ( <b>frm</b> + <b>fflags</b> )
<b>Machine Configuration CSRs</b>				
0x30A	<b>menvcfg</b>	<b>CSR_MENVCFG</b>	M R W	Machine environment configuration register - low word
0x31A	<b>menvcfgh</b>	<b>CSR_MENVCFGH</b>	M R W	Machine environment configuration register - low word
<b>Machine Trap Setup CSRs</b>				

Address	Name [ASM]	Name [C]	AC C	Function
0x300	<code>mstatus</code>	<code>CSR_MSTATUS</code>	M R W	Machine status register - low word
0x301	<code>misa</code>	<code>CSR_MISA</code>	M R W	Machine CPU ISA and extensions
0x304	<code>mie</code>	<code>CSR_MIE</code>	M R W	Machine interrupt enable register
0x305	<code>mtvec</code>	<code>CSR_MTVEC</code>	M R W	Machine trap-handler base address for ALL traps
0x306	<code>mcounteren</code>	<code>CSR_MCOUNTEREN</code>	M R W	Machine counter-enable register
0x310	<code>mstatush</code>	<code>CSR_MSTATUSH</code>	M R W	Machine status register - high word

### Machine Trap Handling CSRs

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

### Machine Physical Memory Protection CSRs

0x3A0 .. 0x3AF	<code>pmpcfg0 .. pmpcfg15</code>	<code>CSR_PMPCFG0 .. CSR_PMPCFG15</code>	M R W	Physical memory protection configuration for region 0..15
----------------	----------------------------------	--	-------------	---

Address	Name [ASM]	Name [C]	AC C	Function
0x3B0 .. 0x3EF	pmpaddr0 .. pmpaddr63	CSR_PMPADDR0 .. CSR_PMPADDR63	M R W	Physical memory protection address register region 0..15

**Trigger Module CSRs**

0x7A0	tselect	CSR_TSELECT	M R W	Trigger select register
0x7A1	tdata1	CSR_TDATA1	M R W	Trigger data register 1
0x7A2	tdata2	CSR_TDATA2	M R W	Trigger data register 2
0x7A3	tdata3	CSR_TDATA3	M R W	Trigger data register 3
0x7A4	tinfo	CSR_TINFO	M R W	Trigger information register
0x7A5	tcontrol	CSR_TCONTROL	M R W	Trigger control register
0x7A8	mcontext	CSR_MCONTEXT	M R W	Machine context register
0x7AA	scontext	CSR_SCONTEXT	M R W	Supervisor context register

**CPU Debug Mode CSRs**

0x7B0	dcsr	-	DR W	Debug control and status register
0x7B1	dpc	-	DR W	Debug program counter
0x7B2	dscratch0	-	DR W	Debug scratch register 0

**(Machine) Counter and Timer CSRs**

Address	Name [ASM]	Name [C]	AC C	Function
0xB00	<code>mcycle</code>	<code>CSR_MCYCLE</code>	M R W	Machine cycle counter low word
0xB02	<code>minstret</code>	<code>CSR_MINSTRET</code>	M R W	Machine instruction-retired counter low word
0xB80	<code>mcycleh</code>	<code>CSR_MCYCLEH</code>	M R W	Machine cycle counter high word
0xB82	<code>minstreth</code>	<code>CSR_MINSTRETH</code>	M R W	Machine instruction-retired counter high word
0xC00	<code>cycle</code>	<code>CSR_CYCLE</code>	UR O	Cycle counter low word
0xC02	<code>instret</code>	<code>CSR_INSTRET</code>	UR O	Instruction-retired counter low word
0xC80	<code>cycleh</code>	<code>CSR_CYCLEH</code>	UR O	Cycle counter high word
0xC82	<code>instreth</code>	<code>CSR_INSTRETH</code>	UR O	Instruction-retired counter high word

**Hardware Performance Monitors (HPM) CSRs**

0x323 .. 0x33F	<code>mhpmevent3 .. mhpmevent31</code>	<code>CSR_MHPMEVENT3 .. CSR_MHPMEVENT31</code>	M R W	Machine performance-monitoring event select for counter 3..31
0xB03 .. 0xB1F	<code>mhpmcOUNTER3 .. mhpmcOUNTER31</code>	<code>CSR_MHPMCOUNTER3 .. CSR_MHPMCOUNTER31H</code>	M R W	Machine performance-monitoring counter 3..31 low word
0xB83 .. 0xB9F	<code>mhpmcOUNTER3h .. mhpmcOUNTER31h</code>	<code>CSR_MHPMCOUNTER3H .. CSR_MHPMCOUNTER31H</code>	M R W	Machine performance-monitoring counter 3..31 high word
0xC03 .. 0xC1F	<code>hpmcounter3 .. hpmcounter31</code>	<code>CSR_HPMCOUNTER3 .. CSR_HPMCOUNTER3H</code>	UR O	User performance-monitoring counter 3..31 low word
0xC83 .. 0xC9F	<code>hpmcounter3h .. hpmcounter31h</code>	<code>CSR_HPMCOUNTER3H .. CSR_HPMCOUNTER31H</code>	UR O	User performance-monitoring counter 3..31 high word

**Machine Counter Setup CSRs**

Address	Name [ASM]	Name [C]	AC C	Function
0x320	<code>mcountinhibit</code>	<code>CSR_MCOUNTINHIBIT</code>	M R W	Machine counter-inhibit register
<b>Machine Information CSRs</b>				
0xF11	<code>mvendorid</code>	<code>CSR_MVENDORID</code>	M RO	Machine vendor ID
0xF12	<code>marchid</code>	<code>CSR_MARCHID</code>	M RO	Machine architecture ID
0xF13	<code>mimpid</code>	<code>CSR_MIMPID</code>	M RO	Machine implementation ID / version
0xF14	<code>mhartid</code>	<code>CSR_MHARTID</code>	M RO	Machine hardware thread ID
0xF15	<code>mconfigptr</code>	<code>CSR_MCONFIGPTR</code>	M RO	Machine configuration pointer register
<b>NEORV32-Specific CSRs</b>				
0xFC0	<code>mxisa</code>	<code>CSR_MXISA</code>	M RO	NEORV32-specific "extended" machine CPU ISA and extensions

### 3.9.1. Floating-Point CSRs

#### fflags

Name Floating-point accrued exceptions

Address **0x001**

Reset **0x00000000**

value

ISA **Zicsr + Zfinx**

Description FPU status flags.  
on

*Table 50. 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
31:5	r/-	<i>reserved</i> , writes are ignored; reads always return 0

#### 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 51. frm CSR bits*

Bit	R/W	Function
2:0	r/w	Rounding mode
31:3	r/-	<i>reserved</i> , writes are ignored; reads always return 0

**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 52. **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> )
31:6	r/-	<i>reserved</i> , writes are ignored; reads always return 0

### 3.9.2. Machine Configuration CSRs

#### menvcfg

Name	Machine environment configuration register
Address	<code>0x30a</code>
Reset value	<code>0x00000000</code>
ISA	<code>Zicsr + U</code>
Description	The features of this CSR are not implemented yet. The register is read-only and always returns zero.

#### menvcfgh

Name	Machine environment configuration register - high word
Address	<code>0x31a</code>
Reset value	<code>0x00000000</code>
ISA	<code>Zicsr + U</code>
Description	The features of this CSR are not implemented yet. The register is read-only and always returns zero.

### 3.9.3. Machine Trap Setup CSRs

#### `mstatus`

Name Machine status register - low word

Address `0x300`

Reset `0x00000000`

value

ISA `Zicsr`

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

on

Table 53. `mstatus` CSR bits

Bit	Name [C]	R/W	Function
3	<code>CSR_MSTATUS_MIE</code>	r/w	<b>MIE:</b> Machine global interrupt enable flag
7	<code>CSR_MSTATUS_MPIE</code>	r/w	<b>MPIE:</b> Previous machine global interrupt enable flag state
12:11	<code>CSR_MSTATUS_MPP_H</code> : <code>CSR_MSTATUS_MPP_L</code>	r/w	<b>MPP:</b> Previous machine privilege level, 11 = machine (M) level, 00 = user (U) level
17	<code>CSR_MSTATUS_MPRV</code>	r/w	<b>MPRV:</b> Effective privilege level for load/stores in machine mode; use `MPP's as effective privilege level 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

on 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 54. `misa` CSR bits

Bit	Name [C]	R/W	Function
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: extension bit is always set to indicate custom non-standard extensions
31:30	<code>CSR_MISA_MXL_HI_EXT</code>	r/-	<b>MXL</b> : 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 `0x304`

Reset `0x00000000`

value

ISA `Zicsr`

Description The `mie` CSR is used to enable/disable individual interrupt sources.  
on

Table 55. `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

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

## mtvec

Name	Machine trap-handler base address
Address	0x305
Reset value	0x00000000
ISA	Zicsr
Description	The <b>mtvec</b> CSR contain the address of the primary trap handler, which gets executed whenever an interrupt is triggered or an exception is raised.

Table 56. **mtvec** CSR bits

Bit	R/W	Function
31:2	r/w	<b>BASE</b> : 4-byte aligned base address of trap base handler
1:0	r/-	<b>MODE</b> : always zero; BASE defines entry for <i>all</i> traps

## mcounteren

Name	Machine counter enable
Address	0x306
Reset value	0x00000000
ISA	Zicsr + U
Description	The <b>mcounteren</b> CSR is used to constrain user-level access to the CPU's counter CSRs. This CSR is also available if U mode is disabled, but the register is hardwired to all-zero in this case.

Table 57. **mcounteren** CSR bits

Bit	R/W	Function
0	r/w	<b>CY</b> : User-level code is allowed to read <b>cycle[h]</b> CSRs when set

Bit	R/W	Function
1	r/-	TM: Hardwired to zero as <code>time[h]</code> CSRs are not implemented
2	r/w	IR: User-level code is allowed to read <code>instret[h]</code> CSRs when set
31:3	r/w	HPM: user-level code is allowed to read <code>hpmcounter[h]</code> CSRs when set

### `mstatush`

Name	Machine status register - high word
Address	<code>0x310</code>
Reset value	<code>0x00000000</code>
ISA	<code>Zicsr</code>
Description	The features of this CSR are not implemented yet. The register is read-only and always returns zero.

### 3.9.4. Machine Trap Handling CSRs

#### `mscratch`

Name      Scratch register for machine trap handlers

Address    `0x340`

Reset     `0x00000000`

value

ISA       `Zicsr`

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

#### `mepc`

Name      Machine exception program counter

Address    `0x341`

Reset     `0x00000000`

value

ISA       `Zicsr`

Description The `mepc` CSR provides the instruction address where execution has stopped/failed  
on when an instruction is triggered / an exception is raised. See section [Traps, Exceptions and Interrupts](#) for a list of all legal values.

#### `mcause`

Name      Machine trap cause

Address    `0x342`

Reset     `0x00000000`

value

ISA       `Zicsr`

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 58. `mcause` CSR bits

Bit	R/W	Function
4:0	r/w	<b>Exception code:</b> see <a href="#">NEORV32 Trap Listing</a>
30:5	r/-	<i>Reserved</i> , read as zero

Bit	R/W	Function
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 register

Address **0x343**

Reset value **0x00000000**

ISA **Zicsr**

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



Note that the NEORV32 **mtval** register is read only. Any write-access will be ignored and will not cause an exception to maintain RISC-V compatibility.

Table 59. **mtval** CSR bits

Trap cause	mtval value
misaligned instruction fetch address or instruction fetch access fault	address of faulting instruction fetch
misaligned load address, load access fault, misaligned store address or store access fault	program counter (= address) of faulting instruction
illegal instruction	instruction word that caused the exception (zero-extended if compressed instruction)
everything else (including all interrupts)	all-zero

**mip**

Name Machine interrupt pending

Address **0x344**

Reset value **0x00000000**

ISA **Zicsr**

Descripti on The `mip` CSR shows currently *pending* machine-level interrupt requests. The bits for the standard RISC-V machine-level interrupts (`MEIP`, `MTIP`, `MSIP`) are read-only. Hence, these interrupts cannot be cleared/set using the `mip` register. These interrupts are cleared/acknowledged by mechanism that are specific for the interrupt-causing modules. the according interrupt-generating device.

*Table 60. `mip` CSR bits*

Bit	Name [C]	R/W	Function
3	<code>CSR_MIP_MSIP</code>	r/-	<b>MSIP</b> : Machine <i>software</i> interrupt pending; <i>cleared by platform-defined mechanism</i>
7	<code>CSR_MIP_MTIP</code>	r/-	<b>MTIP</b> : Machine <i>timer</i> interrupt pending; <i>cleared by platform-defined mechanism</i>
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> : <code>CSR_MIP_FIRQ0P</code>	r/c	<b>FIRQxP</b> : Fast interrupt channel 15..0 pending; has to be cleared manually by writing zero



#### *FIRQ Channel Mapping*

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

### 3.9.5. 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 [PMP ISA Extension](#) for more information.

#### `pmpcfg`

Name	PMP region configuration registers
Address	<code>0x3a0</code> ( <code>pmpcfg0</code> ) ... <code>0x3af</code> ( <code>pmpcfg15</code> )
Reset value	<code>0x00000000</code>
ISA	<code>Zicsr + PMP</code>
Description	Configuration of physical memory protection regions. Each region provides an individual 8-bit array in these CSRs.



Note that only PMP configuration registers `pmpcfg0` to `pmpcfg3` will be implemented if `PMP_NUM_REGIONS` is set to its maximum value (=16). The remaining `pmpcfg4` to `pmpcfg15` CSRs are read-only and will always read as zero.

Table 61. `pmpcfg0` CSR Bits

Bit	Name [C]	R/W	Function
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
6:5	-	r/-	<i>reserved</i> , read as zero
4:3	<code>PMPCFG_A_MS</code>	r/w	A: Mode configuration ( <code>00</code> = OFF, <code>01</code> = TOR, <code>10</code> = NA4, <code>11</code> = NAPOT)
	B :		
	<code>PMPCFG_A_LS</code>		
	B		
2	<code>PMPCFG_X</code>	r/w	X: Execute permission
1	<code>PMPCFG_W</code>	r/w	W: Write permission
0	<code>PMPCFG_R</code>	r/w	R: Read permission

#### `pmpaddr`

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

Name	Physical memory protection address registers
------	--

Address `0x3b0 (pmpaddr0) ... 0x3ef (pmpaddr63)`

Reset `0x00000000`  
value

ISA `Zicsr + PMP`

Description Region address configuration. The two MSBs of each CSR are hardwired to zero (= bits 33:32 of the physical address).



Note that only PMP address registers `pmpaddr0` to `pmpaddr15` will be implemented if `PMP_NUM_REGIONS` is set to its maximum value (=16). The remaining `pmpaddr16` to `pmpaddr63` CSRs are read-only and will always read as zero.

#### *Address Register Update Latency*



After writing a `pmpaddr` CSR the hardware requires up to 32 clock cycles to compute the according address masks. Make sure to wait for this time before completing the PMP region configuration (only relevant for `NA4` and `NAPOT` modes).

### 3.9.6. (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 `0xc00 (cycle), 0xc80 (cycleh)`

Reset `0x00000000`

value

ISA `Zicsr + Zicntr`

Description The `cycle[h]` CSRs are user-mode shadow copies of the according `mcycle[h]` CSRs. The user-level counter are read-only. Any write access will raise an illegal instruction exception.

#### `instret[h]`

Name Instructions-retired counter

Address `0xc02 (instret), 0xc82 (instreth)`

Reset `0x00000000`

value

ISA `Zicsr + Zicntr`

Description The `instret[h]` CSRs are user-mode shadow copies of the according `minstret[h]` CSRs. The user-level counter are read-only. Any write access will raise an illegal instruction exception.

#### `mcycle[h]`

Name Machine cycle counter

Address `0xb00` (`mcycle`), `0xb80` (`mcycleh`)

Reset `0x00000000`  
value

ISA `Zicsr + Zicntr`

Description If not halted via the `mcountinhibit` CSR the `cycle[h]` 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 `0xb02` (`minstret`), `0xb82` (`minstreth`)

Reset `0x00000000`  
value

ISA `Zicsr + Zicntr`

Description If not halted via the `mcountinhibit` CSR the `minstret[h]` CSRs will increment with every retired instruction. These registers are read/write only for machine-mode software

---

### 3.9.7. 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 28 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>0x232</code> ( <code>mhpmevent3</code> ) ... <code>0x33f</code> ( <code>mhpmevent31</code> )
Reset value	<code>0x00000000</code>
ISA	<code>Zicsr</code> + <code>Zihpm</code>
Descripti on	<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 62. `mhpmevent*` CSR Bits

Bit	Name [C]	R/W	Event Description
0	<code>HPMCNT_EVENT_CY</code>	r/w	active clock cycle (CPU not in sleep mode)
1	-	r/-	<i>not implemented, always read as zero</i>
2	<code>HPMCNT_EVENT_IR</code>	r/w	retired instruction (compressed or uncompressed)
3	<code>HPMCNT_EVENT_CIR</code>	r/w	retired compressed instruction
4	<code>HPMCNT_EVENT_WAIT_I F</code>	r/w	instruction fetch memory wait cycle
5	<code>HPMCNT_EVENT_WAIT_I I</code>	r/w	instruction issue pipeline wait cycle
6	<code>HPMCNT_EVENT_WAIT_M C</code>	r/w	multi-cycle ALU operation wait cycle (like iterative shift operation)
7	<code>HPMCNT_EVENT_LOAD</code>	r/w	memory data load operation
8	<code>HPMCNT_EVENT_STORE</code>	r/w	memory data store operation

Bit	Name [C]	R/W	Event Description
9	HPMCNT_EVENT_WAIT_L S	r/w	load/store memory wait cycle
10	HPMCNT_EVENT_JUMP	r/w	unconditional jump
11	HPMCNT_EVENT_BRANCH	r/w	conditional branch ( <i>taken or not taken</i> )
12	HPMCNT_EVENT_TBRANC H	r/w	<i>taken</i> conditional branch
13	HPMCNT_EVENT_TRAP	r/w	entered trap (synchronous exception or interrupt)
14	HPMCNT_EVENT_ILLEGA L	r/w	illegal instruction exception
31:15	-	r/-	<i>reserved</i> , read as zero

**mhpmcOUNTER[h]**

Name	Machine hardware performance monitor
Address	0xb03 (mhpmcOUNTER3) ... 0xb1f (mhpmcOUNTER31) 0xb83 (mhpmcOUNTER3h) ... 0xb9f (mhpmcOUNTER31h)
Reset value	0x00000000
ISA	Zicsr + Zihpm
Description	If not halted via the <b>mcountinhibit</b> CSR the [ <b>mhpmcOUNTER*[h]</b> ] counter CSR increment whenever a configured event from the according <b>mhpmevent</b> 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
Address	0xc03 (hpmcounter3) ... 0xc1f (hpmcounter31) 0xc83 (hpmcounter3h) ... 0xc9f (hpmcounter31h)
Reset value	0x00000000
ISA	Zicsr + Zihpm
Description	The <b>hpmcounter*[h]</b> are user-level shadow copies of the according <b>mhpmcOUNTER[h]</b> CSRs. The user level counter CSRs are read-only. Any write access will raise an illegal instruction exception.

### 3.9.8. 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. on

Table 63. mcountinhibit CSR Bits

Bit	Name [C]	R/W	Event
0	CSR_MCOUNTINHIBIT_I R	r/w	IR: Set to 1 to halt [ $m$ ]instret[h]; hardwired to zero if Zicntr ISA extension is disabled
1	-	r/-	TM: Hardwired to zero as time[h] CSRs are not implemented
2	CSR_MCOUNTINHIBIT_C Y	r/w	CY: Set to 1 to halt [ $m$ ]cycle[h]; hardwired to zero if Zicntr ISA extension is disabled
3:31	CSR_MCOUNTINHIBIT_H PM3 : CSR_MCOUNTINHIBIT_H PM31	r/w	HPMx: Set to 1 to halt [ $m$ ]hpmcount*[h]; hardwired to zero if Zihpm ISA extension is disabled

### 3.9.9. 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), assigned via the <code>VENDOR_ID</code> top generic ( <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: <code>0x00000013</code> ).

#### **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</code> = <code>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 registerD

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.

### 3.9.10. 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>0x7c0</code>
Reset value	<code>DEFINED</code>
ISA	<code>Zicsr + 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 64. `mxisa` CSR Bits

Bit	Name [C]	R/W	Function
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	<code>CSR_MXISA_ZICOND</code>	r/-	<code>Zicond</code> ISA Extension available
5	<code>CSR_MXISA_ZFINX</code>	r/-	<code>Zfinx</code> ISA Extension available
6	-	r/-	<i>reserved</i> , read as zero
7	<code>CSR_MXISA_ZICNTR</code>	r/-	<code>Zicntr</code> ISA Extension available
8	<code>CSR_MXISA_PMP</code>	r/-	<code>PMP</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/-	<i>reserved</i> , read as zero
20	<code>CSR_MXISA_IS_SIM</code>	r/-	set if CPU is being <b>simulated</b> (not guaranteed)
31:21	-	r/-	<i>reserved</i> , read as zero
30	<code>CSR_MXISA_FASTMUL</code>	r/-	fast multiplication available when set ( <code>FAST_MUL_EN</code> )
31	<code>CSR_MXISA_FASTSHIFT</code>	r/-	fast shifts available when set ( <code>FAST_SHIFT_EN</code> )

### 3.9.11. 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 transfers control to the address stored in **mtvec** CSR. The cause of the 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. This might be the address of the instruction that actually caused the trap or that has not been executed yet as it was interrupted by a trap. Additional information regarding the cause of the trap can be retrieved from the **mtval** CSR and the processor's [Internal Bus Monitor \(BUSKEEPER\)](#) (for bus access exceptions).

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 standard RISC-V interrupt request signals are **high-active**. A request has to stay at high-level until it is explicitly acknowledged by the CPU software (for example by writing to a specific memory-mapped register).



#### *Interrupt Signal Requirements - NEORV32-Specific Fast Interrupt Requests*

The NEORV32-specific FIRQ request lines are triggered (= becoming pending) by a one-shot high-level.



#### *Instruction Atomicity*

All instructions execute as atomic operations - interrupts can only trigger *between* consecutive instructions. Even 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. Load exceptions caused by a misalignment or a physical memory protection fault do not trigger a bus/memory read-operation at all. Vice versa, exceptions caused by a store address misalignment or a store physical memory protection fault do not trigger a bus/memory write-operation 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. The highest priority is 1. The "mcause" column shows the cause ID of the according trap that is written to `mcause` CSR. The "ID [C]" names are defined by the NEORV32 core library (the runtime environment *RTE*) and can be used in plain C code. The `mepc` and `mtval` columns show the values written to the according CSRs when a trap is triggered:

- **I-PC** - address of interrupted instruction (instruction has *not* been executed yet)
- **PC** - address of instruction that caused the trap (instruction has been executed)
- **ADR** - bad memory access address that caused the trap
- **CMD** - the instruction word that caused the trap (zero-extended if compressed instruction)
- **0** - zero

Table 65. NEORV32 Trap Listing

Pr io.	mcause	ID [C]	Cause	mepc	mtval
Exceptions (synchronous to instruction execution)					
1	<code>0x00000000</code>	<code>TRAP_CODE_I_MISALIGNED</code>	instruction address misaligned	<b>I-PC</b>	<b>0</b>
2	<code>0x00000001</code>	<code>TRAP_CODE_I_ACCESS</code>	instruction access bus fault	<b>I-PC</b>	<b>0</b>
3	<code>0x00000002</code>	<code>TRAP_CODE_I_ILLEGAL</code>	illegal instruction	<b>PC</b>	<b>CMD</b>
4	<code>0x0000000B</code>	<code>TRAP_CODE_MENV_CALL</code>	environment call from M-mode <i>(ecall)</i>	<b>PC</b>	<b>0</b>

Pr io.	mcause	ID [C]	Cause	mepc	mtval
5	0x00000008	TRAP_CODE_UENV_CALL	environment call from U-mode ( <code>ecall</code> )	PC	0
6	0x00000003	TRAP_CODE_BREAKPOINT	software breakpoint ( <code>ebreak</code> )	PC	0
7	0x00000006	TRAP_CODE_S_MISALIGNED	store address misaligned	PC	ADR
8	0x00000004	TRAP_CODE_L_MISALIGNED	load address misaligned	PC	ADR
9	0x00000007	TRAP_CODE_S_ACCESS	store access bus fault	PC	ADR
10	0x00000005	TRAP_CODE_L_ACCESS	load access bus fault	PC	ADR
Interrupts (asynchronous to instruction execution)					
11	0x80000010	TRAP_CODE_FIRQ_0	fast interrupt request channel 0	I-PC	0
12	0x80000011	TRAP_CODE_FIRQ_1	fast interrupt request channel 1	I-PC	0
13	0x80000012	TRAP_CODE_FIRQ_2	fast interrupt request channel 2	I-PC	0
14	0x80000013	TRAP_CODE_FIRQ_3	fast interrupt request channel 3	I-PC	0
15	0x80000014	TRAP_CODE_FIRQ_4	fast interrupt request channel 4	I-PC	0
16	0x80000015	TRAP_CODE_FIRQ_5	fast interrupt request channel 5	I-PC	0
17	0x80000016	TRAP_CODE_FIRQ_6	fast interrupt request channel 6	I-PC	0
18	0x80000017	TRAP_CODE_FIRQ_7	fast interrupt request channel 7	I-PC	0
19	0x80000018	TRAP_CODE_FIRQ_8	fast interrupt request channel 8	I-PC	0
20	0x80000019	TRAP_CODE_FIRQ_9	fast interrupt request channel 9	I-PC	0
21	0x8000001a	TRAP_CODE_FIRQ_10	fast interrupt request channel 10	I-PC	0
22	0x8000001b	TRAP_CODE_FIRQ_11	fast interrupt request channel 11	I-PC	0
23	0x8000001c	TRAP_CODE_FIRQ_12	fast interrupt request channel 12	I-PC	0
24	0x8000001d	TRAP_CODE_FIRQ_13	fast interrupt request channel 13	I-PC	0
25	0x8000001e	TRAP_CODE_FIRQ_14	fast interrupt request channel 14	I-PC	0
26	0x8000001f	TRAP_CODE_FIRQ_15	fast interrupt request channel 15	I-PC	0
27	0x8000000B	TRAP_CODE_MEI	machine external interrupt (MEI)	I-PC	0
28	0x80000003	TRAP_CODE_MSI	machine software interrupt (MSI)	I-PC	0
29	0x80000007	TRAP_CODE_MTI	machine timer interrupt (MTI)	I-PC	0

Table 66. NEORV32 Trap Description

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 ID [C]	Triggered when ...
TRAP_CODE_I_ACCESS	bus timeout or bus access error during instruction word fetch
TRAP_CODE_I_ILLEGAL	trying to execute an invalid instruction word (malformed or not supported) or on a privilege violation
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 <b>Trigger Module</b> fires
TRAP_CODE_S_MISALIGNED	storing data to an address that is not naturally aligned to the data size (byte, half, word)
TRAP_CODE_L_MISALIGNED	loading data from an address that is not naturally aligned to the data size (byte, half, word)
TRAP_CODE_S_ACCESS	bus timeout or bus access error during load data operation
TRAP_CODE_L_ACCESS	bus timeout or bus access error during store data operation
TRAP_CODE_FIRQ_0 ... TRAP_CODE_FIRQ_15	caused by interrupt-condition of processor-internal modules, see <a href="#">NEORV32-Specific Fast Interrupt Requests</a>
TRAP_CODE_MEI	machine external interrupt (via dedicated top-entity signal)
TRAP_CODE_MSI	machine software interrupt (via dedicated top-entity signal)
TRAP_CODE_MTI	machine timer interrupt (internal machine timer or via dedicated top-entity signal)

#### *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.

#### *Misaligned Instruction Address Exception*



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*. In both cases bit 0 of the program counter (and all related CSRs) is hardwired to zero.

### 3.9.12. Bus Interface

The NEORV32 CPU provides separated instruction and data 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 following table shows the signals of the data and instruction interfaces as seen from the CPU (\*`_o` signals are driven by the CPU / outputs, \*`_i` signals are read by the CPU / inputs).

*Table 67. CPU Bus Interface Signals*

Signal	Width	Direction	Description
		n	
<code>i/d_bus_addr_o</code>	32	out	access address
<code>i/d_bus_rdata_i</code>	32	in	data input for read operations
<code>d_bus_wdata_o</code>	32	out	data output for write operations
<code>d_bus_ben_o</code>	4	out	byte enable signal for write operations
<code>d_bus_we_o</code>	1	out	bus write access request (one-shot)
<code>i/d_bus_re_o</code>	1	out	bus read access request (one-shot)
<code>i/d_bus_ack_i</code>	1	in	accessed peripheral indicates a successful completion of the bus transaction
<code>i/d_bus_err_i</code>	1	in	accessed peripheral indicates an error during the bus transaction
<code>i/d_bus_fence_o</code>	1	out	this signal is set for one cycle when the CPU executes an instruction/data fence command
<code>i/d_bus_priv_o</code>	1	out	shows the effective privilege level of the bus access



#### Pipelined Transfers

Currently, pipelined or overlapping operations (within the same bus interface) are not implemented. So only a single transfer request can be in progress at once.



#### Unaligned Memory Accesses

Please note that the NEORV32 CPU does **not support the handling of unaligned memory accesses** in hardware. Any unaligned memory access will raise an exception that can be used to handle unaligned accesses in software (via emulation).



#### Signal State

All outgoing bus interface signals (that are driven by the CPU) remain stable until the bus access is completed.

## Bus Interface Protocol

A new bus request is triggered either by the `*_bus_re_o` signal (for reading data) or by the `*_bus_we_o` signal (for writing data). In case of a request, the according signal is high for exactly one clock cycle. The transaction is completed when the accessed peripheral/memory either sets the `*_bus_ack_i` signal (indicating successful completion) or the `*_bus_err_i` signal (indicating failed completion). These bus response signals have to be also set only for just one cycle. If a bus request is terminated by the `*_bus_err_i` signal the CPU will raise the according "bus access fault" exception.

### Minimal Response Latency

The transfer can be completed within in the same cycle as it was initiated (asynchronous response) if the accessed module directly sets `*_bus_ack_i` or `*_bus_err_i` high for one cycle. However, in order to shorten the critical path such an "asynchronous" response should be avoided. The default NEORV32 processor-internal modules use a registered response with exactly **one cycle delay** between initiation and completion of transfers.

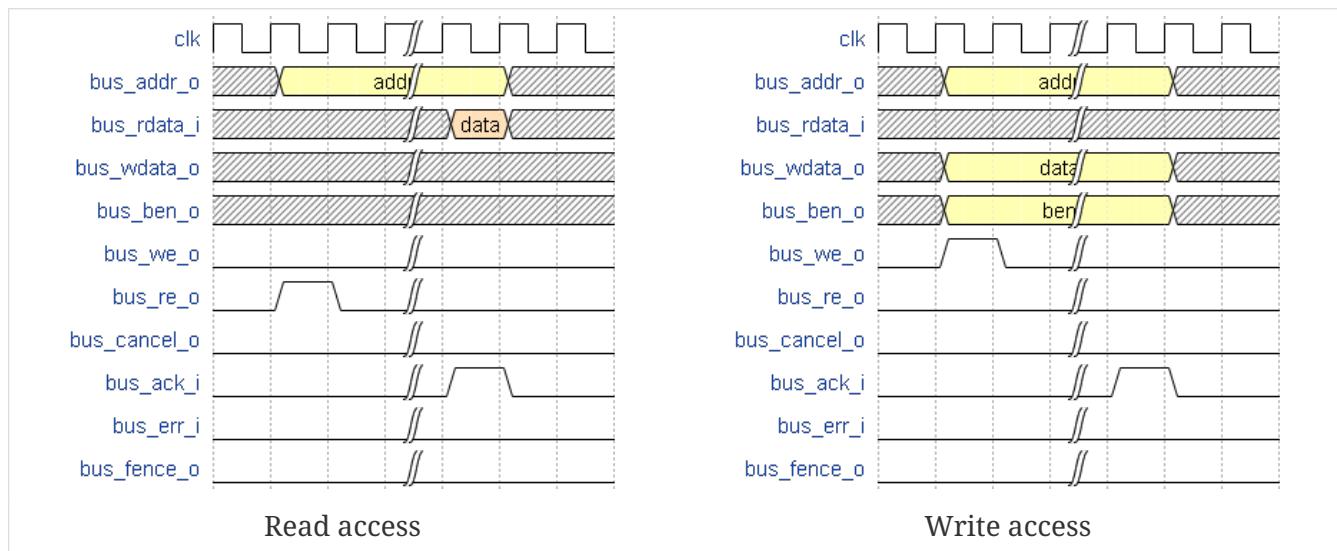
### Maximal Response Latency

The processor-internal modules do not have to respond within one cycle after a bus request has been initiated. However, the bus transaction has to be completed (= acknowledged) within a certain **response time window**. This time window is defined by the global `max_proc_int_response_time_c` constant (default = 15 cycles) defined in the processor's VHDL package file `rtl/neorv32_package.vhd`. It defines the maximum number of cycles after which an *unacknowledged* (`*_bus_ack_i` or `*_bus_err_i` signals both not set) transfer will **time out** and will raise a bus access fault exception. The **Internal Bus Monitor (BUSKEEPER)** keeps track of all bus transactions to enforce this time window.

If any bus operations times out - for example when accessing "address space holes" - the BUSKEEPER will issue a bus error to the CPU (via the according `*bus_err_i` signal) that will raise the according instruction fetch or data access bus exception. Note that the bus keeper does not track external accesses via the external memory bus interface. However, the external memory bus interface also provides an `_optional` bus timeout (see section **Processor-External Memory Interface (WISHBONE)**).

## Exemplary Bus Accesses

Table 68. Example CPU Bus Accesses



## Read and Write Accesses

For a write access the according access address (`bus_addr_o`), the data to-be-written (`bus_wdata_o`) and the byte enable (`bus_ben_o`) are set when `bus_we_o` goes high. These three signals remain unchanged until the transaction is completed.

For a read access the according access address (`bus_addr_o`) is set when `bus_re_o` goes high. The address remains unchanged until the transaction is completed.

## Access Boundaries

The instruction interface will always access memory on word (= 32-bit) boundaries even if fetching compressed (16-bit) instructions. The data interface can access memory on byte (= 8-bit), half-word (= 16-bit) and word (= 32-bit) boundaries, but not all processor module support sub-word accesses.

## Memory Barriers

Whenever the CPU executes a `fence` instruction, the according interface signal is set high for one cycle (`d_bus_fence_o` for a `fence` instruction; `i_bus_fence_o` for a `fence.i` instruction). It is the task of the memory system to perform the necessary operations (for example a cache flush/reload).

# 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/neorv32/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_zicsr`
- `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 are automatically included and linked by adding the following include statement:

```
#include <neorv32.h> // NEORV32 HAL, core and runtime libraries
```

*Table 69. NEORV32 HAL File List*

C source file	C header file	Description
-	neorv32.h	Main NEORV32 library file
-	neorv32_buskeeper.h	Internal Bus Monitor (BUSKEEPER) HAL
neorv32_cfs.c	neorv32_cfs.h	Custom Functions Subsystem (CFS) HAL
neorv32_cpu.c	neorv32_cpu.h	NEORV32 Central Processing Unit (CPU) HAL
	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_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_spi.c	neorv32_spi.h	Serial Peripheral Interface Controller (SPI) HAL
-	neorv32_sysinfo.h	System Configuration Information Memory (SYSINFO) HAL
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

C source file	C header file	Description
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"
-	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](#).

## 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 SW Application Makefile >>>
Make sure to add the bin folder of RISC-V GCC to your PATH variable.

==== Targets ====
help      - show this text
check     - check toolchain
info      - show makefile/toolchain configuration
gdb       - run GNU debugger
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], default ""
USER_LIBS       - Custom libraries [append only], default ""
EFFORT         - Optimization level, default "-Os"
MARCH           - Machine architecture, default "rv32i_zicsr"
MABI            - Machine binary interface, default "ilp32"
APP_INC         - C include folder(s) [append only], default "-I ."
ASM_INC         - ASM include folder(s) [append only], default "-I ."
RISCV_PREFIX    - Toolchain prefix, default "riscv32-unknown-elf-"
NEORV32_HOME    - NEORV32 home folder, default ".../..."
```

### 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 clean_all exe` overrides the default `MARCH` variable definitions.

*Listing 8. 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
MABI ?= ilp32
# User flags for additional configuration (will be added to compiler flags)
USER_FLAGS ?=
# User libraries (will be included by linker)
```

```

USER_LIBS ?=
# Relative or absolute path to the NEORV32 home folder
NEORV32_HOME ?= ../../..
# ****

```

Table 70. 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>i1p32</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>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

### 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.

<b>-Wall</b>	Enable all compiler warnings.
<b>-ffunction-sections</b>	Put functions and data segment in independent sections. This allows a code optimization as dead code and unused data can be easily removed.
<b>-nostartfiles</b>	Do not use the default start code. Instead, the NEORV32-specific start-up code ( <b>sw/common/crt0.S</b> ) is used (pulled-in by the linker script).
<b>-Wl,--gc-sections</b>	Make the linker perform dead code elimination.
<b>-lm</b>	Include/link with <b>math.h</b> .
<b>-lc</b>	Search for the standard C library when linking.
<b>-lgcc</b>	Make sure we have no unresolved references to internal GCC library subroutines.
<b>-mno-fdiv</b>	Use built-in software functions for floating-point divisions and square roots (since the according instructions are not supported yet).

**-g**

Include debugging information/symbols in ELF.

#### 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 9. Using the `USER_FLAGS` Variable for Customization*

```
$ make USER_FLAGS+=" -g -Wl,--__neorv32_heap_size,__heap_size=4096" clean_all exe
```

## 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 [sw/common/neorv32.ld](#) for linking all object files that were generated during compilation. In general, the linker script defines three memory sections: `rom`, `ram` and `iodev`.

*Table 71. Linker script - memory sections*

Memory section	Description
<code>ram</code>	Data memory address space (processor-internal/external DMEM)
<code>rom</code>	Instruction memory address space (processor-internal/external IMEM) or internal bootloader ROM
<code>iodev</code>	Processor-internal memory-mapped IO/peripheral devices address space



The `iodev` section is entirely defined by the processor hardware layout and should not be modified at all.



The `rom` section is automatically re-mapped to the processor-internal [Bootloader ROM \(BOOTROM\)](#) when (re-)compiling the bootloader

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. The attributes are configured indirectly via variables that provide default values.

*Listing 10. 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 - do not change this unless the hardware-defined
address space layout is changed! */
__neorv32_rom_base = DEFINED(__neorv32_rom_base) ? __neorv32_rom_base : 0x00000000; /* = VHDL package's "ispace_base_c" */
__neorv32_ram_base = DEFINED(__neorv32_ram_base) ? __neorv32_ram_base : 0x80000000; /* = VHDL package's "dspace_base_c" */
```

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 11. Overriding default rom size configuration (setting 4096 bytes)

```
$ make USER_FLAGS+=“-Wl,--defsym,__neorv32_rom_size=4096” clean_all exe
```



`neorv32_rom_base` (= `ORIGIN` of the `ram` section) has to be always identical to the processor's `dspace_base_c` hardware configuration. Also, `neorv32_ram_base` (= `ORIGIN` of the `rom` section) has to be always identical to the processor's `ispace_base_c` hardware configuration.



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 72. 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 create four areas. Note that depending on the application some areas might not be existent at all.

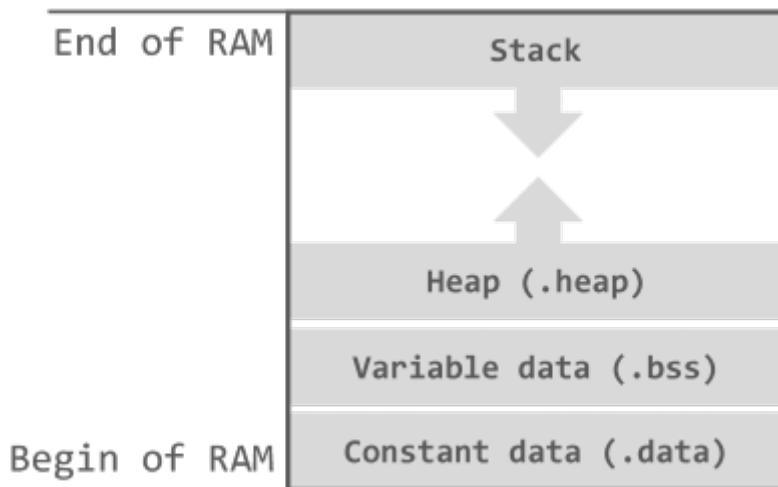


Figure 11. Default RAM Layout

- 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.

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.
<b>-raw_bin</b>	Generates a plain binary file <b>neorv32_raw_exe.bin</b> (no header) for custom purpose.

**-bld\_img**

Generates an executable VHDL memory initialization image (no header) for the processor-internal BOOT ROM. This option generates the `rtl/core/neorv32_bootloader_image.vhd` 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 add 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-boot dummy 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:
  - Interrupts are disabled by clearing `mie`.
  - `mains's return value is copied to the `mscratch` CSR to allow inspection by the debugger.
  - Call all *destructors* (if there are any).
  - An optional **After-Main Handler** is called (if defined at all).
  - 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.

### After-Main Handler

If the application's `main()` function actually returns, an *after main handler* can be executed. This handler is a "normal" function as the C runtime is still available when executed. If this handler uses any kind of peripheral/IO modules make sure these are already initialized within the application. Otherwise you have to initialize them *inside* the handler.

#### *Listing 12. After-main handler - function prototype*

```
void __neorv32_crt0_after_main(int32_t return_code);
```

The function has exactly one argument (`return_code`) that provides the *return value* of the application's main function. For instance, this variable contains `-1` if the main function returned with `return -1;`. The after-main handler itself does not provide a return value.

A simple UART output can be used to inform the user when the application's main function returns (this example assumes that UART0 has been already properly configured in the actual application):

#### *Listing 13. After-main handler - simple example*

```
void __neorv32_crt0_after_main(int32_t return_code) {
    neorv32_uart0_printf("\n<RTE> main function returned with exit code %i. </RTE>\n",
    return_code); ①
}
```

① Use `<RTE>` here to make clear this is a message comes from the runtime environment.



The after-main handler is executed *after* executing all destructor functions (if there are any at all).

## 4.5. Bootloader



This section refers to the **default** NEORV32 bootloader.

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 <a href="#">INT_BOOTLOADER_EN</a> top generic is <b>true</b> . This will automatically select the CPU's <a href="#">Indirect Boot</a> boot configuration.
<b>REQUIRED</b>	The bootloader requires the privileged architecture CPU extension ( <a href="#">Zicsr ISA Extension</a> ) 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 <a href="#">Bootloader Console</a> (like uploading executables) the primary UART ( <a href="#">Primary Universal Asynchronous Receiver and Transmitter (UART0)</a> ) is required.
<b>RECOMMENDED</b>	The default bootloader uses bit 0 of the <a href="#">General Purpose Input and Output Port (GPIO)</a> output port to drive a high-active "heart beat" status LED.
<b>RECOMMENDED</b>	The <a href="#">Machine System Timer (MTIME)</a> is used to control blinking of the status LED and also to automatically trigger the <a href="#">Auto Boot Sequence</a> .
<b>OPTIONAL</b>	The SPI controller ( <a href="#">Serial Peripheral Interface Controller (SPI)</a> ) is needed to store/load executable from external flash using the <a href="#">Auto Boot Sequence</a> .
<b>OPTIONAL</b>	The XIP controller ( <a href="#">Execute In Place Module (XIP)</a> ) 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
CID: 0x00000000
CLK: 0x05f5e100
MISA: 0x40901106
XISA: 0xc0000fab
SOC: 0xfffff402f
IMEM: 0x00008000 bytes @0x00000000
DMEM: 0x00002000 bytes @0x80000000
```

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 <b>mimpid</b> CSR); in BCD format; example: <b>0x01040606</b> = v1.4.6.6).
<b>CID</b>	Custom user-defined ID (via the <b>CUSTOM_ID</b> register from <a href="#">System Configuration Information Memory (SYSINFO)</a> ).
<b>CLK</b>	Processor clock speed in Hz (via the <b>CLK</b> register from <a href="#">System Configuration Information Memory (SYSINFO)</a> ).
<b>MISA</b>	RISC-V CPU extensions ( <b>misa</b> CSR).
<b>XISA</b>	NEORV32-specific CPU extensions ( <b>mxisa</b> CSR).
<b>SOC</b>	Processor configuration (via the <b>SOC</b> register from the <a href="#">System Configuration Information Memory (SYSINFO)</a> ).
<b>IMEM</b>	IMEM memory base address and size in byte (via the <b>IMEM_SIZE</b> and <b>ISPACE_BASE</b> registers from the <a href="#">System Configuration Information Memory (SYSINFO)</a> ).
<b>DMEM</b>	DMEM memory base address and size in byte (via the <b>DMEM_SIZE</b> and <b>DSPACE_BASE</b> registers 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
CID: 0x00000000
CLK: 0x05f5e100
MISA: 0x40901106
```

```
XISA: 0xc0000fab
SOC: 0xfffff402f
IMEM: 0x000008000 bytes @0x00000000
DMEM: 0x00002000 bytes @0x80000000
```

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.

#### *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.

**ERR\_EXC**

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 `l` or `s` (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, which provides the base entry point for all trap handlers. 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

The NEORV32 is part of the default NEORV32 software framework. However, it has to explicitly enabled by calling the RTE's setup function:

*Listing 14. RTE Setup (Function Prototype)*

```
void nerv32_rte_setup(void);
```



The RTE should be enabled right at the beginning of the application's `main` function.

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 15. 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 16. 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 */
    ...
}
```

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 17. 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 18. 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 19. 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 20. Example: Removing the Custom MTIME IRQ Handler*

```
neorv32_rte_handler_uninstall(RTE_TRAP_MTI);
```

#### 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 21. RTE Default Trap Handler Output Examples*

```
<RTE> [M] Illegal instruction @ PC=0x000002d6, MVAL=0x000000FF </RTE> ①
<RTE> [U] Illegal instruction @ PC=0x00000302, MVAL=0x00000000 </RTE> ②
<RTE> [U] Load address misaligned @ PC=0x00000440, MVAL=0x80000101 </RTE> ③
<RTE> [M] Fast IRQ 0x00000003 @ PC=0x00000820, MVAL=0x00000000 </RTE> ④
<RTE> [M] Instruction access fault @ PC=0x90000000, MVAL=0x00000000 [FATAL EXCEPTION]
Halting CPU. </RTE>\n ⑤
```

- ① Illegal 32-bit instruction `0x000000FF` at address `0x000002d6` while the CPU was in machine-mode ([M]).
- ② Illegal 16-bit instruction `0x0000` (zero-extended) at address `0x00000302` while the CPU was in user-mode ([U]).
- ③ Misaligned load access at address `0x00000440` (trying to load a full 32-bit word from address `0x80000101`) while the CPU was in machine-mode.
- ④ Fast interrupt request from channel 3 before executing instruction at address `0x00000820` while the CPU was in machine-mode.
- ⑤ Instruction bus access fault at address `0x90000000` - this is fatal for the default debug trap handler while the CPU was in machine-mode.

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 73. 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>

Trap identifier	According <code>mcause</code> CSR value
"Fast IRQ 0x00000000"	0x80000010
"Fast IRQ 0x00000001"	0x80000011
"Fast IRQ 0x00000002"	0x80000012
"Fast IRQ 0x00000003"	0x80000013
"Fast IRQ 0x00000004"	0x80000014
"Fast IRQ 0x00000005"	0x80000015
"Fast IRQ 0x00000006"	0x80000016
"Fast IRQ 0x00000007"	0x80000017
"Fast IRQ 0x00000008"	0x80000018
"Fast IRQ 0x00000009"	0x80000019
"Fast IRQ 0x0000000a"	0x8000001a
"Fast IRQ 0x0000000b"	0x8000001b
"Fast IRQ 0x0000000c"	0x8000001c
"Fast IRQ 0x0000000d"	0x8000001d
"Fast IRQ 0x0000000e"	0x8000001e
"Fast IRQ 0x0000000f"	0x8000001f
"Unknown trap cause"	undefined

## Bus Access Faults

For bus access faults the RTE default trap handlers also outputs the error code obtained from the **Internal Bus Monitor (BUSKEEPER)** to show the cause of the bus fault. One example is shown below.

*Listing 22. RTE Default Trap Handler Output Example (Load Access Bus Fault)*

```
<RTE> Load access fault [TIMEOUT_ERR] @ PC=0x00000150, MVAL=0xFFFFF70 </RTE>
```

The additional message encapsulated in [ ] shows the actual cause of the bus access fault. Several different message identifiers are possible here:

- **[TIMEOUT\_ERR]**: The accessed memory-mapped module did not respond within the valid access time window. In Most cases this is caused by accessing a module that has not been implemented or when accessing "address space holes" via unused/unmapped addresses (see section **Bus Interface Protocol**).
- **[DEVICE\_ERR]**: The accesses memory-mapped module asserted its error signal to indicate an invalid access. For example this can be caused by trying to write to read-only registers.
- **[PMP\_ERR]**: This indicates an access right violation caused by the **PMP ISA Extension**.

# Chapter 5. On-Chip Debugger (OCD)

The NEORV32 Processor features an *on-chip debugger* (OCD) implementing the **execution-based debugging** scheme, which is compatible to the **Minimal RISC-V Debug Specification Version 1.0**. A copy of the specification is available in [docs/references](#).

## Section Structure

- [Debug Transport Module \(DTM\)](#)
- [Debug Module \(DM\)](#)
- [CPU Debug Mode](#)
- [Trigger Module](#)

## Key Features

- standard JTAG access port
- run-control of the CPU: halting, single-stepping and resuming
- executing arbitrary programs during debugging
- 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



### OCD Security Note

Access via the OCD is **always authenticated** (`dmstatus.authenticated = 1`). Hence, the entire system can always be accessed via the on-chip debugger.



### 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 complex is based on four hardware modules:

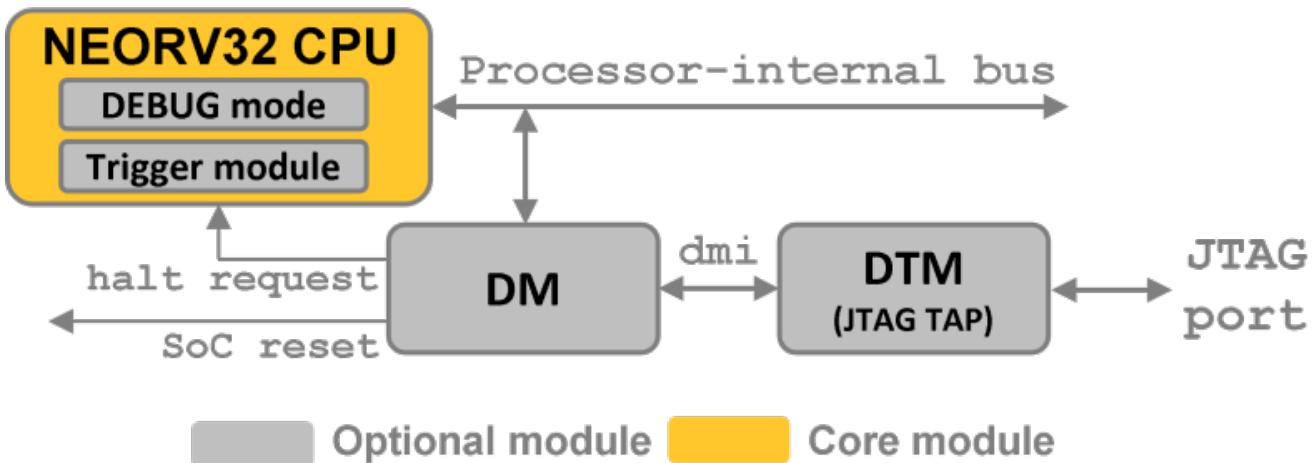


Figure 12. NEORV32 on-chip debugger complex

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)* - this interface is compatible to the interface description shown in Appendix 3 of the "RISC-V debug" specification.
2. **Debug Module (DM)** ([rtl/core/neorv32\\_debug\\_tm.vhd](#)): Debugger control unit 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 **CPU Debug Mode** extension (part of [rtl/core/neorv32\\_cpu\\_control.vhd](#)): This extension provides the "debug execution mode" which executes the "park loop" code from the DM. The mode also provides additional CSRs.
4. CPU **Trigger Module** (also part of [rtl/core/neorv32\\_cpu\\_control.vhd](#)): This module provides a single *hardware breakpoint*, which allows to debug code executed from ROM.

## 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 state, 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 CPU executes the "park loop" code from the *code ROM* of the DM. This park loop implements an endless loop, where the CPU polls the memory-mapped *status register* that is controlled by the *debug module (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 fired while being in debug mode.

## 5.1. Debug Transport Module (DTM)

The debug transport module (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 74. JTAG top level signals*

Name	Width	Direction	Description
jtag_trst_i	1	in	TAP reset (low-active); this signal is optional, make sure to pull it <i>high</i> if not used
jtag_tck_i	1	in	serial clock
jtag_tdi_i	1	in	serial data input
jtag_tdo_o	1	out	serial data output
jtag_tms_i	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`).



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 75. JTAG TAP registers*

Address (via IR)	Name	Size [bits]	Description
00001	IDCODE	32	identifier, hardwired to <code>0x00000001</code>
10000	DTMCS	32	debug transport module control and status register
10001	DMI	40	debug module interface ( <i>dmi</i> ); 6-bit address, 32-bit read/write data, 2-bit operation ( <b>00</b> = NOP; <b>10</b> = write; <b>01</b> = read)
others	BYPASS	1	default JTAG bypass register

*Table 76. DTMCS - DTM Control and Status Register*

Bit(s)	Name	R/W	Description
31:18	-	r/-	<i>reserved</i> , hardwired to zero
17	<code>dmihardreset</code>	r/w	setting this bit will reset the debug module interface; this bit auto-clears
16	<code>dmireset</code>	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	<code>idle</code>	r/-	recommended idle states (= 0, no idle states required)
11:10	<code>dmistat</code>	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	<code>abits</code>	r/-	number of address bits in <code>DMI</code> register (= 6)
3:0	<code>version</code>	r/-	<b>0001</b> = DTM is compatible to spec. version 0.13 & 1.0



The DTM's instruction and data registers can be accessed using OpenOCD's `irscan` and `drscan` commands. OpenOCD also provides low-level RISC-V-specific commands for direct DMI accesses (`riscv dmi_read` & `riscv dmi_write`).

## 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) and the platform-specific debugger (circuit) implementation. It supports the following features:

- Gives the debugger necessary information about the implementation.
- Allows the hart to be halted and resumed and provides status of the current state.
- Provides abstract read and write access to the halted hart's GPRs.
- Provides access to a reset signal that allows debugging from the very first instruction after reset.
- Provides a mechanism to allow debugging the hart immediately out of reset. (*still experimental*)
- 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 level. It implements the **execution based debugging scheme** for a single hart and provides the following hardware features:

- program buffer with 2 entries and implicit `ebreak` instruction afterwards
- no *direct* bus access; indirect bus access via the CPU using the program buffer
- abstract commands: "access register" plus auto-execution
- no *dedicated* halt-on-reset capabilities yet (but can be emulated)

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. From the DTM's point of view, the DM implements a set of **DM Registers** that are used to control and monitor the actual debugging. 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 *dmi*. The following registers are implemented:



Write accesses to registers that are not implemented are simply ignored and read accesses will always return zero. Register names that are encapsulated in "( )" are not actually implemented; however, they are listed to explicitly show their functionality.

*Table 77. Available DM registers*

Address	Name	Description
0x04	data0	Abstract data 0, used for data transfer between debugger and processor
0x10	dmcontrol	Debug module control
0x11	dmstatus	Debug module status
0x12	hartinfo	Hart information
0x16	abstracts	Abstract control and status
0x17	command	Abstract command
0x18	abstractauto	Abstract command auto-execution
0x1d	(nextdm)	Base address of <i>next</i> DM; reads as zero to indicate there is only <i>one</i> DM
0x20	progbuf0	Program buffer 0
0x21	progbuf1	Program buffer 1
0x38	(sbcs)	System bus access control and status; reads as zero to indicate there is <b>no</b> direct system bus access

data

**0x04 Abstract data 0** data0  
Reset value: 0x00000000  
Basic read/write registers 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 78. dmcontrol Register Bits*

Bit	Name [RISC-V]	R/W	Description
31	haltreq	-/w	set/clear hart halt request
30	resumereq	-/w	request hart to resume
28	ackhavereset	-/w	write 1 to clear *havereset flags
1	ndmreset	r/w	put whole processor into reset state when 1
0	dmaactive	r/w	DM enable; writing 0-1 will reset the DM

**dmstatus**0x11      **Debug module status register**[dmstatus](#)Reset value: **0x00400083**

Current status of the overall debug module and the hart. The entire register is read-only.

*Table 79. 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>	<b>1</b> when the hart is in reset
18	<b>anyhavereset</b>	
17	<b>allresumeack</b>	<b>1</b> 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>	<b>1</b> when the DM is disabled to indicate the hart is unavailable
12	<b>anyunavail</b>	
11	<b>allrunning</b>	<b>1</b> when the hart is running
10	<b>anyrunning</b>	
9	<b>allhalted</b>	<b>1</b> 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>hasresethaltr eq</b>	always <b>0</b> ; halt-on-reset is not supported (directly)
4	<b>confstrptrval id</b>	always <b>0</b> ; no configuration string available
3:0	<b>version</b>	<b>0011</b> - DM is compatible to spec. version 1.0

**hartinfo**0x12      **Hart information**[hartinfo](#)Reset value: *see below*

This register gives information about the hart. The entire register is read-only.

*Table 80. `hartinfo` Register Bits*

Bit	Name [RISC-V]	Description
31:24	<code>reserved</code>	reserved; always zero
23:20	<code>nscratch</code>	<code>0001</code> , number of <code>dscratch*</code> CPU registers = 1
19:17	<code>reserved</code>	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 81. `abstracts` Register Bits*

Bit	Name [RISC-V]	R/W	Description
31:29	<code>reserved</code>	r/-	reserved; always zero
28:24	<code>progbuftype</code>	r/-	always <code>0010</code> : size of the program buffer ( <code>progbuf</code> ) = 2 entries
23:11	<code>reserved</code>	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	<code>reserved</code>	r/-	reserved; always zero
3:0	<code>datacount</code>	r/-	always <code>0001</code> : number of implemented <code>data</code> 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 (abstract command register index **0x1000 - 0x101f**).

*Table 82. command Register Bits*

Bit	Name [RISC-V]	R/W	Description / required value
31:24	<b>cmdtype</b>	-/w	<b>00000000</b> to indicate "access register" command
23	<i>reserved</i>	-/w	reserved, has to be <b>0</b> when writing
22:20	<b>aarsize</b>	-/w	<b>010</b> to indicate 32-bit accesses
21	<b>aarpostincrement</b>	-/w	<b>0</b> , post-increment is not supported
18	<b>postexec</b>	-/w	if set the program buffer is executed <i>after</i> the command
17	<b>transfer</b>	-/w	if set the operation in <b>write</b> is conducted
16	<b>write</b>	-/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	<b>regno</b>	-/w	GPR-access only; has to be <b>0x1000 - 0x101f</b>

### abstractauto

#### 0x18 Abstract command auto-execution

abstractauto

Reset value: **0x00000000**

Register to configure when a read/write access to a DM repeats execution of the last abstract command.

*Table 83. abstractauto Register Bits*

Bit	Name [RISC-V]	R/W	Description
17	<b>autoexecprogbuf[1]</b>	r/w	when set reading/writing from/to <b>progbuf1</b> will execute <b>command</b> again
16	<b>autoexecprogbuf[0]</b>	r/w	when set reading/writing from/to <b>progbuf0</b> will execute <b>command</b> again
0	<b>autoexecdata[0]</b>	r/w	when set reading/writing from/to <b>data0</b> will execute <b>command</b> again

## proobuf

0x20	<b>Program buffer 0</b>	probuf0
0x21	<b>Program buffer 1</b>	probuf1

Reset value: `0x00000013` ("NOP")

Program buffer (two entries) for the DM. Note that this register is read-only for the DM (allowed since spec. version 1.0)!

### 5.2.2. DM CPU Access

From the CPU's perspective, the DM behaves as a memory-mapped peripheral that includes the following sub-modules:

- a small ROM that contains the code for the "park loop", which is executed when the CPU is *in* debug mode
- a program buffer populated by the debugger host to execute small programs
- a data buffer to transfer data between the processor and the debugger host
- a status register to communicate debugging requests and status

The DM 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 to fill the entire section.

Table 84. DM CPU access - address map (divided into four sections)

Base address	Name [VHDL package]	Actual size	Description
<code>0xfffff800</code>	<code>dm_code_base_c</code> (= <code>dm_base_c</code> )	64 bytes	ROM for the "park loop" code
<code>0xfffff840</code>	<code>dm_pbuf_base_c</code>	16 bytes	Program buffer, provided by DM
<code>0xfffff880</code>	<code>dm_data_base_c</code>	4 bytes	Data buffer ( <code>dm.data0</code> )
<code>0xfffff8c0</code>	<code>dm_sreg_base_c</code>	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`. Please note, that these sources are not intended to be changed by the user. Hence, the makefile does not provide an automatic option to compile and "install" the debugger ROM code into the HDL sources and require a manual copy

## 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 request has been issued (for example a halt request) or when an exception has occurred *while executing* the park loop itself.

Table 85. 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 address of the *normal entry point* for the park loop code defined by the `CPU_DEBUG_PARK_ADDR` 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* defined by 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 host-controlled debug module (DM). 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* instruction. This eliminates the need to perform bit-masking in the park loop code leading to less code size and faster execution.

Table 86. DM Status Register - CPU Access

Bit	Name	CPU access	Description
0	<code>sreg_halt_ack</code>	read	<i>this bit is write-only</i>
		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
		write	Set by the CPU before it starts resuming

Bit	Name	CPU access	Description
16	sreg_execute_req	read	Set by the DM to request execution of the program buffer
	sreg_execute_ack	write	Set by the CPU before it starts executing the program buffer
24	-	read	<i>this bit is write-only</i>
	sreg_execute_ack	write	Set by the CPU if an exception occurs while being in debug mode

## 5.3. CPU Debug Mode

The NEORV32 CPU Debug Mode `DB` or `DEBUG` is compatible to the **Minimal RISC-V Debug Specification 1.0** `Sdext` (external debug) ISA extension. When enabled via the `Sdext ISA Extension` generic (CPU) and/or the `ON_CHIP_DEBUGGER_EN` (Processor) it adds a new CPU operation mode ("debug mode"), three additional CSRs (section **CPU Debug Mode CSRs**) and one additional instruction (`dret`) to the core.



The CPU debug mode requires the `Zicsr` and `Zifencei` CPU extension to be implemented (top generics `Zicsr ISA Extension` and `Zifencei ISA Extension` = true).

The CPU debug-mode is entered on any of the following events:

1. executed `ebreak` instruction (when in machine-mode and `dcsr` .ebreakm`` is set OR when in user-mode and `dcsr` .ebreaku`` is set)
2. debug halt request from external DM (via CPU signal `db_halt_req_i`, high-active, triggering on rising-edge)
3. finished executing of a single instruction while in single-step debugging mode (enabled via `dcsr` .step``)
4. hardware trigger by the **Trigger Module**



From a hardware point of view, these "entry conditions" are special traps 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
- move `pc` to `dpc`
- copy the hart's current privilege level to `dcsr .prv`
- set `dcrs.cause` 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 `pc` to jump 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
- 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; note that the [Machine System Timer \(MTIME\)](#) keep running as well as its shadowed copies in the `[m]time[h]` CSRs
- 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 `dcsr.prv`
- restore `pc` from `dpc`
- resume normal operation at `pc`

### 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.



The debug-mode 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`

0x7b0     **Debug control and status register**

`dcsr`

Reset value: `0x40000413`

The `dcsr` CSR is compatible to the RISC-V debug spec. It is used to configure debug mode and provides additional status information.

*Table 87. Debug control and status register `dcsr` bits*

Bit	Name [RISC-V]	R/W	Description
31:28	xdebugver	r/-	0100 - CPU debug mode is compatible to spec. version 1.0
27:16	-	r/-	000000000000 - reserved
15	ebreakm	r/w	ebreak instructions in machine mode will enter debug mode when set
14	ebreakh	r/-	0 - hypervisor mode not supported
13	ebreaks	r/-	0 - supervisor mode not supported
12	ebreaku	r/w	ebreak instructions in user mode will enter debug mode when set
11	stepie	r/-	0 - IRQs are disabled during single-stepping
10	stopcount	r/-	1 - standard counters and HPMs are stopped when in debug mode
9	stoptime	r/-	0 - timers increment as usual
8:6	cause	r/-	cause identifier - why debug mode was entered (see below)
5	-	r/-	0 - reserved
4	mprvn	r/-	1 - mstatus `.mprv` is also evaluated when in debug mode
3	nmiip	r/-	0 - non-maskable interrupt is pending
2	step	r/w	enable single-stepping when set
1:0	prv	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

0x7b1    **Debug program counter**

dpc

Reset value: 0x00000000

The `dcsr` CSR is compatible to the RISC-V debug spec. It is used to store the current program counter when debug mode is entered. The `dret` instruction will return to `dpc` by moving `dpc` to `pc`.

### dscratch0

0x7b2    **Debug scratch register 0**

dscratch0

Reset value: 0x00000000

The `dscratch0` CSR is compatible to the RISC-V debug spec. It provides a general purpose debug mode-only scratch register.

## 5.4. Trigger Module

The RISC-V [Sdtrig](#) ISA extension add a programmable *trigger module* to the processor when enabled (via the [Sdtrig ISA Extension](#)). The NEORV32 trigger module implements a subset of the features described in the "RISC-V Debug Specification / Trigger Module".

The trigger module only provides a *single* trigger supporting only the "instruction address match" type. This limitation is granted by the RISC-V specs. and is sufficient to **debug code executed from read-only memory (ROM)**. "Normal" *software* breakpoints (using GDB's `b/break` command) are implemented by temporarily replacing the according instruction word by an `ebreak` instruction. 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 \(XIP\)](#)). Therefore, the NEORV32 trigger module provides a single "instruction address match" trigger to enter debug mode when executing the instruction at a specific address. These "hardware-assisted breakpoints" are used by GDB's `hb/hbreak` command.

The trigger module can also be used independently of the CPU debug-mode (so independent of the on-chip debugger). Machine-mode software can use the trigger module to raise a Break exception when the instruction at a programmable address gets executed.

### 5.4.1. Trigger Module CSRs

The [Sdtrig](#) ISA extension add 8 additional CSRs, which are accessible in debug-mode and also in machine-mode. Machine-level accesses can be ignore by setting `tdata1`.dmode``. This is automatically done by GDB if it uses the trigger module for implementing a "hardware breakpoint"

#### [tselect](#)

---

0x7a0    **Trigger select register**

[tselect](#)

Reset value: `0x00000000`

This CSR is hardwired to zero indicating there is only one trigger available. Any write access is ignored.

---

#### [tdata1](#)

---

0x7a1    **Trigger data register 1 / match control register**

[tdata1 / mcontrol](#)

Reset value: `0x20040040`

This CSR is used to configure the address match trigger. Only one bit is writable, the remaining bits are hardwired (see table below). Write attempts to the hardwired bits are ignored.

---

Table 88. Match Control CSR ([tdata1](#)) Bits

Bit	Name [RISC-V]	R/W	Description
31:28	<a href="#">type</a>	r/-	<code>0010</code> - address match trigger
27	<a href="#">dmode</a>	r/w	set to ignore <a href="#">tdata*</a> CSR accesses from machine-mode

Bit	Name [RISC-V]	R/W	Description
26:21	<code>maskmax</code>	r/-	<code>000000</code> - only exact values
20	<code>hit</code>	r/-	<code>0</code> - feature not supported
19	<code>select</code>	r/-	<code>0</code> - fire trigger on address match
18	<code>timing</code>	r/-	<code>1</code> - trigger <b>after</b> executing the triggering instruction
17:16	<code>sizelo</code>	r/-	<code>00</code> - match against an access of any size
15:12	<code>action</code>	r/w	<code>0001</code> = enter debug-mode on trigger fire; <code>0000</code> = normal m-mode break exception on trigger fire
11	<code>chain</code>	r/-	<code>0</code> - chaining is not supported - there is only one trigger
10:6	<code>match</code>	r/-	<code>0000</code> - only full-address match
6	<code>m</code>	r/-	<code>1</code> - trigger enabled when in machine-mode
5	<code>h</code>	r/-	<code>0</code> - hypervisor-mode not supported
4	<code>s</code>	r/-	<code>0</code> - supervisor-mode not supported
3	<code>u</code>	r/-	trigger enabled when in user-mode, set when <code>U</code> ISA extension is enabled
2	<code>exe</code>	r/w	set to enable trigger
1	<code>store</code>	r/-	<code>0</code> - store address/data matching not supported
0	<code>load</code>	r/-	<code>0</code> - load address/data matching not supported

## tdata2

0x7a2 Trigger data register 2

tdata2

Reset value: `0x00000000`

Since only the "address match trigger" type is supported, this r/w CSR is used to configure the address of the triggering instruction.

## tdata3

0x7a3 Trigger data register 3

tdata3

Reset value: `0x00000000`

This CSR is not required for the NEORV32 trigger module. Hence, it is hardwired to zero and any write access is ignored.

## tinfo

0x7a4 Trigger information register

tinfo

Reset value: `0x00000004`

This CSR is hardwired to "4" indicating there is only an "address match trigger" available. Any write access is ignored.

### tcontrol

0x7a5    **Trigger control register**

tcontrol

Reset value: **0x00000000**

This CSR is not required for the NEORV32 trigger module. Hence, it is hardwired to zero and any write access is ignored.

### mcontext

0x7a8    **Machine context register**

mcontext

Reset value: **0x00000000**

This CSR is not required for the NEORV32 trigger module. Hence, it is hardwired to zero and any write access is ignored.

### scontext

0x7aa    **Supervisor context register**

scontext

Reset value: **0x00000000**

This CSR is not required for the NEORV32 trigger module. Hence, it is hardwired to zero and any write access is ignored.

# Chapter 6. Legal

## License

### BSD 3-Clause License

Copyright (c) 2023, 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.

**The NEORV32 RISC-V Processor**

HQ: <https://github.com/stnolting/neorv32>

By Dipl.-Ing. (M.Sc.) Stephan Nolting

European Union, Germany

Contact: [stnolting@gmail.com](mailto:stnolting@gmail.com)

# Proprietary Notice

- "GitHub" is a Subsidiary of Microsoft Corporation.
- "Vivado" and "Artix" are trademarks of Xilinx 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 proudly made with *asciidoc*.
- "Segger Embedded Studio" and "J-Link" are trademarks of Segger Microcontroller Systems GmbH.
- All further/unreferenced products belong to their according copyright holders.

PDF icons from <https://www.flaticon.com> and made by **Freepik**, **Good Ware**, **Pixel perfect**, **Vectors Market**

# Disclaimer

This project is released under the BSD 3-Clause license. No copyright infringement intended. Other implied or used projects 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 ☺

Please add as many [contributors](#) as possible to the [author](#) field.  
This project would not be where it is without them.

If you are using the NEORV32 or parts of the project in some kind of publication, please cite it as follows:

*Listing 23. BibTeX*

```
@misc{nolting22,
  author      = {Nolting, S. and ...},
  title       = {The NEORV32 RISC-V Processor},
  year        = {2022},
  publisher   = {GitHub},
  journal     = {GitHub repository},
  howpublished = {\url{https://github.com/stnolting/neorv32}}
}
```



### DOI

This project also provides a *digital object identifier* provided by [zenodo](#):  
DOI 10.5281/zenodo.5018888

## Acknowledgments

A big shout-out to the community and all [contributors](#), who helped improving this project! ☺

[RISC-V](#) - instruction sets want to be free!

Continuous integration provided by [GitHub Actions](#) and powered by [GHDL](#).