



The NEORV32 RISC-V Processor - Datasheet

Version v1.13.3-r63-g99ef4e9f



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	7
1.1. Project Key Features	8
1.2. Project Folder Structure	10
1.3. VHDL File Hierarchy	11
1.3.1. File-List File	12
1.4. VHDL Coding Style	14
1.5. Performance	15
1.6. RISC-V Compatibility	15
1.7. RISC-V Documentation	28
2. NEORV32 Processor (SoC)	29
2.1. Processor Top Entity - Generics	31
2.2. Processor Top Entity - Signals	38
2.3. Processor Clocking	42
2.3.1. Peripheral Clocks	42
2.4. Processor Reset	43
2.5. Processor Interrupts	44
2.5.1. RISC-V Standard Interrupts	44
2.5.2. NEORV32-Specific Fast Interrupt Requests	44
2.6. Address Space	46
2.6.1. Bus System	47
2.6.2. Bus Gateway	49
2.6.3. IO Switch	49
2.6.4. Atomic Memory Operations Controller	50
2.6.5. Memory Coherence	51
2.7. Boot Configuration	53
2.7.1. Booting via Bootloader	53
2.7.2. Boot from Custom Address	53
2.7.3. Boot IMEM Image	53
2.8. Processor-Internal Modules	55
2.8.1. Instruction Memory (IMEM)	56
2.8.2. Data Memory (DMEM)	58
2.8.3. Bootloader ROM (BOOTROM)	60
2.8.4. Instruction Cache (iCache)	61
2.8.5. Data Cache (dCache)	63
2.8.6. Serial Memory Controller (SMC)	66
2.8.7. Direct Memory Access Controller (DMA)	70
2.8.8. Processor-External Bus Interface (XBUS)	74

2.8.9. Stream Link Interface (SLINK)	79
2.8.10. General Purpose Input and Output Port (GPIO)	83
2.8.11. Watchdog Timer (WDT)	86
2.8.12. Core-Local Interruptor (CLINT)	88
2.8.13. Primary Universal Asynchronous Receiver and Transmitter (UART0)	90
2.8.14. Secondary Universal Asynchronous Receiver and Transmitter (UART1)	94
2.8.15. Serial Peripheral Interface Controller (SPI)	95
2.8.16. Serial Data Interface Controller (SDI)	99
2.8.17. Two-Wire Serial Interface Controller (TWI)	102
2.8.18. Two-Wire Serial Device Controller (TWD)	106
2.8.19. One-Wire Serial Interface Controller (ONEWIRE)	111
2.8.20. Pulse-Width Modulation Controller (PWM)	116
2.8.21. True Random-Number Generator (TRNG)	120
2.8.22. Custom Functions Subsystem (CFS)	123
2.8.23. Smart LED Interface (NEOLED)	125
2.8.24. General Purpose Timer (GPTMR)	129
2.8.25. Execution Trace Buffer (TRACER)	132
2.8.26. System Configuration Information Memory (SYSINFO)	137
3. NEORV32 Central Processing Unit (CPU)	141
3.1. CPU Top Entity - Signals	142
3.2. CPU Top Entity - Generics	143
3.3. Architecture	144
3.3.1. CPU Front-End	144
3.3.2. CPU Back-End	145
3.3.3. CPU Register File	145
3.3.4. CPU Arithmetic Logic Unit	146
3.3.5. CPU Load/Store Unit	147
3.3.6. Execution Trace Port	147
3.3.7. CPU Tuning Options	149
3.3.8. Sleep Mode	151
3.3.9. Full Virtualization	152
3.4. Bus Interface	153
3.4.1. Bus Interface Protocol	154
3.4.2. Locked Bus Accesses and Bursts	155
3.5. Instruction Sets and Extensions	157
3.5.1. Latency Definitions	159
3.5.2. A ISA Extension	159
3.5.3. B ISA Extension	160
3.5.4. C ISA Extension	160

3.5.5. E ISA Extension	161
3.5.6. I ISA Extension	161
3.5.7. M ISA Extension	161
3.5.8. U ISA Extension	162
3.5.9. X ISA Extension	162
3.5.10. Zaamo ISA Extension	162
3.5.11. Zalrsc ISA Extension	163
3.5.12. Zcb ISA Extension	163
3.5.13. Zifencei ISA Extension	163
3.5.14. Zfinx ISA Extension	164
3.5.15. Zibi ISA Extension	164
3.5.16. Zicntr ISA Extension	165
3.5.17. Zicond ISA Extension	165
3.5.18. Zicsr ISA Extension	165
3.5.19. Zihpm ISA Extension	166
3.5.20. Zimop ISA Extension	166
3.5.21. Zcmop ISA Extension	166
3.5.22. Zba ISA Extension	166
3.5.23. Zbb ISA Extension	167
3.5.24. Zbc ISA Extension	167
3.5.25. Zbs ISA Extension	168
3.5.26. Zbkb ISA Extension	168
3.5.27. Zbkc ISA Extension	168
3.5.28. Zbkx ISA Extension	169
3.5.29. Zkn ISA Extension	169
3.5.30. Zknd ISA Extension	169
3.5.31. Zkne ISA Extension	169
3.5.32. Zknh ISA Extension	170
3.5.33. Zks ISA Extension	170
3.5.34. Zksed ISA Extension	170
3.5.35. Zksh ISA Extension	171
3.5.36. Zkt ISA Extension	171
3.5.37. Zmmul - ISA Extension	172
3.5.38. Xcfu ISA Extension	172
3.5.39. Smpmp ISA Extension	172
3.5.40. Smcntrpmf - ISA Extension	173
3.5.41. Sdext ISA Extension	173
3.5.42. Sdtrig ISA Extension	174
3.6. Custom Functions Unit (CFU)	175

3.6.1. CFU Instruction Formats	175
3.6.2. Using Custom Instructions in Software	176
3.6.3. Custom Instructions Hardware	177
3.7. Control and Status Registers (CSRs)	179
3.7.1. Floating-Point CSRs	182
3.7.2. Machine Trap Setup CSRs	184
3.7.3. Machine Trap Handling CSRs	188
3.7.4. Machine Configuration CSRs	191
3.7.5. Machine Physical Memory Protection CSRs	192
3.7.6. (Machine) Counter and Timer CSRs	194
3.7.7. Hardware Performance Monitors (HPM) CSRs	196
3.7.8. Machine Counter Setup CSRs	198
3.7.9. Machine Information CSRs	200
3.7.10. NEORV32-Specific CSRs	202
3.8. Traps, Exceptions and Interrupts	204
3.8.1. Memory Access Exceptions	204
3.8.2. Nested Interrupts	205
3.8.3. Custom Fast Interrupt Request Lines	205
3.8.4. NEORV32 Trap Listing	205
3.9. Dual-Core Configuration	209
3.9.1. Dual-Core Boot	210
4. Software Framework	212
4.1. Compiler Toolchain	212
4.2. Core Libraries	213
4.3. System View Description File (SVD)	215
4.4. Application Makefile	216
4.4.1. Makefile Targets	217
4.4.2. Default Compiler Flags	218
4.5. Linker Script	220
4.5.1. RAM Layout	221
4.5.2. ROM Layout	222
4.6. C Standard Library	223
4.7. Start-Up Code (crt0)	224
4.8. Executable Image Formats	225
4.9. Bootloader	227
4.9.1. Bootloader Console	227
4.9.2. Bootloader Image Format	229
4.9.3. Auto Boot Sequence	230
4.9.4. Uploading an Executable	231

4.9.5. Programming an SPI or TWI Flash	232
4.9.6. Booting from SD Card	232
4.9.7. Customizing the Internal Bootloader	233
4.9.8. Bootloader Error Codes	235
4.10. NEORV32 Runtime Environment	237
4.10.1. RTE Operation	237
4.10.2. Using the RTE	237
4.10.3. Default RTE Trap Handlers	239
4.10.4. Application Context Handling	240
5. On-Chip Debugger (OCD)	241
5.1. openOCD	244
5.2. Semihosting	245
5.3. Debug Transport Module (DTM)	247
5.4. Debug Module (DM)	250
5.4.1. DM Registers	251
5.4.2. DM CPU Access	256
5.5. Debug Authentication	259
5.5.1. Default Authentication Mechanism	259
5.6. CPU Debug Mode	261
5.6.1. CPU Debug Mode CSRs	262
5.7. Trigger Module	265
5.7.1. Trigger Module CSRs	265
6. Legal	269
About	269
DOI	269
License	269
Proprietary Notice	271
Disclaimer	271
Limitation of Liability for External Links	271
Acknowledgments	271

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.

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**

1.1. 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 by GHDL)
- 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
- official RISC-V open source architecture ID
- fully compatible to the RISC-V ISA specs. - checked by the official RISC-V architecture compatibility 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
- full execution tracing; compatible to the RISC-V formal interface spec (RVFI)

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 serial memory interface controller (supporting XIP)
- optional external memory interface for custom connectivity
- optional DMA controller for CPU-independent data transfers
- on-chip debugger compatible with OpenOCD and GDB including hardware trigger module and optional authentication

Software framework

- written in C and based on GCC
- 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>
- Ada support at <https://github.com/GNAT-Academic-Program/neorv32-hal>

OS Support

- FreeRTOS port: <https://github.com/stnolting/neorv32-freertos>
- Upstream Zephyr support: <https://docs.zephyrproject.org/latest/boards/others/neorv32/doc/index.html>
- MicroPython port: <https://github.com/stnolting/neorv32-micropython>

Extensibility and Customization

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

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



A more detailed comparison of the extension/customization options can be found in section **Adding Custom Hardware Modules** of the user guide.

1.2. Project Folder Structure

The root directory of the repository is considered the NEORV32 base or home folder (i.e. `neorv32/`).

Folder Structure

neorv32	- Project home folder
— docs	- Documentation
— datasheet	- AsciiDoc sources for the NEORV32 data sheet
— figures	- Figures and logos
— userguide	- AsciiDoc sources for the NEORV32 user guide
— rtl	- HDL sources
— core	- Core sources of the CPU & SoC
— system_integration	- System wrappers and bridges for advanced SoC
connectivity	
— test_setups	- Minimal test setup "SoCs" used in the User Guide
— verilog	- Scripts and examples for converting NEORV32 to Verilog
— sim	- Simulation files
— sw	- Software framework
— bootloader	- Sources of the default builtin bootloader
— common	- Linker script, crt0.S start-up code and central makefile
— example	- Example programs for the core and the SoC modules
— eclipse	- Pre-configured Eclipse IDE project
— ...	- Several example programs
— image_gen	- Helper program to generate executables & memory images
— lib	- NEORV32 core library
— ocd_firmware	- Firmware for the on-chip debugger "park loop"
— openocd	- OpenOCD configuration files
— svd	- Processor system view description file (CMSIS-SVD)

1.3. VHDL File Hierarchy

All required VHDL hardware source files are located in the project's **rtl/core** folder.



VHDL Library

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



Compilation Order

See section **File-List File** for more information.



Replacing Modules for Customization or Optimization (Especially Memory Modules)

Any module of the core can be replaced for customization purpose. For ASIC implementations (and for some FPGA implementations), this is particularly recommended for the underlying memories. Therefore, all (larger) memories are encapsulated as easily replaceable IP wrappers (i.e. ***_ram.vhd** and ***_rom.vhd** files). By default, these wrappers use a generic memory description that should infer blockRAM on most FPGA platforms.

RTL File List (in alphabetical order)

```
rtl/core
|
|— neorv32_bootrom.vhd          - Bootloader ROM
|— neorv32_bootrom_image.vhd   - Bootloader ROM initialization image
|— neorv32_bootrom_rom.vhd     - Bootloader ROM primitive wrapper
|— neorv32_bus.vhd             - SoC bus infrastructure modules
|— neorv32_cache.vhd           - Generic cache module
|— neorv32_cache_ram.vhd       - Cache tag and data RAM primitive wrapper
|— neorv32_cfs.vhd             - Custom functions subsystem
|— neorv32_clint.vhd           - Core local interruptor
|— neorv32_cpu.vhd             - NEORV32 CPU TOP ENTITY
|— neorv32_cpu_alu.vhd         - Arithmetic/logic unit
|— neorv32_cpu_control.vhd     - CPU control, exception system and CSRs
|— neorv32_cpu_counters.vhd    - Hardware counters (Zicntr & Zihpm ext.)
|— neorv32_cpu_alu_bitmanip.vhd - Bit-manipulation unit (B ext.)
|— neorv32_cpu_alu_cfu.vhd     - Custom instructions unit (Xcfu ext.)
|— neorv32_cpu_alu_cond.vhd    - Integer conditional unit (Zicond ext.)
|— neorv32_cpu_alu_crypto.vhd  - Scalar cryptography unit (Zk*/Zbk* ext.)
|— neorv32_cpu_alu_fpu.vhd     - Floating-point unit (Zfinx ext.)
|— neorv32_cpu_alu_muldiv.vhd  - Integer MUL/DIV unit (M ext.)
|— neorv32_cpu_alu_shifter.vhd - Bit-shift unit (base ISA)
|— neorv32_cpu_decompressor.vhd - Compressed instructions decoder (C ext.)
|— neorv32_cpu_frontend.vhd    - Instruction fetch and issue
|— neorv32_cpu_hwtrig.vhd      - Hardware trigger module (Sdtrig ext.)
|— neorv32_cpu_lsu.vhd         - Load/store unit
```

— neorv32_cpu_pmp.vhd	- Physical memory protection unit (Smpmp ext.)
— neorv32_cpu_regfile.vhd	- Data register file
— neorv32_cpu_trace.vhd	- Trace generator
— neorv32_debug_auth.vhd	- On-chip debugger: authentication module
— neorv32_debug_dm.vhd	- On-chip debugger: debug module
— neorv32_debug_dtm.vhd	- On-chip debugger: debug transfer module
— neorv32_dma.vhd	- Direct memory access controller
— neorv32_dmem.vhd	- Data memory
— neorv32_dmem_ram.vhd	- Data memory RAM primitive wrapper
— neorv32_gpio.vhd	- General purpose input/output port unit
— neorv32_gptmr.vhd	- General purpose 32-bit timer
— neorv32_imem.vhd	- Instruction memory
— neorv32_imem_image.vhd	- Instruction memory initialization image
— neorv32_imem_ram.vhd	- Instruction memory RAM primitive wrapper
— neorv32_imem_rom.vhd	- Instruction memory ROM primitive wrapper
— neorv32_neoled.vhd	- NeoPixel (TM) compatible smart LED interface
— neorv32_owewire.vhd	- One-Wire serial interface controller
— neorv32_package.vhd	- Main VHDL package file
— neorv32_prim.vhd	- Generic RTL primitives
— neorv32_pwm.vhd	- Pulse-width modulation controller
— neorv32_sdi.vhd	- Serial data interface controller (SPI device)
— neorv32_slink.vhd	- Stream link interface
— neorv32_smc.vhd	- Serial memory controller
— neorv32_spi.vhd	- Serial peripheral interface controller (SPI host)
— neorv32_sys.vhd	- System infrastructure modules
— neorv32_sysinfo.vhd	- System configuration information memory
— neorv32_top.vhd	- NEORV32 PROCESSOR/SOC TOP ENTITY
— neorv32_tracer.vhd	- Instruction trace buffer
— neorv32_trng.vhd	- True random number generator
— neorv32_twd.vhd	- Two wire serial device controller
— neorv32_twi.vhd	- Two wire serial interface controller
— neorv32_uart.vhd	- Universal async. receiver/transmitter
— neorv32_wdt.vhd	- Watchdog timer
— neorv32_xbus.vhd	- External bus interface gateway

1.3.1. File-List File

Most of the RTL sources use **entity instantiation**. Hence, the RTL compile order might be relevant (depending on the synthesis/simulation tool. Therefore, a file-list file is provided in the **rtl** folder that list all required HDL files for the entire processor and also represent their recommended compile order. Note that this file lists the architecture "bottom-up" - the according top entity is listed in the very last line. The file-list file can be consumed by EDA tools to simplify project setup.

- **file_list_core.f** - HDL files and compile order for the entire processor/SoC; top module: **neorv32_top**

A simple bash script **file_list_regenerate.sh** is provided for regenerating the file-list (using GHDL's *elaborate* command). This script can also be invoked using the default application makefile (see

Makefile Targets).

By default, the file-list file includes a **placeholder** (`$NEORV32_HOME`) in the path of each included hardware source file. These placeholders need to be replaced/substituted by the actual path before being used. Example:

- default: `$NEORV32_HOME/rtl/core/neorv32_package.vhd`
- substituted: `path/to/neorv32/rtl/core/neorv32_package.vhd`

Example: Makefile*Listing 1. Processing the File-List Files in a Makefile*

```
NEORV32_HOME ?= path/to/neorv32 ①  
NEORV32_FFILE := $(shell cat $(NEORV32_HOME)/rtl/file_list_core.f) ②  
NEORV32_SRCS := $(subst $$NEORV32_HOME,$(NEORV32_HOME),$(NEORV32_FFILE)) ③
```

- ① Path to the NEORV32 home folder (i.e. the root folder of the GitHub repository).
- ② Load the content of the `file_list_core.f` file-list into a variable `NEORV32_SOC_FILE`.
- ③ Substitute the file-list file's path placeholder "`$NEORV32_HOME`" by the actual path and store to `NEORV32_SRCS` variable.

Example: TCL*Listing 2. Processing the File-List Files in a TCL Script*

```
set neorv32_home path/to/neorv32 ①  
set file_list_file [read [open "$neorv32_home/rtl/file_list_core.f" r]] ②  
set file_list [string map [list {$NEORV32_HOME} $neorv32_home] $file_list_file] ③
```

- ① Path to the NEORV32 home folder (i.e. the root folder of the GitHub repository).
- ② Load the content of the `file_list_core.f` file-list into a variable `file_list_file`.
- ③ Substitute the file-list file's path placeholder "`$NEORV32_HOME`" by the actual path and store to `file_list` variable.

Example: Bash*Listing 3. Processing the File-List Files in a Bash Script*

```
neorv32/rtl$ export NEORV32_HOME=path/to/neorv32 ①  
neorv32/rtl$ envsubst < file_list_core.f > file_list.f ②
```

- ① Set path to the NEORV32 home folder (i.e. the root folder of the GitHub repository) as environment variable.
- ② Substitute `$NEORV32_HOME` by the content of the according environment variable and store to `file_list.f` file.

1.4. VHDL Coding Style

- The entire processor, including the CPU core, is written in platform-/technology-independent VHDL. The code makes minimal use of VHDL 2008 features in order to remain compatible with older EDA tools.
- A specific VHDL library **neorv32** is used for all sources.
- All registers in the control path use an asynchronous reset. The registers in the data path do not have a dedicated reset if their state is gated by a "valid" signal. This allows using of smaller FFs and relaxes the reset routing.
- The entire setup uses a single clock domain. External "clock" signals are synchronized into this clock domain using 2-stage shift registers.
- A single package/library file (**neorv32_package.vhd**) is used to provide global definitions and auxiliary functions. The user-defined configuration is done entirely via the generics of the top entity.
- Internally, all generics are checked to ensure correct configuration. Asserts are used to inform the user about the actual processor configuration and possible invalid settings.
- The code uses entity instantiation for all internal modules. However, if multiple submodules are specified within the same source file, component instantiation is used for them. * When instantiating the top-level processor module (**neorv32_top.vhd**) in a user-defined design, either entity instantiation or component instantiation can be used, as the NEORV32 package file/library file already provides a corresponding component declaration.
- Larger memories (IMEM, DMEM, caches) are implemented as encapsulated wrappers, which are instantiated by component instantiation. The interfaces of these blocks are static (fixed signal widths; not dependent on the configuration). This means they can be easily replaced by corresponding Verilog IP wrappers (also via black-box instantiation and late binding).



All-Verilog Setup

Scripts and instructions for an automatic to-Verilog conversion as well as an all-Verilog setup that uses custom Verilog memory IPs can be found in **rtl/verilog**. See **UG: NEORV32 in Verilog** for more information.

1.5. Performance

Area Utilization

The NEORV32 processor is optimized for minimal size. However, the actual size (silicon area or FPGA resources) depends on the specific configuration. For example, an RTOS-capable setup based on a `rv32imc_Zicsr_Zicntr` CPU configuration including peripheral and memories requires about 2300 LUTs and 1000 FFs and can run at up to 130 MHz (implementation results for an Altera Cyclone IV E `EP4CE22F17C6` FPGA).



NEORV32 Setups

The processor has been successfully ported to AMD, Altera, Lattice, Microchip, Gowin, Cologne Chip and NanoXplore FPGAs. Some pre-configured example setup are available online: <https://github.com/stnolting/neorv32-setups>

Processing Speed

The computational performance of the NEORV32 is evaluated using **Core Mark CPU benchmark**. The NEORV32 CoreMark port can be found in a separate repository: [neorv32-coremark](#)

Table 1. CoreMark Results

CPU Configuration	CoreMark Score	CoreMarks/MHz
<i>small</i> (<code>rv32i_Zicsr_Zifencei</code>)	33.89	0.33
<i>medium</i> (<code>rv32imc_Zicsr_Zifencei</code>)	62.50	0.62
<i>performance</i> (<code>rv32imc_Zicsr_Zifencei</code> + perf. options)	95.23	0.95
<i>dual-core</i> (<code>rv32imc_Zicsr_Zifencei</code> + perf. options)	190.47	1.90

Performance can be further improved by enabling additional ISA extensions. Note that the compiler also needs to support these extensions (make sure to use a **multilib** GCC configuration for optimal code results).

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, tuning options and memory latency.

1.6. RISC-V Compatibility

The NEORV32 CPU passes the official **RISC-V Architectural Certification Tests (ACTs)** to "*certify it faithfully implements the RISC-V specification*". The NEORV32 port of this test framework is available at <https://github.com/stnolting/neorv32-riscv-act>.

<details>

<summary>ACT Results for NEORV32 v1.13.3.1, 24th of July, 2026 (click to expand)</summary>

<pre>

RESULT: All 301 tests passed.

priv/ExceptionsSm/ExceptionsSm-00.log

RVCP-SUMMARY: TEST PASSED - Test File

"ExceptionsSm-00.S"

priv/ExceptionsU/ExceptionsU-00.log

RVCP-SUMMARY: TEST PASSED - Test File

"ExceptionsU-00.S"

priv/ExceptionsZaamo/ExceptionsZaamo-00.log

RVCP-SUMMARY: TEST PASSED - Test File

"ExceptionsZaamo-00.S"

priv/ExceptionsZalrsc/ExceptionsZalrsc-00.log

RVCP-SUMMARY: TEST PASSED - Test File

"ExceptionsZalrsc-00.S"

priv/ExceptionsZc/ExceptionsZc-00.log

RVCP-SUMMARY: TEST PASSED - Test File

"ExceptionsZc-00.S"

priv/ExceptionsZicboU/ExceptionsZicboU-00.log

RVCP-SUMMARY: TEST PASSED - Test File

"ExceptionsZicboU-00.S"

priv/InterruptsSm/InterruptsSm-00.log

RVCP-SUMMARY: TEST PASSED - Test File

"InterruptsSm-00.S"

priv/InterruptsU/InterruptsU-00.log

RVCP-SUMMARY: TEST PASSED - Test File

"InterruptsU-00.S"

priv/SsstrictSm/SsstrictSm-12.log

RVCP-SUMMARY: TEST PASSED - Test File

"SsstrictSm-12.S"

priv/SsstrictSm/SsstrictSm-13.log

RVCP-SUMMARY: TEST PASSED - Test File

"SsstrictSm-13.S"

priv/SsstrictSm/SsstrictSm-14.log

RVCP-SUMMARY: TEST PASSED - Test File

"SsstrictSm-14.S"

priv/U/U-00.log

RVCP-SUMMARY: TEST PASSED - Test File

"U-00.S"

priv/ZicntrU/ZicntrU-00.log

RVCP-SUMMARY: TEST PASSED - Test File

"ZicntrU-00.S"

priv/pmp/pmp32/PMPSm/pmpsm_all_entries_check.log

RVCP-SUMMARY: TEST PASSED - Test File

"pmpsm_all_entries_check.S"

priv/pmp/pmp32/PMPSm/pmpsm_cfg_A_all.log

RVCP-SUMMARY: TEST PASSED - Test File

"pmpsm_cfg_A_all.S"

priv/pmp/pmp32/PMPSm/pmpsm_cfg_A_off_all.log

RVCP-SUMMARY: TEST PASSED - Test File

"pmpsm_cfg_A_off_all.S"

priv/pmp/pmp32/PMPSm/pmpsm_cfg_A_tor_bot.log

RVCP-SUMMARY: TEST PASSED - Test File

"pmpsm_cfg_A_tor_bot.S"

priv/pmp/pmp32/PMPSm/pmpsm_cfg_A_tor_zero.log

RVCP-SUMMARY: TEST PASSED - Test File

"pmpsm_cfg_A_tor_zero.S"

priv/pmp/pmp32/PMPSm/pmpsm_cfg_L_access_all.log

RVCP-SUMMARY: TEST PASSED - Test File

"pmpsm_cfg_L_access_all.S"

priv/pmp/pmp32/PMPSm/pmpsm_cfg_L_modify_napot.log

RVCP-SUMMARY: TEST PASSED - Test File

"pmpsm_cfg_L_modify_napot.S"

priv/pmp/pmp32/PMPSm/pmpsm_cfg_L_modify_off.log

RVCP-SUMMARY: TEST PASSED - Test File

"pmpsm_cfg_L_modify_off.S"

priv/pmp/pmp32/PMPSm/pmpsm_cfg_L_modify_tor.log

RVCP-SUMMARY: TEST PASSED - Test File

"pmpsm_cfg_L_modify_tor.S"

priv/pmp/pmp32/PMPSm/pmpsm_cfg_XWR_all-01.log

RVCP-SUMMARY: TEST PASSED - Test File

"pmpsm_cfg_XWR_all-01.S"

priv/pmp/pmp32/PMPSm/pmpsm_cfg_XWR_all-02.log

RVCP-SUMMARY: TEST PASSED - Test File

"pmpsm_cfg_XWR_all-02.S"	
priv/pmp/pmp32/PMPSm/pmpsm_cfg_XWR_all-03.log	RVCP-SUMMARY: TEST PASSED - Test File
"pmpsm_cfg_XWR_all-03.S"	
priv/pmp/pmp32/PMPSm/pmpsm_cfg_XWR_all-04.log	RVCP-SUMMARY: TEST PASSED - Test File
"pmpsm_cfg_XWR_all-04.S"	
priv/pmp/pmp32/PMPSm/pmpsm_cfg_na4_all.log	RVCP-SUMMARY: TEST PASSED - Test File
"pmpsm_cfg_na4_all.S"	
priv/pmp/pmp32/PMPSm/pmpsm_cfg_napot_all.log	RVCP-SUMMARY: TEST PASSED - Test File
"pmpsm_cfg_napot_all.S"	
priv/pmp/pmp32/PMPSm/pmpsm_cfg_tor_all.log	RVCP-SUMMARY: TEST PASSED - Test File
"pmpsm_cfg_tor_all.S"	
priv/pmp/pmp32/PMPSm/pmpsm_cfg_tor_check-01.log	RVCP-SUMMARY: TEST PASSED - Test File
"pmpsm_cfg_tor_check-01.S"	
priv/pmp/pmp32/PMPSm/pmpsm_cfg_tor_check-02.log	RVCP-SUMMARY: TEST PASSED - Test File
"pmpsm_cfg_tor_check-02.S"	
priv/pmp/pmp32/PMPSm/pmpsm_cfg_tor_check-03.log	RVCP-SUMMARY: TEST PASSED - Test File
"pmpsm_cfg_tor_check-03.S"	
priv/pmp/pmp32/PMPSm/pmpsm_csr_walk-1.log	RVCP-SUMMARY: TEST PASSED - Test File
"pmpsm_csr_walk-01.S"	
priv/pmp/pmp32/PMPSm/pmpsm_csr_walk-10.log	RVCP-SUMMARY: TEST PASSED - Test File
"pmpsm_csr_walk-10.S"	
priv/pmp/pmp32/PMPSm/pmpsm_csr_walk-2.log	RVCP-SUMMARY: TEST PASSED - Test File
"pmpsm_csr_walk-02.S"	
priv/pmp/pmp32/PMPSm/pmpsm_csr_walk-3.log	RVCP-SUMMARY: TEST PASSED - Test File
"pmpsm_csr_walk-03.S"	
priv/pmp/pmp32/PMPSm/pmpsm_csr_walk-4.log	RVCP-SUMMARY: TEST PASSED - Test File
"pmpsm_csr_walk-04.S"	
priv/pmp/pmp32/PMPSm/pmpsm_csr_walk-5.log	RVCP-SUMMARY: TEST PASSED - Test File
"pmpsm_csr_walk-05.S"	
priv/pmp/pmp32/PMPSm/pmpsm_csr_walk-6.log	RVCP-SUMMARY: TEST PASSED - Test File
"pmpsm_csr_walk-06.S"	
priv/pmp/pmp32/PMPSm/pmpsm_csr_walk-7.log	RVCP-SUMMARY: TEST PASSED - Test File
"pmpsm_csr_walk-07.S"	
priv/pmp/pmp32/PMPSm/pmpsm_csr_walk-8.log	RVCP-SUMMARY: TEST PASSED - Test File
"pmpsm_csr_walk-08.S"	
priv/pmp/pmp32/PMPSm/pmpsm_csr_walk-9.log	RVCP-SUMMARY: TEST PASSED - Test File
"pmpsm_csr_walk-09.S"	
priv/pmp/pmp32/PMPSm/pmpsm_grain.log	RVCP-SUMMARY: TEST PASSED - Test File
"pmpsm_grain.S"	
priv/pmp/pmp32/PMPSm/pmpsm_grain_check.log	RVCP-SUMMARY: TEST PASSED - Test File
"pmpsm_grain_check.S"	
priv/pmp/pmp32/PMPSm/pmpsm_na4_legal_lwxr.log	RVCP-SUMMARY: TEST PASSED - Test File
"pmpsm_cfg_napot_all.S"	
priv/pmp/pmp32/PMPSm/pmpsm_napot_legal_lwxr.log	RVCP-SUMMARY: TEST PASSED - Test File
"pmpsm_napot_legal_lwxr.S"	
priv/pmp/pmp32/PMPSm/pmpsm_priority.log	RVCP-SUMMARY: TEST PASSED - Test File
"pmpsm_priority.S"	
priv/pmp/pmp32/PMPSm/pmpsm_priority_off.log	RVCP-SUMMARY: TEST PASSED - Test File
"pmpsm_priority_off.S"	

priv/pmp/pmp32/PMPsm/pmpsm_tor_legal_lwxr-01.log "pmpsm_tor_legal_lwxr.S"	RVCP-SUMMARY: TEST PASSED - Test File
priv/pmp/pmp32/PMPsm/pmpsm_tor_legal_lwxr-02.log "pmpsm_tor_legal_lwxr.S"	RVCP-SUMMARY: TEST PASSED - Test File
priv/pmp/pmp32/PMPU/pmpu_cfg_A_off.log "pmpu_cfg_A_all.S"	RVCP-SUMMARY: TEST PASSED - Test File
priv/pmp/pmp32/PMPU/pmpu_cfg_XWR.log "pmpu_cfg_XWR.S"	RVCP-SUMMARY: TEST PASSED - Test File
priv/pmp/pmp32/PMPU/pmpu_cfg_XWR_unlocked.log "pmpu_cfg_XWR_unlocked.S"	RVCP-SUMMARY: TEST PASSED - Test File
priv/pmp/pmp32/PMPU/pmpu_csr_access.log "pmpu_csr_access.S"	RVCP-SUMMARY: TEST PASSED - Test File
priv/pmp/pmp32/PMPU/pmpu_mprv_check-01.log "pmpu_mprv_check-01.S"	RVCP-SUMMARY: TEST PASSED - Test File
priv/pmp/pmp32/PMPU/pmpu_mprv_check-02.log "pmpu_mprv_check-02.S"	RVCP-SUMMARY: TEST PASSED - Test File
priv/pmp/pmp32/PMPU/pmpu_napot_legal_lxwr-01.log "pmpu_napot_legal_lxwr.S"	RVCP-SUMMARY: TEST PASSED - Test File
priv/pmp/pmp32/PMPU/pmpu_napot_legal_lxwr-02.log "pmps_napot_legal_lxwr.S"	RVCP-SUMMARY: TEST PASSED - Test File
priv/pmp/pmp32/PMPU/pmpu_tor_legal_lxwr-01.log "pmpu_tor_legal_lxwr.S"	RVCP-SUMMARY: TEST PASSED - Test File
priv/pmp/pmp32/PMPU/pmpu_tor_legal_lxwr-02.log "pmpu_tor_legal_lxwr.S"	RVCP-SUMMARY: TEST PASSED - Test File
priv/pmp/pmp32/PMPZaamo/pmpzaamo_cfg_wr.log "pmpzaamo_cfg_wr.S"	RVCP-SUMMARY: TEST PASSED - Test File
priv/pmp/pmp32/PMPZalrsc/pmpzalrsc_cfg_wr.log "pmpzalrsc_cfg_wr.S"	RVCP-SUMMARY: TEST PASSED - Test File
priv/pmp/pmp32/PMPZca/pmpzca_aligned_na4.log "pmpzca_aligned_na4.S"	RVCP-SUMMARY: TEST PASSED - Test File
priv/pmp/pmp32/PMPZca/pmpzca_aligned_napot.log "pmpzca_aligned_napot.S"	RVCP-SUMMARY: TEST PASSED - Test File
priv/pmp/pmp32/PMPZca/pmpzca_aligned_off.log "pmpzca_aligned_off.S"	RVCP-SUMMARY: TEST PASSED - Test File
priv/pmp/pmp32/PMPZca/pmpzca_aligned_tor.log "pmpzca_aligned_tor.S"	RVCP-SUMMARY: TEST PASSED - Test File
priv/pmp/pmp32/PMPZca/pmpzca_cret_na4.log "pmpzca_cret_napot.S"	RVCP-SUMMARY: TEST PASSED - Test File
priv/pmp/pmp32/PMPZca/pmpzca_cret_napot.log "pmpzca_cret_napot.S"	RVCP-SUMMARY: TEST PASSED - Test File
priv/pmp/pmp32/PMPZca/pmpzca_cret_tor.log "pmpzca_cret_napot.S"	RVCP-SUMMARY: TEST PASSED - Test File
priv/pmp/pmp32/PMPZca/pmpzca_legal_lwxr.log "pmpzca_legal_lxwr.S"	RVCP-SUMMARY: TEST PASSED - Test File
priv/pmp/pmp32/PMPZca/pmpzca_misaligned_na4.log "pmpzca_aligned_na4.S"	RVCP-SUMMARY: TEST PASSED - Test File
priv/pmp/pmp32/PMPZca/pmpzca_misaligned_napot.log "pmpzca_misaligned_napot.S"	RVCP-SUMMARY: TEST PASSED - Test File
priv/pmp/pmp32/PMPZca/pmpzca_misaligned_off.log	RVCP-SUMMARY: TEST PASSED - Test File

"pmpzca_misaligned_off.S"	
priv/pmp/pmp32/PMPZca/pmpzca_misaligned_tor.log	RVCP-SUMMARY: TEST PASSED - Test File
"pmpzca_misaligned_tor.S"	
priv/pmp/pmp32/PMPZca/pmpzcb_legal_lxwr.log	RVCP-SUMMARY: TEST PASSED - Test File
"pmpzcb_legal_lxwr.S"	
rv32i/I/I-add-00.log	RVCP-SUMMARY: TEST PASSED - Test File
"I-add-00.S"	
rv32i/I/I-addi-00.log	RVCP-SUMMARY: TEST PASSED - Test File
"I-addi-00.S"	
rv32i/I/I-and-00.log	RVCP-SUMMARY: TEST PASSED - Test File
"I-and-00.S"	
rv32i/I/I-andi-00.log	RVCP-SUMMARY: TEST PASSED - Test File
"I-andi-00.S"	
rv32i/I/I-auipc-00.log	RVCP-SUMMARY: TEST PASSED - Test File
"I-auipc-00.S"	
rv32i/I/I-beq-00.log	RVCP-SUMMARY: TEST PASSED - Test File
"I-beq-00.S"	
rv32i/I/I-bge-00.log	RVCP-SUMMARY: TEST PASSED - Test File
"I-bge-00.S"	
rv32i/I/I-bgeu-00.log	RVCP-SUMMARY: TEST PASSED - Test File
"I-bgeu-00.S"	
rv32i/I/I-bltn-00.log	RVCP-SUMMARY: TEST PASSED - Test File
"I-bltn-00.S"	
rv32i/I/I-bltnu-00.log	RVCP-SUMMARY: TEST PASSED - Test File
"I-bltnu-00.S"	
rv32i/I/I-bne-00.log	RVCP-SUMMARY: TEST PASSED - Test File
"I-bne-00.S"	
rv32i/I/I-fence-00.log	RVCP-SUMMARY: TEST PASSED - Test File
"I-fence-00.S"	
rv32i/I/I-jal-00.log	RVCP-SUMMARY: TEST PASSED - Test File
"I-jal-00.S"	
rv32i/I/I-jalr-00.log	RVCP-SUMMARY: TEST PASSED - Test File
"I-jalr-00.S"	
rv32i/I/I-lb-00.log	RVCP-SUMMARY: TEST PASSED - Test File
"I-lb-00.S"	
rv32i/I/I-lbu-00.log	RVCP-SUMMARY: TEST PASSED - Test File
"I-lbu-00.S"	
rv32i/I/I-lh-00.log	RVCP-SUMMARY: TEST PASSED - Test File
"I-lh-00.S"	
rv32i/I/I-lhu-00.log	RVCP-SUMMARY: TEST PASSED - Test File
"I-lhu-00.S"	
rv32i/I/I-lui-00.log	RVCP-SUMMARY: TEST PASSED - Test File
"I-lui-00.S"	
rv32i/I/I-lw-00.log	RVCP-SUMMARY: TEST PASSED - Test File
"I-lw-00.S"	
rv32i/I/I-nop-00.log	RVCP-SUMMARY: TEST PASSED - Test File
"I-nop-00.S"	
rv32i/I/I-or-00.log	RVCP-SUMMARY: TEST PASSED - Test File
"I-or-00.S"	

rv32i/I/I-ori-00.log	RVCP-SUMMARY: TEST PASSED - Test File
"I-ori-00.S"	
rv32i/I/I-sb-00.log	RVCP-SUMMARY: TEST PASSED - Test File
"I-sb-00.S"	
rv32i/I/I-sh-00.log	RVCP-SUMMARY: TEST PASSED - Test File
"I-sh-00.S"	
rv32i/I/I-sll-00.log	RVCP-SUMMARY: TEST PASSED - Test File
"I-sll-00.S"	
rv32i/I/I-slli-00.log	RVCP-SUMMARY: TEST PASSED - Test File
"I-slli-00.S"	
rv32i/I/I-slt-00.log	RVCP-SUMMARY: TEST PASSED - Test File
"I-slt-00.S"	
rv32i/I/I-slti-00.log	RVCP-SUMMARY: TEST PASSED - Test File
"I-slti-00.S"	
rv32i/I/I-sltiu-00.log	RVCP-SUMMARY: TEST PASSED - Test File
"I-sltiu-00.S"	
rv32i/I/I-sltu-00.log	RVCP-SUMMARY: TEST PASSED - Test File
"I-sltu-00.S"	
rv32i/I/I-sra-00.log	RVCP-SUMMARY: TEST PASSED - Test File
"I-sra-00.S"	
rv32i/I/I-srai-00.log	RVCP-SUMMARY: TEST PASSED - Test File
"I-srai-00.S"	
rv32i/I/I-srl-00.log	RVCP-SUMMARY: TEST PASSED - Test File
"I-srl-00.S"	
rv32i/I/I-srli-00.log	RVCP-SUMMARY: TEST PASSED - Test File
"I-srli-00.S"	
rv32i/I/I-sub-00.log	RVCP-SUMMARY: TEST PASSED - Test File
"I-sub-00.S"	
rv32i/I/I-sw-00.log	RVCP-SUMMARY: TEST PASSED - Test File
"I-sw-00.S"	
rv32i/I/I-xor-00.log	RVCP-SUMMARY: TEST PASSED - Test File
"I-xor-00.S"	
rv32i/I/I-xori-00.log	RVCP-SUMMARY: TEST PASSED - Test File
"I-xori-00.S"	
rv32i/M/M-div-00.log	RVCP-SUMMARY: TEST PASSED - Test File
"M-div-00.S"	
rv32i/M/M-divu-00.log	RVCP-SUMMARY: TEST PASSED - Test File
"M-divu-00.S"	
rv32i/M/M-mul-00.log	RVCP-SUMMARY: TEST PASSED - Test File
"M-mul-00.S"	
rv32i/M/M-mulh-00.log	RVCP-SUMMARY: TEST PASSED - Test File
"M-mulh-00.S"	
rv32i/M/M-mulhsu-00.log	RVCP-SUMMARY: TEST PASSED - Test File
"M-mulhsu-00.S"	
rv32i/M/M-mulhu-00.log	RVCP-SUMMARY: TEST PASSED - Test File
"M-mulhu-00.S"	
rv32i/M/M-rem-00.log	RVCP-SUMMARY: TEST PASSED - Test File
"M-rem-00.S"	
rv32i/M/M-remu-00.log	RVCP-SUMMARY: TEST PASSED - Test File

"M-remu-00.S"	
rv32i/Zaamo/Zaamo-amoadw-00.log	RVCP-SUMMARY: TEST PASSED - Test File
"Zaamo-amoadw-00.S"	
rv32i/Zaamo/Zaamo-amoadn-00.log	RVCP-SUMMARY: TEST PASSED - Test File
"Zaamo-amoadn-00.S"	
rv32i/Zaamo/Zaamo-amomax-00.log	RVCP-SUMMARY: TEST PASSED - Test File
"Zaamo-amomax-00.S"	
rv32i/Zaamo/Zaamo-amomaxu-00.log	RVCP-SUMMARY: TEST PASSED - Test File
"Zaamo-amomaxu-00.S"	
rv32i/Zaamo/Zaamo-amomin-00.log	RVCP-SUMMARY: TEST PASSED - Test File
"Zaamo-amomin-00.S"	
rv32i/Zaamo/Zaamo-amominu-00.log	RVCP-SUMMARY: TEST PASSED - Test File
"Zaamo-amominu-00.S"	
rv32i/Zaamo/Zaamo-amoor-00.log	RVCP-SUMMARY: TEST PASSED - Test File
"Zaamo-amoor-00.S"	
rv32i/Zaamo/Zaamo-amoswap-00.log	RVCP-SUMMARY: TEST PASSED - Test File
"Zaamo-amoswap-00.S"	
rv32i/Zaamo/Zaamo-amoxor-00.log	RVCP-SUMMARY: TEST PASSED - Test File
"Zaamo-amoxor-00.S"	
rv32i/Zalrsc/Zalrsc-lr-00.log	RVCP-SUMMARY: TEST PASSED - Test File
"Zalrsc-lr-00.S"	
rv32i/Zalrsc/Zalrsc-sc-00.log	RVCP-SUMMARY: TEST PASSED - Test File
"Zalrsc-sc-00.S"	
rv32i/Zba/Zba-sh1add-00.log	RVCP-SUMMARY: TEST PASSED - Test File
"Zba-sh1add-00.S"	
rv32i/Zba/Zba-sh2add-00.log	RVCP-SUMMARY: TEST PASSED - Test File
"Zba-sh2add-00.S"	
rv32i/Zba/Zba-sh3add-00.log	RVCP-SUMMARY: TEST PASSED - Test File
"Zba-sh3add-00.S"	
rv32i/Zbb/Zbb-andn-00.log	RVCP-SUMMARY: TEST PASSED - Test File
"Zbb-andn-00.S"	
rv32i/Zbb/Zbb-clz-00.log	RVCP-SUMMARY: TEST PASSED - Test File
"Zbb-clz-00.S"	
rv32i/Zbb/Zbb-cpop-00.log	RVCP-SUMMARY: TEST PASSED - Test File
"Zbb-cpop-00.S"	
rv32i/Zbb/Zbb-ctz-00.log	RVCP-SUMMARY: TEST PASSED - Test File
"Zbb-ctz-00.S"	
rv32i/Zbb/Zbb-max-00.log	RVCP-SUMMARY: TEST PASSED - Test File
"Zbb-max-00.S"	
rv32i/Zbb/Zbb-maxu-00.log	RVCP-SUMMARY: TEST PASSED - Test File
"Zbb-maxu-00.S"	
rv32i/Zbb/Zbb-min-00.log	RVCP-SUMMARY: TEST PASSED - Test File
"Zbb-min-00.S"	
rv32i/Zbb/Zbb-minu-00.log	RVCP-SUMMARY: TEST PASSED - Test File
"Zbb-minu-00.S"	
rv32i/Zbb/Zbb-orc.b-00.log	RVCP-SUMMARY: TEST PASSED - Test File
"Zbb-orc.b-00.S"	
rv32i/Zbb/Zbb-orn-00.log	RVCP-SUMMARY: TEST PASSED - Test File
"Zbb-orn-00.S"	

rv32i/Zbb/Zbb-rev8-00.log "Zbb-rev8-00.S"	RVCP-SUMMARY: TEST PASSED - Test File
rv32i/Zbb/Zbb-rol-00.log "Zbb-rol-00.S"	RVCP-SUMMARY: TEST PASSED - Test File
rv32i/Zbb/Zbb-ror-00.log "Zbb-ror-00.S"	RVCP-SUMMARY: TEST PASSED - Test File
rv32i/Zbb/Zbb-rori-00.log "Zbb-rori-00.S"	RVCP-SUMMARY: TEST PASSED - Test File
rv32i/Zbb/Zbb-sext.b-00.log "Zbb-sext.b-00.S"	RVCP-SUMMARY: TEST PASSED - Test File
rv32i/Zbb/Zbb-sext.h-00.log "Zbb-sext.h-00.S"	RVCP-SUMMARY: TEST PASSED - Test File
rv32i/Zbb/Zbb-xnor-00.log "Zbb-xnor-00.S"	RVCP-SUMMARY: TEST PASSED - Test File
rv32i/Zbb/Zbb-zext.h-00.log "Zbb-zext.h-00.S"	RVCP-SUMMARY: TEST PASSED - Test File
rv32i/Zbc/Zbc-clmul-00.log "Zbc-clmul-00.S"	RVCP-SUMMARY: TEST PASSED - Test File
rv32i/Zbc/Zbc-clmulh-00.log "Zbc-clmulh-00.S"	RVCP-SUMMARY: TEST PASSED - Test File
rv32i/Zbc/Zbc-clmulr-00.log "Zbc-clmulr-00.S"	RVCP-SUMMARY: TEST PASSED - Test File
rv32i/Zbkb/Zbkb-andn-00.log "Zbkb-andn-00.S"	RVCP-SUMMARY: TEST PASSED - Test File
rv32i/Zbkb/Zbkb-brev8-00.log "Zbkb-brev8-00.S"	RVCP-SUMMARY: TEST PASSED - Test File
rv32i/Zbkb/Zbkb-orn-00.log "Zbkb-orn-00.S"	RVCP-SUMMARY: TEST PASSED - Test File
rv32i/Zbkb/Zbkb-pack-00.log "Zbkb-pack-00.S"	RVCP-SUMMARY: TEST PASSED - Test File
rv32i/Zbkb/Zbkb-packh-00.log "Zbkb-packh-00.S"	RVCP-SUMMARY: TEST PASSED - Test File
rv32i/Zbkb/Zbkb-rev8-00.log "Zbkb-rev8-00.S"	RVCP-SUMMARY: TEST PASSED - Test File
rv32i/Zbkb/Zbkb-rol-00.log "Zbkb-rol-00.S"	RVCP-SUMMARY: TEST PASSED - Test File
rv32i/Zbkb/Zbkb-ror-00.log "Zbkb-ror-00.S"	RVCP-SUMMARY: TEST PASSED - Test File
rv32i/Zbkb/Zbkb-rori-00.log "Zbkb-rori-00.S"	RVCP-SUMMARY: TEST PASSED - Test File
rv32i/Zbkb/Zbkb-unzip-00.log "Zbkb-unzip-00.S"	RVCP-SUMMARY: TEST PASSED - Test File
rv32i/Zbkb/Zbkb-xnor-00.log "Zbkb-xnor-00.S"	RVCP-SUMMARY: TEST PASSED - Test File
rv32i/Zbkb/Zbkb-zip-00.log "Zbkb-zip-00.S"	RVCP-SUMMARY: TEST PASSED - Test File
rv32i/Zbkc/Zbkc-clmul-00.log "Zbkc-clmul-00.S"	RVCP-SUMMARY: TEST PASSED - Test File
rv32i/Zbkc/Zbkc-clmulh-00.log	RVCP-SUMMARY: TEST PASSED - Test File

"Zbkc-clmulh-00.S"	
rv32i/Zbkx/Zbkx-xperm4-00.log	RVCP-SUMMARY: TEST PASSED - Test File
"Zbkx-xperm4-00.S"	
rv32i/Zbkx/Zbkx-xperm8-00.log	RVCP-SUMMARY: TEST PASSED - Test File
"Zbkx-xperm8-00.S"	
rv32i/Zbs/Zbs-bclr-00.log	RVCP-SUMMARY: TEST PASSED - Test File
"Zbs-bclr-00.S"	
rv32i/Zbs/Zbs-bclri-00.log	RVCP-SUMMARY: TEST PASSED - Test File
"Zbs-bclri-00.S"	
rv32i/Zbs/Zbs-bext-00.log	RVCP-SUMMARY: TEST PASSED - Test File
"Zbs-bext-00.S"	
rv32i/Zbs/Zbs-bexti-00.log	RVCP-SUMMARY: TEST PASSED - Test File
"Zbs-bexti-00.S"	
rv32i/Zbs/Zbs-binv-00.log	RVCP-SUMMARY: TEST PASSED - Test File
"Zbs-binv-00.S"	
rv32i/Zbs/Zbs-binvi-00.log	RVCP-SUMMARY: TEST PASSED - Test File
"Zbs-binvi-00.S"	
rv32i/Zbs/Zbs-bset-00.log	RVCP-SUMMARY: TEST PASSED - Test File
"Zbs-bset-00.S"	
rv32i/Zbs/Zbs-bseti-00.log	RVCP-SUMMARY: TEST PASSED - Test File
"Zbs-bseti-00.S"	
rv32i/Zca/Zca-c.add-00.log	RVCP-SUMMARY: TEST PASSED - Test File
"Zca-c.add-00.S"	
rv32i/Zca/Zca-c.addi-00.log	RVCP-SUMMARY: TEST PASSED - Test File
"Zca-c.addi-00.S"	
rv32i/Zca/Zca-c.addi16sp-00.log	RVCP-SUMMARY: TEST PASSED - Test File
"Zca-c.addi16sp-00.S"	
rv32i/Zca/Zca-c.addi4spn-00.log	RVCP-SUMMARY: TEST PASSED - Test File
"Zca-c.addi4spn-00.S"	
rv32i/Zca/Zca-c.and-00.log	RVCP-SUMMARY: TEST PASSED - Test File
"Zca-c.and-00.S"	
rv32i/Zca/Zca-c.andi-00.log	RVCP-SUMMARY: TEST PASSED - Test File
"Zca-c.andi-00.S"	
rv32i/Zca/Zca-c.beqz-00.log	RVCP-SUMMARY: TEST PASSED - Test File
"Zca-c.beqz-00.S"	
rv32i/Zca/Zca-c.bnez-00.log	RVCP-SUMMARY: TEST PASSED - Test File
"Zca-c.bnez-00.S"	
rv32i/Zca/Zca-c.j-00.log	RVCP-SUMMARY: TEST PASSED - Test File
"Zca-c.j-00.S"	
rv32i/Zca/Zca-c.jal-00.log	RVCP-SUMMARY: TEST PASSED - Test File
"Zca-c.jal-00.S"	
rv32i/Zca/Zca-c.jalr-00.log	RVCP-SUMMARY: TEST PASSED - Test File
"Zca-c.jalr-00.S"	
rv32i/Zca/Zca-c.jr-00.log	RVCP-SUMMARY: TEST PASSED - Test File
"Zca-c.jr-00.S"	
rv32i/Zca/Zca-c.li-00.log	RVCP-SUMMARY: TEST PASSED - Test File
"Zca-c.li-00.S"	
rv32i/Zca/Zca-c.lui-00.log	RVCP-SUMMARY: TEST PASSED - Test File
"Zca-c.lui-00.S"	

rv32i/Zca/Zca-c.lw-00.log	RVCP-SUMMARY: TEST PASSED - Test File
"Zca-c.lw-00.S"	
rv32i/Zca/Zca-c.lwsp-00.log	RVCP-SUMMARY: TEST PASSED - Test File
"Zca-c.lwsp-00.S"	
rv32i/Zca/Zca-c.mv-00.log	RVCP-SUMMARY: TEST PASSED - Test File
"Zca-c.mv-00.S"	
rv32i/Zca/Zca-c.nop-00.log	RVCP-SUMMARY: TEST PASSED - Test File
"Zca-c.nop-00.S"	
rv32i/Zca/Zca-c.or-00.log	RVCP-SUMMARY: TEST PASSED - Test File
"Zca-c.or-00.S"	
rv32i/Zca/Zca-c.slli-00.log	RVCP-SUMMARY: TEST PASSED - Test File
"Zca-c.slli-00.S"	
rv32i/Zca/Zca-c.srai-00.log	RVCP-SUMMARY: TEST PASSED - Test File
"Zca-c.srai-00.S"	
rv32i/Zca/Zca-c.srli-00.log	RVCP-SUMMARY: TEST PASSED - Test File
"Zca-c.srli-00.S"	
rv32i/Zca/Zca-c.sub-00.log	RVCP-SUMMARY: TEST PASSED - Test File
"Zca-c.sub-00.S"	
rv32i/Zca/Zca-c.sw-00.log	RVCP-SUMMARY: TEST PASSED - Test File
"Zca-c.sw-00.S"	
rv32i/Zca/Zca-c.swsp-00.log	RVCP-SUMMARY: TEST PASSED - Test File
"Zca-c.swsp-00.S"	
rv32i/Zca/Zca-c.xor-00.log	RVCP-SUMMARY: TEST PASSED - Test File
"Zca-c.xor-00.S"	
rv32i/Zcb/Zcb-c.lbu-00.log	RVCP-SUMMARY: TEST PASSED - Test File
"Zcb-c.lbu-00.S"	
rv32i/Zcb/Zcb-c.lh-00.log	RVCP-SUMMARY: TEST PASSED - Test File
"Zcb-c.lh-00.S"	
rv32i/Zcb/Zcb-c.lhu-00.log	RVCP-SUMMARY: TEST PASSED - Test File
"Zcb-c.lhu-00.S"	
rv32i/Zcb/Zcb-c.not-00.log	RVCP-SUMMARY: TEST PASSED - Test File
"Zcb-c.not-00.S"	
rv32i/Zcb/Zcb-c.sb-00.log	RVCP-SUMMARY: TEST PASSED - Test File
"Zcb-c.sb-00.S"	
rv32i/Zcb/Zcb-c.sh-00.log	RVCP-SUMMARY: TEST PASSED - Test File
"Zcb-c.sh-00.S"	
rv32i/Zcb/Zcb-c.zext.b-00.log	RVCP-SUMMARY: TEST PASSED - Test File
"Zcb-c.zext.b-00.S"	
rv32i/ZcbM/ZcbM-c.mul-00.log	RVCP-SUMMARY: TEST PASSED - Test File
"ZcbM-c.mul-00.S"	
rv32i/ZcbZbb/ZcbZbb-c.sext.b-00.log	RVCP-SUMMARY: TEST PASSED - Test File
"ZcbZbb-c.sext.b-00.S"	
rv32i/ZcbZbb/ZcbZbb-c.sext.h-00.log	RVCP-SUMMARY: TEST PASSED - Test File
"ZcbZbb-c.sext.h-00.S"	
rv32i/ZcbZbb/ZcbZbb-c.zext.h-00.log	RVCP-SUMMARY: TEST PASSED - Test File
"ZcbZbb-c.zext.h-00.S"	
rv32i/Zcmop/Zcmop-c.mop.1-00.log	RVCP-SUMMARY: TEST PASSED - Test File
"Zcmop-c.mop.1-00.S"	
rv32i/Zcmop/Zcmop-c.mop.11-00.log	RVCP-SUMMARY: TEST PASSED - Test File

"Zcmop-c.mop.11-00.S"	RVCP-SUMMARY: TEST PASSED - Test File
rv32i/Zcmop/Zcmop-c.mop.13-00.log	
"Zcmop-c.mop.13-00.S"	RVCP-SUMMARY: TEST PASSED - Test File
rv32i/Zcmop/Zcmop-c.mop.15-00.log	
"Zcmop-c.mop.15-00.S"	RVCP-SUMMARY: TEST PASSED - Test File
rv32i/Zcmop/Zcmop-c.mop.3-00.log	
"Zcmop-c.mop.3-00.S"	RVCP-SUMMARY: TEST PASSED - Test File
rv32i/Zcmop/Zcmop-c.mop.5-00.log	
"Zcmop-c.mop.5-00.S"	RVCP-SUMMARY: TEST PASSED - Test File
rv32i/Zcmop/Zcmop-c.mop.7-00.log	
"Zcmop-c.mop.7-00.S"	RVCP-SUMMARY: TEST PASSED - Test File
rv32i/Zcmop/Zcmop-c.mop.9-00.log	
"Zcmop-c.mop.9-00.S"	RVCP-SUMMARY: TEST PASSED - Test File
rv32i/Zicntr/Zicntr-csrrc-00.log	
"Zicntr-csrrc-00.S"	RVCP-SUMMARY: TEST PASSED - Test File
rv32i/Zicntr/Zicntr-csrrs-00.log	
"Zicntr-csrrs-00.S"	RVCP-SUMMARY: TEST PASSED - Test File
rv32i/Zicnd/Zicnd-czero.eqz-00.log	
"Zicnd-czero.eqz-00.S"	RVCP-SUMMARY: TEST PASSED - Test File
rv32i/Zicnd/Zicnd-czero.nez-00.log	
"Zicnd-czero.nez-00.S"	RVCP-SUMMARY: TEST PASSED - Test File
rv32i/Zicsr/Zicsr-csrrc-00.log	
"Zicsr-csrrc-00.S"	RVCP-SUMMARY: TEST PASSED - Test File
rv32i/Zicsr/Zicsr-csrrci-00.log	
"Zicsr-csrrci-00.S"	RVCP-SUMMARY: TEST PASSED - Test File
rv32i/Zicsr/Zicsr-csrrs-00.log	
"Zicsr-csrrs-00.S"	RVCP-SUMMARY: TEST PASSED - Test File
rv32i/Zicsr/Zicsr-csrrsi-00.log	
"Zicsr-csrrsi-00.S"	RVCP-SUMMARY: TEST PASSED - Test File
rv32i/Zicsr/Zicsr-csrrw-00.log	
"Zicsr-csrrw-00.S"	RVCP-SUMMARY: TEST PASSED - Test File
rv32i/Zicsr/Zicsr-csrrwi-00.log	
"Zicsr-csrrwi-00.S"	RVCP-SUMMARY: TEST PASSED - Test File
rv32i/Zifencei/Zifencei-fence.i-00.log	
"Zifencei-fence.i-00.S"	RVCP-SUMMARY: TEST PASSED - Test File
rv32i/Zihpm/Zihpm-csrrc-00.log	
"Zihpm-csrrc-00.S"	RVCP-SUMMARY: TEST PASSED - Test File
rv32i/Zihpm/Zihpm-csrrs-00.log	
"Zihpm-csrrs-00.S"	RVCP-SUMMARY: TEST PASSED - Test File
rv32i/Zimop/Zimop-mop.r.0-00.log	
"Zimop-mop.r.0-00.S"	RVCP-SUMMARY: TEST PASSED - Test File
rv32i/Zimop/Zimop-mop.r.1-00.log	
"Zimop-mop.r.1-00.S"	RVCP-SUMMARY: TEST PASSED - Test File
rv32i/Zimop/Zimop-mop.r.10-00.log	
"Zimop-mop.r.10-00.S"	RVCP-SUMMARY: TEST PASSED - Test File
rv32i/Zimop/Zimop-mop.r.11-00.log	
"Zimop-mop.r.11-00.S"	RVCP-SUMMARY: TEST PASSED - Test File
rv32i/Zimop/Zimop-mop.r.12-00.log	
"Zimop-mop.r.12-00.S"	RVCP-SUMMARY: TEST PASSED - Test File

rv32i/Zimop/Zimop-mop.r.13-00.log "Zimop-mop.r.13-00.S"	RVCP-SUMMARY: TEST PASSED - Test File
rv32i/Zimop/Zimop-mop.r.14-00.log "Zimop-mop.r.14-00.S"	RVCP-SUMMARY: TEST PASSED - Test File
rv32i/Zimop/Zimop-mop.r.15-00.log "Zimop-mop.r.15-00.S"	RVCP-SUMMARY: TEST PASSED - Test File
rv32i/Zimop/Zimop-mop.r.16-00.log "Zimop-mop.r.16-00.S"	RVCP-SUMMARY: TEST PASSED - Test File
rv32i/Zimop/Zimop-mop.r.17-00.log "Zimop-mop.r.17-00.S"	RVCP-SUMMARY: TEST PASSED - Test File
rv32i/Zimop/Zimop-mop.r.18-00.log "Zimop-mop.r.18-00.S"	RVCP-SUMMARY: TEST PASSED - Test File
rv32i/Zimop/Zimop-mop.r.19-00.log "Zimop-mop.r.19-00.S"	RVCP-SUMMARY: TEST PASSED - Test File
rv32i/Zimop/Zimop-mop.r.2-00.log "Zimop-mop.r.2-00.S"	RVCP-SUMMARY: TEST PASSED - Test File
rv32i/Zimop/Zimop-mop.r.20-00.log "Zimop-mop.r.20-00.S"	RVCP-SUMMARY: TEST PASSED - Test File
rv32i/Zimop/Zimop-mop.r.21-00.log "Zimop-mop.r.21-00.S"	RVCP-SUMMARY: TEST PASSED - Test File
rv32i/Zimop/Zimop-mop.r.22-00.log "Zimop-mop.r.22-00.S"	RVCP-SUMMARY: TEST PASSED - Test File
rv32i/Zimop/Zimop-mop.r.23-00.log "Zimop-mop.r.23-00.S"	RVCP-SUMMARY: TEST PASSED - Test File
rv32i/Zimop/Zimop-mop.r.24-00.log "Zimop-mop.r.24-00.S"	RVCP-SUMMARY: TEST PASSED - Test File
rv32i/Zimop/Zimop-mop.r.25-00.log "Zimop-mop.r.25-00.S"	RVCP-SUMMARY: TEST PASSED - Test File
rv32i/Zimop/Zimop-mop.r.26-00.log "Zimop-mop.r.26-00.S"	RVCP-SUMMARY: TEST PASSED - Test File
rv32i/Zimop/Zimop-mop.r.27-00.log "Zimop-mop.r.27-00.S"	RVCP-SUMMARY: TEST PASSED - Test File
rv32i/Zimop/Zimop-mop.r.28-00.log "Zimop-mop.r.28-00.S"	RVCP-SUMMARY: TEST PASSED - Test File
rv32i/Zimop/Zimop-mop.r.29-00.log "Zimop-mop.r.29-00.S"	RVCP-SUMMARY: TEST PASSED - Test File
rv32i/Zimop/Zimop-mop.r.3-00.log "Zimop-mop.r.3-00.S"	RVCP-SUMMARY: TEST PASSED - Test File
rv32i/Zimop/Zimop-mop.r.30-00.log "Zimop-mop.r.30-00.S"	RVCP-SUMMARY: TEST PASSED - Test File
rv32i/Zimop/Zimop-mop.r.31-00.log "Zimop-mop.r.31-00.S"	RVCP-SUMMARY: TEST PASSED - Test File
rv32i/Zimop/Zimop-mop.r.4-00.log "Zimop-mop.r.4-00.S"	RVCP-SUMMARY: TEST PASSED - Test File
rv32i/Zimop/Zimop-mop.r.5-00.log "Zimop-mop.r.5-00.S"	RVCP-SUMMARY: TEST PASSED - Test File
rv32i/Zimop/Zimop-mop.r.6-00.log "Zimop-mop.r.6-00.S"	RVCP-SUMMARY: TEST PASSED - Test File
rv32i/Zimop/Zimop-mop.r.7-00.log	RVCP-SUMMARY: TEST PASSED - Test File

"Zimop-mop.r.7-00.S"	
rv32i/Zimop/Zimop-mop.r.8-00.log	RVCP-SUMMARY: TEST PASSED - Test File
"Zimop-mop.r.8-00.S"	
rv32i/Zimop/Zimop-mop.r.9-00.log	RVCP-SUMMARY: TEST PASSED - Test File
"Zimop-mop.r.9-00.S"	
rv32i/Zimop/Zimop-mop.rr.0-00.log	RVCP-SUMMARY: TEST PASSED - Test File
"Zimop-mop.rr.0-00.S"	
rv32i/Zimop/Zimop-mop.rr.1-00.log	RVCP-SUMMARY: TEST PASSED - Test File
"Zimop-mop.rr.1-00.S"	
rv32i/Zimop/Zimop-mop.rr.2-00.log	RVCP-SUMMARY: TEST PASSED - Test File
"Zimop-mop.rr.2-00.S"	
rv32i/Zimop/Zimop-mop.rr.3-00.log	RVCP-SUMMARY: TEST PASSED - Test File
"Zimop-mop.rr.3-00.S"	
rv32i/Zimop/Zimop-mop.rr.4-00.log	RVCP-SUMMARY: TEST PASSED - Test File
"Zimop-mop.rr.4-00.S"	
rv32i/Zimop/Zimop-mop.rr.5-00.log	RVCP-SUMMARY: TEST PASSED - Test File
"Zimop-mop.rr.5-00.S"	
rv32i/Zimop/Zimop-mop.rr.6-00.log	RVCP-SUMMARY: TEST PASSED - Test File
"Zimop-mop.rr.6-00.S"	
rv32i/Zimop/Zimop-mop.rr.7-00.log	RVCP-SUMMARY: TEST PASSED - Test File
"Zimop-mop.rr.7-00.S"	
rv32i/Zknd/Zknd-aes32dsi-00.log	RVCP-SUMMARY: TEST PASSED - Test File
"Zknd-aes32dsi-00.S"	
rv32i/Zknd/Zknd-aes32dsmi-00.log	RVCP-SUMMARY: TEST PASSED - Test File
"Zknd-aes32dsmi-00.S"	
rv32i/Zkne/Zkne-aes32esi-00.log	RVCP-SUMMARY: TEST PASSED - Test File
"Zkne-aes32esi-00.S"	
rv32i/Zkne/Zkne-aes32esmi-00.log	RVCP-SUMMARY: TEST PASSED - Test File
"Zkne-aes32esmi-00.S"	
rv32i/Zknh/Zknh-sha256sig0-00.log	RVCP-SUMMARY: TEST PASSED - Test File
"Zknh-sha256sig0-00.S"	
rv32i/Zknh/Zknh-sha256sig1-00.log	RVCP-SUMMARY: TEST PASSED - Test File
"Zknh-sha256sig1-00.S"	
rv32i/Zknh/Zknh-sha256sum0-00.log	RVCP-SUMMARY: TEST PASSED - Test File
"Zknh-sha256sum0-00.S"	
rv32i/Zknh/Zknh-sha256sum1-00.log	RVCP-SUMMARY: TEST PASSED - Test File
"Zknh-sha256sum1-00.S"	
rv32i/Zknh/Zknh-sha512sig0h-00.log	RVCP-SUMMARY: TEST PASSED - Test File
"Zknh-sha512sig0h-00.S"	
rv32i/Zknh/Zknh-sha512sig0l-00.log	RVCP-SUMMARY: TEST PASSED - Test File
"Zknh-sha512sig0l-00.S"	
rv32i/Zknh/Zknh-sha512sig1h-00.log	RVCP-SUMMARY: TEST PASSED - Test File
"Zknh-sha512sig1h-00.S"	
rv32i/Zknh/Zknh-sha512sig1l-00.log	RVCP-SUMMARY: TEST PASSED - Test File
"Zknh-sha512sig1l-00.S"	
rv32i/Zknh/Zknh-sha512sum0r-00.log	RVCP-SUMMARY: TEST PASSED - Test File
"Zknh-sha512sum0r-00.S"	
rv32i/Zknh/Zknh-sha512sum1r-00.log	RVCP-SUMMARY: TEST PASSED - Test File
"Zknh-sha512sum1r-00.S"	

```

rv32i/Zksed/Zksed-sm4ed-00.log          RVCP-SUMMARY: TEST PASSED - Test File
"Zksed-sm4ed-00.S"
rv32i/Zksed/Zksed-sm4ks-00.log          RVCP-SUMMARY: TEST PASSED - Test File
"Zksed-sm4ks-00.S"
rv32i/Zksh/Zksh-sm3p0-00.log           RVCP-SUMMARY: TEST PASSED - Test File
"Zksh-sm3p0-00.S"
rv32i/Zksh/Zksh-sm3p1-00.log           RVCP-SUMMARY: TEST PASSED - Test File
"Zksh-sm3p1-00.S"
rv32i/Zmmul/Zmmul-mul-00.log           RVCP-SUMMARY: TEST PASSED - Test File
"Zmmul-mul-00.S"
rv32i/Zmmul/Zmmul-mulh-00.log          RVCP-SUMMARY: TEST PASSED - Test File
"Zmmul-mulh-00.S"
rv32i/Zmmul/Zmmul-mulhsu-00.log        RVCP-SUMMARY: TEST PASSED - Test File
"Zmmul-mulhsu-00.S"
rv32i/Zmmul/Zmmul-mulhu-00.log        RVCP-SUMMARY: TEST PASSED - Test File
"Zmmul-mulhu-00.S"
</pre>
</details>

```



Unsupported/Unavailable ISA Extensions and Illegal Instruction

Executing instructions or accessing CSRs from unsupported ISA extensions will raise a precise illegal instruction exception allowing full software emulation (see section [Full Virtualization](#)).

1.7. RISC-V Documentation

The ratified RISC-V specifications (ISA specs, frameworks, ABI, etc.) are available online at riscv.org:

- <https://docs.riscv.org/reference/home>

Additional resources and specs currently under development are available on GitHub:

- ISA Specifications: <https://github.com/riscv>)
- Non-ISA Specifications: <https://github.com/riscv-non-isa>
- RISC-V Software: <https://github.com/riscv-software-src>

Elementary software references:

- RISC-V C API: <https://github.com/riscv-non-isa/riscv-c-api-doc>
- RISC-V ELF psABI: <https://github.com/riscv-non-isa/riscv-elf-psabi-doc>

Chapter 2. NEORV32 Processor (SoC)

The NEORV32 Processor provides a RISC-V-based full-scale microcontroller-like SoC platform build around the **NEORV32 Central Processing Unit (CPU)**.

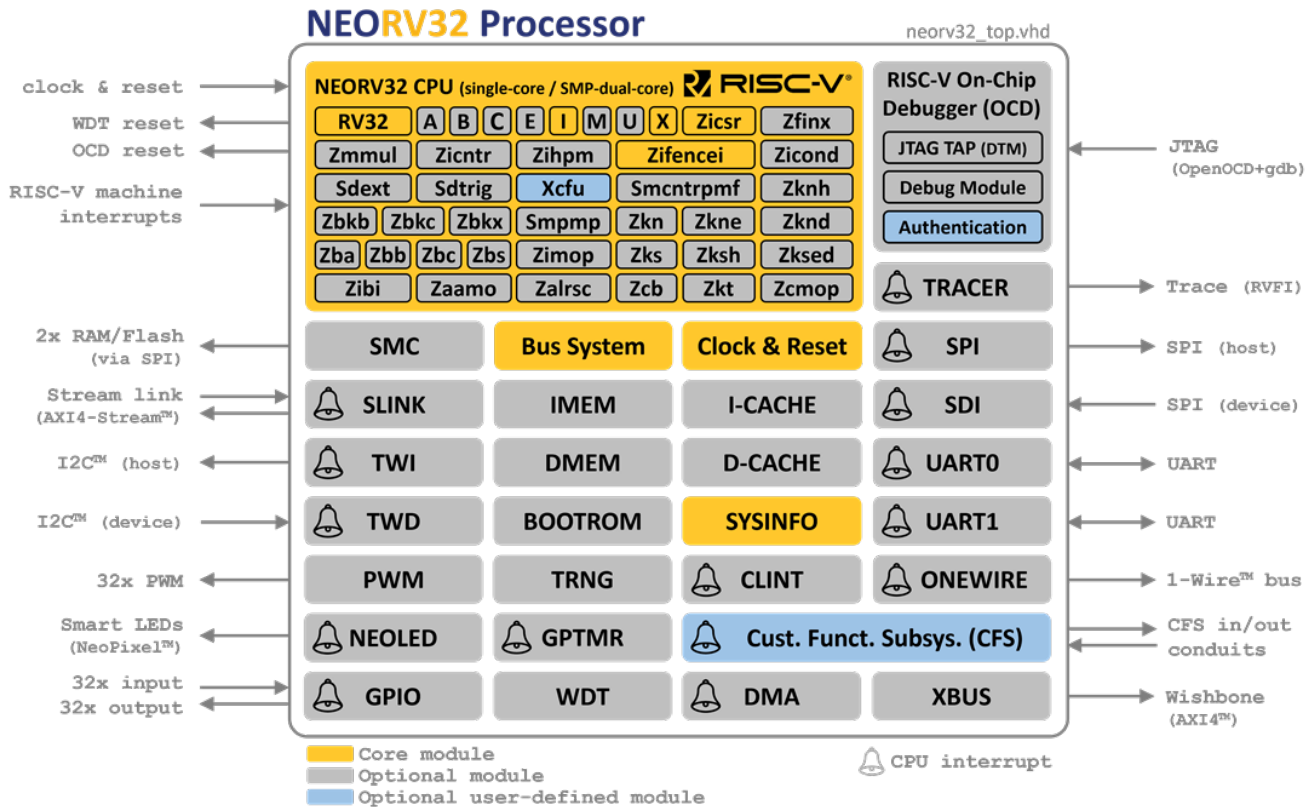


Figure 1. The NEORV32 Processor (Block Diagram)

Section Structure

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

Key Features

- optional SMP **Dual-Core Configuration**
- optional data and instruction memories (**DMEM/IMEM**)
- optional caches (**I-CACHE**, **D-CACHE**)
- optional serial memory controller **SMC** to map external serial (PS)RAM/flash to the processor address space
- optional internal **Bootloader (BOOTROM)** with UART console & SPI/TWI flash and SD card boot

options

- *optional* RISC-V-compatible core local interruptor (**CLINT**)
- *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 I²C standard
- *optional* two-wire serial device controller (**TWD**), compatible to the I²C standard
- *optional* general purpose parallel IO port (**GPIO**), 32 inputs (interrupt capable), 32 outputs
- *optional* 32-bit external bus interface, Wishbone-compatible (**XBUS**), AXI4-compatible bridge available
- *optional* watchdog timer (**WDT**)
- *optional* PWM controller with up to 32 individual channels (**PWM**)
- *optional* ring-oscillator-based true random number generator (**TRNG**)
- *optional* NeoPixel™/WS2812-compatible smart LED interface (**NEOLED**)
- *optional* custom functions subsystem for custom co-processor extensions (**CFS**)
- *optional* general purpose 32-bit timer (**GPTMR**) with up to 16 individual timer slices
- *optional* 1-wire serial interface controller (**ONEWIRE**), compatible to the 1-wire standard
- *optional* autonomous direct memory access controller (**DMA**)
- *optional* stream link interface (**SLINK**), AXI4-Stream-compatible
- *optional* on-chip debugger with JTAG TAP (**OCD**), optional authentication and hardware breakpoint
- *optional* execution trace buffer to debug program flow (**TRACER**) via branch tracing
- *optional* RVFI-compatible **Execution Trace Port** for advanced debugging, profiling and verification
- *optional* system configuration information memory to determine hardware configuration via software (**SYSINFO**)

2.1. Processor Top Entity - Generics

This section lists all configuration generics of the NEORV32 processor top entity (`rtl/neorv32_top.vhd`). These generics allow to configure the system according to your needs. The generics are used to control implementation of certain CPU extensions and peripheral modules and even allow to optimize the system for certain design goals like minimal area or maximum performance.



Default Values

All configuration generics provide default values in case they are not explicitly assigned during instantiation. By default, all configuration options are **disabled**.



Software Discovery of Configuration

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

Table 2. NEORV32 Processor Generic List; table abbreviations: the generic type “`suv(x:y)`” is an abbreviation for “`std_ulogic_vector(x downto y)`”

Name	Type	Default	Description
General			
<code>CLOCK_FREQUENCY</code>	natural	0	The clock frequency of the processor’s <code>clk_i</code> input port in Hertz (Hz). See Processor Clocking .
<code>TRACE_PORT_EN</code>	boolean	false	Enable external CPU execution Execution Trace Port .
<code>DUAL_CORE_EN</code>	boolean	false	Enable the SMP Dual-Core Configuration .
Boot Configuration			
<code>BOOT_MODE_SELECT</code>	natural	0	Boot mode select; see Boot Configuration .
<code>BOOT_ADDR_CUSTOM</code>	<code>suv(31:0)</code>	<code>x"00000000</code> "	Custom CPU boot address (available if <code>BOOT_MODE_SELECT = 1</code>).
On-Chip Debugger (OCD)			
<code>OCD_EN</code>	boolean	false	Implement the on-chip debugger and the CPU debug mode (Sdext ISA Extension).
<code>OCD_NUM_HW_TRIGGER</code> <code>S</code>	natural	0	Number of implemented HW triggers (Trigger Module / Sdtrig ISA Extension) for hardware break-/watchpoints (0..16).
<code>OCD_AUTHENTICATION</code>	boolean	false	Implement Debug Authentication module.
<code>OCD_JEDEC_ID</code>	<code>suv(10:0)</code>	<code>"0000000000</code> <code>00"</code>	JEDEC ID: continuation codes plus vendor ID.

Name	Type	Default	Description
CPU Instruction Sets and Extensions			
RISCV_ISA_C	boolean	false	Enable C ISA Extension (compressed instructions).
RISCV_ISA_E	boolean	false	Enable E ISA Extension (reduced register file size).
RISCV_ISA_M	boolean	false	Enable M ISA Extension (hardware-based integer multiplication and division).
RISCV_ISA_U	boolean	false	Enable U ISA Extension (less-privileged user mode).
RISCV_ISA_Zaamo	boolean	false	Enable Zaamo ISA Extension (atomic read-modify-write operations).
RISCV_ISA_Zalrsc	boolean	false	Enable Zalrsc ISA Extension (atomic reservation-set operations).
RISCV_ISA_Zba	boolean	false	Enable Zba ISA Extension (shifted-add bit-manipulation instructions).
RISCV_ISA_Zbb	boolean	false	Enable Zbb ISA Extension (basic bit-manipulation instructions).
RISCV_ISA_Zbc	boolean	false	Enable Zbc ISA Extension (carry-less multiplication instructions).
RISCV_ISA_Zbkb	boolean	false	Enable Zbkb ISA Extension (scalar cryptography bit manipulation instructions).
RISCV_ISA_Zbkc	boolean	false	Enable Zbkc ISA Extension (scalar cryptography carry-less multiplication instructions).
RISCV_ISA_Zbkx	boolean	false	Enable Zbkx ISA Extension (scalar cryptography crossbar permutations).
RISCV_ISA_Zbs	boolean	false	Enable Zbs ISA Extension (single-bit bit-manipulation instructions).
RISCV_ISA_Zcb	boolean	false	Enable Zcb ISA Extension (additional code size reduction instruction; builds upon C).
RISCV_ISA_Zcmop	boolean	false	Enable Zcmop ISA Extension (compressed may-be-operations; builds upon C and Zimop).
RISCV_ISA_Zfinx	boolean	false	Enable Zfinx ISA Extension (single-precision floating-point unit).
RISCV_ISA_Zibi	boolean	false	Enable Zibi ISA Extension (CPU base counters).
RISCV_ISA_Zicntr	boolean	false	Enable Zicntr ISA Extension (CPU base counters).
RISCV_ISA_Zicond	boolean	false	Enable Zicond ISA Extension (integer conditional instructions).
RISCV_ISA_Zihpm	boolean	false	Enable Zihpm ISA Extension (hardware performance monitors).

Name	Type	Default	Description
<code>RISCV_ISA_Zimop</code>	boolean	false	Enable Zimop ISA Extension (may-be-operations).
<code>RISCV_ISA_Zknd</code>	boolean	false	Enable Zknd ISA Extension (scalar cryptography NIST AES decryption instructions).
<code>RISCV_ISA_Zkne</code>	boolean	false	Enable Zkne ISA Extension (scalar cryptography NIST AES encryption instructions).
<code>RISCV_ISA_Zknh</code>	boolean	false	Enable Zknh ISA Extension (scalar cryptography NIST hash instructions).
<code>RISCV_ISA_Zksed</code>	boolean	false	Enable Zksed ISA Extension (scalar cryptography ShangMi block cyphers).
<code>RISCV_ISA_Zksh</code>	boolean	false	Enable Zksh ISA Extension (scalar cryptography ShangMi hash functions).
<code>RISCV_ISA_Zmmul</code>	boolean	false	Enable Zmmul - ISA Extension (hardware-based integer multiplication).
<code>RISCV_ISA_Smcntrpmf</code>	boolean	false	Enable Smcntrpmf - ISA Extension (counter privilege-mode filtering).
<code>RISCV_ISA_Xcfu</code>	boolean	false	Enable NEORV32-specific Xcfu ISA Extension (custom RISC-V instructions).
CPU Tuning Options			
<code>CPU_CONSTT_BR_EN</code>	boolean	false	Implement constant-time branches (same execution times for taken and not-taken branches).
<code>CPU_FAST_MUL_EN</code>	boolean	false	Implement fast but large full-parallel multipliers (trying to infer DSP blocks); see section CPU Arithmetic Logic Unit .
<code>CPU_FAST_MUL_REG</code>	boolean	false	Add a pipeline register to the fast multiplier (requires <code>CPU_FAST_MUL_EN</code>); see section CPU Arithmetic Logic Unit .
<code>CPU_FAST_SHIFT_EN</code>	boolean	false	Implement fast but large full-parallel barrel shifters; see section CPU Arithmetic Logic Unit .
<code>CPU_RF_ARCH_SEL</code>	natural	0	CPU register file implementation style select; see section CPU Register File .
Physical Memory Protection (Smpmp ISA Extension)			
<code>PMP_NUM_REGIONS</code>	natural	0	Number of implemented PMP regions (0..16).
<code>PMP_MIN_GRANULARITY</code>	natural	4	Minimal region granularity in bytes. Has to be a power of two, min 4.
<code>PMP_TOR_MODE_EN</code>	boolean	false	Implement support for top-of-region (TOR) mode.
<code>PMP_NAP_MODE_EN</code>	boolean	false	Implement support for naturally-aligned power-of-two (NAPOT & NA4) modes.

Name	Type	Default	Description
Hardware Performance Monitors (Zihpm ISA Extension)			
HPM_NUM_CNTS	natural	0	Number of implemented hardware performance monitor counters (0..29).
HPM_CNT_WIDTH	natural	64	Total LSB-aligned size of each HPM counter. Min 0, max 64.
Internal Instruction Memory (IMEM)			
IMEM_EN	boolean	false	Implement instruction memory.
IMEM_BASE	suv(31:0)	x"00000000 "	Base address of the instruction memory (has to be naturally-aligned to its size). Handle with care.
IMEM_SIZE	natural	16*1024	Size in bytes of the instruction memory (use a power of 2).
IMEM_OUTREG_EN	boolean	false	Add IMEM output register stage (improves mapping/timing at the expense of latency).
Internal Data Memory (DMEM)			
DMEM_EN	boolean	false	Implement data memory.
DMEM_BASE	suv(31:0)	x"80000000 "	Base address of the data memory (has to be naturally-aligned to its size). Handle with care.
DMEM_SIZE	natural	8*1024	Size in bytes of the data memory (use a power of 2).
DMEM_OUTREG_EN	boolean	false	Add DMEM output register stage (improves mapping/timing at the expense of latency).
CPU Caches (Instruction Cache (iCache) & Data Cache (dCache))			
ICACHE_EN	boolean	false	Implement the instruction cache ("I\$").
ICACHE_NUM_BLOCKS	natural	4	Number of blocks ("lines"). Has to be a power of two.
DCACHE_EN	boolean	false	Implement the data cache ("D\$")
DCACHE_NUM_BLOCKS	natural	4	Number of blocks ("lines"). Has to be a power of two.
CACHE_BLOCK_SIZE	natural	64	global cache block size (I\$ and D\$) in bytes. Has to be a power of two, min 4.
CACHE_BURSTS_EN	boolean	true	Enable burst transfers for cache updates.
CACHE_UC_BASE	suv(31:0)	x"F0000000 "	Base address of uncached address space. Has to be 256MB-aligned.
Serial Memory Controller (SMC)			
SMC_EN	boolean	false	Implement serial memory controller.

Name	Type	Default	Description
SMC_BASE	suv(31:0)	x"E0000000 "	serial memory base address. Has to be 256MB-aligned.
Processor-External Bus Interface (XBUS) (Wishbone / AXI4-Compatible Bridging)			
XBUS_EN	boolean	false	Implement the external bus interface.
XBUS_TIMEOUT	natural	2048	Number of clock cycles after which an unacknowledged external bus access will auto-terminate (0 = disabled).
XBUS_REGSTAGE_EN	boolean	false	Implement XBUS register stages to ease timing closure.
General Purpose Input and Output Port (GPIO)			
IO_GPIO_NUM	natural	0	Number of GPIO pairs, max 32.
IO_GPIO_DIR_EN	boolean	false	Enable optional GPIO direction control interface (<code>gpio_dir_o</code>).
Core-Local Interruptor (CLINT)			
IO_CLINT_EN	boolean	false	Implement CLINT.
Primary Universal Asynchronous Receiver and Transmitter (UART0)			
IO_UART0_EN	boolean	false	Implement UART0.
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.
Secondary Universal Asynchronous Receiver and Transmitter (UART1)			
IO_UART1_EN	boolean	false	Implement UART1.
IO_UART1_RX_FIFO	natural	1	UART1 RX FIFO depth, has to be a power of two, minimum value is 1, max 32768.
IO_UART1_TX_FIFO	natural	1	UART1 TX FIFO depth, has to be a power of two, minimum value is 1, max 32768.
Serial Peripheral Interface Controller (SPI)			
IO_SPI_EN	boolean	false	Implement SPI controller.
IO_SPI_FIFO	natural	1	Depth of the SPI FIFO. Has to be a power of two, min 1, max 32768.
Serial Data Interface Controller (SDI)			
IO_SDI_EN	boolean	false	Implement SDI controller.
IO_SDI_FIFO	natural	1	Depth of the SDI FIFO. Has to be a power of two, min 1, max 32768.

Name	Type	Default	Description
Two-Wire Serial Interface Controller (TWI)			
IO_TWI_EN	boolean	false	Implement TWI controller.
IO_TWI_FIFO	natural	1	Depth of the TWI FIFO. Has to be a power of two, min 1, max 32768.
Two-Wire Serial Device Controller (TWD)			
IO_TWD_EN	boolean	false	Implement TWD controller.
IO_TWD_RX_FIFO	natural	1	Depth of the TWD RX FIFO. Has to be a power of two, min 1, max 32768.
IO_TWD_TX_FIFO	natural	1	Depth of the TWD TX FIFO. Has to be a power of two, min 1, max 32768.
Pulse-Width Modulation Controller (PWM)			
IO_PWM_NUM	natural	0	Number of PWM channels to implement (0..32).
Watchdog Timer (WDT)			
IO_WDT_EN	boolean	false	Implement WDT.
True Random-Number Generator (TRNG)			
IO_TRNG_EN	boolean	false	Implement TRNG.
IO_TRNG_FIFO	natural	1	Depth of the TRNG data FIFO. Has to be a power of two, min 1, max 32768.
IO_TRNG_NUM_RO	natural	3	Total number of ring-oscillators.
IO_TRNG_NUM_INV	natural	5	Number of inverters in first ring-oscillator; has to be odd.
IO_TRNG_NUM_RBIT	natural	64	Number of raw random bits to process for one output byte; has to be power of 2.
Custom Functions Subsystem (CFS)			
IO_CFS_EN	boolean	false	Implement CFS.
Smart LED Interface (NEOLED)			
IO_NEOLED_EN	boolean	false	Implement NEOLED controller.
IO_NEOLED_TX_FIFO	natural	1	TX FIFO depth of the NEOLED controller. Has to be a power of two, min 1, max 32768.
General Purpose Timer (GPTMR)			
IO_GPTMR_NUM	natural	0	Number of individual GPTMR timer slices (0..16).
One-Wire Serial Interface Controller (ONEWIRE)			
IO_ONEWIRE_EN	boolean	false	Implement 1-wire controller.

Name	Type	Default	Description
<code>IO_ONEWIRE_FIFO</code>	natural	1	Depth of the 1-Wire FIFO. Has to be a power of two, min 1, max 32768.
Direct Memory Access Controller (DMA)			
<code>IO_DMA_EN</code>	boolean	false	Implement DMA controller.
<code>IO_DMA_DSC_FIFO</code>	natural	4	Depth of the DMA transfer descriptor FIFO. Has to be a power of two, min 4, max 512.
Stream Link Interface (SLINK)			
<code>IO_SLINK_EN</code>	boolean	false	Implement stream link interface (AXI4-Stream-Compatible).
<code>IO_SLINK_RX_FIFO</code>	natural	1	SLINK RX FIFO depth, has to be a power of two, minimum value is 1, max 32768.
<code>IO_SLINK_TX_FIFO</code>	natural	1	SLINK TX FIFO depth, has to be a power of two, minimum value is 1, max 32768.
Execution Trace Buffer (TRACER)			
<code>IO_TRACER_EN</code>	boolean	false	Implement the trace buffer.
<code>IO_TRACER_BUFFER</code>	natural	1	Depth of the trace buffer. Has to be a power of two, min 1, max 32768.
<code>IO_TRACER_SIMLOG_EN</code>	boolean	false	Write full trace log to file (simulation-only).

2.2. Processor Top Entity - Signals

The following table shows the interface signals of the processor top entity (`rtl/core/neorv32_top.vhd`). All signals are of type `std_ulogic` or `std_ulogic_vector`, respectively. Custom interface type are used only for the **Execution Trace Port** (`trace_*`).



Default Values of Inputs

All input signals - except for clock and reset - provide default values in case they are not explicitly assigned during instantiation. The weak driver strengths of VHDL ('L' and 'H') are used to model a pull-down or pull-up resistor for these unconnected inputs. After synthesis, these will result in 0 or 1, respectively.



Variable-Sized Ports

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, the TWD and the 1-Wire bus) require explicit tri-state drivers in the final top module.



Input/Output Registers

By default all output signals are driven by register and all input signals are synchronized into the processor's clock domain also using registers. However, for ASIC implementations it is recommended to add another register state to all inputs and output so the synthesis tool can insert an explicit IO (boundary) scan chain.

Table 3. NEORV32 Processor Signal List

Name	Width	Direction	Default	Description
Global Control (Processor Clocking and Processor Reset)				
<code>clk_i</code>	1	in	none	global clock line, all registers triggering on rising edge
<code>rstn_i</code>	1	in	none	global reset, asynchronous, low-active
<code>rstn_ocd_o</code>	1	out	-	On-Chip Debugger (OCD) reset output, synchronous, low-active
<code>rstn_wdt_o</code>	1	out	-	Watchdog Timer (WDT) reset output, synchronous, low-active

Name	Width	Direction	Default	Description
Execution Trace Port				
trace_cpu0_o	*	out	-	CPU core 0 trace port (type <code>trace_port_t</code>).
trace_cpu1_o	*	out	-	CPU core 1 trace port (type <code>trace_port_t</code>).
JTAG Access Port for On-Chip Debugger (OCD)				
jtag_tck_i	1	in	'L'	serial clock
jtag_tdi_i	1	in	'L'	serial data input
jtag_tdo_o	1	out	-	serial data output
jtag_tms_i	1	in	'L'	mode select
Serial Memory Controller (SMC)				
smc_ioen_o	1	out	-	SMC pin enable, can be used for IO multiplexing
smc_sck_o	1	out	-	serial clock
smc_csn_o	2	out	-	bank/chip select, low-active
smc_sdo_o	1	out	-	serial data output
smc_sdi_i	1	in	'L'	serial data input
Processor-External Bus Interface (XBUS)				
xbus_adr_o	32	out	-	destination address
xbus_dat_o	32	out	-	read data
xbus_cti_o	3	out	-	cycle type
xbus_tag_o	3	out	-	access tag
xbus_we_o	1	out	-	write enable ('0' = read transfer)
xbus_sel_o	4	out	-	byte enable
xbus_stb_o	1	out	-	strobe
xbus_cyc_o	1	out	-	valid cycle
xbus_dat_i	32	in	'L'	write data
xbus_ack_i	1	in	'L'	transfer acknowledge
xbus_err_i	1	in	'L'	transfer error
Stream Link Interface (SLINK)				
slink_rx_dat_i	32	in	'L'	RX data
slink_rx_src_i	4	in	'L'	RX source routing information
slink_rx_val_i	1	in	'L'	RX data valid
slink_rx_lst_i	1	in	'L'	RX last element of stream

Name	Width	Direction	Default	Description
slink_rx_rdy_o	1	out	-	RX ready to receive
slink_tx_dat_o	32	out	-	TX data
slink_tx_dst_o	4	out	-	TX destination routing information
slink_tx_val_o	1	out	-	TX data valid
slink_tx_lst_o	1	out	-	TX last element of stream
slink_tx_rdy_i	1	in	'L'	TX allowed to send
General Purpose Input and Output Port (GPIO)				
gpio_dir_o	32	out	-	general purpose port direction control (0 = input, 1 = output)
gpio_o	32	out	-	general purpose parallel output port
gpio_i	32	in	'L'	general purpose parallel input port (interrupt-capable)
Primary Universal Asynchronous Receiver and Transmitter (UART0)				
uart0_txd_o	1	out	-	serial transmitter
uart0_rxd_i	1	in	'L'	serial receiver
uart0_rtsn_o	1	out	-	RX ready to receive new char
uart0_ctsn_i	1	in	'L'	TX allowed to start sending, low-active
Secondary Universal Asynchronous Receiver and Transmitter (UART1)				
uart1_txd_o	1	out	-	serial transmitter
uart1_rxd_i	1	in	'L'	serial receiver
uart1_rtsn_o	1	out	-	RX ready to receive new char
uart1_ctsn_i	1	in	'L'	TX allowed to start sending, low-active
Serial Peripheral Interface Controller (SPI)				
spi_clk_o	1	out	-	controller clock line
spi_dat_o	1	out	-	serial data output
spi_dat_i	1	in	'L'	serial data input
spi_csn_o	8	out	-	chip select, low-active
Serial Data Interface Controller (SDI)				
sdi_clk_i	1	in	'L'	controller clock line
sdi_dat_o	1	out	-	serial data output
sdi_dat_i	1	in	'L'	serial data input
sdi_csn_i	1	in	'H'	chip select, low-active
Two-Wire Serial Interface Controller (TWI)				

Name	Width	Direction	Default	Description
<code>twi_sda_i</code>	1	in	'H'	serial data line sense input
<code>twi_sda_o</code>	1	out	-	serial data line output (pull low only)
<code>twi_scl_i</code>	1	in	'H'	serial clock line sense input
<code>twi_scl_o</code>	1	out	-	serial clock line output (pull low only)
Two-Wire Serial Device Controller (TWD)				
<code>twd_sda_i</code>	1	in	'H'	serial data line sense input
<code>twd_sda_o</code>	1	out	-	serial data line output (pull low only)
<code>twd_scl_i</code>	1	in	'H'	serial clock line sense input
One-Wire Serial Interface Controller (ONEWIRE)				
<code>onewire_i</code>	1	in	'H'	1-wire bus sense input
<code>onewire_o</code>	1	out	-	1-wire bus output (pull low only)
Pulse-Width Modulation Controller (PWM)				
<code>pwm_o</code>	16	out	-	pulse-width modulated channels
Custom Functions Subsystem (CFS)				
<code>cfs_in_i</code>	256	in	'L'	custom CFS input signal conduit
<code>cfs_out_o</code>	256	out	-	custom CFS output signal conduit
Smart LED Interface (NEOLED)				
<code>neoled_o</code>	1	out	-	asynchronous serial data output
Core-Local Interruptor (CLINT)				
<code>mtime_time_o</code>	64	out	-	CLINT.MTIMER system time output
RISC-V Machine-Mode Processor Interrupts				
<code>irq_msi_i</code>	1	in	'L'	machine software interrupt (RISC-V), high-level-active; available only if CLINT is not enabled
<code>irq_mti_i</code>	1	in	'L'	machine timer interrupt (RISC-V), high-level-active; available only if CLINT is not enabled
<code>irq_mei_i</code>	1	in	'L'	machine external interrupt (RISC-V), high-level-active

2.3. Processor Clocking

The entire system is implemented as fully-synchronous logic design using a single clock domain that is driven by the top's `clk_i` signal. This clock signal is used by all internal registers and memories. All of them trigger on the **rising edge** of this clock signal. External "clocks" like the OCD's JTAG clock or the SDI's serial clock are synchronized into the processor's clock domain before being used as "general logic signal" (and not as a dedicated clock).

2.3.1. Peripheral Clocks

Many processor modules like the UARTs or the timers provide a programmable time base for operations. In order to simplify the hardware, the processor implements a global "clock generator" (`neorv32_sys.vhd`) that provides single-cycle *clock enables* for certain frequencies which are derived from the main clock. These clock enable signals are synchronous to the system's main clock. The processor modules can use these enables for sub-main-clock operations while still providing a single clock domain only.

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

Prescaler bits:	<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 clock:	$f/2$	$f/4$	$f/8$	$f/64$	$f/128$	$f/1024$	$f/2048$	$f/4096$

2.4. Processor Reset

The NEORV32 processor includes a central reset sequencer (`neorv32_sys.vhd`) that handles all reset requests and controls the internal reset nets. The processor-wide reset ("system reset") can be triggered by any of the following sources:

- the asynchronous low-active `rstn_i` top entity input signal (external source)
- the **On-Chip Debugger (OCD)** (internal source)
- the **Watchdog Timer (WDT)** (internal source)



Reset Cause

The actual reset cause can be determined via the **Watchdog Timer (WDT)**.

If any of these sources triggers a reset, the internal system-wide reset will be active for at least 4 clock cycles ensuring a valid reset of the entire processor. This system reset is asserted *asynchronously* if triggered by the external `rstn_i` signal and is asserted *synchronously* if triggered by an internal reset source. However, the system reset is always de-asserted *synchronously* at the next rising clock edge.

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

2.5. Processor Interrupts

The system offers several interrupts (IRQs). Some can be used freely, others are reserved for specific tasks.



Trigger Type

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

2.5.1. RISC-V Standard Interrupts

The processor supports the standard RISC-V machine-level interrupts for "machine timer interrupt (MTI)", "machine software interrupt (MSI)" and "machine external interrupt (MEI)". Their usage is defined by the RISC-V privileged architecture specification. See CPU section [Traps, Exceptions and Interrupts](#) for more information.

Table 4. Processor Top RISC-V Machine-Level Interrupts

Top signal	Description
<code>irq_msi_i</code>	Machine software interrupt (MSI). This IRQ is only available if the processor-internal Core-Local Interruptor (CLINT) is not implemented.
<code>irq_mti_i</code>	Machine timer interrupt (MTI). This IRQ is only available if the processor-internal Core-Local Interruptor (CLINT) is not implemented.
<code>irq_mei_i</code>	Machine external interrupt (MEI). This interrupt is always available for custom use.

2.5.2. NEORV32-Specific Fast Interrupt Requests

As part of the NEORV32-specific CPU extensions, the processor core features 16 fast interrupt request signals (**FIRQ0** to **FIRQ15**) with dedicated bits in **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 5. NEORV32 Fast Interrupt Request (FIRQ) Mapping

Channel (priority)	Source
0	<i>reserved</i>
1	Custom Functions Subsystem (CFS)

Channel (priority)	Source
2	Primary Universal Asynchronous Receiver and Transmitter (UART0)
3	Secondary Universal Asynchronous Receiver and Transmitter (UART1)
4	Two-Wire Serial Device Controller (TWD)
5	Execution Trace Buffer (TRACER)
6	Serial Peripheral Interface Controller (SPI)
7	Two-Wire Serial Interface Controller (TWI)
8	General Purpose Input and Output Port (GPIO)
9	Smart LED Interface (NEOLED)
10	Direct Memory Access Controller (DMA)
11	Serial Data Interface Controller (SDI)
12	General Purpose Timer (GPTMR)
13	One-Wire Serial Interface Controller (ONEWIRE)
14	Stream Link Interface (SLINK)
15	True Random-Number Generator (TRNG)

2.6. Address Space

As a 32-bit architecture, NEORV32 can access up to 4GB of physical address space. The following figure shows the **default** address space layout with the according memory capabilities (*physical memory attributes* "PMAs"):

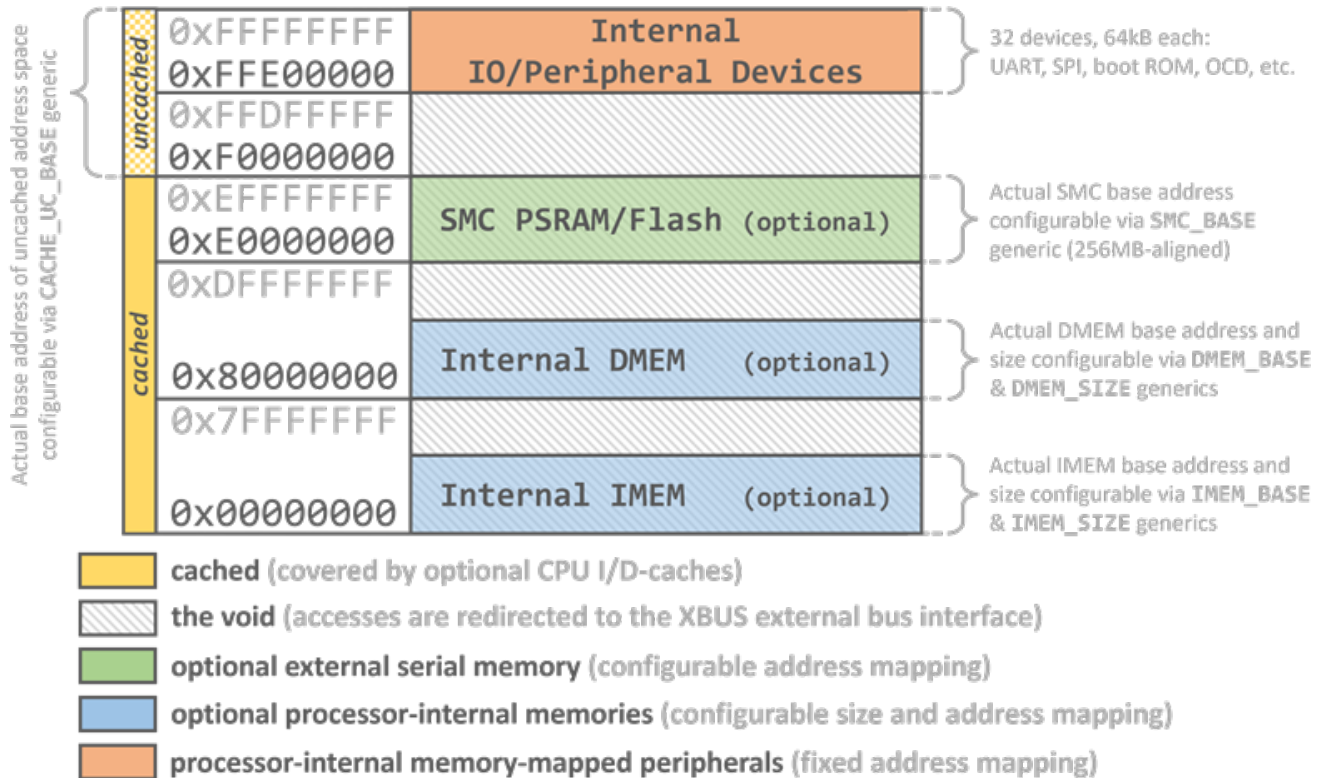


Figure 2. NEORV32 Processor Address Space (Default Configuration)

The address space layout is highly configurable. Only the very last 2 MB (`0xFFE00000` .. `0xFFFFFFFF`) are exclusively reserved for the NEORV32's internal memory-mapped peripherals and I/O modules and cannot be repurposed. The processor-internal **Instruction Memory (IMEM)** and **Data Memory (DMEM)** (if enabled), on the other hand, can be mapped to any address via the configuration generics. The same applies to off-chip serial memory, which can be mapped via the **Serial Memory Controller (SMC)** to any (256MB-aligned) address.

Accesses to "unmapped" addresses (a.k.a. **the void**) are redirected to the **Processor-External Bus Interface (XBUS)**. For example, if the processor-internal IMEM is disabled or re-mapped, accesses to the addresses starting at `0x00000000` are also redirected to the external bus interface. If the external bus interface is not enabled any access to the void will raise a bus error exception.

By default, all addresses starting at `0xF0000000` are not cached by the CPU caches to provide direct access to memory-mapped peripherals. However, this address boundary can be lowered by the according configuration generic (`CACHE_UC_BASE`) to increase the size of the uncached address space.

Uncached Address Space



Please note that the uncached address space can only be altered in 256 MB increments. The uncached address space starts at the configured 256 MB page and

always extends to the end of the address space (**0xFFFFFFFF**).

2.6.1. Bus System

The CPU core provides individual interfaces for instruction fetch and data access. Both interface access the same address space making the core a von-Neumann architecture. Optionally, the two interface can be equipped with data and/or instruction caches (**i-cache** & **d-cache**).

The two CPU interfaces are multiplexed by a simple bus switch into a single processor-internal bus. Optionally, this bus is further multiplexed by another instance of the bus switch so the **Direct Memory Access Controller (DMA)** can also access the entire address space.

Accesses via the resulting SoC bus are routed by the **Bus Gateway** that is used to redirect the access to the processor-internal **Instruction Memory (IMEM)** (if implemented), the processor-internal **Data Memory (DMEM)** (if implemented), the processor-internal peripheral devices or - if the access does not target any of these - the **Processor-External Bus Interface (XBUS)**. Accesses to the processor-internal IO/peripheral devices are further redirected via a dedicated **IO Switch**.

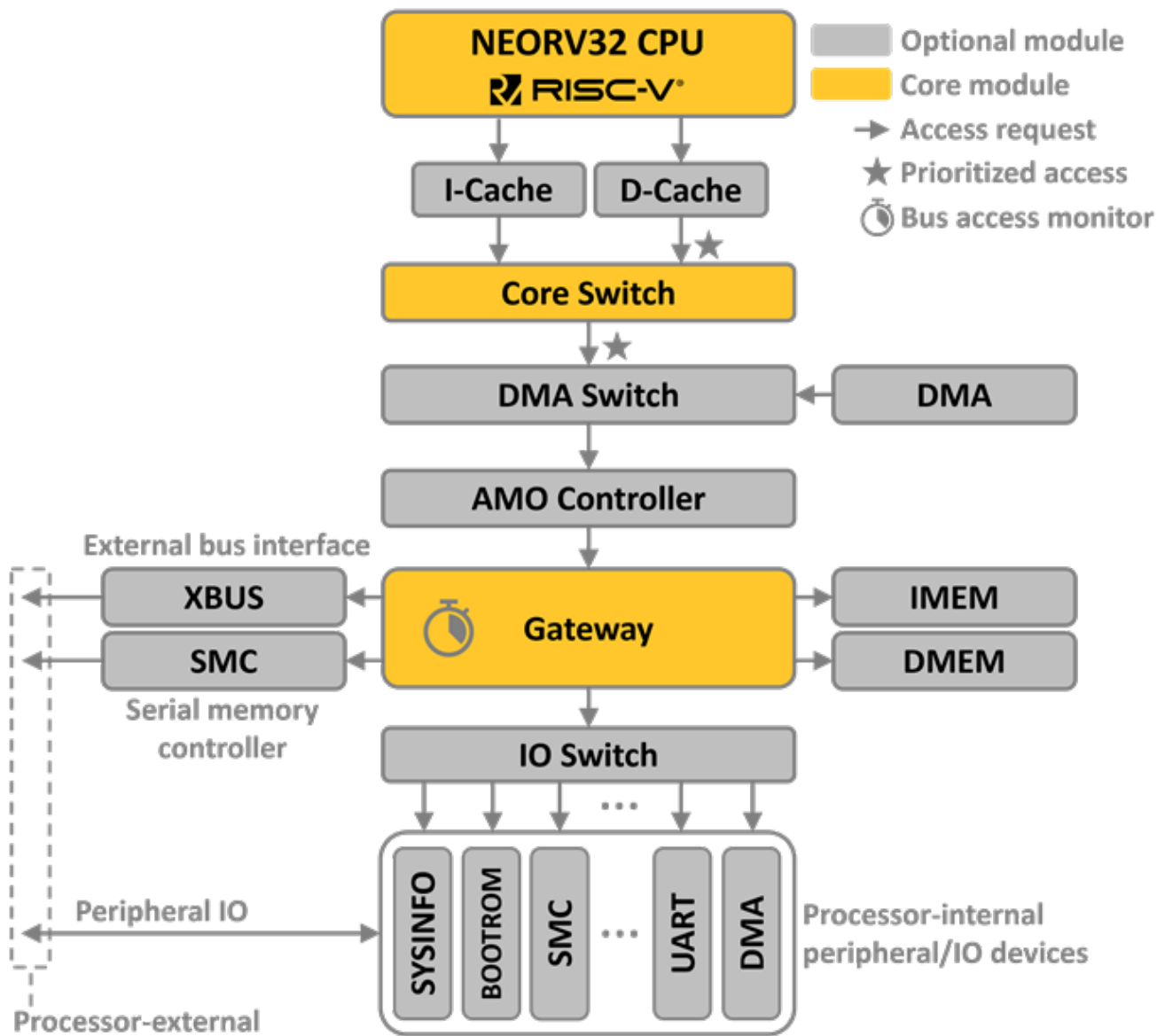


Figure 3. Processor-Internal Bus Architecture

*Bus System Infrastructure IP*

The individual components of the processor's bus system infrastructure are defined in `rtl/core/neorv32_bus.vhd`.

*Bus Interface*

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

*SMP Dual-Core Configuration*

The dual-core configuration adds a second CPU core complex in parallel to the first one. See section [Dual-Core Configuration](#) for more information.

2.6.2. Bus Gateway

The central bus gateway (VHDL component `neorv32_bus_gateway`) distributes CPU accesses to the two main memory regions. Depending on the configuration, it routes accesses to:

1. **Instruction Memory (IMEM)** (if implemented)
2. **Data Memory (DMEM)** (if implemented)
3. processor-internal IO/peripherals
4. **Processor-External Bus Interface (XBUS)** (if implemented)

Additionally, the gateway also provides a **bus monitor** (aka "the bus keeper") that *tracks* all bus transactions to ensure safe and deterministic operations.

Bus Monitor and Timeout

For each single access the bus monitor starts an internal countdown. The accessed module has to respond to the bus request within a bound time window. For **processor-internal** accesses this time windows is defined by a constant in the main NEORV32 package file. For **processor-external accesses** (via the external bus interface) this timeout is defined by the `XBUS_TIMEOUT` top configuration generic.

Listing 4. Internal Bus Timeout Configuration (package constant in `neorv32_package.vhd`)

```
constant int_bus_tmo_c : natural := 16; -- internal bus timeout window; has to be a
power of two
```

Listing 5. External Bus Timeout Configuration (top generic of `neorv32_top.vhd`)

```
XBUS_TIMEOUT : natural := 2048; -- cycles after a pending bus access auto-terminates
(0 = disabled)
```

The according time window defines the maximum number of cycles after which a *non-responding* bus request will time out raising a bus access fault exception. For example this can happen when accessing "address space holes" - addresses that are not mapped to any physical module at all. The specific bus access exception type corresponds to the according access type (i.e. instruction fetch bus access fault, load bus access fault or store bus access fault).



Serial Memory Controller Bus Timeout

The **Serial Memory Controller (SMC)** uses a hard-coded timeout value of 2048 cycles so that it can complete serial accesses even at the lowest SPI clock speed.

2.6.3. IO Switch

The IO switch (VHDL component `neorv32_bus_io_switch`) decodes accesses to the *procesor-internal IO/peripheral* address space. This 2MB address space is used to map all NEORV32 peripherals (like

UART, SPI, timers, etc.), the bootloader ROM and the on-chip debugger. Each device occupies an address range of 64kB; so up to 32 individual devices can be mapped. The according address map is defined in the main VHDL package file (`rtl/core/neorv32_package.vhd`).

2.6.4. Atomic Memory Operations Controller

The CPU's **A ISA Extension** adds support for atomic memory accesses. These atomic memory accesses are handles by dedicated modules of the processor bus system. The SoC bus that is driven by the CPU core(s) is fed through two atomic memory controller modules that handle read-modify-write operations as well as reservation-set operation:

Hardware Module	CPU ISA Extensions	Description
<code>neorv32_bus_amo_rmw</code>	Zaamo ISA Extension	Atomic Read-Modify-Write Controller
<code>neorv32_bus_amo_rvs</code>	Zalrsc ISA Extension	Atomic Reservation-Set Controller



Direct/Non-Caches Atomic Accesses

Atomic operations **always bypass** the CPU's **data cache** using direct/uncached accesses. Care must be taken to maintain data **Memory Coherence**.



Physical Memory Attributes

Atomic memory operations can be executed targeting *any* address. This also includes cached memory, memory-mapped IO devices and processor-external address spaces.

Atomic Read-Modify-Write Controller

This modules converts a single read-modify-write (RMW) memory operations request into two consecutive and atomic bus transactions. For each request, the controller executes an atomic set of three operations:

Table 6. Simplified AMO Controller Operation

Step	Pseudo Code	Description
1	<code>tmp1 ← MEM[address];</code>	Perform a read operation from the targeted memory address and temporarily store the loaded data into an internal buffer (<code>tmp1</code>).
2	<code>tmp2 ← tmp1 OP cpu_wdata;</code>	The buffered data from the first step is processed using the write data provided by the CPU. The result is stored to another temporary internal buffer (<code>tmp2</code>).
3	<code>MEM[address] ← tmp2;</code> <code>cpu_rdata ← tmp1;</code>	The data from the second buffer (<code>tmp2</code>) is written to the addressed memory location. In parallel, the data from the first buffer (<code>tmp1</code> = original content of the accessed address) is sent back to the requesting CPU.

Atomic Reservation-Set Controller

In NEORV32 a reservation is defined by a 64-byte address granule that provides a guarding mechanism to support atomic accesses (as specified by the RISC-V **Za1rsc ISA Extension**). A new reservation is registered by the **LR** instruction. The address provided by this instruction defines the 64-byte-aligned memory region that is monitored for exclusive accesses. The according **SC** instruction evaluates the state of this reservation. If the reservation is still valid (same owner core / issuing core and same address granule) the write access performed by the **SC** instruction is finally executed and the instruction returns a *success* state (i.e. `rd = 0`). If the reservation has been invalidated or does not match, the **SC** instruction will not update main memory and will return a *failed* state (i.e. `rd = 1`).



In either case, the **SC** instruction issues a downstream bus access to main memory (converted to a READ in case of failure) to allow the detection of access faults for the target address also for failed **SC** instructions which is required by the RISC-V specs.

The reservation-set controller implements a single reservation set per instance with a fixed granularity of 64 bytes. Each reservation tracks the owning core's ID and the aligned base address. A reservation is invalidated only when a write access targets the same 64-byte address granule - read accesses, instruction fetches, and **fence[.i]** instructions do not affect the reservation state.

The reservation-set controller is shared by both cores in the SMP **Dual-Core Configuration**. To ensure forward progress during concurrent atomic sequences, a turn-based arbitration mechanism is implemented:

1. An **LR** instruction can create (or update) a reservation if no reservation is active, if the requesting core already owns the current reservation, or if it is the requesting core's "turn" (priority).
2. A successful **SC** flips the turn, ensuring that the other core receives priority for its next **LR** attempt.
3. A failed **SC** from the owning core also clears the reservation, preventing stale reservations from blocking progress.

This guarantees atomic accesses and mutually exclusive access without starvation.



External Reservations

Note that the reservation-set controller cannot track memory accesses outside of the NEORV32 processor (i.e. atomic accesses via the external bus interface) that are executed by processor-external agents.

2.6.5. Memory Coherence

Depending on the configuration, the NEORV32 memory hierarchy provides several levels/layers of memory.

1. The CPU execution pipeline and its *instruction prefetch buffer* (CPU Front-End)
2. The optional **Data Cache (dCache)** and **Instruction Cache (iCache)**
3. Processor-internal and processor-external memories (depending on setup configuration)

NEORV32 features up to three bus hosts: up to two CPU cores and the **Direct Memory Access Controller (DMA)**. Since the levels of the memory hierarchy are transparent to the software, special care must be taken to ensure memory coherence is maintained throughout the entire system/application. Note that coherence and especially cache synchronization is **not** performed automatically by the hardware itself as there is no snooping implemented. NEORV32 supports two RISC-V memory ordering instructions that are used for manual memory synchronization:

- **fence**: Flush the CPU's data cache: modified ("dirty") blocks are copied back to main memory; all clean blocks are invalidated so that the next CPU access to any block results in a cache miss that reloads the cache with up-to-date data from main memory.
- **fence.i**: Perform the same operations as **fence** (i.e. flushing the data cache), but also clear the CPU's instruction prefetch buffer and instruction cache. The next instruction cache access therefore results in a miss that fetches up-to-date data from main memory.

Weak Coherence Model



The NEORV32-specific implementation of the **fence[i]** ordering instructions only provides a rather **weak** coherence model. A core's **fence** just orders all memory accesses towards main memory. Hence, they *can* become visible by other agents (the secondary CPU core, the DMA, processor-external modules) if these agents also synchronize (e.g. reload) their cache(s).

Atomic memory accesses also require special attention. All atomic memory accesses bypass the CPU caches. However, in this case, the caches explicitly ensure memory coherence. If the address of an atomic memory access is cached, the corresponding line is first written back to main memory if it has been modified. Afterward, the corresponding line is invalidated, and the atomic access is executed as a cache bypass. The next access to the atomic address bypasses the cache if it is again an atomic memory operation. Normal memory accesses result in a cache miss (since the corresponding line was automatically invalidated previously), so that the cache is reloaded with current data from main memory.

C-Language `_Atomic` Variables



Using plain **Atomic** access (e.g., `_Atomic int counter = 0;`) defaults to GCC's `memory_order_seq_cst` builtin, which inserts **fence** instructions *before* and *after* the access. This is the safest but slowest option.

2.7. Boot Configuration

The NEORV32 processor provides pre-defined boot configurations to adjust system start-up to the requirements of the application. The actual boot configuration is defined by the **BOOT_MODE_SELECT** generic (see **Processor Top Entity - Generics**). For the SMP dual-core CPU configuration boot procedure see section **Dual-Core Boot**.

Table 7. NEORV32 Boot Configurations

BOOT_MODE_SELECT	Name	Boot address	Description
0 (default)	Bootloader	Base of internal BOOTROM	Implement the processor-internal Bootloader ROM (BOOTROM) as pre-initialized ROM and boot from there.
1	Custom Address	BOOT_ADDR_CUSTOM generic	Start booting at user-defined address (BOOT_ADDR_CUSTOM top generic).
2	IMEM Image	Base of internal IMEM	Implement the processor-internal Instruction Memory (IMEM) as pre-initialized ROM and boot from there.

2.7.1. Booting via Bootloader

This is the default configuration. When selected, the processor-internal **Bootloader ROM (BOOTROM)** is enabled. This ROM contains the executable image (`rtl/core/neorv32_bootrom_image.vhd`) of the default NEORV32 **Bootloader** that will be executed right after reset. The bootloader provides an interactive user console for executable upload as well as an automatic mode to boot from external (SPI) flash memory.

If the processor-internal **Instruction Memory (IMEM)** is enabled it will be implemented as *blank* RAM.

2.7.2. Boot from Custom Address

This is the most flexible boot configuration as it allows the user to specify a custom boot address via the **BOOT_ADDR_CUSTOM** generic. Note that this address has to be aligned to 4-byte boundary. The processor will start executing from the defined address right after reset. For example, this boot configuration can be used to execute a *custom bootloader* from a memory that is attached via the **Processor-External Bus Interface (XBUS)**.

For this configuration the processor-internal **Bootloader ROM (BOOTROM)** is not enabled / implement at all. If the processor-internal **Instruction Memory (IMEM)** is enabled it will be implemented as *blank* RAM.

2.7.3. Boot IMEM Image

This configuration will implement the **Instruction Memory (IMEM)** as *pre-initialized ROM*. The ROM

is initialized during synthesis with the according application image file (`rtl/core/neorv32_imem_image.vhd`). After reset, the CPU will directly start executing this image. Since the IMEM is implemented as ROM, the executable cannot be altered at runtime at all.

For this configuration the **Bootloader ROM (BOOTROM)** is not enabled / implement at all. Note that this boot configuration requires the IMEM to be enabled (`IMEM_EN` = true).



Simulation Setup

This boot configuration is handy for simulations as the application software is executed right away without the need for an explicit initialization / executable upload.

2.8. Processor-Internal Modules



Full-Word Write Accesses Only

Unless otherwise specified, all peripheral/IO devices should only be accessed using full-word accesses (i.e. 32-bit).



IO Module Address Space

Each peripheral/IO module occupies an address space of 64kB bytes. Most devices do not fully utilize this address space and will mirror the available memory-mapped registers across the device's address space. However, only the specified memory-mapped registers addresses should be used.



Unimplemented Modules / Address Holes

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



Writing to Read-Only Registers

Unless otherwise specified, writing to registers that are listed as read-only does not trigger an exception as the write access is just ignored by the corresponding hardware module.



Module Interrupts

Several peripheral/IO devices provide some kind of interrupt. These interrupts are mapped to the CPU's **Custom Fast Interrupt Request Lines**. See section **Processor Interrupts** for more information.



CMSIS System Description View (SVD)

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

2.8.1. Instruction Memory (IMEM)

Hardware source files:	neorv32_imem.vhd	generic processor-internal instruction memory (RAM or ROM)
	neorv32_imem_image.vhd	memory initialization image (VHDL package)
	neorv32_imem_ram.vhd	RAM component (primitive/IP wrapper)
	neorv32_imem_rom.vhd	ROM component (primitive/IP wrapper)
Software driver files:	none	
Top entity ports:	none	
Configuration generics:	IMEM_EN	implement IMEM when true
	IMEM_BASE	32-bit base address (naturally aligned to IMEM size)
	IMEM_SIZE	IMEM size in bytes (use a power of 2)
	IMEM_OUTREG_EN	add IMEM output register stage
	BOOT_MODE_SELECT	implement IMEM as ROM when BOOT_MODE_SELECT = 2; see Boot Configuration
CPU interrupts:	none	

Key Features

- Burst-capable tightly-coupled on-chip instruction memory
- Configurable RAM size
- Accessible at the byte level
- Mapped to memory primitives
- Optional output register for improved timing
- Can be implemented as pre-initialized ROM containing application firmware

Overview

Implementation of the processor-internal instruction memory is enabled by the processor's **IMEM_EN** generic. The total memory size in bytes is defined via the **IMEM_SIZE** generic. Only sizes that are powers of two are supported. If enabled, the IMEM is mapped to base address **0x00000000** by default (see section **Address Space**).



Custom IMEM Base Address

The base address of the IMEM can be customized via the **IMEM_BASE** generic. Please

note that the selected address must be naturally aligned to the IMEM size (**IMEM_SIZE**) and that there must be no overlap with other sections of the internal **Address Space** (internal DMEM if enabled and the internal IO/peripheral address space). Any non-default address must also be passed to the **Linker Script** (e.g. `USER_FLAGS += -Wl,--defsym,__neorv32_rom_base=<IMEM_BASE>`).

By default the IMEM is implemented as true RAM so the content can be modified during run time. Alternatively, the IMEM can be implemented as **pre-initialized read-only memory (ROM)**, so the processor can directly boot from it after reset. This option is configured via the **BOOT_MODE_SELECT** generic. See section **Boot Configuration** for more information. If the IMEM is implemented as ROM any write attempt to it will raise a *store access fault* exception.

The software framework provides an option to generate and override the default VHDL initialization file `rtl/core/neorv32_imem_image.vhd`, which is automatically inserted into the IMEM (see **Makefile Targets**). If the IMEM is implemented as RAM (default), the memory block will not be initialized at all.



Output Register Stage

An optional output register stage can be enabled via **IMEM_OUTREG_EN**. For FPGA targets this might improve mapping/timing results. Note that this option will increase the read latency by one clock cycle. Write accesses are not affected by this at all.



Retrieve Memory Configuration by Software

Software can retrieve the DMEM size from the **SYSINFO - Miscellaneous Configuration** register.



Memory Primitives

The memory's memory core is encapsulated as individual wrapper (`neorv32_imem_ram.vhd` / `neorv32_imem_ram.vhd`) that can be replaced by a technology-specific IP. All interface ports provide static signal widths to ease replacement.

2.8.2. Data Memory (DMEM)

Hardware source files:	neorv32_dmem.vhd	generic processor-internal data memory (RAM)
	neorv32_dmem_ram.vhd	RAM component (primitive/IP wrapper)
Software driver files:	none	
Top entity ports:	none	
Configuration generics:	DMEM_EN	implement DMEM when true
	DMEM_BASE	32-bit base address (naturally aligned to DMEM size)
	DMEM_SIZE	DMEM size in bytes (use a power of 2)
	DMEM_OUTREG_EN	add DMEM output register stage
CPU interrupts:	none	

Key Features

- Burst-capable tightly-coupled on-chip data memory
- Configurable RAM size
- Accessible at the byte level
- Can also be used for executing code
- Mapped to memory primitives
- Optional output register for improved timing

Overview

Implementation of the processor-internal data memory is enabled by the processor's **DMEM_EN** generic. The total memory size in bytes is defined via the **DMEM_SIZE** generic. Only sizes that are powers of two are supported. If enabled, the DMEM is mapped to base address **0x80000000** by default (see section [Address Space](#)).



Custom DMEM Base Address

The base address of the DMEM can be customized via the **DMEM_BASE** generic. Please note that the selected address must be naturally aligned to the DMEM size (**DMEM_SIZE**) and that there must be no overlap with other sections of the internal [Address Space](#) (internal IMEM if enabled and the internal IO/peripheral address space). Any non-default address must also be passed to the [Linker Script](#) (e.g. **USER_FLAGS += -Wl,--defsym,__neorv32_ram_base=<DMEM_BASE>**).



Platform-Specific Memory Primitives

If required, the default DMEM can be replaced by a platform-/technology-specific

primitive to optimize area utilization, timing and power consumption.



Output Register Stage

An optional output register stage can be enabled via `DMEM_OUTREG_EN`. For FPGA targets this might improve mapping/timing results. Note that this option will increase the read latency by one clock cycle. Write accesses are not affected by this at all.



Retrieve Memory Configuration by Software

Software can retrieve the DMEM size from the `SYSINFO - Miscellaneous Configuration` register.



Memory Primitives

The memory's memory core is encapsulated as individual wrapper (`neorv32_dmem_ram.vhd`) that can be replaced by a technology-specific IP. All interface ports provide static signal widths to ease replacement.

2.8.3. Bootloader ROM (BOOTROM)

Hardware source files:	neorv32_bootrom.vhd	default platform-agnostic bootloader ROM
	neorv32_bootrom_image.vhd	memory initialization image
	neorv32_bootrom_rom.vhd	ROM component (primitive/IP wrapper)
Software driver files:	none	<i>implicitly used</i>
Top entity ports:	none	
Configuration generics:	BOOT_MODE_SELECT	implement BOOTROM when BOOT_MODE_SELECT = 0; see Boot Configuration
CPU interrupts:	none	

Key Features

- Tightly-coupled on-chip ROM
- Accessible at the byte level
- Mapped to memory primitives
- Pre-initialized with default NEORV32 bootloader

Overview

The boot ROM contains the pre-compiled executable image of the default NEORV32 **Bootloader**. When the **Boot Configuration** is set to *bootloader* mode (0) via the **BOOT_MODE_SELECT** generic, the boot ROM is automatically enabled and the CPU boot address is adjusted to the base address of the boot ROM. Note that the entire boot ROM is read-only.



Bootloader Image

The bootloader ROM is initialized during synthesis with the default bootloader image (`rtl/core/neorv32_bootrom_image.vhd`). The physical size of the ROM is automatically adjusted to the next power of two of the image size. However, note that the BOOTROM is constrained to a maximum size of 64kB.



Memory Primitives

The memory's memory core is encapsulated as individual wrapper (`neorv32_bootrom_rom.vhd`) that can be replaced by a technology-specific IP. All interface ports provide static signal widths to ease replacement.

2.8.4. Instruction Cache (iCache)

Hardware source files:	neorv32_cache.vhd	generic cache module
	neorv32_cache_ram.vhd	tag and data RAM (primitive/IP wrapper)
Software driver files:	none	
Top entity ports:	none	
Configuration generics:	ICACHE_EN	implement CPU-exclusive instruction cache (I\$) when true
	ICACHE_NUM_BLOCKS	number of cache blocks ("cache lines"); has to be a power of two
	CACHE_BLOCK_SIZE	size of a cache block in bytes (global configuration for I\$ and D\$); has to be a power of two, min 4
	CACHE_BURSTS_EN	enable burst transfers for cache update
	CACHE_UC_BASE	base address of uncached address space; has to be 256MB-aligned
CPU interrupts:	none	

Key Features

- Direct-mapped read-only cache
- Configurable number of lines
- Configurable line size
- Allows bypassing for *uncached* accesses
- Tag and data RAM mapped to memory primitives

Overview

The processor features an optional instruction cache (i-cache) that is directly connected to the **CPU Front-End** instruction fetch interface providing full-transparent accesses. The cache is direct-mapped and read-only. For the **Dual-Core Configuration** each CPU core is equipped with a private instruction cache.

The instruction cache is enabled by the **ICACHE_EN** generic. The total cache memory size in bytes is defined by **ICACHE_NUM_BLOCKS** x **CACHE_BLOCK_SIZE**. **ICACHE_NUM_BLOCKS** defines the number of cache blocks ("lines") and **CACHE_BLOCK_SIZE** defines the block (line) size in bytes; note that the latter configuration is global for all caches.



Retrieve Cache Configuration by Software

Software can retrieve the cache configuration/layout from the **SYSINFO - Cache**

Configuration register.



Memory Primitives

The cache's tag and data RAMs are encapsulated as wrapper (`neorv32_cache_ram.vhd`) that can be modified or replaced by a technology-specific IP. All interface ports provide static signal widths to ease replacement.

Burst Transfers

Cache update operations can use **burst transfers** to increase performance. Burst operations are enabled (for all caches) by the `CACHE_BURSTS_EN` top generic. When bursts are enabled all cache block transfers are **always executed as burst transfers**. Hence, all devices, memories and endpoints that can be accessed by the cache must also be able to process bursts (including modules connected to the **Processor-External Bus Interface (XBUS)**).

Cache Bypass

The cache provides direct/uncached accesses (bypassing the cache) in order to access memory-mapped peripherals. The base address of the uncached address space is configured by the `CACHE_UC_BASE` generic. Note that only the 4 MSBs of this address are evaluated resulting in a 256MB granularity. All accesses that target the address range from `CACHE_UC_BASE` to `0xFFFFFFFF` will bypass the cache. See section **Address Space** for more information.

Cache Synchronization and Memory Coherence

By executing the `fence.i` instruction the entire i-cache is invalidated. The next access therefore results in a cache miss that reloads the cache with up-to-date data from main memory. See section **Memory Coherence** for more information.

Cache Bus Error Handling

The cache responds differently to bus errors depending on which operation caused the error. Note that all instruction cache accesses are entirely read-only.

1. **Cache Bypass:** If a bus error occurs during a cache bypass operation - for example, when accessing memory-mapped I/O - the bus error is reported to the CPU *immediately* and with *full precision*. The address reported in `mepc` represents the *exact* address that caused the error.
2. **Cache Miss:** If the instruction request results in a cache miss, the according memory block is fetched from main memory. If any memory access within this block transfer causes a bus error, the entire cache block is discarded and the bus error is reported to the CPU *immediately* and with *partial precision*. The address reported in `mepc` represents an address within the cache/memory block whose fetch caused the error.
3. **Cache Synchronization:** A cache synchronization operation (`fence.i`) cannot cause any bus error as it just invalidated all current cache blocks without performing any memory access at all.

2.8.5. Data Cache (dCache)

Hardware source files:	neorv32_cache.vhd	generic cache module
	neorv32_cache_ram.vhd	tag and data RAM (primitive/IP wrapper)
Software driver files:	none	
Top entity ports:	none	
Configuration generics:	DCACHE_EN	implement CPU-exclusive data cache (D\$) when true
	DCACHE_NUM_BLOCKS	number of cache blocks ("cache lines"); has to be a power of two
	CACHE_BLOCK_SIZE	size of a cache block in bytes (global configuration for I\$ and D\$); has to be a power of two, min 4
	CACHE_BURSTS_EN	enable burst transfers for cache update
	CACHE_UC_BASE	base address of uncached address space; has to be 256MB-aligned
CPU interrupts:	none	

Key Features

- Direct-mapped cache with *write-back* & *write-allocated* write policies
- Configurable number of lines
- Configurable line size
- Allows bypassing for *uncached* accesses
- Tag and data RAM mapped to memory primitives

Overview

The processor features an optional data cache (d-cache) that is directly connected to the **CPU Load/Store Unit** data interface providing full-transparent accesses. The cache is *direct-mapped* and uses *write-back* (copy modified cache lines back to main memory when evicted) and *write-allocate* (fetch entire line on a write-miss) write policies. For the **Dual-Core Configuration** each CPU core is equipped with a private data cache.

The data cache is enabled by the **DCACHE_EN** generic. The total cache memory size in bytes is defined by **DCACHE_NUM_BLOCKS** × **CACHE_BLOCK_SIZE**. **DCACHE_NUM_BLOCKS** defines the number of cache blocks ("lines") and **CACHE_BLOCK_SIZE** defines the block (line) size in bytes; note that the latter configuration is global for all caches.



Retrieve Cache Configuration by Software

Software can retrieve the cache configuration/layout from the **SYSINFO - Cache Configuration** register.



Memory Primitives

The cache's tag and data RAM is encapsulated as individual wrapper (`neorv32_cache_ram.vhd`) that can be replaced by a technology-specific IP. All interface ports provide static signal widths to ease replacement.

Burst Transfers

Cache update operations can use **burst transfers** to increase performance. Burst operations are enabled (for all caches) by the **CACHE_BURSTS_EN** top generic. When bursts are enabled all cache block transfers are **always executed as burst transfers**. Hence, all devices, memories and endpoints that can be accessed by the cache must also be able to process bursts (including modules connected to the **Processor-External Bus Interface (XBUS)**).

Cache Bypass

The cache provides direct/uncached accesses (bypassing the cache) in order to access memory-mapped peripherals. The base address of the uncached address space is configured by the **CACHE_UC_BASE** generic. Note that only the 4 MSBs of this address are evaluated resulting in a 256MB granularity. All accesses that target the address range from **CACHE_UC_BASE** to **0xFFFFFFFF** will bypass the cache. See section **Address Space** for more information.

All atomic memory accesses also bypass the cache regardless of the address; however, in this case, the cache explicitly ensures **Memory Coherence**: if the address of an atomic memory access is cached, the corresponding line is first written back to main memory if it has been modified. The corresponding line is then invalidated, and the atomic access is finally executed as a cache-bypassing access.

Cache Synchronization and Memory Coherence

By executing the **fence** instruction the data cache is flushed: modified ("dirty") blocks are copied back to main memory; all clean blocks are invalidated so that the next CPU access to any block results in a cache miss that reloads the cache with up-to-date data from main memory. See section **Memory Coherence** for more information.

Cache Bus Error Handling

The cache responds differently to bus errors depending on which operation caused the error.

1. **Cache Bypass:** If a bus error occurs during a cache bypass operation - for example, when accessing memory-mapped I/O - the bus error is reported to the CPU *immediately* and with *full precision*. The address reported in **mtval** represents the *exact* address that caused the error.
2. **Cache Miss Without Eviction:** If the CPU request results in a cache miss that can be completed without, the according memory block is fetched from main memory. If any memory access within this block transfer causes a bus error, the entire cache block is discarded and the bus

error is reported to the CPU *immediately* and with *partial precision*. The address reported in `mtval` represents an address within the cache/memory block whose fetch caused the error.

3. **Cache Miss With Eviction:** If the CPU request results in a cache miss that requires the write-back of a modified ("dirty") cache block, the modified is copied back to main memory before fetching the actual requested memory block. If this write-back causes any bus error, the modified cache block being written back is discarded and the bus error is reported to the CPU *immediately* and *imprecisely*. The address reported in `mtval` only represents an address inside the requested memory block - not an address within the block that the cache was trying to write back.



Bus-Errors During Cache Write-Back

This type of cache-related "background" bus error is difficult to debug and - like most other bus errors - might be fatal when occurring in machine mode. A future version of NEORV32 will include an improved mechanism for determining the *exact* address of this type of bus error.

1. **Cache Synchronization:** If the cache encounters a bus error while writing modified cache blocks back to main memory during a cache synchronization operation (*fence*), the cache block that caused the error is remains valid but also "dirty" (not sync with main memory). The next CPU access that results in a **cache miss with eviction** of this block will encounter a bus error that is reported to the CPU *immediately* and *imprecisely* (see above).

2.8.6. Serial Memory Controller (SMC)

Hardware source files:	neorv32_smc.vhd	
Software driver files:	neorv32_smc.c	Online software reference (Doxygen)
	neorv32_smc.h	Online software reference (Doxygen)
Top entity ports:	<code>smc_ioen_o</code>	pin enable, can be used for IO multiplexing the external serial memory interface (1-bit)
	<code>smc_sck_o</code>	serial clock (1-bit)
	<code>smc_csn_o</code>	bank/chip select, low-active (2-bit, one bit per memory chip)
	<code>smc_sdo_o</code>	controller data out, memory data in (1-bit)
	<code>smc_sdi_i</code>	controller data in, memory data out (1-bit)
Configuration generics:	<code>SMC_EN</code>	implement serial memory controller when <code>true</code>
	<code>SMC_BASE</code>	serial memory base address (32-bit, has to be 256MB-aligned)
CPU interrupts:	none	

Key Features

- Transparent memory mapping into the processor address space (*execute in-place*, XIP)
- Based on the SPI protocol (clock mode 0) with programmable clock speed
- Manages up to two devices as one linear address space (e.g. 2x PSRAM or PSRAM + flash)
- Fixed protocol frame: 8-bit command + 24-bit address (max 16 MB per memory device)
- Programmable read/write commands with configurable read dummy/wait cycles
- Programmable initialization sequence (up to 3 SPI commands)

Theory of Operation

The SMC transparently maps SPI memory (PSRAM, flash) into the processor's address space so it can be used as normal main memory. The controller is enabled via then `SMC_EN` generic. The `SMC_BASE` generic defines the 256MB-aligned base address, which must not conflict with other regions in the processor [Address Space](#).

Configuration is done via two registers: `CSR1` holds the initialization sequence (see below) and provides the 4 most significant address bits of the mapped region (read-only, derived from `SMC_BASE`). `CSR0` configures the actual memory accesses.

To bring up the controller, set the memory geometry (`SMC_CSR0_DUAL`, `SMC_CSR0_MSIZE`), clock speed (`SMC_CSR0_CDIV`), access commands (`SMC_CSR0_RCMD`, `SMC_CSR0_WCMD`) and read wait cycles (`SMC_CSR0_RWAIT`), then set both `SMC_CSR0_EN` and `SMC_CSR0_IOEN`. Setting `SMC_CSR0_EN` also starts the automatic initialization sequence; clearing it resets the module. `SMC_CSR0_BUSY` indicates an ongoing memory operation (including initialization).



Access Latency

A single memory word access can take several clock cycles, especially at low SPI clock speeds and with many read wait cycles. Using the CPU caches to *hide* some of the latency is highly recommended.



Memory Access Range

Transparent memory accesses are only performed when **both** `SMC_CSR0_EN` and `SMC_CSR0_IOEN` are set and the access is within the configured memory size; otherwise a load/store access fault is raised. Accesses inside the SMC region but outside the configured size are *not* acknowledged and trigger a bus-error exception via the gateway's bus monitor.

Initialization Sequence

Right after being enabled, the SMC issues up to three user-defined single-byte commands to reset/initialize both serial memory devices (at the same time):

1. Eight dummy clock cycles with all chip-selects *disabled*.
2. Command `SMC_CSR1_ICMD0` with both chip-selects *active*.
3. Command `SMC_CSR1_ICMD1` with both chip-selects *active*.
4. Command `SMC_CSR1_ICMD2` with both chip-selects *active*.

Unused command slots can be set to `0x00`.

Serial Clock Speed

The SPI clock is derived from the processor main clock (f_{main}) via the 3-bit divider `SMC_CSR0_CDIV`. Range: $f_{main}/2$ (max) to $f_{main}/32$ (min). The clock is always symmetric (50% duty cycle, SPI clock mode 0).

$$f_{SPI} = f_{main} / (2 * (1 + SMC_CSR0_CDIV))$$

Memory Size and Dual-Chip/Bank Mode

The SMC operates either one device or two **identically-sized** devices combined into a single linear address space, selected by `SMC_CSR0_DUAL` together with `SMC_CSR0_MSIZE`. Each device may be up to 16 MB (24-bit address).

Table 8. SMC dual-mode and memory size configurations

DUAL	MSIZE	Configuration	Size per device	Total mapped size
0	00	1 x 2 MB	2 MB	2 MB
0	01	1 x 4 MB	4 MB	4 MB
0	10	1 x 8 MB	8 MB	8 MB
0	11	1 x 16 MB	16 MB	16 MB
1	00	2 x 2 MB	2 MB	4 MB
1	01	2 x 4 MB	4 MB	8 MB
1	10	2 x 8 MB	8 MB	16 MB
1	11	2 x 16 MB	16 MB	32 MB

In dual mode the active device is selected by the address bit right above the per-device range: the first device occupies the lower half, the second the upper half of the mapped region. The `smc_csn_o` lines are low-active; in single-chip mode (i.e. `SMC_CSR0_DUAL = 0`) only `smc_csn_o(0)` is used.

IO/Pin Control

The `smc_ioen_o` output signal is controlled by the `SMC_CSR0_IOEN` control bit and can be used to enable/multiplex the external serial memory interface pins (`smc_sck_o`, `smc_csn_o`, `smc_sdo_o`, `smc_sdi_i`) on the IO pad ring. This can be used, for example, to multiplex the serial memory interface with the [Serial Peripheral Interface Controller \(SPI\)](#) to provide a software-driven manual access mechanism (e.g. to program an XIP flash, which the SMC is not capable of).

Register Map

Table 9. SMC register map (`struct NEORV32_SMC`)

Address	Name [C]	Bit(s), Name [C]	R/W	Description
0xffef0000	CSR0	0 SMC_CSR0_EN	r/w	SMC module enable, reset module when cleared
		1 SMC_CSR0_IOEN	r/w	Enable control of <code>smc_*</code> SMC IO pins
		2 -	r/-	<i>reserved</i> , read as zero
		3 SMC_CSR0_DUAL	r/w	Enable dual-chip mode
		4 SMC_CSR0_BUSY	r/-	Memory operation in progress when set
		6:5 -	r/-	<i>reserved</i> , read as zero
		8:7 SMC_CSR0_MSIZE_MSB : SMC_CSR0_MSIZE_LSB	r/w	Memory size select
		11:9 SMC_CSR0_CDIV_MSB : SMC_CSR0_CDIV_LSB	r/w	SPI clock divider
		15:12 SMC_CSR0_RWAIT_MSB : SMC_CSR0_RWAIT_LSB	r/w	Read access dummy/wait cycles
		23:16 SMC_CSR0_RCMD_MSB : SMC_CSR0_RCMD_LSB	r/w	Memory read command
0xffef0004	CSR1	31:24 SMC_CSR0_WCMD_MSB : SMC_CSR0_WCMD_LSB	r/w	Memory write command
		7:0 SMC_CSR1_ICMD0_MSB : SMC_CSR1_ICMD0_LSB	r/w	1st memory initialization command
		15:8 SMC_CSR1_ICMD1_MSB : SMC_CSR1_ICMD1_LSB	r/w	2nd memory initialization command
		23:16 SMC_CSR1_ICMD2_MSB : SMC_CSR1_ICMD2_LSB	r/w	3rd memory initialization command
		27:24 -	r/-	<i>reserved</i> , read as zero
		31:28 SMC_CSR1_MEM_BASE_MSB : SMC_CSR1_MEM_BASE_LSB	r/-	Memory base address (top 4 address bits / 256MB page)

2.8.7. Direct Memory Access Controller (DMA)

Hardware source files:	neorv32_dma.vhd	
Software driver files:	neorv32_dma.c	Online software reference (Doxygen)
	neorv32_dma.h	Online software reference (Doxygen)
Top entity ports:	none	
Configuration generics:	IO_DMA_EN	implement DMA when true
	IO_DMA_DSC_FIFO	descriptor FIFO depth, has to be a power of 2, min 4, max 512
CPU interrupts:	fast IRQ channel 10	DMA transfer(s) done (see Processor Interrupts)

Key Features

- CPU-independent data movement
- Byte-wide or word-wide data transfers
- Up to 16MB (bytes) or 64MB (words) per transfer
- Optional Endianness conversion
- Optional descriptor FIFO
- Chaining of pre-programmed transfers
- Transfer-done interrupt

Overview

The NEORV32 DMA features a lightweight direct memory access controller that allows to move and modify data independently of the CPU. Only a single read/write channel is implemented. So only one programmed transfer can be in progress at a time. However, a configurable descriptor FIFO is provided which allows to program several transfers so the DMA can execute them one after the other.

The DMA is connected to the central processor-internal bus system (see section [Address Space](#)) and can access the entire/same address space as the CPU core. It uses *interleaving mode* accessing the central processor bus only if the CPU does not currently request a bus access. The DMA controller can handle different data quantities (e.g. read bytes and write them back as zero-extended words) and can also change the Endianness of data while transferring. It supports reading/writing data from/to fixed or auto-incrementing addresses.



DMA Bus Access

Transactions performed by the DMA are executed as bus transactions with elevated **machine-mode** privilege level. Note that any physical memory protection rules ([Smpmp ISA Extension](#)) are not applied to DMA transfers. Furthermore, the DMA uses single-transfers only (i.e. no burst transfers).



DMA Demo Program

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

Theory of Operation

The DMA provides just two memory-mapped interface registers: A status and control register **CTRL** and another one for writing the transfer descriptor(s) to the internal descriptor FIFO.

The DMA is enabled by setting the **DMA_CTRL_EN** bit of the control register. Clearing this flag will abort any outstanding transfer and will also reset/clear the descriptor FIFO. A programmed DMA transfer is initiated by setting the control register's **DMA_CTRL_START** bit. Setting this bit while the descriptor FIFO is empty has no effect. The current status of the FIFO can be checked via the **DMA_CTRL_D*** flags.

The DMA uses an atomic read-modify-write transfer process. Data is read from the current source address, modified/aligned internally and then written back to the current destination address. If the DMA controller encounters a bus error during operation, it will set the **DMA_CTRL_ERROR** flag and will terminate the current transfer. An new transfer can only start if the **DMA_CTRL_ERROR** flag is cleared manually.

When the **DMA_CTRL_DONE** flag is set the DMA has completed all programmed transfers, i.e. all descriptors from the FIFO were executed. This flag also triggers the DMA controller's interrupt request signal. The application software has to clear **DMA_CTRL_DONE** in order to acknowledge the interrupt and to start further transfers.

DMA Descriptor

All DMA transfers are executed based on *descriptors*. A descriptor contains the data source and destination base addresses as well as the number of elements to transfer and the data type and handling configuration. A complete descriptor is encoded as 3 consecutive 32-bit words:

Table 10. DMA Descriptor

Index	Size	Description
0	32-bit	Source data base address
1	32-bit	Destination data base address
2	32-bit	Transfer configuration word (see next table)



Descriptor FIFO Size

The descriptor FIFO has a minimal size of 4 entries. This can be extended by the **IO_DMA_DSC_FIFO** generic.



Incomplete Descriptors

The DMA controller consumes 3 entries from the FIFO for each transfer. If the FIFO does not provide a complete DMA descriptor, the controller will wait until a

complete descriptor is available.

The source and destination data addresses can target any memory location in the entire 32-bit address space including memory-mapped peripherals. The number of elements to transfer as well as incrementing or constant byte- or word-level transfers are configured via the transfer configuration word (3rd descriptor word):

Table 11. DMA Descriptor - Transfer Configuration Word

Bit(s)	Name	Description
23:0	DMA_CONF_NUM	Number of elements to transfer; must be greater than zero
26:24	-	<i>reserved</i> , set to zero
27	DMA_CONF_BSWAP	Set to swap byte order ("Endianness" conversion)
29:28	DMA_CONF_SRC	Source data configuration (see list below)
31:30	DMA_CONF_DST	Destination data configuration (see list below)

Source and destination data accesses are configured by a 2-bit selector individually for the source and the destination data:

- **00**: Constant byte - transfer data as byte (8-bit); do not alter address during transfer
- **01**: Constant word - transfer data as word (32-bit); do not alter address during transfer
- **10**: Incrementing byte - transfer data as byte (8-bit); increment the source address by 1
- **11**: Incrementing word - transfer data as word (32-bit); increment the source address by 4

Optionally, the controller can automatically swap the logical byte order ("Endianness") of the transferred data when the **DMA_CONF_BSWAP** bit is set.

Register Map

*Table 12. DMA Register Map (**struct** NEORV32_DMA)*

Address	Name [C]	Bit(s), Name [C]	R/W	Function
0xffed0000	CTRL	0 DMA_CTRL_EN	r/w	DMA module enable; reset module when cleared
		1 DMA_CTRL_START	-/w	Start programmed DMA transfer(s)
		15:2 reserved	r/-	reserved, read as zero
		19:16 DMA_CTRL_DFIFO_MSB : DMA_CTRL_DFIFO_LSB	r/-	Descriptor FIFO depth, log2(IO_DMA_DSC_FIFO)
		26:20 reserved	r/-	reserved, read as zero
		27 DMA_CTRL_ACK	-/w	Write 1 to clear DMA interrupt (also clears DMA_CTRL_ERROR and DMA_CTRL_DONE)
		27 DMA_CTRL_DEMPTY	r/-	Descriptor FIFO is empty
		28 DMA_CTRL_DFULL	r/-	Descriptor FIFO is full
		29 DMA_CTRL_ERROR	r/-	Bus access error during transfer or incomplete descriptor data
		30 DMA_CTRL_DONE	r/1	All transfers executed
		31 DMA_CTRL_BUSY	r/-	DMA transfer(s) in progress
0xffed0004	DESC	31:0	-/w	Descriptor FIFO write access

2.8.8. Processor-External Bus Interface (XBUS)

Hardware source files:	neorv32_xbus.vhd	External bus gateway
Software driver files:	none	
Top entity ports:	<code>xbus_adr_o</code>	address output (32-bit)
	<code>xbus_dat_i</code>	data input (32-bit)
	<code>xbus_dat_o</code>	data output (32-bit)
	<code>xbus_cti_o</code>	cycle type (3-bit)
	<code>xbus_tag_o</code>	access tag (3-bit)
	<code>xbus_we_o</code>	write enable (1-bit)
	<code>xbus_sel_o</code>	byte enable (4-bit)
	<code>xbus_stb_o</code>	bus strobe (1-bit)
	<code>xbus_cyc_o</code>	valid cycle (1-bit)
	<code>xbus_ack_i</code>	acknowledge (1-bit)
	<code>xbus_err_i</code>	bus error (1-bit)
Configuration generics:	<code>XBUS_EN</code>	enable external bus interface when <code>true</code>
	<code>XBUS_TIMEOUT</code>	number of clock cycles after which an unacknowledged external bus access will auto-terminate (0 = disabled)
	<code>XBUS_REGSTAGE_EN</code>	implement XBUS register stages
	<code>(CACHE_BLOCK_SIZE)</code>	burst size (defined by global cache block size)
	<code>(CACHE_BURSTS_EN)</code>	enable burst transfers for cache update
CPU interrupts:	none	

Key Features

- Gateway for processor-external modules
- Wishbone-compatible bus protocol
- Optional burst support
- Optional AXI4-compatible bridging

Overview

The external bus interface provides a **Wishbone**-compatible on-chip bus interface. This bus interface can be used to attach processor-external modules like memories, custom hardware accelerators or additional peripheral devices.

*Burst Transfers*

If any cache (**i-cache** or **d-cache**) is implemented and bursts are globally enabled (by the **CACHE_BURSTS_EN** top generic) all cache block transfers are **always executed as burst transfers** with a burst size equal to the cache block size (**CACHE_BLOCK_SIZE** top generic). Burst transfers should **not** be enabled if any external module mapped to *cached Address Space* does not support bursts. Note that burst transactions need to set **ACK** or **ERR** for each burst element. Note that the cycle type identifier signal (**xbus_cti_o**) does not support the end-of-burst identifier.

*Address Mapping*

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

*Wishbone Specs*

The official Wishbone specification can be found online: <https://wishbone-interconnect.readthedocs.io/en/latest/index.html>

*AXI4-Compatible Interface Bridge*

A bridge that converts the processor's XBUS interface into an AXI4-compatible host interface is available in `rtl/system_integration/xbus2axi4_bridge.vhd`. This bridge is also used for the ENORV32 Vivado IP block: https://stnolting.github.io/neorv32/ug/#_packaging_the_processor_as_vivado_ip_block

XBUS Bus Protocol

The external bus interface implements a subset of the **pipelined Wishbone** protocol. Basically, three types of bus transfer are implemented which are illustrated in the following figures. The actual transfer type is indicated via the *cycle type identifier* (**xbus_cti_o**) signal.

1. **Single-access** (**xbus_cti_o = 000**) transfers perform a single read or write operation.
2. **Atomic-access** (**xbus_cti_o = 001**) transfers perform a read followed by a write operation. The bus is locked during the entire transfer (keeping **cyc** high) to maintain exclusive bus access. This transfer type is used by the CPU to perform atomic read-modify-write operations.
3. **Burst read** (**xbus_cti_o = 010**) transfers perform several consecutive read accesses. This transfer type is used by cache block operations.

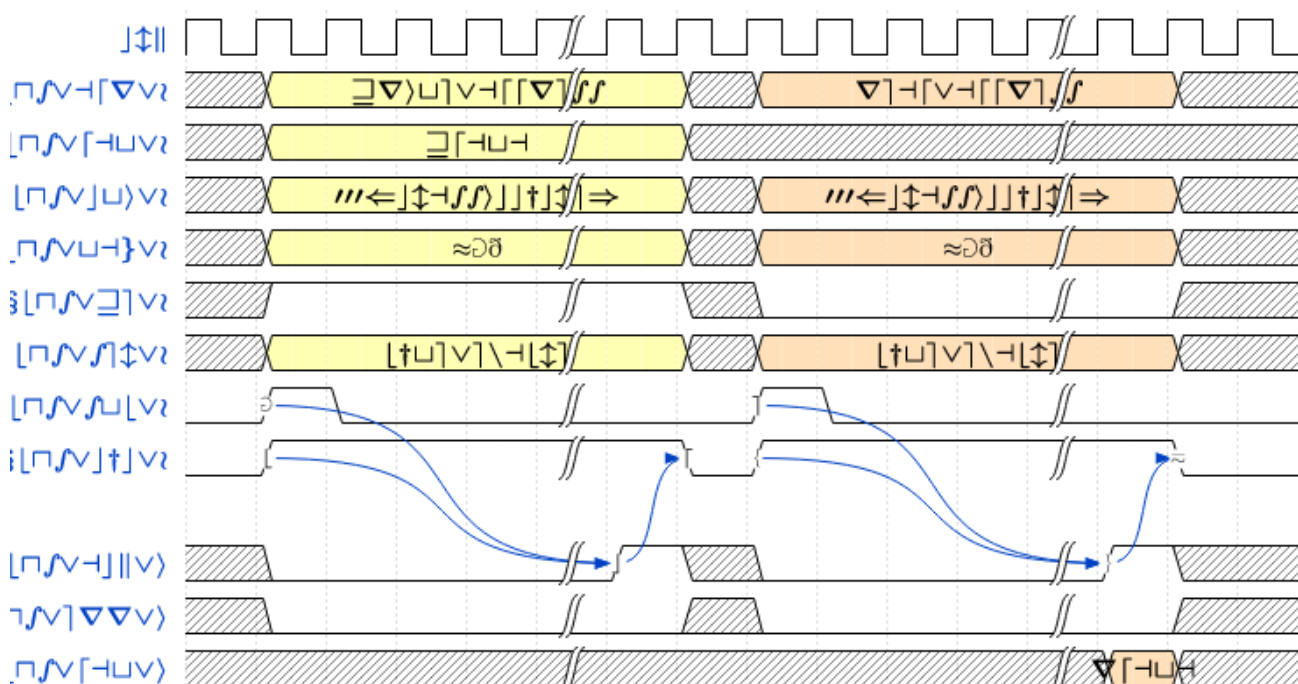


Figure 4. **XBUS Single Access Transfers: Write (left) and Read (right)**

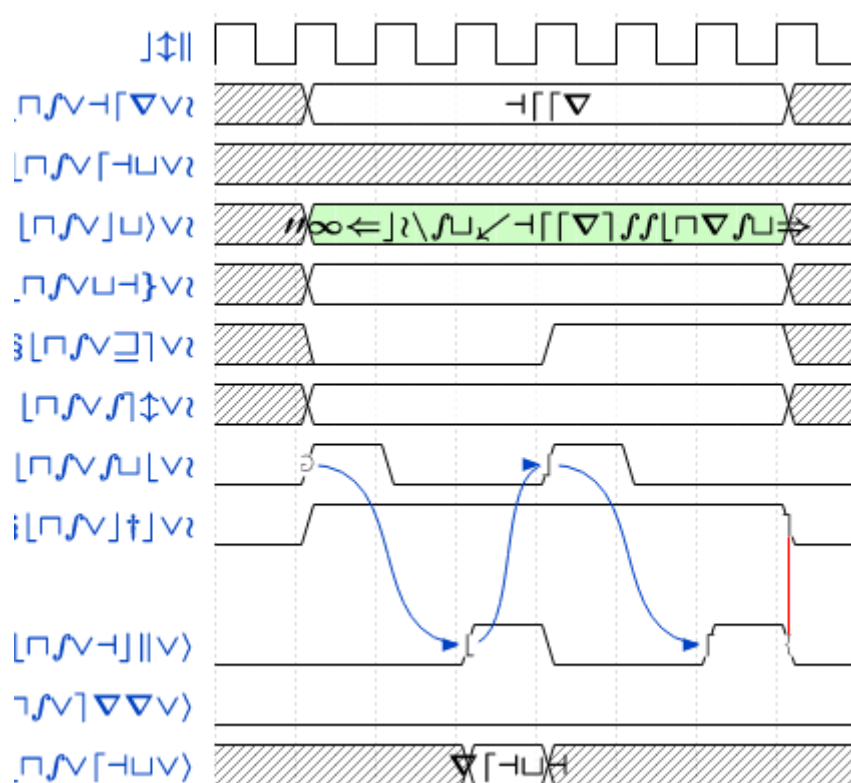


Figure 5. XBUS Atomic Access Transfer

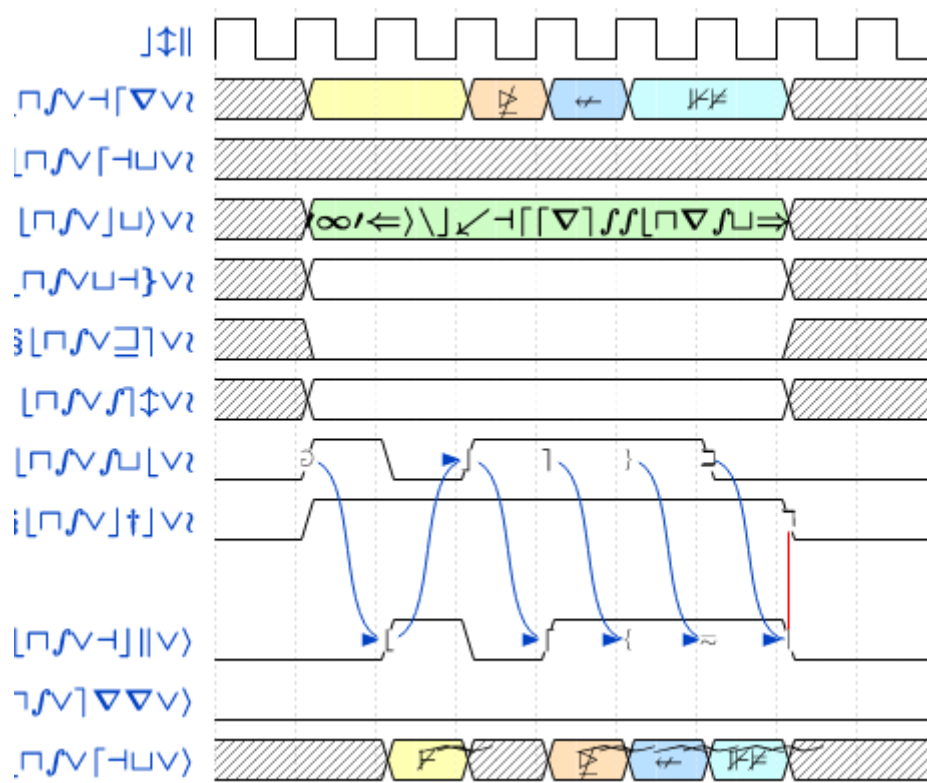


Figure 6. XBUS Burst Read Transfer (4-Words)

Bus Latency

An accessed XBUS device does not have to respond immediately to a bus request by sending an **ACK**. Instead, there is a **time window** where the device has to acknowledge the transfer. This time window is configured by the **XBUS_TIMEOUT** generic. Note that the value provided by this generic is internally extended to the next power of two.

All XBUS transactions have to be acknowledged within this time window. Otherwise the transfer is terminated and a bus fault exception is raised. See section **Bus Monitor and Timeout** for more information.

Furthermore, an accesses XBUS device can signal an error condition at any time by setting the **ERR** signal high for one cycle. This will also terminate the current bus transaction raising a CPU bus fault exception.



Register Stage

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

Access Tag

The XBUS tag signal **xbus_tag_o** provides additional information about the current access cycle. The encoding is compatible to the AXI4 **xPROT** signal.

- `xbus_tag_o(2)` **I**: access is an **instruction** fetch when set; access is a data access when cleared
- `xbus_tag_o(1)` **NS**: this bit is hardwired to `0` indicating a **secure** access
- `xbus_tag_o(0)` **P**: access is performed from **privileged** mode (machine-mode) when set

2.8.9. Stream Link Interface (SLINK)

Hardware source files:	neorv32_slink.vhd	
Software driver files:	neorv32_slink.c	Online software reference (Doxygen)
	neorv32_slink.h	Online software reference (Doxygen)
Top entity ports:	<code>slink_rx_dat_i</code>	RX link data (32-bit)
	<code>slink_rx_src_i</code>	RX routing information (4-bit, optional)
	<code>slink_rx_val_i</code>	RX link data valid (1-bit)
	<code>slink_rx_lst_i</code>	RX link last element of stream (1-bit, optional)
	<code>slink_rx_rdy_o</code>	RX link ready to receive (1-bit)
	<code>slink_tx_dat_o</code>	TX link data (32-bit)
	<code>slink_tx_dst_o</code>	TX routing information (4-bit, optional)
	<code>slink_tx_val_o</code>	TX link data valid (1-bit)
	<code>slink_tx_lst_o</code>	TX link last element of stream (1-bit, optional)
	<code>slink_tx_rdy_i</code>	TX link allowed to send (1-bit)
Configuration generics:	<code>IO_SLINK_EN</code>	implement SLINK when <i>true</i>
	<code>IO_SLINK_RX_FIFO</code>	RX FIFO depth, has to be a power of two, min 1
	<code>IO_SLINK_TX_FIFO</code>	TX FIFO depth, has to be a power of two, min 1
CPU interrupts:	fast IRQ channel 14	SLINK IRQ (see Processor Interrupts)

Key Features

- Compatible to the AXI-4 stream standard
- Independent RX and TX channels
- Optional per-channel FIFOs
- Supports "last" and "source/destination" stream signals
- Interrupt based on FIFO status

Overview

The stream link interface provides independent RX and TX channels for sending and receiving stream data. Each channel features a configurable internal data FIFO (`IO_SLINK_RX_FIFO` and `IO_SLINK_TX_FIFO`). The SLINK interface provides higher bandwidth and less latency than the

external bus interface making it ideally suited for coupling custom stream processors or streaming peripherals.



Example Program

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

Interface & Protocol

Each link/channel consists of the following signals:

- **dat** provides the actual data word
- **val** indicates the current transmission cycle is valid
- **lst** indicates the current transmission cycle is the last element of a stream (optional)
- **rdy** indicates the receiver is ready to receive
- **src** and **dst** provide source/destination routing information (optional)

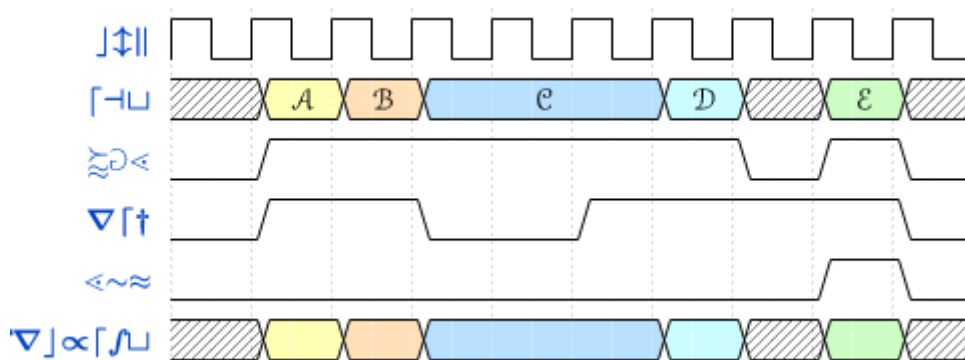


Figure 7. SLINK example transmissions

AXI4-Stream Compatibility



The interface names (except for **src** and **dst**) and the underlying protocol is compatible to the AXI4-Stream protocol standard. A processor top entity with AXI4-Stream-compatible interfaces can be found in [rtl/system_integration](#). More information can be found in the user guide: https://stnolting.github.io/neorv32/ug/#_packaging_the_processor_as_vivado_ip_block

Theory of Operation

The SLINK provides four interface register. The control register (**CTRL**) is used to configure the module and to check its status. The **ROUTE** register can be used to configure a 4-bit routing ID. Two individual data registers (**DATA** and **DATA_LAST**) are used to send and receive the actual data.

The **DATA** register provides direct access to the RX/TX FIFO buffers. Read accesses return data from the RX FIFO. The end-of-stream delimiter as well as the RX routing information are also buffered in the RX FIFO and can be obtained from control register's **SLINK_CTRL_RX_LAST** flag and the **ROUTE** register, respectively, after reading the actual RX data.

Writing to the **DATA** register will immediately write to the TX link FIFO. Writing to the **DATA_LAST**

register will also push data to the TX link FIFO and will also set the end-of-stream delimiter for that data word. TX routing information can be configured via the `ROUTE` register before writing data to `DATA / DATA_LAST`.

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

The SLINK is globally enabled by setting the control register's enable bit `SLINK_CTRL_EN`. Clearing this bit will reset all internal logic and will also clear both FIFOs. Writing to the TX channel's FIFO while it is *full* will have no effect. Reading from the RX channel's FIFO while it is *empty* will also have no effect and will return the last received data word. The current status of the RX and TX FIFOs can be determined via the control register's `SLINK_CTRL_RX_EMPTY`, `SLINK_CTRL_RX_FULL`, `SLINK_CTRL_TX_EMPTY` and `SLINK_CTRL_TX_FULL` flags.

Interrupt

The SLINK module provides a single interrupt request that can be used to signal certain RX/TX data FIFO conditions. The interrupt conditions are based on the RX/TX FIFO status flags `SLINK_CTRL_RX_*` / `SLINK_CTRL_TX_*` and are configured via the according `SLINK_CTRL_IRQ_RX_*` / `SLINK_CTRL_IRQ_TX_*` bits. The SLINK interrupt will fire when the module is enabled (`SLINK_CTRL_EN`) and **any** of the enabled interrupt conditions is met. Hence, all enabled interrupt conditions are logically OR-ed. The interrupt remains active until all interrupt-causing conditions are resolved.

Register Map

Table 13. SLINK register map (`struct NEORV32_SLINK`)

Address	Name [C]	Bit(s)	R/W	Function
0xffec0000	CTRL	0 SLINK_CTRL_EN	r/w	SLINK global enable
		7:1 <i>reserved</i>	r/-	<i>reserved</i> , read as zero
		8 SLINK_CTRL_RX_EMPTY	r/-	RX FIFO empty
		9 SLINK_CTRL_RX_FULL	r/-	RX FIFO full
		10 SLINK_CTRL_TX_EMPTY	r/-	TX FIFO empty
		11 SLINK_CTRL_TX_FULL	r/-	TX FIFO full
		12 SLINK_CTRL_RX_LAST	r/-	Last word read from DATA is end-of-stream
		15:13 <i>reserved</i>	r/-	<i>reserved</i> , read as zero
		16 SLINK_CTRL_IRQ_RX_NEMPTY	r/w	Fire interrupt if RX FIFO not empty
		17 SLINK_CTRL_IRQ_RX_FULL	r/w	Fire interrupt if RX FIFO full
		18 SLINK_CTRL_IRQ_TX_EMPTY	r/w	Fire interrupt if TX FIFO empty
		19 SLINK_CTRL_IRQ_TX_NFULL	r/w	Fire interrupt if TX FIFO not full
		23:20 <i>reserved</i>	r/-	<i>reserved</i> , read as zero
		27:24 SLINK_CTRL_RX_FIFO_MSB : SLINK_CTRL_RX_FIFO_LSB	r/-	log2(RX FIFO size)
		31:28 SLINK_CTRL_TX_FIFO_MSB : SLINK_CTRL_TX_FIFO_LSB	r/-	log2(TX FIFO size)
0xffec0004	ROUTE	3:0	-/w	TX destination routing information (<i>slink_tx_dst_o</i>)
		3:0	r/-	RX source routing information (<i>slink_rx_src_i</i>)
		31:4	r/-	<i>reserved</i> , read as zero
0xffec0008	DATA	31:0	r/w	Write data to TX FIFO; read data from RX FIFO
0xffec000c	DATA_LAST	31:0	r/w	Write data to TX FIFO (and also set end-of-stream delimiter); read data from RX FIFO

2.8.10. General Purpose Input and Output Port (GPIO)

Hardware source files:	neorv32_gpio.vhd	
Software driver files:	neorv32_gpio.c	Online software reference (Doxygen)
	neorv32_gpio.h	Online software reference (Doxygen)
Top entity ports:	<code>gpio_dir_o</code>	32-bit direction control output port
	<code>gpio_o</code>	32-bit parallel output port
	<code>gpio_i</code>	32-bit parallel input port
Configuration generics:	<code>IO_GPIO_NUM</code>	number of input/output pairs to implement (0..32)
	<code>IO_GPIO_DIR_EN</code>	enable direction control feature when <i>true</i>
CPU interrupts:	fast IRQ channel 8	input-pin change interrupt (see Processor Interrupts)

Key Features

- Up to 32 inputs and 32 outputs
- Each input pin is interrupt-capable
- Configurable interrupt triggers (positive/negative level or edge)

Overview

The general purpose IO module provides simple unidirectional - or optionally bidirectional - inputs and outputs. These connections can be used externally (for example, to control status LEDs, connect buttons, etc.) or internally to provide control signals for other IP modules. The input port features programmable, pin-specific level- or edge-triggered interrupts.

Data written to the `PORT_OUT` will appear on the processor's `gpio_o` port. Vice versa, the `PORT_IN` register represents the current state of the processor's `gpio_i`.

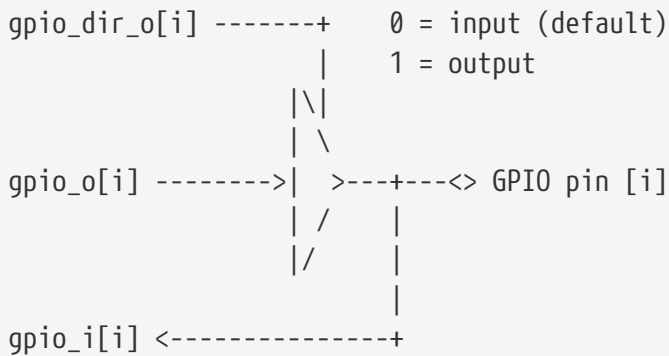
The actual number of input/output pairs is defined by the `IO_GPIO_NUM` generic. When set to zero, the GPIO module is excluded from synthesis and the output port `gpio_o` is tied to all-zero. If `IO_GPIO_NUM` is less than the maximum value of 32, only the LSB-aligned bits of `gpio_o` and `gpio_i` are actually connected while the remaining ones are tied to zero or are left unconnected, respectively. This also applies to all memory-mapped interface registers of the GPIO module (i.e. the according most-significant bits are hardwired to zero and are read-only).

Direction Control

The IO ports can also be used as bidirectional ports. The direction feature must be explicitly enabled via the `IO_GPIO_DIR_EN` generic. When enabled, the `PORT_DIR` register is used to control the direction of each individual IO signal by directly driving the top's `gpio_dir_o` port. Setting bit *i* to 1 configures the according pin as output. Clearing bit *i* configures the according pin as input. Note

that the `PORT_DIR` register resets to all-zero configuring all bi-directional IOs as inputs.

Listing 6. GPIO Tristate Driver Wiring



Listing 7. GPIO VHDL Tristate Driver Example (external `gpio_io` port is of type `std_logic_vector`)

```

gpio_tristate_gen:
for i in 0 to IO_GPIO_NUM-1 generate
  gpio_io(i) <= std_logic(gpio_o(i)) when (gpio_dir_o(i) = '1') else 'Z'; -- drive
end generate;

gpio_i(IO_GPIO_NUM-1 downto 0) <= std_ulogic_vector(gpio_io(IO_GPIO_NUM-1 downto 0));
-- sense
  
```



GPIO_DIR As Additional Output Port

If the bi-directional feature is not required, the `gpio_dir_o` port can be used as additional output port.

Interrupts

Each input pin (`gpio_i`) provides an individual programmable interrupt trigger. The actual interrupt trigger type can be configured individually for each input pin using the `IRQ_TYPE` and `IRQ_POLARITY` registers. `IRQ_TYPE` defines the actual trigger type (level-triggered or edge-triggered), while `IRQ_POLARITY` defines the trigger's polarity (low-level/falling-edge or high-level/rising-edge). The position of each bit in these registers corresponds to the according `gpio_i` input pin.

Each pin interrupt channel can be enabled or disabled individually using the `IRQ_ENABLE` register. Each bit in this register corresponds to the according input pin. If the programmed trigger of a disabled input (`IRQ_ENABLE(i) = 0`) fires, the interrupt request is entirely ignored.

Table 14. GPIO Trigger Configuration for Pin *i*

<code>IRQ_ENABLE(i)</code>	<code>IRQ_TYPE(i)</code>	<code>IRQ_POLARITY(i)</code>	Resulting trigger of <code>gpio_i(i)</code>
1	0	0	low-level (<code>GPIO_TRIG_LEVEL_LOW</code>)
1	0	1	high-level (<code>GPIO_TRIG_LEVEL_HIGH</code>)
1	1	0	falling-edge (<code>GPIO_TRIG_EDGE_FALLING</code>)

IRQ_ENABLE(i)	IRQ_TYPE(i)	IRQ_POLARITY(i)	Resulting trigger of gpio_i(i)
1	1	1	rising-edge (GPIO_TRIG_EDGE_RISING)
0	-	-	interrupt disabled

If the configured trigger of an enabled input pin (`IRQ_ENABLE(i) = 1`) fires, the according interrupt request is buffered internally in the `IRQ_PENDING` register. When this register contains a non-zero value (i.e. any bit becomes set) an interrupt request is sent to the CPU via FIRQ channel 8 (see [Processor Interrupts](#)).

The CPU can determine the interrupt-triggering pins by reading the `IRQ_PENDING` register. Each set bit in this register indicates that the according input pin's interrupt trigger has fired. Then, the CPU can clear those pending interrupt pin by setting all set bits to zero.



GPIO Interrupts Demo Program

A demo program for the GPIO input interrupts can be found in [sw/example/demo_gpio](#).

Register Map

Table 15. GPIO unit register map (`struct NEORV32_GPIO`)

Address	Name [C]	Bit(s)	R/W	Function
0xfffc0000	PORT_IN	31:0	r/-	Parallel input port; <code>PORT_IN(i)</code> corresponds to <code>gpio_i(i)</code>
0xfffc0004	PORT_OUT	31:0	r/w	Parallel output port; <code>PORT_OUT(i)</code> corresponds to <code>gpio_o(i)</code>
0xfffc0008	PORT_DIR	31:0	r/w	Port direction; <code>PORT_DIR(i)</code> corresponds to <code>gpio_o(i)</code> ; 0 = input (default), 1 = output; this register is hardwired to all-zero if <code>IO_GPIO_DIR_EN = false</code>
0xfffc000c	-	31:0	r/-	reserved, read as zero
0xfffc0010	IRQ_TYPE	31:0	r/w	Trigger type select (0 = level trigger, 1 = edge trigger); <code>IRQ_TYPE(i)</code> corresponds to <code>gpio_i(i)</code>
0xfffc0014	IRQ_POLARITY	31:0	r/w	Trigger polarity select (0 = low-level/falling-edge, 1 = high-level/rising-edge); <code>IRQ_POLARITY(i)</code> corresponds to <code>gpio_i(i)</code>
0xfffc0018	IRQ_ENABLE	31:0	r/w	Per-pin interrupt enable; <code>IRQ_ENABLE(i)</code> corresponds to <code>gpio_i(i)</code>
0xfffc001c	IRQ_PENDING	31:0	r/c	Per-pin interrupt pending, can be cleared by writing zero to the according bit(s); <code>IRQ_PENDING(i)</code> corresponds to <code>gpio_i(i)</code>

2.8.11. Watchdog Timer (WDT)

Hardware source files:	neorv32_wdt.vhd	
Software driver files:	neorv32_wdt.c	Online software reference (Doxygen)
	neorv32_wdt.h	Online software reference (Doxygen)
Top entity ports:	<code>rstn_wdt_o</code>	synchronous watchdog reset output, low-active
Configuration generics:	<code>IO_WDT_EN</code>	implement watchdog when <code>true</code>
CPU interrupts:	none	

Key Features

- 32-bit timeout counter with 3-bit clock divider
- Password-protected; lockable configuration
- Provides information about the cause of the last reset

Overview

The watchdog (WDT) provides a last resort for safety-critical applications. When a pre-programmed timeout value is reached a system-wide hardware reset is generated. The internal counter has to be reset explicitly by the application program every now and then to prevent a timeout.

Theory of Operation

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

The internal 24-bit timeout counter is clocked at 1/4096 of the processor's main clock. Whenever this counter reaches the programmed timeout value (24-bit `WDT_CTRL_TIMEOUT` value) a hardware reset is triggered. The timeout counter is reset by writing the reset **PASSWORD** to the `RESET` register ("feeding the watchdog"). The password is hardwired to hexadecimal `0x709D1AB3`.

$$T_{\text{timeout}} = (1 / f_{\text{main}}[\text{Hz}]) * 4096 * \text{WDT_CTRL_TIMEOUT}$$



Operation in Sleep-Mode and During Debugging

Once enabled, the watchdog keeps operating even if the CPU is in **Sleep Mode** or if the processor is being debugged via the **On-Chip Debugger (OCD)**.

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. The lock bit can only be cleared by a *hardware* reset

In the locked state any write access to the control register will trigger an immediate hardware reset (read accesses are still possible and have no side effects). Additionally, writing an incorrect password to the **RESET** register will also trigger an immediate hardware reset.

Cause of last Hardware Reset

The cause of the last system hardware reset can be determined via the two **WDT_CTRL_RCAUSE** bits:

- **WDT_RCAUSE_EXT** (0b00): Reset caused by external reset signal pin
- **WDT_RCAUSE_OCD** (0b01): Reset caused by on-chip debugger
- **WDT_RCAUSE_TMO** (0b10): Reset caused by watchdog timeout
- **WDT_RCAUSE_ACC** (0b11): Reset caused by illegal watchdog access

External Reset Output

The WDT provides a dedicated output (**Processor Top Entity - Signals: rstn_wdt_o**) to reset processor-external modules when the watchdog times out. This signal is low-active and synchronous to the processor clock. It is available if the watchdog is implemented; otherwise it is hardwired to **1**. Note that the signal also becomes active (low) when the processor's main reset signal is active (even if the watchdog is deactivated or disabled for synthesis).

Register Map

Table 16. WDT register map (**struct** **NEORV32_WDT**)

Address	Name [C]	Bit(s), Name [C]	R/W	Reset value	Writable if locked	Function
0xffffb0000	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
		3:2 WDT_CTRL_RCAUSE_HI : WDT_CTRL_RCAUSE_LO	r/-	0	-	Cause of last system reset
		7:4 -	r/-	-	-	<i>reserved</i> , reads as zero
		31:8 WDT_CTRL_TIMEOUT_MSB : WDT_CTRL_TIMEOUT_LSB	r/w	0	no	Timeout value (24-bit)
0xffffb0004	RESET	31:0	-/w	-	yes	Write <i>PASSWORD</i> to reset WDT timeout counter

2.8.12. Core-Local Interruptor (CLINT)

Hardware source files:	neorv32_clint.vhd	
Software driver files:	neorv32_clint.c	Online software reference (Doxygen)
	neorv32_clint.h	Online software reference (Doxygen)
Top entity ports:	<code>irq_msi_i</code>	RISC-V software IRQ if CLINT is not implemented
	<code>irq_mti_i</code>	RISC-V machine timer IRQ if CLINT is not implemented
	<code>mtime_time_o</code>	Current system time (from CLINT's MTIMER)
Configuration generics:	<code>IO_CLINT_EN</code>	implement core local interruptor when <code>true</code>
CPU interrupts:	<code>MTI</code>	machine timer interrupt (see Processor Interrupts)
	<code>MSI</code>	machine software interrupt (see Processor Interrupts)

Key Features

- Compatible to the RISC-V ACLINT specification
- Global 64-bit system time
- Timer and software interrupts for each CPU core

Overview

The core-local interruptor provides machine-level timer and software interrupts for a set of CPU cores ("harts"). It is compatible to the original SiFive® CLINT specifications and it is also backwards-compatible to the RISC-V *Advanced Core-Local Interruptor (ACLINT)* specifications. In terms of the ACLINT spec the NEORV32 CLINT implements three *devices* that are placed next to each other in the address space: an MTIMER including time-compares and an MSWI device. The CLINT can support up to 4095 harts. However, the NEORV32 CLINT is configured for up to two harts only. Hence, only the according registers are implemented while the remaining ones are hardwired to zero.



RISC-V ACLINT Specs.

The original RISC-V ACLINT specifications are available online at <https://github.com/riscvarchive/riscv-aclint>.

MTIMER Device

The MTIMER device provides a global 64-bit machine timer (`NEORV32_CLINT→MTIME`) that increments with every main processor clock tick. Upon reset the timer is reset to all zero. Each hart provides an

individual 64-bit timer-compare register (**NEORV32_CLINT**→**MTIMECMP[0]** for hart 0). Whenever **MTIMECMP** $\geq$ **MTIME** the according hart's machine timer interrupt becomes pending.



CSR Access

The MTIME system time is also available as read-only shadow copy via the **time[h]** CSRs.

MSWI Device

The MSWI provides a software interrupt for each hart via hart-individual memory-mapped registers (e.g. **NEORV32_CLINT**→**MSWI[0]** for hart 0). Setting bit 0 of this register will bring the machine software interrupt for the specific hart into pending state.



External Machine Timer and Software Interrupts

If the NEORV32 CLINT module is **disabled** (**IO_CLINT_EN** = **false**) the core's machine timer interrupt and machine software interrupt become available as processor-external signals (**irq_mti_i** and **irq_msi_i**, respectively).

Register Map

Table 17. CLINT register map (**struct** **NEORV32_CLINT**)

Address	Name [C]	Bits	R/W	Function
0xfff40000	MSWI[0]	0	r/w	trigger machine software interrupt for hart 0 when set
		31:1	r/-	hardwired to zero
0xfff40004	MSWI[1]	0	r/w	trigger machine software interrupt for hart 1 when set
		31:1	r/-	hardwired to zero
0xfff44000	MTIMECMP[0]	63:0	r/w	64-bit time compare for hart 0
0xfff44008	MTIMECMP[1]	63:0	r/w	64-bit time compare for hart 1
0xfff4bff8	MTIME	63:0	r/w	64-bit global machine timer

2.8.13. Primary Universal Asynchronous Receiver and Transmitter (UART0)

Hardware source files:	neorv32_uart.vhd	
Software driver files:	neorv32_uart.c	Online software reference (Doxygen)
	neorv32_uart.h	Online software reference (Doxygen)
Top entity ports:	uart0_txd_o	serial transmitter output
	uart0_rxd_i	serial receiver input
	uart0_rtsn_o	flow control: RX ready to receive, low-active
	uart0_ctsn_i	flow control: RX ready to receive, low-active
Configuration generics:	IO_UART0_EN	implement UART0 when true
	UART0_RX_FIFO	RX FIFO depth (power of 2, min 1)
	UART0_TX_FIFO	TX FIFO depth (power of 2, min 1)
CPU interrupts:	fast IRQ channel 2	Programmable FIFO status interrupt (see Processor Interrupts)

Key Features

- Independent RX and TX lines
- Fixed format: 8 data bits, 1 stop bit, no parity bit
- Programmable baud rate
- Optional RX and TX FIFO buffers
- Optional support for hardware flow-control
- Interrupt based on FIFO buffer status

Overview

The NEORV32 UART provides a a standard universal asynchronous serial interface with independent transmitter and receiver channels, each equipped with a configurable FIFO. The transmission frame is fixed to **8-N-1**: 8 data bits, no parity bit, 1 stop bit. The actual transmission baud rate is programmable via software.



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 control register `CTRL`. A new TX transmission is started by writing to the `DATA` register. RX data is available via the `DATA` register. 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 select (`clock_prescaler`).

Table 18. UART0 Clock Configuration

<code>UART_CTRL_PRSC[2:0]</code>	0b000	0b001	0b010	0b011	0b100	0b101	0b110	0b111
Resulting <code>clock_prescaler</code>	2	4	8	64	128	1024	2048	4096

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

RX and TX FIFOs

The UART provides individual data FIFOs for RX and TX to allow data transmission without CPU intervention. The sizes of these FIFOs can be configured via the according configuration generics (`UART0_RX_FIFO` and `UART0_TX_FIFO`). Write to `DATA` and reads from `DATA` are automatically buffered by the according FIFO. Both FIFOs are automatically cleared when disabling the module via the `UART_CTRL_EN` flag. The control register's `UART_CTRL_RX_*` and `UART_CTRL_TX_*` flags provide information about the RX and TX FIFO fill level.



RX/TX FIFO Size

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

UART Interrupt

The UART module provides a single interrupt line that can be used to signal certain RX/TX data FIFO conditions. The interrupt conditions are based on the RX/TX FIFO status flags (`UART_CTRL_RX_*` / `UART_CTRL_TX_*`) and are configured via the according `UART_CTRL_IRQ_RX_*` / `UART_CTRL_IRQ_TX_*` bits. The UART interrupt will fire when the module is enabled (`SLINK_CTRL_EN`) and **any** of the enabled interrupt conditions is met. Hence, all enabled interrupt conditions are logically OR-ed. The interrupt remains active until all interrupt-causing conditions are resolved.

RTS/CTS Hardware Flow Control

The NEORV32 UART supports optional hardware flow control using the standard CTS `uart0_ctsn_i` ("clear to send") and RTS `uart0_rtsn_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:

- The UART's transmitter will not start a new transmission until the `uart0_ctsn_i` signal goes low. During this time the UART busy flag `UART_CTRL_TX_BUSY` remains set.
- The UART will set `uart0_rtsn_o` signal low if the RX FIFO is not already full. `uart0_rtsn_o` is

always low if hardware flow-control is disabled. Disabling the UART (setting `UART_CTRL_EN` low) while having hardware flow-control enabled, will set `uart0_rtsn_o` high to indicate that the UART is not capable of receiving data.

Simulation Mode

The simulation mode of the UART allows to redirect TX data to the simulator console instead of sending it via the physical `uart0_txd_o` signal. Simulation mode is enabled by setting the `UART_CTRL_SIM_MODE` bit. When enabled, all data written to the `DATA` register is sent to the simulator and printed as ASCII characters to the simulator console. Note that this feature is only available within a simulation.

Register Map

Table 19. UART0 register map (`struct NEORV32_UART0`)

Address	Name [C]	Bit(s), Name [C]	R/W	Function
0xfff50000	CTRL	0 UART_CTRL_EN	r/w	UART enable
		1 UART_CTRL_SIM_MODE	r/w	enable simulation mode
		2 UART_CTRL_HWFC_EN	r/w	enable RTS/CTS hardware flow-control
		5:3 UART_CTRL_PRSC_MSB : UART_CTRL_PRSC_LSB	r/w	baud rate clock prescaler select
		15:6 UART_CTRL_BAUD_MSB : UART_CTRL_BAUD_LSB	r/w	10-bit baud value configuration value
		16 UART_CTRL_RX_EMPTY	r/-	RX FIFO not empty (data available)
		17 UART_CTRL_RX_FULL	r/-	RX FIFO full
		18 UART_CTRL_TX_EMPTY	r/-	TX FIFO empty
		19 UART_CTRL_TX_NFULL	r/-	TX FIFO not full
		20 UART_CTRL_IRQ_RX_EMPTY	r/w	fire RX-IRQ if RX FIFO not empty
		21 UART_CTRL_IRQ_RX_FULL	r/w	fire RX-IRQ if RX FIFO full
		22 UART_CTRL_IRQ_TX_EMPTY	r/w	fire TX-IRQ if TX FIFO empty
		23 UART_CTRL_IRQ_TX_NFULL	r/w	fire TX-IRQ if TX not full
		29:24 -	r/-	<i>reserved</i> , read as zero
		30 UART_CTRL_RX_OVER	r/-	RX FIFO overflow; cleared on read access
		31 UART_CTRL_TX_BUSY	r/-	TX busy or TX FIFO not empty

Address	Name [C]	Bit(s), Name [C]	R/W	Function
0xfff50004	DATA	7:0 UART_DATA_RTX_MSB : UART_DATA_RTX_LSB	r/w	receive/transmit data
		11:8 UART_DATA_RX_FIFO_MSB : UART_DATA_RX_FIFO_LSB	r/-	log2(RX FIFO size)
		15:12 UART_DATA_TX_FIFO_MSB : UART_DATA_TX_FIFO_LSB	r/-	log2(TX FIFO size)
		31:16	r/-	<i>reserved</i> , read as zero

2.8.14. Secondary Universal Asynchronous Receiver and Transmitter (UART1)

Hardware source files:	neorv32_uart.vhd	
Software driver files:	neorv32_uart.c	Online software reference (Doxygen)
	neorv32_uart.h	Online software reference (Doxygen)
Top entity ports:	uart1_txd_o	serial transmitter output
	uart1_rxd_i	serial receiver input
	uart1_rtsn_o	flow control: RX ready to receive, low-active
	uart1_ctsn_i	flow control: RX ready to receive, low-active
Configuration generics:	IO_UART1_EN	implement UART1 when true
	UART1_RX_FIFO	RX FIFO depth (power of 2, min 1)
	UART1_TX_FIFO	TX FIFO depth (power of 2, min 1)
CPU interrupts:	fast IRQ channel 3	Programmable FIFO status interrupt (see Processor Interrupts)

Overview

The secondary UART (UART1) is functionally identical to the primary UART ([Primary Universal Asynchronous Receiver and Transmitter \(UART0\)](#)). UART1 uses different addresses for the control register (**CTRL**) and the data register (**DATA**) and uses a different CPU fast interrupt (FIRQ) channel.

Register Map

Table 20. UART1 register map (**struct** **NEORV32_UART1**)

Address	Name [C]	Bit(s), Name [C]	R/W	Function
0xffff60000	CTRL	...	...	Same as UART0
0xffff60004	DATA	...	...	Same as UART0

2.8.15. Serial Peripheral Interface Controller (SPI)

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

Key Features

- SPI-compatible host controller
- Programmable clock phase and polarity
- Fine-grained programmable clock
- 8 dedicated chip-select lines
- Optional data/command FIFO (ring-buffer)
- Interrupt if programmed transfer sequences have completed

Overview

The NEORV32 SPI module provides a **host-mode** serial peripheral interface. The module operates on byte-wide data, supports all 4 standard SPI clock modes, provides a precise SPI clock generator and implements 8 dedicated chip select signals via the top entity's **spi_csn_o** signal. An optional receive/transmit ring-buffer/FIFO can be configured via the **IO_SPI_FIFO** generic to support programming of complete SPI transmissions without CPU interaction.



Host-Mode Only

The NEORV32 SPI module only supports *host mode*. Transmission are initiated only by the SPI module itself. 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 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 and for issuing chip-select commands.

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 resets the entire module, clears the RX/TX FIFO and terminates any transfer being in process.

The actual SPI transfer (receiving one byte while also sending one byte) as well as control of the chip-select lines is handled by the module's **DATA** register. Note that this register will access the TX FIFO of the ring-buffer when writing to it and will access the RX FIFO of the ring-buffer when reading from it.

The most significant bit of the **DATA** register (**SPI_DATA_CMD**) is used to select the purpose of the data being written. When the **SPI_DATA_CMD** is cleared, the lowest 8-bit represent the actual SPI TX data that will be transmitted by the SPI engine. After completion, the according receive data is pushed to the RX FIFO.

If **SPI_DATA_CMD** is set, the lowest 4-bit control the chip-select lines. In this case, bits **2:0** select one of the eight chip-select lines. The selected line will become enabled when bit **3** is set. If bit **3** is cleared, all chip-select lines will be disabled at once. Note that the bits of the **spi_csn_o** port are low-active. Only one chip-select can be active at a time.

Since all SPI operations are controlled via the FIFOs, entire SPI sequences (chip-enable, data transmission(s), chip-disable) can be *programmed*. Thus, SPI operations can be executed without any CPU interaction at all and can also be coordinated by the system's DMA controller.

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*.

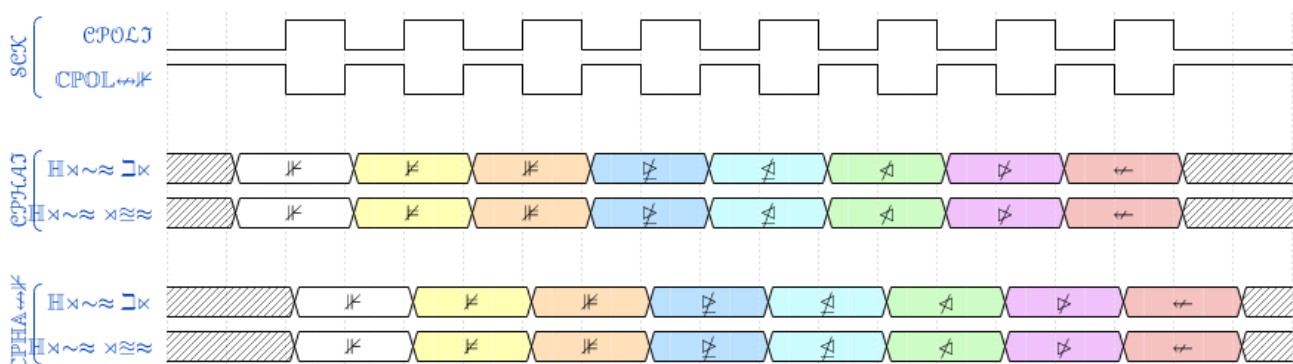


Figure 8. SPI Clock Modes

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 21. SPI prescaler configuration

SPI_CTRL_PRSC[2:0]	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_{SPI} is derived from the processor's main clock f_{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 exactly 50%.

SPI Interrupt

The SPI module provides a single interrupt that gets triggered when the programmed SPI sequence has completed (i.e. the TX FIFO is empty and the SPI engine is idle).

Register Map

Table 22. SPI register map (`struct NEORV32_SPI`)

Address	Name [C]	Bit(s), Name [C]	R/W	Function
0xfff80000	CTRL	0 SPI_CTRL_EN	r/w	SPI module enable
		1 SPI_CTRL_CPHA	r/w	clock phase
		2 SPI_CTRL_CPOL	r/w	clock polarity
		5:3 SPI_CTRL_PRSC2 : SPI_CTRL_PRSC0	r/w	3-bit clock prescaler select
		9:6 SPI_CTRL_CDIV3 : SPI_CTRL_CDIV0	r/w	4-bit clock divider for fine-tuning
		15:10 -	r/-	<i>reserved</i> , read as zero
		16 SPI_CTRL_RX_AVAIL	r/-	RX FIFO data available (RX FIFO not empty)
		17 SPI_CTRL_TX_EMPTY	r/-	TX FIFO empty
		18 SPI_CTRL_TX_FULL	r/-	TX FIFO full
		23:19 -	r/-	<i>reserved</i> , read as zero
		27:24 SPI_CTRL_FIFO_MSB : SPI_CTRL_FIFO_LSB	r/-	FIFO depth; $\log_2(\text{IO_SPI_FIFO})$
		29:28 -	r/-	<i>reserved</i> , read as zero
		30 SPI_CS_ACTIVE	r/-	Set if any chip-select line is active
		31 SPI_CTRL_BUSY	r/-	SPI module busy when set (serial engine operation in progress and TX FIFO not empty yet)

Address	Name [C]	Bit(s), Name [C]	R/W	Function
0xfff80004	DATA	7:0 SPI_DATA_MSB : SPI_DATA_LSB	r/w	receive/transmit data (FIFO), only for data mode (SPI_DATA_CMD = 0)
		3:0	-/w	chip-select-enable (bit 3) and chip-select (bit 2:0), only for command mode (SPI_DATA_CMD = 1)
		30:8 -	r/-	reserved, read as zero
		31 SPI_DATA_CMD	-/w	0 = data, 1 = chip-select-command

2.8.16. Serial Data Interface Controller (SDI)

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

Key Features

- SPI-compatible device-side controller
- Programmable SPI clock polarity
- Optional data RX/TX FIFO
- Interrupt based on FIFO status

Overview

The serial data interface module provides a **device-side** SPI interface to receive communications from an **external SPI host**, which is responsible for performing the actual transmission (i.e. the host generates the SPI clock and control the chip-select line). An optional receive/transmit FIFO can be configured via the `IO_SDI_FIFO` generic to support transmissions without CPU interaction.



Device-Mode Only

The NEORV32 SDI module only supports *device mode*. If you are looking for a *host-mode* serial peripheral interface (transactions performed 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 its status and a single data register `DATA` for receiving/transmitting data. Any access to the `DATA` register actually accesses the internal RX/TX FIFO.

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 and will also clear the entire RX/TX FIFO.

The SDI operates on byte-level. Data bytes written to the **DATA** register will be pushed to the internal TX FIFO. This TX data will be sent back to the external host during an SPI transfer. If no data is available in the TX FIFO all-zero is sent. Received bytes are pushed to the RX FIFO and can be retrieved by reading the RX FIFO via the **DATA** register. Data is always transferred MSB-first. The current state of these FIFOs is available via the control register's **SDI_CTRL_RX_*** and **SDI_CTRL_TX_*** status flags. The RX/TX FIFOs can be cleared at any time by the **SDI_CTRL_CLR_*** control register bits.

Data is only transferred (and pushed to the RX FIFO / popped from the TX FIFO) when the chip-select input **sdi_csn_i** is active (low-active). If the external SPI host aborts the transmission by setting the chip-select signal high again *before* 8 data bits have been transferred, no data is written to the RX FIFO and no data is retrieved from the TX FIFO.

SDI Clocking

The SDI module supports both SPI clock polarity modes ("CPOL"), but only "CPHA=0"-clock-phase is *officially* supported yet. However, experiments have shown that the SDI module can also deal with both clock phase modes.

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. This clock synchronization requires the external SDI clock (**sdi_clk_i**) to **not exceed 1/4 of the processor's main clock**.

SDI Interrupt

The SDI module provides a single interrupt that is triggered by a set of programmable interrupt conditions that are based on the status of the RX & TX FIFOs. The different interrupt sources are enabled by setting the according **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.

Register Map

Table 23. SDI register map (**struct NEORV32_SDI**)

Address	Name [C]	Bit(s), Name [C]	R/W	Function
0xfff70000	CTRL	0 SDI_CTRL_EN	r/w	SDI module enable
		1 SDI_CTRL_CLR_RX	-/w	clear RX FIFO, flag auto-clears
		2 SDI_CTRL_CLR_TX	-/w	clear TX FIFO, flag auto-clears
		3 -	r/-	<i>reserved</i> , read as zero
		7:4 SDI_CTRL_FIFO_MSB : SDI_CTRL_FIFO_LSB	r/-	FIFO depth; $\log_2(IO_SDI_FIFO)$
		15:8 -	r/-	reserved, read as zero
		16 SDI_CTRL_IRQ_RX_EMPTY	r/w	fire interrupt if RX FIFO is not empty
		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
		23:19 -	r/-	<i>reserved</i> , read as zero
		24 SDI_CTRL_RX_EMPTY	r/-	RX FIFO empty
		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
		30:28 -	r/-	<i>reserved</i> , read as zero
		31 SDI_CTRL_CS_ACTIVE	r/-	Chip-select is active when set
0xfff70004	DATA	7:0 SDI_DATA_MSB : SDI_DATA_LSB	r/w	receive/transmit data (FIFO)
		31:8 -	r/-	<i>reserved</i> , read as zero

2.8.17. Two-Wire Serial Interface Controller (TWI)

Hardware source files:	neorv32_twi.vhd	
Software driver files:	neorv32_twi.c	Online software reference (Doxygen)
	neorv32_twi.h	Online software reference (Doxygen)
Top entity ports:	<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>
	<code>IO_TWI_FIFO</code>	FIFO depth, has to be a power of two, min 1
CPU interrupts:	fast IRQ channel 7	FIFO empty and module idle interrupt (see Processor Interrupts)

Key Features

- Philips I²C-compatible host controller
- Programmable clock speed
- Support for host-ACK generated by the controller
- Optional support for clock stretching
- Optional data/command FIFO
- Interrupt based on FIFO status

Overview

The NEORV32 TWI implements an I²C-compatible host controller to communicate with arbitrary I²C-devices. Note that multi-controller mode and bus arbitration are not supported.



Host-Mode Only

The NEORV32 TWI controller only supports **host mode**. Transmission are initiated by the processor's TWI controller and not by an external I²C module. If you are looking for a *device-mode* module (transactions initiated by an external host) check out the [Two-Wire Serial Device Controller \(TWD\)](#).

The TWI controller provides two memory-mapped registers that are used for configuration & status check (**CTRL**) and for accessing transmission data (**DATA**). The **DATA** register is transparently buffered by separate RX and TX FIFOs. The size of those FIFOs can be configured by the `IO_TWI_RX_FIFO` and `IO_TWI_TX_FIFO` generics. Software can determine the FIFO size via the control register's

`TWI_CTRL_FIFO_*` bits. The current status of the RX and TX FIFO can be polled by software via the `TWD_CTRL_RX_*` and `TWI_CTRL_TX_*` flags.

The module is globally enabled by setting the control register's `TWI_CTRL_EN` bit. Clearing this bit will disable and reset the entire module also clearing the internal RX and TX FIFOs.



Current Bus State

The current state of the I2C bus lines (SCL and SDA) can be checked by software via the `TWI_CTRL_SENSE_*` control register bits. Note that the TWI module needs to be enabled in order to sample the bus state.

TWI Clock Speed

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

Table 24. TWI prescaler configuration

<code>TWI_CTRL_PRSC[2:0]</code>	0b000	0b001	0b010	0b011	0b100	0b101	0b110	0b111
Resulting <code>clock_prescaler</code>	2	4	8	64	128	1024	2048	4096

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

$$f_{SCL} = f_{main}[Hz] / (4 * \text{clock_prescaler} * (1 + \text{TWI_CTRL_CDIV}))$$

Hence, the maximum TWI clock is $f_{main} / 8$ and the lowest TWI clock is $f_{main} / 262144$. The generated TWI clock is always symmetric having a duty cycle of exactly 50% (if the clock is not halted by a device during clock stretching).



Clock Stretching

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

TWI Transfers

All TWI operations are controlled by the `DCMD` register, which is buffered by a FIFO. This command/data FIFO is internally split into a TX FIFO and a RX FIFO. Writing to `DCMD` will write to the TX FIFO while reading from `DCMD` will read from the RX FIFO. Thus, complete TWI sequences can be programmed and executed without further CPU intervention. The actual operation is selected by a 2-bit value that is written to the register's `TWI_DCMD_CMD_*` bit-field:

- `00`: NOP (no-operation); all further bit-fields in `DCMD` are ignored; no bus operation is executed

- **01**: Generate a (repeated) START condition; all further bit-fields in **DCMD** are ignored
- **10**: Generate a STOP condition; all further bit-fields in **DCMD** are ignored
- **11**: Trigger a data transmission; see below

For any transmission the data to be send has to be written to the register's **TWI_DCMD_MSB** : **TWI_DCMD_LSB** bit-field. If **TWI_DCMD_ACK** is set the controller will generate an ACK by it's own. If **TWI_DCMD_ACK** is cleared the controller will sample ACK/NACK from the accessed device. Vice versa, the device's data response can be read from **TWI_DCMD_MSB** : **TWI_DCMD_LSB** bit-field. The ACK/NACK generated by the device can be read from the **TWI_DCMD_ACK** bit.

The control register's busy flag **TWI_CTRL_BUSY** is set as long as the TX FIFO contains data (i.e. programmed TWI operations that have not been executed yet) or if the TWI bus engine is still processing an operation. An active transmission can be terminated at any time by disabling the TWI module. This will also clear the data/command FIFO.

Tristate Drivers

The TWI module requires two tristate drivers (actually: open-drain drivers - signals can only be actively driven low) for the SDA and SCL lines, which have to be implemented by the user in the setup's top module / IO ring. A generic VHDL example is shown below (here, **sda_io** and **scl_io** are the actual I²C bus lines, which are of type **std_logic**).

Listing 8. TWI VHDL Tristate Driver Example

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

TWI Interrupt

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

Register Map

*Table 25. TWI register map (struct **NEORV32_TWI**)*

Address	Name [C]	Bit(s), Name [C]	R/W	Function
0xfff90000	CTRL	0 TWI_CTRL_EN	r/w	TWI enable, reset if cleared
		3:1 TWI_CTRL_PRSC2 : TWI_CTRL_PRSC0	r/w	3-bit clock prescaler select
		7:4 TWI_CTRL_CDIV3 : TWI_CTRL_CDIV0	r/w	4-bit clock divider
		8 TWI_CTRL_CLKSTR	r/w	Enable (allow) clock stretching
		14:9 -	r/-	<i>reserved</i> , read as zero
		18:15 TWI_CTRL_FIFO_MSB : TWI_CTRL_FIFO_LSB	r/-	FIFO depth; log2(IO_TWI_FIFO)
		26:12 -	r/-	<i>reserved</i> , read as zero
		27 TWI_CTRL_SENSE_SCL	r/-	current state of the SCL bus line
		28 TWI_CTRL_SENSE_SDA	r/-	current state of the SDA bus line
		29 TWI_CTRL_TX_FULL	r/-	set if the TWI bus is claimed by any controller
		30 TWI_CTRL_RX_AVAIL	r/-	RX FIFO data available
		31 TWI_CTRL_BUSY	r/-	TWI bus engine busy or TX FIFO not empty
0xfff90004	DCMD	7:0 TWI_DCMD_MSB : TWI_DCMD_LSB	r/w	write: TX data byte; read: RX data byte
		8 TWI_DCMD_ACK	r/w	write: ACK issued by controller; read: 1 = device NACK, 0 = device ACK
		10:9 TWI_DCMD_CMD_HI : TWI_DCMD_CMD_LO	-/w	TWI operation (00 = NOP, 01 = START conditions, 10 = STOP condition, 11 = data transmission)
		31:11 -	r/-	<i>reserved</i> , read as zero

2.8.18. Two-Wire Serial Device Controller (TWD)

Hardware source files:	neorv32_twd.vhd	
Software driver files:	neorv32_twd.c	Online software reference (Doxygen)
	neorv32_twd.h	Online software reference (Doxygen)
Top entity ports:	<code>twd_sda_i</code>	1-bit serial data line sense input
	<code>twd_sda_o</code>	1-bit serial data line output (pull low only)
	<code>twd_scl_i</code>	1-bit serial clock line sense input
Configuration generics:	<code>IO_TWD_EN</code>	implement TWD controller when <code>true</code>
	<code>IO_TWD_RX_FIFO</code>	RX FIFO depth, has to be a power of two, min 1
	<code>IO_TWD_TX_FIFO</code>	TX FIFO depth, has to be a power of two, min 1
CPU interrupts:	fast IRQ channel 4	FIFO status interrupt (see Processor Interrupts)

Key Features

- I²C-compatible device-side controller
- Programmable 7-bit device address
- Optional RX/TX data FIFOs
- Programmable interrupt conditions

Overview

The NEORV32 TWD implements an I2C-compatible **device-side** interface. Processor-external I2C hosts can communicate with this module by issuing I2C transactions. The TWD is entirely passive and only reacts to external transmissions.



Device-Mode Only

The NEORV32 TWD controller only supports **device mode**. Transmission are initiated by processor-external modules and not by an external TWD. If you are looking for a *host-mode* module (transactions initiated by the processor) check out the [Two-Wire Serial Interface Controller \(TWI\)](#).

Theory of Operation

The TWD module provides two memory-mapped registers that are used for configuration & status check (**CTRL**) and for accessing transmission data (**DATA**). The **DATA** register is transparently buffered by separate RX and TX FIFOs. The size of those FIFOs can be configured by the `IO_TWD_RX_FIFO` and `IO_TWD_TX_FIFO` generics. Software can determine the FIFO size via the control register's

`TWD_CTRL_FIFO_*` bits. The current status of the RX and TX FIFO can be polled by software via the `TWD_CTRL_RX_*` and `TWD_CTRL_TX_*` flags.

The module is globally enabled by setting the control register's `TWD_CTRL_EN` bit. Clearing this bit will disable and reset the entire module also clearing the internal RX and TX FIFOs. Each FIFO can also be cleared individually at any time by setting `TWD_CTRL_CLR_RX` or `TWD_CTRL_CLR_TX` control register bits, respectively.

The external I²C bus is sampled and synchronized into the processor's clock domain with a sampling frequency of 1/8 of the processor's main clock. In order to increase the resistance to glitches the sampling frequency can be lowered to 1/64 of the processor clock by setting the control register's `TWD_CTRL_FSEL` bit.

The TWD only responds to a configurable 7-bit device address that is programmed by the `TWD_CTRL_DEV_ADDR` bits. Specific general-call or broadcast addresses are not supported yet.

Depending on the transaction type, data is either read from the TX FIFO and transferred back to the host (**read operation**) or data is received from the host and written to the RX FIFO (**write operation**).

Read Operation: Host Reads Data from TWD Module

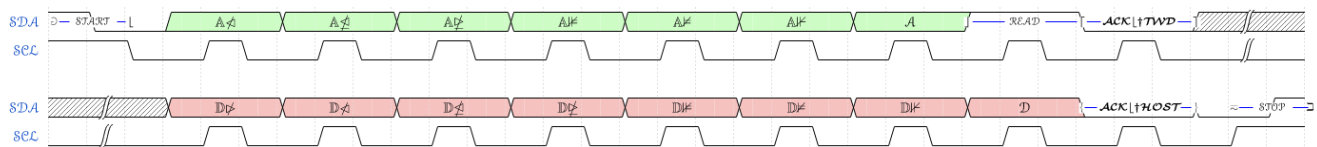


Figure 9. Exemplary single-byte TWD read operation (split across two lines)

For a read operation the host first generates START or REPEATED-START condition. Then, the host transmits the 7 bit device address (green signals `A6` to `A0`) plus the read-command bit. If the transferred address matches the one programmed to `TWD_CTRL_DEV_ADDR` control register bits the TWD module will response with an ACK by pulling the SDA bus line actively low during the 9th SCL clock pulse. If there is no address match the TWD will not interfere with the bus and will wait for the next START or REPEATED-START condition.

For the actual data transmission the host keeps the SDA line at high state while sending the clock pulses. The TWD will read a byte from the internal TX FIFO and will transmit it MSB-first to the host. During the 9th clock pulse the host has to acknowledged the transfer (ACK) by pulling SDA low. If no ACK is received by the TWD no data is taken from the TX FIFO and the same byte is transmitted in the next data phase. If the TX FIFO becomes empty while the host keeps reading data, all-one bytes are sent back to the host. The transaction is terminated by a STOP condition.

Write Operation: Host Writes Data to TWD Module

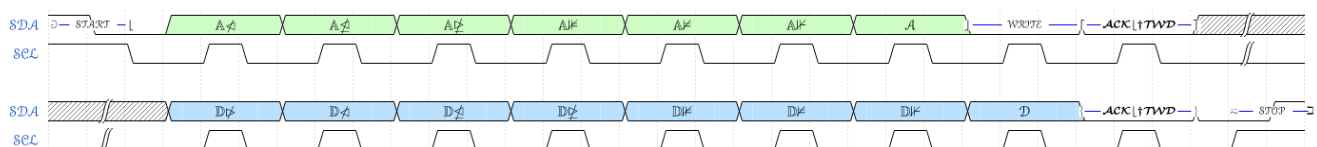


Figure 10. Exemplary single-byte TWD write operation timing (split across two lines)

For a write operation the host first generates START or REPEATED-START condition. Then, the host transmits the 7 bit device address (green signals **A6** to **A0**) plus the write-command bit. If the transferred address matches the one programmed to **TWD_CTRL_DEV_ADDR** control register bits the TWD module will response with an ACK by pulling the SDA bus line actively low during the 9th SCL clock pulse. If there is no address match the TWD will not interfere with the bus and will wait for the next START or REPEATED-START condition.

For the actual data transmission the host sends the write-data MSB first together with the clock pulses. During the 9th clock pulse the TWD will respond with an ACK by pulling SDA low. Note that the TWD will respond with a NACK if the RX FIFO is full. The transaction is terminated by a STOP condition.

Communication Status

Actual communication with the TWD is (re-)**started** when a START or REPEATED-START condition followed by the TWD device address is received. This communication is **terminated** as soon as a STOP condition is received. The current state of the communication can be polled via the **TWD_CTRL_COM** control register flag.

Whenever a communication start has been observed the **TWD_CTRL_COM_BEG** control register flag becomes set. Accordingly, the **TWD_CTRL_COM_END** control register flag becomes set whenever a communication end / termination has been observed. These flags retain their value until they are either manually deleted (by writing a **1** to the respective flag) or the TWD module is deactivated and thus reset.

TWD Interrupt

The TWD module provides a single interrupt to signal certain FIFO conditions and/or communication states to the CPU. The control register's **TWD_CTRL_IRQ_*** bits are used to enabled individual interrupt conditions. Note that all enabled conditions are logically OR-ed.

Table 26. TWD Interrupts Sources

Interrupt Enable	Description	Interrupt Condition
TWD_CTRL_IRQ_RX_AVAIL	Interrupt if at least one data byte is available in the RX FIFO.	TWD_CTRL_RX_AVAIL == 1
TWD_CTRL_IRQ_RX_FULL	Interrupt if the RX FIFO is full.	TWD_CTRL_RX_FULL == 1
TWD_CTRL_IRQ_TX_EMPTY	Interrupt if the TX FIFO is empty.	TWD_CTRL_TX_EMPTY == 1
TWD_CTRL_IRQ_TX_NFULL	Interrupt if the TX FIFO is not full yet.	TWD_CTRL_TX_FULL == 0
TWD_CTRL_IRQ_COM_BEG	Interrupt if a communication-start has been observed.	TWD_CTRL_COM_BEG == 1
TWD_CTRL_IRQ_COM_END	Interrupt if a communication-end has been observed.	TWD_CTRL_COM_END == 1

The CPU interrupt request remains active until all enabled interrupt-causing conditions are resolved. The interrupt can only trigger if the module is actually enabled (**TWD_CTRL_EN** is set).

Tristate Driver

The TWD module requires a single tristate drives (actually: open-drain drivers - signals can only be actively driven low) for the SDA line, which have to be implemented by the user in the setup's top module / IO ring. The SCL signal is only read by the TWD module (input only) and therefore does not require a tristate driver. A generic VHDL example is shown below.

Listing 9. TWD Tristate Driver Example (VHDL)

```
-- IO ports / bus lines --
sda_io : inout std_logic; -- bidirectional
scl_io : in    std_logic; -- input only

-- tristate driver --
sda_io <= '0' when (twd_sda_o = '0') else 'Z'; -- drive (pull low)

-- sense --
twd_sda_i <= std_ulogic(sda_io);
twd_scl_i <= std_ulogic(scl_io);
```

Register Map

Table 27. TWD register map (`struct NEORV32_TWD`)

Address	Name [C]	Bit(s), Name [C]	R/W	Function
0xffea0000	CTRL	0 TWD_CTRL_EN	r/w	TWD enable, reset if cleared
		1 TWD_CTRL_CLR_RX	-/w	Clear RX FIFO, flag auto-clears
		2 TWD_CTRL_CLR_TX	-/w	Clear TX FIFO, flag auto-clears
		3 TWD_CTRL_FSEL	r/w	Bus sample clock / filter select
		10:4 TWD_CTRL_DEV_ADDR6 : TWD_CTRL_DEV_ADDR0	r/w	Device address (7-bit)
		11 TWD_CTRL_IRQ_RX_AVAIL	r/w	IRQ if RX FIFO data available
		12 TWD_CTRL_IRQ_RX_FULL	r/w	IRQ if RX FIFO full
		13 TWD_CTRL_IRQ_TX_EMPTY	r/w	IRQ if TX FIFO empty
		14 TWD_CTRL_IRQ_TX_NFULL	r/w	IRQ if TX FIFO is not full
		15 TWD_CTRL_IRQ_COM_BEG	r/w	IRQ if begin of communication
		16 TWD_CTRL_IRQ_COM_END	r/w	IRQ if end of communication
		20:17 TWD_CTRL_RX_FIFO_MSB : TWD_CTRL_RX_FIFO_LSB	r/-	FIFO depth; log2(IO_TWD_RX_FIFO)
		24:21 TWD_CTRL_TX_FIFO_MSB : TWD_CTRL_TX_FIFO_LSB	r/-	FIFO depth; log2(IO_TWD_TX_FIFO)
		25 TWD_CTRL_RX_AVAIL	r/-	RX FIFO data available
		26 TWD_CTRL_RX_FULL	r/-	RX FIFO full
		27 TWD_CTRL_TX_EMPTY	r/-	TX FIFO empty
		28 TWD_CTRL_TX_FULL	r/-	TX FIFO full
		29 TWD_CTRL_COM_BEG	r/c	Communication has started; clear by writing 1
		30 TWD_CTRL_COM_END	r/c	Communication has ended; clear by writing 1
		31 TWD_CTRL_COM	r/-	Active communication
0xffea0004	DATA	7:0 TWD_DATA_MSB : TWD_DATA_LSB	r/w	RX/TX data FIFO access
		31:8 -	r/-	<i>reserved</i> , read as zero

2.8.19. One-Wire Serial Interface Controller (ONEWIRE)

Hardware source files:	neorv32_owewire.vhd	
Software driver files:	neorv32_owewire.c	Online software reference (Doxygen)
	neorv32_owewire.h	Online software reference (Doxygen)
Top entity ports:	owewire_i	1-bit 1-wire bus sense input
	owewire_o	1-bit 1-wire bus output (pull low only)
Configuration generics:	IO_ONEWIRE_EN	implement ONEWIRE interface controller when true
	IO_ONEWIRE_FIFO	RTX FIFO depth, has to be zero or a power of two, min 1
CPU interrupts:	fast IRQ channel 13	operation done interrupt (see Processor Interrupts)

Key Features

- Dallas® 1-Wire™-compatible host controller
- Hardware-based protocol handling (RESET, presence, data, etc.)
- Fine-grained programmable bus timings
- Optional data/command FIFO
- Interrupt based on FIFO status

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 (plus ground) for communication.

The bus is based on a single open-drain signal. The controller as well as all devices on the bus can only pull-down the bus (similar to TWI/I2C). The default high-level is provided by a single pull-up resistor connected to the positive power supply close to the bus controller. Recommended values are between 1kΩ and 10kΩ depending on the bus characteristics (wire length, number of devices, etc.).

Tri-State Drivers

The ONEWIRE module requires a tristate driver (actually, just an open-drain driver) for the 1-wire bus line, which has to be implemented in the top module / IO ring of the design. A generic VHDL example is given below (**owewire_io** is the actual 1-wire bus signal, which is of type **std_logic**; **owewire_o** and **owewire_i** are the processor's ONEWIRE port signals).

Listing 10. ONEWIRE VHDL Tristate Driver Example

```
owewire_io <= '0' when (owewire_o = '0') else 'Z'; -- drive (low)
```



```
onewire_i <= std_ulogic(onewire_io); -- sense
```

Theory of Operation

The ONEWIRE controller provides two interface registers: **CTRL** and **DCMD**. The control register (**CTRL**) is used to configure the module and to monitor the current state. The **DCMD** register, which can optionally be buffered by a configurable FIFO (**IO_ONEWIRE_FIFO** generic), is used to read/write data from/to the bus and to trigger bus operations.

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, any bus operation is aborted, the bus is brought to high-level (due to the external pull-up resistor) and the internal FIFO is cleared. The basic timing configuration is programmed via a coarse clock prescaler (**ONEWIRE_CTRL_PRSCx** bits) and a fine clock divider (**ONEWIRE_CTRL_CLKDIVx** bits).

The controller can execute four basic bus operations, which are triggered by writing the according command bits in the **DCMD** register (**ONEWIRE_DCMD_DATA_*** bits) while also writing the actual data bits (**ONEWIRE_DCMD_CMD_*** bits).

1. **0b00** (**ONEWIRE_CMD_NOP**) - no operation (dummy)
2. **0b01** (**ONEWIRE_CMD_BIT**) - transfer a single-bit (read-while-write)
3. **0b10** (**ONEWIRE_CMD_BYTE**) - transfer a full-byte (read-while-write)
4. **0b11** (**ONEWIRE_CMD_RESET**) - generate reset pulse and check for device presence

Every command (except NOP) will result in a bus operation when dispatched from the data/command FIFO. Each command (except NOP) will also sample a bus response (a read bit, a read byte or a presence pulse) to a shadowed receive FIFO that is accessed when reading the **DCMD** register.

When the single-bit operation (**ONEWIRE_CMD_BIT**) is executed, the data previously written to **DCMD[0]** will be sent to the bus and the response is sampled to **DCMD[7]**. Accordingly, a full-byte transmission (**ONEWIRE_CMD_BYTE**) will send the byte written to **DCMD[7:0]** to the bus and will sample the response to **DCMD[7:0]** (LSB-first). Finally, the reset command (**ONEWIRE_CMD_RESET**) will generate a bus reset and will also sample the "presence pulse" from the device(s) to the **DCMD[ONEWIRE_DCMD_PRESENCE]**.



Read from Bus

In order to read a single bit from the bus **DCMD[0]** has to be set to **1** before triggering the bit transmission operation to allow the accessed device to pull-down the bus. Accordingly, **DCMD[7:0]** has to be set to **0xFF** before triggering the byte transmission operation when the controller shall read a byte from the bus.

As soon as the current bus operation has completed (and there are no further operations pending in the FIFO) the **ONEWIRE_CTRL_BUSY** bit in the control registers clears.

Bus Timing

The control register provides a 2-bit clock prescaler select (`ONEWIRE_CTRL_PRSC`) and a 8-bit clock divider (`ONEWIRE_CTRL_CLKDIV`) 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 28. ONEWIRE Clock Prescaler Configurations

<code>ONEWIRE_CTRL_PRSC[2:0]</code>	0b00	0b01	0b10	0b11
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

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

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 T_{base} . The following images show the two basic operations of the ONEWIRE controller: single-bit (0 or 1) transaction and reset with presence detect. Note that the full-byte operations just repeats the single-bit operation eight times. The relevant points in time are shown as *absolute* time points (in multiples of the time base T_{base}) with the falling edge of the bus as reference points.

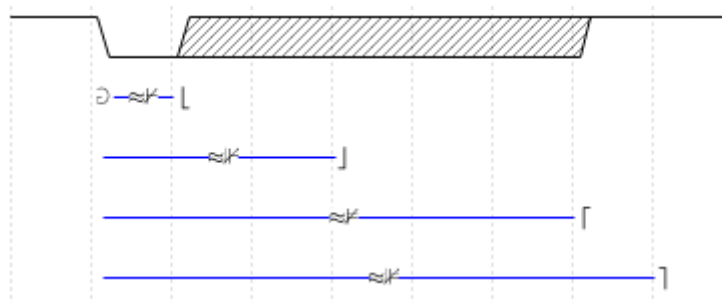


Figure 11. Single-bit data transmission (not to scale)

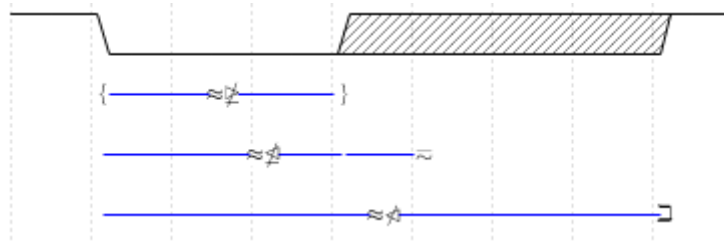


Figure 12. Reset pulse and presence detect (not to scale)

Table 29. Data Transmission Timing

Symbol	Description	Multiples of T_{base}	Time when $T_{base} = 10\mu s$
Single-bit data transmission			
t0 (a → b)	Time until end of active low-phase when writing a '1' or when reading	1	10 μs
t1 (a → c)	Time until controller samples bus state (read operation)	2	20 μs
t2 (a → d)	Time until end of bit time slot (when writing a '0' or when reading)	7	70 μs
t3 (a → e)	Time until end of inter-slot pause (= total duration of one bit)	9	90 μs
Reset pulse and presence detect			
t4 (f → g)	Time until end of active reset pulse	48	480 μs
t5 (f → h)	Time until controller samples bus presence	55	550 μs
t6 (f → i)	Time until end of presence phase	96	960 μs

*Default Timing Parameters*

The "known-good" default values for base time multiples were chosen for stable and reliable bus operation and 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 (if necessary) by editing the ONEWIRE's VHDL source file. The times t0 to t6 correspond to the previous timing diagrams.

Listing 11. Hardwired timing configuration in `neorv32_owewire.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 Mode

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

Interrupt

A single interrupt is provided by the ONEWIRE module to signal "idle" condition to the CPU. Whenever the controller is idle (again) and the data/command FIFO is empty, the interrupt becomes active.

Register Map

Table 30. ONEWIRE register map (`struct NEORV32_ONEWIRE`)

Address	Name [C]	Bit(s), Name [C]	R/W	Function
0xfff20000	CTRL	0 ONEWIRE_CTRL_EN	r/w	ONEWIRE enable, reset if cleared
		1 ONEWIRE_CTRL_CLEAR	-/w	clear RXT FIFO, auto-clears
		3:2 ONEWIRE_CTRL_PRSC1 : ONEWIRE_CTRL_PRSC0	r/w	2-bit clock prescaler select
		11:4 ONEWIRE_CTRL_CLKDIV7 : ONEWIRE_CTRL_CLKDIV0	r/w	8-bit clock divider value
		14:12 -	r/-	<i>reserved</i> , read as zero
		18:15 ONEWIRE_CTRL_FIFO_MSB : ONEWIRE_CTRL_FIFO_LSB	r/-	FIFO depth; $\log_2(\text{IO_ONEWIRE_FIFO})$
		27:19 -	r/-	<i>reserved</i> , read as zero
		28 ONEWIRE_CTRL_TX_FULL	r/-	TX FIFO full
		29 ONEWIRE_CTRL_RX_AVAIL	r/-	RX FIFO data available
		30 ONEWIRE_CTRL_SENSE	r/-	current state of the bus line
		31 ONEWIRE_CTRL_BUSY	r/-	operation in progress when set or TX FIFO not empty
0xfff20004	DCMD	7:0 ONEWIRE_DCMD_DATA_MSB : ONEWIRE_DCMD_DATA_LSB	r/w	receive/transmit data
		9:8 ONEWIRE_DCMD_CMD_HI : ONEWIRE_DCMD_CMD_LO	-/w	operation command LSBs
		10 ONEWIRE_DCMD_PRESENCE	r/-	bus presence detected
		31:11 -	r/-	<i>reserved</i> , read as zero

2.8.20. Pulse-Width Modulation Controller (PWM)

Hardware source files:	neorv32_pwm.vhd	
Software driver files:	neorv32_pwm.c	Online software reference (Doxygen)
	neorv32_pwm.h	Online software reference (Doxygen)
Top entity ports:	<code>pwm_o</code>	PWM output channels (32-bit)
Configuration generics:	<code>IO_PWM_NUM</code>	number of PWM channels to implement (0..32)
CPU interrupts:	none	

Key Features

- Up to 32 individual channels with up to 16-bit resolution and programmable polarity
- Fast-PWM or phase-correct operation mode with counter-compare and counter-wrap registers
- Toggle-free 0% and 100% duty cycle output rates
- Global clock prescaler

Overview

The PWM module implements a pulse-width modulation controller with up to 32 independent channels. Period length (and thus, the carrier frequency), duty cycle, polarity and operation mode can be programmed individually for each channel. However, the clock for the PWM counter increment is defined by a single global clock prescaler. PWM operation is based on 16-bit wide period counters that are constrained by programmable wrapping values.

The total number of implemented channels is defined by the `IO_PWM_NUM` generic. The PWM output signal `pwm_o` has a static size of 32 bit. Channel 0 corresponds to bit 0, channel 1 to bit 1 and so on. If less than 32 channels are configured, only the LSB-aligned channel bits are connected while the remaining ones are hardwired to zero.

Theory of Operation

The PWM module provides several configuration registers: an `ENABLE` register, a `POLARITY` registers, a `CLKPRSC` register, and a `MODE` register. Each of these register provides one bit (LSB-aligned) for each available PWM channel. Additional, up to `IO_PWM_NUM` channel register `CHANNEL` are provided.

The `ENABLE` register is used to enable individual PWM channels. By using bit-masks several channels can be enabled at once so they operate in perfect lockstep. The PWM counter of a disabled channel is halted and reset to zero; the according PWM output (`pwm_o(i)`) is reset to the configured polarity. The polarity of each channel's output can be inverted by setting the corresponding bit in the `POLARITY` register.

The `CLKPRSC` register configures the *global* clock prescaler that is used for the PWM counter increment of all channels. Eight pre-defined prescaler values are available:

Table 31. PWM prescaler configuration

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

The `MODE` register defines the PWM operation mode of a channel. If bit i is `0` channel i operates in **fast-PWM** mode. If bit i is `1` channel i operates in **phase-correct PWM** mode.

Duty cycle and period length of each channel are defined by the according `CHANNEL` registers. These registers are split in two half-words: the lowest half-word defines the counter-compare value (`CMP`) while the upper half-word defines the counter-wrap value (`TOP`). Regardless of the configured PWM operation mode, the duty cycle of channel i is defined by the following formula:

$$\text{Duty Cycle}[i] = 100\% * (\text{CMP}[i] / (\text{TOP}[i] + 1))$$

The NEORV32 PWM module supports toggle-free 0% and 100% duty cycle output rates. 0% duty cycle is achieved by setting `CMP = 0`. 100% duty cycle is achieved by setting `CMP = TOP + 1`.



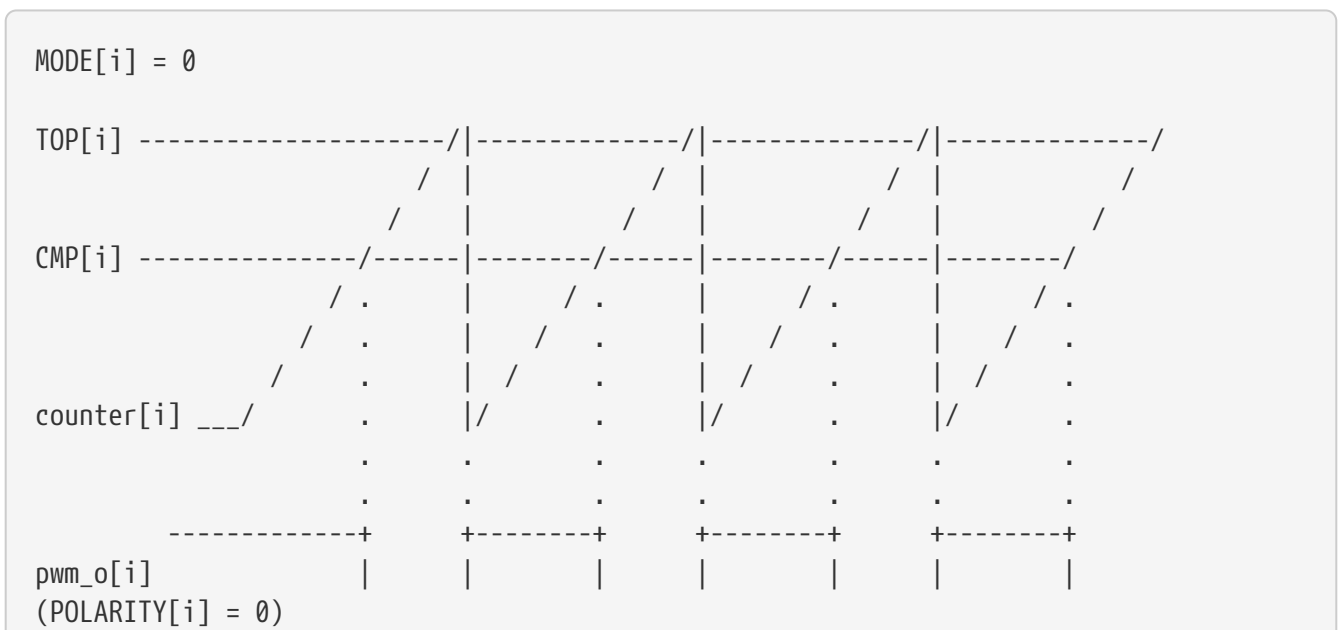
`CHANNEL` Half-Word Access

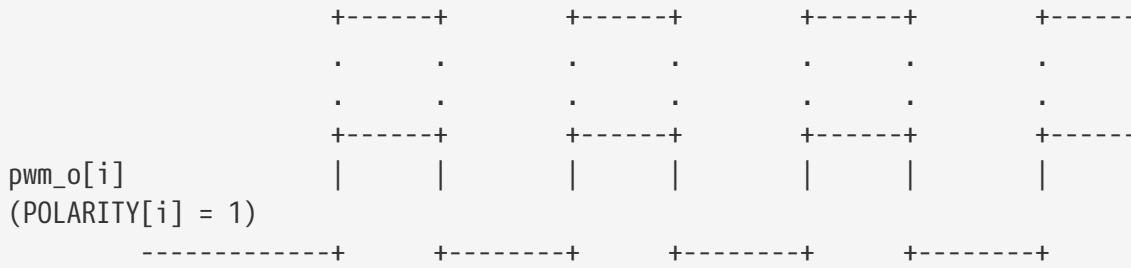
In contrast to many other NEORV32 peripherals, the PWM `CHANNEL[i]` registers can also be accessed using half-word load/store operation. This allows to update `TOP` and `CMP` independent of each other.

Fast-PWM Mode

When in fast-PWM mode (`MODE[i] = 0`) the channel's PWM counter generates a sawtooth-like waveform. The counter automatically wraps to all-zero when reaching the programmed `TOP[i]` value. Whenever the counter value is less equal to or greater than the `CMP[i]` value the PWM output is set to the programmed polarity (`POLARITY[i]`) value; otherwise it is set to the inverse polarity.

Listing 12. Fast-PWM Operation





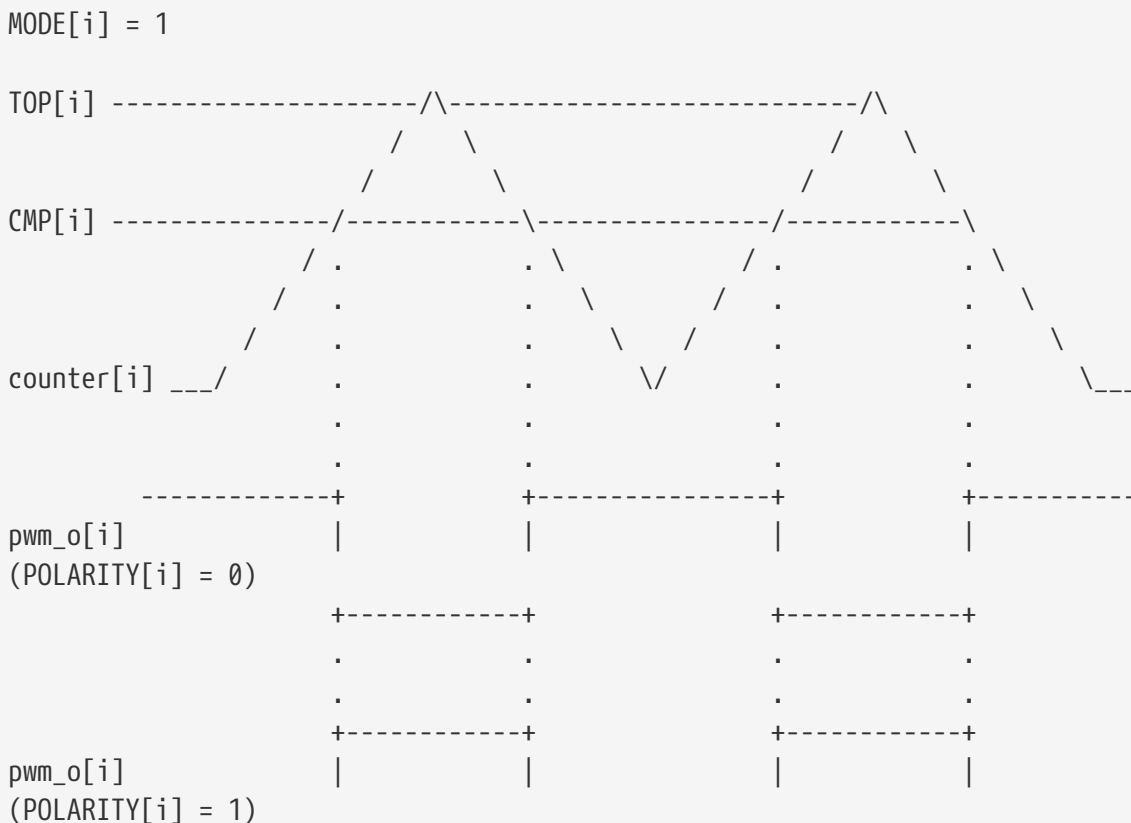
Based on the global clock prescaler and the **TOP** wrap value, the carrier frequency of a channel is defined by the following formula (f_{main} is the main clock frequency of the processor; **clock_prescaler** is the global pre-scaling factor according to the selected **CLKPRSC** value):

$$f_{PWM}[i] = f_{main} / (\text{clock_prescaler} * (\text{TOP}[i] + 1))$$

Phase-Correct PWM Mode

When in phase-correct PWM mode (**MODE[i] = 1**) the channel's PWM counter generates a triangle-like waveform. Phase-correct mode centers the pulse on the same point regardless of the duty cycle. After enabling, the PWM counters counts upward until it reaches the **TOP** value. After that the counter counts downward until it reaches 0 again. Whenever the counter value is equal to or greater than the **CMP[i]** value the PWM output is set to the programmed polarity (**POLARITY[i]**) value; otherwise it is set to the inverse polarity.

Listing 13. Phase-Correct PWM Operation



-----+ +-----+ +-----

Based on the global clock prescaler and the **TOP** wrap value, the carrier frequency of a channel is defined by the following formula (f_{main} is the main clock frequency of the processor; **clock_prescaler** is the global pre-scaling factor according to the selected **CLKPRSC** value):

$$f_{PWM}[i] = f_{main} / (\text{clock_prescaler} * 2 * \text{TOP}[i])$$



PWM Dead Time

The phase-correct PWM mode can also be used for controlling modules that require a certain dead time (e.g. H-bridges). This requires two PWM channels, both of which operate in phase-correct mode, but the second channel is configured with an inverse polarity. Both channels need to have the same **TOP** value and must be enabled at once to have a perfect phase and frequency match. The dead time is define by the difference of the channel's **CMP** values:

$$T_{dead} = 0.5 * T * (\text{CMP}[i] - \text{CMP}[i+1])$$

Register Map

Table 32. PWM register map (**struct** **NEORV32_PWM**)

Address	Name [C]	Bit(s)	R/W	Function
0xfff00000	ENABLE	31:0	r/w	Channel enable, one bit per channel
0xfff00004	POLARITY	31:0	r/w	Channel polarity, one bit per channel
0xfff00008	CLKPRSC	2:0	r/w	Global clock prescaler select
0xfff0000c	MODE	31:0	r/w	Channel operation mode (0 = fast-PWM, 1 = phase-correct PWM), one bit per channel
0xfff00080	CHANNEL[0].TOPCMP	31:0	r/w	Channel 0 full word access alias (TOP and CMP together at 32-bit value)
0xfff00080	CHANNEL[0].CMP	15:0	r/w	Channel 0 top/wrap value register
0xfff00082	CHANNEL[0].TOP	15:0	r/w	Channel 0 compare value register
...	...	...	r/w	...
0xfff000fc	CHANNEL[31].TOPCMP	31:0	r/w	Channel 31 full word access alias (TOP and CMP together at 32-bit value)
0xfff000fc	CHANNEL[31].CMP	15:0	r/w	Channel 31 top/wrap value register
0xfff000fe	CHANNEL[31].TOP	15:0	r/w	Channel 31 compare value register

2.8.21. True Random-Number Generator (TRNG)

Hardware source files:	neorv32_trng.vhd	
Software driver files:	neorv32_trng.c	Online software reference (Doxygen)
	neorv32_trng.h	Online software reference (Doxygen)
Top entity ports:	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
	<code>IO_TRNG_NUM_RO</code>	total number of ring-oscillators
	<code>IO_TRNG_NUM_INV</code>	number of inverters in first ring-oscillator, has to be odd
	<code>IO_TRNG_NUM_RBIT</code>	number of raw bits to process for one output byte, has to be power of two
CPU interrupts:	fast IRQ channel 15	data available interrupt (see Processor Interrupts)

Key Features

- True random number generator based on phase noise
- Completely technology-independent architecture
- Configurable TRNG layout to adjust to technology and quality requirements
- Optional random data FIFO
- Interrupt based on FIFO status

Overview

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



In-Depth Documentation

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



Inferring Latches

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

Theory of Operation

The TRNG provides two memory mapped interface register. One control register (**CTRL**) for configuration and status check and one data register (**DATA**) for obtaining the random data. The TRNG is enabled by setting the control register's **TRNG_CTRL_EN**. As soon as the **TRNG_CTRL_AVAIL** bit is set a new random data byte is available and can be obtained from the lowest 8 bits of the **DATA** register. If this bit is cleared, there is no valid data available and the reading **DATA** will return all-zero.

The actual neoTRNG is configured by three generics:

1. **IO_TRNG_NUM_RO** defines the total number of ring-oscillators.
2. **IO_TRNG_NUM_INV** defines the number of inverters in the first ring-oscillator. Each subsequent RO is two inverters longer than the previous one.
3. **IO_TRNG_NUM_RBIT** defines the number of raw entropy bits obtained from the ring-oscillators that are scrambled and combined to produce a single random output byte.

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 larger number of random data in a short time. The random data FIFO can be cleared at any time either by disabling the TRNG or by setting the **TRNG_CTRL_FIFO_CLR** flag. The FIFO depth can be retrieved by software via the **TRNG_CTRL_FIFO_*** bits.

Simulation



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

Interrupt

The TRNG provides a single interrupt request signal that gets triggered when the TRNG is enabled and the data FIFO is completely full (indicating that at least **IO_TRNG_FIFO** bytes of random data are available).

Register Map

Table 33. TRNG register map (**struct** **NEORV32_TRNG**)

Address	Name [C]	Bit(s), Name [C]	R/W	Function
0xffffa0000	CTRL	0 TRNG_CTRL_EN	r/w	TRNG enable
		1 TRNG_CTRL_FIFO_CLR	-/w	flush random data FIFO when set; auto-clears
		2 TRNG_CTRL_SIM_MODE	r/-	simulation mode (PRNG!)
		3 TRNG_CTRL_AVAIL	r/-	random data available when set
		7:4 TRNG_CTRL_FIFO_MSB : TRNG_CTRL_FIFO_LSB	r/-	FIFO depth ($\log_2(\text{IO_TRNG_FIFO})$)
		11:8 TRNG_CTRL_NBIT_MSB : TRNG_CTRL_NBIT_LSB	r/-	FIFO depth ($\log_2(\text{IO_TRNG_NUM_RBIT})$)
		19:12 TRNG_CTRL_NRO_MSB : TRNG_CTRL_NRO_LSB	r/-	Total number of ring-oscillators (IO_TRNG_NUM_RO)
		31:20 TRNG_CTRL_NINV_MSB : TRNG_CTRL_NINV_LSB	r/-	Number of inverters in first ring-oscillator (IO_TRNG_NUM_INV)
0xffffa0004	DATA	7:0 TRNG_DATA_MSB : TRNG_DATA_LSB	r/-	random data byte
		31:8 <i>reserved</i>	r/-	reserved, read as zero

2.8.22. Custom Functions Subsystem (CFS)

Hardware source files:	neorv32_cfs.vhd	
Software driver files:	neorv32_cfs.c	Online software reference (Doxygen)
	neorv32_cfs.h	Online software reference (Doxygen)
Top entity ports:	<code>cfs_in_i</code>	custom input conduit, 256-bit
	<code>cfs_out_o</code>	custom output conduit, 256-bit
Configuration generics:	<code>IO_CFS_EN</code>	implement CFS when <code>true</code>
CPU interrupts:	fast IRQ channel 1	CFS interrupt (see Processor Interrupts)

Key Features

- Tightly-coupled co-processor unit for custom hardware accelerators
- Can operate independently of the CPU
- Access via memory-mapped registers (64kB)
- Pre-defined 256-bit input and output conduits for custom IO
- One dedicated interrupt channel

Overview

The custom functions subsystem is a template module that allows to implement custom tightly-coupled co-processors or interfaces. It provides up to 64kB of address space for memory-mapped registers (**REG**, see register map below) that can be accessed by the CPU via normal load/store operations. The actual functionality is to be defined by the hardware designer. Furthermore, the CFS provides two input/output conduits to implement custom on-chip or off-chip interfaces.

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 (e.g. AES), signal processing (e.g. FFT), AI applications (e.g. CNNs), custom IO controllers (e.g. SDIO), networking (e.g. CAN), real-time data transport (e.g. I2S) or just replication of existent NEORV32 peripherals (e.g. a second SPI controller).



Custom ISA Instructions

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



CFS Template

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.

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 14. 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 Custom IOs

The CFS provides two unidirectional input and output conduits (`cfs_in_i` and `cfs_out_o`). Both signals are 256 bit wide and are directly propagated to the processor's top entity. These conduits can be used to implement application-specific interfaces like memory or peripheral connections.

CFS Interrupt

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

Register Map

Table 34. CFS register map (`struct NEORV32_CFS`)

Address	Name [C]	Bit(s)	R/W	Function
<code>0xffeb0000</code>	<code>REG[0]</code>	<code>31:0</code>	(r)/(w)	custom CFS register 0
<code>0xffeb0004</code>	<code>REG[1]</code>	<code>31:0</code>	(r)/(w)	custom CFS register 1
...	...	<code>31:0</code>	(r)/(w)	...
<code>0xffebfff8</code>	<code>REG[16382]</code>	<code>31:0</code>	(r)/(w)	custom CFS register 16382
<code>0xffebfffc</code>	<code>REG[16383]</code>	<code>31:0</code>	(r)/(w)	custom CFS register 16383

2.8.23. Smart LED Interface (NEOLED)

Hardware source files:	neorv32_neoled.vhd	
Software driver files:	neorv32_neoled.c	Online software reference (Doxygen)
	neorv32_neoled.h	Online software reference (Doxygen)
Top entity ports:	neoled_o	1-bit serial data output
Configuration generics:	IO_NEOLED_EN	implement NEOLED controller when true
	IO_NEOLED_TX_FIFO	transmission FIFO depth, has to be a power of 2, min 1
CPU interrupts:	fast IRQ channel 9	configurable NEOLED data FIFO interrupt (see Processor Interrupts)

Key Features

- "NeoPixel™"-compatible LED controller
- Hardware-based protocol handling
- Fine-grained programmable bus timings
- Optional data/command FIFO
- Interrupt based on FIFO status

Overview

The NEOLED module provides a dedicated interface for "smart LEDs" like WS2812, WS2811 and other compatible ones. These LEDs provide a single-wire interface that uses an asynchronous serial protocol for transmitting data. The NEOLED module handles the tight timing constraints of the interface entirely in hardware. A configurable data and command buffer (FIFO) allows to utilize (e.g. DMA-backed) block transfer operation without CPU intervention.

LEDs are connected to the **neoled_o** output and can be arbitrarily chained. The module also supports 24-bit and 32-bit LED data. Hence, mixed setups with RGB LEDs (24-bit color) and RGBW LEDs (32-bit color including a dedicated white LED chip) are fully supported.

The NEOLED modules provides four accessible interface registers: the control register **CTRL** and the write-only **DATA24**, **DATA32** and **STROBE** registers. The precise interface timing is configured via the control registers. The data and strobe registers are used for sending 24-bit/32-bit data and reset/strobe commands.

Protocol

The interface of the smart LEDs is based on duty-cycle modulation of a specific carrier signal. Data is transmitted in a serial manner starting with the MSB. The intensity for each R/G/B (/W) LED chip is defined via an 8-bit value. The actual data bits are transferred by modifying the duty cycle of the base carriers. A RESET command is send by pulling the data line LOW for at least 50µs.

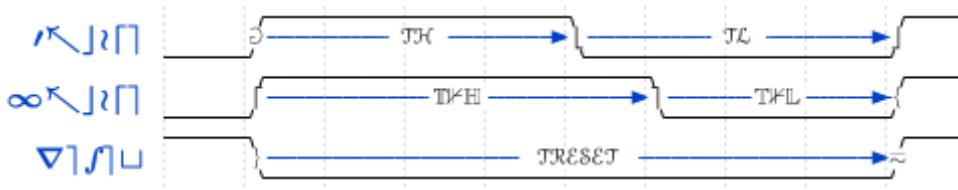


Figure 13. WS2812 bit-level timing (timing does not scale)

Table 35. Example: WS2812 Protocol Timings

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

Timing Configuration

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

Table 36. NEOLED Prescaler Configuration

NEOLED_CTRL_PRSC[2:0]	0b000	0b001	0b010	0b011	0b100	0b101	0b110	0b111
Resulting clock_prescaler	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_1H*** and **NEOLED_CTRL_T_0H*** values, respectively. These programmable timing constants allow to adapt the interface for a wide variety of smart LED protocols.

Timing Configuration - Example (WS2812)

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

- processor clock $f_{\text{main}} = 100 \text{ MHz}$
- NEOLED_CTRL_PRSCx** = **0b001** = $f_{\text{main}} / 4$

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

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

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_0H** = **0b01010** (= decimal 10)
- **NEOLED_CTRL_T_1H** = **0b10100** (= decimal 20)

$$T_{carrier} = T_{TX} * \text{NEOLED_CTRL_T_TOT} = 40\text{ns} * 30 = 1.4\mu\text{s}$$

$$T_{0H} = T_{TX} * \text{NEOLED_CTRL_T_0H} = 40\text{ns} * 10 = 0.4\mu\text{s}$$

$$T_{1H} = T_{TX} * \text{NEOLED_CTRL_T_1H} = 40\text{ns} * 20 = 0.8\mu\text{s}$$



The NEOLED SW driver library (**neorv32_neoled.h**) provides a simplified configuration function that configures all timing parameters for driving WS2812 LEDs based on the processor clock frequency.

NEOLED Data/Command (FIFO)

The interface features a configurable TX data and command buffer (a FIFO) to allow more CPU-independent operation. The buffer depth is configured via the **IO_NEOLED_TX_FIFO** top generic. The FIFO size configuration can be read by software from the **NEOLED_CTRL_FIFO_x** control register bits. If the module is disabled by clearing **NEOLED_CTRL_EN** the entire data and command buffer is cleared.

LED data and commands are written to the **DATA24**, **DATA32** and **STROBE** registers, which are transparently buffered by the optional FIFO. Writing data to **DATA24** will finally transmit the 24 LSB-aligned bits from the written data word to the LED string. Writing data to **DATA32** will finally transmit all 32-bit bits from the written data word to the LED string. Writing any data to **STROBE**, which is also buffered by the FIFO, will send a STROBE (RESET) command to the LED string.

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

NEOLED Interrupt

The NEOLED modules features a single interrupt that gets triggered when the TX data/command FIFO is empty.

Register Map

Table 37. NEOLED register map (**struct** **NEORV32_NEOLED**)

Address	Name [C]	Bit(s), Name [C]	R/W	Function
0xffffd0000	CTRL	0 NEOLED_CTRL_EN	r/w	module enable
		3:1 NEOLED_CTRL_PRSC_MSB : NEOLED_CTRL_PRSC_LSB	r/w	clock prescaler select
		8:4 NEOLED_CTRL_T_TOT_MSB : NEOLED_CTRL_T_TOT_LSB	r/w	pre-scaled clock ticks per total single-bit period (T_{total})
		13:9 NEOLED_CTRL_T_0H_MSB : NEOLED_CTRL_T_0H_LSB	r/w	pre-scaled clock ticks per high- time for sending a zero-bit (T_{0H})
		18:14 NEOLED_CTRL_T_1H_MSB : NEOLED_CTRL_T_1H_LSB	r/w	pre-scaled clock ticks per high- time for sending a one-bit (T_{1H})
		24:19 -	r/-	<i>reserved</i> , read as zero
		28:25 NEOLED_CTRL_FIFO_MSB : NEOLED_CTRL_FIFO_LSB	r/w	log(FIFO size)
		29 NEOLED_CTRL_TX_EMPTY	r/-	TX FIFO is empty
		30 NEOLED_CTRL_TX_FULL	r/-	TX FIFO is full
		31 NEOLED_CTRL_TX_BUSY	r/-	TX serial engine is busy when set
0xffffd0004	DATA2 4	23:0	-/w	write 24-bit RGB data to transmission buffer
0xffffd0008	DATA3 2	31:0	-/w	write 32-bit RGB data to transmission buffer
0xffffd000c	STROBE	-	-/w	write strobe/reset command to transmission buffer

2.8.24. General Purpose Timer (GPTMR)

Hardware source files:	neorv32_gptmr.vhd	
Software driver files:	neorv32_gptmr.c	Online software reference (Doxygen)
	neorv32_gptmr.h	Online software reference (Doxygen)
Top entity ports:	none	
Configuration generics:	IO_GPTMR_NUM	number of individual GPTMR timer slices (0..16)
CPU interrupts:	fast IRQ channel 12	timer interrupt (see Processor Interrupts)

Key Features

- Up to 16 individual timer slices with 32-bit-wide timer and timer-match registers
- Per-slice enable, mode and interrupt control/status
- Single-shot or interval operation mode
- Global clock prescaler

Overview

The general purpose timer module provides up to 16 individual timers organized as slices (the number of implemented slices is configured by **IO_GPTMR_NUM**). Each slice can operate as single-shot timer or as continuous/interval timer and provides a 32-bit counter (**SLICE[i].CNT**) and a 32-bit threshold registers (**SLICE[i].THR**). Operation is controlled by two 32-bit control and status registers (**CSR0** and **CSR1**). An interrupt is triggered when a slice counter matches the according threshold value.



CSR Access

CSR0 and **CSR1** can be accessed in full-word mode (**CSR[0/1].WORD**) and also in half-word mode (e.g. **CSR0.ENABLE**) to update configuration sub-words with a single memory operation.

Theory of Operation

A slice i is enabled when the according bit of **CSR0.ENABLE** is set. Only the lowest **IO_GPTMR_NUM** bits are available. When enabled, the slice's counter register **SLICE[i].CNT** starts incrementing at a prescaled clock frequency. This prescaler is globally configured (= for all slices) by the **CSR1.PRSC** register:

Table 38. GPTMR prescaler configuration

CSR1.PRSC	0b000	0b001	0b010	0b011	0b100	0b101	0b110	0b111
Resulting prescaler	2	4	8	64	128	1024	2048	4096

When `SLICE[i].CNT` matches the threshold value in `SLICE[i].THR` the according interrupt-pending bit in `CSR1.IRQ` becomes set. Disabled slices (`CSR0.ENABLE(i) = 0`) cannot generate any new interrupt requests. However, pending interrupt in `CSR1.IRQ` will remain pending even if the according slice is disabled via. Software can clear pending interrupts by writing zero to the according `CSR1.IRQ` bits. A CPU interrupt request is generated if any slice interrupt i is pending and slice i is enabled (`CSR0.ENABLE(i) = 1` and `CSR1.IRQ(i) = 1`).

The time until a slice interrupt is triggered ($T[i]$) depends on the processor clock (f_{main}) global clock prescaler (selected by `CSR1.PRSC`) and the slice threshold value (`SLICE[i].THR`):

$$T[i] = f_{main} / (\text{prescaler} * \text{SLICE}[i].\text{THR})$$

Each slice can operate as single-shot timer or as continuous interval timer: If `CSR0.MODE(i) = 0` the according slice operates in **single-shot mode**: if the counter matches the programmed threshold (`SLICE[i].CNT == SLICE[i].THR`) the slice interrupt becomes pending and the counter stops at the threshold value. If `CSR0.MODE(i) = 1` the according slice operates in **continuous mode**: if the counter matches the programmed threshold (`SLICE[i].CNT == SLICE[i].THR`) the slice interrupt becomes pending and the counter is reset to zero and starts counting again up to the threshold value.

Register Map

Table 39. GPTMR register map (`struct NEORV32_GPTMR`)

Address	Name [C]	Bit(s), Name [C]	R/W	Description
0xffff10000	CSR0.WORD	31:0	r/w	Control and status register 0 full-word access.
0xffff10000	CSR0.ENABLE	15:0	r/w	Control and status register 0 - Slice enable (one bit per implemented slice starting at LSB).
0xffff10002	CSR0.MODE	15:0	r/w	Control and status register 0 - Slice operation mode: 0 = single shot, 1 = continuous (one bit per implemented slice starting at LSB).
0xffff10004	CSR1.WORD	31:0	r/w	Control and status register 1 full-word access.
0xffff10004	CSR1.IRQ	15:0	r/w	Control and status register 1 - Slice interrupt pending (one bit per implemented slice starting at LSB); write 0 to clear interrupt.
0xffff10006	CSR1.PRSC	2:0	r/w	Control and status register 1 - Global clock prescaler select.
0xffff10080	SLICE[0].CNT	31:0	r/w	Slice 0 counter register.

Address	Name [C]	Bit(s), Name [C]	R/W	Description
0xfff10084	SLICE[0].THR	31:0	r/w	Slice 0 threshold register.
...	...	...	...	...
0xfff100F8	SLICE[15].CNT	31:0	r/w	Slice 15 counter register.
0xfff100FC	SLICE[15].THR	31:0	r/w	Slice 15 threshold register.

2.8.25. Execution Trace Buffer (TRACER)

Hardware source files:	neorv32_tracer.vhd	
Software driver files:	neorv32_tracer.c	Online software reference (Doxygen)
	neorv32_tracer.h	Online software reference (Doxygen)
Top entity ports:	none	
Configuration generics:	<code>IO_TRACER_EN</code>	implement TRACER module when <code>true</code>
	<code>IO_TRACER_BUFFER</code>	trace buffer depth, has to be zero or a power of two, min 1
	<code>IO_TRACER_SIMLOG_EN</code>	write full trace log to file when <code>true</code> (simulation-only)
CPU interrupts:	fast IRQ channel 5	tracing stop-address match interrupt (see Processor Interrupts)

Key Features

- Delta-tracing of instruction execution
- Auto-stop tracing (and issue interrupt) when reaching a programmable instruction address
- Dump and inspect delta-trace directly from the firmware or via GDB

Overview

The NEORV32 tracer module allows to record program execution in order to find out how the CPU arrived at a certain code point. It is fed by the CPU's [Execution Trace Port](#). In order to keep the data rate low, so-called *instruction delta tracing* is used:

Instruction delta tracing, also known as branch tracing, works by tracking execution from a known start address by sending information about the deltas taken by the program. Deltas are typically introduced by jump, call, return and branch type instructions, although interrupts and exceptions are also types of deltas. Instruction delta tracing provides an efficient encoding of an instruction sequence by exploiting the deterministic way the processor behaves based on the program it is executing.

— RISC-V Trace Specification, github.com/riscv-non-isa/riscv-trace-spec

This means that only non-linear changes of the program counter are recorded. These *deltas* are stored in a local memory (the trace buffer) and can be read by the debugger (e.g. GDB). The size of this trace buffer is configurable via the `IO_TRACER_BUFFER` generic.

The program being executed can also read the trace buffer itself (stand-alone-mode, not debugger required). In the dual-core configuration, for example, one CPU core can analyze the trace of the

other CPU core.



TRACER Demo Program

A demo program for the tracer can be found in `sw/example/demo_tracer`.

Theory of Operation

The tracer module provides four memory-mapped registers: the status and control register **CTRL**, a "stop-tracing" address register **STOP_ADDR** and two read-only registers to read trace data (**DELTA_SRC** and **DELTA_DST**).

The trace module is enabled by setting the **TRACER_CTRL_EN** bit in the control register. If this bit is cleared, the module is disabled and the internal trace buffer is cleared. Bit **TRACER_CTRL_HSEL** selects the hart / CPU core that shall be traced. This bit is read-only zero for the single-core configuration.

Tracing is started by writing **1** to the **TRACER_CTRL_START** control register bit. Tracing can be stopped at any time by manually writing **1** to the **TRACER_CTRL_STOP** control register bit. Software can check if tracing is in progress by reading the **TRACER_CTRL_RUN**. Tracing is *automatically stopped* when program execution reaches the address in the **STOP_ADDR** register. Automatic trace stopping can be disabled by writing **-1** to this register.



Halt Tracing during Debug-Mode

Whenever the CPU enters debug mode, either by a halt request from the debugger or by a (hard-coded) breakpoint or watchpoint, tracing is paused until the CPU resumes normal operation.

During tracing, the module writes the instruction deltas to the internal buffer. The buffer is implemented as FIFO, so that only the last **IO_TRACER_BUFFER** instruction deltas are kept. At least one instruction delta is available in the trace buffer when **TRACER_CTRL_AVAIL** is set. A single entry ("packet") of the trace buffer data can be read via the two registers **DELTA_SRC** and **DELTA_DST**:

- **DELTA_SRC** provides the 31-bit instruction address of the branch source. Bit 0 is set if this trace packet is the very first after trace start; it is cleared for all further packets.
- **DELTA_DST** provides the 31-bit instruction address of the branch destination. Bit 0 is set if this branch was caused by a trap-entry; it is cleared if this branch was caused by a jump/call/branch instruction.



Trace Buffer Read Access

The trace buffer's FIFO read-pointer automatically increments when reading **DELTA_DST**, so make sure to read **DELTA_SRC** before **DELTA_DST**. The implementation as FIFO also means that the **trace data can only be read once**.

Tracer Interrupt

The tracer module features a single interrupt that gets triggered when the traced program reached the address stored in the module's **STOP_ADDR** register. The pending interrupt needs to be explicitly

acknowledged/cleared by writing 1 to the `TRACER_CTRL_IRQ_CLR` control register bit.

Evaluating Trace Data from GDB

Two simple functions for handling the tracer are implemented as GDB script. The script is located in the tracer example program and can be imported into GDB using the `source` command:

Listing 15. Importing the Tracer GDB Script

```
(gdb) source path/to/neorv32/sw/example/demo_tracer/neorv32_tracer.gdb
```

After sourcing, two additional commands are available that can be executed from the GDB command line:

- `tracer_get`: read-out the data from the trace buffer and print the execution history
- `tracer_start arg`: restart the tracer; the CPU ID has to be provided as argument (`arg`, 0 or 1)

Using the tracer from GDB is briefly illustrated in the following example:

Listing 16. Using the Tracer from GDB (GDB started in `neorv32/sw/example/demo_tracer`)

```
(gdb) make clean elf ①
...
(gdb) load ②
...
(gdb) c ③
...
(gdb) source neorv32_tracer.gdb ④
(gdb) tracer_get ⑤
[0] SRC: 0x3b0 -> DST: 0x2cc <TRACE_START>
Line 128 of "main.c" starts at address 0x3b0 <main+212> and ends at 0x3b4 <main+216>.
Line 68 of "main.c" starts at address 0x2cc <test_code> and ends at 0x2d0
<test_code+4>.
...
```

- ① Compile the demo program.
- ② Upload the compiled ELF.
- ③ Start executing. The program will stop re-entering debug-mode when reaching main's `return` statement.
- ④ Import the tracer helper functions script.
- ⑤ Get trace log and print.

Tracer Simulation Logging



Simulation-Only

This feature is available in simulation only.

The NEORV32 tracer module can also be used to generate a complete log of all executed instructions. Simulation trace logging is enabled by the `IO_TRACER_SIMLOG_EN` top generic. This also requires the tracer is implemented (`IO_TRACER_EN` = true). However, no specific configuration of the control register `CTRL` is required. During simulation, all traced instructions are written to log files in the simulator's home folder:

- `neorv32.tracer0.log` for CPU 0
- `neorv32.tracer1.log` for CPU 1 (only if [Dual-Core Configuration](#) is enabled)

The trace log is structured line by line where each line describes an executed instruction. The start of an exemplary trace log might look like this:

Listing 17. Exemplary cut-out from a simulation trace log (here: `neorv32.tracer1.log` showing boot of core 1)

```
52168 256630 0x000024f6 0xc00026f3 U csrrs      x13, cycle, x0
52169 256633 0x000024fa 0xc02025f3 U csrrs      x11, instret, x0
52170 256635 0x000024fe 0x0001      U c.addi      x0, x0, 0
52171 256647 0x00002500 0x0001      U c.addi      x0, x0, 0
52172 256650 0x00002502 0xc0002673 U csrrs      x12, cycle, x0
52173 256653 0x00002506 0xc0202573 U csrrs      x10, instret, x0
52174 256656 0x0000250a 0x00000073 U ecall
52175 256665 0x00003e94 0x34011073 M csrrw      x0, mscratch, x2 <TRAP_ENTRY> ①
52176 256667 0x00003e98 0x7119      M c.addi16sp  x2, x2, -128
52177 256672 0x00003e9a 0xc206      M c.swsp      x1, 4(x2)
52178 256675 0x00003e9c 0x340110f3 M csrrw      x1, mscratch, x2
```

① This line is used for explanation (see below).

Column structure:

1. **52175**: Instruction index ("order"); a linear increasing counter that starts at zero and increments with each executed instruction; printed as decimal integer.
2. **256665**: Time stamp; a linear increasing counter that starts at zero and increments with each clock cycle; printed as decimal integer.
3. **0x00003e94**: Instruction address (program counter); printed as hexadecimal 32-bit value (**0x** prefix).
4. **0x34011073**: instruction word; printed as 32-bit or 16-bit (compressed instruction) hexadecimal value (**0x** prefix).
5. **M**: Current operating mode / privilege level (**M** = machine-mode, **U** = user-mode, **D** = debug-mode); printed as single character.
6. **csrrw**: The decoded instruction mnemonic. For unknown instructions **INVALID** is printed instead. Decompressed 16-bit instructions have a prefixed **c.** (e.g. **c.addi**).
7. **x0, mscratch, x2**: Operands of the decoded instruction.
8. If the CPU encounters a trap (synchronous exception or interrupt) **<TRAP_ENTRY>** is printed at the

end of the line corresponding to the first instruction of the trap handler.



Instruction Execution Time

The execution time of instruction i (number of cycles required for retiring) can be calculated by subtracting the current time stamp i from the next time stamp $i+1$.

Register Map

Table 40. TRACER register map (`struct NEORV32_TRACER`)

Address	Name [C]	Bit(s), Name [C]	R/W	Function
0xfff30000	CTRL	0 TRACER_CTRL_EN	r/w	TRACER enable, reset module when 0
		1 TRACER_CTRL_HSEL	r/w	Hart select for tracing (0 = CPU0, 1 = CPU1)
		2 TRACER_CTRL_START	r/w	Start tracing, flag always reads as zero
		3 TRACER_CTRL_STOP	r/w	Manually stop tracing, flag always reads as zero
		4 TRACER_CTRL_RUN	r/-	Tracing in progress when set
		5 TRACER_CTRL_AVAIL	r/-	Trace data available when set
		6 TRACER_CTRL_IRQ_CLR	r/w	Clear pending interrupt when writing 1, flag always reads as zero
		10:7 TRACER_CTRL_TBM_MSB : TRACER_CTRL_TBM_LSB	r/-	$\log_2(\text{IO_TRACER_BUFFER})$: trace buffer depth
		31:11 reserved	r/-	reserved, hardwired to zero
0xfff30004	STOP_ADDR	31:0	r/w	Stop-tracing-address register
0xfff30008	DELTA_SRC	31:1	r/-	Branch source address, set to -1 to disable automatic stopping
		0	r/-	1 = very first instruction delta in current trace; 0 = any further instruction delta
0xfff3000c	DELTA_DST	31:1	r/-	Branch destination address
		0	r/-	1 = branch due trap entry (interrupt or synchronous exception); 0 = branch due to jump/call/branch instruction

2.8.26. System Configuration Information Memory (SYSINFO)

Hardware source files:	neorv32_sysinfo.vhd	
Software driver files:	neorv32_sysinfo.h	Online software reference (Doxygen)
Top entity ports:	none	
Configuration generics:	all	most of the top's configuration generics
CPU interrupts:	none	

Key Features

- Provides detailed SoC configuration information

Overview

The SYSINFO module allows the application software to determine the setting of most of the **Processor Top Entity - Generics** that are related to CPU and processor/SoC configuration. This device is always implemented - regardless of the actual hardware configuration since the NEORV32 software framework requires information from this device for correct operation. However, advanced users that do not want to use the default NEORV32 software framework can choose to disable the entire SYSINFO module. This might also be suitable for setups that use the processor just as wrapper for a CPU-only configuration.

Register Map

All registers of this module are read-only except for the **CLK** register. Upon reset, the **CLK** registers is initialized from the **CLOCK_FREQUENCY** top entity generic. Application software can override this default value in order, for example, to take into account a dynamic frequency scaling of the processor.

Table 41. SYSINFO register map (**struct** **NEORV32_SYSINFO**)

Address	Name [C]	R/W	Description
0xfffe0000	CLK	r/w	clock frequency in Hz (initialized from top's CLOCK_FREQUENCY generic)
0xfffe0004	MISC[4]	r/-	miscellaneous system configurations (see SYSINFO - Miscellaneous Configuration)
0xfffe0008	SOC	r/-	specific SoC configuration (see SYSINFO - SoC Configuration)
0xfffe000c	CACHE	r/-	cache configuration information (see SYSINFO - Cache Configuration)

SYSINFO - Miscellaneous Configuration

Table 42. SYSINFO **MEM** Bits

Byte	Name [C]	Description
7:0	SYSINFO_MISC_IMEM_MBS : SYSINFO_MISC_IMEM_LSB	$\log_2$ (internal IMEM size in bytes), via top's IMEM_SIZE generic
15:8	SYSINFO_MISC_DMEM_MSB : SYSINFO_MISC_DMEM_LSB	$\log_2$ (internal DMEM size in bytes), via top's DMEM_SIZE generic
19:16	SYSINFO_MISC_HART_MSB : SYSINFO_MISC_HART_LSB	number of physical CPU cores ("harts")
21:20	SYSINFO_MISC_BOOT_MSB : SYSINFO_MISC_BOOT_LSB	boot mode configuration, via top's BOOT_MODE_SELECT generic (see Boot Configuration)
26:22	SYSINFO_MISC_ITMO_MSB : SYSINFO_MISC_ITMO_LSB	$\log_2$ (internal bus timeout cycles), see Bus Monitor and Timeout
31:27	SYSINFO_MISC_ETMO_MSB : SYSINFO_MISC_ETMO_LSB	$\log_2$ (external bus timeout cycles), see Bus Monitor and Timeout

SYSINFO - SoC Configuration

Table 43. SYSINFO SOC Bits

Bit	Name [C]	Description
0	SYSINFO_SOC_BOOTLOADER	set if processor-internal Bootloader is implemented (via top's BOOT_MODE_SELECT generic; see Boot Configuration)
1	SYSINFO_SOC_XBUS	set if Processor-External Bus Interface (XBUS) is implemented (via top's XBUS_EN generic)
2	SYSINFO_SOC_IMEM	set if processor-internal Instruction Memory (IMEM) is implemented (via top's IMEM_EN generic)
3	SYSINFO_SOC_DMEM	set if processor-internal Data Memory (DMEM) is implemented (via top's DMEM_EN generic)
4	SYSINFO_SOC_OCD	set if On-Chip Debugger (OCD) is implemented (via top's OCD_EN generic)
5	SYSINFO_SOC_ICACHE	set if Instruction Cache (iCache) is implemented (via top's ICACHE_EN generic)
6	SYSINFO_SOC_DCACHE	set if Data Cache (dCache) is implemented (via top's DCACHE_EN generic)
7	SYSINFO_SOC_SMC	set if Serial Memory Controller (SMC) is implemented (via top's SMC_EN generic)
8	-	<i>reserved</i> , read as zero
9	-	<i>reserved</i> , read as zero
10	-	<i>reserved</i> , read as zero
11	SYSINFO_SOC_OCD_AUTH	set if on-chip debugger Debug Authentication is implemented (via top's OCD_AUTHENTICATION generic)

Bit	Name [C]	Description
12	SYSINFO_SOC_IMEM_ROM	set if processor-internal Instruction Memory (IMEM) is implemented as pre-initialized ROM (via top's BOOT_MODE_SELECT generic; see Boot Configuration)
13	SYSINFO_SOC_IO_TWD	set if Two-Wire Serial Device Controller (TWD) is implemented (via top's IO_TWD_EN generic)
14	SYSINFO_SOC_IO_DMA	set if Direct Memory Access Controller (DMA) is implemented (via top's IO_DMA_EN generic)
15	SYSINFO_SOC_IO_GPIO	set if General Purpose Input and Output Port (GPIO) is implemented (via top's IO_GPIO_EN generic)
16	SYSINFO_SOC_IO_CLINT	set if Core-Local Interruptor (CLINT) is implemented (via top's IO_CLINT_EN generic)
17	SYSINFO_SOC_IO_UART0	set if Primary Universal Asynchronous Receiver and Transmitter (UART0) is implemented (via top's IO_UART0_EN generic)
18	SYSINFO_SOC_IO_SPI	set if Serial Peripheral Interface Controller (SPI) is implemented (via top's IO_SPI_EN generic)
19	SYSINFO_SOC_IO_TWI	set if Two-Wire Serial Interface Controller (TWI) is implemented (via top's IO_TWI_EN generic)
20	SYSINFO_SOC_IO_PWM	set if Pulse-Width Modulation Controller (PWM) is implemented (via top's IO_PWM_NUM generic)
21	SYSINFO_SOC_IO_WDT	set if Watchdog Timer (WDT) is implemented (via top's IO_WDT_EN generic)
22	SYSINFO_SOC_IO_CFS	set if Custom Functions Subsystem (CFS) is implemented (via top's IO_CFS_EN generic)
23	SYSINFO_SOC_IO_TRNG	set if True Random-Number Generator (TRNG) is implemented (via top's IO_TRNG_EN generic)
24	SYSINFO_SOC_IO_SDI	set if Serial Data Interface Controller (SDI) is implemented (via top's IO_SDI_EN generic)
25	SYSINFO_SOC_IO_UART1	set if Secondary Universal Asynchronous Receiver and Transmitter (UART1) is implemented (via top's IO_UART1_EN generic)
26	SYSINFO_SOC_IO_NEOLED	set if Smart LED Interface (NEOLED) is implemented (via top's IO_NEOLED_EN generic)
27	SYSINFO_SOC_IO_TRACER	set if Execution Trace Buffer (TRACER) is implemented (via top's IO_TRACER_EN generic)
28	SYSINFO_SOC_IO_GPTMR	set if General Purpose Timer (GPTMR) is implemented (via top's IO_GPTMR_NUM generic)

Bit	Name [C]	Description
29	SYSINFO_SOC_IO_SLINK	set if Stream Link Interface (SLINK) is implemented (via top's IO_SLINK_EN generic)
30	SYSINFO_SOC_IO_ONEWIRE	set if One-Wire Serial Interface Controller (ONEWIRE) interface is implemented (via top's IO_ONEWIRE_EN generic)
31	SYSINFO_SOC_SIM	set if NEORV32 is being simulated

SYSINFO - Cache Configuration

The SYSINFO cache register provides information about the configuration of the processor caches:

- **Instruction Cache (iCache)**
- **Data Cache (dCache)**

Table 44. SYSINFO **CACHE** Bits

Bit	Name [C]	Description
3:0	SYSINFO_CACHE_BLOCK_SIZE_3 : SYSINFO_CACHE_BLOCK_SIZE_0	$\log_2$ (cache block size in bytes), via top's CACHE_BLOCK_SIZE generic
7:4	SYSINFO_CACHE_I_NUM_BLOCKS_3 : SYSINFO_CACHE_I_NUM_BLOCKS_0	$\log_2$ (i-cache number of cache blocks), via top's ICACHE_NUM_BLOCKS generic
11:8	SYSINFO_CACHE_D_NUM_BLOCKS_3 : SYSINFO_CACHE_D_NUM_BLOCKS_0	$\log_2$ (d-cache number of cache blocks), via top's DCACHE_NUM_BLOCKS generic
15:1 2	SYSINFO_CACHE_UC_BEGIN_3 : SYSINFO_CACHE_UC_BEGIN_0	start of non-cached address space, 256MB page, via top's CACHE_UC_BASE generic
16	SYSINFO_CACHE_BURSTS_EN	cache burst transfers enabled, via top's CACHE_BURSTS_EN generic
31:1 7	-	<i>reserved</i> , read as zero

Chapter 3. NEORV32 Central Processing Unit (CPU)

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



RISC-V Specifications

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

Section Structure

- **CPU Top Entity - Signals** and **CPU Top Entity - Generics**
- **Architecture** and **Full Virtualization**
- **Execution Trace Port**
- **CPU Tuning Options**
- **Instruction Sets and Extensions** and **Custom Functions Unit (CFU)**
- **Control and Status Registers (CSRs)**
- **Traps, Exceptions and Interrupts**
- **Bus Interface**

3.1. CPU Top Entity - Signals

The following table shows all interface signals of the CPU top entity `rtl/core/neorv32_cpu.vhd`. All ports (except for the data/instruction bus interfaces) use `std_ulogic` or `std_ulogic_vector` types, respectively. The bus ports are wrapped as custom interface types (`bus_req_t` and `bus_rsp_t`). See section **Bus Interface** for the protocol and type documentation.

Table 45. NEORV32 CPU Signal List (Dir = direction seen from the CPU)

Signal	Width/Type	Dir	Description
Clock and Reset			
<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.
Status			
<code>mtime_i</code>	64	in	System time from the Core-Local Interruptor (CLINT) (MTIME).
<code>trace_o</code>	<code>trace_port_t</code>	out	Execution Trace Port.
<code>sleep_o</code>	1	out	Set while the CPU is in Sleep Mode .
Interrupts (Traps, Exceptions and Interrupts)			
<code>msi_i</code>	1	in	RISC-V machine software interrupt.
<code>mei_i</code>	1	in	RISC-V machine external interrupt.
<code>mti_i</code>	1	in	RISC-V machine timer interrupt.
<code>firq_i</code>	16	in	Custom fast interrupt request signals.
<code>dbi_i</code>	1	in	Request CPU to halt and enter debug mode (RISC-V On-Chip Debugger (OCD)).
Instruction Bus Interface			
<code>ifence_o</code>	1	out	Instruction fence / memory ordering.
<code>ibus_req_o</code>	<code>bus_req_t</code>	out	Instruction fetch bus request.
<code>ibus_rsp_i</code>	<code>bus_rsp_t</code>	in	Instruction fetch bus response.
Data Bus Interface			
<code>dfence_o</code>	1	out	Data fence / memory ordering.
<code>dbus_req_o</code>	<code>bus_req_t</code>	out	Data access (load/store) bus request.
<code>dbus_rsp_i</code>	<code>bus_rsp_t</code>	in	Data access (load/store) bus response.

3.2. CPU Top Entity - Generics

Most of the CPU configuration generics are a subset of the **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)" represents a `std_ulogic_vector(x downto y)`.

Table 46. NEORV32 CPU-Exclusive Generic List

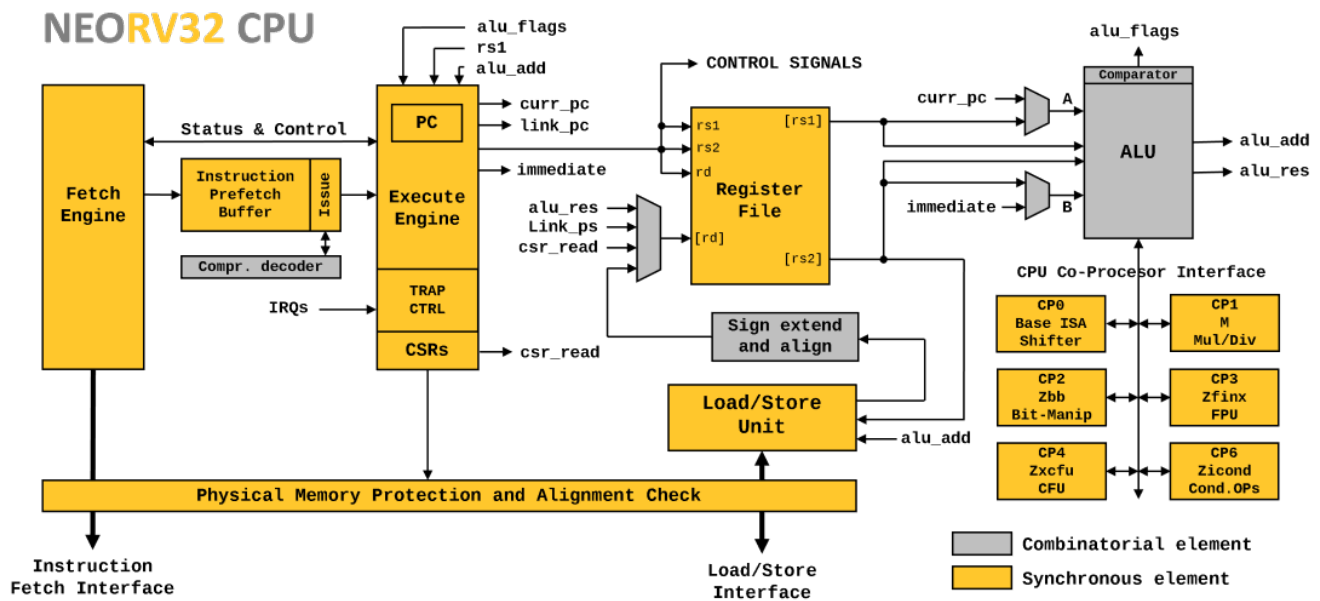
Name	Type	Description
HART_ID	natural	ID of the core (for mhartid CSR).
VENDOR_ID	suv(31:0)	Vendor ID (for mvendorid CSR).
BOOT_ADDR	suv(31:0)	CPU reset address. See section Address Space .
DEBUG_PARK_ADDR	suv(31:0)	"Park loop" entry address for the On-Chip Debugger (OCD) , has to be 4-byte aligned.
DEBUG_EXC_ADDR	suv(31:0)	"Exception" entry address for the On-Chip Debugger (OCD) , has to be 4-byte aligned.
RISCV_ISA_Sdext	boolean	Implement RISC-V-compatible "debug" CPU operation mode required for the On-Chip Debugger (OCD) .
RISCV_ISA_Sdtrig	boolean	Implement RISC-V-compatible Trigger Module . See section On-Chip Debugger (OCD) .
RISCV_ISA_Smpmp	boolean	Implement RISC-V-compatible physical memory protection (PMP). See section Smpmp ISA Extension .
NUM_HW_TRIGGERS	natural	Number of hardware triggers for the Trigger Module



Tuning Option Generics

Additional generics that are related to certain *tuning options* are listed in section **CPU Tuning Options**.

3.3. 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-couples 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**.



No Hardware Support for Misaligned Memory Accesses

The CPU does not support resolving unaligned memory access by the hardware. Any kind of unaligned memory access will raise an exception to allow a software-based emulation. See section **Application Context Handling** for more information.

3.3.1. CPU 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 16-bit instructions or a mixture of both. Unaligned 32bit instructions are explicitly supported. The instruction chunks are queued within the instruction prefetch buffer

(IPB). A consecutive logic reads data from this FIFO, optionally decompresses 16-bit instructions and assembles 32-bit instruction words that are forwarded to the CPU back-end for execution.

The IPB 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 instruction fetch and instruction execution so both stages can operate in parallel to increase performance.

3.3.2. CPU Back-End

The actual instruction execution is conducted by a micro-sequencer-based approach. Execution is split into several micro-operations, each of which generates special control signals, e.g., to configure the ALU or to trigger memory accesses. Simple arithmetic/logical operations require two cycles, while more complex operations require additional cycles depending on the CPU configuration. Since the front-end essentially works in parallel "in the background", an optimal CPI (cycles per instruction) value of 2 can be achieved.

The division into individual micro-operations not only allows for small hardware that reuses CPU parts in different cycles (e.g., the ALU: for arithmetic/logical calculations and also for calculating the next program counter). It also allows for precise monitoring of execution and the generation of precise traps. Hence, the back-end also includes **Control and Status Registers (CSRs)**, the **Traps, Exceptions and Interrupts** logic and the **CPU Debug Mode** controller.

3.3.3. CPU Register File

The data register file (VHDL file `neorv32_cpu_regfile.vhd`) implements the general purpose RISC-V architecture registers `x0` to `x31`. For the "embedded" `rv32e` ISA option only the lower 16 registers are implemented. Register zero (`x0/zero`) always reads as zero and any write access to it has no effect. Read and write accesses are mutually exclusive as they occur in different execution cycles. Hence, the "read-during-write" behavior of the register file is irrelevant.

The register file can be realized using different architecture styles to optimize the implementation for the target platform/technology. The `CPU_RF_ARCH_SEL` CPU tuning option generic is used to select a specific architecture style:

Table 47. CPU Register File Implementation Flavors

<code>CPU_RF_ARCH_SEL</code>	Implementation Style	Note
0 (<code>sram_sync</code>)	Register-Based with Synchronous Read	can infer FPGA block-RAM
1 (<code>sram_async</code>)	Register-Based with Asynchronous Read	can infer FPGA distributed-RAM
2 (<code>reg</code>)	Register-Based with Reset	provides a full hardware reset
3 (<code>latch</code>)	Latch-Based	improved efficiency for ASIC implementations



No Hardware Reset

Apart from the **Register-Based with Reset** style, the register file does not provide a

dedicated hardware reset. Hence, all **x** registers power-up in an undefined state and need software-based initialization (e.g. by the **Start-Up Code (crt0)**).



Technology-Specific Register File

For maximum efficiency regarding area, performance and energy the default register file VHDL module can be replaced by a platform-/technology-specific macro or primitive.

Register-Based with Synchronous Read

This style implements the architecture registers as register-based SRAM array forming a synchronous simple-dual-port RAM. One port is used in time division multiplexing for writing (**rd**) and reading the first operand (**rs1**). The second port is used exclusively for reading the second operand (**rs2**). Read and write accesses are implemented as synchronous operations triggering on the rising clock edge. Hence, this architecture style can infer **FPGA block RAM**.

The **x0/zero** register is also implemented as another physical register within the RAM. However, this requires that these register bits are explicitly initialized (set to zero) by the hardware. This is done transparently by the CPU control without requiring additional processing effort.

Register-Based with Asynchronous Read

For this style, the register file is implemented as register-based SRAM, too. Writing data (**rd**) and reading operands (**rs1** & **rs2**) is performed by three individual port, whereby the read accesses are implemented *asynchronously*. Hence, this architecture style can infer **FPGA distributed RAM**. If the **x0/zero** register is read, the respective operand is masked before the read data is buffered in the operand output register.

Register-Based with Reset

The register file can also be constructed from individual rising-edge-triggered flip flops (registers). No registers are implemented for **x0/zero** which is hardwired to zero. In contrast to the other architecture styles, this implementation flavor provides a **dedicated hardware reset** that initializes all registers to zero during CPU reset.

Latch-Based

The latch-based architecture version implements the individual architecture registers as simple latches that are transparent during the low phase of the clock. A data input register (for **rd**) and two output registers (for **rs1** & **rs2**) ensure correct control and improved timing. No latches are implemented for architecture register **x0/zero**; it is hardwired to zero. This architecture styles provides superior performance and area efficiency for **ASIC implementations** when realized with standard cells.

3.3.4. CPU Arithmetic Logic Unit

The arithmetic/logic unit ("ALU", VHDL file **neorv32_cpu_alu.vhd**) is used for actual data processing

as well as generating memory and branch addresses. All "simple" **I ISA Extension** computational instructions (like **add** and **or**) are implemented as plain combinatorial logic requiring only a single cycle to complete. More sophisticated instructions like shift operations or multiplications are processed by so-called "ALU co-processors" (VHDL files **neorv32_cpu_alu_*.vhd**).

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 unprivileged ISA extensions (e.g. **M ISA Extension**, **Zfinx ISA Extension** and also custom instructions implemented within the **Custom Functions Unit (CFU)**). If the CPU logic detects an illegal instruction or an instruction fetch alignment or bus error no co-processor will be triggered to prevent corruption of the co-processor's internal state (like FPU CSRs).



Multi-Cycle Execution Monitor

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

3.3.5. CPU Load/Store Unit

The load/store unit (VHDL file **neorv32_cpu_lsu.vhd**) takes care of handling data memory accesses via load and store instructions. It performs data adjustment when accessing sub-word data quantities (16-bit or 8-bit) and applies sign-extension for signed load operations.

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



Unaligned Accesses

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

3.3.6. Execution Trace Port

The execution trace port provides information to trace the CPU's instruction execution. This can be used for advanced debugging, profiling or verification. Optionally, the trace data can be tracked by the **Execution Trace Buffer (TRACER)** to perform online branch delta tracing. Furthermore, trace data can also be made available externally by the two **trace_cpu0_o** and **trace_cpu1_o top module ports**.

Trace data is generated by the CPU's trace generator (VHDL file **neorv32_cpu_trace.vhd**), which is enabled if either the **Execution Trace Buffer (TRACER)** is implemented or if the external trace ports are enabled by the **TRACE_PORT_EN top generic** (via the CPU core's **CPU_TRACE_EN** tuning option).

The trace port protocol is partly compatible to the **RISC-V Formal Interface (RVFI)** defined by Yosys (<https://yosyshq.readthedocs.io/projects/riscv-formal/en/latest/rvfi.html>).

The trace ports uses a custom interface type `trace_port_t` (VHDL record) which is defined in the processor's main RTL package file (`neorv32_package.vhd`).

Table 48. Trace Port Signals (`trace_port_t` VHDL type)

Signal	Width	RVFI Equivalent	Description
<code>valid</code>	1	<code>rvfi_valid</code>	All others signals are valid when set; high for one cycle
Instruction Metadata			
<code>order</code>	64	<code>rvfi_order</code>	Instruction index (linear incrementing counter; resets to zero on overflow); bits <code>63:32</code> are hardwired to all-zero
<code>insn</code>	32	<code>rvfi_insn</code>	Instruction word; for compressed instructions the upper 16-bit are set to zero
<code>trap</code>	1	<code>rvfi_trap</code>	Set if the current instruction causes a synchronous exception
<code>halt</code>	1	<code>rvfi_halt</code>	Set if the current instruction is last instruction before halting
<code>intr</code>	1	<code>rvfi_intr</code>	Set if the current instruction is the first instruction of a trap handler
<code>mode</code>	2	<code>rvfi_mode</code>	Current privilege level; <code>00</code> = user, <code>11</code> = machine
<code>ixl</code>	2	<code>rvfi_ixl</code>	Current <code>XLEN</code> ; hardwired to <code>01</code> = 32-bit
<code>debug</code>	1	-	Set if the current instruction is executed in debug-mode
<code>compr</code>	1	-	Set if the current instruction is a decompressed instruction
<code>delta</code>	1	-	Set if the current instruction is the target of a control-flow transfer
<code>cmd32</code>	32	-	32-bit instruction word; compressed instructions are shown in their decompressed 32-bit format
Integer Register			
<code>rs1_addr</code>	5	<code>rvfi_rs1_addr</code>	register <code>rs1</code> address
<code>rs2_addr</code>	5	<code>rvfi_rs2_addr</code>	register <code>rs2</code> address
<code>rs1_rdata</code>	32	<code>rvfi_rs1_rdata</code>	register <code>rs1</code> read data (<i>before</i> instruction execution)
<code>rs2_rdata</code>	32	<code>rvfi_rs2_rdata</code>	register <code>rs2</code> read data (<i>before</i> instruction execution)
<code>rd_addr</code>	5	<code>rvfi_rd_addr</code>	register <code>rd</code> address; all-zero if no write
<code>rd_rdata</code>	32	<code>rvfi_rd_wdata</code>	register <code>rd</code> write data (<i>after</i> instruction execution)

Signal	Width	RVFI Equivalent	Description
Program Counter			
<code>pc_rdata</code>	32	<code>rvfi_pc_rdata</code>	address of current instruction
<code>pc_wdata</code>	32	<code>rvfi_pc_wdata</code>	address of next address; e.g. branch destination (<i>after</i> instruction execution)
Control and Status Register			
<code>csr_addr</code>	12	-	CSR access address; all-zero if no CSR access
<code>csr_rdata</code>	32	-	CSR read data (valid if <code>csr_addr</code> $\neq$ 0)
<code>csr_wdata</code>	32	-	CSR write data
Memory Access			
<code>mem_addr</code>	32	<code>rvfi_mem_addr</code>	memory access address
<code>mem_rmask</code>	4	<code>rvfi_mem_rmask</code>	memory read-mask (byte-wise read-enable; all-zero if no memory read)
<code>mem_wmask</code>	4	<code>rvfi_mem_wmask</code>	memory write-mask (byte-wise write-enable; all-zero if no memory write)
<code>mem_rdata</code>	32	<code>rvfi_mem_rdata</code>	memory read data (valid if <code>mem_rmask</code> $\neq$ 0)
<code>mem_wdata</code>	32	<code>rvfi_mem_wdata</code>	memory write data (valid if <code>mem_wmask</code> $\neq$ 0)

3.3.7. CPU Tuning Options

The top module provides several tuning options to optimize the CPU for performance, size and even security. Note that these configuration options have no impact on the actual functionality (e.g. ISA compatibility).

`CPU_TRACE_EN`

Name	Execution trace generation
Type	<code>boolean</code>
Default	<code>false</code> (disabled)
Description	When this option is enabled, the CPU code generates execution trace data that can be consumed by the Execution Trace Buffer (TRACER) for advanced debugging and profiling. When disabled, no trace data is generated.
Note	This option is automatically enabled if the Execution Trace Buffer (TRACER) is implemented or if the <code>TRACE_PORT_EN</code> top generic is <code>true</code> .

CPU_CONSTT_BR_EN

Name	Constant-time branches (data-independent timing)
Type	boolean
Default	false (disabled)
Description	When this option is enabled, all conditional branch instructions have identical execution times for taken and not taken branch conditions. Thus, all branches behave as if they were always taken (including a complete CPU pipeline flush). Enabling this feature makes execution times more predictable and makes timing side-channel attacks more difficult. Furthermore, this option can help to shorten the CPU's critical path simplifying timing closure. When disabled, not-taken conditional branches are executed faster without clearing the CPU pipeline. Hence, for maximum performance, this feature should be disabled.

CPU_FAST_MUL_EN

Name	Fast multiplication
Type	boolean
Default	false (disabled)
Description	When enabled the M/Zmmul extension's multiplier is implemented as "plain multiplication" allowing the synthesis tool to infer DSP blocks / multiplication primitives. Multiplication operations only require a few cycles due to the DSP-internal register stages. The execution time is time-independent of the provided operands. When disabled the M/Zmmul extension's multiplier is implemented as bit-serial multiplier that computes one result bit in every cycle. Multiplication operations always require 32 clock cycles.

CPU_FAST_MUL_REG

Name	Fast multiplier pipeline register
Type	boolean
Default	false (disabled)
Description	Only effective together with CPU_FAST_MUL_EN. When enabled a pipeline register is added to the fast multiplier so the synthesis tool can retime it into the DSP cascade that is inferred when the DSP blocks are narrower than the operands, shortening the critical path. Costs one extra latency cycle per multiplication (T_mul_latency = 2). When disabled the fast multiplier is bit-identical to before. Wide-DSP targets (e.g. Xilinx UltraScale+) do not need this.

CPU_FAST_SHIFT_EN

Name	Fast bit-shifting
Type	boolean
Default	false (disabled)
Description	When enabled the ALU's shifter unit is implemented as full-parallel barrel shifter that is capable of shifting a data word by an arbitrary number of positions within a single cycle. Hence, the execution time of any base-ISA shift operation is independent of the provided operands. Furthermore, this feature can help to reduce the risk of timing side-channel attacks. Note that the barrel shifter requires a lot of hardware resources for implementation. When disabled the ALU's shifter unit is implemented as bit-serial shifter that can shift the input data only by one position per cycle. Hence, several cycles might be required to complete any base-ISA shift-related operations. Therefore, the execution time of the serial approach is not time-independent of the provided operands. However, the serial approach requires only a few hardware resources and does not impact the critical path.

CPU_RF_ARCH_SEL

Name	Register file implementation style
Type	natural (0..2)
Default	0 (simple-dual-port RAM with synchronous read)
Description	This variable can be used to select a specific architecture for the CPU's general-purpose register file that can be implemented most efficiently for the target platform/technology. see section CPU Register File .

3.3.8. 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 RISC-V **wfi** ("wait for interrupt") instruction.

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

*Sleep during Debugging*

When executed in debug-mode or during single-stepping, **wfi** will behave as simple

`nop` without entering sleep mode.

3.3.9. 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 level 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.

All reserved, unsupported, unavailable, or illegal instructions always raise a precise exception, which, for example, also allows for software-based emulation of additional ISA extensions or system features.

3.4. Bus Interface

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

The bus interface uses two custom interface types: `bus_req_t` is used to propagate the bus access requests downstream from a host to a device. These signals are driven by the request-issuing device (i.e. the CPU core). Vice versa, `bus_rsp_t` is used to return the bus response upstream from a device back to the host and is driven by the accessed device or bus system (i.e. a processor-internal memory or IO device).

Table 49. Bus Interface Signals

Signal	Width	Description
Request Bus (VHDL <code>bus_req_t</code> type)		
<code>meta</code>	5	Access meta information. <code>meta(4:3)</code> : core ID from the lowest two bits of the <code>mhartid</code> CSR; the DMA has core ID <code>0b10</code> <code>meta(2)</code> : set when CPU is in debug-mode <code>meta(1)</code> : current privilege mode, <code>0</code> = user-mode, <code>1</code> = machine-mode <code>meta(0)</code> : access source, <code>0</code> = data access, <code>1</code> = instruction fetch
<code>addr</code>	32	Access address using byte-wise addressing. Half-word (16-bit) and word (32-bit) addresses are aligned accordingly (e.g. LSB = 0 for half-word accesses).
<code>data</code>	32	Write data. Writing individual bytes is controlled via <code>ben</code> .
<code>ben</code>	4	Byte-enable for each byte in <code>data</code> . This signal is used for the write-data as well as for the read-data.
<code>stb</code>	1	Request trigger ("strobe") . This signal is high for exactly one cycle triggering a bus request.
<code>rw</code>	1	Access direction (<code>0</code> = read, <code>1</code> = write).
<code>amo</code>	1	Set if current access is an atomic memory operation.
<code>amoop</code>	4	Actual type of atomic memory operation. Irrelevant if <code>amo</code> is not set.
<code>burst</code>	1	Set during a burst transaction (always set together with <code>lock</code>).
<code>lock</code>	1	Set if locked access; allow exclusive access to the bus system.
Response Bus (VHDL <code>bus_rsp_t</code> type)		
<code>ack</code>	1	Transfer acknowledge, single-shot.
<code>err</code>	1	Transfer error when set, valid if <code>ack</code> = 1.
<code>data</code>	32	Read data, valid if <code>ack</code> = 1.



Adding Register Stages

Arbitrary pipeline stages can be added to the request and response buses at any point to relax timing (at the cost of additional latency).

3.4.1. Bus Interface Protocol

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

The transaction is completed when the accessed device returns an acknowledge via the **ack** signal, which is high for exactly one cycle. Together with **ack** the accessed device returns an optional error status (**err**) and read data (**rdata**) in case of a read-access. **err** and **rdata** are evaluated only when **ack** is high.

The figure below shows three exemplary single-access bus transactions:

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

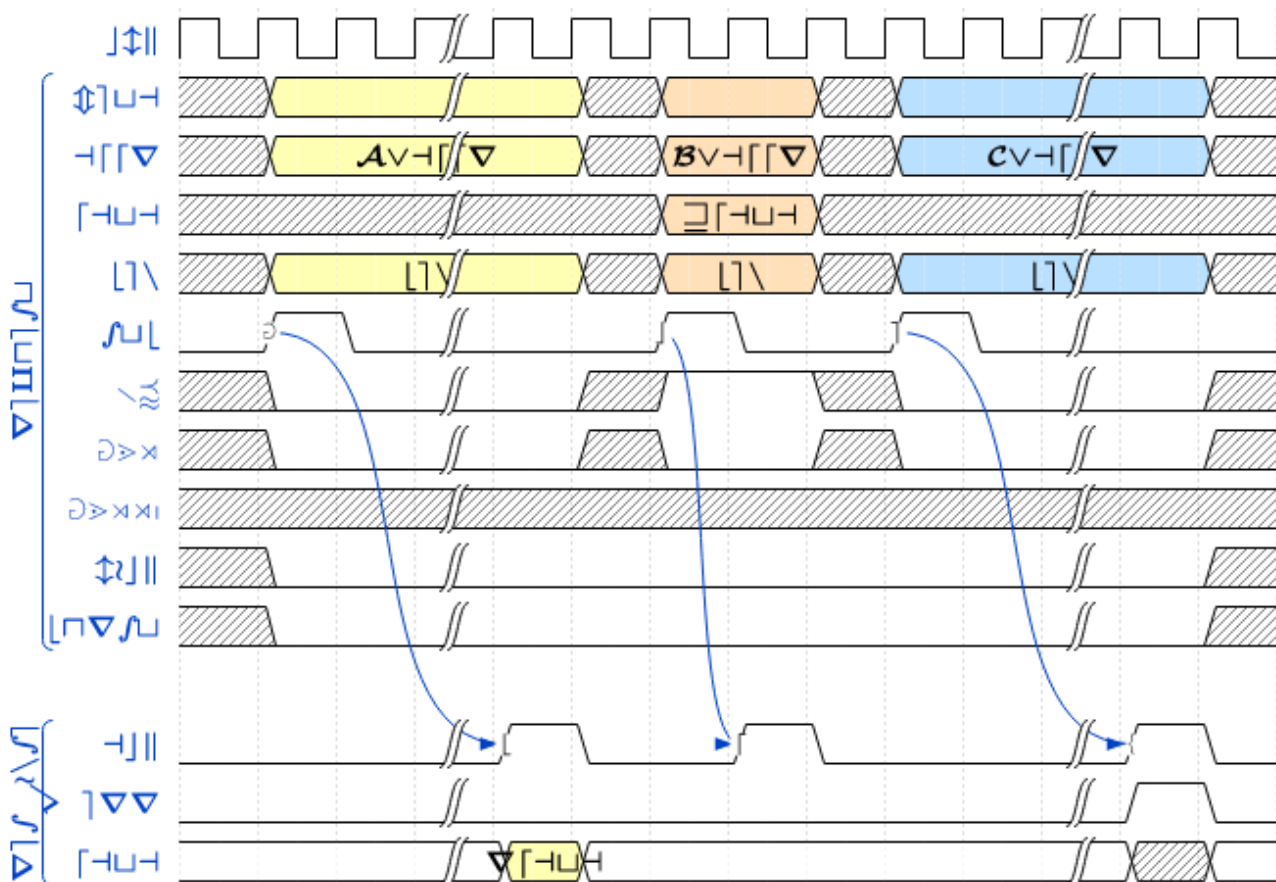


Figure 14. Three Exemplary **Single-Access** Transactions

3.4.2. Locked Bus Accesses and Bursts

Bus hosts like the CPU or the caches can request exclusive access to the downstream bus system using the "bus lock" mechanism. When the bus is locked the locking host has exclusive access and can issue an arbitrary number of transfers that cannot be interleaved by any other host. This feature is used to implement atomic read-modify-write operations and for burst transfers.

The following figure shows a simplified read-followed-by-write transaction implemented as uninterrupted exclusive bus access and a 4-word incrementing-address read burst.

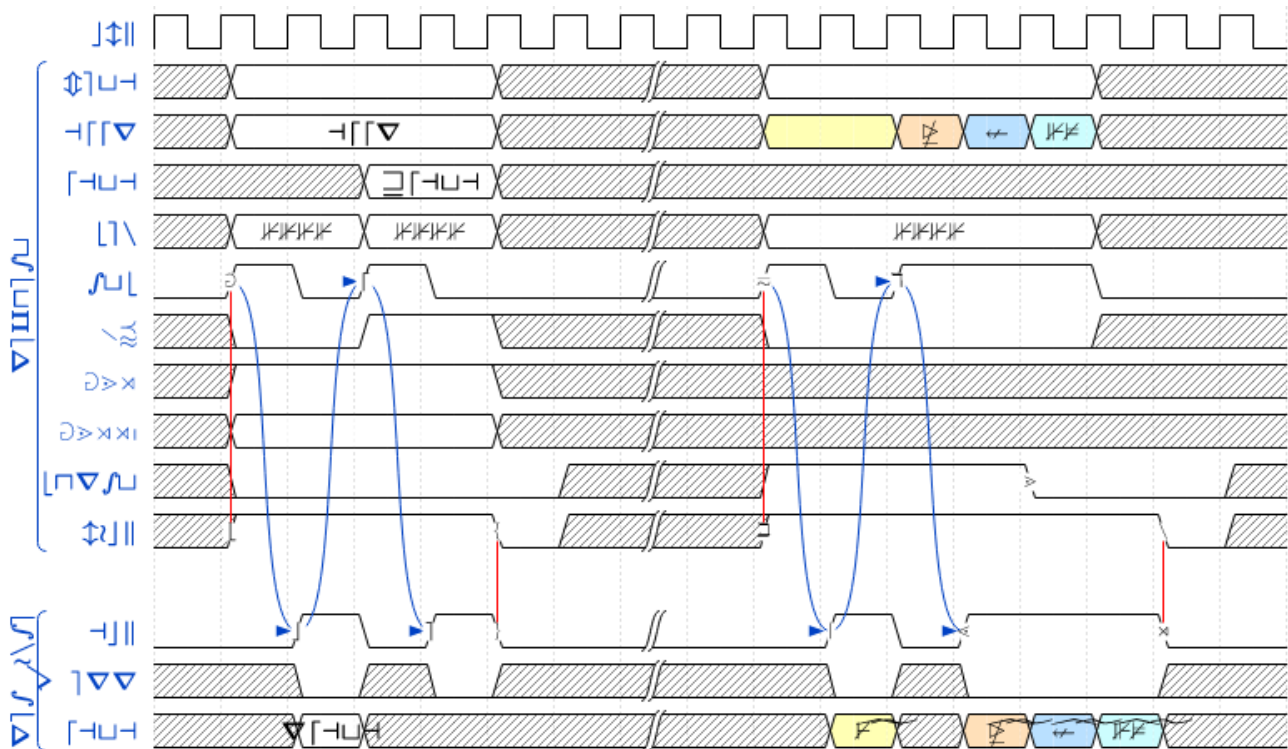


Figure 15. **Atomic Access Transfer** (left) and **4-Word Incrementing-Address Burst Read Transfer** (right)

Atomic Access Transfer

1. The lock request is sent by setting **lock** together with the first **stb** ("a"). **lock** stays high until the end of the entire read-write transfer ("f").
2. The bus is locked when the host received the first **ack** ("c").
3. Now the host can send further requests using **stb** ("d").
4. When the exclusive access is completed the host clears **lock** after receiving the last **ack** of the transfer ("f").
5. **lock** has to be low for at least one cycle after the last **ack** ("f"). This will release the bus lock.

Burst Read Transfer

1. The locked burst request is sent by setting **lock** and **burst** together with the first **stb** ("h"). The address applied to **addr** defines the base/start address of the burst. **lock** stays high until the very last **stb** has been acknowledged ("n").
2. The bus is locked when the host received the first **ack** ("j"). Now the host has exclusive bus

access and can request the remaining burst strobes ("k").

3. For the very last transfer element **burst** is deasserted ("m"). This indicates the end of the burst.
4. The burst transfer is completed when all required **stb** pulses are sent and when all corresponding **ack** have been received ("o"). **burst** is cleared. **lock** is cleared and has to be low for at least one cycle to release the bus lock.



Burst Generators

Only the caches (**i-cache** / **d-cache**) can generate burst transfers (and only if explicitly enabled via **CACHE_BURSTS_EN**).



Fast Burst Response

The accessed device/memory can return all data words of the burst already after receiving the first strobe of the locked transfer ("j").

3.5. Instruction Sets and Extensions

The NEORV32 CPU provides several optional RISC-V-compliant and custom/user-defined ISA extensions. The extensions can be enabled/configured via the according **Processor Top Entity - Generics**. This chapter gives a brief overview of all available ISA extensions.

Table 50. NEORV32 Instruction Set Extensions

Name	Description	Enabled by Generic
A	Atomic memory instructions	<i>Implicitly enabled</i>
B	Bit manipulation instructions	<i>Implicitly enabled</i>
C	Compressed (16-bit) instructions	RISCV_ISA_C
E	Embedded CPU extension (reduced register file size)	RISCV_ISA_E
I	Integer base ISA	Enabled if RISCV_ISA_E is not enabled
M	Integer multiplication and division instructions	RISCV_ISA_M
U	Less-privileged <i>user</i> mode extension	RISCV_ISA_U
X	Platform-specific / NEORV32-specific extension	Always enabled
Zaamo	Atomic read-modify-write memory operations	RISCV_ISA_Zaamo
Zalrsc	Atomic reservation-set memory operations	RISCV_ISA_Zalrsc
Zba	Shifted-add bit manipulation instructions	RISCV_ISA_Zba
Zbb	Basic bit manipulation instructions	RISCV_ISA_Zbb
Zbc	Carry-less multiplication instructions	RISCV_ISA_Zbc
Zbkb	Scalar cryptography bit manipulation instructions	RISCV_ISA_Zbkb
Zbkc	Scalar cryptography carry-less multiplication instructions	RISCV_ISA_Zbkc
Zbkx	Scalar cryptography crossbar permutation instructions	RISCV_ISA_Zbkx
Zbs	Single-bit bit manipulation instructions	RISCV_ISA_Zbs
Zca (an alias)	Compressed instruction without floating-point operations	Implicitly enabled via RISCV_ISA_C
Zcb	Additional code size reduction instructions (build upon C)	RISCV_ISA_Zcb
Zcmop	Compressed may-be-operation (build upon C + Zimop)	RISCV_ISA_Zcmop

Name	Description	Enabled by Generic
Zfinx	Floating-point instructions using integer registers	RISCV_ISA_Zfinx
Zibi	Branches with immediate-comparison	RISCV_ISA_Zibi
Zicntr	Base counters extension	RISCV_ISA_Zicntr
Zicond	Integer conditional operations	RISCV_ISA_Zicond
Zicsr	Control and status register access instructions	Always enabled
Zifencei	Instruction stream synchronization instruction	Always enabled
Zihpm	Hardware performance monitors extension	RISCV_ISA_Zihpm
Zimop	May-be-operations	RISCV_ISA_Zimop
Zkn	Scalar cryptography NIST algorithm suite	<i>Implicitly</i> enabled
Zknd	Scalar cryptography NIST AES decryption instructions	RISCV_ISA_Zknd
Zkne	Scalar cryptography NIST AES encryption instructions	RISCV_ISA_Zkne
Zknh	Scalar cryptography NIST hash function instructions	RISCV_ISA_Zknh
Zks	Scalar cryptography ShangMi algorithm suite	<i>Implicitly</i> enabled
Zksed	Scalar cryptography ShangMi block cypher instructions	RISCV_ISA_Zksed
Zksh	Scalar cryptography ShangMi hash instructions	RISCV_ISA_Zksh
Zkt	Data independent execution time (of cryptography operations)	<i>Implicitly</i> enabled
Zmmul	Integer multiplication-only instructions (subset of M)	RISCV_ISA_Zmmul
Sdext	External debug support extension (on-chip debugger)	OCD_EN
Sdtrig	Debug trigger module extension (hardware breakpoint)	OCD_NUM_HW_TRIGGERS
Smcntrpmf	Counter privilege-mode filtering	RISCV_ISA_Smcntrpmf
Smpmp	Physical memory protection (PMP) extension	RISCV_ISA_Smpmp
Xcfu	Custom / user-defined instructions	RISCV_ISA_Xcfu



RISC-V ISA Specification

For more information regarding the RISC-V ISA extensions please refer to the official specification at riscv.org: <https://docs.riscv.org/reference/home/>

*Illegal Instructions*

The NEORV32 instruction-decoding logic follows a strict illegal instruction detection scheme: All reserved, unavailable, or unsupported instructions always raise an illegal instruction exception when executed. See [Full Virtualization](#) for more information.

*Discovering ISA Extensions*

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

3.5.1. Latency Definitions

This chapter shows the CPI values (cycles per instruction) for each individual instruction/type. For some instructions a placeholder variable from the table below is used. Note that the provided values reflect *optimal bus accesses* (i.e. no congestion/wait states due to dual-core or DMA accesses) and no pipeline wait cycles. To benchmark a certain processor configuration for its setup-specific CPI value please refer to the [sw/example/performance_tests](#) test programs.

Alias	Description
<code>T_shift_latency</code>	1 if <code>CPU_FAST_SHIFT_EN</code> is <code>true</code> ; otherwise <code>T_shift_latency</code> equals the shift offset
<code>T_data_latency</code>	1 for accessing the default processor-internal memories; 1 if accessing the data cache results in a hit; 2 for accessing the default processor-internal peripherals; uncached accesses via the Processor-External Bus Interface (XBUS) may have additional wait states
<code>T_inst_latency</code>	1 for accessing the default processor-internal memories; 1 if accessing the instruction cache results in a hit; 2 for accessing the default processor-internal peripherals; uncached accesses via the Processor-External Bus Interface (XBUS) may have additional wait states; branches to an unaligned address require additional $2 \times T_inst_latency$ cycles
<code>T_mul_latency</code>	1 if <code>CPU_FAST_MUL_EN</code> is <code>true</code> (2 if <code>CPU_FAST_MUL_REG</code> is also <code>true</code>); 32 otherwise
<code>T_cust_latency</code>	latency is defined by the custom Custom Functions Unit (CFU) logic; however, the minimum is 1 and the maximum is 512
<code>T_cache_sync</code>	cycles required to clear/flush the CPU cache(s) (D\$ and/or I\$); depends on actual cache configuration and main memory latency

3.5.2. A ISA Extension

The **A** ISA extension adds instructions and mechanisms for atomic memory operations that can be

used to manage mutually-exclusive access to shared resources. Note that this ISA extension cannot be enabled by a specific generic. Instead, it is enabled if a specific set of sub-extensions is enabled. The **A** extension is shorthand for the following set of other extensions:

- **Zaamo ISA Extension** - Atomic read-modify-write instructions.
- **Zalrsc ISA Extension** - Atomic reservation-set instructions.



A ISA Extension Exceptions

Note that according to the RISC-V spec. load and load-reservate instructions generate load exceptions, whereas store, store-conditional, and AMO instructions generate store exceptions. Accessing an unaligned address will raise an alignment exception (and not a bus fault exception which is also valid according to the RISC-V spec).

3.5.3. **B** ISA Extension

The **B** ISA extension adds instructions for bit-manipulation operations. This ISA extension cannot be enabled by a specific generic. Instead, it is enabled if a specific set of sub-extensions is enabled. The **B** extension is shorthand for the following set of other extensions:

- **Zba ISA Extension** - Address-generation / shifted-add instructions.
- **Zbb ISA Extension** - Basic bit manipulation instructions.
- **Zbs ISA Extension** - Single-bit operations.

A processor configuration which implements **B** must implement all of the above extensions.

3.5.4. **C** ISA Extension

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

Table 51. Instructions and Timing

Class	Instructions	Execution cycles
ALU	c.addi4spn c.nop c.add[i] c.li c.addi16sp c.lui c.and[i] c.sub c.xor c.or c.mv	2
ALU shifts	c.srli c.srai c.slli	2 + T_shift_latency
Branch	c.beqz c.bnez	not taken: 3 taken: 5 + T_inst_latency
Jump/call	c.jal[r] c.j c.jr	5 + T_inst_latency
Load/store	c.lw c.sw c.lwsp c.swsp	4 + T_data_latency
System	c.break	7 + T_inst_latency

Note that the NEORV32 **C** ISA extension only includes the **Zca** instructions; i.e. all instructions from **C**

excluding the single-precision (**F**) and double-precision (**D**) floating-point instructions.

3.5.5. **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 base **I** ISA extension. Due to the reduced register file size an alternate toolchain ABI (**ilp32e***) is required.

3.5.6. **I** ISA Extension

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

Table 52. Instructions and Timing

Class	Instructions	Execution cycles
ALU	add [i] slt [i] slt [i] u xor [i] or [i] and [i] sub 2 lui auipc	
ALU shifts	sll [i] srl [i] sra [i]	3 + T_shift_latency
Branch	beq bne blt bge bltu bgeu	not taken: 3 taken: 5 + T_inst_latency
Jump/call	jal [r]	5 + T_inst_latency
Load/store	lb lh lw lbu lhu sb sh sw	4 + T_data_latency
Data fence	fence	2 + T_cache_sync
System	ecall ebreak mret	7 + T_inst_latency
System	wfi	3



fence Instruction

The **fence** instruction triggers a load/store memory ordering operation by setting the CPU's **dfence_o** signal for exactly one cycle. See section **Memory Coherence** for more information. If both the predecessor (**PREC**) and successor (**SUCC**) fields are zero, the instruction is executed as a NOP without triggering any memory ordering.



wfi Instruction

The **wfi** instruction is used to enter CPU **Sleep Mode**.

3.5.7. **M** ISA Extension

Hardware-accelerated integer multiplication and division operations are available via the RISC-V **M** ISA extension. This ISA extension is implemented as multi-cycle ALU co-process (**rtl/core/neorv32_cpu_alu_muldiv.vhd**).

Table 53. Instructions and Timing

Class	Instructions	Execution cycles
Multiplication	<code>mul mulh mulhsu mulhu</code>	$3 + T_{mul_latency}$
Division	<code>div divu rem remu</code>	$3 + 32$



Multiplication Tuning Options

The physical implementation of the multiplier can be tuned for certain design goals like area or throughput. See section [CPU Tuning Options](#) for more information.

3.5.8. 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](#).

3.5.9. 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](#)). * Additional [NEORV32-Specific CSRs](#).

3.5.10. Zaamo ISA Extension

The **Zaamo** ISA extension is a sub-extension of the RISC-V [A ISA Extension](#) and compromises instructions for atomic read-modify-write operations. It is enabled by the top's **RISCV_ISA_Zaamo** generic. Atomic read-modify-write operations are handled by the processor's [Atomic Memory Operations Controller](#).

Table 54. Instructions and Timing

Class	Instructions	Execution cycles
Atomic read-modify-write	<code>amoswap.w amoaddd.w amoand.w amoor.w amoxor.w amomax[u].w amomin[u].w</code>	$4 + 2x T_{data_latency} + 1$

aq and rl Bits



The instruction word's **aq** and **rl** memory ordering bits are not evaluated by the hardware at all. However, the CPU and especially the [Data Cache \(dCache\)](#) treats all atomic memory operation as if the **aq** and **rl** bits were set (**for the targeted address / cache line**). See [Memory Coherence](#) for more information.

3.5.11. Zalrsc ISA Extension

The **Zalrsc** ISA extension is a sub-extension of the RISC-V **A ISA Extension** and compromises instructions for reservation-set operations. It is enabled by the top's **RISCV_ISA_Zalrsc** generic. Atomic reservation-set operations are handled by the processor's **Atomic Reservation-Set Controller**.

Table 55. Instructions and Timing

Class	Instructions	Execution cycles
Atomic reservation-set	lr.w sc.w	4 + T_data_latency



aq and **rl** Bits

The instruction word's **aq** and **rl** memory ordering bits are not evaluated by the hardware at all. However, the CPU and especially the **Data Cache (dCache)** treats all atomic memory operation as if the **aq** and **rl** bits were set (**for the targeted address / cache line**). See **Memory Coherence** for more information.

3.5.12. Zcb ISA Extension

This ISA extension is part of the "code size reduction" ISA extension **Zc*** and adds additional compressed instruction for common operation. **Zcb** requires the **C ISA Extension** to be enabled. Some instructions may require additional ISA (sub-) extensions.

Table 56. Instructions and Timing

Class	Instructions	Depends on ISA Ext.	Execution cycles
Memory	c.lbu c.lh c.lhu c.sb c.sh	-	4 + T_data_latency
Logic	c.not c.zext.b	-	2
Logic	c.sext.b c.zext.h c.zext.h	B or Zbb	3
Arithmetic	c.mul	M or Zmmul	3 + T_mul_latency



RISC-V GCC ISA String

Technically, **Zcb** requires the **Zca** extension which is **C** excluding the floating-point operations. Therefore, **Zca** and **Zcb** must be contained in the ISA string so that the compiler generates **Zcb** instructions.. Example: **MARCH=rv32imc_zca_zcb_...**

3.5.13. Zifencei ISA Extension

This instruction is the only standard mechanism to ensure that stores visible to a hart will also be visible to its instruction fetch. When executed, the CPU sets its **ifence_o** signal for exactly one cycle and flushes its instruction prefetch buffer and. See section **Memory Coherence** for more information.

Table 57. Instructions and Timing

Class	Instructions	Execution cycles
Instruction fence	<code>fence.i</code>	2 + <code>T_cache_sync</code>

3.5.14. **Zfinx** ISA Extension

The **Zfinx** floating-point extension is an *alternative* of the standard **F** floating-point ISA extension. It also uses the integer register file `x` to store and operate on floating-point data instead of a dedicated floating-point register file. Thus, the **Zfinx** extension requires less hardware resources and features faster context changes. This also implies that there are NO dedicated `f` register file-related load/store or move instructions. The **Zfinx** extension's floating-point unit is controlled via dedicated **Floating-Point CSRs**. This ISA extension is implemented as multi-cycle ALU co-process (`rtl/core/neorv32_cpu_alu_fpu.vhd`).



Fused / Multiply-Add and Division / Square Root Instructions

Fused multiply-add instructions `f[n]m[add/sub].s` and division `fdiv.s` and square root `fsqrt.s` instructions are not supported yet. are not supported. Special GCC flags are used to prevent the compiler from emitting these operations (see **Default Compiler Flags**).



Subnormal Numbers

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

Table 58. Instructions and Timing

Class	Instructions	Execution cycles
Add/sub	<code>fadd.s fsub.s</code>	21
Multiply	<code>fmul.s</code>	13
Compare	<code>fmin.s fmax.s feq.s flt.s fle.s</code>	5
Convert (float → int)	<code>fcvt.w.s fcvt.wu.s</code>	10
Convert (int → float)	<code>fcvt.s.w fcvt.s.wu</code>	42
Sign-injection	<code>fsgnj.s fsgnjn.s fsgnjx.s</code>	4
Classify	<code>fclass.s</code>	4

3.5.15. **Zibi** ISA Extension

The RISC-V **Zibi** ISA extension adds two new "branch with immediate" instructions. The instruction word's `rs2` bit-field selects one out of 32 pre-defined immediate values which is used for the actual

comparison.



Non-Ratified ISA Extension

The RISC-V **Zibi** ISA extension has not been ratified yet!

Table 59. Instructions and Timing

Class	Instructions	Execution cycles
Branch with immediate	beqi bnei	not taken: 3 taken: 5 + T_inst_latency

3.5.16. 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.



Constrained Access

User-level access to the counter CSRs can be constrained by the **mcounteren** CSR.

3.5.17. Zicond ISA Extension

The **Zicond** ISA extension adds integer conditional move primitives that allow to implement branch-less control flows. It is enabled by the top's **RISCV_ISA_Zicond** generic. This ISA extension is implemented as multi-cycle ALU co-processor (**rtl/core/neorv32_cpu_alu_cond.vhd**).

Table 60. Instructions and Timing

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

3.5.18. 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.



Side-Effects if Destination is Zero-Register

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

Table 61. Instructions and Timing

Class	Instructions	Execution cycles
System	csrrw[i] csrrs[i] csrrc[i]	3

3.5.19. **Zihpm** ISA Extension

In additions to the base counters the NEORV32 CPU provides up to 29 hardware performance monitors (HPM; **mhpmcounter[h]** and **hpmcounter[h]**), which can be used to benchmark applications. Each HPM consists of a counter (up to 64-bit) and a corresponding 32-bit event configuration CSR (**mhpmevent**). The counter CRSs are split into a high-word and a low-word CSR. The effective counter width can be defined via the top's **HPM_CNT_WIDTH** generic.

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.



Increment Inhibit

The event-driven increment of the HPMs can be deactivated individually via the **mcounthinhibit** CSR.



Constrained Access

User-level access to the counter CSRs can be constrained by the **mcounteren** CSR.

3.5.20. **Zimop** ISA Extension

The **Zimop** ISA extension introduces the concept of instructions that may be operations (MOPs). MOPs are initially defined to simply write zero to the destination register, but are designed to be redefined by later extensions to perform some other system/environment-class actions. NEORV32 can implement this ISA extension for software compatibility. Without the **Zimop** extension, any **mop.r*** instruction will raise an illegal instruction exception.

Table 62. Instructions and Timing

Class	Instructions	Execution cycles
System	mop.r.[0..31] mop.rr.[0..7]	3

3.5.21. **Zcmop** ISA Extension

The **Zcmop** ISA extension build up upon the **Zimop ISA Extension** by adding compressed may-be-operation that decompress into their 32-bit counterparts (i.e. **c.mop.n** → **mop.r.n** with **r = x0**). Note that this extensions requires the **Zimop ISA Extension** & **C ISA Extension**.

Table 63. Instructions and Timing

Class	Instructions	Execution cycles
System	c.mop.n	3

3.5.22. **Zba** ISA Extension

The **Zba** sub-extension is part of the *RISC-V bit manipulation* ISA specification (**B ISA Extension**) and

adds shifted-add / address-generation instructions. It is enabled by the top's **RISCV_ISA_Zba** generic. This ISA extension is implemented as multi-cycle ALU co-processor (`rtl/core/neorv32_cpu_alu_bitmanip.vhd`).

Table 64. Instructions and Timing

Class	Instructions	Execution cycles
Shifted-add	<code>sh1add sh2add sh3add</code>	4

3.5.23. **Zbb** ISA Extension

The **Zbb** sub-extension is part of the *RISC-V bit manipulation* ISA specification (**B ISA Extension**) and adds the basic bit manipulation instructions. It is enabled by the top's **RISCV_ISA_Zbb** generic. This ISA extension is implemented as multi-cycle ALU co-processor (`rtl/core/neorv32_cpu_alu_bitmanip.vhd`).

Table 65. Instructions and Timing

Class	Instructions	Execution cycles
Logic with negate	<code>andn orn xnor</code>	4
Bit count	<code>clz ctz cpop</code>	4 + T_shift_latency
Integer maximum/minimum	<code>min[u] max[u]</code>	4
Sign/zero extension	<code>sext.b sext.h zext</code>	4
Bitwise rotation	<code>rol ror[i]</code>	4 + T_shift_latency
OR-combine	<code>orc.b</code>	4
Byte-reverse	<code>rev8</code>	4



Shifter Tuning Options

The physical implementation of the bit-shifter can be tuned for certain design goals like area or throughput. See section **CPU Tuning Options** for more information.

3.5.24. **Zbc** ISA Extension

The **Zbc** sub-extension adds carry-less multiplication instructions. It provides the same instructions as the **Zbkc** ISA extension but also provides a *reversed* instruction (`clmulr`). It is enabled by the top's **RISCV_ISA_Zbc** generic. This ISA extension is implemented as multi-cycle ALU co-processor (`rtl/core/neorv32_cpu_alu_bitmanip.vhd`).

Table 66. Instructions and Timing

Class	Instructions	Execution cycles
Carry-less multiply	<code>clmul clmulh clmulr</code>	4 + 32

3.5.25. **Zbs** ISA Extension

The **Zbs** sub-extension is part of the *RISC-V bit manipulation* ISA specification (**B ISA Extension**) and adds single-bit operations. It is enabled by the top's `RISCV_ISA_Zbs` generic. This ISA extension is implemented as multi-cycle ALU co-processor (`rtl/core/neorv32_cpu_alu_bitmanip.vhd`).

Table 67. Instructions and Timing

Class	Instructions	Execution cycles
Single-bit	<code>sbset[i] sbclr[i] sbinv[i] sbext[i]</code>	4

3.5.26. **Zbkb** ISA Extension

The **Zbkb** sub-extension is part of the *RISC-V scalar cryptography* ISA specification and extends the *RISC-V bit manipulation* ISA extension with additional instructions. It is enabled by the top's `RISCV_ISA_Zbkb` generic. Note that enabling this extension will also enable the **Zbb** basic bit-manipulation ISA extension (which is extended by **Zknb**). This ISA extension is implemented as multi-cycle ALU co-processor (`rtl/core/neorv32_cpu_alu_bitmanip.vhd`).

Table 68. Instructions and Timing (in addition to **Zbb**)

Class	Instructions	Execution cycles
Packing	<code>pack packh</code>	4
Interleaving	<code>zip unzip</code>	4
Byte-wise bit reversal	<code>brev8</code>	4

3.5.27. **Zbkc** ISA Extension

The **Zbkc** sub-extension is part of the *RISC-V scalar cryptography* ISA extension and adds carry-less multiplication instruction. ISA extension with additional instructions. It is enabled by the top's `RISCV_ISA_Zbkc` generic. This ISA extension is implemented as multi-cycle ALU co-processor (`rtl/core/neorv32_cpu_alu_bitmanip.vhd`).

Table 69. Instructions and Timing

Class	Instructions	Execution cycles
Carry-less multiply	<code>clmul clmulh</code>	4 + 32

3.5.28. **Zbxx** ISA Extension

The **Zbxx** sub-extension is part of the *RISC-V scalar cryptography* ISA specification and adds crossbar permutation instructions. It is enabled by the top's **RISCV_ISA_Zbxx** generic. This ISA extension is implemented as multi-cycle ALU co-processor (`rtl/core/neorv32_cpu_alu_crypto.vhd`).

Table 70. Instructions and Timing

Class	Instructions	Execution cycles
Crossbar permutation	xperm8 xperm4	4

3.5.29. **Zkn** ISA Extension

The **Zkn** ISA extension is part of the *RISC-V scalar cryptography* ISA specification and defines the "NIST algorithm suite". This ISA extension cannot be enabled by a specific generic. Instead, it is enabled if a specific set of cryptography-related sub-extensions is enabled.

The **Zkn** extension is shorthand for the following set of other extensions:

- **Zbkb ISA Extension** - Bit manipulation instructions for cryptography.
- **Zbkc ISA Extension** - Carry-less multiply instructions.
- **Zbxx ISA Extension** - Cross-bar permutation instructions.
- **Zkne ISA Extension** - AES encryption instructions.
- **Zknd ISA Extension** - AES decryption instructions.
- **Zknh ISA Extension** - SHA2 hash function instructions.

A processor configuration which implements **Zkn** must implement all of the above extensions.

3.5.30. **Zknd** ISA Extension

The **Zknd** sub-extension is part of the *RISC-V scalar cryptography* ISA specification and adds NIST AES decryption instructions. It is enabled by the top's **RISCV_ISA_Zknd** generic. This ISA extension is implemented as multi-cycle ALU co-processor (`rtl/core/neorv32_cpu_alu_crypto.vhd`).

Table 71. Instructions and Timing

Class	Instructions	Execution cycles
AES decryption	aes32dsi aes32dsmi	6

3.5.31. **Zkne** ISA Extension

The **Zkne** sub-extension is part of the *RISC-V scalar cryptography* ISA specification and adds NIST AES encryption instructions. It is enabled by the top's **RISCV_ISA_Zkne** generic. This ISA extension is implemented as multi-cycle ALU co-processor (`rtl/core/neorv32_cpu_alu_crypto.vhd`).

Table 72. Instructions and Timing

Class	Instructions	Execution cycles
AES decryption	<code>aes32esi aes32esmi</code>	6

3.5.32. **Zknh** ISA Extension

The **Zknh** sub-extension is part of the *RISC-V scalar cryptography* ISA specification and adds NIST hash function instructions. It is enabled by the top's `RISCV_ISA_Zknh` generic. This ISA extension is implemented as multi-cycle ALU co-processor (`rtl/core/neorv32_cpu_alu_crypto.vhd`).

Table 73. Instructions and Timing

Class	Instructions	Execution cycles
sha256	<code>sha256sig0 sha256sig1 sha256sum0 sha256sum1</code>	4
sha512	<code>sha512sig0h sha512sig0l sha512sig1h sha512sig1l sha512sum0r sha512sum1r</code>	4

3.5.33. **Zks** ISA Extension

The **Zks** ISA extension is part of the *RISC-V scalar cryptography* ISA specification and defines the "ShangMi algorithm suite". This ISA extension cannot be enabled by a specific generic. Instead, it is enabled if a specific set of cryptography-related sub-extensions is enabled.

The **Zks** extension is shorthand for the following set of other extensions:

- **Zbkb ISA Extension** - Bit manipulation instructions for cryptography.
- **Zbkc ISA Extension** - Carry-less multiply instructions.
- **Zbkx ISA Extension** - Cross-bar permutation instructions.
- **Zksed ISA Extension** - SM4 block cipher instructions.
- **Zksh ISA Extension** - SM3 hash function instructions.

A processor configuration which implements **Zks** must implement all of the above extensions.

3.5.34. **Zksed** ISA Extension

The **Zksed** sub-extension is part of the *RISC-V scalar cryptography* ISA specification and adds ShangMi block cypher and key schedule instructions. It is enabled by the top's `RISCV_ISA_Zksed` generic. This ISA extension is implemented as multi-cycle ALU co-processor (`rtl/core/neorv32_cpu_alu_crypto.vhd`).

Table 74. Instructions and Timing

Class	Instructions	Execution cycles
Block cyphers	<code>sm4ed</code>	6

Class	Instructions	Execution cycles
Key schedule	sm4ks	6

3.5.35. **Zksh** ISA Extension

The **Zksh** sub-extension is part of the *RISC-V scalar cryptography* ISA specification and adds ShangMi hash function instructions. It is enabled by the top's **RISCV_ISA_Zksh** generic. This ISA extension is implemented as multi-cycle ALU co-processor (**rtl/core/neorv32_cpu_alu_crypto.vhd**).

Table 75. Instructions and Timing

Class	Instructions	Execution cycles
Hash	sm3p0 sm3p1	6

3.5.36. **Zkt** ISA Extension

The **Zkt** sub-extension is part of the *RISC-V scalar cryptography* ISA specification and guarantees data independent execution times of cryptography and cryptography-related instructions. The ISA extension cannot be enabled by a specific generic. Instead, it is enabled implicitly by certain CPU configurations.

The RISC-V **Zkt** specifications provides a list of instructions that are included within this specification. However, not all instructions are required to be implemented. Rather, every one of these instructions that the core does implement must adhere to the requirements of **Zkt**.

Table 76. **Zkt** instruction listing

Parent extension	Instructions	Data independent execution time?
RVI	lui auipc add[i] slt[i][u] xor[i] or[i] and[i] sub	yes
	sll[i] srl[i] sra[i]	yes if CPU_FAST_SHIFT_EN enabled
RVM	mul[h] mulh[s]u	yes
RVC	c.nop c.addi c.lui c.andi c.sub c.xor c.and c.mv c.add	yes
	c.srli c.srai c.slli	yes if CPU_FAST_SHIFT_EN enabled
	c.mul c.not c.zext.b (Zcb)	yes
RVK	aes32ds[m]i aes32es[m]i sha256sig* sha512sig* sha512sum* sm3p0 sm3p1 sm4ed sm4ks	yes
RVB	xperm4 xperm8 andn orn xnor pack[h] brev8 rev8	yes
	ror[i] rol	yes if CPU_FAST_SHIFT_EN enabled



Data-Independent Timing of Branches

To further reduce the possibility of timing side-channel attacks, conditional branches can be executed with constant time (i.e. data-independent execution times). See the [CPU_CONSTT_BR_EN CPU Tuning Options](#) for more information.

3.5.37. Zmmul - ISA Extension

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

3.5.38. Xcfu ISA Extension

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

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



CFU Execution Time

The actual CFU execution time depends on the logic being implemented. The CPU architecture requires a minimal execution time of 3 cycles (purely combinatorial CFU operation) and automatically terminates execution after 512 cycles if the CFU does not complete operation within this time window.

Table 77. Instructions and Timing

Class	Instructions	Execution cycles
Custom instructions	Instruction words with custom-0 or custom-1 opcode	$3 + T_{\text{cust_latency}}$

3.5.39. Smpmp ISA Extension

The NEORV32 physical memory protection (PMP) provides an elementary memory protection mechanism that can be used to configure read/write(execute permission of arbitrary memory regions. 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 NEORV32 PMP is fully compatible to the RISC-V Privileged Architecture Specifications and is configured via several CSRs ([Machine Physical Memory Protection CSRs](#)). Several [Processor Top Entity - Generics](#) are provided to adjust the CPU's PMP capabilities according to the application requirements (pre-synthesis):

1. **PMP_NUM_REGIONS** defines the number of implemented PMP regions (0..16); setting this generic to zero will result in absolutely no PMP logic being implemented
2. **PMP_MIN_GRANULARITY** defines the minimal granularity of each region (has to be a power of 2, minimal granularity = 4 bytes); note that a smaller granularity will lead to wider comparators and thus, to higher area footprint and longer critical path
3. **PMP_TOR_MODE_EN** controls the implementation of the top-of-region (TOR) mode (default = true); disabling this mode will reduce area footprint
4. **PMP_NAP_MODE_EN** controls the implementation of the naturally-aligned-power-of-two (NA4 and NAPOT) modes (default = true); disabling this mode will reduce area footprint and critical path length



PMP Permissions when in Debug Mode

When in debug-mode all PMP rules are bypassed/ignored granting the debugger maximum access permissions.



PMP Time-Multiplex

Instructions are executed in a multi-cycle manner. Hence, data access (load/store) and instruction fetch cannot occur at the same time. Therefore, the PMP hardware uses only a single set of comparators for memory access permissions checks that are switched in an iterative, time-multiplex style reducing hardware footprint by approx. 50% while maintaining full security features and RISC-V compatibility.



PMP Memory Accesses

Load/store accesses for which there are insufficient access permission do not trigger any memory/bus accesses at all. In contrast, instruction accesses for which there are insufficient access permission nevertheless lead to a memory/bus access (causing potential side effects on the memory side). However, the fetched instruction will be discarded and the corresponding exception will still be triggered precisely.

3.5.40. **Smcptrpmf** - ISA Extension

The counter privilege-mode filtering (**Smcptrpmf**) ISA extension allows to halt the standard cycle (**[m]cycle[h]**) and instructions-retired (**[m]instret[h]**) counters if the CPU operates in a specific privilege mode (machine or user). Four additional CSRs are provided by this ISA extension:

- **mcyclecfg[h]**
- **minstretcfg[h]**

3.5.41. **Sdext** ISA Extension

This ISA extension enables the RISC-V-compatible "external debug support" by implementing the CPU "debug mode", which is required for the on-chip debugger. See section **On-Chip Debugger**

(OCD) / [CPU Debug Mode](#) for more information.

Table 78. Instructions and Timing

Class	Instructions	Execution cycles
System	dret	5

3.5.42. **Sdtrig** ISA Extension

This ISA extension implements the RISC-V-compatible "trigger module" which provides support for hardware breakpoints for the on-chip-debugger. See section [On-Chip Debugger \(OCD\) / Trigger Module](#) for more information.

3.6. Custom Functions Unit (CFU)

The Custom Functions Unit (CFU) is a NEORV32-specific ISA extension (**Xcfu**) providing a generic hardware module that can be used to implement **custom RISC-V instructions**.



CFU vs. Other Customization Options

The CFU is *not* intended for complex and **CPU-independent** functional units that implement complete accelerators (like full block-based AES encryption). These kind of accelerators should be implemented as stream- or memory-mapped co-processor (e.g. **Custom Functions Subsystem (CFS)**) to allow CPU-independent operation. A comparative survey of all NEORV32-specific hardware extension/customization options is provided in the user guide section **Adding Custom Hardware Modules**.



Default CFU Hardware/Software Example

The default CFU module (`rtl/core/neorv32_cpu_alu_cfu.vhd`) implements the *Extended Tiny Encryption Algorithm (XTEA)* as example application. The according demo program is located in `sw/example/demo_cfu`.

3.6.1. CFU Instruction Formats

The CFU instructions can utilize a limited set of RISC-V OPCODES in **rv32** 32-bit instruction encoding space for custom purpose. It is up to the designer to decide exactly what function these opcodes serve and what type of instruction type/format they use. So far, four OPCODES are supported. Two of them ("custom-0" and "custom-1") have been explicitly reserved by the RISC-V ISA spec. for user-defined extensions ("Guaranteed Non-Standard Encoding Space"). These should be the first choice for implementing custom CFU instruction.

The other two OPCODES ("op-32" and "op-imm-32") are reserved by the RISC-V spec. for extended operations and future ISA extensions. Note that the support/handling of the latter two OPCODES in NEORV32 might change in future.

Listing 18. Available CFU OPCODES (defined in `sw/lib/include/neorv32_intrinsics.h`)

```
#define RISC_V_OPCODE_CUSTOM0 0b0001011 // RISC-V "CUSTOM-0"
#define RISC_V_OPCODE_CUSTOM1 0b0101011 // RISC-V "CUSTOM-1"
#define RISC_V_OPCODE_OP32    0b0111011 // RISC-V "OP-32"
#define RISC_V_OPCODE_OPIMM32 0b0011011 // RISC-V "OP-IMM-32"
```



Default CFU Example

For the provided XTEA CFU example, the "custom-0" opcode is used to implement custom **RISC-V R-Type Instructions** while the "custom-1" is used to implement custom **RISC-V I-Type Instructions**.

RISC-V R-Type Instructions

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



Figure 16. R-Type Instruction Format

RISC-V I-Type Instructions

The I-type instructions operate on one source registers **rs1** and a 12-bit immediate operand **imm12** and return the processing result via the destination register **rd**. The actual operation can be defined by using the **funct3** bit field. Alternatively, this immediate can also be used to pass additional data to the CFU like offsets, look-up-tables addresses or shift-amounts. However, the actual functionality is entirely user-defined. Note that all immediate values are always compile-time-static.



Figure 17. I-Type Instruction Format

3.6.2. Using Custom Instructions in Software

The custom instructions provided by the CFU can be utilized in plain C-code using **intrinsics**. Intrinsics behave like normal C-functions but under the hood they substitute a sequence of assembly instruction for the original function call. For now, the NEORV32 software framework provides two pre-defined prototypes for custom instructions:

Listing 19. RISC-V Instruction Prototypes (simplified; defined in `sw/lib/include/neorv32_intrinsics.h`)

```
uint32_t RISCV_INSTR_R_TYPE(int opcode, int funct3, int funct7, uint32_t rs1, uint32_t rs2);
uint32_t RISCV_INSTR_I_TYPE(int opcode, int funct3, uint32_t rs1, int imm12);
```

These intrinsics always return a 32-bit value of type **uint32_t** (the processing result) which can be discarded if not needed. The **funct3**, **funct7** and **imm12** bit-fields are used to pass compile-time-static literals to the CFU. The **rs1** and **rs2** arguments pass actual runtime data to the CFU via register addresses. These register arguments can be populated with variables or literals; the compiler will add the required code to move the data into a register before passing it to the CFU.

Listing 20. CFU Instruction Examples

```
uint32_t foo = 12345;
uint32_t res = RISCV_INSTR_I_TYPE(RISCV_OPCODE_CUSTOM1, 0b001, foo, 0xABC);
```

```
res = RISCV_INSTR_R_TYPE(RISCV_OPCODE_CUSTOM0, 0b101, 0b1100110, res, foo);
```

3.6.3. Custom Instructions Hardware

The CFU hardware module (`rtl/core/neorv32_cpu_alu_cfu.vhd`) is tightly integrated into to CPU and coupled as ALU co-processor. A simple interface is used to communicate operands, operation definition and the processing result.

Table 79. CFU Interface

Signal	Direction	Width	Description
Global			
<code>clk_i</code>	in	1	CPU clock, registers trigger on a rising-edge
<code>rstn_i</code>	in	1	CPU reset, low-active, asynchronous
Request			
<code>start_i</code>	in	1	Trigger CFU operation; high for exactly one cycle
<code>inst_i</code>	in	32	CFU instruction word
<code>rs1_i</code>	in	32	Register operand 1 read from the CPU register file
<code>rs2_i</code>	in	32	Register operand 2 read from the CPU register file
Response			
<code>result_o</code>	out	1	Processing result that is written to the destination register (<code>rd</code>)
<code>valid_o</code>	out	32	High for exactly one cycle when the CFU operation is done and <code>result</code> is valid

CFU operations can be entirely combinatorial (e.g. for simple operation like bit-reversal) so the result is already valid at the end of the current clock cycle. CFU operations can also take several clock cycles to complete (like an iterative multiplications) and may also include internal states and memories. However, the CFU has to complete its operation within a **bound time window** (default = 512 clock cycles). Otherwise, 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 `valid` signal at all which will cause an execution timeout. For example this can be used to signal invalid configurations/operations to the runtime environment. See the documentation in the CFU's HDL source file for more information.

The following figure shows two timing sequences for CFU operations. The first one (orange) resembles a pure combinatorial operation that completes within the same cycle in which it was triggered. The second (blue) one performs a multi-cycle operation and is completed after several clock cycles.

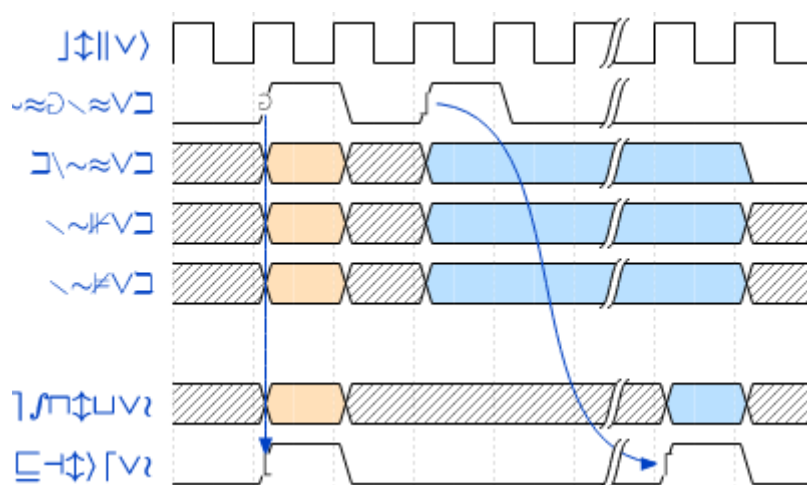


Figure 18. CFU Interface Timing Examples

3.7. Control and Status Registers (CSRs)

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



Unused, Reserved, Unimplemented and Disabled CSRs

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

Table 80. NEORV32 Control and Status Registers (CSRs)

Address	Name [ASM]	Name [C]	Access	Description
Floating-Point CSRs				
0x001	fflags	CSR_FFLAGS	URW	Floating-point accrued exceptions
0x002	frm	CSR_FRM	URW	Floating-point dynamic rounding mode
0x003	fcsr	CSR_FCSR	URW	Floating-point control and status
Machine Trap Setup CSRs				
0x300	mstatus	CSR_MSTATUS	MRW	Machine status register - low word
0x301	misa	CSR_MISA	MRW	Machine CPU ISA and extensions
0x304	mie	CSR_MIE	MRW	Machine interrupt enable register
0x305	mtvec	CSR_MTVEC	MRW	Machine trap-handler base address for ALL traps
0x306	mcounteren	CSR_MCOUNTEREN	MRW	Machine counter-enable register
0x310	mstatush	CSR_MSTATUSH	MRW	Machine status register - high word
Machine Configuration CSRs				
0x30a	menvcfg	CSR_MENVCFG	MRW	Machine environment configuration register - low word
0x31a	menvcfgh	CSR_MENVCFGH	MRW	Machine environment configuration register - high word
Machine Counter Setup CSRs				
0x320	mcountinhibit	CSR_MCOUNTINHIBIT	MRW	Machine counter-inhibit register

Address	Name [ASM]	Name [C]	Access	Description
0x321	<code>mcyclecfg</code>	<code>CSR_MCYCLECFG</code>	MRW	Machine cycle counter privilege mode filtering - low word
0x322	<code>minstretcfg</code>	<code>CSR_MINSTRETCFG</code>	MRW	Machine instret counter privilege mode filtering - low word
0x721	<code>mcyclecfgh</code>	<code>CSR_MCYCLECFGH</code>	MRW	Machine cycle counter privilege mode filtering - high word
0x722	<code>minstretcfgh</code>	<code>CSR_MINSTRETCFHH</code>	MRW	Machine instret counter privilege mode filtering - high word
Machine Trap Handling CSRs				
0x340	<code>mscratch</code>	<code>CSR_MSCRATCH</code>	MRW	Machine scratch register
0x341	<code>mepc</code>	<code>CSR_MEPC</code>	MRW	Machine exception program counter
0x342	<code>mcause</code>	<code>CSR_MCAUSE</code>	MRW	Machine trap cause
0x343	<code>mtval</code>	<code>CSR_MTVAL</code>	MRW	Machine trap value
0x344	<code>mip</code>	<code>CSR_MIP</code>	MRW	Machine interrupt pending register
Machine Physical Memory Protection CSRs				
0x3a0 .. 0x303	<code>pmpcfg0</code> .. <code>pmpcfg3</code>	<code>CSR_PMPCFG0</code> .. <code>CSR_PMPCFG3</code>	MRW	Physical memory protection configuration registers
0x3b0 .. 0x3bf	<code>pmpaddr0</code> .. <code>pmpaddr15</code>	<code>CSR_PMPADDR0</code> .. <code>CSR_PMPADDR15</code>	MRW	Physical memory protection address registers
Trigger Module CSRs				
0x7a0	<code>tselect</code>	<code>CSR_TSELECT</code>	MRW	Trigger select register
0x7a1	<code>tdata1</code>	<code>CSR_TDATA1</code>	MRW	Trigger data register 1
0x7a2	<code>tdata2</code>	<code>CSR_TDATA2</code>	MRW	Trigger data register 2
0x7a4	<code>tinfo</code>	<code>CSR_TINFO</code>	MRW	Trigger information register
CPU Debug Mode CSRs				
0x7b0	<code>dcsr</code>	<code>CSR_DCSR</code>	DRW	Debug control and status register
0x7b1	<code>dpc</code>	<code>CSR_DPC</code>	DRW	Debug program counter
0x7b2	<code>dscratch0</code>	<code>CSR_DSCRATCH0</code>	DRW	Debug scratch register 0
(Machine) Counter and Timer CSRs				
0xb00	<code>mcycle</code>	<code>CSR_MCYCLE</code>	MRW	Machine cycle counter low word
0xb02	<code>minstret</code>	<code>CSR_MINSTRET</code>	MRW	Machine instruction-retired counter low word
0xb80	<code>mcycleh</code>	<code>CSR_MCYCLEH</code>	MRW	Machine cycle counter high word

Address	Name [ASM]	Name [C]	Access	Description
0xb82	<code>minstreth</code>	<code>CSR_MINSTRETH</code>	MRW	Machine instruction-retired counter high word
0xc00	<code>cycle</code>	<code>CSR_CYCLE</code>	URO	Cycle counter low word
0xc01	<code>time</code>	<code>CSR_TIME</code>	URO	System time low word
0xc02	<code>instret</code>	<code>CSR_INSTRET</code>	URO	Instruction-retired counter low word
0xc80	<code>cycleh</code>	<code>CSR_CYCLEH</code>	URO	Cycle counter high word
0xc81	<code>timeh</code>	<code>CSR_TIMEH</code>	URO	System time high word
0xc82	<code>instreth</code>	<code>CSR_INSTRETH</code>	URO	Instruction-retired counter high word
Hardware Performance Monitors (HPM) CSRs				
0x323 .. 0x33f	<code>mhpmevent3 .. mhpmevent31</code>	<code>CSR_MHPMEVENT3 .. CSR_MHPMEVENT31</code>	MRW	Machine performance-monitoring event select
0xb03 .. 0xb1f	<code>mhpmcouter3 .. mhpmcouter31</code>	<code>CSR_MHPMCOUNTER3 .. CSR_MHPMCOUNTER31</code>	MRW	Machine performance-monitoring counter low word
0xb83 .. 0xb9f	<code>mhpmcouter3h .. mhpmcouter31h</code>	<code>CSR_MHPMCOUNTER3H .. CSR_MHPMCOUNTER31H</code>	MRW	Machine performance-monitoring counter high word
0xc03 .. 0xc1f	<code>hpmcounter3 .. hpmcounter31</code>	<code>CSR_HPMCounter3 .. CSR_HPMCounter31</code>	URO	Performance-monitoring counter low word
0xc83 .. 0xc9f	<code>hpmcounter3h .. hpmcounter31h</code>	<code>CSR_HPMCounter3H .. CSR_HPMCounter31H</code>	URO	Performance-monitoring counter high word
Machine Information CSRs				
0xf11	<code>mvendorid</code>	<code>CSR_MVENDORID</code>	MRO	Machine vendor ID
0xf12	<code>marchid</code>	<code>CSR_MARCHID</code>	MRO	Machine architecture ID
0xf13	<code>mimpid</code>	<code>CSR_MIMPID</code>	MRO	Machine implementation ID / version
0xf14	<code>mhartid</code>	<code>CSR_MHARTID</code>	MRO	Machine hardware thread ID
0xf15	<code>mconfigptr</code>	<code>CSR_MCONFIGPTR</code>	MRO	Machine configuration pointer register
NEORV32-Specific CSRs				
0xfc0	<code>mxisa</code>	<code>CSR_MXISA</code>	MRO	Extended machine CPU ISA and extensions low word
0xfc1	<code>mxisah</code>	<code>CSR_MXISAH</code>	MRO	Extended machine CPU ISA and extensions high word

3.7.1. Floating-Point CSRs

fflags

Name	Floating-point accrued exceptions
Address	0x001
Reset value	0x00000000
ISA	Zicsr & Zfinx
Description	FPU status flags.

Table 81. **fflags** CSR bits

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

frm

Name	Floating-point dynamic rounding mode
Address	0x002
Reset value	0x00000000
ISA	Zicsr & Zfinx
Description	The frm CSR is used to configure the rounding mode of the FPU.

Table 82. **frm** CSR bits

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

fcsr

Name	Floating-point control and status register
------	--

Address	0x003
Reset value	0x00000000
ISA	Zicsr & Zfinx
Description	The fcsr provides combined access to the fflags and frm flags.

Table 83. fcsr CSR bits

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

3.7.2. Machine Trap Setup CSRs

mstatus

Name	Machine status register - low word
Address	0x300
Reset value	0x00000000
ISA	Zicsr
Description	The mstatus CSR is used to configure general machine environment parameters.

Table 84. **mstatus** CSR bits

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

misalr

Name	ISA and extensions
Address	0x301
Reset value	DEFINED , according to enabled ISA extensions
ISA	Zicsr
Description	The misalr CSR provides information regarding the availability of basic RISC-V ISA extensions.



The NEORV32 **misalr** 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 85. **mis**a CSR bits

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



Machine-mode software can discover available **Z*** sub-extensions (like **Zicsr** or **Zfinx**) by checking the NEORV32-specific **mxisa[h]** CSRs.

mie

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

Table 86. **mie** CSR bits

Bit	Name [C]	R/W	Function
31:16	CSR_MIE_FIRQ15E : CSR_MIE_FIRQ0E	r/w	Fast interrupt channel 15..0 enable

Bit	Name [C]	R/W	Function
11	CSR_MIE_MEIE	r/w	MEIE : Machine <i>external</i> interrupt enable
7	CSR_MIE_MTIE	r/w	MTIE : Machine <i>timer</i> interrupt enable (from Core-Local Interruptor (CLINT))
3	CSR_MIE_MSIE	r/w	MSIE : Machine <i>software</i> interrupt enable (from Core-Local Interruptor (CLINT))

mtvec

Name	Machine trap-handler base address
Address	0x305
Reset value	0x00000000
ISA	Zicsr
Description	The mtvec CSR holds the trap vector configuration.

Table 87. **mtvec** CSR bits

Bit	R/W	Function
31:2	r/w	BASE : in DIRECT mode = 4-byte-aligned base address of trap base handler, <i>all</i> traps jump to pc = BASE ; in VECTORED mode = 128-byte-aligned base address of trap vector table, interrupts cause a jump to pc = BASE + 4 * mcause and exceptions a jump to pc = BASE .
1:0	r/w	MODE : mode configuration, 00 = DIRECT, 01 = VECTORED; other encodings are <i>reserved</i> .

mcounteren

Name	Machine counter enable
Address	0x306
Reset value	0x00000000
ISA	Zicsr & U
Description	The mcounteren CSR is used to constrain user-mode access to the CPU's counter CSRs.

Table 88. **mcounteren** CSR bits

Bit	Name [C]	R/W	Function
31:3	CSR_MCOUNTEREN_HPM31 : CSR_MCOUNTEREN_HPM3	r/w	HPMn : User-mode is allowed to read according hpmcounter[h] (31:3) CSRs when set
2	CSR_MCOUNTEREN_IR	r/w	IR : User-mode is allowed to read instret[h] CSRs when set
1	CSR_MCOUNTEREN_TM	r/w	TM : User-mode is allowed to read time[h] CSRs when set
0	CSR_MCOUNTEREN_CY	r/w	CY : User-mode is allowed to read cycle[h] CSRs when set

mstatush

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

3.7.3. Machine Trap Handling CSRs

mscratch

Name	Scratch register for machine trap handlers
Address	0x340
Reset value	0x00000000
ISA	Zicsr
Description	The mscratch is a general-purpose machine-mode scratch register.

mepc

Name	Machine exception program counter
Address	0x341
Reset value	0x00000000
ISA	Zicsr
Description	The mepc CSR provides the instruction address where execution has stopped/failed when an interrupt is triggered / an exception is raised. See section Traps, Exceptions and Interrupts for a list of all legal values. The mret instruction will return to the address stored in mepc by automatically moving mepc to the program counter.



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

mcause

Name	Machine trap cause
Address	0x342
Reset value	0x00000000
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 89. **mcause** CSR bits

Bit	R/W	Function
31	r/w	Interrupt: 1 if the trap is caused by an interrupt (0 if the trap is caused by an exception)
30:5	r/-	Hardwired to zero
4:0	r/w	Exception code: see NEORV32 Trap Listing

mtval

Name	Machine trap value
Address	0x343
Reset value	0x00000000
ISA	Zicsr
Description	The mtval CSR provides additional information why a trap was entered. See section Traps, Exceptions and Interrupts for more information.

mip

Name	Machine interrupt pending
Address	0x344
Reset value	0x00000000
ISA	Zicsr
Description	The mip CSR shows currently <i>pending</i> machine-mode interrupt requests. Any write access to this register is ignored.

Table 90. **mip** CSR bits

Bit	Name [C]	R/W	Function
31:16	CSR_MIP_FIRQ15P : CSR_MIP_FIRQ0P	r/-	FIRQxP : Fast interrupt channel 15..0 pending, see NEORV32-Specific Fast Interrupt Requests ; cleared by source-specific mechanism
11	CSR_MIP_MEIP	r/-	MEIP : Machine <i>external</i> interrupt pending, triggered by mti_i top port (see CPU Top Entity - Signals) or by the processor-internal Core-Local Interruptor (CLINT) ; cleared by source-specific mechanism

Bit	Name [C]	R/W	Function
7	CSR_MIP_MTIP	r/-	MTIP : Machine <i>timer</i> interrupt pending, triggered by mei_i top port (see CPU Top Entity - Signals) or by the processor-internal Core-Local Interruptor (CLINT) ; cleared by source-specific mechanism
3	CSR_MIP_MSIP	r/-	MSIP : Machine <i>software</i> interrupt pending, triggered by msi_i top port (see CPU Top Entity - Signals); cleared by source-specific mechanism



FIRQ Channel Mapping

See section **NEORV32-Specific Fast Interrupt Requests** for the mapping of the FIRQ channels and the according interrupt-triggering processor module.

3.7.4. Machine Configuration CSRs

menvcfg

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

menvcfgh

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

3.7.5. 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** (but below PMP configuration/address register 16) **no illegal instruction exception** is triggered. The according CSRs are read-only (writes are ignored) and always return zero. See section **Smpmp ISA Extension** for more information.

pmpcfg

Name	Physical memory protection region configuration registers
Address	0x3a0 (pmpcfg0) 0x3a1 (pmpcfg1) 0x3a2 (pmpcfg2) 0x3a3 (pmpcfg3)
Reset value	all 0x00000000
ISA	Zicsr & Smpmp
Description	Configuration of physical memory protection regions. Each region provides an individual 8-bit array in these CSRs. Note that, depending on PMP_NUM_REGIONS , a maximum of the lowest 4 configuration registers are physically implemented (pmpcfg0 .. pmpcfg3). All remaining register are hardwired to zero.

Table 91. **pmpcfg*** CSR Bits

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



Implemented Modes

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

pmpaddr

Name	Physical memory protection region address registers
Address	0x3b0 (pmpaddr1) 0x3b1 (pmpaddr2) 0x3b2 (pmpaddr3) 0x3b3 (pmpaddr4) 0x3b4 (pmpaddr5) 0x3b5 (pmpaddr6) 0x3b6 (pmpaddr6) 0x3b7 (pmpaddr7) 0x3b8 (pmpaddr8) 0x3b9 (pmpaddr9) 0x3ba (pmpaddr10) 0x3bb (pmpaddr11) 0x3bc (pmpaddr12) 0x3bd (pmpaddr13) 0x3be (pmpaddr14) 0x3bf (pmpaddr15)
Reset value	all 0x00000000
ISA	Zicsr & Smpmp
Description	Region address/boundaries configuration (address/mask bits 33:2). Note that, depending on PMP_NUM_REGIONS , a maximum of the lowest 16 address registers are physically implemented (pmpaddr0 .. pmpaddr15). All remaining register are hardwired to zero.

Table 92. **pmpaddr*** CSR Bits

Bit	R/W	Description
29:0	r/w	address bits 31 downto 2
31:30	r-w	address bits 33 downto 32' , hardwired to zero

3.7.6. (Machine) Counter and Timer CSRs

cycle[h]

Name	Cycle counter
Address	0xc00 (cycle) 0xc80 (cycleh)
Reset value	0x00000000_00000000
ISA	Zicsr & Zicntr
Description	The cycle[h] CSRs are user-mode shadow copies of the according mcycle[h] CSRs. The user-mode counter are read-only. Any write access will raise an illegal instruction exception.

time[h]

Name	System time counter
Address	0xc01 (time) 0xc81 (timeh)
Reset value	0x00000000_00000000
ISA	Zicsr & Zicntr
Description	The time[h] CSRs are user-mode shadow copies the memory-mapped machine timer ("MTIMER) of the Core-Local Interruptor (CLINT) . These CSR are read-only. Any write access will raise an illegal instruction exception.

instret[h]

Name	Instructions-retired counter
Address	0xc02 (instret) 0xc82 (instreth)
Reset value	0x00000000_00000000
ISA	Zicsr & Zicntr
Description	The instret[h] CSRs are user-mode shadow copies of the according minstret[h] CSRs. The user-mode 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 value	0x00000000_00000000
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 value	0x00000000_00000000
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

*Instruction Retiring*

According to the RISC-V specification, instructions that trigger an asynchronous exception or are interrupted by a debug mode or an interrupt request are not considered "retired". Hence, these events are not counted by **[m]instret[h]**.

3.7.7. 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 29 HPM counter and configuration registers (**mhpcounter*[h]** and **mhpmevent***) are accessible, 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 reads 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	0x323 ... 0x33f (mhpmevent3 ... mhpmevent31)
Reset value	all 0x00000000
ISA	Zicsr & Zihpm
Description	The value in these CSRs define the micro-architectural events that cause an increment of the according mhpcounter*[h] counter(s). All available events are listed in the table below. If more than one event is enabled, 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.

Table 93. **mhpmevent*** CSR Bits (Micro-Architectural Counter Events)

Bit	Name [C]	R/W	Event Description
8	HPMCNT_EVENT_STORE	r/w	memory store operation (read-modify-write AMOs are counted as 1x load + 1x store operation)
7	HPMCNT_EVENT_LOAD	r/w	memory load operation (read-modify-write AMOs are counted as 1x load + 1x store operation)
6	HPMCNT_EVENT_DELTA	r/w	control flow transfer (jump, taken branch, trap entry/exit)
5	HPMCNT_EVENT_WAIT_LSU	r/w	memory/bus/cache/etc. delay/wait cycle while executing any load or store operation (caused by a data bus wait cycle))
4	HPMCNT_EVENT_WAIT_ALU	r/w	delay/wait cycle caused by a <i>multi-cycle</i> CPU Arithmetic Logic Unit operation (e.g. bit-serial shift)
3	HPMCNT_EVENT_WAIT_DIS	r/w	instruction dispatch wait cycle (wait for instruction prefetch-buffer refill); caused by a fence instruction, a control flow transfer or a instruction fetch bus wait cycle (e.g. cache miss)
2	HPMCNT_EVENT_CI	r/w	retired compressed instruction (C ISA Extension)
1	HPMCNT_EVENT_IR	r/w	retired instruction (16-bit/compressed or 32-bit/uncompressed)

Bit	Name [C]	R/W	Event Description
0	HPMCNT_EVENT_CY	r/w	active clock cycle (CPU not in Sleep Mode)



Atomic Memory Access

The read-modify-write instructions of the **Zaamo ISA Extension** operate as simple load for the CPU hardware. Hence, they will only trigger **HPMCNT_EVENT_LOAD** and only once.

mhpmcounter[h]

Name	Machine hardware performance monitor (HPM) counter
Address	0xb03 ... 0xb1f (mhpmcounter3 ... mhpmcounter31) 0xb83 ... 0xb9f (mhpmcounter3h ... mhpmcounter31h)
Reset value	all 0x00000000
ISA	Zicsr & Zihpm
Description	If not halted via the mcounthinhibit CSR the HPM counter CSRs increment whenever the configured events from the according mhpmevent CSR occur. The HPM counter registers are read/write for machine mode software and are not accessible for lower-privileged software.

hpmcounter[h]

Name	Hardware performance monitor (HPM) counter
Address	0xc03 ... 0xc1f (hpmcounter3 ... hpmcounter31) 0xc83 ... 0xc9f (hpmcounter3h ... hpmcounter31h)
Reset value	all 0x00000000
ISA	Zicsr & Zihpm
Description	The hpmcounter[h] CSRs are user-mode shadow copies of the according mhpmcounter[h] CSRs. The user-mode counters are read-only. Any write access will raise an illegal instruction exception.

3.7.8. 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.

Table 94. **mcountinhibit** CSR Bits

Bit	Name [C]	R/W	Description
15:3	CSR_MCOUNTINHIBIT_HP3 : CSR_MCOUNTINHIBIT_HP15	r/w	HPMx : Set to 1 to halt [m]hpmcount*[h] ; hardwired to zero if Zihpm ISA extension is disabled
2	CSR_MCOUNTINHIBIT_CY	r/w	CY : Set to 1 to halt [m]cycle[h] ; hardwired to zero if Zicntr ISA extension is disabled
1	-	r/-	TM : Hardwired to zero as time[h] CSRs are not implemented
0	CSR_MCOUNTINHIBIT_IR	r/w	IR : Set to 1 to halt [m]instret[h] ; hardwired to zero if Zicntr ISA extension is disabled

mcyclecfg[h]

Name	Machine cycle counter privilege mode filtering
Address	0x321 (mcyclecfg) 0x721 (mcyclecfgh)
Reset value	all 0x00000000
ISA	Zicsr & Zicntr & Smcntrpmf
Description	Halt cycle counter when CPU operates in a specific privilege mode. Note that mcyclecfg is hardwired to all-zero.

Table 95. **mcyclecfg** CSR Bits

Bit	Name [C]	R/W	Description
30	CSR_MCYCLECFGH_MINH	r/w	MINH : Set to 1 to halt [m]cycle[h] counter when CPU is in machine-mode

Bit	Name [C]	R/W	Description
28	CSR_MCYCLECFGH_UINH	r/w	UINH : Set to 1 to halt [m]cycle[h] counter when CPU is in user-mode

minstretcfg[h]

Name	Machine instret counter privilege mode filtering		
Address	0x322 (minstretcfg) 0x722 (minstretcfgh)		
Reset value	all 0x00000000		
ISA	Zicsr & Zicntr & Smcntrpmf		
Description	Halt instruction-retired counter when CPU operates in a specific privilege mode. Note that minstretcfg is hardwired to all-zero.		

Table 96. **minstretcfgh** CSR Bits

Bit	Name [C]	R/W	Description
30	CSR_MINSTRETCFGH_MINH	r/w	MINH : Set to 1 to halt [m]instret[h] counter when CPU is in machine-mode
28	CSR_MINSTRETCFGH_UINH	r/w	UINH : Set to 1 to halt [m]instret[h] counter when CPU is in user-mode

3.7.9. Machine Information CSRs

mvendorid

Name	Machine vendor ID
Address	0xf11
Reset value	0x00000000
ISA	Zicsr
Description	This register is hardwired to the value provided by the OCD_JEDEC_ID Processor Top Entity - Generics (the upper bits are filled with zeros).

Table 97. **mvendorid** CSR Bits

Bits	R/W	Description
31:7	r/-	JEDEC ID bank
6:0	r/-	JEDEC ID offset

marchid

Name	Machine architecture ID
Address	0xf12
Reset value	0x00000013
ISA	Zicsr
Description	The marchid CSR is read-only and provides the NEORV32 official RISC-V open-source architecture ID (decimal: 19, 32-bit hexadecimal: 0x00000013).

mimpid

Name	Machine implementation ID
Address	0xf13
Reset value	DEFINED
ISA	Zicsr
Description	The mimpid CSR is read-only and provides the version of the NEORV32 as BCD-coded number (example: mimpid = 0x01020312 → 01.02.03.12 → version 1.2.3.12).

mhartid

Name	Machine hardware thread ID
Address	0xf14
Reset value	DEFINED
ISA	Zicsr
Description	The mhartid CSR is read-only and provides the core's hart ID. In a multi-core system, the hard ID of each core is unique and starts at 0 and is incremented continuously.

mconfigptr

Name	Machine configuration pointer register
Address	0xf15
Reset value	0x00000000
ISA	Zicsr
Description	The features of this CSR are not implemented yet. The register is read-only and always returns zero.

3.7.10. NEORV32-Specific CSRs



RISC-V-Compliant Mapping

NEORV32-specific CSRs are mapped to addresses that are explicitly reserved for custom/implementation-specific use.

mxisa[h]

Name	Machine extended ISA and extensions register
Address	0xfc0 (mxisa , low word) 0xfc1 (mxisah , high word)
Reset value	DEFINED
ISA	Zicsr & X
Description	The mxisa[h] CSRs are NEORV32-specific read-only CSRs that help machine-mode software to discover additional ISA (sub-)extensions (beyond the <i>base</i> extensions listed in misa).

Table 98. **mxisa** CSR Bits

Bit	Name [C]	R/W	Description
31	CSR_MXISA_SMCNTRPMF	r/-	Smcntrpmf - ISA Extension available
30	CSR_MXISA_ZIMOP	r/-	Zimop ISA Extension available
29	CSR_MXISA_ZIBI	r/-	Zibi ISA Extension available
28	CSR_MXISA_ZCA	r/-	"`C` without floating-point", available when C ISA Extension is available
27	CSR_MXISA_ZCB	r/-	Zcb ISA Extension available
26	CSR_MXISA_ZALRSC	r/-	Zalrsc ISA Extension available
25	CSR_MXISA_ZAAMO	r/-	Zaamo ISA Extension available
24	CSR_MXISA_ZBS	r/-	Zbs ISA Extension available
23	CSR_MXISA_ZBB	r/-	Zbb ISA Extension available
22	CSR_MXISA_ZBA	r/-	Zba ISA Extension available
21	CSR_MXISA_ZKS	r/-	Zks ISA Extension available
20	CSR_MXISA_ZKSED	r/-	Zksed ISA Extension available
19	CSR_MXISA_ZKSH	r/-	Zksh ISA Extension available
18	CSR_MXISA_ZKN	r/-	Zkn ISA Extension available
17	CSR_MXISA_ZBKC	r/-	Zbkc ISA Extension available

Bit	Name [C]	R/W	Description
16	CSR_MXISA_ZBKB	r/-	Zbkb ISA Extension available
15	CSR_MXISA_ZKNH	r/-	Zknh ISA Extension available
14	CSR_MXISA_ZKNE	r/-	Zkne ISA Extension available
13	CSR_MXISA_ZKND	r/-	Zknd ISA Extension available
12	CSR_MXISA_ZBKX	r/-	Zbkx ISA Extension available
11	CSR_MXISA_SDTRIG	r/-	Sdtrig ISA Extension available
10	CSR_MXISA_SDEXT	r/-	Sdext ISA Extension available
9	CSR_MXISA_ZIHPM	r/-	Zihpm ISA Extension available
8	CSR_MXISA_SMPMP	r/-	Smpmp ISA Extension available
7	CSR_MXISA_ZICNTR	r/-	Zicntr ISA Extension available
6	CSR_MXISA_ZICOND	r/-	Zicond ISA Extension available
5	CSR_MXISA_ZFINX	r/-	Zfinx ISA Extension available
4	CSR_MXISA_ZKT	r/-	Zkt ISA Extension available
3	CSR_MXISA_XCFU	r/-	Xcfu ISA Extension available
2	CSR_MXISA_ZMMUL	r/-	Zmmul - ISA Extension available
1	CSR_MXISA_ZIFENCEI	r/-	Zifencei ISA Extension available
0	CSR_MXISA_ZICSR	r/-	Zicsr ISA Extension available

Table 99. *mxisah* CSR Bits

Bit	Name [C]	R/W	Description
31:2	-	r/-	reserved, read as zero
1	CSR_MXISAH_ZCMOP	r/-	Zcmop ISA Extension available
0	CSR_MXISAH_ZBC	r/-	Zbc ISA Extension available

3.8. Traps, Exceptions and Interrupts

In this document the following terminology is used (derived from the RISC-V trace specification available at <https://github.com/riscv-non-isa/riscv-trace-spec>):

- **exception**: an unusual condition occurring at run time associated (i.e. *synchronous*) with an instruction in a RISC-V hart
- **interrupt**: an external *asynchronous* event that may cause a RISC-V hart to experience an unexpected transfer of control
- **trap**: the transfer of control to a trap handler caused by either an *exception* or an *interrupt*

Whenever an exception or interrupt is triggered, the CPU switches to machine-mode (if not already in machine-mode) and continues operation at the address being stored in the **mtvec** CSR. The cause of the trap can be determined via the **mcause** CSR. A list of all implemented **mcause** values and the according description can be found below in section **NEORV32 Trap Listing**. The address that reflects the current program counter when a trap was taken is stored to **mepc** CSR. Additional information regarding the cause of the trap can be retrieved from the **mtval** CSR.

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

All CPU interrupt request signals are **high-active**. Once triggered, the interrupt request line should stay high until it is explicitly acknowledged by a source-specific mechanism (for example by writing to a specific memory-mapped register).



Instruction Atomicity and Forward-Progress

All instructions execute as atomic operations - interrupts can only trigger *between* consecutive instructions. Additionally, if there is a permanent interrupt request, exactly one instruction from the interrupted program will be executed before another interrupt handler can start. This allows program progress even if there are permanent interrupt requests.

3.8.1. Memory Access Exceptions

If a load operation causes any exception, the instruction's destination register is **not written** at all. Furthermore, exceptions caused by a misaligned memory address a physical memory protection

fault do not trigger a memory access request at all.

For 32-bit-only instructions (= no **C** extension) the misaligned instruction exception is raised if bit 1 of the fetch address is set (i.e. not on a 32-bit boundary). If the **C** extension is implemented there will **never** be a misaligned instruction exception at all.

3.8.2. Nested Interrupts

The CPU supports nested interrupt handling in software. It automatically disables interrupts upon entering any trap handler by clearing **mstatus.MIE**. Otherwise, further interrupts during the critical part of the handler (i.e. before saving the context: **mcause**, **mepc**, stack frame, etc.) would cause a context loss. However, software can explicitly enable interrupts again by manually setting **mstatus.MIE** from within the handler to allow another interrupt to trigger.

3.8.3. 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.

3.8.4. 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.



FIRQ Mapping

See section **NEORV32-Specific Fast Interrupt Requests** for the mapping of the FIRQ channels to the according hardware modules.

Table Annotations

The "RTE Trap ID" aliases are defined by the NEORV32 core library and can be used in plain C code when interacting with the pre-defined RTE functions. The **mcause**, **mepc** and **mtval** columns show the value being written to the according CSRs when a trap is encountered:

- **I-PC** - address of intercepted instruction (instruction at this address has not been executed yet and must be executed after the trap handler to maintain program flow)
- **PC** - address of instruction that caused the trap (instruction has been executed)
- **ADDR** - memory access address that caused the trap
- **INST** - the actual instruction word that caused the trap
- **0** - zero

Table 100. NEORV32 Trap Listing

Priority	mcause	RTE Trap ID	Cause	mepc	mtval
Exceptions (<i>synchronous</i> to instruction execution)					
highest	0x00000001	TRAP_CODE_I_ACCESS	illegal access fault	PC	0
	0x00000002	TRAP_CODE_I_ILLEGAL	illegal instruction	PC	INST
	0x00000000	TRAP_CODE_I_MISALIGNED	instruction address misaligned	PC	ADDR
	0x0000000b	TRAP_CODE_MENV_CALL	environment call from M-mode	PC	0
	0x00000008	TRAP_CODE_UENV_CALL	environment call from U-mode	PC	0
	0x00000003	TRAP_CODE_BREAKPOINT	software breakpoint / HW trigger	PC	0
	0x00000006	TRAP_CODE_S_MISALIGNED	store address misaligned	PC	ADDR
	0x00000004	TRAP_CODE_L_MISALIGNED	load address misaligned	PC	ADDR
	0x00000007	TRAP_CODE_S_ACCESS	store access fault	PC	ADDR
	0x00000005	TRAP_CODE_L_ACCESS	load access fault	PC	ADDR
Interrupts (<i>asynchronous</i> to instruction execution)					
	0x80000010	TRAP_CODE_FIRQ_0	fast interrupt request channel 0	I-PC	0
	0x80000011	TRAP_CODE_FIRQ_1	fast interrupt request channel 1	I-PC	0
	0x80000012	TRAP_CODE_FIRQ_2	fast interrupt request channel 2	I-PC	0
	0x80000013	TRAP_CODE_FIRQ_3	fast interrupt request channel 3	I-PC	0
	0x80000014	TRAP_CODE_FIRQ_4	fast interrupt request channel 4	I-PC	0
	0x80000015	TRAP_CODE_FIRQ_5	fast interrupt request channel 5	I-PC	0
	0x80000016	TRAP_CODE_FIRQ_6	fast interrupt request channel 6	I-PC	0
	0x80000017	TRAP_CODE_FIRQ_7	fast interrupt request channel 7	I-PC	0
	0x80000018	TRAP_CODE_FIRQ_8	fast interrupt request channel 8	I-PC	0
	0x80000019	TRAP_CODE_FIRQ_9	fast interrupt request channel 9	I-PC	0
	0x8000001a	TRAP_CODE_FIRQ_10	fast interrupt request channel 10	I-PC	0
	0x8000001b	TRAP_CODE_FIRQ_11	fast interrupt request channel 11	I-PC	0
	0x8000001c	TRAP_CODE_FIRQ_12	fast interrupt request channel 12	I-PC	0
	0x8000001d	TRAP_CODE_FIRQ_13	fast interrupt request channel 13	I-PC	0
	0x8000001e	TRAP_CODE_FIRQ_14	fast interrupt request channel 14	I-PC	0
	0x8000001f	TRAP_CODE_FIRQ_15	fast interrupt request channel 15	I-PC	0

Priority	mcause	RTE Trap ID	Cause	mepc	mtval
	0x8000000b	TRAP_CODE_MEI	machine external interrupt (MEI)	I-PC	0
	0x80000003	TRAP_CODE_MSI	machine software interrupt (MSI)	I-PC	0
lowest	0x80000007	TRAP_CODE_MTI	machine timer interrupt (MTI)	I-PC	0

Table 101. NEORV32 Trap Description

Trap ID [C]	Triggered when ...
TRAP_CODE_I_ACCESS	bus timeout, bus access error or PMP rule violation during instruction fetch
TRAP_CODE_I_ILLEGAL	trying to execute an invalid instruction word (malformed or not supported) or on a privilege violation
TRAP_CODE_I_MISALIGNED	fetching a 32-bit instruction word that is not 32-bit-aligned (see note below)
TRAP_CODE_MENV_CALL	executing ecall instruction in machine-mode
TRAP_CODE_UENV_CALL	executing ecall instruction in user-mode
TRAP_CODE_BREAKPOINT	executing ebreak instruction or if Trigger Module fires
TRAP_CODE_S_MISALIGNED	storing data to an address that is not naturally aligned to the data size (half/word)
TRAP_CODE_L_MISALIGNED	loading data from an address that is not naturally aligned to the data size (half/word)
TRAP_CODE_S_ACCESS	bus timeout, bus access error or PMP rule violation during store data operation
TRAP_CODE_L_ACCESS	bus timeout, bus access error or PMP rule violation during load data operation
TRAP_CODE_FIRQ_*	caused by interrupt-condition of processor-internal modules , see NEORV32-Specific Fast Interrupt Requests
TRAP_CODE_MEI	machine external interrupt (via dedicated Processor Top Entity - Signals)
TRAP_CODE_MSI	machine software interrupt (internal Core-Local Interruptor (CLINT) or via dedicated Processor Top Entity - Signals)
TRAP_CODE_MTI	machine timer interrupt (internal Core-Local Interruptor (CLINT) or via dedicated Processor Top Entity - Signals)

*Resumable Exceptions*

Note that not all exceptions are resumable. The "instruction access fault" and the "instruction address misaligned" exceptions are not resumable in most cases.

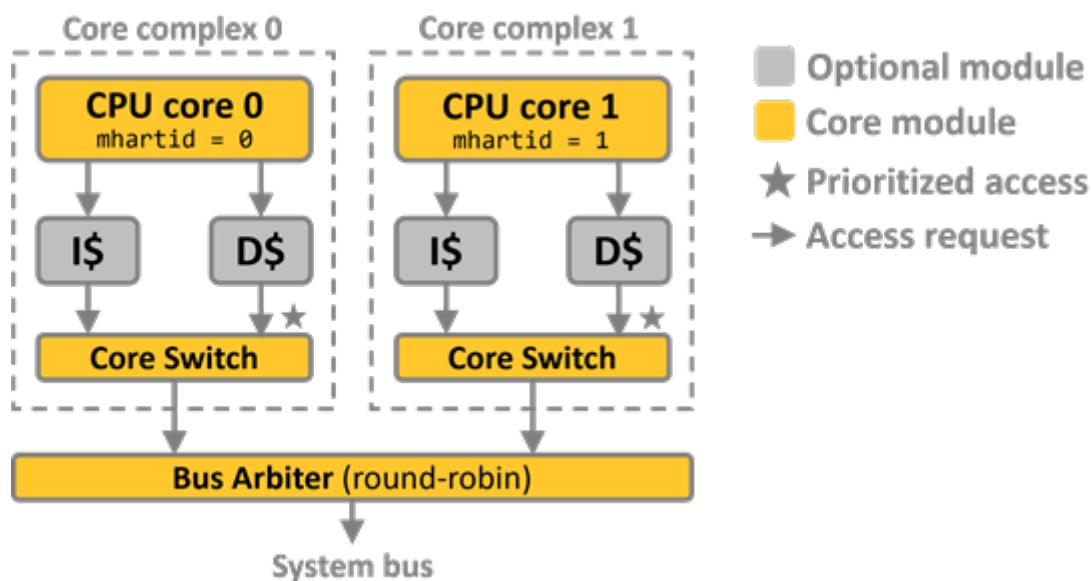
3.9. Dual-Core Configuration



Dual-Core Example Programs

A set of rather simple dual-core example programs can be found in [sw/example/demo_dual_core*](#).

Optionally, the CPU core can be implemented as **symmetric multiprocessing (SMP) dual-core** system. This dual-core configuration is enabled by the **DUAL_CORE_EN** top generic. When enabled, two *core complexes* are implemented. Each core complex consists of a CPU core and optional instruction (I\$) and data (D\$) caches. Similar to the single-core **Bus System**, the instruction and data interfaces are switched into a single bus interface by a prioritizing bus switch. The bus interfaces of both core complexes are further switched into a single system bus using a round-robin arbiter.



Both CPU cores are fully identical and use the same ISA, tuning and cache configurations provided by the according top generics. However, each core can be identified by the according "hart ID" that can be retrieved from the **mhartid** CSR. CPU core 0 (the *primary* core) has **mhartid = 0** while core 1 (the *secondary* core) has **mhartid = 1**.

The following table summarizes the most important aspects when using the dual-core configuration.

CPU configuration	Both cores use the same cache, CPU and ISA configuration provided by the according top generics.
Debugging	A special SMP openOCD script (sw/openocd/openocd_neorv32.dual_core.cfg) is required to debug both cores at once. SMP-debugging is fully supported by the RISC-V gdb port.
Clock and reset	Both cores use the same global processor clock and reset.
Address space	Both cores have full access to the same physical Address Space .

Interrupts	All Processor Interrupts are routed to both cores. Hence, each core has access to all NEORV32-Specific Fast Interrupt Requests (FIRQs). Additionally, the RISC-V machine-level <i>external interrupt</i> (via the top <code>irq_mei_i</code> port) is also sent to both cores. In contrast, the RISC-V machine level <i>software</i> and <i>timer</i> interrupts are core-exclusive (provided by the Core-Local Interruptor (CLINT)).
RTE	The NEORV32 Runtime Environment can be used for both cores. However, the RTE needs to be explicitly initialized on each core (executing <code>neorv32_rte_setup()</code>). Note that the installed trap handlers apply to both cores. The installed user-defined trap handlers can check the according core's ID via the <code>mhartid</code> CSR to perform core-specific trap handling.
Memory	Each core has its own stack. The top of stack of core 0 is defined by the Linker Script while the top of stack of core 1 has to be explicitly defined by core 0 (see Dual-Core Boot). Both cores share the same heap, <code>.data</code> and <code>.bss</code> sections. Hence, only core 0 setups the <code>.data</code> and <code>.bss</code> sections at boot-up.
Constructors and destructors	Constructors and destructors are supported by core 0 only (see section C Standard Library).
Cache layout	If enabled, each CPU core has its own data and/or instruction cache.
Cache coherency	Be aware that there is no cache snooping available. If any CPU1 cache is enabled care must be taken to prevent access to outdated data - either by using cache synchronization (<code>fence</code> / <code>fence.i</code> instructions) or by using atomic memory accesses. See Memory Coherence for more information.
Bootloader	Only core 0 will boot and execute the bootloader while core 1 is held in standby.
Booting	See section Dual-Core Boot .

3.9.1. Dual-Core Boot

After reset, both cores start booting at the same reset vector using the same executable. However, core 1 will - regardless of the **Boot Configuration** - always enter **Sleep Mode** right inside the default **Start-Up Code (crt0)**. The primary core (core 0) will continue booting, executing either the **Bootloader** or the pre-installed image from the internal instruction memory (depending on the boot configuration). To explicitly boot-up core 1, the primary core has to use a special library function provided by the NEORV32 software framework:

Listing 21. CPU Core 1 Launch Function Prototype (note that this function can only be executed on core 0)

```
int neorv32_smp_launch(int (*entry_point)(void), uint8_t* stack_memory, size_t
stack_size_bytes);
```



Core 1 Boot-Up

Core 0 will ensure that core 1 only uses start-up code from the shared executable. See <https://github.com/stnolting/neorv32/pull/1450>.

Core 0 uses the two 32-bit **MTIMECMP** registers of the **Core-Local Interruptor (CLINT)** to submit a *launch configuration* to core 1. This launch configuration consists of the stack layout (via the **stack_memory** and **stack_size_bytes** arguments) and the actual entry point for core 1. When these registers have been populated, core 0 will trigger the software interrupt of core 1 (also via the CLINT) to wake it from sleep mode. After that, core 1 will fetch the launch configuration and will start execution at the configured entry point.

Listing 22. CPU Core 1 Main Function

```
int core1_main(void) { // return `int`, no arguments
    return 0; // return to crt0 and go to sleep mode
}
```

Core 1 Stack Memory



The memory for the stack of core 1 (**stack_memory**) can be either statically allocated (i.e. a global volatile memory array; placed in the **.data** or **.bss** section of core 0) or dynamically allocated (using **malloc**; placed on the heap of core 0). In any case the memory should be aligned to a 16-byte boundary.

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**
- **System View Description File (SVD)**
- **Application Makefile**
- **Default Compiler Flags**
- **Linker Script**
- **C Standard Library**
- **Start-Up Code (crt0)**
- **Executable Image Formats**
- **NEORV32 Runtime Environment**
- **Bootloader**



Software Documentation

All core libraries and example programs are documented *in-code* using **Doxygen**. The API documentation is automatically built and deployed to GitHub pages: <https://stnolting.github.io/neorv32/sw/files.html>.



Example Programs

A collection of annotated example programs illustrating how to use certain NEORV32 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 in the official RISC-V GNU toolchain GitHub repository: <https://github.com/riscv-collab/riscv-gnu-toolchain>.



Toolchain Installation

More information regarding the toolchain (building from scratch or downloading prebuilt ones) can be found in the user guide section **Software Toolchain Setup**.

4.2. Core Libraries

The NEORV32 project provides a set of pre-defined C libraries that allow an easy integration of the processor/CPU features (also called "HAL" - *hardware abstraction layer*). All driver and runtime-related files are located in [sw/lib](#). These library files are automatically included and linked by adding the following include statement:

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

The NEORV32 HAL consists of the following files.

Table 102. NEORV32 Hardware Abstraction Layer File List

C source file	C header file	Description
-	neorv32.h	Main NEORV32 library file
neorv32_aux.c	neorv32_aux.h	General auxiliary/helper function
neorv32_cfs.c	neorv32_cfs.h	Custom Functions Subsystem (CFS) HAL
neorv32_clint.c	neorv32_clint.h	Core-Local Interruptor (CLINT) HAL
neorv32_cpu.c	neorv32_cpu.h	NEORV32 Central Processing Unit (CPU) HAL
	neorv32_csr.h	Control and Status Registers (CSRs) definitions
neorv32_dma.c	neorv32_dma.h	Direct Memory Access Controller (DMA) HAL
neorv32_gpio.c	neorv32_gpio.h	General Purpose Input and Output Port (GPIO) HAL
neorv32_gptmr.c	neorv32_gptmr.h	General Purpose Timer (GPTMR) HAL
-	neorv32_intrinsics.h	Macros for intrinsics and custom instructions*)
neorv32_neoled.c	neorv32_neoled.h	Smart LED Interface (NEOLED) HAL
neorv32_owewire.c	neorv32_owewire.h	One-Wire Serial Interface Controller (ONEWIRE) HAL
neorv32_pwm.c	neorv32_pwm.h	Pulse-Width Modulation Controller (PWM) HAL
neorv32_rte.c	neorv32_rte.h	NEORV32 Runtime Environment
neorv32_sdi.c	neorv32_sdi.h	Serial Data Interface Controller (SDI) HAL
neorv32_slink.c	neorv32_slink.h	Stream Link Interface (SLINK) HAL
neorv32_smp.c	neorv32_smp.h	HAL for the SMP Dual-Core Configuration
neorv32_spi.c	neorv32_spi.h	Serial Peripheral Interface Controller (SPI) HAL
	neorv32_sysinfo.h	System Configuration Information Memory (SYSINFO) HAL
neorv32_tracer.c	neorv32_tracer.h	Execution Trace Buffer (TRACER) HAL

C source file	C header file	Description
<code>neorv32_trng.c</code>	<code>neorv32_trng.h</code>	True Random-Number Generator (TRNG) HAL
<code>neorv32_twd.c</code>	<code>neorv32_twd.h</code>	Two-Wire Serial Device Controller (TWD) HAL
<code>neorv32_twi.c</code>	<code>neorv32_twi.h</code>	Two-Wire Serial Interface Controller (TWI) HAL
<code>neorv32_uart.c</code>	<code>neorv32_uart.h</code>	Primary Universal Asynchronous Receiver and Transmitter (UART0) and UART1 HAL
<code>neorv32_wdt.c</code>	<code>neorv32_wdt.h</code>	Watchdog Timer (WDT) HAL
<code>neorv32_newlib.c</code>	-	Platform-specific system calls for <i>newlib</i>



Doxygen API Documentation

The Doxygen-based documentation of all core libraries is available online at <https://stnolting.github.io/neorv32/sw/files.html>.

4.3. System View Description File (SVD)

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

4.4. Application Makefile

Application compilation is based on a centralized GNU makefile (`sw/common/common.mk`). Each software project (for example the ones in `sw/example` folder) should provide a local makefile that just includes the central makefile:

```
# Set path to NEORV32 root directory
NEORV32_HOME ?= ../../..
# Include the main NEORV32 makefile
include $(NEORV32_HOME)/sw/common/common.mk
```

Thus, the functionality of the central makefile (including all targets) becomes available for the project. A project-local makefile should be used to define all setup-relevant configuration options instead of changing the central makefile to keep the code base clean. Setting variables in the project-local makefile will override the default configuration. Most example projects already provide a makefile that list all relevant configuration options.

The following example shows the configuration of a local Makefile:

```
# Override the default CPU ISA
MARCH = rv32imc_zicsr_zifencei

# Override the default RISC-V GCC prefix
RISCV_PREFIX ?= riscv-none-elf-

# Override default optimization goal
EFFORT = -Os

# Add extended debug symbols for Eclipse
USER_FLAGS += -ggdb -gdwarf-3

# Additional sources
APP_SRC += $(wildcard ./*.c)
APP_INC += -I .

# Adjust processor IMEM size and base address
USER_FLAGS += -Wl,--defsym,__neorv32_rom_size=16k
USER_FLAGS += -Wl,--defsym,__neorv32_rom_base=0x00000000

# Adjust processor DMEM size and base address
USER_FLAGS += -Wl,--defsym,__neorv32_ram_size=8k
USER_FLAGS += -Wl,--defsym,__neorv32_ram_base=0x80000000

# Adjust maximum heap size
USER_FLAGS += -Wl,--defsym,__neorv32_heap_size=2k

# Set path to NEORV32 root directory
```

```
NEORV32_HOME ?= ../../..
```

```
# Include the main NEORV32 makefile
include $(NEORV32_HOME)/sw/common/common.mk
```

4.4.1. Makefile Targets

Invoking a project-local makefile (executing **make** or **make help**) will show the help menu that lists all available targets as well as all variable including their *current* setting.

```
neorv32/sw/example/hello_world$ make
NEORV32 Software Makefile
Find more information at https://github.com/stnolting/neorv32
Use 'make V=1' or set BUILD_VERBOSE to increase build verbosity

Targets:

help      show this text
check     check toolchain and list supported ISA extensions
info      show project/makefile configuration
gdb       start GNU debugging session
asm       build and generate <main.asm> assembly listing file
elf       build and generate <main.elf> ELF file
exe       build and generate <neorv32_exe.bin> executable file for bootloader
upload
bin       build and generate <neorv32_raw_exe.bin> executable memory image
hex       build and generate <neorv32_raw_exe.hex> executable memory image
coe       build and generate <neorv32_raw_exe.coe> executable memory image
mem       build and generate <neorv32_raw_exe.mem> executable memory image
mif       build and generate <neorv32_raw_exe.mif> executable memory image
image    build and generate VHDL IMEM application memory image
<neorv32_imem_image.vhd> in local folder
install  build, generate and install VHDL IMEM application memory image
<neorv32_imem_image.vhd>
clean     clean up project home folder
clean_all clean up project home folder and image generator
all       elf + asm + exe + bin + coe + mem + mif + image + install
```

Additional targets:

```
sim       in-console simulation using default testbench (sim folder) and GHDL
hdl_list  regenerate HDL file-list (*.f) in NEORV32_HOME/rtl
upload    upload executable to bootloader via UART (/dev/ttyUSB1)
elf_info  show ELF information
elf_sections show ELF sections
bl_image  build and generate VHDL BOOTROM bootloader memory image
<neorv32_bootrom_image.vhd> in local folder
bootloader build, generate and install VHDL BOOTROM bootloader memory image
```

<neorv32_bootrom_image.vhd>

Variables:

```

BUILD_VERBOSE    Set to increase build verbosity: "0"
USER_FLAGS       Custom toolchain flags [append only]: "-ggdb -gdwarf-3 -Wl,
--defsym,__neorv32_rom_size=16k -Wl,--defsym,__neorv32_rom_base=0x00002000 -Wl,
--defsym,__neorv32_ram_size=8k"
USER_LIBS        Custom libraries [append only]: ""
EFFORT           Optimization level: "-Os"
MARCH            Machine architecture: "rv32i_zicsr_zifencei"
MABI             Machine binary interface: "ilp32"
APP_INC          C include folder(s) [append only]: "-I ."
APP_SRC          C source folder(s) [append only]: "./main.c  "
APP_OBJ          Object file(s) [append only]: ""
ASM_INC          ASM include folder(s) [append only]: "-I ."
RISCV_PREFIX     Toolchain prefix: "riscv-none-elf-"
NEORV32_HOME     NEORV32 home folder: "../.."
GDB_ARGS         GDB arguments: "-ex target extended-remote localhost:3333"
UART_TTY         Serial port for upload to bootloader: "/dev/ttyUSB1"
GHDL_RUN_FLAGS   GHDL simulation run arguments: ""

```

Build Artifacts

All *intermediate* build artifacts (like object files and binaries) will be places into a (new) project-local folder named **build**. The *resulting* build artifacts (like executable, the main ELF and all memory initialization/image files) will be placed in the root project folder.

*Increase Verbosity*

Use **make V=1** or set **BUILD_VERBOSE** in your environment to increase build verbosity.

*Compile Flags as C-String*

The makefile defines a C string **CC_FLAGS_STR** that contains all flags passed to the compiler. This string can be processed at runtime: **printf("CC_FLAGS: %s\n", CC_FLAGS_STR);**

4.4.2. Default Compiler Flags

The central makefile uses specific compiler flags to tune the code to the NEORV32 hardware. Hence, these flags should not be altered. However, experienced users can modify them to further tune compilation.

Table 103. Compiler Options (CC_OPTS)

-Wall, -Wextra	Enable all/extra compiler warnings.
-----------------------	-------------------------------------

<code>-ffunction-sections</code>	Put functions in independent sections. This allows a code optimization as dead code can be easily removed.
<code>-fdata-sections</code>	Put data segment in independent sections. This allows a code optimization as unused data can be easily removed.
<code>-mno-fdiv</code>	Use built-in software functions for floating-point divisions and square roots (since the according instructions are not supported yet).
<code>-mstrict-align</code>	Unaligned memory accesses cannot be resolved by the hardware and require emulation.
<code>-mbranch-cost=10</code>	Branching is expensive.
<code>-ffp-contract=off</code>	Disable floating-point expression contraction (fused multiply-add/sub; not supported by the NEORV32 FPU).
<code>-g</code>	Add (simple) debug information.

Table 104. Linker Libraries (`LD_LIBS`)

<code>-lm</code>	Include/link with <code>math.h</code> .
<code>-lc</code>	Search for the standard C library when linking.
<code>-lgcc</code>	Make sure we have no unresolved references to internal GCC library subroutines.
<code>-nostartfiles</code>	Do not use the default start code. Instead, the NEORV32-specific start-up code (<code>sw/common/crt0.S</code>) is used (pulled-in by the linker script).
<code>-Wl,--gc-sections</code>	Make the linker perform dead code elimination.

Advanced Debug Symbols

By default, only "simple" symbols are added to the ELF (`-g`). Extended debug flags (e.g. for Eclipse) can be added using the `USER_FLAGS` variable (e.g. `USER_FLAGS += -ggdb -gdwarf-3`). Note that other debug flags may be required depending of the GCC/GDB version

4.5. Linker Script

The NEORV32-specific linker script (`sw/common/neorv32.ld`) is used to link the compiled sources according to the processor's **Address Space**). For the final executable, only two memory segments are required:

Table 105. Linker script - Memory Segments

Memory section	Description
<code>rom</code>	Instruction memory address space (processor-internal Instruction Memory (IMEM) and/or external memory)
<code>ram</code>	Data memory address space (processor-internal Data Memory (DMEM) and/or external memory)

These two sections are configured by several variables defined in the linker script and exposed to the build framework (aka the makefile). Those variable allow to customized the RAM/ROM sizes and base addresses. Additionally, a certain amount of the RAM can be reserved for the software-managed heap (see **RAM Layout**).

Table 106. Linker script - Configuration

Memory section	Description	Default
<code>__neorv32_rom_size</code>	"ROM" size (instruction memory / IMEM)	16kB
<code>__neorv32_ram_size</code>	"RAM" size (data memory / DMEM)	8kB
<code>__neorv32_rom_base</code>	"ROM" base address (instruction memory / IMEM)	0x00000000
<code>__neorv32_ram_base</code>	"RAM" base address (data memory / DMEM)	0x80000000
<code>__neorv32_heap_size</code>	Maximum heap size; part of the "RAM"	0kB

Each variable provides a default value (e.g. "16K" for the instruction memory /ROM /IMEM size). These defaults can be overridden by setup-specific values to take the user-defined processor configuration into account (e.g. a different IMEM size). The `USER_FLAGS` variable provided by the **Application Makefile** can also be used to customize the memory configuration. For example, the following line can be added to a project-specific local makefile to adjust the memory sizes:

Listing 23. Overriding Default Memory Sizes (configuring 64kB IMEM and 32kB DMEM)

```
USER_FLAGS += "-Wl,--defsym,__neorv32_rom_size=64k -Wl,
--defsym,__neorv32_ram_size=32k"
```



Memory Configuration Constraints

Memory sizes have to be a multiple of 4 bytes. Memory base addresses have to be 32-bit-aligned.

4.5.1. RAM Layout

The default NEORV32 linker script uses the defined RAM (**DMEM**) size to map several sections. Note that depending on the application some sections might have zero size.

Listing 24. Default RAM Layout

```
<End of RAM>
+-----+
|           Stack           |
+-----+
.   stack grows downwards   .
.                             .
.                             .
.   heap grows upwards      .
+-----+
| Dynamic memory (".heap")  |
+-----+
| Uninitialized data (".bss")|
+-----+
| Initialized data (".data")|
+-----+
<Start of RAM>
```

1. **Initialized data (".data")**: The data section is placed right at the beginning of the RAM. For example, this contains *explicitly initialized* global variables which are initialized by the **Start-Up Code (crt0)**.
2. **Non-initialized data (".bss")**: This dynamic data section contains data like global variables *without explicit initialization*. This memory section is cleared by the **Start-Up Code (crt0)**.
3. **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.
4. **Stack**: The stack starts at the end of the RAM at the last 16-byte aligned address. According to the RISC-V ABI / calling convention the stack is 128-bit-aligned before procedure entry. The stack grows downwards.

Heap Size



The maximum size of the heap is defined by the `__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).

Heap-Stack Collision



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. The **physical memory protection** extension can be used to implement a guarding mechanism.

4.5.2. ROM Layout

The default NEORV32 linker script uses the defined ROM (**IMEM**) size to map several sections into an executable layout. Note that depending on the application some sections might have zero size.

Listing 25. Default ROM Layout

```
<End of ROM>
+-----+
| Initial data for ".data" |
+-----+
| Read-only data ".rodata" |
+-----+
| Program code ".text" |
+-----+
<Start of ROM>
```

1. **Program code** (**.text**): This section contains the actual instructions of the program.
2. **Read-only data** (**.rodata**): Read-only data, for example for constants and strings.
3. **Heap** (**.heap**): Initialization data that is copied at runtime to the RAM's **data** section by the **Start-Up Code** (**crt0**).



Binary Executable Image

The above sections are concatenated by the **Application Makefile** to create a standalone and portable executable. This can be exported in various **Executable Image Formats**, e.g. for transfer via the **Bootloader** or to initialize read-only memory.

4.6. C Standard Library

The default software framework relies on **newlib** as default C standard library. Newlib provides hooks for common "system calls" (like file handling and standard input/output) that are used by other C libraries like **stdio**. These hooks are available in [sw/lib/source/newlib.c](#) and were adapted for the NEORV32 processor.



Standard Input/Output Streams / Consoles

The **UART0** is used to implement all the standard input, output and error consoles (**STDIN** = file number 0, **STDOUT** = file number 1, **STDERR** = file number 2). All other input/output streams (other file number than 0,1,2) are redirected to **UART1**.



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. Note that constructor and destructors are only executed by core 0 (primary core) in the SMP **Dual-Core Configuration**.



Newlib Test/Demo Program

A simple test and demo program that uses some of newlib's system functions (like **malloc/free** and **read/write**) is available in [sw/example/demo_newlib](#).



Executing System Functions on a Host Computer

The NEORV32 on-chip debugger supports **Semihosting** which allow to use the input/output facilities on a host computer.

4.7. Start-Up Code (crt0)

The CPU and also the processor require a minimal start-up and initialization code to bring the hardware into an operational state. Furthermore, the C runtime requires an initialization before compiled code can be executed. This setup is done by the start-up code (`sw/common/crt0.S`) which is automatically linked with every application program and gets mapped before the actual application code so it gets executed right after boot. The `crt0.S` start-up performs the following operations:

Table 107. CTR0 operations executed by each CPU core

CPU 0 (primary core)	CPU 1 (secondary core)
setup the stack pointer (<code>sp</code>) and the global pointer (<code>gp</code>) according to the RAM Layout	
initialize <code>mstatus</code> disabling machine-level interrupts	
install an endless loop as trap handler ("crt0 panic") to <code>mtvec</code> : the core will halt if any trap occurs	
clear <code>mie</code> disabling all interrupt sources	
initialize all integer register <code>x1..x31</code> (only <code>x1..x15</code> if E ISA extension is enabled)	
initialize <code>.data</code> section	wait in SMP sleep mode for Dual-Core Boot .
clear <code>.bss</code> section	
call constructors	
call core0's <code>main</code> function	call core1's <code>main</code> function
<i>execute application</i>	
<code>main</code> returns	
clear <code>mie</code> disabling all interrupt sources	
reinstall an endless loop as trap handler ("crt0 panic") to <code>mtvec</code> .	
<code>main</code> return value is copied to <code>mscratch</code> for inspection by a debugger	
call destructors	
execute <code>ebreak</code> instruction to notify external debugger	
halt: CPU enters sleep mode executing <code>wfi</code> in an endless loop	
CPU 0 (primary core)	CPU 1 (secondary core)

4.8. Executable Image Formats

The **Application Makefile** generates a flattened binrary from the final ELF file, which is further processed by the NEORV32 executable image generator (**sw/image_gen**). This utility tool can generate various types of standalone and portable executable file formats, which are used, for example, for uploading via the **Bootloader** or for initializing SoC ROM modules.

Listing 26. Invoking the Image Generator

```
neorv32/sw/image_gen$ ./image_gen
NEORV32 executable image generator

Usage:    image_gen [options]
Example:  image_gen -i elf.bin -o main_exe.bin -b A0000000 -t exe

Options:
  -h                Show this help text and exit
  -i file_name      Flattened binary input file
  -o file_name      Image output file
  -t format         Image output format
  -b address        Bootloader exe base address (hex; for "-t exe" only)

Image formats (using little-Endian byte ordering):
  exe  Executable for bootloader upload (binary file with header)
  vhd  VHDL memory image (raw executable)
  bin  Binary file (raw executable)
  hex  Plain hexdump file (raw executable)
  coe  COE file (8x hex per line, ASCII, raw executable)
  mem  MEM file (8x hex per line, ASCII, raw executable)
  mif  MIF file (8x hex per line, ASCII, raw executable)
```

The image generator can be used as a stand-alone program, but it is also integrated into the application makefile. This provides corresponding **Makefile Targets** for the different image types:

Table 108. Image Generation Targets

exe	Generates an executable binary file (including a header; see Bootloader Image Format) for upload via the Bootloader .
vhd	Generates a raw VHDL memory image (package file, package name is the same as the output file name).
bin	Generates a raw binary file for custom purpose.
hex	Generates a raw HEX file for custom purpose (like memory initialization via <code>\$readmemh()</code> / <code>hread()</code>).
coe	Generates a raw COE file for FPGA memory initialization.
mem	Generates a raw MEM file for FPGA memory initialization.

mif

Generates a raw MIF file for FPGA memory initialization.

*Image Generator Compilation*

The sources of the image generator are automatically compiled when invoking the according makefile targets. Note that compiling the image generator requires a *native* C compiler.

*VHDL Memory Initialization Files*

The generated VHDL images are plain VHDL packages providing program data as constant array. Note that this array is always extended to a power of 2. Unused elements are set to zero.

4.9. Bootloader

The NEORV32 bootloader provides an optional built-in firmware that allows to upload new application firmware at any time without the need to re-synthesize the FPGA's bitstream. The bootloader is automatically executed after reset when enabled via the **Boot Configuration** `BOOT_MODE_SELECT` generic. The bootloader provides the following key features:

- interactive user console via UART
- upload executable via UART
- load executable from SPI flash; store executable to SPI flash
- load executable from TWI flash; store executable to TWI flash (*option disabled by default*)
- load executable from SD card (*option disabled by default*)
- automatic boot sequence: automatically boot from TWI/SPI flash / SD card
- highly configurable to adapt to application requirements



Pre-Built Bootloader Image

This section refers to the **default** NEORV32 bootloader. The **Bootloader ROM (BOOTROM)** image file already contains the pre-compiled bootloader (`sw/bootloader/bootloader.c`) in its default configuration.



Minimal Hardware Requirements

The default bootloader image was compiled for a minimal `rv32e_zicsr_zifencei` ISA configuration and requires a RAM (DMEM) size of at least 256 bytes. These constraints ensure that the bootloader can be executed on any CPU/processor configuration. It is recommended to enable at least **UART0**, the **CLINT** and the **GPIO** controller. See **Customizing the Internal Bootloader** for more information.

4.9.1. Bootloader Console

The default bootloader provides a serial console via UART0 for user interaction using the following terminal settings:

- 19200 Baud, 8 data bits, no parity bit, 1 stop bit (`19200-8-N-1`)
- line breaks: carriage return + newline (`\r\n`)



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). Note that some terminal programs struggle with transmitting files larger than 4kB (see <https://github.com/stnolting/neorv32/pull/215>). **For Windows I can recommend TeraTerm (GUI) and SimplySerial (Window terminal). For Linux I can recommend picocom.**

By default the bootloader uses the LSB of the GPIO output port (`gpio_o(0)`) for a high-active status LED. If the GPIO's direction control feature is implemented, only pin 0 is configured as output; the remaining pins are configured as inputs. All other output pins are set to low. After reset, the status LED will start blinking at 2Hz and the **Auto Boot Sequence** is started. This auto-boot sequence can be skipped within 8s by pressing any key.

Listing 27. Default Bootloader Console

```
NEORV32 Bootloader
build: Jul 18 2026 ①

Auto-boot in 8s. Press any key to abort. ②
Aborted. ③

Type 'h' for help.
CMD:> ④
```

- ① Bootloader version (compile date).
- ② Start of auto-boot sequence.
- ③ Auto-boot sequence aborted due to user console input.
- ④ Command prompt.

To see a list of all available commands press `h` to see the help menu:

Listing 28. Bootloader Help Menu

```
CMD:> h
Available CMDs:
h: Help          ①
i: System info   ②
r: Restart       ③
u: Upload via UART ④
t: TWI flash - load ⑤
w: TWI flash - program ⑥
l: SPI flash - load ⑦
s: SPI flash - program ⑧
c: SD card - load ⑨
e: Start executable ⑩
x: Exit          ⑪
CMD:>
```

- ① Show "Available CMDs" help text again.
- ② Show hardware configuration information.
- ③ Restart bootloader and auto-boot sequence.
- ④ Upload new executable (`neorv32_exe.bin`) via UART.

- ⑤ Load executable from TWI flash (*this option is disabled by default*).
- ⑥ Store executable to TWI flash (*this option is disabled by default*).
- ⑦ Load executable from SPI flash.
- ⑧ Program previously-uploaded executable to SPI flash.
- ⑨ Load executable from SD card (*this option is disabled by default*).
- ⑩ Start the (up)loaded executable.
- ⑪ Raise a breakpoint exception: If a debugger is connected this will transfer control to the debugger. If no debugger is connected this will print an exception error (**Bootloader Error Codes**) shutting down the processor.



Unavailable Commands

Note that not all options are enabled in the default bootloader configuration in order to keep the bootloader ROM size below 4kB. However, these options can be enabled in `config.h` (see **Customizing the Internal Bootloader**).

Available commands can be executed by typing the according letter. For example, when executing the `i` command the general system configuration is printed:

Listing 29. System Configuration Information (Example)

```
CMD:> i
HWV: 0x01130207 ①
CLK: 0x05F5E100 ②
MISA: 0x40901104 ③
XISA: 0x00000000_0x10000F93 ④
SOC: 0x18BF80FF ⑤
MISC: 0x59011010 ⑥
CMD:>
```

- ① Processor hardware version in BCD format (**mimpid** CSR).
- ② Processor clock speed in Hz (**CLK** register of **System Configuration Information Memory (SYSINFO)**).
- ③ RISC-V CPU extensions (**misa** CSR).
- ④ Additional RISC-V CPU (sub-)extensions (high & low words of **mxisa[h]** CSRs).
- ⑤ Processor configuration (**SOC** register of **System Configuration Information Memory (SYSINFO)**).
- ⑥ Miscellaneous memory and SoC configuration (**MISC** register of **System Configuration Information Memory (SYSINFO)**).

4.9.2. Bootloader Image Format

In order to keep the bootloader as small as possible, complete ELF parsing was omitted. Therefore, a compiled ELF cannot be loaded directly into the processor via the bootloader. Instead, the image

generator (see **Executable Image Formats**) is used to create a simplified executable file for the bootloader that contains all the necessary information from the ELF and can be processed by the bootloader with little overhead.

After the executable has been compiled, the image generator extracts the relevant sections and combines them into an image that corresponds to the **ROM Layout** of the processor configuration. This image is preceded by a simple header containing information for transmission as well as for relocation and booting. The header consists of four individual 32-bit words, which, like the actual image, are stored in low-endian byte order.

Listing 30. Bootloader Executable Header

```
// bootloader executable header, little-Endian
typedef struct __attribute__((packed,aligned(4))) {
    uint32_t signature; // = 0x214F454E ("NEO!")
    uint32_t base_addr; // executable base address & entry point
    uint32_t size;      // size in bytes
    uint32_t checksum;  // sum-complement checksum
} exe_header_t;
```

The first word represents the signature (**0x214F454E**, "NEO!" as string representation). Based on this, the bootloader can recognize a valid executable that also matches the bootloader version.

The next word defines the base address of the image. The bootloader stores the received image starting at this address. This allows the bootloader to store different programs at different memory addresses, for example. The base address is ultimately defined by the **Linker Script** using the `__neorv32_rom_base` symbol. This also corresponds to the entry point of the program that is executed during bootup.

The next word is the size of the image in bytes. The size of the generated executable is always a multiple of 4.

The last word represents a checksum that ensures the transfer and integrity of the executable. To keep the bootloader simple, a simple "sum complement" checksum is used here.

Listing 31. Exemplary Header

```
neorv32/sw/example/processor_check$ xxd -e -l 32 neorv32_exe.bin
00000000: 214f454e 00000000 00007078 2af7f9c6   NEO!....xp.....*
```

4.9.3. Auto Boot Sequence

After reset, the bootloader waits 8 seconds for UART console input before it starts the automatic boot sequence. Depending on the configuration (**Customizing the Internal Bootloader**) the bootloader will try to fetch a valid executable from different sources:

1. **TWI boot:** Try to load an executable from TWI flash (default device ID is **0xA0**); if enabled.

2. **SPI boot:** Try to load an executable from SPI flash (default SPI chip select line is `spi_csn_o(0)`); if enabled.
3. **SD boot:** Try to load file `boot.bin` from SD flash (default SPI chip select line is `spi_csn_o(1)`); if enabled.
4. **Direct boot:** Directly execute the code at a pre-configured address; if enabled.

If a valid boot image is loaded it will be immediately started. If no valid executable can be fetched the interactive bootloader console is started.

4.9.4. Uploading an Executable

1. Connect the primary UART (UART0) interface of the processor to a serial port of your host computer.
2. Start a serial terminal program.
3. Open a connection to the the serial port your UART is connected to.
4. Press the NEORV32 reset button to restart the bootloader. The status LED starts blinking and the bootloader intro screen appears in the console. Press any key to abort the automatic boot sequence and to start the actual bootloader user interface console.
5. Execute the "Upload" command by typing `u`. Now the bootloader is waiting for a binary executable to be send: `Awaiting neorv32_exe.bin...`
6. Use the "send file" option of your terminal program to send a valid NEORV32 executable (`neorv32_exe.bin`). Make sure the terminal sends the executable in raw binary mode.
7. If everything went fine, `Awaiting neorv32_exe.bin... OK` is printed in the terminal.
8. The executable is now in the instruction memory of the processor. To execute the program right now run the "start executable" command by typing `e`.



Uploading via `make upload` right from the console

The **Application Makefile** provides an "upload" target that allows you to transfer a compiled executable directly to the bootloader via UART right from the console. This script handles the communication with the bootloader and is based on a simple shell helper script `neorv32/sw/image_gen/uart_upload.sh`. Example: `neorv32/sw/example/demo_blink_led$ make UART_TTY=/dev/ttyUSB1 upload`. The serial interface device is defined by `UART_TTY`.



UART Overflow Detection and Hardware Flow Control

The bootloader **does not check for UART RX overflows** during transmissions. However, such an error during transmission will most likely result in a checksum error, so the error will certainly be detected anyway. For setups that use very high custom baud rates values and/or provide very high main memory latency (e.g. external serial RAM), it is recommended to enable the UART RTS/CTS hardware flow control. See **Customizing the Internal Bootloader** for more information.

4.9.5. Programming an SPI or TWI Flash

This guide shows how to write an executable to the SPI flash via the bootloader so it can be automatically fetched and executed after processor reset. If the TWI flash option is enabled, an according command for programming the TWI flash is available.

1. Reset the NEORV32 processor and wait until the bootloader start screen appears.
2. Abort the auto boot sequence and start the user console by pressing any key.
3. Press **u** to upload the executable that shall be programmed to the flash.
4. Send the binary via the terminal program. When the upload is completed and "OK" appears, press **s** to trigger the SPI flash programming:

```
CMD:> u
Awaiting neorv32_exe.bin... OK
CMD:> s
Write 0x00001614 bytes to SPI flash @0x00400000 (y/n)?
```

5. The bootloader shows the size of the executable and the base address of the SPI flash where the executable will be stored. A prompt appears: type **y** to start the programming or type **n** to abort.

```
CMD:> u
Awaiting neorv32_exe.bin... OK
CMD:> s
Write 0x00001614 bytes to SPI flash @0x00400000 (y/n)?
Flashing... OK
CMD:>
```

6. Note that flash programming can take some time (depending on the TWI/SPI clock configuration in **config.h**). If "OK" appears, the programming process was successful and the flash can be used for the auto-boot sequence.

4.9.6. Booting from SD Card

The SD card is accessed in SPI mode. The card has to be formatted using the **FAT32** file system with a sector size of at least 512 bytes. The executable has to be placed in the card's root directory. By default the bootloader will fetch the **boot.bin** file, which is just a renamed copy of the default **neorv32_exe.bin** file that is generated by the application compilation flow. The name of the SD card's boot file can be changed to a custom file name (see **Customizing the Internal Bootloader**), but it has to use the 8.3 DOS format (max 8 character for the name plus dot plus 3 characters suffix).

SD-card and FAT32 support is provided by the great **Petit FatFs** library by Elm-Chan: https://elm-chan.org/fsw/ff/00index_p.html

4.9.7. Customizing the Internal Bootloader

The NEORV32 bootloader provides several options to configure it for a custom setup. It configured via set of C-language **defines** in `sw/bootloader/config.h`. All defines provide default value that can be edited or overridden by Makefile directives.

Table 109. Bootloader Configuration Parameters

Parameter	Default	Legal values	Description
Additional includes			
-	-	-	Add further #include statements right at the beginning of the file.
Serial console - requires Primary Universal Asynchronous Receiver and Transmitter (UART0)			
UART_EN	1	0,1	Set to 1 to enable the serial console.
UART_BAUD	19200	any	Baud rate of UART0.
UART_HWFC	0	0,1	Enable RTS/CTS hardware flow control. Recommended for very high baud rates / high memory latency.
Status LED - requires General Purpose Input and Output Port (GPIO)			
STATUS_LED_EN	1	0,1	Enable bootloader status led ("heart beat") at GPIO output port pin STATUS_LED_PIN when 1.
STATUS_LED_PIN	0	0..31	GPIO output pin used for the high-active status LED. Note that if the GPIO module's direction control feature is enabled only this pin is configured as output.
Serial memory - requires Serial Memory Controller (SMC)			
SMC_EN	0	0,1	Enable initialization of Serial Memory Controller (SMC) .
SMC_SETUP_ARGS	...	...	List of configuration arguments for SMC setup function (<code>neorv32_smc_setup(...)</code>).
Auto-boot - requires Core-Local Interruptor (CLINT)			
AUTO_BOOT_EN	1	0,1	Auto-boot enabled when 1.
AUTO_BOOT_TIMEOUT	8	any	Timeout in seconds after which the auto-boot sequence starts (if there is no UART input by the user).
Direct-boot			
DIRECT_BOOT_EN	0	0,1	Enable direct-boot option when 1.
DIRECT_BOOT_ADDR	0x00000000	32-bit	Direct boot address of executable; can be used to directly execute from pre-initialized memory.

Parameter	Default	Legal values	Description
TWI flash - requires Two-Wire Serial Interface Controller (TWI)			
TWI_FLASH_EN	0	0, 1	Set to 1 to enable the TWI flash (boot) options.
TWI_FLASH_PROG_EN	1	0, 1	Set to 1 to enable programming the TWI flash from the bootloader menu.
TWI_FLASH_CLK_PRSC	CLK_PRSC_C_1024	CLK_PRSC_2 CLK_PRSC_4 CLK_PRSC_8 CLK_PRSC_64 CLK_PRSC_128 CLK_PRSC_1024 CLK_PRSC_2024 CLK_PRSC_4096	TWI clock prescaler.
TWI_FLASH_CLK_DIV	0	0..15	TWI clock divider value.
TWI_FLASH_ID	0xA0	any	8-bit TWI device address (with R/W bit cleared).
TWI_FLASH_BASE_ADDR	0x00000000	32-bit	Defines the TWI flash base address for the executable.
TWI_FLASH_ADDR_BYTES	2	1, 2, 3, 4	Number of TWI flash address bytes.
SPI flash - requires Serial Peripheral Interface Controller (SPI)			
SPI_FLASH_EN	1	0, 1	Set to 1 to enable the SPI flash (boot) options.
SPI_FLASH_PROG_EN	1	0, 1	Set to 1 to enable programming the SPI flash from the bootloader menu.
SPI_FLASH_CS	0	0..7	SPI chip select line (port <code>spi_csn_o</code>) for selecting the SPI flash.
SPI_FLASH_CLK_PRSC	CLK_PRSC_C_64	CLK_PRSC_2 CLK_PRSC_4 CLK_PRSC_8 CLK_PRSC_64 CLK_PRSC_128 CLK_PRSC_1024 CLK_PRSC_2024 CLK_PRSC_4096	SPI clock prescaler.
SPI_FLASH_CLK_DIV	0	0..15	SPI clock divider value.
SPI_FLASH_BASE_ADDR	0x00400000	32-bit	Defines the SPI flash base address for the executable.
SPI_FLASH_ADDR_BYTES	3	1, 2, 3, 4	SPI flash address size in number of bytes.
SPI_FLASH_SECTOR_SIZE	16*1024	any	Number of SPI flash address bytes.

Parameter	Default	Legal values	Description
SPI SD card - requires Serial Peripheral Interface Controller (SPI)			
<code>SPI_SDCARD_EN</code>	0	0, 1	Set to 1 to enable booting from SD card.
<code>SPI_SDCARD_CS</code>	1	0..7	SPI chip select line (port <code>spi_csn_o</code>) for selecting the SD card.
<code>SPI_SDCARD_CLK_P</code> <code>RSC</code>	<code>CLK_PRSC_64</code>	<code>CLK_PRSC_2</code> <code>CLK_PRSC_4</code> <code>CLK_PRSC_8</code> <code>CLK_PRSC_64</code> <code>CLK_PRSC_128</code> <code>CLK_PRSC_1024</code> <code>CLK_PRSC_2024</code> <code>CLK_PRSC_4096</code>	SPI clock prescaler.
<code>SPI_SDCARD_CLK_DIV</code>	0	0..15	SPI clock divider value.
<code>SPI_SDCARD_FILE</code>	"boot.bin"	8.3 DOS format	File name of the boot image. Has to be located in the root directory.
Branding			
<code>THEME_INTRO</code>	"NEORV32 Bootloader"	any string	Intro text that is shown in the bootloader console.
<code>THEME_EXE</code>	"neorv32_exe.bin"	any string	Name of executable that is shown in the upload menu.
Custom Code			
<code>USER_CODE_INIT</code>	(void)0 (empty statement)	any code	User-defined initialization code (wrapped in a macro) that is executed right after the bootloader's hardware initialization.

4.9.8. Bootloader Error Codes

ERROR_DEVICE	A device-accessing function returned an error code. Make sure that the device is properly connected and that all required processor modules/interface are actually enabled (by the according Processor Top Entity - Generics). This error can also occur if there was an RX overflow on the UART (if overflow detection is enabled). In this case, either UART hardware flow control should be enabled or the baud rate should be reduced; both adjustments require recompiling the bootloader.
---------------------	---

ERROR_SIGNATURE	The signature that indicates a valid NEORV32 executable of the accessed executable is incorrect. This can be caused by a temporary transmission error or by an invalid or corrupted executable.
ERROR_CHECKSUM	The checksum of the loaded executable is incorrect. This can be caused by a temporary transmission error or by an invalid or corrupted executable.
ERROR_EXCEPTION	An unexpected exception has occurred. This can be caused by an invalid bootloader configuration (non-available processor modules, memory layout, ...). For debugging purpose the error message will also display the content of the mcause , mepc and mtval CSRs. Example: ERROR_EXCEPTION 0x00000003 0xffe00cb4 0x00000000

4.10. NEORV32 Runtime Environment

The NEORV32 software framework provides a minimal runtime environment ("RTE") that takes care of a stable and *safe* execution environment by providing a unified interface for handling of *all* traps (exceptions and interrupts). Once initialized, the RTE provides **Default RTE Trap Handlers** that catch all possible traps. These default handlers just output a message via UART 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** for bare-metal / non-OS applications. 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.

4.10.1. RTE Operation

The RTE manages the trap-related CSRs of the CPU's privileged architecture (see **Machine Trap Handling CSRs**). It initializes the **mtvec** CSR in DIRECT mode, which provides the base entry point for *all* traps. The address stored to this register defines the address of the **first-level trap handler**, which is provided by the NEORV32 RTE. Whenever an exception or interrupt is triggered this first-level trap handler is executed.

The first-level handler performs a complete context save, analyzes the source of the trap and calls the according **second-level trap handler**, which takes care of the actual exception/interrupt handling. The RTE manages an internal look-up table to track 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 debugging message via the primary UART (UART0) (if enabled) to inform the user that a trap has occurred that is not (yet) handled by a proper application-specific trap handler. After sending this message, the RTE tries to resume normal execution by moving on to the next linear instruction.



Dual-Core Configuration

The RTE's internal trap handler look-up table is used globally for **both** cores. If a core-specific handling is required, the according user-defined trap handler need to retrieve the core's ID from **mhartid** and branch accordingly.

4.10.2. Using the RTE

The NEORV32 runtime environment is part of the default NEORV32 software framework. The links to the according software references are listed below.

```
neorv32_ Online software reference \(Doxygen\)
rte.c
neorv32_ Online software reference \(Doxygen\)
rte.h
```

The RTE has to be explicitly enabled by calling the according setup function. It is recommended to do this right at the beginning of the application's `main` function. For the SMP [Dual-Core Configuration](#) the RTE setup functions has to be called on each core that wants to use the RTE.

Listing 32. RTE Setup Right at the Beginning of "main"

```
int main() {

    neorv32_rte_setup(); // setup NEORV32 runtime environment

    ...
}
```

After setup, all traps will trigger execution of the RTE's [Default RTE Trap Handlers](#) at first. In order to use application-specific trap handlers the default debug handlers can be overridden by installing user-defined ones:

Listing 33. 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`. However, more convenient device-specific aliases are also defined in `sw/lib/include/neorv32.h`. The second argument `handler` is the actual function that implements the user-defined trap handler. The custom handler functions must have a specific type without any arguments and with no return value:

Listing 34. 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 would place an `mret` instruction at the end of the handler making it impossible to return to the first-level trap handler of the RTE core.



`mscratch` CSR

The **mscratch** CSR should not be used inside an application trap handler as this register is used by the RTE to provide the base address of the application's stack frame **Application Context Handling** (i.e. modifying the registers of application code that caused a trap).

The following example shows how to install trap handlers for exemplary traps.

Listing 35. Installing Custom Trap Handlers Examples

```
neorv32_rte_handler_install(TRAP_CODE_MTI, machine_timer_irq_handler); // handler for
machine timer interrupt
neorv32_rte_handler_install(TRAP_CODE_MENV_CALL, environment_call_handler); // handler
for machine environment call exception
neorv32_rte_handler_install(SLINK_TRAP_CODE, slink_rx_handler); // handler for SLINK
receive interrupt
```

4.10.3. 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 (exemplary RTE outputs are shown below). 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**).

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 unhandled **trap** like a bus access fault exception, continuing execution will most likely fail making the CPU crash.

Listing 36. RTE Default Trap Handler UART0 Output Examples

```
<NEORV32-RTE-PANIC> [cpu0|M] Illegal instruction MEPC=0x000002d6 MTVAL=0x000000FF
</NEORV32-RTE-PANIC> ①
<NEORV32-RTE-PANIC> [cpu0|U] Illegal instruction MEPC=0x00000302 MTVAL=0x00000000
</NEORV32-RTE-PANIC> ②
<NEORV32-RTE-PANIC> [cpu0|U] Load address misaligned MEPC=0x00000440 MTVAL=0x80000101
</NEORV32-RTE-PANIC> ③
<NEORV32-RTE-PANIC> [cpu1|M] FIRQ channel 0x00000003 MEPC=0x00000820 MTVAL=0x00000000
</NEORV32-RTE-PANIC> ④
<NEORV32-RTE-PANIC> [cpu1|M] Instruction access fault MEPC=0x90000000 MTVAL=0x00000000
FATAL! HALTING CPU </NEORV32-RTE-PANIC>\n ⑤
```

- ① Illegal 32-bit instruction **MTVAL=0x000000FF** at address **MEPC=0x000002d6** while CPU 0 was in machine-mode (**[cpu0|M]**).
- ② Illegal 16-bit instruction **MTVAL=0x00000000** at address **MEPC=0x00000302** while CPU 0 was in user-mode (**[cpu0|U]**).

- ③ Misaligned load access at address **MEPC=0x00000440** (trying to load a full 32-bit word from address **MTVAL=0x80000101**) while CPU 0 was in user-mode (**[cpu0|U]**).
- ④ Fast interrupt request from channel 3 before executing instruction at address **MEPC=0x00000820** while CPU 1 was in machine-mode (**[cpu0|M]**).
- ⑤ Instruction bus access fault at address **MEPC=0x90000000** while CPU 1 was in machine-mode (**[cpu0|M]**). This is a fatal/non-resumable exception halting the CPU.

4.10.4. Application Context Handling

Upon trap entry the RTE backups the entire application context (i.e. all **x** general purpose registers) to the stack. The context is restored automatically after trap completion. The base address of the according stack frame is copied to the **mscratch** CSR. By having this information available, the RTE provides dedicated functions for accessing and altering the application context:

Listing 37. RTE Context Access Functions

```
// Prototypes
uint32_t neorv32_rte_context_get(int x); // read register
void      neorv32_rte_context_put(int x, uint32_t data); // write data to register

// Examples
uint32_t tmp = neorv32_rte_context_get(9); // read register 'x9'
neorv32_rte_context_put(28, tmp); // write 'tmp' to register 'x28'
```

The **x** argument is used to specify one of the RISC-V general purpose register **x0** to **x31**. Note that registers **x16** to **x31** are not available if the RISC-V **E ISA Extension** is enabled. For the SMP **Dual-Core Configuration** the provided context functions will access the stack frame of the interrupted application code that was running on the specific CPU core that caused the trap entry.

The context access functions can be used by application-specific trap handlers to *emulate* unsupported CPU / SoC features like unimplemented IO modules, unsupported instructions and even unaligned memory accesses.



Demo Program: Emulate Unaligned Memory Access

A demo program, which showcases how to emulate unaligned memory accesses using the NEORV32 runtime environment can be found in [sw/example/demo_emulate_unaligned](#).

Chapter 5. On-Chip Debugger (OCD)

The NEORV32 Processor features an *on-chip debugger* (OCD) compatible to the **Minimal RISC-V Debug Specification** implementing the **execution-based debugging** scheme (<https://docs.riscv.org/reference/debug/index.html>).

The on-chip debugger is implemented if the **OCD_EN** processor top generic is set to **true**. Optionally, up to 16 hardware triggers can be implemented (**Sdtrig ISA Extension**) to support hardware-assisted break- and watchpoints. Furthermore, the OCD supports an optional **Debug Authentication** module to constrain OCD access to authorized parties.



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.

Section Structure

- **openOCD**
- **Semihosting**
- **Debug Transport Module (DTM)**
- **Debug Module (DM)**
- **Debug Authentication**
- **CPU Debug Mode**
- **Trigger Module**

Key Features

- standard 4-wire JTAG access port
- debugging of up to 4 CPU cores ("harts")
- full control of the CPU: halting, single-stepping and resuming
- indirect access to all core registers and the entire processor address space (via program buffer)
- execution of arbitrary programs via the program buffer
- compatible with upstream OpenOCD and GDB
- optional trigger module for up to 16 hardware break- and watchpoints
- optional authentication for advanced security

Configuration Options

Table 110. NEORV32 OCD Configuration Generics

Name	Type	Default	Description
<code>OCD_EN</code>	boolean	false	Implement the on-chip debugger and the CPU debug mode.
<code>OCD_NUM_HW_TRIGGER S</code>	natural	0	Number of implemented HW triggers (Trigger Module / Sdtrig ISA Extension) for hardware break-/watchpoints (0..16).
<code>OCD_AUTHENTICATION</code>	boolean	false	Implement optional Debug Authentication module.
<code>OCD_JEDEC_ID</code>	suu(10:0)	"0000000000 00"	JEDEC ID; continuation codes plus vendor ID (passed to the JTAG Debug Transport Module (DTM)).

Overview

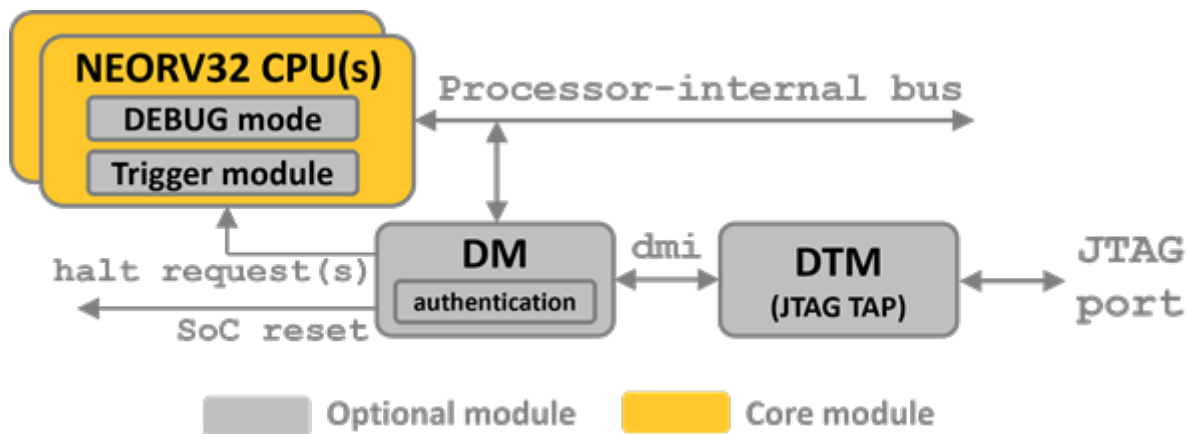


Figure 19. NEORV32 on-chip debugger complex

The NEORV32 on-chip debugger is based on five hardware modules:

1. **Debug Transport Module (DTM)**: JTAG access tap to allow an external adapter to interface with the *debug module (DM)*.
2. **Debug Module (DM)**: The RISC-V debug module is the main bridge between the external debugger and the processor being debugged. It provides a *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 between the debugger and the CPU.
3. **Debug Authentication**: Authenticator module to secure on-chip debugger access. By default this module implements a very simple authentication mechanism as example. Users can modify/replace this default logic to implement arbitrary authentication mechanism.
4. **CPU Debug Mode** ISA extension: This ISA extension provides the "debug execution mode" as another CPU operation mode that is used to execute the park loop code from the DM. This mode also provides additional CSRs and instructions.
5. **CPU Trigger Module**: This module provides up to 16 hardware triggers that can be used as hardware-assisted break- or watchpoints.

Theory of Operation

When debugging the system using the OCD, the external debugger (e.g. GDB) issues a halt request to the CPU to make it enter so-called *debug mode*. In this mode the application-defined architectural state of the system/CPU is "frozen" so the debugger can monitor it without interfering with the actual application. However, the OCD can also modify the entire architectural state at any time. While in debug mode, the debugger has full control over the entire CPU core.

After halting, the CPU executes the "park loop" code from the code ROM of the debug module (DM). This park loop implements an endless loop that is used to poll a memory-mapped **Status Register** of the DM. The flags in this register are used to communicate requests from the DM and to acknowledge their processing 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 or to signal that an exception has been raised while being in debug mode.

5.1. openOCD

By default, the free and open-source openOCD (<https://openocd.org/>) is used to establish a debugger connection. However, other interfaces can be used (like Segger or Lauterbach tools) if they provide RISC-V support.

openOCD is configured by a set of configuration scripts which are located in `neorv32/sw/openocd`. Two top-level scripts are provided: one for the single-core processor configuration (`neorv32.cfg`) and another one for the SMP dual-core configuration (`neorv32.dual-core.cfg`). Both scripts have the same format and include several helper scripts.

Listing 38. NEORV32 openOCD single-core CFG file (`neorv32.cfg`)

```
# configuration
set PATH [file dirname [file normalize [info script]]] ①
set CORENAME neorv32 ②
set NUMCORES 1 ③

# configure JTAG interface, setup target, authenticate and start
source [file join $PATH lib/interface.cfg] ④
source [file join $PATH lib/target.cfg] ⑤
source [file join $PATH lib/authenticate.cfg] ⑥
source [file join $PATH lib/start.cfg] ⑦
```

- ① Get the absolute path of the script.
- ② Set the core name that will show up in openOCD and GDB. This can be changed to any custom name.
- ③ Number of CPU cores (1 for the single-core setup, 2 for the dual-core setup).
- ④ Interface adapter (JTAG) configuration script. Replace by custom adapter setup.
- ⑤ NEORV32-specific target configuration and initialization.
- ⑥ Optional authentication process; the **Default Authentication Mechanism** is implemented as example; Replace this when using a custom authentication mechanism.
- ⑦ Reset and halt the target.

5.2. Semihosting

Semihosting is a mechanism developed by ARM that enables code running on the NEORV32 to communicate and use the input/output facilities on a host computer that is running a debugger. Examples of these facilities include keyboard input, screen output, and disk I/O. For example, you can use this mechanism to enable functions in the C library, such as `printf()` and `scanf()`, to use the screen and keyboard of the host instead of having a screen and keyboard on the target system. (source: <https://developer.arm.com/documentation/dui0471/i/semihosting/what-is-semihosting?lang=en>)

For example, with semihosting you can

- print to the host's `stdout` console
- read from the host's `stdin` console
- execute arbitrary commands in the host's shell
- retrieve the host's system time
- read and write files on the host system
- ...



Semihosting Example Program

A simple semihosting example program can be found in `sw/example/demo_semihosting`.

The RISC-V semihosting builds upon the ARM standard. Hence, semihosting has to be **explicitly enabled** on the host side (in `GDB`) using the ARM command set:

```
(gdb) monitor arm semihosting enable
semihosting is enabled
```

File accesses need to be explicitly enabled. Additionally, the base folder for accessing those files should be defined:

```
(gdb) monitor arm semihosting_fileio enable
(gdb) monitor arm semihosting_basedir path/to/neorv32/sw/example/demo_semihosting
```

The NEORV32 software framework provides a build-in library for semihosting primitives (`sw/lib/include/neorv32_semihosting.h`). Additionally, accesses to the standard IO streams (`stdin` and `stdout`) can be automatically mapped to the host's console. Functions such as `printf` and `puts` can then print right to the host's `stdout` console. Vice versa, functions like `scanf` and `fgets` will read from the host's `stdin`. To enable this automatic mapping, the `define STDIO_SEMIHOSTING` needs to be defined and the application firmware needs to be recompiled.

Listing 39. Forward `stdio.h` Calls to Host Computer

```
USER_FLAGS += -DSTDIO_SEMIHOSTING
```

When `STDIO_SEMIHOSTING` is defined *all* file accesses provided by `stdio.h` (via Newlib) will be redirected to the host computer via semihosting services.

*Redirecting all UART data via Semihosting*

You can redirect **all** physical UART data via semihosting by defining `UART_SEMIHOSTING` (e.g. "`USER_FLAGS+=-DSTDIO_SEMIHOSTING -DUART_SEMIHOSTING`").

*Semihosting Services Without a Host*

If any semihosting request is issued without a host being connected, an environment breakpoint exception is raised.

Further references:

- A great overview: <https://interrupt.memfault.com/blog/arm-semihosting>
- RISC-V semihosting spec.: <https://docs.riscv.org/reference/semihosting/index.html>
- Implementing semihosting on RISC-V: <https://embeddedinn.com/articles/tutorial/understanding-riscv-semihosting/>
- Description of the service calls by ARM: <https://developer.arm.com/documentation/dui0203/j/semihosting/semihosting-operations?lang=en>

5.3. Debug Transport Module (DTM)

The debug transport module "DTM" (VHDL module: `rtl/core/neorv32_debug_dtm.vhd`) provides a bridge between a standard 4-wire JTAG test access port ("tap") and the internal debug module interface.

Table 111. JTAG Top Level Signals of the DTM

Name	Width	Direction	Description
<code>jtag_tck_i</code>	1	in	serial clock
<code>jtag_tdi_i</code>	1	in	serial data input
<code>jtag_tdo_o</code>	1	out	serial data output
<code>jtag_tms_i</code>	1	in	mode select



Maximum JTAG Clock

All JTAG signals are synchronized to the processor's clock domain. Hence, no additional clock domain is required for the DTM. However, this constrains the maximal JTAG clock frequency (`jtag_tck_i`) to be less than or equal to 1/5 of the processor clock frequency (`clk_i`).



JTAG TAP Reset

The NEORV32 JTAG TAP does not provide a dedicated reset signal ("TRST"). However, JTAG-level resets can be triggered using TMS signaling.



Maintaining the JTAG Chain

If the on-chip debugger is disabled the JTAG serial input `jtag_tdi_i` is directly connected to the JTAG serial output `jtag_tdo_o` to maintain the JTAG chain.



JTAG TAP Access via FPGA Vendor Infrastructure

Most FPGAs are programmed via JTAG and also provide mechanisms to access user-defined JTAG logic from within the FPGA fabric using vendor-specific primitives. Therefore, a dedicated second JTAG connection for the OCD may be omitted. Instead, the debugger can share the same physical JTAG connection that is already used for FPGA configuration. See the setups in `[neorv32-setups]` (<https://github.com/stnolting/neorv32-setups>) for example implementations.

The DTM implements a single 5-bit *instruction register* `IR` and several *data registers* `DR` with different sizes. The individual data registers are accessed by writing the respective address to the instruction register. The following table shows all available data registers and their addresses:

Table 112. JTAG TAP registers

Address (via IR)	Name	Size (bits)	Description
00001	IDCODE	32	identification code (see below)
10000	DTMCS	32	debug transport module control and status register (see below)
10001	DMI	41	debug module interface (see below)
others	BYPASS	1	default JTAG bypass register

Table 113. IDCODE - DTM Identification Code Register

Bit(s)	Name	R/W	Description
31:28	version	r/-	version ID, hardwired to zero
27:12	partid	r/-	part ID, hardwired to zero
11:1	manid	r/-	JEDEC manufacturer ID, assigned via the JEDEC_ID generic
0	-	r/-	hardwired to 1

Table 114. DTMCS - DTM Control and Status Register

Bit(s)	Name	R/W	Description
31:21	-	r/-	reserved, hardwired to zero
20:18	errinfo	r/-	not implemented; hardwired to zero
17	dmihardreset	r/w	setting this bit will reset the debug module interface; this bit auto-clears
16	dmireset	r/w	setting this bit will clear the sticky error state; this bit auto-clears
15	-	r/-	reserved, hardwired to zero
14:12	idle	r/-	recommended idle states (= 0, no idle states required)
11:10	dmistat	r/-	read-only alias of DMI.op; hardwired to 00 (NOP)
9:4	abits	r/-	number of address bits in DMI register (= 7)
3:0	version	r/-	0001 = DTM is compatible to RISC-V debug spec. versions v0.13 and v1.0

Table 115. DMI - DTM Debug Module Interface Register

Bit(s)	Name	R/W	Description
40:34	addr	r/w	7-bit address, see DM Registers
33:2	data	r/w	32-bit data to write/read to/from the addressed DM register

Bit(s)	Name	R/W	Description
1:0	op	r/-	Read access - status: 00 = idle, 11 = access while DMI is busy (sticky)
		-/w	Write access - operation: 00 = NOP, 01 = read, 10 = write

5.4. Debug Module (DM)

The debug module "DM" (VHDL module: `rtl/core/neorv32_debug_dm.vhd`) acts as a translation interface between abstract operations issued by the debugger application (like GDB) and the platform-specific debugger hardware. It supports the following features:

- Gives the debugger necessary information about the implementation.
- Allows the hart to be halted/resumed/reset and provides the current status.
- Provides abstract read and write access to the halted hart's general purpose registers.
- Provides access to a reset signal that allows debugging from the very first instruction after reset.
- Provides a *program buffer* to force the hart to execute arbitrary instructions.
- Allows memory accesses (to the entire address space) from a hart's point of view.
- Optionally implements an authentication mechanism to secure on-chip debugger access.

The NEORV32 DM follows the "Minimal RISC-V External Debug Specification" to provide full debugging capabilities while keeping resource/area requirements at a minimum. It implements the **execution based debugging scheme** for up to four individual CPU cores ("harts") and provides the following architectural core features:

- program buffer with 2 entries and an implicit `ebreak` instruction at the end
- indirect bus access via the CPU using the program buffer
- abstract commands: "access register" plus auto-execution
- halt-on-reset capability
- optional authentication



DM Spec. Version

The NEORV32 DM complies to the RISC-V DM spec version 1.0.

From the DTM's point of view, the DM implements a set of **DM Registers** that are used to control and monitor the debugging session. From the CPU's point of view, the DM implements several memory-mapped registers that are used for communicating data, instructions, debugging control and status (**DM CPU Access**).

External Reset Output

The entire processor can be reset at any time by the debugger via the `ndmreset` bit of the `dmcontrol` register. This signal is also available as processor top signal (**Processor Top Entity - Signals**: `rstn_ocrd_o`) and can be used to reset processor-external modules via the on-chip debugger. This signal is low-active and synchronous to the processor clock. It is available if the on-chip debugger is actually implemented; otherwise it is hardwired to 1. Note that the signal also becomes active (low) when the processor's main reset signal is active (even if the on-chip debugger is deactivated or disabled for synthesis).

5.4.1. DM Registers

The DM is controlled via a set of registers that are accessed via the DTM. The following registers are implemented:



Unimplemented Registers

Write accesses to registers that are not implemented are simply ignored and read accesses to these registers will always return zero. In both cases no error condition is signaled to the DTM.

Table 116. Available DM registers

Address	Name	Description
0x04	data0	Abstract data register 0
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 next DM; reads as zero to indicate there is only one DM
0x20	progbuf0	Program buffer 0
0x21	progbuf1	Program buffer 1
0x30	authdata	Data to/from the authentication module
0x38	sbscs	System bus access control and status; reads as zero to indicate there is no system bus access
0x40	haltsum0	Hart halt summary

data0

0x04 **Abstract data 0** **data0**

Reset value: **0x00000000**

Basic read/write data exchange register to be used with abstract commands (for example to read/write data from/to CPU GPRs).

dmcontrol

0x10 **Debug module control register** **dmcontrol**

Reset value: **0x00000000**

Control of the overall debug module and the hart. The following table shows all implemented bits. All remaining bits/bit-fields are configured as "zero" and are read-only. Writing '1' to these bits/fields will be ignored.

Table 117. **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
27	-	r/-	reserved, hardwired to zero
26	hasel	r/-	0 : only a single hart can be selected at once
25:16	hartsello	r/w	hart select; only the lowest 3 bits are implemented
15:6	hartselhi	r/-	hardwired to zero
5:4	-	r/-	reserved, hardwired to zero
3	setresethaltreq	r/-	0 : halt-on-reset not implemented
2	clrresethaltreq	r/-	0 : halt-on-reset not implemented
1	ndmreset	r/w	put whole system (except OCD) into reset state when 1
0	dmactive	r/w	DM enable; writing 0-1 will reset the DM

dmstatus0x11 **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 118. **dmstatus** Register Bits

Bit	Name [RISC-V]	Description
31:25	<i>reserved</i>	reserved; zero
24	ndmresetpending	DM in reset state
23	stickyunavail	*unavail bits reflect the current state
22	impebreak	1 : indicates an implicit ebreak instruction after the last program buffer entry
21:20	<i>reserved</i>	reserved; zero

Bit	Name [RISC-V]	Description
19	allhavereset	1 when the selected hart is in reset state
18	anyhavereset	
17	allresumeack	1 when the selected hart has acknowledged a resume request
16	anyresumeack	
15	allnonexistent	1 when the selected hart is not available
14	anynonexistent	
13	allunavail	1 when the DM is disabled to indicate the selected hart is unavailable
12	anyunavail	
11	allrunning	1 when the selected hart is running
10	anyrunning	
9	allhalted	1 when the selected hart is halted
8	anyhalted	
7	authenticated	set if authentication passed; see Debug Authentication
6	authbusy	set if authentication is busy, see Debug Authentication
5	hasresethaltreq	0: halt-on-reset is not supported (directly)
4	confstrptrvalid	0: no configuration string available
3:0	version	0011: DM compatible to debug spec. version v1.0

hartinfo0x12 **Hart information****hartinfo**Reset value: *see below*

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

Table 119. **hartinfo** Register Bits

Bit	Name [RISC-V]	Description
31:24	<i>reserved</i>	reserved; zero
23:20	nscratch	0001: number of dscratch* CPU registers = 1
19:17	<i>reserved</i>	reserved; zero
16	dataaccess	0: the data registers are shadowed in the hart's address space
15:12	datasize	0001: number of 32-bit words in the address space dedicated to shadowing the data registers (1 register)

Bit	Name [RISC-V]	Description
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 DM CPU Access)

abstracts0x16 **Abstract control and status**[abstracts](#)Reset value: `0x02000801`

Command execution info and status.

Table 120. [abstracts](#) Register Bits

Bit	Name [RISC-V]	R/W	Description
31:29	<i>reserved</i>	r/-	reserved; zero
28:24	<code>progbuFSIZE</code>	r/-	<code>0010</code> : size of the program buffer (<code>progbuf</code>) = 2 entries
23:11	<i>reserved</i>	r/-	reserved; zero
12	<code>busy</code>	r/-	set when a command is being executed
11	<code>relaxedpriv</code>	r/-	<code>1</code> : PMP rules are ignored when in debug mode
10:8	<code>cmderr</code>	r/w	error during command execution (see below); has to be cleared by writing <code>111</code>
7:4	<i>reserved</i>	r/-	reserved; zero
3:0	<code>datacount</code>	r/-	<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

command0x17 **Abstract command**[command](#)Reset value: `0x00000000`

Writing this register will trigger the execution of an abstract command. A new command can only be executed if `cmderr` is zero. The entire register is write-only (reads will return zero).



The NEORV32 DM only supports **Access Register** abstract commands. These commands can only access the hart's GPRs x0 - x15/31 (abstract command register

index **0x1000 - 0x101f**).

Table 121. **command** Register Bits

Bit	Name [RISC-V]	R/W	Description / required value
31:24	cmdtype	-/w	00000000 : indicates "access register" command
23	<i>reserved</i>	-/w	reserved, has to be 0 when writing
22:20	aarsize	-/w	010 : indicates 32-bit accesses
21	aarpostincrement	-/w	0 : post-increment is not supported
18	postexec	-/w	set if the program buffer is executed <i>after</i> the command
17	transfer	-/w	set if the operation in write is conducted
16	write	-/w	1 : copy data0 to [regno], 0 : copy [regno] to data0
15:0	regno	-/w	GPR-access only; has to be 0x1000 - 0x101f

abstractauto

0x18 **Abstract command auto-execution**

abstractauto

Reset value: **0x00000000**

Register to configure if a read/write access to a DM register re-triggers execution of the last abstract command.

Table 122. **abstractauto** Register Bits

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

progbuf

0x20 **Program buffer 0**

progbuf0

0x21 **Program buffer 1**

progbuf1

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

Program buffer (two entries) for the DM.

authdata

0x30 **Authentication data**

authdata

Reset value: *user-defined*

This register serves as a 32-bit serial port to/from the authentication module. See [Debug Authentication](#).

haltsum0

0x40 **Halt summary 0**

haltsum0

Reset value: 0x00000000

Each bit corresponds to a hart being halted. Only the lowest four bits are implemented.

5.4.2. DM CPU Access

From the CPU's perspective the DM acts like another memory-mapped peripheral. It occupies 512 bytes of the CPU's address space starting at address `base_io_dm_c` (0xffff0000). This address space is divided into four sections with 64 bytes each to provide access to the *park loop code ROM*, the *program buffer*, the *data buffer* and the *status register*. The program buffer, the data buffer and the status register do not fully occupy the 64-byte-wide sections and are mirrored several times across the according section.

Table 123. DM CPU Access - Address Map

Base address	Physical size	Description
0xffffffff00	64 bytes	ROM for the "park loop" code (Code ROM)
0xffffffff40	16 bytes	Program buffer (progbuf)
0xffffffff80	4 bytes	Data buffer (data0)
0xfffffffc0	4 bytes	Status Register

DM Register Access



All memory-mapped registers of the DM can only be accessed by the CPU when in debug mode. Hence, the DM registers are not accessible for normal CPU operations. Any CPU access outside of debug mode will raise a bus access fault exception.

Code ROM

The code ROM contains the minimal OCD firmware that implements the debugger's park loop.



Park Loop Code Sources ("OCD Firmware")

The assembly sources of the park loop code are available in `sw/oed-firmware/park_loop.S`.

The park loop code provides two entry points where code execution can start. These are used to enter the park loop either when an explicit debug-entry/halt request has been issued (for example a

halt request) or when an exception has occurred while executing code in debug mode (from the program buffer).

Table 124. Park Loop Entry Points

Address	Description
<code>dm_exc_entry_c (base_io_dm_c + 0)</code>	Exception entry address
<code>dm_park_entry_c (base_io_dm_c + 4)</code>	Normal entry address (halt request)

When the CPU enters (via an explicit halt request from the debugger) or re-enters debug mode (for example via an **ebreak** in the DM's program buffer), it jumps to the **normal entry point** that is configured via the `CPU_DEBUG_PARK_ADDR` CPU generic. By default, this address is set to `dm_park_entry_c`, which is defined in the main package file. If an exception is encountered during debug mode, the CPU jumps to the address of the **exception entry point** configured via the `CPU_DEBUG_EXC_ADDR` CPU generic. By default, this address is set to `dm_exc_entry_c`, which is also defined in the main package file.

Status Register

The status register provides a direct communication channel between a CPU executing the park loop and the debugger-controlled DM. This register is used to communicate **requests** which are issued by the DM, and the according **acknowledges** which are issued by a CPU core.

From the CPU side the register is accessed using byte-wide load and store operations. Each of the four bytes is associated to a specific request when read or to an according acknowledge when written.

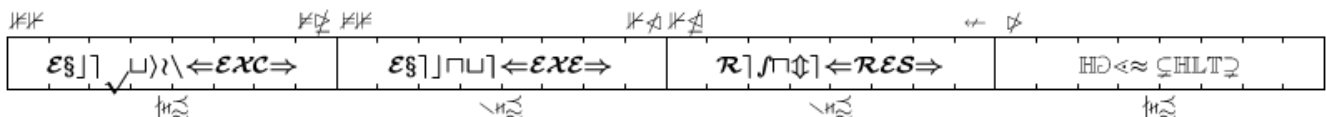


Figure 20. DM Status Register Layout

If the park-loop code of a CPU reads a non-zero value from `RES` then it is requested to resume operation. If it reads a non-zero value from `EXE` then it is requested to execute the program buffer. Accordingly, the park-loop code acknowledges these requests by writing *any* value (including zero) to the according byte. The `HLT` and `EXC` bytes are used to acknowledge the external halt request and to indicate that executing the program buffer has encountered an exception, respectively.

Table 125. DM Status Register (byte-wise access)

Name	Bits	CPU Read Access (Request)	CPU Write Access (Acknowledge)
<code>EXC</code>	<code>31:24</code>	-	CPU has encountered an exception
<code>EXE</code>	<code>23:16</code>	DM requests program buffer execution	CPU has started to execute program buffer
<code>RES</code>	<code>15:8</code>	DM requests resuming	CPU has started to resume
<code>HLT</code>	<code>7:0</code>	-	CPU has halted

In the SMP **Dual-Core Configuration** all cores use the same request/acknowledge communication mechanism by reading and writing the DM status register. The CPU-specific requests and acknowledges can still be clearly assigned, as the CPU **Bus Interface** implicitly transmits the ID of the accessing CPU core. Therefore, CPU core i only sees the requests intended for it and can only set the acknowledges originating from itself when accessing the DM status register.

5.5. Debug Authentication

Optionally, the on-chip debugger DM can be equipped with an *authenticator module* to secure debugger access. This authentication is enabled by the **OCDAUTHENTICATION** top generic. When disabled, the debugger is always authorized and has unlimited access. When enabled, the debugger is required to authenticate in order to gain access.

The authenticator module is implemented as individual RTL module (`rtl/core/neorv32_debug_auth.vhd`). By default, it implements a very simple authentication mechanism. Note that this default mechanism is not secure in any way - it is intended as example logic to illustrate the interface and authentication process. Users can modify the default logic or replace the entire module to implement a more sophisticated custom authentication mechanism.

The authentication interface is compliant to the RISC-V debug spec and is based on a single CSR and two additional status bits:

- **authdata** CSR: this 32-bit register is used to read/write data from/to the authentication module. It is hardwired to all-zero if authentication is not implemented.
- **dmstatus** CSR:
 - The **authenticated** bit (read-only) is set if authentication was successful. The debugger can access the processor only if this bit is set. It is automatically hardwired to **1** (always authenticated) if the authentication module is not implemented.
 - The **authbusy** bit (read-only) indicates if the authentication module is busy. When set, no data should be written/read to/from **authdata**. This bit is automatically hardwired to **0** (never busy) if the authentication module is not implemented.

openOCD provides dedicated commands to exchange data with the authenticator module:

Listing 40. openOCD RISC-V Authentication Commands

```
riscv authdata_read      // read 32-bit from authdata CSR
riscv authdata_write value // write 32-bit value to authdata CSR
```

Based on these two primitives arbitrary complex authentication mechanism can be implemented.

5.5.1. Default Authentication Mechanism



The default authentication mechanism is not secure at all. Replace it by a custom design.

The default authenticator hardware implements a very simple authentication mechanism: a single read/write bit is implemented that directly corresponds to the **authenticated** bit in **dmstatus**. This bit can be read/written as bit zero (LSB) of the **authdata** CSR. Writing 1 to this register will result in a successful authentication.

The default openOCD configuration script provides a helper script for authentication. This script

also provides several helper functions for interaction with the RISC-V debug mechanism. Additionally, the default authentication mechanism is implemented there (as example):

Listing 41. Default authentication process ([sw/openocd/authentication.cfg](#))

```
# read challenge
set CHALLENGE [authenticator_read]
# compute response (default authenticator module)
set RESPONSE [expr {$CHALLENGE | 1}]
# send response
authenticator_write $RESPONSE
# success?
authenticator_check
```

5.6. CPU Debug Mode

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

Debug-mode is entered on any of the following events:

1. The CPU executes an **ebreak** instruction (when in machine-mode and **ebreakm** in **dcsr** is set OR when in user-mode and **ebreaku** in **dcsr** is set).
2. A debug halt request is issued by the DM (via CPU **db_halt_req_i** signal, high-active).
3. The CPU completes executing a single instruction while being in single-step debugging mode (**step** in **dcsr** is set).
4. A hardware trigger from the **Trigger Module** fires (if **exe** in **tdata1** / **mcontrol** is set).



From a hardware point of view these debug-mode-entry conditions are normal traps (synchronous exceptions or asynchronous interrupts) that are handled transparently by the control logic.

Whenever the CPU enters debug-mode it performs the following operations:

- wake-up CPU if it was send to sleep mode by the **wfi** instruction
- switch to debug-mode privilege level
- move the current program counter to **dpc**
- copy the hart's current privilege level to the **prv** flags in **dcsr**
- set **cause** in **dcsr** according to the cause why debug mode is entered
- **no update** of **mtval**, **mcause**, **mtval** and **mstatus[h]** CSRs
- load the address configured via the CPU's (**CPU_DEBUG_PARK_ADDR**) generic to the program counter jumping to the "debugger park loop" code stored in the debug module (DM)

When the CPU is in debug-mode:

- 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 the **CPU_DEBUG_PARK_ADDR** generic) 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 the **CPU_DEBUG_EXC_ADDR** generic) to signal an exception to the DM; the CPU restarts the park loop again afterwards
- interrupts are disabled; however, they will remain pending and will get executed after the CPU has left debug mode and is not being single-stepped
- if the DM makes a resume request, the park loop exits and the CPU leaves debug mode by executing **dret**
- the standard counters (**Machine**) Counter and Timer CSRs **[m]cycle[h]** and **[m]instret[h]** are stopped
- all **Hardware Performance Monitors (HPM) CSRs** are stopped

Debug mode is left either by executing the **dret** instruction or by performing a hardware reset of the CPU. Executing **dret** outside of debug mode will raise an illegal instruction exception.

Whenever the CPU leaves debug mode it performs the following operations:

- set the hart's current privilege level according to the **prv** flags of **dcsr**
- restore the original program counter from **dpc** resuming normal operation

5.6.1. CPU Debug Mode CSRs

Two additional CSRs are required by the "Minimal RISC-V Debug Specification": the debug mode control and status register **dcsr** and the debug program counter **dpc**. An additional general purpose scratch register for debug-mode-only (**dscratch0**) allows faster execution by having a fast-accessible backup register. These CSRs are only accessible if the CPU is in debug mode. If these CSRs are accessed outside of debug mode an illegal instruction exception is raised.

dcsr

Name	Debug control and status register
Address	0x7b0
Reset value	0x40000410
ISA	Zicsr & Sdext
Description	This register is used to configure the debug mode environment and provides additional status information.

Table 126. 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

Bit	Name [RISC-V]	R/W	Description
15	ebreakm	r/w	ebreak instructions in machine mode will <i>enter</i> 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 <i>enter</i> 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 : <i>reserved</i>
4	mprven	r/-	1 : mprv in mstatus is also evaluated when in debug mode
3	nmip	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

Name	Debug program counter
Address	0x7b1
Reset value	0x00000000
ISA	Zicsr & Sdext
Description	The register is used to store the current program counter when debug mode is entered. The dret instruction will return to the address stored in dpc by automatically moving dpc to the program counter.



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

dscratch0

Name	Debug scratch register 0
Address	0x7b2
Reset value	0x00000000
ISA	Zicsr & Sdext
Description	The register provides a general purpose debug mode-only scratch register.

5.7. Trigger Module

The RISC-V **Sdtrig** ISA extension adds a *trigger module* to the CPU core. The number of hardware triggers is configured via the **OCD_NUM_HW_TRIGGERS** top generic. Up to 16 hardware triggers can be implemented. If **OCD_NUM_HW_TRIGGERS** is set to a value greater than 0 the **Sdtrig ISA Extension** is automatically enabled. The trigger module implements a subset of the features described in the "RISC-V Debug Specification / Trigger Module (**Sdtrig**)" and complies to version v1.0 of that spec. Note that the NEORV32 trigger module only supports "type 6 - instruction address match" triggers. These triggers can be used as hardware-assisted breakpoints or as hardware-assisted watchpoints.

- A **breakpoint** stops the program whenever a particular point in the program is reached (i.e. after executing the instruction at the specified address): GDB's **hbreak** command.
- A **watchpoint** stops the program whenever the value of a variable or expression changes (i.e. after a load and/or store operation has accessed data at the specified address): GDB's **[a|r]watch** commands.

From a hardware point of view the trigger modules raises an *asynchronous* exception right after the triggering operation has been retired. The **dpc** CSR shows the instruction address where normal execution must be resumed to preserve the program flow



Debug-Mode Only

The trigger module can be used by debug-mode only. Write accesses from machine-mode are constrained as **tdata1.dmode** is hardwired to one. However, the **Smpmp ISA Extension** can be used for machine-mode instruction or data address traps.

5.7.1. Trigger Module CSRs

The **Sdtrig** ISA extension adds 4 additional CSRs. These CSRs can also be read from machine-mode, but write-accesses are restricted as **tdata1.dmode** is hardwired to one.

tselect

Name	Trigger select register
Address	0x7a0
Reset value	0x00000000
ISA	Zicsr & Sdtrig
Description	This CSR is used to select which hardware trigger is available via the tdata1 , tdata2 and tinfo CSRs. Only the lowest $\log_2(\text{OCD_NUM_HW_TRIGGERS})+1$ bit are writable. However, software should check if tinfo is non-zero for the written tselect value to determine if the selected trigger exists.

tdata1

Name	Trigger data register 1, visible as trigger "type 6 match control" (mcontrol6)
Address	0x7a1
Reset value	0x60000048
ISA	Zicsr & Sdtrig
Description	This CSR provides access to the configuration bits of the currently selected trigger (via tselect). Note that only "type 6" triggers are supported. The according configuration bits are listed below. Write accesses from machine-mode are ignored (because tdata1.dmode is hardwired to one).

Table 127. Match control CSR (**tdata1**) bits of the trigger selected via **tselect**

Bit	Name [RISC-V]	R/W	Description
31:28	type	r/-	0110 : address match trigger type 6
27	dmode	r/-	'1': ignore write accesses to tdata1 and tdata2 from machine-mode
26	uncertain	r/-	0 : trigger satisfies the configured conditions
25	hit1	r/c	see hit0 bit
24	vs	r/-	0 : VS-mode not supported
23	vu	r/-	0 : VU-mode not supported
22	hit0	r/c	hit1 (MSB) and hit0 (LSB) combine into a single 2-bit field; this field is set to 0b11 when the trigger just fired; timing: the trigger fires immediately after the instruction that triggered has retired, but before any subsequent instructions were executed; the debugger has to write zeros to acknowledge the firing trigger
21	select	r/-	0 : only address matching is supported
20:19	reserved	r/-	00 : hardwired to zero
18:16	size	r/-	000 : match accesses of any size
15:12	action	r/-	0001 : enter debug-mode on trigger match
11	chain	r/-	0 : chaining is not supported
10:6	match	r/-	0000 : equal-match only
6	m	r/-	1 : trigger enabled when in machine-mode
5	uncertainen	r/-	0 : feature not supported, hardwired to zero
4	s	r/-	0 : S-mode not supported
3	u	r/-	trigger enabled when in user-mode, set if U ISA extension is enabled

Bit	Name [RISC-V]	R/W	Description
2	execute	r/w	set to enable this trigger to fire on instruction address match (i.e. "breakpoint")
1	store	r/w	set to enable this trigger to fire on data store address match (i.e. "watchpoint")
0	load	r/w	set to enable this trigger to fire on data load address match (i.e. "watchpoint")

tdata2

Name	Trigger data register 2
Address	0x7a2
Reset value	0x00000000
ISA	Zicsr & Sdtrig
Description	This CSR provides access to the address bits (instruction address for breakpoints; data access address for watchpoints) of the currently selected trigger (via tselect). Write accesses from machine-mode are ignored (because tdata1.dmode is hardwired to one).

tdata3

Name	Trigger data register 3
Address	0x7a3
Reset value	0x00000000
ISA	Zicsr & Sdtrig
Description	The features provided by this CSR are not implemented. Hence, this CSR is hardwired to all-zero.

tinfo

Name	Trigger information register
Address	0x7a4
Reset value	0x01000040
ISA	Zicsr & Sdtrig
Description	The CSR shows additional trigger information of the currently selected trigger (via tselect). This CSR is read-only, any write access is ignored.

*Table 128. Trigger info CSR (**tinfo**) bits of the trigger selected via **tselect***

Bit	Name [RISC-V]	R/W	Description
31:24	version	r/-	0x01 : compatible to Sdtrig spec. version v1.0
23:15	reserved	r/-	0x00 : hardwired to zero
15:0	info	r/-	0x0040 : only "type 6" (= bit 6 set) trigger is supported; for invalid value of tselect this bit-field read as 0x0001 ("type 0" = none)

Chapter 6. Legal

About

The NEORV32 RISC-V Processor

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

Stephan Nolting, M.Sc.

stnolting[at]gmail[dot]com

European Union, Germany

DOI

This project provides a *digital object identifier (DOI)* provided by **zenodo**: <https://doi.org/10.5281/zenodo.5018888>

License

BSD 3-Clause License

Copyright (c) NEORV32 contributors.

Copyright (c) 2020 - 2026, Stephan Nolting. All rights reserved.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

1. Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
3. Neither the name of the copyright holder nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT HOLDER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN

IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.



SPDX Identifier

SPDX-License-Identifier: BSD-3-Clause

Proprietary Notice

- "GitHub" is a subsidiary of Microsoft Corporation.
- "Vivado" and "Artix" are trademarks of AMD Inc.
- "ARM", "AMBA", "AXI", "AXI4", "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.
- "GateMate" is a trademark of Cologne Chip AG.
- "Windows" is a trademark of Microsoft Corporation.
- "Tera Term" copyright by T. Teranishi.
- "NeoPixel" is a trademark of Adafruit Industries.
- "Segger Embedded Studio" and "J-Link" are trademarks of Segger Microcontroller Systems GmbH.
- Images/figures made with *Microsoft Power Point*.
- Diagrams made with *WaveDrom*.
- Documentation made with **asciidoc**tor.

All further/unreferenced projects/products/brands belong to their according copyright holders. No copyright infringement intended.

Disclaimer

This project is released under the BSD 3-Clause license. NO COPYRIGHT INFRINGEMENT INTENDED. Other implied or used projects/sources might have different licensing – see their according documentation for 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.

Acknowledgments

A big shout-out to the community and all the **contributors**, who helped improving this

project! This project would not be where it is without them. ☐☐

RISC-V - instruction sets want to be free!

Continuous integration provided by **GitHub Actions** and powered by **GHDL**.