The NEORV32 Processor

by Dipl.-Ing. Stephan Nolting

A customizable, lightweight and open-source
32-bit RISC-V soft-core microcontroller.

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” and “Radiant” are trademarks of Lattice Semiconductor Corporation.
“Windows” is a trademark of Microsoft Corporation.
“Tera Term” copyright by T. Teranishi.

Disclaimer

This file is part of the NEORV32 Processor project by Stephan Nolting.

This is a hobby project released under the BSD 3-Clause license.
No copyright infringement intended.

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

RISC-V

https://github.com/riscv

The **NEORV32 Processor** Project
©2020 Dipl.-Ing. Stephan Nolting, Hannover, Germany
For any kind of feedback, feel free to drop me a line: stnolting@gmail.com

BSD 3-Clause License

Table of Content

Proprietary and Legal Notice **2**
BSD 3-Clause License **3**
1. Overview **6**
1.1. Design Principles 7
1.2. Citing 7
1.3. Processor Key Features 8
1.4. Project Folder Structure 9
1.5. VHDL File Hierarchy 10
1.6. Processor Top Entity – Signals 11
1.7. Processor Top Entity – Configuration Generics 12
1.8. FPGA Implementation Results 15
1.8.1. CPU 15
1.8.2. Peripherals 16
1.9. CPU Performance 17
1.9.1. CoreMark Benchmark 17
1.9.2. Instruction Timing 18
1.9.3. Evaluation 18
2. Central Processing Unit **19**
2.1. Instruction Set 20
2.2. Instruction Timing 21
2.3. Control and Status Registers (CSRs) 22
2.4. Exceptions and Interrupts 29
2.5. Address Space 30
2.5.1. Processor-Internal Instruction Memory (IMEM) 31
2.5.2. Processor-Internal Data Memory (DMEM) 31
2.5.3. Processor-Internal Bootloader ROM (BOOTROM) 31
2.5.4. Processor-External Memory Interface (WISHBONE) 32
2.5.5. Processor-Internal Peripheral/IO Devices 33
3. Peripheral/IO Devices **35**
3.1. General Purpose Input and Output Port (GPIO) 36
3.2. Core-Local Interrupt Controller (CLIC) 37
3.3. Watchdog Timer (WDT) 39
3.4. Machine System Timer (MTIME) 41
3.5. Universal Asynchronous Receiver and Transmitter (UART) 42
3.6. Serial Peripheral Interface Master (SPI) 44
3.7. Two Wire Serial Interface Master (TWI) 46
3.8. Pulse Width Modulation Controller (PWM) 48
3.9. True Random Number Generator (TRNG) 50
4. Software Architecture **52**
4.1. Toolchain 52
4.2. Core Software Libraries 53
4.3. Application Makefile 54
4.3.1. Makefile Targets 54
4.3.2. Makefile Configuration 55
4.4. Executable Image Format 56
4.5. Bootloader 57
4.5.1. Auto Boot Sequence 59
4.5.2. External SPI Flash for Booting 60
4.5.3. Bootloader Error Codes 61
4.6. NEORV32 Runtime Environment 62

5. Let's Get It Started! **65**
5.1. Toolchain Setup 65
5.1.1. Making the Toolchain from Scratch 65
5.1.2. Downloading and Installing the Prebuilt Toolchain 66
5.1.3. Installation 66
5.1.4. Testing the Installation 66
5.2. General Hardware Setup 67
5.3. General Software Framework Configuration 70
5.4. Building the Software Documentation 71
5.5. Application Program Compilation 71
5.6. Uploading and Starting of a Binary Executable Image via UART 72
5.7. Setup of a New Application Program Project 74
5.8. Enabling RISC-V CPU Extensions 75
5.9. Building a Non-Volatile Application (Program Fixed in IMEM) 76
5.10. Re-Building the Internal Bootloader 77
5.11. Programming the Bootloader SPI Flash 78
6. Troubleshooting **79**
7. Change Log **80**

1. Overview

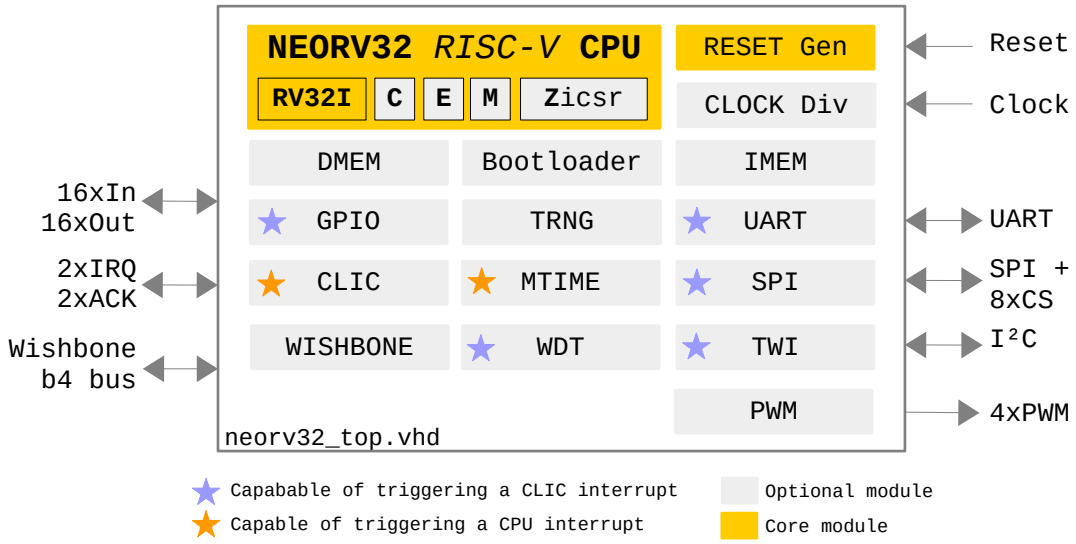

Figure 1: NEORV32 processor block diagram

The **NEORV32**[1] is a customizable mikrocontroller-like processor system based on a RISC-V `rv32i` or `rv32e` CPU with optional `M`, `E`, `C` and `Zicsr` extensions. The CPU was built from scratch and is compliant to the **Unprivileged ISA Specification Version 2.1** and a subset of the **Privileged Architecture Specification Version 1.12**. The NEORV32 is intended as auxiliary processor within a larger SoC designs or as stand-alone custom microcontroller.

The processor provides common peripherals and interfaces like input and output ports, serial interfaces for UART, I²C and SPI, interrupt controller, timers and embedded memories. External memories peripherals and custom IP can be attached via a Wishbone-based external memory interface. All optional features beyond the base CPU can be enabled configured via VHDL generics.

This project comes with a complete software ecosystem that features core libraries for high-level usage of the provided functions and peripherals, application makefiles and example programs. All software source files provide a doxygen-based documentary.

The project 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.

Quickstart: If you don't want to read all the documentary, jump directly to the 5. Let's Get It Started! chapter.

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

1.1. Design Principles

I've worked on and with several soft-core architecture. And 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 parameters. 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 require latches) and clock gating to halt certain parts of a circuit.

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

Probably we all are fanboys/girls/you-name-it of a specific FPGA architectures, toolchains or manufacturers. 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.**

1.2. Citing

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

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

1.3. Processor Key Features

- 32-bit `rv32i/e` RISC-V-compliant base CPU (→ 19)
  - Optional `C` extension for compressed instructions
  - Optional `E` extensions for small-outlined embedded CPU version
  - Optional `M` extension for multiplication and division instructions
  - Optional `Zicsr` extension for control and status register access and exception/interrupt system
- Toolchain based on free RISC-V GCC port, prebuilt toolchains available (→ 52)
- Application compilation based on GNU makefiles (→ 54)
- Doxygen-based documentation of the software framework (→ 71)
- Completely described in behavioral, platform-independent VHDL – no primitives, macros, etc.
- Fully synchronous design, no latches, no gated clocks
- Small hardware footprint and high operating frequency (→ 15)
- Highly customizable processor configuration (→ 12)
- Optional processor-internal data and instruction memories (**DMEM/IMEM** → 31)
- Optional internal **bootloader** with UART console and automatic SPI flash boot option (→ 57)
- Optional machine system timer (**MTIME** → 41), RISC-V-compliant
- Optional universal asynchronous receiver and transmitter (**UART** → 42)
- Optional 8/16/24/32-bit serial peripheral interface master (**SPI** → 44) with 8 dedicated CS lines
- Optional two wire serial interface master (**TWI** → 46), compatible to the I²C standard
- Optional general purpose parallel IO port (**GPIO** → 36), 16xOut, 16xIn
- Optional 32-bit external bus interface, Wishbone b4 compliant (**WISHBONE** → 32)
- Optional watchdog timer (**WDT** → 39)
- Optional PWM controller with 4 channels and 8-bit duty cycle resolution (**PWM** → 48)
- Optional GARO-based true random number generator (**TRNG** → 50)
- Optional core-local interrupt controller with 8 channels (**CLIC** → 37)

1.4. Project Folder Structure

```
neorv32                 Project home folder.
├─docs                  Project documentary: RISC-V specifications implemented in this
│ │                     project, Wishbone bus specification, NEORV32 data sheet, doxygen
│ │                     makefiles.
│ ├─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
│ │                     processor. 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 a simple VHDL testbench and additional
│                       simulation files.
└─sw                    The software folder contains the processor's core libraries,
  │                     makefiles, linker scripts, start-up codes and example programs.
  ├─bootloader          Source and compilation script of the NEORV32-internal bootloader.
  ├─common              Application & bootloader linker scripts and startup codes.
  ├─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 folders starting with a `.` 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.

1.5. 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 processor core VHDL files have to be assigned to a new **`library`** called **`neorv32`**.

```
neorv32_top.vhd                    Processor core top entity
├─neorv32_boot_rom.vhd             Bootloader ROM
│ └─neorv32_bootloader_image.vhd   Boot ROM initialization image for the bootloader
├─neorv32_clic.vhd                 Core-local interrupt controller
├─neorv32_cpu.vhd                  NEORV32 CPU top entity
│ ├─neorv32_cpu_alu.vhd            Arithmetic/logic unit
│ ├─neorv32_cpu_bus.vhd            Bus interface unit
│ ├─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_reg_file.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_package.vhd              Processor VHDL package file
├─neorv32_pwm.vhd                  Pulse-width modulation controller
├─neorv32_spi.vhd                  Serial peripheral interface master
├─neorv32_trng.vhd                 True random number generator
├─neorv32_twi.vhd                  Two wire serial interface master
├─neorv32_uart.vhd                 Universal asynchronous receiver/transmitter
├─neorv32_wdt.vhd                  Watchdog timer
└─neorv32_wb_interface.vhd         External Wishbone bus gateway
```

1.6. Processor Top Entity – Signals

The following table shows all interface ports of the processor top entity (`neorv32_top.vhd`). The type of all signals is `std_ulogic` or `std_ulogic_vector`, respectively – except for the TWI signals, which are of type `std_logic`.

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	Slave 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	
General Purpose Inputs & Outputs (GPIO)				
gpio_o	16	Output	General purpose parallel output[2]	GPIO
gpio_i	16	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 Master (SPI)				
spi_sclk_o	1	Output	SPI master clock line	SPI
spi_mosi_o	1	Output	SPI serial data output	
spi_miso_i	1	Input	SPI serial data input	
spi_csn_o	8	Output	SPI dedicated chip select lines 0..7[3] (low-active)	
Two-Wire Interface (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
External Interrupts				
ext_irq_i	2	Input	Interrupt request signals, high-active	CLIC
ext_ack_o	2	Output	Interrupt request acknowledges, single-shot	

Table 1: neorv32_top.vhd – processor's top entity interface ports

2 Bit #0 is used by the bootloader to drive a high-active status LED.
3 Chip select #0 is used by the bootloader to access the external boot SPI flash.

1.7. 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.

General

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

`HART_ID` **std_ulogic_vector(31 downto 0)** `x"00000000"`
Custom hardware thread ID. Can be read by software via the `mhartid` CSR.

`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 `MEM_ISPACE_BASE` to the base address of the boot ROM.

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_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**. The bootloader and also the default application start-up code (`crt0.S`) rely on system information provided by (custom) CSRs.

Memory Configuration: Instruction Memory

MEM_ISPACE_BASE **std_ulogic_vector(31 downto 0)** x"00000000"
Base address of the instruction memory space. This is also the default boot address, if the bootloader is not implemented.

MEM_ISPACE_SIZE **natural** 16*1024
Size of the instruction memory space in bytes. Starts at MEM_ISPACE_BASE.

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) when true. Has no effect when MEM_INT_IMEM_USE is false. Must no be greater than MEM_ISPACE_SIZE.

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.

Memory Configuration: Data Memory

MEM_DSPACE_BASE **std_ulogic_vector(31 downto 0)** x"80000000"
Base address of the data memory space.

MEM_DSPACE_SIZE **natural** 8*1024
Size of the data memory space in bytes. Starts at MEM_DSPACE_BASE.

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) when true. Has no effect when MEM_INT_DMEM_USE is false. Must no be greater than MEM_DSPACE_SIZE.

Memory Configuration: External Memory Interface

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.

MEM_EXT_TIMEOUT **natural** 15
Maximum length of bus access in main clock cycles. If a bus access is not acknowledged within the specified time, the access is aborted and a load/store/instruction access fault is triggered.

Processor Peripherals

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.

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.

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. The UART interrupt can only be used when `CPU_EXTENSION_RISCV_Zicsr` and `IO_CLIC_USE` are enabled.

IO_SPI_USE **boolean** `true`
Implement serial peripheral interface master (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. The SPI interrupt can only be used when `CPU_EXTENSION_RISCV_Zicsr` and `IO_CLIC_USE` are enabled.

IO_TWI_USE **boolean** `true`
Implement two-wire interface master (TWI) when `true`. When disabled, the `twi_sda_io` and `twi_scl_io` signals are unconnected. The TWI interrupt can only be used when `CPU_EXTENSION_RISCV_Zicsr` and `IO_CLIC_USE` are enabled.

IO_PWM_USE **boolean** `true`
Implement pulse-width modulation controller (PWM) when `true`. When disabled, the `pwm_o` signal is always low.

IO_WDT_USE **boolean** `true`
Implement watchdog timer (WDT) when `true`. The WDT interrupt can only be used when `CPU_EXTENSION_RISCV_Zicsr` and `IO_CLIC_USE` are enabled.

IO_CLIC_USE **boolean** `true`
Implement core-local interrupt controller (CLIC) when `true`. When disabled, the CPU's machine external interrupt is not available. The `CPU_EXTENSION_RISCV_Zicsr` has to be enabled to use the CLIC interrupt.

IO_TRNG_USE **boolean** `false`
Implement true-random number generator (TRNG) when `true`.

1.8. FPGA Implementation Results

This chapter shows exemplary implementation results of the NEORV32 processor for an **Intel Cyclone IV EP4CE22F17C6N** FPGA on a *Terasic DE0-Nano*[4] 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). 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.

1.8.1. CPU

```
Hardware Version:    0.0.2.3
```

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_Zicsr = false`
		852	326	2048	0	111 MHz

`rv32i`	`CPU_EXTENSION_RISCV_C = false`
	`CPU_EXTENSION_RISCV_E = false`
	`CPU_EXTENSION_RISCV_M = false`
	`CPU_EXTENSION_RISCV_Zicsr = true`
		1488	694	2048	0	107 MHz

`rv32im`	`CPU_EXTENSION_RISCV_C = false`
	`CPU_EXTENSION_RISCV_E = false`
	`CPU_EXTENSION_RISCV_M = true`
	`CPU_EXTENSION_RISCV_Zicsr = true`
		2057	941	2048	0	102 MHz

`rv32imc`	`CPU_EXTENSION_RISCV_C = true`
	`CPU_EXTENSION_RISCV_E = false`
	`CPU_EXTENSION_RISCV_M = true`
	`CPU_EXTENSION_RISCV_Zicsr = true`
		2209	958	2048	0	102 MHz

`rv32e`	`CPU_EXTENSION_RISCV_C = false`
	`CPU_EXTENSION_RISCV_E = true`
	`CPU_EXTENSION_RISCV_M = false`
	`CPU_EXTENSION_RISCV_Zicsr = false`
		848	326	1024	0	111 MHz

`rv32e`	`CPU_EXTENSION_RISCV_C = false`
	`CPU_EXTENSION_RISCV_E = true`
	`CPU_EXTENSION_RISCV_M = false`
	`CPU_EXTENSION_RISCV_Zicsr = true`
		1316	594	1024	0	106 MHz

`rv32em`	`CPU_EXTENSION_RISCV_C = false`
	`CPU_EXTENSION_RISCV_E = true`
	`CPU_EXTENSION_RISCV_M = true`
	`CPU_EXTENSION_RISCV_Zicsr = true`
		1879	841	1024	0	101 MHz

`rv32emc`	`CPU_EXTENSION_RISCV_C = true`
	`CPU_EXTENSION_RISCV_E = true`
	`CPU_EXTENSION_RISCV_M = true`
	`CPU_EXTENSION_RISCV_Zicsr = true`
		2065	858	1024	0	100 MHz

Table 2: Hardware utilization for different CPU configurations

4 https://www.terasic.com.tw/cgi-bin/page/archive.pl?No=593

1.8.2. Peripherals

Hardware Version: **0.0.2.3**

Module	Description	LEs	FFs	MEM bits	DSPs
Boot ROM	Bootloader ROM (4kB)	3	1	32 768	0
DMEM	Processor-internal data memory (8kB)	12	2	65 536	0
GPIO	General purpose input/output ports	37	33	0	0
IMEM	Processor-internal instruction memory (16kB)	7	2	131 072	0
MTIME	Machine system timer	369	168	0	0
PWM	Pulse_width modulation controller	77	69	0	0
SPI	Serial peripheral interface	198	125	0	0
TRNG	True random number generator	103	93	0	0
TWI	Two-wire interface	76	44	0	0
UART	Universal asynchronous receiver/transmitter	154	108	0	0
WDT	Watchdog timer	57	45	0	0

Table 3: Hardware utilization by the different peripheral modules

1.9. CPU Performance

Configuration

Hardware Version:	**0.0.2.3**
Hardware:	32kB IMEM, 16kB DMEM, 100MHz clock
CoreMark:	**2000** iteration, MEM_METHOD is **MEM_STACK**
CPU Extensions:	`rv32i` or `rv32im` or `rv32imc`
Compiler:	**RISCV32-GCC 9.2.0**
Used peripherals:	**MTIME** for time measurement, **UART** for printing the results

1.9.1. CoreMark Benchmark

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 iterationsper second:

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

The **relative CoreMark score** is defined as the CoreMark score divided by the clock frequency in Mhz:

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

Results

Architecture	Optimization	Executable Size	CoreMark Score	CoreMarks/Mhz
`rv32i`	`-Os`	17 944	**23.26**	0.232
	`-O2`	20 264	**25.64**	0.256
`rv32im`	`-Os`	16 880	**40.81**	0.408
	`-O2`	19 312	**47.62**	0.476
`rv32imc`	`-Os`	13 000	**32.78**	0.327
	`-O2`	15 004	**37.04**	0.370

Table 4: NEORV32 CoreMark results

1.9.2. 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 (all of CoreMark – not only the timed core) by the number of executed instructions (`instret[h]` CSRs). The executables were generated using optimization `-O2`.

CPU	Toolchain Configuration	Required Clock Cycles	Executed Instructions	Average CPI
`rv32i`	`rv32i`	10 385 023 697	1 949 310 506	**5.3**
`rv32im`	`rv32im`	6 276 943 488	995 011 883	**6.3**
`rv32imc`	`rv32imc`	7 340 734 652	934 952 588	**7.6**

More information regarding the execution time of each implemented instruction can be found in chapter 2.2. Instruction Timing2.2. Instruction Timing.

1.9.3. Evaluation

Based on the provided performance measurement and the hardware utilization for the different CPU configurations, the following configurations are suggested:

Highest performance:	`rv32im`
Lowest memory requirements:	`rv32imc`
Lowest hardware requirements[5]:	`rv32ec`

5 Including on-chip memory hardware requirements.

2. Central Processing Unit

This chapter takes a detailed look at the NEORV32 CPU. The CPU itself consists of the following VHDL files (from the project's `rtl/core` folder):

`neorv32_cpu.vhd`	CPU top entity
`neorv32_cpu_alu.vhd`	Arithmetic/logic unit
`neorv32_cpu_bus.vhd`	Bus interface unit
`neorv32_cpu_control.vhd`	CPU control and CSRs
`neorv32_cpu_cp_muldiv.vhd`	MULDIV co-processor (if CPU-M ext.)
`neorv32_cpu_decompressor.vhd`	Compressed instructions decoder (if CPU-C ext.)
`neorv32_cpu_regfile.vhd`	Data register file
`neorv32_package.vhd`	Processor/CPU package files

Architecture

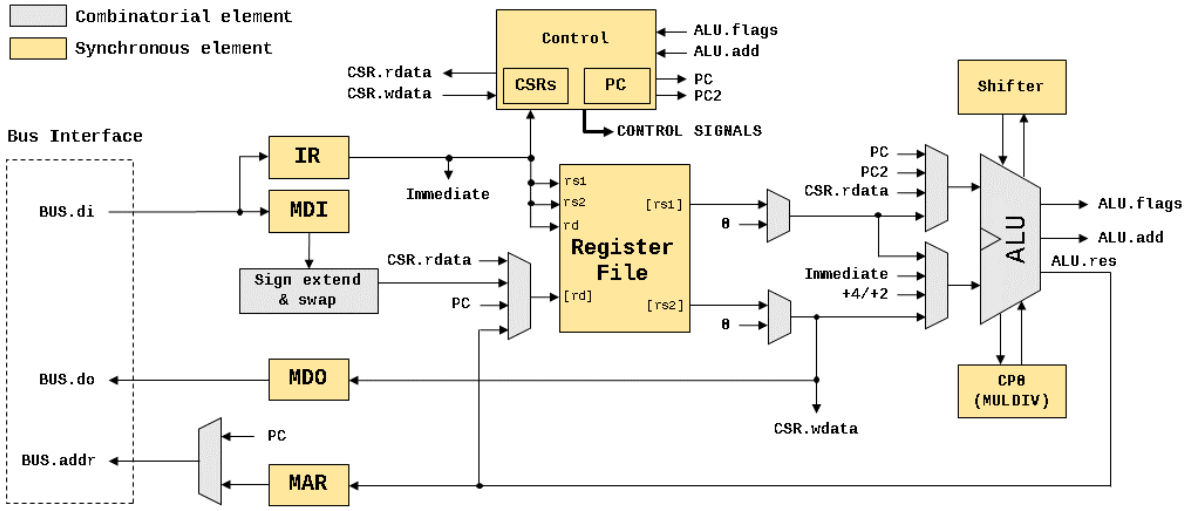

Figure 2: Simplified daata path of the NEORV32 CPU

CPU Key Features

- RISC-V 32-bit base integer ISA `rv32i/e(c)(m)`
- Optional privileged architecture (`Zicsr`) extension supporting RISC-V-compliant control and status registers (CSRs), exceptions and interrupts
- Privilege levels: Machine mode (`M-mode`)
- 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 (causing an exception only when the address' LSB is set).
- Multi-cycle in-order instruction execution
- Mostly compliant to the RISC-V specifications (except for some CSR accesses)
- NEORV32-specific custom CSRs are mapped to the official RISC-V designated address spaces

2.1. Instruction Set

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`

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

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

Compressed 16-bit instructions are available when the `CPU_EXTENSION_C` 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`

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

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

The “wait for interrupt instruction” `WFI` works like a *sleep* command. When executed, the CPU is halted until a valid interrupt request occurs.

2.2. 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 (like bus accesses to processor-internal memories or peripherals).

Class	Instruction(s)	Execution Cycles
ALU	ADDI SLTI SLTIU XORI ORI ANDI ADD SUB SLT SLTU XOR OR AND LUI AUIPC	4
ALU – Shifts	SLLI SRLI SRAI SLL SRL SRA	4 + sha[6] + 1
Branches	BEQ BNE BLT BGE BLTU BGEU	4
Jumps	JAL JALR	4
Memory	LB LH LW LBU LHU SB SH SW	7
Multiplication	MUL MULH MULHSU MULHU	44
Division	DIV DIVU REM REMU	44
CSR Access	CSRRW CSRRS CSRRC CSRRWI CSRRSI CSRRCI	5
System	ECALL EBREAK MRET WFI	4
Compressed	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	Like according uncompressed

Table 5: Required clock cycles per instruction

The instruction fetch for the `rv32i(m)` base CPU (without support for compressed instructions) always requires 3 cycles (without additional bus delays).

If the compressed instructions CPU extensions is enabled (`CPU_EXTENSION_RISCV_C` generic is `true`), the instruction fetch requires an additional cycle for decoding the compressed instruction (instruction fetch then always requires 3+1 cycles). Also, access to unaligned uncompressed instructions (32-bit instruction not on 32-bit boundary) will require an additional instruction fetch to load the missing part of the instruction word.

The average CPI (cycles per instructions) for executing the CoreMark benchmark is evaluated in chapter 1.9.2. Instruction Timing.

6 Shift amount: 0..31

2.3. 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 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 or written. According to the RISC-V specs writing an implemented read-only CSR does not trigger an exception. The CSR marked with a red underlined "`r/-`" have a different read/write access compared to the official RISC-V specifications.

The CSRs with a blue underlined address are counter CSRs. These register are not available for embedded CPU extensions (`CPU_EXTENSION_E` generic is `true`) and will cause an illegal instruction exception. The counters have a constrained size. In contrast to the official RISC-V specifications, the counters are not 64-bit wide but 48-bit wide (to reduce hardware costs). Also, the (user) `cycle[h]` and `instret[h]` counters simply reflect the counters from machine mode (`mcycle[h]`, `instret[h]`). Also, the `time[h]` counters are identical to the `cycle[h]` and `mcycle[h]`, respectively. However, all listed counter CSR addresses can be accesses without an exception fault.

The NEORV32-specific CSRs are mapped to the official "machine custom (read-only) CSRs" CSR address space address space. These custom CSRs provide information regarding the general processor configuration, like clock speed, memory configuration and implemented processor peripherals.

Address	Name [ASM]	Name [C]	R/W	Function
Machine Trap Setup (RISC-V compliant)				
0x300	mstatus	CSR_MSTATUS	r/w	Machine status register
0x301	misa	CSR_MISA	r/-	Machine CPU ISA and extensions
0x304	mie	CSR_MIE	r/w	Machine interrupt enable register
0x305	mtvec	CSR_MTVEC	r/w	Machine trap-handler base address (for ALL traps)
Machine Trap Handling (RISC-V compliant)				
0x340	mscratch	SCR_MACRATCH	r/w	Machine scratch register
0x341	mepc	CSR_MEPC	r/w	Machine exception program counter
0x342	mcause	CSR_MCAUSE	r/-	Machine trap cause
0x343	mtval	CSR_MTVAL	r/-	Machine bad address or instruction
0x344	mip	CSR_MIP	r/w	Machine interrupt pending register
0x34a	mtinst	CSR_MTINST	r/-	Machine trap instruction (transformed)
Counters and Timers (RISC-V compliant)				
0xb00	mcycle	CSR_MCYCLE	r/-	Machine cycle counter low word
0xb02	minstret	CSR_MINSTRET	r/-	Machine instructions-retired counter low word
0xb80	mcycleh*	CSR_MCYCLEH	r/-	Machine cycle counter low word
0xb82	minstreth*	CSR_MINSTRETH	r/-	Machine instructions-retired counter high word
0xc00	cycle	CSR_CYCLE	r/-	Cycle counter low word
0xc01	time	CSR_TIME	r/-	Timer 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/-	Timer high word
0xc82	instreth*	CSR_INSTRETH	r/-	Instructions-retired counter high word
Machine Information Registers (RISC-V compliant)				
0xf13	mimpid	CSR_MIMPID	r/-	Machine implementation ID/version
0xf14	mhartid	CSR_MHARTID	r/-	Machine thread ID
Custom NEORV32-specific CSRs				
0xfc0	-	CSR_MFEATURES	r/-	Implemented processor devices/peripherals
0xfc1	-	CSR_MCLOCK	r/-	Processor primary clock speed
0xfc4	-	CSR_MISPACEBASE	r/-	Base address of instruction address space
0xfc5	-	CSR_MDSPACESIZE	r/-	Size of instruction address space in byte
0xfc6	-	CSR_MDSPACEBASE	r/-	Base address of data address space
0xfc7	-	CSR_MDSPACESIZE	r/-	Size of data address space in bytes

Table 6: NEORV32 Control and Status Registers (CSRs)

*) These high timers/counters are constrained to 16-bit (instead of original 32 bit).

CSR_MCYLE[H] CSR_CYCLE[H] CSR_TIME[H]

The MCYCLE[H], CYCLE[H] and TIME[H] registers are identical for the NEORV32. These counters count the executed clock cycles. These registers are cleared during reset and increment with the primary clock. Note, that these register are only 48-bit wide (in contrast to the original 64-bit). These registers are not available for embedded CPUs (CPU_EXTENSION_E generic is true).

CSR_MINSTRET[H] CSR_INSTRET[H]

The MINSTRET[H] and INSTRET[H] registers are identical for the NEORV32. These counters count the executed instructions. These registers are cleared during reset and increment when the CPU control state machine is in EXECUTE state. Note, that these register are only 48-bit wide (in contrast to the original 64-bit). These registers are not available for embedded CPUs (CPU_EXTENSION_E generic is true).

CSR_MSTATUS

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

Bit#	Name	R/W	Function
12:11	MPP	r/-	CPU operation mode, always “11” (machine mode / M-mode)
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.

CSR_MISA

The MISA register is compliant to the RISC-V misa CSR, but with the exception that it is read-only. Hence, all implemented CPU extension are always enabled and cannot be de-/reactivated during runtime. The following bits are implemented (all remaining bits are always zero and are read-only):

Bit#	R/W	Function
31:30	r/-	32-bit indicator (“01”)
25	r/-	Z CPU extension, set when CPU_EXTENSION_RISCV_Zicsr enabled
23	r/-	X CPU extension, always set to indicate non-standard extensions
12	r/-	M CPU extension, set when CPU_EXTENSION_RISCV_M enabled
8	r/-	I CPU extension, always set, cleared when CPU_EXTENSION_RISCV_E enabled
4	r/-	E CPU extension, set when CPU_EXTENSION_RISCV_E enabled
2	r/-	C CPU extension, set when CPU_EXTENSION_RISCV_C enabled

CSR_MIE

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

Bit#	Name	R/W	Function
11	MEIE	r/w	Machine external interrupt enable (from CLIC)
7	MTIE	r/w	Machine timer interrupt enable (from MTIME)
3	MSIE	r/w	Machine software interrupt enable (via MIP CSR)

CSR_MTVEC

The MTVEC register is compliant to the RISC-V mtvec CSR. This register stores the base address for the machine trap handler. The CPU jumps to this address, regardless of the trap source.

CSR_MEPC

The MEPC register is compliant to the RISC-V mepc CSR. For exceptions (like an illegal instruction), this register provides the address of the exception-causing instruction. On return (via the MRET instruction), the mepc CSR has to increased by 4 to get to the next instruction. For exceptions (like a machine timer interrupt), this register provides the address of the next not-yet-executed instruction. On return(via the MRET instruction), the mepc CSR must not be modified.

CSR_MCAUSE

The MCAUSE register is compliant to the RISC-V mcause CSR, but with the exception that it is read-only. It shows the cause of the current exception.

CSR_MTVAL

The MTVAL register is compliant to the RISC-V mtval CSR, but with the exception that it is read-only. It shows the bad address or the illegal instruction, that caused the exception.

CSR_MSCRATCH

The MSCRATCH register is compliant to the RISC-V mscratch CSR.

CSR_MTINST

The MSCRATCH register is *partly* compliant to the RISC-V mtinst CSR. Bit #1 is used to indicate to indicate an expanded compressed instruction (when low). The rest of the register contains the (expanded) instruction, that caused the exception (no further instruction transformation is conducted yet). The register is read-only.

CSR_MIP

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

Bit#	Name	R/W	Function
11	MEIP	r/-	Machine external interrupt pending (from CLIC)
7	MTIP	r/-	Machine timer interrupt pending (from MTIME)
3	MSIP	r/w	Machine software interrupt pending (via MIP CSR)

CSR_MIMPIP

The MIMPID register is compliant to the RISC-V mimpid CSR. It shows the version of the NEORV32.

CSR_MHARTID

The MHARTID register is compliant to the RISC-V mhartid CSR. It is set via the HART_ID generic.

CSR_MFEATURES

The MFEATURES register is a custom CSR and is read-only. The following bits are implemented (all remaining bits are always zero). The [C] names are defined by the NEORV32 core library and can be used as immediates in normal C code.

Bit#	Name [C]	Function
24	CPU_MFEATURES_IO_TRNG	Set when the TRNG is implemented (via the IO_TRNG_USE generic)
23	CPU_MFEATURES_IO_CLIC	Set when the CLIC is implemented (via the IO_CLIC_USE generic)
22	CPU_MFEATURES_IO_WDT	Set when the WDT is implemented (via the IO_WDT_USE generic)
21	CPU_MFEATURES_IO_PWM	Set when the PWM is implemented (via the IO_PWM_USE generic)
20	CPU_MFEATURES_IO_TWI	Set when the TWI is implemented (via the IO_TWI_USE generic)
19	CPU_MFEATURES_IO_SPI	Set when the SPI is implemented (via the IO_SPI_USE generic)
18	CPU_MFEATURES_IO_UART	Set when the UART is implemented (via the IO_UART_USE generic)
17	CPU_MFEATURES_IO_MTIME	Set when the MTIME is implemented (via the IO_MTIME_USE generic)
16	CPU_MFEATURES_IO_GPIO	Set when the GPIO is implemented (via the IO_GPIO_USE generic)
4	CPU_MFEATURES_MEM_INT_DMEM	Set when the processor-internal IMEM is implemented (via the MEM_INT_IMEM_USE generic)
3	CPU_MFEATURES_MEM_INT_IMEM_ROM	Set when the processor-internal IMEM is read-only (via the MEM_INT_IMEM_ROM generic)
2	CPU_MFEATURES_MEM_INT_IMEM	Set when the processor-internal DMEM implemented (via the MEM_INT_DMEM_USE generic)
1	CPU_MFEATURES_MEM_EXT	Set when the external Wishbone bus interface is implemented (via the MEM_EXT_USE generic)
0	CPU_MFEATURES_BOOTLOADER	Set when the processor-internal bootloader is implemented (via the BOOTLOADER_USE generic)

CSR_MCLOCK

The MCLOCK register is a custom CSR. It contains the clock frequency of the processor in Hz (assigned via the CLOCK_FREQUENCY generic) and is read-only.

CSR_MISPACEBASE

The MISPACEBASE register is a custom CSR. It shows the base address of the instruction address space and is defined via the MEM_ISPACE_BASE generic and is read-only.

CSR_MISPACESIZE

The MISPACESIZE register is a custom CSR. It shows size of the instruction address space in bytes and is defined via the MEM_ISPACE_SIZE generic. This register is read-only.

CSR_MDSPACEBASE

The MDSPACEBASE register is a custom CSR. It shows the base address of the data address space and is defined via the MEM_DSPACE_BASE generic and is read-only.

CSR_MDSPACESIZE

The MDSPACESIZE register is a custom CSR. It shows size of the data address space in bytes and is defined via the MEM_DSPACE_SIZE generic. This register is read-only.

2.4. Exceptions and Interrupts

The NEORV32 supports the following exceptions and instructions. The identifier codes and the priority are compliant to the RISC-V specifications. 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:

Priority	`mcause` value	ID [C]	Function
1 (highest)	0x8000000B	EXCID_MEI	Machine external interrupt (via CLIC)
2	0x80000007	EXCID_MTI	Machine timer interrupt (via MTIME)
3	0x80000003	EXCID_MSI	Machine software interrupt (via `mip` CSR)
4	0x00000000	EXCID_I_MISALIGNED	Instruction address misaligned
5	0x00000001	EXCID_I_ACCESS	Instruction access fault
6	0x00000002	EXCID_I_ILLEGAL	Illegal instruction
7	0x0000000B	EXCID_MENV_CALL	Environment call from M-mode (`ECALL` instruction)
8	0x00000003	EXCID_BREAKPOINT	Breakpoint (`EBREAK` instruction)
9	0x00000006	EXCID_S_MISALIGNED	Store address misaligned
10	0x00000004	EXCID_L_MISALIGNED	Load address misaligned
11	0x00000007	EXCID_S_ACCESS	Store access fault
12 (lowest)	0x00000005	EXCID_L_ACCESS	Load access fault

2.5. Address Space

The CPU is a 32-bit architecture. Hence, it can access an address space of up to 2^{32} bytes (4GB). This address space is divided into 4 main region: The instruction memory space for instructions, the data memory space for application runtime data, the bootloader address space for the processor-internal bootloader and the IO address space for the processor-internal peripheral/IO devices.

The beginning of the memory space for instructions is defined via the `MEM_MISPACEBASE` generic. This generic must be 4-byte aligned. The complete size of the instruction memory space is defined via the `MEM_MISPACESIZE` generic (in bytes). Analogous, the beginning of the memory space for data is defined via the `MEM_MDSPACEBASE` generic. This generic must be 4-byte aligned. The complete size of the data memory space is defined via the `MEM_MDSPACESIZE` generic (in bytes). The instruction and data memory spaces may overlap.

The base address of the bootloader and the IO region for the peripheral devices are fixed. These address regions cannot be used for other applications – even if the bootloader or all IO devices are not implemented.

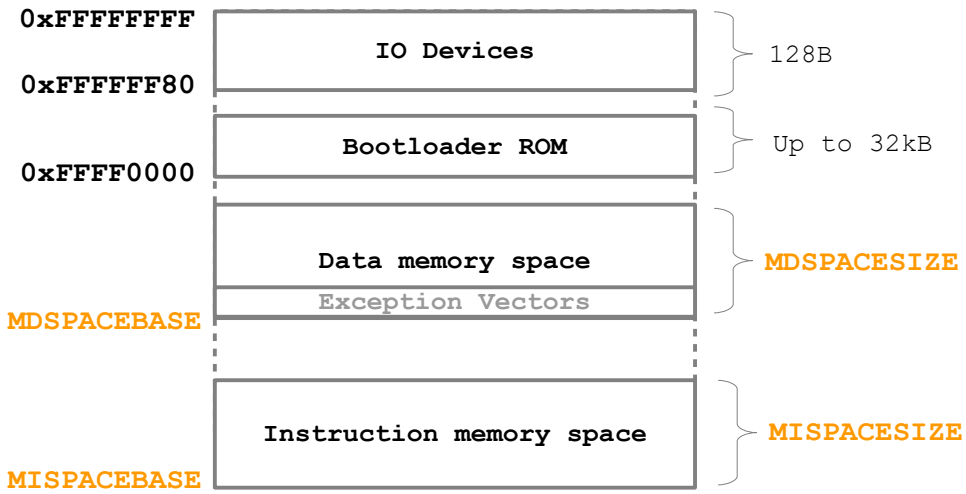

Figure 3: General NEORV32 address space layout

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). The processor-internal instruction memory (IMEM) can be implemented as ROM (`MEM_INT_IMEM_ROM`), which is initialized with the application code during synthesis.

If the processor-internal IMEM is implemented, it is located at the base address of the instruction address space. Also, the processor-internal data memory is located at the beginning of the data address space if implemented.

If the configured instruction/data memory space is greater than the size of the IMEM/DMEM, the accesses to the according addresses are forwarded to the external bus interface to interface processor-external memories and peripheral devices (when `MEM_EXT_USE` generic is `true`).

2.5.1. Processor-Internal Instruction Memory (IMEM)

A processor-internal instruction memory (`rtl/core/neorv32_imem.vhd`) can be enabled 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 (defined via the `MEM_MISPACESIZE` generic) and located at the beginning of the instruction memory space (defined via the `MEM_MISPACEBASE` generic).

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.

2.5.2. Processor-Internal Data Memory (DMEM)

A processor-internal data memory (`rtl/core/neorv32_dmem.vhd`) can be enabled 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 (defined via the `MEM_MDSPACESIZE` generic) and located at the beginning of the data memory space (defined via the `MEM_MDSPACEBASE` generic). The DMEM is always implemented as RAM.

2.5.3. Processor-Internal Bootloader ROM (BOOTROM)

As the name already suggests, the boot ROM (`rtl/core/neorv32_boot_rom.vhd`) 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

When 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 `MEM_MISPACEBASE` generic. 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.

2.5.4. 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
Configuration generics:	`MEM_EXT_USE`	Enable external memory interface when `true`
	`MEM_EXT_REG_STAGES`	Number of interface register stages
	`MEM_EXT_TIMEOUT`	Maximum length of bus accesses

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, after which an exception will be raised, is defined via the `MEM_EXT_TIMEOUT` generic of the NEORV32 processor.

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 uses Classic Pipelined Wishbone Transactions. There is always a delay of at least one clock cycle between issuing a bus access and read-back / acknowledge. The transactions are always in order and cannot overlap.

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.

2.5.5. Processor-Internal Peripheral/IO 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 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.

The peripheral/IO address space cannot be used for other applications even when all of the devices are not implemented.

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

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 provide a time-based configuration, provide a 3 bit prescaler select in their according control register to select 1 out of the 8 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 `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

3. Peripheral/IO Devices

This chapter gives detailed information about the available processor-internal peripheral / IO devices. You do not need to worry about peripheral device registers and register bits when writing an application for the NEORV32. The core software library completely abstracts the underlying hardware via high-level C functions.

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.asm`.

Nomenclature

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 defined in the main main processor core library file `sw/lib/include/neorv32.h`. The registers and/or register bits, which can be directly accessed 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.

3.1. 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` `gpio_i`	16-bit parallel output port 16-bit parallel input port
Configuration generics:	`IO_GPIO_USE`	Implement GPIO port unit when `true`
CPU interrupts:	`none`	
CLIC interrupts:	`CLIC channel 2`	Pin-change interrupt

Theory of Operation

The general purpose parallel IO port unit provides a simple 16-bit parallel input port and a 16-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.

The parallel input port features a single pin-change interrupt. Whenever an input pin has a low-to-high or high-to-low transition, the interrupt is triggered. When the modules is disabled for implementation, the pin-change interrupt is permanently disabled.

Register Map

Address	Name [C]	Bit(s) (Name) [C]	R/W	Function
0xFFFFFF80	GPIO_INPUT	0..15	r/-	Parallel input port
0xFFFFFF84	GPIO_OUTPUT	0..15	r/w	Parallel output port

Table 7: GPIO port unit register map

3.2. Core-Local Interrupt Controller (CLIC)

Overview

Hardware source file(s):	`neorv32_clic.vhd`	
Software driver file(s):	`neorv32_clic.c` `neorv32_clic.h`	
Top entity ports:	`ext_irq_i` `ext_ack_o`	2-bit external interrupt request 2-bit external acknowledge
Configuration generics:	`IO_CLIC_USE`	Implement CLIC when `true`
CPU interrupts:	`MEI`	Machine external interrupt
CLIC interrupts:	`none`	

Theory of Operation

The core-local interrupt controller implements a simple interrupt controller for the processor-internal peripherals. It features eight independent IRQ channels. Whenever a channel is triggered, the interrupt request is stored and forwarded to the CPU's machine external interrupt.

The CLIC features a single control register The unit is globally enabled when setting the `CLIC_CT_EN` bit. If this bit is cleared during operation, all buffered interrupt requests are deleted. Each interrupt request channel features a unique enable signal (`CLIC_CT_IRQx_EN`), which activates the according channel when set. Channel 0 has the highest priority, while channel 7 has the lowest. If several interrupt requests arise at the same time, the one with highest priority will be processed while the remaining ones are internally buffered. Note, that all interrupt request channels **trigger on high-level**.

The following table shows the CLIC interrupt channels and according sources.

Channel #	Channel ID [C]	Priority	Source	Function
0	`CLIC_CH_WDT`	`highest`	`WDT`	Watchdog timeout
1			`-`	*reserved*
2	`CLIC_CH_GPIO`		`GPIO`	GPIO input pin-change
3	`CLIC_CH_UART`		`UART`	RX or TX done
4	`CLIC_CH_SPI`		`SPI`	Transmission done
5	`CLIC_CH_TWI`		`TWI`	Transmission done
6	`CLIC_CH_EXT0`		`external`	External via `ext_irq_i(0)`
7	`CLIC_CH_EXT1`	`lowest`	`external`	External via `ext_irq_i(1)`

Table 8: CLIC interrupt request channels

When an interrupt is signaled to the CPU, the application program can determine which channel caused the request by reading the `CLIC_CT_SRCx` bits. A "000" indicates channel 0, a "001" channel 1 and so on. The current interrupt is acknowledged by writing a 1 to the `CLIC_CT_ACK` control register bit.

Each of the eight interrupt channels can also be triggered by software. For that, the SW IRQ enable bit `CLIC_CT_SW_IRQ_EN` has to be set while the `CLIC_CT_SW_IRQ_SRCx` bits define the channel to be triggered (e.g., “000” for channel 0). A software triggering is only possible when the according IRQ channel is enabled and the CLIC is activated at all.

The CLIC also provides two processor-external interrupt request lines with according acknowledge via the top entity’s `ext_irq_i` and `ext_ack_o` ports.

When the CLIC is disabled from implementation, the `MEI` CPU interrupt is permanently disabled.

Register Map

Address	Name [C]	Bit(s) (Name) [C]		R/W	Function
0xFFFFFF88	CLIC_CT	0	CLIC_CT_SRC0	r/-	IRQ source bit 0
		1	CLIC_CT_SRC1	r/-	IRQ source bit 1
		2	CLIC_CT_SRC2	r/-	IRQ source bit 2
		3	CLIC_CT_ACK	0/w	ACK current IRQ when set
		4	CLIC_CT_EN	r/w	CLIC enable
		8	CLIC_CT_IRQ0_EN	r/w	Enable CLIC channel 0
		9	CLIC_CT_IRQ1_EN	r/w	Enable CLIC channel 1
		10	CLIC_CT_IRQ2_EN	r/w	Enable CLIC channel 2
		11	CLIC_CT_IRQ3_EN	r/w	Enable CLIC channel 3
		12	CLIC_CT_IRQ4_EN	r/w	Enable CLIC channel 4
		13	CLIC_CT_IRQ5_EN	r/w	Enable CLIC channel 5
		14	CLIC_CT_IRQ6_EN	r/w	Enable CLIC channel 6
		15	CLIC_CT_IRQ7_EN	r/w	Enable CLIC channel 7
		16	CLIC_CT_SW_IRQ_SRC0	0/w	SW IRQ trigger select bit 0
		17	CLIC_CT_SW_IRQ_SRC1	0/w	SW IRQ trigger select bit 1
		18	CLIC_CT_SW_IRQ_SRC2	0/w	SW IRQ trigger select bit 2
		19	CLIC_CT_SW_IRQ_EN	0/w	SW IRQ trigger enable

Table 9: CLIC register map

3.3. 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:	`none`	
CLIC interrupts:	`CLIC 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 CLIC. 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

Table 10: WDT register map

3.4. 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
CLIC interrupts:	`none`	

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`. The interrupt is directly forwarded to the CPU's `MTI` interrupt. An MTIME interrupt is acknowledged by writing to `MTIMECMP_HI`.

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 11: MTIME register map

3.5. 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` `uart_rxd_o`	Serial transmitter output Serial receiver input
Configuration generics:	`IO_UART_USE`	Implement UART when `true`
CPU interrupts:	`none`	
CLIC interrupts:	`CLIC channel 3`	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 20-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	20-bit BAUD configuration value
		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	UART_DATA_AVAIL	r/-	RX data available when set

Table 12: UART register map

3.6. Serial Peripheral Interface Master (SPI)

Overview

Hardware source file(s):	`neorv32_spi.vhd`	
Software driver file(s):	`neorv32_spi.c` `neorv32_spi.h`	

Top entity ports:	`spi_sclk_o`
	`spi_mosi_o`
	`spi_miso_i`
	`spi_csn_o`
		1-bit serial master clock output
		1-bit serial master data output
		1-bit serial master data input
		8-bit chip select port (low-active)

Configuration generics:	`IO_SPI_USE`	Implement SPI when `true`
CPU interrupts:	`none`	
CLIC interrupts:	`CLIC channel 4`	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 13: SPI transceiver register map

3.7. Two Wire Serial Interface Master (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` `twi_scl_io`	Bi-directional serial data line Bi-directional serial clock line
Configuration generics:	`IO_TWI_USE`	Implement TWI when `true`
CPU interrupts:	`none`	
CLIC interrupts:	`CLIC channel 5`	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 slave nodes. The NEORV32 TWI implements a TWI **master**. It features "**clock stretching**", so a slow slave can halt the transmission by pulling the SCL line low. **Currently no multi-master support is available. Also, the TWI unit cannot operate in slave 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 master 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 slave has to acknowledge each transferred byte. When the `TWI_CT_ACK` bit is set after a completed transmission, the accessed slave has send an acknowledge. If it is cleared after a transmission, the slave has send a not-acknowledge (NACK). The NEORV32 TWI master can also send an ACK (→ master 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 slave 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 master. Hence, external pull-up resistors are required for the SDA and SCL 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 master 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 14: TWI register map

3.8. 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 PWM output
Configuration generics:	`IO_PWM_USE`	Implement PWM controller when `true`
CPU interrupts:	`none`	
CLIC 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:

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

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 15: PWM controller register map

3.9. 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`	
CLIC interrupts:	`none`	

Theory of Operation

The NEORV32 true random number generator provides 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. It features a platform independent architecture based on two papers which are cited at the bottom of the following pages.

When the `TRNG_CT_EN` bit is set, the TRNG starts operation. Make sure to configure the GARO taps using the `TRNG_CT_TAPx` bits in advance. As soon as the `TRNG_DATA_VALID` bit in the `TRNG_DATA` register is set, the current sampled 16-bit random data can be obtained from the lowest 16 bits of the `TRNG_DATA` register. Note, that the TRNG needs at least 16 clock cycles to generate a new random byte. During this sampling time the current output random data is kept in the output register until a valid sampling of the new byte has completed.

Architecture

The NEORV32 TRNG is based on the *GARO **Ga**lois **R**ing **O**scillator TRNG*[7]. Basically, this architecture is an asynchronous LFSR constructed from a chain of inverters. Before the output signal of one oscillator 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).

The default setup of the TRNG uses a total of 16 inverters and a software configurable GARO tap maks. 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[8] since no specific VHDL attributes are required.
The single-bit output signal of the GARO array is fed through flip flops to eliminate any metastability beyond this point. Afterwards, a Von-Neuman de-biasing is applied to get rid of any any bias introduced by the GARO array. If the de-biasing fails, an additional cycle is required to obtain a now random sample. This process might replicate depending on the quality of the GARo oscillation.
The single-bit output signal of the GARO array is fed through flip flops to eliminate any metastability

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

beyond this point. Afterwards, a Von-Neuman de-biasing is applied to get rid of any any bias introduced by the GARO array. If the de-biasing fails, an additional cycle is required to obtain a now random sample. This process might replicate depending on the quality of the GARo oscillation.

This de-biased signal is used as input for a simple chaos machine post-processing to provide a 'better' uniform distribution. This chaos machine is implemented as a 16-bit LFSR. As soon as 16 valid bits (so no errors during the de-biasing) have bin sampled, the resulting data is moved to the output register and is available for fetching by the CPU bus.

Register Map

Address	Name [C]	Bit(s) (Name) [C]		R/W	Function
0xFFFFFFC0	TRNG_CT	15:0	TRNG_CT_TAP	r/w	16-bit GARo tap mask configuration
		31	TRNG_CT_EN	r/w	TRNG enable
0xFFFFFFC4	TRNG_DATA	15:0	TRNG_DATA	r/-	Random data output
		31	TRNG_DATA_VALID	r/-	Random data valid when set

Table 16: TRNG register map

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 and bootloader start-up codes	`sw/common/bootloader_crt0.S` `sw/common/crt0.S`
Application and bootloader linker scripts	`sw/common/bootloader_neorv32.ld` `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 booloader	`sw/bootloader/bootloader.c`

The complete 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 NEORV32 uses 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

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_clic.c	neorv32_clic.h	HW driver functions for the CLIC.
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/build` folder. To view the documentation, open the `docs/build/html/index.html` file with your browser of choice. Click on the "files" tab to see a list of all documented files.

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 varibales 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. Makefile 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 toolchain. You should run this target at least once after installation.
`info`	Show the makefile configuration (see next chapter).
`compile`	Compile all sources and generate `compile` executable for upload via bootloader
`install`	Compile all sources, generate executable (via `compile` target) for upload via bootloader and generate and install IMEM VHDL initialization image file `rtl/core/neorv32_application_image.vhd`.
`all`	Execute `compile` 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 uses the bootloader-specific start-up code (`sw/common/bootloader_crt0.S`) and linker script (`sw/common/bootloader_neorv32.ld`) instead.

4.3.2. Makefile Configuration

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

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

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

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

# Compiler toolchain (use default if not set by user)
RISCV_TOOLCHAIN ?= riscv32-unknown-elf

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

# Path to runtime c library (use default if not set by user)
LIBC_PATH   ?= $(dir $(shell which $(CC)))../$(RISCV_TOOLCHAIN)/lib/libc.a
LIBGCC_PATH ?= $(dir $(shell which $(CC)))../lib/gcc/$(RISCV_TOOLCHAIN)/*/libgcc.a

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

Description of Makefile Configuration Variables

`EFFORT`	Optimization level, optimize for size (`-Os`) is default; legal values: `-O0` `-O1` `-O2` `-O3` `-Os`
`APP_SRC`	The *.c source files of the application. *.c-files in the current folder are automatically added via wildcard. Additional files can be added; separated by white spaces
`APP_INC`	Include file folders; separated by white spaces; must be defined with `-I` prefix
`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.
`LIBC_PATH`	Location of the standard C library.
`LIBGCC_PATH`	Locations of the standard GCC C-library.
`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.

The makefile configuration variable can be re-defined directly when invoking the makefile:
`$ make MARCH=-march=rv32ic clean_all compile`

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`. This file is parsed 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 actual program image. 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 you to upload new program executable at every time. If you have an external SPI flash connected to the processor (for example the FPGA configuration memory), you can store the program executable to it and the system can directly boot it after reset without any user interaction.

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

The bootloader requires the UART for manual executable upload or 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 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)`
- `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: Jun 17 2020
HWV:  0.0.2.3
CLK:  0x05F5E100 Hz
MISA: 0x42801104
CONF: 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 version and several system parameters:

`BLDV`	Bootloader version (built time).
`HWV`	Processor hardware version (from the `mimpid` CSR).
`CLK`	Processor clock speed in Hz (via the `mclock` CSR from the `CLOCK_FREQUENCY` generic).
`MISA`	CPU extensions (from the `misa` CSR).
`CONF`	Processor configuration (via the `mfeatures` CSR from the IO and MEM config. generics).
`IMEM`	Instructions memory base address and size in byte (via the `mispacebase` & `mispacesize` CSRs from the `MEM_ISPACE_BASE` & `MEM_ISPACE_SIZE` generics).
`DMEM`	Data memory base address and size in byte (via the `mdspacebase` & `mdspacesize` CSRs from the `MEM_DSPACE_BASE` & `MEM_DSPACE_SIZE` generics).

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: Jun 17 2020
HWV:  0.0.2.3
CLK:  0x05F5E100 Hz
MISA: 0x42801104
CONF: 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

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.

4.5.1. Auto Boot Sequence

When you reset the NEORV32processor, 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.

4.5.2. 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 for the executable image inside the SPI flash is defined in the "user configuration" section 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     0x00040000

/** 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.

4.5.3. Bootloader Error Codes

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

`ERR_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.
`ERR_1`	Your program is way too big for the internal processor's instructions memory. Increase the memory size or reduce (optimize!) your application code.
`ERR_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.
`ERR_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.
`ERR_4`	The instruction memory is marked as read-only. Set the `MEM_INT_IMEM_ROM` generic to `false` to allow write accesses.
`ERR_5`	This error pops up when an exception was triggered. Such an error with exception code "0x00000002" will be generated when you try to boot from the instruction memory, but no valid executable has been loaded yet.

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 runtime environment is implemented in the `sw/common/crt0.asm` 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` – `x31` (`x1` – `x15` only for embedded CPU mode) with zero.
- Setup stack-pointer (`x2`): The stack always starts at the very end of the data address space (`DSPACEBASE` + `DSPACESIZE` – 4).
- Initialize the global pointer (`x3`) according to the `.data` segment layout provided by the linker script.
- **Initialize the NEORV32 runtime environment for catching exceptions and interrupts.**
- 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 also processed by 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 initialize 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 the `WFI` instruction if implemented).

The most interesting point in terms of this chapter is the NEORV32 runtime environment `neorv32_rte`. This minimal runtime environment is executed right after booting the application. It basically 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* implemented in the `sw/common/crt0.asm` file. 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 first-level exception handler uses an interrupt/exception vector table located right at the beginning of the data memory space. This vector table is not available for "normal" application data storage, which is enforced by the NEORV32 linker script (`sw/common/neorv32.ld`):

```
MEMORY
{
  rom  (rx) : ORIGIN = 0x00000000,          LENGTH = 16*1024
  ram (rwx) : ORIGIN = 0x80000000 + 2*16*4, LENGTH =  8*1024
}
```

The vector tables has 32 x 4-byte entries. The first 16 entries are reserved for exception handlers, the second 16 entries are reserved for the interrupt handlers. While executing the start-up code each entry is initialized with a default handler, that actually does nothing but return (via `ret`). By this, every exception and every interrupt is safely catched. Of course, this default handler cannot handle system-critical events. Hence, these default handlers are only relevant for an early system state. The actual application program should replace these dummy handler entries in the vector table with actual "real" handlers.

Using the NEORV32 Runtime Environment (RTE)

The actual application should not directly mess with the exception vector table. Instead, the NEORV32 runtime environment (`sw/lib/include/neorv32_rte.h`) provides functions to install the real *second-level handlers* for each of the implemented exceptions and interrupts:

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

The following `exc_id` exception IDs are available:

ID name [C]	Description / exception or interrupt causing event
`EXCID_I_MISALIGNED`	Instruction address misaligned
`EXCID_I_ACCESS`	Instruction (bus) access fault
`EXCID_I_ILLEGAL`	Illegal instruction
`EXCID_BREAKPOINT`	Breakpoint (EBREAK instruction)
`EXCID_L_MISALIGNED`	Load address misaligned
`EXCID_L_ACCESS`	Load (bus) access fault
`EXCID_S_MISALIGNED`	Store address misaligned
`EXCID_S_ACCESS`	Store (bus) access fault
`EXCID_MENV_CALL`	Environment call from machine mode (ECALL instruction)
`EXCID_MSI`	Machine software interrupt
`EXCID_MTI`	Machine timer interrupt (via MTIME)
`EXCID_MEI`	Machine external interrupt (via CLIC)

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

```
void handler_xyz(void) {

}
```

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 default dummy handler, so even uninstalled exceptions and interrupts are further captured.

```
int neorv32_rte_exception_uninstall(uint8_t exc_id);
```

Example: Removing the MTIME interrupt handler:

```
neorv32_rte_exception_uninstall(EXC_MTI);
```

Debugging

For debugging purpose, the NEORV32 runtime environment features a "debug handler" that can be installed for all available exceptions and interrupts. This debug handler will give detailed information about the triggered exception/interrupt via UART. After that, it will try to resume normal application execution. This attempt might fail – but at least the debug handler gives detailed information for digging into the faulty code. The installation of the debug handler is triggered via the following function:

```
void neorv32_rte_enable_debug_mode(void);
```

Installing the debug handler will override all previous interrupt/exception handler installations. Hence, enable the NEORV32 RTE debug mode right at program start and install your custom exception/interrupt handler after that.

More information regarding the NEORV32 runtime environment can be found in the `doxygen` software documentation.

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.

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 `rv32i`):

```
riscv-gnu-toolchain$ ./configure --prefix=/opt/riscv --with-arch=rv32i –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):

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

Alternatively, you can directly download the according toolchain archive 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 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 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. 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: Add the `rtl/core/top_templates/neorv32_test_setup.vhd` VHDL file to your project too, and select it as top entity. This test setup provides a minimal test hardware setup:

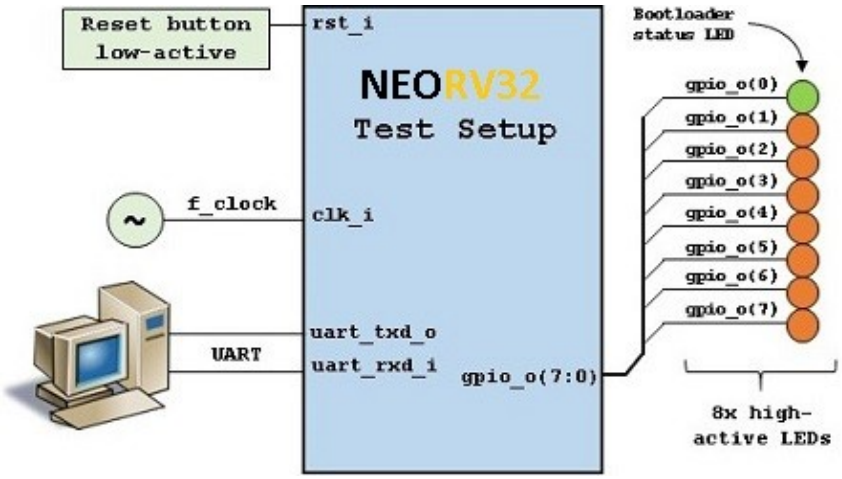

Figure 4: Hardware configuration of the NEORV32 test setup

5. 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.

6. 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). But 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, that you need to adapt, is marked in **orange**.

```
neorv32_top_inst: neorv32_top
generic map (
  -- General --
  CLOCK_FREQUENCY           => 100000000, -- in Hz
  HART_ID                   => x"00000000",
  BOOTLOADER_USE            => true,
  -- RISC-V CPU Extensions --
  CPU_EXTENSION_RISCV_C     => false,
  CPU_EXTENSION_RISCV_E     => false,
  CPU_EXTENSION_RISCV_M     => false,
  CPU_EXTENSION_RISCV_Zicsr => true,
  -- Memory configuration: Instruction memory --
  MEM_ISPACE_BASE           => x"00000000",
  MEM_ISPACE_SIZE           => 16*1024, -- in BYTES
  MEM_INT_IMEM_USE          => true,
  MEM_INT_IMEM_SIZE         => 16*1024, -- in BYTES
  MEM_INT_IMEM_ROM          => false,
  -- Memory configuration: Data memory --
  MEM_DSPACE_BASE           => x"80000000",
  MEM_DSPACE_SIZE           => 8*1024, -- in BYTES
  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_CLIC_USE               => true,
  IO_TRNG_USE               => false
)
```

7. 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. We will 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 as these values will be required for setting up the software framework in the next chapter.

8. 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;
```

9. 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 must be **low-active**, so maybe you need to invert the input signal. 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). Finally, connect the UART signals `uart_txd_o` and `uart_rxd_i` to your serial host interface (dedicated pins, USB-to-serial converter, etc.).

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

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

12. 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.

1. You need to tell the linker the size of the processor's instruction and data memories. We have just configured the test setup – so you should remember the memory configuration.

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

```
MEMORY
{
  rom  (rx) : ORIGIN = 0x00000000,          LENGTH = 16*1024
  ram (rwx) : ORIGIN = 0x80000000 + 2*16*4, LENGTH =  8*1024
}
```

3. There are four parameters that are relevant here: 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 `MEM_ISPACE_BASE` generic.
The `rom` `LENGTH` parameter has to be equal to the configuration of the `MEM_ISPACE_SIZE` generic.

The `ram` `ORIGIN` parameter has to be equal to the configuration of the `MEM_DSPACE_BASE` generic.
The `ram` `LENGTH` parameter has to be equal to the configuration of the `MEM_DSPACE_SIZE` generic.

Make sure you **do not** delete the `+ 2*16*4` right after the origin of the RAM! This offset is required to reserve space for the exception vector table managed by the NEORV32 runtime environment.

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.

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/build`. 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, execute:

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

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 process takes this file and creates a final executable. The makefile will show the resulting memory utilization and the executable size in the console:

```
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
```

4. That's it. The compile 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.

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. In this tutorial, we will use **TeraTerm** as an exemplary serial terminal program for **Windows**, but the general procedure is the same for other terminal programs, build environments or operating systems.

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

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**

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)`

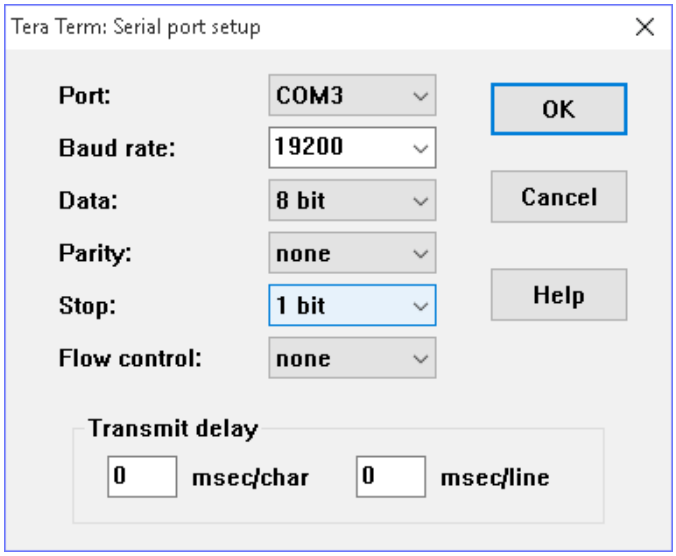

Figure 5: Serial configuration of TeraTerm

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: Jun 17 2020
HWV:  0.0.2.3
CLK:  0x05F5E100 Hz
MISA: 0x42801104
CONF: 0x01FF0015
IMEM: 0x00008000 bytes @ 0x00000000

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:

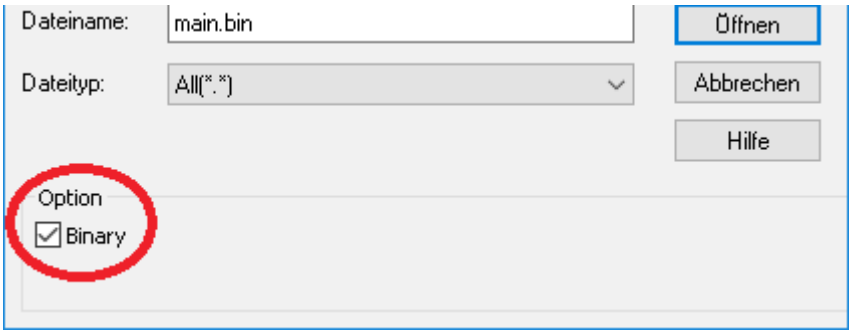

Figure 6: Transfer executable in binary mode (German version of TeraTerm)

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.

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
```

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).

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=rv32icm`	
`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=rv32ecm`	

The toolchain always supports the privileged instructions (like `ECALL` or CSR access instructions) regardless of the `MARCH` or `MABI` configuration. However, the compiler will not generate the according instructions by itself. Privileged instructions are only used when explicitly coded as inline assembly or when using according libraries. When the `CPU_EXTENSION_RISCV_Zicsr` extension is not synthesized, all these instructions will behave like NOPs. Instructions returning data will always return zero.

When using the embedded CPU mode (`CPU_EXTENSION_RISCV_E` generic set `true`) a C-library, which was compiled for the `ilp32e` ABI, is required.

The CSR access extension (`CPU_EXTENSION_RISCV_Zicsr`) is supported by all toolchain configurations and all ABIs and need no further makefile configuration.

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 bootloader linker script `sw/common/bootloader_neorv32.ld` and adapt the `LENGTH` parameter of the `bootrom` according to your new memory size. `boot_size_c` and `LENGTH` have to be always identical.

```
MEMORY
{
  bootrom  (rx) : ORIGIN = 0xFFFF0000, LENGTH = 4*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).

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 @ 0x00040000? (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 @ 0x00040000? (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.

6. Troubleshooting

If your setup does not work as expected, scroll through the following point. Maybe you have missed something during setup. If you are still encountering problems, open a new issue on GitHub or contact me.

- Check the correct installation of the toolchain with a `$ make check`.
- Check the synthesis tool for errors and warnings. Double-check the timing report.
- Does the processor simulate correctly?
- Synthesis tools can be a little bit obscure sometimes. Use the default synthesis options and do not start with a target frequency of 800MHz for your first setup.
- If your application does not run, make a clean rebuild: `$ make clean_all compile`
- If it still does not run, enable the debug feature of the NEORV32 runtime environment.
- Make sure your hardware configuration (like memory sizes, CPU extensions,...) is always sync with the toolchain (linker script configuration, targeted architecture (`march`), ABI (`mabi`),...).
- ...

7. Change Log

Date (DD.MM.YYYY)	HW version	Modifications
23.06.2020	`0.0.2.3`	Publication