The NEORV32 Processor

Dipl.-Ing. Stephan Nolting

A small, customizable and open-source
full-scale 32-bit RISC-V soft-core CPU & SoC.

Proprietary and Legal Notice

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

Limitation of Liability for External Links

Our website contains links to the websites of third parties („external links"). As the content of these websites is not under our control, we cannot assume any liability for such external content. In all cases, the provider of information of the linked websites is liable for the content and accuracy of the information provided. At the point in time when the links were placed, no infringements of the law were recognizable to us. As soon as an infringement of the law becomes known to us, we will immediately remove the link in question.

Disclaimer

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

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

RISC-V – Instruction Sets Want To Be Free!

https://riscv.org/
https://github.com/riscv

Citing

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

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

The **NEORV32 Processor** Project
©2020 Dipl.-Ing. Stephan Nolting, Hanover, Germany
https://github.com/stnolting/neorv32
Contact: stnolting@gmail.com

BSD 3-Clause License

Table of Content

1. Overview

The **NEORV32[1] Processor** is a customizable microcontroller-like system on chip (SoC) that is based on the RISC-V-compliant **NEORV32 CPU**. The project and this documentary are split into several parts:

This very first chapter gives a brief overview of the project and its key features. The second chapter (2. NEORV32 Central Processing Unit (CPU)) focuses on the CPU. Chapter three (3. NEORV32 Processor (SoC)) shows the processor setup and all its modules. The fourth chapter (4. Software Architecture) introduces the software framework. Chapter five (5. Let's Get It Started!) presents tutorials for getting started and for several aspects of the project.

The NEORV32 CPU

The **CPU** implements a `rv32i` RISC-V core with optional `C`, `E`, `M`, `U`, `Zicsr`, `Zifencei` and `PMP` (physical memory protection) extensions. It passes the official RISC-V compliance tests and is compliant to the Unprivileged ISA Specification Version 2.2 and a subset of the Privileged Architecture Specification Version 1.12-draft.

If you do not want to use the NEORV32 *Processor* setup, you can also use the CPU in stand-alone mode and build your own SoC around it. Chapter 2. NEORV32 Central Processing Unit (CPU) explains all the details of the CPU: Architecture, instruction sets, extensions, timing and interfaces.

The NEORV32 Processor

Based on the NEORV32 CPU, the **Processor** is a full-scale RISC-V microcontroller system that already provides common peripherals like GPIO, serial interfaces, timers, embedded memories and an external bus interface for connectivity and custom extension. All optional features and modules beyond the base CPU can be enabled and configured via VHDL generics.

The processor is intended as ready-to-use auxiliary processor within a larger SoC designs or as stand-alone custom microcontroller. Its top entity can be directly synthesized for any target technology without modifications. Chapter 3. NEORV32 Processor (SoC) shows the provides processor modules, their functionality and how they interact with the CPU.

This project comes with a complete software ecosystem that features core libraries for high-level usage of the provided functions and peripherals, makefiles, a runtime environment, several example programs to start with and even a builtin bootloader for easy program upload via UART. All software source files provide a *doxygen*-based documentary (available on GitHub pages).

How to get started?

The processor is intended to work "out of the box". Just synthesize the test setup from this project, upload it to your FPGA board of choice and start playing with the NEORV32. If you do not want to compile the GCC toolchain by yourself, you can also download pre-compiled binaries for Linux.

Jump directly to the **5. Let's Get It Started!** chapter to get started!

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

1.1. Project Key Features

- ✔ 32-bit `rv32i` RISC-V-compliant base CPU (→ p.16)
  - The CPU passes the official RISC-V-Compliance tests (→ p.18)
  - Optional `C` extension for compressed instructions
  - Optional `E` extensions for embedded CPU version
  - Optional `M` extension for multiplication and division instructions
  - Optional `U` extension for user mode privilege level
  - Optional `Zicsr` extension for control and status register access and exception/interrupt system
  - Optional `Zifencei` extension for instruction stream synchronization
  - Optional Physical Memory Protection (PMP)
- ✔ Toolchain based on free RISC-V GCC port; prebuilt toolchains available (→ p.75)
- ✔ Application compilation based on GNU makefiles (→ p.77)
- ✔ Doxygen-based documentation of the software framework (→ p.94); a deployed version is available at https://stnolting.github.io/neorv32/files.html
- ✔ NEORV32 Processor: Customizable full-scale microcontroller-like processor system/SoC (→ p.42) based on the NEORV32 CPU
  - Serial interfaces (UART, TWI, SPI)
  - Timers and counters (WDT, MTIME)
  - Embedded memories for data, instructions and bootloader
  - External memory interface to attach custom modules
  - Optional slots for two tightly-coupled custom co-processor (CFUs)
- ✔ Fully synchronous design, no latches, no gated clocks
- ✔ Completely described in behavioral, platform-independent VHDL – no primitives, macros, etc. used
- ✔ Small hardware footprint and high operating frequency (→ p.11)
- ✔ FreeRTOS port + demo available (→ p.103)
- ✔ Official RISC-V open-source architecture ID

1.2. Design Principles

I've worked on and with several soft-core architecture. And I think I have studied even more of them. There are so many good projects on GitHub: Great processor designs and projects with the best of intentions. Unfortunately, many of them lack a good documentation that covers everything down from the rtl level to the software library and how all the parts get together.

➔ **From zero to `main()`: Completely open source and documented.**

Everyone uses different FPGAs and evaluations boards. This variety is a good thing. Though, it can be quite frustrating if you have to dig into the deepest corners of a HDL project if you just want to do a quick test synthesis for your FPGA board.. This project comes in a technology-independent form. Nevertheless, it also provides *optional alternative* components, that are tailored to specific FPGAs.

➔ **Plain VHDL without technology-specific parts like attributes, macros or primitives.**

I just talked about it. Sometimes you just want to check out a project. This project also tries to be useful for beginners, too.

➔ **Easy to use – working out of the box.**

Some of the open-source processors out there have nice CPI and benchmark results. But at least some of these architectures have quite "unrealistic" memory interfaces (read response within the same cycle) most memories cannot provide. Also, I do not like asynchronous interfaces (some of them even use latches) nor clock gating to halt certain parts of a circuit.

➔ **Clean synchronous design, no wacky combinatorial interfaces.**

There are a lot of nice RISC-V soft-cores out there that are more powerful than this approach. The NEORV32 processor is intended to be a small and still RISC-V-compliant processor system that also fits into the smallest FPGAs and the tiniest niches within larger FPGA systems.

➔ **Be as small as possible – but with a reasonable size-performance tradeoff.**

Probably we all are fans of a specific FPGA architectures toolchain or manufacturer. All FPGA vendors out there have their individual benefits, but I really like the Lattice iCE40 FPGAs. They are so tiny and have a very clean and simple architecture (no fancy multiplexers and stuff like that). The large embedded memory blocks are a nice extra.

➔ **The processor has to fit in a Lattice iCE40 UltraPlus 5k FPGA running at 20+ MHz.**

1.3. Project Folder Structure

```
neorv32                 Project home folder.
│
├─.ci                   Scripts for continuous integration.
│
├─CHANGELOG.md          Project change log.
│
├─docs                  Project documentary: RISC-V specifications implemented in this
│ │                     project, Wishbone bus specification, NEORV32 data sheet, doxygen
│ │                     makefiles.
│ │
│ ├─doxygen_build       Software documentary HTML files (generated by doxygen).
│ │
│ └─figures             Images mainly for the GitHub front page.
│
├─rtl                   Processor's VHDL source files.
│ │
│ ├─core                This folder contains all the rtl (VHDL) core files of the NEORV32.
│ │                     Make sure to add ALL of them to your FPGA EDA project.
│ │
│ ├─top_templates       Here you can find alternative top entities of the NEORV32.
│ │
│ └─fpga_specific       This folder provides FPGA technology-specific optimized HW modules.
│
├─sim                   The sim folder contains the default VHDL testbench and additional
│ │                     simulation files.
│ │
│ ├─ghdl                Simulation script for GHDL.
│ │
│ └─vivado              Default Xilinx Vivado simulation waveform configuration.
│
└─sw                    The software folder contains the processor's core libraries,
  │                     makefiles, linker script, start-up code and example programs.
  │
  ├─bootloader          Source and compilation script of the NEORV32-internal bootloader.
  │
  ├─common              Linker script and startup code.
  │
  ├─example             Here you can find several example programs. Each project folder
  │ │                   includes the program's C sources and a makefile. Add your own
  │ └─...               projects to this folder.
  │
  ├─image_gen           Helper program to generate executables for the NEORV32.
  │
  └─lib                 This folder contains the processor's core libraries.
    │
    ├─include           NEORV32 hardware driver library C source files and the according
    └─source            header/include files.
```

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

1.4. 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 have to be assigned to a new **library** called **neorv32**.

```
neorv32_top.vhd                       NEORV32 Processor top entity
│
├─neorv32_boot_rom.vhd                Bootloader ROM
│ │
│ └─neorv32_bootloader_image.vhd      Boot ROM initialization image for the bootloader
│
├─neorv32_busswitch.vhd               Processor bus switch for CPU buses (I&D)
│
├─neorv32_cfu0.vhd                    Custom functions unit 0
│
├─neorv32_cfu1.vhd                    Custom functions unit 1
│
├─neorv32_cpu.vhd                     NEORV32 CPU top entity
│ │
│ ├─neorv32_package.vhd               Processor/CPU main VHDL package file
│ │
│ ├─neorv32_cpu_alu.vhd               Arithmetic/logic unit
│ │
│ ├─neorv32_cpu_bus.vhd               Bus interface unit + physical memory protection
│ │
│ ├─neorv32_cpu_control.vhd           CPU control, exception/IRQ system and CSRs
│ │ │
│ │ └─neorv32_cpu_decompressor.vhd    Compressed instructions decoder
│ │
│ ├─neorv32_cpu_cp_muldiv.vhd         Multiplication/division co-processor
│ │
│ └─neorv32_cpu_regfile.vhd           Data register file
│
├─neorv32_dmem.vhd                    Processor-internal data memory
│
├─neorv32_gpio.vhd                    General purpose input/output port unit
│
├─neorv32_imem.vhd                    Processor-internal instruction memory
│ │
│ └─neor32_application_image.vhd      IMEM application initialization image
│
├─neorv32_mtime.vhd                   Machine system timer
│
├─neorv32_pwm.vhd                     Pulse-width modulation controller
│
├─neorv32_spi.vhd                     Serial peripheral interface controller
│
├─neorv32_sysinfo.vhd                 System configuration information memory
│
├─neorv32_trng.vhd                    True random number generator
│
├─neorv32_twi.vhd                     Two wire serial interface controller
│
├─neorv32_uart.vhd                    Universal asynchronous receiver/transmitter
│
├─neorv32_wdt.vhd                     Watchdog timer
│
└─neorv32_wb_interface.vhd            External Wishbone bus gateway
```

1.5. FPGA Implementation Results

This chapter shows exemplary implementation results of the **NEORV32 processor/CPU** for an **Intel Cyclone IV EP4CE22F17C6N** FPGA on a *Terasic © DE0-Nano* board. The design was synthesized using **Intel Quartus Prime Lite 19.1** ("balanced implementation"). The timing information is derived from the Timing Analyzer / Slow 1200mV 0C Model. If not other specified, the default configuration of the processor's generics is assumed. No constraints were used.

The first chapter shows the implementation results for different CPU configurations (via the `CPU_EXTENSION_*` generics only) while the second chapter shows the implementation results for each of the available peripherals. The results were taken from the fitter report (Resource Section / Resource Utilization by Entity) and reflect the resource utilization by the CPU only.

Please note, that the provided results are just a relative measure as logic functions of different modules might be merged between entity boundaries, so the actual utilization results might vary a bit.

1.5.1. CPU

```
Hardware Version:    1.4.4.8
```

CPU	CPU Configuration Generics	LEs	FFs	MEM bits	DSPs	F_{max}
`rv32i`	CPU_EXTENSION_RISCV_C = false CPU_EXTENSION_RISCV_E = false CPU_EXTENSION_RISCV_M = false CPU_EXTENSION_RISCV_U = false CPU_EXTENSION_RISCV_Zicsr = false CPU_EXTENSION_RISCV_Zifencei = false	983	438	2048	0	120 MHz
`rv32iu` + `Zicsr` + `Zifencei`	CPU_EXTENSION_RISCV_C = false CPU_EXTENSION_RISCV_E = false CPU_EXTENSION_RISCV_M = false CPU_EXTENSION_RISCV_U = true CPU_EXTENSION_RISCV_Zicsr = true CPU_EXTENSION_RISCV_Zifencei = true	1877	802	2048	0	112 MHz
`rv32imu` + `Zicsr` + `Zifencei`	CPU_EXTENSION_RISCV_C = false CPU_EXTENSION_RISCV_E = false CPU_EXTENSION_RISCV_M = true CPU_EXTENSION_RISCV_U = true CPU_EXTENSION_RISCV_Zicsr = true CPU_EXTENSION_RISCV_Zifencei = true	2374	1048	2048	0	110 MHz
`rv32imcu` + `Zicsr` + `Zifencei`	CPU_EXTENSION_RISCV_C = true CPU_EXTENSION_RISCV_E = false CPU_EXTENSION_RISCV_M = true CPU_EXTENSION_RISCV_U = true CPU_EXTENSION_RISCV_Zicsr = true CPU_EXTENSION_RISCV_Zifencei = true	2650	1064	2048	0	110 MHz
`rv32emcu` + `Zicsr` + `Zifencei`	CPU_EXTENSION_RISCV_C = true CPU_EXTENSION_RISCV_E = true CPU_EXTENSION_RISCV_M = true CPU_EXTENSION_RISCV_U = true CPU_EXTENSION_RISCV_Zicsr = true CPU_EXTENSION_RISCV_Zifencei = true	2680	1061	1024	0	110 MHz

Table 1: Hardware utilization for different CPU configurations

1.5.2. Processor Modules

Hardware Version: **1.4.4.8**

Module	Description	LEs	FFs	MEM bits	DSPs
Boot ROM	Bootloader ROM (4kB)	4	1	32 768	0
BUSSWITCH	Mux for CPU I & D interfaces	62	8	0	0
CFU0	Custom functions unit 0[2]	–	–	–	–
CFU1	Custom functions unit 1	–	–	–	–
DMEM	Processor-internal data memory (8kB)	13	2	65 536	0
GPIO	General purpose input/output ports	66	65	0	0
IMEM	Processor-internal instruction memory (16kB)	7	2	131 072	0
MTIME	Machine system timer	268	166	0	0
PWM	Pulse_width modulation controller	72	69	0	0
SPI	Serial peripheral interface	184	125	0	0
SYSINFO	System configuration information memory	11	9	0	0
TRNG	True random number generator	132	105	0	0
TWI	Two-wire interface	74	44	0	0
UART	Universal asynchronous receiver/transmitter	175	132	0	0
WDT	Watchdog timer	58	45	0	0
WISHBONE	External memory interface (`MEM_EXT_REG_STAGES` = 2)	106	104	0	0

Table 2: Hardware utilization by the different peripheral modules

2 Hardware requirements for the CFUs depend on actual user-defined implementation.

1.5.3. Exemplary Processor Setups

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

`Hardware Version:`	**`1.4.4.8`**
`CPU Configuration:`	`rv32i(m)cu` + `Zicsr` + `Zifencei` + `(PMP)`

Vendor	FPGA	Board	Toolchain	Impl. strategy	LUT / LE	FF / REG	DSP	Embedded memory	*f* [MHz]
Intel	Cyclone IV EP4CE22F17C6N	Terasic DE0-Nano	Quartus Prime Lite 19.1	balanced	4008 (18%)	1849 (9%)	0 (0%)	Memory bits: 231424 (38%)	105
Lattice	iCE40 UltraPlus iCE40UP5K-SG48I	Upduino v2.0	Radiant 2.1 (Sinplify Pro)	default	4296 (81%)	1611 (30%)	0 (0%)	EBR: 12 (40%) SPRAM: 4 (100%)	22.5*
Xilinx	Artix-7 XC7A35TICSG324 -1L	Arty A7-35T	Vivado 2019.2	default	2390 (11%)	1888 (5%)	0 (0%)	BRAM: 8 (16%)	100*

Table 3: Hardware utilization for different FPGA platforms

Notes

- The Lattice iCE40 UltraPlus setup uses the FPGA's SPRAM memory primitives for the internal IMEM and DEMEM (each 64kb). The according FPGA-specific memory components for the IMEM and DMEM can be found in the `rtl/fpga_specific` folder. Also, the Lattice setup does not implement the `M` extension and not the physical memory protection (`PMP`).
- The clock frequencies marked with an asterisk (*) are constrained clocks. The remaining ones are "f_max" results from the place and route timing reports.
- The Upduino and the Arty board have on-board SPI flash memories for storing the FPGA configuration. These device can also be used by the default NEORV32 bootloader to store and automatically boot an application program after reset (both tested successfully).
- The setups with `PMP` implement 2 regions with a minimal granularity of 32kB.

Regarding Lattice Radiant

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

1.6. CPU Performance

1.6.1. CoreMark Benchmark

Configuration

Hardware:	32kB IMEM, 16kB DMEM, 100MHz clock
CoreMark:	**2000** iteration, MEM_METHOD is **MEM_STACK**
Compiler:	**RISCV32-GCC 10.1.0**
Peripherals:	**UART** for printing the results
Flags:	default, see makefile
Hardware Version:	**1.4.5.2**

The performance of the NEORV32 was tested and evaluated using the CoreMark CPU benchmark. This benchmark focuses on testing the capabilities of the CPU core itself rather than the performance of the whole system. The according source code and the SW project can be found in the `sw/example/coremark` folder. All NEORV32-specific modifications were done in the port-me files - “outside” of the time-critical benchmark core.

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

$$\text{CoreMark Score} = \frac{\text{CoreMark iterations}}{\text{Time in seconds}}$$

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

$$\text{Relative CoreMark Score} = \frac{\text{CoreMark Score}}{\text{Clock frequency [MHz]}}$$

Results

CPU	Executable Size	Optimization	CoreMark Score	CoreMarks/Mhz
`rv32i`	26 940 bytes	-O3	33.89	**0.3389**
`rv32im`	25 772 bytes	-O3	64.51	**0.6451**
`rv32imc`	20 524 bytes	-O3	64.51	**0.6451**
`rv32imc + FAST_MUL_EN`	20 524 bytes	-O3	80.00	**0.8000**
`rv32imc + FAST_MUL_EN + FAST_SHIFT_EN`	20 524 bytes	-O3	83.33	**0.8333**

Table 4: NEORV32 CoreMark results

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

1.6.2. Instruction Timing

The NEORV32 CPU is based on a multi-cycle architecture. Each instruction is executed in a sequence of several consecutive micro operations. Hence, each instruction requires several clock cycles to execute. The average CPI (cycles per instruction) depends on the instruction mix of a specific applications and also on the available CPU extensions. The following table shows the performance results for successfully (!) running 2000 CoreMark iterations. The average CPI is computed by dividing the total number of required clock cycles (only the timed core to avoid distortion due to IO wait cycles) by the number of executed instructions (`[m]instret[h]` CSRs). The executables were generated using optimization `-O3`.

Hardware Version:	**1.4.5.2**

CPU	Required Clock Cycles	Executed Instructions	Average CPI
`rv32i`	5 945 938 586	1 469 587 406	**4.05**
`rv32im`	3 110 282 586	602 225 760	**5.16**
`rv32imc`	3 172 969 968	615 388 890	**5.16**
`rv32imc + FAST_MUL_EN`	2 590 417 968	615 388 890	**4.21**
`rv32imc + FAST_MUL_EN + FAST_SHIFT_EN`	2 456 318 408	615 388 890	**3.99**

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

More information regarding the execution time of each implemented instruction can be found in chapter 2.5. Instruction Timing2.5. Instruction Timing.

2. NEORV32 Central Processing Unit (CPU)

The NEORV32 CPU is the heart of the NEORV32 processor. You can use it as part of the NEORV32 processor or you can use the CPU to build your very own processor system.

CPU Key Features

- 32-bit RISC-V CPU: `rv32[i/e][c][m]x + [U][Zicsr][Zifencei]`
- Compliant to the RISC-V user specifications – passes the official RISC-V compliance tests
- Mostly compliant to the RISC-V privileged specifications
- Optional privileged architecture (`Zicsr`) extension supporting RISC-V-compliant control and status registers (CSRs), CSR access instructions, exception/interrupt/trap support
- Optional `Zifencei` extension for instruction stream synchronization via `fence.i` instruction
- Privilege levels: Machine level (`M-mode`), User level (`U-mode`, when `U` extension is enabled); after reset the CPU is in `M-mode`
- Optional Physical Memory Configuration (PMP), compliant to the RISCV-specs., only supports `NAPOT` mode and up to 8 regions yet
- Von-Neumann architecture, separated interfaces for instruction fetch and data access (merged into single bus via a bus switch for the NEORV32 processor)
- Little-endian byte order
- No hardware support of unaligned data/instructions accesses – they will trigger an exception. When the `C` extension is enabled, instructions can also be 16-bit aligned and a misaligned instruction address exception is not possible anymore
- All reserved or unimplemented instructions will raise an illegal instruction exception
- Two-stage pipelined multi-cycle in-order instruction execution
- Four custom fast interrupt request lines (custom extension)
- NEORV32-specific custom CSRs are mapped to the official RISC-V custom address spaces (custom extension)
- Official RISC-V open-source architecture ID

Architecture

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

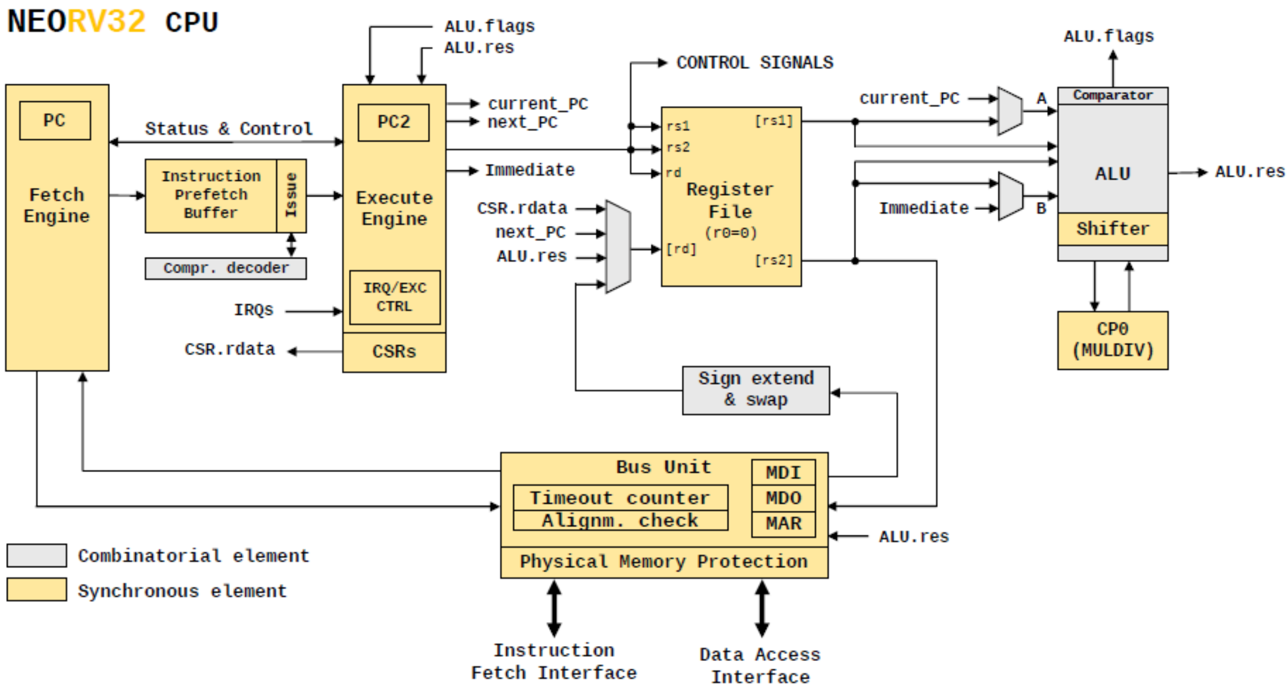

Figure 1: Simplified architecture of the NEORV32 CPU

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

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

Basically, the NEORV32 CPU is somewhere between a classical pipelined architecture, where each stage requires exactly one processing cycle (if not stalled) and a classical multi-cycle architecture, which executes every single instruction in a series of consecutive micro-operations. The combination of these two classical design paradigms allows an increased instruction execution in contrast to a pure multi-cycle approach (due to the pipelined approach) at a reduced hardware footprint (due to the multi-cycle approach). This seems to be a quite good trade-off – at least for me.
The CPU provides independent interfaces for instruction fetch and data access. These two bus interfaces are merged into a single processor-internal bus via a bus switch. Hence, memory locations including peripheral devices are mapped to a single 32-bit address space making the architecture a modified Von-Neumann Architecture.

2.1. RISC-V Compliance

The NEORV32 CPU passes the official RISC-V compliance test. The port of this test suite for the NEORV32 can be found in the neorv32_compliance_test GitHub repository.

RISC-V `rv32i` Tests

```
Check                 I-ADD-01 ... OK
Check                I-ADDI-01 ... OK
Check                 I-AND-01 ... OK
Check                I-ANDI-01 ... OK
Check               I-AUIPC-01 ... OK
Check                 I-BEQ-01 ... OK
Check                 I-BGE-01 ... OK
Check                I-BGEU-01 ... OK
Check                 I-BLT-01 ... OK
Check                I-BLTU-01 ... OK
Check                 I-BNE-01 ... OK
Check         I-DELAY_SLOTS-01 ... OK
Check              I-EBREAK-01 ... OK
Check               I-ECALL-01 ... OK
Check           I-ENDIANESS-01 ... OK
Check                  I-IO-01 ... OK
Check                 I-JAL-01 ... OK
Check                I-JALR-01 ... OK
Check                  I-LB-01 ... OK
Check                 I-LBU-01 ... OK
Check                  I-LH-01 ... OK
Check                 I-LHU-01 ... OK
Check                 I-LUI-01 ... OK
Check                  I-LW-01 ... OK
Check        I-MISALIGN_JMP-01 ... OK
Check       I-MISALIGN_LDST-01 ... OK
Check                 I-NOP-01 ... OK
Check                  I-OR-01 ... OK
Check                 I-ORI-01 ... OK
Check             I-RF_size-01 ... OK
Check            I-RF_width-01 ... OK
Check               I-RF_x0-01 ... OK
Check                  I-SB-01 ... OK
Check                  I-SH-01 ... OK
Check                 I-SLL-01 ... OK
Check                I-SLLI-01 ... OK
Check                 I-SLT-01 ... OK
Check                I-SLTI-01 ... OK
Check               I-SLTIU-01 ... OK
Check                I-SLTU-01 ... OK
Check                 I-SRA-01 ... OK
Check                I-SRAI-01 ... OK
Check                 I-SRL-01 ... OK
Check                I-SRLI-01 ... OK
Check                 I-SUB-01 ... OK
Check                  I-SW-01 ... OK
Check                 I-XOR-01 ... OK
Check                I-XORI-01 ... OK
--------------------------------
OK: 48/48 RISCV_TARGET=neorv32 RISCV_DEVICE=rv32i RISCV_ISA=rv32i
```

RISC-V rv32im Tests

```
Check                     DIV ... OK
Check                    DIVU ... OK
Check                     MUL ... OK
Check                    MULH ... OK
Check                  MULHSU ... OK
Check                   MULHU ... OK
Check                     REM ... OK
Check                    REMU ... OK
--------------------------------
OK: 8/8 RISCV_TARGET=neorv32 RISCV_DEVICE=rv32im RISCV_ISA=rv32im
```

RISC-V rv32imc Tests

```
Check                   C-ADD ... OK
Check                  C-ADDI ... OK
Check              C-ADDI16SP ... OK
Check              C-ADDI4SPN ... OK
Check                   C-AND ... OK
Check                  C-ANDI ... OK
Check                  C-BEQZ ... OK
Check                  C-BNEZ ... OK
Check                     C-J ... OK
Check                   C-JAL ... OK
Check                  C-JALR ... OK
Check                    C-JR ... OK
Check                    C-LI ... OK
Check                   C-LUI ... OK
Check                    C-LW ... OK
Check                  C-LWSP ... OK
Check                    C-MV ... OK
Check                    C-OR ... OK
Check                  C-SLLI ... OK
Check                  C-SRAI ... OK
Check                  C-SRLI ... OK
Check                   C-SUB ... OK
Check                    C-SW ... OK
Check                  C-SWSP ... OK
Check                   C-XOR ... OK
--------------------------------
OK: 25/25 RISCV_TARGET=neorv32 RISCV_DEVICE=rv32imc RISCV_ISA=rv32imc
```

RISC-V rv32Zicsr Tests

```
Check              I-CSRRC-01 ... OK
Check             I-CSRRCI-01 ... OK
Check              I-CSRRS-01 ... OK
Check             I-CSRRSI-01 ... OK
Check              I-CSRRW-01 ... OK
Check             I-CSRRWI-01 ... OK
--------------------------------
OK: 6/6 RISCV_TARGET=neorv32 RISCV_DEVICE=rv32Zicsr RISCV_ISA=rv32Zicsr
```

RISC-V `rv32Zifencei` Tests

```
Check              I-FENCE.I-01 ... OK
--------------------------------
OK: 1/1 RISCV_TARGET=neorv32 RISCV_DEVICE=rv32Zifencei RISCV_ISA=rv32Zifencei
```

2.1.1 RISC-V Non-Compliance Issues

This list shows the *currently known* issues regarding full RISC-V-compliance.

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

The *Physical Memory Protection* (PMP) only supports the modes `OFF` and `NAPOT` yet. Also, the CPU only supports up to 8 regions.

2.1.2 NEORV32-Specific (Custom) Extensions

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

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

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

2.2. CPU Top Entity – Signals

The following table shows all interface ports of the CPU top entity (`rtl/core/neorv32_cpu.vhd`). The type of all signals is **`std_ulogic`** or **`std_ulogic_vector`**, respectively. A CPU wrapper providing resolved port signals can be found in `rtl/top_templates/neorv32_cpu_stdlogic.vhd`.

Signal Name	Width	Direction	Function	HW Module
Global Control				
clk_i	1	Input	Global clock line, all registers triggering on rising edge	global
rstn_i	1	Input	Global reset, **low-active**	
Instruction Bus Interface				
i_bus_addr_o	32	Output	Destination address	BUS_UNIT
i_bus_rdata_i	32	Input	Write data	
i_bus_wdata_o	32	Output	Read data	
i_bus_ben_o	4	Output	Byte enable	
i_bus_we_o	1	Output	Write transaction	
i_bus_re_o	1	Output	Read transaction	
i_bus_cancel_o	1	Output	Cancel current transfer	
i_bus_ack_i	1	Input	Bus transfer acknowledge from accessed peripheral	
i_bus_err_i	1	Input	Bus transfer terminate from accessed peripheral	
i_bus_fence_o	1	Output	Indicates an executed FENCEI instruction	
Data Bus Interface				
i_bus_addr_o	32	Output	Destination address	BUS_UNIT
i_bus_rdata_i	32	Input	Write data	
i_bus_wdata_o	32	Output	Read data	
i_bus_ben_o	4	Output	Byte enable	
i_bus_we_o	1	Output	Write transaction	
i_bus_re_o	1	Output	Read transaction	
i_bus_cancel_o	1	Output	Cancel current transfer	
i_bus_ack_i	1	Input	Bus transfer acknowledge from accessed peripheral	
i_bus_err_i	1	Input	Bus transfer terminate from accessed peripheral	
i_bus_fence_o	1	Output	Indicates an executed FENCE instruction	
System Time				
time_i	64	Input	System time input (from MTIME)	CONTROL
Interrupts (RISC-V-compliant)				
msw_irq_i	1	Input	RISC-V machine software interrupt	CONTROL
mext_irq_i	1	Input	RISC-V machine external interrupt	
mtime_irq_i	1	Input	RISC-V machine timer interrupt	
Fast Interrupts (custom extension)				
firq_i	4	Input	Fast interrupt request signals	CONTROL

Table 5: neorv32_cpu.vhd – CPU top entity interface ports

2.3. CPU Top Entity – Configuration Generics

This is a list of all configuration generics of the NEORV32 CPU top entity `rtl/neorv32_cpu.vhd`. The generic's name is shown in orange, the type in **black** and the default value in light gray.

2.3.1 General

HW_THREAD_ID **std_ulogic_vector(31 downto 0)** x"00000000"
The hart ID of the CPU. Can be read via the `mhartid` CSR. Hart IDs must be unique within a system.

CPU_BOOT_ADDR **std_ulogic_vector(31 downto 0)** x"00000000"
Defines the boot address of the CPU after reset.

2.3.2. RISC-V CPU Extensions

CPU_EXTENSION_RISCV_C **boolean** false
Implement the CPU extension for compressed instructions when `true`.

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

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

CPU_EXTENSION_RISCV_U **boolean** false
Implement user privilege level when `true`.

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 exception system will be excluded from synthesis. Hence, no interrupts and no exceptions can be detected.

The `CPU_EXTENSION_RISCV_Zicsr` should be **always enabled**.

CPU_EXTENSION_RISCV_Zifencei **boolean** true
Implement the instruction fetch synchronization instruction `ifetch.i`. For example, this option is required for self-modifying code.

2.3.3. Extension Options

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

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

2.3.4. Physical Memory Protection

PMP_USE **boolean** false

Implement physical memory protection (PMP) when `true`.

PMP_NUM_REGIONS **natural** 4

Defines the number of PMP regions. Allowed configurations: 1 to 8. With each additional region the according `pmpcfgx` and `pmpaddrx` CSR / CSR bits become available.

PMP_GRANULARITY **natural** 14

The PMP only supports the `NATOP` mode. This generic defines the minimal granularity. Allowed values: 1 (8-byte region), 1 (16-byte region), …, 30 (4GB region). Default is 14 (64kB region).

2.4. Instruction Set and CPU Extensions

32-bit Base ISA (`I` extension)

The CPU supports the complete RV32I base integer instruction set:

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

In order to keep the hardware footprint low, the CPU's shift unit uses a bit-serial approach. Shift operations are split in coarse shifts (multiples of 4) and a final fine shift (0 to 3). The total execution time depends on the shift amount.

The `FENCE` instruction performs the same instruction sync operations as the `FENCE.I` instruction (even if `Zifencei` is not enabled). Also, the top's `fence_o` signal is set high for one cycle to inform the memory system.

Embedded CPU Architecture (`E` extension)

This extensions does not feature additional instructions. However, the embedded CPU version only implements the lower 16 registers and uses a specific ABI (`ilp32e`) when the `CPU_EXTENSION_RISCV_E` configuration generic is `true`.

Compressed Instructions (`C` extension)

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

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

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

Integer Multiplication and Division (`M` extension)

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

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

By default, multiplication and division operations are executed in a bit-serial approach. Alternatively, the multiplier core can be implemented using DSP blocks when the `FAST_MUL_EN` generic is `true`. In that case, multiplications complete within 6 cycles.

User Privilege Level (`U` extension)

Add the less-privileged *user mode* when the `CPU_EXTENSION_RISCV_U` configuration generic is `true`.

Control and Status Register Access (`Zicsr` extension)

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

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

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

Instruction Coherency Operation (`Zifencei` extension)

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

- `FENCE.I`

The `FENCE.I` instruction resets the CPU's instruction fetch engine and flushes all prefetch buffers. This allows a clean re-fetch of modified data from memory. Also, the top's `fencei_o` signal is set high for one cycle to inform the memory system.

Physical Memory Protection (`PMP` extension)

The NEORV32 physical memory protection is compliant to the PPM specified by the RISC-V specs. The circuitry is implemented when the `PMP_USE` configuration generic is `true`. The actual number of available PMP configuration registers (`pmpcfgx`) and PMP address registers (`pmpaddrx`) is defined by the configured number of regions (`PMP_NUM_REGIONS`).

The NEORV32 PMP only supports the NAPOT mode. The minimal available region granularity can be configured via the `PMP_GRANULARITY` generic (min 8 bytes).

- CSRs: `pmpcfg0` `pmpcfg1` `pmpaddr0` `pmpaddr1` `pmpaddr2` `pmpaddr3` `pmpaddr4` `pmpaddr5` `pmpaddr6` `pmpaddr7`

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

2.5. Instruction Timing

The following table shows the required clock cycles for executing a certain instruction. The execution cycles assume a bus access without additional wait states and a filled pipelined.

Class	ISA / Extension	Instructions	Execution Cycles
ALU	I/E	ADDI SLTI SLTIU XORI ORI ANDI ADD SUB SLT SLTU XOR OR AND LUI AUIPC	2
	C	C.ADDI4SPN C.NOP C.ADDI C.LI C.ADDI16SP C.LUI C.ANDI C.SUB C.XOR C.OR C.AND C.ADD C.MV	
ALU – Shifts	I/E	SLLI SRLI SRAI SLL SRL SRA	2 + *sha*[3]/4 + *sha*%4 + 1 ***FAST_SHIFT***[4]: 3
	C	C.SRLI C.SRAI C.SLLI	
Branches	I/E	BEQ BNE BLT BGE BLTU BGEU	Taken: 2 + 5 Not taken: 3
	C	C.BEQZ C.BNEZ	
Jumps	I/E	JAL JALR	2 + 5
	C	C.JAL C.J C.JR C.JALR	
Memory	I/E	LB LH LW LBU LHU SB SH SW	5
	C	C.LW C.SW C.LWSP C.SWSP	
Multiplication	M	MUL MULH MULHSU MULHU	2 + 32 + 4 ***FAST_MUL***[5]*: 5*
Division	M	DIV DIVU REM REMU	2 + 32 + 6
CSR Access	Zicsr	CSRRW CSRRS CSRRC CSRRWI CSRRSI CSRRCI	4
System	I/E	ECALL EBREAK FENCE	3
	C	C.EBREAK	
	Zicsr	MRET WFI	
	Zifencei	FENCE.I	

Table 6: Clock cycles per instruction (optimal)

Average CPI for "Real" Applications

The average CPI (cycles per instructions) for executing the CoreMark benchmark for different CPU configurations is presented in chapter 1.6.2. Instruction Timing.

3 Shift amount: 0..31
4 Using a fast (but huge) barrel shifter for the CPU's shift operations; enabled via top's FAST_SHIFT_EN generic
5 Using DSPs for multiplication; enabled via top's FAST_MUL_EN generic

2.6. Control and Status Registers (CSRs)

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

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

If not otherwise mentioned, all CSRs are initialized with `0x0000_0000` after reset.

The NEORV32-specific CSRs (if available at all) are mapped to the official “*custom CSRs*” CSR address space.

When trying to write to a read-only CSR (like the `time` CSR) or when trying to access a non-existent CSR an illegal instruction exception is triggered.

CSR Listing

Notes for the following listing:

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

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

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

Address	Name [ASM]	Name [C]	R/W	Function	Note
Machine Trap Setup					
0x300	mstatus	CSR_MSTATUS	r/w	Machine status register	S
0x301	misa	CSR_MISA	r/-	Machine CPU ISA and extensions	R
0x304	mie	CSR_MIE	r/w	Machine interrupt enable register	C
0x305	mtvec	CSR_MTVEC	r/w	Machine trap-handler base address (for ALL traps)	
Machine Trap Handling					
0x340	mscratch	SCR_MSCRATCH	r/w	Machine scratch register	
0x341	mepc	CSR_MEPC	r/w	Machine exception program counter	
0x342	mcause	CSR_MCAUSE	r/w	Machine trap cause	
0x343	mtval	CSR_MTVAL	r/w	Machine bad address or instruction	
0x344	mip	CSR_MIP	r/w	Machine interrupt pending register	C
(Machine) Physical Memory Protection					
0x3a0	pmpcfg0	CSR_PMPCFG0	r/w	Physical memory protection configuration for region 0..3	S
0x3a1	pmpcfg1	CSR_PMPCFG1	r/w	Physical memory protection configuration for region 4..7	S
0x3b0	pmpaddr0	CSR_PMPADDR0	r/w	Physical memory protection address register region 0	

Address	Name [ASM]	Name [C]	R/W	Function	Note
0x3b1	pmpaddr1	CSR_PMPADDR1	r/w	Physical memory protection address register region 1	
0x3b2	pmpaddr2	CSR_PMPADDR2	r/w	Physical memory protection address register region 2	
0x3b3	pmpaddr3	CSR_PMPADDR3	r/w	Physical memory protection address register region 3	
0x3b4	pmpaddr4	CSR_PMPADDR4	r/w	Physical memory protection address register region 4	
0x3b5	pmpaddr5	CSR_PMPADDR5	r/w	Physical memory protection address register region 5	
0x3b6	pmpaddr6	CSR_PMPADDR6	r/w	Physical memory protection address register region 6	
0x3b7	pmpaddr7	CSR_PMPADDR7	r/w	Physical memory protection address register region 7	
Counters and Timers					
0xb00	mcycle	CSR_MCYCLE	r/w	Machine cycle counter low word	
0xb02	minstret	CSR_MINSTRET	r/w	Machine instructions-retired counter low word	
0xb80	mcycleh	CSR_MCYCLEH	r/w	Machine cycle counter low word	
0xb82	minstreth	CSR_MINSTRETH	r/w	Machine instructions-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/-	Instructions-retired counter low word	
0xc80	cycleh	CSR_CYCLEH	r/-	Cycle counter high word	
0xc81	timeh	CSR_TIMEH	r/-	System time (from MTIME) high word	
0xc82	instreth	CSR_INSTRETH	r/-	Instructions-retired counter high word	
Machine Information Registers, read-only					
0xf11	mvendorid	CSR_MVENDORID	r/-	Vendor ID	
0xf12	marchid	CSR_MARCHID	r/-	Architecture ID	
0xf13	mimpid	CSR_MIMPID	r/-	Machine implementation ID / version	
0xf14	mhartid	CSR_MHARTID	r/-	Machine thread ID	
NEORV32-Specific Custom CSRs					
0xfc0	-	CSR_MZEXT	r/-	Available Z* CPU extensions	C

Table 7: NEORV32 Control and Status Registers (CSRs)

2.6.1. Machine Trap Setup

Machine Status Register (mstatus) [0x300]

The `mstatus` CSR is compliant to the RISC-V specs. The following bits are implemented (all remaining bits are always zero and are read-only):

Bit#	Name [C]	R/W	Function
12:11	`MPP`	r/w	Previous machine privilege level, 11= machine (M) mode, 00= user (U) level
7	`MPIE`	r/w	Previous machine interrupt enable flag
3	`MIE`	r/w	Machine 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`.

ISA and Extensions (misa) [0x301]

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

The `misa` CSR is compliant to the RISC-V specs. The lowest 26 bits show the implemented CPU extensions. The following bits are implemented (all remaining bits are always zero and are read-only):

Bit#	Name [C]	R/W	Function
31:30	`CPU_MISA_MXL_HI_EXT` `CPU_MISA_MXL_LO_EXT`	r/-	32-bit architecture indicator (always “01”)
23	`CPU_MISA_X_EXT`	r/-	The **X** extension bit is always set to indicate custom non-standard extensions
20	`CPU_MISA_U_EXT`	r/-	**U** CPU extensions (user mode), wet when `CPU_EXTENSION_RISCV_U` enabled
12	`CPU_MISA_M_EXT`	r/-	**M** CPU extension (muld/div HW), set when `CPU_EXTENSION_RISCV_M` enabled
8	`CPU_MISA_I_EXT`	r/-	**I** CPU extension, always set, cleared when `CPU_EXTENSION_RISCV_E` enabled
4	`CPU_MISA_E_EXT`	r/-	**E** CPU extension (embedded), set when `CPU_EXTENSION_RISCV_E` enabled
2	`CPU_MISA_C_EXT`	r/-	**C** CPU extension (compressed instructions), set when `CPU_EXTENSION_RISCV_C` enabled

Machine Interrupt-Enable Register (mie) [0x304]

The `mie` CSR is compliant to the RISC-V specs. The following bits are implemented (all remaining bits are always zero and are read-only):

Bit#	Name [C]	R/W	Function
19	CPU_MIE_FIRQ3E	r/w	Fast interrupt channel 3 enable
18	CPU_MIE_FIRQ2E	r/w	Fast interrupt channel 2 enable
17	CPU_MIE_FIRQ1E	r/w	Fast interrupt channel 1 enable
16	CPU_MIE_FIRQ0E	r/w	Fast interrupt channel 0 enable
11	CPU_MIE_MEIE	r/w	Machine external interrupt enable
7	CPU_MIE_MTIE	r/w	Machine timer interrupt enable (from MTIME)
3	CPU_MIE_MSIE	r/w	Machine software interrupt enable

Machine Trap-Handler Base Address (mtvec) [0x305]

The `mtvec` CSR is compliant to the RISC-V specs. This register stores the base address for the machine trap handler. The CPU jumps to this address, regardless of the trap source. The lowest two bits of this register are always zero and cannot be altered.

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

2.6.2. Machine Trap Handling

Scratch Register for Machine Trap Handlers (mscratch) [0x340]

The `mscratch` CSR is compliant to the RISC-V specs. It is a general purpose scratch register that can be used by the exception/interrupt handler.

Machine Exception Program Counter (mepc) [0x341]

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

Machine Trap Cause (mcause) [0x342]

The `mcause` CSR is compliant to the RISC-V specs. It shows the cause of the current exception / interrupt (see chapter 2.7. Traps, Exceptions and Interrupts). The following bits are implemented:

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

Machine Bad Address or Instruction (mtval) [0x343]

The `mtval` CSR is compliant to the RISC-V specs. When a trap is triggered, the CSR shows either the faulting address (for misaligned/faulting load/stores/fetch) or the faulting instruction itself (for illegal instructions). For interrupts the CSR is set to zero.

Machine Interrupt Pending (mip) [0x344]

The `mip` CSR is compliant to the RISC-V specs but hast custom extension. The following bits are implemented (all remaining bits are always zero and are read-only):

Bit#	Name [C]	Note	R/W	Function
19	`CPU_MIP_FIRQ3P`	*custom*	r/-	Fast interrupt channel 3 pending
18	`CPU_MIP_FIRQ2P`	*custom*	r/-	Fast interrupt channel 2 pending
17	`CPU_MIP_FIRQ1P`	*custom*	r/-	Fast interrupt channel 1 pending
16	`CPU_MIP_FIRQ0P`	*custom*	r/-	Fast interrupt channel 0 pending
11	`CPU_MIP_MEIP`	RISC-V	r/-	Machine external interrupt pending
7	`CPU_MIP_MTIP`	RISC-V	r/-	Machine timer interrupt pending (from MTIME)
3	`CPU_MIP_MSIP`	RISC-V	r/-	Machine software interrupt pending

2.6.3. Physical Memory Protection

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

Physical Memory Protection Configuration Register 0 & 1 (pmpcfg0 & pmpcfg1) [0x3a0 - 0x3a1]

The `pmpcfg0` to `pmpcfg1` CSRs are compliant to the RISC-V specs. They are used to configure up to 8 protection regions. The following bits (for the first PMP configuration entry) are implemented (all remaining bits are always zero and are read-only):

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

Physical Memory Protection Address Registers 0 to 7 (pmaddrg0 to pmpaddr7) [0x3b0 - 0x3b7]

The `pmpaddr0` to `pmpaddr7` CSRs are compliant to the RISC-V specs. They are used to configure the base address and the region size for up to 8 regions.

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`.

2.6.4. Counters and Timers

These timers and counter can be used for performance evaluation of an application. The `[m]instret[h]` counters increment when an instruction enters the *execute* stage in the CPU's execute engine. The `[m]cycle[h]` counters increment with the CPU clock when the CPU is not in sleep mode.

Machine Cycle Counter – Low (mcycle) [0xb00]

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

Machine Instruction-Retired Counter – Low (minstret) [0xb02]

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

Machine Cycle Counter – High (mcycleh) [0xb80]

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

Machine Instruction-Retired Counter – High (minstreth) [0xb82]

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

Cycle Counter for RDCYCLE Instruction – Low (cycle) [0xc00]

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

System Time for RDTIME Instruction – Low (time) [0xc01]

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

Instructions-Retired Counter for RDINSTRET Instruction – Low (instret) [0xc02]

The `instret` CSR is compliant to the RISC-V specs. It shows the lower 32-bit of the number of retired instruction. The `instret` CSR is read-only and is a shadowed copy from the `minstret` CSR.

Cycle Counter for RDCYCLEH Instruction – High (cycleh) [0xc80]

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

System Time for RDTIMEH Instruction – High (timeh) [0xc81]

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

Instructions-Retired Counter for RDINSTRETH Instruction – High (instreth) [0xc82]

The `instreth` CSR is compliant to the RISC-V specs. It shows the upper 32-bit of the number of retired instruction. The `instreth` CSR is read-only and is a shadowed copy from the `minstreth` CSR.

2.6.5. Machine Information Registers

Vendor ID (mvendorid) [0xf11]

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

Architecture ID (marchid) [0xf12]

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

Implementation ID (mimpid) [0xf13]

The `mimpid` CSR is compliant to the RISC-V specs. It is read-only and shows the version of the NEORV32 as BCD-coded number (like 1.2.3.4).

Hardware Thread ID (mhartid) [0xf14]

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

2.6.6. NEORV32-Specific Custom CSRs

Z* CPU Extensions Indicator Register (mzext) [0xfc0]

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

Bit#	Name [C]	R/W	Function
2	`CPU_MZEXT_PMP`	`r/-`	Physical memory protection available (enabled via `PMP_USE` generic)
1	`CPU_MZEXT_ZIFENCEI`	`r/-`	`Zifencei` extensions available (enabled via `CPU_EXTENSION_RISCV_Zifencei` generic)
0	`CPU_MZEXT_ZICSR`	`r/-`	`Zicsr` extensions available (enabled via `CPU_EXTENSION_RISCV_Zicsr` generic)

2.7. Traps, Exceptions and Interrupts

The NEORV32 supports the following exceptions and instructions (traps). Whenever an exception or interrupt is triggered, the CPU transfers control to the address stored in the `mtvec` CSR. The cause of the according interrupt or exception can be determined via the content of the `mcause` CSR The address that was the current program counter when a trap was taken is stored to `mepc`. Additional information regarding the cause of the trap can be retrieved from `mtval`.

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

Custom Fast Interrupt Request Lines

As a custom extension, the NEORV32 CPU features 4 fast interrupt request lines via the `firq_i(3:0)` CPU top entity signals. These four interrupts have unique configuration and status flags in the `mie` and `mip` CSRs and also provide custom trap codes (see below).

Notes

The lines marked with an "***C***" are custom extensions. The **mepc** and **mtval** columns show the value written to `mepc`/`mtval` when a trap is triggered:

- `I-PC` Address of *interrupted* instruction
- `B-ADR` Bad memory access address that cause the trap
- `PC` Address of instruction that caused the trap
- `0` Zero
- `Inst` The faulting instruction itself

Priority	mcause	ID [C]	Function	mepc	mtval	
1	0x8000000B	TRAP_CODE_MEI	Machine external interrupt	I-PC	0	
2	0x80000007	TRAP_CODE_MTI	Machine timer interrupt (from MTIME)	I-PC	0	
3	0x80000003	TRAP_CODE_MSI	Machine software interrupt	I-PC	0	
4	0x80000010	TRAP_CODE_FIRQ_0	Fast interrupt request channel 0	I-PC	0	*C*
5	0x80000011	TRAP_CODE_FIRQ_1	Fast interrupt request channel 1	I-PC	0	*C*
6	0x80000012	TRAP_CODE_FIRQ_2	Fast interrupt request channel 2	I-PC	0	*C*
7	0x80000013	TRAP_CODE_FIRQ_3	Fast interrupt request channel 3	I-PC	0	*C*
8	0x00000001	TRAP_CODE_I_ACCESS	Instruction access fault	B-ADR	PC	
9	0x00000002	TRAP_CODE_I_ILLEGAL	Illegal instruction	PC	Inst	
10	0x00000000	TRAP_CODE_I_MISALIGNED	Instruction address misaligned	B-ADR	PC	
11	0x0000000B	TRAP_CODE_MENV_CALL	Environment call from M-mode (ECALL)	PC	PC	
12	0x00000003	TRAP_CODE_BREAKPOINT	Breakpoint (EBREAK)	PC	PC	
13	0x00000006	TRAP_CODE_S_MISALIGNED	Store address misaligned	B-ADR	B-ADR	
14	0x00000004	TRAP_CODE_L_MISALIGNED	Load address misaligned	B-ADR	B-ADR	
15	0x00000007	TRAP_CODE_S_ACCESS	Store access fault	B-ADR	B-ADR	
16	0x00000005	TRAP_CODE_L_ACCESS	Load access fault	B-ADR	B-ADR	

The [C] names are defined by the NEORV32 core library and can be used in plain C code.

2.8. Address Space

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

2.9. Bus Interface

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

2.9.1. Interface Signals

The following table shows the signals of the interfaces seen from the CPU (`*_o` signals are driven by the CPU, `*_i` signals are read by the CPU).

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

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

If there is not active transfer in progress (data or instructions) the state of the `bus_cancel_o` signal is irrelevant.

2.9.2. Protocol

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

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

The transfer can be completed directly in the next cycle after 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.

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

Bus Accesses

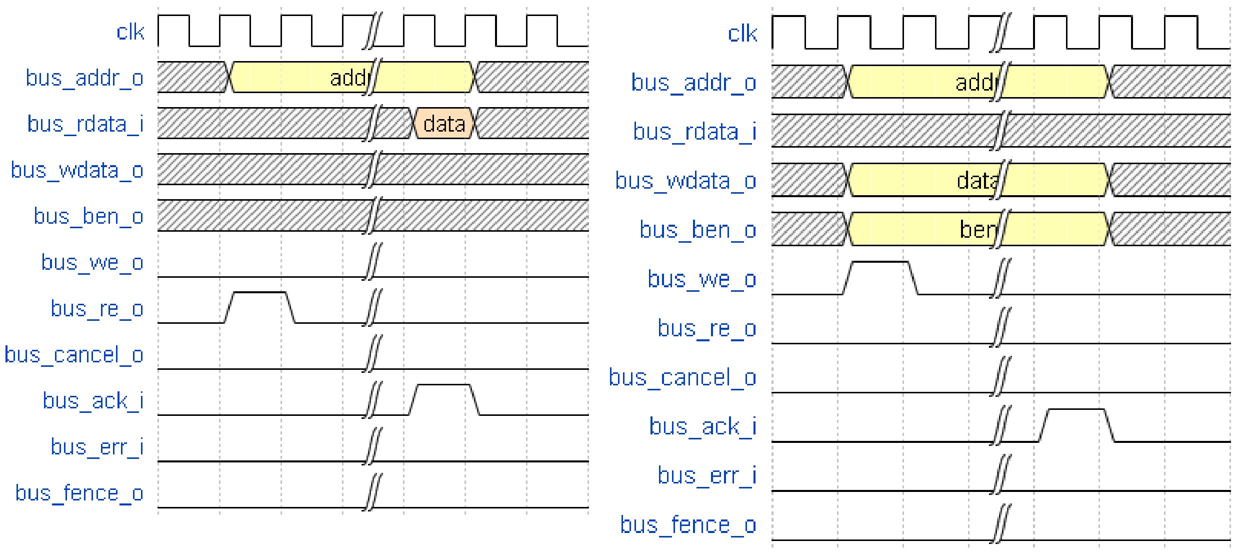

Figure 2: CPU interface read access (left) and write access (right)

Write Access

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

Read Access

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

Memory Barriers

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

Access Boundaries

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

3. NEORV32 Processor (SoC)

The NEORV32 Processor is built from the *NEORV32 CPU* together with common peripheral interfaces and embedded memories to provide a RISC-V-based full-scale microcontroller-like SoC platform.

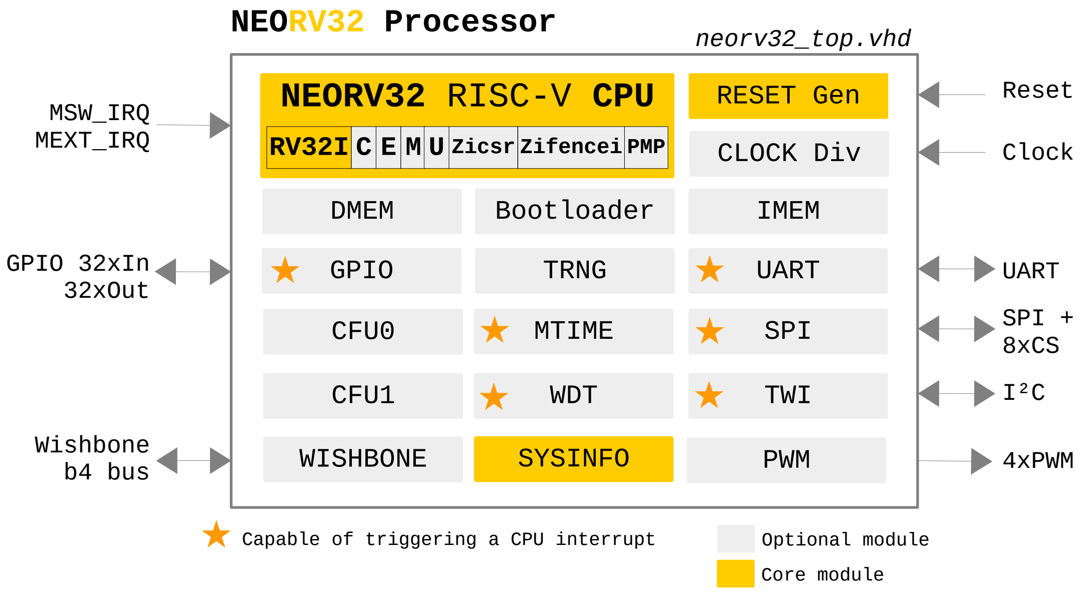

Figure 3: NEORV32 processor block diagram

Processor Key Features

- ✔ Optional processor-internal data and instruction memories (**DMEM/IMEM** → p.53)
- ✔ Optional internal **bootloader** with UART console and automatic SPI flash boot option (→ p.81)
- ✔ Optional machine system timer (**MTIME** → p.60), RISC-V-compliant
- ✔ Optional universal asynchronous receiver and transmitter (**UART** → p.61) with simulation output option via text.io
- ✔ Optional 8/16/24/32-bit serial peripheral interface controller (**SPI** → p.63) with 8 dedicated CS lines
- ✔ Optional two wire serial interface controller (**TWI** → p.65), compatible to the I²C standard
- ✔ Optional general purpose parallel IO port (**GPIO** → p.57), 16xOut, 16xIn
- ✔ Optional 32-bit external bus interface, Wishbone b4 compliant (**WISHBONE** → p.55)
- ✔ Optional watchdog timer (**WDT** → p.58)
- ✔ Optional PWM controller with 4 channels and 8-bit duty cycle resolution (**PWM** → p.67)
- ✔ Optional GARO-based true random number generator (**TRNG** → p.69)
- ✔ Optional custom functions units for custom co-processor extensions (**CFU0** & **CFU1** → p.71)
- ✔ System configuration information memory to check HW config. via software (**SYSINFO** → p.73)

3.1. Processor Top Entity – Signals

The following table shows all interface ports of the processor top entity (`rtl/core/neorv32_top.vhd`). The type of all signals is **`std_ulogic`** or **`std_ulogic_vector`**, respectively. A wrapper providing resolved port signals can be found in `rtl/top_templates/neorv32_top_stdlogic.vhd`.

Signal Name	Width	Direction	Function	HW Module
Global Control				
clk_i	1	Input	Global clock line, all registers triggering on rising edge	global
rstn_i	1	Input	Global reset, **low-active**	
External bus interface (Wishbone-compatible)				
wb_adr_o	32	Output	Destination address	WISHBONE
wb_dat_i	32	Input	Write data	
wb_dat_o	32	Output	Read data	
wb_we_o	1	Output	Write enable ('0' = read transfer)	
wb_sel_o	4	Output	Byte enable	
wb_stb_o	1	Output	Strobe	
wb_cyc_o	1	Output	Valid cycle	
wb_ack_i	1	Input	Transfer acknowledge	
wb_err_i	1	Input	Transfer error	
Advanced memory control signals				
fence_o	1	Output	Indicates an executed `fence` instruction	CPU
fencei_o	1	Output	Indicates an executed `fencei` instruction	
General Purpose Inputs & Outputs (GPIO)				
gpio_o	32	Output	General purpose parallel output	GPIO
gpio_i	32	Input	General purpose parallel input	
Universal Asynchronous Receiver/Transmitter (UART)				
uart_txd_o	1	Output	UART serial transmitter	UART
uart_rxd_i	1	Input	UART serial receiver	
Serial Peripheral Interface Controller (SPI)				
spi_sck_o	1	Output	SPI controller clock line	SPI
spi_sdo_o	1	Output	SPI serial data output	
spi_sdi_i	1	Input	SPI serial data input	
spi_csn_o	8	Output	SPI dedicated chip select lines 0..7 (low-active)	
Two-Wire Interface Controller (TWI)				
twi_sda_io	1	InOut	TWI serial data line	TWI
twi_scl_io	1	InOut	TWI serial clock line	
Pulse-Width Modulation Channels (PWM)				
pwm_o	4	Output	Pulse-width modulated channels	PWM
Interrupts				
mtime_irq_i	1	Input	Machine timer interrupt[6] (RISC-V)	CPU
msw_irq_i	1	Input	Machine software interrupt (RISC-V)	
mext_irq_i	1	Input	Machine external interrupt (RISC-V)	

Table 8: neorv32_top.vhd – processor's top entity interface ports

6 Only available if processor-internal machine system timer (MTIME) is disabled (IO_MTIME_USE = false)

3.2. Processor Top Entity – Configuration Generics

This is a list of all configuration generics of the NEORV32 processor top entity `rtl/neorv32_top.vhd`. The generic name is shown in **orange**, the type in **black** and the default value in light gray. Most of the configured settings can be determined by the software via the SYSINFO IO module (3.4.14. System Configuration Information Memory (SYSINFO)).

3.2.1 General

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

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

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

3.2.2. RISC-V CPU Extensions

See chapter 2. NEORV32 Central Processing Unit (CPU) for more information.

`CPU_EXTENSION_RISCV_C` **boolean** `false`
Implement the CPU extension for compressed instructions when `true`.

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

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

`CPU_EXTENSION_RISCV_U` **boolean** `false`
Implement user privilege level when `true`.

`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 exception system will be excluded from synthesis. Hence, no interrupts and no exceptions can be detected.

The `CPU_EXTENSION_RISCV_Zicsr` should be **always enabled**.

`CPU_EXTENSION_RISCV_Zifencei` **boolean** `true`
Implement the instruction fetch synchronization instruction `ifetch.i`. For example, this option is required for self-modifying code.

3.2.3. Extension Options

FAST_MUL_EN **boolean** false

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

FAST_SHIFT_EN **boolean** false

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

3.2.4. Physical Memory Protection

PMP_USE **boolean** false

Implement physical memory protection (PMP) when `true`.

PMP_NUM_REGIONS **natural** 4

Defines the number of PMP regions. Allowed configurations: 1 to 8. With each additional region the according `pmpcfgx` and `pmpaddrx` CSR / CSR bits become available.

PMP_GRANULARITY **natural** 14

The PMP only supports the `NATOP` mode. This generic defines the minimal granularity. Allowed values: 1 (8-byte region), 1 (16-byte region), …, 30 (4GB region). Default is 14 (64kB region).

3.2.5. Internal Instruction Memory

See chapter 3.3. Address Space and 3.4.1. Instruction Memory (IMEM) for more information.

MEM_INT_IMEM_USE **boolean** true

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

MEM_INT_IMEM_SIZE **natural** 16*1024

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

MEM_INT_IMEM_ROM **boolean** false

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

3.2.6. Internal Data Memory

See chapter 3.3. Address Space and 3.4.2. Data Memory (DMEM) for more information.

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

MEM_INT_DMEM_SIZE **natural** 8*1024
Size in bytes of the processor-internal data memory (DMEM). Has no effect when `MEM_INT_DMEM_USE` is `false`.

3.2.7. External Memory Interface

See chapter 3.3. Address Space and 3.4.4. Processor-External Memory Interface (WISHBONE) for more information.

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

MEM_EXT_REG_STAGES **natural** 2
Defines the number of register stages inside the external bus gateway. Allowed configurations: 0, 1 or 2. Adding register stages increases the bus access latency but will also improve timing.

3.2.8. Processor Peripherals

See chapter 3.4. Processor-Internal Modules for more information.

IO_GPIO_USE **boolean** true
Implement general purpose input/output port unit (GPIO) when `true`. When disabled, the `gpio_i` signal is unconnected and the `gpio_o` signal is always low. See chapter 3.4.5. General Purpose Input and Output Port (GPIO) for more information.

IO_MTIME_USE **boolean** true
Implement machine system timer (MTIME) when `true`. When disabled, the CPU's machine timer interrupt is not available. The `CPU_EXTENSION_RISCV_Zicsr` has to be enabled if you want to use the machine system timer's interrupt. See chapter 3.4.7. Machine System Timer (MTIME) for more information.

IO_UART_USE **boolean** true
Implement universal asynchronous receiver/transmitter (UART) when `true`. When disabled, the `uart_rxd_i` signal is unconnected and the `uart_txd_o` signal is always low. See chapter 3.4.8. Universal Asynchronous Receiver and Transmitter (UART) for more information.

IO_SPI_USE **boolean** true
Implement serial peripheral interface controller (SPI) when `true`. When disabled, the `spi_miso_i` signal is unconnected, the `spi_sclk_o` and `spi_mosi_o` signals are always low and the `spi_csn_o` signal is always high. See chapter 3.4.9. Serial Peripheral Interface Controller (SPI) for more information.

IO_TWI_USE **boolean** true
Implement two-wire interface controller (TWI) when `true`. When disabled, the `twi_sda_io` and `twi_scl_io` signals are unconnected. See chapter 3.4.10. Two Wire Serial Interface Controller (TWI) for more information.

IO_PWM_USE **boolean** true
Implement pulse-width modulation controller (PWM) when `true`. When disabled, the `pwm_o` signal is always low. See chapter 3.4.11. Pulse Width Modulation Controller (PWM) for more information.

IO_WDT_USE **boolean** true
Implement watchdog timer (WDT) when `true`. See chapter 3.4.6. Watchdog Timer (WDT) for more information.

IO_TRNG_USE **boolean** false
Implement true-random number generator (TRNG) when `true`. See chapter 3.4.12. True Random Number Generator (TRNG) for more information.

IO_CFU0_USE **boolean** false
Implement custom functions unit 0 (CFU0) when `true`. See chapter 3.4.13. Custom Functions Units 0 and 1 (CFU0 & CFU1) or more information.

IO_CFU1_USE **boolean** false
Implement custom functions unit 1 (CFU1) when `true`. See chapter 3.4.13. Custom Functions Units 0 and 1 (CFU0 & CFU1) or more information.

3.3. Address Space

The 4GB address space of the NEORV32 Processor is divided into 4 main regions:

- The **instruction memory space** for instructions and constants.
- The **data memory space** for application runtime data (heap, stack, …).
- The **bootloader** address space for the processor-internal bootloader.
- The **IO/peripheral address space** for the processor-internal IO/peripheral devices (e.g., UART).

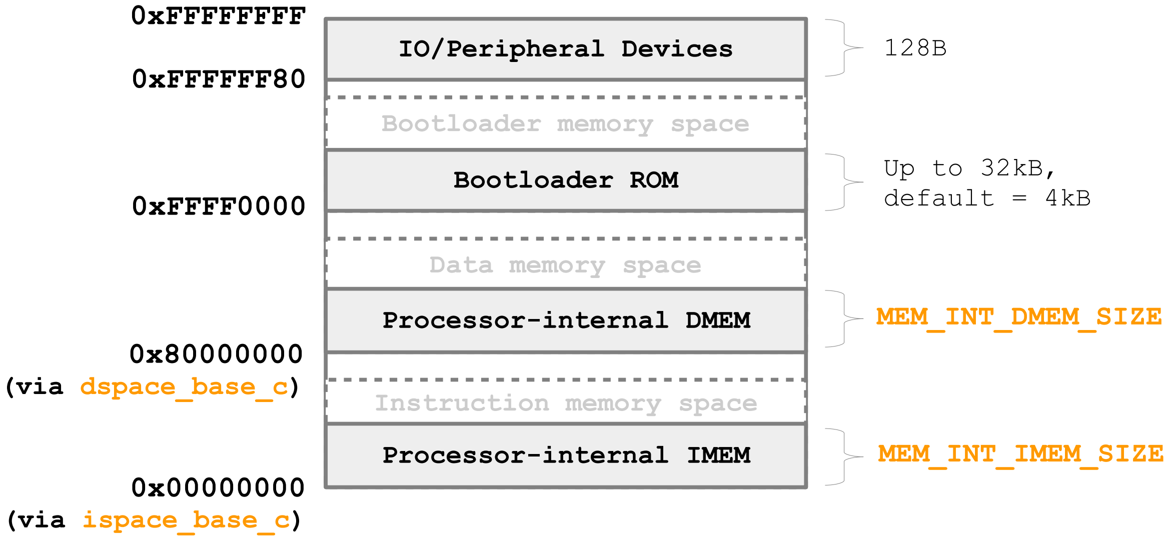

Figure 4: Default NEORV32 processor address space layout

General Address Space Layout

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

```
-- Architecture Configuration ----------------------------------------------------
-- ----------------------------------------------------------------------------------
constant ispace_base_c : std_ulogic_vector(data_width_c-1 downto 0) := x"00000000";
constant dspace_base_c : std_ulogic_vector(data_width_c-1 downto 0) := x"80000000";
```

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

The base address of the bootloader (at `0xFFFF0000`) and the IO region (at `0xFFFFFF80`) for the peripheral devices are also defined in the package and are fixed. These address regions cannot be used for other applications – even if the bootloader or all IO devices are not implemented.

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

- Both base addresses have to be aligned to a 4-byte boundary.
- Both base addresses have to be aligned to the according internal memory sizes. For example the `dspace_base_c` data space base address has to be aligned to the size of the `DMEM` (`MEM_INT_DMEM_SIZE`).

CPU Access

The CPU can access all of the 4GB address space from the instruction fetch interface and also from the data access interface. These two CPU interfaces are multiplexed by a simple bus switch[7] (`rtl/core/neorv32_busswitch.vhd`) into a single processor-internal bus. All internal memories, peripherals and also the external memory interface are connected to this internal bus. Hence, both CPU interfaces access the same (identical) address space.

Internal Memories

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

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

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

External Memory Interface

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

- Access to the processor-internal IMEM and processor-internal IMEM is implemented
- Access to the processor-internal DMEM and processor-internal DMEM is implemented
- Access to the bootloader ROM – even if the bootloader is not implemented
- Access to the IO area – even if some/all IO/peripheral devices are not implemented

7 The bus switch allows the CPU's data accesses to have higher priority than instruction fetch accesses.

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

External Memory Interface – Instruction Memory Example

`MEM_INT_IMEM_USE` = `true`, `MEM_INT_IMEM_SIZE` = `1024 byte`, `MEM_EXT_USE` = `true`

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

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

Internal Clock Divider

An internal clock divider generates 8 clock signals derived from the processor's main clock input `clk_i`. These derived clock signals are not actual *clock signals*. Instead, they are derived from a simple counter and are used as "clock enable" signal by the different processor modules. Thus, the whole design operates using only the main clock signal (single clock domain). Some of the processor peripherals like the Watchdog or the UART can select one of the derived clock enabled signals for their internal operation. If none of the connected modules require a clock signal from the divider, it is automatically deactivated to reduce dynamic power.
The peripheral devices, which feature a time-based configuration, provide a three-bit prescaler select in their according control register to select one out of the eight available clocks. The mapping of the prescaler select bits to the actually obtained clock are shown in the table below. Here, *f* represents the processor main clock from the top entity's `clk_i` signal.

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

Fast Interrupt Request Lines

The NEORV32 CPU features four independent fast interrupt channels implemented as custom extensions. The channels are used to signal interrupts from the IO/peripheral modules to the CPU.

Channel	Priority	Source Module	Description
0	`1 (highest)`	`WDT`	Watchdog interrupt
1	`2`	`GPIO`	GPIO input pin-change interrupt
2	`3`	`UART`	UART RX done interrupt **or** TX completed interrupt
3	`4 (lowest)`	`SPI` **or** `TWI`	SPI **or** TWI transmission done interrupt

Table 9: Fast IRQ mapping for the NEORV32 processor

Peripheral Devices

The processor-internal peripheral/IO devices are located at the end of the 32-bit address space at base address `0xFFFFFF80`. A region of `128 bytes` is reserved for this devices. Hence, all peripheral/IO devices are accessed using a memory-mapped scheme. A special linker script as well as the NEORV32 core software library abstract the specific memory layout for the user.

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

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

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.

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

Nomenclature for the Peripheral/IO Devices Listing

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

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

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

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

`0/w` These registers / register 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 / register 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.

3.4.1. Instruction Memory (IMEM)

Overview

Hardware source file(s):	`neorv32_imem.vhd`	
Software driver file(s):	`none`	Implicitly used
Top entity ports:	`none`	
Configuration generics:	`MEM_INT_IMEM_USE`	Implement processor-internal IMEM when `true`
	`MEM_INT_IMEM_SIZE`	IMEM size in bytes
	`MEM_INT_IMEM_ROM`	Implement IMEM as ROM when `true`

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

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

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

3.4.2. Data Memory (DMEM)

Overview

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

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

The DMEM is always implemented as RAM.

3.4.3. Bootloader ROM (BOOTROM)

Overview

Hardware source file(s):	`neorv32_boot_rom.vhd`	
Software driver file(s):	`none`	Implicitly used
Top entity ports:	`none`	
Configuration generics:	`BOOTLOADER_USE`	Implement bootloader when `true`

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

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

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

Boot Configuration

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

3.4.4. Processor-External Memory Interface (WISHBONE)

Overview

Hardware source file(s):	`neorv32_wishbone.vhd`	
Software driver file(s):	`none`	Implicitly used
Top entity ports:	`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
	`wb_sel_o`	Byte enable (4-bit)
	`wb_stb_o`	Strobe
	`wb_cyc_o`	Valid cycle
	`wb_ack_i`	Acknowledge
	`wb_err_i`	Bus error
	`fence_o`	Indicates an executed `fence` instruction
	`fencei_o`	Indicates an executed `fence.i` instruction
Configuration generics:	`MEM_EXT_USE`	Enable external memory interface when `true`
	`MEM_EXT_REG_STAGES`	Number of interface register stages
Configuration constants: → VHDL package file `neorv32_package.vhd`	`bus_timeout_c`	Number of cycles after which a valid bus transaction will abort and trigger an exception
	`wb_pipe_mode_c`	When false (default): Classic/standard Wishbone protocol; when true: Pipelined Wishbone protocol

The external memory interface uses the Wishbone interface protocol. The external interface port is available when the `MEM_EXT_USE` generic is `true`. This interface can be used to attach external memories, custom hardware accelerators additional IO devices or all other kinds of IP blocks.

All memory accesses from the CPU, that do not target the internal bootloader ROM, the internal IO region or the internal data/instruction memories (if implemented at all) are forwarded to the Wishbone gateway and thus to the external memory interface.

Latency

The Wishbone gateway can be configured to provide additional register stages to ease timing closure. The `MEM_EXT_REG_STAGES` generic defines the number of register stages:

- 0: No register stages; no additional latency
- 1: Processor-outgoing signals are buffered; 1 cycle additional latency
- 2: Processor-outgoing and -incoming signals are buffered; 2 cycles additional latency

Bus Access Timeout

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

The processor-internal memories and peripherals will always acknowledge the transfers within two cycles. Of course, a bus timeout will occur if accessing unused address locations. For example, a bus timeout and thus, a load/store bus access fault will occur when trying to access an IO device that has not been implemented.

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

```
-- Architecture Configuration ----------------------------------------------------
-- ----------------------------------------------------------------------------------
...
constant bus_timeout_c : natural := 127;
```

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

The bus timeout value is defined for the external memory interface but also applies when accessing processor-internal modules like memories or IO device. Hence, this parameter must not be less than one cycle.

Wishbone Bus Protocol

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

```
-- Architecture Configuration ----------------------------------------------------
-- ----------------------------------------------------------------------------------
...
constant wb_pipe_mode_c : boolean := false;
```

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

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.

3.4.5. General Purpose Input and Output Port (GPIO)

Overview

Hardware source file(s):	`neorv32_gpio.vhd`	
Software driver file(s):	`neorv32_gpio.c` `neorv32_gpio.h`	
Top entity ports:	`gpio_o`	32-bit parallel output port
	`gpio_i`	32-bit parallel input port
Configuration generics:	`IO_GPIO_USE`	Implement GPIO port unit when `true`
CPU interrupts:	`Fast IRQ channel 1`	Pin-change interrupt

Theory of Operation

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

Pin-Change Interrupt

The parallel input port `gpio_i` features a pin-change interrupt. Whenever an input pin has a low-to-high or high-to-low transition, the interrupt is triggered. By default, the pin-change interrupt is disabled and can be enabled using a bit mask that has to be written to the `GPIO_INPUT` register. Each set bit in this mask enables the pin-change interrupt for the corresponding input pin. If the modules is disabled for implementation, the pin-change interrupt is also permanently disabled.

Register Map

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

Table 10: GPIO port unit register map

3.4.6. Watchdog Timer (WDT)

Overview

Hardware source file(s):	`neorv32_wdt.vhd`	
Software driver file(s):	`neorv32_wdt.c` `neorv32_wdt.h`	
Top entity ports:	`none`	
Configuration generics:	`IO_WDT_USE`	Implement Watchdog timer when `true`
CPU interrupts:	`Fast IRQ channel 0`	Watchdog timer overflow

Theory of Operation

The watchdog (WDT) provides a last resort for safety-critical applications. The WDT has a free running 20-bit 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.

The watchdog is enabled by setting the `WDT_CT_EN` bit. The clock used to increment the internal counter is selected via the 3-bit `WDT_CT_CLK_SWLx` prescaler:

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

Whenever the internal timer overflow, the watchdog executes one of two possible actions: Either a hard processor reset or an interrupt request to the CPU's fast interrupt channel #0. The `WDT_CT_MODE` bit defines the action to take on overflow: When cleared, the Watchdog will trigger an IRQ, when set the WDT will cause a system reset.

The cause of the last action of the Watchdog can be determined via the `WDT_CT_CAUSE` flag. If this flag I zero, the processor has been reset via the external reset pin. If this flag is set, the last action (reset or interrupt) was caused by a Watchdog timer overflow. The `WDT_CT_PWFAIL` flag is set, when the last Watchdog action was triggered by an illegal access to the Watchdog control register.

The Watchdog control register can only be accessed when the access password is present in bits 15:8 of the written data. The default Watchdog password is: `0x47`
The watchdog is reset whenever a valid write access to the unit's control register is performed.

Register Map

Address	Name [C]	Bit(s) (Name) [C]		R/W	Function
0xFFFFFF8C	WDT_CT	0	WDT_CT_CLK_SEL0	r/w	Clock prescaler select bit 0
		1	WDT_CT_CLK_SEL1	r/w	Clock prescaler select bit 1
		2	WDT_CT_CLK_SEL2	r/w	Clock prescaler select bit 2
		3	WDT_CT_EN	r/w	Watchdog enable
		4	WDT_CT_MODE	r/w	Overflow action: 1: reset, 0: IRQ
		5	WDT_CT_CAUSE	r/-	Cause of last WDT action
		6	WDT_CT_PWFAIL	r/-	Last WDT action caused by wrong pwd
		15:8	WDT_CT_PASSWORD	0/w	Watchdog access password
		31..16	-	r/-	*Reserved, read as zero*

Table 11: WDT register map

3.4.7. Machine System Timer (MTIME)

Overview

Hardware source file(s):	`neorv32_mtime.vhd`	
Software driver file(s):	`neorv32_mtime.c` `neorv32_mtime.h`	
Top entity ports:	`none`	
Configuration generics:	`IO_MTIME_USE`	Implement MTIME when `true`
CPU interrupts:	`MTI`	Machine timer interrupt

Theory of Operation

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

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

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

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

Register Map

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

Table 12: MTIME register map

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

3.4.8. Universal Asynchronous Receiver and Transmitter (UART)

Overview

Hardware source file(s):	`neorv32_uart.vhd`	
Software driver file(s):	`neorv32_uart.c` `neorv32_uart.h`	
Top entity ports:	`uart_txd_o`	Serial transmitter output
	`uart_rxd_o`	Serial receiver input
Configuration generics:	`IO_UART_USE`	Implement UART when `true`
CPU interrupts:	`Fast IRQ channel 2`	TX done or RX done

Theory of Operation

In most cases, the UART is a standard interface used to establish a communication channel between the computer/user and an application running on the processor platform. The NEORV32 UART features a standard configuration frame configuration: 8 data bits, 1 stop bit and no parity bit. These values are fixed. The actual Baudrate is configurable by software.

The UART is enabled when the `UART_CT_EN` bit in the UART control register is set. The actual transmission Baudrate (like “19200”) is configured via the 12-bit `UART_CT_BAUDxx` value and the 3-bit `UART_CT_PRSCx` clock prescaler.

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

$$Baudrate = \frac{f_{main}[Hz]}{\text{Prescaler} \cdot \text{UART\_CT\_BAUD}}$$

A new transmission is started by writing the data byte to the lowest byte of the `UART_DATA` register. The transfer is completed when the `UART_CT_TX_BUSY` control register flag returns to zero. A new received byte is available when the `UART_DATA_AVAIL` flag of the `UART_DATA` register is set. If a new byte is received before the previous one has been read by the CPU, the receiver overrun flag `UART_CT_RXOR` is set.

The UART has a single interrupt, which can be trigger by two sources: The interrupt is triggered when a transmission has finished and the `UART_CT_TX_IRQ` flag is set. Additionally, the interrupt can also be triggered when a data byte has been received and the `UART_CT_RX_IRQ` flag is set.

If the UART is not implemented, the UART’s serial output port is tied to zero and the UART’s interrupt is unavailable.

Register Map

Address	Name [C]	Bit(s) (Name) [C]		R/W	Function
0xFFFFFFA0	UART_CT	11:0	UART_CT_BAUDxx	r/w	12-bit BAUD configuration value
		12	*UART_CT_SIM_MODE*	*r/w*	*Enable simulation output mode (see below)*
		24	UART_CT_PRSC0	r/w	Baudrate clock prescaler select bit 0
		25	UART_CT_PRSC1	r/w	Baudrate clock prescaler select bit 1
		26	UART_CT_PRSC2	r/w	Baudrate clock prescaler select bit 2
		27	UART_CT_RXOR	r/-	UART receiver overrun
		28	UART_CT_EN	r/w	UART enable
		29	UART_CT_RX_IRQ	r/w	RX complete IRQ enable
		30	UART_CT_TX_IRQ	r/w	TX done IRQ enable
		31	UART_CT_TX_BUSY	r/-	Transceiver busy flag
0xFFFFFFA4	UART_DATA	7:0	UART_DATA_LSB/MSB	r/w	Receive/transmit data (8-bit)
		31:0	*-*	*-/w*	*Simulation data output*
		31	UART_DATA_AVAIL	r/-	RX data available when set

Table 13: UART register map

Simulation Mode

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

The simulation mode is enabled by setting the `UART_CT_SIM_MODE` bit in the UART's control register `UART_CT`. Any further UARt configuration bits are irrelevant, but the UART has to be enabled via the `UART_CT_EN` bit.

When the simulation mode is enabled, every written char (in bits 7:0) to `UART_DATA` is directly output as ASCII char to the simulator console. Additionally, all text is also stored to a text file `neorv32.uart.sim_mode.text.out` in the simulation home folder. Furthermore, the whole 32-bit word written to `UART_DATA` is stored as plain 8-char hexadecimal value to a second text file `neorv32.uart.sim_mode.data.out` also located in the simulation home folder

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

If the UART simulation mode is enabled "on real hardware" there will be no UART transmissions at all.

3.4.9. Serial Peripheral Interface Controller (SPI)

Overview

Hardware source file(s):	`neorv32_spi.vhd`	
Software driver file(s):	`neorv32_spi.c` `neorv32_spi.h`	
Top entity ports:	`spi_sck_o`	1-bit serial controller clock output
	`spi_sdo_o`	1-bit serial controller data output
	`spi_dsi_i`	1-bit serial controller data input
	`spi_csn_o`	8-bit chip select port (low-active)
Configuration generics:	`IO_SPI_USE`	Implement SPI when `true`
CPU interrupts:	`Fast IRQ channel 3`	Transmission done interrupt

Theory of Operation

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

The SPI unit is enabled via the `SPI_CT_EN` bit. The idle clock polarity is configured via the `SPI_CT_CPHA` bit and can be low (0) or high (1) during idle. Data is shifted in/out with MSB first when the `SPI_CT_DIR` bit is cleared; data is sifted in/out LSB-first when the flag is set. The data quantity to be transferred within a single transmission is defined via the `SPI_CT_SIZEx` bits. The unit supports 8-bit ("00"), 16-bit ("01"), 24-bit ("10") and 32-bit ("11") transfers. Whenever a transfer is completed, an interrupt is triggered when the `SPI_CT_IRQ_EN` bit is set. A transmission is still in progress as long as the `SPI_CT_BUSY` flag is set. T he SPI controller features 8 dedicated chip-select lines. These lines are controlled via the control register's `SPI_CT_CSx` bits. When the CSx bit is set, the according chip select line `spi_csn_o(x)` goes low (low-active chip select lines)

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

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

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

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

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

Register Map

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

Table 14: SPI transceiver register map

3.4.10. Two Wire Serial Interface Controller (TWI)

Overview

Hardware source file(s):	neorv32_twi.vhd	
Software driver file(s):	neorv32_twi.c neorv32_twi.h	
Top entity ports:	twi_sda_io	Bi-directional serial data line
	twi_scl_io	Bi-directional serial clock line
Configuration generics:	IO_TWI_USE	Implement TWI when true
CPU interrupts:	Fast IRQ channel 3	Transmission done interrupt

Theory of Operation

The two wire interface – actually called I²C – is a quite famous interface for connecting several on-board components. Since this interface only needs two signals (the serial data line SDA and the serial clock line SCL) – despite of the number of connected devices – it allows easy interconnections of several peripheral nodes. The NEORV32 TWI implements a TWI **controller**. It features "**clock stretching**", so a slow peripheral can halt the transmission by pulling the SCL line low. **Currently no multi-controller support is available. Also, the TWI unit cannot operate in peripheral mode.**

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

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

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

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

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

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

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

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

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

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

Register Map

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

Table 15: TWI register map

3.4.11. Pulse Width Modulation Controller (PWM)

Overview

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

Theory of Operation

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

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

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

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

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

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

The resulting PWM frequency is defined by:

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

Register Map

Address	Name [C]	Bit(s) (Name) [C]		R/W	Function
0xFFFFFFB8	PWM_CT	0	PWM_CT_EN	r/w	PWM controller enable
		1	PWM_CT_PRSC0	r/w	Clock prescaler select bit 0
		2	PWM_CT_PRSC1	r/w	Clock prescaler select bit 1
		3	PWM_CT_PRSC2	r/w	Clock prescaler select bit 2
0xFFFFFFBC	PWM_DUTY	7:0	PWM_DUTY_CH0	r/w	8-bit duty cycle for channel 0
		15:8	PWM_DUTY_CH1	r/w	8-bit duty cycle for channel 1
		23:16	PWM_DUTY_CH2	r/w	8-bit duty cycle for channel 2
		31:24	PWM_DUTY_CH3	r/w	8-bit duty cycle for channel 3

Table 16: PWM controller register map

3.4.12. True Random Number Generator (TRNG)

Overview

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

Theory of Operation

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

The TRNG features a platform independent architecture without primitives or attributes. The concept is based on two papers, which are cited at the bottom of the following pages.

Architecture

The NEORV32 TRNG is based on the *GARO **Ga**lois **R**ing **O**scillator TRNG*[8]. Basically, this architecture is an asynchronous LFSR constructed from a chain of inverters. Before the output signal of one inverter is passed to the input of the next one, the signal can be XORed with the final output signal of the inverter chain (see image below) using a switching mask (f).

To prevent the synthesis tool from doing logic optimization and thus, removing all but one inverter, the TRNG uses simple latches to decouple an inverter and its actual output. The latches are reset when the TRNG is disabled and are enabled one by one by a simple shift register when the TRNG is activated. By this, the TRNG provides a platform independent architecture[9] since no specific VHDL attributes are required.

The default setup of the TRNG uses a total of 15 inverters and 2 GARO chains. The outputs of both chains are XORed to generate a final 1-bit random signal. This output signal is de-biased using a simple 2-bit Von-Neuman randomness extractor. The output from the de-biasing stage is fed to a simple 8-bit LFSR-based post-processing circuit to improve whitening. If the de-biasing fails, additional cycles are required to obtain a new random sample. This process might repeat depending on the quality of the GARO oscillation.

Each GARO chain features a simple online health monitoring, which checks the output stream for being stuck at zero or one, respectively.

8 "Enhancing the Randomness of a Combined True Random Number Generator Based on the Ring Oscillator Sampling Method" by Mieczyslaw Jessa and Lukasz Matuszewski
9 "Extended Abstract: The Butterfly PUF Protecting IP on every FPGA" by Sandeep S. Kumar, Jorge Guajardo, Roel

Using the TRNG

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

If the `TRNG_CT_ERROR_0` or the `TRNG_CT_ERROR_1` bit is set, the online health monitoring has detected a stuck-at-zero/stuck-at-one error in one of the GARO chains. In this case, the TRNG has to be disabled and re-enabled via `TRNG_CT_EN` to clear these error flags and to resume normal operation.

The `TRNG_CT_VALID` bit might also be set even if there is a stuck-at-zero/stuck-at-one error. Hence, the health monitoring bits should be checked before checking the actual valid flag.

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

Register Map

Address	Name [C]	Bit(s) (Name) [C]		R/W	Function
`0xFFFFFF88`	`TRNG_CT`	`7:0`	`TRNG_CT_DATA_LSB` `TRNG_CT_DATA_MSB`	`r/-`	8-bit random data output
		`15`	`TRNG_CT_VALID`	`r/-`	Random data output is valid when set
		`16`	`TRNG_CT_ERROR_0`	`r/-`	Stuck-at-zero error
		`17`	`TRNG_CT_ERROR_1`	`r/-`	Stuck-at-one error
		`31`	`TRNG_CT_EN`	`r/w`	TRNG enable

Table 17: TRNG register map

3.4.13. Custom Functions Units 0 and 1 (CFU0 & CFU1)

Overview

Hardware source file(s):	`neorv32_cfu0.vhd` `neorv32_cfu1.vhd`	
Software driver file(s):	`none`	Has to be implemented by user
Top entity ports:	`none`	None by default, can be implemented by user
Configuration generics:	`IO_CFU0_USE` `IO_CFU1_USE`	Implement CFU 0 when `true` Implement CFU 1 when `true`

Theory of Operation

The custom functions units (CFU0 and CFU1) are intended for tightly-coupled custom co-processors. In contrast to connecting custom hardware accelerators via the external memory interface, the CFUs provide a convenient and low-latency extension/customization option.

The default VHDL sources files, which are simple templates, are `rtl/core/neorv32_cfu0.vhd` for CFU 0 and `rtl/core/neorv32_cfu1.vhd` for CFU 1.

Each CFU provides four memory-mapped interface registers (see tables below). The actual function of these register has to be defined by the hardware designer. By default, all registers provide simple read and write access capabilities. The CFU VHDL source files provides several comments and notes for the implementing custom hardware.

As an example the four registers of CFU 0 could be used in the following way:

- `CFU0_REG_0`: Global control register
- `CFU0_REG_1`: Data read/write FIFO
- `CFU0_REG_2`: Command FIFO
- `CFU0_REG_3`: Status register

The CFU interface register can be accessed using the provided C-language aliases (see tables below).

```
// C code usage example
CFU0_REG_1 = some_data_array(i);       // write to CFU 0 register 0
some_other_data_array(i) = CFU0_REG_1; // read from CFU 0 register 0
```

CFU Signals

Besides the clock signal and the CPU interface bus, each CPU also provides a low-active asynchronous reset input (`rstn_i`) and 8 “derived clocks” (`clkgen_i`) for generating precise timing tasks. These signals have to be used as clock enable signals rather than as “real clocks”. The derived clock inputs are available when the clock generator enable output (`clkgen_en_o`) is set. See the CFU VHDL source files for more information.

Register Map

Address	Name [C]	Bit(s)	R/W	Function
0xFFFFFFC0	CFU0_REG_0	31:0	(r)/(w)	CFU 0 custom interface register 0
0xFFFFFFC4	CFU0_REG_1	31:0	(r)/(w)	CFU 0 custom interface register 1
0xFFFFFFC8	CFU0_REG_2	31:0	(r)/(w)	CFU 0 custom interface register 2
0xFFFFFFCC	CFU0_REG_3	31:0	(r)/(w)	CFU 0 custom interface register 3

Table 18: CFU 0 register map

Address	Name [C]	Bit(s)	R/W	Function
0xFFFFFFD0	CFU1_REG_0	31:0	(r)/(w)	CFU 1 custom interface register 0
0xFFFFFFD4	CFU1_REG_1	31:0	(r)/(w)	CFU 1 custom interface register 1
0xFFFFFFD8	CFU1_REG_2	31:0	(r)/(w)	CFU 1 custom interface register 2
0xFFFFFFDC	CFU1_REG_3	31:0	(r)/(w)	CFU 1 custom interface register 3

Table 19: CFU 1 register map

3.4.14. System Configuration Information Memory (SYSINFO)

Overview

Hardware source file(s):	`neorv32_sysinfo.vhd`	
Software driver file(s):	`(neorv32.h)`	(Registers and bits definitions)
Top entity ports:	`none`	
Configuration generics:	`*`	Shows the settings of most configuration generics
CPU interrupts:	`none`	

Theory of Operation

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

This devices is always implemented – regardless of the actual hardware configuration. The bootloader as well as the NEORV32 software runtime environment require information (like memory layout) for correct operation.

Register Map

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

Table 20: SYSINFO register map

SYSINFO_FEATURES

Bit#	Name [C]	Function
25	SYSINFO_FEATURES_IO_CFU1	Set when the custom functions unit 1 is implemented (via the IO_CFU1_USE generic)
24	SYSINFO_FEATURES_IO_TRNG	Set when the TRNG is implemented (via the IO_TRNG_USE generic)
23	SYSINFO_FEATURES_IO_CFU0	Set when the custom functions unit 0 is implemented (via the IO_CFU0_USE generic)
22	SYSINFO_FEATURES_IO_WDT	Set when the WDT is implemented (via the IO_WDT_USE generic)
21	SYSINFO_FEATURES_IO_PWM	Set when the PWM is implemented (via the IO_PWM_USE generic)
20	SYSINFO_FEATURES_IO_TWI	Set when the TWI is implemented (via the IO_TWI_USE generic)
19	SYSINFO_FEATURES_IO_SPI	Set when the SPI is implemented (via the IO_SPI_USE generic)
18	SYSINFO_FEATURES_IO_UART	Set when the UART is implemented (via the IO_UART_USE generic)
17	SYSINFO_FEATURES_IO_MTIME	Set when the MTIME is implemented (via the IO_MTIME_USE generic)
16	SYSINFO_FEATURES_IO_GPIO	Set when the GPIO is implemented (via the IO_GPIO_USE generic)
4	SYSINFO_FEATURES_MEM_INT_DMEM	Set when the processor-internal IMEM is implemented (via the MEM_INT_IMEM_USE generic)
3	SYSINFO_FEATURES_MEM_INT_IMEM_ROM	Set when the processor-internal IMEM is read-only (via the MEM_INT_IMEM_ROM generic)
2	SYSINFO_FEATURES_MEM_INT_IMEM	Set when the processor-internal DMEM implemented (via the MEM_INT_DMEM_USE generic)
1	SYSINFO_FEATURES_MEM_EXT	Set when the external Wishbone bus interface is implemented (via the MEM_EXT_USE generic)
0	SYSINFO_FEATURES_BOOTLOADER	Set when the processor-internal bootloader is implemented (via the BOOTLOADER_USE generic)

4. Software Architecture

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

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

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

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

4.1. Toolchain

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

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

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

The default toolchain used by the project's makefiles is: **`riscv32-unknown-elf`**

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

4.2. Core Software Libraries

The NEORV32 project provides a set of C libraries that allow an easy usage of all of the core's peripheral and CPU features. All you need to do is to include the main NEORV32 library file in your application's source file(s):

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

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

C source file	C header file	Function
-	`neorv32.h`	Main NEORV32 definitions and library file.
`neorv32_cfu.c`	`neorv32_cfu.h`	HW driver (dummy)[10] functions for the custom functions units
`neorv32_cpu.c`	`neorv32_cpu.h`	HW driver functions for the NEORV32 CPU.
`neorv32_gpio.c`	`neorv32_gpio.h`	HW driver functions for the GPIO.
`neorv32_mtime.c`	`neorv32_mtime.h`	HW driver functions for the MTIME.
`neorv32_pwm.c`	`neorv32_pwm.h`	HW driver functions for the PWM.
`neorv32_rte.c`	`neorv32_rte.h`	NEORV32 runtime environment helper functions.
`neorv32_spi.c`	`neorv32_spi.h`	HW driver functions for the SPI.
`neorv32_trng.c`	`neorv32_trng.h`	HW driver functions for the TRNG.
`neorv32_twi.c`	`neorv32_twi.h`	HW driver functions for the TWI.
`neorv32_uart.c`	`neorv32_uart.h`	HW driver functions for the UART.
`neorv32_wdt.c`	`neorv32_wdt.h`	HW driver functions for the WDT.

Documentation

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

```
neorv32/docs$ doxygen doxygen_makefile_sw
```

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

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

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

4.3. Application Makefile

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

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

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

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

4.3.1. Targets

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

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

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

4.3.2. Configuration

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

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

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

# Optimization
EFFORT ?= -Os

# Compiler toolchain
RISCV_TOOLCHAIN ?= riscv32-unknown-elf

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

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

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

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

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

4.3.3. Default Compilation Flags

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

`-Wall`	Enable all compiler warnings.
`-ffunction-sections` `-fdata-sections`	Put functions and data segment in independent sections. This allows a code optimization as dead code and unused data can be easily removed.
`-nostartfiles`	Do not use the default start code. The makefiles use the NEORV32-specific start-up code instead (`sw/common/crt0.S`).
`-Wl,--gc-sections`	Make the linker perform dead code elimination.
`-lm`	Include/link with `math.h`
`-lc`	Search for the standard C library when linking
`-lgcc`	Make sure we have no unresolved references to internal GCC library subroutines.
`-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.

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

4.4. Executable Image Format

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

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

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

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

Executable Image Generator

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

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

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

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

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

4.5. Bootloader

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

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

The bootloader requires the UART for user interaction, executable upload and SPI flash programming (`IO_UART_USE` generic is `true`).

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

To interact with the bootloader, attach the UART signals (`uart_txd_o` and `uart_rxd_o`) of the processor's top entity via a COM port (-adapter) to a computer, configure your terminal program using the following settings and perform a reset of the processor.

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

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

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

```
<< NEORV32 Bootloader >>

BLDV: Jul  2 2020
HWV:  0.1.0.1
CLK:  0x05F5E100 Hz
USER: 0x00000000
MISA: 0x42801104
PROC: 0x01FF0015
IMEM: 0x00008000 bytes @ 0x00000000
DMEM: 0x00002000 bytes @ 0x80000000

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

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

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

`BLDV`	Bootloader version (built time).
`HWV`	Processor hardware version (from the `mimpid` CSR).
`USER`	Custom user code (from the `USER_CODE` generic).
`CLK`	Processor clock speed in Hz (via the `mclock` CSR from the `CLOCK_FREQUENCY` generic).
`MISA`	CPU extensions (from the `misa` CSR).
`PROC`	Processor configuration (via the `mfeatures` CSR from the IO and MEM config. generics).
`IMEM`	IMEM memory base address and size in byte.
`DMEM`	DMEM memory base address and size in byte.

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

```
<< NEORV32 Bootloader >>

BLDV: Jul  2 2020
HWV:  0.1.0.1
CLK:  0x05F5E100 Hz
USER: 0x00000000
MISA: 0x42801104
PROC: 0x01FF0015
IMEM: 0x00008000 bytes @ 0x00000000
DMEM: 0x00002000 bytes @ 0x80000000

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

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

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

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

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

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

4.5.1. External SPI Flash for Booting

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

SPI Flash Requirements

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

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

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

SPI Flash Configuration

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

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

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

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

4.5.2. Auto Boot Sequence

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

4.5.3. Bootloader Error Codes

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

`ERROR_0` If you try to transfer an invalid executable (via UART or from the external SPI flash), this error message shows up. Also, if no SPI flash was found during a boot attempt, this message will be displayed.

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

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

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

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

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

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

4.5.4. Final Notes

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

4.6. NEORV32 Runtime Environment

The software architecture of the NEORV32 comes with a minimal runtime environment that takes care of clean application start and also of all interrupts and exceptions during execution.

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

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

Using the NEORV32 Runtime Environment (RTE)

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

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

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

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

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

For instance, an illegal instruction exception catched by the RTE might look like this:

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

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

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

The following `id` exception IDs are available:

ID name [C]	Description / exception or interrupt causing event
`RTE_TRAP_I_MISALIGNED`	Instruction address misaligned
`RTE_TRAP_I_ACCESS`	Instruction (bus) access fault
`RTE_TRAP_I_ILLEGAL`	Illegal instruction
`RTE_TRAP_BREAKPOINT`	Breakpoint (EBREAK instruction)
`RTE_TRAP_L_MISALIGNED`	Load address misaligned
`RTE_TRAP_L_ACCESS`	Load (bus) access fault
`RTE_TRAP_S_MISALIGNED`	Store address misaligned
`RTE_TRAP_S_ACCESS`	Store (bus) access fault
`RTE_TRAP_MENV_CALL`	Environment call from machine mode (ECALL instruction)
`RTE_TRAP_MTI`	Machine timer interrupt (via MTIME)
`RTE_TRAP_MEI`	Machine external interrupt
`RTE_TRAP_MSI`	Machine software interrupt
`RTE_TRAP_FIRQ_0`	Fast interrupt channel 0 (via WDT)
`RTE_TRAP_FIRQ_1`	Fast interrupt channel 1 (via GPIO)
`RTE_TRAP_FIRQ_2`	Fast interrupt channel 2 (via UART)
`RTE_TRAP_FIRQ_3`	Fast interrupt channel 3 (via SPI or TWI)

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

```
void handler_xyz(void) {

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

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

Example: Installation of the MTIME interrupt handler:

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

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

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

Example: Removing the MTIME interrupt handler:

```
neorv32_rte_exception_uninstall(EXC_MTI);
```

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

5. Let's Get It Started!

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

5.1. Toolchain Setup

At first, we need to get the RISC-V GCC toolchain. There are two possibilities to do this:

- Download and compile the official RISC-V GNU toolchain
- Download and install a prebuilt version of the toolchain

Compilation of the toolchain is done using the guide from the official https://github.com/riscv/riscv-gnu-toolchain GitHub page. I have done that on my computer and you can download my prebuilt version from https://github.com/stnolting/riscv_gcc_prebuilt.

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

Besides of the RISC-V GCC, you will need a native GCC to compile the NEORV32 image generator.

Keep in mind that – for instance – a `rv32imc` toolchain only provides library code compiled with compressed (`c`) and mul/div instructions (`m`)! Hence, this code cannot be executed (without emulation) on an architecture without these extensions!

5.1.1. Making the Toolchain from Scratch

The official RISC-V repository uses submodules. You need the --recursive option to fetch the submodules automatically:

```
$ git clone --recursive https://github.com/riscv/riscv-gnu-toolchain
```

Download and install the prerequisite standard packages:

```
$ sudo apt-get install autoconf automake autotools-dev curl python3 libmpc-dev libmpfr-dev libgmp-dev gawk build-essential bison flex texinfo gperf libtool patchutils bc zlib1g-dev libexpat-dev
```

To build the Linux cross-compiler, pick an install path. If you choose, say, `/opt/riscv`, then add `/opt/riscv/bin` to your `PATH` now.

```
$ export PATH:$PATH:/opt/riscv/bin
```

Then, simply run the following commands in the RISC-V GNU toolchain source folder (for `rv32gc`):

```
riscv-gnu-toolchain$ ./configure --prefix=/opt/riscv --with-arch=rv32gc –with-abi=ilp32
riscv-gnu-toolchain$ make
```

After a while (hours!) you will get `riscv32-unknown-elf-gcc` and all of its friends in your `/opt/riscv/bin` folder.

5.1.2. Downloading and Installing the Prebuilt Toolchain

Alternatively, you can download a prebuilt version of the toolchain. I have compiled the toolchain on a 64-bit x86 Ubuntu (Ubuntu on Windows, actually). **Make sure `git lfs` is installed.**

```
$ git clone https://github.com/stnolting/riscv_gcc_prebuilt.git
```

Alternatively, you can directly download the according toolchain archive as single zip-file or as **packed release** zip-file from https://github.com/stnolting/riscv_gcc_prebuilt.

Unpack the archive and copy the content to a location in your file system (e.g. `/opt/riscv`).

Of course you can also use any other prebuilt version of the toolchain. Make sure it is a `riscv32-unknown-elf` or `riscv64-unknown-elf` (that can also emit 32-bit code) toolchain, supports the `rv32i/e` architecture and uses the `ilp32` or `ilp32e` ABI.

5.1.3. Installation

Now you have the binaries. The last step is to add them to your `PATH` environment variable (if you have not already done so). Make sure to add the binaries folder (`bin`) of your toolchain.

```
$ export PATH:$PATH:/opt/riscv/bin
```

You should add this command to your `.bashrc` (if you are using bash) to automatically add the RISC-V toolchain at every console start.

5.1.4. Testing the Installation

To make sure everything works fine, navigate to an example project in the NEORV32 example folder and execute the following command:

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

This will test all the tools required for the NEORV32. Everything is working fine if `Toolchain check OK` appears at the end.

5.2. General Hardware Setup

The following steps are required to generate a bitstream for your FPGA board. If you want to run the **NEORV32 processor** in simulation only, the following steps might also apply.

In this tutorial we will use a test implementation of the **processor** – using most of the processor's optional modules but just propagating the minimal signals to the outer world. Hence, this guide is intended as evaluation or "hello world" project to check out the NEORV32. A little note: The order of the following steps might be a little different for your specific EDA tool.

1. Create a new project with your FPGA EDA tool of choice.

2. Add all VHDL files from the project's `rtl/core` folder to your project. Make sure to *reference* the files only – do not copy them.

3. Make sure to add all the rtl files to a new **library** called `neorv32`. If your FPGA tools does not provide a field to enter the library name, check out the "properties" menu of the rtl files.

4. The `rtl/core/neorv32_top.vhd` VHDL file is the top entity of the NEORV32 processor. If you already have a design, instantiate this unit into your design and proceed.

5. If you do not have a design yet and just want to check out the NEORV32 – no problem! In this guide we will use a simplified top entity, that encapsulated the actual processor top entity. Add the `rtl/core/top_templates/neorv32_test_setup.vhd` VHDL file to your project too, and select it as top entity.

6. This test setup provides a minimal test hardware setup:

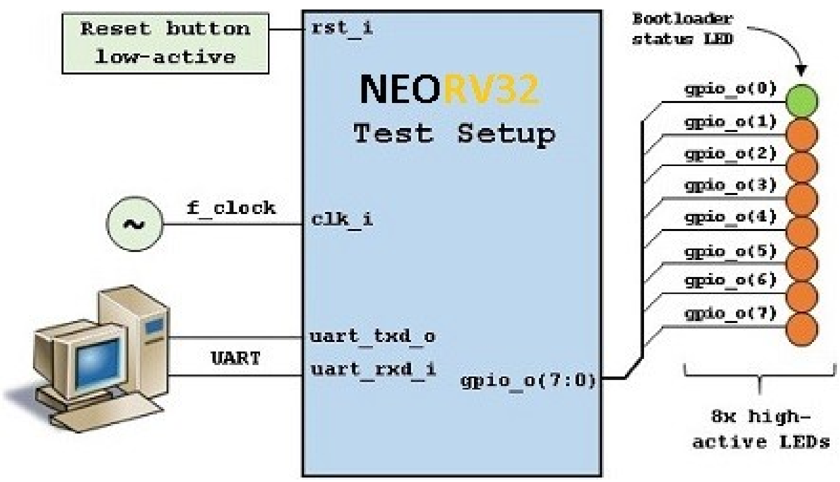

Figure 5: Hardware configuration of the NEORV32 test setup

7. This test setup only implements some very basic processor and CPU features. Also, only the minimum number of signals is propagated to the outer world.

8. The configuration of the NEORV32 processor is done using the generics of the instantiated processor top entity. Let's keep things simple at first and use the default configuration (see below).

9. There is one generic that has to be set according to your FPGA / board: The clock frequency of the top's clock input signal (`clk_i`). Use the `CLOCK_FREQUENCY` generic to specify your clock source's frequency in Hertz (Hz) (→ the default value you need to adapt is marked in **orange**).

```
neorv32_top_inst: neorv32_top
generic map (
  -- General --
  CLOCK_FREQUENCY              => 100000000, -- in Hz
  BOOTLOADER_USE               => true,
  USER_CODE                    => x"00000000",
  -- RISC-V CPU Extensions --
  CPU_EXTENSION_RISCV_C        => true,
  CPU_EXTENSION_RISCV_E        => false,
  CPU_EXTENSION_RISCV_M        => false,
  CPU_EXTENSION_RISCV_Zicsr    => true,
  CPU_EXTENSION_RISCV_Zifencei => true,
  -- Extension Options --
  FAST_MUL_EN                  => false,
  FAST_SHIFT_EN                => false,
  -- Physical Memory Protection (PMP) --
  PMP_USE                      => false,
  PMP_NUM_REGIONS              => 4,
  PMP_GRANULARITY              => 15,
  -- Memory configuration: Instruction memory --
  MEM_INT_IMEM_USE             => true,
  MEM_INT_IMEM_SIZE            => 16*1024, -- in BYTES
  MEM_INT_IMEM_ROM             => false,
  -- Memory configuration: Data memory --
  MEM_INT_DMEM_USE             => true,
  MEM_INT_DMEM_SIZE            => 8*1024, -- in BYTES
  -- Memory configuration: External memory interface --
  MEM_EXT_USE                  => false,
  MEM_EXT_REG_STAGES           => 2,
  -- Processor peripherals --
  IO_GPIO_USE                  => true,
  IO_MTIME_USE                 => true,
  IO_UART_USE                  => true,
  IO_SPI_USE                   => false,
  IO_TWI_USE                   => false,
  IO_PWM_USE                   => false,
  IO_WDT_USE                   => true,
  IO_TRNG_USE                  => false,
  IO_CFU0_USE                  => false,
  IO_CFU1_USE                  => false
)
```

10. If you feel like it – or if your FPGA does not provide enough resources – you can modify the memory sizes (`MEM_INT_IMEM_SIZE` and `MEM_INT_DMEM_SIZE`, marked in **red** and **blue**) or exclude certain peripheral modules from implementation. But as mentioned above, let's keep things simple and use the standard configuration for now.

11. For this setup, we will only use the processor-internal data and instruction memories for the test setup. So make sure, the instruction and data space sizes are always equal to the sizes of the internal memories (i.e. `MEM_INT_IMEM_SIZE` == `MEM_ISPACESIZE` and `MEM_INT_DMEM_SIZE` == `MEM_DSPACESIZE`).

Keep the internal instruction and data memory sizes in mind – these values are required for setting up the software framework in the next chapter.

12. Depending on your FPGA tool of choice, it is time to assign the signals of the test setup top entity to the according pins of your FPGA board. All the signals can be found in the entity declaration:

```
entity neorv32_test_setup is
  port (
    -- Global control --
    clk_i      : in  std_ulogic := '0'; -- global clock, rising edge
    rstn_i     : in  std_ulogic := '0'; -- global reset, low-active, async
    -- GPIO --
    gpio_o     : out std_ulogic_vector(7 downto 0); -- parallel output
    -- UART --
    uart_txd_o : out std_ulogic; -- UART send data
    uart_rxd_i : in  std_ulogic := '0' -- UART receive data
  );
end neorv32_test_setup;
```

13. Attach the clock input `clk_i` to your clock source and connect the reset line `rstn_i` to a button of your FPGA board. Check whether it is low-active or high-active – the reset signal of the processor is **low-active**, so maybe you need to invert the input signal.

14. If possible, connected at least bit #0 of the GPIO output port `gpio_o` to a high-active LED (invert the signal when your LEDs are low-active).

15. Finally, connect the UART signals `uart_txd_o` and `uart_rxd_i` to your serial host interface (dedicated pins, USB-to-serial converter, etc.).

16. Perform the project HDL compilation (synthesis, mapping, bitstream generation).

17. Download the generated bitstream into your FPGA ("program" it) and press the reset button (just to make sure everything is sync).

18. Done! If you have assigned the bootloader status LED (bit #0 of the GPIO output port), it should be flashing now and you should receive the bootloader start prompt via the UART.

5.3. General Software Framework Configuration

While your synthesis tool is crunching the NEORV32 HDL files, it is time to configure the project's software framework for your processor hardware setup.

1. You need to tell the linker the size of the processor's instruction and data memories. This has to be identical to the hardware memory configuration (see 5.2. General Hardware Setup).

2. Open the NEORV32 linker script `sw/common/neorv32.ld` with a text editor. Right at the beginning of the linker script you will find the memory configuration:

The `rom` region provides conditional assignments (via symbol `make_bootloader`) for the origin and the length depending on whether the executable is built as normal application (for the IMEM) or as bootloader code (for the BOOTROM). **To modify the IMEM configuration of the `rom` region, make sure to only edit the second values for *ORIGIN* and *LENGTH* (marked in red).**

```
MEMORY
{
  rom  (rx)  :  ORIGIN  =  DEFINED(make_bootloader)  ?  0xFFFF0000  :  0x00000000,  LENGTH  =
DEFINED(make_bootloader) ? 4*1024 : 16*1024

  ram (rwx) : ORIGIN = 0x80000000, LENGTH =  8*1024
}
```

3. There are four parameters that are relevant here (only the right value for the rom section): The origin and the length of the instruction memory (named `rom`) and the origin and the length of the data memory (named `ram`). These four parameters have to be always sync to your hardware memory configuration:

The `rom ORIGIN` parameter has to be equal to the configuration of the NEORV32 `ispace_base_c` (default: `0x00000000`) VHDL package configuration constant. The `ram ORIGIN` parameter has to be equal to the configuration of the NEORV32 `dspace_base_c` (default: `0x80000000`) VHDL package configuration constant.

The `rom LENGTH` and the `ram LENGTH` parameters have to match the available memory sizes. For instance, if the system does not have any external memories connected, the `rom LENGTH` parameter has to fit the size of the processor-internal IMEM (defined via top's `MEM_INT_IMEM_SIZE` generic) and the `ram LENGTH` parameter has to fit the size of the processor-internal DMEM (defined via top's `MEM_INT_DMEM_SIZE` generic)

5.4. Building the Software Documentation

If you wish, you can generate the documentation of the NEORV32 software framework. This doxygen-based documentation illustrates the core libraries as well as all the example programs. A deployed version of the documentation can be found online at **GitHub pages**.

1. Make sure `doxygen` is installed. Navigate to the `docs` folder and generate the documentation files using the provided doxygen makefile:

```
neorv32/docs$ doxygen doxygen_makefile_sw
```

2. Doxygen will generate a HTML-based documentary. The output files are placed in (a new folder) `docs/doxygen_build/html`. Move to this folder and open `index.html` with your browser. Click on the “files” tab to see an overview of all documented files.

5.5. Application Program Compilation

1. Open a terminal console and navigate to one of the project’s example programs. For example the simple `sw/example_blink_led` program. This program uses the NEORV32 `GPIO` unit to display an 8-bit counter on the lowest eight bit of the `gpio_o` port.

2. To compile the project and generate an executable simply execute:

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

3. This will compile and link the application sources together with all the included libraries. At the end, your application is put into an ELF file (main.elf). The image generator takes this file and creates a final executable. The makefile will show the resulting memory utilization and the executable size:

```
neorv32/sw/example/blink_led$ make exe
Memory utilization:
   text    data     bss     dec     hex filename
    852       0       0     852     354 main.elf
Executable (neorv32_exe.bin) size in bytes:
864
```

4. That’s it. The `exe` target has created the actual executable (`neorv32_exe.bin`) in the current folder, which is ready to be uploaded to the processor via the bootloader and a UART interface.

The compilation process will also create a `main.asm` assembly listing file in the project directory. This shows the actual assembly code of the complete application

5.6. Uploading and Starting of a Binary Executable Image via UART

We have just created the executable. Now it is time to upload it to the processor. There are basically two options to do so.

Option 1

The NEORV32 makefiles provide an `upload` target that allows to directly upload an executable from the command line. Reset the processor and execute:

```
sw/example/blink_led$ make COM_PORT=/dev/ttyUSB1 upload
```

Replace `/dev/ttyUSB1` with the actual serial port you are using to communicate with the processor. You might have to use `sudo` if the targeted tty device requires elevated access rights.

Option 2

Alternatively, you can use a standard terminal program to upload an executable. This provides a more "secure" way as you can directly interact with the bootloader console. Additionally, using a terminal program allows to directly communicate with the uploaded program.

1. Connect the UART interface of your FPGA (board) to a COM port of your computer or use an USB-to-serial adapter.

2. Start a terminal program. In this tutorial, I am using TeraTerm for Windows. You can download it from https://ttssh2.osdn.jp/index.html.en

Make sure your terminal program can transfer the executable in raw byte mode without any protocol stuff around it.

3. Open a connection to the corresponding COM port. Configure the terminal according to the following parameters:

   - `19200 Baud`
   - `8 data bits`
   - `1 stop bit`
   - `No parity bits`
   - **`No transmission/flow control protocol! (just raw byte mode)`**
   - `Newline on \r\n = carriage return & newline (if configurable at all)`

4. Also make sure, that single chars are transmitted without any consecutive "new line" or "carriage return" commands (this is highly dependent on your terminal application of choice, TeraTerm only sends the raw chars by default).

5. Press the NEORV32 reset button to restart the bootloader. The status LED starts blinking and the bootloader intro screen appears in your console. Hurry up and press any key (hit space!) to abort the automatic boot sequence and to start the actual bootloader user interface console.

```
<< NEORV32 Bootloader >>

BLDV: Jul  2 2020
HWV:  0.1.0.1
CLK:  0x05F5E100 Hz
USER: 0x00000000
MISA: 0x42801104
PROC: 0x01FF0015
IMEM: 0x00008000 bytes @ 0x00000000
DMEM: 0x00002000 bytes @ 0x80000000

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

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

6. Execute the "Upload" command by typing `u`. Now the bootloader is waiting for a binary executable to be send.

```
CMD:> u
Awaiting neorv32_exe.bin...
```

7. Use the "send file" option of your terminal program to transmit the previously generated binary executable `neorv32_exe.bin`.

8. Again, make sure to transmit the executable in **raw binary mode** (no transfer protocol, no additional header stuff). When using TeraTerm, select the "binary" option in the send file dialog:

Figure 6: Transfer executable in binary mode (German version of TeraTerm)

9. If everything went fine, **`OK`** will appear in your terminal:

```
CMD:> u
Awaiting neorv32_exe.bin... OK
```

10. The executable now resides in the instruction memory of the processor. To execute the program right now execute the "Execute" command by typing `e`.

```
CMD:> u
Awaiting neorv32_exe.bin... OK
CMD:> e
Booting...

Blinking LED demo program
```

11. Now you should see the LEDs counting.

5.7. Setup of a New Application Program Project

Done with all the introduction tutorials and those example programs? Then it is time to start your own application project!

1. The easiest way of creating a new project is to make a copy of an existing project (like the `blink_led` project) inside the `example` folder. By this, all file dependencies are kept and you can start coding and compiling.

2. If you want to have he project folder somewhere else, you need to adapt the project's makefile. In the makefile you will find a variable that keeps the relative or absolute path to the NEORV32 home folder. Just modify this variable according to your project's location:

```
# Relative or absolute path to the NEORV32 home folder (use default if not set by user)
NEORV32_HOME ?= ../../..
```

3. If your project contains additional source files outside of the project folder, you can add them to the `APP_SRC` variable:

```
# User's application sources (add additional files here)
APP_SRC = $(wildcard *.c) ../somewhere/some_file.c
```

4. You also need to add the folder containing the include files of your new project to the `APP_INC` variable (do not forget the `-I` prefix):

```
# User's application include folders (don't forget the '-I' before each entry)
APP_INC = -I . -I ../somewhere/include_stuff_folder
```

5. If you feel like it, you can change the default optimization level:

```
# Compiler effort
EFFORT = -Os
```

5.8. Enabling RISC-V CPU Extensions

Whenever you enable a RISC-V compliant CPU extensions via the `CPU_EXTENSION_RISCV_*` generics, you need to adapt the toolchain configuration, so the compiler actually can make use of the extension(s).

To do so, open the makefile of your project (e.g., `sw/example/blink_led/makefile`) and scroll to the "USER CONFIGURATION" section right at the beginning of the file. You need to modify the `MARCH` and `MABI` variables according to your CPU hardware configuration.

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

Alternatively, the `MARCH` and `MABI` configurations can be overridden when invoking the makefile:

```
$ make MARCH=-march=rv32imc clean_all all
```

The following table shows the different combinations of CPU extensions and the according configuration for the `MARCH` and `MABI` variables. Of course you can also just use a subset of the available extensions (e.g. `march=rv32im` for a `rv32imc` CPU). All remaining CPU extension options do not require a modification of MARCH or MABI.

Enabled CPU Extension(s)	Toolchain MARCH	Toolchain MABI
`none`	`MARCH=-march=rv32i`	`MABI = -mabi=ilp32`
`CPU_EXTENSION_RISCV_C`	`MARCH=-march=rv32ic`	
`CPU_EXTENSION_RISCV_M`	`MARCH=-march=rv32im`	
`CPU_EXTENSION_RISCV_C` `CPU_EXTENSION_RISCV_M`	`MARCH=-march=rv32imc`	
`CPU_EXTENSION_RISCV_E`	`MARCH=-march=rv32e`	`MABI = -mabi=ilp32e`
`CPU_EXTENSION_RISCV_E` `CPU_EXTENSION_RISCV_C`	`MARCH=-march=rv32ec`	
`CPU_EXTENSION_RISCV_E` `CPU_EXTENSION_RISCV_M`	`MARCH=-march=rv32em`	

`CPU_EXTENSION_RISCV_E`
`CPU_EXTENSION_RISCV_C`
`CPU_EXTENSION_RISCV_M`
	`MARCH=-march=rv32emc`	

The **RISC-V ISA string** (for `MARCH`) follows a certain canonical structure:
`RV[32/64/128][I/E][M][A][F][D][G][Q][L][C][B][J][T][P][V][N][…]`
Example: `rv32imc` is valid while `rv32icm` is not.

5.9. Building a Non-Volatile Application (Program Fixed in IMEM)

The purpose of the bootloader is to allow an easy and fast update of the application being currently executed. But maybe at some time your project has become mature and you want to actually embed your processor including the application. Of course you can store the executable to the SPI flash and let the bootloader fetch and execute it a system start. But if you don't have an SPI flash available or you want a really fast start of your applications, you can directly implement your executable within the processor internal instruction memory. When using this approach, the bootloader is no longer required. To have your application to permanently reside in the internal instruction memory, follow the upcoming steps.

This works only for the internal instruction memory. Also make sure that the memory components the IMEM is mapped to support an initialization via the bitstream.

1. At first, compile your application code by running the `make install` command (the memory utilization is not shown again when your code has already been compiled):

```
neorv32/sw/example/blink_led$ make compile
Memory utilization:
   text    data     bss     dec     hex filename
    852       0       0     852     354 main.elf
Executable (neorv32_exe.bin) size in bytes:
864
Installing application image to ../../../rtl/core/neorv32_application_image.vhd
```

2. The `install` target has created an executable, too, but this time in the form of a VHDL memory initialization file. At synthesis, this initialization will become part of the final FPGA bitstream, which in terms initializes the IMEM's blockram.

3. You need the processor to directly execute the code in the IMEM. Deactivate the implementation of the bootloader via the top entity's generic:

```
BOOTLOADER_USE => false, -- implement processor-internal bootloader?
```

4. When the bootloader is deactivated, the according ROM is removed and the CPU will start booting at the base address of the instruction memory space. Thus, the CPU directly executed your application code after reset.

5. The IMEM could be still modified, since it is implemented as RAM. This might corrupt your executable. To prevent this and to implement the IMEM as true ROM (and eventually saving some more hardware resources), active the IMEM as ROM feature using the processor's top entity generic:

```
MEM_INT_IMEM_ROM => true, -- implement processor-internal instruction memory as ROM
```

6. Perform a synthesis and upload your new bitstream. Your application code resides now unchangeable in the processor's IMEM and is directly executed after reset.

5.10. Re-Building the Internal Bootloader

If you have modified any of the configuration parameters of the default bootloader (in `sw/bootloader/bootloader.c`), if you have added additional features or if you have implemented your own bootloader, you need to re-compile and re-install the bootloader.

1. The NEORV32 default bootloader uses 4kB of boot ROM space. This is also the default boot ROM size. If your new/modified bootloader exceeds this size, you need to modify the boot ROM configurations.

2. Open the processor's main package file `rtl/core/neorv32_package.vhd` and edit the `boot_size_c` constant according to your requirements. The boot ROM size **must not exceed 32kB** and should be a power of two (for optimal hardware mapping).

```
-- Bootloader ROM --
constant boot_size_c : natural := 4*1024; -- bytes
```

3. Now open the NEORV32 linker script `sw/common/neorv32.ld` and adapt the `LENGTH` parameter of the `rom` according to your new memory size. `boot_size_c` and `LENGTH` have to be always identical. Do not modify the `ORIGIN` of the boot memory.

The `rom` region provides conditional assignments (via symbol `make_bootloader`) for the origin and the length depending on whether the executable is built as normal application (for the IMEM) or as bootloader code (for the BOOTROM). **To modify the BOOTLOADER memory size, make sure to edit the first value for the origin (marked in red).**

```
MEMORY
{
  rom  (rx)  :  ORIGIN  =  DEFINED(make_bootloader)  ?  0xFFFF0000  :  0x00000000,  LENGTH  =
DEFINED(make_bootloader) ? 4*1024 : 16*1024
   ram (rwx) : ORIGIN = 0x80000000, LENGTH =  8*1024
}
```

4. Compile and install the bootloader using the explicit `bootloader` makefile target. This target uses the bootloader-specific start-up code and linker script instead of the regular application files.

```
neorv32/sw/bootloader$ make bootloader
```

5. Now perform a new synthesis / HDL compilation to update the bitstream with the new bootloader image.

The bootloader is intended to work regardless of the actual NEORV32 hardware configuration – especially when it comes to CPU extensions. Hence, the bootloader should be build using the minimal `rv32i` ISA only (`rv32e` would be even better).

See chapter 4.5. Bootloader for more information regarding the bootloader.

5.11. Programming the Bootloader SPI Flash

1. At first, reset the NEORV32 processor and wait until the bootloader start screen appears in your terminal program.

2. Abort the auto boot sequence and start the user console by pressing any key.

3. Press `u` to upload the program image, that you want to store to the external flash:

```
CMD:> u
Awaiting neorv32_exe.bin...
```

4. Send the binary in raw binary via your terminal program. When the uploaded is completed and `OK` appears, press `p` to trigger the programming of the flash (do not execute the image via the `e` command as this might corrupt the image):

```
CMD:> u
Awaiting neorv32_exe.bin... OK
CMD:> p
Write 0x000013FC bytes to SPI flash @ 0x00800000? (y/n)
```

5. The bootloader shows the size of the executable and the base address inside the SPI flash where the executable is going to be stored. A prompt appears: Type `y` to start the programming or type `n` to abort.

```
CMD:> u
Awaiting neorv32_exe.bin... OK
CMD:> p
Write 0x000013FC bytes to SPI flash @ 0x00800000? (y/n) y
Flashing... OK
CMD:>
```

6. If `OK` appears in the terminal line, the programming process was successful. Now you can use the auto boot sequence to automatically boot your application from the flash at system start-up without any user interaction.

See chapter 4.5. Bootloader for more information regarding the bootloader.

5.12. Simulating the Processor

The NEORV32 project features a simple testbench (`sim/neorv32_tb.vhd`) that can be used to simulate and test the processor and the CPU itself. This testbench features a 100MHz clock and enables all optional peripheral devices and all optional CPU extensions (but not the embedded CPU mode).

Please note that the true-random number generator (TRNG) **CANNOT** be simulated due to its combinatorial oscillator architecture.

The simulated NEORV32 does not use the bootloader and directly boots the current application image (from the `rtl/core/neorv32_application_image.vhd` image file). Make sure to use the `all` target of the makefile to **install** your application as VHDL image after compilation:

```
sw/example/blink_led$ make clean_all all
```

Simulation Console Output

Data written to the NEORV32 UART transmitter is send to a virtual UART receiver implemented within the testbench. This receiver uses the default (bootloader) UART configuration. Received chars are send to the simulator console and are also stored to a file (`neorv32.testbench_uart.out`) in the simulator home folder.

Faster Simulation Console Output

When printing data via the UART the communication will always be based on the configured BAUD rate. For a simulation this will take a very long time. To have a faster output you can enable the UART's simulation mode (see chapter 3.4.8. Universal Asynchronous Receiver and Transmitter (UART)). ASCII data written to the UART will be immediately printed to the simulator console. Additionally, the ASCII data is logged in a file (`neorv32.uart.sim_mode.text.out`) in the simulator home folder. All written 32-bit data is also dumped as 8-char hexadecimal value into a file `neorv32.uart.sim_mode.data.out` in the simulation home folder.

You can automatically the UART's sim mode when compiling an application. In this case the "real" UART transmitter unit is permanently disabled. To enable the simulation mode just compile and install your application and add `UART_SIM_MODE` to the compiler `USER_FLAGS` variable (do not forget the `-D` suffix flag):

```
sw/example/blink_led$ make USER_FLAGS+=-DUART_SIM_MODE clean_all all
```

The UART simulation output (to file and to screen) outputs "complete lines" at once. A line is completed with a line feed (newline, ASCII `\n` = 10).

Simulation with Xilinx Vivado

The project features a Vivado simulation waveform configuration in `sim/vivado`.

Simulation with GHDL

To simulate the processor using GHDL navigate to the `sim` folder and run the provided shell script. The simulation time can be configured in the script via the `--stop-time=4ms` argument.

```
neorv32/sim$ sh ghdl_sim.sh
```

5.13. Continuous Integration

This project uses continuous integration provided by Travis CI. The project includes a `.travis.yml` file for configuring Travis CI. This configuration file uses the continuous integration scripts located in `.ci`.

What the continuous integration does so far:

- Builds the doxygen-based software documentation and deploys it to GitHub pages
- Downloads, unpacks and installs the pre-built GCC toolchain
- Test the toolchain
- Compile all example projects from the `sw/example` folder
- Compile the bootloader from the `sw/bootloader` folder
- Compile and install the CPU test code from the `sw/bootloader/cpu_test` folder, the generated executable uses the UART's simulation output
- Simulate the processor using its default testbench (`sim/neorv32_tb.vhd`) using GHDL
- The UART simulation text output is searched for a reference string; if the string is found the test was successful

5.14. FreeRTOS Support

A NEORV32-specific port and a simple demo for FreeRTOS (https://github.com/FreeRTOS/FreeRTOS) are available in the `sw/example/demo_freeRTOS` folder.

See the documentation (`sw/example/demo_freeRTOS/README.md`) for more information.

6. Change Log

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