

□ | *neorv32\_logo\_riscv.png*[pdfwidth=6.25in,align=center]

# The NEORV32 RISC-V Processor - User Guide

The NEORV32 Community and Stephan Nolting

Version v1.11.1-r20-g2b7d619f



### Documentation

The online documentation of the project (a.k.a. the **data sheet**) is available on GitHub-pages: <https://stnolting.github.io/neorv32/> The online documentation of the **software framework** is also available on GitHub-pages: <https://stnolting.github.io/neorv32/sw/files.html>

# Table of Contents

1. Software Toolchain Setup .....	5
2. General Hardware Setup .....	7
3. General Software Framework Setup .....	10
3.1. Modifying the Linker Script .....	10
3.2. Overriding the Default Configuration .....	11
4. Application Program Compilation .....	12
5. Uploading and Starting of a Binary Executable Image via UART .....	13
6. Installing an Executable Directly Into Memory .....	16
7. Setup of a New Application Program Project .....	18
8. Application-Specific Processor Configuration .....	19
8.1. Optimize for Performance .....	19
8.2. Optimize for Size .....	19
8.3. Optimize for Clock Speed .....	20
8.4. Optimize for Energy .....	21
9. Adding Custom Hardware Modules .....	22
9.1. Standard ( <i>External</i> ) Interfaces .....	22
9.2. External Bus Interface .....	22
9.3. Custom Functions Subsystem .....	23
9.4. Custom Functions Unit .....	23
9.5. Comparative Summary .....	23
10. Customizing the Internal Bootloader .....	25
10.1. Auto-Boot Configuration .....	27
11. Programming an External SPI Flash via the Bootloader .....	28
11.1. Programming an Executable .....	28
12. Packaging the Processor as Vivado IP Block .....	30
13. Simulating the Processor .....	33
13.1. Testbench .....	33
13.2. Faster Simulation Console Output .....	34
13.3. GHDL Simulation .....	34
13.4. Simulation using Application Makefiles .....	34
13.4.1. Hello World! .....	35
14. Building the Documentation .....	37
15. Zephyr RTOS Support .....	38
16. FreeRTOS Support .....	39
17. LiteX SoC Builder Support .....	40
17.1. LiteX Setup .....	40
17.2. LiteX Simulation .....	42

18. Debugging using the On-Chip Debugger .....	44
18.1. Hardware Requirements .....	44
18.2. OpenOCD .....	45
18.3. Debugging with GDB .....	46
18.3.1. Software Breakpoints .....	48
18.3.2. Hardware Breakpoints .....	49
18.4. Segger Embedded Studio .....	49
19. NEORV32 in Verilog .....	51
20. Eclipse IDE .....	52
20.1. Eclipse Prerequisites .....	52
20.2. Import The Provided Eclipse Example Project .....	53
20.3. Setup a new Eclipse Project from Scratch .....	53
20.3.1. Create a new Project .....	53
20.3.2. Add Initial Files .....	54
20.3.3. Add Build Targets (optional) .....	54
20.3.4. Configure Build Tools .....	55
20.3.5. Adjust Default Build Configuration (optional) .....	55
20.3.6. Add NEORV32 Software Framework .....	55
20.3.7. Setup OpenOCD .....	56
20.3.8. Setup Serial Terminal .....	57
20.4. Eclipse Setup References .....	57
21. Legal .....	58
About .....	58
License .....	58
Proprietary Notice .....	59
Disclaimer .....	59
Limitation of Liability for External Links .....	59
Citing .....	59
Acknowledgments .....	60

## Let's Get It Started!

This user guide uses the NEORV32 project *as is* from the official **neorv32** repository. To make your first NEORV32 project run, follow the guides from the upcoming sections. It is recommended to follow these guides step by step and eventually in the presented order.



This guide uses the minimalistic and platform/toolchain agnostic SoC **test setups** from **rtl/test\_setups** for illustration. You can use one of the provided test setups for your first FPGA tests.

For more sophisticated example setups have a look at the **neorv32-setups** repository, which provides **SoC setups** for various FPGAs, boards and toolchains.

## Quick Links

- **Toolchain**, **hardware** and **general software framework** setup
- **compile** an application and **upload** it or making it **persistent** in internal memory
- setup a new **application project**
- **optimizing** the core for your application
- add **custom hardware extensions** and **customizing the bootloader**
- **program** an external SPI flash for persistent application storage
- generate an AMD Vivado **IP block**
- **simulate** the processor and **build the documentation**
- RTOS support for **Zephyr** and **FreeRTOS**
- build SoCs using **LiteX**
- in-system **debugging** of the whole processor
- **NEORV32 in Verilog** - an all-Verilog "version" of the processor
- use the **Eclipse IDE** to develop and debug code for the NEORV32

# Chapter 1. Software Toolchain Setup

To compile (and debug) executables for the NEORV32 a RISC-V-compatible toolchain is required. By default, the project's software framework uses the GNU C Compiler RISC-V port "RISC-V GCC". Basically, there are two options to obtain such a toolchain:

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



### Default GCC Prefix

The default toolchain prefix for this project is **riscv-none-elf-** (**RISCV\_PREFIX** variable).

## Toolchain Requirements



### Library/ISA Considerations

Note that a toolchain build with **--with-arch=rv32imc** provides library code (like the C standard library) compiled entirely using compressed (**C**) and **mul/div** instructions (**M**). Hence, this pre-compiled library code CANNOT be executed (without emulation) on an architecture that does not support these ISA extensions.

## Building the Toolchain from Scratch

The official RISC-V GCC GitHub repository (<https://github.com/riscv-collab/riscv-gnu-toolchain>) provides instructions for building the toolchain from scratch:

*Listing 1. Preparing GCC build for rv32i (minimal ISA only in this example)*

```
$ git clone https://github.com/riscv/riscv-gnu-toolchain
$ cd riscv-gnu-toolchain
$ riscv-gnu-toolchain$ ./configure --prefix=/opt/riscv --with-arch=rv32i --with
-abi=ilp32
$ riscv-gnu-toolchain$ make
```

## Downloading and Installing a Prebuilt Toolchain

Alternatively, a prebuilt toolchain can be used. Some OS package managers provide embedded RISC-V GCC toolchain. However, I can highly recommend the toolchain provided by the X-Pack project (MIT license): <https://github.com/xpack-dev-tools/riscv-none-elf-gcc-xpack>

## Toolchain Installation

To integrate the toolchain of choice into the NEORV32 software framework, the toolchain's binaries need to be added to the system path (e.g. **PATH** environment variable) so they can be used by a shell. Therefore, the absolute path to the toolchain's **bin** folder has to be appended to the **PATH** variable:

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

*bashrc*

This command can be added to `.bashrc` (or similar) to automatically add the RISC-V toolchain at every console start.

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

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

This will test all the tools required for generating NEORV32 executables. Everything is working fine if "Toolchain check OK" appears at the end of the log output.

## Chapter 2. General Hardware Setup

This guide shows the basics of setting up a NEORV32 project for simulation or synthesis *from scratch*. It uses a simple, exemplary test "SoC" setup of the processor to keep things simple at the beginning. This simple setup is intended for a first test / evaluation of the NEORV32.

The NEORV32 project features three minimalistic pre-configured test setups in `rtl/test_setups`. These test setups only implement very basic processor and CPU features and mainly differ in the actual boot configuration.

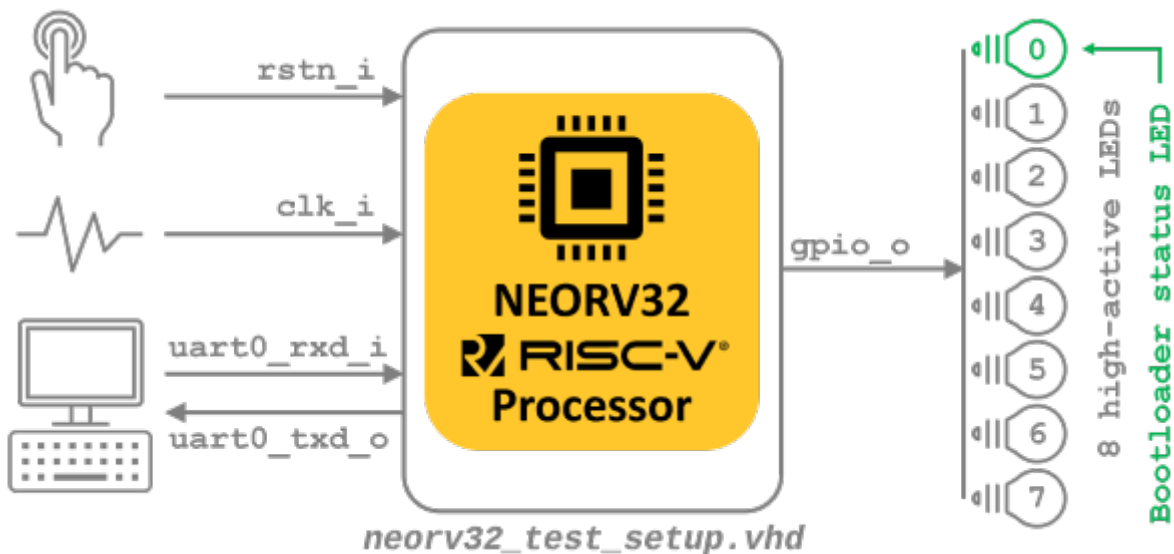


Figure 1. NEORV32 "hello world" test setup (`rtl/test_setups/neorv32_test_setup_bootloader.vhd`)

1. Create a new project with your FPGA/ASIC/simulator EDA tool of choice.
2. Add all VHDL files from the project's `rtl/core` folder to your project. Make sure to add all these rtl files to a new library called `neorv32`. If your toolchain does not provide a field to enter the library name, check out the "properties" menu of the added rtl files.



### Compile Order and File-List Files

Some tools (like Lattice Radiant) might require a *manual compile order* of the VHDL source files to identify the dependencies. The `rtl` folder features file-list files that list all required HDL files in their recommended compilation order (see [https://stnolting.github.io/neorv32/#\\_file\\_list\\_files](https://stnolting.github.io/neorv32/#_file_list_files)).

3. The `rtl/core/neorv32_top.vhd` VHDL file is the top entity of the NEORV32 processor, which can be instantiated within the actual project. However, in this tutorial we will use one of the pre-defined test setups from `rtl/test_setups` (see above).



Make sure to include the `neorv32` package into your design when instantiating the processor: add `library neorv32;` and `use neorv32.neorv32_package.all;` to your design unit.



4. Add the pre-defined test setup of choice to the project, too, and select it as **top entity**.
5. The entity of the test setups provides a minimal set of configuration generics, that might have to be adapted to match your FPGA and board:

*Listing 2. Test setup entity - configuration generics*

```
generic (
  -- adapt these for your setup --
  CLOCK_FREQUENCY   : natural := 100000000; ①
  MEM_INT_IMEM_SIZE : natural := 16*1024;    ②
  MEM_INT_DMEM_SIZE : natural := 8*1024      ③
);
```

① Clock frequency of `clk_i` signal in Hertz

② Default size of internal instruction memory: 16kB

③ Default size of internal data memory: 8kB

6. If you want to or if your FPGA does not provide sufficient resources you can modify the *memory sizes* (`MEM_INT_IMEM_SIZE` and `MEM_INT_DMEM_SIZE`).
7. There is one generic that has to be set according to your FPGA board setup: the actual 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).

#### Memory Layout



If you have changed the default memory configuration (`MEM_INT_IMEM_SIZE` and `MEM_INT_DMEM_SIZE` generics) keep those new sizes in mind - these values are required for setting up the software framework in the next section **General Software Framework Setup**.

8. 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 of the corresponding test setup, e.g.:

*Listing 3. Ports of `neorv32_test_setup_bootloader.vhd`*

```
port (
  -- Global control --
  clk_i      : in  std_ulogic; -- global clock, rising edge
  rstn_i     : in  std_ulogic; -- global reset, low-active, async
  -- GPIO --
  gpio_o     : out std_ulogic_vector(7 downto 0); -- parallel output
  -- UART0 --
  uart0_txd_o : out std_ulogic; -- UART0 send data
  uart0_rxd_i : in  std_ulogic  -- UART0 receive data
);
```

*Signal Polarity*

If your FPGA board has inverse polarity for certain input/output you need to add inverters. Example: The reset signal `rstn_i` is low-active by default; the LEDs connected to `gpio_o` are high-active by default.

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 is **low-active**, so maybe you need to invert the input signal.
10. If possible, connected at least bit `0` of the GPIO output port `gpio_o` to a LED (see "Signal Polarity" note above).
11. If you are using a UART-based test setup connect the UART communication signals `uart0_txd_o` and `uart0_rxd_i` to the host interface (e.g. a USB-UART converter).
12. If you are using the on-chip debugger setup connect the processor's JTAG signal `jtag_*` to a suitable JTAG adapter.
13. Perform the project HDL compilation (synthesis, mapping, placement, routing, bitstream generation).
14. Program the generated bitstream into your FPGA and press the button connected to the reset signal.
15. Done! The LED(s) connected to `gpio_o` should be flashing now.

# Chapter 3. General Software Framework Setup

To allow executables to be *actually executed* on the NEORV32 Processor the configuration of the software framework has to be aware to the hardware configuration. This guide focuses on the **memory configuration**. To enable certain CPU ISA features refer to the [\[enabling\\_risc\\_v\\_cpu\\_extensions\]](#) section.

This guide shows how to configure the linker script for a given hardware memory configuration. More information regarding the linker script itself can be found in the according section of the data sheet: [https://stnolting.github.io/neorv32/#\\_linker\\_script](https://stnolting.github.io/neorv32/#_linker_script)



If you have **not** changed the *default* memory configuration in section **General Hardware Setup** you are already done and you can skip the rest of this section.



Always keep the processor's **Address Space** layout in mind when modifying the linker script

There are two options to modify the default memory configuration of the linker script:

1. **Modifying the Linker Script**
2. **Overriding the Default Configuration** (recommended!)

## 3.1. Modifying the Linker Script

This will modify the linker script *itself*.

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

*Listing 4. Cut-out of the linker script `neorv32.ld`*

```
/* Default rom/ram (IMEM/DMEM) sizes */
__neorv32_rom_size = DEFINED(__neorv32_rom_size) ? __neorv32_rom_size : 2048M; ①
__neorv32_ram_size = DEFINED(__neorv32_ram_size) ? __neorv32_ram_size : 8K; ②

/* Default HEAP size (= 0; no heap at all) */
__neorv32_heap_size = DEFINED(__neorv32_heap_size) ? __neorv32_heap_size : 0; ③

/* Default section base addresses - do not change this unless the hardware-defined
address space layout is changed! */
__neorv32_rom_base = DEFINED(__neorv32_rom_base) ? __neorv32_rom_base : 0x00000000; /*
= VHDL package's "ispace_base_c" */ ④
__neorv32_ram_base = DEFINED(__neorv32_ram_base) ? __neorv32_ram_base : 0x80000000; /*
= VHDL package's "dspace_base_c" */ ⑤
```

- ① Default (max) size of the instruction memory address space (right-most value) (internal/external IMEM): 2048MB
  - ② Default size of the data memory address space (right-most value) (internal/external DMEM): 8kB
  - ③ Default size of the HEAP (right-most value): 0kB
  - ④ Default base address of the instruction memory address space (right-most value): `0x00000000`
  - ⑤ Default base address of the data memory address space (right-most value): `0x80000000`
2. Only the `neorv32_ram_size` variable needs to be modified! If you have changed the default DMEM (`MEM_INT_DMEM_SIZE` generic) size then change the right-most parameter (here: 8kB) so it is equal to your DMEM hardware configuration. The `neorv32_rom_size` does not need to be modified even if you have changed the default IMEM size. For more information see [https://stnolting.github.io/neorv32/#\\_linker\\_script](https://stnolting.github.io/neorv32/#_linker_script)
3. Done! Save your changes and close the linker script.

## 3.2. Overriding the Default Configuration

This will not change the default linker script at all. Hence, **this approach is recommended** as it allows a per-project memory configuration without changing the code base.

The RAM and ROM sizes from [Modifying the Linker Script](#) (as well as the according base addresses) can also be modified by overriding the default values when invoking `make`. Therefore, the command needs to pass the according values to the linker using the makefile's `USER_FLAGS` variable.



See section "Application Makefile" of the data sheet for more information regarding the default makefile variables: [https://stnolting.github.io/neorv32/#\\_application\\_makefile](https://stnolting.github.io/neorv32/#_application_makefile)

*Listing 5. Example: override default RAM (DMEM) and ROM (IMEM) size while invoking make*

```
$ make USER_FLAGS+="-Wl,--defsym,__neorv32_ram_size=16k -Wl,--defsym,__neorv32_rom_size=32k" clean_all exe
```

The `-Wl` passes the following command/flag to the linker while `--defsym` defines a symbol for the linker. Hence, the default linker script section sizes are overridden. In this example the RAM size (=DMEM) is set to 16kB and the ROM size (=IMEM) is set to 32kB.



When using this approach the customized attributes have to be specified every time the makefile is invoked! You can put the RAM/ROM override commands into the project's local makefile or define a simple shell script that defines all the setup-related parameters (memory sizes, RISC-V ISA extensions, optimization goal, further tuning flags, etc.).

# Chapter 4. Application Program Compilation

This guide shows how to compile an example C-code application into a NEORV32 executable that can be uploaded via the bootloader or the on-chip debugger.

1. Open a terminal console and navigate to one of the project's example programs. For instance, navigate to the simple `sw/example_demo_blink_led` example program. This program uses the NEORV32 GPIO module to display an 8-bit counter on the lowest eight bit of the `gpio_o` output port.
2. To compile the project and generate an executable simply execute:

```
neorv32/sw/example/demo_blink_led$ make clean_all exe
```

3. The `clean_all` target is used (instead of just `clean`) to ensure everything is re-build.
4. The `exe` target will compile and link the application sources together with all the included libraries. At the end an ELF file (`main.elf`) is generated. The *NEORV32 image generator* (in `sw/image_gen`) takes this file and creates the final executable (`neorv32_exe.bin`). The makefile will show the resulting memory utilization and the executable size:

```
neorv32/sw/example/demo_blink_led$ make clean_all exe
Memory utilization:
  text  data  bss   dec   hex filename
  1004    0    0   1004   3ec main.elf
Compiling ../../sw/image_gen/image_gen
Executable (neorv32_exe.bin) size in bytes:
1016
```

## Build Artifacts



All *intermediate* build artifacts (like object files and binaries) will be placed into a (new) project-local folder named `build`. The *resulting* build artifacts (like executable, the main ELF and all memory initialization/image files) will be placed in the root project folder.

5. That's it. The `exe` target has created the actual executable `neorv32_exe.bin` in the current folder that is ready to be uploaded to the processor using the build-in bootloader. Alternatively, the ELF file can be uploaded using the on-chip debugger.

# Chapter 5. Uploading and Starting of a Binary Executable Image via UART

Follow this guide to use the bootloader to upload an executable via UART.



This concept uses the default "Indirect Boot" scenario that uses the bootloader to upload new executables. See datasheet section **Indirect Boot** for more information.



If your FPGA board does not provide such an interface - don't worry! Section **Installing an Executable Directly Into Memory** shows how to run custom programs on your FPGA setup without having a UART.

1. Connect the primary UART (UART0) interface of your FPGA board to a serial port of your host computer.
2. Start a terminal program. In this tutorial, I am using TeraTerm for Windows. You can download it for free from <https://ttssh2.osdn.jp/index.html.en> . On Linux you could use **cutecom** (recommended) or **GTKTerm**, which you can get here <https://github.com/Jeija/gtkterm.git> (or install via your package manager).



Any terminal program that can connect to a serial port should work. However, make sure the program can transfer data in *raw* byte mode without any protocol overhead around it. Some terminal programs struggle with transmitting files larger than 4kB (see <https://github.com/stnolting/neorv32/pull/215>). Try a different program if uploading a binary does not work (terminal stall).

3. Open a connection to the the serial port your UART is connected to. Configure the terminal setting according to the following parameters:
  - 19200 Baud
  - 8 data bits
  - 1 stop bit
  - no parity bits
  - *no* transmission/flow control protocol
  - newline on `\r\n` (carriage return **and** line feed)
4. Also make sure that single chars are send from your computer *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. In **cutecom**, change **LF** to **None** in the drop-down menu next to the input text box.
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.

*Listing 6. Bootloader console; aborted auto-boot sequence*

```
<< NEORV32 Bootloader >>

BLDV: Mar  7 2023
HWV:  0x01080107
CID:  0x00000000
CLK:  0x05f5e100
MISA: 0x40901106
XISA: 0xc0000fab
SOC:  0xffff402f
IMEM: 0x00008000
DMEM: 0x00002000

Autoboot in 10s. Press any key to abort.
Aborted.

Available CMDs:
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 send a NEORV32 executable (**neorv32\_exe.bin**).
8. Again, make sure to transmit the executable in raw binary mode (no transfer protocol). When using TeraTerm, select the "binary" option in the send file dialog.
9. If everything went fine, OK will appear in your terminal:



Make sure to upload the NEORV32 executable **neorv32\_exe.bin**. Uploading any other file (like **main.bin**) will cause an **ERR\_EXE** bootloader error (see [https://stnolting.github.io/neorv32/#\\_bootloader\\_error\\_codes](https://stnolting.github.io/neorv32/#_bootloader_error_codes)).

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

10. The executable is now in the instruction memory of the processor. To execute the program right now run the "Execute" command by typing **e**:

```
CMD:> u
Awaiting neorv32_exe.bin... OK
CMD:> e
Booting...
Blinking LED demo program
```

11. If everything went fine, you should see the LEDs blinking.



The bootloader will print error codes if something went wrong. See section **Bootloader** of the NEORV32 datasheet for more information.



See section **Programming an External SPI Flash via the Bootloader** to learn how to use an external SPI flash for nonvolatile program storage.



The bootloader also supports booting from external TWI memory. Enable it in the bootloader makefile, but be careful, enabling all features may result in a too-big binary.



Executables can also be uploaded via the **on-chip debugger**. See section **Debugging with GDB** for more information.



## Chapter 6. Installing an Executable Directly Into Memory

If you do not want to use the bootloader (or the on-chip debugger) for executable upload or if your setup does not provide a serial interface for that, you can also directly install an application into embedded memory.

This concept uses the "Direct Boot" scenario that implements the processor-internal IMEM as ROM, which is pre-initialized with the application's executable during synthesis. Hence, it provides *non-volatile* storage of the executable inside the processor. This storage cannot be altered during runtime and any source code modification of the application requires to re-program the FPGA via the bitstream.



See datasheet section **Direct Boot** for more information.

Using the IMEM as ROM:

- for this boot concept the bootloader is no longer required
  - this concept only works for the internal IMEM (but can be extended to work with external memories coupled via the processor's bus interface)
  - make sure that the memory components (like block RAM) the IMEM is mapped to support an initialization via the bitstream
1. At first, make sure your processor setup actually implements the internal IMEM: the `MEM_INT_IMEM_EN` generics has to be set to `true`:

*Listing 7. Processor top entity configuration - enable internal IMEM*

```
-- Internal Instruction memory --  
MEM_INT_IMEM_EN => true, -- implement processor-internal instruction memory
```

2. For this setup we do not want the bootloader to be implemented at all. Disable implementation of the bootloader by setting the `INT_BOOTLOADER_EN` generic to `false`. This will also modify the processor-internal IMEM so it is initialized with the executable during synthesis.

*Listing 8. Processor top entity configuration - disable internal bootloader*

```
-- General --  
INT_BOOTLOADER_EN => false, -- boot configuration: false = boot from int/ext (I)MEM
```

3. To generate an "initialization image" for the IMEM that contains the actual application, run the `install` target when compiling your application:

```
neorv32/sw/example/demo_blink_led$ make clean_all install
```

```
Memory utilization:
```

text	data	bss	dec	hex	filename
1004	0	0	1004	3ec	main.elf

```
Compiling ../../sw/image_gen/image_gen
```

```
Executable (neorv32_exe.bin) size in bytes:
```

```
1016
```

```
Installing application image to ../../rtl/core/neorv32_application_image.vhd
```

4. The **install** target has compiled all the application sources but instead of creating an executable (**neorv32\_exe.bit**) that can be uploaded via the bootloader, it has created a VHDL memory initialization image **core/neorv32\_application\_image.vhd**.
5. This VHDL file is automatically copied to the core's rtl folder (**rtl/core**) so it will be included for the next synthesis.
6. Perform a new synthesis. The IMEM will be build as pre-initialized ROM (inferring embedded memories if possible).
7. Upload your bitstream. Your application code now resides unchangeable in the processor's IMEM and is directly executed after reset.

The synthesis tool / simulator will print asserts to inform about the (IMEM) memory / boot configuration:

```
NEORV32 PROCESSOR CONFIG NOTE: Boot configuration: Direct boot from memory (processor-internal IMEM).
```

```
NEORV32 PROCESSOR CONFIG NOTE: Implementing processor-internal IMEM as ROM (1016 bytes), pre-initialized with application.
```

# Chapter 7. Setup of a New Application Program Project

1. The easiest way of creating a *new* software application project is to copy an *existing* one. This will keep all file dependencies. For example you can copy `sw/example/demo_blink_led` to `sw/example/flux_capacitor`.
2. If you want to place you application somewhere outside `sw/example` you need to adapt the application's makefile. In the makefile you will find a variable that keeps the relative or absolute path to the NEORV32 repository home folder. Just modify this variable according to your new project's home 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 can add a folder containing your application's include files 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
```

# Chapter 8. Application-Specific Processor Configuration

The processor's configuration options, which are mainly defined via the top entity VHDL generics, allow to tailor the entire SoC according to the application-specific requirements. Note that this chapter does not focus on optional *SoC features* like IO/peripheral modules - it rather gives ideas on how to optimize for *overall goals* like performance and area.



Please keep in mind that optimizing the design in one direction (like performance) will also effect other potential optimization goals (like area and energy).

## 8.1. Optimize for Performance

The following points show some concepts to optimize the processor for performance regardless of the costs (i.e. increasing area and energy requirements):

- Enable all performance-related RISC-V CPU extensions that implement dedicated hardware accelerators instead of emulating operations entirely in software: `M`, `C`, `Zfinx`
- Enable mapping of complex CPU operations to dedicated hardware: `FAST_MUL_EN`  $\Rightarrow$  `true` to use DSP slices for multiplications, `FAST_SHIFT_EN`  $\Rightarrow$  `true` use a fast barrel shifter for shift operations.
- Implement the instruction cache: `ICACHE_EN`  $\Rightarrow$  `true`
- Use as many *internal* memory as possible to reduce memory access latency: `MEM_INT_IMEM_EN`  $\Rightarrow$  `true` and `MEM_INT_DMEM_EN`  $\Rightarrow$  `true`, maximize `MEM_INT_IMEM_SIZE` and `MEM_INT_DMEM_SIZE`
- *To be continued...*

## 8.2. Optimize for Size

The NEORV32 is a size-optimized processor system that is intended to fit into tiny niches within large SoC designs or to be used a customized microcontroller in really tiny / low-power FPGAs (like Lattice iCE40). Here are some ideas how to make the processor even smaller while maintaining it's *general purpose system* concept and maximum RISC-V compatibility.

### SoC

- This is obvious, but exclude all unused optional IO/peripheral modules from synthesis via the processor configuration generics.
- If an IO module provides an option to configure the number of "channels", constrain this number to the actually required value (e.g. the PWM module's `IO_PWM_NUM_CH` generic).
- Disable the instruction cache (`ICACHE_EN`  $\Rightarrow$  `false`) if the design only uses processor-internal IMEM and DMEM memories.
- *To be continued...*

## CPU

- Use the *embedded* RISC-V CPU architecture extension (`CPU_EXTENSION_RISCV_E`) to reduce block RAM utilization.
- The compressed instructions extension (`CPU_EXTENSION_RISCV_C`) requires additional logic for the decoder but also reduces program code size by approximately 30%.
- If not explicitly used/required, exclude the CPU standard counters `[m]instret[h]` (number of instruction) and `[m]cycle[h]` (number of cycles) from synthesis by disabling the `Zicntr` ISA extension (note, this is not RISC-V compliant).
- Map CPU shift operations to a small and iterative shifter unit (`FAST_SHIFT_EN`  $\Rightarrow$  `false`).
- If you have unused DSP block available, you can map multiplication operations to those slices instead of using LUTs to implement the multiplier (`FAST_MUL_EN`  $\Rightarrow$  `true`).
- If there is no need to execute division in hardware, use the `Zmmul` extension instead of the full-scale `M` extension.
- Disable CPU extension that are not explicitly used.
- *To be continued...*

## 8.3. Optimize for Clock Speed

The NEORV32 Processor and CPU are designed to provide minimal logic between register stages to keep the critical path as short as possible. When enabling additional extension or modules the impact on the existing logic is also kept at a minimum to prevent timing degrading. If there is a major impact on existing logic (example: many physical memory protection address configuration registers) the VHDL code automatically adds additional register stages to maintain critical path length. Obviously, this increases operation latency.



Enable the "ASIC style" / full-reset register file option (`REGFILE_HW_RST`) to obtain maximum clock speed for the CPU (at the cost of an increased hardware footprint).

In order to optimize for a minimal critical path (= maximum clock speed) the following points should be considered:

- Complex CPU extensions (in terms of hardware requirements) should be avoided (examples: floating-point unit, physical memory protection).
- Large carry chains (>32-bit) should be avoided (i.e. constrain the HPM counter sizes via `HPM_CNT_WIDTH`).
- If the target FPGA provides sufficient DSP resources, CPU multiplication operations can be mapped to DSP slices (`FAST_MUL_EN`  $\Rightarrow$  `true`) reducing LUT usage and critical path impact while also increasing overall performance.
- Use the synchronous (registered) RX path configuration of the external bus interface (`XBUS_ASYNC_RX`  $\Rightarrow$  `false`).
- *To be continued...*



The short and fixed-length critical path allows to integrate the core into existing clock domains. So no clock domain-crossing and no sub-clock generation is required. However, for very high clock frequencies (this is technology / platform dependent) clock domain crossing becomes crucial for chip-internal connections.

## 8.4. Optimize for Energy

There are no *dedicated* configuration options to optimize the processor for energy (minimal consumption; energy/instruction ratio) yet. However, a reduced processor area (**Optimize for Size**) will also reduce static energy consumption.

To optimize your setup for low-power applications, you can make use of the CPU sleep mode (**wfi** instruction). Put the CPU to sleep mode whenever possible. If the clock gating feature is enabled (**CLOCK\_GATING\_EN**) the entire CPU complex will be disconnected from the clock tree while in sleep mode.

Disable all processor modules that are not actually used (exclude them from synthesis if they will be *never* used; disable a module via it's control register if the module is not *currently* used).



### *Processor-internal clock generator shutdown*

If *no* IO/peripheral module is currently enabled, the processor's internal clock generator circuit will be shut down reducing switching activity and thus, dynamic energy consumption.

# Chapter 9. Adding Custom Hardware Modules

In resemblance to the RISC-V ISA, the NEORV32 processor was designed to ease customization and *extensibility*. The processor provides several predefined options to add application-specific custom hardware modules and accelerators. A **Comparative Summary** is given at the end of this section.



## *Debugging/Testing Custom Hardware Modules*

Custom hardware IP modules connected via the external bus interface or integrated as CFU can be debugged "in-system" using the "bus explorer" example program (`sw/example_bus_explorer`). This program provides an interactive console (via UART0) that allows to perform arbitrary read and write access from/to any memory-mapped register.

## 9.1. Standard (*External*) Interfaces

The processor already provides a set of standard interfaces that are intended to connect *chip-external* devices. However, these interfaces can also be used chip-internally. The most suitable interfaces are **GPIO**, **UART**, **SPI** and **TWI**.

The SPI and especially the GPIO interfaces might be the most straightforward approaches since they have a minimal protocol overhead. Beyond simplicity, these interface only provide a very limited bandwidth and require more sophisticated software handling ("bit-banging" for the GPIO). Hence, it is not recommend to use them for *chip-internal* communication.

## 9.2. External Bus Interface

The **External Bus Interface** provides the classic approach for attaching custom IP. By default, the bus interface implements the widely adopted Wishbone interface standard. This project also includes wrappers to convert to other protocol standards like ARM's AXI4-Lite or Intel's Avalon protocols. By using a full-featured bus protocol, complex SoC designs can be implemented including several modules and even multi-core architectures. Many FPGA EDA tools provide graphical editors to build and customize whole SoC architectures and even include pre-defined IP libraries.

Custom hardware modules attached to the processor's bus interface have no limitations regarding their functionality. User-defined interfaces (like DDR memory access) can be implemented and the hardware module can operate completely independent of the CPU.

The bus interface uses a memory-mapped approach. All data transfers are handled by simple load/store operations since the external bus interface is mapped into the processor's **address space**. This allows a very simple still high-bandwidth communications. However, high bus traffic may increase access latencies.

## 9.3. Custom Functions Subsystem

The **Custom Functions Subsystem (CFS)** is an "empty" template for a memory-mapped, processor-internal module.

The basic idea of this subsystem is to provide a convenient, simple and flexible platform, where the user can concentrate on implementing the actual design logic rather than taking care of the communication between the CPU/software and the design logic. Note that the CFS does not have direct access to memory. All data (and control instruction) have to be send by the CPU.

The use-cases for the CFS include medium-scale hardware accelerators that need to be tightly-coupled to the CPU. Potential use cases could be DSP modules like CORDIC, cryptographic accelerators or custom interfaces (like IIS).

## 9.4. Custom Functions Unit

The **Custom Functions Unit (CFU)** is a functional unit that is integrated right into the CPU's pipeline. It allows to implement custom RISC-V instructions. This extension option is intended for rather small logic that implements operations, which cannot be emulated in pure software in an efficient way. Since the CFU has direct access to the core's register file it can operate with minimal data latency.

## 9.5. Comparative Summary

The following table gives a comparative summary of the most important factors when choosing one of the chip-internal extension options:

- **Custom Functions Unit (CFU)** for CPU-internal custom RISC-V instructions
- **Custom Functions Subsystem (CFS)** for tightly-coupled processor-internal co-processors
- **External Bus Interface (WISHBONE)** for processor-external memory-mapped modules

*Table 1. Comparison of On-Chip Extension Options*

	<b>Custom Functions Unit (CFU)</b>	<b>Custom Functions Subsystem (CFS)</b>	<b>External Bus Interface</b>
<b>RTL location</b>	CPU-internal	processor-internal	processor-external
<b>HW complexity/size</b>	small	medium	large
<b>CPU-independent operation</b>	no	yes	yes
<b>CPU interface</b>	register file access	memory-mapped	memory-mapped
<b>Low-level access mechanism</b>	custom instructions	load/store	load/store
<b>Access latency</b>	minimal	low	medium to high



	Custom Functions Unit (CFU)	Custom Functions Subsystem (CFS)	External Bus Interface
<b>External IO interfaces</b>	not supported	yes, but limited	yes, user-defined
<b>Exception capability</b>	yes	no	no
<b>Interrupt capability</b>	no	yes	user-defined

# Chapter 10. Customizing the Internal Bootloader

The NEORV32 bootloader provides several options to configure and customize it for a certain application setup. This configuration is done by passing *defines* when compiling the bootloader. Of course you can also modify the bootloader source code to provide a setup that perfectly fits your needs.



Each time the bootloader sources are modified, the bootloader has to be re-compiled (and re-installed to the bootloader ROM) and the processor has to be re-synthesized.



Keep in mind that the maximum size for the bootloader is limited to 8kB and it should be compiled using the minimal base & privileged ISA `rv32e_zicsr_zifencei` only to ensure it can work with any actual CPU configuration.

Table 2. Bootloader configuration parameters

Parameter	Default	Legal values	Description
Memory layout			
<code>EXE_BASE_ADDR</code>	<code>0x00000000</code>	<i>any</i>	Base address / boot address for the executable (see section "Address Space" in the NEORV32 data sheet)
Serial console interface			
<code>UART_EN</code>	<code>1</code>	<code>0, 1</code>	Set to <code>0</code> to disable UART0 (no serial console at all)
<code>UART_BAUD</code>	<code>19200</code>	<i>any</i>	Baud rate of UART0
<code>UART_HW_HANDSHAKE_EN</code>	<code>0</code>	<code>0, 1</code>	Set to <code>1</code> to enable UART0 hardware flow control
Status LED			
<code>STATUS_LED_EN</code>	<code>1</code>	<code>0, 1</code>	Enable bootloader status led ("heart beat") at <code>GPIO</code> output port pin <code>#STATUS_LED_PIN</code> when <code>1</code>
<code>STATUS_LED_PIN</code>	<code>0</code>	<code>0 ... 31</code>	<code>GPIO</code> output pin used for the high-active status LED
Auto-boot configuration			
<code>AUTO_BOOT_TIMEOUT</code>	<code>10</code>	<i>any</i>	Time in seconds after the auto-boot sequence starts (if there is no UART input by the user); set to <code>0</code> to disabled auto-boot sequence
SPI configuration			
<code>SPI_EN</code>	<code>1</code>	<code>0, 1</code>	Set <code>1</code> to enable the usage of the SPI module (including load/store executables from/to SPI flash options)
<code>SPI_FLASH_CS</code>	<code>0</code>	<code>0 ... 7</code>	SPI chip select output ( <code>spi_csn_o</code> ) for selecting flash

Parameter	Default	Legal values	Description
<code>SPI_FLASH_ADDR_BYTES</code>	3	2, 3, 4	SPI flash address size in number of bytes (2=16-bit, 3=24-bit, 4=32-bit)
<code>SPI_FLASH_SECTOR_SIZE</code>	65536	<i>any</i>	SPI flash sector size in bytes
<code>SPI_FLASH_CLK_PRSC</code>	<code>CLK_PRSC_8</code>	<code>CLK_PRSC_2</code> <code>CLK_PRSC_4</code> <code>CLK_PRSC_8</code> <code>CLK_PRSC_64</code> <code>CLK_PRSC_128</code> <code>CLK_PRSC_1024</code> <code>CLK_PRSC_2024</code> <code>CLK_PRSC_4096</code>	SPI clock pre-scaler (dividing main processor clock)
<code>SPI_BOOT_BASE_ADDR</code>	<code>0x00400000</code>	<i>any</i> 32-bit value	Defines the <i>base</i> address of the executable in external flash
TWI configuration			
<code>TWI_EN</code>	0	0, 1	Set 1 to enable the usage of the TWI module (including load executables from TWI device option)
<code>TWI_CLK_PRSC</code>	<code>CLK_PRSC_64</code>	<code>CLK_PRSC_2</code> <code>CLK_PRSC_4</code> <code>CLK_PRSC_8</code> <code>CLK_PRSC_64</code> <code>CLK_PRSC_128</code> <code>CLK_PRSC_1024</code> <code>CLK_PRSC_2024</code> <code>CLK_PRSC_4096</code>	TWI clock pre-scaler (dividing main processor clock)
<code>TWI_CLK_DIV</code>	3	0 ... 15	TWI clock divider (dividing twi clock)
<code>TWI_DEVICE_ID</code>	<code>0x50</code>	<code>0x00</code> ... <code>0x7F</code>	First TWI device ID to start. Is incremented until the end of the program is reached, when <code>TWI_ADDR_BYTES</code> is 1.
<code>TWI_ADDR_BYTES</code>	1	1, 2	TWI memory address size in number of bytes. When <code>TWI_ADDR_BYTES</code> is 1, <code>TWI_DEVICE_ID</code> the gets incremented as well.



Enabling all features while sticking to the minimal RISC-V ISA will result in a too-large binary!

Each configuration parameter is implemented as C-language `define` that can be manually overridden (*redefined*) when invoking the bootloader's makefile. The according parameter and its new value has to be *appended* (using `+=`) to the makefile `USER_FLAGS` variable. Make sure to use the `-D` prefix here. The configuration is also listed in the makefile of the bootloader.

For example, to configure a UART Baud rate of 57600 and redirecting the status LED to GPIO output

pin 20 use the following command:

*Listing 9. Example: customizing, re-compiling and re-installing the bootloader*

```
sw/bootloader$ make USER_FLAGS+=-DUART_BAUD=57600 USER_FLAGS+=-DSTATUS_LED_PIN=20  
clean_all bootloader
```



The `clean_all` target ensure that all libraries are re-compiled. The `bootloader` target will automatically compile and install the bootloader to the HDL boot ROM (updating `rtl/core/neorv32_bootloader_image.vhd`).

## 10.1. Auto-Boot Configuration

The default bootloader provides a UART-based user interface that allows to upload new executables at any time. Optionally, the executable can also be programmed to an external SPI flash by the bootloader (see section [Programming an External SPI Flash via the Bootloader](#)).

The bootloader also provides an *automatic boot sequence* (auto-boot) which will start copying an executable from external SPI flash to IMEM using the default SPI configuration. By this, the default bootloader provides a "non-volatile program storage" mechanism that automatically boots from external SPI flash (after `AUTO_BOOT_TIMEOUT`) while still providing the option to re-program the SPI flash at any time via the UART console.

# Chapter 11. Programming an External SPI Flash via the Bootloader

The default processor-internal NEORV32 bootloader supports automatic booting from an external SPI flash. This guide shows how to write an executable to the SPI flash via the bootloader so it can be automatically fetched and executed after processor reset. For example, you can use a section of the FPGA bitstream configuration memory to store an application executable.



## Customization

This section assumes the *default* configuration of the NEORV32 bootloader. See section [Customizing the Internal Bootloader](#) on how to customize the bootloader and its setting (for example the SPI chip-select port, the SPI clock speed or the **flash base address** for storing the executable).

## 11.1. Programming an Executable

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 executable 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 upload is completed and "OK" appears, press **s** 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:> s
Write 0x000013FC bytes to SPI flash @ 0x02000000? (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.



Section [Customizing the Internal Bootloader](#) show the according C-language **define** that can be modified to specify the base address of the executable inside the SPI flash.

```
CMD:> u
Awaiting neorv32_exe.bin... OK
CMD:> s
Write 0x000013FC bytes to SPI flash @ 0x02000000? (y/n) y
Flashing... OK
CMD:>
```



The bootloader stores the executable in **little-endian** byte-order to the flash.

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.

# Chapter 12. Packaging the Processor as Vivado IP Block

Packaging the entire processor as IP module allows easy integration of the core (or even several cores) into a block-design-based Vivado project. The NEORV32 repository provides a full-scale TCL script that automatically packages the processor as Vivado IP block. For this, a specialized wrapper for the processor's top entity is provided (`rtl/system_integration/neorv32_vivado_ip.vhd`) that features AXI4-Lite (via XBUS) and AXI4-Stream (via SLINK) compatible interfaces.



## General AXI Wrapper

The provided AXI wrapper can also be used for custom (AXI) setups outside of Vivado and/or IP block designs.

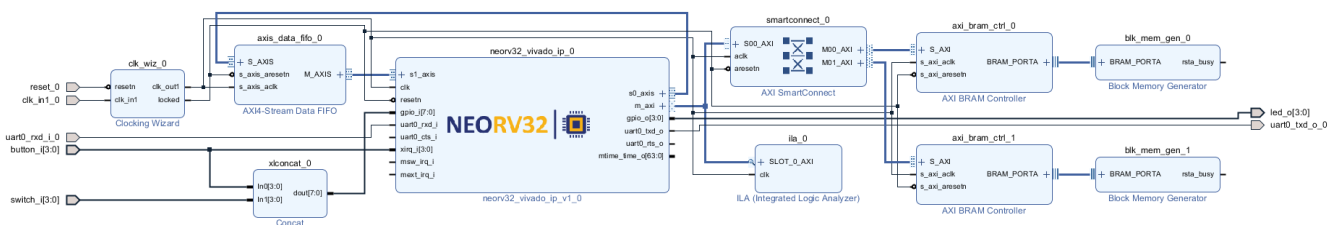


Figure 2. Example Vivado SoC using the NEORV32 IP Block

Besides packaging the HDL modules, the TCL script also generates a simplified customization GUI that allows an easy and intuitive configuration of the processor. The rather complex VHDL configuration generics are renamed and provided with tool tips to make it easier to understand the different configuration options.

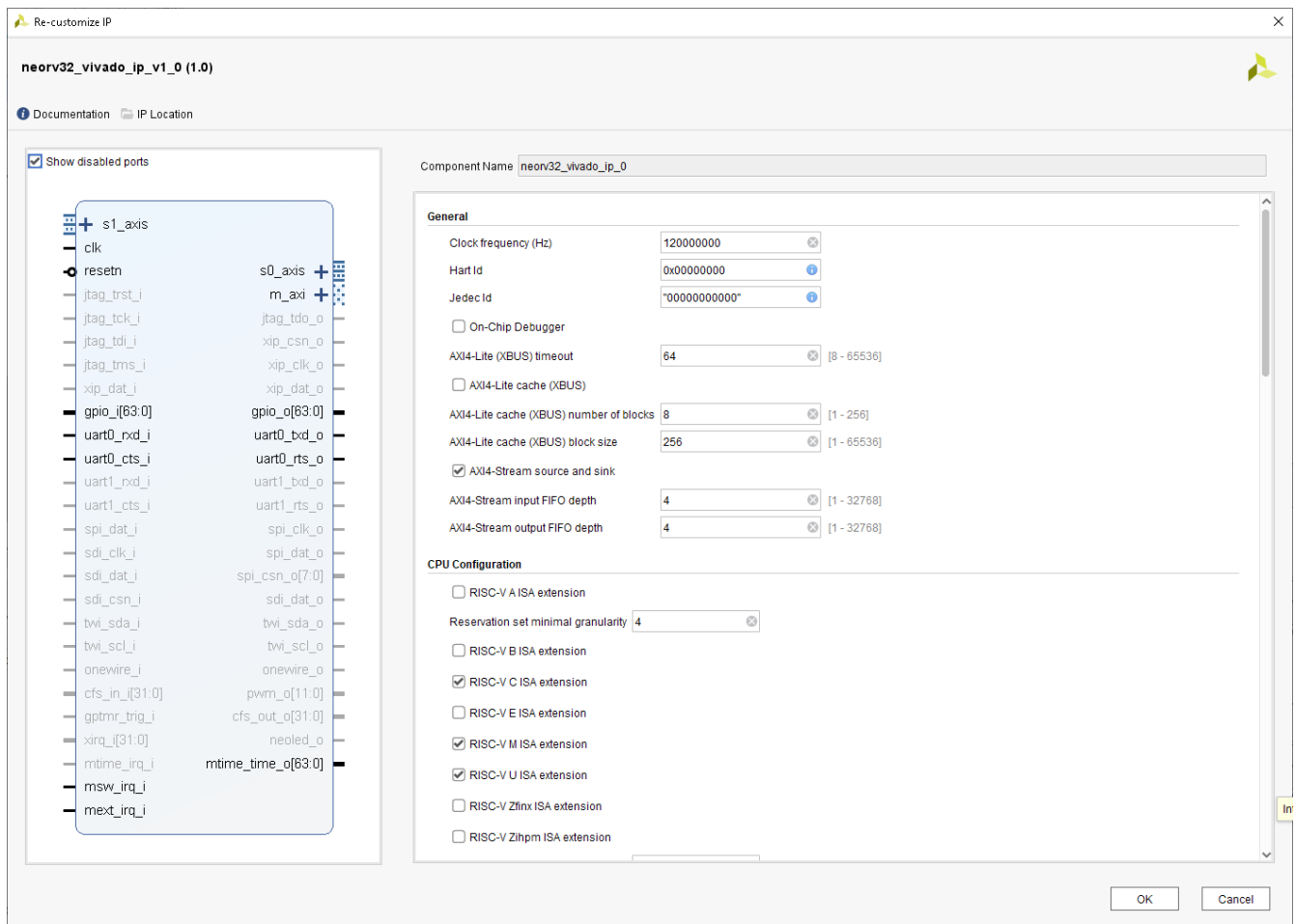


Figure 3. NEORV32 IP Customization GUI

The following steps show how to package the processor using the provided TCL script and how to import the generated IP block into the Vivado IP repository.

1. Open the Vivado GUI.
2. In GUI mode select either "Tools → Run TCL Script" to directly execute the script or open the TCL shell ("Window → Tcl Console") to manually invoke the script.
3. Use `cd` in the TCL console to navigate to the project's `neorv32/rtl/system_integration` folder.
4. Execute `source neorv32_vivado_ip.tcl` in the TCL console.
5. A second Vivado instance will open automatically packaging the IP module. After this process is completed, the second Vivado instance will automatically close again.
6. A new folder `neorv32_vivado_ip_work` is created in `neorv32/rtl/system_integration` which contains the IP-packaging Vivado project.
7. Inside, the `packaged_ip` folder provides the actual IP module.
8. Open your design project in Vivado.
9. Click on "Settings" in the "Project Manager" on the left side.
10. Under "Project Settings" expand the "IP" section and click on "Repository".
11. Click the large plus button and select the previously generated IP folder (`path/to/neorv32/rtl/system_integration/neorv32_vivado_ip_work/packaged_ip`).



12. Click "Select" and close the Settings menu with "Apply" and "OK".
13. You will find the NEORV32 in the "User Repository" section of the Vivado IP catalog.



#### *Combinatorial Loops DRC Errors*

If the TRNG is enabled it is recommended to add the following commands to the project's constraints file in order to prevent DRC errors during bitstream generation.

#### *Listing 10. Allow Combinatorial Loops*

```
set_property SEVERITY {warning} [get_drc_checks LUTLP-1]
set_property IS_ENABLED FALSE [get_drc_checks LUTLP-1]
set_property ALLOW_COMBINATORIAL_LOOPS TRUE
```



#### *Re-Packaging the IP Core*

For every change that is made right to the hardware (excluding configuration made via the customization GUI!) the NEORV32 IP module needs to be re-packaged by re-executing the packing script (`neorv32_vivado_ip.tcl`).

This also applies if an executable installed right into the IMEM (see section [Installing an Executable Directly Into Memory](#)) shall be updated. It is **not** possible to replace the IMEM image (`neorv32_application_image.vhd`) file in the `packaged_ip` folder. For the Vivado design suite, the new program to be executed must be compiled and installed using the `install` makefile target. Next, the `neorv32_vivado_ip.tcl` script has to be executed again. Finally, Vivado will prompt to upgrade the NEORV32 IP.



#### *AMD Vivado / ISIM - Incremental Compilation of Simulation Sources*

When using AMD Vivado (ISIM for simulation) make sure to **TURN OFF** "incremental compilation" (*Project Setting* → *Simulation* → *Advanced* → *\_Enable incremental compilation*). This will slow down simulation relaunch but will ensure that all application images (`*_image.vhd`) are reanalyzed when recompiling the NEORV32 application or bootloader.

# Chapter 13. Simulating the Processor

Due to the complexity of the NEORV32 processor and all the different configuration options, there is a wide range of possible testing and verification strategies.

On the one hand, a simple smoke testbench allows ensuring that functionality is correct from a software point of view. That is used for running the RISC-V architecture tests, in order to guarantee compliance with the ISA specification(s). All required simulation sources are located in **sim**.

On the other hand, **VUnit** and **Verification Components** are used for verifying the functionality of the various peripherals from a hardware point of view.

## *AMD Vivado / ISIM - Incremental Compilation*



When using AMD Vivado (ISIM for simulation) make sure to **TURN OFF** "incremental compilation" (*Project Setting* → *Simulation* → *Advanced* → *\_Enable incremental compilation*). This will slow down simulation relaunch but will ensure that all application images (*\*\_image.vhd*) are reanalyzed when recompiling the NEORV32 application or bootloader.

## 13.1. Testbench



### *VUnit Testbench*

A more sophisticated testbench using **VUnit** is available in a separate repository: <https://github.com/stnolting/neorv32-vunit>

A plain-VHDL testbench without any third-party libraries / dependencies (*sim/neorv32\_tb.vhd*) can be used for simulating and testing the processor and all its configurations. This testbench features clock and reset generators and enables all optional peripheral and CPU extensions. The processor check program (*sw/example/processor\_check*) is develop in close relation to the default testbench in order to test all primary processor functions.

The simulation setup is configured via the "User Configuration" section located right at the beginning of the testbench architecture. Each configuration generic provides a default value and a comments to explain the functionality. Basically, these configuration generics represent most of the processor's **top generics**.

### *UART output during simulation*



Data written to the NEORV32 UART0 / UART1 transmitter is send to a virtual UART receiver implemented as part of the default testbench. The received chars are send to the simulator console and are also stored to a log file (*neorv32\_tb.uart0\_rx.out* for UART0, *neorv32\_tb.uart1\_rx.out* for UART1) inside the simulator's home folder. **Please note that printing via the native UART receiver takes a lot of time.** For faster simulation console output see section **Faster Simulation Console Output**.

## 13.2. Faster Simulation Console Output

When printing data via the physical UART the communication speed will always be based on the configured BAUD rate. For a simulation this might take some time. To have faster output you can enable the **simulation mode** for UART0/UART1 (see section [https://stnolting.github.io/neorv32/#\\_primary\\_universal\\_asynchronous\\_receiver\\_and\\_transmitter\\_uart0](https://stnolting.github.io/neorv32/#_primary_universal_asynchronous_receiver_and_transmitter_uart0)).

ASCII data sent to UART0 / UART1 will be immediately printed to the simulator console and logged to files in the simulator's home directory.

- `neorv32.uart0_sim_mode.out`: ASCII data send via UART0
- `neorv32.uart1_sim_mode.out`: ASCII data send via UART1



### Automatic Simulation Mode

You can "automatically" enable the simulation mode of UART0/UART1 when compiling an application. In this case, the "real" UART0/UART1 transmitter unit is permanently disabled by setting the UART's "sim-mode" bit. To enable the simulation mode just compile and install the application and add `-DUART0_SIM_MODE` `-DUART0_SIM_MODE` / `-DUART1_SIM_MODE` to the compiler's `USER_FLAGS` variable (do not forget the `-D` suffix flag):

*Listing 11. Auto-Enable UART0 Simulation-Mode while Compiling*

```
sw/example/demo_blink_led$ make USER_FLAGS+=-DUART0_SIM_MODE clean_all all
```

## 13.3. GHDL Simulation

The default simulation setup that is also used by the project's CI pipeline is based on the free and open-source VHDL simulator **GHDL**. The `sim` folder also contains a simple script that evaluates and simulates all core files. This script can be called right from the command. Optionally, additional GHDL flags can be passes.

*Listing 12. Invoking the default GHDL simulation script*

```
neorv32/sim$ sh ghdl.sh --stop-time=20ms
```

## 13.4. Simulation using Application Makefiles

The **GHDL Simulation** can also be started by the main application makefile (i.e. from each SW project folder).

*Listing 13. Starting the GHDL simulation from the application makefile*

```
sw/example/demo_blink_led$ make USER_FLAGS+=-DUART0_SIM_MODE clean_all install sim
[...]
```

## Blinking LED demo program

Makefile targets:

- **clean\_all**: delete all artifacts and rebuild everything
- **install**: install executable
- **sim**: run GHDL simulation



#### Adjusting the Testbench Configuration

The testbench provides several generics for customization. These can be adjusted in-console using the makefile's **GHDL\_RUN\_FLAGS** variable. E.g.: **make GHDL\_RUN\_FLAGS+="-gBOOT\_MODE\_SELECT=1" sim**

### 13.4.1. Hello World!

To do a quick test of the NEORV32 make and the required tools navigate to the project's **sw/example/hello\_world** folder and run **make USER\_FLAGS+="-DUART0\_SIM\_MODE" clean\_all install sim**:

```
neorv32/sw/example/hello_world$ make USER_FLAGS+="-DUART0_SIM_MODE" clean_all install
sim
../../sw/lib/source/neorv32_uart.c: In function 'neorv32_uart_setup':
../../sw/lib/source/neorv32_uart.c:109:2: warning: #warning UART0_SIM_MODE (primary
UART) enabled! Sending all UART0.TX data to text.io simulation output instead of real
UART0 transmitter. Use this for simulation only! [-Wcpp]
  109 | #warning UART0_SIM_MODE (primary UART) enabled! Sending all UART0.TX data to
text.io simulation output instead of real UART0 transmitter. Use this for simulation
only! ❶
      | ~~~~~~
Memory utilization:
   text   data   bss    dec    hex filename
   5540     0    116   5656   1618 main.elf ❷
Compiling image generator...
Generating neorv32_application_image.vhd
Installing application image to ../../rtl/core/neorv32_application_image.vhd ❸
Simulating processor using default testbench...
GHDL simulation run parameters: --stop-time=10ms ❹
../rtl/core/neorv32_top.vhd:351:5:@0ms:(assertion note): [NEORV32] The NEORV32 RISC-V
Processor (v1.10.7.6), github.com/stnolting/neorv32
../rtl/core/neorv32_top.vhd:357:5:@0ms:(assertion note): [NEORV32] Processor
Configuration: CPU IMEM-ROM DMEM I-CACHE D-CACHE XBUS XBUS-CACHE CLINT GPIO UART0
UART1 SPI SDI TWI TWD PWM WDT TRNG CFS NEOLED XIRQ GPTMR ONEWIRE DMA SLINK CRC SYSINFO
OCD-AUTH
../rtl/core/neorv32_top.vhd:411:5:@0ms:(assertion note): [NEORV32] BOOT_MODE_SELECT =
2: booting IMEM image
../rtl/core/neorv32_clockgate.vhd:38:3:@0ms:(assertion warning): [NEORV32] Clock
gating enabled (using default/generic clock switch).
```

```

../rtl/core/neorv32_cpu.vhd:135:3:@0ms:(assertion note): [NEORV32] CPU ISA:
rv32ibmux_zalrsc_zba_zbb_zbkb_zbkc_zbkx_zbs_zicntr_zicond_zicsr_zifencei_zihpm_zfinx_z
kn_zknd_zkne_zknh_zks_zksed_zksh_zkt_zmmul_zxcfu_sdext_sdtrig_smpmp
../rtl/core/neorv32_cpu.vhd:171:3:@0ms:(assertion note): [NEORV32] CPU tuning options:
fast_mul fast_shift rf_hw_rst
../rtl/core/neorv32_cpu.vhd:178:3:@0ms:(assertion warning): [NEORV32] Assuming this is
a simulation.
../rtl/core/neorv32_imem.vhd:59:3:@0ms:(assertion note): [NEORV32] Implementing
processor-internal IMEM as pre-initialized ROM.
../rtl/core/neorv32_trng.vhd:277:3:@0ms:(assertion note): [neoTRNG] The neoTRNG (v3.2)
- A Tiny and Platform-Independent True Random Number Generator,
https://github.com/stnolting/neoTRNG
../rtl/core/neorv32_trng.vhd:284:3:@0ms:(assertion warning): [neoTRNG] Simulation-mode
enabled (NO TRUE/PHYSICAL RANDOM)!
../rtl/core/neorv32_debug_auth.vhd:48:3:@0ms:(assertion warning): [NEORV32] using
DEFAULT on-chip debugger authenticator. Replace by custom module.

```

⑤

```

##      ##  ##  ##
##      ##  #####  #####  #####  ##      ##  #####  #####
##      #####
####    ##  ##      ##      ##  ##      ##  ##      ##  ##      ##
##      ####      ####      ##  ##      ##  ##      ##      ##
##      ##  #####  ##
##  ##  ##  #####  ##      ##  #####  ##      ##      #####  ##
##      #####  #####  ####
##  ##  ##  ##      ##      ##  ##      ##      ##      ##      ##
##      ##  #####  ##
##      #####  ##      ##  ##      ##      ##  ##      ##      ##
##      #####  ####
##      ##  #####  #####  ##      ##      ##      #####  #####
##      #####
##      ##  ##  ##
Hello world! :)

```

- ① Notifier that "simulation mode" of UART0 is enabled (by the `USER_FLAGS+--DUART0_SIM_MODE` makefile flag). All UART0 output is send to the simulator console.
- ② Final executable size (**text**) and *static* data memory requirements (**data**, **bss**).
- ③ The application code is *installed* as pre-initialized IMEM. This is the default approach for simulation.
- ④ List of (default) arguments that were send to the simulator. Here: maximum simulation time (10ms).
- ⑤ Execution of the actual program starts. UART0 TX data is printed right to the console.

## Chapter 14. Building the Documentation

The documentation (datasheet + user guide) is written using **asciidoc**. The according source files can be found in **docs/...**. The documentation of the software framework is written *in-code* using **doxygen**.

A makefiles in the project's **docs** directory is provided to build all of the documentation as HTML pages or as PDF documents.



Pre-rendered PDFs are available online as *nightly pre-releases*: <https://github.com/stnolting/neorv32/releases>. The HTML-based documentation is also available online at the project's **GitHub Pages**.

The makefile provides a help target to show all available build options and their according outputs.

```
neorv32/docs$ make help
```

*Listing 14. Example: Generate HTML documentation (data sheet) using **asciidoc***

```
neorv32/docs$ make html
```



If you don't have **asciidoc** / **asciidoc-pdf** installed, you can still generate all the documentation using a *docker container* via **make container**.

# Chapter 15. Zephyr RTOS Support

The NEORV32 processor is supported by upstream Zephyr RTOS: <https://docs.zephyrproject.org/latest/boards/others/neorv32/doc/index.html>



The absolute path to the NEORV32 executable image generator binary (`.../neorv32/sw/image_gen`) has to be added to the `PATH` variable so the Zephyr build system can generate executables and memory-initialization images.



Zephyr OS port provided by GitHub user [henrikbrixandersen](#) (see <https://github.com/stnolting/neorv32/discussions/172>). ☐☐

# Chapter 16. FreeRTOS Support

A NEORV32-specific port and a simple demo for FreeRTOS (<https://github.com/FreeRTOS/FreeRTOS>) are available in a separate repository on GitHub: <https://github.com/stnolting/neorv32-freertos>



# Chapter 17. LiteX SoC Builder Support

**LiteX** is a SoC builder framework by **Enjoy-Digital** that allows easy creation of complete system-on-chip designs - including sophisticated interfaces like Ethernet, serial ATA and DDR memory controller. The NEORV32 has been ported to the LiteX framework to be used as central processing unit.

The default microcontroller-like NEORV32 processor is not directly supported as all the peripherals would provide some *redundancy*. Instead, the LiteX port uses a *core complex wrapper* that only includes the actual NEORV32 CPU, the instruction cache (optional), the RISC-V machine system timer (optional), the on-chip debugger (optional) and the internal bus infrastructure. The specific implementation of optional modules as well as RISC-V ISA configuration and performance optimization options are controlled by a single *CONFIGURATION* option wrapped in the LiteX build flow. The external bus interface is used to connect to other LiteX SoC parts.



### Core Complex Wrapper

The NEORV32 core complex wrapper used by LiteX for integration can be found in [rtl/system\\_integration/neorv32\\_litex\\_core\\_complex.vhd](#).



### LiteX NEORV32 Documentation

More information can be found in the "NEORV32" section of the LiteX project wiki: <https://github.com/enjoy-digital/litex/wiki/CPUs>



### Work-In-Progress ☐

UG: synthesis - how to create a whole NEORV32 + LiteX SoC for a FPGA

LiteX: debugger - the NEORV32 on-chip-debugger is not supported by the LiteX port yet

LiteX: external interrupt - the "RISC-V machine external interrupt" is not supported by the LiteX port yet

## 17.1. LiteX Setup

1. Install LiteX and the RISC-V compiler following the excellent quick start guide: <https://github.com/enjoy-digital/litex/wiki#quick-start-guide>
2. The NEORV32 port for LiteX uses GHDL and yosys for converting the VHDL files via the **GHDL-yosys-plugin**. You can download prebuilt packages for example from <https://github.com/YosysHQ/fpga-toolchain>, which is no longer maintained. It is superseded by <https://github.com/YosysHQ/fpga-toolchain>.
3. **EXPERIMENTAL**: GHDL provides a **synthesis options**, which converts a VHDL setup into a plain-Verilog module (not tested on LiteX yet). Check out **neorv32-verilog** for more information.



### GHDL-yosys Plugin

If you would like to use the experimental GHDL Yosys plugin for VHDL on Linux or

MacOS, you will need to set the `GHDL_PREFIX` environment variable. e.g. `export GHDL_PREFIX=<install_dir>/fpga-toolchain/lib/ghdl`. On Windows this is not necessary.

If you are using an existing Makefile set up for ghdl-yosys-plugin and see ERROR: This version of yosys is built without plugin support you probably need to remove `-m ghdl` from your yosys parameters. This is because the plugin is typically loaded from a separate file but it is provided built into yosys in this package.  
- from <https://github.com/YosysHQ/fpga-toolchain>

**This means you might have to edit the call to yosys in `litex/soc/cores/cpu/neorv32/core.py`.**

3. Add the `bin` folder of the ghdl-yosys-plugin to your `PATH` environment variable. You can test your yosys installation and check for the GHDL plugin:

```
$ yosys -H
```

```
/-----\
|
| yosys -- Yosys Open SYnthesis Suite
|
| Copyright (C) 2012 - 2020 Claire Xenia Wolf <claire@yosyshq.com>
|
| Permission to use, copy, modify, and/or distribute this software for any
| purpose with or without fee is hereby granted, provided that the above
| copyright notice and this permission notice appear in all copies.
|
| THE SOFTWARE IS PROVIDED "AS IS" AND THE AUTHOR DISCLAIMS ALL WARRANTIES
| WITH REGARD TO THIS SOFTWARE INCLUDING ALL IMPLIED WARRANTIES OF
| MERCHANTABILITY AND FITNESS. IN NO EVENT SHALL THE AUTHOR BE LIABLE FOR
| ANY SPECIAL, DIRECT, INDIRECT, OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES
| WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS, WHETHER IN AN
| ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT OF
| OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.
|
\-----/
```

```
Yosys 0.10+12 (open-tool-forge build) (git sha1 356ec7bb, gcc 9.3.0-17ubuntu1~20.04-0s)
```

```
-- Running command 'help' --
```

```
... ①
ghdl          load VHDL designs using GHDL ②
...
```

- ① A long list of plugins...
- ② This is the plugin we need.

## 17.2. LiteX Simulation

Start a simulation right in your console using the NEORV32 as target CPU:

```
$ litex_sim --cpu-type=neorv32
```

LiteX will start running its BIOS:

```
  _ _  _ _  _ _  
 / /  ( ) / ____ | | / /  
 / / _ / / _ / - _ > <  
 / ____ / _ \ _ \ _ / | |  
Build your hardware, easily!
```

(c) Copyright 2012-2022 Enjoy-Digital  
(c) Copyright 2007-2015 M-Labs

BIOS built on Jul 19 2022 12:21:36  
BIOS CRC passed (6f76f1e8)

LiteX git sha1: 0654279a

----- SoC -----

CPU: NEORV32-standard @ 1MHz  
BUS: WISHBONE 32-bit @ 4GiB  
CSR: 32-bit data  
ROM: 128KiB  
SRAM: 8KiB

----- Boot -----

Bootling from serial...  
Press Q or ESC to abort boot completely.  
sL5DdSMmkekro  
Timeout  
No boot medium found

----- Console -----

litex> help

LiteX BIOS, available commands:

```
flush_cpu_dcache    - Flush CPU data cache
crc                  - Compute CRC32 of a part of the address space
ident                - Identifier of the system
help                 - Print this help

serialboot           - Boot from Serial (SFL)
reboot               - Reboot
boot                 - Boot from Memory

mem_cmp              - Compare memory content
mem_speed            - Test memory speed
mem_test             - Test memory access
mem_copy             - Copy address space
mem_write            - Write address space
mem_read             - Read address space
mem_list             - List available memory regions
```

```
litex>
```

You can use the provided console to execute LiteX commands.

# Chapter 18. Debugging using the On-Chip Debugger

The NEORV32 on-chip debugger ("OCD") allows *online* in-system debugging via an external JTAG access port from a host machine. The general flow is independent of the host machine's operating system. However, this tutorial uses Windows and Linux (Ubuntu on Windows / WSL) in parallel running the upstream version of OpenOCD and the RISC-V *GNU debugger* `gdb`.



## TLDR

You can start a pre-configured debug session (using default `main.elf` as executable and `target extended-remote localhost:3333` as GDB connection configuration) by using the **GDB** makefile target (i.e. `make gdb`).



## OCD Hardware Implementation

See datasheet section [On Chip Debugger \(OCD\)](#) for more information regarding the actual hardware.



## OCD CPU Requirements

The on-chip debugger is only implemented if the `ON_CHIP_DEBUGGER_EN` generic is set *true*. Furthermore, it requires the `Zicsr` and `Zifencei` CPU extension, which are always enabled by the CPU.

## 18.1. Hardware Requirements

Make sure the on-chip debugger of your NEORV32 setup is implemented (`ON_CHIP_DEBUGGER_EN` generic = *true*). This tutorial uses `gdb` to **directly upload an executable** to the processor. If you are using the default processor setup *with* internal instruction memory (IMEM) make sure it is implemented as RAM (`INT_BOOTLOADER_EN` generic = *true*).

Connect a JTAG adapter to the NEORV32 `jtag_*` interface signals. If you do not have a full-scale JTAG adapter, you can also use a FTDI-based adapter like the "FT2232H-56Q Mini Module", which is a simple and inexpensive FTDI breakout board.

Table 3. JTAG pin mapping

NEORV32 top signal	JTAG signal	Default FTDI port
<code>jtag_tck_i</code>	TCK	D0
<code>jtag_tdi_i</code>	TDI	D1
<code>jtag_tdo_o</code>	TDO	D2
<code>jtag_tms_i</code>	TMS	D3



## JTAG TAP Reset

The NEORV32 JTAG TAP does not provide a dedicated reset signal ("TRST"). However, the missing TRST is not a problem, since JTAG-level resets can be triggered using with TMS signaling.

## 18.2. OpenOCD

The NEORV32 on-chip debugger can be accessed using the upstream version of OpenOCD. A pre-configured OpenOCD configuration file is provided ([sw/openocd/openocd\\_neorv32.cfg](#)) that allows an easy access to the NEORV32 CPU.



You might need to adapt `ftdi vid_pid`, `ftdi channel` and `ftdi layout_init` in [sw/openocd/openocd\\_neorv32.cfg](#) according to your interface chip and your operating system.



If you want to modify the JTAG clock speed (via `adapter speed` in [sw/openocd/openocd\\_neorv32.cfg](#)) make sure to meet the clock requirements noted in [Documentation: Debug Transport Module \(DTM\)](#).

To access the processor using OpenOCD, open a terminal and start OpenOCD with the pre-configured configuration file.

*Listing 15. Connecting via OpenOCD (on Windows) using the default [openocd\\_neorv32.cfg](#) script*

```
N:\Projects\neorv32\sw\openocd>openocd -f openocd_neorv32.cfg
Open On-Chip Debugger 0.11.0 (2021-11-18) [https://github.com/sysprogs/openocd]
Licensed under GNU GPL v2
libusb1 09e75e98b4d9ea7909e8837b7a3f00dda4589dc3
For bug reports, read
    http://openocd.org/doc/doxygen/bugs.html
Info : clock speed 1000 kHz
Info : JTAG tap: neorv32.cpu tap/device found: 0x00000000 (mfg: 0x000 (<invalid>),
part: 0x0000, ver: 0x0)
Info : datacount=1 progbufsize=2
Info : Disabling abstract command reads from CSRs.
Info : Examined RISC-V core; found 1 harts
Info : hart 0: XLEN=32, misa=0x40901107
Info : starting gdb server for neorv32.cpu.0 on 3333
Info : Listening on port 3333 for gdb connections
Target HALTED.
Ready for remote connections.
Info : Listening on port 6666 for tcl connections
Info : Listening on port 4444 for telnet connections
```

OpenOCD has successfully connected to the NEORV32 on-chip debugger and has examined the CPU (showing the content of the `misa` CSRs). The processor is halted and OpenOCD waits for `gdb` to connect via port 3333.

## 18.3. Debugging with GDB



### GDB + SVD

Together with a third-party plugin the processor's SVD file can be imported right into GDB to allow comfortable debugging of peripheral/I/O devices (see <https://github.com/stnolting/neorv32/discussions/656>).

This guide uses the simple "blink example" from `sw/example/demo_blink_led` as simplified test application to show the basics of in-system debugging.

At first, the application needs to be compiled. We will use the minimal machine architecture configuration (`rv32i`) here to be independent of the actual processor/CPU configuration. Navigate to `sw/example/demo_blink_led` and compile the application:

### Listing 16. Compile the test application

```
.../neorv32/sw/example/demo_blink_led$ make MARCH=rv32i USER_FLAGS+=-g clean_all all
```



### Adding debug symbols to the executable

`USER_FLAGS+=-g` passes the `-g` flag to the compiler so it adds debug information/symbols to the generated ELF file. This is optional but will provide more sophisticated debugging information (like source file line numbers).

This will generate an ELF file `main.elf` that contains all the symbols required for debugging. Furthermore, an assembly listing file `main.asm` is generated that we will use to define breakpoints.

Open another terminal in `sw/example/demo_blink_led` and start `gdb`.

### Listing 17. Starting GDB (on Linux (Ubuntu on Windows))

```
.../neorv32/sw/example/demo_blink_led$ riscv32-unknown-elf-gdb
GNU gdb (GDB) 10.1
Copyright (C) 2020 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.
Type "show copying" and "show warranty" for details.
This GDB was configured as "--host=x86_64-pc-linux-gnu --target=riscv32-unknown-elf".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<https://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
  <http://www.gnu.org/software/gdb/documentation/>.

For help, type "help".
Type "apropos word" to search for commands related to "word".
```

(gdb)

Now connect to OpenOCD using the default port 3333 on your machine. We will use the previously generated ELF file `main.elf` from the `demo_blink_led` example. Finally, upload the program to the processor and start debugging.



The executable that is uploaded to the processor is **not** the default NEORV32 executable (`neorv32_exe.bin`) that is used for uploading via the bootloader. Instead, all the required sections (like `.text`) are extracted from `main.elf` by GDB and uploaded via the debugger's indirect memory access.

*Listing 18. Running GDB*

```
(gdb) target extended-remote localhost:3333 ①
Remote debugging using localhost:3333
warning: No executable has been specified and target does not support
determining executable automatically. Try using the "file" command.
0xffff0c94 in ?? () ②
(gdb) file main.elf ③
A program is being debugged already.
Are you sure you want to change the file? (y or n) y
Reading symbols from main.elf...
(gdb) load ④
Loading section .text, size 0xd0c lma 0x0
Loading section .rodata, size 0x39c lma 0xd0c
Start address 0x00000000, load size 4264
Transfer rate: 43 KB/sec, 2132 bytes/write.
(gdb)
```

① Connect to OpenOCD

② The CPU was still executing code from the bootloader ROM - but that does not matter here

③ Select `main.elf` from the `demo_blink_led` example

④ Upload the executable

After the upload, GDB will make the processor jump to the beginning of the uploaded executable (by default, this is the beginning of the instruction memory at `0x00000000`) skipping the bootloader and halting the CPU right before executing the `demo_blink_led` application.



After gdb has connected to the CPU, it is recommended to disable the CPU's global interrupt flag (`mstatus.mie`, = bit #3) to prevent unintended calls of potentially outdated trap handlers. The global interrupt flag can be cleared using the following gdb command: `set $mstatus = ($mstatus & ~(1<<3))`. Interrupts can be enabled globally again by the following command: `set $mstatus = ($mstatus | (1<<3))`.



### 18.3.1. Software Breakpoints

The following steps are just a small showcase that illustrate a simple debugging scheme.

While compiling `demo_blink_led`, an assembly listing file `main.asm` was generated. Open this file with a text editor to check out what the CPU is going to do when resumed.

The `demo_blink_led` example implements a simple counter on the 8 lowest GPIO output ports. The program uses "busy wait" to have a visible delay between increments. This waiting is done by calling the `neorv32_cpu_delay_ms` function. We will add a *breakpoint* right at the end of this wait function so we can step through the iterations of the counter.

*Listing 19. Cut-out from `main.asm` generated from the `demo_blink_led` example*

```
00000688 <__neorv32_cpu_delay_ms_end>:
688: 01c12083      lw  ra,28(sp)
68c: 02010113      addi sp,sp,32
690: 00008067      ret
```

The very last instruction of the `neorv32_cpu_delay_ms` function is `ret` (= return) at hexadecimal `690` in this example. Add this address as *breakpoint* to GDB.



The address might be different if you use a different version of the software framework or if different ISA options are configured.

*Listing 20. Adding a GDB software breakpoint*

```
(gdb) b * 0x690 ①
Breakpoint 1 at 0x690
```

① `b` is an alias for `break`, which adds a *software* breakpoint.



#### *How do software breakpoints work?*

Software breakpoints are used for debugging programs that are accessed from read/write memory (RAM) like IMEM. The debugger temporarily replaces the instruction word of the instruction, where the breakpoint shall be inserted, by a `ebreak` / `c.ebreak` instruction. Whenever execution reaches this instruction, debug mode is entered and the debugger restores the original instruction at this address to maintain original program behavior.

When debugging programs executed from ROM *hardware-assisted* breakpoints using the core's trigger module have to be used. See section **Hardware Breakpoints** for more information.

Now execute `c` (= continue). The CPU will resume operation until it hits the break-point. By this we can move from one counter increment to another.

Listing 21. Iterating from breakpoint to breakpoint

```
Breakpoint 1 at 0x690
(gdb) c
Continuing.

Breakpoint 1, 0x00000690 in neorv32_cpu_delay_ms ()
(gdb) c
Continuing.

Breakpoint 1, 0x00000690 in neorv32_cpu_delay_ms ()
(gdb) c
Continuing.
```



#### Hardcoded EBREAK Instructions In The Program Code

If your original application code uses the BREAK instruction (for example for some OS calls/signaling) this instruction will cause an enter to debug mode when executed. These situation cannot be continued using gdb's `c` nor can they be "stepped-over" using the single-step command `s`. You need to declare the `ebreak` instruction as breakpoint to be able to resume operation after executing it. See <https://sourceware.org/pipermail/gdb/2021-January/049125.html>

### 18.3.2. Hardware Breakpoints

Hardware-assisted breakpoints using the CPU's trigger module are required when debugging code that is executed from read-only memory (ROM) as GDB cannot temporarily replace instructions by BREAK instructions.

From a user point of view hardware breakpoints behave like software breakpoints. GDB provides a command to setup a hardware-assisted breakpoint:

Listing 22. Adding a GDB hardware breakpoint

```
(gdb) hb * 0x690 ①
Breakpoint 1 at 0x690
```

① `hb` is an alias for `hbreak`, which adds a *hardware* breakpoint.



The CPU's trigger module only provides a single *instruction address match* type trigger. Hence, only a single `hb` hardware-assisted breakpoint can be used.

## 18.4. Segger Embedded Studio

Software for the NEORV32 processor can also be developed and debugged *in-system* using Segger Embedded Studio and a Segger J-Link probe. The following links provide further information as well as an excellent tutorial.

- Segger Embedded Studio: <https://www.segger.com/products/development-tools/embedded-studio>
- Segger notes regarding NEORV32: [https://wiki.segger.com/J-Link\\_NEORV32](https://wiki.segger.com/J-Link_NEORV32)
- Excellent tutorial: <https://www.emb4fun.com/riscv/ses4rv/index.html>

## Chapter 19. NEORV32 in Verilog

If you are more of a Verilog fan or if your EDA toolchain does not support VHDL or mixed-language designs you can use an **all-Verilog** version of the processor provided by the [neorv32-verilog](#) repository.



Note that this is **not a manual re-implementation of the core in Verilog** but rather an automated conversion.

GHDL's synthesis feature is used to convert a pre-configured NEORV32 setup - including all peripherals, memories and memory images - into a single, unoptimized plain-Verilog module file without any (technology-specific) primitives.



### GHDL Synthesis

More information regarding GHDL's synthesis option can be found at <https://ghdl.github.io/ghdl/using/Synthesis.html>.

An intermediate VHDL wrapper is provided that can be used to configure the processor (using VHDL generics) and to customize the interface ports. After conversion, a single Verilog file is generated that contains the whole NEORV32 processor. The original processor module hierarchy is preserved as well as most (all?) signal names, which allows easy inspection and debugging of simulation waveforms and synthesis results.

*Listing 23. Example: interface of the resulting NEORV32 Verilog module (for a minimal SoC configuration)*

```
module neorv32_verilog_wrapper
  (input  clk_i,
   input  rstn_i,
   input  uart0_rxd_i,
   output uart0_txd_o);
```

The generated Verilog code has been simulated and verified with **Icarus Verilog** (simulation) and AMD Vivado (simulation and synthesis).



For detailed information check out the [neorv32-verilog](#) repository at <https://github.com/stnolting/neorv32-verilog>.

# Chapter 20. Eclipse IDE

Eclipse (<https://www.eclipse.org/>) is an interactive development environment that can be used to develop, debug and profile application code for the NEORV32 RISC-V Processor. This chapter shows how to import the provided **example setup** from the NEORV32 project repository. Additionally, all the required steps to create a compatible project from scratch are illustrated in this chapter.

*This is a Makefile-Based Project!*



Note that the provided Eclipse example project (as well as the setup tutorial in this section) implements a **makefile-based project**. Hence, the makefile in the example folder is used for building the application instead of the Eclipse-managed build system. Therefore, **all compiler options, include folder, source files, etc. have to be defined within this makefile**.

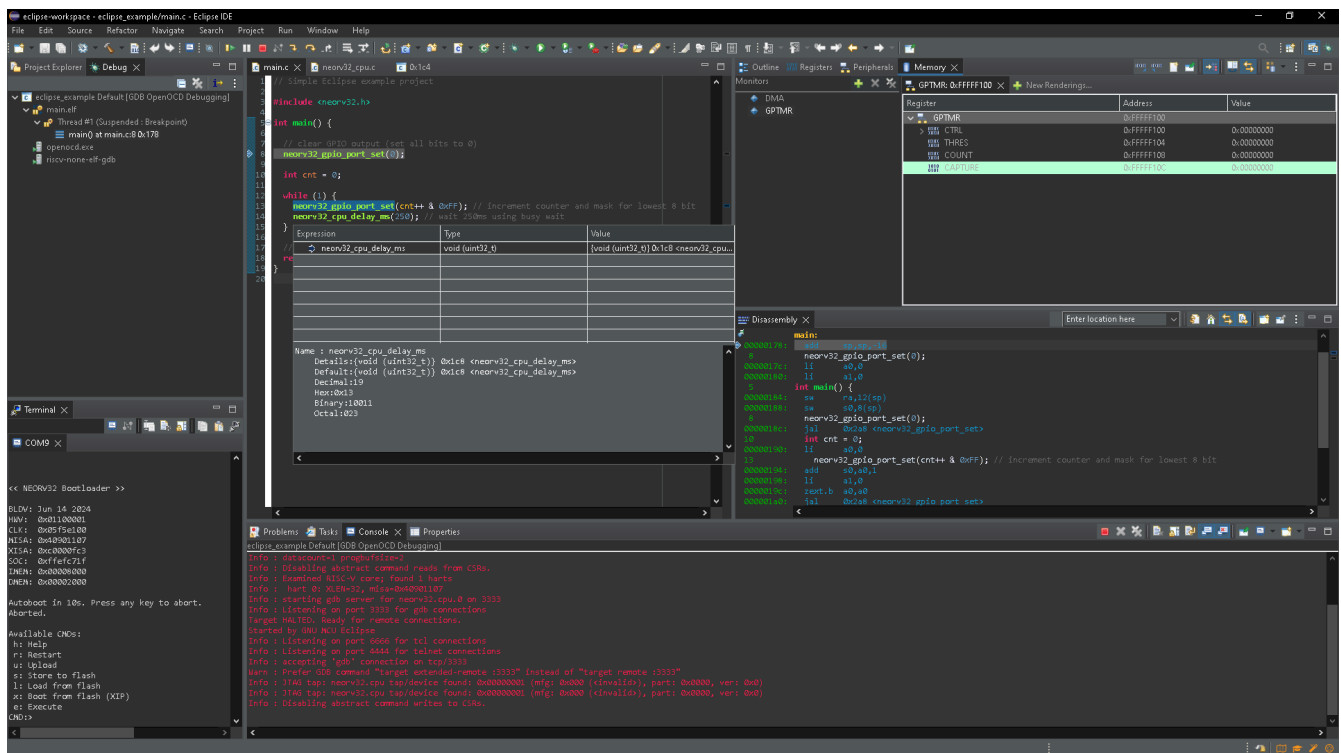


Figure 4. Developing and debugging code for the NEORV32 using the Eclipse IDE

## 20.1. Eclipse Prerequisites

The following tools are required:

- Eclipse IDE (**Eclipse IDE for Embedded C/C++ Developers**): <https://www.eclipse.org/downloads/>
- Precompiled RISC-V GCC toolchain: e.g. <https://github.com/xpack-dev-tools/riscv-none-elf-gcc-xpack>
- Precompiled OpenOCD binaries: e.g. <https://github.com/xpack-dev-tools/openocd-xpack>
- Build tools like make and busybox: e.g. <https://github.com/xpack-dev-tools/windows-build-tools-xpack>

*XPack Windows Build Tools*

The NEORV32 makefile relies on the **basename** command which might not be part of the default XPack Windows Build Tools. However, you can just open the according **bin** folder, copy **busybox.exe** and rename that copy to **basename.exe**.

## 20.2. Import The Provided Eclipse Example Project

A preconfigured Eclipse project is available in **neorv32/sw/example/eclipse**. To import it:

1. Open Eclipse.
2. Click on **File > Import**, expand **General** and select **Projects from Folder or Archive**.
3. Click **Next**.
4. Click on **Directory** and select the provided example project folder (see directory above).
5. Click **Finish**.

*NEORV32 Folder and File Paths*

The provided example project uses **relative paths** for including all the NEORV32-specific files and folders (in the Eclipse configuration files). Note that these paths need to be adjusted when moving the example setup to a different location.

*Tool Configuration*

Make sure to adjust the binaries / installation folders of the RISC-V GCC toolchain, openOCD and Windows build tools according to your installation. See the section **Configure Build Tools** for more information.

*Makefile Adjustment*

Make sure to adjust the variables inside the project's makefile to match your processor configuration (memory sizes, CPU ISA configuration, etc.): [https://stnolting.github.io/neorv32/#\\_application\\_makefile](https://stnolting.github.io/neorv32/#_application_makefile)

## 20.3. Setup a new Eclipse Project from Scratch

This chapter shows all the steps required to create an Eclipse project for the NEORV32 entirely from scratch.

### 20.3.1. Create a new Project

1. Select **File > New > Project**.
2. Expand **C/C\*\*** and select **\*\*C project**.
3. In the **C++ Project** wizard:
  - Enter a **Project name**.

- Uncheck the box next to **Use default location** and specify a location using **Browse** where you want to create the project.
  - From the **Project type** list expand **Makefile project** and select **Empty Project**.
  - Select **RISC-V Cross GCC** from the **Toolchain** list on the right side.
  - Click **Next**.
  - Skip the next page using the default configuration by clicking **Next**.
4. In the **GNU RISC-V Cross Toolchain** wizard configure the **Toolchain name** and **Toolchain path** according to your RISC-V GCC installation.
- Example: **Toolchain name:** xPack GNU RISC-V Embedded GCC (riscv-none-elf-gcc)
  - Example: **Toolchain path:** C:\Program Files (x86)\xpack-riscv-none-elf-gcc-13.2.0-2\bin
5. Click **Finish**.

If you need to reconfigure the RISC-V GCC binaries and/or paths:

1. right-click on the project in the left view, select **Properties**
2. expand **MCU** and select **RISC-V Toolchain Paths**
3. adjust the **Toolchain folder** and the **Toolchain name** if required
4. Click **Apply**.

### 20.3.2. Add Initial Files

Start a simple project by adding two initial files. Further files can be added later. Only the makefile is really relevant here.

1. Add a new file by right-clicking on the project and select **New > File** and enter **main.c** in the filename box.
2. Add another new file by right-clicking on the project and select **New > File** and enter **makefile** in the filename
3. Copy the makefile of an existing NEORV32 example program and paste it to the new (empty) makefile.

### 20.3.3. Add Build Targets (optional)

This step adds some of the targets of the NEORV32 makefile for easy access. This step is optional.

1. In the project explorer right-click on the project and select **Build Target > Create....**
2. Add “all” as **Target name** (keep all the default checked boxes).
3. Repeat these steps for all further targets that you wish to add (e.g **clean\_all**, **exe**, **elf**).



#### *Clean-All Target*

Adding the **clean\_all** target is highly recommended. Executing this target once

after importing the project ensures that there are no (incompatible) artifacts left from previous builds.



#### *Available Target*

See the NEORV32 data sheet for a list and description of all available makefile targets: [https://stnolting.github.io/neorv32/#\\_makefile\\_targets](https://stnolting.github.io/neorv32/#_makefile_targets)

### 20.3.4. Configure Build Tools

This step is only required if your system does not provide any build tools (like **make**) by default.

1. In the project explorer right-click on the project and select **Properties**.
2. Expand **MCU** and click on **Build Tools Path**.
3. Configure the **Build tools folder**.
  - Example: **Build tools folder**: `C:/xpack/xpack-windows-build-tools-4.4.1-2/bin`
4. Click **Apply and Close**.

### 20.3.5. Adjust Default Build Configuration (optional)

This will simplify the auto-build by replacing the default **make all** command by **make elf**. Thus, only the required **main.elf** file gets generated instead of *all* executable files (like HDL and memory image files).

1. In the project explorer right-click on the project and select **Properties**.
2. Select **C/C++ Build** and click on the **Behavior** Tab.
3. Update the default targets in the **Workbench Build Behavior** box:
  - **Build on resource save**: **elf** (only build the ELF file)
  - **Build (Incremental build)**: **elf** (only build the ELF file)
  - **Clean**: **clean** (only remove project-local build artifacts)
4. Click **Apply and Close**.

### 20.3.6. Add NEORV32 Software Framework

1. In the project explorer right-click on the project and select **Properties**.
2. Expand **C/C++ General**, click on **Paths and Symbols** and highlight **Assembly** under **Languages**.
3. In the **Include** tab click **Add...**
  - Check the box in front of **Add to all languages** and click on **File System...** and select the NEORV32 library include folder (`path/to/neorv32/sw/lib/include`).
  - Click **OK**.
4. In the **Include** tab click **Add...**



- Check the box in front of **Add to all languages** and click on **File System...** and select the NEORV32 commons folder (`path/to/neorv32/sw/common`).
  - Click **OK**.
5. Click on the **Source Location** tab and click **Link Folder...\***.
    - Check the box in front of **Link to folder in the system** and click the **Browse** button.
    - Select the source folder of the NEORV32 software framework (`path/to/neorv32/sw/lib/source`).
    - Click **OK**.
  6. Click **Apply and Close**.

### 20.3.7. Setup OpenOCD

1. In the project explorer right-click on the project and select **Properties**.
2. Expand **MCU** and select **OpenOCD Path**.
  - Configure the **Executable** and **Folder** according to your openOCD installation.
  - Example: **Executable:** `openocd.exe`
  - Example: **Folder:** `C:\OpenOCD\bin`
  - Click **Apply and Close**.
3. In the top bar of Eclipse click on the tiny arrow right next to the **Debug** bug icon and select **Debug Configurations**.
4. Double-click on **GDB OpenOCD Debugging**; several menu tabs will open on the right.
  - In the **Main** tab add `main.elf` to the **C/C++ Application** box.
  - In the **Debugger** tab add the NEORV32 OpenOCD script with a `-f` in front of it-
  - Example: **Config options:** `-f ../../openocd/openocd_neorv32.cfg`
  - In the **Startup** tab uncheck the box in front of **Initial Reset** and add `monitor reset halt` to the box below.
  - In the "Common" tab mark **Shared file** to store the run-configuration right in the project folder instead of the workspace(optional).
  - In the **SVD Path** tab add the NEORV32 SVD file (`path/to/neorv32/sw/svd/neorv32.svd`).
5. Click **Apply** and then **Close**.



#### *Default Debug Configuration*

When you start debugging the first time you might need to select the provided debug configuration: **GDB OpenOCD Debugging > eclipse\_example Default**



#### *Debug Symbols*

For debugging the ELF has to be compiled to contain according debug symbols. Debug symbols are enabled by the project's local makefile: `USER_FLAGS += -ggdb -gdwarf-3`

(this configuration seems to work best for Eclipse - at least for me).

If you need to reconfigure OpenOCD binaries and/or paths:

1. right-click on the project in the left view, select **Properties**
2. expand **MCU** and select **OpenOCD Path**
3. adjust the **Folder** and the **Executable** name if required
4. Click **Apply**.

### 20.3.8. Setup Serial Terminal

A serial terminal can be added to Eclipse by installing it as a plugin. I recommend "TM Terminal" which is already installed in some Eclipse bundles.

Open a TM Terminal serial console:

1. Click on **Window > Show View > Terminal** to open the terminal.
2. A **Terminal** tab appears on the bottom. Click the tiny screen button on the right (or press Ctrl+Alt+Shift) to open the terminal configuration.
3. Select **Serial Terminal** in **Choose Terminal** and configure the settings according to the processor's UART configuration.

Installing TM Terminal from the Eclipse market place:

1. Click on **Help > Eclipse Marketplace....**
2. Enter "TM Terminal" to the **Find** line and hit enter.
3. Select **TM Terminal** from the list and install it.
4. After installation restart Eclipse.

## 20.4. Eclipse Setup References

- Eclipse help: <https://help.eclipse.org/latest/index.jsp>
- Importing an existing project into Eclipse: [https://help.eclipse.org/latest/index.jsp?topic=%2Forg.eclipse.cdt.doc.user%2Fgetting\\_started%2Fcdt\\_w\\_import.htm](https://help.eclipse.org/latest/index.jsp?topic=%2Forg.eclipse.cdt.doc.user%2Fgetting_started%2Fcdt_w_import.htm)
- Eclipse OpenOCD Plug-In: <https://eclipse-embed-cdt.github.io/debug/openocd/>

# Chapter 21. Legal

## About

The NEORV32 RISC-V Processor

<https://github.com/stnolting/neorv32>

Stephan Nolting, M.Sc.

☐ European Union

[stnolting@gmail.com](mailto:stnolting@gmail.com)

## License

### BSD 3-Clause License

Copyright (c) NEORV32 contributors. Copyright (c) 2020 - 2025, Stephan Nolting. All rights reserved.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

1. Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
3. Neither the name of the copyright holder nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT HOLDER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.



*SPDX Identifier*

SPDX-License-Identifier: BSD-3-Clause

## Proprietary Notice

- "GitHub" is a subsidiary of Microsoft Corporation.
- "Vivado" and "Artix" are trademarks of AMD Inc.
- "AXI", "AXI4", "AXI4-Lite", "AXI4-Stream", "AHB", "AHB3" and "AHB3-Lite" are trademarks of Arm Holdings plc.
- "ModelSim" is a trademark of Mentor Graphics – A Siemens Business.
- "Quartus [Prime]" and "Cyclone" are trademarks of Intel Corporation.
- "iCE40", "UltraPlus" and "Radiant" are trademarks of Lattice Semiconductor Corporation.
- "GateMate" is a trademark of Cologne Chip AG.
- "Windows" is a trademark of Microsoft Corporation.
- "Tera Term" copyright by T. Teranishi.
- "NeoPixel" is a trademark of Adafruit Industries.
- "Segger Embedded Studio" and "J-Link" are trademarks of Segger Microcontroller Systems GmbH.
- Images/figures made with *Microsoft Power Point*.
- Timing diagrams made with *WaveDrom Editor*.
- Documentation made with **asciidoc**.

All further/unreferenced projects/products/brands belong to their according copyright holders. No copyright infringement intended.

## Disclaimer

This project is released under the BSD 3-Clause license. NO COPYRIGHT INFRINGEMENT INTENDED. Other implied or used projects/sources might have different licensing – see their according documentation for more information.

## Limitation of Liability for External Links

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

## Citing



This is an open-source project that is free of charge. Use this project in any way

you like (as long as it complies to the permissive license). Please cite it appropriately. □



#### *Contributors & Community* □

Please add as many **contributors** as possible to the **author** field.  
This project would not be where it is without them.



#### *DOI*

This project provides a *digital object identifier* provided by **zenodo**:  
DOI 10.5281/zenodo.5018888

## Acknowledgments

A big shout-out to the community and all the **contributors**, who helped improving this project! This project would not be where it is without them. □□

**RISC-V** - instruction sets want to be free!

Continuous integration provided by **GitHub Actions** and powered by **GHDL**.