The NEORV32 RISC-V Processor

Datasheet

Version v1.7.0-r8-gcc64b5cd

Documentation

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

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

Table of Contents

Chapter 1. Overview

The NEORV32[1] is an open-source RISC-V compatible processor system that is intended as **ready-to-go** auxiliary processor within a larger SoC designs or as stand-alone custom / customizable microcontroller.

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

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

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

Check out the processor's **online User Guide** that provides hands-on tutorials to get you started.

Structure

Annotations

❗	Warning
⚠	Important
📝	Note
ℹ	Tip

1.1. Rationale

Why did you make this?

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

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

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

Why a *soft-core* processor?

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

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

Why RISC-V?

> RISC-V is a free and open ISA enabling a new era of processor innovation through open standard collaboration.
>
> — RISC-V International, https://riscv.org/about/

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

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

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

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

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

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

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

1.2. Project Key Features

- open-source and documented; including user guides to get started
- completely described in behavioral, platform-independent VHDL (yet platform-optimized modules are provided)
- fully synchronous design, no latches, no gated clocks
- small hardware footprint and high operating frequency for easy integration
- **NEORV32 CPU**: 32-bit `rv32i` RISC-V CPU
  - RISC-V compatibility: passes the official architecture tests
  - base architecture + privileged architecture (optional) + ISA extensions (optional)
  - option to add custom RISC-V instructions (as custom ISA extension)
  - rich set of customization options (ISA extensions, design goal: performance / area (/ energy), …)
  - aims to support Full Virtualization capabilities (CPU *and* SoC) to increase execution safety
  - official RISC-V open source architecture ID
- **NEORV32 Processor (SoC)**: highly-configurable full-scale microcontroller-like processor system
  - based on the NEORV32 CPU

  - optional serial interfaces (UARTs, TWI, SPI)
  - optional timers and counters (WDT, MTIME)
  - optional general purpose IO and PWM and native NeoPixel (c) compatible smart LED interface
  - optional embedded memories / caches for data, instructions and bootloader
  - optional external memory interface (Wishbone / AXI4-Lite) and stream link interface (AXI4-Stream) for custom connectivity
  - optional execute in place (XIP) module
  - on-chip debugger compatible with OpenOCD and gdb including hardware trigger module
- **Software framework**
  - GCC-based toolchain - prebuilt toolchains available; application compilation based on GNU makefiles
  - internal bootloader with serial user interface
  - 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; a deployed version is available at https://stnolting.github.io/neorv32/sw/files.html
  - FreeRTOS port + demos available

For more in-depth details regarding the feature provided by he hardware see the according sections: NEORV32 Central Processing Unit (CPU) and NEORV32 Processor (SoC).

Extensibility and Customization

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

- Processor-External Memory Interface (WISHBONE) (AXI4-Lite) for processor-external modules
- Custom Functions Subsystem (CFS) for tightly-coupled processor-internal co-processors
- Custom Functions Unit (CFU) for custom RISC-V instructions

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

1.3. Project Folder Structure

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

1.4. VHDL File Hierarchy

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

All core VHDL files from the list below 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_fifo.vhd                - General purpose FIFO component
 ├neorv32_package.vhd             - Processor/CPU main VHDL package file
 │
 ├neorv32_cpu.vhd                 - NEORV32 CPU top entity
 │ ├neorv32_cpu_alu.vhd            - Arithmetic/logic unit
 │ │ ├neorv32_cpu_cp_bitmanip.vhd   - Bit-manipulation co-processor (B ext.)
 │ │ ├neorv32_cpu_cp_cfu.vhd        - Custom functions (instruction) co-processor
(Zxcfu ext.)
 │ │ ├neorv32_cpu_cp_fpu.vhd        - Floating-point co-processor (Zfinx ext.)
 │ │ ├neorv32_cpu_cp_muldiv.vhd     - Mul/Div co-processor (M extension)
 │ │ └neorv32_cpu_cp_shifter.vhd    - Bit-shift co-processor
 │ ├neorv32_cpu_bus.vhd            - Bus interface + physical memory protection
 │ ├neorv32_cpu_control.vhd        - CPU control, exception/IRQ system and CSRs
 │ │ └neorv32_cpu_decompressor.vhd  - Compressed instructions decoder
 │ └neorv32_cpu_regfile.vhd        - Data register file
 │
 ├neorv32_boot_rom.vhd            - Bootloader ROM
 │ └neorv32_bootloader_image.vhd   - Bootloader boot ROM memory image
 ├neorv32_busswitch.vhd           - Processor bus switch for CPU buses (I&D)
 ├neorv32_bus_keeper.vhd          - Processor-internal bus monitor
 ├neorv32_cfs.vhd                 - Custom functions subsystem
 ├neorv32_debug_dm.vhd            - on-chip debugger: debug module
 ├neorv32_debug_dtm.vhd           - on-chip debugger: debug transfer module
 ├neorv32_dmem.entity.vhd         - Processor-internal data memory (entity-only!)
 ├neorv32_gpio.vhd                - General purpose input/output port unit
 ├neorv32_gptmr.vhd               - General purpose 32-bit timer
 ├neorv32_icache.vhd              - Processor-internal instruction cache
 ├neorv32_imem.entity.vhd         - Processor-internal instruction memory (entity-
only!)
 │ └neor32_application_image.vhd   - IMEM application initialization image
 ├neorv32_mtime.vhd               - Machine system timer
 ├neorv32_neoled.vhd              - NeoPixel (TM) compatible smart LED interface
 ├neorv32_pwm.vhd                 - Pulse-width modulation controller
 ├neorv32_slink.vhd               - Stream link 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
 ├neorv32_wdt.vhd                 - Watchdog timer
 ├neorv32_wishbone.vhd            - External (Wishbone) bus interface
 ├neorv32_xip.vhd                 - Execute in place module
 ├neorv32_xirq.vhd                - External interrupt controller
 │
 ├mem/neorv32_dmem.default.vhd    - _Default_ data memory (architecture-only)
 └mem/neorv32_imem.default.vhd    - _Default_ instruction memory (architecture-only)
```

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

1.5. FPGA Implementation Results

This section shows *exemplary* FPGA implementation results for the NEORV32 CPU and NEORV32 Processor modules. Note that certain configuration options might also have an impact on other configuration options. Furthermore, this report cannot cover all possible option combinations. Hence, the presented implementation results are just *exemplary*. If not otherwise mentioned all implementations use the default generic configurations.

1.5.1. CPU

HW version:	`1.6.9.8`
Top entity:	`rtl/core/neorv32_cpu.vhd`
FPGA:	Intel Cyclone IV E `EP4CE22F17C6`
Toolchain:	Quartus Prime Lite 21.1
Constraints:	**no timing constraints**, "*balanced optimization*", f_{max} from "*Slow 1200mV 0C Model*"

CPU ISA Configuration	LEs	FFs	MEM bits	DSPs	f_{max}
`rv32e`	830	400	512	0	129 MHz
`rv32i`	834	400	1024	0	129 MHz
`rv32i_Zicsr`	1328	678	1024	0	128 MHz
`rv32i_Zicsr_Zicntr`	1614	808	1024	0	128 MHz
`rv32im_Zicsr_Zicntr`	2087	983	1024	0	128 MHz
`rv32ima_Zicsr_Zicntr`	2129	987	1024	0	128 MHz
`rv32imac_Zicsr_Zicntr`	2338	992	1024	0	128 MHz
`rv32imacb_Zicsr_Zicntr`	3175	1247	1024	0	128 MHz
`rv32imacbu_Zicsr_Zicntr`	3186	1254	1024	0	128 MHz
`rv32imacbu_Zicsr_Zicntr_Zifencei`	3187	1254	1024	0	128 MHz

CPU ISA Configuration	LEs	FFs	MEM bits	DSPs	f_{max}
`rv32imacbu_Zicsr_Zicntr_Zifencei_Zfinx`	4450	1906	1024	7	123 MHz
`rv32imacbu_Zicsr_Zicntr_Zifencei_Zfinx_DebugMode`	4825	2018	1024	7	123 MHz

RISC-V Compliance

The `Zicsr` ISA extension implements the privileged machine architecture (see **`Zicsr`** Control and Status Register Access / Privileged Architecture). The `Zicntr` ISA extension implements the basic counters and timers (see **`Zicntr`** CPU Base Counters). Both extensions are *mandatory* in order to comply with the RISC-V architecture specifications.

The table above does not show *all* CPU ISA extensions. More sophisticated and application-specific options like PMP and HMP are not included in this overview.

Goal-Driven Optimization

The CPU provides further options to reduce the area footprint (for example by constraining the CPU-internal counter sizes) or to increase performance (for example by using a barrel-shifter; at cost of extra hardware). See section Processor Top Entity - Generics for more information. Also, take a look at the User Guide section Application-Specific Processor Configuration.

1.5.2. Processor - Modules

HW version:	`1.6.8.3`
Top entity:	`rtl/core/neorv32_top.vhd`
FPGA:	Intel Cyclone IV E `EP4CE22F17C6`
Toolchain:	Quartus Prime Lite 21.1
Constraints:	**no timing constraints**, "*balanced optimization*"

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

Module	Description	LEs	FFs	MEM bits	DSPs
Boot ROM	Bootloader ROM (4kB)	3	2	32768	0
BUSKEEPER	Processor-internal bus monitor	28	15	0	0
BUSSWITCH	Bus multiplexer for CPU instr. and data interface	69	8	0	0
CFS	Custom functions subsystem[2]	-	-	-	-

Module	**Description**	**LEs**	**FFs**	**MEM bits**	**DSPs**
DM	On-chip debugger - debug module	473	240	0	0
DTM	On-chip debugger - debug transfer module (JTAG)	259	221	0	0
DMEM	Processor-internal data memory (8kB)	18	2	65536	0
GPIO	General purpose input/output ports	102	98	0	0
GPTMR	General Purpose Timer	153	105	0	0
iCACHE	Instruction cache (2x4 blocks, 64 bytes per block)	417	297	4096	0
IMEM	Processor-internal instruction memory (16kB)	12	2	131072	0
MTIME	Machine system timer	345	166	0	0
NEOLED	Smart LED Interface (NeoPixel/WS28128) (FIFO_depth=1)	227	184	0	0
PWM	Pulse_width modulation controller (8 channels)	128	qq7	0	0
SLINK	Stream link interface (2xRX, 2xTX, FIFO_depth=1)	136	116	0	0
SPI	Serial peripheral interface	114	94	0	0
SYSINFO	System configuration information memory	13	11	0	0
TRNG	True random number generator	89	79	0	0
TWI	Two-wire interface	77	43	0	0
UART0, UART1	Universal asynchronous receiver/transmitter 0/1 (FIFO_depth=1)	195	143	0	0
WDT	Watchdog timer	61	46	0	0
WISHBONE	External memory interface	120	112	0	0
XIP	Execute in place module	318	244	0	0
XIRQ	External interrupt controller (32 channels)	245	200	0	0

Note that not all IOs were actually connected to FPGA pins (for example some GPIO inputs and outputs) when generating these reports.

1.5.3. Exemplary Setups

Check out the `neorv32-setups` repository (@GitHub: https://github.com/stnolting/neorv32-setups), which provides several demo setups for various FPGA boards and toolchains.

1.6. CPU Performance

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

Dhrystone

A *simple* port of the Dhrystone benchmark is also available in `sw/example/dhrystone`.

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 in MHz.

Table 2. Configuration

HW version:	`1.5.7.10`
Hardware:	32kB int. IMEM, 16kB int. DMEM, no caches, 100MHz clock
CoreMark:	2000 iterations, MEM_METHOD is MEM_STACK
Compiler:	RISCV32-GCC 10.2.0
Compiler flags:	default, see makefile

Table 3. CoreMark results

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

The CoreMark results were generated using a `rv32i` toolchain. This toolchain supports standard extensions like `M` and `C` but the built-in libraries only use the base `I` ISA.

The "*performance*" CPU configuration uses the *FAST_MUL_EN* and *FAST_SHIFT_EN* options.

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

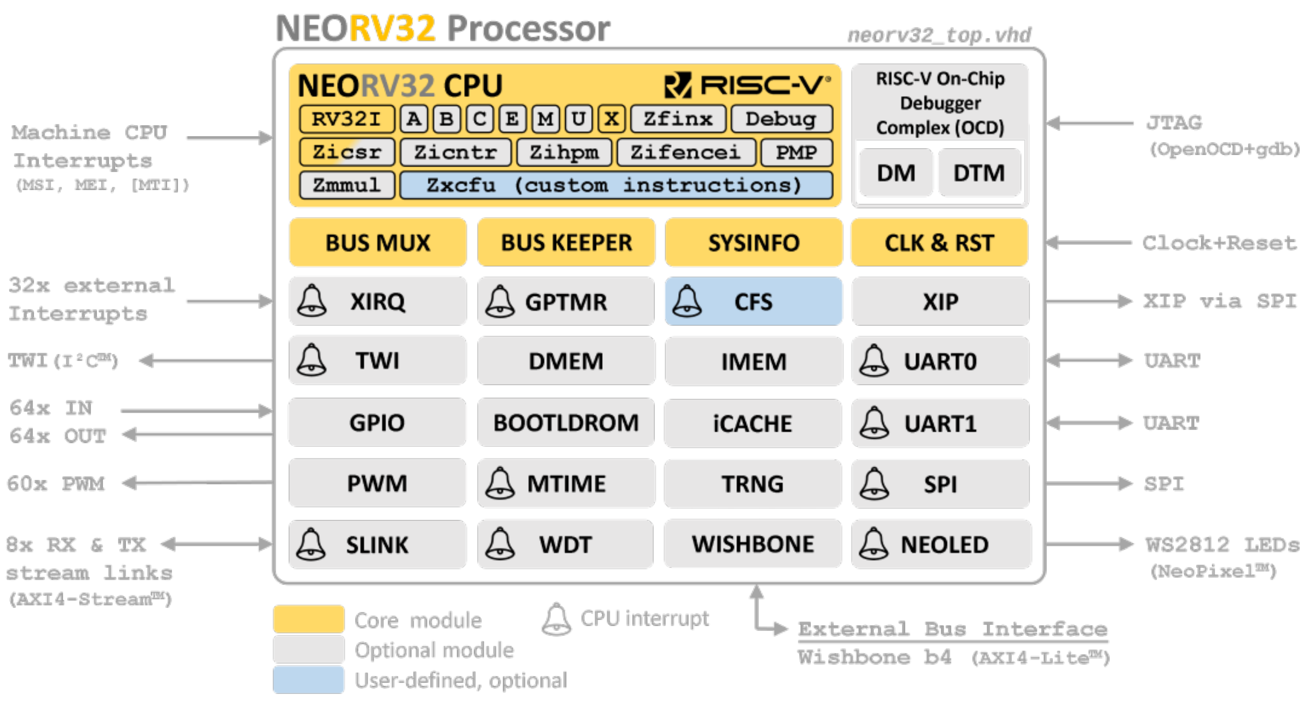
Instruction Timing.

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

[2] Resource utilization depends on custom design logic.

Chapter 2. NEORV32 Processor (SoC)

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


NEORV32 Processor
neorv32_top.vhd
NEORV32 CPU
RISC-V
RV32I A B C E M U X Zfinx Debug
Zicsr Zicntr Zihpm Zifencei PMP
Zmmul Zxcfu (custom instructions)
RISC-V On-Chip Debugger Complex (OCD)
DM DTM
BUS MUX BUS KEEPER SYSINFO CLK & RST
XIRQ GPTMR CFS XIP
TWI DMEM IMEM UART0
GPIO BOOTLDROM iCACHE UART1
PWM MTIME TRNG SPI
SLINK WDT WISHBONE NEOLED
Machine CPU Interrupts (MSI, MEI, [MTI])
32x external Interrupts
TWI (I²C™)
64x IN
64x OUT
60x PWM
8x RX & TX stream links (AXI4-Stream™)
JTAG (OpenOCD+gdb)
Clock+Reset
XIP via SPI
UART
UART
SPI
WS2812 LEDs (NeoPixel™)
Core module
Optional module
User-defined, optional
CPU interrupt
External Bus Interface Wishbone b4 (AXI4-Lite™)

Key Features

- *optional* processor-internal data and instruction memories (DMEM/IMEM) + cache (iCACHE)
- *optional* internal bootloader (BOOTROM) with UART console & SPI flash boot option
- *optional* machine system timer (MTIME), RISC-V-compatible
- *optional* two independent universal asynchronous receivers and transmitters (UART0, UART1) with optional hardware flow control (RTS/CTS) and optional RX/TX FIFOs
- *optional* 8/16/24/32-bit serial peripheral interface controller (SPI) with 8 dedicated CS lines
- *optional* two wire serial interface controller (TWI), compatible to the I²C standard
- *optional* general purpose parallel IO port (GPIO), 64xOut, 64xIn
- *optional* 32-bit external bus interface, Wishbone b4 / AXI4-Lite compatible (WISHBONE)
- *optional* 32-bit stream link interface with up to 8 independent links, AXI4-Stream compatible (SLINK)
- *optional* watchdog timer (WDT)
- *optional* PWM controller with up to 60 channels & 8-bit duty cycle resolution (PWM)
- *optional* ring-oscillator-based true random number generator (TRNG)
- *optional* custom functions subsystem for custom co-processor extensions (CFS)

- *optional* NeoPixel™/WS2812-compatible smart LED interface (**NEOLED**)
- *optional* external interrupt controller with up to 32 channels (**XIRQ**)
- *optional* general purpose 32-bit timer (**GPTMR**)
- *optional* execute in place module (**XIP**)
- *optional* on-chip debugger with JTAG TAP (**OCD**)
- bus keeper to monitor processor-internal bus transactions (**BUSKEEPER**)
- system configuration information memory to check HW configuration via software (**SYSINFO**)

2.1. Processor Top Entity - Signals

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

Default Values of Ports

All *input signals* provide default values in case they are not explicitly assigned during instantiation. For control signals the value `L` (weak pull-down) is used. For serial and parallel data signals the value `U` (unknown) is used. Pulled-down signals will not cause "accidental" system crashes since all control signals have defined level.

Configurable Amount of Channels

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

Signal	Width	Dir.	Function
			Global Control
`clk_i`	1	in	global clock line, all registers triggering on rising edge
`rstn_i`	1	in	global reset, asynchronous, **low-active**
			JTAG Access Port for On-Chip Debugger (OCD)
`jtag_trst_i`	1	in	TAP reset, low-active (optional [3])
`jtag_tck_i`	1	in	serial clock
`jtag_tdi_i`	1	in	serial data input
`jtag_tdo_o`	1	out	serial data output [4]
`jtag_tms_i`	1	in	mode select
			External Bus Interface (WISHBONE)
`wb_tag_o`	3	out	tag (access type identifier)
`wb_adr_o`	32	out	destination address
`wb_dat_i`	32	in	write data
`wb_dat_o`	32	out	read data
`wb_we_o`	1	out	write enable ('0' = read transfer)
`wb_sel_o`	4	out	byte enable

Signal	Width	Dir.	Function
`wb_stb_o`	1	out	strobe
`wb_cyc_o`	1	out	valid cycle
`wb_lock_o`	1	out	exclusive access request
`wb_ack_i`	1	in	transfer acknowledge
`wb_err_i`	1	in	transfer error
			Advanced Memory Control Signals
`fence_o`	1	out	indicates an executed *fence* instruction
`fencei_o`	1	out	indicates an executed *fencei* instruction
			Execute In Place Interface (XIP)
`xip_csn_o`	1	out	chi select, low-active
`xip_clk_o`	1	out	serial clock
`xip_sdi_i`	1	in	serial data input
`xip_sdo_o`	1	out	serial data output
			Stream Link Interface (SLINK)
`slink_tx_dat_o`	8x32	out	TX link *n* data
`slink_tx_val_o`	8	out	TX link *n* data valid
`slink_tx_rdy_i`	8	in	TX link *n* allowed to send
`slink_rx_dat_i`	8x32	in	RX link *n* data
`slink_rx_val_i`	8	in	RX link *n* data valid
`slink_rx_rdy_o`	8	out	RX link *n* ready to receive
			General Purpose Inputs & Outputs (GPIO)
`gpio_o`	64	out	general purpose parallel output
`gpio_i`	64	in	general purpose parallel input
			Primary Universal Asynchronous Receiver/Transmitter (UART0)
`uart0_txd_o`	1	out	UART0 serial transmitter
`uart0_rxd_i`	1	in	UART0 serial receiver
`uart0_rts_o`	1	out	UART0 RX ready to receive new char
`uart0_cts_i`	1	in	UART0 TX allowed to start sending
			Primary Universal Asynchronous Receiver/Transmitter (UART1)
`uart1_txd_o`	1	out	UART1 serial transmitter
`uart1_rxd_i`	1	in	UART1 serial receiver
`uart1_rts_o`	1	out	UART1 RX ready to receive new char

Signal	Width	Dir.	Function
`uart1_cts_i`	1	in	UART1 TX allowed to start sending
			Serial Peripheral Interface Controller (SPI)
`spi_sck_o`	1	out	SPI controller clock line
`spi_sdo_o`	1	out	SPI serial data output
`spi_sdi_i`	1	in	SPI serial data input
`spi_csn_o`	8	out	SPI dedicated chip select (low-active)
			Two-Wire Interface Controller (TWI)
`twi_sda_io`	1	inout	TWI serial data line
`twi_scl_io`	1	inout	TWI serial clock line
			Pulse-Width Modulation Channels (PWM)
`pwm_o`	60	out	pulse-width modulated channels
			Custom Functions Subsystem (CFS)
`cfs_in_i`	32	in	custom CFS input signal conduit
`cfs_out_o`	32	out	custom CFS output signal conduit
			Smart LED Interface - NeoPixel™ compatible (NEOLED)
`neoled_o`	1	out	asynchronous serial data output
			System time (MTIME)
`mtime_i`	64	in	machine timer time (to `time[h]` CSRs) from *external MTIME* unit if the processor-internal *MTIME* unit is NOT implemented
`mtime_o`	64	out	machine timer time from *internal MTIME* unit if processor-internal *MTIME* unit IS implemented
			External Interrupts (XIRQ)
`xirq_i`	32	in	external interrupt requests (up to 32 channels)
			RISC-V Machine-Level CPU Interrupts
`mtime_irq_i`	1	in	machine timer interrupt13 (RISC-V), high-active
`msw_irq_i`	1	in	machine software interrupt (RISC-V), high-active
`mext_irq_i`	1	in	machine external interrupt (RISC-V), high-active

2.2. Processor Top Entity - 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.

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

More information can be found in the user guides' section Application-Specific Processor Configuration.

Privileged software can determine the actual CPU and processor configuration via the `misa` and `mxisa` CSRs (CPU) and the SYSINFO (processor) memory-mapped registers.

Run a quick simulation using the provided simulation/GHDL scripts (https://stnolting.github.io/neorv32/ug/#_hello_world) to verify the configuration of the processor generics is valid.

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

Not all configuration combinations are valid. The processor RTL code provides sanity checks to inform the user during synthesis/simulation if an invalid combination has been detected.

Generic Description

The description of each generic provides the following summary:

Table 4. Generic description

Generic name	*type*	*default value*
Description		

2.2.1. General

See section System Configuration Information Memory (SYSINFO) for more information.

CLOCK_FREQUENCY

CLOCK_FREQUENCY	*natural*	*none*

The clock frequency of the processor's `clk_i` input port in Hertz (Hz). This value can be retrieved by software from the SYSINFO module.

INT_BOOTLOADER_EN

INT_BOOTLOADER_EN	*boolean*	false

Implement the processor-internal boot ROM, pre-initialized with the default 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 section Boot Configuration for more information.

HW_THREAD_ID

HW_THREAD_ID	*natural*	0

The hart ID of the CPU. Software can retrieve this value from the `mhartid` CSR. Note that hart IDs must be unique within a system.

ON_CHIP_DEBUGGER_EN

ON_CHIP_DEBUGGER_EN	*boolean*	false

Implement the on-chip debugger (OCD) and the CPU debug mode. See chapter On-Chip Debugger (OCD) for more information.

2.2.2. RISC-V CPU Extensions

Discovering ISA Extensions

See section Instruction Sets and Extensions for more information. The configuration of the RISC-V *main* ISA extensions (like `M`) can be determined via the `misa` CSR. The configuration of ISA *sub-extensions* (like `Zicsr`) and *tuning options* can be determined via the NEORV32-specific `mxisa` CSR.

CPU_EXTENSION_RISCV_A

CPU_EXTENSION_RISCV_A	*boolean*	false

Implement atomic memory access operations when *true*. See section `A` - Atomic Memory Access.

CPU_EXTENSION_RISCV_B

CPU_EXTENSION_RISCV_B	*boolean*	false
Implement the `B` bit-manipulation sub-extension when *true*. See section `B` - Bit-Manipulation Operations for more information.		

CPU_EXTENSION_RISCV_C

CPU_EXTENSION_RISCV_C	*boolean*	false
Implement compressed instructions (16-bit) when *true*. Compressed instructions can reduce program code size by approx. 30%. See section `C` - Compressed Instructions.		

CPU_EXTENSION_RISCV_E

CPU_EXTENSION_RISCV_E	*boolean*	false
Implement the embedded CPU extension (only implement the first 16 data registers) when *true*. This reduces embedded memory requirements for the register file. See section `E` - Embedded CPU for more information. Note that this RISC-V extensions requires a different application binary interface (ABI).		

CPU_EXTENSION_RISCV_M

CPU_EXTENSION_RISCV_M	*boolean*	false
Implement hardware accelerators for integer multiplication and division instructions when *true*. If this extensions is not enabled, multiplication and division operations (*not* instructions) will be computed entirely in software. If only a hardware multiplier is required use the *CPU_EXTENSION_RISCV_Zmmul* extension. Multiplication can also be mapped to DSP slices via the *FAST_MUL_EN* generic. See section `M` - Integer Multiplication and Division for more information.		

CPU_EXTENSION_RISCV_U

CPU_EXTENSION_RISCV_U	*boolean*	false
Implement less-privileged user mode when *true*. See section `U` - Less-Privileged User Mode for more information.		

CPU_EXTENSION_RISCV_Zfinx

CPU_EXTENSION_RISCV_Zfinx	*boolean*	false
Implement the 32-bit single-precision floating-point extension (using integer registers) when *true*. See section `Zfinx` Single-Precision Floating-Point Operations for more information.		

CPU_EXTENSION_RISCV_Zicsr

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 privileged architecture / trap system will be excluded from synthesis. Hence, no interrupts, no exceptions and no machine information will be available. See section `Zicsr` Control and Status Register Access / Privileged Architecture for more information.		

CPU_EXTENSION_RISCV_Zicntr

CPU_EXTENSION_RISCV_Zicntr	*boolean*	true
Implement the basic CPU (Machine) Counter and Timer CSRs (`time[h]`, `[m]cycle[h]`, `[m]instret[h]`) when true. See section `Zicntr` CPU Base Counters for more information.		

CPU_EXTENSION_RISCV_Zihpm

CPU_EXTENSION_RISCV_Zihpm	*boolean*	false
Implement hardware performance monitor CSRs when true. See section `Zihpm` Hardware Performance Monitors for more information.		

CPU_EXTENSION_RISCV_Zifencei

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 instruction cache and CPU prefetch buffer flushes). See section `Zifencei` Instruction Stream Synchronization for more information.		

CPU_EXTENSION_RISCV_Zmmul

CPU_EXTENSION_RISCV_Zmmul	*boolean*	false
Implement integer multiplication-only instructions when *true*. This is a sub-extension of the `M` extension, which cannot be used together with the `M` extension. See section `Zmmul` - Integer Multiplication for more information.		

CPU_EXTENSION_RISCV_Zxcfu

CPU_EXTENSION_RISCV_Zxcfu	*boolean*	false
NEORV32-specific "custom RISC-V" ISA extensions: Implement the Custom Functions Unit (CFU) for user-defined custom instruction when *true*. See section `Zxcfu` Custom Instructions Extension (CFU) for more information.		

2.2.3. Tuning Options

These are generics to fine-tune certain ISA extensions and CPU features. See section Instruction Sets and Extensions for more information.

FAST_MUL_EN

FAST_MUL_EN	*boolean*	false
When this generic is enabled, the multiplier of the `M` extension is implemented using DSPs blocks instead of an iterative bit-serial approach. Performance will be increased and LUT utilization will be reduced at the cost of DSP slice utilization. This generic is only relevant when a hardware multiplier CPU extension is enabled (*CPU_EXTENSION_RISCV_M* or *CPU_EXTENSION_RISCV_Zmmul* is *true*). **Note that the multipliers of the `Zfinx` Single-Precision Floating-Point Operations extension are always mapped to DSP block (if available).**		

FAST_SHIFT_EN

FAST_SHIFT_EN	*boolean*	false
If this generic is set *true* the shifter unit of the CPU's ALU is implemented as fast barrel shifter (requiring more hardware resources but completing within two clock cycles). If it is set *false*, the CPU uses a serial shifter that only performs a single bit shift per cycle (requiring less hardware resources, but requires up to 32 clock cycles to complete - depending on shift amount). **Note that this option also implements barrel shifters for *all* shift-related operations of the `B` - Bit-Manipulation Operations extension.**		

CPU_CNT_WIDTH

CPU_CNT_WIDTH	*natural*	64
This generic configures the total size of the CPU's `[m]cycle` and `[m]instret` CSRs (low word + high word). The maximum value is 64, the minimum value is 0. See section (Machine) Counter and Timer CSRs for more information. This generic is only relevant if the `Zicntr` ISa extension is enabled (*CPU_EXTENSION_RISCV_Zicntr*). Note: configurations with *CPU_CNT_WIDTH* less than 64 bits do not comply to the RISC-V specs.		

CPU_IPB_ENTRIES

CPU_IPB_ENTRIES	*natural*	2
This generic configures the number of entries in the CPU's instruction prefetch buffer (a FIFO). The value has to be a power of two and has to be greater than or equal to *two* (>= 2). Long linear sequences of code can benefit from an increased IPB size.		

2.2.4. Physical Memory Protection (PMP)

See section `PMP` Physical Memory Protection for more information.

PMP_NUM_REGIONS

PMP_NUM_REGIONS	*natural*	0

Total number of implemented protection regions (0..16). If this generics is zero no physical memory protection logic will be implemented at all.		

PMP_MIN_GRANULARITY

PMP_MIN_GRANULARITY	*natural*	4
Minimal region granularity in bytes. Has to be a power of two and has to be at least 4 bytes. A larger granularity will reduce hardware utilization and impact on critical path but will also reduce the minimal region size.		

2.2.5. Hardware Performance Monitors (HPM)

These generics allow to customize the `Zihpm` ISA extension. Note that the following generics are ignored if the *CPU_EXTENSION_RISCV_Zihpm* generic is *false*. See section **`Zihpm`** Hardware Performance Monitors for more information.

HPM_NUM_CNTS

HPM_NUM_CNTS	*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

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 0. If the size is less than 64-bit, the unused MSB-aligned counter bits are hardwired to zero.		

2.2.6. Internal Instruction Memory

See sections Address Space and Instruction Memory (IMEM) for more information.

MEM_INT_IMEM_EN

MEM_INT_IMEM_EN	*boolean*	false
Implement processor internal instruction memory (IMEM) when *true*.		

MEM_INT_IMEM_SIZE

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*.		

2.2.7. Internal Data Memory

See sections Address Space and Data Memory (DMEM) for more information.

MEM_INT_DMEM_EN

MEM_INT_DMEM_EN	*boolean*	false
Implement processor internal data memory (DMEM) when *true*.		

MEM_INT_DMEM_SIZE

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*.		

2.2.8. Internal Cache Memory

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

ICACHE_EN

ICACHE_EN	*boolean*	false
Implement processor internal instruction cache when *true*. Note: if the setup only uses processor-internal data and instruction memories there is not point of implementing the i-cache.		

ICACHE_NUM_BLOCKS

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_EN* is false.		

ICACHE_BLOCK_SIZE

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

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` = 2-way set-associative. Has no effect when *ICACHE_EN* is *false*.		

2.2.9. External Memory Interface

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

MEM_EXT_EN

MEM_EXT_EN	*boolean*	false
Implement external bus interface (WISHBONE) when *true*.		

MEM_EXT_TIMEOUT

MEM_EXT_TIMEOUT	*natural*	255
Clock cycles after which a pending external bus access will auto-terminate and raise a bus fault exception. If set to zero, there will be no auto-timeout and no bus fault exception (might permanently stall system!).		

MEM_EXT_PIPE_MODE

MEM_EXT_PIPE_MODE	*boolean*	false
Use *standard* ("classic") Wishbone protocol for external bus when *false*. Use *pipelined* Wishbone protocol when *true*.		

MEM_EXT_BIG_ENDIAN

MEM_EXT_BIG_ENDIAN	*boolean*	false
Use BIG endian interface for external bus when *true*. Use little endian interface when *false*.		

MEM_EXT_ASYNC_RX

MEM_EXT_ASYNC_RX	*boolen*	false
By default, *MEM_EXT_ASYNC_RX* = *false* implements a registered read-back path (RX) for incoming data in the bus interface in order to shorten the critical path. By setting *MEM_EXT_ASYNC_RX* = *true* an *asynchronous* ("direct") read-back path is implemented reducing access latency by one cycle but eventually increasing the critical path.		

2.2.10. Stream Link Interface

See section Stream Link Interface (SLINK) for more information.

SLINK_NUM_TX

SLINK_NUM_TX	*natural*	0
Number of TX (send) links to implement. Valid values are 0..8.		

SLINK_NUM_RX

SLINK_NUM_RX	*natural*	0
Number of RX (receive) links to implement. Valid values are 0..8.		

SLINK_TX_FIFO

SLINK_TX_FIFO	*natural*	1
Internal FIFO depth for *all* implemented TX links. Valid values are 1..32k and have to be a power of two.		

SLINK_RX_FIFO

SLINK_RX_FIFO	*natural*	1
Internal FIFO depth for *all* implemented RX links. Valid values are 1..32k and have to be a power of two.		

2.2.11. External Interrupt Controller

See section External Interrupt Controller (XIRQ) for more information.

XIRQ_NUM_CH

XIRQ_NUM_CH	*natural*	0
Number of external interrupt channels o implement. Valid values are 0..32.		

XIRQ_TRIGGER_TYPE

XIRQ_TRIGGER_TYPE	*std_ulogic_vector(31 downto 0)*	0xFFFFFFFF
Interrupt trigger type configuration (one bit for each IRQ channel): `0` = level-triggered, '1' = edge triggered. *XIRQ_TRIGGER_POLARITY* generic is used to specify the actual level (high/low) or edge (falling/rising).		

XIRQ_TRIGGER_POLARITY

XIRQ_TRIGGER_POLARITY	*std_ulogic_vector(31 downto 0)*	0xFFFFFFFF
Interrupt trigger polarity configuration (one bit for each IRQ channel): `0` = low-level/falling-edge, '1' = high-level/rising-edge. *XIRQ_TRIGGER_TYPE* generic is used to specify the actual type (level or edge).		

2.2.12. Processor Peripheral/IO Modules

See section Processor-Internal Modules for more information.

IO_GPIO_EN

IO_GPIO_EN	*boolean*	false
Implement general purpose input/output port unit (GPIO) when *true*. See section General Purpose Input and Output Port (GPIO) for more information.		

IO_MTIME_EN

IO_MTIME_EN	*boolean*	false
Implement machine system timer (MTIME) when *true*. See section Machine System Timer (MTIME) for more information.		

IO_UART0_EN

IO_UART0_EN	*boolean*	false
Implement primary universal asynchronous receiver/transmitter (UART0) when *true*. See section Primary Universal Asynchronous Receiver and Transmitter (UART0) for more information.		

IO_UART0_RX_FIFO

IO_UART0_RX_FIFO	*natural*	1
UART0 receiver FIFO depth, has to be a power of two, minimum value is 1 (implementing simple double-buffering). See section Primary Universal Asynchronous Receiver and Transmitter (UART0) for more information.		

IO_UART0_TX_FIFO

IO_UART0_TX_FIFO	*natural*	1
UART0 transmitter FIFO depth, has to be a power of two, minimum value is 1 (implementing simple double-buffering). See section Primary Universal Asynchronous Receiver and Transmitter (UART0) for more information.		

IO_UART1_EN

IO_UART1_EN	*boolean*	false
Implement secondary universal asynchronous receiver/transmitter (UART1) when *true*. See section Secondary Universal Asynchronous Receiver and Transmitter (UART1) for more information.		

IO_UART1_RX_FIFO

IO_UART1_RX_FIFO	*natural*	1

UART1 receiver FIFO depth, has to be a power of two, minimum value is 1 (implementing simple double-buffering). See section Primary Universal Asynchronous Receiver and Transmitter (UART0) for more information.

IO_UART1_TX_FIFO

IO_UART1_TX_FIFO	*natural*	1
UART1 transmitter FIFO depth, has to be a power of two, minimum value is 1 (implementing simple double-buffering). See section Primary Universal Asynchronous Receiver and Transmitter (UART0) for more information.		

IO_SPI_EN

IO_SPI_EN	*boolean*	false
Implement serial peripheral interface controller (SPI) when *true*. See section Serial Peripheral Interface Controller (SPI) for more information.		

IO_TWI_EN

IO_TWI_EN	*boolean*	false
Implement two-wire interface controller (TWI) when *true*. See section Two-Wire Serial Interface Controller (TWI) for more information.		

IO_PWM_NUM_CH

IO_PWM_NUM_CH	*natural*	0
Number of pulse-width modulation (PWM) channels (0..60) to implement. The PWM controller is *not* implemented if zero. See section Pulse-Width Modulation Controller (PWM) for more information.		

IO_WDT_EN

IO_WDT_EN	*boolean*	false
Implement watchdog timer (WDT) when *true*. See section Watchdog Timer (WDT) for more information.		

IO_TRNG_EN

IO_TRNG_EN	*boolean*	false
Implement true-random number generator (TRNG) when *true*. See section True Random-Number Generator (TRNG) for more information.		

IO_CFS_EN

IO_CFS_EN	*boolean*	false
Implement custom functions subsystem (CFS) when *true*. See section Custom Functions Subsystem (CFS) for more information.		

IO_CFS_CONFIG

IO_CFS_CONFIG	*std_ulogic_vector(31 downto 0)*	0x"00000000"
This is a "conduit" generic that can be used to pass user-defined CFS implementation flags to the custom functions subsystem entity. See section Custom Functions Subsystem (CFS) for more information.		

IO_CFS_IN_SIZE

IO_CFS_IN_SIZE	*positive*	32
Defines the size of the CFS input signal conduit (`cfs_in_i`). See section Custom Functions Subsystem (CFS) for more information.		

IO_CFS_OUT_SIZE

IO_CFS_OUT_SIZE	*positive*	32
Defines the size of the CFS output signal conduit (`cfs_out_o`). See section Custom Functions Subsystem (CFS) for more information.		

IO_NEOLED_EN

IO_NEOLED_EN	*boolean*	false
Implement smart LED interface (WS2812 / NeoPixel™-compatible) (NEOLED) when *true*. See section Smart LED Interface (NEOLED) for more information.		

IO_NEOLED_TX_FIFO

IO_NEOLED_TX_FIFO	*natural*	1
TX FIFO depth of the the NEOLED module. Minimal value is 1, maximal value is 32k, has to be a power of two. See section Smart LED Interface (NEOLED) for more information.		

IO_GPTMR_EN

IO_GPTMR_EN	*boolean*	false
Implement general purpose 32-bit timer (GPTMR) when *true*. See section General Purpose Timer (GPTMR) for more information.		

IO_XIP_EN

IO_XIP_EN	*boolean*	false
Implement the execute in place module (XIP) when *true*. See section Execute In Place Module (XIP) for more information.		

2.3. Processor Interrupts

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

2.3.1. RISC-V Standard Interrupts

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

Top signal	Width	Description
`mtime_irq_i`	1	Machine timer interrupt from *processor-external* MTIME unit. This IRQ is only available if the processor-internal MTIME unit is not used (*IO_MTIME_EN* = false).
`msw_irq_i`	1	Machine software interrupt. This interrupt is used for inter-processor interrupts in multi-core systems. However, it can also be used for any custom purpose.
`mext_irq_i`	1	Machine external interrupt. This interrupt is used for any processor-external interrupt source (like a platform interrupt controller).

Trigger type

The fast interrupt request channels become pending after being triggering by **a rising edge**. A pending FIRQ has to be explicitly cleared by writing zero to the according `mip` CSR bit.

2.3.2. Platform External Interrupts

Top signal	Width	Description
`xirq_i`	up to 32	External platform interrupts (user-defined).

The processor provides an optional interrupt controller for up to 32 user-defined external interrupts (see section External Interrupt Controller (XIRQ)). These external IRQs are mapped to a *single* CPU fast interrupt request so a software handler is required to differentiate / prioritize these interrupts.

Trigger type

The trigger for these interrupt can be defined via generics. See section External Interrupt Controller (XIRQ) for more information. Depending on the trigger type, users can implement custom acknowledge mechanisms. All *external interrupts* are mapped to a single processor-internal *fast interrupt request* (see below).

2.3.3. NEORV32-Specific Fast Interrupt Requests

As part of the custom/NEORV32-specific CPU extensions, the CPU features 16 fast interrupt request signals (`FIRQ0` - `FIRQ15`). These are reserved for *processor-internal* modules only (for example for the communication interfaces to signal "available incoming data" or "ready to send new data").

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

Table 5. NEORV32 fast interrupt channel mapping

Channel	Source	Description
0	WDT	watchdog timeout interrupt
1	CFS	custom functions subsystem (CFS) interrupt (user-defined)
2	UART0	UART0 data received interrupt (RX complete)
3	UART0	UART0 sending done interrupt (TX complete)
4	UART1	UART1 data received interrupt (RX complete)
5	UART1	UART1 sending done interrupt (TX complete)
6	SPI	SPI transmission done interrupt
7	TWI	TWI transmission done interrupt
8	XIRQ	External interrupt controller interrupt
9	NEOLED	NEOLED TX buffer interrupt
10	SLINK	RX data buffer interrupt
11	SLINK	TX data buffer interrupt
12	GPTMR	General purpose timer interrupt
13:15	-	*reserved*, will never fire

Trigger type

The fast interrupt request channels become pending after being triggering by **a rising edge**. A pending FIRQ has to be explicitly cleared by writing zero to the according `mip` CSR bit.

2.4. Address Space

The NEORV32 Processor provides a 32-bit / 4GB (physical) address space By default, this address space is divided into five main regions:

1. **Instruction address space** - memory address space for instructions (=code) and constants. A configurable section of this address space is used by the internal/external *instruction memory* (*MEM_INT_IMEM_SIZE* for the internal IMEM).
2. **Data address space** - memory address space for application runtime data (heap, stack, etc.). A configurable section of this address space is used by the internal/external *data memory* (*MEM_INT_DMEM_SIZE* for the internal DMEM).
3. **Bootloader address space**. A *fixed* section of this address space is used by the internal *bootloader memory* (BOOTLDROM).
4. **On-Chip Debugger address space**. This *fixed* section is entirely used by the processor's On-Chip Debugger (OCD).
5. **IO/peripheral address space**. Also a *fixed* section used for the processor-internal memory-mapped IO/peripheral devices (e.g., UART).

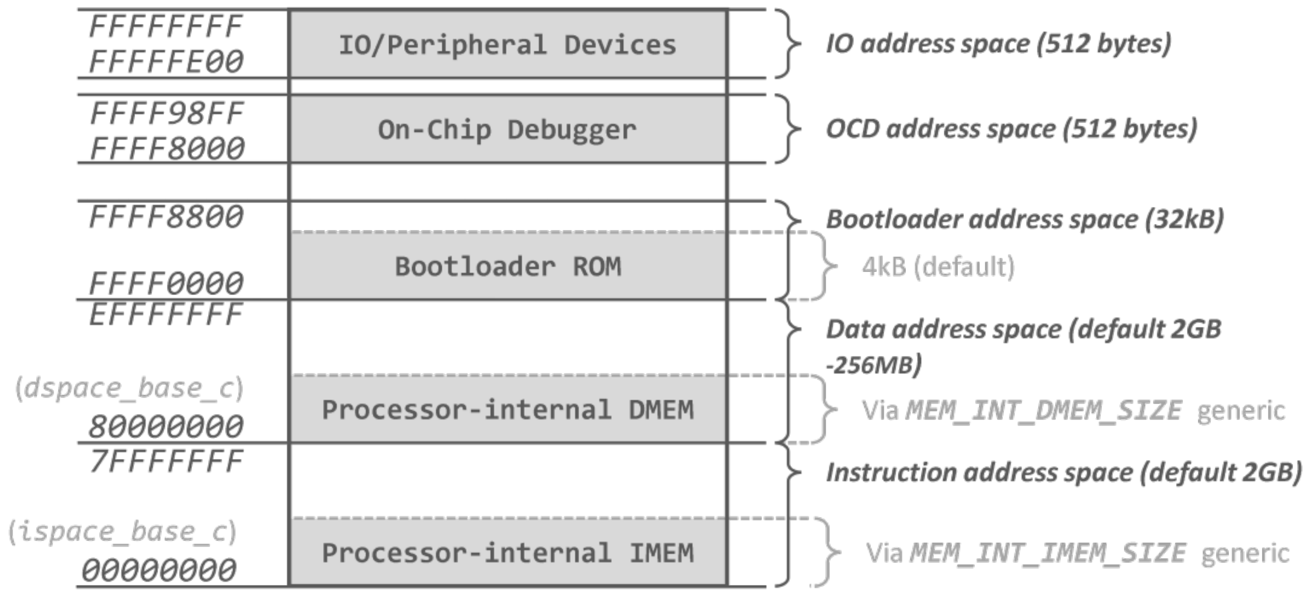

FFFFFFFF
FFFFFE00
IO/Peripheral Devices
IO address space (512 bytes)
FFFF98FF
FFFF8000
On-Chip Debugger
OCD address space (512 bytes)
FFFF8800
FFFF0000
Bootloader ROM
Bootloader address space (32kB)
4kB (default)
EFFFFFFF
(dspace_base_c)
80000000
Processor-internal DMEM
Data address space (default 2GB -256MB)
Via MEM_INT_DMEM_SIZE generic
7FFFFFFF
(ispace_base_c)
00000000
Processor-internal IMEM
Instruction address space (default 2GB)
Via MEM_INT_IMEM_SIZE generic

Figure 1. NEORV32 processor - address space (default configuration)

RAM Layout - Usage of the Data Address Space

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

2.4.1. CPU Data and Instruction Access

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

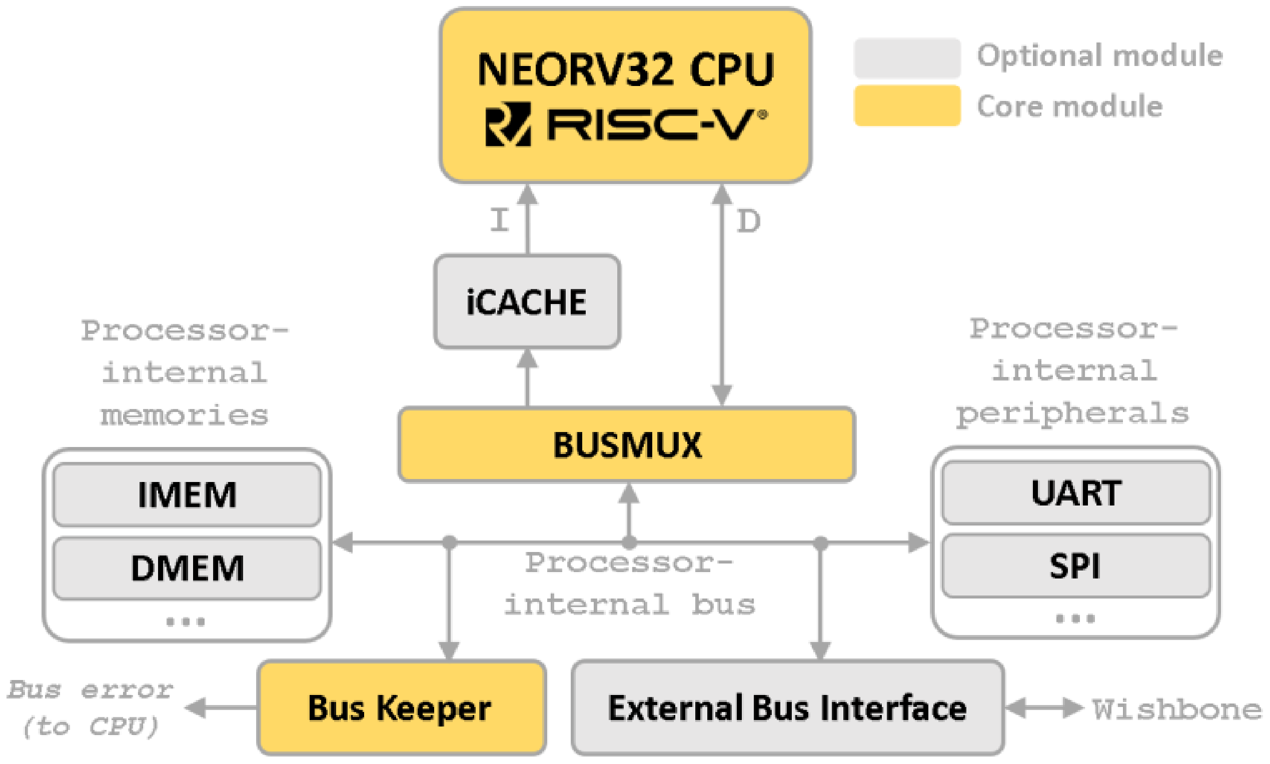


Figure 2. Processor-internal bus architecture

The internal processor bus might appear as bottleneck. In order to reduce traffic jam on this bus (when instruction fetch and data interface access the bus at the same time) the instruction fetch of the CPU is equipped with a prefetch buffer. 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 the *MTIME* configuration corrupting this interrupt. This kind of security issues can be compensated using the PMP system (see Machine Physical Memory Protection CSRs).

2.4.2. 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. Make sure that both base addresses are *aligned* to a 4-byte boundary.

The base address of the internal bootloader (at *0xFFFF0000*) and the internal IO region (at *0xFFFFFE00*) for peripheral devices are also defined in the package and are fixed. These address regions cannot not be used for other applications - even if the bootloader or all IO devices are not implemented - without modifying the core's hardware sources.

2.4.3. Memory Configuration

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

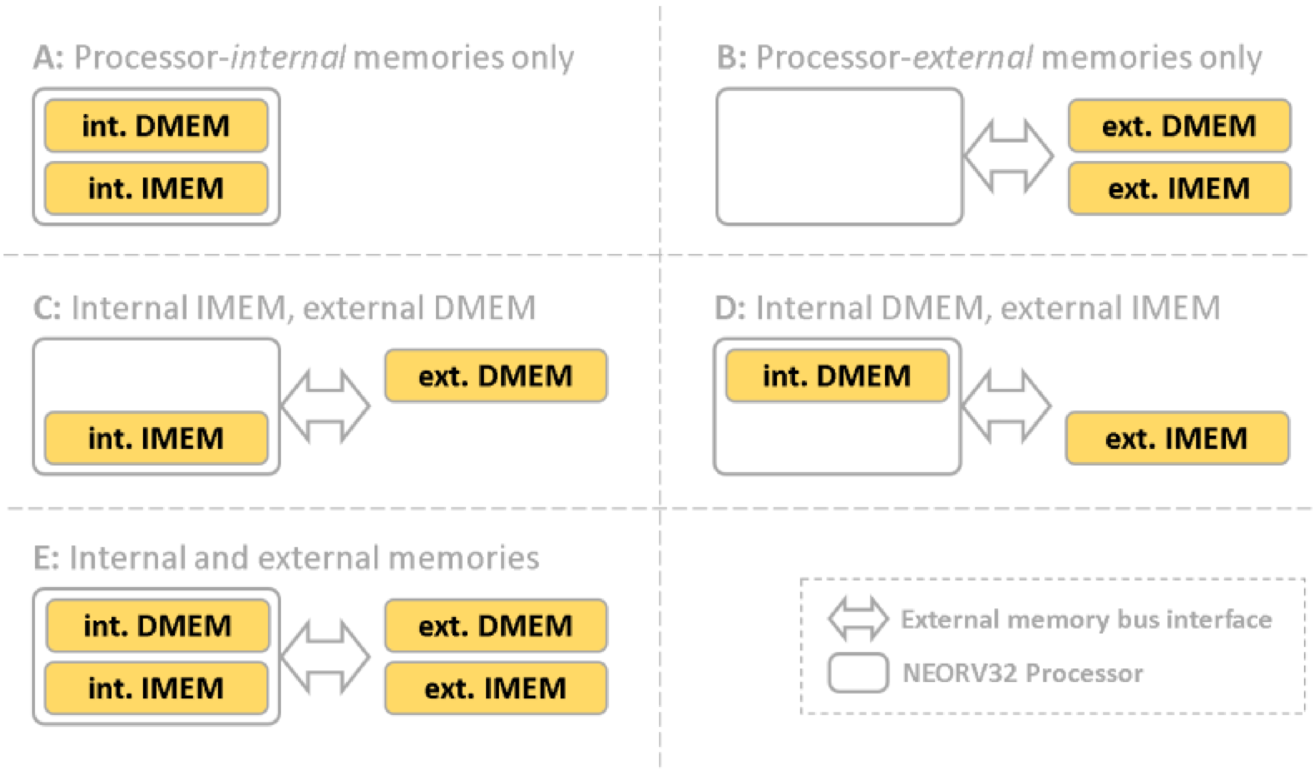

A: Processor-internal memories only
int. DMEM
int. IMEM
B: Processor-external memories only
ext. DMEM
ext. IMEM
C: Internal IMEM, external DMEM
ext. DMEM
int. IMEM
D: Internal DMEM, external IMEM
int. DMEM
ext. IMEM
E: Internal and external memories
int. DMEM
int. IMEM
ext. DMEM
ext. IMEM
External memory bus interface
NEORV32 Processor

Figure 3. Exemplary memory configurations

Internal Memories

The processor-internal memories (Instruction Memory (IMEM) and Data Memory (DMEM)) are enabled (=implemented) via the *MEM_INT_IMEM_EN* and *MEM_INT_DMEM_EN* generics. Their sizes are configures via the according *MEM_INT_IMEM_SIZE* and *MEM_INT_DMEM_SIZE* generics.

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.

The default processor setup uses only *internal* memories.

If the IMEM (internal or external) is less than the (default) maximum size (2GB), there is a "dead address space" between it and the DMEM. This provides an additional safety feature since data corrupting scenarios like stack overflow cannot directly corrupt the content of the IMEM: any access to the "dead address space" in between will raise an exception that can be caught by the runtime environment.

External Memories

If external memories (or further IP modules) shall be connected via the *processor's external bus interface*, the interface has to be enabled via *MEM_EXT_EN* generic (=*true*). More information regarding this interface can be found in section Processor-External Memory Interface (WISHBONE) (AXI4-Lite).

Any CPU access (data or instructions), which does not fulfill *at least one* of the following conditions, is forwarded via the processor's bus interface to external components:

- access to the processor-internal IMEM and processor-internal IMEM is implemented
- access to the processor-internal DMEM and processor-internal DMEM is implemented
- access to the bootloader ROM and beyond → addresses >= *BOOTROM_BASE* (default 0xFFFF0000) will never be forwarded to the external memory interface

If the Execute In Place module (XIP) is implemented accesses map to this module are not forwarded to the external memory interface. See section Execute In Place Module (XIP) for more information.

If no (or not all) processor-internal memories are implemented, the according base addresses are mapped to external memories. For example, if the processor-internal IMEM is not implemented (*MEM_INT_IMEM_EN* = *false*), the processor will forward any access to the instruction address space (starting at `ispace_base_c`) via the external bus interface to the external memory system.

If the external interface is deactivated, any access exceeding the internal memory address space (instruction, data, bootloader) or the internal peripheral address space will trigger a bus access fault exception.

2.4.4. Boot Configuration

Due to the flexible memory configuration concept, the NEORV32 Processor provides several different boot concepts. The figure below shows the exemplary concepts for the two most common boot scenarios.

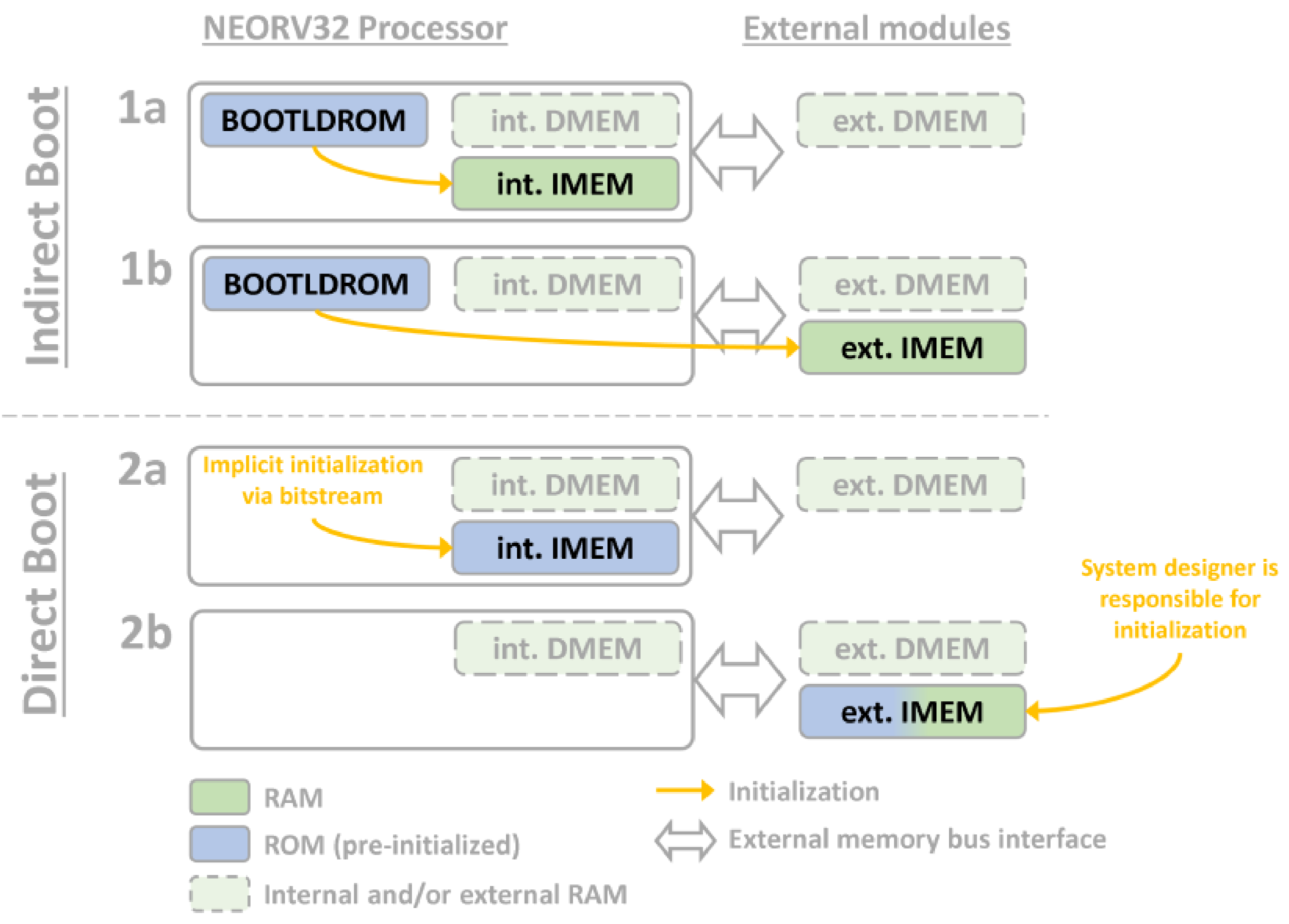


Figure 4. NEORV32 boot configurations

The configuration of internal or external data memory (DMEM; *MEM_INT_DMEM_EN* = *true* / *false*) is not further relevant for the boot configuration itself. Hence, it is not further illustrated here.

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

Please note that the provided boot scenarios are just exemplary setups that (should) fit most common requirements. Much more sophisticated boot scenarios are possible by combining internal and external memories. For example, the default internal bootloader could be used as first-level bootloader that loads (from extern SPI flash) a second-level bootloader that is placed and execute in internal IMEM. This second-level bootloader could then fetch the actual application and store it to external *data* memory and transfers CPU control to that.

Indirect Boot

The *indirect* boot scenarios **1a** and **1b** use the processor-internal Bootloader. This boot setup is enabled by setting the *INT_BOOTLOADER_EN* generic to *true*, which will implement the processor-internal Bootloader ROM (BOOTROM). This read-only memory is pre-initialized during synthesis with the default bootloader firmware. The bootloader provides several options to upload an executable (via UART or from external SPI flash) and copies it to the beginning of the *instruction address space* so the CPU can execute it.

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

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

Direct Boot

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

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

In contrast, boot scenario **2b** uses a processor-external IMEM (*MEM_INT_IMEM_EN* = *false*). In this scenario the system designer is responsible for providing an initialized external memory that contains the actual application to be executed. If the external is not already initialized after reset, a simple ROM containing a "polling loop" can be implemented that is exited as soon as the application logic has finished initializing the memory with the acutal application code.

2.5. 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 (`crt0.S`) will perform 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), by the internal watchdog timer (if implemented) or by the on-chip debugger. The external reset signal `rstn_i` is extended to be active for at least 4 cycles when triggered.

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:	`0b000`	`0b001`	`0b010`	`0b011`	`0b100`	`0b101`	`0b110`	`0b111`
Resulting clock:	*f/2*	*f/4*	*f/8*	*f/64*	*f/128*	*f/1024*	*f/2048*	*f/4096*

Peripheral / IO Devices

The processor-internal peripheral/IO devices are located at the end of the 32-bit address space at base address *0xFFFFFE00*. A region of 512 bytes is reserved for 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.

Address Space Mapping

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

Full-Word Write Accesses Only

All peripheral/IO devices can only be written in full-word mode (i.e. 32-bit). Byte or half-word (8/16-bit) 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.

Unimplemented Modules

When accessing an IO device that hast not been implemented (via the according generic), a load/store access fault exception is triggered.

Hardware Reset

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`.

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.

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

Interrupts of Processor-Internal Modules

Most peripheral/IO devices provide some kind of interrupt (for example to signal available incoming data). These interrupts are entirely mapped to the CPU's Custom Fast Interrupt Request Lines. Note that all these interrupt lines are high-active and are permanently triggered until the IRQ-causing condition is resolved.

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. C-language code can directly interact with these registers via pre-defined `struct`. Each IO/peripheral module provides a unique `struct`. All accessible interface registers of this module are defined as members of this `struct`. The pre-defined `struct` are defined int the main processor core library include file `sw/lib/include/neorv32.h`.

The naming scheme of these low-level hardware access structs is `NEORV32_<module_name>.<register_name>`.

Listing 1. Low-level hardware access example in C using the pre-defined `struct`

```
// Read from SYSINFO "CLK" register
uint32_t temp = NEORV32_SYSINFO.CLK;
```

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
- `-/w` these registers / bits are write-only; they auto-clear in the next cycle and are always read as zero

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.

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.

2.5.1. Instruction Memory (IMEM)

Hardware source file(s):	neorv32_imem.entity.vhd	entity-only definition
	mem/neorv32_imem.default.vhd	default *platform-agnostic* memory architecture
Software driver file(s):	none	*implicitly used*
Top entity port:	none	
Configuration generics:	*MEM_INT_IMEM_EN*	implement processor-internal IMEM when *true*
	MEM_INT_IMEM_SIZE	IMEM size in bytes
	INT_BOOTLOADER_EN	use internal bootloader when *true* (implements IMEM as *uninitialized* RAM, otherwise the IMEM is implemented an *pre-intialized* ROM)
CPU interrupts:	none	

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

By default the IMEM is implemented as true RAM so the content can be modified during run time. This is required when using a bootloader that can update the content of the IMEM at any time. If you do not need the bootloader anymore - since your application development has completed and you want the program to permanently reside in the internal instruction memory - the IMEM is automatically implemented as *pre-intialized* ROM when the processor-internal bootloader is disabled (*INT_BOOTLOADER_EN* = *false*).

When the IMEM is implemented as ROM, it will be initialized during synthesis (actually, by the bitstream) with the actual application program image. The compiler toolchain will 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 (default), the memory will **not be initialized at all**.

The actual IMEM is split into two design files: a plain entity definition (`neorv32_imem.entity.vhd`) and the actual architecture definition (`mem/neorv32_imem.default.vhd`). This **default architecture** provides a *generic* and *platform independent* memory design that (should) infers embedded memory block. You can replace/modify the architecture source file in order to use platform-specific features (like advanced memory resources) or to improve technology mapping and/or timing.

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

2.5.2. Data Memory (DMEM)

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

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

The actual DMEM is split into two design files: a plain entity definition (`neorv32_dmem.entity.vhd`) and the actual architecture definition (`mem/neorv32_dmem.default.vhd`). This **default architecture** provides a *generic* and *platform independent* memory design that (should) infers embedded memory block. You can replace/modify the architecture source file in order to use platform-specific features (like advanced memory resources) or to improve technology mapping and/or timing.

2.5.3. Bootloader ROM (BOOTROM)

Hardware source file(s):	neorv32_boot_rom.vhd	
Software driver file(s):	none	*implicitly used*
Top entity port:	none	
Configuration generics:	*INT_BOOTLOADER_EN*	implement processor-internal bootloader when *true*
CPU interrupts:	none	

The default `neorv32_boot_rom.vhd` HDL source file provides a *generic* memory design that infers embedded memory for *larger* memory configurations. You might need to replace/modify the source file in order to use platform-specific features (like advanced memory resources) or to improve technology mapping and/or timing.

This HDL modules provides a read-only memory that contain the executable code image of the bootloader. If the *INT_BOOTLOADER_EN* generic is *true* this module will be implemented and the CPU boot address is modified to directly execute the code from the bootloader ROM after reset.

The bootloader ROM is located at address `0xFFFF0000` and can occupy a address space of up to 32kB. The base address as well as the maximum address space size are fixed and cannot (should not!) be modified as this might address collision with other processor modules.

The bootloader memory is *read-only* and is automatically initialized with the bootloader executable image `rtl/core/neorv32_bootloader_image.vhd` during synthesis. The actual *physical* size of the ROM is also determined via synthesis and expanded to the next power of two. For example, if the bootloader code requires 10kB of storage, a ROM with 16kB will be generated. The maximum size must not exceed 32kB.

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

Bootloader - Software

See section Bootloader for more information regarding the actual bootloader software/executable itself.

Boot Configuration

See section Boot Configuration for more information regarding the processor's different boot scenarios.

2.5.4. Processor-Internal Instruction Cache (iCACHE)

Hardware source file(s):	neorv32_icache.vhd	
Software driver file(s):	none	*implicitly used*
Top entity port:	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 improve performance when using memories with high access latencies. The cache is directly connected to the CPU's instruction fetch interface and provides full-transparent buffering of instruction fetch accesses to the entire address space.

The cache is implemented if the *ICACHE_EN* generic is true. The size of the cache memory is defined via *ICACHE_BLOCK_SIZE* (the size of a single cache block/page/line in bytes; has to be a power of two and >= 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 greater than 1 the LRU replacement policy (least recently used) is used.

Cache Memory HDL

The default `neorv32_icache.vhd` HDL source file provides a *generic* memory design that infers embedded memory. You might need to replace/modify the source file in order to use platform-specific features (like advanced memory resources) or to improve technology mapping and/or timing. Also, 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.

Caching Internal Memories

The instruction cache is intended to accelerate instruction fetches from *processor-external* memories. Since all processor-internal memories provide an access latency of one cycle (by default), caching internal memories does not bring a relevant performance gain. However, it will slightly reduce traffic on the processor-internal bus.

Manual Cache Clear/Reload

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

Retrieve Cache Configuration from Software

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

Bus Access Fault Handling

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

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

Hardware source file(s):	neorv32_wishbone.vhd	
Software driver file(s):	none	*implicitly used*
Top entity port:	`wb_tag_o`	request tag output (3-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_lock_o`	exclusive access request (1-bit)
	`wb_ack_i`	acknowledge (1-bit)
	`wb_err_i`	bus error (1-bit)
	`fence_o`	an executed `fence` instruction
	`fencei_o`	an executed `fence.i` instruction
Configuration generics:	*MEM_EXT_EN*	enable external memory interface when *true*
	MEM_EXT_TIMEOUT	number of clock cycles after which an unacknowledged external bus access will auto-terminate (0 = disabled)
	MEM_EXT_PIPE_MODE	when *false* (default): classic/standard Wishbone protocol; when *true*: pipelined Wishbone protocol
	MEM_EXT_BIG_ENDIAN	byte-order (Endianness) of external memory interface; true=BIG, false=little (default)
	MEM_EXT_ASYNC_RX	use registered RX path when *false* (default); use async/direct RX path when *true*
CPU interrupts:	none	

The external memory interface provides a Wishbone b4-compatible on-chip bus interface. The bus interface is implemented 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.

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

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

If the Execute In Place module (XIP) is implemented accesses map to this module are not forwarded to the external memory interface. See section Execute In Place Module (XIP) for more information.

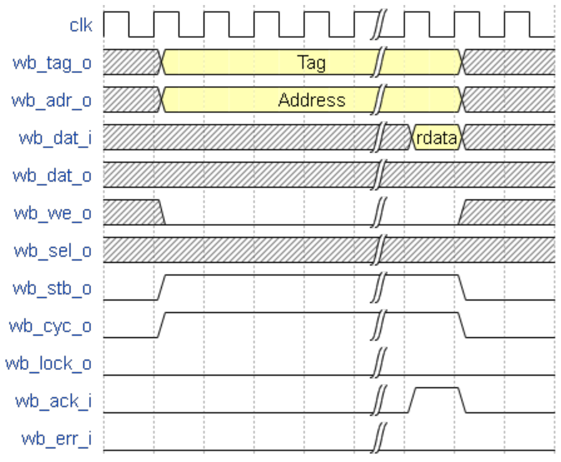
See section Address Space for more information.

Wishbone Bus Protocol

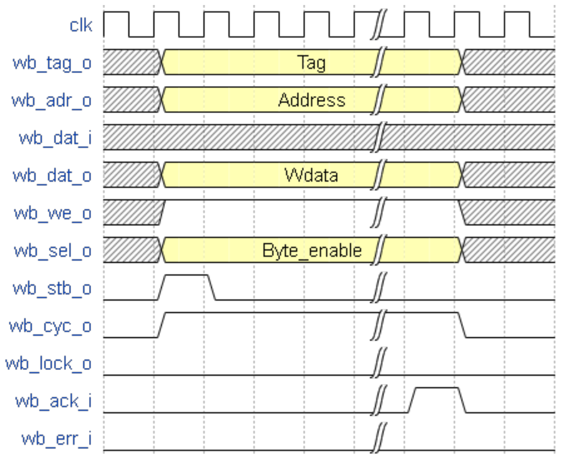
The external memory interface either uses the **standard** ("classic") Wishbone transaction protocol (default) or **pipelined** Wishbone transaction protocol. The transaction protocol is configured via the *MEM_EXT_PIPE_MODE* generic:

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

Table 6. Exemplary Wishbone bus accesses using "classic" and "pipelined" protocol



Classic Wishbone read access



Pipelined Wishbone write access

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

Bus Access

The NEORV32 Wishbone gateway does not support burst transfer yet, so there is always just one transfer in progress. Hence, the Wishbone `STALL` signal is not implemented. An accessed Wishbone device does not have to respond immediately to a bus request by sending an ACK. instead, there is a *time window* where the device has to acknowledge the transfer. This time window id configured by the *MEM_EXT_TIMEOUT* top generic that defines the maximum time (in clock cycles) a bus access can be pending before it is automatically terminated. If *MEM_EXT_TIMEOUT* is set to zero, the timeout disabled an a bus access can take an arbitrary number of cycles to complete.

When *MEM_EXT_TIMEOUT* is greater than zero, the Wishbone gateway starts an internal countdown whenever the CPU accesses a memory address via the external memory interface. If the accessed memory / device does not acknowledge (via `wb_ack_i`) or terminate (via `wb_err_i`) the transfer within *MEM_EXT_TIMEOUT* clock cycles, the bus access is automatically canceled setting `wb_cyc_o` low again and a CPU load/store/instruction fetch bus access fault exception is raised.

> Setting *MEM_EXT_TIMEOUT* to zero will permanently stall the CPU if the targeted Wishbone device never responds. Hence, *MEM_EXT_TIMEOUT* should be always set to a value greater than zero.
>
> This feature can be used as **safety guard** if the external memory system does not check for "address space holes". That means that accessing addresses, which do not belong to a certain memory or device, do not permanently stall the processor due to an unacknowledged/unterminated bus access. If the external memory system can guarantee to access **any** bus access (even it targets an unimplemented address) the timeout feature should be disabled (*MEM_EXT_TIMEOUT* = 0).

Wishbone Tag

The 3-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

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_lock_o` signal telling the bus interconnect to establish a reservation for the current accessed address (start of an exclusive access). This signal will stay asserted until another memory access instruction is executed (for example a `sc.w`).

The memory system has to make sure that no other entity can access the reservated address until `wb_lock_o` is released again. If this attempt fails, the memory system has to assert `wb_err_i` in order to indicate that the reservation was broken.

See section Bus Interface for the CPU bus interface protocol.

Endianness

The NEORV32 CPU and the Processor setup are **little-endian** architectures. To allow direct connection to a big-endian memory system the external bus interface provides an *Endianness configuration*. The Endianness (of the external memory interface) can be configured via the *MEM_EXT_BIG_ENDIAN* generic. By default, the external memory interface uses little-endian byte-order (like the rest of the processor / CPU).

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

Gateway Latency

By default, the Wishbone gateway introduces two additional latency cycles: processor-outgoing ("TX") and processor-incoming ("RX") signals are fully registered. Thus, any access from the CPU to a processor-external devices via Wishbone requires 2 additional clock cycles (at least; depending on device’s latency).

If the attached Wishbone network / peripheral already provides output registers or if the Wishbone network is not relevant for timing closure, the default buffering of incoming ("RX") data within the gateway can be disabled by implementing an "asynchronous" RX path. The configuration is done via the *MEM_EXT_ASYNC_RX* generic.

AXI4-Lite Connectivity

The AXI4-Lite wrapper (`rtl/system_integration/neorv32_SystemTop_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.

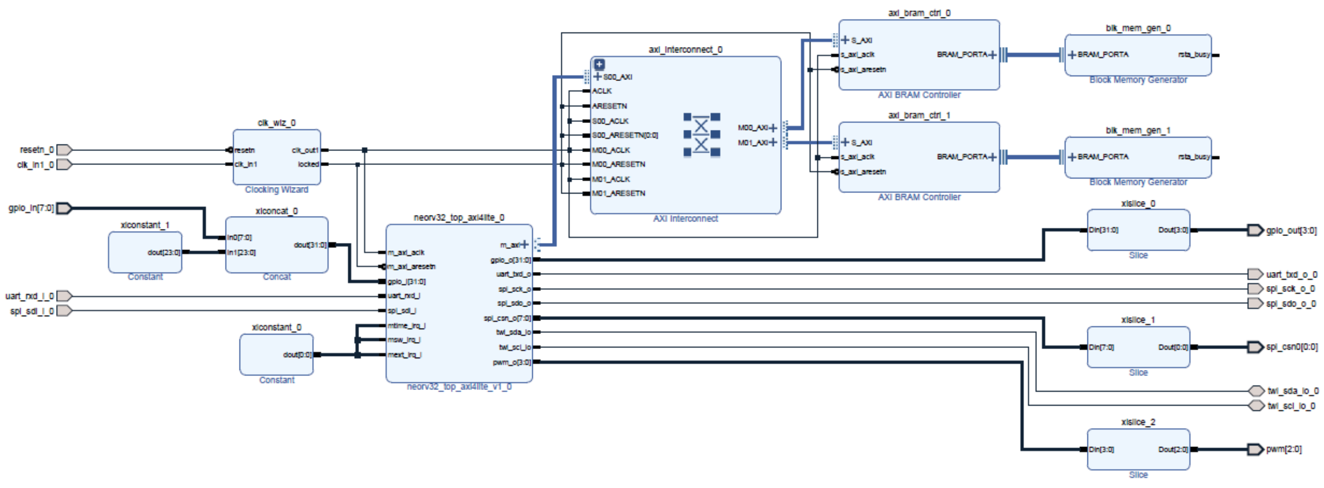


Figure 5. Example AXI SoC using Xilinx Vivado

Using the auto-termination timeout feature (*MEM_EXT_TIMEOUT* greater than zero) is **not AXI4 compliant** as the AXI protocol does not support canceling of bus transactions. Therefore, the NEORV32 top wrapper with AXI4-Lite interface (`rtl/system_integration/neorv32_SystemTop_axi4lite`) configures *MEM_EXT_TIMEOUT* = 0 by default.

2.5.6. Internal Bus Monitor (BUSKEEPER)

Hardware source file(s):	neorv32_buskeeper.vhd	
Software driver file(s):	none	
Top entity port:	none	
Configuration generics:	none	
Package constants:	`max_proc_int_response_time_c`	Access time window (#cycles)
CPU interrupts:	none	

Theory of Operation

The Bus Keeper is a fundamental component of the processor's internal bus system that ensures correct bus operations to maintain execution safety. The Bus Keeper monitors every single bus transactions that is intimated by the CPU. If an accessed device responds with an error condition or do not respond within a specific *access time window*, the according bus access fault exception is raised. The following exceptions can be raised by the Bus Keeper (see section NEORV32 Trap Listing for all CPU exceptions):

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

The **access time window**, in which an accessed device has to respond, is defined by the `max_proc_int_response_time_c` constant from the processor's VHDL package file (`rtl/neorv32_package.vhd`). The default value is **15 clock cycles**.

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

- `0` - "Device Error": The bus access exception was cause by the memory-mapped device that has been accessed (the device asserted it's `err_o`).
- `1` - "Timeout Error": The bus access exception was caused by the Bus Keeper because the accessed memory-mapped device did not respond within the access time window. Note that this error type can also be raised by the optional timeout feature of the Processor-External Memory Interface (WISHBONE) (AXI4-Lite)).

Bus access fault exceptions are also raised if a physical memory protection (PMP) rule is violated. In this case the *BUSKEEPER_ERR_FLAG* bit remains zero (since the error signal is not triggered by the BUSKEEPER but by the CPU's PMP logic).

Table 7. BUSKEEPER register map (`struct NEORV32_BUSKEEPER`)

Address	Name [C]	Bit(s), Name [C]	R/W	Function
`0xffffff7C`	`NEORV32_BUSKEEPER.CTRL`	`0` *BUSKEEPER_ERR_TYPE*	r/-	Bus error type, valid if *BUSKEEPER_ERR_FLAG*
		`31` *BUSKEEPER_ERR_FLAG*	r/c	Sticky error flag, clears after read or write access

2.5.7. Stream Link Interface (SLINK)

Hardware source file(s):	neorv32_slink.vhd	
Software driver file(s):	neorv32_slink.c	
	neorv32_slink.h	
Top entity port:	`slink_tx_dat_o`	TX link data (8x32-bit)
	`slink_tx_val_o`	TX link data valid (8-bit)
	`slink_tx_rdy_i`	TX link allowed to send (8-bit)
	`slink_rx_dat_i`	RX link data (8x32-bit)
	`slink_rx_val_i`	RX link data valid (8-bit)
	`slink_rx_rdy_o`	RX link ready to receive (8-bit)
Configuration generics:	*SLINK_NUM_TX*	Number of TX links to implement (0..8)
	SLINK_NUM_RX	Number of RX links to implement (0..8)
	SLINK_TX_FIFO	FIFO depth (1..32k) of TX links, has to be a power of two
	SLINK_RX_FIFO	FIFO depth (1..32k) of RX links, has to be a power of two
CPU interrupts:	fast IRQ channel 10	SLINK RX IRQ (see Processor Interrupts)
	fast IRQ channel 11	SLINK TX IRQ (see Processor Interrupts)

The SLINK component provides up to 8 independent RX (receiving) and TX (sending) links for transmitting stream data. The interface provides higher bandwidth (and less latency) than the external memory bus interface, which makes it ideally suited to couple custom stream processing units (like CORDIC, FFTs or cryptographic accelerators).

Each individual link provides an internal FIFO for data buffering. The FIFO depth is globally defined for all TX links via the *SLINK_TX_FIFO* generic and for all RX links via the *SLINK_RX_FIFO* generic. The FIFO depth has to be at least 1, which will implement a simple input/output register. The maximum value is limited to 32768 entries. Note that the FIFO depth has to be a power of two (for optimal logic mapping).

The actual number of implemented RX/TX links is configured by the *SLINK_NUM_RX* and *SLINK_NUM_TX* generics. The SLINK module will be synthesized only if at least one of these generics is greater than zero. All unimplemented links are internally terminated and their according output signals are pulled to low level.

The SLINK interface does not provide any additional tag signals (for example to define a "stream destination address" or to indicate the last data word of a "package"). Use a custom controller connected via the external memory bus interface or use some of the processor's GPIO ports to implement custom data tag signals.

Theory of Operation

The SLINK provides eight data registers (`DATA[i]`) to access the links (read accesses will access the RX links, write accesses will access the TX links), one control register (`CTRL`) and one status register (`STATUS`).

The SLINK is globally activated by setting the control register's enable bit *SLINK_CTRL_EN*. The actual data links are accessed by reading or writing the according link data registers `DATA[0]` to `DATA[7]`. For example, writing the `DATA[0]` will put the according data into the FIFO of TX link 0. Accordingly, reading from `DATA[0]` will return one data word from the FIFO of RX link 0.

The configuration (done via the SLINK generics) can be checked by software by evaluating bit fields in the control register. The *SLINK_CTRL_TX_FIFO_Sx* and *SLINK_CTRL_RX_FIFO_Sx* indicate the TX & RX FIFO sizes. The *SLINK_CTRL_TX_NUMx* and *SLINK_CTRL_RX_NUMx* bits represent the absolute number of implemented TX and RX links.

The status register shows the FIFO status flags of each RX and TX link. The *SLINK_CTRL_RXx_AVAIL* flags indicate that there is *at least* one data word in the according RX link's FIFO. The *SLINK_CTRL_TXx_FREE* flags indicate there is *at least* one free entry in the according TX link's FIFO. The *SLINK_STATUS_RXx_HALF* and *SLINK_STATUS_RXx_HALF* flags show if a certain FIFO's fill level has exceeded half of its capacity.

Blocking Link Access

When directly accessing the link data registers (without checking the according FIFO status flags) the access is as *blocking*. That means the CPU access will stall until the accessed link responds. For example, when reading RX link 0 (via `DATA[0]` register) the CPU will stall, if there is not data available in the according FIFO yet. The CPU access will complete as soon as RX link 0 receives new data.

Vice versa, writing data to TX link 0 (via `DATA[0]` register) will stall the CPU access until there is at least one free entry in the link's FIFO.

The NEORV32 processor ensures that *any* CPU access to memory-mapped devices (including the SLINK module) will **time out** after a certain number of cycles (see section Bus Interface). Hence, blocking access to a stream link that does not complete within a certain amount of cycles will raise a *store bus access exception* when writing to a *full* TX link's FIFO or a *load bus access exception* when reading from an *empty* RX 's FIFO. Hence, this concept should only be used when evaluating the half-full FIFO condition (for example via the SLINK interrupts) before actual accessing links.

There is no RX FIFO overflow mechanism available yet.

Non-Blocking Link Access

For a non-blocking link access concept, the FIFO status flags in `STATUS` need to be checked *before* reading/writing the actual link data register. For example, a non-blocking write access to a TX link 0 has to check *SLINK_STATUS_TX0_FREE* first. If the bit is set, the FIFO of TX link 0 can take another data word and the actual data can be written to `DATA[0]`. If the bit is cleared, the link's FIFO is full and the status flag can be polled until it there is free space in the available.

This concept will not raise any exception as there is no "direct" access to the link data registers. However, non-blocking accesses require additional instructions to check the according status flags prior to the actual link access, which will reduce performance for high-bandwidth data streams.

Stream Link Interface & Protocol

The SLINK interface consists of three signals `dat`, `val` and `rdy` for each RX and TX link. Each signal is an "array" with eight entires (one for each link). Note that an entry in `slink_*x_dat` is 32-bit wide while entries in `slink_*x_val` and `slink_*x_rdy` are are just 1-bit wide.

The stream link protocol is based on a simple FIFO-like interface between a source (sender) and a sink (receiver). Each link provides two signals for implementing a simple FIFO-style handshake. The `slink_*x_val` signal is set by the source if the according `slink_*x_dat` (also set by the source) contains valid data. The stream source has to ensure that both signals remain stable until the according `slink_*x_rdy` signal is set by the stream sink to indicate it can accept another data word.

In summary, a data word is transferred if both `slink_*x_val(i)` and `slink_*x_rdy(i)` are high.

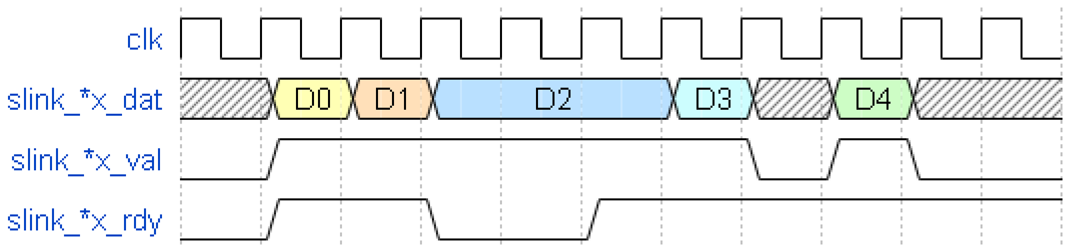


Figure 6. Exemplary stream link transfer

The SLINK handshake protocol is compatible with the AXI4-Stream base protocol.

SLINK Interrupts

The stream interface provides two independent interrupts that are *globally* driven by the RX and TX link's FIFO fill level status. Each RX and TX link provides an individual interrupt enable flag and an individual interrupt type flag that allows to configure interrupts only for certain (or all) links and for application- specific FIFO conditions. The interrupt configuration is done using the `NEORV32_SLINK.IRQ` register. Any interrupt can only become pending if the SLINK module is enabled at all.

There is no RX FIFO overflow mechanism available yet.

The current FIFO fill-level of a specific **RX link** can only raise an interrupt request if it's interrupt enable flag *SLINK_IRQ_RX_EN* is set. Vice versa, the current FIFO fill-level of a specific **TX link** can only raise an interrupt request if it's interrupt enable flag *SLINK_IRQ_TX_EN* is set.

The **RX link's** *SLINK_IRQ_RX_MODE* flags define the FIFO fill-level condition for raising an RX interrupt request: * If a link's interrupt mode flag is `0` an IRQ is generated when the link's FIFO *becomes* not empty ("RX data available"). * If a link's interrupt mode flag is `1` an IRQ is generated when the link's FIFO *becomes* at least half-full ("time to get data from RX FIFO to prevent overflow").

The **TX link's** *SLINK_IRQ_TX_MODE* flags define the FIFO fill-level condition for raising an TX interrupt request: * If a link's interrupt mode flag is `0` an IRQ is generated when the link's FIFO *becomes* not full ("space left in FIFO for new TX data"). * If a link's interrupt mode flag is `1` an IRQ is generated when the link's FIFO *becomes* less than half-full ("SW can send *SLINK_TX_FIFO*/2 data words without checking any flags").

Once the SLINK's RX or TX interrupt has become pending, it has to be explicitly cleared again by writing zero to the according **`mip`** CSR bit.

The interrupt configuration register `NEORV32_SLINK.IRQ` should we written *before* the SLINK module is actually enabled.

If *SLINK_RX_FIFO* is 1 all *SLINK_IRQ_RX_MODE* bits are hardwired to one. If *SLINK_TX_FIFO* is 1 all *SLINK_IRQ_TX_MODE* bits are hardwired to one.

Table 8. SLINK register map (`struct NEORV32_SLINK`)

Address	Name [C]	Bit(s)	R/W	Function
`0xfffffec0`				

Address	Name [C]	Bit(s)	R/W	Function
		UM3 : SLINK_CTRL_TX_NUM0		
		`3:0` *SLINK_CTRL_RX_NUM3 : SLINK_CTRL_RX_NUM0*	r/-	Number of implemented RX links
`0xfffffec4`	-	`31:0`	r/-	*reserved*

Address	Name [C]	Bit(s)	R/W	Function
`0xfffffec8`	`NEORV32_SLINK.IRQ`	`31:24` *SLINK_IRQ_RX_EN_MSB* : *SLINK_IRQ_RX_EN_LSB*	r/w	RX interrupt enable for link 7..0
		`23:16` *SLINK_IRQ_RX_MODE_MSB* : *SLINK_IRQ_RX_MODE_LSB*	r/w	RX IRQ mode for link 7..0: `0` = FIFO rises above half-full; `1` = FIFO not empty
		`15:8` *SLINK_IRQ_TX_EN_MSB* : *SLINK_IRQ_TX_EN_LSB*	r/w	TX interrupt enable for link 7..0
		`7:0` *SLINK_IRQ_TX_MODE_MSB* : *SLINK_IRQ_TX_MODE_LSB*	r/w	TX IRQ mode for link 7..0: `0` = FIFO falls below half-full; `1` = FIFO not full

Address	Name [C]	Bit(s)	R/W	Function
`0xfffffeec`	-	`31:0`	r/-	*reserved*

Address	Name [C]	Bit(s)	R/W	Function
`0xfffffed0`				

Address	Name [C]	Bit(s)	R/W	Function
		SLINK_STATUS_TX0_FREE		
		`7:0` *SLINK_STATUS_RX7_AVAIL* : *SLINK_STATUS_RX0_AVAIL*	r/-	At least one data word in RX FIFO available for link 7..0
`0xffffffed4` : `0xfffffedc`	-	`31:0`	r/-	*reserved*
`0xfffffee0`	`NEORV32_SLINK.DATA[0]`	`31:0`	r/w	Link 0 RX/TX data
`0xfffffee4`	`NEORV32_SLINK.DATA[1]`	`31:0`	r/w	Link 1 RX/TX data
`0xfffffee8`	`NEORV32_SLINK.DATA[2]`	`31:0`	r/w	Link 2 RX/TX data
`0xfffffeec`	`NEORV32_SLINK.DATA[3]`	`31:0`	r/w	Link 3 RX/TX data
`0xfffffef0`	`NEORV32_SLINK.DATA[4]`	`31:0`	r/w	Link 4 RX/TX data
`0xfffffef4`	`NEORV32_SLINK.DATA[5]`	`31:0`	r/w	Link 5 RX/TX data
`0xfffffef8`	`NEORV32_SLINK.DATA[6]`	`31:0`	r/w	Link 6 RX/TX data
`0xfffffefc`	`NEORV32_SLINK.DATA[7]`	`31:0`	r/w	Link 7 RX/TX data

2.5.8. General Purpose Input and Output Port (GPIO)

Hardware source file(s):	neorv32_gpio.vhd	
Software driver file(s):	neorv32_gpio.c	
	neorv32_gpio.h	
Top entity port:	`gpio_o`	64-bit parallel output port
	`gpio_i`	64-bit parallel input port
Configuration generics:	*IO_GPIO_EN*	implement GPIO port when *true*
CPU interrupts:	none	

The general purpose parallel IO port unit provides a simple 64-bit parallel input port and a 64-bit parallel output port. These ports can be used chip-externally (for example to drive status LEDs, connect buttons, etc.) or chip-internally to provide control signals for other IP modules. The component is disabled for implementation when the *IO_GPIO_EN* generic is set *false*. In this case the GPIO output port `gpio_o` is tied to all-zero.

Access Atomicity

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

INPUT is read-only

Write accesses to the `NEORV32_GPIO.INPUT_LO` and `NEORV32_GPIO.INPUT_HI` registers will raise a store bus error exception. The BUSKEEPER will indicate a "DEVICE_ERR" in this case.

Table 9. GPIO unit register map (`struct NEORV32_GPIO`)

Address	Name [C]	Bit(s)	R/W	Function
`0xffffffc0`	`NEORV32_GPIO.INPUT_LO`	31:0	r/-	parallel input port pins 31:0
`0xffffffc4`	`NEORV32_GPIO.INPUT_HI`	31:0	r/-	parallel input port pins 63:32
`0xffffffc8`	`NEORV32_GPIO.OUTPUT_LO`	31:0	r/w	parallel output port pins 31:0
`0xffffffcc`	`NEORV32_GPIO.OUTPUT_HI`	31:0	r/w	parallel output port pins 63:32

2.5.9. Watchdog Timer (WDT)

Hardware source file(s):	neorv32_wdt.vhd	
Software driver file(s):	neorv32_wdt.c	
	neorv32_wdt.h	
Top entity port:	none	
Configuration generics:	*IO_WDT_EN*	implement watchdog when *true*
CPU interrupts:	fast IRQ channel 0	watchdog timer overflow (see 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). The *WDT_CTRL_HALF* flag of the control register `CTRL` indicates that at least half of the maximum timeout value has been reached.

The watchdog is enabled by setting the *WDT_CTRL_EN* bit. The clock used to increment the internal counter is selected via the 3-bit *WDT_CTRL_CLK_SELx* prescaler:

`WDT_CTRL_CLK_SELx`	Main clock prescaler	Timeout period in clock cycles
`0b000`	2	2 097 152
`0b001`	4	4 194 304
`0b010`	8	8 388 608
`0b011`	64	67 108 864
`0b100`	128	134 217 728
`0b101`	1024	1 073 741 824
`0b110`	2048	2 147 483 648
`0b111`	4096	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_CTRL_MODE bit defines the action to be taken on an overflow: When cleared, the Watchdog will assert an IRQ, when set the WDT will cause a system reset. The configured action can also be triggered manually at any time by setting the *WDT_CTRL_FORCE* bit. The watchdog is reset by setting the *WDT_CTRL_RESET* bit.

A watchdog interrupt can only occur if the watchdog is enabled and interrupt mode is enabled. A triggered interrupt has to be cleared again by writing zero to the according **`mip`** CSR bit.

The cause of the last action of the watchdog can be determined via the *WDT_CTRL_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_CTRL_LOCK*. In the locked state any write access to the configuration flags is ignored (see table below, "writable 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).

Watchdog Operation during Debugging

By default the watchdog pauses operation when the CPU enters debug mode and will resume normal operation after the CPU has left debug mode. This will prevent an unintended watchdog timeout (and a hardware reset if configured) during a debug session. However, the watchdog can be configured to keep operating even when the CPU is in debug mode by setting the control register's *WDT_CTRL_DBEN* bit. If the CPU's debug mode is not implemented this flag is hardwired to zero.

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

Address	Name [C]	Bit(s), Name [C]	R/W	Reset value	Writable if locked	Function
`0xffffffbc`	`NEORV32_WDT.CTRL`	`0` *WDT_CTRL_EN*	r/w	`0`	no	watchdog enable
		`1` *WDT_CTRL_CLK_SEL0*	r/w	`0`	no	3-bit clock prescaler select
		`2` *WDT_CTRL_CLK_SEL1*	r/w	`0`	no	
		`3` *WDT_CTRL_CLK_SEL2*	r/w	`0`	no	
		`4` *WDT_CTRL_MODE*	r/w	`0`	no	overflow action: `1`=reset, `0`=IRQ
		`5` *WDT_CTRL_RCAUSE*	r/-	`0`	-	cause of last system reset: `0`=caused by external reset signal, `1`=caused by watchdog
		`6` *WDT_CTRL_RESET*	-/w	-	yes	watchdog reset when set, auto-clears
		`7` *WDT_CTRL_FORCE*	-/w	-	yes	force configured watchdog action when set, auto-clears
		`8` *WDT_CTRL_LOCK*	r/w	`0`	no	lock access to configuration when set, clears only on system reset (via external reset signal OR watchdog reset action = reset)
		`9` *WDT_CTRL_DBEN*	r/w	`0`	no	allow WDT to continue operation even when in debug mode
		`10` *WDT_CTRL_HALF*	r/-	`0`	-	set if at least half of the max. timeout counter value has been reached

2.5.10. Machine System Timer (MTIME)

Hardware source file(s):	neorv32_mtime.vhd	
Software driver file(s):	neorv32_mtime.c	
	neorv32_mtime.h	
Top entity port:	`mtime_i`	System time input from external MTIME
	`mtime_o`	System time output (64-bit) for SoC
Configuration generics:	*IO_MTIME_EN*	implement MTIME when *true*
CPU interrupts:	`MTI`	machine timer interrupt (see Processor Interrupts)

The MTIME module implements the memory-mapped MTIME machine timer from the official RISC-V specifications. This module features a 64-bit system timer incrementing with the primary processor clock. Besides accessing the MTIME register via memory operation the current system time can also be obtained using the `time[h]` CSRs. Furthermore, the current system time is made available for processor-external usage via the top's `mtime_o` signal.

The 64-bit system time can be accessed via the `TIME_LO` and `TIME_HI` memory-mapped registers (read/write) and also via the CPU's `time[h]` CSRs (read-only). A 64-bit time compare register - accessible via the memory-mapped `TIMECMP_LO` and `TIMECMP_HI` registers - is used to configure the CPU's MTI (machine timer interrupt). The interrupt is triggered whenever `TIME` (high & low part) is greater than or equal to `TIMECMP` (high & low part). The interrupt remain active (=pending) until `TIME` becomes less `TIMECMP` again (either by modifying `TIME` or `TIMECMP`).

If the processor-internal **MTIME module is NOT implemented**, the top's `mtime_i` input signal is used to update the `time[h]` CSRs and the `MTI` machine timer CPU interrupt (`MTI`) is directly connected to the top's `mtime_irq_i` input. The `mtime_o` signal is hardwired to zero in this case.

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

Address	Name [C]	Bits	R/W	Function
`0xffffff90`	`NEORV32_MTIME.TIME_LO`	31:0	r/w	machine system time, low word
`0xffffff94`	`NEORV32_MTIME.TIME_HI`	31:0	r/w	machine system time, high word
`0xffffff98`	`NEORV32_MTIME.TIMECMP_LO`	31:0	r/w	time compare, low word
`0xffffff9c`	`NEORV32_MTIME.TIMECMP_HI`	31:0	r/w	time compare, high word

2.5.11. Primary Universal Asynchronous Receiver and Transmitter (UART0)

Hardware source file(s):	neorv32_uart.vhd	
Software driver file(s):	neorv32_uart.c	
	neorv32_uart.h	
Top entity port:	`uart0_txd_o`	serial transmitter output UART0
	`uart0_rxd_i`	serial receiver input UART0
	`uart0_rts_o`	flow control: RX ready to receive
	`uart0_cts_i`	flow control: TX allowed to send
Configuration generics:	*IO_UART0_EN*	implement UART0 when *true*
	UART0_RX_FIFO	RX FIFO depth (power of 2, min 1)
	UART0_TX_FIFO	TX FIFO depth (power of 2, min 1)
CPU interrupts:	fast IRQ channel 2	RX interrupt
	fast IRQ channel 3	TX interrupt (see Processor Interrupts)

The UART is a standard serial interface mainly used to establish a communication channel between a host computer computer/user and an application running on the embedded processor.

The NEORV32 UARTs feature independent transmitter and receiver with a fixed frame configuration of 8 data bits, an optional parity bit (even or odd) and a fixed stop bit. The actual transmission rate - the Baudrate - is programmable via software. Optional FIFOs with custom sizes can be configured for the transmitter and receiver independently.

The UART features two memory-mapped registers `CTRL` and `DATA`, which are used for configuration, status check and data transfer.

Standard Console(s)

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. Furthermore, UART0 is used to implement all the standard input, output and error consoles (`STDIN`, `STDOUT` and `STDERR`).

Theory of Operation

UART0 is enabled by setting the *UART_CTRL_EN* bit in the UART0 control register `CTRL`. The Baud rate is configured via a 12-bit *UART_CTRL_BAUDxx* baud prescaler (`baud_prsc`) and a 3-bit *UART_CTRL_PRSCx* clock prescaler (`clock_prescaler`) that scales the processor's primary clock (f_{main}).

Table 12. UART0 prescaler configuration

`UART_CTRL_PRSCx`	`0b000`	`0b001`	`0b010`	`0b011`	`0b100`	`0b101`	`0b110`	`0b111`
Resulting `clock_prescaler`	2	4	8	64	128	1024	2048	4096

Baud rate = (f_{main}*[Hz]* / `clock_prescaler`) / (`baud_prsc` + 1)

A new transmission is started by writing the data byte to be send to the lowest byte of the `DATA` register. The transfer is completed when the *UART_CTRL_TX_BUSY* control register flag returns to zero. A new received byte is available when the *UART_DATA_AVAIL* flag of the `DATA` register is set. A "frame error" in a received byte (invalid stop bit) is indicated via the *UART_DATA_FERR* flag in the `DATA` register. The flag is cleared by reading the `DATA` register.

A transmission (RX or TX) can be terminated at any time by disabling the UART module by clearing the *UART_CTRL_EN* control register bit.

RX and TX FIFOs

UART0 provides optional FIFO buffers for the transmitter and the receiver. The *UART0_RX_FIFO* generic defines the depth of the RX FIFO (for receiving data) while the *UART0_TX_FIFO* defines the depth of the TX FIFO (for sending data). Both generics have to be a power of two with a minimal allowed value of 1. This minimal value will implement simple "double-buffering" instead of full-featured FIFOs. Both FIFOs are cleared whenever UART0 is disabled (clearing *UART_CTRL_EN* in `CTRL`).

The state of both FIFO (*empty*, *at lest half-full*, *full*) is available via the *UART_CTRL*?X_EMPTY_, *UART_CTRL*?X_HALF_ and *UART_CTRL**X_FULL_ flags in the `CTRL` register.

If the RX FIFO is already full and new data is received by the receiver unit, the *UART_DATA_OVERR* flag in the `DATA` register is set indicating an "overrun". This flag is cleared by reading the `DATA` register.

In contrast to other FIFO-equipped peripherals, software **cannot** determine the UART's FIFO size configuration by reading specific control register bits (simply because there are no bits left in the control register).

Hardware Flow Control - RTS/CTS

UART0 supports optional 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 individually.

- If **RTS hardware flow control** is enabled by setting the *UART_CTRL_RTS_EN* control register flag, the UART will pull the `uart0_rts_o` signal low if the UART's receiver is ready to receive new data. As long as this signal is low the connected device can send new data. `uart0_rts_o` is always LOW if the UART is disabled. The RTS line is de-asserted (going high) as soon as the start bit of a new incoming char has been detected.
- If **CTS hardware flow control** is enabled by setting the *UART_CTRL_CTS_EN* control register flag, the UART's transmitter will not start sending a new data until the `uart0_cts_i` signal goes low. During this time, the UART busy flag *UART_CTRL_TX_BUSY* remains set. If `uart0_cts_i` is asserted, no new data transmission will be started by the UART. The state of the `uart0_cts_i` signal has no effect on a transmission being already in progress. Application software can check

the current state of the `uart0_cts_o` input signal via the *UART_CTRL_CTS* control register flag.

Parity Modes

An optional parity bit can be added to the data stream if the *UART_CTRL_PMODE1* flag is set. When *UART_CTRL_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_PERR* flag in the `DATA` register. This flag is updated with each new received character and is cleared by reading the `DATA` register.

UART Interrupts

UART0 features two independent interrupt for signaling certain RX and TX conditions. The behavior of these conditions differs based on the configured FIFO sizes. If the according FIFO size is greater than 1, the *UART_CTRL_RX_IRQ* and *UART_CTRL_TX_IRQ* `CTRL` flags allow a more fine-grained IRQ configuration. An interrupt can only become pending if the according interrupt condition is fulfilled and the UART is enabled at all.

- If *UART0_RX_FIFO* is exactly 1, the RX interrupt goes pending when data *becomes* available in the RX FIFO (→ *UART_CTRL_RX_EMPTY* clears). *UART_CTRL_RX_IRQ* is hardwired to `0` in this case.
- If *UART0_TX_FIFO* is exactly 1, the TX interrupt goes pending when at least one entry in the TX FIFO *becomes* free (→ *UART_CTRL_TX_FULL* clears). *UART_CTRL_TX_IRQ* is hardwired to `0` in this case.
- If *UART0_RX_FIFO* is greater than 1: If *UART_CTRL_RX_IRQ* is `0` the RX interrupt goes pending when data *becomes* available in the RX FIFO (→ *UART_CTRL_RX_EMPTY* clears). If *UART_CTRL_RX_IRQ* is `1` the RX interrupt becomes pending the RX FIFO *becomes* at least half-full (→ *UART_CTRL_RX_HALF* sets).
- If *UART0_TX_FIFO* is greater than 1: If *UART_CTRL_TX_IRQ* is `0` the TX interrupt goes pending when at least one entry in the TX FIFO *becomes* free (→ *UART_CTRL_TX_FULL* clears). If *UART_CTRL_TX_IRQ* is `1` the TX interrupt goes pending when the RX FIFO *becomes* less than half-full (→ *UART_CTRL_TX_HALF* clears).

Once the RX or TX interrupt has become pending, it has to be explicitly cleared again by writing zero to the according **`mip`** CSR bit.

Simulation Mode

The default UART0 operation will transmit any data written to the `DATA` register via the serial TX line at the defined baud rate via the physical link. To accelerate UART0 output during simulation (and also to dump large amounts of data) the UART0 features a *simulation mode*.

Simulation mode is enabled by setting the *UART_CTRL_SIM_MODE* bit in the UART0's control register `CTRL`. Any other UART0 configuration bits are irrelevant for this mode but UART0 has to be enabled via the *UART_CTRL_EN* bit. There will be no physical UART0 transmissions via `uart0_txd_o` at all when simulation mode is enabled. Furthermore, no interrupts (RX & TX) will be triggered.

When the simulation mode is enabled any data written to `DATA[7:0]` is directly output as ASCII char to the simulator console. Additionally, all chars are 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 `DATA[31:0]` 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.

More information regarding the simulation-mode of the UART0 can be found in the User Guide section Simulating the Processor.

Table 13. UART0 register map (`struct NEORV32_UART0`)

Address	Name [C]	Bit(s), Name [C]	R/W	Function
`0xffffffa0`				

Address	Name [C]	Bit(s), Name [C]	R/W	Function
		26 *UART_CTRL_PRSC2*	r/w	
		27 *UART_CTRL_CTS*	r/-	current state of UART's CTS input signal
		28 *UART_CTRL_EN*	r/w	UART enable
		29 *UART_CTRL_RX_IRQ*	r/w	RX IRQ mode: 1=FIFO at least half-full; 0=FIFO not empty
		30 *UART_CTRL_TX_IRQ*	r/w	TX IRQ mode: 1=FIFO less than half-full; 0=FIFO not full
		31 *UART_CTRL_TX_BUSY*	r/-	transmitter busy flag
`0xffffffa4`	`NEORV32_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
		29 *UART_DATA_FERR*	r/-	RX data frame error (stop bit nt set)
		30 *UART_DATA_OVERR*	r/-	RX data overrun
		31 *UART_DATA_AVAIL*	r/-	RX data available when set

2.5.12. Secondary Universal Asynchronous Receiver and Transmitter (UART1)

Hardware source file(s):	neorv32_uart.vhd	
Software driver file(s):	neorv32_uart.c	
	neorv32_uart.h	
Top entity port:	`uart1_txd_o`	serial transmitter output UART1
	`uart1_rxd_i`	serial receiver input UART1
	`uart1_rts_o`	flow control: RX ready to receive
	`uart1_cts_i`	flow control: TX allowed to send
Configuration generics:	*IO_UART1_EN*	implement UART1 when *true*
	UART1_RX_FIFO	RX FIFO depth (power of 2, min 1)
	UART1_TX_FIFO	TX FIFO depth (power of 2, min 1)
CPU interrupts:	fast IRQ channel 4	RX interrupt
	fast IRQ channel 5	TX interrupt (see Processor Interrupts)

Theory of Operation

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

Simulation Mode

The secondary UART (UART1) provides the same simulation options as the primary UART. 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 data words.

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

Address	Name [C]	Bit(s), Name [C]	R/W	Function
`0xffffffd0`				

Address	Name [C]	Bit(s), Name [C]	R/W	Function
		24 *UART_CTRL_PRSC0*	r/w	3-bit baudrate clock prescaler select
		25 *UART_CTRL_PRSC1*	r/w	
		26 *UART_CTRL_PRSC2*	r/w	
		27 *UART_CTRL_CTS*	r/-	current state of UART's CTS input signal
		28 *UART_CTRL_EN*	r/w	UART enable
		29 *UART_CTRL_RX_IRQ*	r/w	RX IRQ mode: 1=FIFO at least half-full; 0=FIFO not empty; hardwired to zero if *UART0_RX_FIFO* = 1
		30 *UART_CTRL_TX_IRQ*	r/w	TX IRQ mode: 1=FIFO less than half-full; 0=FIFO not full; hardwired to zero if *UART0_TX_FIFO* = 1
		31 *UART_CTRL_TX_BUSY*	r/-	transmitter busy flag
0xffffffd4	NEORV32_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
		29 *UART_DATA_FERR*	r/-	RX data frame error (stop bit nt set)
		30 *UART_DATA_OVERR*	r/-	RX data overrun
		31 *UART_DATA_AVAIL*	r/-	RX data available when set

2.5.13. Serial Peripheral Interface Controller (SPI)

Hardware source file(s):	neorv32_spi.vhd	
Software driver file(s):	neorv32_spi.c	
	neorv32_spi.h	
Top entity port:	`spi_sck_o`	1-bit serial clock output
	`spi_sdo_o`	1-bit serial data output
	`spi_sdi_i`	1-bit serial data input
	`spi_csn_i`	8-bit dedicated chip select (low-active)
Configuration generics:	*IO_SPI_EN*	implement SPI controller when *true*
CPU interrupts:	fast IRQ channel 6	transmission done interrupt (see Processor Interrupts)

Theory of Operation

SPI is a synchronous serial transmission interface for fast on-board communications. The NEORV32 SPI transceiver supports 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, which are directly controlled by the SPI module (no additional GPIO required).

The NEORV32 SPI module only supports *host mode*. Transmission are initiated only by the processor's SPI module (and not by an external SPI module).

The SPI unit is enabled by setting the *SPI_CTRL_EN* bit in the `CTRL` control register. No transfer can be initiated and no interrupt request will be triggered if this bit is cleared. Furthermore, a transfer being in process can be terminated at any time by clearing this bit.

Changes to the `CTRL` control register should be made only when the SPI module is idle as they directly effect transmissions being in-progress.

A transmission can be terminated at any time by disabling the SPI module by clearing the *SPI_CTRL_EN* control register bit.

The data quantity to be transferred within a single transmission is defined via the *SPI_CTRL_SIZEx* bits. The SPI module supports 8-bit (`00`), 16-bit (`01`), 24-bit (`10`) and 32-bit (`11`) transfers.

A transmission is started when writing data to the `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 `DATA`. Vice versa, the received data is also always LSB-aligned. Application software should only actually process the amount of bits that were configured using *SPI_CTRL_SIZEx* when reading `DATA`.

The NEORV32 SPI module only support MSB-first mode. Data can be reversed before writing `DATA` (for TX) / after reading `DATA` (for RX) to implement LSB-first transmissions. Note that in both cases data in ` DATA` still needs to be LSB-aligned.

The actual transmission length is left to the user: after asserting chip-select an arbitrary amount of transmission with arbitrary data quantity (*SPI_CTRL_SIZEx*) can be made before de-asserting chip-select again.

The SPI controller features 8 dedicated chip-select lines. These lines are controlled via the control register's *SPI_CTRL_CSx* bits. When a specific *SPI_CTRL_CSx* bit is **set**, the according chip-select line `spi_csn_o(x)` goes **low** (low-active chip-select lines).

The dedicated SPI chip-select signals can be seen as *general purpose* outputs. These are intended to control the accessed device's chip-select signal but can also be use for controlling other shift register signals (like data strobe or output-enables).

SPI Clock Configuration

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

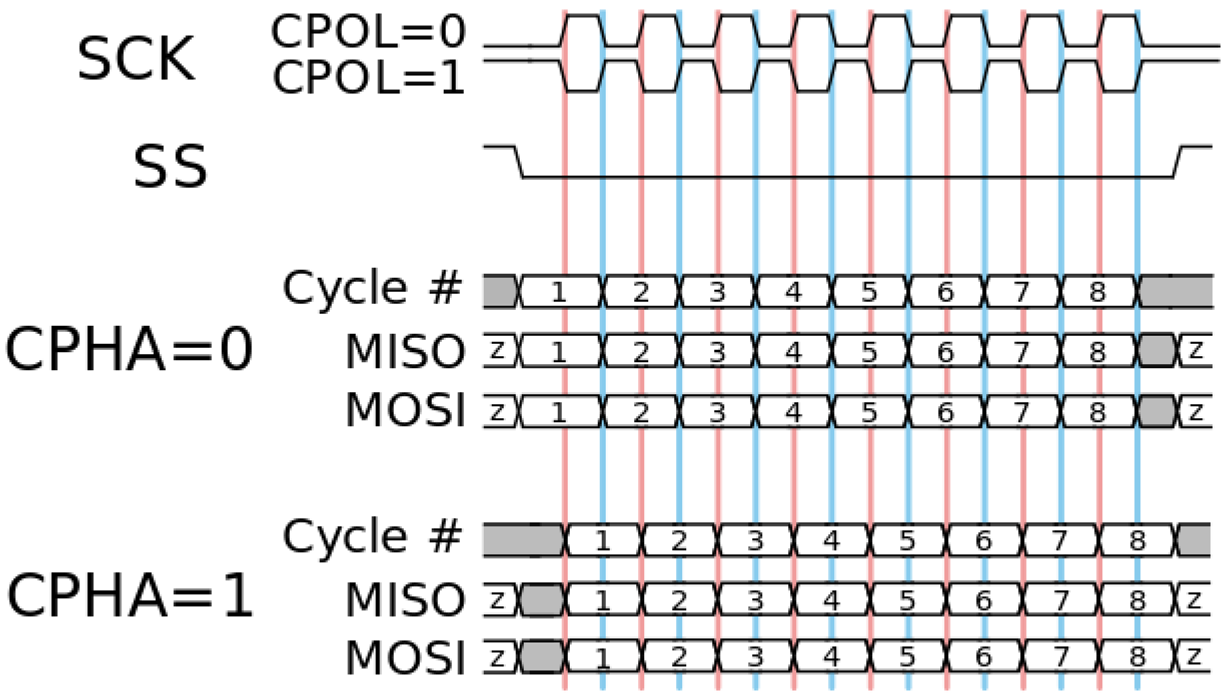


Figure 7. SPI clock modes; image from https://en.wikipedia.org/wiki/File:SPI_timing_diagram2.svg (license: (Wikimedia) Creative Commons Attribution-Share Alike 3.0 Unported)

Table 15. SPI standard clock modes

	Mode 0	**Mode 1**	**Mode 2**	**Mode 4**
SPI_CTRL_CPOL	`0`	`0`	`1`	`1`
SPI_CTRL_CPHA	`0`	`1`	`0`	`1`

The SPI clock frequency (`spi_sck_o`) is programmed by the 3-bit *SPI_CTRL_PRSCx* clock prescaler. The following prescalers are available:

Table 16. SPI prescaler configuration

`SPI_CTRL_PRSCx`	`0b000`	`0b001`	`0b010`	`0b011`	`0b100`	`0b101`	`0b110`	`0b111`
Resulting `clock_prescaler`	2	4	8	64	128	1024	2048	4096

Based on the *SPI_CTRL_PRSCx* configuration, the actual SPI clock frequency f_{SPI} is derived from the processor's main clock f_{main} and is determined by:

$f_{SPI} = f_{main}[Hz] / (2 *$ `clock_prescaler`$)$

Hence, the maximum SPI clock is f_{main} / 4.

High-Speed SPI mode

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

SPI Interrupt

The SPI module provides a single interrupt to signal "transmission done" to the CPU. Whenever the SPI module completes the current transfer operation, the interrupt is triggered and has to be explicitly cleared again by writing zero to the according **`mip`** CSR bit.

Table 17. SPI register map (`struct NEORV32_SPI`)

Address	Name [C]	Bit(s), Name [C]	R/W	Function
0xffffffa8	NEORV32_SPI.CTRL	0 *SPI_CTRL_CS0*	r/w	Direct chip-select 0..7; setting spi_csn_o(x) low when set
		1 *SPI_CTRL_CS1*	r/w	
		2 *SPI_CTRL_CS2*	r/w	
		3 *SPI_CTRL_CS3*	r/w	
		4 *SPI_CTRL_CS4*	r/w	
		5 *SPI_CTRL_CS5*	r/w	
		6 *SPI_CTRL_CS6*	r/w	
		7 *SPI_CTRL_CS7*	r/w	
		8 *SPI_CTRL_EN*	r/w	SPI enable
		9 *SPI_CTRL_CPHA*	r/w	clock phase (0=sample RX on rising edge & update TX on falling edge; 1=sample RX on falling edge & update TX on rising edge)
		10 *SPI_CTRL_PRSC0*	r/w	3-bit clock prescaler select
		11 *SPI_CTRL_PRSC1*	r/w	
		12 *SPI_CTRL_PRSC2*	r/w	
		13 *SPI_CTRL_SIZE0*	r/w	transfer size (00=8-bit, 01=16-bit, 10=24-bit, 11=32-bit)
		14 *SPI_CTRL_SIZE1*	r/w	
		15 *SPI_CTRL_CPOL*	r/w	clock polarity
		16 *SPI_CTRL_HIGHSPEED*	r/w	enable SPI high-speed mode (ignoring *SPI_CTRL_PRSC*)
		17:30	r/-	_reserved, read as zero
		31 *SPI_CTRL_BUSY*	r/-	transmission in progress when set
0xffffffac	NEORV32_SPI.DATA	31:0	r/w	receive/transmit data, LSB-aligned

2.5.14. Two-Wire Serial Interface Controller (TWI)

Hardware source file(s):	neorv32_twi.vhd	
Software driver file(s):	neorv32_twi.c	
	neorv32_twi.h	
Top entity port:	`twi_sda_io`	1-bit bi-directional serial data
	`twi_scl_io`	1-bit bi-directional serial clock
Configuration generics:	*IO_TWI_EN*	implement TWI controller when *true*
CPU interrupts:	fast IRQ channel 7	transmission done interrupt (see Processor Interrupts)

Theory of Operation

The two wire interface - also 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 supports "clock so a slow peripheral can halt the transmission by pulling the SCL line low. Currently, **no multi-controller support** is available. Also, the NEORV32 TWI unit cannot operate in peripheral mode.

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

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

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

A transmission can be terminated at any time by disabling the TWI module by clearing the *TWI_CTRL_EN* control register bit.

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-bit *TWI_CTRL_PRSCx* clock prescaler. The following prescalers are available:

Table 18. TWI prescaler configuration

`TWI_CTRL_PRSCx`	`0b000`	`0b001`	`0b010`	`0b011`	`0b100`	`0b101`	`0b110`	`0b111`
Resulting `clock_prescaler`	2	4	8	64	128	1024	2048	4096

Based on the *TWI_CTRL_PRSCx* configuration, the actual TWI clock frequency f_{SCL} is derived from the processor main clock f_{main} and is determined by:

f_{SCL} = f_{main}*[Hz]* / (4 * `clock_prescaler`)

TWI Interrupt

The SPI module provides a single interrupt to signal "operation done" to the CPU. Whenever the TWI module completes the current operation (generate stop condition, generate start conditions or transfer byte), the interrupt is triggered. Once triggered, the interrupt has to be explicitly cleared again by writing zero to the according **`mip`** CSR bit.

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

Address	Name [C]	Bit(s), Name [C]	R/W	Function
`0xffffffb0`	`NEORV32_TWI.CTRL`	`0` *TWI_CTRL_EN*	r/w	TWI enable
		`1` *TWI_CTRL_START*	r/w	generate START condition
		`2` *TWI_CTRL_STOP*	r/w	generate STOP condition
		`3` *TWI_CTRL_PRSC0*	r/w	3-bit clock prescaler select
		`4` *TWI_CTRL_PRSC1*	r/w	
		`5` *TWI_CTRL_PRSC2*	r/w	
		`6` *TWI_CTRL_MACK*	r/w	generate controller ACK for each transmission ("MACK")
		`30` *TWI_CTRL_ACK*	r/-	ACK received when set
		`31` *TWI_CTRL_BUSY*	r/-	transfer/START/STOP in progress when set
`0xffffffb4`	`NEORV32_TWI.DATA`	`7:0` *TWI_DATA_MSB* : TWI_DATA_LSB_	r/w	receive/transmit data

2.5.15. Pulse-Width Modulation Controller (PWM)

Hardware source file(s):	neorv32_pwm.vhd	
Software driver file(s):	neorv32_pwm.c	
	neorv32_pwm.h	
Top entity port:	`pwm_o`	up to 60 PWM output channels (60-bit, fixed)
Configuration generics:	*IO_PWM_NUM_CH*	number of PWM channels to implement (0..60)
CPU interrupts:	none	

The PWM controller implements a pulse-width modulation controller with up to 60 independent channels and 8- bit resolution per channel. The actual number of implemented channels is defined by the *IO_PWM_NUM_CH* generic. Setting this generic to zero will completely remove the PWM controller from the design.

The `pwm_o` has a static size of 60-bit. Is less than 60 PWM channels are configured, only the LSB-aligned channels (bits) are used while the remaining bits are hardwired to zero.

The PWM controller is based on an 8-bit base counter with a programmable threshold comparators for each channel that defines the actual duty cycle. The controller can be used to drive fancy RGB-LEDs with 24- bit true color, to dim LCD back-lights or even for "analog" control. An external integrator (RC low-pass filter) can be used to smooth the generated "analog" signals.

Theory of Operation

The PWM controller is activated by setting the *PWM_CTRL_EN* bit in the module's control register `CTRL`. When this bit is cleared, the unit is reset and all PWM output channels are set to zero. The 8-bit duty cycle for each channel, which represents the channel's "intensity", is defined via an 8-bit value. The module provides up to 15 duty cycle registers `DUTY[0]` to `DUTY[14]` (depending on the number of implemented channels). Each register contains the duty cycle configuration for 4 consecutive channels. For example, the duty cycle of channel 0 is defined via bits 7:0 in `DUTY[0]`. The duty cycle of channel 2 is defined via bits 15:0 in `DUTY[0]`. Channel 4's duty cycle is defined via bits 7:0 in `DUTY[1]` and so on.

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

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

Intensity $_x$ = `DUTY[y](i*8+7 downto i*8)` / (2^8)

The base frequency of the generated PWM signals is defined by the PWM core clock. This clock is derived from the main processor clock and divided by a prescaler via the 3-bit PWM_CTRL_PRSCx in the unit’s control register. The following pre-scalers are available:

Table 20. PWM prescaler configuration

`PWM_CTRL_PRSCx`	`0b000`	`0b001`	`0b010`	`0b011`	`0b100`	`0b101`	`0b110`	`0b111`
Resulting `clock_prescaler`	2	4	8	64	128	1024	2048	4096

The resulting PWM base frequency is defined by:

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

Table 21. PWM register map (`struct neorv32_pwm_t`)

Address	Name [C]	Bit(s), Name [C]	R/W	Function
`0xfffffe80`	`NEORV32_PWM.CTRL`	`0` *PWM_CTRL_EN*	r/w	PWM enable
		`1` *PWM_CTRL_PRSC0*	r/w	3-bit clock prescaler select
		`2` *PWM_CTRL_PRSC1*	r/w	
		`3` *PWM_CTRL_PRSC2*	r/w	
`0xfffffe84`	`NEORV32_PWM.DUTY[0]`	`7:0`	r/w	8-bit duty cycle for channel 0
		`15:8`	r/w	8-bit duty cycle for channel 1
		`23:16`	r/w	8-bit duty cycle for channel 2
		`31:24`	r/w	8-bit duty cycle for channel 3
…	…	…	r/w	…
`0xfffffebc`	`NEORV32_PWM.DUTY[14]`	`7:0`	r/w	8-bit duty cycle for channel 56
		`15:8`	r/w	8-bit duty cycle for channel 57
		`23:16`	r/w	8-bit duty cycle for channel 58
		`31:24`	r/w	8-bit duty cycle for channel 59

2.5.16. True Random-Number Generator (TRNG)

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

Theory of Operation

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

The TRNG is based on the *neoTRNG*, which is a "spin-off project" of the NEORV32 processor. The TRNG uses the default neoTRNG configuration, which showed very good results in the `dieharder` battery of random number tests. More detailed information about the neoTRNG, it's architecture and a detailed evaluation of the random number quality can be found it it's repository: https://github.com/stnolting/neoTRNG

Platform Independent Architecture

The TRNG features a platform independent architecture without FPGA-specific primitives, macros or attributes so it can be synthesized for *any* FPGA.

Inferring Latches

The synthesis tool might emit a warning like *"inferring latches for … neorv32_trng …"*. This is no problem as this is what we actually want (the TRNG is based on latches).

Using the TRNG

The TRNG features a single register for status and data access. When the *TRNG_CTRL_EN* control register (`CTRL`) bit is set, the TRNG is enabled and starts operation. As soon as the *TRNG_CTRL_VALID* bit is set, the currently sampled 8-bit random data byte can be obtained from the lowest 8 bits of the `CTRL` register (*TRNG_CTRL_DATA_MSB* : *TRNG_CTRL_DATA_LSB*). These bits always keep the latest valid data obtained from the TRNG entropy source. The *TRNG_CTRL_VALID* bit is automatically cleared when reading the control register.

TRNG Reset

The TRNG core does not provide a dedicated reset. In order to ensure correct operations, the TRNG should be disabled (=reset) by clearing the *TRNG_CTRL_EN* and waiting some milliseconds before re-enabling it.

Table 22. TRNG register map (`struct NEORV32_TRNG`)

Address	Name [C]	Bit(s), Name [C]	R/W	Function
`0xffffffb8`	`NEORV32_TRNG.CTRL`	`7:0` *TRNG_CTRL_DATA_MSB* : *TRNG_CTRL_DATA_MSB*	r/-	8-bit random data
		`30` *TRNG_CTRL_EN*	r/w	TRNG enable
		`31` *TRNG_CTRL_VALID*	r/-	random data is valid when set

2.5.17. Custom Functions Subsystem (CFS)

Hardware source file(s):	neorv32_gfs.vhd	
Software driver file(s):	neorv32_gfs.c	
	neorv32_gfs.h	
Top entity port:	`cfs_in_i`	custom input conduit
	`cfs_out_o`	custom output conduit
Configuration generics:	*IO_CFS_EN*	implement CFS when *true*
	IO_CFS_CONFIG	custom generic conduit
	IO_CFS_IN_SIZE	size of `cfs_in_i`
	IO_CFS_OUT_SIZE	size of `cfs_out_o`
CPU interrupts:	fast IRQ channel 1	CFS interrupt (see Processor Interrupts)

Theory of Operation

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

In contrast to connecting custom hardware accelerators via external memory interfaces (like SPI or the processor's external bus interface), the CFS provide a convenient, low-latency and tightly-coupled extension and customization option.

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

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

Take a look at the template CFS VHDL source file (`rtl/core/neorv32_cfs.vhd`). The file is highly commented to illustrate all aspects that are relevant for implementing custom CFS-based co-processor designs.

CFS Software Access

The CFS memory-mapped registers can be accessed by software using the provided C-language aliases (see register map table below). Note that all interface registers are declared as 32-bit words of type uint32_t.

Listing 2. CFS Software Access Example

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

A very simple example program that uses the *default* CFS hardware module can be found in sw/example/cfs_demo.

CFS Interrupt

The CFS provides a single high-level-triggered interrupt request signal mapped to the CPU's fast interrupt channel 1. Once triggered, the interrupt becomes pending (if enabled in the mis CSR) and has to be explicitly cleared again by writing zero to the according **mip** CSR bit. See section Processor Interrupts for more information.

CFS Configuration Generic

By default, the CFS provides a single 32-bit std_(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 directly down to the CFS. The actual definition of the generic and it's usage inside the CFS is left to the hardware designer.

CFS Custom IOs

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

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

Table 23. CFS register map (struct NEORV32_CFS)

Address	Name [C]	Bit(s)	R/W	Function
0xffffffe00	NEORV32_CFS.REG[0]	31:0	(r)/(w)	custom CFS interface register 0
0xffffffe04	NEORV32_CFS.REG[1]	31:0	(r)/(w)	custom CFS interface register 1
…	…	31:0	(r)/(w)	…

Address	Name [C]	Bit(s)	R/W	Function
`0xfffffe78`	`NEORV32_CFS.REG[30]`	`31:0`	(r)/(w)	custom CFS interface register 30
`0xfffffe7c`	`NEORV32_CFS.REG[31]`	`31:0`	(r)/(w)	custom CFS interface register 31

2.5.18. Smart LED Interface (NEOLED)

Hardware source file(s):	neorv32_neoled.vhd	
Software driver file(s):	neorv32_neoled.c	
	neorv32_neoled.h	
Top entity port:	`neoled_o`	1-bit serial data output
Configuration generics:	*IO_NEOLED_EN*	implement NEOLED when *true*
	IO_NEOLED_TX_FIFO	TX FIFO depth (1..32k, has to be a power of two)
CPU interrupts:	fast IRQ channel 9	NEOLED interrupt (see Processor Interrupts)

Theory of Operation

The NEOLED module provides a dedicated interface for "smart RGB 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 (see link:https://learn.adafruit.com/adafruit-neopixel-uberguide).

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

Theory of Operation - NEOLED Module

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

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 module.

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 `DATA` to the LEDs. If *NEOLED_MODE_EN* is set, the NEOLED interface operates in 32-bit mode and will transmit bits `31:0` of the data written to `DATA` to the LEDs.

The mode bit can be configured before writing each new data word in order to support an arbitrary setup of RGB and RGBW LEDs.

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 (& W) LED chip (= color code) is defined via an 8-bit value. The actual data bits are transferred by modifying the duty cycle of the signal (the timings for the WS2812 are shown below). A RESET command is "send" by pulling the data line LOW for at least 50µs.

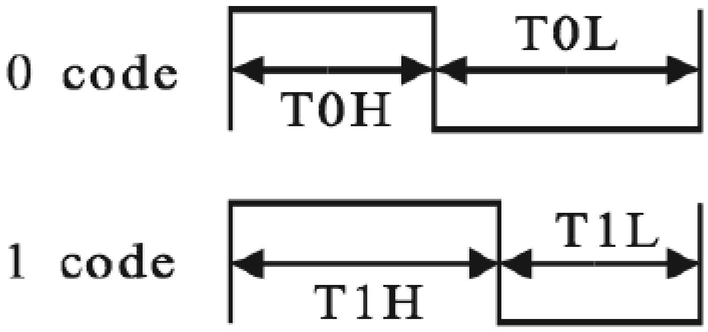


Figure 8. WS2812 bit-level protocol - taken from the "Adafruit NeoPixel Überguide"

Table 24. WS2812 interface timing

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

Timing Configuration

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

Table 25. NEOLED prescaler configuration

`NEOLED_CTRL_PRSCx`	`0b000`	`0b001`	`0b010`	`0b011`	`0b100`	`0b101`	`0b110`	`0b111`
Resulting `clock_prescaler`	2	4	8	64	128	1024	2048	4096

The duty-cycles (or more precisely: the high- and low-times for sending either a '1' bit or a '0' bit) are defined via the 5-bit *NEOLED_CTRL_T_ONE_H_x* and *NEOLED_CTRL_T_ZERO_H_x* values, respectively. These programmable timing constants allow to adapt the interface for a wide variety of smart LED protocol (for example WS2812 vs. WS2811).

Timing Configuration - Example (WS2812)

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

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

$\boldsymbol{f_{TX}}$ = *f*f_{main}*[Hz]* / `clock_prescaler` = 100MHz / 4 = 25MHz

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

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

- *NEOLED_CTRL_T_TOT* = `0b11110` (= decimal 30)
- *NEOLED_CTRL_T_ZERO_H* = `0b01010` (= decimal 10)
- *NEOLED_CTRL_T_ONE_H* = `0b10100` (= decimal 20)

$\boldsymbol{T_{carrier}}$ = $\boldsymbol{T_{TX}}$ * *NEOLED_CTRL_T_TOT* = 40ns * 30 = 1.4µs

$\boldsymbol{T_{0H}}$ = $\boldsymbol{T_{TX}}$ * *NEOLED_CTRL_T_ZERO_H* = 40ns * 10 = 0.4µs

$\boldsymbol{T_{1H}}$ = $\boldsymbol{T_{TX}}$ * *NEOLED_CTRL_T_ONE_H* = 40ns * 20 = 0.8µs

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

TX Data FIFO

The interface features a TX data buffer (a FIFO) to allow more CPU-independent operation. The buffer depth is configured via the *IO_NEOLED_TX_FIFO* top generic (default = 1 entry). The FIFO size configuration can be read via the *NEOLED_CTRL_BUFS_x* control register bits, which result log2(*IO_NEOLED_TX_FIFO*).

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

Software can check the FIFO fill level via the control register's *NEOLED_CTRL_TX_EMPTY*, *NEOLED_CTRL_TX_HALF* and *NEOLED_CTRL_TX_FULL* flags. The *NEOLED_CTRL_TX_BUSY* flags provides additional information if the the TX unit is still busy sending data.

Please note that the timing configurations (*NEOLED_CTRL_PRSCx*, *NEOLED_CTRL_T_TOT_x*, *NEOLED_CTRL_T_ONE_H_x* and *NEOLED_CTRL_T_ZERO_H_x*) are **NOT** stored to the buffer. Changing these value while the buffer is not empty or the TX engine is still busy will cause data corruption.

- Strobe Command ("RESET") **

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

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

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

NEOLED Interrupt

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

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

One the NEOLED interrupt has been triggered and became pending, it has to explicitly cleared again by writing zero to according `mip` CSR bit.

The *NEOLED_CTRL_IRQ_CONF* is hardwired to one if *IO_NEOLED_TX_FIFO* = 1 (→ IRQ if FIFO is empty). If the FIFO is configured to contain only a single entry (*IO_NEOLED_TX_FIFO* = 1) the interrupt will become pending if the FIFO (which is just a single register providing simple *double-buffering*) is empty.

Table 26. NEOLED register map (`struct NEORV32_NEOLED`)

Address	Name [C]	Bit(s), Name [C]	R/W	Function
`0xffffffd8`				

Address	Name [C]	Bit(s), Name [C]	R/W	Function
		21 *NEOLED_CTRL_T_ONE_H_1*	r/w	high time for sending a one bit (T_{1H})
		22 *NEOLED_CTRL_T_ONE_H_2*	r/w	
		23 *NEOLED_CTRL_T_ONE_H_3*	r/w	
		24 *NEOLED_CTRL_T_ONE_H_4*	r/w	
		27 *NEOLED_CTRL_IRQ_CONF*	r/w	TX FIFO interrupt configuration: `0`=IRQ if FIFO is less than half-full, `1`=IRQ if FIFO is empty
		28 *NEOLED_CTRL_TX_EMPTY*	r/-	TX FIFO is empty
		29 *NEOLED_CTRL_TX_HALF*	r/-	TX FIFO is *at least* half full
		30 *NEOLED_CTRL_TX_FULL*	r/-	TX FIFO is full
		31 *NEOLED_CTRL_TX_BUSY*	r/-	TX serial engine is busy when set
`0xffffffdc`	`NEORV32_NEOLED.DATA`	`31:0` / `23:0`	-/w	TX data (32-/24-bit)

2.5.19. External Interrupt Controller (XIRQ)

Hardware source file(s):	neorv32_xirq.vhd	
Software driver file(s):	neorv32_xirq.c	
	neorv32_xirq.h	
Top entity port:	`xirq_i`	IRQ input (32-bit, fixed)
Configuration generics:	*XIRQ_NUM_CH*	Number of IRQs to implement (0..32)
	XIRQ_TRIGGER_TYPE	IRQ trigger type configuration
	XIRQ_TRIGGER_POLARITY	IRQ trigger polarity configuration
CPU interrupts:	fast IRQ channel 8	XIRQ (see Processor Interrupts)

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

Theory of Operation

The XIRQ provides up to 32 interrupt *channels* (configured via the *XIRQ_NUM_CH* generic). Each bit in the `xirq_i` input signal vector represents one interrupt channel. If less than 32 channels are configure, only the LSB-aligned channels are used while the remaining bits are left unconnected. An interrupt channel is enabled by setting the according bit in the interrupt enable register `IER`.

If the configured trigger (see below) of an enabled channel fires, the request is stored into an internal buffer. This buffer is available via the interrupt pending register `IPR`. A `1` in this register indicates that the corresponding interrupt channel has fired but has not yet been serviced (so it is pending). An interrupt channel can become pending if the according `IER` bit is set. Pending IRQs can be cleared by writing `0` to the according `IPR` bit. As soon as there is a least one pending interrupt in the buffer, an interrupt request is send to the CPU.

A disabled interrupt channel can still be pending if it has been triggered before clearing the according `IER` bit.

The CPU can determine active external interrupt request either by checking the bits in the `IPR` register, which show all pending interrupt channels, or by reading the interrupt source register `SCR`. This register provides a 5-bit wide ID (0..31) that shows the interrupt request with *highest priority*. Interrupt channel `xirq_i(0)` has highest priority and `xirq_i(XIRQ_NUM_CH-1)` has lowest priority. This priority assignment is fixed and cannot be altered by software. The CPU can use the ID from `SCR` to service IRQ according to their priority. To acknowledge the according interrupt the CPU can write `1 << SCR` to `IPR`.

In order to clear a pending FIRQ interrupt from the external interrupt controller again, the according **`mip`** CSR bit has to be cleared. Additionally, the XIRQ interrupt has to be acknowledged by writing *any* value to the interrupt source register `SRC`.

An interrupt handler should clear the interrupt pending bit that caused the interrupt first before acknowledging the interrupt by writing the `SCR` register.

IRQ Trigger Configuration

The controller does not provide a configuration option to define the IRQ triggers *during runtime*. Instead, two generics are provided to configure the trigger of each interrupt channel before synthesis: the *XIRQ_TRIGGER_TYPE* and *XIRQ_TRIGGER_POLARITY* generic. Both generics are 32 bit wide representing one bit per interrupt channel. If less than 32 interrupt channels are implemented the remaining configuration bits are ignored.

XIRQ_TRIGGER_TYPE is used to define the general trigger type. This can be either *level-triggered* (`0`) or *edge-triggered* (`1`). *XIRQ_TRIGGER_POLARITY* is used to configure the polarity of the trigger: a `0` defines low-level or falling-edge and a `1` defines high-level or rising-edge.

Listing 3. Example trigger configuration: channel 0 for rising-edge, IRQ channels 1 to 31 for high-level

```
XIRQ_TRIGGER_TYPE     => x"00000001";
XIRQ_TRIGGER_POLARITY => x"ffffffff";
```

Table 27. XIRQ register map (`struct NEORV32_XIRQ`)

Address	Name [C]	Bit(s)	R/W	Function
`0xffffff80`	`NEORV32_XIRQ.IER`	`31:0`	r/w	Interrupt enable register (one bit per channel, LSB-aligned)
`0xffffff84`	`NEORV32_XIRQ.IPR`	`31:0`	r/w	Interrupt pending register (one bit per channel, LSB-aligned); writing 0 to a bit clears according pending interrupt
`0xffffff88`	`NEORV32_XIRQ.SCR`	`4:0`	r/w	Channel id (0..31) of firing IRQ (prioritized!); writing *any* value will acknowledge the current interrupt
`0xffffff8c`	-	`31:0`	r/-	*reserved*, read as zero

2.5.20. General Purpose Timer (GPTMR)

Hardware source file(s):	neorv32_gptmr.vhd	
Software driver file(s):	neorv32_gptmr.c	
	neorv32_gptmr.h	
Top entity port:	none	
Configuration generics:	*IO_GPTMR_EN*	implement general purpose timer when *true*
CPU interrupts:	fast IRQ channel 12	transmission done interrupt (see Processor Interrupts)

Theory of Operation

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

The timer is enabled by setting the *GPTMR_CTRL_EN* bit in the device's control register `CTRL`. The `COUNT` register will start incrementing at a programmable rate, which scales the main processor clock. The pre-scaler value is configured via the three *GPTMR_CTRL_PRSCx* control register bits:

Table 28. GPTMR prescaler configuration

`GPTMR_CTRL_PRSCx`	`0b000`	`0b001`	`0b010`	`0b011`	`0b100`	`0b101`	`0b110`	`0b111`
Resulting `clock_prescaler`	2	4	8	64	128	1024	2048	4096

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

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

Timer Interrupt

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

Table 29. GPTMR register map (`struct NEORV32_GPTMR`)

Address	Name [C]	Bit(s), Name [C]	R/W	Function
`0xffffff60`	`NEORV32_GPTMR.CTRL`	`0` *GPTMR_CTRL_EN*	r/w	Timer enable flag
		`1` *GPTMR_CTRL_PRSC0*	r/w	3-bit clock prescaler select
		`2` *GPTMR_CTRL_PRSC1*	r/w	
		`3` *GPTMR_CTRL_PRSC2*	r/w	
		`4` *GPTMR_CTRL_MODE*	r/w	Counter mode: `0`=single-shot, `1`=continuous
`0xffffff64`	`NEORV32_GPTMR.THRES`	`31:0`	r/w	Threshold value register
`0xffffff68`	`NEORV32_GPTMR.COUNT`	`31:0`	r/w	Counter register

2.5.21. Execute In Place Module (XIP)

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

Overview

The execute in place (XIP) module is probably one of the more complicated modules of the NEORV32. The module allows to execute code (and read constant data) directly from a SPI flash memory. Hence, it uses the standard serial peripheral interface (SPI) as transfer protocol under the hood.

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

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

An example program for the XIP module is available in `sw/example/demo_xip`.

Quad-SPI (QSPI) support, which is about 4x times faster, is planned for the future.

SPI Protocol

The XIP module accesses external flash using the standard SPI protocol. The module always sends data MSB-first and provides all of the standard four clock modes (0..3), which are configured via the *XIP_CTRL_CPOL* (clock polarity) and *XIP_CTRL_CPHA* (clock phase) control register bits, respectively. The clock speed of the interface (`xip_clk_o`) is defined by a three-bit clock pre-scaler configured using the *XIP_CTRL_PRSCx* bits:

Table 30. XIP prescaler configuration

`XIP_CTRL_PRSCx`	`0b000`	`0b001`	`0b010`	`0b011`	`0b100`	`0b101`	`0b110`	`0b111`
Resulting `clock_prescaler`	2	4	8	64	128	1024	2048	4096

Based on the *XIP_CTRL_PRSCx* configuration the actual XIP SPI clock frequency f_{XIP} is derived from the processor's main clock f_{main} and is determined by:

f_{XIP} = $f_{main}[Hz]$ / (2 * `clock_prescaler`)

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

High-Speed SPI mode

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

The flash's "read command", which initiates a read access, is defined by the *XIP_CTRL_RD_CMD* control register bits. For most SPI flash memories this is `0x03` for normal SPI mode.

Direct SPI Access

The XIP module allows to initiate *direct* SPI transactions. This feature can be used to configure the attached SPI flash or to perform direct read and write accesses to the flash memory. Two data registers `NEORV32_XIP.DATA_LO` and `NEORV32_XIP.DATA_HI` are provided to send up to 64-bit of SPI data. The `NEORV32_XIP.DATA_HI` register is write-only, so a total of 32-bit receive data is provided. Note that the module handles the chip-select line (`xip_csn_o`) by itself so it is not possible to construct larger consecutive transfers.

The actual data transmission size in bytes is defined by the control register's *XIP_CTRL_SPI_NBYTES* bits. Any configuration from 1 byte to 8 bytes is valid. Other value will result in unpredictable behavior.

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

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

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

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

Address Mapping

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

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

Example: to map the XIP flash to the address space starting at `0x20000000` write a "2" (`0b0010`) to the *XIP_CTRL_XIP_PAGE* control register bits. Any access within `0x20000000 .. 0x2fffffff` will be forwarded to the XIP flash. Note that the SPI access address might wrap around.

Using the FPGA Bitstream Flash also for XIP

You can also use the FPGA's bitstream SPI flash for storing XIP programs. To prevent overriding the bitstream, a certain offset needs to be added to the executable (which might require linker script modifications). To execute the program stored in the SPI flash simply jump to the according base address. For example if the executable starts at flash offset `0x8000` and the XIP flash is mapped to the base address `0x20000000` then add the offset to the base address and use that as jump/call destination (=`0x20008000`).

Using the XIP Mode

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

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

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

- 8-bit command: configured by the *XIP_CTRL_RD_CMD* control register bits ("SPI read command")

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

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

There is no *continuous read* feature (i.e. a burst SPI transmission fetching several data words at once) implemented yet.

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

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

If the XIP module is disabled (*XIP_CTRL_EN* = `0`) any accesses to the programmed XIP memory segment are ignored by the module and might be forwarded to the processor's external memory interface (if implemented) or will cause a bus exception. If the XIP module is enabled (*XIP_CTRL_EN* = `1`) but XIP mode is not enabled yet (*XIP_CTRL_XIP_EN* = '0') any access to the programmed XIP memory segment will raise a bus exception.

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

Table 31. XIP register map (`struct NEORV32_XIP`)

Address	Name [C]	Bit(s), Name [C]	R/W	Function
`0xffffff40`	`NEORV32_XIP.CTRL`	`0` *XIP_CTRL_EN*	r/w	XIP module enable
		`1` *XIP_CTRL_PRSC0*	r/w	3-bit SPI clock prescaler select
		`2` *XIP_CTRL_PRSC1*	r/w	
		`3` *XIP_CTRL_PRSC2*	r/w	
		`4` *XIP_CTRL_CPOL*	r/w	SPI clock polarity
		`5` *XIP_CTRL_CPHA*	r/w	SPI clock phase
		`9:6` *XIP_CTRL_SPI_NBYTES_MSB* : *XIP_CTRL_SPI_NBYTES_LSB*	r/w	Number of bytes in SPI transaction (1..9)
		`10` *XIP_CTRL_XIP_EN*	r/w	XIP mode enable
		`12:11` *XIP_CTRL_XIP_ABYTES_MSB* : *XIP_CTRL_XIP_ABYTES_LSB*	r/w	Number of address bytes for XIP flash (minus 1)
		`20:13` *XIP_CTRL_RD_CMD_MSB* : *XIP_CTRL_RD_CMD_LSB*	r/w	Flash read command
		`24:21` *XIP_CTRL_XIP_PAGE_MSB* : *XIP_CTRL_XIP_PAGE_LSB*	r/w	XIP memory page
		`25` *XIP_CTRL_SPI_CSEN*	r/w	Allow SPI chip-select to be actually asserted when set
		`26` *XIP_CTRL_HIGHSPEED*	r/w	enable SPI high-speed mode (ignoring *XIP_CTRL_PRSC*)
		`29:27`	r/-	*reserved*, read as zero
		`30` *XIP_CTRL_PHY_BUSY*	r/-	SPI PHY busy when set
		`31` *XIP_CTRL_XIP_BUSY*	r/-	XIP access in progress when set
`0xffffff44`	*reserved*	`31:0`	r/-	*reserved*, read as zero
`0xffffff48`	`NEORV32_XIP.DATA_LO`	`31:0`	r/w	Direct SPI access - data register low

Address	Name [C]	Bit(s), Name [C]	R/W	Function
`0xffffff4C`	`NEORV32_XIP.DATA_HI`	`31:0`	-/w	Direct SPI access - data register high; write access triggers SPI transfer

2.5.22. System Configuration Information Memory (SYSINFO)

Hardware source file(s):	neorv32_sysinfo.vhd	
Software driver file(s):	neorv32.h	
Top entity port:	none	
Configuration generics:	*	most of the top's configuration generics
CPU interrupts:	none	

Theory of Operation

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

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

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

Table 32. SYSINFO register map (`struct NEORV32_SYSINFO`)

Address	Name [C]	Function
`0xffffffe0`	`NEORV32_SYSINFO.CLK`	clock speed in Hz (via top's *CLOCK_FREQUENCY* generic)
`0xffffffe4`	-	*reserved*, read as zero
`0xffffffe8`	`NEORV32_SYSINFO.SOC`	specific SoC configuration (see SYSINFO - SoC Configuration)
`0xffffffec`	`NEORV32_SYSINFO.CACHE`	cache configuration information (see SYSINFO - Cache Configuration)
`0xfffffff0`	`NEORV32_SYSINFO.ISPACE_BASE`	instruction address space base (via package's `ispace_base_c` constant)
`0xfffffff4`	`NEORV32_SYSINFO.IMEM_SIZE`	internal IMEM size in bytes (via top's *MEM_INT_IMEM_SIZE* generic)
`0xfffffff8`	`NEORV32_SYSINFO.DSPACE_BASE`	data address space base (via package's `sdspace_base_c` constant)
`0xfffffffc`	`NEORV32_SYSINFO.DMEM_SIZE`	internal DMEM size in bytes (via top's *MEM_INT_DMEM_SIZE* generic)

SYSINFO - SoC Configuration

Table 33. SYSINFO_SOC bits

Bit	Name [C]	Function
`0`	*SYSINFO_SOC_BOOTLOADER*	set if the processor-internal bootloader is implemented (via top's *INT_BOOTLOADER_EN* generic)
`1`	*SYSINFO_SOC_MEM_EXT*	set if the external Wishbone bus interface is implemented (via top's *MEM_EXT_EN* generic)
`2`	*SYSINFO_SOC_MEM_INT_IMEM*	set if the processor-internal DMEM implemented (via top's *MEM_INT_DMEM_EN* generic)
`3`	*SYSINFO_SOC_MEM_INT_DMEM*	set if the processor-internal IMEM is implemented (via top's *MEM_INT_IMEM_EN* generic)
`4`	*SYSINFO_SOC_MEM_EXT_ENDIAN*	set if external bus interface uses BIG-endian byte-order (via top's *MEM_EXT_BIG_ENDIAN* generic)
`5`	*SYSINFO_SOC_ICACHE*	set if processor-internal instruction cache is implemented (via top's *ICACHE_EN* generic)
`13`	*SYSINFO_SOC_IS_SIM*	set if processor is being **simulated** (□□ not guaranteed)
`14`	*SYSINFO_SOC_OCD*	set if on-chip debugger implemented (via top's *ON_CHIP_DEBUGGER_EN* generic)
`15`	*SYSINFO_SOC_HW_RESET*	set if a dedicated hardware reset of all core registers is implemented (via package's `dedicated_reset_c` constant)
`16`	*SYSINFO_SOC_IO_GPIO*	set if the GPIO is implemented (via top's *IO_GPIO_EN* generic)
`17`	*SYSINFO_SOC_IO_MTIME*	set if the MTIME is implemented (via top's *IO_MTIME_EN* generic)
`18`	*SYSINFO_SOC_IO_UART0*	set if the primary UART0 is implemented (via top's *IO_UART0_EN* generic)
`19`	*SYSINFO_SOC_IO_SPI*	set if the SPI is implemented (via top's *IO_SPI_EN* generic)
`20`	*SYSINFO_SOC_IO_TWI*	set if the TWI is implemented (via top's *IO_TWI_EN* generic)
`21`	*SYSINFO_SOC_IO_PWM*	set if the PWM is implemented (via top's *IO_PWM_NUM_CH* generic)
`22`	*SYSINFO_SOC_IO_WDT*	set if the WDT is implemented (via top's *IO_WDT_EN* generic)

Bit	Name [C]	Function
23	*SYSINFO_SOC_IO_CFS*	set if the custom functions subsystem is implemented (via top's *IO_CFS_EN* generic)
24	*SYSINFO_SOC_IO_TRNG*	set if the TRNG is implemented (via top's *IO_TRNG_EN* generic)
25	*SYSINFO_SOC_IO_SLINK*	set if the SLINK is implemented (via top's *SLINK_NUM_TX* and/or *SLINK_NUM_RX* generics)
26	*SYSINFO_SOC_IO_UART1*	set if the secondary UART1 is implemented (via top's *IO_UART1_EN* generic)
27	*SYSINFO_SOC_IO_NEOLED*	set if the NEOLED is implemented (via top's *IO_NEOLED_EN* generic)
28	*SYSINFO_SOC_IO_XIRQ*	set if the XIRQ is implemented (via top's *XIRQ_NUM_CH* generic)
29	*SYSINFO_SOC_IO_GPTMR*	set if the GPTMR is implemented (via top's *IO_GPTMR_EN* generic)
30	*SYSINFO_SOC_IO_XIP*	set if the XIP module is implemented (via top's *IO_XIP_EN* generic)

SYSINFO - Cache Configuration

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

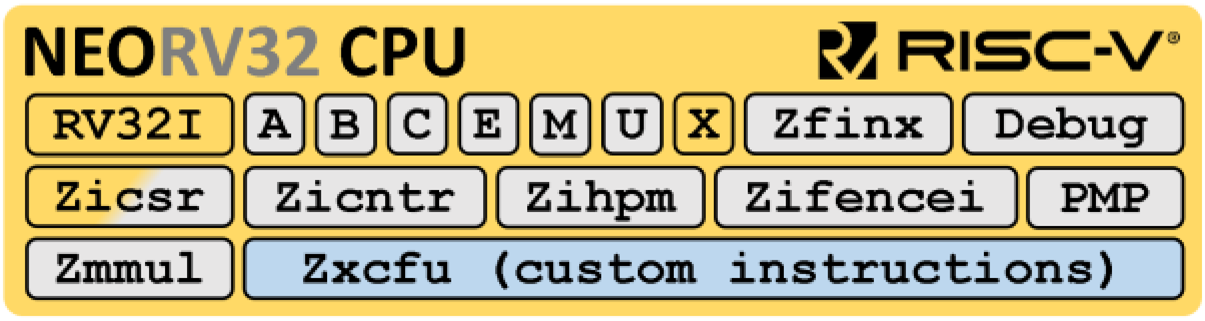
Table 34. SYSINFO_CACHE bits

Bit	Name [C]	Function
`3:0`	*SYSINFO_CACHE_IC_BLOCK_SIZE_3* : *SYSINFO_CACHE_IC_BLOCK_SIZE_0*	*log2*(i-cache block size in bytes), via top's *ICACHE_BLOCK_SIZE* generic
`7:4`	*SYSINFO_CACHE_IC_NUM_BLOCKS_3* : *SYSINFO_CACHE_IC_NUM_BLOCKS_0*	*log2*(i-cache number of cache blocks), via top's *ICACHE_NUM_BLOCKS* generic
`11:9`	*SYSINFO_CACHE_IC_ASSOCIATIVITY_3* : *SYSINFO_CACHE_IC_ASSOCIATIVITY_0*	*log2*(i-cache associativity), via top's *ICACHE_ASSOCIATIVITY* generic
`15:12`	*SYSINFO_CACHE_IC_REPLACEMENT_3* : *SYSINFO_CACHE_IC_REPLACEMENT_0*	i-cache replacement policy (`0001` = LRU if associativity > 0)
`32:16`	-	zero, reserved for d-cache

[3] Pull high if not used.

[4] If the on-chip debugger is not implemented (*ON_CHIP_DEBUGGER_EN* = false) `jtag_tdi_i` is directly forwarded to `jtag_tdo_o` to maintain the JTAG chain.

Chapter 3. NEORV32 Central Processing Unit (CPU)



Section Structure

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

Key Features

- 32-bit little-endian, multi-cycle, in-order `rv32` RISC-V CPU
- Compatible to the RISC-V. **Privileged Architecture - Machine ISA Version 1.12** specifications
- Available Instruction Sets and Extensions:
  - `A` - atomic memory access operations
  - `B` - bit-manipulation instructions
  - `C` - 16-bit compressed instructions
  - `I` - integer base ISA (always enabled)
  - `E` - embedded CPU version (reduced register file size)
  - `M` - integer multiplication and division hardware
  - `U` - less-privileged *user* mode
  - `Zfinx` - single-precision floating-point unit
  - `Zicsr` - control and status register access (privileged architecture)
  - `Zicntr` - CPU base counters
  - `Zihpm` - hardware performance monitors
  - `Zifencei` - instruction stream synchronization

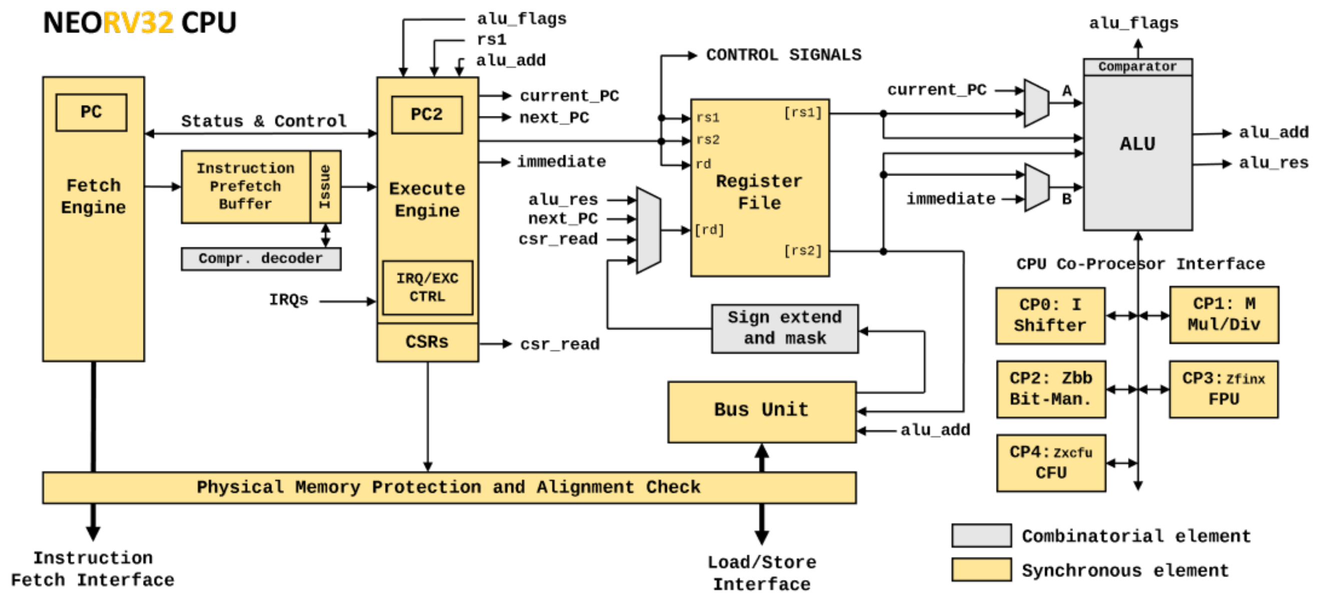
  - `Zmmul` - integer multiplication hardware
  - `Zxcfu` - custom instructions extension
  - `PMP` - physical memory protection
  - `Debug` - CPU Debug Mode (part of the on.chip debugger) including hardware Trigger Module
- RISC-V Compatibility: Compatible to the RISC-V user specifications and a subset of the RISC-V privileged architecture specifications - passes the official RISC-V Architecture Tests (v2+)
- Official RISC-V open-source architecture ID
- Supports *all* of the machine-level Traps, Exceptions and Interrupts from the RISC-V specifications (including bus access exceptions and all unimplemented/illegal/malformed instructions)
  - This is a special aspect on *execution safety* by Full Virtualization
  - Standard RISC-V interrupts (*external*, *timer*, *software*) plus 16 custom *fast* interrupts
- Optional physical memory configuration (PMP), compatible to the RISC-V specifications
- Optional hardware performance monitors (HPM) for application benchmarking
- Separated Bus Interfaces for instruction fetch and data access

It is recommended to use the **NEORV32 Processor** as default top instance 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.

This documentation assumes the reader is familiar with the official RISC-V "User" and "Privileged Architecture" specifications.

3.1. Architecture

The NEORV32 CPU was designed from scratch based only on the official ISA / privileged architecture specifications. The following figure shows the simplified architecture of the CPU.



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

The front-end is responsible for fetching 32-bit chunks of instruction words (one aligned 32-bit instruction, two 16-bit instructions or a mixture if 32-bit instructions are not aligned to 32-bit boundaries). The instruction data is stored to a FIFO queue - the instruction prefetch buffer.

The back-end is responsible for the actual execution of the instruction. It includes an "issue engine", which takes data from the instruction prefetch buffer and assembles 32-bit instruction words (plain 32-bit instruction or decompressed 16-bit instructions) for execution.

Front-end and back-end 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 (accessing memory-mapped devices with high latency), executing multi-cycle ALU operations (like divisions) or when the CPU front-end has to reload the prefetch buffer 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 (*including* fetch) 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 overlapping operation of fetch and execute) at a reduced hardware footprint (due to the multi-cycle concept).

As a Von-Neumann machine, the CPU provides independent interfaces for instruction fetch and data access. These two bus interfaces are merged into a single processor-internal bus via a prioritizing bus switch (data accesses have higher priority). Hence, ALL memory locations including peripheral devices are mapped to a single unified 32-bit address space.

3.2. Full Virtualization

Just like the RISC-V ISA the NEORV32 aims to provide *maximum virtualization* capabilities on CPU and SoC level to allow a high standard of **execution safety**. The CPU supports **all** traps specified by the official RISC-V specifications. [5] Thus, the CPU provides defined hardware fall-backs via traps for any expected and unexpected situation (e.g. executing a malformed instruction or accessing a non-allocated memory address). For any kind of trap the core is always in a defined and fully synchronized state throughout the whole architecture (i.e. there are no out-of-order operations that might have to be reverted). This allows a defined and predictable execution behavior at any time improving overall execution safety.

Execution Safety - NEORV32 Virtualization Features

- Due to the acknowledged memory accesses the CPU is *always* sync with the memory system (i.e. there is no speculative execution / no out-of-order states).
- The CPU supports *all* RISC-V compatible bus exceptions including access exceptions, which are triggered if an accessed address does not respond or encounters an internal device error during access.
- Accessed memory addresses (plain memory, but also memory-mapped devices) need to respond within a fixed time window. Otherwise a bus access exception is raised.
- The RISC-V specs. state that executing an malformed instruction results in unpredictable behavior. As an additional execution safety feature the NEORV32 CPU ensures that *all* unimplemented/malformed/illegal instructions do raise an illegal instruction exceptions and do not commit any state-changing operation (like writing registers or triggering memory operations).
- To be continued…

3.3. RISC-V Compatibility

The NEORV32 CPU passes the tests of the *RISC-V Architecture Test Framework*. This framework is used to check RISC-V implementations for compatibility with the official RISC-V ISA specifications. The NEORV32 port of this test framework has been moved to a separate repository: https://github.com/stnolting/neorv32-verif

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/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            ... IGNORED ①
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
Check fence-01          ... OK
--------------------------------
OK: 39/39 RISCV_TARGET=neorv32 RISCV_DEVICE=I XLEN=32
```

① Test is skipped due to a GHDL simulation issue.

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/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
```

3.3.1. RISC-V Incompatibility Issues and Limitations

This list shows the currently identified issues regarding full RISC-V-compatibility.

Read-Only "Read-Write" CSRs

The `misa` and `mtval` CSRs in the NEORV32 are *read-only*. Any machine-mode write access to them is ignored and will *not* cause any exceptions or side-effects to maintain RISC-V compatibility.

Physical Memory Protection

The RISC-V-compatible NEORV32 Machine Physical Memory Protection CSRs only implements the **TOR** (top of region) mode and only up to 16 PMP regions. Furthermore, the `pmpcfg`'s *lock bits* only lock the according PMP entry and not the entries below. All region rules are checked in parallel **without** prioritization so for identical memory regions the most restrictive PMP rule will be enforced.

Atomic Memory Operations

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

No HW-Support of Misaligned Memory Accesses

The CPU does not support the resolution of unaligned memory access by the hardware. This is not a RISC-V-compatibility issue but an important thing to know. Any kind of unaligned memory access will raise an exception to allow a software-based emulation.

3.4. CPU Top Entity - Signals

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

Table 35. NEORV32 CPU top entity signals

Signal	Width	Dir.	Description
			Global Signals
`clk_i`	1	in	global clock line, all registers triggering on rising edge
`rstn_i`	1	in	global reset, low-active
`sleep_o`	1	out	CPU is in sleep mode when set
`debug_o`	1	out	CPU is in debug mode when set
			Instruction Bus Interface
`i_bus_addr_o`	32	out	access address
`i_bus_rdata_i`	32	in	read data
`i_bus_wdata_o`	32	out	write data (always zero)
`i_bus_ben_o`	4	out	byte enable
`i_bus_we_o`	1	out	write transaction (always zero)
`i_bus_re_o`	1	out	read transaction
`i_bus_lock_o`	1	out	exclusive access request (always zero)
`i_bus_ack_i`	1	in	bus transfer acknowledge from accessed peripheral
`i_bus_err_i`	1	in	bus transfer terminate from accessed peripheral
`i_bus_fence_o`	1	out	indicates an executed `fence.i` instruction
`i_bus_priv_o`	1	out	current *effective* CPU privilege level (`0` user, `1` machine or debug)
			Data Bus Interface
`d_bus_addr_o`	32	out	access address
`d_bus_rdata_i`	32	in	read data
`d_bus_wdata_o`	32	out	write data
`d_bus_ben_o`	4	out	byte enable
`d_bus_we_o`	1	out	write transaction
`d_bus_re_o`	1	out	read transaction
`d_bus_lock_o`	1	out	exclusive access request
`d_bus_ack_i`	1	in	bus transfer acknowledge from accessed peripheral

Signal	Width	Dir.	Description
d_bus_err_i	1	in	bus transfer terminate from accessed peripheral
d_bus_fence_o	1	out	indicates an executed fence instruction
d_bus_priv_o	1	out	current *effective* CPU privilege level (0 user, 1 machine or debug)
System Time (for time[h] CSR)			
time_i	64	in	system time input from Machine System Timer (MTIME)
Interrupts, RISC-V-compatible (Traps, Exceptions and Interrupts)			
msw_irq_i	1	in	RISC-V machine software interrupt
mext_irq_i	1	in	RISC-V machine external interrupt
mtime_irq_i	1	in	RISC-V machine timer interrupt
Interrupts, NEORV32-specific (Traps, Exceptions and Interrupts)			
firq_i	16	in	fast interrupt request signals
Enter Debug Mode Request (On-Chip Debugger (OCD))			
db_halt_req_i	1	in	request CPU to halt and enter debug mode

3.5. CPU Top Entity - Generics

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

CPU_BOOT_ADDR	*std_ulogic_vector(31 downto 0)*	*no default value*
This address defines the reset address at which the CPU starts fetching instructions after reset. In terms of the NEORV32 processor, this generic is configured with the base address of the bootloader ROM (default) or with the base address of the processor-internal instruction memory (IMEM) if the bootloader is disabled (*INT_BOOTLOADER_EN* = *false*). See section Address Space for more information.		

CPU_DEBUG_ADDR	*std_ulogic_vector(31 downto 0)*	*no default value*
This address defines the entry address for the "execution based" on-chip debugger. By default, this generic is configured with the base address of the debugger memory. See section On-Chip Debugger (OCD) for more information.		

CPU_EXTENSION_RISCV_DEBUG	*boolean*	*no default value*
Implement RISC-V-compatible "debug" CPU operation mode. See section CPU Debug Mode for more information.		

3.6. Instruction Sets and Extensions

The basic NEORV32 is a RISC-V `rv32i` architecture that provides several *optional* 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* and *The RISC-V Instruction Set Manual Volume II: Privileged Architecture*, which are available in the projects `docs/references` folder.

Discovering ISA Extensions

The CPU can discover available ISA extensions via the **`misa`** & **`mxisa`** CSRs or by executing an instruction and checking for an *illegal instruction exception* (→ Full Virtualization).

Executing an instruction from an extension that is not supported yet or that is currently not enabled (via the according top entity generic) will raise an illegal instruction exception.

3.6.1. `A` - Atomic Memory Access

Atomic memory access instructions allow more sophisticated memory operations like implementing semaphores and mutexes. The RICS-C specs. defines a specific *atomic* extension that provides instructions for atomic memory accesses. The `A` ISA extension is enabled if 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 raise an illegal instruction exception.

The **load-reservate** instruction behaves as a "normal" load-word instruction (`lw`) but will also set a CPU-internal *data memory access lock*. Executing a **store-conditional** behaves as "normal" store-word instruction (`sw`) that will only conduct an actual memory write operations if the lock is still intact. Additionally, the store-conditional instruction will also return the lock state (returns zero if the lock is still intact or non-zero if the lock has been broken). After the execution of the `sc` instruction, the lock is automatically removed.

The lock is broken if at least one of the following conditions occur: . executing any data memory access instruction other than `lr.w` . raising *any* t (for example an interrupt or a memory access exception)

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

3.6.2. `B` - Bit-Manipulation Operations

The `B` ISA extension adds instructions for bit-manipulation operations. This extension is enabled if the *CPU_EXTENSION_RISCV_B* configuration generic is *true*. The official RISC-V specifications can be found here: https://github.com/riscv/riscv-bitmanip A copy of the spec is also available in `docs/references`.

The NEORV32 `B` ISA extension includes the following sub-extensions (according to the RISC-V bit-manipulation spec. v.093) and their corresponding instructions:

- **`Zba` - Address-generation instructions**
  - `sh1add` `sh2add` `sh3add`
- **`Zbb` - Basic bit-manipulation instructions**
  - `andn` `orn` `xnor`
  - `clz` `ctz` `cpop`
  - `max` `maxu` `min` `minu`
  - `sext.b` `sext.h` `zext.h`
  - `rol` `ror` `rori`
  - `orc.b` `rev8`
- **`Zbc` - Carry-less multiplication instructions**
  - `clmul` `clmulh` `clmulr`
- **`Zbs` - Single-bit instructions**
  - `bclr` `bclri`
  - `bext` `bexti`
  - `bext` `binvi`
  - `bset` `bseti`

By default, the bit-manipulation unit uses an *iterative* approach to compute shift-related operations like `clz` and `rol`. To increase performance (at the cost of additional hardware resources) the *FAST_SHIFT_EN* generic can be enabled to implement full-parallel logic (like barrel shifters) for all shift-related `B` instructions.

The `B` extension is frozen and officially ratified. However, there is no software support for this extension in the upstream GCC RISC-V port yet. An intrinsic library is provided to utilize the provided `B` extension features from C-language code (see `sw/example/bitmanip_test`) to circumvent this.

3.6.3. C - Compressed Instructions

The *compressed* ISA extension provides 16-bit encodings of commonly used instructions to reduce code space size. The `C` extension is 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`

When the compressed instructions extension is enabled, branches to an *unaligned* and *uncompressed* instruction require an additional instruction fetch to load the according second half-word of that instruction. The performance can be increased again 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` and `-falign-jumps=4` compile flags (via the makefile).

3.6.4. E - Embedded CPU

The embedded CPU extensions reduces the size of the general purpose register file from 32 entries to 16 entries to decrease physical hardware requirements (for example block RAM). This extensions is enabled when the *CPU_EXTENSION_RISCV_E* configuration generic is *true*. Accesses to registers beyond `x15` will raise and *illegal instruction exception*. This extension does not add any additional instructions or features.

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

3.6.5. I - Base Integer ISA

The CPU always supports the complete `rv32i` base integer instruction set. This base set is always enabled regardless of the setting of the remaining exceptions. The base instruction set includes the following instructions:

- immediate: `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`

In order to keep the hardware footprint low, the CPU's shift unit uses a bit-serial approach. Hence, shift operations take up to 32 cycles (plus overhead) depending on the actual shift amount. Alternatively, the shift operations can be processed completely in parallel by a fast (but large) barrel shifter if the `FAST_SHIFT_EN` generic is *true*. In that case, shift operations complete within 2 cycles (plus overhead) regardless of the actual shift amount.

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

3.6.6. `M` - Integer Multiplication and Division

Hardware-accelerated integer multiplication and division operations 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`

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* allowing faster execution. Multiplications and divisions always require a fixed amount of cycles to complete - regardless of the input operands.

Regardless of the setting of the *FAST_MUL_EN* generic multiplication and division instructions operate *independently* of the input operands. Hence, there is **no early completion** of multiply by one/zero and divide by zero operations.

3.6.7. `Zmmul` - Integer Multiplication

This is a *sub-extension* of the `M` ISA extension. It implements the multiplication-only operations of the `M` extensions and is intended for size-constrained setups that require hardware-based integer multiplications but not hardware-based divisions, which will be computed entirely in software. This extension requires only ~50% of the hardware utilization of the "full" `M` extension. It is implemented if the *CPU_EXTENSION_RISCV_Zmmul* configuration generic is *true*.

- multiplication: `mul` `mulh` `mulhsu` `mulhu`

If `Zmmul` is enabled, executing any division instruction from the `M` ISA extension (`div`, `divu`, `rem`, `remu`) will raise an *illegal instruction exception*.

Note that `M` and `Zmmul` extensions *cannot* be enabled at the same time.

If your RISC-V GCC toolchain does not (yet) support the `_Zmmul` ISA extensions, it can be "emulated" using a `rv32im` machine architecture and setting the `-mno-div` compiler flag (example `$ make MARCH=rv32im USER_FLAGS+=-mno-div clean_all exe`).

3.6.8. `U` - Less-Privileged User Mode

In addition to the basic (and highest-privileged) machine-mode, the *user-mode* ISA extensions adds a second less-privileged operation mode. It is implemented if the *CPU_EXTENSION_RISCV_U* configuration generic is *true*. Code executed in user-mode cannot access machine-mode CSRs. Furthermore, user-mode access to the address space (like peripheral/IO devices) can be constrained via the physical memory protection (*PMP*). Any kind of privilege rights violation will raise an exception to allow Full Virtualization.

Additional CSRs:

- **`mcounteren`** - machine counter enable to constrain user-mode access to timer/counter CSRs

3.6.9. `X` - NEORV32-Specific (Custom) Extensions

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

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

3.6.10. `Zfinx` Single-Precision Floating-Point Operations

The `Zfinx` floating-point extension is an *alternative* of the standard `F` floating-point ISA extension. The `Zfinx` extensions also uses the integer register file `x` to store and operate on floating-point data instead of a dedicated floating-point register file (hence, `F-in-x`). Thus, the `Zfinx` extension requires less hardware resources and features faster context changes. This also implies that there are NO dedicated `f` register file-related load/store or move instructions. The official RISC-V specifications can be found here: https://github.com/riscv/riscv-zfinx

The NEORV32 floating-point unit used by the `Zfinx` extension is compatible to the *IEEE-754* specifications.

The `Zfinx` extensions only supports single-precision (`.s` instruction suffix), 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`
- compressed instructions: `c.flw` `c.flwsp` `c.fsw` `c.fswsp`

Additional CSRs:

- `fcsr` - FPU control register
- `frm` - rounding mode control
- `fflags` - FPU status flags

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

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

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

3.6.11. `Zicsr` Control and Status Register Access / Privileged Architecture

The CSR access instructions as well as the exception and interrupt system (= the privileged architecture) is implemented when the *CPU_EXTENSION_RISCV_Zicsr* configuration generic is *true*.

If the `Zicsr` extension is disabled the CPU does not provide any *privileged architecture* features at all! In order to provide the full set of privileged functions that are required to run more complex tasks like operating system and to allow a secure execution environment the `Zicsr` extension should be always enabled.

In this case the following instructions are available:

- CSR access: `csrrw` `csrrs` `csrrc` `csrrwi` `csrrsi` `csrrci`
- environment: `mret` `wfi`

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

- `wfi` Instruction **

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

Executing the `wfi` instruction is user-mode will raise an illegal instruction exception if **`mstatus`**.`TW` is set.

3.6.12. `Zicntr` CPU Base Counters

The `Zicntr` ISA extension adds the basic cycle `[m]cycle[h]`), instruction-retired (`[m]instret[h]`) and time (`time[h]`) counters. This extensions is stated is *mandatory* by the RISC-V spec. However, size-constrained setups may remove support for these counters. Section (Machine) Counter and Timer CSRs shows a list of all `Zicntr`-related CSRs. These are available if the `Zicntr` ISA extensions is enabled via the *CPU_EXTENSION_RISCV_Zicntr* generic.

Additional CSRs:

- **`cycle[h]`**, **`mcycle[h]`** - cycle counter
- **`instret[h]`**, **`minstret[h]`** - instructions-retired counter
- **`time[h]`** - system *wall-clock* time

Disabling the `Zicntr` extension does not remove the `time[h]`-driving MTIME unit.

If `Zicntr` is disabled, all accesses to the according counter CSRs will raise an illegal instruction exception.

3.6.13. `Zihpm` Hardware Performance Monitors

In additions to the base cycle, instructions-retired and time counters the NEORV32 CPU provides up to 29 hardware performance monitors (HPM 3..31), which can be used to benchmark applications. Each HPM consists of an N-bit wide counter (split in a high-word 32-bit CSR and a low-word 32-bit CSR), where N is defined via the top's *HPM_CNT_WIDTH* generic (0..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 HPM counters are available if the `Zihpm` ISA extensions is enabled via the *CPU_EXTENSION_RISCV_Zihpm* generic. The actual number of implemented HPM counters is defined by the *HPM_NUM_CNTS* generic.

Additional CSRs:

- `mhpmevent` 3..31 (depending on *HPM_NUM_CNTS*) - event configuration CSRs
- `mhpmcounter[h]` 3..31 (depending on *HPM_NUM_CNTS*) - counter CSRs

The HPM counter CSRs can only be accessed in machine-mode. Hence, the according `mcounteren` CSR bits are always zero and read-only. Any access from less-privileged modes will raise an illegal instruction exception.

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

3.6.14. `Zifencei` Instruction Stream Synchronization

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

- `fence.i`

The `fence.i` instruction resets the CPU's front-end (instruction fetch) and flushes the prefetch buffer. This allows a clean re-fetch of modified instructions from memory. Also, the top's `i_bus_fencei_o` signal is set high for one cycle to inform the memory system (like the i-cache to perform a flush/reload. Any additional flags within the `fence.i` instruction word are ignore by the hardware.

3.6.15. `Zxcfu` Custom Instructions Extension (CFU)

The `Zxcfu` presents a NEORV32-specific *custom RISC-V* ISA extension (`Z` = sub-extension, `x` = platform-specific custom extension, `cfu` = name of the custom extension). When enabled via the *CPU_EXTENSION_RISCV_Zxcfu* configuration generic, this ISA extensions adds the Custom Functions Unit (CFU) to the CPU core. The CFU is a module that allows to add **custom RISC-V instructions** to the processor core.

The CPU is implemented as ALU co-processor and is integrated right into the CPU's pipeline providing minimal data transfer latency as it has direct access to the core's register file. Up to 1024 custom instructions can be implemented within the CFU. These instructions are mapped to an OPCODE space that has been explicitly reserved by the RISC-V spec for custom extensions.

Software can utilize the custom instructions by using *intrinsic functions*, which are inline assembly functions that behave like "regular" C functions.

For more information regarding the CFU see section Custom Functions Unit (CFU).

The CFU / `Zxcfu` ISA extension is intended for application-specific *instructions*. If you like to add more complex accelerators or interfaces that can also operate independently of the CPU take a look at the memory-mapped Custom Functions Subsystem (CFS).

3.6.16. `PMP` Physical Memory Protection

The NEORV32 physical memory protection (PMP) provides an elementary memory protection mechanism that can be used to constrain read, write and execute rights of arbitrary memory regions. The PMP is compatible to the *RISC-V Privileged Architecture Specifications*. For detailed information see the according spec.'s sections.

The NEORV32 PMP only supports **TOR** (top of region) mode, which basically is a "base-and-bound" concept, and only up to 16 PMP regions.

The physical memory protection logic is implemented if the *PMP_NUM_REGIONS* configuration generic is greater than zero. This generic also defines the total number of available configurable protection regions. The minimal granularity of a protected region is defined by the *PMP_MIN_GRANULARITY* generic. Larger granularity will reduce hardware complexity but will also decrease granularity as the minimal region sizes increases. The default value is 4 bytes, which allows a minimal region size of 4 bytes.

If implemented the PMP provides the following additional CSRs:

- `pmpcfg` 0..3 (depending on configuration) - PMP configuration registers, 4 entries per CSR
- `pmpaddr` 0..15 (depending on configuration) - PMP address registers

Operation Summary

Any CPU access address (from the instruction fetch or data access interface) is tested if it matches *any* of the specified PMP regions. If there is a match, the configured access rights are enforced:

- 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

If an access to a protected region does not have the according access rights it will raise the according instruction/load/store *bus access fault* exception.

By default, all PMP checks are enforced for user-mode only. However, PMP rules can also be enforced for machine-mode when the according PMP region has the "LOCK" bit set. This will also prevent any write access to according region's PMP CSRs until the CPU is reset.

Rule Prioritization

All rules are checked in parallel **without** prioritization so for identical memory regions the most restrictive PMP rule will be enforced.

PMP Example Program

A simple PMP example program can be found in `sw/example/demo_pmp`.

Impact on Critical Path

When implementing more PMP regions that a "*certain critical limit*" an **additional register stage** is automatically inserted into the CPU's memory interfaces to keep impact on the critical path as short as minimal as possible. Unfortunately, this will also increase the latency of instruction fetches and data access by one cycle. The *critical limit* can be modified by a constant from the main VHDL package file (`rtl/core/neorv32_package.vhd`, default value = 8):

```
-- "critical" number of PMP regions --
constant pmp_num_regions_critical_c : natural := 8;
```

Reducing the minimal PMP region size / granularity via the *PMP_MIN_GRANULARITY* to entity generic will also reduce hardware utilization and impact on critical path.

3.7. Custom Functions Unit (CFU)

The Custom Functions Unit is the central part of the `Zxcfu` Custom Instructions Extension (CFU) and represents the actual hardware module, which is used to implement *custom RISC-V instructions*. The concept of the NEORV32 CFU has been highly inspired by google's CFU-Playground.

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

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

The CFU is not intended for complex and autonomous functional units that implement complete accelerators like block-based AES de-/encoding). Such accelerator can be implemented within the Custom Functions Subsystem (CFS). A comparison of all chip-internal hardware extension options is provided in the user guide section Adding Custom Hardware Modules.

3.7.1. Custom CFU Instructions - General

The custom instruction utilize a specific instruction space that has been explicitly reserved for user-defined extensions by the RISC-V specifications ("*Guaranteed Non-Standard Encoding Space*"). The NEORV32 CFU uses the *CUSTOM0* opcode to identify custom instructions. The binary encoding of this opcode is `0001011`.

The custom instructions processed by the CFU use the 32-bit **R2-type** RISC-V instruction format, which consists of six bit-fields:

- `funct7`: 7-bit immediate
- `rs2`: address of second source register
- `rs1`: address of first source register
- `funct3`: 3-bit immediate
- `rd`: address of destination register
- `opcode`: always `0001011` to identify custom instructions

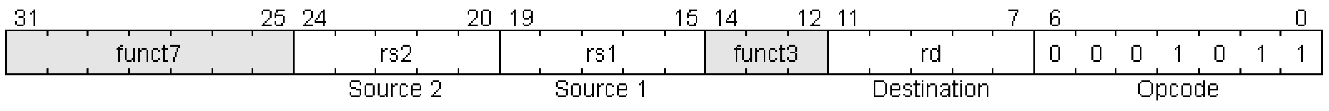


Figure 9. CFU instruction format (RISC-V R2-type)

Obviously, all bit-fields including the immediates have to be static at compile time.

Custom Instructions - Exceptions

The CPU control logic can only check the *CUSTOM0* opcode of the custom instructions to check if the instruction word is valid. It cannot check the `funct3` and `funct7` bit-fields since they are implementation-defined. Hence, a custom CFU instruction can never raise an illegal instruction exception. However, custom will raise an illegal instruction exception if the CFU is not enabled/implemented (i.e. `Zxcfu` ISA extension is not enabled).

The CFU operates on the two source operands and return the processing result to the destination register. The actual instruction to be performed can be defined by using the `funct7` and `funct3` bit fields. These immediate bit-fields can also be used to pass additional data to the CFU like offsets, look-up-tables addresses or shift-amounts. However, the actual functionality is completely user-defined.

3.7.2. Using Custom Instructions in Software

The custom instructions provided by the CFU are included into plain C code by using **intrinsics**. Intrinsics behave like "normal" functions but under the hood they are a set of macros that hide the complexity of inline assembly. Using such intrinsics removes the need to modify the compiler, built-in libraries and the assembler when including custom instructions.

The NEORV32 software framework provides 8 pre-defined custom instructions macros, which are defined in `sw/lib/include/neorv32_cpu_cfu.h`. Each intrinsic provides an implicit definition of the instruction word's `funct3` bit-field:

Listing 4. CFU instruction prototypes

```
neorv32_cfu_cmd0(funct7, rs1, rs2) // funct3 = 000
neorv32_cfu_cmd1(funct7, rs1, rs2) // funct3 = 001
neorv32_cfu_cmd2(funct7, rs1, rs2) // funct3 = 010
neorv32_cfu_cmd3(funct7, rs1, rs2) // funct3 = 011
neorv32_cfu_cmd4(funct7, rs1, rs2) // funct3 = 100
neorv32_cfu_cmd5(funct7, rs1, rs2) // funct3 = 101
neorv32_cfu_cmd6(funct7, rs1, rs2) // funct3 = 110
neorv32_cfu_cmd7(funct7, rs1, rs2) // funct3 = 111
```

Each intrinsic functions always returns a 32-bit value (the processing result). Furthermore, each intrinsic function requires three arguments:

- `funct7` - 7-bit immediate
- `rs2` - source operand 2, 32-bit
- `rs1` - source operand 1, 32-bit

The `funct7` bit-field is used to pass a 7-bit literal to the CFU. The `rs1` and `rs2` arguments to pass the actual data to the CFU. These arguments can be populated with variables or literals. The following example show how to pass arguments when executing `neorv32_cfu_cmd6`: `funct7` is set to all-zero, `rs1` is given the literal *2751* and `rs2` is given a variable that contains the return value from `some_function()`.

Listing 5. CFU instruction usage example

```
uint32_t opb = some_function();
uint32_t res = neorv32_cfu_cmd6(0b0000000, 2751, opb);
```

CFU Example Program

There is a simple example program for the CFU, which shows how to use the *default* CFU hardware module. The example program is located in `sw/example/demo_cfu`.

3.7.3. Custom Instructions Hardware

The actual functionality of the CFU's custom instruction is defined by the logic in the CFU itself. It is the responsibility of the designer to implement this logic within the CFU hardware module `rtl/core/neorv32_cpu_cp_cfu.vhd`.

The CFU hardware module receives the data from instruction word's immediate bit-fields and also the operation data, which is fetched from the CPU's register file.

Listing 6. CFU instruction data passing example

```
uint32_t opb = 0x12345678;
uint32_t res = neorv32_cfu_cmd6(0b0100111, 0x00cafe00, opb);
```

In this example the CFU hardware module receives the two source operands as 32-bit signal and the immediate values as 7-bit and 3-bit signals:

- `rs1_i` (32-bit) contains the data from the `rs1` register (here = `0x00cafe00`)
- `rs2_i` (32-bit) contains the data from the `rs2` register (here = 0x12345678)
- `control.funct3` (3-bit) contains the immediate value from the `funct3` bit-field (here = `0b110`; "cmd6")
- `control.funct7` (7-bit) contains the immediate value from the `funct7` bit-field (here = `0b0100111`)

The CFU executes the according instruction (for example this is selected by the `control.funct3` signal) and provides the operation result in the 32-bit `control.result` signal. The processing can be entirely combinatorial, so the result is available at the end of the current clock cycle. Processing can also take several clock cycles and may also include internal states and memories. As soon as the CFU has completed operations it sets the `control.done` signal high.

CFU Hardware Example & More Details

The default CFU module already implement some exemplary instructions that are used for illustration by the CFU example program. See the CFU's VHDL source file (`rtl/core/neorv32_cpu_cp_cfu.vhd`), which is highly commented to explain the available signals and the handshake with the CPU pipeline.

CFU Execution Time

The CFU is not required to finish processing within a bound time. However, the designer should keep in mind that the CPU is **stalled** until the CFU has finished processing. This also means the CPU cannot react to pending interrupts. Nevertheless, interrupt requests will still be queued.

3.8. Instruction Timing

The instruction timing listed in the table below 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.

Average CPI (cycles per instructions) values for "real applications" like for executing the CoreMark benchmark for different CPU configurations are presented in CPU Performance.

Table 36. Clock cycles per instruction

Class	ISA	Instruction(s)	Execution cycles
ALU	`I/E`	`addi` `slti` `sltiu` `xori` `ori` `andi` `add` `sub` `slt` `sltu` `xor` `or` `and` `lui` `auipc`	2
ALU	`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`	2
ALU	`I/E`	`slli` `srli` `srai` `sll` `srl` `sra`	3 + SA[6]/4 + SA%4; FAST_SHIFT[7]: 4; TINY_SHIFT[8]: 2..32
ALU	`C`	`c.srli` `c.srai` `c.slli`	3 + SA[9]; FAST_SHIFT[10]:
Branches	`I/E`	`beq` `bne` `blt` `bge` `bltu` `bgeu`	Taken: 5 + (ML-1)[11]; Not taken: 3
Branches	`C`	`c.beqz` `c.bnez`	Taken: 5 + (ML-1); Not taken: 3
Jumps / Calls	`I/E`	`jal` `jalr`	5 + (ML-1)
Jumps / Calls	`C`	`c.jal` `c.j` `c.jr` `c.jalr`	5 + (ML-1)
Memory access	`I/E`	`lb` `lh` `lw` `lbu` `lhu` `sb` `sh` `sw`	5 + (ML-2)
Memory access	`C`	`c.lw` `c.sw` `c.lwsp` `c.swsp`	5 + (ML-2)
Memory access	`A`	`lr.w` `sc.w`	5 + (ML-2)
MulDiv	`M`	`mul` `mulh` `mulhsu` `mulhu`	2+32+2; FAST_MUL[12]: 4
MulDiv	`M`	`div` `divu` `rem` `remu`	2+32+2
System	`Zicsr`	`csrrw` `csrrs` `csrrc` `csrrwi` `csrrsi` `csrrci`	3
System	`Zicsr`	`ecall` `ebreak`	3
System	`Zicsr+C`	`c.break`	3
System	`Zicsr`	`wfi`	3
System	`Zicsr`	`mret` `dret`	5
Fence	`I/E`	`fence`	4 + ML
Fence	`Zifencei`	`fence.i`	4 + ML

Class	ISA	Instruction(s)	Execution cycles
Floating-point - artihmetic	`Zfinx`	`fadd.s`	110
Floating-point - artihmetic	`Zfinx`	`fsub.s`	112
Floating-point - artihmetic	`Zfinx`	`fmul.s`	22
Floating-point - compare	`Zfinx`	`fmin.s` `fmax.s` `feq.s` `flt.s` `fle.s`	13
Floating-point - misc	`Zfinx`	`fsgnj.s` `fsgnjn.s` `fsgnjx.s` `fclass.s`	12
Floating-point - conversion	`Zfinx`	`fcvt.w.s` `fcvt.wu.s`	47
Floating-point - conversion	`Zfinx`	`fcvt.s.w` `fcvt.s.wu`	48
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 - arithmetic/logic	`B(Zba)`	`sh1add` `sh2add` `sh3add`	3
Bit-manipulation - shifts	`B(Zbb)`	`clz` `ctz`	3 + 0..32
Bit-manipulation - shifts	`B(Zbb)`	`cpop`	3 + 32
Bit-manipulation - shifts	`B(Zbb)`	`rol` `ror` `rori`	3 + SA
Bit-manipulation - single-bit	`B(Zbs)`	`sbset[i]` `sbclr[i]` `sbinv[i]` `sbext[i]`	3
Bit-manipulation - shifted-add	`B(Zba)`	`sh1add` `sh2add` `sh3add`	3
Bit-manipulation - carry-less multiply	`B(Zbc)`	`clmul` `clmulh` `clmulr`	3 + 32
Custom instructions (CFU)	`Zxcfu`	-	min. 4
Illegal instructions	`Zicsr`	-	min. 2

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.

3.9. Control and Status Registers (CSRs)

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 immediate 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.

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*.

When trying to write to a read-only CSR (like the `time` CSR) or when trying to access a nonexistent CSR or when trying to access a machine-mode CSR from less-privileged user-mode an illegal instruction exception is raised.

CSR reset value: Please note that most of the CSRs do **NOT** provide a dedicated 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). For more information see section CPU Hardware Reset.

CSR Listing

The description of each single CSR provides the following summary:

Table 37. CSR description

Address	*Description*	*ASM alias*
Reset value: *CSR content after hardware reset* (also see CPU Hardware Reset)		
Detailed description		

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.

Debug Mode CSRs

The *debug mode* CSRs are not listed here since they are only accessible in debug mode and not during normal CPU operation. See section CPU Debug Mode CSRs.

CSR Listing Notes

CSRs with the following notes …

- `X`: *custom* - have or are a custom CPU-specific extension (that is allowed by the RISC-V specs)
- `R`: *read-only* - are read-only (in contrast to the originally specified r/w capability)
- `C`: *constrained* - have a constrained compatibility, not all specified bits are implemented

Table 38. NEORV32 Control and Status Registers (CSRs)

Address	Name [ASM]	Name [C]	R/W	Function	Note
Floating-Point CSRs					
0x001	`fflags`	*CSR_FFLAGS*	r/w	Floating-point accrued 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 Configuration CSRs					
0x30a	`menvcfg`	*CSR_MENVCFG*	r/-	Machine environment configuration register - low word	`R`
0x31a	`menvcfgh`	*CSR_MENVCFGH*	r/-	Machine environment configuration register - low word	`R`
Machine Trap Setup CSRs					
0x300	`mstatus`	*CSR_MSTATUS*	r/w	Machine status register - low word	`C`
0x301	`misa`	*CSR_MISA*	r/-	Machine CPU ISA and extensions	`R`
0x304	`mie`	*CSR_MIE*	r/w	Machine interrupt enable register	`X`
0x305	`mtvec`	*CSR_MTVEC*	r/w	Machine trap-handler base address (for ALL traps)	
0x306	`mcounteren`	*CSR_MCOUNTEREN*	r/w	Machine counter-enable register	`C`
0x310	`mstatush`	*CSR_MSTATUSH*	r/-	Machine status register - high word	`C`
Machine Trap Handling CSRs					
0x340	`mscratch`	*CSR_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	`X`
0x343	`mtval`	*CSR_MTVAL*	r/-	Machine bad address or instruction	`R`
0x344	`mip`	*CSR_MIP*	r/w	Machine interrupt pending register	`X`
Machine Physical Memory Protection CSRs					

Address	Name [ASM]	Name [C]	R/W	Function	Note
0x3a0 .. 0x3af	`pmpcfg0` .. `pmpcfg3`	*CSR_PMPCFG0 .. CSR_PMPCFG3*	r/w	Physical memory protection config. for region 0..15	`C`
0x3b0 .. 0x3ef	`pmpaddr0` .. `pmpaddr15`	*CSR_PMPADDR0 .. CSR_PMPADDR15*	r/w	Physical memory protection addr. register region 0..15	
(Machine) Counter and Timer CSRs					
0xb00	`mcycle`	*CSR_MCYCLE*	r/w	Machine cycle counter low word	
0xb02	`minstret`	*CSR_MINSTRET*	r/w	Machine instruction-retired counter low word	
0xb80	`mcycle[h]`	*CSR_MCYCLE*	r/w	Machine cycle counter high word	
0xb82	`minstret[h]`	*CSR_MINSTRET*	r/w	Machine instruction-retired counter 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/-	Instruction-retired counter low word	
0xc80	`cycle[h]`	*CSR_CYCLEH*	r/-	Cycle counter high word	
0xc81	`time[h]`	*CSR_TIMEH*	r/-	System time (from MTIME) high word	
0xc82	`instret[h]`	*CSR_INSTRETH*	r/-	Instruction-retired counter high word	
Hardware Performance Monitors (HPM) CSRs					
0x323 .. 0x33f	`mhpmevent3` .. `mhpmevent31`	*CSR_MHPMEVENT3 .. CSR_MHPMEVENT31*	r/w	Machine performance-monitoring event selector 3..31	`X`
0xb03 .. 0xb1f	`mhpmcounter3` .. `mhpmcounter31`	*CSR_MHPMCOUNTER3 .. CSR_MHPMCOUNTER31*	r/w	Machine performance-monitoring counter 3..31 low word	
0xb83 .. 0xb9f	`mhpmcounter3h` .. `mhpmcounter31h`	*CSR_MHPMCOUNTER3H .. CSR_MHPMCOUNTER31H*	r/w	Machine performance-monitoring counter 3..31 high word	
Machine Counter Setup CSRs					
0x320	`mcountinhibit`	*CSR_MCOUNTINHIBIT*	r/w	Machine counter-enable register	
Machine Information CSRs					
0xf11	`mvendorid`	*CSR_MVENDORID*	r/-	Vendor ID	
0xf12	`marchid`	*CSR_MARCHID*	r/-	Architecture ID	
0xf13	`mimpid`	*CSR_MIMPID*	r/-	Machine implementation ID / version	

Address	Name [ASM]	Name [C]	R/W	Function	Note
0xf14	`mhartid`	*CSR_MHARTID*	r/-	Machine thread ID	
0xf15	`mconfigptr`	*CSR_MCONFIGPTR*	r/-	Machine configuration pointer register	
NEORV32-Specific CSRs					
0xfc0	`mxisa`	*CSR_MXISA*	r/-	NEORV32-specific "extended" machine CPU ISA and extensions	

3.9.1. 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.

`fflags`

0x001 **Floating-point accrued exceptions** `fflags`

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.

`frm`

0x002 **Floating-point dynamic rounding mode** `frm`

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.

`fcsr`

0x003 **Floating-point control and status register** `fcsr`

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.

3.9.2. Machine Configuration CSRs

`menvcfg`

0x30a	**Machine environment configuration register**	`menvcfg`

Reset value: *0x00000000*

The features of this CSR are not implemented yet. The register is read-only. NOTE: This register only exists if the `U` ISA extensions is enabled.

`menvcfgh`

0x31a	**Machine environment configuration register - high word**	`menvcfgh`

Reset value: *0x00000000*

The features of this CSR are not implemented yet. The register is read-only. NOTE: This register only exists if the `U` ISA extensions is enabled.

3.9.3. Machine Trap Setup CSRs

`mstatus`

0x300 **Machine status register** `mstatus`

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).

Table 39. Machine status register

Bit	Name [C]	R/W	Function
21	*CSR_MSTATUS_TW*	r/w	**TW**: Disallows execution of `wfi` instruction in user mode when set; hardwired to zero if user-mode not implemented
12:11	*CSR_MSTATUS_MPP_H* : *CSR_MSTATUS_MPP_L*	r/w	***MPP**: Previous machine privilege level, 11 = machine (M) level, 00 = user (U) level
7	*CSR_MSTATUS_MPIE*	r/w	**MPIE**: Previous machine global interrupt enable flag state
3	*CSR_MSTATUS_MIE*	r/w	**MIE**: 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`.

`misa`

0x301 **ISA and extensions** `misa`

Reset value: *defined*

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).

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.

Table 40. Machine ISA and extension register

Bit	Name [C]	R/W	Function
31:30	*CSR_MISA_MXL_HI_EXT* : *CSR_MISA_MXL_LO_EXT*	r/-	**MXL**: 32-bit architecture indicator (always *01*)
23	*CSR_MISA_X_EXT*	r/-	**X**: extension bit is always set to indicate custom non-standard extensions

Bit	Name [C]	R/W	Function
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 (mul/div) available, set when *CPU_EXTENSION_RISCV_M* enabled
8	*CSR_MISA_I_EXT*	r/-	**I**: CPU base ISA, 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 instruction) available, set when *CPU_EXTENSION_RISCV_C* enabled
0	*CSR_MISA_A_EXT*	r/-	**A**: CPU extension (atomic memory access) available, set when *CPU_EXTENSION_RISCV_A* enabled

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

`mie`

0x304 **Machine interrupt-enable register** `mie`

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):

Table 41. Machine ISA and extension register

Bit	Name [C]	R/W	Function
31:16	*CSR_MIE_FIRQ15E* : *CSR_MIE_FIRQ0E*	r/w	Fast interrupt channel 15..0 enable
11	*CSR_MIE_MEIE*	r/w	**MEIE**: Machine *external* interrupt enable
7	*CSR_MIE_MTIE*	r/w	**MTIE**: Machine *timer* interrupt enable (from *MTIME*)
3	*CSR_MIE_MSIE*	r/w	**MSIE**: Machine *software* interrupt enable

`mtvec`

0x305 **Machine trap-handler base address** `mtvec`

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 (= address mode only). Hence, the trap handler's base address has to be aligned to a 4-byte boundary.

Table 42. Machine trap-handler base address

Bit	R/W	Function
31:2	r/w	**BASE**: 4-byte aligned base address of trap base handler
1:0	r/-	**MODE**: Always zero; BASE defined entry for *all* traps

mcounteren

0x306 **Machine counter enable** mcounteren

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 modes. For example, if user-level code tries to read from a counter/timer CSR without enabled access, an illegal instruction exception is raised. NOTE: If the U ISA extension is not enabled this CSR does not exist.

Table 43. Machine counter enable register

Bit	Name [C]	R/W	Function
31:3	0	r/-	Always zero: user-level code is **not** allowed to read HPM counters
2	*CSR_MCOUNTEREN_IR*	r/w	**IR**: User-level code is allowed to read cycle[h] CSRs when set
1	*CSR_MCOUNTEREN_TM*	r/w	**TM**: User-level code is allowed to read time[h] CSRs when set
0	*CSR_MCOUNTEREN_CY*	r/w	**CY**: User-level code is allowed to read instret[h] CSRs when set

HPM Access

Bits 3 to 31 are used to control user-level access to the Hardware Performance Monitors (HPM) CSRs. In the NEORV32 CPU these bits are hardwired to zero. Hence, user-level software cannot access the HPMs. Accordingly, the pmcounter*[h] CSRs are **not** implemented and any access will raise an illegal instruction exception.

`mstatush`

0x310	**Machine status register - high word**	`mstatush`

Reset value: *0x00000000*

The `mstatush` CSR is compatible to the RISC-V specifications. In combination with **`mstatus`** it shows additional execution state information. The NEORV32 `mstatush` CSR is read-only and all bits are hardwired to zero.

3.9.4. Machine Trap Handling CSRs

mscratch

0x340 **Scratch register for machine trap handlers** `mscratch`

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.

mepc

0x341 **Machine exception program counter** `mepc`

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.

mcause

0x342 **Machine trap cause** `mcause`

Reset value: *UNDEFINED*

The `mcause` CSR is compatible to the RISC-V specifications. It show the cause ID for a taken exception.

Table 44. Machine trap cause register

Bit	R/W	Function
31	r/w	**Interrupt**: `1` if the trap is caused by an interrupt (`0` if the trap is caused by an exception)
30:5	r/-	*Reserved*, read as zero
4:0	r/w	**Trap ID**: see NEORV32 Trap Listing

mtval

0x343 **Machine bad address or instruction** `mtval`

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/store/fetch) or the faulting (decompressed) instruction word itself (for illegal instructions). For all other exceptions (including interrupts) the CSR is set to zero.

Table 45. Machine bad address or instruction register

Trap cause	`mtval` content
misaligned instruction fetch address or instruction fetch access fault	address of faulting instruction fetch
misaligned load address, load access fault, misaligned store address or store access fault	program counter (= address) of faulting instruction
illegal instruction	actual instruction word of faulting instruction (decoded 32-bit instruction word if caused by a compressed instruction)
anything else including interrupts	*0x00000000* (always zero)

The NEORV32 `mtval` CSR is read-only. However, a write access will *NOT* raise an illegal instruction exception.

In case an invalid compressed instruction raised an illegal instruction exception, `mtval` will show the according de-compressed instruction word. To get the "real" 16-bit instruction that caused the exception perform a memory load using the address stored in `mepc`.

`mip`

0x344 **Machine interrupt Pending** `mip`

Reset value: *0x00000000*

The `mip` CSR is compatible to the RISC-V specifications and also provides custom extensions. It shows currently *pending* interrupts. The bits for the standard RISC-V interrupts are read-only. Hence, these interrupts cannot be cleared using the `mip` register and must be cleared/acknowledged within the according interrupt-generating device. The upper 16 bits represent the status of the CPU's fast interrupt request lines (FIRQ). Once triggered, these bit have to be cleared manually by writing zero to the according `mip` bits (in the interrupt handler routine) to clear the current interrupt request.

Table 46. Machine interrupt pending register

Bit	Name [C]	R/W	Function
31:16	*CSR_MIP_FIRQ15P* : *CSR_MIP_FIRQ0P*	r/c	**FIRQxP**: Fast interrupt channel 15..0 pending; cleared request by writing 1
11	*CSR_MIP_MEIP*	r/-	**MEIP**: Machine *external* interrupt pending; *cleared by user-defined mechanism*
7	*CSR_MIP_MTIP*	r/-	**MTIP**: Machine *timer* interrupt pending; cleared by incrementing MTIME's time compare register
3	*CSR_MIP_MSIP*	r/-	**MSIP**: Machine *software* interrupt pending; *cleared by user-defined mechanism*

FIRQ Channel Mapping

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

3.9.5. Machine Physical Memory Protection CSRs

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 implementation of the available *PMP entries*. Each PMP entry consists of an 8-bit `pmpcfg` CSR entry and a complete `pmpaddr*` CSR. See section `PMP` Physical Memory Protection for more information.

If trying to access an PMP-related CSR beyond *PMP_NUM_REGIONS* **no illegal instruction exception** is triggered. The according CSRs are read-only (writes are ignored) and always return zero. However, any access beyond `pmpcfg3` or `pmpaddr15`, which are the last physically implemented registers if *PMP_NUM_REGIONS* == 16, will raise an illegal instruction exception as these CSRs are not implemented at all.

`pmpcfg`

0x3a0 - 0x3a3	**Physical memory protection configuration registers**	`pmpcfg0` - `pmpcfg3`

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 (8-bit per region). The actual number of available `pmpcfg` CSRs and CSR entries is defined by the *PMP_NUM_REGIONS* generic.

Table 47. Physical memory protection configuration register layout (1 entry out of 4)

Bit	Name [C]	R/W	Function
7	*PMPCFG_L*	r/w	**L**: Lock bit, prevents further write accesses, also enforces access rights in machine-mode, can only be cleared by CPU reset
6:5	-	r/-	*reserved*, read as zero
4	*PMPCFG_A_MSB*	r/-	**A**: Mode configuration; only **OFF** (`00`) and **TOR** (`01`) modes are supported, any other value will map back to OFF/TOR as the MSB is hardwired to zero
3	*PMPCFG_A_LSB*	r/w	
2	*PMPCFG_X*	r/w	**X**: Execute permission
1	*PMPCFG_W*	r/w	**W**: Write permission
0	*PMPCFG_R*	r/w	**R**: Read permission

Setting the lock bit `L` **only locks the according PMP entry** and not the PMP entries below!

`pmpaddr`

0x3b0 - 0x3bf	**Physical memory protection address registers**	`pmpaddr0` - `pmpaddr15`

Reset value: *UNDEFINED*

The `pmpaddr*` CSRs are compatible to the RISC-V specifications. They are used to configure bits 33:2 of the PMP region's physical memory address. The actual number of available `pmpaddr` CSRs is defined by the *PMP_NUM_REGIONS* generic.

Table 48. Physical memory protection address register layout

Bit	R/W	Function
31:30	r/-	Hardwired to zero
29 : *log2(PMP_MIN_GRANULARITY)-2*	r/w	Bits 31 downto *log2(PMP_MIN_GRANULARITY)* of the region's address
log2(PMP_MIN_GRANULARITY)-2 : 0	r/-	Hardwired to zero

When configuring the 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*`.

3.9.6. (Machine) Counter and Timer CSRs

The (machine) counters and timers are implemented when the `Zicntr` ISA extensions is enabled (default) via the *CPU_EXTENSION_RISCV_Zicntr* generic.

The *CPU_CNT_WIDTH* generic defines the total size of the CPU's **`cycle[h]`** and **`instret[h]`** / **`mcycle[h]`** and **`minstret[h]`** counter CSRs (low and high words combined); the time CSRs are not affected by this generic. Note that any configuration with *CPU_CNT_WIDTH* less than 64 is not RISC-V compliant.

Effective CPU counter width (`[m]cycle` & `[m]instret`)

If *CPU_CNT_WIDTH* is less than 64 (the default value) and greater than or equal 32, the according MSBs of `[m]cycleh` and `[m]instreth` are read-only and always read as zero. This configuration will also set the *CSR_MXISA_ZXSCNT* flag ("small counters") in the **`mxisa`** CSR.

If *CPU_CNT_WIDTH* is less than 32 and greater than 0, the `[m]cycleh` and `[m]instreth` CSRs are hardwired to zero and any write access to them is ignored. Furthermore, the according MSBs of `[m]cycle` and `[m]instret` are read-only and always read as zero. This configuration will also set the *CSR_MXISA_ZXSCNT* flag ("small counters") in the **`mxisa`** CSR.

If *CPU_CNT_WIDTH* is 0, the **`cycle[h]`** and **`instret[h]`** / **`mcycle[h]`** and **`minstret[h]`** CSRs are hardwired to zero and any write access to them is ignored.

Counter Increment During Debugging

The `[m]cycle[h]` and `[m]instret[h]` counters do not increment when the CPU is in debug mode. See section CPU Debug Mode for more information.

`cycle[h]`

0xc00	**Cycle counter - low word**	`cycle`
0xc80	**Cycle counter - high word**	`cycleh`

Reset value: *UNDEFINED*

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

`time[h]`

0xc01	**System time - low word**	`time`
0xc81	**System time - high word**	`timeh`

Reset value: *UNDEFINED*

The `time[h]` CSR is compatible to the RISC-V specifications. It shows the lower/upper 32-bit of the 64-bit system time. The system time is either generated by the processor-internal *MTIME* system timer unit (if *IO_MTIME_EN* = *true*) or can be provided by an external timer unit via the processor's `mtime_i` signal (if *IO_MTIME_EN* = *false*). CSR is read-only. Change the system time via the *MTIME* unit.

`instret[h]`

0xc02	**Instructions-retired counter - low word**	`instret`
0xc82	**Instructions-retired counter - high word**	`instreth`

Reset value: *UNDEFINED*

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

`mcycle[h]`

0xb00	**Machine cycle counter - low word**	`mcycle`
0xb80	**Machine cycle counter - high word**	`mcycleh`

Reset value: *UNDEFINED*

The `mcycle[h]` CSR is compatible to the RISC-V specifications. It shows the lower/upper 32-bit of the 64-bit cycle counter. The `mcycle[h]` CSR can also be written when in machine mode and is mirrored to the `cycle[h]` CSR.

`minstret[h]`

0xb02	**Machine instructions-retired counter - low word**	`minstret`
0xb82	**Machine instructions-retired counter - high word**	`minstreth`

Reset value: *UNDEFINED*

The `minstret[h]` CSR is compatible to the RISC-V specifications. It shows the lower/upper 32-bit of the 64-bit retired instructions counter. The `minstret[h]` CSR also be written when in machine mode and is mirrored to the `instret[h]` CSR.

3.9.7. Hardware Performance Monitors (HPM) CSRs

The hardware performance monitor CSRs are implemented when the `Zihpm` ISA extension is enabled via the *CPU_EXTENSION_RISCV_Zihpm* generic.

The actually implemented hardware performance logic is configured via the *HPM_NUM_CNTS* top entity generic, which defines the number of implemented performance monitors. Note that always all 28 HPM counter and configuration registers (`mhpmcounter*[h]` and `mhpmevent*`) are implemented, but only the actually configured ones are real registers and not hardwired to zero.

If trying to access an HPM-related CSR beyond *HPM_NUM_CNTS* **no illegal instruction exception is triggered**. The according CSRs are read-only (writes are ignored) and always return zero.

The HPM system only allows machine-mode access. Hence, `hpmcounter*[h]` CSR are not implemented and any access (even) from machine mode will raise an exception. Furthermore, the according bits of **`mcounteren`** used to configure user-mode access to `hpmcounter*[h]` are hard-wired to zero.

The total counter width of the HPMs can be configured before synthesis via the *HPM_CNT_WIDTH* generic (0..64-bit).

The total LSB-aligned HPM counter size (low word CSR + high word CSR) is defined via the *HPM_NUM_CNTS* generic (0..64-bit). If *HPM_NUM_CNTS* is less than 64, all unused MSB-aligned bits are hardwired to zero.

Counter Increment During Debugging

All HPM counters do not increment when the CPU is in debug mode. See section CPU Debug Mode for more information.

`mhpmevent`

0x232 -0x33f	**Machine hardware performance monitor event selector**	`mhpmevent3` - `mhpmevent31`

Reset value: *UNDEFINED*

The `mhpmevent*` CSRs are compatible to the RISC-V specifications. The configuration of these CSR define the architectural events that cause the according `mhpmcounter*[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 or debug mode, no HPM counter will increment at all.

Table 49. HPM event selector

Bit	Name [C]	R/W	Event
0	*HPMCNT_EVENT_CY*	r/w	active clock cycle (not in sleep)
1	-	r/-	*not implemented, always read as zero*
2	*HPMCNT_EVENT_IR*	r/w	retired instruction (compressed or uncompressed)
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	memory data load operation
8	*HPMCNT_EVENT_STORE*	r/w	memory data 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 (synchronous exception or interrupt)
14	*HPMCNT_EVENT_ILLEGAL*	r/w	illegal instruction exception

`mhpmcounter[h]`

0xb03 - 0xb1f	**Machine hardware performance monitor - counter low**	`mhpmcounter3` - `mhpmcounter31`
0xb83 - 0xb9f	**Machine hardware performance monitor - counter high**	`mhpmcounter3h` - `mhpmcounter31h`

Reset value: *UNDEFINED*

The `mhpmcounter*[h]` CSRs are compatible to the RISC-V specifications. These CSRs provide the lower/upper 32- bit of arbitrary event counters. The event(s) that trigger an increment of theses counters are selected via the according `mhpmevent*` CSRs bits.

3.9.8. Machine Counter Setup CSRs

`mcountinhibit`

0x320	**Machine counter-inhibit register**	`mcountinhibit`

Reset value: *UNDEFINED*

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).

Table 50. Machine counter-inhibit register

Bit	Name [C]	R/W	Event
0	*CSR_MCOUNTINHIBIT_IR*	r/w	**IR**: The `[m]instret[h]` CSRs will auto-increment with each committed instruction when set
2	*CSR_MCOUNTINHIBIT_CY*	r/w	**CY**: The `[m]cycle[h]` CSRs will auto-increment with each clock cycle (if CPU is not in sleep state) when set
3:31	*CSR_MCOUNTINHIBIT_HPM3* : *CSR_MCOUNTINHIBIT_HPM31*	r/w	**HPMx**: The `mhpmcount*[h]` CSRs will auto-increment according to the configured `mhpmevent*` selector

3.9.9. Machine Information CSRs

All machine information registers can only be accessed in machine mode and are read-only.

`mvendorid`

0xf11 **Machine vendor ID** `mvendorid`

Reset value: *0x00000000*

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

`marchid`

0xf12 **Machine architecture ID** `marchid`

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).

`mimpid`

0xf13 **Machine implementation ID** `mimpid`

Reset value: *defined*

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).

`mhartid`

0xf14 **Machine hardware thread ID** `mhartid`

Reset value: *defined*

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.

`mconfigptr`

0xf15 **Machine configuration pointer register** `mconfigptr`

Reset value: *0x00000000*

This register holds a physical address (if not zero) that points to the base address of an architecture configuration structure. Software can traverse this data structure to discover information about the harts, the platform, and their configuration. **NOTE: Not assigned yet.**

3.9.10. NEORV32-Specific CSRs

All NEORV32-specific CSRs are mapped to addresses that are explicitly reserved for custom **Machine-Mode, read-only** CSRs (assured by the RISC-V privileged specifications). Hence, these CSRs can only be accessed when in machine-mode. Any access outside of machine-mode will raise an illegal instruction exception.

`mxisa`

0x7c0 **Machine EXTENDED ISA and Extensions register** `mxisa`

Reset value: *defined*

NEORV32-specific read-only CSR that helps machine-mode software to discover `Z*` sub-extensions and CPU options.

Table 51. Machine EXTENDED ISA and Extensions register bits

Bit	Name [C]	R/W	Function
31	*CSR_MXISA_FASTSHIFT*	r/-	fast shifts available when set (via top's *FAST_SHIFT_EN* generic)
30	*CSR_MXISA_FASTMUL*	r/-	fast multiplication available when set (via top's *FAST_MUL_EN* generic)
31:11	-	r/-	*reserved*, read as zero
10	*CSR_MXISA_DEBUGMODE*	r/-	RISC-V CPU `debug_mode` available when set (via top's *ON_CHIP_DEBUGGER_EN* generic)
9	*CSR_MXISA_ZIHPM*	r/-	`Zihpm` (hardware performance monitors) extension available when set (via top's *CPU_EXTENSION_RISCV_Zihpm* generic)
8	*CSR_MXISA_PMP*	r/-	PMP` (physical memory protection) extension available when set (via top's *PMP_NUM_REGIONS* generic)
7	*CSR_MXISA_ZICNTR*	r/-	`Zicntr` extension (`I` sub-extension) available when set - `[m]cycle`, `[m]instret` and `[m]time` CSRs available when set (via top's *CPU_EXTENSION_RISCV_Zicntr* generic)
6	*CSR_MXISA_ZXSCNT*	r/-	Custom extension - *Small* CPU counters: `[m]cycle` & `[m]instret` CSRs have less than 64-bit when set (via top's *CPU_CNT_WIDTH* generic)
5	*CSR_MXISA_ZFINX*	r/-	`Zfinx` extension (`F` sub-/alternative-extension: FPU using `x` registers) available when set (via top's *CPU_EXTENSION_RISCV_Zfinx* generic)
4	-	r/-	*reserved*, read as zero

Bit	Name [C]	R/W	Function
3	*CSR_MXISA_ZXCFU*	r/-	`Zxcfu` extension (custom functions unit for custom RISC-V instructions) available when set (via top's *CPU_EXTENSION_RISCV_Zxcfu* generic)
2	*CSR_MXISA_ZMMUL*	r/-	`Zmmul` extension (`M` sub-extension) available when set (via top's *CPU_EXTENSION_RISCV_Zmmul* generic)
1	*CSR_MXISA_ZIFENCEI*	r/-	`Zifencei` extension (`I` sub-extension) available when set (via top's *CPU_EXTENSION_RISCV_Zifencei* generic)
0	*CSR_MXISA_ZICSR*	r/-	`Zicsr` extension (`I` sub-extension) available when set (via top's *CPU_EXTENSION_RISCV_Zicsr* generic)

3.9.11. Traps, Exceptions and Interrupts

In this document the following 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 `mtvec` CSR. The cause of the according interrupt or exception can be determined via the content of `mcause` CSR. The address that reflects the current program counter when a trap was taken is stored to `mepc` CSR. Additional information regarding the cause of the trap can be retrieved from `mtval` CSR and the processor's Internal Bus Monitor (BUSKEEPER) (for memory access exceptions)

The traps are prioritized. If several *synchronous exceptions* occur at once only the one with highest priority is triggered while all remaining exceptions are ignored. If several *asynchronous exceptions* (interrupts) trigger at once, the one with highest priority is serviced first while the remaining ones stay *pending*. After completing the interrupt handler the interrupt with the second highest priority will get serviced and so on until no further interrupts are pending.

Interrupt Signal Requirements - Standard RISC-V Interrupts

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

Interrupt Signal Requirements - Fast Interrupt Requests

The NEORV32-specific FIRQ request lines are triggered by a one-shot high-level (i.e. rising edge). Each request is buffered in the CPU control unit until the channel is either disabled (by clearing the according `mie` CSR bit) or the request is explicitly cleared (by writing zero to the according `mip` CSR bit).

Instruction Atomicity

All instructions execute as atomic operations - interrupts can only trigger *between* two instructions. So even if there is a permanent interrupt request, exactly one instruction from the interrupt program will be executed before another interrupt handler can start. This allows program progress even if there are permanent interrupt requests.

Memory Access Exceptions

If a load operation causes any exception, the instruction's destination register is *not written* at all. Load exceptions caused by a misalignment or a physical memory protection fault do not trigger a bus/memory read-operation at all. Vice versa, exceptions caused by a store address misalignment or a store physical memory protection fault do not trigger a bus/memory write-operation at all.

Custom Fast Interrupt Request Lines

As a custom extension, the NEORV32 CPU features 16 fast interrupt request (FIRQ) lines via the `firq_i` CPU top entity signals. These interrupts have custom configuration and status flags in the **`mie`** and **`mip`** CSRs and also provide custom trap codes in **`mcause`**. These FIRQs are reserved for NEORV32 processor-internal usage only.

NEORV32 Trap Listing

The following table shows all traps that are currently supported by the NEORV32 CPU. It also shows the prioritization and the CSR side-effects. A more detailed description of the actual trap triggering events is provided in a further table.

Asynchronous exceptions (= interrupts) set the MSB of **`mcause`** while *synchronous exception* (= "software exception") clear the MSB.

Table Annotations

The "Prio." column shows the priority of each trap. The highest priority is 1. The “mcause” column shows the cause ID of the according trap that is written to **`mcause`** CSR. The "[RISC-V]" columns show the interrupt/exception code value from the official RISC-V privileged architecture spec. The "ID [C]" names are defined by the NEORV32 core library (the runtime environment *RTE*) and can be used in plain C code. The “mepc” and “mtval” columns show the value written to **`mepc`** and **`mtval`** CSRs when a trap is triggered:

- **IPC** - address of interrupted instruction (instruction has not been executed yet)
- **PC** - address of instruction that caused the trap
- **ADR** - bad memory access address that caused the trap
- **INST** - the faulting instruction word itself
- **0** - zero

Table 52. NEORV32 Trap Listing

Prio.	`mcause`	[RISC-V]	ID [C]	Cause	`mepc`	`mtval`
			Synchronous Exceptions			
1	`0x00000000`	0.0	*TRAP_CODE_I_MISALIGNED*	instruction address misaligned	**PC**	**ADR**
2	`0x00000001`	0.1	*TRAP_CODE_I_ACCESS*	instruction access bus fault	**PC**	**ADR**
3	`0x00000002`	0.2	*TRAP_CODE_I_ILLEGAL*	illegal instruction	**PC**	**INST**
4	`0x0000000B`	0.11	*TRAP_CODE_MENV_CALL*	environment call from M-mode (`ecall`)	**PC**	**0**
5	`0x00000008`	0.8	*TRAP_CODE_UENV_CALL*	environment call from U-mode (`ecall`)	**PC**	**0**

Prio.	`mcause`	[RISC-V]	ID [C]	Cause	`mepc`	`mtval`
6	`0x00000003`	0.3	*TRAP_CODE_BREAKPOINT*	software breakpoint (`ebreak`)	**PC**	**0**
7	`0x00000006`	0.6	*TRAP_CODE_S_MISALIGNED*	store address misaligned	**PC**	**ADR**
8	`0x00000004`	0.4	*TRAP_CODE_L_MISALIGNED*	load address misaligned	**PC**	**ADR**
9	`0x00000007`	0.7	*TRAP_CODE_S_ACCESS*	store access bus fault	**PC**	**ADR**
10	`0x00000005`	0.5	*TRAP_CODE_L_ACCESS*	load access bus fault	**PC**	**ADR**
			Asynchronous Exceptions (Interrupts)			
11	`0x80000010`	1.16	*TRAP_CODE_FIRQ_0*	fast interrupt request channel 0	**IPC**	**0**
12	`0x80000011`	1.17	*TRAP_CODE_FIRQ_1*	fast interrupt request channel 1	**IPC**	**0**
13	`0x80000012`	1.18	*TRAP_CODE_FIRQ_2*	fast interrupt request channel 2	**IPC**	**0**
14	`0x80000013`	1.19	*TRAP_CODE_FIRQ_3*	fast interrupt request channel 3	**IPC**	**0**
15	`0x80000014`	1.20	*TRAP_CODE_FIRQ_4*	fast interrupt request channel 4	**IPC**	**0**
16	`0x80000015`	1.21	*TRAP_CODE_FIRQ_5*	fast interrupt request channel 5	**IPC**	**0**
17	`0x80000016`	1.22	*TRAP_CODE_FIRQ_6*	fast interrupt request channel 6	**IPC**	**0**
18	`0x80000017`	1.23	*TRAP_CODE_FIRQ_7*	fast interrupt request channel 7	**IPC**	**0**
19	`0x80000018`	1.24	*TRAP_CODE_FIRQ_8*	fast interrupt request channel 8	**IPC**	**0**
20	`0x80000019`	1.25	*TRAP_CODE_FIRQ_9*	fast interrupt request channel 9	**IPC**	**0**
21	`0x8000001a`	1.26	*TRAP_CODE_FIRQ_10*	fast interrupt request channel 10	**IPC**	**0**
22	`0x8000001b`	1.27	*TRAP_CODE_FIRQ_11*	fast interrupt request channel 11	**IPC**	**0**
23	`0x8000001c`	1.28	*TRAP_CODE_FIRQ_12*	fast interrupt request channel 12	**IPC**	**0**
24	`0x8000001d`	1.29	*TRAP_CODE_FIRQ_13*	fast interrupt request channel 13	**IPC**	**0**

Prio.	`mcause`	[RISC-V]	ID [C]	Cause	`mepc`	`mtval`
25	`0x8000001e`	1.30	*TRAP_CODE_FIRQ_14*	fast interrupt request channel 14	**IPC**	**0**
26	`0x8000001f`	1.31	*TRAP_CODE_FIRQ_15*	fast interrupt request channel 15	**IPC**	**0**
27	`0x8000000B`	1.11	*TRAP_CODE_MEI*	machine external interrupt (MEI)	**IPC**	**0**
28	`0x80000003`	1.3	*TRAP_CODE_MSI*	machine software interrupt (MSI)	**IPC**	**0**
29	`0x80000007`	1.7	*TRAP_CODE_MTI*	machine timer interrupt (MTI)	**IPC**	**0**

The following table provides a summarized description of the actual events for triggering a specific trap.

Table 53. NEORV32 Trap Description

Trap ID [C]	Triggered when …
TRAP_CODE_I_MISALIGNED	fetching a 32-bit instruction word that is not 32-bit-aligned (*see note below!*)
TRAP_CODE_I_ACCESS	bus timeout or bus error during instruction word fetch
TRAP_CODE_I_ILLEGAL	trying to execute an invalid instruction word (malformed or not supported) or on a privilege violation
TRAP_CODE_MENV_CALL	executing `ecall` instruction in machine-mode
TRAP_CODE_UENV_CALL	executing `ecall` instruction in user-mode
TRAP_CODE_BREAKPOINT	executing `ebreak` instruction
TRAP_CODE_S_MISALIGNED	storing data to an address that is not naturally aligned to the data size (byte, half, word) being stored
TRAP_CODE_L_MISALIGNED	loading data from an address that is not naturally aligned to the data size (byte, half, word) being loaded
TRAP_CODE_S_ACCESS	bus timeout or bus error during load data operation
TRAP_CODE_L_ACCESS	bus timeout or bus error during store data operation
TRAP_CODE_FIRQ_0 … *TRAP_CODE_FIRQ_15*	caused by interrupt-condition of processor-internal modules, see NEORV32-Specific Fast Interrupt Requests
TRAP_CODE_MEI	user-defined processor-external source (via dedicated top-entity signal)
TRAP_CODE_MSI	user-defined processor-external source (via dedicated top-entity signal)

Trap ID [C]	Triggered when …
TRAP_CODE_MTI	processor-internal machine timer overflow OR user-defined processor-external source (via dedicated top-entity signal)

Misaligned Instruction Address Exception

For 32-bit-only instructions (= no `C` extension) the misaligned instruction exception is raised if bit 1 of the fetch address is set (i.e. not on a 32-bit boundary). If the `C` extension is implemented there will never be a misaligned instruction exception *at all*. In both cases bit 0 of the program counter (and all related CSRs) is hardwired to zero.

3.9.12. Bus Interface

The NEORV32 CPU implements a 32-bit machine with separated instruction and data interfaces making the CPU a **Harvard Architecture**: the *instruction fetch interface* (`i_bus_*`) is used for fetching instruction and the *data access interface* (`d_bus_*`) is used to access data via load and store operations. Each of this interfaces can access an address space of up to 2^{32} bytes (4GB). The following table shows the signals of the data and instruction interfaces as seen from the CPU (`*_o` signals are driven by the CPU / outputs, `*_i` signals are read by the CPU / inputs). Both interfaces use the same protocol.

Table 54. CPU bus interfaces ()

Signal	Width	Direction	Description
`i/d_bus_addr_o`	32	out	access address
`i/d_bus_rdata_i`	32	in	data input for read operations
`i/d_bus_wdata_o`	32	out	data output for write operations
`i/d_bus_ben_o`	4	out	byte enable signal for write operations
`i/d_bus_we_o`	1	out	bus write access (always zero for instruction fetches)
`i/d_bus_re_o`	1	out	bus read access
`i/d_bus_lock_o`	1	out	exclusive access request
`i/d_bus_ack_i`	1	in	accessed peripheral indicates a successful completion of the bus transaction
`i/d_bus_err_i`	1	in	accessed peripheral indicates an error during the bus transaction
`i/d_bus_fence_o`	1	out	this signal is set for one cycle when the CPU executes an instruction/data fence operation
`i/d_bus_priv_o`	2	out	current CPU privilege level

Pipelined Transfers

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" (pending) at once. However, this is no real drawback. The minimal possible latency for a single access is two cycles, which equals the CPU's minimal execution latency for a single instruction.

Unaligned Memory Accesses

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

Protocol

An actual bus request is triggered either by the `*_bus_re_o` signal (for reading data) or by the `*_bus_we_o` signal (for writing data). In case of a request, one of these signals is high for exactly one cycle. The transaction is completed when the accessed peripheral/memory either sets the `*_bus_ack_i` signal (→ successful completion) or the `*_bus_err_i` signal (→ failed completion). These bus response signal are also set only for one cycle active. An error indicated by the `*_bus_err_i` signal will raise the according "instruction bus access fault" or "load/store bus access fault" exception.

Minimal Response Latency

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. However, in order to shorten the critical path such "asynchronous" completion should be avoided. The default NEORV32 processor-internal modules provide exactly **one cycle delay** between initiation and completion of transfers.

Maximal Response Latency

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

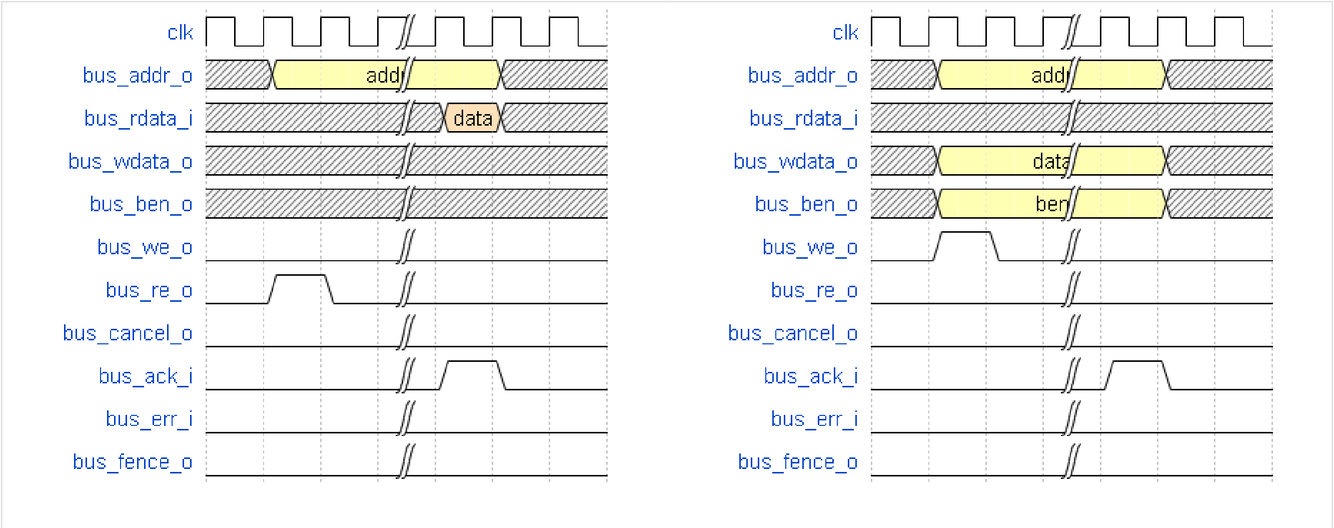
If any bus operations times out (for example when accessing "address space holes") the BUSKEEPER will issue a bus error to the CPU that will raise the according instruction fetch or data access bus exception. Note that **the bus keeper does not track external accesses via the external memory bus interface**. However, the external memory bus interface also provides an *optional* bus timeout (see section Processor-External Memory Interface (WISHBONE) (AXI4-Lite)).

Interface Response

Please note that any CPU access via the data or instruction interface has to be terminated either by asserting the CPU's *_bus_ack_i` or `*_bus_err_i` signal. Otherwise the CPU will be stalled permanently. The BUSKEEPER ensures that any kind of access is always properly terminated.

Exemplary Bus Accesses

Table 55. Example bus accesses: see read/write access description below



Read access Write access

Write Access

For a write access, the access 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 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 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).

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.

Exclusive (Atomic) Access

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

The CPU starts an exclusive access to memory via the *load-reservate instruction* (`lr.w`). This instruction will set the CPU-internal *exclusive access lock*, which directly drives the `d_bus_lock_o`. 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 executes a *store-conditional instruction* (`sc.w`) the *CPU-internal exclusive access lock* is evaluated to check if the exclusive access was successful. If the lock is still OK, the instruction will write-back zero and will allow the according store operation to the memory system. If the lock is broken, the instruction will write-back non-zero and will not generate an actual memory store operation.

The CPU-internal exclusive access lock is broken if at least one of the situations appear.

- when executing any other memory-access operation than `lr.w`
- when any trap (sync. or async.) is triggered (for example to force a context switch)
- when the memory system signals a bus error (via the `bus_err_i` signal)

For more information regarding the SoC-level behavior and requirements of atomic operations see section 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 (for example a cache flush and refill).

3.9.13. CPU Hardware Reset

In order to reduce routing constraints (and by this the actual hardware requirements), most uncritical registers of the NEORV32 CPU as well as most register of the whole NEORV32 Processor do not use **a dedicated hardware reset**. "Uncritical registers" in this context means that the initial value of these registers after power-up is not relevant for a defined CPU boot process.

Rationale

A good example to illustrate the concept of uncritical registers is a pipelined processing engine. Each stage of the engine features an N-bit *data register* and a 1-bit *status register*. The status register is set when the data in the according data register is valid. At the end of the pipeline the status register might trigger a write-back of the processing result to some kind of memory. The initial status of the data registers after power-up is irrelevant as long as the status registers are all reset to a defined value that indicates there is no valid data in the pipeline's data register. Therefore, the pipeline data register do no require a dedicated reset as they do not control the actual operation (in contrast to the status register). This makes the pipeline data registers from this example "uncritical registers".

NEORV32 CPU Reset

In terms of the NEORV32 CPU, there are several pipeline registers, state machine registers and even status and control registers (CSRs) that do not require a defined initial state to ensure a correct boot process. The pipeline register will get initialized by the CPU's internal state machines, which are initialized from the main control engine that actually features a defined reset. The initialization of most of the CPU's core CSRs (like interrupt control) is done by the software (to be more specific, this is done by the `crt0.S` start-up code).

During the very early boot process (where `crt0.S` is running) there is no chance for undefined behavior due to the lack of dedicated hardware resets of certain CSRs. For example the machine interrupt-enable CSR **`mie`** does not provide a dedicated reset. The value after reset of this register is uncritical as interrupts cannot fire because the global interrupt enabled flag in the status register (`mstatsus(mie)`) *do* provide a dedicated hardware reset setting this bit to low (globally disabling interrupts).

Reset Configuration

Most CPU-internal register do provide an asynchronous reset in the VHDL code, but the "don't care" value (VHDL `'-'`) is used for initialization of all uncritical registers, effectively generating a flip-flop without a reset. However, certain applications or situations (like advanced gate-level / timing simulations) might require a more deterministic reset state. For this case, a defined reset level (reset-to-low) of all CPU registers can be enabled by enabling a constant in the main VHDL package file (`rtl/core/neorv32_package.vhd`):

```
-- use dedicated hardware reset value for UNCRITICAL registers --
-- FALSE=reset value is irrelevant (might simplify HW), default; TRUE=defined LOW
reset value
constant dedicated_reset_c : boolean := false;
```

[5] If the `Zicsr` CPU extension is enabled (implementing the full set of the privileged architecture).

[6] Shift amount.

[7] Barrel shift when `FAST_SHIFT_EN` is enabled.

[8] Serial shift when `TINY_SHIFT_EN` is enabled.

[9] Shift amount (0..31).

[10] Barrel shifter when `FAST_SHIFT_EN` is enabled.

[11] Memory latency.

[12] DSP-based multiplication; enabled via `FAST_MUL_EN`.

Chapter 4. Software Framework

To make actual use of the NEORV32 processor, the project comes with a complete software ecosystem. This ecosystem is based on the RISC-V port of the GCC GNU Compiler Collection and consists of the following elementary parts:

- Compiler Toolchain
- Core Libraries
- Application Makefile
- Executable Image Format
  - Linker Script
  - RAM Layout
  - C Standard Library
  - Start-Up Code (crt0)
- Bootloader
- NEORV32 Runtime Environment

A summarizing list of the most important elements of the software framework and their according files and folders is shown below:

Application start-up code	`sw/common/crt0.S`
Application linker script	`sw/common/neorv32.ld`
Core hardware driver libraries ("HAL")	`sw/lib/include/` & `sw/lib/source/`
Central application makefile	`sw/common/common.mk`
Tool for generating NEORV32 executables	`sw/image_gen/`
Default bootloader	`sw/bootloader/bootloader.c`
Example programs	`sw/example`

Software Documentation

All core libraries and example programs are highly documented using **Doxygen**. See section [Building the Software Framework Documentation]. The documentation is automatically built and deployed to GitHub pages and is available online at https://stnolting.github.io/neorv32/sw/files.html .

4.1. Compiler 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-gnutoolchain.

The NEORV32 implements a 32-bit RISC-V architecture and uses a 32-bit integer and soft-float ABI by default. Make sure the toolchain / toolchain build is configured accordingly.

- MARCH = `rv32i`
- MABI = `ilp32`

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

The default toolchain prefix used by the project's makefiles is **`riscv32-unknown-elf`**, which can be changes using makefile flags at any time.

More information regarding the toolchain (building from scratch or downloading the prebuilt ones) can be found in the user guides' section Software Toolchain Setup.

4.2. Core Libraries

The NEORV32 project provides a set of C libraries that allows an easy usage of the processor/CPU features (also called "HAL" - hardware abstraction layer). All driver and runtime-related files are located in `sw/lib`. These are automatically included and linked by adding the following *include statement*:

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

C source file	C header file	Description
-	`neorv32.h`	main NEORV32 definitions and library file
`neorv32_cfs.c`	`neorv32_cfs.h`	HW driver (stubs) functions for the custom functions subsystem [13]
`neorv32_cpu.c`	`neorv32_cpu.h`	HW driver functions for the NEORV32 **CPU**
`neorv32_cpu_cfu.c`	`neorv32_cpu_cfu.h`	HW driver functions for the NEORV32 **CFU** (custom instructions)
`neorv32_gpio.c`	`neorv32_gpio.h`	HW driver functions for the **GPIO**
`neorv32_gptmr.c`	`neorv32_gptmr.h`	HW driver functions for the **GPTRM**
-	`neorv32_intrinsics.h`	macros for custom intrinsics & instructions
`neorv32_mtime.c`	`neorv32_mtime.h`	HW driver functions for the **MTIME**
`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** and helper functions
`neorv32_slink.c`	`neorv32_slink.h`	HW driver functions for the **SLINK**
`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**
`neorv32_xip.c`	`neorv32_xip.h`	HW driver functions for the **XIP**
`neorv32_xirq.c`	`neorv32_xirq.h`	HW driver functions for the **XIRQ**
`syscalls.c`	-	newlib system calls

A CMSIS-SVD-compatible **System View Description (SVD)** file including all peripherals is available in `sw/svd`. `sw/lib/include`. Currently, the following library files are available:

4.3. Application Makefile

Application compilation is based on a single, centralized **GNU makefiles** `sw/common/common.mk`. Each project in the `sw/example` folder features a makefile that just includes this central makefile. When creating a new project copy an existing project folder or at least the makefile to the new project folder. It is suggested to create new projects also in `sw/example` to keep the file dependencies. However, these dependencies can be manually configured via makefiles variables when the new project is located somewhere else.

Before the makefile can be used to compile applications, the RISC-V GCC toolchain needs to be installed. Furthermore, the `bin` folder of the compiler needs to be added to the system's `PATH` variable. More information can be found in User Guide: Software Toolchain Setup.

The makefile is invoked by simply executing `make` in the console. For example:

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

4.3.1. Targets

Just executing `make` (or executing `make help`) will show the help menu listing all available targets.

```
$ make
<<< NEORV32 SW Application Makefile >>>
Make sure to add the bin folder of RISC-V GCC to your PATH variable.

== Targets ==
 help         - show this text
 check        - check toolchain
 info         - show makefile/toolchain configuration
 exe          - compile and generate <neorv32_exe.bin> executable for upload via
bootloader
 hex          - compile and generate <neorv32_exe.hex> executable raw file
 image        - compile and generate VHDL IMEM boot image (for application) in local
folder
 install      - compile, generate and install VHDL IMEM boot image (for application)
 sim          - in-console simulation using default/simple testbench and GHDL
 all          - exe + hex + install
 elf_info     - show ELF layout info
 clean        - clean up project
 clean_all    - clean up project, core libraries and image generator
 bl_image     - compile and generate VHDL BOOTROM boot image (for bootloader only!) in
local folder
 bootloader   - compile, generate and install VHDL BOOTROM boot image (for bootloader
only!)

== Variables ==
 USER_FLAGS   - Custom toolchain flags [append only], default ""
 EFFORT       - Optimization level, default "-Os"
 MARCH        - Machine architecture, default "rv32i"
 MABI         - Machine binary interface, default "ilp32"
 APP_INC      - C include folder(s) [append only], default "-I ."
 ASM_INC      - ASM include folder(s) [append only], default "-I ."
 RISCV_PREFIX - Toolchain prefix, default "riscv32-unknown-elf-"
 NEORV32_HOME - NEORV32 home folder, default "../../.."
```

4.3.2. Configuration

The compilation flow is configured via variables right at the beginning of the central makefile (`sw/common/common.mk`):

The makefile configuration variables can be overridden or extended directly when invoking the makefile. For example `$ make MARCH=rv32ic clean_all exe` overrides the default `MARCH` variable definitions. Permanent modifications/definitions can be made in the project-local makefile (e.g., `sw/example/blink_led/makefile`).

Listing 7. Default Makefile Configuration

```
# *****************************************************************************
# USER CONFIGURATION
# *****************************************************************************
# User's application sources (*.c, *.cpp, *.s, *.S); add additional files here
APP_SRC ?= $(wildcard ./*.c) $(wildcard ./*.s) $(wildcard ./*.cpp) $(wildcard ./*.S)
# User's application include folders (don't forget the '-I' before each entry)
APP_INC ?= -I .
# User's application include folders - for assembly files only (don't forget the '-I'
before each
entry)
ASM_INC ?= -I .
# Optimization
EFFORT ?= -Os
# Compiler toolchain
RISCV_PREFIX ?= riscv32-unknown-elf-
# CPU architecture and ABI
MARCH ?= rv32i
MABI  ?= ilp32
# User flags for additional configuration (will be added to compiler flags)
USER_FLAGS ?=
# Relative or absolute path to the NEORV32 home folder
NEORV32_HOME ?= ../../..
# *****************************************************************************
```

Table 56. Variables Description

`APP_SRC`	The source files of the application (`.c, .cpp`, `.S and .s` files are allowed; files of these types in the project folder are automatically added via wild cards). Additional files can be added separated by white spaces
`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`, `-Ofast`, …
`RISCV_PREFIX`	The toolchain prefix to be used; follows the triplet naming convention `[architecture]-[host_system]-[output]-…`
`MARCH`	The targeted RISC-V architecture/ISA; enable compiler support of optional CPU extension by adding the according extension name (e.g. `rv32im` for `M` CPU extension; see User Guide: Enabling RISC-V CPU Extensions for more information
`MABI`	Application binary interface (default: 32-bit integer ABI `ilp32`)
`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 default `sw/example` folder

4.3.3. Default Compiler Flags

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

`-Wall`	Enable all compiler warnings.
`-ffunction-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. Instead, use the NEORV32-specific start-up code (`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 built-in software 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.

4.4. Executable Image Format

In order to generate a file, which can be executed by the processor, all source files have to be compiler, linked and packed into a final *executable*.

4.4.1. Linker Script

When all the application sources have been compiled, they need to be *linked* in order to generate a unified program file. For this purpose the makefile uses the NEORV32-specific linker script `sw/common/neorv32.ld` for linking all object files that were generated during compilation.

The linker script defines three memory *sections*: `rom`, `ram` and `iodev`. Each section provides specific access *attributes*: read access (`r`), write access (`w`) and executable (`x`).

Table 57. Linker memory sections - general

Memory section	Attributes	Description
`ram`	`rwx`	Data memory address space (processor-internal/external DMEM)
`rom`	`rx`	Instruction memory address space (processor-internal/external IMEM) *or* internal bootloader ROM
`iodev`	`rw`	Processor-internal memory-mapped IO/peripheral devices address space

These sections are defined right at the beginning of the linker script:

Listing 8. Linker memory sections - cut-out from linker script `neorv32.ld`

```
MEMORY
{
  ram  (rwx) : ORIGIN = 0x80000000, LENGTH = DEFINED(make_bootloader) ? 512 : 8*1024
  rom   (rx) : ORIGIN = DEFINED(make_bootloader) ? 0xFFFF0000 : 0x00000000, LENGTH =
DEFINED(make_bootloader) ? 32K : 2048M
  iodev (rw) : ORIGIN = 0xFFFFFE00, LENGTH = 512
}
```

Each memory section provides a *base address* `ORIGIN` and a *size* `LENGTH`. The base address and size of the `iodev` section is fixed and should not be altered. The base addresses and sizes of the `ram` and `rom` regions correspond to the total available instruction and data memory address space (see section Address Space Layout) as defined in `rtl/core/neorv32_package.vhd`.

> ⚠ `ORIGIN` of the `ram` section has to be always identical to the processor's `dspace_base_c` hardware configuration.
>
> `ORIGIN` of the `rom` section has to be always identical to the processor's `ispace_base_c` hardware configuration.

The sizes of `rom` section is a little bit more complicated. The default linker script configuration assumes a *maximum* of 2GB *logical* memory space, which is also the default configuration of the processor's hardware instruction memory address space. This size does not have to reflect the *actual* physical size of the instruction memory (internal IMEM and/or processor-external memory). It just provides a maximum limit. When uploading new executable via the bootloader, the bootloader itself checks if sufficient *physical* instruction memory is available. If a new executable is embedded right into the internal-IMEM the synthesis tool will check, if the configured instruction memory size is sufficient (e.g., via the *MEM_INT_IMEM_SIZE* generic).

The `rom` region uses a conditional assignment (via the `make_bootloader` symbol) for `ORIGIN` and `LENGTH` that is used to place "normal executable" (i.e. for the IMEM) or "the bootloader image" to their according memories.

The `ram` region also uses a conditional assignment (via the `make_bootloader` symbol) for `LENGTH`. When compiling the bootloader (`make_bootloader` symbol is set) the generated bootloader will only use the *first* 512 bytes of the data address space. This is a fall-back to ensure the bootloader can operate independently of the actual *physical* data memory size.

The linker maps all the regions from the compiled object files into five final sections: `.text`, `.rodata`, `.data`, `.bss` and `.heap`. These regions contain everything required for the application to run:

Table 58. Linker memory regions

Region	Description
`.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.
`.heap`	This section is required for the address generation of dynamic memory constructs only.

The `.text` and `.rodata` sections are mapped to processor's instruction memory space and the `.data`, `.bss` and `heap` sections are mapped to the processor's data memory space. Finally, the `.text`, `.rodata` and `.data` sections are extracted and concatenated into a single file `main.bin`.

4.4.2. RAM Layout

The default NEORV32 linker script uses all of the defined RAM (linker script memory section `ram`) to create four areas. Note that depending on the application some areas might not be existent at all.

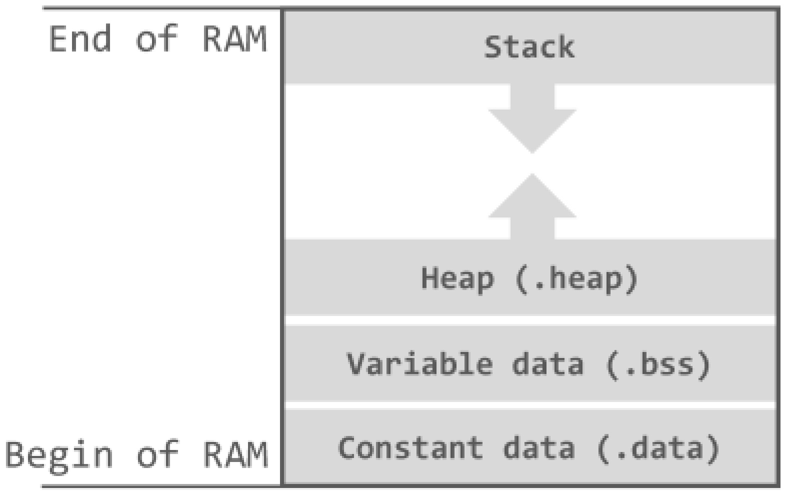


Figure 10. Default RAM Layout

1. **Constant data (`.data`)**: The constant data section is placed right at the beginning of the RAM. For example, this section contains *explicitly initialized* global variables. This section is initialized by the executable.
2. **Dynamic data (`.bss`)**: The constant data section is followed by the dynamic data section, which contains *uninitialized* data like global variables without explicit initialization. This section is cleared by the start-up code `crt0.S`.
3. **Heap (`.heap`)**: The heap is used for dynamic memory that is managed by functions like `malloc()` and `free()`. The heap grows upwards. This section is not initialized at all.
4. **Stack**: The stack starts at the very end of the RAM at address `ORIGIN(ram) + LENGTH(ram) - 4`. The stack grows downwards.

There is *no explicit limit* for the maximum stack size as this is hard to check. However, a physical memory protection rule could be used to configure a maximum size by adding a "protection area" between stack and heap (a PMP region without any access rights).

The maximum size of the heap is defined by the linker script's `__heap_size` symbol. This symbol can be overridden at any time. By default, the maximum heap size is 1/4 of the total RAM size.

Heap-Stack Collisions

Take care when using dynamic memory to avoid collision of the heap and stack memory areas. There is no compile-time protection mechanism available as the actual heap and stack size are defined by *runtime* data. Also beware of fragmentation when using dynamic memory allocation.

4.4.3. C Standard Library

Constructors and Deconstructors

The NEORV32 processor is an embedded system intended for running bare-metal or RTOS applications. To simplify this setup explicit constructors and deconstructors are not supported by default. However, a minimal "deconstructor-alike" support is provided by the After-Main Handler.

The NEORV32 is a processor for *embedded* applications. Hence, it is not capable of running desktop OSs like Linux (at least not without emulation). Hence, the software framework relies on a "bare-metal" setup that uses **newlib** as default C standard library.

RTOS Support

The NEORV32 CPU and processor **do support** embedded RTOS like FreeRTOS and Zephyr. See the User guide section Zephyr RTOS Support and FreeRTOS Support for more information.

Newlib provides stubs for common "system calls" (like file handling and standard input/output) that are used by other C libraries like `stdio`. These stubs are available in `sw/source/syscalls.c` and were adapted for the NEORV32 processor.

Standard Console(s)

UART0 is used to implement all the standard input, output and error consoles (`STDIN`, `STDOUT` and `STDERR`).

Newlib Test/Demo Program

A simple test and demo program, which uses some of newlib's core functions (like `malloc`/`free` and `read`/`write`) is available in `sw/example/demo_newlib`

4.4.4. Executable Image Generator

The `main.bin` file is packed by the NEORV32 image generator (`sw/image_gen`) to generate the final executable file.

The sources of the image generator are automatically compiled when invoking the makefile.

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_hex`	Generates a plain ASCII hex-char file `neorv32_exe.hex` that can be used to initialize custom (instruction-) memories (in synthesis/simulation).

`-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. The *normal application* compilation flow will generate the `neorv32_exe.bin` executable to be upload via UART to the NEORV32 bootloader.

The image generator add a small header to the `neorv32_exe.bin` executable, which consists of three 32-bit words located right at the beginning of the file. The first word of the executable is the signature word and is always `0x4788cafe`. Based on this word the bootloader can identify a valid image file. The next word represents the size in bytes of the actual program image in bytes. A simple "complement" checksum of the actual program image is given by the third word. This provides a simple protection against data transmission or storage errors.

4.4.5. Start-Up Code (crt0)

The CPU and also the processor require a minimal start-up and initialization code to bring the CPU (and the SoC) into a stable and initialized state and to initialize the C runtime environment before the actual application can be executed. This start-up code is located in `sw/common/crt0.S` and is automatically linked *every* application program and placed right before the actual application code so it gets executed right after reset.

The `crt0.S` start-up performs the following operations:

1. Disable interrupts globally by clearing **`mstatus`**`.mie`.
2. Initialize all integer registers `x1 - x31` (or just `x1 - x15` when using the `E` CPU extension) to a defined value.
3. Initialize all CPU core CSRs and also install a default "dummy" trap handler for *all* traps. This handler catches all traps
   - All interrupt sources are disabled and all pending interrupts are cleared.
4. Initialize the global pointer `gp` and the stack pointer `sp` according to the RAM Layout provided by the linker script. during the early boot phase.
5. Clear all counter CSRs and stop auto-increment.
6. Clear IO area: Write zero to all memory-mapped registers within the IO region (`iodev` section). If certain devices have not been implemented, a bus access fault exception will occur. This exception is captured by the dummy trap handler.
7. Clear the `.bss` section defined by the linker script.
8. Copy read-only data from the `.text` section to the `.data` section to set initialized variables.
9. Call the application's `main` function (with *no* arguments: `argc` = `argv` = 0).
10. If the main function returns…
    - the return value is copied to the **`mscratch`** CSR to allow inspection by the on-chip debugger.

  - an optional After-Main Handler is called (if defined at all).
  - the last step the CPU does is entering endless sleep mode (using the `wfi` instruction).

After-Main Handler

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

Listing 9. After-main handler - function prototype

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

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

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

Listing 10. After-main handler - simple example

```
void __neorv32_crt0_after_main(int32_t return_code) {

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

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

4.5. Bootloader

This section refers to the **default** bootloader from the repository. The bootloader can be customized to target application-specific scenarios. See User Guide section Customizing the Internal Bootloader for more information.

The NEORV32 bootloader (source code `sw/bootloader/bootloader.c`) provides an optional build-in firmware that allows to upload new application executables without the need to re-synthesize the FPGA's bitstream. A UART connection is used to provide a simple text-based user interface that allows to upload executables.

Furthermore, the bootloader provides options to program executable to a processor-external SPI flash. An "auto boot" feature can optionally fetch this executable right after reset if there is no user interaction via UART. This allows to build processor setup with non-volatile application storage, which can still be updated at any time.

The bootloader is implemented if the *INT_BOOTLOADER_EN* generic is *true* (default). This will automatically select the CPU's Indirect Boot boot configuration.

Hardware Requirements for the Default NEORV32 Bootloader

REQUIRED: The bootloader requires the privileged architecture CPU extension (`Zicsr` Control and Status Register Access / Privileged Architecture) and at least 512 bytes of data memory (processor-internal DMEM or external DMEM).

RECOMMENDED: For user interaction via UART (like uploading executables) the primary UART (Primary Universal Asynchronous Receiver and Transmitter (UART0)) has to be implemented. Without UART0 the auto-boot via SPI is still supported but the bootloader should be customized (see User Guide) for this purpose.

RECOMMENDED: The default bootloader uses bit 0 of the GPIO controller's (General Purpose Input and Output Port (GPIO)) output port to drive a high-active "heart beat" status LED.

RECOMMENDED: The MTIME machine timer (Machine System Timer (MTIME) generic is *true*) is used to control blinking of the status LED and also to automatically trigger the auto-boot sequence.

OPTIONAL: The SPI controller (Serial Peripheral Interface Controller (SPI)) is required to store/load executable from external flash (for the auto boot feature).

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

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

- 19200 Baud
- 8 data bits
- no parity bit
- 1 stop bit
- newline on `\r\n` (carriage return, newline)
- no transfer protocol / control flow protocol - just raw bytes

Any terminal program that can connect to a serial port should work. However, make sure the program can transfer data in *raw* byte mode without any protocol overhead around it. Some terminal programs struggle with transmitting files larger than 4kB (see https://github.com/stnolting/neorv32/pull/215). Try a different program if uploading a binary does not work.

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

```
<< NEORV32 Bootloader >>

BLDV: Feb 16 2022
HWV:  0x01060709
CLK:  0x05f5e100
ISA:  0x40901107 + 0xc000068b
SOC:  0x7b7f402f
IMEM: 0x00008000 bytes @0x00000000
DMEM: 0x00004000 bytes @0x80000000

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

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

`BLDV`	Bootloader version (built date).
`HWV`	Processor hardware version (the **`mimpid`** CSR); in BCD format; example: `0x01040606` = v1.4.6.6).
`CLK`	Processor clock speed in Hz (via the `CLK` register from System Configuration Information Memory (SYSINFO); defined by the *CLOCK_FREQUENCY* generic).
`ISA`	CPU extensions (**`misa`** CSR + **`mxisa`** CSR).
`SOC`	Processor configuration (via the `SOC` register from the System Configuration Information Memory (SYSINFO); defined by the `IO_*` and `MEM_*` configuration generics).

`IMEM`	IMEM memory base address and size in byte (via the `IMEM_SIZE` and `ISPACE_BASE` registers from the System Configuration Information Memory (SYSINFO); defined by the *MEM_INT_IMEM_SIZE* generic).
`DMEM`	DMEM memory base address and size in byte (via the `DMEM_SIZE` and `DSPACE_BASE` registers from the System Configuration Information Memory (SYSINFO); defined by the *MEM_INT_DMEM_SIZE* generic).

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: Feb 16 2022
HWV:  0x01060709
CLK:  0x05f5e100
ISA:  0x40901107 + 0xc000068b
SOC:  0x7b7f402f
IMEM: 0x00008000 bytes @0x00000000
DMEM: 0x00004000 bytes @0x80000000

Autoboot in 8s. Press any key to abort.
Aborted. ①

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

① Auto boot sequence aborted due to user console input.

The auto boot countdown is stopped and the bootloader's user console is ready to receive one of the following commands:

- `h`: Show the help text (again)
- `r`: Restart the bootloader and the auto-boot sequence
- `u`: Upload new program executable (`neorv32_exe.bin`) via UART into the instruction memory
- `s`: Store executable to SPI flash at `spi_csn_o(0)` (little-endian byte order)
- `l`: Load executable from SPI flash at `spi_csn_o(0)` (little-endian byte order)
- `e`: Start the application, which is currently stored in the instruction memory (IMEM)

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

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.

For detailed information on using an SPI flash for application storage see User Guide section Programming an External SPI Flash via the Bootloader.

4.5.1. Auto Boot Sequence

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

4.5.2. Bootloader Error Codes

If something goes wrong during bootloader operation, an error code and a short message is shown. In this case the processor stalls,, the bootloader status LED is permanently activated and the processor must be reset manually.

In many cases the error source is just *temporary* (like some HF spike during an UART upload). Just try again.

`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. 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 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 - Unexpected exception!`	The bootloader encountered an exception during operation. This might be caused when it tries to access peripherals that were not implemented during synthesis. Example: executing commands `l` or `s` (SPI flash operations) without the SPI module beeing implemented.

4.6. NEORV32 Runtime Environment

The NEORV32 software framework provides a minimal runtime environment (**RTE**) that takes care of a stable and *safe* execution environment by handling *all* traps (= exceptions & interrupts). The RTE simplifies trap handling by wrapping the CPU's *privileged architecture* (i.e. trap-related CSRs) into a unified software API. The NEORV32 RTE is a software library (`sw/lib/source/neorv32_rte.c`) that is part of the default processor library set. It provides public functions via `sw/lib/include/neorv32_rte.h` for application interaction.

Once initialized, the RTE provides Default RTE Trap Handlers that catch all possible exceptions. These default handlers just output a message via UART to inform the user when a certain trap has been triggered. The default handlers can be overridden by the application code to install application-specific handler functions for each trap.

Using the RTE is **optional but highly recommended**. The RTE provides a simple and comfortable way of delegating traps to application-specific handlers while making sure that all traps (even though they are not explicitly used by the application) are handled correctly. Performance-optimized applications or embedded operating systems should not use the RTE for delegating traps.

For the **C standard runtime library** see section [c_standard_library].

4.6.1. RTE Operation

The RTE handles the trap-related CSRs of the CPU's privileged architecture (Machine Trap Handling CSRs). It initializes the **`mtvec`** CSR, which provides the base entry point for all trap handlers. The address stored to this register reflects the **first-level exception handler**, which is provided by the NEORV32 RTE. Whenever an exception or interrupt is triggered this first-level handler is executed.

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 takes care of the actual 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 RTE setup, each entry in the RTE's trap handler's look-up table is initialized with a Default RTE Trap Handlers. These default handler do not execute any trap-related operations - they just output a message via the **primary UART (UART0)** to inform the user that a trap has occurred, that is not handled by the actual application. After sending this message, the RTE tries to continue executing the user program.

4.6.2. Using the RTE

The NEORV32 is enabled by calling the RTE's setup function:

Listing 11. Function Prototype: RTE Setup

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

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

As mentioned above, *all* traps will only trigger execution of the RTE's Default RTE Trap Handlers. To use application-specific handlers, which actually *handle* a trap, the default handlers can be overridden by installing user-defined ones:

Listing 12. Function Prototype: Installing an Application-Specific Trap Handler

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

The first argument `id` defines the "trap ID" (for example a certain interrupt request) that shall be handled by the user-defined handler. The second argument `*handler` is the actual function that implements the trap handler. The function return zero on success and a non-zero value if an error occurred (invalid `id`). In this case no modifications to the RTE's trap look-up-table will be made.

The custom handler functions need to have a specific format without any arguments an with no return value:

Listing 13. Function Prototype: Custom Trap Handler

```
void custom_trap_handler_xyz(void) {

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

Custom Trap Handler Attributes

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

The trap identifier `id` specifies the according trap cause. These can be an *asynchronous trap* like an interrupt from one of the processor modules or a *synchronous trap* triggered by software-caused events like an illegal instruction or an environment call instruction. The `sw/lib/include/neorv32_rte.h` library files provides aliases for trap events supported by the CPU (see NEORV32 Trap Listing) that can be used when installing custom trap handler functions:

Table 59. RTE Trap ID List

ID alias [C]	Description / trap 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
`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

The following example shows how to install a custom handler (`custom_mtime_irq_handler`) for handling the RISC-V machine timer (MTIME) interrupt:

Listing 14. Example: Installing the MTIME IRQ Handler

```
neorv32_rte_exception_install(RTE_TRAP_MTI, custom_mtime_irq_handler);
```

User-defined trap handlers can also be un-installed. This will remove the users trap handler from the RTE core and will re-install the Default RTE Trap Handlers for the specific trap.

Listing 15. Function Prototype: Installing an Application-Specific Trap Handler

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

The argument `id` defines the identifier of the according trap that shall be un-installed. The function return zero on success and a non-zero value if an error occurred (invalid `id`). In this case no modifications to the RTE's trap look-up-table will be made.

The following example shows how to un-install the custom handler `custom_mtime_irq_handler` from the RISC-V machine timer (MTIME) interrupt:

Listing 16. Example: Removing the Custom MTIME IRQ Handler

```
neorv32_rte_exception_uninstall(RTE_TRAP_MTI);
```

4.6.3. Default RTE Trap Handlers

The default RTE trap handlers are executed when a certain trap is triggered that is not handled by a user-defined application-specific trap handler. These default handler will just output a message giving additional debug information via UART0 to inform the user and will try to resume normal execution of the application.

Continuing Execution

In most cases the RTE can successfully continue operation when it catches an interrupt request, which is not handled by the actual application program. However, if the RTE catches an un_handled exception like a bus access fault continuing execution will most likely fail and the CPU will crash.

Listing 17. RTE Default Trap Handler Output Example (Illegal Instruction)

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

In this example the "Illegal instruction" *message* describes the cause of the trap, which is an illegal instruction exception here. `PC` shows the current program counter value when the trap occurred and `MTVAL` shows additional debug information from the **`mtval`** CSR. In this case it shows the encoding of the illegal instruction.

The specific *message* corresponds to the trap code from the **`mcause`** CSR (see NEORV32 Trap Listing). A full list of all messages and the according `mcause` trap codes are shown below.

Table 60. RTE Default Trap Handler Messages and According `mcause` Values

Trap identifier	According `mcause` CSR value
"Instruction address misaligned"	`0x00000000`
"Instruction access fault"	`0x00000001`
"Illegal instruction"	`0x00000002`
"Breakpoint"	`0x00000003`
"Load address misaligned"	`0x00000004`
"Load access fault"	`0x00000005`
"Store address misaligned"	`0x00000006`
"Store access fault"	`0x00000007`
"Environment call from U-mode"	`0x00000008`
"Environment call from M-mode"	`0x0000000b`
"Machine software interrupt"	`0x80000003`
"Machine timer interrupt"	`0x80000007`
"Machine external interrupt"	`0x8000000b`
"Fast interrupt 0"	`0x80000010`
"Fast interrupt 1"	`0x80000011`
"Fast interrupt 2"	`0x80000012`
"Fast interrupt 3"	`0x80000013`
"Fast interrupt 4"	`0x80000014`
"Fast interrupt 5"	`0x80000015`
"Fast interrupt 6"	`0x80000016`
"Fast interrupt 7"	`0x80000017`
"Fast interrupt 8"	`0x80000018`
"Fast interrupt 9"	`0x80000019`
"Fast interrupt a"	`0x8000001a`
"Fast interrupt b"	`0x8000001b`
"Fast interrupt c"	`0x8000001c`
"Fast interrupt d"	`0x8000001d`
"Fast interrupt e"	`0x8000001e`
"Fast interrupt f"	`0x8000001f`
"Unknown trap cause"	*not defined*

Bus Access Faults

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

Listing 18. RTE Default Trap Handler Output Example (Load Access Bus Fault)

```
<RTE> Load access fault [TIMEOUT_ERR] @ PC=0x00000150, MTVAL=0xFFFFFF70 </RTE>
```

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

- `[TIMEOUT_ERR]`: The accessed memory-mapped module did not respond within the valid access time window. In Most cases this is caused by accessing a module that has not been implemented or when accessing "address space holes" (unused/unmapped addresses).
- `[DEVICE_ERR]`: The accesses memory-mapped module asserted it's error signal to indicate an invalid access. For example this can be caused by trying to write to read-only registers or by writing data quantities (like a byte) to devices that do not support sub-word write accesses.
- `[PMP_ERR]`: This indicates an access right violation caused by the **`PMP`** Physical Memory Protection.

[13] This driver file only represents a stub, since the real CFS drivers are defined by the actual CFS implementation.

Chapter 5. On-Chip Debugger (OCD)

The NEORV32 Processor features an *on-chip debugger* (OCD) implementing **execution-based debugging** that is compatible to the **Minimal RISC-V Debug Specification Version 0.13.2**. Please refer to this spec for in-deep information. A copy of the specification is available in `docs/references/riscv-debug-release.pdf`.

The NEORV32 OCD provides the following key features:

- JTAG access port
- run-control of the CPU: halting, single-stepping and resuming
- executing arbitrary programs during debugging
- accessing core registers
- indirect access to the whole processor address space (via program buffer)
- trigger module for hardware breakpoints
- compatible with upstream OpenOCD

OCD Security Note

Access via the OCD is *always authenticated* (`dmstatus.authenticated == 1`). Hence, the *whole system* can always be accessed via the on-chip debugger. Currently, there is no option to disable the OCD via software. The OCD can only be disabled by disabling implementation (setting *ON_CHIP_DEBUGGER_EN* generic to *false*).

Hands-On Tutorial

A simple example on how to use NEORV32 on-chip debugger in combination with OpenOCD and the GNU debugger is shown in section Debugging using the On-Chip Debugger of the User Guide.

The NEORV32 on-chip debugger complex is based on four hardware modules:

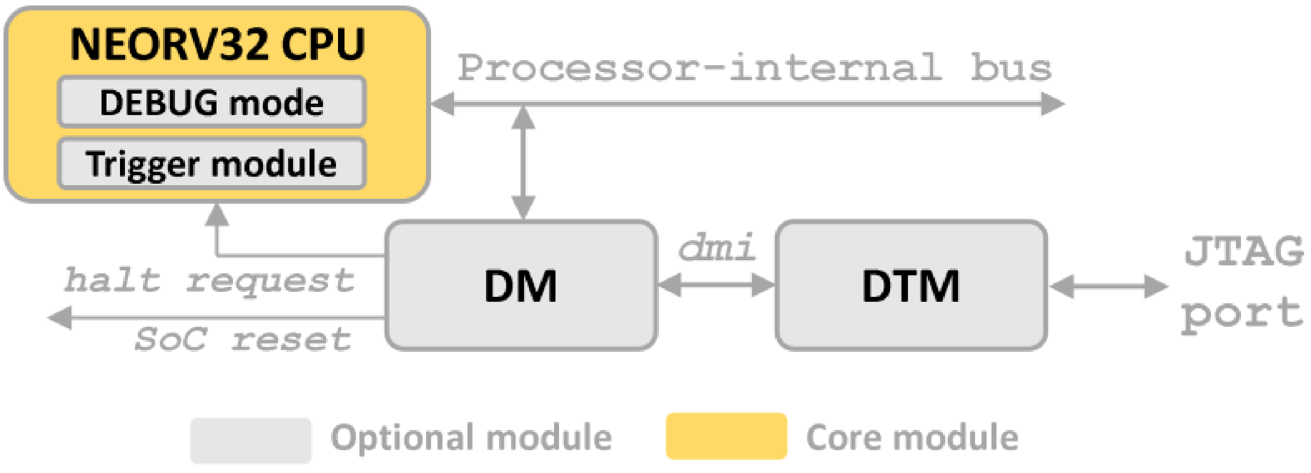


Figure 11. NEORV32 on-chip debugger complex

1. Debug Transport Module (DTM) (`rtl/core/neorv32_debug_dtm.vhd`): External JTAG access tap to allow an external adapter to interface with the *debug module(DM)* using the *debug module interface (dmi)*.
2. Debug Module (DM) (`rtl/core/neorv32_debug_tm.vhd`): Debugger control unit that is configured by the DTM via the the *dmi*. Form the CPU's "point of view" this module behaves as a memory-mapped "peripheral" that can be accessed via the processor-internal bus. The memory-mapped registers provide an internal *data buffer* for data transfer from/to the DM, a *code ROM* containing the "park loop" code, a *program buffer* to allow the debugger to execute small programs defined by the DM and a *status register* that is used to communicate *halt*, *resume* and *execute* requests/acknowledges from/to the DM.
3. CPU CPU Debug Mode extension (part of`rtl/core/neorv32_cpu_control.vhd`): This extension provides the "debug execution mode" which executes the "park loop" code from the DM. The mode also provides additional CSRs.
4. (CPU Trigger Module (also part of`rtl/core/neorv32_cpu_control.vhd`): This module provides a single *hardware* breakpoint, which allows to debug code executed from ROM.)

Theory of Operation

When debugging the system using the OCD, the debugger issues a halt request to the CPU (via the CPU's `db_halt_req_i` signal) to make the CPU enter *debug mode*. In this state, the application-defined architectural state of the system/CPU is "frozen" so the debugger can monitor and even modify it. While in debug mode, the CPU executes the "park loop" code from the *code ROM* of the DM. This park loop implements an endless loop, in which the CPU polls the memory-mapped *status register* that is controlled by the *debug module (DM)*. The flags of these register are used to communicate *requests* from the DM and to *acknowledge* them by the CPU: trigger execution of the program buffer or resume the halted application.

5.1. Debug Transport Module (DTM)

The debug transport module (VHDL module: `rtl/core/neorv32_debug_dtm.vhd`) provides a JTAG test access port (TAP). The DTM is the first entity in the debug system, which connects and external debugger via JTAG to the next debugging entity: the debug module (DM). External JTAG access is provided by the following top-level ports.

Table 61. JTAG top level signals

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

Maximum JTAG Clock

All JTAG signals are synchronized to the processor clock domain by oversampling them in DTM. Hence, no additional clock domain is required for the DTM. However, this constraints the maximal JTAG clock frequency (`jtag_tck_i`) to be less than or equal to **1/5** of the processor clock frequency (`clk_i`).

If the on-chip debugger is disabled (*ON_CHIP_DEBUGGER_EN* = false) the JTAG serial input `jtag_tdi_i` is directly connected to the JTAG serial output `jtag_tdo_o` to maintain the JTAG chain.

The NEORV32 JTAG TAP does not provide a *boundary check* function (yet?). Hence, physical device pins cannot be accessed.

The DTM uses the "debug module interface (dmi)" to access the actual debug module (DM). These accesses are controlled by TAP-internal registers. Each registers is selected by the JTAG instruction register (`IR`) and accessed through the JTAG data register (`DR`).

The DTM's instruction and data registers can be accessed using OpenOCDs `irscan` and `drscan` commands. The RISC-V port of OpenOCD also provides low-level command (`riscv dmi_read` & `riscv dmi_write`) to access the *dmi* debug module interface.

JTAG access is conducted via the **instruction register** `IR`, which is 5 bit wide, and several **data registers** `DR` with different sizes. The data registers are accessed by writing the according address to the instruction register. The following table shows the available data registers:

Table 62. JTAG TAP registers

Address (via IR)	Name	Size [bits]	Description
`00001`	`IDCODE`	32	identifier, default: `0x0CAFE001` (configurable via package's `jtag_tap_idcode_*` constants)
`10000`	`DTMCS`	32	debug transport module control and status register
`10001`	`DMI`	41	debug module interface (*dmi*); 7-bit address, 32-bit read/write data, 2-bit operation (`00` = NOP; `10` = write; `01` = read)
others	`BYPASS`	1	default JTAG bypass register

Table 63. `DTMCS` - DTM Control and Status Register

Bit(s)	Name	r/w	Description
31:18	-	r/-	*reserved*, hardwired to zero
17	`dmihardreset`	r/w	setting this bit will reset the DM interface; this bit auto-clears
16	`dmireset`	r/w	setting this bit will clear ste sticky error state; this bit auto-clears
15	-	r/-	*reserved*, hardwired to zero
14:12	`idle`	r/-	recommended idle states (= 0, no idle states required)
11:10	`dmistat`	r/-	DMI statu: `00` = no error, `01` = reserved, `10` = operation failed, `11` = failed operation during pending DMI operation
9:4	`abits`	r/-	number of DMI address bits (= 7)
3:0	`version`	r/-	`0001` = spec version 0.13

See the RISC-V debug specification for more information regarding the data registers and operations. A local copy can be found in `docs/references`.

5.2. Debug Module (DM)

According to the RISC-V debug specification, the DM (VHDL module: `rtl/core/neorv32_debug_dm.vhd`) acts as a translation interface between abstract operations issued by the debugger and the platform-specific debugger implementation. It supports the following features (excerpt from the debug spec):

- Gives the debugger necessary information about the implementation.
- Allows the hart to be halted and resumed and provides status of the current state.
- Provides abstract read and write access to the halted hart's GPRs.
- Provides access to a reset signal that allows debugging from the very first instruction after reset.
- Provides a mechanism to allow debugging the hart immediately out of reset. (*still experimental*)
- Provides a Program Buffer to force the hart to execute arbitrary instructions.
- Allows memory access from a hart's point of view.

The NEORV32 DM follows the "Minimal RISC-V External Debug Specification" to provide full debugging capabilities while keeping resource (area) requirements at a minimum level. It implements the **execution based debugging scheme** for a single hart and provides the following hardware features:

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

The DM provides two "sides of access": access from the DTM via the *debug module interface (dmi)* and access from the CPU via the processor-internal bus. From the DTM's point of view, the DM implements a set of DM Registers that are used to control and monitor the actual debugging. From the CPU's point of view, the DM implements several memory-mapped registers (within the *normal* address space) that are used for communicating debugging control and status (DM CPU Access).

5.2.1. DM Registers

The DM is controlled via a set of registers that are accessed via the DTM's *dmi*. The "Minimal RISC-V Debug Specification" requires only a subset of the registers specified in the spec. The following registers are implemented. Write accesses to any other registers are ignored and read accesses will always return zero. Register names that are encapsulated in "( )" are not actually implemented; however, they are listed to explicitly show their functionality.

Table 64. Available DM registers

Address	Name	Description
`0x04`	`data0`	Abstract data 0, used for data transfer between debugger and processor
`0x10`	`dmcontrol`	Debug module control
`0x11`	`dmstatus`	Debug module status
`0x12`	`hartinfo`	Hart information
`0x16`	`abstracts`	Abstract control and status
`0x17`	`command`	Abstract command
`0x18`	`abstractauto`	Abstract command auto-execution
`0x1d`	(`nextdm`)	Base address of *next* DM; read as zero to indicate there is only *one* DM
`0x20`	`progbuf0`	Program buffer 0
`0x21`	`progbuf1`	Program buffer 1
`0x38`	(`sbcs`)	System bus access control and status; read as zero to indicate there is no *direct* system bus access
`0x40`	`haltsum0`	Halt summary 0

`data`

0x04 **Abstract data 0** `data0`

Reset value: *UNDEFINED*

Basic read/write registers to be used with abstract command (for example to read/write data from/to CPU GPRs).

`dmcontrol`

0x10 **Debug module control register** `dmcontrol`

Reset value: 0x00000000

Control of the overall debug module and the hart. The following table shows all implemented bits. All remaining bits/bit-fields are configures as "zero" and are read-only. Writing '1' to these bits/fields will be ignored.

Table 65. `dmcontrol` - debug module control register bits

Bit	Name [RISC-V]	R/W	Description
31	`haltreq`	-/w	set/clear hart halt request
30	`resumereq`	-/w	request hart to resume
28	`ackhavereset`	-/w	write `1` to clear `*havereset` flags
1	`ndmreset`	r/w	put whole processor into reset when `1`

Bit	Name [RISC-V]	R/W	Description
0	`dmactive`	r/w	DM enable; writing `0`-`1` will reset the DM

`dmstatus`

0x11	**Debug module status register**	`dmstatus`

Reset value: 0x00000000

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

Table 66. `dmstatus` - debug module status register bits

Bit	Name [RISC-V]	Description
31:23	*reserved*	reserved; always zero
22	`impebreak`	always `1`; indicates an implicit `ebreak` instruction after the last program buffer entry
21:20	*reserved*	reserved; always zero
19	`allhavereset`	`1` when the hart is in reset
18	`anyhavereset`	
17	`allresumeack`	`1` when the hart has acknowledged a resume request
16	`anyresumeack`	
15	`allnonexistent`	always zero to indicate the hart is always existent
14	`anynonexistent`	
13	`allunavail`	`1` when the DM is disabled to indicate the hart is unavailable
12	`anyunavail`	
11	`allrunning`	`1` when the hart is running
10	`anyrunning`	
9	`allhalted`	`1` when the hart is halted
8	`anyhalted`	
7	`authenticated`	always `1`; there is no authentication
6	`authbusy`	always `0`; there is no authentication
5	`hasresethaltreq`	always `0`; halt-on-reset is not supported (directly)
4	`confstrptrvalid`	always `0`; no configuration string available
3:0	`version`	`0010` - DM is compatible to version 0.13

hartinfo

0x12 **Hart information** hartinfo

Reset value: see below

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

Table 67. hartinfo - hart information register bits

Bit	Name [RISC-V]	Description
31:24	*reserved*	reserved; always zero
23:20	nscratch	0001, number of dscratch* CPU registers = 1
19:17	*reserved*	reserved; always zero
16	dataccess	0, the data registers are shadowed in the hart's address space
15:12	datasize	0001, number of 32-bit words in the address space dedicated to shadowing the data registers (1 register)
11:0	dataaddr	= dm_data_base_c(11:0), signed base address of data words (see address map in DM CPU Access)

abstracts

0x16 **Abstract control and status** abstracts

Reset value: see below

Command execution info and status.

Table 68. abstracts - abstract control and status register bits

Bit	Name [RISC-V]	R/W	Description
31:29	*reserved*	r/-	reserved; always zero
28:24	progbufsize	r/-	0010; size of the program buffer (progbuf) = 2 entries
23:11	*reserved*	r/-	reserved; always zero
12	busy	r/-	1 when a command is being executed
11	*reserved*	r/-	reserved; always zero
10:8	cmerr	r/w	error during command execution (see below); has to be cleared by writing 111
7:4	*reserved*	r/-	reserved; always zero
3:0	datacount	r/-	0001; number of implemented data registers for abstract commands = 1

Error codes in cmderr (highest priority first):

- 000 - no error

- `100` - command cannot be executed since hart is not in expected state
- `011` - exception during command execution
- `010` - unsupported command
- `001` - invalid DM register read/write while command is/was executing

`command`

0x17 **Abstract command** `command`

Reset value: 0x00000000

Writing this register will trigger the execution of an abstract command. New command can only be executed if `cmderr` is zero. The entire register in write-only (reads will return zero).

The NEORV32 DM only supports **Access Register** abstract commands. These commands can only access the hart's GPRs (abstract command register index `0x1000` - `0x101f`).

Table 69. `command` - abstract command register - "access register" commands only

Bit	Name [RISC-V]	R/W	Description / required value
31:24	`cmdtype`	-/w	`00000000` to indicate "access register" command
23	*reserved*	-/w	reserved, has to be `0` when writing
22:20	`aarsize`	-/w	`010` to indicate 32-bit accesses
21	`aarpostincrement`	-/w	`0`, post-increment is not supported
18	`postexec`	-/w	if set the program buffer is executed *after* the command
17	`transfer`	-/w	if set the operation in `write` is conducted
16	`write`	-/w	`1`: copy `data0` to `[regno]`, `0`: copy `[regno]` to `data0`
15:0	`regno`	-/w	GPR-access only; has to be `0x1000` - `0x101f`

`abstractauto`

0x18 **Abstract command auto-execution** `abstractauto`

Reset value: 0x00000000s

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

Table 70. `abstractauto` - Abstract command auto-execution register bits

Bit	Name [RISC-V]	R/W	Description
17	`autoexecprogbuf[1]`	r/w	when set reading/writing from/to `progbuf1` will execute `command again`

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

progbuf

0x20	**Program buffer 0**	progbuf0
0x21	**Program buffer 1**	progbuf1

Reset value: NOP-instruction

General purpose program buffer (two entries) for the DM.

haltsum0

0x40	**Halt summary 0**	haltsum0

Reset value: *UNDEFINED*

Bit 0 of this register is set if the hart is halted (all remaining bits are always zero). The entire register is read-only.

5.2.2. DM CPU Access

From the CPU's point of view, the DM behaves as a memory-mapped peripheral that includes

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

DM Register Access

All memory-mapped registers of the DM can only be accessed by the CPU if it is actually *in* debug mode. Hence, the DM registers are not "visible" for normal CPU operations. Any access outside of debug mode will raise a bus error exception.

Park Loop Code Sources

The assembly sources of the **park loop code** are available in sw/ocd-firmware/park_loop.S. Please note, that these sources are not intended to be changed by the used. Hence, the makefile does not provide an automatic option to compile and "install" the debugger ROM code into the HDL sources and require a manual copy (see sw/ocd-firmware/README.md).

The DM uses a total address space of 128 words of the CPU's address space (= 512 bytes) divided into four sections of 32 words (= 128 bytes) each. Please note, that the program buffer, the data buffer and the status register only uses a few effective words in this address space. However, these effective addresses are mirrored to fill up the whole 128 bytes of the section. Hence, any CPU access within this address space will succeed.

Table 71. DM CPU access - address map (divided into four sections)

Base address	Name [VHDL package]	Actual size	Description
`0xffff800`	`dm_code_base_c` (= `dm_base_c`)	128 bytes	Code ROM for the "park loop" code
`0xffff880`	`dm_pbuf_base_c`	16 bytes	Program buffer, provided by DM
`0xffff900`	`dm_data_base_c`	4 bytes	Data buffer (`dm.data0`)
`0xffff980`	`dm_sreg_base_c`	4 bytes	Control and status register

From the CPU's point of view, the DM is mapped to an *"unused"* address range within the processor's Address Space right between the bootloader ROM (BOOTROM) and the actual processor-internal IO space at addresses `0xffff800` - `0xffff9ff`

When the CPU enters or re-enters (for example via `ebreak` in the DM's program buffer) debug mode, it jumps to the beginning of the DM's "park loop" code ROM at `dm_code_base_c`. This is the *normal entry point* for the park loop code. If an exception is encountered during debug mode, the CPU jumps to `dm_code_base_c + 4`, which is the *exception entry point*.

Status Register

The status register provides a direct communication channel between the CPU executing the park loop and the host-controlled controller of the DM. Note that all bits that can be written by the CPU (acknowledge flags) cause a single-shot (1-cycle) signal to the DM controller and auto-clear (always read as zero). The bits that are driven by the DM controller and are read-only to the CPU and keep their state until the CPU acknowledges the according request.

Table 72. DM CPU access - status register

Bit	Name	CPU access	Description
0	`halt_ack`	-/w	Set by the CPU to indicate that the CPU is halted and keeps iterating in the park loop
1	`resume_req`	r/-	Set by the DM to tell the CPU to resume normal operation (leave parking loop and leave debug mode via `dret` instruction)
2	`resume_ack`	-/w	Set by the CPU to acknowledge that the CPU is now going to leave parking loop & debug mode

Bit	Name	CPU access	Description
3	`execute_req`	r/-	Set by the DM to tell the CPU to leave debug mode and execute the instructions from the program buffer; CPU will re-enter parking loop afterwards
4	`execute_ack`	-/w	Set by the CPU to acknowledge that the CPU is now going to execute the program buffer
5	`exception_ack`	-/w	Set by the CPU to inform the DM that an exception occurred during execution of the park loop or during execution of the program buffer

5.3. CPU Debug Mode

The NEORV32 CPU Debug Mode `DB` or `DEBUG` (part of `rtl/core/neorv32_cpu_control.vhd`) is compatible to the "Minimal RISC-V Debug Specification 0.13.2". It is enabled/implemented by setting the CPU generic *CPU_EXTENSION_RISCV_DEBUG* to "true" (done by setting processor generic *ON_CHIP_DEBUGGER_EN*). It provides a new operation mode called "debug mode". When enabled, three additional CSRs are available (section CPU Debug Mode CSRs) and the "return from debug mode" instruction `dret` are available.

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

The CPU debug-mode is entered when one of the following events appear:

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

From a hardware point of view, these "entry conditions" are special synchronous (e.g. `ebreak` instruction) and asynchronous (e.g. halt request "interrupt") traps, that are handled invisibly by the control logic.

Whenever the CPU enters debug-mode it performs the following operations:

- wake-up CPU if it was send to sleep mode by the `wfi` instruction
- move `pc` to `dpc`
- copy the hart's current privilege level to `dcsr.prv`
- set `dcrs.cause` according to the cause why debug mode is entered
- **no update** of `mtval`, `mcause`, `mtval` and `mstatus` CSRs
- load the address configured via the CPU's *CPU_DEBUG_ADDR* generic to the `pc` to jump to the "debugger park loop" code stored in the debug module (DM)

When the CPU is in debug-mode the following things are important:

- while in debug mode, the CPU executes the parking loop and the program buffer provided by the DM if requested
- effective CPU privilege level is `machine` mode, any active physical memory protection (PMP) configuration is bypassed

- the `wfi` instruction acts as a `nop` (also during single-stepping)
- if an exception occurs:
  - if the exception was caused by any debug-mode entry action the CPU jumps to the *normal entry point* (= *CPU_DEBUG_ADDR*) of the park loop again (for example when executing `ebreak` *in* debug-mode)
  - for all other exception sources the CPU jumps to the *exception entry point* (= *CPU_DEBUG_ADDR* + 4) to signal an exception to the DM; the CPU restarts the park loop again afterwards
- interrupts are disabled; however, they will remain pending and will get executed after the CPU has left debug mode
- if the DM makes a resume request, the park loop exits and the CPU leaves debug mode (executing `dret`)
- the standard counters (Machine) Counter and Timer CSRs `[m]cycle[h]` and `[m]instret[h]` are stopped; note that the Machine System Timer (MTIME) keep running as well as it's shadowed copies in the `[m]time[h]` CSRs
- all Hardware Performance Monitors (HPM) CSRs are stopped

Debug mode is left either by executing the `dret` instruction [14] (*in* debug mode) or by performing a hardware reset of the CPU. Executing `dret` outside of debug mode will raise an illegal instruction exception.

Whenever the CPU leaves debug mode it performs the following operations:

- set the hart's current privilege level according to `dcsr.prv`
- restore `pc` from `dpcs`
- resume normal operation at `pc`

5.3.1. CPU Debug Mode CSRs

Two additional CSRs are required by the *Minimal RISC-V Debug Specification*: The debug mode control and status register `dcsr` and the program counter `dpc`. Providing a general purpose scratch register for debug mode (`dscratch0`) allows faster execution of program provided by the debugger, since *one* general purpose register can be backup-ed and directly used.

The debug-mode control and status registers (CSRs) are only accessible when the CPU is *in* debug mode. If these CSRs are accessed outside of debug mode (for example when in `machine` mode) an illegal instruction exception is raised.

`dcsr`

0x7b0	**Debug control and status register**	`dcsr`
Reset value: 0x40000000		

The `dcsr` CSR is compatible to the RISC-V debug spec. It is used to configure debug mode and provides additional status information. The following bits are implemented. The reaming bits are read-only and always read as zero.

Table 73. Debug control and status register `dcsr` bits

Bit	Name [RISC-V]	R/W	Description
31:28	`xdebugver`	r/-	`0100` - indicates external debug support exists
27:16	-	r/-	`000000000000` - *reserved*
15	`ebereakm`	r/w	`ebreak` instructions in `machine` mode will *enter* debug mode when set
14	`ebereakh`	r/-	`0` - hypervisor mode not supported
13	`ebereaks`	r/-	`0` - supervisor mode not supported
12	`ebereaku`	r/w	`ebreak` instructions in `user` mode will *enter* debug mode when set
11	`stepie`	r/-	`0` - IRQs are disabled during single-stepping
10	`stopcount`	r/-	`1` - standard counters and HPMs are stopped when in debug mode
9	`stoptime`	r/-	`0` - timers increment as usual
8:6	`cause`	r/-	cause identifier - why debug mode was entered (see below)
5	-	r/-	`0` - *reserved*
4	`mprven`	r/-	`0` - `mstatus.mprv` is ignored when in debug mode
3	`nmip`	r/-	`0` - non-maskable interrupt is pending
2	`step`	r/w	enable single-stepping when set
1:0	`prv`	r/w	CPU privilege level before/after debug mode

Cause codes in `dcsr.cause` (highest priority first):

- `010` - trigger by hardware Trigger Module
- `001` - executed EBREAK instruction
- `011` - external halt request (from DM)
- `100` - return from single-stepping

dpc

0x7b1 **Debug program counter** `dpc`

Reset value: *UNDEFINED*

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

`dscratch0`

0x7b2	**Debug scratch register 0**	`dscratch0`
Reset value: *UNDEFINED*		
The `dscratch0` CSR is compatible to the RISC-V debug spec. It provides a general purpose debug mode-only scratch register.		

5.4. Trigger Module

The NEORV32 trigger module implements a subset of the features described in the "RISC-V Debug Specification / Trigger Module". It is always implemented when the CPU debug mode / the on-chip debugger is implemented.

⚠ The trigger module only provides a single trigger of *instruction address match* type. This trigger will fire **after** the instruction at the specific address has been executed.

The trigger module only provides a single trigger supporting only the "instruction address match" type. This limitation is granted by the RISC-V specs. and is sufficient to **debug code executed from read-only memory (ROM)**. "Normal" *software* breakpoints (using gdb's `b`/`break` command) are implemented by temporarily replacing the according instruction word by a BREAK instruction. This is not possible when debugging code that is executed from read-only memory (for example when debugging programs that are executed via the Execute In Place Module (XIP)). Therefore, the NEORV32 trigger module provides a single "instruction address match" trigger to enter debug mode when executing the instruction at a specific address. These "hardware-assisted breakpoints" are used by gdb's `hb`/`hbreak` command.

5.4.1. Trigger Module CSRs

The trigger module provides 8 additional CSRs, which accessible in debug mode and also in machine-mode. Since the trigger module does not support *native mode* writes from machine-mode software to those CSRs are ignored. Hence, the CSRs of this module are only relevant for the debugger.

`tselect`

0x7a0 **Trigger select register** `tselect`

Reset value: 0x00000000

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

`tdata1`

0x7a1 **Trigger data register 1 / match control register** `tdata1` / `mcontrol`

Reset value: 0x28041048

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

Table 74. Match control CSR (`tdata1`) bits

Bit	Name [RISC-V]	R/W	Description
31:28	`type`	r/-	`0010` - address match trigger

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

`tdata2`

0x7a2 **Trigger data register 2** `tdata2`

Reset value: *UNDEFINED*

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

`tdata3`

0x7a3 **Trigger data register 3** `tdata3`

Reset value: 0x00000000

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

`tinfo`

0x7a4 **Trigger information register** `tinfo`

Reset value: 0x00000004

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

`tcontrol`

0x7a5	**Trigger control register**	`tcontrol`

Reset value: 0x00000000

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

`mcontext`

0x7a8	**Machine context register**	`mcontext`

Reset value: 0x00000000

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

`scontext`

0x7aa	**Supervisor context register**	`scontext`

Reset value: 0x00000000

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

[14] `dret` should only be executed *inside* the debugger "park loop" code (→ code ROM in the debug module (DM).)

Chapter 6. Legal

License

The NEORV32 RISC-V Processor
HQ: https://github.com/stnolting/neorv32
By Dipl.-Ing. Stephan Nolting
Hanover, Germany
Contact: stnolting@gmail.com

Proprietary Notice

Disclaimer

Limitation of Liability for External Links

Citing

This is an open-source project that is free of charge. Use this project in any way you like (as long as it complies to the permissive license). Please quote it appropriately.

Contributors ❤️

Please add as many contributors as possible to the `author` field.
This project would not be where it is without them.

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

Listing 19. BibTeX

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

DOI

This project also provides a *digital object identifier* provided by zenodo: [zenodo.5018888]

Acknowledgments

A big shout-out to the community and all contributors, who helped improving this project! ❤️

RISC-V - instruction sets want to be free!

Continuous integration provided by GitHub Actions and powered by GHDL.