The NEORV32 RISC-V Processor

by Dipl.-Ing. Stephan Nolting

A small, customizable and open-source
full-scale 32-bit RISC-V soft-core CPU and SoC
written in platform-independent VHDL.

Proprietary and Legal Notice

"GitHub" is a Subsidiary of Microsoft Corporation.
"Vivado" and "Artix" are trademarks of Xilinx Inc.
"AXI" and "AXI4-Lite" are trademarks of Arm Holdings plc.
"ModelSim" is a trademark of Mentor Graphics – A Siemens Business.
"Quartus Prime" and "Cyclone" are trademarks of Intel Corporation.
"iCE40", "UltraPlus" and "Radiant" are trademarks of Lattice Semiconductor Corporation.
"Windows" is a trademark of Microsoft Corporation.
"Tera Term" copyright by T. Teranishi.
Timing diagrams made with *WaveDrom Editor*.
"NeoPixel" is a trademark of Adafruit Industries.

Limitation of Liability for External Links

Our website 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.

Disclaimer

This project is released under the BSD 3-Clause license. No copyright infringement intended. Other implied or used projects might have different licensing – see their documentation to get more information.

This project is not affiliated with or endorsed by the Open Source Initiative.
https://www.oshwa.org
https://opensource.org

RISC-V – Instruction Sets Want To Be Free!
https://riscv.org/
https://github.com/riscv

Citing

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

```
S. Nolting, "The NEORV32 RISC-V Processor", github.com/stnolting/neorv32
```

The **NEORV32 Processor** Project
©2021 Dipl.-Ing. Stephan Nolting, Hannover, Germany
https://github.com/stnolting/neorv32
Contact: stnolting@gmail.com

BSD 3-Clause License

Copyright (c) 2021, 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.

Project Logo

The NEORV32 project logo consists of the "NEORV32" name written in capital letters (*Calibri* font), where "NEO" is painted in black and "RV32" in moderate orange, a vertical black dash and a stylized microchip with 5 "pins" on each side. This chip has an orange block on the inside surrounded by a black box and a smaller black square in the center. The logo comes in three versions: the normal version with white background, the dark logo where black is replaced by gray color and the inverse version with black background. The inverse version does not invert the orange parts of the original logo. The logo image files (*.png) are located in `docs/figures`.

Normal logo (white background)   Dark logo (transparent bg)   Inverse logo (black background)

Table of Content

1. Overview

NEORV32

The **NEORV32**[1] **Processor** is a customizable microcontroller-like system on chip (SoC) that is based on the RISC-V NEORV32 CPU. The processor is intended as ready-to-go auxiliary processor within a larger SoC designs or as stand-alone custom microcontroller. Its top entity can be directly synthesized for any target technology without modifications.

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 peripherals.

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 (a prebuilt toolchain is also available on GitHub).

The project's change log is available in the CHANGELOG.md file in the root directory of the NEORV32 repository.

Structure

2. NEORV32 Central Processing Unit (CPU)
- instruction set(s) and extensions
- instruction timing
- control ans status registers
- traps, exceptions and interrupts
- hardware execution safety
- native bus interface

3. NEORV32 Processor (SoC)
- top entity signals and configuration generics
- address space layout
- internal peripheral devices and interrupts
- internal memories and caches
- internal bus architecture
- external bus interface

4. Software Architecture (Software Framework)
- core libraries
- bootloader
- makefiles
- runtime environment

5. Let's Get It Started! (Tutorials and Guides)
- toolchain installation and setup
- hardware setup
- software setup
- application compilation
- simulating the processor

---

1   Pronounced "*neo-R-V-thirty-two*" or "*neo-risc-five-thirty-two*" in its long form.

1.1. Project Key Features

- ✔ **NEORV32 CPU:** 32-bit `rv32i` RISC-V base CPU (→ p.17); **passes the official RISC-V architecture tests (→ p.19)**; optional RISC-V CPU extensions:

  - `A` extension for atomic memory operations
  - `B` extension for bit manipulation instructions
  - `C` extension for compressed instructions (16-bit)
  - `E` extensions for embedded CPU version (reduced register file size)
  - `M` extension for integer multiplication and division hardware
  - `U` extension for less-privileged user mode
  - `Zfinx` single-precision floating-point extension
  - `Zicsr` extension for control and status register access and exception/interrupt system
  - `Zifencei` extension for instruction stream synchronization
  - `PMP` extension for RISC-V-compatible physical memory protection extension
  - `HPM` extensions for hardware performance monitors

- ✔ Safe execution hardware (→ p. 52)

- ✔ Official RISC-V open-source architecture ID

- ✔ **Software framework**

  - GCC-based toolchain (→ p.117) - prebuilt toolchains available; application compilation based on *GNU makefiles* (→ p.119)

  - Bootloader supporting application upload via UART; programming/booting of/from external SPI flash

  - Core libraries for high-level usage of the provided functions and peripherals

  - Runtime environment and several example programs

  - Doxygen-based documentation of the software framework (→ p.138); a deployed version is available at https://stnolting.github.io/neorv32/files.html

  - FreeRTOS port + demos available (→ p.149)

- ✔ **NEORV32 Processor:** Highly-configurable full-scale microcontroller-like processor system / **SoC** (→ p.60) based on the NEORV32 CPU:

  - Serial interfaces (UARTs, TWI, SPI)

  - Timers and counters (WDT, MTIME, NCO)

  - General purpose IO and PWM and native NeoPixel (c) compatible smart LED interface

  - Embedded memories / caches for data, instructions and bootloader

  - External memory interface (Wishbone or AXI4-Lite → p.80),...

- ✔ Fully synchronous design, no latches, no gated clocks

- ✔ Completely described in behavioral, platform-independent VHDL

- ✔ Small hardware footprint and high operating frequency (→ p.11)

1.2. Project Folder Structure

```
neorv32              Project home folder

├─.ci                Project change log

├─boards             Exemplary setups for various FPGA boards

├─CHANGELOG.md       Project documentary: RISC-V specifications implemented within this
                     project, Wishbone bus specification, NEORV32 data sheet, doxygen
├─docs               makefiles

                     Software documentary HTML files (generated by doxygen)

  ├─doxygen_build    Images mainly for the GitHub front page + project logos

  └─figures          Port files for the RISC-V architecture test framework; see section
                     2.1. RISC-V Architecture Tests
├─riscv-arch-test

                     Processor's VHDL source files

├─rtl                This folder contains all the rtl (VHDL) core files of the NEORV32 –
                     make sure to add ALL of them to your FPGA EDA project
  ├─core             Alternative top entities of the NEORV32 processor

  ├─top_templates    This folder provides FPGA technology-specific optimized HW modules

  └─fpga_specific    The sim folder contains the default VHDL testbench and additional
                     simulation files (see section 5.12. Simulating the Processor)
├─sim                Simulation scripts for GHDL

  ├─ghdl             Simulation(-only!)-optimized CPU/processor components

  ├─rtl_modules      Pre-configured Xilinx ISIM wafeform

  └─Vivado           The software folder contains the processor's core libraries,
                     makefiles, linker script, start-up code and example programs
└─sw                 Source and compilation script of the NEORV32-internal bootloader

  ├─bootloader       Linker script and startup code

  ├─common           Here you can find several example programs. Each project folder
                     includes the program's C sources and a makefile – add your own
  ├─example          projects to this folder

    └─...            Helper program to generate executables for the NEORV32

  ├─image_gen        This folder contains the processor's core libraries

  └─lib              NEORV32 hardware driver library C source files and the according
                     header/include files
    ├─include
    └─source
```

> ℹ There are further files and folders starting with a dot which – for example – contain
> data/configurations only relevant for `git` or for the continuous integration framework (`.ci`). These
> files and folders are not relevant for the actual checked-out NEORV32 project.

1.3. VHDL File Hierarchy

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

> ⚠️ All core VHDL files have to be assigned to a new design **library** named **neorv32**. Additional files, like alternative top entities, can be assigned to any library.

```
neorv32_top.vhd                          NEORV32 Processor top entity

├─neorv32_boot_rom.vhd                    Bootloader ROM
│  └─neorv32_bootloader_image.vhd         Boot ROM initialization image for the bootloader
│
├─neorv32_busswitch.vhd                   Processor bus switch for CPU buses (I&D)
├─neorv32_icache.vhd                      Processor-internal instruction cache
├─neorv32_cfs.vhd                         Custom functions subsystem
│
├─neorv32_cpu.vhd                         NEORV32 CPU top entity
│  │
│  ├─neorv32_package.vhd                  Processor/CPU main VHDL package file
│  ├─neorv32_cpu_alu.vhd                  Arithmetic/logic unit
│  ├─neorv32_cpu_bus.vhd                  Bus interface unit + physical memory protection
│  ├─neorv32_cpu_control.vhd              CPU control, exception/IRQ system and CSRs
│  │  └─neorv32_cpu_decompressor.vhd      Compressed instructions decoder
│  │
│  ├─neorv32_cpu_cp_bitmanip.vhd          Bit manipulation co-processor (B extension)
│  ├─neorv32_cpu_cp_fpu.vhd               Floating-point co-processor (Zfinx extension)
│  ├─neorv32_cpu_cp_muldiv.vhd            Multiplication/division co-processor (M extension)
│  └─neorv32_cpu_regfile.vhd              Data register file
│
├─neorv32_dmem.vhd                        Processor-internal data memory
├─neorv32_gpio.vhd                        General purpose input/output port unit
├─neorv32_imem.vhd                        Processor-internal instruction memory
│  └─neor32_application_image.vhd         IMEM application initialization image
│
├─neorv32_mtime.vhd                       Machine system timer
├─neorv32_nco.vhd                         Numerically-controlled oscillator
├─neorv32_neoled.vhd                      NeoPixel (c) compatible smart LED interface
├─neorv32_pwm.vhd                         Pulse-width modulation controller
├─neorv32_spi.vhd                         Serial peripheral interface controller
├─neorv32_sysinfo.vhd                     System configuration information memory
├─neorv32_trng.vhd                        True random number generator
├─neorv32_twi.vhd                         Two wire serial interface controller
├─neorv32_uart.vhd                        Universal async. receiver/transmitter (UART0,UART1)
├─neorv32_wdt.vhd                         Watchdog timer
└─neorv32_wb_interface.vhd                External (Wishbone) bus interface
```

1.4. FPGA Implementation Results

This chapter shows exemplary implementation results of the NEORV32 CPU and Processor. Please note, that the provided results are just a relative measure as logic functions of different modules might be merged between entity boundaries, so the actual utilization results might vary a bit.

1.4.1. CPU

Implementation results for an `Intel Cyclone IV EP4CE22F17C6N` FPGA using Intel Quartus Prime Lite 20.1 ("balanced implementation, Slow 1200mV 0C Model"). The default configuration of the CPU-related processor generics is assumed (e.g. no physical memory protection, no hardware performance monitors). No constraints were used. Setups with enabled "embedded CPU extension" `E` show the same LUT and FF utilization and identical f_max. However, the size of the register file is cut in half.

Hardware Version:	**1.5.3.2**
Top entity:	**rtl/core/neorv32_cpu.vhd**

CPU	CPU Core Configuration Generics		LEs	FFs	MEM bits	DSPs[2]	f$_{max}$
rv32i	CPU_EXTENSION_RISCV_A = false CPU_EXTENSION_RISCV_B = false CPU_EXTENSION_RISCV_C = false CPU_EXTENSION_RISCV_E = false CPU_EXTENSION_RISCV_M = false CPU_EXTENSION_RISCV_U = false CPU_EXTENSION_RISCV_Zfinx = false CPU_EXTENSION_RISCV_Zicsr = false CPU_EXTENSION_RISCV_Zifencei = false		980	409	1024	0	123 MHz
rv32i + Zicsr	CPU_EXTENSION_RISCV_A = false CPU_EXTENSION_RISCV_B = false CPU_EXTENSION_RISCV_C = false CPU_EXTENSION_RISCV_E = false CPU_EXTENSION_RISCV_M = false CPU_EXTENSION_RISCV_U = false CPU_EXTENSION_RISCV_Zfinx = false CPU_EXTENSION_RISCV_Zicsr = true CPU_EXTENSION_RISCV_Zifencei = false		1835	856	1024	0	124 MHz
rv32im + Zicsr	CPU_EXTENSION_RISCV_A = false CPU_EXTENSION_RISCV_B = false CPU_EXTENSION_RISCV_C = false CPU_EXTENSION_RISCV_E = false CPU_EXTENSION_RISCV_M = true CPU_EXTENSION_RISCV_U = false CPU_EXTENSION_RISCV_Zfinx = false CPU_EXTENSION_RISCV_Zicsr = true CPU_EXTENSION_RISCV_Zifencei = false		2443	1134	1024	0	124 MHz
rv32imc + Zicsr	CPU_EXTENSION_RISCV_A = false CPU_EXTENSION_RISCV_B = false CPU_EXTENSION_RISCV_C = true CPU_EXTENSION_RISCV_E = false CPU_EXTENSION_RISCV_M = true CPU_EXTENSION_RISCV_U = false CPU_EXTENSION_RISCV_Zfinx = false CPU_EXTENSION_RISCV_Zicsr = true CPU_EXTENSION_RISCV_Zifencei = false		2669	1149	1024	0	125 MHz

2 9x9 DSP Elements

CPU	CPU Core Configuration Generics		LEs	FFs	MEM bits	DSPs	f_{max}
rv32imac + Zicsr	CPU_EXTENSION_RISCV_A CPU_EXTENSION_RISCV_B CPU_EXTENSION_RISCV_C CPU_EXTENSION_RISCV_E CPU_EXTENSION_RISCV_M CPU_EXTENSION_RISCV_U CPU_EXTENSION_RISCV_Zfinx CPU_EXTENSION_RISCV_Zicsr CPU_EXTENSION_RISCV_Zifencei	= true = false = true = false = true = false = false = true = false	2685	1156	1024	0	124 MHz
rv32imac + u + Zicsr	CPU_EXTENSION_RISCV_A CPU_EXTENSION_RISCV_B CPU_EXTENSION_RISCV_C CPU_EXTENSION_RISCV_E CPU_EXTENSION_RISCV_M CPU_EXTENSION_RISCV_U CPU_EXTENSION_RISCV_Zfinx CPU_EXTENSION_RISCV_Zicsr CPU_EXTENSION_RISCV_Zifencei	= true = false = true = false = true = true = false = true = false	2698	1162	1024	0	124 MHz
rv32imac + u + Zicsr + Zifencei	CPU_EXTENSION_RISCV_A CPU_EXTENSION_RISCV_B CPU_EXTENSION_RISCV_C CPU_EXTENSION_RISCV_E CPU_EXTENSION_RISCV_M CPU_EXTENSION_RISCV_U CPU_EXTENSION_RISCV_Zfinx CPU_EXTENSION_RISCV_Zicsr CPU_EXTENSION_RISCV_Zifencei	= true = false = true = false = true = true = false = true = true	2715	1162	1024	0	122 MHz
rv32imac + u + Zfinx + Zicsr + Zifencei	CPU_EXTENSION_RISCV_A CPU_EXTENSION_RISCV_B CPU_EXTENSION_RISCV_C CPU_EXTENSION_RISCV_E CPU_EXTENSION_RISCV_M CPU_EXTENSION_RISCV_U CPU_EXTENSION_RISCV_Zfinx CPU_EXTENSION_RISCV_Zicsr CPU_EXTENSION_RISCV_Zifencei	= true = false = true = false = true = true = true = true = true	4004	1812	1024	7	121 MHz

1.4.2. Processor Modules

Implementation results for an `Intel Cyclone IV EP4CE22F17C6N` FPGA using Intel Quartus Prime Lite 20.1 ("balanced implementation"). The timing information is derived from the Timing Analyzer / Slow 1200mV 0C Model. If not other specified, the default configuration of the CPU's generics is assumed. No constraints were used.

Hardware Version:	**1.5.2.4**
Top entity:	**rtl/core/neorv32_top.vhd**

Module	Description	LEs	FFs	MEM bits	DSPs
Boot ROM	Bootloader ROM (4kB)	3	1	32 768	0
BUSSWITCH	Bus mux for CPU instr. and data interfaces	65	8	0	0
iCACHE	Instruction cache (4 blocks, 256 bytes per block)	234	156	8192	0
CFS	Custom functions subsystem[3]	–	–	–	–
DMEM	Processor-internal data memory (8kB)	6	2	65 536	0
GPIO	General purpose input/output ports	67	65	0	0
IMEM	Processor-internal instruction memory (16kB)	6	2	131 072	0
MTIME	Machine system timer	274	166	0	0
NCO	Numerically-controlled oscillator	254	226	0	0
NEOLED	Smart LED Interface (NeoPixel/WS28128) [4xFIFO]	347	309	0	0
PWM	Pulse_width modulation controller	71	69	0	0
SPI	Serial peripheral interface	138	124	0	0
SYSINFO	System configuration information memory	10	10	0	0
TRNG	True random number generator	132	105	0	0
TWI	Two-wire interface	77	44	0	0
UART0/1	Universal asynchronous receiver/transmitter	176	132	0	0
WDT	Watchdog timer	60	45	0	0
WISHBONE	External memory interface	129	104	0	0

Table 1: Hardware utilization by the processor modules

---

3   Hardware requirements for the CFS depends on actual user-defined implementation.

1.4.3. Exemplary Processor Setups

The following table shows exemplary NEORV32 processor implementation results for different FPGA platforms. The processor setup uses **the default peripheral configuration** (like no CFS, no caches and no TRNG), no external memory interface and only internal instruction and data memories. IMEM uses 16kB and DMEM uses 8kB memory space. The setup top entity connects most of the processor's top entity signals to FPGA pins – except for the Wishbone bus and the external interrupt signals. The "default" strategy of each toolchain is used.

```
Hardware Version:     1.4.9.0
CPU Configuration:    rv32i(m)cu + Zicsr + Zifencei + (PMP)
```

Vendor	FPGA	Board	Toolchain	CPU config	LUT / LE	FF / REG	DSP	Embedded memory	f [MHz]
Intel	Cyclone IV EP4CE22F17 C6N	Terasic DE0-Nano	Quartus Prime Lite 20.1	rv32imc + Zicsr + Zifencei + PMP	3813 (17%)	18904 (8%)	0 (0%)	Memory bits: 231424 (38%)	119
Lattice	iCE40 UltraPlus iCE40UP5K-SG48I	Upduino v2.0	Radiant 2.1 (Sinplify Pro)	rv32ic + Zicsr + Zifencei	4397 (83%)	1679 (31%)	0 (0%)	EBR: 12 (40%) SPRAM: 4 (100%)	22.15*
Xilinx	Artix-7 XC7A35TIC SG324-1L	Arty A7-35T	Vivado 2019.2	rv32imc + Zicsr + Zifencei	2465 (12%)	1912 (5%)	0 (0%)	BRAM: 8 (16%)	100*

Table 2: Hardware utilization for different FPGA platforms

Notes

- The Lattice iCE40 UltraPlus setup uses the FPGA's SPRAM memory primitives for the internal IMEM and DEMEM (each 64kb). The according FPGA-specific memory components for the IMEM and DMEM can be found in the `rtl/fpga_specific` folder.

- The clock frequencies marked with an asterisk (*) are constrained clocks. The remaining ones are "f_max" results from the place and route timing reports.

- The Upduino and the Arty board have on-board SPI flash memories for storing the FPGA configuration. These device can also be used by the default NEORV32 bootloader to store and automatically boot an application program after reset (both tested successfully).

- The setups with `PMP` implement 2 regions with a minimal granularity of 64kB.

- No HPM counters are used.

- **Regarding Lattice Radiant:** I have used Lattice Radiant 2.1.0.27.2 to generate the bitstream for the Lattice iCE40 UltraPlus FPGA. I highly encourage you to use *Sinplify Pro* as synthesis engine instead of the default LSE (Lattice Synthesis Engine). The LSE generates slightly faster results, but sometimes LSE results lead to strange behavior of the CPU (like trap codes that are *impossible*)...

1.5. CPU Performance

1.5.1. CoreMark Benchmark

Configuration

Hardware:	32kB IMEM, 16kB DMEM, no caches(!), 100MHz clock
CoreMark:	2000 iteration, MEM_METHOD is MEM_STACK
Compiler:	RISCV32-GCC 10.1.0
Peripherals:	UART for printing the results
Compiler flags:	default, see makefile

The performance of the NEORV32 was tested and evaluated using the CoreMark CPU benchmark. This benchmark focuses on testing the capabilities of the CPU core itself rather than the performance of the whole system. The according source code and the SW project can be found in the `sw/example/coremark` folder.

The resulting *CoreMark score* is defined as CoreMark iterations per second:

The execution time is determined via the RISC-V `[m]cycle[h]` CSRs. The *relative CoreMark score* is defined as CoreMark score divided by the CPU's clock frequency [MHz]:

Results

Hardware Version:	**1.4.9.8**

CPU (incl. `Zicsr`)	Executable Size	Optimization	CoreMark Score	CoreMarks/Mhz
`rv32i`	28 756 bytes	-O3	36.36	**0.3636**
`rv32im`	27 516 bytes	-O3	68.97	**0.6897**
`rv32imc`	22 008 bytes	-O3	68.97	**0.6897**
`rv32imc` + `FAST_MUL_EN`	22 008 bytes	-O3	86.96	**0.8696**
`rv32imc` + `FAST_MUL_EN` + `FAST_SHIFT_EN`	22 008 bytes	-O3	90.91	**0.9091**

Table 3: NEORV32 CoreMark results

> The `FAST_MUL_EN` configuration uses DSPs for the multiplier of the `M` extension (enabled via the `FAST_MUL_EN` generic). The `FAST_SHIFT_EN` configuration uses a barrel shifter for CPU shift operations (enabled via the `FAST_SHIFT_EN` generic).

1.5.2. Instruction Timing

The NEORV32 CPU is based on a multi-cycle architecture. Each instruction is executed in a sequence of several consecutive micro operations. Hence, each instruction requires several clock cycles to execute.

The average CPI (cycles per instruction) depends on the instruction mix of a specific applications and also on the available CPU extensions. The following table shows the performance results for successfully (!) running 2000 CoreMark iterations.

The average CPI is computed by dividing the total number of required clock cycles (only the timed core to avoid distortion due to IO wait cycles) by the number of executed instructions (`[m]instret[h]` CSRs). The executables were generated using optimization `-O3`.

Hardware Version: **1.4.9.8**

CPU	Required Clock Cycles	Executed Instructions	Average CPI
`rv32i`	5 595 750 503	1 466 028 607	**3.82**
`rv32im`	2 966 086 503	598 651 143	**4.95**
`rv32imc`	2 981 786 734	611 814 918	**4.87**
`rv32imc + FAST_MUL_EN`	2 399 234 734	611 814 918	**3.92**
`rv32imc + FAST_MUL_EN + FAST_SHIFT_EN`	2 265 135 174	611 814 948	**3.70**

> ℹ The `FAST_MUL_EN` configuration uses DSPs for the multiplier of the `M` extension (enabled via the `FAST_MUL_EN` generic). The `FAST_SHIFT_EN` configuration uses a barrel shifter for CPU shift operations (enabled via the `FAST_SHIFT_EN` generic).

> ℹ More information regarding the execution time of each implemented instruction can be found in chapter 2.5. Instruction Timing.

2. NEORV32 Central Processing Unit (CPU)

Key Features

- ✔ 32-bit pipelined/multi-cycle in-order RISC-V CPU

- ✔ Optional RISC-V extensions: `rv32[i/e][m][a][c][b]`[4]`[Zfinx] + [u][Zicsr][Zifencei]`

- ✔ Compatible to the RISC-V *user specifications* and a subset of the RISC-V *privileged architecture specifications* – **passes the official RISC-V Architecture Tests (v2+)**

- ✔ Official RISC-V open-source architecture ID

- ✔ Safe execution hardware (see section 2.7. Execution Safety); among other things, the CPU supports *all* traps from the RISC-V specifications (including bus access exceptions) and traps on *all* unimplemented/illegal/malformed instructions

- ✔ Optional physical memory configuration (PMP), compatible to the RISC-V specifications

- ✔ Optional hardware performance monitors (HPM) for application benchmarking

- ✔ Separated interfaces for instruction fetch and data access (merged into single bus via a bus switch for the NEORV32 processor)

- ✔ BIG-endian byte order

- ✔ **No hardware support of unaligned data/instruction accesses** – they will trigger an exception. If the `C` extension is enabled instructions can also be 16-bit aligned and a misaligned instruction address exception is not possible anymore

⚠ See section 2.1. RISC-V Architecture Tests for a list of all
- • RISC-V architecture tests the CPU is passing
- • main non-RISC-V-compatible issues
- • NEORV32-specific custom extensions

ⓘ It is recommended to use the **NEORV32 Processor** setup even if you only want to use the actual CPU. Simply disable all the processor-internal modules via the generics and you will get a "CPU wrapper" that provides a minimal CPU environment and an external bus interface (like AXI4). This setup also allows to further use the default bootloader and software framework. From this base you can start building your own SoC. Of course you can also use the CPU in it's *true* stand-alone mode.

---

4 Work-in-progress

Architecture

The NEORV32 CPU was designed from scratch based only on the official ISA and privileged architecture specifications. The following figure shows the simplified architecture of the CPU.
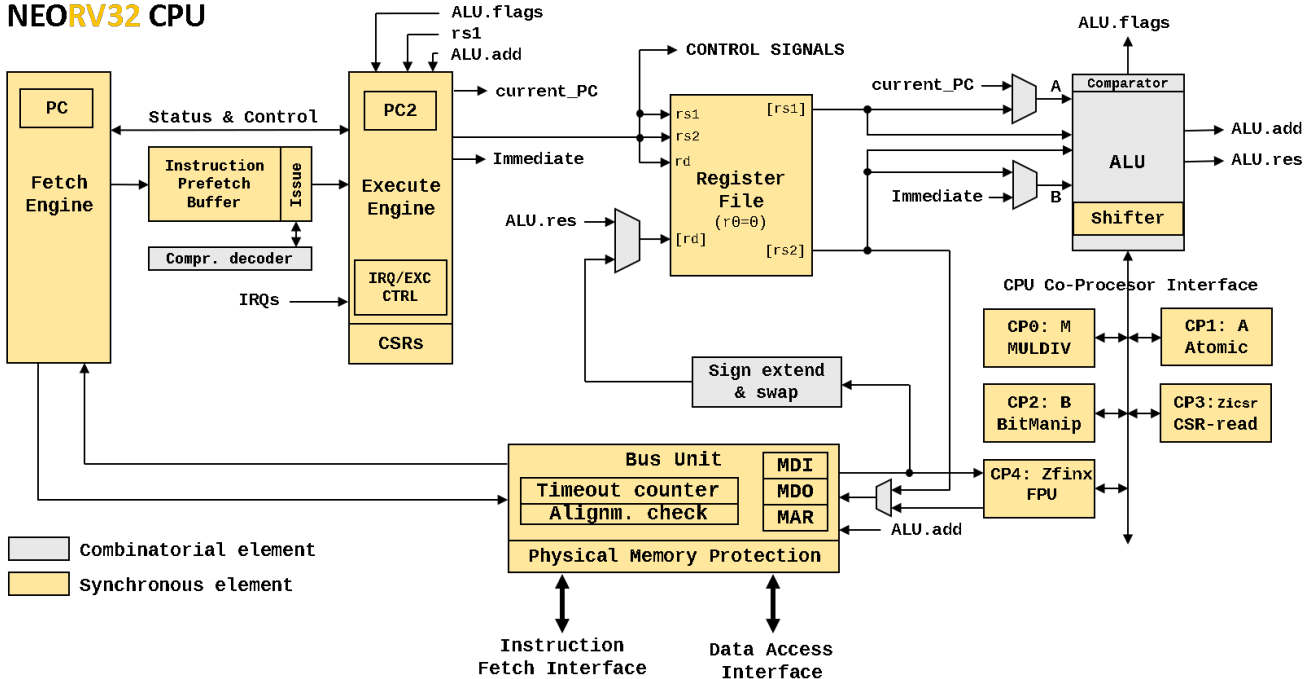
Figure 1: Simplified architecture of the NEORV32 CPU

The CPU uses a pipelined architecture with basically two main stages. The first stage (IF – instruction fetch) is responsible for fetching new instruction data from memory via the *fetch engine*. The instruction data is stored to a FIFO – the instruction *prefetch buffer*. The *issue engine* takes this data and assembles 32-bit instruction words for the next pipeline stage. Compressed instructions – if enabled – are also decompressed in this stage. The second stage (EX – execution) is responsible for actually executing the fetched instructions via the *execute engine*.

These two pipeline stages are based on a multi-cycle processing engine. So the processing of each stage for a certain operations can take several cycles. Since the IF and EX stages are decoupled via the instruction prefetch buffer, both stages can operate in parallel and with overlapping operations. Hence, the optimal CPI (cycles per instructions) is 2, but it can be significantly higher: For instance when executing loads/stores, multi-cycle operations like shifts and multiplications or when the instruction fetch engine has to reload the prefetch buffers due to a taken branch.

Basically, the NEORV32 CPU 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 in a series of consecutive micro-operations. The combination of these two classical design paradigms allows an increased instruction execution in contrast to a pure multi-cycle approach (due to the pipelined approach) at a reduced hardware footprint (due to the multi-cycle approach). This seems to be a quite good trade-off – at least for me.

The CPU provides independent interfaces for instruction fetch and data access. These two bus interfaces are merged into a single processor-internal bus via a bus switch. Hence, memory locations including peripheral devices are mapped to a single 32-bit address space making the architecture a modified Von-Neumann Architecture.

2.1. RISC-V Architecture Tests

The NEORV32 CPU passes the `rv32_m/I`, `rv32_m/M`, `rv32_m/C`, `rv32_m/privilege`, and `rv32_m/Zifencei` tests of the official RISC-V Architecture Tests (GitHub). The port files for the NEORV32 processor are located in `riscv-arch-test` folder. See section **5.14. RISC-V Architecture Test Framework** for more information.

RISC-V `rv32_m/I` Tests

```
Check                   add-01 ... OK
Check                  addi-01 ... OK
Check                   and-01 ... OK
Check                  andi-01 ... OK
Check                 auipc-01 ... OK
Check                   beq-01 ... OK
Check                   bge-01 ... OK
Check                  bgeu-01 ... OK
Check                   blt-01 ... OK
Check                  bltu-01 ... OK
Check                   bne-01 ... OK
Check                 fence-01 ... OK
Check                   jal-01 ... OK
Check                  jalr-01 ... OK
Check              lb-align-01 ... OK
Check             lbu-align-01 ... OK
Check              lh-align-01 ... OK
Check             lhu-align-01 ... OK
Check                   lui-01 ... OK
Check              lw-align-01 ... OK
Check                    or-01 ... OK
Check                   ori-01 ... OK
Check              sb-align-01 ... OK
Check              sh-align-01 ... OK
Check                   sll-01 ... OK
Check                  slli-01 ... OK
Check                   slt-01 ... OK
Check                  slti-01 ... OK
Check                 sltiu-01 ... OK
Check                  sltu-01 ... OK
Check                   sra-01 ... OK
Check                  srai-01 ... OK
Check                   srl-01 ... OK
Check                  srli-01 ... OK
Check                   sub-01 ... OK
Check              sw-align-01 ... OK
Check                   xor-01 ... OK
Check                  xori-01 ... OK
-------------------------------
 OK: 38/38 RISCV_TARGET=neorv32 RISCV_DEVICE=I XLEN=32
```

RISC-V `rv32_m/M` Tests

```
Check                   div-01 ... OK
Check                  divu-01 ... OK
Check                   mul-01 ... OK
Check                  mulh-01 ... OK
Check                mulhsu-01 ... OK
Check                 mulhu-01 ... OK
Check                   rem-01 ... OK
Check                  remu-01 ... OK
-------------------------------
OK: 8/8 RISCV_TARGET=neorv32 RISCV_DEVICE=M XLEN=32
```

RISC-V `rv32_m/C` Tests

```
Check                      cadd-01 ... OK
Check                     caddi-01 ... OK
Check                  caddi16sp-01 ... OK
Check                  caddi4spn-01 ... OK
Check                      cand-01 ... OK
Check                     candi-01 ... OK
Check                     cbeqz-01 ... OK
Check                     cbnez-01 ... OK
Check                   cebreak-01 ... OK
Check                        cj-01 ... OK
Check                      cjal-01 ... OK
Check                     cjalr-01 ... OK
Check                       cjr-01 ... OK
Check                       cli-01 ... OK
Check                      clui-01 ... OK
Check                       clw-01 ... OK
Check                     clwsp-01 ... OK
Check                       cmv-01 ... OK
Check                      cnop-01 ... OK
Check                       cor-01 ... OK
Check                     cslli-01 ... OK
Check                     csrai-01 ... OK
Check                     csrli-01 ... OK
Check                      csub-01 ... OK
Check                       csw-01 ... OK
Check                     cswsp-01 ... OK
Check                       cxor-01 ... OK
-------------------------------
OK: 27/27 RISCV_TARGET=neorv32 RISCV_DEVICE=C XLEN=32
```

RISC-V `rv32_m/privilege` Tests

```
Check                      ebreak ... OK
Check                       ecall ... OK
Check             misalign-beq-01 ... OK
Check             misalign-bge-01 ... OK
Check            misalign-bgeu-01 ... OK
Check             misalign-blt-01 ... OK
Check            misalign-bltu-01 ... OK
Check             misalign-bne-01 ... OK
Check             misalign-jal-01 ... OK
Check              misalign-lh-01 ... OK
Check             misalign-lhu-01 ... OK
Check              misalign-lw-01 ... OK
Check              misalign-sh-01 ... OK
Check              misalign-sw-01 ... OK
Check           misalign1-jalr-01 ... OK
Check           misalign2-jalr-01 ... OK
-------------------------------
OK: 16/16 RISCV_TARGET=neorv32 RISCV_DEVICE=privilege XLEN=32
```

RISC-V `rv32_m/Zifencei` Tests

```
Check                       Fencei ... OK
---------------------------------
OK: 1/1 RISCV_TARGET=neorv32 RISCV_DEVICE=Zifencei XLEN=32
```

2.1.1 RISC-V Incompatibility Issues and Limitations

This list shows the *currently known* issues regarding full RISC-V-compatibility. More specific information can be found in section 2.4. Instruction Sets and CPU Extensions.

⚠ CPU and Processor are BIG-ENDIAN, but this should be no problem as the external memory bus interface provides big- and little-endian configurations. See section 3.5.5. Processor-External Memory Interface (WISHBONE) (AXI4-Lite) for more information.

⚠ The `misa` CSR is read-only. It reflects the *synthesized* CPU extensions. Hence, all implemented CPU extensions are always active and cannot be enabled/disabled dynamically during runtime. Any write access to it (in machine mode) is ignored and will not cause any exception or side-effects.

⚠ The *Physical Memory Protection* (PMP, → 2.4.11. Physical Memory Protection (PMP Extension)) only supports the modes `OFF` and `NAPOT` yet and a minimal granularity of 8 bytes per region.

⚠ The `A` CPU extension (atomic memory access) only implements the `lr.w` and `sc.w` instructions yet. However, these instructions are sufficient to emulate all further AMO operations.

2.1.2 NEORV32-Specific (Custom) Extensions

The NEORV32-specific extensions are always enabled and are indicated by the set `X` bit in the `misa` CSR.

ⓘ The CPU provides eight "fast interrupt" interrupts, which are controlled via custom bit in the `mie` and `mip` CSR. This extension is mapped to bits, that are available for custom use (according to the RISC-V specs). Also, custom trap codes for `mcause` are provided.

ⓘ A custom CSR (`mzext`) is available that can be used to check for implemented Z* CPU extensions (for example `Zifencei`). This CSR is mapped to the official "custom CSR address region".

ⓘ All undefined/unimplemented/malformed/illegal instructions do raise an illegal instruction exception (to increase 2.7. Execution Safety).

2.2. CPU Top Entity – Signals

The following table shows all interface ports of the CPU top entity (`rtl/core/neorv32_cpu.vhd`). The type of all signals is **std_ulogic** or **std_ulogic_vector**, respectively.

Signal Name	Width	Direction	Function	HW Module
Global Control				
clk_i	1	Input	Global clock line, all registers triggering on rising edge	global
rstn_i	1	Input	Global reset, **low-active**	
sleep_o	1	Output	CPU is in sleep mode when set	CONTROL
Instruction Bus Interface				
i_bus_addr_o	32	Output	Destination address	
i_bus_rdata_i	32	Input	Write data	
i_bus_wdata_o	32	Output	Read data	
i_bus_ben_o	4	Output	Byte enable	
i_bus_we_o	1	Output	Write transaction	
i_bus_re_o	1	Output	Read transaction	
i_bus_cancel_o	1	Output	Cancel current transfer	BUS_UNIT
i_bus_ack_i	1	Input	Bus transfer acknowledge from accessed peripheral	
i_bus_err_i	1	Input	Bus transfer terminate from accessed peripheral	
i_bus_fence_o	1	Output	Indicates an executed **FENCE.I** instruction	
i_bus_priv_o	2	Output	Current CPU privilege level	
Data Bus Interface				
d_bus_addr_o	32	Output	Destination address	
d_bus_rdata_i	32	Input	Write data	
d_bus_wdata_o	32	Output	Read data	
d_bus_ben_o	4	Output	Byte enable	
d_bus_we_o	1	Output	Write transaction	
d_bus_re_o	1	Output	Read transaction	
d_bus_cancel_o	1	Output	Cancel current transfer	BUS_UNIT
d_bus_ack_i	1	Input	Bus transfer acknowledge from accessed peripheral	
d_bus_err_i	1	Input	Bus transfer terminate from accessed peripheral	
d_bus_fence_o	1	Output	Indicates an executed **FENCE** instruction	
d_bus_priv_o	2	Output	Current CPU privilege level	
d_bus_excl_o	1	Output	Exclusive bus access request	
d_bus_excl_i	1	Input	Exclusive bus access successful	
System Time				
time_i	64	Input	System time input (from MTIME)	CONTROL
Interrupts (RISC-V-compatible)				
msw_irq_i	1	Input	RISC-V machine software interrupt	
mext_irq_i	1	Input	RISC-V machine external interrupt	CONTROL
mtime_irq_i	1	Input	RISC-V machine timer interrupt	
Fast Interrupts (custom extension)				
firq_i	16	Input	Fast interrupt request signals	CONTROL
firq_ack_o	6	Output	Fast interrupt acknowledge signals	

Table 4: neorv32_cpu.vhd – CPU top entity interface ports

2.3. CPU Top Entity – Configuration Generics

The *CPU top module* (`rtl/neorv32_cpu.vhd`) provides a <u>subset</u> of the configuration generics that are provided by the *Processor top module* (`rtl/neorv32_top.vhd`). See section 3.2. Processor Top Entity – Configuration Generics for a complete list of all configuration generics.

> **ⓘ** Privileged software can determine the actual processor/CPU configuration via the `misa` and `mzext` control and status register (→ section 2.6. Control and Status Registers (CSRs)) and via the memory-mapped SYSINFO (→ section 3.5.18. System Configuration Information Memory (SYSINFO)) memory, respectively.

2.4. Instruction Sets and CPU Extensions

The NEORV32 is an RISC-V `rv32i` architecture that provides several <u>optional</u> RISC-V CPU and ISA (instruction set architecture) extensions. For more information regarding the RISC-V ISA extensions please see the *The RISC-V Instruction Set Manual – Volume I: Unprivileged ISA*, which is available in the `docs/` folder.

2.4.1. Atomic Memory Access Instructions (A Extension)

Atomic memory access instructions (for implementing semaphores and mutexes) are available when the `CPU_EXTENSION_RISCV_A` configuration generic is `true`. In this case the following additional instructions are available:

- `LR.W` (load-reservate)
- `SC.W` (store-conditional)

> **ⓘ** Even though only `LR.W` and `SC.W` instructions are implemented yet, all further atomic operations (load-modify-write instruction) can be emulated using these two instruction. Furthermore, the instruction's ordering flags (*aq* and *lr*) are ignored by the CPU hardware. Using any other (not yet implemented) AMO (atomic memory operation) will trigger an illegal instruction exception.

> **ⓘ** The atomic instructions have special requirements for memory system / bus interconnect. More information can be found in sections 2.10. Bus Interface and 3.5.5. Processor-External Memory Interface (WISHBONE) (AXI4-Lite), respectively.

2.4.2. Bit Manipulation Instructions (B Extension)

Bit manipulation instructions (**Zbb**, **Zbs** and **Zbb** subsets only yet) are available when the `CPU_EXTENSION_RISCV_B` configuration generic is `true`. In this case the following instructions are available:

- Base subset **Zbb**: `CLZ` `CTZ` `CPOP` `SEXT.B` `SEXT.H` `MIN[U]` `MAX[U]` `ANDN` `ORN` `XNOR` `ROL` `ROR` `RORI` `C.XOR` `zext` (pseudo instruction for `PACK rd, rs, zero`) `rev8` (pseudo instruction for `GREVI rd, rs, -8`) `orc.b` (pseudo instruction for `GORCI rd, rs, 7`)

- Single-bit operations **Zbs**: `SBSET[I]` `SBCLR[I]` `SBCLR[I]` `SBEXT[I]`

- shifted-add operations **Zba**: `SH1ADD` `SH2ADD` `SH3ADD`

> ⚠ **The bit manipulation extension is not yet officially ratified**, but is expected to stay unchanged. There is no software support in the upstream GCC RISC-V port yet. However, an **intrinsic library** is provided to utilize the provided bit manipulation extension from C-language code (see `sw/example/bit_manipulation`).

> ⚠ The current version of the bit manipulation specs that are supported by the NEORV32 can be found in `docs/bitmanip-draft.pdf`.

2.4.3. Compressed Instructions (C Extension)

Compressed 16-bit instructions are available when the `CPU_EXTENSION_RISCV_C` configuration generic is `true`. In this case the following instructions are available:

- `C.ADDI4SPN` `C.LW` `C.SW` `C.NOP` `C.ADDI` `C.JAL` `C.LI` `C.ADDI16SP` `C.LUI` `C.SRLI` `C.SRAI` `C.ANDI` `C.SUB` `C.XOR` `C.OR` `C.AND` `C.J` `C.BEQZ` `C.BNEZ` `C.SLLI` `C.LWSP` `C.JR` `C.MV` `C.EBREAK` `C.JALR` `C.ADD` `C.SWSP`

> ℹ When the compressed instructions extension is enabled branches to an unaligned uncompressed (i.e. 32-bit) require an additional instruction fetch to load the required second half-word of that instruction. The performance can be increased by forcing a 32-bit alignment of branch target addresses. By default, this is enforced via the GCC `-falign-functions=4` `-falign-labels=4` `-falign-loops=4` `-falign-jumps=4` flags (via the makefile).

2.4.4. Embedded CPU Architecture (E Extension)

This extensions does not feature additional instructions or functions. However, the embedded CPU version reduces the general purpose register file from 32 entries to 16 entries to reduce hardware requirements. This extensions is enabled when the `CPU_EXTENSION_RISCV_E` configuration generic is `true`.

Due to the reduced register file an alternate ABI (`ilp32e`) is required for the toolchain.

2.4.5. 32-bit Base ISA (I Extension)

The CPU supports the complete RV32I base integer instruction set. This "extensions" is always enabled regardless of the setting of the remaining exceptions. The base instruction set includes the following instructions:

- Immediates: `LUI` `AUIPC`
- Jumps: `JAL` `JALR`
- Branches: `BEQ` `BNE` `BLT` `BGE` `BLTU` `BGEU`
- Memory: `LB` `LH` `LW` `LBU` `LHU` `SB` `SH` `SW`
- ALU: `ADDI` `SLTI` `SLTIU` `XORI` `ORI` `ANDI` `SLLI` `SRLI` `SRAI` `ADD` `SUB` `SLL` `SLT` `SLTU` `XOR` `SRL` `SRA` `OR` `AND`
- Environment: `ECALL` `EBREAK` `FENCE`

> ℹ️ In order to keep the hardware footprint low, the CPU's shift unit uses a bit-serial approach. Shift operations are split in coarse shifts (multiples of 4) and a final fine shift (0 to 3). The total execution time depends on the shift amount. Alternatively, the shift operations can be processed using a fast (but large) barrel shifter when the `FAST_SHIFT_EN` generic is `true`. In that case, shift operations complete within 2 cycles regardless of the shift amount.

> ℹ️ Internally, the `FENCE` instruction does not perform any operation inside the CPU. It only sets the top's `fence_o` signal high for one cycle to inform the memory system a `FENCE` instruction has been executed. Any flags *within* the `FENCE` instruction word are ignore by the hardware.

2.4.6. Integer Multiplication and Division (M Extension)

Hardware-accelerated integer multiplication and division instructions are available when the `CPU_EXTENSION_RISCV_M` configuration generic is `true`. In this case the following instructions are available:

- Multiplication: `MUL` `MULH` `MULHSU` `MULHU`
- Division: `DIV` `DIVU` `REM` `REMU`

> ℹ️ By default, multiplication and division operations are executed in a bit-serial approach. Alternatively, the multiplier core can be implemented using DSP blocks if the `FAST_MUL_EN` generic is `true`. In this case, multiplications complete within 6 cycles. Multiplications and divisions always require a fixed amount of cycles to complete – regardless of the input operands.

2.4.7. User Privilege Level (U Extension)

Adds the less-privileged *user mode* when the `CPU_EXTENSION_RISCV_U` configuration generic is `true`. For instance, use-level code cannot access machine-mode CSRs. Furthermore, access to the address space (like peripheral/IO devices) can be limited via the physical memory protection unit for code running in user mode.

2.4.8. Single-Precision Floating-Point Extension (`Zfinx` Extension)

The `Zfinx` floating-point extension is an alternative of the `F` floating-point instruction that also uses the integer register file `x` to store and operate on floating-point data (hence, `F-in-x`). Since not dedicated `f` register file exists, 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 official RISC-V specifications can be found here: github.com/riscv/riscv-zfinx

The `Zfinx` extensions only supports single-precision (`.S` suffix) yet (so it is a *direct alternative* to the `F` extension). The `Zfinx` extension is implemented when the `CPU_EXTENSION_RISCV_Zfinx` configuration generic is `true`. In this case the following instructions and CSRs are available:

- Conversion: `FCVT.S.W FCVT.S.WU FCVT.W.S FCVT.WU.S`
- Comparison: `FMIN.S FMAX.S FEQ.S FLT.S FLE.S`
- Computational: `FADD.S FSUB.S FMUL.S`
- Sign-injection: `FSGNJ.S FSGNJN.S FSGNJX.S`
- Number classification: `FCLASS.S`

- CSRs: `fcsr` `frm` `fflags`

⚠ Fused multiply-add instructions (`F[N]M[ADD/SUB].S`) are not supported!
Division (`FDIV`) and square root (`FSQRT`) instructions are not supported yet!

ⓘ The NEORV32 floating-point unit (FPU) used by the `Zfinx` extension is compatible to the IEEE-754 specifications.

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

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

2.4.9. Control and Status Register Access (Zicsr Extension)

The CSR access instructions as well as the exception and interrupt system are implemented when the `CPU_EXTENSION_RISCV_Zicsr` configuration generic is `true`. In this case the following instructions are available:

- CSR access: `CSRRW` `CSRRS` `CSRRC` `CSRRWI` `CSRRSI` `CSRRCI`
- Environment: `MRET` `WFI`

⚠️ If the `Zicsr` extension is disabled the CPU does **not** provide any kind of interrupt or exception support at all. In order to provide the full spectrum of functions and to allow a secure executions environment, the e `Zicsr` extension should always be enabled.

ℹ️ The "wait for interrupt instruction" `WFI` works like a *sleep* command. When executed, the CPU is halted until a valid **interrupt** request occurs. To wake up again, the according interrupt source has to be enabled via the `mie` CSR and the global interrupt enable flag in `mstatus` has to be set.

2.4.10. Instruction Coherency Operation (Zifencei Extension)

The `Zifencei` CPU extension is implemented if the `CPU_EXTENSION_RISCV_Zifencei` configuration generic is `true`. It allows manual synchronization of the instruction stream.

- `FENCE.I`

ℹ️ The `FENCE.I` instruction resets the CPU's instruction fetch engine and flushes the prefetch buffer. This allows a clean re-fetch of modified data from memory. Also, he top's `fencei_o` signal is set high for one cycle to inform the memory system. Any additional flags within the `FENCE.I` instruction word are ignore by the hardware.

2.4.11. Physical Memory Protection (PMP Extension)

The NEORV32 physical memory protection (PMP) is compatible to the PMP specified by the RISC-V specs. The CPU PMP only supports `NAPOT` mode yet and a minimal region size (granularity) of 8 bytes (configured via the `PMP_MIN_GRANULARITY` generic). The physical memory protection system is implemented when the `PMP_NUM_REGIONS` configuration generic is >0. In this case the following additional CSRs are available:

- CSRs: `pmpcfg*` (0..15, depending on configuration) `pmpaddr*` (0..63, depending on configuration)

See section 2.6.4. Machine Physical Memory Protection for more information regarding the PMP CSRs.

Configuration

The actual number of regions and the minimal region granularity are defined via the top entity generics `PMP_MIN_GRANULARITY` and `PMP_NUM_REGIONS`. `PMP_MIN_GRANULARITY` defines the minimal available granularity of each region in bytes. `PMP_NUM_REGIONS` defines the implemented regions and thus, the available `pmpcfg*` and `pmpaddr*` CSRs.

> ⚠️ **When implementing <u>more</u> PMP regions that a certain *critical limit* an additional register stage is automatically inserted into the CPU's memory interfaces increasing the latency of instruction fetches and data access by +1 cycle.**
> The *critical limit* can be adapted for custom use by a constant from the main VHDL package file (`rtl/core/neorv32_package.vhd`). The default value is 8.
>
> ```
> -- "critical" number of PMP regions --
> constant pmp_num_regions_critical_c : natural := 8;
> ```

Operation

Any memory access address (from the CPU's instruction fetch or data access interface) is tested for accessing any of the specified (configured via `pmpaddr*` and enabled via `pmpcfg*`) PMP regions. If an address accesses one of these regions, the configures access rights (*attributes* via `pmpcfg*`) are checked:

- A write access (store) will fail if no write-attribute is set
- A read access (load) will fail if no read-attribute is set
- An instruction fetch access will fail if no execute-attribute is set

An illegal access to a protected region will trigger the according instruction/load/store *access fault* exception.

By default, all PMP checks are enforced for user-level programs only. If you wish to enforce the physical memory protection also for machine-level programs you need to active the **locked bit** in the according `pmpcfg` configuration.

> ⚠️ After updating the address configuration registers `pmpaddr*` the system requires up to 33 cycles for internal (iterative) computations before the configuration becomes valid.

> ℹ️ For more information regarding RISC-V physical memory protection see the official *The RISC-V Instruction Set Manual – Volume II: Privileged Architecture*.

2.4.12. Hardware Performance Monitors (HPM Extension)

The CPU provides up to 29 hardware performance monitors (HPM 3..31), which can be used to benchmark applications. Each HPM consists of an N-bit counter (split in two CSRs), where N is defined via the `HPM_CNT_WIDTH` generic (1..64-bit), and a corresponding event configuration CSR. The event configuration CSR defines the architectural events that lead to an increment of the associated HPM counter.

The cycle, time and instructions-retired counters (`[m]cycle[h]`, `time[h]`, `[m]instret[h]`) are mandatory performance monitors on every RISC-V platform and have fixed increment event. For example, the instructions-retired counter increments with each executed instructions. The actual hardware performance monitors are optional and can be configured to increment on arbitrary hardware events. The number of available HPM is configured via the `HPM_NUM_CNTS` generic at synthesis time. Assigning a zero will exclude all HPM logic from the design.

Depending on the configuration, the following additional CSR are available:

- `Counters:` `[m]hpmcounter*[h]` (3..31, depending on configuration)
- `Event configuration:` `mhpmevent*` (3..31, depending on configuration)

User-level access to the counter registers (`hpmcounter*[h]`) can be restricted via the `mcounteren` CSR. Auto-increment of the HPMs can be deactivated via the `mcountinhibit` CSR.

If `HPM_NUM_CNTS` is lower than the maximumg value (=29) the remaining HPMs are not implemented. However, accessing their associated CSRs will not trigger an illegal instructions exception. These CSR are read-only and will always return 0.

> For a list of all allocated HPM-related CSRs and all provided event configurations see section 2.6.6. Hardware Performance Monitors (HPM).

2.5. Instruction Timing

i The instruction timing listed in the table shows the required clock cycles for executing a certain instruction. These instruction cycles assume a bus access without additional wait states and a filled pipeline.

i **Average CPI for "Real" Applications**
The average CPI (cycles per instructions) for executing the CoreMark benchmark for different CPU configurations is presented in chapter 1.5.2. Instruction Timing.

Class	ISA / Extension	Instruction(s)	Execution Cycles
ALU	I/E	ADDI SLTI SLTIU XORI ORI ANDI ADD SUB SLT SLTU XOR OR AND LUI AUIPC	2
	C	C.ADDI4SPN C.NOP C.ADDI C.LI C.ADDI16SP C.LUI C.ANDI C.SUB C.XOR C.OR C.AND C.ADD C.MV	
	I/E	SLLI SRLI SRAI SLL SRL SRA	$3 + SA^5/4 + SA\%4$
	C	C.SRLI C.SRAI C.SLLI	***FAST_SHIFT[6]***: 3
Branches	I/E	BEQ BNE BLT BGE BLTU BGEU	Taken: $5 + ML^7$
	C	C.BEQZ C.BNEZ	Not taken: 3
Jumps / Calls	I/E	JAL JALR	$5 + ML$
	C	C.JAL C.J C.JR C.JALR	
Memory access	I/E	LB LH LW LBU LHU SB SH SW	$4 + ML$
	C	C.LW C.SW C.LWSP C.SWSP	
	A	LR.W SC.W	
Multiplication	M	MUL MULH MULHSU MULHU	$2 + 32 + 4$ ***FAST_MUL[8]: 5***
Division		DIV DIVU REM REMU	$2 + 32 + 6$
Bit manipulation – arithmetic/logic	B (Zbb)	SEXT.B SEXT.H MIN MINU MAX MAXU ANDN ORN XNOR zext(PACK) rev8(GREVI) orc.b(GORCI)	3
Bit manipulation – shifts		CLZ CTZ	$3 + BP^9$
		CPOP	$3 + 32$
		ROL ROR RORI	$3 + SA$
Single-bit operations	B (Zbs)	SBSET[I] SBCLR[I] SBINV[I] SBEXT[I]	3

5 *SA* = Shift amount, 0..31(immediate or register value)
6 Using a fast (but huge) barrel shifter for the CPU's shift operations; enabled via top's `FAST_SHIFT_EN` generic
7 *ML* = memory latency; all processor-internal memories and IO devices have 1 cycle access latency (ML = 1); branches / jumps /calls to 32-bit instructions placed on unaligned addresses (= not 32-bit aligned) require an extra instruction fetch (= plus 1+ML cycles)
8 Using DSPs for multiplication; enabled via top's `FAST_MUL_EN` generic
9 *BP* = Position (0..32 (!)) of first set bit starting from the LSB (for CTZ) / MSB (for CLZ)

Class	ISA / Extension	Instruction(s)	Execution Cycles
Shifted-add operations	`B (Zba)`	`SH1ADD SH2ADD SH3ADD`	3
CSR Access	`Zicsr`	`CSRRW CSRRS CSRRC CSRRWI CSRRSI CSRRCI`	4
System	`I/E+Zicsr`	`ECALL EBREAK`	4
	`I/E`	`FENCE`	3
	`C+Zicsr`	`C.EBREAK`	4
	`Zicsr`	`MRET WFI`	5
	`Zifencei`	`FENCE.I`	3

Floating-Point arithmetic operations	`Zfinx`	`FADD.S`
		`FSUB.S`
		`FMUL.S`
			110
			112
			22

Floating-Point comparison		`FMIN.S FMAX.S FEQ.S FLT.S FLE.S`	13

Floating-Point sign-injection, classification		`FSGNJ.S FSGNJN.S FSGNJX.S`
		
		`FCLASS.S`
			12

Floating-Point conversion		`FCVT.W.S FCVT.WU.S` `FCVT.S.W FCVT.S.WU`	47 48

Table 5: Clock cycles per instruction (optimal)

> ℹ️ The presented values of the **floating-point execution cycles** are average values – obtained from 4096 instruction executions using pseudo-random input values. The execution time for emulating the instructions (using pure-software libraries) is ~17..140 times higher.

2.6. Control and Status Registers (CSRs)

⚠️ The CSRs, the CSR-related instructions as well as the complete exception/interrupt processing system are only available when the `CPU_EXTENSION_RISCV_Zicsr` generic is `true`.

The following table shows a summary of all available CSRs. The address field defines the CSR address for the CSR access instructions. The [ASM] name can be used for (inline) assembly code and is directly understood by the assembler/compiler. The [C] names are defined by the NEORV32 core library and can be used as immediates in plain C code. The "R/W" column shows whether the CSR can be read and/or written.

The NEORV32-specific CSRs are mapped to the official "*custom CSRs*" CSR address space.

ℹ️ When trying to write to a read-only CSR (like the `time` CSR), when trying to access a non-existent CSR or when trying to access a "machine" CSR from user-mode an illegal instruction exception is triggered.

⚠️ **Reset value:** Please note that not all of the CSRs provide a dedicated hardware reset. Hence, these CSRs are not initialized by a hardware reset and keep an "UNDEFINED" value until they are explicitly initialized by the software (normally this is already done by the NEORV32-specific crt0.S start-up code).

CSR Listing

*Notes for the following listing:

C   CSRs with this note have or are a custom CPU extension (that is allowed by the RISC-V specs)

R   This note indicates that a CSR is read-only (in contrast to the originally specified r/w capability)

S   CSRs with this node have a constrained compatibility; for example not all specified bits are available

Address	Name [ASM]	Name [C]	R/W	Function	*
User Floating-Point CSRs					
0x001	fflags	CSR_FFLAGS	r/w	Floating-point accrues exceptions	
0x002	frm	CSR_FRM	r/w	Floating-point dynamic rounding mode	
0x003	fcsr	CSR_FCSR	r/w	Floating-point control and status (frm + fflags)	
Machine Trap Setup					
0x300	mstatus	CSR_MSTATUS	r/w	Machine status register	S
0x301	misa	CSR_MISA	r/-	Machine CPU ISA and extensions	R
0x304	mie	CSR_MIE	r/w	Machine interrupt enable register	C
0x305	mtvec	CSR_MTVEC	r/w	Machine trap-handler base address (for ALL traps)	
0x306	mcounteren	CSR_MCOUNTEREN	r/w	Machine counter-enable register	S
0x310	mstatush	CSR_MSTATUSH	r/-	Machine status register – high word	S R
Machine Trap Handling					
0x340	mscratch	SCR_MSCRATCH	r/w	Machine scratch register	
0x341	mepc	CSR_MEPC	r/w	Machine exception program counter	
0x342	mcause	CSR_MCAUSE	r/w	Machine trap cause	C
0x343	mtval	CSR_MTVAL	r/w	Machine bad address or instruction	
0x344	mip	CSR_MIP	r/w	Machine interrupt pending register	C
Machine Physical Memory Protection					
0x3a0	pmpcfg0	CSR_PMPCFG0	r/w	Physical memory protection config. for region 0..3	S
...	...	...	...	...	
0x3af	pmpcfg15	CSR_PMPCFG15	r/w	Physical memory protection config. for region 60..63	S
0x3b0	pmpaddr0	CSR_PMPADDR0	r/w	Physical memory protection addr. register region 0	
...	...	...	...	...	
0x3ef	pmpaddr63	CSR_PMPADDR63	r/w	Physical memory protection addr. register region 63	
[Machine] Counters and Timers					
0xb00	mcycle	CSR_MCYCLE	r/w	Machine cycle counter low word	
0xb02	minstret	CSR_MINSTRET	r/w	Machine instructions-retired counter low word	
0xb03	mhpmcounter3	CSR_MHPMCOUNTER3	r/-	Machine performance-monitoring counter 3 low word	
...	...	...	...	...	
0xb1f	mhpmcounter31	CSR_MHPMCOUNTER31	r/-	Machine performance-monitoring counter 31 low word	

Address	Name [ASM]	Name [C]	R/W	Function	*
0xb80	mcycleh	CSR_MCYCLEH	r/w	Machine cycle counter low word	
0xb82	minstreth	CSR_MINSTRETH	r/w	Machine instructions-retired counter high word	
0xb83	mhpmcounter3h	CSR_MHPMCOUNTER3H	r/-	Machine performance-monitoring counter 3 high word	
...	...	...	...	...	
0xb9f	mhpmcounter31h	CSR_MHPMCOUNTER31H	r/-	Machine performance-monitoring counter 31 high word	
0xc00	cycle	CSR_CYCLE	r/-	Cycle counter low word	
0xc01	time	CSR_TIME	r/-	System time (from MTIME) low word	
0xc02	instret	CSR_INSTRET	r/-	Instructions-retired counter low word	
0xc03	hpmcounter3	CSR_HPMCOUNTER3	r/-	Performance-monitoring counter 3 low word	
...	...	...	...	...	
0xc1f	hpmcounter31	CSR_HPMCOUNTER31	r/-	Performance-monitoring counter 31 low word	
0xc80	cycleh	CSR_CYCLEH	r/-	Cycle counter high word	
0xc81	timeh	CSR_TIMEH	r/-	System time (from MTIME) high word	
0xc82	instreth	CSR_INSTRETH	r/-	Instructions-retired counter high word	
0xc83	hpmcounter3h	CSR_HPMCOUNTER3H	r/-	Performance-monitoring counter 3 high word	
...	...	...	...	...	
0xc9f	hpmcounter31h	CSR_HPMCOUNTER31H	r/-	Performance-monitoring counter 31 high word	
Machine Counter Setup					
0x320	mcountinhibit	CSR_MCOUNTINHIBIT	r/w	Machine counter-enable register	S
0x323	mhpmevent3	CSR_MHPMEVENT3	r/w	Machine performance-monitoring event selector 3	C
...	...	...	...	...	C
0x33f	mhpmevent3	CSR_MHPMEVENT31	r/w	Machine performance-monitoring event selector 31	C
Machine Information Registers, read-only					
0xf11	mvendorid	CSR_MVENDORID	r/-	Vendor ID	
0xf12	marchid	CSR_MARCHID	r/-	Architecture ID	
0xf13	mimpid	CSR_MIMPID	r/-	Machine implementation ID / version	
0xf14	mhartid	CSR_MHARTID	r/-	Machine thread ID	
NEORV32-Specific Custom Machine CSRs					
0xfc0	-	CSR_MZEXT	r/-	Available Z* CPU extensions	C

Table 6: NEORV32 Control and Status Registers (CSRs)

Not Implemented CSRs / CSR Bits

All CSR bits that are unused / not implemented / not shown are hardwired to zero. All CSRs that are not implemented at all (and are not "disabled" using certain configuration generics) will trigger an exception on access. The CSR that are implemented within the NEORV32 might cause an exception if they are disabled. See the according CSR description for more information.

2.6.1. User Floating-Point CSRs

These CSRs are available if the `Zfinx` extensions is enabled (`CPU_EXTENSION_RISCV_Zfinx` is `true`). Otherwise any access to the floating-point CSRs will raise an illegal instruction exception.

Floating-Point Accrued Exceptions   `fflags`

```
CSR address:   0x001
Reset value:   UNDEFINED
```

The `fflags` CSR is compatible to the RISC-V specifications. It shows the accrued ("accumulated") exception flags in the lowest 5 bits. This CSR is only available if a floating-point CPU extension is enabled. See the RISC-V ISA spec for more information.

Floating-Point Dynamic Rounding Mode   `frm`

```
CSR address:   0x002
Reset value:   UNDEFINED
```

The `frm` CSR is compatible to the RISC-V specifications and is used to configure the rounding modes using the lowest 3 bits. This CSR is only available if a floating-point CPU extension is enabled. See the RISC-V ISA spec for more information.

Floating-Point Control and Status Register   `fcsr`

```
CSR address:   0x003
Reset value:   UNDEFINED
```

The `fcsr` CSR is compatible to the RISC-V specifications. It provides combined read/write access to the `fflags` and `frm` CSRs. This CSR is only available if a floating-point CPU extension is enabled. See the RISC-V ISA spec for more information.

2.6.2. Machine Trap Setup

Machine Status Register – Low Word	mhpmcounter31h

```
CSR address:   0x300
Reset value:   0x00000000
```

The `mstatus` CSR is compatible to the RISC-V specifications. It shows the CPU's current execution state. The following bits are implemented (all remaining bits are always zero and are read-only):

Bit#	Name [C]	R/W	Function
12:11	CSR_MSTATUS_MPP_H CSR_MSTATUS_MPP_L	r/w	Previous machine privilege level, 11 = machine (M) level, 00 = user (U) level
7	CSR_MSTATUS_MPIE	r/w	Previous machine global interrupt enable flag state
6	CSR_MSTATUS_UBE	r/-	User-mode byte-order (Endianness) for load/Store operations, always set indicating BIG-endian byte-order (copy of `CSR_MSTATUSH_MBE`); bit is always zero if user-mode is not implemented
3	CSR_MSTATUS_MIE	r/w	Machine global interrupt enable flag

When entering an exception/interrupt, the `MIE` flag is copied to `MPIE` and cleared afterwards. When leaving the exception/interrupt (via the `MRET` instruction), `MPIE` is copied back to `MIE`.

```
CSR address:  0x301
Reset value:  -
```

⚠️ The `misa` CSR is not fully RISC-V-compatible as it is read-only. Hence, implemented CPU extensions cannot be switch on/off during runtime. For compatibility reasons any write access to this CSR is simply ignored and will **NOT** cause an illegal instruction exception.

The `misa` CSR gives information about the actual CPU features. The lowest 26 bits show the implemented CPU extensions. The following bits are implemented (all remaining bits are always zero and are read-only):

Bit#	Name [C]	R/W	Function
31:30	CSR_MISA_MXL_HI_EXT CSR_MISA_MXL_LO_EXT	r/-	32-bit architecture indicator (always "01")
23	CSR_MISA_X_EXT	r/-	The **X** extension bit is always set to indicate custom non-standard extensions
20	CSR_MISA_U_EXT	r/-	**U** CPU extension (user mode) available, set when `CPU_EXTENSION_RISCV_U` enabled
12	CSR_MISA_M_EXT	r/-	**M** CPU extension (muld/div HW) available, set when `CPU_EXTENSION_RISCV_M` enabled
8	CSR_MISA_I_EXT	r/-	**I** CPU extension, always set, cleared when `CPU_EXTENSION_RISCV_E` enabled
4	CSR_MISA_E_EXT	r/-	**E** CPU extension (embedded) available, set when `CPU_EXTENSION_RISCV_E` enabled
2	CSR_MISA_C_EXT	r/-	**C** CPU extension (compressed instructions) available, set when `CPU_EXTENSION_RISCV_C` enabled
1	CSR_MISA_B_EXT	r/-	**B** CPU extension (bit manipulation instructions) available, set when `CPU_EXTENSION_RISCV_B` enabled
0	CSR_MISA_A_EXT	r/-	**A** CPU extension (atomic memory access instructions) available, set when `CPU_EXTENSION_RISCV_A` enabled

ℹ️ Information regarding the available RISC-V `Z*` *sub-extensions* (like `Zicsr` or `Zfinx`) can found in the `mzext` CSR.

Machine Interrupt-Enable Register   `mie`

```
CSR address:   0x304
Reset value:   UNDEFINED
```

The `mie` CSR is compatible to the RISC-V specifications and features custom extensions for the fast interrupt channels. It is used to enabled specific interrupts sources. Please note that interrupts also have to be globally enabled via the `CSR_MSTATUS_MIE` flag of the `mstatus` CSR. The following bits are implemented (all remaining bits are always zero and are read-only):

Bit#	Name [C]	R/W	Function
31	CSR_MIE_FIRQ15E	r/w	Fast interrupt channel 15 enable
30	CSR_MIE_FIRQ14E	r/w	Fast interrupt channel 14 enable
29	CSR_MIE_FIRQ13E	r/w	Fast interrupt channel 13 enable
28	CSR_MIE_FIRQ12E	r/w	Fast interrupt channel 12 enable
27	CSR_MIE_FIRQ11E	r/w	Fast interrupt channel 11 enable
26	CSR_MIE_FIRQ10E	r/w	Fast interrupt channel 10 enable
25	CSR_MIE_FIRQ9E	r/w	Fast interrupt channel 9 enable
24	CSR_MIE_FIRQ8E	r/w	Fast interrupt channel 8 enable
23	CSR_MIE_FIRQ7E	r/w	Fast interrupt channel 7 enable
22	CSR_MIE_FIRQ6E	r/w	Fast interrupt channel 6 enable
21	CSR_MIE_FIRQ5E	r/w	Fast interrupt channel 5 enable
20	CSR_MIE_FIRQ4E	r/w	Fast interrupt channel 4 enable
19	CSR_MIE_FIRQ3E	r/w	Fast interrupt channel 3 enable
18	CSR_MIE_FIRQ2E	r/w	Fast interrupt channel 2 enable
17	CSR_MIE_FIRQ1E	r/w	Fast interrupt channel 1 enable
16	CSR_MIE_FIRQ0E	r/w	Fast interrupt channel 0 enable
11	CSR_MIE_MEIE	r/w	Machine external interrupt enable
7	CSR_MIE_MTIE	r/w	Machine timer interrupt enable (from MTIME)
3	CSR_MIE_MSIE	r/w	Machine software interrupt enable

> ℹ The *fast interrupt request signals* (FIRQ) are not used by the CPU itself. These IRQ lines are drives by processor modules. See section 3.3. Processor Interrupts for more information.

Machine Trap-Handler Base Address   `mtvec`

```
CSR address:   0x305
Reset value:   UNDEFINED
```

The `mtvec` CSR is compatible to the RISC-V specifications. It stores the base address for **ALL** machine traps. Thus, it defines the main entry point for exception/interrupt handling regardless of the actual trap source. The lowest two bits of this register are always zero and cannot be modified (= fixed address mode).

Bit#	R/W	Function
31:2	r/w	4-byte aligned base address of trap base handler
1:0	r/-	Always zero

Machine Counter Enable   `mcounteren`

```
CSR address:   0x306
Reset value:   UNDEFINED
```

The `mcounteren` CSR is compatible to the RISC-V specifications. The bits of this CSR define which counter/timer CSR can be accessed (read) from code running in a less-privileged environment. For example, if user-level code tries to read from a counter/timer CSR without having access, the illegal instruction exception is raised. The following table shows all implemented bits (all remaining bits are always zero and are read-only). If user mode in not implemented (`CPU_EXTENSION_RISCV_U` = `false`) all bits of the `mcounteren` CSR are tied to zero.

Bit#	Name [C]	R/W	Function
2	CSR_MCOUNTEREN_IR	r/w	User-level code is allowed to read `cycle[h]` CSRs when set
1	CSR_MCOUNTEREN_TM	r/w	User-level code is allowed to read `time[h]` CSRs when set
0	CSR_MCOUNTEREN_CY	r/w	User-level code is allowed to read `instret[h]` CSRs when set

Machine Status Register – High Word   `mstatush`

```
CSR address:   0x310
Reset value:   0x00000020
```

The `mstatus` CSR is compatible to the RISC-V specifications. It provides additional CPU status information. The following bits are implemented (all remaining bits are always zero and are read-only):

Bit#	Name [C]	R/W	Function
5	CSR_MSTATUSH_MBE	r/-	Machine-mode byte-order (Endianness) for load/Store operations, always set indicating BIG-endian byte-order

2.6.3. Machine Trap Handling

Scratch Register for Machine Trap Handlers   `mscratch`

```
CSR address:  0x340
Reset value:  UNDEFINED
```

The `mscratch` CSR is compatible to the RISC-V specifications. It is a general purpose scratch register that can be used by the exception/interrupt handler. The content pf this register after reset is undefined.

Machine Exception Program Counter   `mepc`

```
CSR address:  0x341
Reset value:  UNDEFINED
```

The `mepc` CSR is compatible to the RISC-V specifications. For exceptions (like an illegal instruction) this register provides the address of the exception-causing instruction. For Interrupt (like a machine timer interrupt) this register provides the address of the next not-yet-executed instruction.

Machine Trap Cause   `mcause`

```
CSR address:  0x342
Reset value:  UNDEFINED
```

The `mcause` CSR is compatible to the RISC-V specifications. It shows the cause of the current exception / interrupt (see chapter 2.8. Traps, Exceptions and Interrupts). The following bits are implemented:

Bit#	R/W	Function
31	r/w	1: Indicates an interrupt; 0: Indicates an exception
30:5	r/-	Always zero
4:0	r/w	Exception ID code

Machine Bad Address or Instruction   `mtval`

```
CSR address:   0x343
Reset value:   UNDEFINED
```

The `mtval` CSR is compatible to the RISC-V specifications. When a trap is triggered, the CSR shows either the faulting address (for misaligned/faulting load/stores/fetch) or the faulting instruction itself (for illegal instructions). For interrupts the CSR is set to zero. The `mtval` CSR content provides the following information after entering a trap:

Trap cause	mcause CSR	mtval CSR content
Misaligned instruction fetch address Instruction fetch access fault	0x00000000 0x00000001	Address of faulting instruction fetch
Breakpoint	0x00000003	Program counter (= address) of faulting instruction itself
Misaligned load address Load access fault Misaligned store address Store access fault	0x00000006 0x00000005 0x00000006 0x00000007	Access address of faulting load/store operation
Illegal instruction	0x00000002	Instruction word of faulting instruction
Anything else (including interrupts)	...	0x00000000 (always zero)

Machine Interrupt Pending mip

```
CSR address:   0x344
Reset value:   0x00000000
```

The `mip` CSR is compatible to the RISC-V specifications and provides custom extensions. The following bits are implemented (all remaining bits are always zero and are read-only):

Bit#	Name [C]	Note	R/W	Function
31	CSR_MIP_FIRQ15P	*custom*	r/w	Fast interrupt channel 15 pending
30	CSR_MIP_FIRQ14P	*custom*	r/w	Fast interrupt channel 14 pending
29	CSR_MIP_FIRQ13P	*custom*	r/w	Fast interrupt channel 13 pending
28	CSR_MIP_FIRQ12P	*custom*	r/w	Fast interrupt channel 12 pending
27	CSR_MIP_FIRQ11P	*custom*	r/w	Fast interrupt channel 11 pending
26	CSR_MIP_FIRQ10P	*custom*	r/w	Fast interrupt channel 10 pending
25	CSR_MIP_FIRQ9P	*custom*	r/w	Fast interrupt channel 9 pending
24	CSR_MIP_FIRQ8P	*custom*	r/w	Fast interrupt channel 8 pending
23	CSR_MIP_FIRQ7P	*custom*	r/w	Fast interrupt channel 7 pending
22	CSR_MIP_FIRQ6P	*custom*	r/w	Fast interrupt channel 6 pending
21	CSR_MIP_FIRQ5P	*custom*	r/w	Fast interrupt channel 5 pending
20	CSR_MIP_FIRQ4P	*custom*	r/w	Fast interrupt channel 4 pending
19	CSR_MIP_FIRQ3P	*custom*	r/w	Fast interrupt channel 3 pending
18	CSR_MIP_FIRQ2P	*custom*	r/w	Fast interrupt channel 2 pending
17	CSR_MIP_FIRQ1P	*custom*	r/w	Fast interrupt channel 1 pending
16	CSR_MIP_FIRQ0P	*custom*	r/w	Fast interrupt channel 0 pending
11	CSR_MIP_MEIP	RISC-V	r/w	Machine external interrupt pending
7	CSR_MIP_MTIP	RISC-V	r/w	Machine timer interrupt pending
3	CSR_MIP_MSIP	RISC-V	r/w	Machine software interrupt pending

A pending interrupt can be cleared by setting the according bit to zero.

> ℹ The *fast interrupt request signals* (FIRQ) are not used by the CPU itself. These IRQ lines are drives by processor modules. See section 3.3. Processor Interrupts for more information.

2.6.4. Machine Physical Memory Protection

The available physical memory protection logic is configured via the `PMP_NUM_REGIONS` and `PMP_MIN_GRANULARITY` top entity generics. `PMP_NUM_REGIONS` defines the number of implemented protection regions and thus, the availability of the according `pmpcfg*` and `pmpaddr*` CSRs.

> ℹ️ If trying to access an PMP-related CSR beyond `PMP_NUM_REGIONS` **no illegal instruction exception** is triggered and the according CSRs are read-only and always return zero.

> ⚠️ The RISC-V-compatible NEORV32 physical memory protection only implements the `NAPOT` (naturally aligned power-of-two region) mode with a minimal region granularity of 8 bytes.

Physical Memory Protection Configuration Register(s)   `pmpcfg0 – pmpcfg15`

```
CSR address:   0x3a0 – 0x3af
Reset value:   0x00000000
```

The `pmpcfg*` CSRs are compatible to the RISC-V specifications. They are used to configure the protected regions, where each `pmpcfg*` CSR provides configuration bits for four regions. The following bits (for the first PMP configuration entry) are implemented (all remaining bits are always zero and are read-only):

Bit#	RISC-V Name	R/W	Function
7	L	r/w	Lock bit, can be set – but not be cleared again (only via CPU reset)
6:5	-	r/-	Reserved, always read as zero
4:3	A	r/w	Mode configuration; only **OFF** ("00") and **NAPOT** ("11") are supported
2	X	r/w	Execute permission
1	W	r/w	Write permission
0	R	r/w	Read permission

Physical Memory Protection Base Address Register(s)   `pmpaddr0 – pmpaddr63`

```
CSR address:   0x3b0 – 0x3ef
Reset value:   UNDEFINED
```

The `pmpaddr*` CSRs are compatible to the RISC-V specifications. They are used to configure the base address and the region size.

> ℹ️ When configuring PMP make sure to set `pmpaddr*` before activating the according region via `pmpcfg*`; when changing the PMP configuration, deactivate the according region via `pmpcfg*` before modifying `pmpaddr*`.

2.6.5. [Machine] Counters and Timers

Cycle Counter – Low Word   `cycle`

```
CSR address:   0xc00
Reset value:   UNDEFINED
```

The `cycle` CSR is compatible to the RISC-V specifications. It shows the lower 32-bit of the 64-bit cycle counter. The `cycle` CSR is read-only and is a shadowed copy of the `mcycle` CSR.

Machine Cycle Counter – Low Word   `mcycle`

```
CSR address:   0xb00
Reset value:   UNDEFINED
```

The `mcycle` CSR is compatible to the RISC-V specifications. It shows the lower 32-bit of the 64-bit cycle counter. The `mcycle` CSR can also be written and is copied to the `cycle` CSR.

System Time – Low Word   `time`

```
CSR address:   0xc01
Reset value:   UNDEFINED
```

The `time` CSR is compatible to the RISC-V specifications. It shows the lower 32-bit of the 64-bit system time. The system time is generated by the MTIME system timer unit via the CPU `time_i` signal. The `time` CSR is read-only. Change the system time via the MTIME unit.

If the processor-internal machine timer (MTIME) is not implemented (via `IO_MTIME_EN` = `false`), the processor's `mtime_i` top entity signal is accessible via the `time[h]` CSRs.

Instructions-Retired Counter – Low Word   `instret`

```
CSR address:   0xc02
Reset value:   UNDEFINED
```

The `instret` CSR is compatible to the RISC-V specifications. It shows the lower 32-bit of the 64-bit retired instructions counter.. The `instret` CSR is read-only and is a shadowed copy of the `minstret` CSR.

Machine Instructions-Retired Counter – Low Word   `minstret`

```
CSR address:   0xb01
Reset value:   UNDEFINED
```

The `minstret` CSR is compatible to the RISC-V specifications. It shows the lower 32-bit of the 64-bit retired instructions counter. The `minstret` CSR can also be written and is copied to the `instret` CSR.

Cycle Counter – High Word   `cycleh`

```
CSR address:   0xc80
Reset value:   UNDEFINED
```

The `cycleh` CSR is compatible to the RISC-V specifications. It shows the upper 32-bit of the 64-bit cycle counter. The `cycleh` CSR is read-only and is a shadowed copy of the `mcycleh` CSR.

Machine Cycle Counter – High Word   `mcycleh`

```
CSR address:   0xb80
Reset value:   UNDEFINED
```

The `mcycleh` CSR is compatible to the RISC-V specifications. It shows the upper 32-bit of the 64-bit cycle counter. The `mcycleh` CSR can also be written and is copied to the `cycleh` CSR.

System Time – High Word   `timeh`

```
CSR address:   0xc81
Reset value:   UNDEFINED
```

The `timeh` CSR is compatible to the RISC-V specifications. It shows the upper 32-bit of the 64-bit system time. The system time is generated by the MTIME system timer unit via the CPU `time_i` signal. The `timeh` CSR is read-only. Change the system time via the MTIME unit.

If the processor-internal machine timer (MTIME) is not implemented (via `IO_MTIME_EN` = `false`), the processor's `mtime_i` top entity signal is accessible via the `time[h]` CSRs.

Instructions-Retired Counter – High Word   `instreth`

```
CSR address:  0xc82
Reset value:  UNDEFINED
```

The `instreth` CSR is compatible to the RISC-V specifications. It shows the upper 32-bit of the 64-bit retired instructions counter. The `instreth` CSR is read-only and is a shadowed copy of the `minstreth` CSR.

Machine Instructions-Retired Counter – High Word   `minstreth`

```
CSR address:  0xb82
Reset value:  UNDEFINED
```

The `minstreth` CSR is compatible to the RISC-V specifications. It shows the upper 32-bit of the 64-bit retired instructions counter. The `minstreth` CSR can also be written and is copied to the `instreth` CSR.

2.6.6. Hardware Performance Monitors (HPM)

The available hardware performance logic is configured via the `HPM_NUM_CNTS` top entity generic. `HPM_NUM_CNTS` defines the number of implemented performance monitors and thus, the availability of the according `[m]hpmcounter*[h]` and `mhpmevent*` CSRs.

> ℹ️ If trying to access an HPM-related CSR beyond `HPM_NUM_CNTS` **no illegal instruction exception** is triggered and the according CSRs are read-only and always return zero.

Machine Performance-Monitoring Event Selector(s)	mhpmevent3 – mhpmevent31

```
CSR address:  0x323 – 0x33f
Reset value:  UNDEFINED
```

The `mhpmevent*` CSRs are compatible to the RISC-V specifications. The configuration of these CSR define the events that cause the according `[m]hpmcounter*[h]` counters to increment. All available events are listed in the table below. If more than one event is selected, the according counter will increment if **any** 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 event is observed. If the CPU is in sleep mode, no HPM counter will increment at all.

The available hardware performance logic is configured via the `HPM_NUM_CNTS` top entity generic. `HPM_NUM_CNTS` defines the number of implemented performance monitors and thus, the availability of the according `[m]hpmcounter*[h]` and `mhpmevent*` CSRs.

Bit#	Name [C]	R/W	Event that triggers a counter increment
0	HPMCNT_EVENT_CY	r/w	Active clock cycle
1	-	r/-	*Not implemented, always read as zero*
2	HPMCNT_EVENT_IR	r/w	Retired Instruction
3	HPMCNT_EVENT_CIR	r/w	Retired compressed instruction
4	HPMCNT_EVENT_WAIT_IF	r/w	Instruction fetch memory wait cycle (if more than 1 cycle memory latency)
5	HPMCNT_EVENT_WAIT_II	r/w	Instruction issue pipeline wait cycle (if more than 1 cycle latency), caused by pipelines flushes (like taken branches)
6	HPMCNT_EVENT_WAIT_MC	r/w	Multi-cycle ALU operation wait cycle
7	HPMCNT_EVENT_LOAD	r/w	Load operation
8	HPMCNT_EVENT_STORE	r/w	Store operation
9	HPMCNT_EVENT_WAIT_LS	r/w	Load/store memory wait cycle (if more than 1 cycle memory latency)
10	HPMCNT_EVENT_JUMP	r/w	Unconditional jump
11	HPMCNT_EVENT_BRANCH	r/w	Conditional branch (taken or not taken)
12	HPMCNT_EVENT_TBRANCH	r/w	Taken conditional branch
13	HPMCNT_EVENT_TRAP	r/w	Entered trap
14	HPMCNT_EVENT_ILLEGAL	r/w	Illegal instruction exception

> ℹ️ The total LSB-aligned HPM counter size (low word CSR + high word CSR) is defined via the `HPM_CNT_WIDTH` generic (1..64-bit). If `HPM_CNT_WIDTH` is less than 64, all unused MSB-aligned bits are hardwired to zero.

Performance-Monitoring Counter(s) – Low Word   `hpmcounter3 – hpcmcounter31`

```
CSR address:   0xc03 – 0xc1f
Reset value:   UNDEFINED
```

The `hpmcounter*` CSRs are compatible to the RISC-V specifications. These CSRs provide the lower 32-bit of arbitrary event counters (64-bit). These CSRs are read-only and provide a showed copy of the according `mhpmcounter*` CSRs.

Machine Performance-Monitoring Counter(s) – Low Word   `mhpmcounter3 – mhpcmcounter31`

```
CSR address:   0xb03 – 0xb1f
Reset value:   UNDEFINED
```

The `mhpmcounter*` CSRs are compatible to the RISC-V specifications. These CSRs provide the lower 32-bit of arbitrary event counters (64-bit). The `mhpmcounter*` CSRs can also be written and are copied to the `hpmcounter*` CSRs. The event(s) that trigger an increment of theses counters are selected via the according `mhpmevent*` CSRs.

Performance-Monitoring Counter(s) – High Word   `hpmcounter3h – hpcmcounter31h`

```
CSR address:   0xc83 – 0xc9f
Reset value:   UNDEFINED
```

The `hpmcounter*h` CSRs are compatible to the RISC-V specifications. These CSRs provide the upper 32-bit of arbitrary event counters (64-bit). These CSRs are read-only and provide a showed copy of the according `mhpmcounter*h` CSRs.

Machine Performance-Monitoring Counter(s) – Hi. Word   `mhpmcounter3h – mhpcmcounter31h`

```
CSR address:   0xb83 – 0xb9f
Reset value:   UNDEFINED
```

The `mhpmcounter*h` CSRs are compatible to the RISC-V specifications. These CSRs provide the upper 32-bit of arbitrary event counters (64-bit). The `mhpmcounter*h` CSRs can also be written and are copied to the `hpmcounter*h` CSRs. The event(s) that trigger an increment of theses counters are selected via the according `mhpmevent*` CSRs.

2.6.7. Machine Counters Setup

Machine Counter-Inhibit Register   `mcountinhibit`

```
CSR address:   0x320
Reset value:   0xFFFFFFFF
```

The `mcountinhibit` CSR is compatible to the RISC-V specifications. The bits in this register define which counter/timer CSR are allowed to perform an automatic increment. Automatic update is enabled if the according bit in `mcountinhibit` is **cleared**. The following bits are implemented (all remaining bits are always zero and are read-only):

Bit#	Name [C]	R/W	Function
0	CSR_MCOUNTINHIBIT_IR	r/w	The `[m]instret[h]` CSRs will auto-increment with each committed instruction when set
2	CSR_MCOUNTINHIBIT_CY	r/w	The `[m]cycle[h]` CSRs will auto-increment with each clock cycle (if CPU is not in sleep state) when set
3..31	-	r/w	The `[m]hpmcount*[h]` CSRs will auto-increment according to the configured `mhpmevent*` selector

2.6.8. Machine Information Registers

Machine Vendor ID   `mvendorid`

```
CSR address:   0xf11
Reset value:   0x00000000
```

The `mvendorid` CSR is compatible to the RISC-V specifications. It is read-only and always reads zero.

Machine Architecture ID   `marchid`

```
CSR address:   0xf12
Reset value:   0x00000013
```

The `marchid` CSR is compatible to the RISC-V specifications. It is read-only and shows the NEORV32 official RISC-V open-source architecture ID (decimal: `19`, 32-bit hexadecimal: `0x00000013`).

Machine Implementation ID   `mimpid`

```
CSR address:   0xf13
Reset value:   HW version number
```

The `mimpid` CSR is compatible to the RISC-V specifications. It is read-only and shows the version of the NEORV32 as BCD-coded number (example: `mimpid` = 0x01020312 → 01.02.03.12 → version 1.2.3.12).

Machine Hardware Thread ID   `mhartid`

```
CSR address:   0xf14
Reset value:   HW_THREAD_ID generic
```

The `mhartid` CSR is compatible to the RISC-V specifications. It is read-only and shows the core's hart ID, which is assigned via the CPU's `HW_THREAD_ID` generic.

2.6.9. NEORV32-Specific Custom CSRs

Z* CPU Extensions Indicator Register   `mzext`

```
CSR address:  0xfc0
Reset value:  -
```

The `mzext` CSR is a custom read-only CSR that shows the implemented Z* extensions. The following bits are implemented (all remaining bits are always zero).

Bit#	Name [C]	R/W	Function
0	CPU_MZEXT_ZICSR	r/-	`Zicsr` extensions available (enabled via `CPU_EXTENSION_RISCV_Zicsr` generic)
1	CPU_MZEXT_ZIFENCEI	r/-	`Zifencei` extensions available (enabled via `CPU_EXTENSION_RISCV_Zifencei` generic)
2	CPU_MZEXT_ZBB	r/-	`Zbb` extensions available; bit manipulation base subset (enabled via `CPU_EXTENSION_RISCV_B` generic)
3	CPU_MZEXT_ZBS	r/-	`Zbs` extensions available; bit manipulation base subset (enabled via `CPU_EXTENSION_RISCV_B` generic)
4	CPU_MZEXT_ZBA	r/-	`Zba` extensions available; bit manipulation base subset (enabled via `CPU_EXTENSION_RISCV_B` generic)
5	CPU_MZEXT_ZFINX	r/-	`Zfinx` floating-point extensions available (enabled via `CPU_EXTENSION_RISCV_Zfinx` generic)

2.7. Execution Safety

The hardware of the NEORV32 CPU was designed for a maximum of *execution safety*. If the `Zicsr` CPU extension is enabled, the core supports **all** traps specified by the official RISC-V specifications (obviously, not the ones that are related to yet unimplemented extensions/features). Thus, the CPU provides well-defined hardware fall-backs for (nearly) everything that can go wrong. Even if any kind of trap is triggered, the core is always in a precise and fully synchronized state throughout the whole architecture (i.e. no need to make out-of-order operations undone) that allows predictable execution behavior at any time.

Additional and highlighted safety features:

- ✗   The CPU supports ***all bus exceptions*** including <u>bus access exceptions</u> that are triggered if an accessed address does not respond or encounters an internal error during access (which is a rare feature in many open-source RISC-V cores).

- ✗   The CPU raises an illegal instruction trap for **all** unimplemented/malformed/illegal instruction words (which is a rare feature in many open-source RISC-V cores, too).

- ✗   If user-level code tries to read from machine-level-only CSR (like `mstatus`) an illegal instruction exception is raised ($\rightarrow$ illegal access). The results of this operations is always zero (though, machine-level code handling this exception can modify the target register of the illegal access-causing instruction to allow full virtualization). Illegal write accesses to machine CSRs will not be conducted at all and will only result in raising an illegal instruction exception.

- ✗   Illegal user-level memory accesses to protected addresses or address regions (via physical memory protection) will not be conducted at all (no actual write and no actual read; prevents triggering of memory-mapped devices). Illegal load operations will not result any data (the instruction's destination register will not be written at all).

2.8. Traps, Exceptions and Interrupts

In this document a (maybe) special nomenclature regarding traps is used:

- Interrupts = Asynchronous exceptions
- Exceptions = Synchronous exceptions
- Traps = Exceptions + Interrupts (synchronous **or** asynchronous exceptions)

Whenever an exception or interrupt is triggered, the CPU transfers control to the address stored in the `mtvec` CSR. The cause of the according interrupt or exception can be determined via the content of the `mcause` CSR The address that reflected the current program counter when a trap was taken is stored to `mepc`. Additional information regarding the cause of the trap can be retrieved from `mtval`.

The traps are prioritized. If several exceptions occur at once only the one with highest priority is triggered. If several interrupts trigger at once, the one with highest priority is triggered while the remaining ones are queued. After completing the interrupt handler the interrupt with the second highest priority will issues and so on.

Memory Access Exceptions

If a load operation causes any exception, the destination register is not written at all. Exceptions caused by a misalignment or a physical memory protection fault do not trigger a bus read-operation at all.
Exceptions caused by a store address misalignment or a store physical memory protection fault do not trigger a bus write-operation at all.

Instruction Atomicity

All instructions execute as atomic operations – interrupts can only trigger between two consecutive instructions.

Custom Fast Interrupt Request Lines

As a custom extension, the NEORV32 CPU features 16 *fast interrupt request* 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 (see table below).

> *i* The *fast interrupt request signals* (FIRQ) are not used by the CPU itself. These IRQ lines are drives by processor modules. See section 3.3. Processor Interrupts for more information.

Prio.	mcause	[RISC-V]	ID [C]	Cause	mepc	mtval	
1	0x8000000B	1.11	TRAP_CODE_MEI	Machine external interrupt	I-PC	0	
2	0x80000003	1.3	TRAP_CODE_MSI	Machine software interrupt	I-PC	0	
3	0x80000007	1.7	TRAP_CODE_MTI	Machine timer interrupt (from MTIME)	I-PC	0	
4	0x80000010	1.16	TRAP_CODE_FIRQ_0	Fast interrupt request channel 0	I-PC	0	*C*
5	0x80000011	1.17	TRAP_CODE_FIRQ_1	Fast interrupt request channel 1	I-PC	0	*C*
6	0x80000012	1.18	TRAP_CODE_FIRQ_2	Fast interrupt request channel 2	I-PC	0	*C*
7	0x80000013	1.19	TRAP_CODE_FIRQ_3	Fast interrupt request channel 3	I-PC	0	*C*
8	0x80000014	1.20	TRAP_CODE_FIRQ_4	Fast interrupt request channel 4	I-PC	0	*C*
9	0x80000015	1.21	TRAP_CODE_FIRQ_5	Fast interrupt request channel 5	I-PC	0	*C*
10	0x80000016	1.22	TRAP_CODE_FIRQ_6	Fast interrupt request channel 6	I-PC	0	*C*
11	0x80000017	1.23	TRAP_CODE_FIRQ_7	Fast interrupt request channel 7	I-PC	0	*C*
12	0x80000018	1.24	TRAP_CODE_FIRQ_8	Fast interrupt request channel 8	I-PC	0	*C*
13	0x80000019	1.25	TRAP_CODE_FIRQ_9	Fast interrupt request channel 9	I-PC	0	*C*
14	0x8000001a	1.26	TRAP_CODE_FIRQ_10	Fast interrupt request channel 10	I-PC	0	*C*
15	0x8000001b	1.27	TRAP_CODE_FIRQ_11	Fast interrupt request channel 11	I-PC	0	*C*
16	0x8000001c	1.28	TRAP_CODE_FIRQ_12	Fast interrupt request channel 12	I-PC	0	*C*
17	0x8000001d	1.29	TRAP_CODE_FIRQ_13	Fast interrupt request channel 13	I-PC	0	*C*
18	0x8000001e	1.30	TRAP_CODE_FIRQ_14	Fast interrupt request channel 14	I-PC	0	*C*
19	0x8000001f	1.31	TRAP_CODE_FIRQ_15	Fast interrupt request channel 15	I-PC	0	*C*
20	0x00000001	0.1	TRAP_CODE_I_ACCESS	Instruction access fault	B-ADR	PC	
21	0x00000002	0.2	TRAP_CODE_I_ILLEGAL	Illegal instruction	PC	Inst	
22	0x00000000	0.0	TRAP_CODE_I_MISALIGNED	Instruction address misaligned	B-ADR	PC	
23	0x0000000B	0.11	TRAP_CODE_MENV_CALL	Environment call from M-mode (ECALL in machine-mode)	PC	PC	
24	0x00000008	0.8	TRAP_CODE_UENV_CALL	Environment call from U-mode (ECALL in user-mode)	PC	PC	
25	0x00000003	0.3	TRAP_CODE_BREAKPOINT	Breakpoint (EBREAK)	PC	PC	
26	0x00000006	0.6	TRAP_CODE_S_MISALIGNED	Store address misaligned	B-ADR	B-ADR	
27	0x00000004	0.4	TRAP_CODE_L_MISALIGNED	Load address misaligned	B-ADR	B-ADR	
28	0x00000007	0.7	TRAP_CODE_S_ACCESS	Store access fault	B-ADR	B-ADR	
29	0x00000005	0.5	TRAP_CODE_L_ACCESS	Load access fault	B-ADR	B-ADR	

> *i* The [C] names are defined by the NEORV32 core library (`sw/lib/include/neorv32.h`) and can be used in plain C code.

Notes

The priority column shows the priority of each trap. The highest priority is 1. The `mcause` column shows the cause ID of the according trap that is written to `mcause` CSR. Orange values indicate an interrupt (=asynchronous exception), black number indicate an exception (=synchronous exception). Lines marked with a "*C*" are custom extensions. The `RISC-V` columns show the *interrupt/exception code value* from the official *RISC-V privileged architecture* manual. The **mepc** and **mtval** columns show the value written to `mepc` and `mtval` CSRs when a trap is triggered:

- `I-PC`   Address of *interrupted* instruction (instruction has not been execute/completed yet)
- `B-ADR` Bad memory access address that cause the trap
- `PC`   Address of instruction that caused the trap
- `0`   Zero
- `Inst`   The faulting instruction itself
- `BOOT`   CPU boot address

2.9. Address Space

The CPU is a 32-bit architecture with separated instruction and data interfaces making it a *Harvard Architecture*. Each of this interfaces can access an address space of up to 2^{32} bytes (4GB). The memory system is based on 32-bit words with a minimal granularity of 1 byte. Please note, that the NEORV32 CPU does not support unaligned memory accesses <u>in hardware</u> – however, a software-based handling can be implemented as any unaligned memory access will trigger an according exception.

2.10. Bus Interface

The CPU provides two independent bus interfaces: One for fetching instructions (`i_bus_*`) and one for accessing data (`d_bus_*`) via load and store operations. Both interfaces use the same interface protocol.

2.10.1. Interface Signals

The following table shows the signals of the data ("D") and instruction ("I") interfaces seen from the CPU (`*_o` signals are driven by the CPU, `*_i` signals are read by the CPU).

Signal	Interface	Size	Function
bus_addr_o	D/I	32	The access address
bus_rdata_i	D/I	32	Data input for read operations
bus_wdata_o	D/I	32	Data output for write operations
bus_ben_o	D/I	4	Byte enable signal for write operations
bus_we_o	D/I	1	Bus write access
bus_re_o	D/I	1	Bus read access
bus_cancel_o	D/I	1	Indicates that the current bus access is terminated by the controller (the CPU)
bus_ack_i	D/I	1	Accessed peripheral indicates a successful completion of the bus transaction
bus_err_i	D/I	1	Accessed peripheral indicates an error during the bus transaction
bus_fence_o	D/I	1	This signal is set for one cycle when the CPU executes a data/instruction fence operation
bus_excl_o	D	1	Signal is set for one cycle when the current data access requests an exclusive access
bus_excl_i	D	1	Indicates if an exclusive access was successful. This signal is evaluated when a transfer is terminated (via `d_bus_ack_i` or `d_bus_err_i`).

> *i* Currently, there a no pipelined or overlapping operations implemented within the same bus interface. So only a single transfer request can be "on the fly". This also means that there can only be an exclusive active read transaction or an active write transaction – read and write transactions in parallel are not yet implemented.

> *i* If there is no active transfer in progress (data or instructions) the state of the `bus_cancel_o` signal is irrelevant.

2.10.2. Protocol

A bus request is triggered either by the `bus_re_o` signal (for reading data) or by the `bus_we_o` signal (for writing data). These signals are active for one cycle and initiate a new bus transaction. The transaction is completed when the accessed peripheral either sets the `bus_ack_i` signal (→ successful completion) or the `bus_err_i` signal is set (→ failed completion). All these control signals are only active (= high) for one single cycle.

An error indicated via the `bus_err_i` signal during a transfer will trigger the according *instruction bus access fault* or *load/store bus access fault* exception. The CPU will terminate a transfer (when an error during transfer is encountered) via the `bus_cancel_o` signal.

The transfer can be completed directly in the same cycle as it was initiated (via the `bus_re_o` or `bus_we_o` signal) if the peripheral sets `bus_ack_i` or `bus_err_i` high for one cycle.

⚠️ **Memories / memory-mapped devices with variable latency**

All bus transactions require a minimal latency of 1 cycle. In this case the bus transactions takes a total of 2 cycles (setting all signals in the first cycle starting a new request; results are provided in the second cycle.

There is no problem if the accessed peripheral takes more than 1 cycle to process the request (= latency > 1 cycle). However, the bus transaction **has to be completed (= acknowledged)** within the number of cycles specified via the global `bus_timeout_c` constant (default: 127 cycles) from the VHDL package file (`rtl/neorv32_package.vhd`). If not, the according *instruction bus access fault* or *load/store bus access fault* exception is triggered and the CPU cancels the transaction via the `bus_cancel_o` signal.
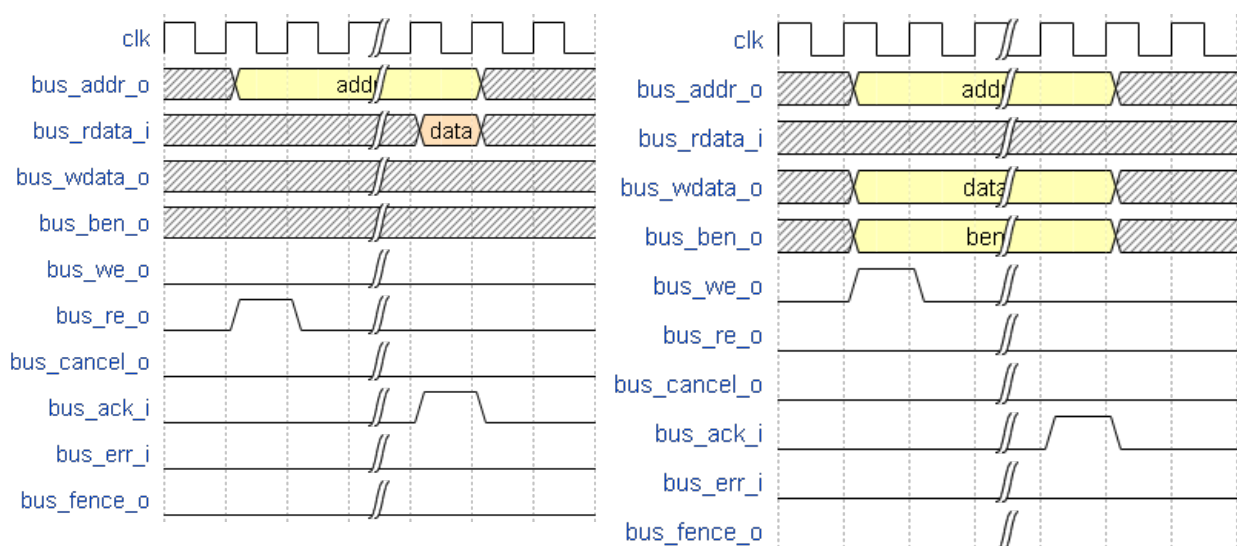
Bus Accesses

Figure 2: CPU interface read access (left) and write access (right)

Write Access

For a write access, the accessed address (`bus_addr_o`), the data to be written (`bus_wdata_o`) and the byte enable signals (`bus_ben_o`) are set when `bus_we_o` goes high. These three signals are kept stable until the transaction is completed. In the example below the accessed peripheral cannot answer directly in the next cycle after issuing. Here, the transaction is successful and the peripheral sets the `bus_ack_i` signal several cycles after issuing.

Read Access

For a read access, the accessed address (`bus_addr_o`) is set when `bus_re_o` goes high. The address is kept stable until the transaction is completed. In the example below the accessed peripheral cannot answer directly in the next cycle after issuing. The peripheral hast to apply the read data right in the same cycle as the bus transaction is completed (here, the transaction is successful and the peripheral sets the `bus_ack_i` signal).

Exclusive (Atomic) Access

The CPU can access memory in an exclusive manner by generating load-reservate and store-conditional combination. Normally, these combinations should target the same memory address.

The CPU starts an exclusive access to a specific address via the *load-reservate instruction* (`LR.W`). This instruction will set the `d_bus_excl_o` signal high for one cycle when requesting a new read-transfer (indicated by `d_bus_re_o` = 1). It is the task of the memory system to manage this exclusive access reservation by storing the according access address and the source of the access itself (for example via the CPU ID in a multi-core system).

When the CPU executed a *store-conditional instruction* (`SC.W`) the `d_bus_excl_i` signal is evaluated to check if the exclusive access was successful. The memory system has to set this signal high when the transfer is acknowledged to indicate a successful exclusive access.

The reservation established via the according load-reservate instruction has to be removed if there is an access to the according address where the `d_bus_excl_i` signal is NOT asserted. This is the case for any non-atomic load/store instructions (including instruction fetches) and also for the store-conditional instruction.

> ℹ️ For more information regarding the SoC-lvel behavior and requirement of atomic operations see chapter 3.5.5. Processor-External Memory Interface (WISHBONE) (AXI4-Lite).

Memory Barriers

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

Access Boundaries

The instruction interface will always access memory on word (= 32-bit) boundaries even if fetching compressed (16-bit) instructions. The data interface can access memory on byte (= 8-bit), half-word (= 16-bit) and word (= 32-bit) boundaries.

3. NEORV32 Processor (SoC)

The NEORV32 Processor is based in the *NEORV32 CPU* (→ pp.17). Together with common peripheral interfaces and embedded memories it provide a RISC-V-based full-scale microcontroller-like SoC platform.
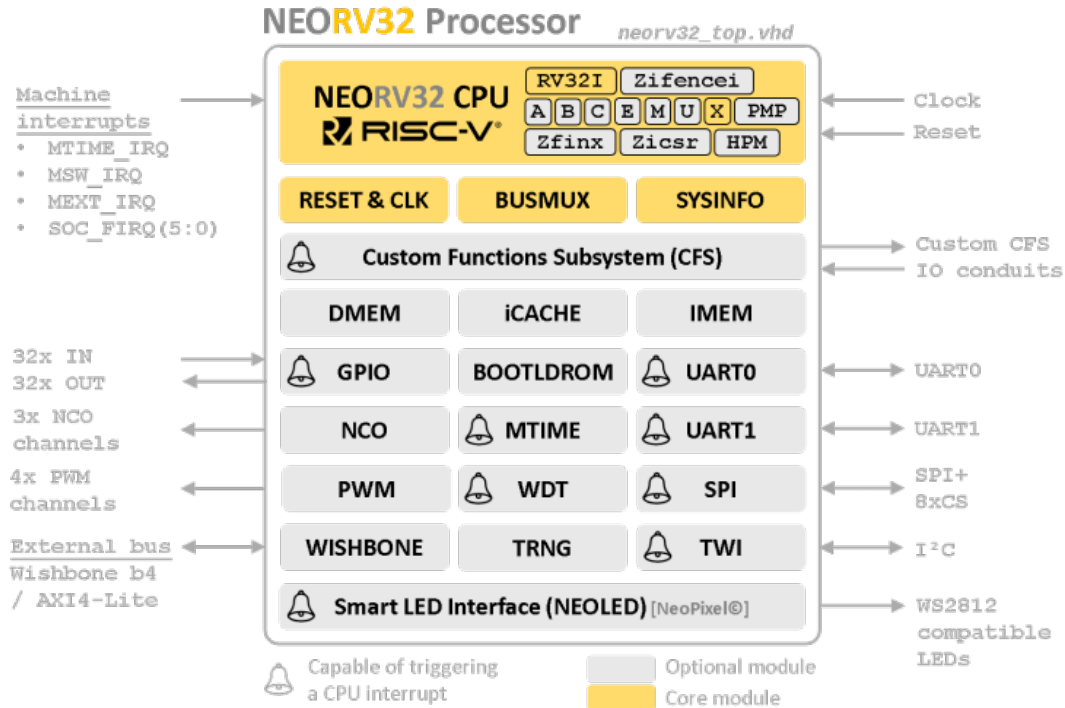

Figure 3: NEORV32 processor block diagram

Key Features

✔ Optional processor-internal data and instruction memories (**DMEM/IMEM** → p.77) + cache (**iCACHE** → p.78)

✔ Optional internal bootloader (**BOOTLDROM**) with UART console & SPI flash boot option (→ p.123)

✔ Optional machine system timer (**MTIME** → p.88), RISC-V-copmatible

✔ Optional two independent universal asynchronous receivers and transmitters (**UART0** → p.89, **UART1** → p.93) with optional hardware flow control (RTS/CTS)

✔ Optional 8/16/24/32-bit serial peripheral interface controller (**SPI** → p.95) with 8 dedicated CS lines

✔ Optional two wire serial interface controller (**TWI** → p.97), compatible to the I²C standard

✔ Optional general purpose parallel IO port (**GPIO** → p.84), 32xOut, 32xIn

✔ Optional 32-bit external bus interface, Wishbone b4 / **AXI4-Lite** compatible (**WISHBONE** → p.80)

✔ Optional watchdog timer (**WDT** → p.85)

✔ Optional PWM controller with 4 channels and 8-bit duty cycle resolution (**PWM** → p.99)

✔ Optional ring-oscillator-based true random number generator (**TRNG** → p.101)

✔ Optional custom functions subsystem for custom co-processor extensions (**CFS** → p.103)

✔ Optional numerically-controlled oscillator (**NCO** → p.106) with 3 independent channels

✔ Optional NeoPixel©/WS2812-compatible smart LED interface (**NEOLED** → p.110)

✔ System configuration information memory to check HW config. via software (**SYSINFO** → p.114)

3.1. Processor Top Entity – Signals

The following table shows all interface ports of the processor top entity (**rtl/core/neorv32_top.vhd**). The type of all signals is **std_ulogic** or **std_ulogic_vector**, respectively.

> ℹ️ A wrapper for the NEORV32 Processor setup providing resolved port signals can be found in rtl/top_templates/neorv32_top_stdlogic.vhd.

Signal Name	Width	Direction	Function	HW Module / Chapter
Global Control				
clk_i	1	Input	Global clock line, all registers triggering on rising edge	global
rstn_i	1	Input	Global reset, **low-active**	
External bus interface (Wishbone-compatible)				
wb_tag_o	4	Output	Tag (access type identifier)	
wb_adr_o	32	Output	Destination address	
wb_dat_i	32	Input	Write data	
wb_dat_o	32	Output	Read data	
wb_we_o	1	Output	Write enable ('0' = read transfer)	
wb_sel_o	4	Output	Byte enable	3.5.5. Processor-External Memory Interface (WISHBONE) (AXI4-Lite)
wb_stb_o	1	Output	Strobe	
wb_cyc_o	1	Output	Valid cycle	
wb_tag_i	1	Input	Tag (response identifier)	
wb_ack_i	1	Input	Transfer acknowledge	
wb_err_i	1	Input	Transfer error	
Advanced memory control signals				
fence_o	1	Output	Indicates an executed fence instruction	2.4. Instruction Sets and CPU Extensions
fencei_o	1	Output	Indicates an executed fencei instruction	
General Purpose Inputs & Outputs (GPIO)				
gpio_o	32	Output	General purpose parallel output	3.5.6. General Purpose Input and Output Port (GPIO)
gpio_i	32	Input	General purpose parallel input	
Primary Universal Asynchronous Receiver/Transmitter (UART0)				
uart0_txd_o	1	Output	UART0 serial transmitter	3.5.9. Primary Universal Asynchronous Receiver and Transmitter (UART0)
uart0_rxd_i	1	Input	UART0 serial receiver	
uart0_rts_o	1	Output	UART0 RX ready to receive new char	
uart0_cts_i	1	Input	UART0 TX allowed to start sending	
Secondary Universal Asynchronous Receiver/Transmitter (UART1)				
uart1_txd_o	1	Output	UART1 serial transmitter	3.5.10. Secondary Universal Asynchronous Receiver and
uart1_rxd_i	1	Input	UART1 serial receiver	

Signal Name	Width	Direction	Function	HW Module / Chapter
uart1_rts_o	1	Output	UART1 RX ready to receive new char	Transmitter (UART1)
uart1_cts_i	1	Input	UART1 TX allowed to start sending	
Serial Peripheral Interface Controller (SPI)				
spi_sck_o	1	Output	SPI controller clock line	3.5.11. Serial Peripheral Interface Controller (SPI)
spi_sdo_o	1	Output	SPI serial data output	
spi_sdi_i	1	Input	SPI serial data input	
spi_csn_o	8	Output	SPI dedicated chip select lines 0..7 (low-active)	
Two-Wire Interface Controller (TWI)				
twi_sda_io	1	InOut	TWI serial data line	3.5.12. Two Wire Serial Interface Controller (TWI)
twi_scl_io	1	InOut	TWI serial clock line	
Custom Functions Subsystem (CFS)				
cfs_in_i	32^{10}	Input	Custom CFS input signal conduit	3.5.15. Custom Functions Subsystem (CFS)
cfs_out_o	32^{11}	Output	Custom CFS output signal conduit	
Pulse-Width Modulation Channels (PWM)				
pwm_o	4	Output	Pulse-width modulated channels	3.5.13. Pulse Width Modulation Controller (PWM)
Numerically-Controller Oscillator (NCO)				
nco_o	3	Output	NCO output channels	3.5.16. Numerically-Controller Oscillator (NCO)
Smart LED Interface (NeoPixel© compatible) (NEOLED)				
neoled_o	1	Output	Async. serial data output	3.5.17. Smart LED Interface (NEOLED) – NeoPixel © Compatible
System time input from external MTIME unit				
mtime_i	32	Input	Machine timer time (to time CSRs) from external MTIME unit if the processor-internal MTIME unit is NOT used	2.6.5. [Machine] Counters and Timers
Interrupts				
soc_firq_i	6	Input	Platform fast interrupt channels (custom)	3.3. Processor Interrupts
mtime_irq_i	1	Input	Machine timer interrupt[12] (RISC-V)	
msw_irq_i	1	Input	Machine software interrupt (RISC-V)	2.8. Traps, Exceptions and Interrupts
mext_irq_i	1	Input	Machine external interrupt (RISC-V)	

Table 7: neorv32_top.vhd – processor's top entity interface ports

---

10 Default value. Can be customized via top's IO_CFS_IN_SIZE configuration generic.
11 Default value. Can be customized via top's IO_CFS_OUT_SIZE configuration generic.
12 Only available if processor-internal machine system timer (MTIME) is disabled (IO_MTIME_EN = false)

3.2. Processor Top Entity – Configuration Generics

This is a list of all configuration generics of the NEORV32 processor top entity `rtl/neorv32_top.vhd`. The generic name is shown in **orange**, followed by the type in printed in **black** and concluded by the default value printed in light gray.

> *i* Privileged software can determine the actual processor/CPU configuration via the `misa` and `mzext` control and status register (→ section 2.6. Control and Status Registers (CSRs)) and via the memory-mapped SYSINFO (→ section 3.5.18. System Configuration Information Memory (SYSINFO)) memory, respectively.

General

CLOCK_FREQUENCY **natural** 0
The clock frequency of the processor's `clk_i` input port in Hertz (Hz).

BOOTLOADER_EN **boolean** true
Implement the boot ROM, pre-initialized with the bootloader image when `true`. This will also change the processor's boot address from the beginning of the instruction memory address space (default = `0x00000000`) to the base address of the boot ROM. See chapter 4.5. Bootloader for more information.

USER_CODE **std_ulogic_vector(31 downto 0)** x"00000000"
Custom user code that can be read by software via the SYSINFO module.

HW_THREAD_ID **natural** 0
The hart ID of the CPU. Can be read via the `mhartid` CSR. Hart IDs must be unique within a system.

RISC-V CPU Extensions
See chapter 2.4. Instruction Sets and CPU Extensions for more information.

CPU_EXTENSION_RISCV_A **boolean** false
Implement atomic memory access operations when `true`.

CPU_EXTENSION_RISCV_B **boolean** false
Implement bit manipulation instructions (NOT ALL subsets are supported yet!) when `true`.

CPU_EXTENSION_RISCV_C **boolean** false
Implement compressed instructions (16-bit) when `true`.

CPU_EXTENSION_RISCV_E **boolean** false
Implement the embedded CPU extension (only implement the first 16 data registers) when `true`.

CPU_EXTENSION_RISCV_M **boolean** false
Implement integer multiplication and division instructions when `true`.

CPU_EXTENSION_RISCV_U **boolean** `false`

Implement user privilege level when `true`.

CPU_EXTENSION_RISCV_Zfinx **boolean** `false`

Implement the 32-bit single-precision **floating-point extension** (using integer registers) when `true`. For more information see section 2.4.8. Single-Precision Floating-Point Extension (Zfinx Extension).

CPU_EXTENSION_RISCV_Zicsr **boolean** `true`

Implement the control and status register (CSR) access instructions when `true`. Note: When this option is disabled, the complete trap system will be excluded from synthesis. Hence, no interrupts, no exceptions and no machine information can be detected.

> ⚠️ The `CPU_EXTENSION_RISCV_Zicsr` should be **always enabled**. Without this extension the CPU does not support any kind of exception/interrupt/trap/machine information system.

CPU_EXTENSION_RISCV_Zifencei **boolean** `false`

Implement the instruction fetch synchronization instruction `fence.i`. For example, this option is required for self-modifying code (and/or for i-cache flushes).

Extension Options

See chapter 2.4. Instruction Sets and CPU Extensions for more information.

FAST_MUL_EN **boolean** `false`

When this generic is enabled, the multiplier of the `M` extension is realized using DSPs blocks instead of an iterative bit-serial approach. This generic is only relevant when the multiplier and divider CPU extension is enabled (`CPU_EXTENSION_RISCV_M` is `true`).

FAST_SHIFT_EN **boolean** `false`

When this generic is enabled the shifter unit of the CPU's ALU is implement as fast barrel shifter (requiring more hardware resources).

Physical Memory Protection (PMP)

See chapter 2.4.11. Physical Memory Protection (PMP Extension) for more information.

PMP_NUM_REGIONS **natural** `0`

Total number of implemented protections regions (0..64). If this generics is zero no physical memory protection logic will be implemented at all.

PMP_MIN_GRANULARITY **natural** `64*1024`

Minimal region granularity in bytes. Has to be a power of two. Has to be at least 8 bytes.

Hardware Performance Monitors (HPM)

See chapter 2.4.12. Hardware Performance Monitors (HPM Extension) for more information.

HPM_NUM_COUNTER **natural** 0

Total number of implemented hardware performance monitor counters (0..29). If this generics is zero no hardware performance monitor logic will be implemented at all.

HPM_CNT_WIDTH **natural** 40

This generic defines the total LSB-aligned size of each HPM counter (size(`[m]hpmcounter*h`) + size(`[m]hpmcounter*`)). The maximum value is 64, the minimal is 1. If the size is less than 64-bit, the unused MSB-aligned counter bits are hardwired to zero.

Internal Instruction Memory

See chapter 3.4. Address Space and 3.5.1. Instruction Memory (IMEM) for more information.

MEM_INT_IMEM_EN **boolean** true

Implement processor internal instruction memory (IMEM) when `true`.

MEM_INT_IMEM_SIZE **natural** 16*1024

Size in bytes of the processor internal instruction memory (IMEM). Has no effect when `MEM_INT_IMEM_EN` is `false`.

MEM_INT_IMEM_ROM **boolean** false

Implement processor-internal instruction memory as read-only memory, which will be initialized with the application image at synthesis time. Has no effect when `MEM_INT_IMEM_EN` is `false`.

Internal Data Memory

See chapter 3.4. Address Space and 3.5.2. Data Memory (DMEM) for more information.

MEM_INT_DMEM_EN **boolean** true

Implement processor internal data memory (DMEM) when `true`.

MEM_INT_DMEM_SIZE **natural** 8*1024

Size in bytes of the processor-internal data memory (DMEM). Has no effect when `MEM_INT_DMEM_EN` is `false`.

Internal Cache Memory

See chapter 3.5.4. Processor-Internal Instruction Cache (iCACHE) for more information.

ICACHE_EN   **boolean**   `false`

Implement processor internal instruction cache when `true`.

ICACHE_NUM_BLOCKS   **natural**   `4`

Number of blocks (cache "pages" or "lines") in the instruction cache. Has to be a power of two. Has no effect when `ICACHE_DMEM_EN` is `false`.

ICACHE_BLOCK_SIZE   **natural**   `64`

Size in bytes of each block in the instruction cache. Has to be a power of two. Has no effect when `ICACHE_EN` is `false`.

ICACHE_ASSOCIATIVITY   **natural**   `1`

Associativity (= number of sets) of the instruction cache. Has to be a power of two. Allowed configurations: 1 = 1 set, direct mapped; 2 = 2way set-associative. Has no effect when `ICACHE_EN` is `false`.

External Memory Interface

See chapter 3.4. Address Space and 3.5.5. Processor-External Memory Interface (WISHBONE) (AXI4-Lite) for more information.

MEM_EXT_EN   **boolean**   `false`

Implement external bus interface (WISHBONE) when `true`.

Processor Peripherals

See chapter 3.5. Processor-Internal Modules for more information.

IO_GPIO_EN **boolean** `true`

Implement general purpose input/output port unit (GPIO) when `true`. When disabled, the `gpio_i` signal is unconnected and the `gpio_o` signal is always low. See chapter 3.5.6. General Purpose Input and Output Port (GPIO) for more information.

IO_MTIME_EN **boolean** `true`

Implement machine system timer (MTIME) when `true`. When disabled, the CPU's machine timer interrupt is not available. The `CPU_EXTENSION_RISCV_Zicsr` has to be enabled if you want to use the machine system timer's interrupt. See chapter 3.5.8. Machine System Timer (MTIME) for more information.

IO_UART0_EN **boolean** `true`

Implement primary universal asynchronous receiver/transmitter (UART0) when `true`. When disabled, the `uart0_rxd_i` and `uart0_cts_i` signals is unconnected and the `uart0_txd_o` and `uart0_rts_o` signals are always low. See chapter 3.5.9. Primary Universal Asynchronous Receiver and Transmitter (UART0) for more information.

IO_UART1_EN **boolean** `true`

Implement secondary universal asynchronous receiver/transmitter (UART1) when `true`. When disabled, the `uart1_rxd_i` and `uart1_cts_i` signals is unconnected and the `uart1_txd_o` and `uart1_rts_o` signals are always low. See chapter 3.5.10. Secondary Universal Asynchronous Receiver and Transmitter (UART1) for more information.

IO_SPI_EN **boolean** `true`

Implement serial peripheral interface controller (SPI) when `true`. When disabled, the `spi_miso_i` signal is unconnected, the `spi_sclk_o` and `spi_mosi_o` signals are always low and the `spi_csn_o` signal is always high. See chapter 3.5.11. Serial Peripheral Interface Controller (SPI) for more information.

IO_TWI_EN **boolean** `true`

Implement two-wire interface controller (TWI) when `true`. When disabled, the `twi_sda_io` and `twi_scl_io` signals are unconnected. See chapter 3.5.12. Two Wire Serial Interface Controller (TWI) for more information.

IO_PWM_EN **boolean** `true`

Implement pulse-width modulation controller (PWM) when `true`. When disabled, the `pwm_o` signal is always low. See chapter 3.5.13. Pulse Width Modulation Controller (PWM) for more information.

IO_WDT_EN **boolean** `true`

Implement watchdog timer (WDT) when `true`. See chapter 3.5.7. Watchdog Timer (WDT) for more information.

IO_TRNG_EN **boolean** `false`

Implement true-random number generator (TRNG) when `true`. See chapter 3.5.14. True Random Number Generator (TRNG) for more information.

IO_CFS_EN **boolean** `false`

Implement custom functions subsystem (CFS) when `true`. See chapter 3.5.15. Custom Functions Subsystem (CFS) for more information.

IO_CFS_CONFIG **std_ulogic_vector(31 downto 0)** `x"00000000"`

This is a "conduit" generic that can be used to pass user-defined CFS implementation flags to the custom functions subsystem entity. See chapter 3.5.15. Custom Functions Subsystem (CFS) for more information.

IO_CFS_IN_SIZE **positive** `32`

Defines the size of the CFS input signal conduit (`cfs_in_i`). See chapter 3.5.15. Custom Functions Subsystem (CFS) for more information.

IO_CFS_OUT_SIZE **positive** `32`

Defines the size of the CFS output signal conduit (`cfs_out_o`). See chapter 3.5.15. Custom Functions Subsystem (CFS) for more information.

IO_NCO_EN **natural** `true`

Implement numerically-controller oscillator (NCO) when `true`. When disabled, the `nco_o` signals is always all-zero. See chapter 3.5.16. Numerically-Controller Oscillator (NCO) for more information.

IO_NEOLED_EN **natural** `true`

Implement smart LED interface (WS2812 / *NeoPixel*©-compatible) (NEOLED) when `true`. When disabled, the `neoled_o` signal is always zero. See chapter 3.5.17. Smart LED Interface (NEOLED) – NeoPixel © Compatible for more information.

3.3. Processor Interrupts

RISC-V Standard Interrupts

The processor setup features the standard RISC-V interrupt lines for "*machine timer interrupt*", "*machine software interrupt*" and "*machine external interrupt*". The software and external interrupt lines are available via the processor's top entity. By default, the timer interrupt is connected to the internal *machine timer* `MTIME` timer unit (→ 3.5.8. Machine System Timer (MTIME)). If this module has not been enabled for synthesis, the machine timer interrupt is also available via the processor's top entity.

NEORV32-Specific Fast Interrupt Requests

As part of the custom/NEORV32-specific CPU extensions, the CPU features 16 fast interrupt request signals ("FIRQ0 – FIRQ15").

> ⓘ The fast interrupt request signals have custom `mip` CSR bits (→ 2.6.2. Machine Trap Setup), custom `mie` CSR bits (→ 2.6.3. Machine Trap Handling) and custom `mcause` CSR trap codes and trap priories (→ 2.8. Traps, Exceptions and Interrupts).

The fast interrupt request signals are divided into two groups. The FIRQs with higher priority (FIRQ0 – FIRQ9) are dedicated for processor-internal usage. The FIRQs with lower priority (FIRQ10 – FIRQ15) are available for custom usage via the processor's top entity signal `soc_firq_i`.

The mapping of the 16 FIRQ channels is shown in the following table:

Channel	Priority	Source (Module)	Description
0	highest	WDT	Watchdog timeout interrupt
1		CFS	Custom functions subsystem (CFS) interrupt (user-defined)
2		UART0 (RXD)	UART0 data received interrupt (RX complete)
3		UART0 (TXD)	UART0 sending done interrupt (TX complete)
4		UART1 (RXD)	UART1 data received interrupt (RX complete)
5		UART1 (TXD)	UART1 sending done interrupt (TX complete)
6		SPI	SPI transmission done interrupt
7		TWI	TWI transmission done interrupt
8		GPIO	GPIO input pin-change interrupt
9		NEOLED	NEOLED buffer TX empty / not full
10		soc_firq_i(0)	Custom platform use; available via processor top signal
11		soc_firq_i(1)	
12		soc_firq_i(2)	
13		soc_firq_i(3)	
14		soc_firq_i(4)	

Channel	Priority	Source (Module)	Description
15	lowest	`soc_firq_i(5)`	

Table 8: Default fast IRQ channel mapping for the NEORV32 processor

3.4. Address Space

The total 32-bit (4GB) address space of the NEORV32 Processor is divided into four main regions:

1. **Instruction memory (IMEM) space** – for instructions and constants.
2. **Data memory (DMEM) space** – for application runtime data (heap, stack, etc.).
3. **Bootloader ROM address space** – for the processor-internal bootloader.
4. **IO/peripheral address space** – for the processor-internal IO/peripheral devices (e.g., UART).
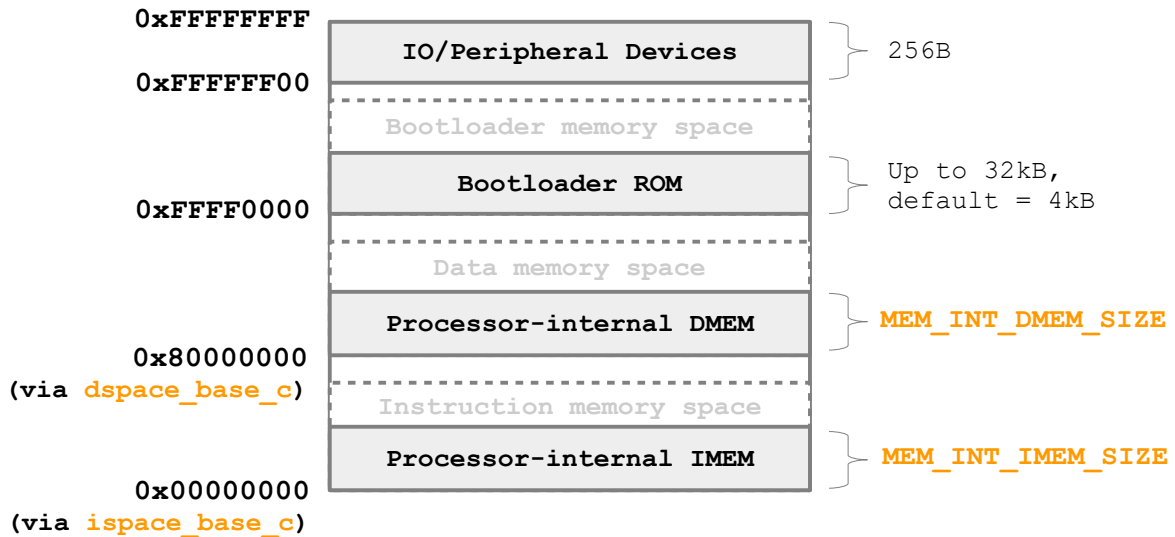
Figure 4: Default NEORV32 processor address space layout

General Address Space Layout

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

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

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

The base address of the bootloader (at `0xFFFF0000`) and the IO region (at `0xFFFFFF00`) for the peripheral devices are also defined in the package and are fixed. These address regions cannot be used for other applications – even if the bootloader or all IO devices are not implemented.

⚠️ When using the processor-internal data and/or instruction memories (`DMEM`/`IMEM`) and using a non-default configuration for the `dspace_base_c` and/or `ispace_base_c` base addresses, the following requirements have to be fulfilled:

- ✔ Both base addresses have to be aligned to a 4-byte boundary.
- ✔ Both base addresses have to be aligned to the according internal memory sizes.

CPU Data and Instruction Access

The CPU can access all of the 4GB address space from the instruction fetch interface (**I**) <u>and also</u> from the data access interface (**D**). These two CPU interfaces are multiplexed by a simple bus switch (`rtl/core/neorv32_busswitch.vhd`) into a <u>single</u> processor-internal bus. All processor-internal memories, peripherals and also the external memory interface are connected to this bus. Hence, both CPU interfaces (instruction fetch & data access) have access to the same (*identical*) address space making the setup a modified von-Neumann architecture.
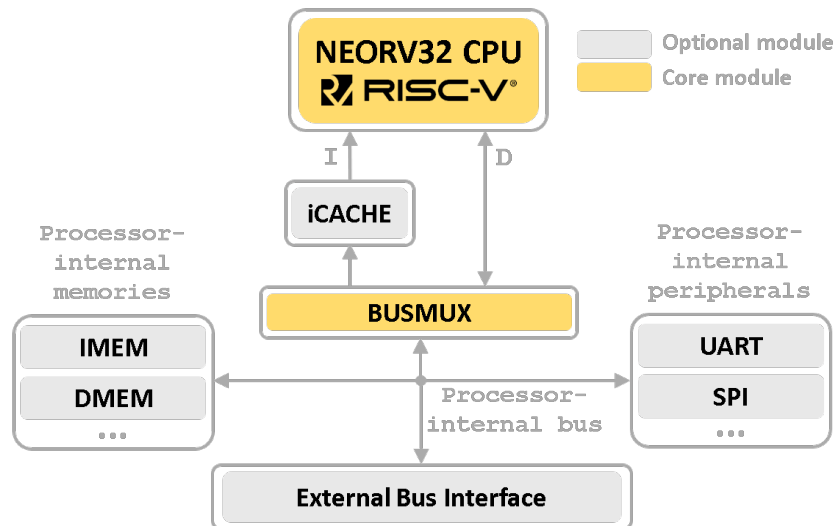

Figure 5: Processor-internal bus architecture

ⓘ The internal processor bus might appear as bottleneck. In order to reduce traffic yam on this bus (when instruction fetch and data interface access the bus at the same time) the instruction fetch of the CPU is equipped with a prefetch buffer. Instruction fetches can be further buffered using the i-cache. Furthermore, data accesses (loads and stores) have higher priority than instruction fetch accesses.

⚠ Please note that all processor-internal components including the peripheral/IO devices can also be accessed from programs running in **less-privileged user mode**. For example, if the system relies on a periodic interrupt from the MTIME timer unit, user-level programs could alter theMTIME configuration corrupting this interrupt. This kind of security issues can be compensated using the 2.4.11. Physical Memory Protection (PMP Extension) system.

Physical Memory Attributes (PMAs)

The processor setup defines four simple attributes for the four processor-internal address space regions:

- **r – read access** (from CPU data access interface, e.g. via "load")
- **w – write access** (from CPU data access interface, e.g. via "store")
- **x – execute access** (from CPU instruction fetch interface)
- **a – atomic access** (from CPU data access interface)

- **b – byte(8-bit)-accessible** (when writing)
- **h – half-word(16-bit)-accessible** (when writing)
- **w – word(64-bit)-accessible** (when writing)

The following table shows the provided physical memory attributes of each region. Additional attributes (like denying execute right for certain region of the IMEM) can be provided using the RISC-V Physical Memory Protection (PMP, → 2.4.11. Physical Memory Protection (PMP Extension)) extension.

#	Region	Base address	Size	Attributes
4	IO/peripheral devices	0xFFFFFF00	256 Byes	r/w[13]/a/w
3	Bootloader ROM	0xFFFF0000	Up to 32kB	r/x
2	DMEM	0x80000000	Up to 2GB (-64kB)	r/w/x/a/b/h/w
1	IMEM	0x00000000	Up to 2GB	r/w/x/a/b/h/w

Table 9: Physical memory attributes of the four main processor address space regions

Only the CPU of the processor has access to the internal memories and IO devices, hence all accesses are always exclusive. Accessing a memory region in a way that violates the provided attributes will trigger a load/store/instruction fetch access exception or will return a failed atomic access result, respectively.

The physical memory attributes of memories and/or devices connected via the external bus interface have to defined by those components or the interconnection fabric.

Internal Memories

The processor can implement internal memories for instructions (IMEM) and data (DMEM), which will be mapped to FPGA block RAMs. The implementation of these memories is controlled via the boolean `MEM_INT_IMEM_EN` and `MEM_INT_DMEM_EN` generics.

The size of these memories are configured via the `MEM_INT_IMEM_SIZE` and `MEM_INT_DMEM_SIZE` generics (in bytes), respectively. The processor-internal instruction memory (IMEM) can optionally be implemented as true ROM (`MEM_INT_IMEM_ROM`), which is initialized with the application code during synthesis.

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

External Memory/Bus Interface

Any CPU access (data or instructions), which **does not fulfill one** of the following conditions, is forwarded to the external memory interface:

- Access to the processor-internal IMEM <u>and</u> processor-internal IMEM is implemented
- Access to the processor-internal DMEM <u>and</u> processor-internal DMEM is implemented
- Access to the bootloader ROM and beyond → addresses >= `BOOTROM_BASE` (default `0xFFFF0000`)

---

13 Read/write accesses depend on the actual device configuration (like which devices are implemented). Also, not all registers of all IO devices provide read and/or write access capabilities.

will never be forwarded to the external memory interface

The external bus interface is available when the `MEM_EXT_EN` generic is `true`. If this interface is deactivated, any access exceeding the internal memories or peripheral devices will trigger a bus access fault exception.

External Memory/Bus Interface – Instruction Memory Example

```
MEM_INT_IMEM_EN   = true
MEM_INT_IMEM_SIZE = 1024 byte
MEM_EXT_EN        = true
```

All accesses beyond address `0x000003ff` (base + size: `0x00000000` + `1024 bytes` - 1) are forwarded to the external memory interface. To connect an external memory with 1024 bytes starting at the end of the processor-internal IMEM, the base address of this external memory has to be `0x00000400`. If the external memory interface is not implemented, any access beyond `0x000003ff` will trigger an instruction bus access fault exception

3.5. Processor-Internal Modules

Basically, the processor is a SoC consisting of the NEORV32 CPU, peripheral/IO devices, embedded memories, an external memory interface and a bus infrastructure to interconnect all units. Additionally, the system implements an internal reset generator and a global clock generator/divider.

Internal Reset Generator

Most processor-internal modules – except for the CPU and the watchdog timer – do not have a dedicated

reset signal. However, all devices can be reset by software by clearing the corresponding unit's control register. The automatically included application start-up code will perform such a software-reset of all modules to ensure a clean system reset state. The hardware reset signal of the processor can either be triggered via the external reset pin (`rstn_i`, **low-active**) or by the internal watchdog timer (if implemented). Before the external reset signal is applied to the system, it is filtered (so no spike can generate a reset, a minimum active reset period of one clock cycle is required) and extended to have a minimal duration of four clock cycles.

Internal Clock Divider

An internal clock divider generates 8 clock signals derived from the processor's main clock input `clk_i`. These derived clock signals are not actual *clock signals*. Instead, they are derived from a simple counter and are used as "clock enable" signal by the different processor modules. Thus, the whole design operates using only the main clock signal (single clock domain). Some of the processor peripherals like the Watchdog or the UARTs can select one of the derived clock enabled signals for their internal operation. If none of the connected modules require a clock signal from the divider, it is automatically deactivated to reduce dynamic power.

The peripheral devices, which feature a time-based configuration, provide a three-bit prescaler select in their according control register to select one out of the eight available clocks. The mapping of the prescaler select bits to the actually obtained clock are shown in the table below. Here, *f* represents the processor main clock from the top entity's `clk_i` signal.

Prescaler bits	000	001	010	011	100	101	110	111
Resulting clock	f/2	f/4	f/8	f/64	f/128	f/1024	f/2048	f/4096

Peripheral / IO Devices

The processor-internal peripheral/IO devices are located at the end of the 32-bit address space at base address `0xFFFFFF00`. A region of 256 bytes is reserved for this devices. Hence, all peripheral/IO devices are accessed using a memory-mapped scheme. A special linker script as well as the NEORV32 core software library abstract the specific memory layout for the user.

⚠️ When accessing an IO device, that hast not been implemented (e.g., via the `IO_XXX_EN` generics), a load/store access fault exception is triggered.

⚠️ **The peripheral/IO devices can only be written in full-word mode** (i.e. 32-bit). Byte or half-word (8/16-bit) writes will trigger a store access fault exception. Read accesses are not size constrained. Processor-internal memories as well as modules connected to the external memory interface can still be written with a byte-wide granularity.

ℹ️ You should use the provided core software library to interact with the peripheral devices. This prevents incompatibilities with future versions, since the hardware driver functions handle all the register and register bit accesses.

ℹ️ Most of the IO devices do not have a hardware reset. Instead, the devices are reset via software by writing zero to the unit's control register. A general software-based reset of all devices is done by the application start-up code `crt0.S`.

Nomenclature for the Peripheral / IO Devices Listing

Each peripheral device chapter features a register map showing accessible control and data registers of the according device including the implemented control and status bits. You can directly interact with these registers/bits via the provided <u>C-code defines</u>. These defines are set in the main processor core library include file `sw/lib/include/neorv32.h`. The registers and/or register bits, which can be accessed directly using plain C-code, are marked with a **[C]**.

Not all registers or register bits can be arbitrarily read/written. The following read/write access types are available:

`r/w`   Registers / bits can be read and written.

`r/-`   Registers / bits are read-only. Any write access to them has no effect.

`0/w`   These registers / bits are write-only. They auto-clear in the next cycle and are always read as zero.

ℹ️ Bits / registers that are not listed in the register map tables are not (yet) implemented. These registers / bits are always read as zero. A write access to them has no effect, but user programs should only write zero to them to keep compatible with future extension.

ℹ️ When writing to read-only registers, the access is nevertheless acknowledged, but no actual data is written. When reading data from a write-only register the result is undefined.

3.5.1. Instruction Memory (IMEM)

Overview

Hardware source file(s):	`neorv32_imem.vhd`	
Software driver file(s):	`none`	Implicitly used
Top entity ports:	`none`	
Configuration generics:	`MEM_INT_IMEM_EN`	Implement processor-internal IMEM when `true`

	`MEM_INT_IMEM_SIZE`	IMEM size in bytes
	`MEM_INT_IMEM_ROM`	Implement IMEM as ROM when `true`
CPU interrupts:	`none`	-

A processor-internal instruction memory can be enabled for synthesis via the processor's `MEM_INT_IMEM_EN` generic. The size in bytes is defined via the `MEM_INT_IMEM_SIZE` generic. If the IMEM is implemented, the memory is mapped into the instruction memory space and located right at the beginning of the instruction memory space (default `ispace_base_c` = `0x00000000`).

By default, the IMEM is implemented as RAM, so the content can be modified during run time. This is required when using a bootloader that can update the content of the IMEM at any time. If you do not need the bootloader anymore – since your application development is done and you want the program to permanently reside in the internal instruction memory – the IMEM can also be implemented as true read-only memory. In this case set the `MEM_INT_IMEM_ROM` generic of the processor's top entity to `true`.

When the IMEM is implemented as ROM, it will be initialized during synthesis with the actual application program image. Based on your application the toolchain will automatically generate a VHDL initialization file `rtl/core/neorv32_application_image.vhd`, which is automatically inserted into the IMEM. If the IMEM is implemented as RAM, the memory will not be initialized at all.

3.5.2. Data Memory (DMEM)

Overview

Hardware source file(s):	`neorv32_dmem.vhd`	
Software driver file(s):	`none`	Implicitly used
Top entity ports:	`none`	
Configuration generics:	`MEM_INT_DMEM_EN`	Implement processor-internal DMEM when `true`
	`MEM_INT_DMEM_SIZE`	DMEM size in bytes
CPU interrupts:	`none`	-

A processor-internal data memory can be enabled for synthesis via the processor's `MEM_INT_DMEM_EN` generic. The size in bytes is defined via the `MEM_INT_DMEM_SIZE` generic. If the DMEM is implemented, the memory is mapped into the data memory space and located right at the beginning of the data memory space (default `dspace_base_c` = `0x80000000`). The DMEM is always implemented as RAM.

3.5.3. Bootloader ROM (BOOTROM)

Overview

Hardware source file(s):	`neorv32_boot_rom.vhd`	
Software driver file(s):	`none`	Implicitly used
Top entity ports:	`none`	

| Configuration generics: | `BOOTLOADER_EN` | Implement bootloader when `true` |
| CPU interrupts: | `none` | - |

As the name already suggests, the boot ROM contains the read-only bootloader image. When the bootloader is enabled via the `BOOTLOADER_EN` generic it is directly executed after system reset.

The bootloader ROM is located at address `0xFFFF0000`. This location is fixed and the bootloader ROM size must not exceed 32kB. The bootloader read-only memory is automatically initialized during synthesis via the `rtl/core/neorv32_bootloader_image.vhd` file, which is generated when compiling and installing the bootloader sources.

The bootloader ROM address space cannot be used for other applications even when the bootloader is not implemented.

Boot Configuration

If the bootloader is implemented, the CPU starts execution after reset right at the beginning of the boot ROM. If the bootloader is *not* implemented, the CPU starts execution at the beginning of the instruction memory space (defined via `ispace_base_c` constant in the `neorv32_package.vhd` VHDL package file, default `ispace_base_c` = `0x00000000`). In this case, the instruction memory has to contain a valid executable – either by using the internal IMEM with an initialization during synthesis or by a user-defined initialization process.

3.5.4. Processor-Internal Instruction Cache (iCACHE)

Overview

| Hardware source file(s): | `neorv32_icache.vhd` | |
| Software driver file(s): | `none` | Implicitly used |

Top entity ports:	none	
Configuration generics:	`ICACHE_EN`	Implement processor-internal instruction cache when `true`
	`ICACHE_NUM_BLOCKS`	Number of cache blocks (/pages/lines)
	`ICACHE_BLOCK_SIZE`	Size of a cache block in bytes
	`ICACHE_ASSOCIATIVITY`	Associativity / number of sets
CPU interrupts:	none	-

The processor features an optional cache for instructions to compensate memories with high latency. The cache is directly connected to the CPU's instruction fetch interface and provides a full-transparent buffering of instruction fetch accesses to the entire 4GB address space.

⚠ The instruction cache is intended to accelerate instruction fetch **via the external memory interface**. Since all processor-internal memories provide an access latency of one cycle (by default), **caching internal memories does not bring any performance gain**. However, it *might* reduce traffic on the processor-internal bus.

The cache is implemented if the `ICACHE_EN` generic is true. The size of the cache memory is defined via `ICACHE_BLOCK_SIZE` (the size of a single cache block/page/line in bytes; has to be a power of two and >= 4 bytes), `ICACHE_NUM_BLOCKS` (the total amount of cache blocks; has to be a power of two and >= 1) and the actual cache associativity `ICACHE_ASSOCIATIVITY` (number of sets; 1 = direct-mapped, 2 = 2-way set-associative, has to be a power of two and >= 1).

If the cache associativity (`ICACHE_ASSOCIATIVITY`) is > 1 the LRU replacement policy (least recently used) is used.

ⓘ Keep the features of the targeted FPGA's memory resources (block RAM) in mind when configuring the cache size/layout to maximize and optimize resource utilization.

By executing the `ifence.i` instruction (`Zifencei` CPU extension) the cache is cleared and a reload from main memory is forced. Among other things, this allows to implement self-modifying code.

Bus Access Fault Handling

The cache always loads a complete cache block (`ICACHE_BLOCK_SIZE` bytes) aligned to the size of a cache block if a *miss* is detected. If any of the accessed addresses within a single block do not successfully acknowledge (i.e. issuing an error signal or timing out) the whole cache block is invalidate and any access to an address within this cache block will also raise an instruction fetch bus error fault exception.

> ⚠️ If the instruction cache is implemented, the default bus timeout (configured in the processor's VHDL package file, see section 2.10. Bus Interface) is automatically modified: The default bus timeout is rounded to the next power-of-two and multiplied by the number of word in one cache block minus one. Example:
>
> ```
> default bus_timeout_c = 63 cycles
> ICACHE_BLOCK_SIZE = 256 (bytes
> actual bus timeout = 64 * (256/4) -1 = 4095 cycles
> ```

3.5.5. Processor-External Memory Interface (WISHBONE) (AXI4-Lite)

Overview

Hardware source file(s):	`neorv32_wishbone.vhd`	
Software driver file(s):	`none`	Implicitly used

Top entity ports:	`wb_tag_o`	Request tag output; access identifier (4-bit)
	`wb_adr_o`	Address output (32-bit)
	`wb_dat_i`	Data input (32-bit)
	`wb_dat_o`	Data output (32-bit)
	`wb_we_o`	Write enable (1-bit)
	`wb_sel_o`	Byte enable (4-bit)
	`wb_stb_o`	Strobe (1-bit)
	`wb_cyc_o`	Valid cycle (1-bit)
	`wb_tag_i`	Response tag (1-bit)
	`wb_ack_i`	Acknowledge (1-bit)
	`wb_err_i`	Bus error (1-bit)
	fence_o	Indicates an executed `fence` instruction
	fencei_o	Indicates an executed `fence.i` instruction
Configuration generics:	`MEM_EXT_EN`	Enable external memory interface when `true`
Configuration constants: → VHDL package file `neorv32_package.vhd`	`wb_pipe_mode_c`	When false (default): Classic/standard Wishbone protocol; when true: Pipelined Wishbone protocol
	`bus_timeout_c`	Cycles after which an unacknowledged bus access will time out, get canceled and triggers a bus exception interrupt, default = 127
	`xbus_big_endian_c`	Byte-order (Endianness) of external memory interface (true=BIG (default), false=little)
CPU interrupts:	`none`	-

The external memory interface uses the Wishbone interface protocol. The external interface port is available when the `MEM_EXT_EN` generic is `true`. This interface can be used to attach external memories, custom hardware accelerators additional IO devices or all other kinds of IP blocks. All memory accesses from the CPU, that do not target the internal bootloader ROM, the internal IO region or the internal data/instruction memories (if implemented at all) are forwarded to the Wishbone gateway and thus to the external memory interface.

> ⓘ When using the default processor setup, all access addresses between `0x00000000` and `0xffff0000` (= beginning of processor-internal BOOT ROM) are delegated to the external memory / bus interface if they are not targeting the (actually enabled/implemented) processor-internal instruction memory (IMEM) or the (actually enabled/implemented) processor-internal data memory (DMEM). See section 3.4. Address Space for more information.

Wishbone Bus Protocol

The external memory interface either uses **Standard ("classic") Wishbone Transactions** (default) or **Pipelined Wishbone Transactions**. The transaction protocol is defined via the `wb_pipe_mode_c` constant in the in the main VHDL package file (`rtl/neorv32_package.vhd`):

```
-- (external) bus interface --
constant wb_pipe_mode_c : boolean := false;
```

When `wb_pipe_mode_c` is disabled, all bus control signals including **STB** are active (and stable) until the transfer is acknowledged/terminated. If `wb_pipe_mode_c` is enabled, all bus control **except STB** are active (and stable) until the transfer is acknowledged/terminated. In this case, **STB** is active only during the very first bus clock cycle.

"Classic" Wishbone read-access   "Pipelined" Wishbone write-access
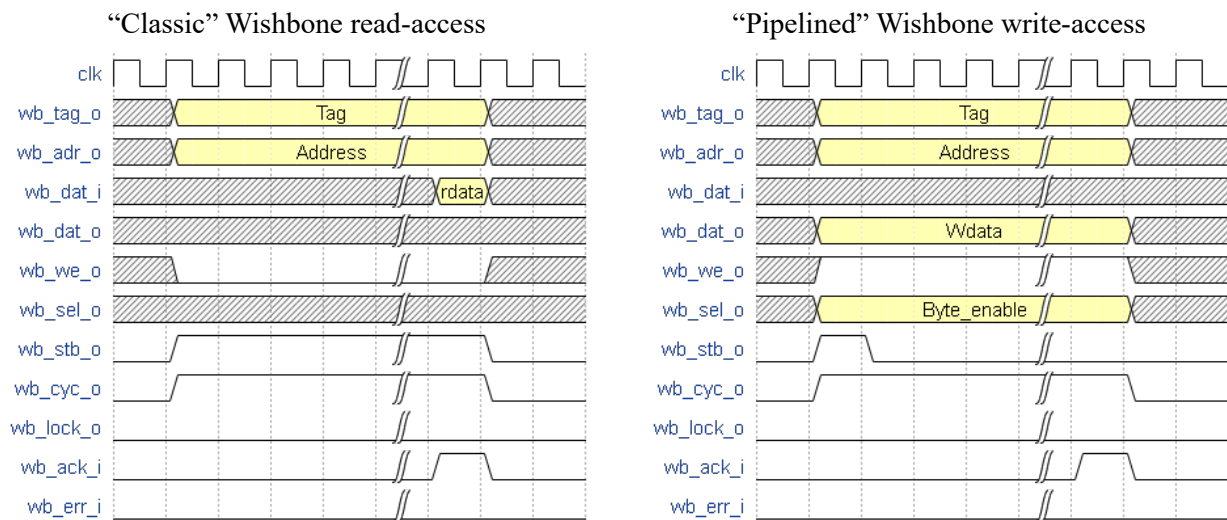
> ℹ️ A detailed description of the implemented Wishbone bus protocol and the according interface signals can be found in the data sheet "*Wishbone B4 – WISHBONE System-on-Chip (SoC) Interconnection Architecture for Portable IP Cores*". A copy of this document can be found in the `docs` folder of this project.

Latency

The Wishbone gateway introduces two additional latency cycles: Processor-outgoing and -incoming signals are fully registered. Thus, any access from the CPU to a processor-external devices takes at least four cycles if the accessed device can respond within the same cycle the external bus access is initiated.

If the CPU cancels an active Wishbone transaction, the bus interface goes into suspend mode, that still keeps the transaction active for some time to allow the bus system to acknowledge the transfer. If the bus system still does not terminate the transfer, the bus interface forces a termination.

Bus Access Timeout

Whenever the CPU starts a memory access, an internal timer is started. If the accessed address (the memory or peripheral device) does not acknowledge the transfer within a certain time, the bus access is canceled and a load/store/instruction fetch bus access fault exception is raised – depending on the bus access type.

The processor-internal memories and peripherals will always acknowledge the transfers within two cycles.

Of course, a bus timeout will occur if accessing unused address locations. For example, a bus timeout and thus, a load/store bus access fault will occur when trying to access an IO device that has not been implemented.

The maximum bus cycle time (default = 127 cycles), after which a **bus access exception will be triggered**, is defined via the global `bus_timeout_c` constant in the project's main VHDL package file (`rtl/neorv32_package.vhd`):

```
-- (external) bus interface --
constant bus_timeout_c : natural := 127;
```

Bus accesses via the external memory interface are acknowledged via the Wishbone-compatible `wb_ack_i` signal. The external bus accesses can be terminated/aborted at any time by an accessed device/memory via the Wishbone-compatible `wb_err_i` signal.

Wishbone Tag

The 4-bit wishbone `wb_tag_o` signal provides additional information regarding the access type. This signal is compatible to the AXI4 `AxPROT` signal.

`wb_tag_o(0)`	1: Privileged access (CPU is in machine mode); 0: Unprivileged access
`wb_tag_o(1)`	Always zero (indicating "secure access")
`wb_tag_o(2)`	1: Instruction fetch access, 0: Data access
`wb_tag_o(3)`	Set when the CPU requests an exclusive bus access

Exclusive / Atomic Bus Access

If the atomic memory access CPU extension (via `CPU_EXTENSION_RISCV_A`) is enabled, the CPU can request an atomic/exclusive bus access via the external memory interface.

The load-reservate instruction (`LR.W`) will set the `wb_tag_o(3)` signal telling the bus interconnect to establish a *reservation* for the current accessed address (start of an exclusive access). The store-conditional instruction (`SC.W`) evaluates the status of the response tag signal `wb_tag_i`. If the reservation was successful and no other controller (CPU core, DMA, …) accessed the reserved address the signal has to be set by the memory system to indicate a successful exclusive access.

The reservation of a specific address is removed whenever a bus transfer accesses this address without having `wb_tag_o(3)` set.

Endianness

The NEORV32 CPU and the Processor setup are BIG-endian architectures. However, to allow a connection to a little-endian memory system the external bus interface provides an Endianness configuration. The Endianness can be configured via the global `xbus_big_endian_c` constant in the main VHDL package file (`rtl/neorv32_package.vhd`). By default, the external memory interface uses BIG-endian byte-order.

```
-- (external) bus interface --
constant xbus_big_endian_c : boolean := true;
```

Application software can check the Endianness configuration of the external bus interface via the SYSINFO_FEATURES_MEM_EXT_ENDIAN flag in the processor's SYSINFO module (see section 3.5.18. System Configuration Information Memory (SYSINFO) for more information).

AXI4-Lite Connectivity

The **AXI4-Lite** wrapper (`rtl/top_templates/neorv32_top_axi4lite.vhd`) provides a Wishbone-to-AXI4-Lite bridge, compatible with Xilinx Vivado (IP packager and block design editor). All entity signals of this wrapper are of type *std_logic* or *std_logic_vector*, respectively.

The AXI Interface has been verified using Xilinx Vivado *IP Packager* and *Block Designer*. The AXI interface port signals are automatically detected when packaging the core.
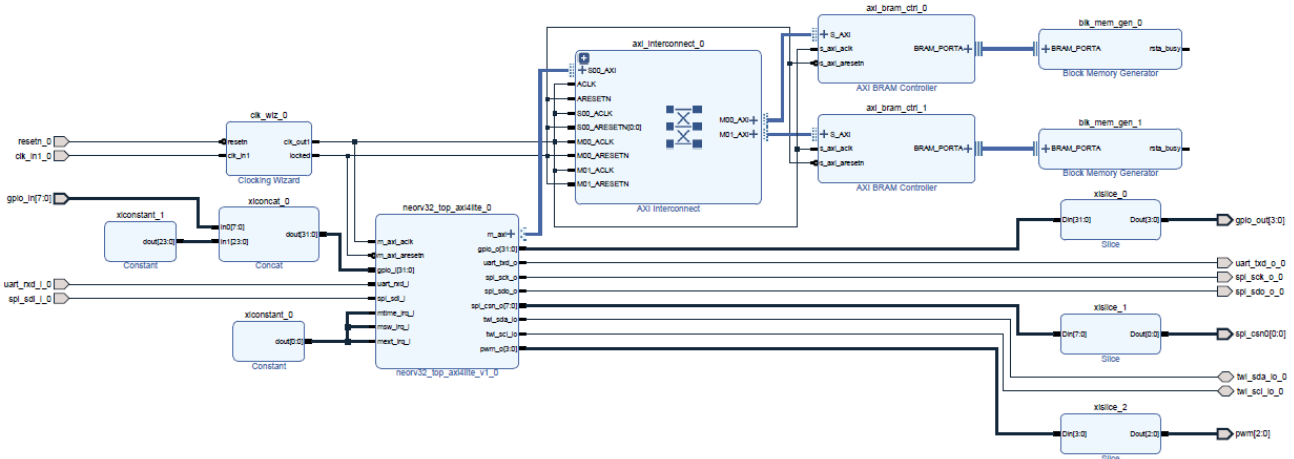

Figure 6: Example AXI SoC using Xilinx Vivado

3.5.6. General Purpose Input and Output Port (GPIO)

Overview

Hardware source file(s):	neorv32_gpio.vhd
Software driver file(s):	neorv32_gpio.c

	`neorv32_gpio.h`	
Top entity ports:	`gpio_o`	32-bit parallel output port
	`gpio_i`	32-bit parallel input port
Configuration generics:	`IO_GPIO_EN`	Implement GPIO port unit when `true`
CPU interrupts:	`Fast IRQ channel 8`	Pin-change interrupt [3.3. Processor Interrupts]

Theory of Operation

The general purpose parallel IO port unit provides a simple 32-bit parallel input port and a 32-bit parallel output port. These ports can be used chip-externally (for example to drive status LEDs, connect buttons, etc.) or system-internally to provide control signals for other IP modules. When the modules is disabled for implementation the GPIO output port is tied to zero.

Pin-Change Interrupt

The parallel input port `gpio_i` features a single pin-change interrupt. Whenever an input pin has a low-to-high or high-to-low transition, the interrupt is triggered. By default, the pin-change interrupt is disabled and can be enabled using a bit mask that has to be written to the `GPIO_INPUT` register. Each set bit in this mask enables the pin-change interrupt for the corresponding input pin. If more than one input pin is enabled for triggering the pin-change interrupt, any transition on one of the enabled input pins will trigger the CPU's pin-change interrupt. If the modules is disabled for implementation, the pin-change interrupt is also permanently disabled.

Register Map

Address	Name [C]	Bit(s)	R/W	Function
`0xFFFFFF80`	`GPIO_INPUT`	`31..0`	`r/-`	Parallel input port
		`31..0`	`-/w`	Parallel input pin-change IRQ enable mask
`0xFFFFFF84`	`GPIO_OUTPUT`	`31..0`	`r/w`	Parallel output port

Table 10: GPIO port unit register map

3.5.7. Watchdog Timer (WDT)

Overview

Hardware source file(s):	`neorv32_wdt.vhd`

Software driver file(s):	`neorv32_wdt.c` `neorv32_wdt.h`
Top entity ports:	`none`
Configuration generics:	`IO_WDT_EN` Implement Watchdog timer when `true`
CPU interrupts:	`Fast IRQ channel 0` Watchdog timer overflow [3.3. Processor Interrupts]

Theory of Operation

The watchdog (WDT) provides a last resort for safety-critical applications. The WDT has an internal 20-bit wide counter that needs to be reset every now and then by the user program. If the counter overflows, either a system reset or an interrupt is generated (depending on the configured operation mode).

Configuration of the watchdog is done by a single controle register `WDT_CT`. The watchdog is enabled by setting the `WDT_CT_EN` bit. The clock used to increment the internal counter is selected via the 3-bit `WDT_CT_CLK_SELx` prescaler:

WDT_CT_CLK_SWLx	000	001	010	011	100	101	110	111
Main clock prescaler:	2	4	8	64	128	1024	2048	4096
Timeout period in clock cycles:	2 097 152	4 194 304	8 388 608	67 108 864	134 217 728	1 073 741 824	2 147 483 648	4 294 967 296

Whenever the internal timer overflows the watchdog executes one of two possible actions: Either a hard processor reset is triggered or an interrupt is requested at CPU's fast interrupt channel #0. The `WDT_CT_MODE` bit defines the action to be taken on an overflow: When cleared, the Watchdog will trigger an IRQ, when set the WDT will cause a system reset. The configured actions can also be triggered manually at any time by setting the `WDT_CT_FORCE` bit. The watchdog is reset by setting the `WDT_CT_RESET` bit.

The cause of the last action of the watchdog can be determined via the `WDT_CT_RCAUSE` flag. If this flag is zero, the processor has been reset via the external reset signal. If this flag is set the last system reset was initiated by the watchdog.

The Watchdog control register can be locked in order to protect the current configuration. The lock is activated by setting bit `WDT_CT_LOCK`. In the locked state any write access to the configuration flags is ignored (see table below, "accessible if locked"). Read accesses to the control register are not effected. The lock can only be removed by a system reset (via external reset signal or via a watchdog reset action).

Register Map

Address	Name [C]	Bit(s) (Name) [C]	R/W	Writable if locked?	Function
0xFFFFFF8C	WDT_CT	0 WDT_CT_EN	r/w	no	Watchdog enable
		1 WDT_CT_CLK_SEL0	r/w	no	Clock prescaler select bit 0
		2 WDT_CT_CLK_SEL1	r/w	no	Clock prescaler select bit 1
		3 WDT_CT_CLK_SEL2	r/w	no	Clock prescaler select bit 2
		4 WDT_CT_MODE	r/w	no	Overflow action: 1=reset, 0=IRQ
		5 WDT_CT_RCAUSE	r/-	-	Cause of last system reset; 0=caused by external reset signal, 1=caused by watchdog
		6 WDT_CT_RESET	-/w	yes	Watchdog reset when set, auto-clears
		7 WDT_CT_FORCE	-/w	yes	Force configured watchdog action when set, auto-clears
		8 WDT_CT_LOCK	r/w	no	Lock access to configuration when set, clears only on system reset (via external reset signal OR watchdog reset action)

Table 11: WDT register map

3.5.8. Machine System Timer (MTIME)

Overview

Hardware source file(s):	`neorv32_mtime.vhd`	
Software driver file(s):	`neorv32_mtime.c` `neorv32_mtime.h`	
Top entity ports:	*mtime_i*	System time input if processor-internal MTIME unit is not used
Configuration generics:	`IO_MTIME_EN`	Implement MTIME when `true`
CPU interrupts:	`MTI`	Machine timer interrupt

Theory of Operation

The MTIME machine system timer implements the memory-mapped `mtime` timer from the official RISC-V specifications. This unit features a 64-bit system timer incremented with the primary processor clock.

The 64-bit system time can be accessed via the `MTIME_LO` and `MTIME_HI` registers. A 64-bit time compare register – accessible via `MTIMECMP_LO` and `MTIMECMP_HI` – can be used to trigger an interrupt to the CPU whenever `MTIME >= MTIMECMP`. This interrupt is directly forwarded to the CPU's `MTI` interrupt. The time and compare registers can also be accessed as single 64-bit registers via the `MTIME` and `MTIMECMP` defines.

The system time is also readable via the CPU's `time[h]` CSRs. If the processor-internal MTIME unit is NOT implemented, the top's `mtime_i` signal is used to update the `time[h]` CSRs.

> ℹ️ There is no need to acknowledge the MTIME interrupt. The interrupt request is a single-shot signal, so the CPU is triggered <u>once</u> if the system time is greater than or equal to the compare time. Hence, another MTIME IRQ is only possible when increasing the compare time.

The 64-bit counter and the 64-bit comparator are implemented as 2×32-bit counters and comparators with a registered carry to prevent a 64-bit carry chain ad thus, to simplify timing closure.

Register Map

Address	Name [C]	Bit(s)	R/W	Function
0xFFFFFF90	MTIME_LO	31:0	r/w	Machine system time, low word
0xFFFFFF94	MTIME_HI	31:0	r/w	Machine system time, high word
0xFFFFFF98	MTIMECMP_LO	31:0	r/w	Time compare, low word
0xFFFFFF9C	MTIMECMP_HI	31:0	r/w	Time compare, high word

Table 12: MTIME register map

> ⚠️ Just like all peripheral/IO devices, the registers of the MTIME system timer can only be written in full 32-bit word mode (using `sw` instruction). All other write accesses will have no effect on MTIME and will trigger a store fault exception.

3.5.9. Primary Universal Asynchronous Receiver and Transmitter (UART0)

Overview

Hardware source file(s):	`neorv32_uart.vhd`	
Software driver file(s):	`neorv32_uart.c` `neorv32_uart.h`	
Top entity ports:	`uart0_txd_o`	Serial transmitter output UART0
	`uart0_rxd_i`	Serial receiver input UART0
	`uart0_rts_o`	HW flow control: UART0.RX ready to receive
	`uart0_cts_i`	HW flow control: UART0.TX allowed to send
Configuration generics:	`IO_UART0_EN`	Implement UART0 when `true`
CPU interrupts:	`Fast IRQ channel 2` `Fast IRQ channel 3`	RX done interrupt TX done interrupt [3.3. Processor Interrupts]

⚠️ Please note that 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**. For compatibility, all C-language function calls to `neorv32_uart_*` are mapped to the according primary UART (UART0) `neorv32_uart0_*` functions.

Theory of Operation

In most cases, the UART is a standard interface used to establish a communication channel between the computer/user and an application running on the processor platform. The NEORV32 UARTs features a standard configuration frame configuration: 8 data bits, an optional parity bit (even or odd) and 1 stop bit. The parity and the actual Baudrate are configurable by software.

The UART0 is enabled by setting the `UART_CT_EN` bit in the UART control register `UART0_CT`. The actual transmission Baudrate (like "19200") is configured via the 12-bit `UART_CT_BAUDxx` **baud prescaler** and the 3-bit `UART_CT_PRSCx` **clock prescaler**.

UART_CT_PRSCx	000	001	010	011	100	101	110	111
Resulting clock prescaler:	2	4	8	64	128	1024	2048	4096

$$Baudrate = \frac{f_{main}[Hz]/\text{clock\_prescaler}}{\text{baud\_prescaler}+1}$$

A new transmission is started by writing the data byte to the lowest byte of the `UART0_DATA` register. The transfer is completed when the `UART_CT_TX_BUSY` control register flag returns to zero. A new received byte is available when the `UART_DATA_AVAIL` flag of the `UART0_DATA` register is set.

A "frame error" in a received byte (broken stop bit) is indicated via the `UART_DATA_FERR` flag in the `UART0_DATA` register.

RX Double-Buffering

The UART receive engine provides a simple data buffer with two entries. These two entries are transparent for the user. The transmitting device can send up to 2 chars to the UART without risking data loss. If another char is sent before at least one char has been read from the buffer data loss occurs. This situation can be detected via the receiver overrun flag `UART_DATA_OVERR` in the `UART0_DATA` register. The flag is automatically cleared after reading `UART0_DATA`.

Parity Modes

The parity flag is added if the `UART_CT_PMODE1` flag is set. When `UART_CT_PMODE0` is zero the UART operates in "even parity" mode. If this flag is set the UART operates in "odd parity" mode. Parity errors in received data are indicated via the `UART_DATA` `UART_DATA_PERR` flag. This flag is updated with each new received character. A frame error in the received data (i.e. stop bit is not set) is indicated via the `UART0_DATA` `UART_DATA_PERR` flag, which is also updated with each new received character

Hardware Flow Control – RTS/CTS

The UART supports hardware flow control using the standard **CTS** (clear to send) and/or **RTS** (ready to send / *ready to receive "RTR"*) signals. Both hardware control flow mechanisms can be enabled in parallel.

If RTS hardware flow control is enabled by setting the `UART_CT_RTS_EN` control register flag, the UART's will pull the `uart0_rts_o` signal **low** if the receiver is idle and no received data is waiting to get read by application software. `uart0_rts_o` is always LOW if the UART is disabled.

> *i* The RTS line is deasserted (going high) as soon as the start bit of a new incoming char has been detected. The transmitting device continues sending the current char and can also **send another char** (due to the RX double-buffering) [*this is done by most terminal programs*]. Any additional data send when RTS is still asserted will override the RX input buffer causing data loss.

If CTS hardware flow control is enabled by setting the `UART_CT_CTS_EN` control register flag, the UART's transmitter will not start sending a new char until the `uart0_cts_i` signal goes **low**. If a new data to be send is written to the UART data register while `uart0_cts_i` is not asserted, the UART will wait for `uart0_cts_i` to become asserted again before sending starts. During this time, the UART busy flag `UART_CT_TX_BUSY` remains set.

> *i* If CTS is asserted, no **new** data transmission will be started by the UART. The state of the CTS signals has no effect on a transmission being in progress.

Signal changes on `uart0_cts_i` during an active transmission are ignored. Application software can check the current state of the `uart0_cts_o` input signal via the `UART_CT_CTS` control register flag.

> *i* Please note that – just like the RXD and TXD signals – the RTS and CTS signals have to be cross-coupled between devices.

Interrupts

The UART features two interrupts: the TX done interrupt is triggered when a transmit has finished. The RX done interrupt is triggered when a data byte has been received. If the UART0 is not implemented, the UART0's serial output port is tied to zero and the UART0 interrupts are permanently tied to zero as well.

> ⓘ The UART's RX interrupt is always triggered when a new data word has arrived – regardless of the state of the RX double-buffer.

Simulation Mode

The default UART0 operation will transmit any data written to the `UART0_DATA` register via the TX line at the defined baud rate. Even though the default testbench provides a simulated UART0 receiver, which outputs any received char to the simulator console, such a transmission takes a lot of time. To accelerate UART0 output during simulation (and also to dump large amounts of data for further processing like verification) the UART0 features a *simulation mode*.

The simulation mode is enabled by setting the `UART_CT_SIM_MODE` bit in the UART0's control register `UART0_CT`. Any further UART0 configuration bits are irrelevant, but the UART0 has to be enabled via the `UART_CT_EN` bit. When the simulation mode is enabled, any written char (bits 7:0) to `UART0_DATA` is directly output as ASCII char to the simulator console. Additionally, all text is also stored to a text file `neorv32.uart0.sim_mode.text.out` in the simulation home folder. Furthermore, the whole 32-bit word written to `UART0_DATA` is stored as plain 8-char hexadecimal value to a second text file `neorv32.uart0.sim_mode.data.out` also located in the simulation home folder.

If the UART is configured for simulation mode there will be no physical UART0 transmissions via `uart0_txd_o` at all. Furthermore, **no interrupts** (RX done or TX done) will be triggered in any situation.

> ⓘ More information regarding the simulation-mode of the UART0 can be found in chapter 5.12. Simulating the Processor.

Register Map

Address	Name [C]	Bit(s) (Name) [C]		R/W	Function
0xFFFFFFA0	UART0_CT	11:0	UART_CT_BAUDxx	r/w	12-bit BAUD configuration value
		12	*UART_CT_SIM_MODE*	*r/w*	*Enable simulation output mode (see below)*
		20	UART_CT_RTS_EN	r/w	Enable RTS hardware flow control
		21	UART_CT_CTS_EN	r/w	Enable CTS hardware flow control
		22	UART_CT_PMODE0	r/w	Parity bit enable and configuration (00/01=
		23	UART_CT_PMODE1	r/w	no parity; 10=even parity; 11=odd parity)
		24	UART_CT_PRSC0	r/w	Baudrate clock prescaler select bit 0
		25	UART_CT_PRSC1	r/w	Baudrate clock prescaler select bit 1
		26	UART_CT_PRSC2	r/w	Baudrate clock prescaler select bit 2
		27	UART_CT_CTS	r/−	Current state of UART's CTS input signal
		28	UART_CT_EN	r/w	UART enable
		31	UART_CT_TX_BUSY	r/−	Transceiver busy flag
0xFFFFFFA4	UART0_DATA	7:0	UART_DATA_MSB UART_DATA_LSB	r/w	Receive/transmit data (8-bit)
		31:0	*−*	*−/w*	*Simulation data output*
		28	UART_DATA_PERR	r/−	RX parity error (if enabled)
		29	UART_DATA_FERR	r/−	RX data frame error (stop bit not set)
		30	UART_DATA_OVERR	r/−	RX data overrun
		31	UART_DATA_AVAIL	r/−	RX data available when set

Table 13: UART0 register map

3.5.10. Secondary Universal Asynchronous Receiver and Transmitter (UART1)

Overview

Hardware source file(s):	`neorv32_uart.vhd`	
Software driver file(s):	`neorv32_uart.c` `neorv32_uart.h`	
Top entity ports:	`uart1_txd_o`	Serial transmitter output UART1
	`uart1_rxd_i`	Serial receiver input UART1
	`uart1_rts_o`	HW flow control: UART1.RX ready to receive
	`uart1_cts_i`	HW flow control: UART1.TX allowed to send
Configuration generics:	`IO_UART1_EN`	Implement UART1 when `true`
CPU interrupts:	Fast IRQ channel 4 Fast IRQ channel 5	RX done interrupt TX done interrupt [3.3. Processor Interrupts]

Theory of Operation

The secondary UART (UART1) is functional identical to the primary UART (UART0 → 3.5.9. Primary Universal Asynchronous Receiver and Transmitter (UART0)). Obviously, UART1 has different addresses for thw control (`UART1_CTRL`) and the data register (`UART1_DATA`) – see register map below. However, the register bits/flags use the same bit positions and namings. Also, the "RX done" and "TX done" interrupts are mapped to different CPU *fast interrupt channels*.

Simulation Mode

The secondary UART (UART1) provides the same simulation options as the primary UART. However, output data is written to UART1-specific files: `neorv32.uart1.sim_mode.text.out` is used to store plain ASCII text and `neorv32.uart1.sim_mode.data.out` is used to store full 32-bit hexadecimal encoded data words.

Register Map

Address	Name [C]	Bit(s) (Name) [C]		R/W	Function
0xFFFFFFD0	UART1_CT	11:0	UART_CT_BAUDxx	r/w	12-bit BAUD configuration value
		12	*UART_CT_SIM_MODE*	*r/w*	*Enable simulation output mode (see below)*
		20	UART_CT_RTS_EN	r/w	Enable RTS hardware flow control
		21	UART_CT_CTS_EN	r/w	Enable CTS hardware flow control
		22	UART_CT_PMODE0	r/w	Parity bit enable and configuration (00/01= no parity; 10=even parity; 11=odd parity)
		23	UART_CT_PMODE1	r/w	
		24	UART_CT_PRSC0	r/w	Baudrate clock prescaler select bit 0
		25	UART_CT_PRSC1	r/w	Baudrate clock prescaler select bit 1
		26	UART_CT_PRSC2	r/w	Baudrate clock prescaler select bit 2
		27	UART_CT_CTS	r/-	Current state of UART's CTS input signal
		28	UART_CT_EN	r/w	UART enable
		31	UART_CT_TX_BUSY	r/-	Transceiver busy flag
0xFFFFFFD4	UART1_DATA	7:0	UART_DATA_MSB UART_DATA_LSB	r/w	Receive/transmit data (8-bit)
		31:0	*–*	*-/w*	*Simulation data output*
		28	UART_DATA_PERR	r/-	RX parity error (if enabled)
		29	UART_DATA_FERR	r/-	RX data frame error (stop bit not set)
		30	UART_DATA_OVERR	r/-	RX data overrun
		31	UART_DATA_AVAIL	r/-	RX data available when set

Table 14: UART1 register map

3.5.11. Serial Peripheral Interface Controller (SPI)

Overview

Hardware source file(s):	`neorv32_spi.vhd`	
Software driver file(s):	`neorv32_spi.c` `neorv32_spi.h`	
Top entity ports:	`spi_sck_o`	1-bit serial controller clock output
	`spi_sdo_o`	1-bit serial controller data output
	`spi_dsi_i`	1-bit serial controller data input
	`spi_csn_o`	8-bit dedicated chip select port (low-active)
Configuration generics:	`IO_SPI_EN`	Implement SPI when `true`
CPU interrupts:	Fast IRQ channel 6	Transmission done interrupt [3.3. Processor Interrupts]

Theory of Operation

SPI is a synchronous serial transmission protocol. The NEORV32 SPI transceiver allows 8-, 16-, 24- and 32-bit wide transmissions. The unit provides 8 dedicated chip select signals via the top entity's `spi_csn_o` signal.

The SPI unit is enabled via the `SPI_CT_EN` bit. The idle clock polarity is configured via the `SPI_CT_CPHA` bit and can be low (0) or high (1) during idle. Data is shifted in/out with MSB first when the `SPI_CT_DIR` bit is cleared; data is sifted in/out LSB-first when the flag is set. The data quantity to be transferred within a single transmission is defined via the `SPI_CT_SIZEx` bits. The unit supports 8-bit ("00"), 16-bit ("01"), 24-bit ("10") and 32-bit ("11") transfers. Whenever a transfer is completed, an interrupt is triggered.

A transmission is still in progress as long as the `SPI_CT_BUSY` flag is set. T he SPI controller features 8 dedicated chip-select lines. These lines are controlled via the control register's `SPI_CT_CSx` bits. When the CSx bit is set, the according chip select line `spi_csn_o(x)` goes low (low-active chip select lines)

The SPI clock frequency is defined via the 3 `SPI_CT_PRSCx` clock prescaler bits. The following prescalers are available:

SPI_CT_PRSCx	000	001	010	011	100	101	110	111
Resulting prescaler:	2	4	8	64	128	1024	2048	4096

Based on the `SPI_CT_PRSCx` configuration, the actual SPI clock frequency f_{SPI} is determined by:

$$f_{SPI} = \frac{f_{main}[Hz]}{2 \cdot Prescaler}$$

A transmission is started when writing data to the `SPI_DATA` register. The data must be LSB-aligned. So if the SPI transceiver is configured for less than 32-bit transfers data quantity, the transmit data must be placed into the lowest 8/16/24 bit of `SPI_DATA`. Vice versa, the received data is also always LSB-aligned.

Register Map

Address	Name [C]	Bit(s) (Name) [C]		R/W	Function
0xFFFFFFA8	SPI_CT	0	SPI_CT_CS0	r/w	Direct chip select 0, csn(0) is low when set
		1	SPI_CT_CS1	r/w	Direct chip select 1, csn(1) is low when set
		2	SPI_CT_CS2	r/w	Direct chip select 2, csn(2) is low when set
		3	SPI_CT_CS3	r/w	Direct chip select 3, csn(3) is low when set
		4	SPI_CT_CS4	r/w	Direct chip select 4, csn(4) is low when set
		5	SPI_CT_CS5	r/w	Direct chip select 5, csn(5) is low when set
		6	SPI_CT_CS6	r/w	Direct chip select 6, csn(6) is low when set
		7	SPI_CT_CS7	r/w	Direct chip select 7, csn(7) is low when set
		8	SPI_CT_EN	r/w	SPI enable
		9	SPI_CT_CPHA	r/w	Idle clock polarity
		10	SPI_CT_PRSC0	r/w	Clock prescaler select bit 0
		11	SPI_CT_PRSC1	r/w	Clock prescaler select bit 1
		12	SPI_CT_PRSC2	r/w	Clock prescaler select bit 2
		13	SPI_CT_DIR	r/w	Shift direction (0: MSB first, 1: LSB first)
		14	SPI_CT_SIZE0	r/w	Transfer size (00: 8.bit, 01: 16-bit, 10: 24-bit, 11: 32-bit)
		15	SPI_CT_SIZE1	r/w	
		31	SPI_CT_BUSY	r/-	Ongoing transfer when set
0xFFFFFFAC	SPI_DATA	31:0		r/w	Receive/transmit data, LSS-aligned

Table 15: SPI transceiver register map

3.5.12. Two Wire Serial Interface Controller (TWI)

Overview

Hardware source file(s):	neorv32_twi.vhd	
Software driver file(s):	neorv32_twi.c neorv32_twi.h	
Top entity ports:	twi_sda_io	Bi-directional serial data line
	twi_scl_io	Bi-directional serial clock line
Configuration generics:	IO_TWI_EN	Implement TWI when `true`
CPU interrupts:	Fast IRQ channel 7	Transmission done interrupt [3.3. Processor Interrupts]

Theory of Operation

The two wire interface – actually called I²C – is a quite famous interface for connecting several on-board components. Since this interface only needs two signals (the serial data line `twi_sda_io` and the serial clock line `twi_scl_io`) – despite of the number of connected devices – it allows easy interconnections of several peripheral nodes.

The NEORV32 TWI implements a TWI **controller**. It features "**clock stretching**" (if enabled via the control register), so a slow peripheral can halt the transmission by pulling the SCL line low. **Currently no multi-controller support is available. Also, the TWI unit cannot operate in peripheral mode.**

The TWI is enabled via the control register `TWI_CT_EN` bit. The user program can start / terminate a transmission by issuing a START or STOP condition. These conditions are generated by setting the according bit (`TWI_CT_START` or `TWI_CT_STOP`) in the control register.

Data is send by writing a byte to the `TWI_DATA` register. Received data can also be obtained from this register. The TWI controller is busy (transmitting or performing a START or STOP condition) as long as the `TWI_CT_BUSY` bit in the control register is set.

An accessed peripheral has to acknowledge each transferred byte. When the `TWI_CT_ACK` bit is set after a completed transmission, the accessed peripheral has send an acknowledge. If it is cleared after a transmission, the peripheral has send a not-acknowledge (NACK). The NEORV32 TWI controller can also send an ACK (→ controller acknowledge "MACK") after a transmission by pulling SDA low during the ACK time slot. Set the `TWI_CT_MACK` bit to activate this feature. If this bit is cleared, the ACK/NACK of the peripheral is sampled in this time slot (normal mode).

In summary, the following independent TWI operations can be triggered by the application program:

- send START condition (also as REPEATED START condition)
- send STOP condition
- send (at least) one byte while also sampling one byte from the bus

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

The TWI clock frequency is defined via the 3 `TWI_CT_PRSCx` clock prescaler bits. The following prescalers are available:

TWI_CT_PRSCx	000	001	010	011	100	101	110	111
Resulting prescaler:	2	4	8	64	128	1024	2048	4096

Based on the TWI`_CT_PRSCx` configuration, the actual TWI clock frequency f_{SCL} is determined by:

$$f_{SCL} = \frac{f_{main}[Hz]}{4 \cdot Prescaler}$$

Register Map

Address	Name [C]	Bit(s) (Name) [C]		R/W	Function
0xFFFFFFB0	TWI_CT	0	TWI_CT_EN	r/w	TWI enable
		1	TWI_CT_STAT	0/w	Generate START condition
		2	TWI_CT_STOP	0/w	Generate STOP condition
		3	TWI_CT_PRSC0	r/w	Clock prescaler select bit 0
		4	TWI_CT_PRSC1	r/w	Clock prescaler select bit 1
		5	TWI_CT_PRSC2	r/w	Clock prescaler select bit 2
		6	TWI_CT_MACK	r/w	Generate controller ACK for each transmission
		7	TWI_CT_CKSTEN	r/w	Enable/allow clock stretching (by peripherals)
		30	TWI_CT_ACK	r/-	ACK received when set
		31	TWI_CT_BUSY	r/-	Transfer in progress when set
0xFFFFFFB4	TWI_DATA	7:0	TWI_DATA_MSB TWI_DATA_LSB	r/-	Receive/transmit data

Table 16: TWI register map

3.5.13. Pulse Width Modulation Controller (PWM)

Overview

Hardware source file(s):	`neorv32_pwm.vhd`	
Software driver file(s):	`neorv32_pwm.c` `neorv32_pwm.h`	
Top entity ports:	`pwm_o`	4-channel (4 x 1-bit) PWM output
Configuration generics:	`IO_PWM_EN`	Implement PWM controller when `true`
CPU interrupts:	`none`	

Theory of Operation

The PWM controller implements a pulse-width modulation controller with four independent channels and 8-bit resolution per channel. It is based on an 8-bit counter with four programmable threshold comparators that control the actual duty cycle of each channel. The controller can be used to drive a fancy RGB-LED with 24-bit true color, to dim LCD backlights or even for motor control. An external integrator (RC low-pass filter) can be used to smooth the generated "analog" signals.

The PWM controller is activated by setting the `PWM_CT_EN` bit in the module's control register. When this flag is cleared, the unit is reset and all PWM output channels are set to zero. The base clock for the PWM generation is defined via the 3 `PWM_CT_PRSCx` bits. The 8-bit duty cycle for each channel, which represents the channel's "intensity", is defined via the according 8-bit `PWM_DUTY_CHx` byte in the `PWM_DUTY` register.

Based on the duty cycle `PWM_DUTY_CHx` the according analog output voltage (relative to the IO supply voltage) of each channel can be computed by the following formula:

$$Intensity_{xx} = \frac{PWM\_DUTY\_CHx}{2^8}\%$$

The frequency of the generated PWM signals is defined by the PWM operating clock. This clock is derived from the main processor clock and divided by a prescaler via the 3 `PWM_CT_PRSCx` bits in the unit's control register. The following prescalers are available:

PWM_CT_PRSCx	000	001	010	011	100	101	110	111
Resulting prescaler:	2	4	8	64	128	1024	2048	4096

The resulting PWM frequency is defined by:

$$f_{PWM} = \frac{f_{main}}{2^8 \cdot Prescaler}$$

> A more sophisticated frequency generation is provided by by the *numerically-controlled oscillator* module (→ 3.5.16. Numerically-Controller Oscillator (NCO)).

Register Map

Address	Name [C]	Bit(s) (Name) [C]		R/W	Function
0xFFFFFFB8	PWM_CT	0	PWM_CT_EN	r/w	PWM controller enable
		1	PWM_CT_PRSC0	r/w	Clock prescaler select bit 0
		2	PWM_CT_PRSC1	r/w	Clock prescaler select bit 1
		3	PWM_CT_PRSC2	r/w	Clock prescaler select bit 2
0xFFFFFFBC	PWM_DUTY	7:0	PWM_DUTY_CH0_MSB PWM_DUTY_CH1_LSB	r/w	8-bit duty cycle for channel 0
		15:8	PWM_DUTY_CH1_MSB PWM_DUTY_CH1_LSB	r/w	8-bit duty cycle for channel 1
		23:16	PWM_DUTY_CH2_MSB PWM_DUTY_CH2_LSB	r/w	8-bit duty cycle for channel 2
		31:24	PWM_DUTY_CH3_MSB PWM_DUTY_CH3_LSB	r/w	8-bit duty cycle for channel 3

Table 17: PWM controller register map

3.5.14. True Random Number Generator (TRNG)

Overview

Hardware source file(s):	`neorv32_trng.vhd`	
Software driver file(s):	`neorv32_trng.c` `neorv32_trng.h`	
Top entity ports:	`none`	
Configuration generics:	`IO_TRNG_EN`	Implement TRNG when `true`
CPU interrupts:	`none`	

Theory of Operation

The NEORV32 true random number generator provides *physical true random numbers* for your application. Instead of using a pseudo RNG like a LFSR, the TRNG of the processor uses a simple, straight-forward ring oscillator as physical entropy source. Hence, voltage and thermal fluctuations are used to provide true physical random data.

The TRNG features a platform independent architecture without FPGA-specific primitives, macros or attributes.

Architecture

The NEORV32 TRNG is based on simple ring oscillators, which are implemented as an inverter chain with an odd number of inverters. A **latch** is used to decouple each individual inverter. Basically, this architecture is some king of *asynchronous LFSR*.

The output of several ring oscillators are synchronized using two registers and are XORed together. The resulting output is de-biased using a von Neumann randomness extractor. This de-biased output is further processed by a simple 8-bit Fibonacci LFSR to improve whitening. After at least 8 clock cycles the state of the LFSR is sampled and provided as final data output.

To prevent the synthesis tool from doing logic optimization and thus, removing all but one inverter, the TRNG uses simple latches to decouple an inverter and its actual output. The latches are reset when the TRNG is disabled and are enabled one by one by a "real" shift register when the TRNG is activated. This construct can be synthesized for any FPGA platform. Thus, the NEORV32 TRNG provides a platform independent architecture.

TRNG Configuration

The TRNG uses several ring-oscillators, where the next oscillator provides a slightly longer chain (more inverters) than the one before. This increment is constant for all implemented oscillators. This setup can be customized by modifying the *"Advanced Configuration"* constants in the TRNG's VHDL file:

The `num_roscs_c` constant defines the total number of ring oscillators in the system. `num_inv_start_c` defines the number of inverters used by the first ring oscillators (has to be an odd number). Each additional ring oscillator provides `num_inv_inc_c` more inverters that the one before (has to be an even number).

The LFSR-based post-processing can be deactivated using the `lfsr_en_c` constant. The polynomial tap mask of the LFSR can be customized using `lfsr_taps_c`.

Using the TRNG

The TRNG features a single register for status and data access. When the `TRNG_CT_EN` control register bit is set, the TRNG is enabled and starts operation. As soon as the `TRNG_CT_VALID` bit is set, the currently sampled 8-bit random data byte can be obtained from the lowest 8 bits of the `TRNG_CT` register (`TRNG_CT_DATA_MSB` downto `TRNG_CT_DATA_LSB`). The `TRNG_CT_VALID` bit is automatically cleared when reading the control register.

Note, that the TRNG needs at least 8 clock cycles to generate a new random byte. During this sampling time the current output random data is kept stable in the output register until a valid sampling of the new byte has completed.

Randomness "Quality"

I have not verified the quality of the generated random numbers (for example using NIST test suites). The quality is highly effected by the actual configuration of the TRNG and the resulting FPGA mapping/routing. However, generating larger histograms of the generated random number shows an equal distribution (binary average of the random numbers = 127). A simple evaluation test/demo program can be found in `sw/example/demo_trng`.

Register Map

Address	Name [C]	Bit(s)	(Name) [C]	R/W	Function
0xFFFFFF88	TRNG_CT	7:0	TRNG_CT_DATA_MSB TRNG_CT_DATA_LSB	r/-	8-bit random data output
		30	TRNG_CT_EN	r/w	TRNG enable
		31	TRNG_CT_VALID	r/-	Random output data is valid when set

Table 18: TRNG register map

3.5.15. Custom Functions Subsystem (CFS)

Overview

Hardware source file(s):	`neorv32_cfs.vhd`	
Software driver file(s):	`neorv32_cfs.c` `neorv32_cfs.h`	Stubs only, have to be implemented by the user
Top entity ports:	`cfs_in_i` `cfs_out_o`	Custom I/O "conduit" signals
Configuration generics:	`IO_CFS_EN`	Implement CFS when `true`
	`IO_CFS_CONFIG`	Custom 32-bit "conduit" generic
	`IO_CFS_IN_SIZE`	Defines size of `cfs_in_i` input signal
	`IO_CFS_OUT_SIZE`	Defines size of `cfs_out_o` output signal
CPU interrupts:	`Fast IRQ channel 1`	CFS interrupt [3.3. Processor Interrupts]

Theory of Operation

The custom functions subsystem can be used to implement application-specific user-defined co-processors (like encryption or arithmetic accelerators) or peripheral/communication interfaces. In contrast to connecting custom hardware accelerators via the external memory interface, the CFS provide a convenient and low-latency extension/customization option.

The CFS provides up to 32 32-bit memory-mapped registers (see register map table below). The actual functionality of these register has to be defined by the hardware designer.

> ℹ️ Take a look at the 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 require/provide 32-bit data of type `uint32_t`.

```
// C-code CFS usage example

CFS_REG_0 = (uint32_t)some_data_array(i);  // write to CFS register 0
uint32_t temp = CFS_REG_20;                // read from CFS register 20
```

CFS Interrupt

The CFS provides a single one-shot interrupt request signal mapped to the CPU's fast interrupt channel 1. See 3.3. Processor Interrupts for more information.

CFS Configuration Generic

By default, the CFS provides a single 32-bit **std_(u)logic_vector** configuration generic `IO_CFS_CONFIG` that is available in the processor's top entity. This generic can be used to pass custom configuration options from the top entity down to the CFS entiy.

CFS Custom IOs

By default, the CFS also provides two unidirectional input and output *conduits* `cfs_in_i` and `cfs_out_o`. These signals are propagated to the processor's top entity. The actual use of these signals has to be defined by the hardware designer.

The size of the input signal conduit `cfs_in_i` is defined via the (top's) `IO_CFS_IN_SIZE` configuration generic (default = 32-bit). The size of the output signal conduit `cfs_out_o` is defined via the (top's) `IO_CFS_OUT_SIZE` configuration generic (default = 32-bit).

If the custom function subsystem is not implemented (`IO_CFS_EN` = `false`) the `cfs_out_o` signal is tied to all-zero.

Register Map

Address	Name [C]	Bit(s)	R/W	Function
0xFFFFFF00	CFS_REG_0	31:0	(r)/(w)	CFS custom interface register 0
0xFFFFFF04	CFS_REG_1	31:0	(r)/(w)	CFS custom interface register 1
0xFFFFFF08	CFS_REG_2	31:0	(r)/(w)	CFS custom interface register 2
0xFFFFFF0C	CFS_REG_3	31:0	(r)/(w)	CFS custom interface register 3
0xFFFFFF10	CFS_REG_4	31:0	(r)/(w)	CFS custom interface register 4
0xFFFFFF14	CFS_REG_5	31:0	(r)/(w)	CFS custom interface register 5
0xFFFFFF18	CFS_REG_6	31:0	(r)/(w)	CFS custom interface register 6
0xFFFFFF1C	CFS_REG_7	31:0	(r)/(w)	CFS custom interface register 7
0xFFFFFF20	CFS_REG_8	31:0	(r)/(w)	CFS custom interface register 8
0xFFFFFF24	CFS_REG_9	31:0	(r)/(w)	CFS custom interface register 9
0xFFFFFF28	CFS_REG_10	31:0	(r)/(w)	CFS custom interface register 10
0xFFFFFF2C	CFS_REG_11	31:0	(r)/(w)	CFS custom interface register 11
0xFFFFFF30	CFS_REG_12	31:0	(r)/(w)	CFS custom interface register 12
0xFFFFFF34	CFS_REG_13	31:0	(r)/(w)	CFS custom interface register 13
0xFFFFFF38	CFS_REG_14	31:0	(r)/(w)	CFS custom interface register 14
0xFFFFFF3C	CFS_REG_15	31:0	(r)/(w)	CFS custom interface register 15
0xFFFFFF40	CFS_REG_16	31:0	(r)/(w)	CFS custom interface register 16
0xFFFFFF44	CFS_REG_17	31:0	(r)/(w)	CFS custom interface register 17
0xFFFFFF48	CFS_REG_18	31:0	(r)/(w)	CFS custom interface register 18
0xFFFFFF4C	CFS_REG_19	31:0	(r)/(w)	CFS custom interface register 19
0xFFFFFF50	CFS_REG_20	31:0	(r)/(w)	CFS custom interface register 20
0xFFFFFF54	CFS_REG_21	31:0	(r)/(w)	CFS custom interface register 21
0xFFFFFF58	CFS_REG_22	31:0	(r)/(w)	CFS custom interface register 22
0xFFFFFF5C	CFS_REG_23	31:0	(r)/(w)	CFS custom interface register 23
0xFFFFFF60	CFS_REG_24	31:0	(r)/(w)	CFS custom interface register 24
0xFFFFFF64	CFS_REG_25	31:0	(r)/(w)	CFS custom interface register 25
0xFFFFFF68	CFS_REG_26	31:0	(r)/(w)	CFS custom interface register 26
0xFFFFFF6C	CFS_REG_27	31:0	(r)/(w)	CFS custom interface register 27
0xFFFFFF70	CFS_REG_28	31:0	(r)/(w)	CFS custom interface register 28
0xFFFFFF74	CFS_REG_29	31:0	(r)/(w)	CFS custom interface register 29
0xFFFFFF78	CFS_REG_30	31:0	(r)/(w)	CFS custom interface register 30
0xFFFFFF7C	CFS_REG_31	31:0	(r)/(w)	CFS custom interface register 31

Table 19: Custom Functions Subsystem register map

3.5.16. Numerically-Controller Oscillator (NCO)

Overview

Hardware source file(s):	`neorv32_nco.vhd`	
Software driver file(s):	`neorv32_nco.c` `neorv32_nco.h`	
Top entity ports:	`nco_o`	NCO output channels (3×1-bit)
Configuration generics:	`IO_NCO_EN`	Implement NCO when `true`
CPU interrupts:	`none`	

Theory of Operation

The numerically-controller oscillator (NCO) provides a precise arbitrary linear frequency generator with three independent channels. Based on a *direct digital synthesis* core, the NCO features a 20-bit wide *accumulator* that is incremented with a programmable *tuning word*. Whenever the accumulator overflows, a flip flop is toggled that provides the actual frequency output. The accumulator increment is driven by one of eight configurable clock sources, which are derived from the processor's main clock.

The NCO features four accessible registers: the control register `NCO_CT` and a `NCO_TUNE_CHi` register for the tuning word of each channel *i*. The NCO is <u>globally</u> enabled by setting the `NCO_CT_EN` bit in the control register. If this bit is cleared, the accumulators of all channels are reset. The clock source for each channel *i* is selected via the three bits `NCO_CT_CHi_PRSCx` prescaler. The resulting clock is generated from the main processor clock (f_{main}) divided y the selected prescaler.

NCO_CT_CHi_PRSCx	000	001	010	011	100	101	110	111
Resulting clock prescaler:	2	4	8	64	128	1024	2048	4096

The resulting output frequency of each channel *i* is defined by the following equation.

$$f_{NCO}(i) = \frac{f_{main}}{\text{clock\_prescaler}(i)} \cdot \frac{\text{tuning\_word}(i)}{2 \cdot 2^{20+1}} \text{ for channel i} = 0,1,2$$

The maximum NCO frequency f_{NCO_max} is configured using the minimal clock prescaler and a maximum all-one tuning word:

$$f_{NCO\_max} = \frac{f_{main}}{2} \cdot \frac{2^{20}-1}{2 \cdot 2^{20+1}}$$

The minimum frequency is always 0 Hz when the tuning word is zero. The frequency resolution f_{NCO_res} is defined using the maximum clock prescaler and a minimal non-zero tuning word (= 1):

$$f_{NCO\_res} = \frac{f_{main}}{4096} \cdot \frac{1}{2 \cdot 2^{20+1}}$$

Assuming a processor frequency of f_{main} = 100 MHz the maximum NCO output frequency is f_{NCO_max} = 12.499 MHz with an NCO frequency resolution of f_{NCO_res} = 0.00582 Hz.

Advanced Configuration

The idle polarity of each channel is configured via the `NCO_CT_CHi_IDLE_POL` flag and can be either 0 (idle low) or 1 (idle high), which basically allows to invert the NCO output. If the NCO is globally disabled by clearing the `NCO_CT_EN` flag, all `nco_o` outputs are set to `NCO_CT_CHi_IDLE_POL`.

The current state of each NCO channel output can be read by software via the `NCO_CT_CHi_OUTPUT` bit. The NCO frequency output is normally available via the top `nco_o` output signal. The according channel output can be permanently set to zero by clearing the according `NCO_CT_CHi_OE` bit.

Each NCO channel can operate either in **standard mode** or in **pulse mode**. The mode is configured via the according channel's `NCO_CT_CHi_MODE` control register bit.

Standard Operation Mode

If this `NCO_CT_CHi_MODE` bit of channel *i* is cleared, the channel operates in standard mode providing a frequency with *exactly 50% duty cycle* ($T_{high} = T_{Low}$).

Pulse Operation Mode

If the `NCO_CT_CHi_MODE` bit of channel *i* is set, the channel operates in pulse mode. In this mode, the duty cycle can be modified to generate *active* pulses with variable length. Note that the "*active*" pulse is defined by the *inverted* `NCO_CT_CHi_IDLE_POL` bit.

Eight different pulse lengths are available. The *active* pulse length is defined as number of NCO clock cycles, where the NCO clock is defined via the clock prescaler bits `NCO_CT_Chi_PRSCx`. The pulse length of channel *i* is programmed by the three-bit `NCO_CT_CHi_PULSEx` configuration:

NCO_CT_CHi_PULSEx	000	001	010	011	100	101	110	111
Pulse length (in NCO clock cycles)	2	4	8	16	32	64	128	256

If `NCO_CT_CHi_IDLE_POL` is cleared, T_{high} is defined by the `NCO_CT_CHi_PULSEx` configuration and $T_{low} = T - T_{high}$. If `NCO_CT_CHi_IDLE_POL` is set, T_{low} is defined by the `NCO_CT_CHi_PULSEx` configuration and $T_{high} = T - T_{low}$.

The actual output frequency of the channel (defined via the clock prescaler and the tuning word) is not affected by the pulse configuration.

For simple PWM applications, that do not require a precise frequency but a more flexible duty cycle configuration, see the PWM module (→ 3.5.13. Pulse Width Modulation Controller (PWM)).

Register Map

Address	Name [C]	Bit(s) (Name) [C]	R/W	Function
0xFFFFFFC0	NCO_CT	0 NCO_CT_EN	r/-	8-bit random data output
		Channel 0		
		1 NCO_CT_CH0_MODE	r/w	Output mode (0=fixed 50% duty cycle; 1=pulse mode)
		2 NCO_CT_CH0_IDLE_POL	r/w	Output idle polarity (0=low, 1=high)
		3 NCO_CT_CH0_OE	r/w	Enable output to nco_o(0)
		4 NCO_CT_CH0_OUTPUT	r/-	Current channel output state
		5 NCO_CT_CH0_PRSC0	r/w	Clock prescaler select bit 0
		6 NCO_CT_CH0_PRSC1	r/w	Clock prescaler select bit 1
		7 NCO_CT_CH0_PRSC2	r/w	Clock prescaler select bit 2
		8 NCO_CT_CH0_PULSE0	r/w	Pulse-mode: Pulse length select bit 0
		9 NCO_CT_CH0_PULSE1	r/w	Pulse-mode: Pulse length select bit 1
		10 NCO_CT_CH0_PULSE2	r/w	Pulse-mode: Pulse length select bit 2
		Channel 1		
		11 NCO_CT_CH1_MODE	r/w	Output mode (0=fixed 50% duty cycle; 1=pulse mode)
		12 NCO_CT_CH1_IDLE_POL	r/w	Output idle polarity (0=low, 1=high)
		13 NCO_CT_CH1_OE	r/w	Enable output to nco_o(1)
		14 NCO_CT_CH1_OUTPUT	r/-	Current channel output state
		15 NCO_CT_CH1_PRSC0	r/w	Clock prescaler select bit 0
		16 NCO_CT_CH1_PRSC1	r/w	Clock prescaler select bit 1
		17 NCO_CT_CH1_PRSC2	r/w	Clock prescaler select bit 2
		18 NCO_CT_CH1_PULSE0	r/w	Pulse-mode: Pulse length select bit 0
		19 NCO_CT_CH1_PULSE1	r/w	Pulse-mode: Pulse length select bit 1
		20 NCO_CT_CH1_PULSE2	r/w	Pulse-mode: Pulse length select bit 2
		Channel 2		
		21 NCO_CT_CH2_MODE	r/w	Output mode (0=fixed 50% duty cycle; 1=pulse mode)
		22 NCO_CT_CH2_IDLE_POL	r/w	Output idle polarity (0=low, 1=high)
		23 NCO_CT_CH2_OE	r/w	Enable output to nco_o(2)
		24 NCO_CT_CH2_OUTPUT	r/-	Current channel output state
		25 NCO_CT_CH2_PRSC0	r/w	Clock prescaler select bit 0
		26 NCO_CT_CH2_PRSC0	r/w	Clock prescaler select bit 1
		27 NCO_CT_CH2_PRSC0	r/w	Clock prescaler select bit 2
		28 NCO_CT_CH2_PULSE0	r/w	Pulse-mode: Pulse length select bit 0
		29 NCO_CT_CH2_PULSE0	r/w	Pulse-mode: Pulse length select bit 1

Address	Name [C]	Bit(s) (Name) [C]	R/W	Function
		30 NCO_CT_CH2_PULSE0	r/w	Pulse-mode: Pulse length select bit 2
0xFFFFFFC4	NCO_TUNE_CH0	19:0	r/w	Tuning word for channel 0
0xFFFFFFC8	NCO_TUNE_CH1	19:0	r/w	Tuning word for channel 1
0xFFFFFFCC	NCO_TUNE_CH2	19:0	r/w	Tuning word for channel 2

Table 20: NCO register map

3.5.17. Smart LED Interface (NEOLED) – NeoPixel © Compatible

Overview

Hardware source file(s):	`neorv32_neoled.vhd`	
Software driver file(s):	`neorv32_neoled.c` `neorv32_neoled.h`	
Top entity ports:	`neoled_o`	Asynchronous serial LED data (single-wire)
Configuration generics:	`IO_NEOLED_EN`	Implement NEOLED module when `true`
CPU interrupts:	`Fast IRQ channel 9`	NEOLED interrupt [3.3. Processor Interrupts]

Introduction

The NEOLED module provides a dedicated interface for "smart color LEDs" like the *WS2812* or *WS2811*. These LEDs provide a single interface wire that uses an asynchronous serial protocol for transmitting color data. Basically, data is transferred via LED-internal shift registers, which allows to cascade an unlimited number of smart LEDs. The protocol provides a RESET command to strobe the transmitted data into the LED PWM driver registers after data has shifted throughout all LEDs in a chain.

✅ The NEOLED interface is compatible to the *Adafruit Industries* **NeoPixel** products, which feature WS2812 (or older WS2811) smart LEDs. → NeoPixel Überguide online @ Adafruit Industries

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

Theory of Operation – Protocol

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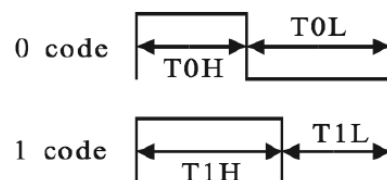

Figure 7: WS2812 bit-level protocol – taken from the "Adafruit NeoPixel Überguide"

`T_TOTAL (T_carrier)`	1.25μs +/- 300ns	Period for a single bit
`T0H`	0.4μs +/- 150ns	High-time for sending a '1'
`T0L`	0.8μs +/- 150ns	Low-time for sending a '1'
`T1H`	0.85μs +/- 150ns	High-time for sending a '0'
`T1L`	0.45μs +/- 150 ns	Low-time for sending a '0'
`RESET`	Above 50μs	Low-time for sending a RESET command

Theory of Operation – NEOLED

The NEOLED modules provides two accessible interface register: the control register `NEOLED_CT` and the TX data register `NEOLED_DATA`. The NEOLED module is globally enabled the control register's `NEOLED_CT_EN` bit. Clearing this bit will terminate any current operation and will also reset the module and set the `neoled_o` output to zero. The precise timing (implementing the WS281* protocol) and transmission mode are fully programmable to provide maximum flexibility.

Timing Configuration

The basic carrier frequency (*800kHz for the WS2812 LED*s) is configured via a 3-bit main clock prescaler that scales the main processor clock (`NEOLED_CT_PRSCx`, see table below) and a 5-bit cycle multiplier `NEOLED_CT_T_TOT_x`.

NEOLED_CT_PRSCx	000	001	010	011	100	101	110	111
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_CT_T_ONE_H_x` and `NEOLED_CT_T_ZERO_H_x` value. These configurable timing constants allow to adapt the interface for a wide variety of smart LED protocol (→ WS2812 vs. WS2811).

Timing Configuration – Example (WS2812)

Generating the base clock for the NEOLED TX engine:

Processor clock f_{main}	100 MHz	
`NEOLED_CT_PRSCx`	`0b001`	→ f_{main} / 4

$$f_{TX} = \frac{f_{main}}{\text{clock prescaler}} = \frac{100\,MHz}{4} = 25\,MHz\,; \quad T_{TX} = \frac{1}{f_{carrier}} = 40\,ns$$

Generating carrier frequency and high-times (duty cycle) for sending '0' or '1' bits:

`NEOLED_CT_T_TOT_x`	`0b11110`	30
`NEOLED_CT_T_ZERO_H_x`	`0b01010`	10
`NEOLED_CT_T_ONE_H_x`	`0b10100`	20

$$T_{carrier} = T_{TX} \cdot \text{NEOLED\_CT\_T\_TOT} = 40\,ns \cdot 30 = 1.2\,\mu s$$

$$T_{0H} = T_{TX} \cdot \text{NEOLED\_CT\_T\_ZERO} = 40\,ns \cdot 10 = 0.4\,\mu s$$

$$T_{1H} = T_{TX} \cdot \text{NEOLED\_CT\_ONE\_H} = 40\,ns \cdot 20 = 0.8\,\mu s$$

> 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 configuration.

RGB / RGBW Configuration

NeoPixel are available in two "color" version: LEDs with three chips providing **RGB** color and LEDs with four chips providing RGB color plus a dedicated white LED chip → **RGBW**. Since the intensity of every LED chip is defined via an 8-bit value the **RGB** LEDs require a frame of **24-bit** per module and the **RGBW** LEDs require a frame of **32-bit** per mdule.

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

TX Data FIFO

The interface features a TX data buffer (a FIFO) to allow more CPU-independent operation. The buffer depth is configured via the `tx_buffer_entries_c` constant (default = 4 entries) in the module's VHDL source file `rtl/core/neorv32_neoled.vhd`. The current configuration can be read via the `NEOLED_CT_BUFS_x` control register bits, which result `log2(tx_buffer_entries_c)`.

When writing data to the `NEOLED_DATA` register the data is automatically written to the TX buffer. Whenever data is available in the buffer the serial transmission engine will take it and transmit it to the LEDs.

The data transfer size (**MODE**) can be modified at every time since this control register bit is also buffered in the FIFO. This allows to arbitrarily mix RGB and RGBW LEDs in the chain.

> ⚠️ Please note that the timing configuration (`NEOLED_CT_PRSCx`, `NEOLED_CT_T_TOT_x`. `NEOLED_CT_T_ONE_H_x` and `NEOLED_CT_T_ZERO_H_x`) are **NOT** stored to the buffer. Changing these value while the buffer is not empty or the TX engine is still sending will cause data corruption.

Status Configuration

The NEOLED modules features two read-only status bits in the control register: `NEOLED_CT_BUSY` and `NEOLED_CT_TX_STATUS`.

If the `NEOLED_CT_TX_STATUS` is set the serial TX engine is still busy sending serial data to the LED stripes. If the flag is cleared, the TX engine is idle and the serial data output `neoled_o` is **LOW**.

The `NEOLED_CT_BUSY` flag provides a programmable option to check for the TX buffer state. The control register's `NEOLED_CT_BSCON` bit is used to configure the "meaning" of the `NEOLED_CT_BUSY` flag. The condition for sending an interrupt request (IRQ) to the CPU is also configured via the `NEOLED_CT_BSCON` bit.

NEOLED_CT_BSCON	NEOLED_CT_BUSY	Sending an IRQ
0	The busy flag will clear if there **is at least one free entry** in the TX buffer	The IRQ will fire if **at least one entry gets free** in the TX buffer
1	The busy flag will clear if the **whole TX buffer is empty**	The IRQ will fire if the **whole TX buffer gets empty**

When `NEOLED_CT_BSCON` is set the CPU can write up to `tx_buffer_entries_c` new data words to `NEOLED_DATA` without checking the busy flag `NEOLED_CT_BUSY`. This highly relaxes time constraints for sending a continuous data stream to the LEDs (as an idle time beyond 50μs will trigger the LED's a RESET command).

Register Map

Address	Name [C]	Bit(s) (Name) [C]		R/W	Function
0xFFFFFFD8	NEOLED_CT	0	NEOLED_CT_EN	r/w	NEOLED enable
		1	NEOLED_CT_MODE	r/w	Data transfer mode; 0 = 24-bit, 1= 32-bit
		2	NEOLED_CT_BSCON	r/w	Busy flag / IRQ trigger configuration (0: clear busy flag and send IRQ when at least one free entry available in buffer; 1: clear busy flag and send IRQ when whole buffer is empty)
		3 ... 5	NEOLED_CT_PRSC2 ... NEOLED_CT_PRSC0	r/w	Clock prescaler select bit 2:0
		9 ... 6	NEOLED_CT_BUFS_3 ... NEOLED_CT_BUFS_0	r/w	Log2(`tx_buffer_entries_c`)
		14 ... 10	NEOLED_CT_T_TOT_4 ... NEOLED_CT_T_TOT_0	r/w	Pulse clock ticks per total single-bit period (`T_total`)
		19 ... 15	NEOLED_CT_T_ZERO_H_4 ... NEOLED_CT_T_ZERO_H_0	r/w	Pulse clock ticks per high-time for sending a zero-bit (`T_high_zero`)
		24 ... 20	NEOLED_CT_T_ONE_H_4 ... NEOLED_CT_T_ONE_H_0	r/w	Pulse clock ticks per high-time for sending a one-bit (`T_high_one`)
		30	NEOLED_CT_TX_STATUS	r/-	Serial TX engine still busy when set
		31	NEOLED_CT_BUSY	r/-	Buffer busy flag; bscon = 0: At least one free buffer entry available when busy flag clears; bscon = 1: Whole buffer is empty when busy flag clears
0xFFFFFFDC	NEOLED_DATA	31:0 / 23:0		-/w	TX data (32-bit or 24-bit)

Table 21: NEOLED register map

3.5.18. System Configuration Information Memory (SYSINFO)

Overview

Hardware source file(s):	`neorv32_sysinfo.vhd`	
Software driver file(s):	`(neorv32.h)`	(Register and bit definitions only)
Top entity ports:	`none`	
Configuration generics:	`*`	Most of the top's configuration generics
CPU interrupts:	`none`	

Theory of Operation

The SYSINFO allows the application software to determine the setting of most of the processor's top entity generics. All registers of this unit are read-only.

This device is always implemented – regardless of the actual hardware configuration. The bootloader as well as the NEORV32 software runtime environment require information from this device (like memory layout and default clock speed) for correct operation.

Register Map

Address	Name [C]	R/W	Function
0xFFFFFFE0	SYSINFO_CLK	r/–	Clock speed in Hz (via top's `CLOCK_FREQUENCY` generic)
0xFFFFFFE4	SYSINFO_USER_CODE	r/–	Custom user code, assigned via top's `USER_CODE` generic
0xFFFFFFE8	SYSINFO_FEATURES	r/–	Implemented hardware (see next table)
0xFFFFFFEC	SYSINFO_CACHE	r/–	Cache configuration information (see next table)
0xFFFFFFF0	SYSINFO_ISPACE_BASE	r/–	Instruction address space base (defined via `ispace_base_c` constant in the `neorv32_package.vhd` file)
0xFFFFFFF4	SYSINFO_IMEM_SIZE	r/–	Internal IMEM size in bytes (defined via top's `MEM_INT_IMEM_SIZE` generic)
0xFFFFFFF8	SYSINFO_DSPACE_BASE	r/–	Data address space base (defined via `sdspace_base_c` constant in the `neorv32_package.vhd` file)
0xFFFFFFFC	SYSINFO_DMEM_SIZE	r/–	Internal DMEM size in bytes (defined via top's `MEM_INT_DMEM_SIZE` generic)

Table 22: SYSINFO register map

SYSINFO_FEATURES

Bit#	Name [C]	Function
27	SYSINFO_FEATURES_IO_NEOLED	Set when the NEOLED is implemented (via top's `IO_NEOLED_EN` generic)
26	SYSINFO_FEATURES_IO_UART1	Set when the secondary UART1 is implemented (via top's `IO_UART1_EN` generic)
25	SYSINFO_FEATURES_IO_NCO	Set when the NCO is implemented (via top's `IO_NCO_EN` generic)
24	SYSINFO_FEATURES_IO_TRNG	Set when the TRNG is implemented (via top's `IO_TRNG_EN` generic)
23	SYSINFO_FEATURES_IO_CFS	Set when the custom functions subsystem is implemented (via top's `IO_CFS_EN` generic)
22	SYSINFO_FEATURES_IO_WDT	Set when the WDT is implemented (via top's `IO_WDT_EN` generic)
21	SYSINFO_FEATURES_IO_PWM	Set when the PWM is implemented (via top's `IO_PWM_EN` generic)
20	SYSINFO_FEATURES_IO_TWI	Set when the TWI is implemented (via top's `IO_TWI_EN` generic)
19	SYSINFO_FEATURES_IO_SPI	Set when the SPI is implemented (via top's `IO_SPI_EN` generic)
18	SYSINFO_FEATURES_IO_UART0	Set when the primary UART0 is implemented (via top's `IO_UART0_EN` generic)
17	SYSINFO_FEATURES_IO_MTIME	Set when the MTIME is implemented (via top's `IO_MTIME_EN` generic)
16	SYSINFO_FEATURES_IO_GPIO	Set when the GPIO is implemented (via top's `IO_GPIO_EN` generic)
5	SYSINFO_FEATURES_MEM_EXT_ENDIAN	Set when external bus interface uses BIG-endian byte-order (via package's `xbus_big_endian_c` constant)
4	SYSINFO_FEATURES_MEM_INT_DMEM	Set when the processor-internal IMEM is implemented (via top's `MEM_INT_IMEM_EN` generic)
3	SYSINFO_FEATURES_MEM_INT_IMEM_ROM	Set when the processor-internal IMEM is read-only (via top's `MEM_INT_IMEM_ROM` generic)
2	SYSINFO_FEATURES_MEM_INT_IMEM	Set when the processor-internal DMEM implemented (via top's `MEM_INT_DMEM_EN` generic)
1	SYSINFO_FEATURES_MEM_EXT	Set when the external Wishbone bus interface is implemented (via top's `MEM_EXT_EN` generic)
0	SYSINFO_FEATURES_BOOTLOADER	Set when the processor-internal bootloader is implemented (via top's `BOOTLOADER_EN` generic)

SYSINFO_CACHE

Bits#	Name [C]	Function
15 ... 12	SYSINFO_CACHE_IC_REPLACEMENT_3 ... SYSINFO_CACHE_IC_REPLACEMENT_0	Instruction cache replacement policy (0000 = none, direct-mapped; 0001 = LRU – least recently used)
11 ... 8	SYSINFO_CACHE_IC_ASSOCIATIVITY_3 ... SYSINFO_CACHE_IC_ASSOCIATIVITY_0	**Instruction** cache associativity = log2(top's `ICACHE_ASSOCIATIVITY` generic)
7 ... 4	SYSINFO_CACHE_IC_NUM_BLOCKS_3 ... SYSINFO_CACHE_IC_NUM_BLOCKS_0	Log2 of **instruction** cache's number of blocks = log2(top's `ICACHE_NUM_BLOCKS` generic)
3 ... 0	SYSINFO_CACHE_IC_BLOCK_SIZE_3 ... SYSINFO_CACHE_IC_BLOCK_SIZE_0	Log2 of **instruction** cache's block size = log2(top's `ICACHE_BLOCK_SIZE` generic)

4. Software Architecture

To make actual use of the **processor**, the NEORV32 project comes with a complete software ecosystem. This ecosystem consists of the following elementary parts.

Application/bootloader start-up code	`sw/common/crt0.S`
Application/bootloader linker script	`sw/common/neorv32.ld`
Core hardware driver libraries	`sw/lib/include/` `sw/lib/source/`
Makefiles	E.g. `sw/example/blink_led/makefile`
Auxiliary tool for generating NEORV32 executables	`sw/image_gen/`
Default bootloader	`sw/bootloader/bootloader.c`

The software ecosystem is based on the RISC-V port of the GCC GNU Compiler Collection.

Last but not least, the NEORV32 ecosystem provides some example programs for testing the hardware, for illustrating the usage of peripherals and for general getting in touch with the project.

4.1. Toolchain

The toolchain for this project is based on the free RISC-V GCC-port. You can find the compiler sources and build instructions on the official RISC-V GNU toolchain GitHub page: https://github.com/riscv/riscv-gnu-toolchain.

The NEORV32uses a 32-bit base integer architecture (`rv32i`) and a 32-bit integer and soft-float ABI (`ilp32`), so make sure you build an according toolchain.

Alternatively, you can download a prebuilt `rv32i/e` toolchain for 64-bit x86 Linux from: github.com/stnolting/riscv-gcc-prebuilt

The default toolchain used by the project's makefiles is (can be changed in the makefiles):
riscv32-unknown-elf

> *i* More information regarding the toolchain (building from scratch or downloading the prebuilt ones) can be found in chapter 5.1. Toolchain Setup.

4.2. Core Software Libraries

The NEORV32 project provides a set of C libraries that allows an easy usage of the core's peripheral and CPU features. Just include the main NEORV32 library file in your application's source file(s):

```
#include <neorv32.h>
```

Together with the makefile, this will automatically include all the processor's header files located in `sw/lib/include` into your application. The actual source files of the core libraries are located in `sw/lib/source` and are automatically included into the source list of your software project. The following files are currently part of the NEORV32 core library:

C source file	C header file	Function
-	neorv32.h	Main NEORV32 definitions and library file.
neorv32_cfs.c	neorv32_cfs.h	HW driver (stubs)[14] functions for the cust. functions subsystem
neorv32_cpu.c	neorv32_cpu.h	HW driver functions for the NEORV32 CPU.
neorv32_gpio.c	neorv32_gpio.h	HW driver functions for the GPIO.
-	neorv32_intrinsics.h	Macros for custom intrinsics/instructions
neorv32_mtime.c	neorv32_mtime.h	HW driver functions for the MTIME.
neorv32_nco.c	neorv32_nco.h	HW driver functions for the NCO.
neorv32_neoled.c	neorv32_neoled.h	HW driver functions for the NEOLED.
neorv32_pwm.c	neorv32_pwm.h	HW driver functions for the PWM.
neorv32_rte.c	neorv32_rte.h	NEORV32 runtime environment helper functions.
neorv32_spi.c	neorv32_spi.h	HW driver functions for the SPI.
neorv32_trng.c	neorv32_trng.h	HW driver functions for the TRNG.
neorv32_twi.c	neorv32_twi.h	HW driver functions for the TWI.
neorv32_uart.c	neorv32_uart.h	HW driver functions for the UART0 and UART1.
neorv32_wdt.c	neorv32_wdt.h	HW driver functions for the WDT.

Documentation

All core library SW functions are highly documented using doxygen. To generate the HTML-based documentation, navigate to the project's `docs` folder and run:

```
neorv32/docs$ doxygen Doxyfile
```

This will generate/update the `docs/doxygen_build` folder. To view the documentation, open the `docs/doxygen_build/html/index.html` file with your browser of choice. Click on the "files" tab to see a list of all documented files.

> The SW documentation is automatically built and deployed to GitHub pages by the CI environment. The online documentation is available at: https://stnolting.github.io/neorv32/files.html

---

14 This driver file only represents a dummy, since the real CFS drivers are defined by the actual CFS implementation.

4.3. Application Makefile

Application compilation is based on a single GNU makefile. Each project in the `sw/example` folder features a **makefile**. All these makefiles are identical. When creating a new project, copy an existing project folder or at least the makefile to your new project folder. I suggest to create new projects also in `sw/example` to keep the file dependencies. Of course, these dependencies can be manually configured via makefiles variables when your project is located somewhere else.

Before you can use the makefiles, you need to install the RISC-V GCC toolchain. Also, you have to add the installation folder of the compiler to your system's PATH variable. More information can be found in chapter 5. Let's Get It Started!.

The makefile is invoked by simply executing `make` in your console:

```
neorv32/sw/example/blink_led$ make
```

4.3.1. Targets

Just executing `make` will show the help menu showing all available targets. The following targets are available:

`help`	Show a short help text explaining all available targets.
`check`	Check the GNU toolchain. You should run this target at least once after installing it.
`info`	Show the makefile configuration (see next chapter).
`exe`	Compile all sources and generate application executable for upload via bootloader.
`install`	Compile all sources, generate executable (via `exe` target) for upload via bootloader and generate and install IMEM VHDL initialization image file `rtl/core/neorv32_application_image.vhd`.
`all`	Execute `exe` and `install`.
`clean`	Remove all generated files in the current folder.
`clean_all`	Remove all generated files in the current folder and also removes the compiled core libraries and the compiled image generator tool.
`bootloader`	Compile all sources, generate executable and generate and install BOOTROM VHDL initialization image file `rtl/core/neorv32_bootloader_image.vhd`. This target modifies the ROM origin and length in the linker script by setting the `make_bootloader` symbol.

An assembly listing file (`main.asm`) is created by the compilation flow for further analysis or debugging purpose.

4.3.2. Configuration

The compilation flow is configured via variables right at the beginning of the makefile:

```
# ****************************************************************************
# USER CONFIGURATION
# ****************************************************************************
# User's application sources (*.c, *.cpp, *.s, *.S); add additional files here
APP_SRC ?= $(wildcard ./*.c) $(wildcard ./*.s) $(wildcard ./*.cpp) $(wildcard ./*.S)

# User's application include folders (don't forget the '-I' before each entry)
APP_INC ?= -I .
# User's application include folders - for assembly files only (don't forget the '-I' before each
entry)
ASM_INC ?= -I .

# Optimization
EFFORT ?= -Os

# Compiler toolchain
RISCV_TOOLCHAIN ?= riscv32-unknown-elf

# CPU architecture and ABI
MARCH ?= -march=rv32i
MABI  ?= -mabi=ilp32

# User flags for additional configuration (will be added to compiler flags)
USER_FLAGS ?=

# Serial port for executable upload via bootloer
COM_PORT ?= /dev/ttyUSB0

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

`APP_SRC`	The source files of the application (*.c, *.cpp, *.S and *.s files are allowed; file of these types in the project folder are automatically added via wildcards). Additional files can be added; separated by white spaces
`APP_INC`	Include file folders; separated by white spaces; must be defined with `-I` prefix
`ASM_INC`	Include file folders that are used only for the assembly source files (*.S/*.s).
`EFFORT`	Optimization level, optimize for size (`-Os`) is default; legal values: `-O0 -O1 -O2 -O3 -Os`
`RISCV_TOOLCHAIN`	The toolchain to be used; follows the naming convention architecture-vendor-output
`MARCH`	The architecture of the RISC-V CPU. Only RV32 is supported by the NEORV32. Enable compiler support of optional CPU extension by adding the according extension letter (e.g. `rv32im` for `M` CPU extension). See 5.8. Enabling RISC-V CPU Extensions
`MABI`	The default 32-bit integer ABI. Do not change.
`USER_FLAGS`	Additional flags that will be forwarded to the compiler tools
`NEORV32_HOME`	Relative or absolute path to the NEORV32 project home folder. Adapt this if the makefile/project is not in the project's `sw/example` folder.
`COM_PORT`	Default serial port for executable upload via bootloader.

4.3.3. Default Compilation Flags

The following default compiler flags are used for compiling an application. These flags are defined via the `CC_OPTS` variable. Custom flags can be added via the `USER_FLAGS` variable to the `CC_OPTS` variable.

`-Wall`	Enable all compiler warnings.
`-ffunction-sections` `-fdata-sections`	Put functions and data segment in independent sections. This allows a code optimization as dead code and unused data can be easily removed.
`-nostartfiles`	Do not use the default start code. The makefiles use the NEORV32-specific start-up code instead (`sw/common/crt0.S`).
`-Wl,--gc-sections`	Make the linker perform dead code elimination.
`-lm`	Include/link with `math.h`
`-lc`	Search for the standard C library when linking
`-lgcc`	Make sure we have no unresolved references to internal GCC library subroutines
`-mno-fdiv`	Use builtin functions for floating-point divisions and square roots (since the according instructions are not supported yet)
`-falign-functions=4` `-falign-labels=4` `-falign-loops=4` `-falign-jumps=4`	Force a 32-bit alignment of functions and labels (branch/jump/call targets). This increases performance as it simplifies instruction fetch when using the `C` extension. As a drawback this will also slightly increase the program code.

> *i* The makefile configuration variables can be (re-)defined directly when invoking the makefile. For example: `$ make MARCH=-march=rv32ic clean_all exe`

4.4. Executable Image Format

When all the application sources have been compiled and linked, a final executable file has to be generated. For this purpose, the makefile uses the NEORV32-specific linker script `sw/common/neorv32.ld` to map all the sections into only four final sections: `.text`, `.rodata`, `.data` and `.bss`. These four section contain everything required for the application to run:

`.text`	Executable instructions generated from the start-up code and all application sources
`.rodata`	Constants (like strings) from the application; also the initial data for initialized variables
`.data`	This section is required for the address generation of fixed (= global) variables only
`.bss`	This section is required for the address generation of dynamic memory constructs only

The `.text` and `.rodata` sections are mapped to processor's instruction memory space and the `.data` and `.bss` sections are mapped to the processor's data memory space.

Finally, the `.text`, `.rodata` and `.data` sections are extracted and concatenated into a single file `main.bin`.

Executable Image Generator

The file `main.bin` is processed by the NEORV32 image generator (`sw/image_gen`) to generate the final executable. The image generator can generate three types of executables, selected by a flag when calling the generator:

`-app_bin`	Generates an executable binary file `neorv32_exe.bin` (for UART uploading via the bootloader)
`-app_img`	Generates an executable VHDL memory initialization image for the processor-internal IMEM. This option generates the `rtl/core/neorv32_application_image.vhd` file.
`-bld_img`	Generates an executable VHDL memory initialization image for the processor-internal BOOT ROM. This option generates the `rtl/core/neorv32_bootloader_image.vhd` file.

All these options are managed by the makefile – so you don't actually have to think about them. The normal application compilation flow will generate the `neorv32_exe.bin` file in the current software project folder ready for upload via UART to the NEORV32 bootloader.

This executable version has a very small header consisting of three 32-bit words located right at the beginning of the file. This header is generated by the image generator (`sw/image_gen`). The image generator is automatically compiled when invoking the makefile.

The first word of the executable is the signature word and is always `0x4788CAFE`. Based on this word, the bootloader can identify a valid image file. The next word represents the size in bytes of the <u>actual program image</u>. A simple "complement" checksum of the actual program image is given by the third word. This provides a simple protection against data transmission or storage errors.

4.5. Bootloader

The default bootloader (`sw/bootloader/bootloader.c`) of the NEORV32 processor allows to upload new program executables at every time. If there is an external SPI flash connected to the processor (like the FPGA's configuration memory), the bootloader can store the program executable to it. After reset, the bootloader can directly boot from the flash without any user interaction.

> ⚠️ The bootloader is only implemented when the `BOOTLOADER_EN` generic is `true` and requires the CSR access CPU extension (`CPU_EXTENSION_RISCV_Zicsr` generic is `true`).

> ℹ️ The bootloader requires the **primary UART** `UART0` for user interaction, executable upload and SPI flash programming (`IO_UART0_EN` generic is `true`).

> ℹ️ For the **automatic boot** from an SPI flash, the SPI controller has to be implemented (`IO_SPI_EN` generic is `true`) and the machine system timer MTIME has to be implemented (`IO_MTIME_EN` generic is `true`), too, to allow an auto-boot timeout counter.

To interact with the bootloader, attach the primary UART (UART0) signals (`uart0_txd_o` and `uart0_rxd_o`) of the processor's top entity via a COM port (-adapter) to a computer, configure your terminal program using the following settings and perform a reset of the processor. Hardware flow control is not used so the according interface signals can be ignored.

Terminal console settings (19200-8-N-1):

- 19200 Baud
- 8 data bits
- No parity bit
- 1 stop bit
- Newline on \r\n (carriage return, newline) – also for sending!
- No transfer protocol for sending data, just the raw byte stuff

The bootloader uses the LSB of the top entity's `gpio_o` output port as high-active status LED (all other output pin are set to low level by the bootloader). After reset, this LED will start blinking at ~2Hz and the following intro screen should show up in your terminal:

```
<< NEORV32 Bootloader >>

BLDV: Mar 23 2021
HWV:  0x01050208
CLK:  0x05F5E100
USER: 0x10000DE0
MISA: 0x40901105
ZEXT: 0x00000023
PROC: 0x0EFF0037
IMEM: 0x00004000 bytes @ 0x00000000
DMEM: 0x00002000 bytes @ 0x80000000

Autoboot in 8s. Press key to abort.
```

> ⚠️ The uploaded executables are always stored to the instruction space starting at the base address of the instruction space.

This start-up screen also gives some brief information about the bootloader and several system parameters:

BLDV   Bootloader version (built date).

HWV   Processor hardware version (from the `mimpid` CSR) in BCD format (example: `0x01040606 = v1.4.6.6`).

USER   Custom user code (from the `USER_CODE` generic).

CLK   Processor clock speed in Hz (via the `SYSINFO` module, from the `CLOCK_FREQUENCY` generic).

MISA   CPU extensions (from the `misa` CSR).

ZEXT   CPU sub-extensions (from the `mzext` CSR)

PROC   Processor configuration (via the `SYSINFO` module, from the IO and MEM config. generics).

IMEM   IMEM memory base address and size in byte.

DMEM   DMEM memory base address and size in byte.

Now you have 8 seconds to press any key. Otherwise, the bootloader starts the auto boot sequence. When you press any key within the 8 seconds, the actual bootloader user console starts:

```
<< NEORV32 Bootloader >>

BLDV: Mar 23 2021
HWV:  0x01050208
CLK:  0x05F5E100
USER: 0x10000DE0
MISA: 0x40901105
ZEXT: 0x00000023
PROC: 0x0EFF0037
IMEM: 0x00004000 bytes @ 0x00000000
DMEM: 0x00002000 bytes @ 0x80000000

Autoboot in 8s. Press key to abort.
Aborted.

Available commands:
 h: Help
 r: Restart
 u: Upload
 s: Store to flash
 l: Load from flash
 e: Execute
CMD:>
```

The auto-boot countdown is stopped and now you can enter a command from the list to perform the corresponding operation:

- **h**: Show the help text (again)
- **r**: Restart the bootloader and the auto-boot sequence
- **u**: Upload new program executable (`neorv32_exe.bin`) via UART into the instruction memory
- **s**: Store executable to SPI flash at `spi_csn_o(0)`
- **l**: Load executable from SPI flash at `spi_csn_o(0)`
- **e**: Start the application, which is currently stored in the instruction memory
- **#**: Shortcut for executing **u** and **e** afterwards (not shown in help menu)

A new executable can be uploaded via UART by executing the `u` command. The executable can be directly executed via the `e` command. To store the recently uploaded executable to an attached SPI flash press `s`. To directly load an executable from the SPI flash press `l`. The bootloader and the auto-boot sequence can be manually restarted via the `r` command.

> ℹ The CPU is in machine level privilege mode after reset. When the bootloader boots an application, this application is also started in machine level privilege mode.

4.5.1. External SPI Flash for Booting

If you want the NEORV32 bootloader to automatically fetch and execute an application at system start, you can store it to an external SPI flash. The advantage of the external memory is to have a non-volatile program storage, which can be re-programmed at any time just by executing some bootloader commands. Thus, no FPGA bitstream recompilation is required at all.

SPI Flash Requirements

The bootloader can access an SPI compatible flash via the processor top entity's SPI port and connected to chip select `spi_csn_o(0)`. The flash must be capable of operating at least at 1/8 of the processor's main clock. Only single read and write byte operations are used. The address has to be 24 bit long. Furthermore, the SPI flash has to support at least the following commands:

- READ   (0x03)
- READ STATUS   (0x05)
- WRITE ENABLE   (0x06)
- PAGE PROGRAM   (0x02)
- SECTOR ERASE   (0xD8)
- READ ID   (0x9E)

Compatible (FGPA configuration) SPI flash memories are for example the **Winbond W25Q64FV** or the **Micron N25Q032A**.

SPI Flash Configuration

The base address `SPI_FLASH_BOOT_ADR` for the executable image inside the SPI flash is defined in the "user configuration" section of the bootloader source code (`sw/bootloader/bootloader.c`). Most FPGAs, that use an external configuration flash, store the golden configuration bitstream at base address 0. Make sure there is no address collision between the FPGA bitstream and the application image. You need to change the default sector size if your Flash has a sector size greater or less than 64kB:

```
/** SPI flash boot image base address */
#define SPI_FLASH_BOOT_ADR     0x00800000

/** SPI flash sector size in bytes */
#define SPI_FLASH_SECTOR_SIZE  (64*1024)
```

> ⚠ For any change you made inside the bootloader, you have to recompile the bootloader (5.10. Customizing the Internal Bootloader) and do a new synthesis of the processor.

4.5.2. Auto Boot Sequence

When you reset the NEORV32 processor, the bootloader waits 8 seconds for a user console input before it starts the automatic boot sequence. This sequence tries to fetch a valid boot image from the external SPI flash, connected to SPI chip select `spi_csn_o(0)`. If a valid boot image is found and can be successfully transferred into the instruction memory, it is automatically started. If no SPI flash was detected or if there was no valid boot image found, the bootloader stalls and the status LED is permanently activated.

4.5.3. Bootloader Error Codes

If something goes wrong during bootloader operation, an error code is shown. In this case the processor stalls, a bell command and one of the following error codes are send to the terminal, the bootloader status LED is permanently activated and the system must be reset manually.

ERROR_0   If you try to transfer an invalid executable (via UART or from the external SPI flash), this error message shows up. There might be a transfer protocol configuration error in the terminal program. See section 5.6. Uploading and Starting of a Binary Executable Image via UART for more information. Also, if no SPI flash was found during an auto-boot attempt, this message will be displayed.

ERROR_1   Your program is way too big for the internal processor's instructions memory. Increase the memory size or reduce (optimize!) your application code.

ERROR_2   This indicates a checksum error. Something went wrong during the transfer of the program image (upload via UART or loading from the external SPI flash). If the error was caused by a UART upload, just try it again. When the error was generated during a flash access, the stored image might be corrupted.

ERROR_3   This error occurs if the attached SPI flash cannot be accessed. Make sure you have the right type of flash and that it is properly connected to the NEORV32 SPI port using chip select #0.

ERROR_4   The instruction memory is marked as read-only. Set the `MEM_INT_IMEM_ROM` generic to `false` to allow write accesses.

ERROR_5   This error pops up when an unexpected exception or interrupt was triggered. The cause of the trap (`mcause` ID) is displayed for further investigation.

ERROR_?   Something really bad happened when there is no specific error code available...

4.5.4. Final Notes

⚠️   The bootloader is intended to work independent of the actual hardware (-configuration). Hence, it should be compiled with the minimal base ISA only. The current version of the bootloader uses the `rv32i` ISA – so it will not work on `rv32e` architectures. To make the bootloader work on embedded CPU, recompile it using the `rv32e` ISA (see chapter 5.10. Customizing the Internal Bootloader).

4.6. NEORV32 Runtime Environment

The NEORV32 provides a minimal runtime environment (RTE) that mainly takes care of

- a clean application start
- a clean handling of exceptions/interrupts

> *i* Performance or latency-optimized applications or embedded operating systems should use a custom trap management.

4.6.1. CRT0 Start-Up Code

The initial part of the runtime environment is the `sw/common/crt0.S` application start-up code. This piece of code is automatically linked with every application program and represents the starting point for every application. Hence, it is directly executed after reset. The start-up code performs the following operations:

- Initialize integer registers `x1` – `x15/x31`.
- Initialize all CPU core CSRs.
- Initialize the global pointer `gp` according to the `.data` segment layout provided by the linker script.
- Clear IO area: Write zero to **all** memory-mapped registers within the IO region. If certain devices have not been implemented, a bus access fault exception will occur. This exception is captured by a simple dummy handler in the start-up code.
- Clear the `.bss` section defined by the linker script.
- Copy read-only data from the `.text` section to the `.data` section to set initialized variables.
- Call the application's `main` function (with no arguments).
- If the main function returns, the processor goes to an endless sleep mode (using a simple loop or via the `WFI` instruction if available).

4.6.2. Using the NEORV32 Runtime Environment (RTE)

After system start-up, the actual runtime environment is responsible for catching all implemented exceptions and interrupts. To activate the NEORV32 RTE execute the following function:

```
void neorv32_rte_setup(void);
```

This setup initializes the RISC-V-compatible `mtvec` CSR, which provides the base address for all trap handlers. The address stored to this register reflects the *first-level exception handler* provided by the NEORV32 RTE. Whenever an exception or interrupt is triggered, this *first-level handler* is called.

The *first-level handler* performs a complete context save, analyzes the source of the exception/interrupt and calls the according *second-level exception* handler, which actually takes care of the exception/interrupt handling. For this, the RTE manages a private look-up table to store the addresses of the according trap handlers.

After the initial setup of the RTE, each entry in the trap handler's look-up table is initialized with a debug handler, that outputs detailed hardware information **via the primary UART (UART0)** when triggered. This is intended as a fall-back for debugging or for accidentally-triggered exceptions/interrupts.

For instance, an illegal instruction exception caught by the RTE debug handler might look like this:

```
<RTE> Illegal instruction @0x000002d6, MTVAL=0x00001537 </RTE>
```

To install the **actual application's trap handlers** the NEORV32 RTE provides functions for installing and uninstalling trap handler for each implemented exception/interrupt source.

```
int neorv32_rte_exception_install(uint8_t id, void (*handler)(void));
```

The following `id` exception IDs are available:

ID name [C]	Description / exception or interrupt causing event
`RTE_TRAP_I_MISALIGNED`	Instruction address misaligned
`RTE_TRAP_I_ACCESS`	Instruction (bus) access fault
`RTE_TRAP_I_ILLEGAL`	Illegal instruction
`RTE_TRAP_BREAKPOINT`	Breakpoint (EBREAK instruction)
`RTE_TRAP_L_MISALIGNED`	Load address misaligned
`RTE_TRAP_L_ACCESS`	Load (bus) access fault
`RTE_TRAP_S_MISALIGNED`	Store address misaligned
`RTE_TRAP_S_ACCESS`	Store (bus) access fault
`RTE_TRAP_MENV_CALL`	Environment call from machine mode (ECALL instruction)
`RTE_TRAP_UENV_CALL`	Environment call from user mode (ECALL instruction)
`RTE_TRAP_MTI`	Machine timer interrupt (via MTIME)
`RTE_TRAP_MEI`	Machine external interrupt
`RTE_TRAP_MSI`	Machine software interrupt
`RTE_TRAP_FIRQ_0`	Fast interrupt channel 0
`RTE_TRAP_FIRQ_1`	Fast interrupt channel 1
`RTE_TRAP_FIRQ_2`	Fast interrupt channel 2
`RTE_TRAP_FIRQ_3`	Fast interrupt channel 3
`RTE_TRAP_FIRQ_4`	Fast interrupt channel 4
`RTE_TRAP_FIRQ_5`	Fast interrupt channel 5
`RTE_TRAP_FIRQ_6`	Fast interrupt channel 6
`RTE_TRAP_FIRQ_7`	Fast interrupt channel 7
`RTE_TRAP_FIRQ_8`	Fast interrupt channel 8
`RTE_TRAP_FIRQ_9`	Fast interrupt channel 9
`RTE_TRAP_FIRQ_10`	Fast interrupt channel 10
`RTE_TRAP_FIRQ_11`	Fast interrupt channel 11
`RTE_TRAP_FIRQ_12`	Fast interrupt channel 12
`RTE_TRAP_FIRQ_13`	Fast interrupt channel 13
`RTE_TRAP_FIRQ_14`	Fast interrupt channel 14
`RTE_TRAP_FIRQ_15`	Fast interrupt channel 15

When installing a custom handler function for any of these exception/interrupts, make sure the function uses <u>no attributes</u> (especially no *interrupt* attribute!), has <u>no arguments</u> and <u>no return value</u> like in the following example:

```
void handler_xyz(void) {

  // handle exception/interrupt...
}
```

> ⚠️ Do **NOT** use the `((interrupt))` attribute for the application exception handler functions! This will place an `mret` instruction to the end of it making it impossible to return to the first-level exception handler of the RTE, which will cause stack corruption.

Example: Installation of the MTIME interrupt handler:

```
neorv32_rte_exception_install(EXC_MTI, handler_xyz);
```

To remove a previously installed exception handler call the according uninstall function from the NEORV32 runtime environment. This will replace the previously installed handler by the initial debug handler, so even uninstalled exceptions and interrupts are further captured.

```
int neorv32_rte_exception_uninstall(uint8_t id);
```

Example: Removing the MTIME interrupt handler:

```
neorv32_rte_exception_uninstall(EXC_MTI);
```

> ℹ️ More information regarding the NEORV32 runtime environment can be found in the `doxygen` software documentation (also available online at GitHub pages).

5. Let's Get It Started!

To make your NEORV32 project run, follow the guides from the upcoming sections. Follow these guides step by step and in the presented order.

5.1. Toolchain Setup

The default toolchain for this project is **`riscv32-unknown-elf`**. Of course you can use any other RISC-V toolchain (like `riscv64-unknown-elf`). Just change the `RISCV_TOOLCHAIN` variable in the application makefile(s) according to your needs or define this variable when invoking the makefile.

There are two possibilities to get the actual RISC-V GCC toolchain:

1. Download and build the official RISC-V GNU toolchain yourself
2. Download and install a prebuilt version of the toolchain

Keep in mind that – for instance – a `rv32imc` toolchain only provides library code compiled with compressed (`c`) and mul/div instructions (`m`)! Hence, this code cannot be executed (without emulation) on an architecture without these extensions!

5.1.1. Building the Toolchain from Scratch

To build the toolchain by yourself you can follow the guide from the official https://github.com/riscv/riscv-gnu-toolchain GitHub page.

The official RISC-V repository uses submodules. You need the --recursive option to fetch the submodules automatically:

```
$ git clone --recursive https://github.com/riscv/riscv-gnu-toolchain
```

Download and install the prerequisite standard packages:

```
$ sudo apt-get install autoconf automake autotools-dev curl python3 libmpc-dev libmpfr-dev libgmp-dev gawk build-essential bison flex texinfo gperf libtool patchutils bc zlib1g-dev libexpat-dev
```

To build the Linux cross-compiler, pick an install path. If you choose, say, `/opt/riscv`, then add `/opt/riscv/bin` to your `PATH` now.

```
$ export PATH=$PATH:/opt/riscv/bin
```

Then, simply run the following commands in the RISC-V GNU toolchain source folder (for **`rv32i`**):

```
riscv-gnu-toolchain$ ./configure --prefix=/opt/riscv --with-arch=rv32i --with-abi=ilp32
riscv-gnu-toolchain$ make
```

⚠️ Keep in mind that – for instance – a `rv32imc` (architecture) toolchain only provides library code compiled <u>with</u> compressed (`c`) and mul/div <u>instructions</u> (`m`)! Hence, this code cannot be executed (without emulation) on an architecture <u>without</u> these extensions!

After a while (hours!) you will get `riscv32-unknown-elf-gcc` and all of its friends in your `/opt/riscv/bin` folder.

5.1.2. Downloading and Installing a Prebuilt Toolchain

Alternatively, you can download a prebuilt version of the toolchain.

Use The Toolchain I have Build

I have compiled the toolchain on a 64-bit x86 Ubuntu (Ubuntu on Windows, actually) and uploaded it to GitHub. You can directly download the according toolchain archive as **single zip-file** within a **packed release** from github.com/stnolting/riscv-gcc-prebuilt.

Unpack the downloaded toolchain archive and copy the content to a location in your file system (e.g. `/opt/riscv`). More information about downloading and installing my prebuilt toolchains can be found in the repository's README.

Use The Toolchain Provided by SiFive

Alternatively, you can also download (for free) and use one of the GCC toolchains provided by SiFive (github.com/sifive/freedom-tools/releases). I've tried their toolchains and they work like a charm. Make sure to set `RISCV_TOOLCHAIN=riscv64-unknown-elf` (in the makefile or in the console when invoking the makefile) when using it. Also keep in mind that the SiFive toolchains were compiled for

⚠️ The SiFive toolchains were build for more sophisticated architectures (`rv64imafdc`). While theses toolchains also allow to emit 32-bit RISC-V code, the toolchain only provides library code compiled <u>with</u> the machine options enabled at compile time! Hence, this code cannot be executed (without emulation) on an architecture <u>without</u> these extensions (does not apply for soft-floates).

Unpack the downloaded toolchain archive and copy the content to a location in your file system (e.g. `/opt/riscv`).

Using Another Toolchain

ℹ️ Of course you can also use any other prebuilt version of the toolchain. Make sure it is a `riscv32-unknown-elf` or `riscv64-unknown-elf` (that can also emit 32-bit code) toolchain, supports the `rv32i/e` architecture and uses the `ilp32` or `ilp32e` ABI.

5.1.3. Installation

Now you have the binaries. The last step is to add them to your `PATH` environment variable (if you have not already done so). Make sure to add the <u>binaries folder</u> (`bin`) of your toolchain.

```
$ export PATH:$PATH:/opt/riscv/bin
```

You should add this command to your `.bashrc` (if you are using bash) to automatically add the RISC-V toolchain at every console start.

5.1.4. Testing the Installation

To make sure everything works fine, navigate to an example project in the NEORV32 example folder and execute the following command:

```
neorv32/sw/example/blink_led$ make check
```

This will test all the tools required for the NEORV32. Everything is working fine if `Toolchain check OK` appears at the end.

5.2. General Hardware Setup

The following steps are required to generate a bitstream for your FPGA board. If you want to run the **NEORV32 processor** in simulation only, the following steps might also apply.

> ⓘ Check out the example setups (online @GitHub) in the boards folder, which provides script-based demo projects for various FPGA boars.

In this tutorial we will use a test implementation of the **processor** – using most of the processor's optional modules but just propagating the minimal signals to the outer world. Hence, this guide is intended as evaluation or "hello world" project to check out the NEORV32. A little note: The order of the following steps might be a little different for your specific EDA tool.

1. Create a new project with your FPGA EDA tool of choice.

2. Add all VHDL files from the project's `rtl/core` folder to your project. Make sure to *reference* the files only – do not copy them.

3. Make sure to add all the rtl files to a new **library** called `neorv32`. If your FPGA tools does not provide a field to enter the library name, check out the "properties" menu of the rtl files.

4. The `rtl/core/neorv32_top.vhd` VHDL file is the top entity of the NEORV32 processor. If you already have a design, instantiate this unit into your design and proceed.

5. If you do not have a design yet and just want to check out the NEORV32 – no problem! In this guide we will use a simplified top entity, that encapsulated the actual processor top entity. Add the `rtl/core/top_templates/`**`neorv32_test_setup.vhd`** VHDL file to your project too, and select it as top entity.

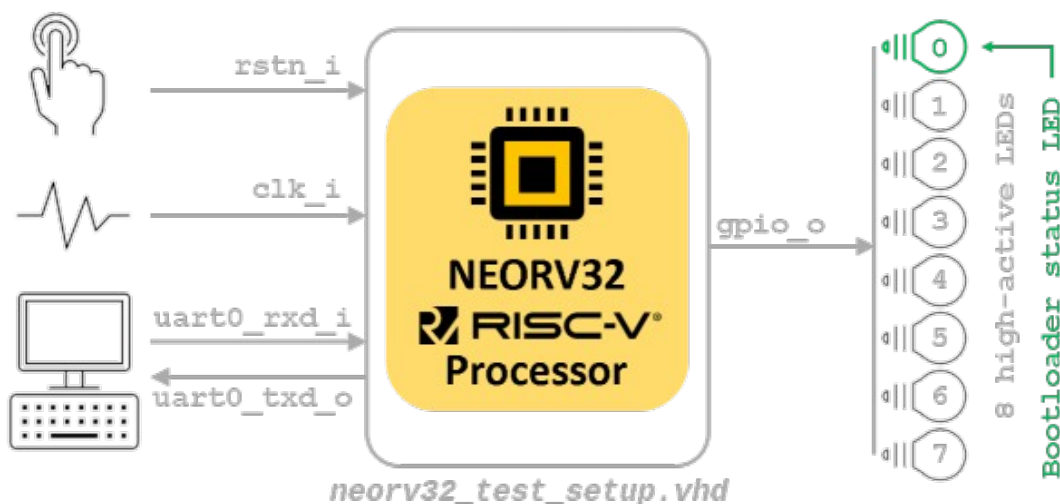6. This test setup provides a minimal test hardware setup:

Figure 8: Hardware configuration of the NEORV32 test setup

7. This test setup only implements some very basic processor and CPU features. Also, only the minimum number of signals is propagated to the outer world. Please note that the **reset** input signal `rstn_i` is **low-active**.

8. The configuration of the NEORV32 processor is done using the generics of the instantiated processor top entity. Let's keep things simple at first and use the default configuration (see below).

9. There is one generic that has to be set according to your FPGA / board: The clock frequency of the top's clock input signal (`clk_i`). Use the `CLOCK_FREQUENCY` generic to specify your clock source's frequency in Hertz (Hz) ($\rightarrow$ the default value you need to adapt is marked in **orange**).

```
neorv32_top_inst: neorv32_top
generic map (
  -- General --
  CLOCK_FREQUENCY             => 100000000, -- in Hz
  BOOTLOADER_EN               => true,
  USER_CODE                   => x"00000000",
  ...
  -- Internal Instruction memory --
  MEM_INT_IMEM_EN             => true,
  MEM_INT_IMEM_SIZE           => 16*1024,
  MEM_INT_IMEM_ROM            => false,
  -- Internal Data memory --
  MEM_INT_DMEM_EN             => true,
  MEM_INT_DMEM_SIZE           => 8*1024,
  ...
```

10. If you feel like it – or if your FPGA does not provide enough resources – you can modify the memory sizes (`MEM_INT_IMEM_SIZE` and `MEM_INT_DMEM_SIZE` – marked in **red** and **blue**) or exclude certain peripheral modules from implementation. But as mentioned above, let's keep things simple and use the standard configuration for now.

11. For this setup, we will only use the processor-internal data and instruction memories for the test setup. So make sure, the instruction and data space sizes are always equal to the sizes of the internal memories (i.e. `MEM_INT_IMEM_SIZE` == `MEM_ISPACESIZE` and `MEM_INT_DMEM_SIZE` == `MEM_DSPACESIZE`).

> *i* Keep the internal instruction and data memory sizes in mind – these values are required for setting up the software framework in the next section 5.3. General Software Framework Configuration.

12. Depending on your FPGA tool of choice, it is time to assign the signals of the test setup top entity to the according pins of your FPGA board. All the signals can be found in the entity declaration:

```
entity neorv32_test_setup is
  port (
    -- Global control --
    clk_i       : in  std_ulogic := '0'; -- global clock, rising edge
    rstn_i      : in  std_ulogic := '0'; -- global reset, low-active, async
    -- GPIO --
    gpio_o      : out std_ulogic_vector(7 downto 0); -- parallel output
    -- UART0 --
    uart0_txd_o : out std_ulogic;        -- UART0 send data
    uart0_rxd_i : in  std_ulogic := '0' -- UART0 receive data
  );
end neorv32_test_setup;
```

13. Attach the clock input `clk_i` to your clock source and connect the reset line `rstn_i` to a button of your FPGA board. Check whether it is low-active or high-active – the reset signal of the processor is **low-active**, so maybe you need to invert the input signal.

14. If possible, connected at least bit #0 of the GPIO output port `gpio_o` to a high-active LED (invert the signal when your LEDs are low-active).

15. Finally, connect the **primary UART (UART0)** communication signals `uart0_txd_o` and `uart0_rxd_i` to your serial host interface (dedicated pins, USB-to-serial converter, etc.). Hardware flow control (RTS/CTS) is not used so the according interface signals can be ignored.

16. Perform the project HDL compilation (synthesis, mapping, bitstream generation).

17. Download the generated bitstream into your FPGA ("program" it) and press the reset button (just to make sure everything is sync).

18. Done! If you have assigned the bootloader status LED (bit #0 of the GPIO output port), it should be flashing now and you should receive the bootloader start prompt via the UART.

5.3. General Software Framework Configuration

While your synthesis tool is crunching the NEORV32 HDL files, it is time to configure the project's software framework for your processor hardware setup.

1. You need to tell the linker the size of the processor's instruction and data memories. This has to be identical to the hardware memory configuration (see 5.2. General Hardware Setup).

2. Open the NEORV32 linker script `sw/common/neorv32.ld` with a text editor. Right at the beginning of the linker script you will find the memory configuration:

> ⚠️ The `rom` region provides conditional assignments (via symbol `make_bootloader`) for the origin and the length depending on whether the executable is built as normal application (for the IMEM) or as bootloader code (for the BOOTROM). **To modify the IMEM configuration of the `rom` region, make sure to only edit the second values for *ORIGIN* and *LENGTH* (marked in red).**

```
MEMORY
{
  rom (rx)  : ORIGIN = DEFINED(make_bootloader) ? 0xFFFF0000 : 0x00000000, LENGTH =
DEFINED(make_bootloader) ? 4*1024 : 16*1024

  ram (rwx) : ORIGIN = 0x80000000, LENGTH =  8*1024
}
```

3. There are four parameters that are relevant here (only the right value for the rom section): The origin and the length of the instruction memory (named `rom`) and the origin and the length of the data memory (named `ram`). These four parameters have to be always sync to your hardware memory configuration as described in section 5.2. General Hardware Setup:

> ⚠️ The `rom` `ORIGIN` parameter has to be equal to the configuration of the NEORV32 `ispace_base_c` (default: `0x00000000`) VHDL package configuration constant. The `ram` `ORIGIN` parameter has to be equal to the configuration of the NEORV32 `dspace_base_c` (default: `0x80000000`) VHDL package configuration constant.

> ⚠️ The `rom` `LENGTH` and the `ram` `LENGTH` parameters have to match the available memory sizes. For instance, if the system does not have any external memories connected, the `rom` `LENGTH` parameter has to fit the size of the processor-internal IMEM (defined via top's `MEM_INT_IMEM_SIZE` generic) and the `ram` `LENGTH` parameter has to fit the size of the processor-internal DMEM (defined via top's `MEM_INT_DMEM_SIZE` generic).

5.4. Building the Software Documentation

If you wish, you can generate the documentation of the NEORV32 software framework. This doxygen-based documentation illustrates the core libraries as well as all the example programs. A deployed version of the documentation can be found online at GitHub pages.

1. Make sure `doxygen` is installed. Navigate to the `docs` folder and generate the documentation files using the provided doxygen makefile:

```
neorv32/docs$ doxygen Doxyfile
```

2. Doxygen will generate a HTML-based documentary. The output files are placed in (a new folder) `docs/doxygen_build/html`. Move to this folder and open `index.html` with your browser. Click on the "files" tab to see an overview of all documented files.

5.5. Application Program Compilation

1. Open a terminal console and navigate to one of the project's example programs. For example the simple `sw/example_blink_led` program. This program uses the NEORV32 `GPIO` unit to display an 8-bit counter on the lowest eight bit of the `gpio_o` port.

2. To compile the project and generate an executable simply execute:

```
neorv32/sw/example/blink_led$ make exe
```

3. This will compile and link the application sources together with all the included libraries. At the end, your application is put into an ELF file (main.elf). The image generator takes this file and creates a final executable. The makefile will show the resulting memory utilization and the executable size:

```
neorv32/sw/example/blink_led$ make exe
Memory utilization:
   text    data     bss     dec     hex filename
    852       0       0     852     354 main.elf
Executable (neorv32_exe.bin) size in bytes:
864
```

4. That's it. The `exe` target has created the actual executable (`neorv32_exe.bin`) in the current folder, which is ready to be uploaded to the processor via the bootloader and a UART interface.

ℹ️ The compilation process will also create a `main.asm` assembly listing file in the project directory. This shows the actual assembly code of the complete application.

5.6. Uploading and Starting of a Binary Executable Image via UART

We have just created the executable. Now it is time to upload it to the processor. There are basically two options to do so.

Option 1

The NEORV32 makefiles provide an `upload` target that allows to directly upload an executable from the command line. Reset the processor and execute:

```
sw/example/blink_led$ make COM_PORT=/dev/ttyUSB1 upload
```

Replace `/dev/ttyUSB1` with the actual serial port you are using to communicate with the processor. You might have to use `sudo` if the targeted tty device requires elevated access rights.

Option 2

Alternatively, you can use a standard terminal program to upload an executable. This provides a more "secure" way as you can directly interact with the bootloader console. Additionally, using a terminal program allows to directly communicate with the uploaded program.

1. Connect the **primary UART (UART0)** interface of your FPGA (board) to a COM port of your computer or use an USB-to-serial adapter.

2. Start a terminal program. In this tutorial, I am using TeraTerm for Windows. You can download it from https://ttssh2.osdn.jp/index.html.en

Make sure your terminal program can transfer the executable in raw byte mode without any protocol stuff around it.

3. Open a connection to the corresponding COM port. Configure the terminal according to the following parameters:

   - 19200 Baud
   - 8 data bits
   - 1 stop bit
   - No parity bits
   - **No transmission/flow control protocol! (just raw byte mode)**
   - Newline on `\r\n` = carriage return & newline (if configurable at all)

4. Also make sure, that single chars are transmitted without any consecutive "new line" or "carriage return" commands (this is highly dependent on your terminal application of choice, TeraTerm only sends the raw chars by default).

5. Press the NEORV32 reset button to restart the bootloader. The status LED starts blinking and the bootloader intro screen appears in your console. Hurry up and press any key (hit space!) to abort the automatic boot sequence and to start the actual bootloader user interface console.

```
<< NEORV32 Bootloader >>

BLDV: Mar 23 2021
HWV:  0x01050208
CLK:  0x05F5E100
USER: 0x10000DE0
MISA: 0x40901105
ZEXT: 0x00000023
PROC: 0x0EFF0037
IMEM: 0x00004000 bytes @ 0x00000000
DMEM: 0x00002000 bytes @ 0x80000000

Autoboot in 8s. Press key to abort.
Aborted.

Available commands:
 h: Help
 r: Restart
 u: Upload
 s: Store to flash
 l: Load from flash
 e: Execute
CMD:>
```

6. Execute the "Upload" command by typing `u`. Now the bootloader is waiting for a binary executable to be send.

```
CMD:> u
Awaiting neorv32_exe.bin...
```

7. Use the "send file" option of your terminal program to transmit the previously generated binary executable `neorv32_exe.bin`.

8. Again, make sure to transmit the executable in **raw binary mode** (no transfer protocol, no additional header stuff). When using TeraTerm, select the "binary" option in the send file dialog:
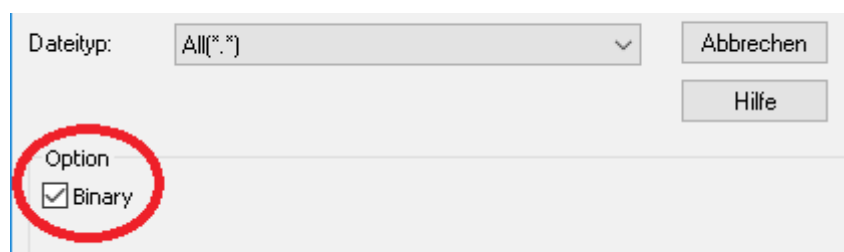

Figure 9: Transfer executable in binary mode (German version of TeraTerm)

9. If everything went fine, `OK` will appear in your terminal:

```
CMD:> u
Awaiting neorv32_exe.bin... OK
```

10. The executable now resides in the instruction memory of the processor. To execute the program right now execute the "Execute" command by typing e.

```
CMD:> u
Awaiting neorv32_exe.bin... OK
CMD:> e
Booting...

Blinking LED demo program
```

11. Now you should see the LEDs counting.

5.7. Setup of a New Application Program Project

Done with all the introduction tutorials and those example programs? Then it is time to start your own application project!

1. The easiest way of creating a new project is to make a copy of an existing project (like the blink_led project) inside the example folder. By this, all file dependencies are kept and you can start coding and compiling.

2. If you want to have he project folder somewhere else, you need to adapt the project's makefile. In the makefile you will find a variable that keeps the relative or absolute path to the NEORV32 home folder. Just modify this variable according to your project's location:

```
# Relative or absolute path to the NEORV32 home folder (use default if not set by user)
NEORV32_HOME ?= ../../..
```

3. If your project contains additional source files outside of the project folder, you can add them to the APP_SRC variable:

```
# User's application sources (add additional files here)
APP_SRC = $(wildcard *.c) ../somewhere/some_file.c
```

4. You also need to add the folder containing the include files of your new project to the APP_INC variable (do not forget the -I prefix):

```
# User's application include folders (don't forget the '-I' before each entry)
APP_INC = -I . -I ../somewhere/include_stuff_folder
```

5. If you feel like it, you can change the default optimization level:

```
# Compiler effort
EFFORT = -Os
```

5.8. Enabling RISC-V CPU Extensions

Whenever you enable a RISC-V CPU extensions via the `CPU_EXTENSION_RISCV_*` generics, you need to adapt the toolchain configuration, so the compiler actually can make use of the extension(s).

To do so, open the makefile of your project (e.g., `sw/example/blink_led/makefile`) and scroll to the "USER CONFIGURATION" section right at the beginning of the file. You need to modify the `MARCH` and `MABI` variables according to your CPU hardware configuration.

```
# CPU architecture and ABI
MARCH = -march=rv32i
MABI  = -mabi=ilp32
```

Alternatively, the `MARCH` and `MABI` configurations can be overridden when invoking the makefile:

```
$ make MARCH=-march=rv32imac clean_all all
```

The following table shows exemplay of CPU extensions and the according configuration for the `MARCH` and `MABI` variables. Of course you can also just use a subset of the available extensions (e.g. `march=rv32im` for a `rv32imc` CPU). All remaining CPU extension options do not require a modification of MARCH or MABI.

Enabled CPU Extension(s)	Toolchain MARCH	Toolchain MABI
-	MARCH=-march=rv32i	MABI = -mabi=ilp32
CPU_EXTENSION_RISCV_C	MARCH=-march=rv32ic	
CPU_EXTENSION_RISCV_C CPU_EXTENSION_RISCV_M	MARCH=-march=rv32imc	
CPU_EXTENSION_RISCV_A CPU_EXTENSION_RISCV_C CPU_EXTENSION_RISCV_M	MARCH=-march=rv32imac	

> ℹ️ The **RISC-V ISA string** (for `MARCH`) follows a certain canonical structure:
> `RV32[i/e][m][a][f][d][g][q][c][b][v][n][…]`
> Example: `rv32imac` is valid while `rv32icma` is not.

> ⚠️ **Not all NEORV32 CPU Extensions are supported by upstream GCC yet.**
> Some extensions are not yet officially ratified, but are expected to stay unchanged. Hence, there is no software support in the upstream GCC RISC-V port yet. However, intrinsic libraries are provided to utilize the extensions. See section 2.4. Instruction Sets and CPU Extensions for more information.

5.9. Building a Non-Volatile Application (Program Fixed in IMEM)

The purpose of the bootloader is to allow an easy and fast update of the application being currently executed. But maybe at some time your project has become mature and you want to actually embed your processor including the application. Of course you can store the executable to the SPI flash and let the bootloader fetch and execute it a system start. But if you don't have an SPI flash available or you want a really fast start of your applications, you can directly implement your executable within the processor internal instruction memory. When using this approach, the bootloader is no longer required. To have your application to permanently reside in the internal instruction memory, follow the upcoming steps.

⚠️ This works only for the internal instruction memory. Also make sure that the memory components (like block RAM) the IMEM is mapped to support an initialization via the bitstream.

1. At first, compile your application code by running the `make install` command (the memory utilization is not shown again when your code has already been compiled):

```
neorv32/sw/example/blink_led$ make compile
Memory utilization:
   text    data     bss     dec     hex filename
    852       0       0     852     354 main.elf
Executable (neorv32_exe.bin) size in bytes:
864
Installing application image to ../../../rtl/core/neorv32_application_image.vhd
```

2. The `install` target has created an executable, too, but this time in the form of a VHDL memory initialization file. At synthesis, this initialization will become part of the final FPGA bitstream, which in terms initializes the IMEM's blockram.

3. You need the processor to directly execute the code in the IMEM. Deactivate the implementation of the bootloader via the top entity's generic:

```
BOOTLOADER_EN => false, -- implement processor-internal bootloader?
```

4. When the bootloader is deactivated, the according ROM is removed and the CPU will start booting at the base address of the instruction memory space. Thus, the CPU directly executed your application code after reset.

5. The IMEM could be still modified, since it is implemented as RAM. This might corrupt your executable. To prevent this and to implement the IMEM as true ROM (and eventually saving some more hardware resources), active the IMEM as ROM feature using the processor's top entity generic:

```
MEM_INT_IMEM_ROM => true, -- implement processor-internal instruction memory as ROM
```

6. Perform a synthesis and upload your new bitstream. Your application code resides now unchangeable in the processor's IMEM and is directly executed after reset.

5.10. Customizing the Internal Bootloader

The bootloader provides several configuration options to customize it for your specific applications. The most important user-defined configuration options are available as C `#defines` right at the beginning of the bootloader source code `sw/bootloader/bootloader.c`):

```
/** UART BAUD rate */
#define BAUD_RATE              (19200)
/** Enable auto-boot sequence if != 0 */
#define AUTOBOOT_EN            (1)
/** Time until the auto-boot sequence starts (in seconds) */
#define AUTOBOOT_TIMEOUT       8
/** Set to 0 to disable bootloader status LED */
#define STATUS_LED_EN          (1)
/** SPI_DIRECT_BOOT_EN: Define/uncomment to enable SPI direct boot */
//#define SPI_DIRECT_BOOT_EN
/** Bootloader status LED at GPIO output port */
#define STATUS_LED             (0)
/** SPI flash boot image base address (warning! address might wrap-around!) */
#define SPI_FLASH_BOOT_ADR     (0x00800000)
/** SPI flash chip select line at spi_csn_o */
#define SPI_FLASH_CS           (0)
/** Default SPI flash clock prescaler */
#define SPI_FLASH_CLK_PRSC     (CLK_PRSC_8)
/** SPI flash sector size in bytes (default = 64kb) */
#define SPI_FLASH_SECTOR_SIZE  (64*1024)
/** ASCII char to start fast executable upload process  */
#define FAST_UPLOAD_CMD        '#'
```

If you have modified any of the configuration parameters of the default bootloader itself you need to re-compile and re-install the bootloader.

Changing the Default Size of the Bootloader ROM

1. The NEORV32 default bootloader uses 4kB of boot ROM space. This is also the default boot ROM size. If your new/modified bootloader exceeds this size, you need to modify the boot ROM configurations.

2. Open the processor's main package file `rtl/core/neorv32_package.vhd` and edit the `boot_size_c` constant according to your requirements. The boot ROM size **must not exceed 32kB** and should be a power of two (for optimal hardware mapping).

```
-- Bootloader ROM --
constant boot_size_c : natural := 4*1024; -- bytes
```

3. Now open the NEORV32 linker script `sw/common/neorv32.ld` and adapt the `LENGTH` parameter of the `rom` according to your new memory size. `boot_size_c` and `LENGTH` have to be always identical. Do not modify the `ORIGIN` of the boot memory.

⚠️ The `rom` region provides conditional assignments (via symbol `make_bootloader`) for the origin and the length depending on whether the executable is built as normal application (for the IMEM) or as bootloader code (for the BOOTROM). **To modify the BOOTLOADER memory size, make sure to edit the first value for the origin (marked in red).**

```
MEMORY
{
  rom (rx) : ORIGIN = DEFINED(make_bootloader) ? 0xFFFF0000 : 0x00000000, LENGTH =
DEFINED(make_bootloader) ? 4*1024 : 16*1024
  ram (rwx) : ORIGIN = 0x80000000, LENGTH =  8*1024
}
```

Re-Compiling and Re-Installing the Bootloader

1. Compile and install the bootloader using the explicit `bootloader` makefile target.

```
neorv32/sw/bootloader$ make bootloader
```

2. Now perform a new synthesis / HDL compilation to update the bitstream with the new bootloader image (some synthesis tools also allow to only update the BRAM initialization without re-running the entire synthesis process).

⚠️ The bootloader is intended to work regardless of the actual NEORV32 hardware configuration – especially when it comes to CPU extensions. Hence, the bootloader should be build using the minimal `rv32i` ISA only (`rv32e` would be even better).

ℹ️ See chapter 4.5. Bootloader for more information regarding the actual bootloader and how to use it within a project.

5.11. Programming the Bootloader SPI Flash

1. At first, reset the NEORV32 processor and wait until the bootloader start screen appears in your terminal program.

2. Abort the auto boot sequence and start the user console by pressing any key.

3. Press `u` to upload the program image, that you want to store to the external flash:

```
CMD:> u
Awaiting neorv32_exe.bin...
```

4. Send the binary in raw binary via your terminal program. When the uploaded is completed and `OK` appears, press `p` to trigger the programming of the flash (do not execute the image via the `e` command as this might corrupt the image):

```
CMD:> u
Awaiting neorv32_exe.bin... OK
CMD:> p
Write 0x000013FC bytes to SPI flash @ 0x00800000? (y/n)
```

5. The bootloader shows the size of the executable and the base address inside the SPI flash where the executable is going to be stored. A prompt appears: Type `y` to start the programming or type `n` to abort.

```
CMD:> u
Awaiting neorv32_exe.bin... OK
CMD:> p
Write 0x000013FC bytes to SPI flash @ 0x00800000? (y/n) y
Flashing... OK
CMD:>
```

6. If `OK` appears in the terminal line, the programming process was successful. Now you can use the auto boot sequence to automatically boot your application from the flash at system start-up without any user interaction.

> *i*   See chapter 4.5. Bootloader for more information regarding the actual bootloader and how to use it within a project.

5.12. Simulating the Processor

Testbench

The NEORV32 project features a simple testbench (`sim/neorv32_tb.vhd`) that can be used to simulate and test the Processor setup (the testbench instantiates the `rtl/core/neorv32_top.vhd` as "device under test"). This testbench features a 100MHz clock and enables all optional peripheral devices and all optional CPU extensions (but not the embedded CPU mode).

⚠️ Please note that the true-random number generator (TRNG) **CANNOT** be simulated due to its combinatorial (looped) oscillator architecture.

The simulation setup is configured via the `User Configuration` section located right at the beginning of the testbench's architecture. Each configuration constant provides comments to explain the functionality.

Besides the actual NEO430 Processor, the testbench also simulates "external" components that are connected to the processor's external bus/memory interface. These components are:

- an external instruction memory (that also allows to booting)
- an external data memory
- an external memory to simulate "external IO devices"
- a memory-mapped registers to trigger the processor's interrupt signals

The following table shows the base addresses of these five components and their default configuration and properties (attributes: r = read, w = write, e = execute, a = atomic accesses possible, b = byte-accessible, h = half-word-accessible, w = word-accessible).

Base address	Size	Attributes	Function
`0x00000000`	`imem_size_c`	`r/w/e  a b/h/w`	External IMEM (is initialized with application image)
`0x80000000`	`dmem_size_c`	`r/w/e  a b/h/w`	External DMEM
`0xF0000000`	`64 bytes`	`r/w/e !a b/h/w`	External "IO" memory
`0xFF000000`	`4 bytes`	`-/w/-  a -/-/w`	Memory-mapped register to trigger "machine external interrupt", "machine software interrupt" and/or "SoC Fast Interrupt Channels 0..7"

Table 23: TRNG register map

The simulated NEORV32 does not use the bootloader and directly boots the current application image (from the `rtl/core/neorv32_application_image.vhd` image file). Make sure to use the `all` target of the makefile to **install** your application as VHDL image after compilation:

```
sw/example/blink_led$ make clean_all all
```

Additional simulations files can be found in `sim/rtl_files`. These files are not used yet for the actual processor project.

Simulation-Optimized CPU/Processors Modules

The `sim/rtl_modules` folder provides simulation-optimized versions of certain CPU/Processor modules. These alternatives can be used to replace the default CPU/Processor HDL files to allow faster/easier/more efficient simulation. **These files are not intended for synthesis!**

Simulation Console Output

Data written to the NEORV32 UART0 / UART1 transmitter is send to a virtual UART receiver implemented as part of the testbench. Received chars are send to the simulator console and are also stored to a file (`neorv32.testbench_uart0.out` for UART0 and `neorv32.testbench_uart1.out` for UART1) in the simulator home folder.

Faster Simulation Console Output

When printing data via the UART the communication speed will always be based on the configured BAUD rate. For a simulation this might take some time. To have a faster output you can enable the simulation mode or UART0/UART1 (see chapter 3.5.9. Primary Universal Asynchronous Receiver and Transmitter (UART0)).

ASCII data written to the UART0 will be immediately printed to the simulator console. Additionally, the ASCII data is logged in a file (`neorv32.uart0.sim_mode.text.out`) in the simulator home folder. All written 32-bit data is also dumped as 8-char hexadecimal value into a file `neorv32.uart0.sim_mode.data.out` in the simulation home folder.

ASCII data written to the UART1 will be immediately printed to the simulator console. Additionally, the ASCII data is logged in a file (`neorv32.uart1.sim_mode.text.out`) in the simulator home folder. All written 32-bit data is also dumped as 8-char hexadecimal value into a file `neorv32.uart1.sim_mode.data.out` in the simulation home folder.

You can automatically the UART0/UART1 simulation mode when compiling an application. In this case the "real" UART0/UART1 transmitter unit is permanently disabled. To enable the simulation mode just compile and install your application and <u>add</u> `UART0_SIM_MODE` for UART0 and/or `UART1_SIM_MODE` for UART1 to the compiler `USER_FLAGS` variable (do not forget the `-D` suffix flag):

```
sw/example/blink_led$ make USER_FLAGS+=-DUART0_SIM_MODE clean_all all
```

> *i*   The UART simulation output (to file and to screen) outputs "complete lines" at once. A line is completed with a <u>line feed</u> (newline, ASCII \n = 10).

Simulation with Xilinx Vivado

The project features a Vivado simulation waveform configuration in `sim/vivado`.

Simulation with GHDL

To simulate the processor using **GHDL** navigate to the `sim` folder and run the provided shell script. The simulation time can be configured in the script via the `--stop-time=4ms` argument.

```
neorv32/sim$ sh ghdl_sim.sh
```

5.13. FreeRTOS Support

A NEORV32-specific port and a simple demo for FreeRTOS (https://github.com/FreeRTOS/FreeRTOS) are available in the `sw/example/demo_freeRTOS` folder.

See the documentation (`sw/example/demo_freeRTOS/README.md`) for more information.

5.14. RISC-V Architecture Test Framework

The NEORV32 Processor passes all the tests provided by the official RISC-V Architecture Test Suite (V2.0+), which is available online at GitHub: https://github.com/riscv/riscv-arch-test

All files required for executing the test framework on a simulated instance of the processor (including port files) are located in the `riscv-arch-test` folder in the root directory of the NEORV32 repository. Take a look at the provided `riscv-arch-test/README.md` file for more information on how to run the tests and how testing is conducted in detail.