The NEORV32 RISC-V Processor - User Guide

Version v1.12.5-r29-gee716a5a

Documentation

The online documentation of the project (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

Quick Links

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-`. This default prefix can be chanced by the `RISCV_PREFIX` variable in the NEORV32 application makefile(s).

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
```

Note that this configuration will build a GCC toolchain with `riscv32-unknown-elf-` as prefix / GCC triplet. Hence, you need to adjust the `RISCV_PREFIX` variable accordingly before running any NEORV32 makefiles.

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 processor.

The NEORV32 project features three minimalist 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.

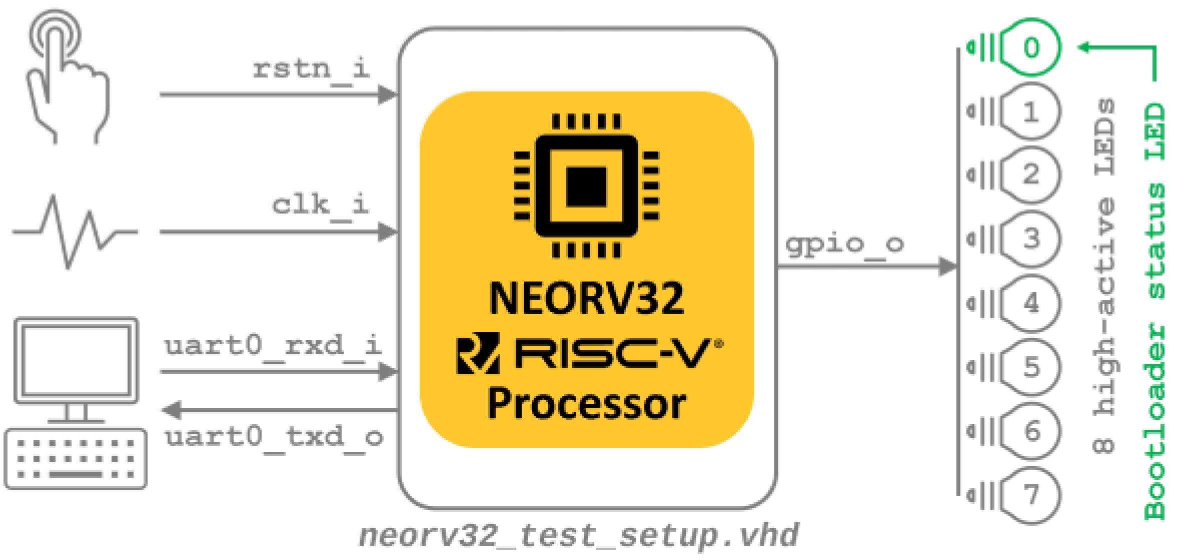


Figure 1. NEORV32 "hello world" Test Setup (`rtl/test_setups/neorv32_test_setup_bootloader.vhd`)

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).

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

NEORV32 VHDL Package

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
  IMEM_SIZE       : natural := 16*1024;   ②
  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 your FPGA does not provide sufficient resources you can modify the memory sizes (`IMEM_SIZE` and `DMEM_SIZE`).
7. The actual clock frequency (Hz) of the top's clock input signal (`clk_i`) needs to be defined using the `CLOCK_FREQUENCY` generic.

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 your 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. The LED(s) connected to `gpio_o` should be flashing now.

Chapter 3. Application Program Compilation

This guide shows how to compile a C-code application for the NEORV32 processor.

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 GPIO controller 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` target is used 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 places into a (new) project-local folder named `build`. The main ELF and the actual executable will be placed in the root project folder.

5. 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. Additionally, the `main.elf` file can be uploaded using the on-chip debugger.

Chapter 4. Installing an Executable Directly Into Memory

An application can be installed as non-volatile image to the internal instruction memory (IMEM) so that it gets executed right after reset. For this concept the IMEM gets implemented as pre-initialized ROM that gets initialized during FPGA bitstream programming.

1. Configure the processor boot mode `BOOT_MODE_SELECT` to "Boot IMEM Image" (select value 2). This boot mode requires the processor-internal instruction memory (IMEM) so ensure is is enabled (`IMEM_EN`).

Listing 4. Processor top entity configuration - enable internal IMEM

```
BOOT_MODE_SELECT => 2,    -- boot from pre-initialized IMEM-ROM
IMEM_EN          => true, -- implement processor-internal instruction memory
```

2. Regenerate and re-install the default IMEM initialization file (`rtl/core/neorv32_application_image.vhd`) so that it contains the image of your actual application firmware.

```
neorv32/sw/example/demo_blink_led$ make clean_all image install
Memory utilization:
   text    data     bss     dec     hex filename
  22192    1352    4216   27760    6c70 main.elf
Compiling image generator...
Generating neorv32_application_image.vhd
Installing application image to ../../../rtl/core/neorv32_application_image.vhd
```

3. Rerun FPGA synthesis and upload the bitstream. Your application code now resides non-volatile in the processor’s IMEM and is executed right after reset.

Chapter 5. 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 coarse Comparative Summary is given at the end of this section.

5.1. 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.

5.2. 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 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 according 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 latency. In addition, the processor's DMA controller can also used to/from transfer data to processor-external modules.

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.

5.3. 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, cryptography accelerators or custom interfaces (like IIS).

5.4. 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.

5.5. 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 Predefined On-Chip Extension Options

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

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

Chapter 6. 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 including an interactive configuration GUI. For this, a specialized wrapper for the processor's top entity is provided (`rtl/system_integration/neorv32_vivado_ip.vhd`) that features AXI4-compatible (via XBUS) and AXI4-Stream-compatible (via SLINK) interfaces.

General AXI Wrapper

The provided AXI4-bridge can also be used for custom (AXI) setups outside of Vivado and/or IP block designs.

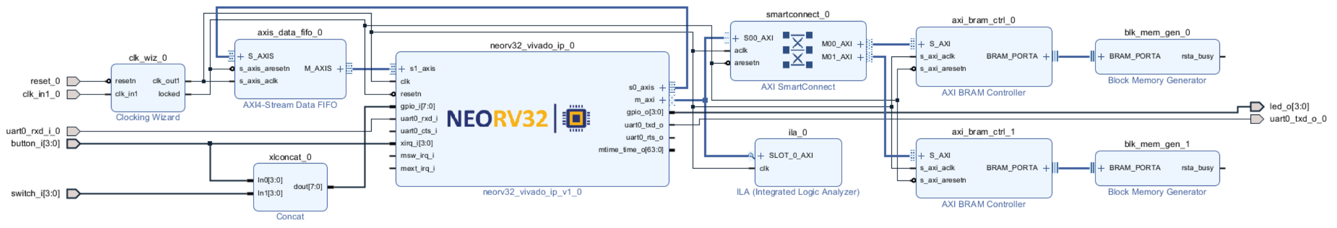
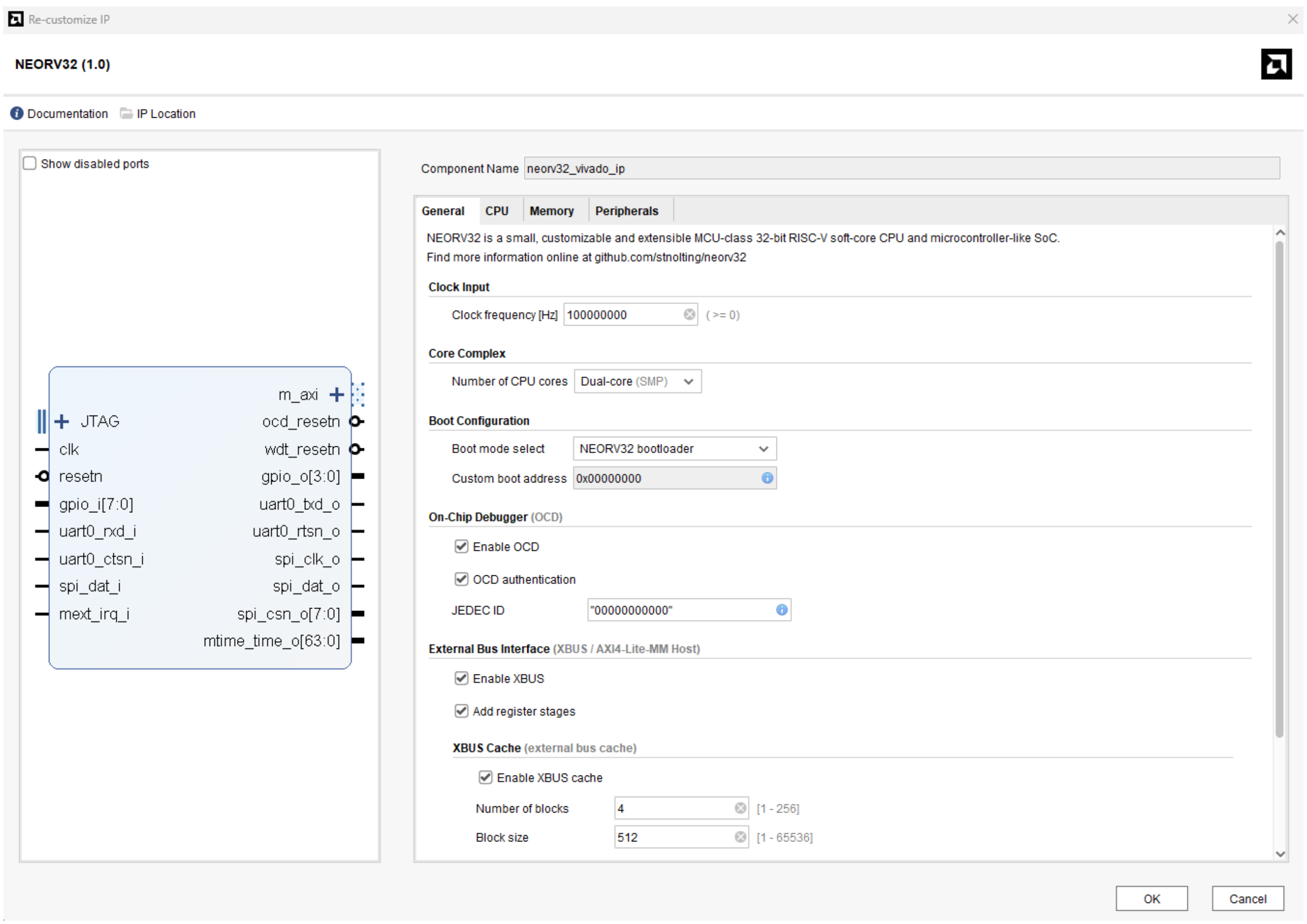


Figure 2. Example Vivado SoC using the NEORV32 IP Block

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



NEORV32 (1.0)

Component Name neorv32_vivado_ip

NEORV32 is a small, customizable and extensible MCU-class 32-bit RISC-V soft-core CPU and microcontroller-like SoC.
Find more information online at github.com/stnolting/neorv32

Clock Input

Clock frequency [Hz] 100000000 (>= 0)

Core Complex

Number of CPU cores Dual-core (SMP)

Boot Configuration

Boot mode select NEORV32 bootloader

Custom boot address 0x00000000

On-Chip Debugger (OCD)

✓ Enable OCD

✓ OCD authentication

JEDEC ID "00000000000"

External Bus Interface (XBUS / AXI4-Lite-MM Host)

✓ Enable XBUS

✓ Add register stages

XBUS Cache (external bus cache)

✓ Enable XBUS cache

Number of blocks 4 [1 - 256]

Block size 512 [1 - 65536]

Figure 3. NEORV32 IP Customization GUI

The following steps show how to package the processor using the provided TCL script (`neorv32/rtl/system_integration/neorv32_vivado_ip.tcl`) and how to import the generated IP block into the Vivado IP repository.

Packaging from within a Vivado Project (recommended)

1. Open your Vivado block design.
2. 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. The `packaged_ip` sub-folder provides the actual IP module which is automatically added to the project's IP repository.
8. You will find the NEORV32 in the "User Repository" section of the Vivado IP catalog.

Stand-Alone Packaging

1. Start Vivado.
2. On the splash screen select "Tools → Run TCL Script".
3. Navigate to the `neorv32/rtl/system_integration` folder and select `neorv32_vivado_ip.tcl`.
4. A second Vivado instance will open automatically packaging the IP module. After this process is completed, the second Vivado instance will automatically close again.
5. A new folder `neorv32_vivado_ip_work` is created in `neorv32/rtl/system_integration` which contains the IP-packaging Vivado project.
6. The `packaged_ip` sub-folder provides the actual IP module.
7. Open your custom design where you want to integrate the NEORV32 IP module.
8. Click on "Settings" in the "Project Manager" on the left side.
9. Under "Project Settings" expand the "IP" section and click on "Repository".
10. Click the large plus button and select the previously generated IP folder (`path/to/neorv32/rtl/system_integration/neorv32_vivado_ip_work/packaged_ip`).
11. Click "Select" and close the Settings menu with "Apply" and "OK".
12. 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 5. 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 to the hardware RTL code (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 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.

Out-of-Context Synthesis

If NEORV32 configurations do not take effect, try using the out-of-context synthesis option.

1. Incremental Compilation of Simulation Sources (ISIM)

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 7. 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 Settings* → *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.

7.1. 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 (`tb.uart0_rx.log` for UART0, `tb.uart1_rx.log` 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.

7.2. 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).

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.log`: ASCII data send via UART0
- `neorv32.uart1.log`: 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 6. Auto-Enable UART0 Simulation-Mode while Compiling

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

7.3. 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 7. Invoking the default GHDL simulation script

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

7.4. Simulation using Application Makefiles

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

Listing 8. 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`

7.4.1. 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 install sim`:

```
neorv32/sw/example/hello_world$ make USER_FLAGS+=-DUART0_SIM_MODE clean install sim
../../../sw/lib/source/neorv32_uart.c: In function 'neorv32_uart_setup':
../../../sw/lib/source/neorv32_uart.c:80:2: 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] ①
   80 | #warning UART0_SIM_MODE (primary UART) enabled! \
      |  ^~~~~~~
Memory utilization:
   text    data     bss     dec     hex filename
   5596       0     116    5712    1650 main.elf ②
Compiling image generator...
Generating neorv32_application_image.vhd
Installing application image to ../../../rtl/core/neorv32_application_image.vhd ③
Simulating processor using default testbench...
GHDL simulation run parameters: --stop-time=10ms ④
../rtl/core/neorv32_top.vhd:329:5:@0ms:(assertion note): [NEORV32] The NEORV32 RISC-V
Processor (v1.11.6.0), github.com/stnolting/neorv32
../rtl/core/neorv32_top.vhd:335:5:@0ms:(assertion note): [NEORV32] Processor
Configuration: CPU (smp-dual-core) IMEM-ROM DMEM I-CACHE D-CACHE XBUS CLINT GPIO UART0
UART1 SPI SDI TWI TWD PWM WDT TRNG CFS NEOLED GPTMR ONEWIRE DMA SLINK SYSINFO OCD OCD-
AUTH OCD-HWBP
../rtl/core/neorv32_top.vhd:388:5:@0ms:(assertion note): [NEORV32] BOOT_MODE_SELECT =
2: booting IMEM image
../rtl/core/neorv32_cpu.vhd:130:5:@0ms:(assertion note): [NEORV32] CPU ISA:
rv32iabmux_zaamo_zalrsc_zba_zbb_zbkb_zbkc_zbkx_zbs_zicntr_zicond_zicsr_zifencei_zihpm_
zfinx_zkn_zknd_zkne_zknh_zks_zksed_zksh_zkt_zmmul_zxcfu_sdext_sdtrig_smpmp
../rtl/core/neorv32_cpu.vhd:168:5:@0ms:(assertion note): [NEORV32] CPU tuning options:
fast_mul fast_shift
```

```
../rtl/core/neorv32_cpu.vhd:175:5:@0ms:(assertion warning): [NEORV32] Assuming this is
a simulation.
../rtl/core/neorv32_imem.vhd:61:3:@0ms:(assertion note): [NEORV32] Implementing
processor-internal IMEM as pre-initialized ROM.
../rtl/core/neorv32_trng.vhd:298:3:@0ms:(assertion note): [neoTRNG] The neoTRNG (v3.3)
- A Tiny and Platform-Independent True Random Number Generator,
https://github.com/stnolting/neoTRNG
../rtl/core/neorv32_trng.vhd:308:3:@0ms:(assertion warning): [neoTRNG] Simulation-mode
enabled (NO TRUE/PHYSICAL RANDOM)!
../rtl/core/neorv32_debug_auth.vhd:44:3:@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 8. Building the Documentation

The data sheet and 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-Built PDFs

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

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

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

Listing 9. Example: Generate HTML documentation (data sheet) using `asciidoctor`

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

Using Containers for Building

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

Chapter 9. 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 10. 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 11. 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 Port Documentation

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

11.1. 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 superdesed 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
-Os)


-- Running command `help' --

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

① A long list of plugins...

② This is the plugin we need.

11.2. 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 ==================--
Booting 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 12. MicroPython Port

A simple out-of-tree port of MicroPython for the NEORV32 RISC-V Processor can be found in a separate repository: https://github.com/stnolting/neorv32-micropython

Work-In-Progress

This port is still under development. Hence, it supports just some simple modules and methods yet. However, it is already fully operational.

Listing 10. MicroPython REPL Console

```
MicroPython v1.25.0 on 2025-04-20; neorv32-default with neorv32
Type "help()" for more information.
>>> help("modules")
__main__          collections       machine           sys
array             gc                micropython       time
builtins          io                struct
Plus any modules on the filesystem
```

Listing 11. Basic build-in Modules

```
>>> import machine
>>> machine.info()
NEORV32 version 1.11.2.9
Clock: 150000000 Hz
MISA:  0x40901105
MXISA: 0x66006cd3
SoC:   0x480ba97b
>>> import time
>>> time.localtime()
(2025, 4, 20, 19, 52, 46, 6, 110)
>>> import builtins
>>> builtins.neorv32.help()
neorv32 - helper functions:
  gpio_pin_set(pin, level)       - Set GPIO.output [pin] to [level]
  gpio_pin_toggle(pin)           - Toggle GPIO.output [pin]
  systick_set_callback(callback) - Call [callback] from SysTICK IRQ
  help()                         - Show this text
```

Chapter 13. Using the On-Chip Debugger

The NEORV32 on-chip debugger ("OCD") allows online in-system debugging via an external JTAG access port. 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: `make gdb`

OCD CPU Requirements

The on-chip debugger is only implemented if the `OCD_EN` generic is set *true*.

13.1. Hardware Requirements

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 breakout adapter.

Table 2. JTAG pin mapping

NEORV32 top signal	JTAG signal	Default FTDI port
`jtag_tck_i`	TCK	D0
`jtag_tdi_i`	TDI	D1
`jtag_tdo_o`	TDO	D2
`jtag_tms_i`	TMS	D3

JTAG TAP Reset

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

13.2. 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`

Interface Configuration

You might need to adapt the default interface configuration in `sw/openocd/interface.cfg` according to your JTAG adapter and your operating system.

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

Listing 12. Connecting via OpenOCD (on Windows) using the default `openocd_neorv32.cfg` script

```
neorv32\sw\openocd>openocd.exe -f openocd_neorv32.cfg
xPack Open On-Chip Debugger 0.12.0+dev-01850-geb6f2745b-dirty (2025-02-07-10:08)
Licensed under GNU GPL v2
For bug reports, read
        http://openocd.org/doc/doxygen/bugs.html
*****************************************
NEORV32 single-core openOCD configuration
*****************************************
Error: libusb_open() failed with LIBUSB_ERROR_NOT_FOUND
Info : clock speed 2000 kHz
Info : JTAG tap: neorv32.cpu tap/device found: 0x00000001 (mfg: 0x000 (<invalid>),
part: 0x0000, ver: 0x0)
Error: Debugger is not authenticated to target Debug Module. (dmstatus=0x3). Use
`riscv authdata_read` and `riscv authdata_write` commands to authenticate.
Info : [neorv32.cpu] Examination succeed
Info : [neorv32.cpu] starting gdb server on 3333
Info : Listening on port 3333 for gdb connections
Info : authdata_write resulted in successful authentication
Info : datacount=1 progbufsize=2
Info : Disabling abstract command reads from CSRs.
Info : Examined RISC-V core; found 2 harts
Info :  hart 0: XLEN=32, misa=0x40901107
Authentication passed.
Info : JTAG tap: neorv32.cpu tap/device found: 0x00000001 (mfg: 0x000 (<invalid>),
part: 0x0000, ver: 0x0)
Target RESET and HALTED. Ready for remote connections.
Info : Listening on port 6666 for tcl connections
Info : Listening on port 4444 for telnet connections
```

Now the processor should be reset, halted and openOCD is waiting for connections.

13.3. Debugging with GDB

System View Description (SVD)

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

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

At first, the application needs to be compiled. Navigate to `sw/example/demo_blink_led` and compile the application:

Listing 13. Compile the test application

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

Beyond others, this will generate an ELF file `main.elf` that contains all the symbols required for debugging. Open another terminal in `sw/example/demo_blink_led` and start `gdb`.

Listing 14. 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 execution.

Listing 15. 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) c ⑤
```

① Connect to OpenOCD

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

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

④ Upload the executable

⑤ Start execution

You can halt execution at any time by `CTRL+c`. Then you can inspect the code, dump and alter variables, set breakpoints, step through the code, etc.

13.4. 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
- Excellent tutorial: https://www.emb4fun.com/riscv/ses4rv/index.html

Chapter 14. NEORV32 in Verilog

If you are more of a Verilog fan or if your EDA toolchain does not support VHDL/mixed-language designs you can use GHDL to convert an **all-Verilog** version of NEORV32. GHDL's synthesis feature is used to convert a pre-configured NEORV32 setup - including all peripherals, memories and memory images - into a single plain-Verilog module.

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 neorv32_verilog_wrapper.vhd that is used to configure the processor (using VHDL generics) and to customize the interface ports. After conversion, a single Verilog file is generated neorv32_verilog_wrapper.v`) that contains the whole NEORV32 processor. The original processor module hierarchy is preserved as well as all module interface names, which allows inspection/debugging of simulation waveforms and synthesis results.

The actual conversion is conducted by a single Makefile, which analyzes all the processor's sources and finally calls GHDL `synth` to create the final Verilog code. After conversion, the interface of the resulting `neorv32_verilog_wrapper` Verilog module is shown in the console which can be used as instantiation template.

```
neorv32/rtl/verilog$ make convert
Converting to Verilog: neorv32_verilog_wrapper.vhd -> neorv32_verilog_wrapper.v

...

------------------------------------------------
Verilog instantiation prototype
------------------------------------------------
module neorv32_verilog_wrapper
  (input  clk_i,
   input  rstn_i,
   input  uart0_rxd_i,
   output uart0_txd_o);
------------------------------------------------
```

Conversion Notes

- GHDL synthesis generates an un-optimized plain Verilog code without any (technology-specific) primitives. However, optimizations will be performed by the technology-specific synthesis tool.
- The output of the GHDL synthesis is a *post-elaboration* result. Therefore, all the processor's configuration options (i.e. VHDL generics) are resolved **before** the actual output is generated. Hence, the entity of the conversion wrapper must not have any generic.
- The interface of the resulting Verilog module lists all inputs first followed by all outputs.

- The original module hierarchy is preserved as well as all module interface names and many internal signal names.
- VHDL records are collapsed into linear arrays.

14.1. Verilog Simulation

A simple Verilog testbench is provided (`neorv32/rtl/verilog/testbench.v`) to simulate the default conversion output. The testbench includes a UART receiver connected to the processor's UART0; received characters are send to the simulator console. The simulation can be started right from the Makefile and it supports [Icarus Verilog](https://github.com/steveicarus/iverilog) and [Verilator](https://github.com/verilator/verilator) as simulators:

- Icarus Verilog: `neorv32/rtl/verilog$ make SIMULATOR=iverilog sim`
- Verilator: `neorv32/rtl/verilog$ make SIMULATOR=verilator sim`

As the default VHDL wrapper enabled the build-in bootloader, the simulation UART receiver checks for the the bootloader intro string ("NEORV32"). As soon as this string has been received `Simulation successful!` is printed to the console.

Listing 16. All-Verilog simulation with Icarus Verilog

```
Running simulation with Icarus Verilog

NEORV32 Verilog testbench





NEORV32
Simulation successful!
testbench.v:83: $finish called at 81865950 (100ps)
```

Generating Waveform Data

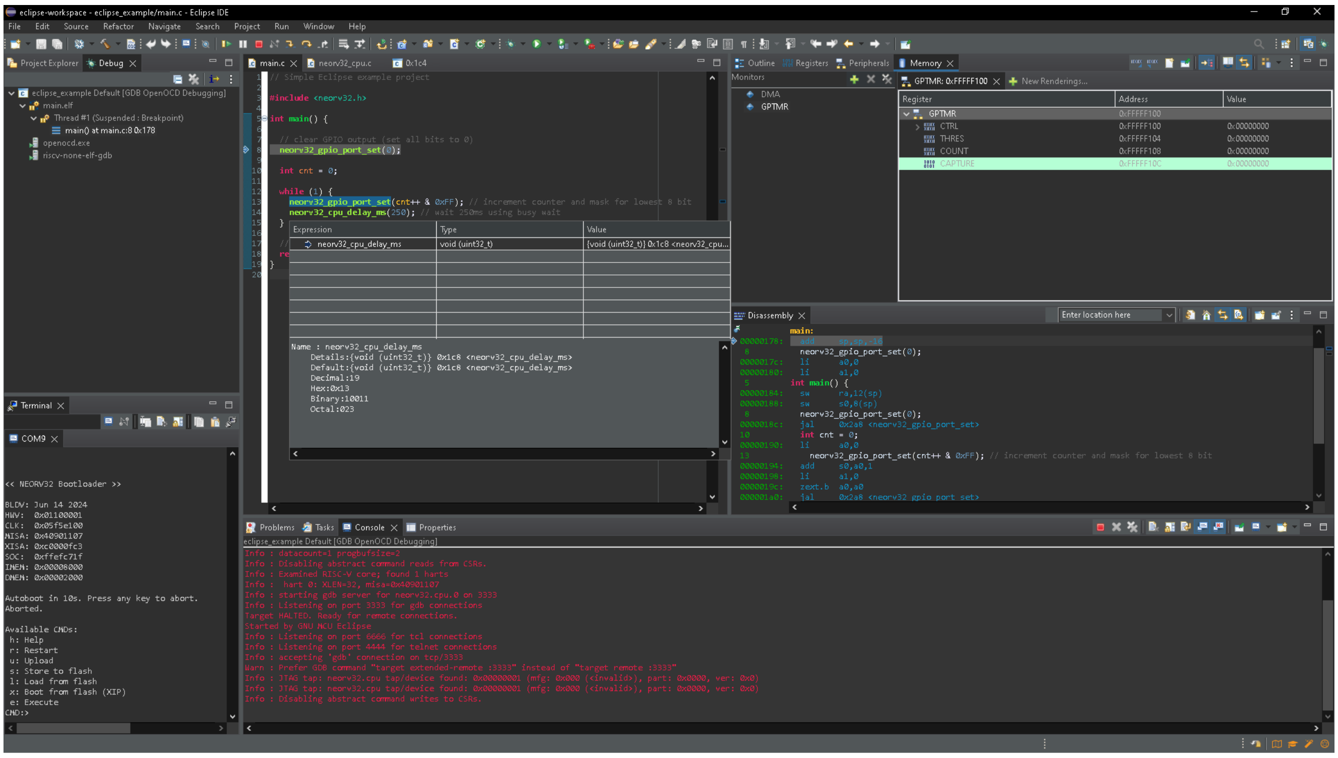
The Verilog simulator will emit waveform data if the `DUMP_WAVE` variable is set to `1`. Example: `neorv32-verilog$ make SIMULATOR=verilator DUMP_WAVE=1 sim` Waveform data is stored as `wave.fst`.

Chapter 15. Eclipse IDE

Eclipse (https://www.eclipse.org/) is a free and open-source 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 Eclipse 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**.



```
eclipse_example Default [GDB OpenOCD Debugging]
  main.elf
    Thread #1 (Suspended : Breakpoint)
      main() at main.c:8 0x178
  openocd.exe
  riscv-none-elf-gdb
```

```c
// Simple Eclipse example project

#include <neorv32.h>

int main() {

  // clear GPIO output (set all bits to 0)
  neorv32_gpio_port_set(0);

  int cnt = 0;

  while (1) {
    neorv32_gpio_port_set(cnt++ & 0xFF); // increment counter and mask for lowest 8 bit
    neorv32_cpu_delay_ms(250); // wait 250ms using busy wait
  }

  //
  re
}
```

Expression	Type	Value
neorv32_cpu_delay_ms	void (uint32_t)	{void (uint32_t)} 0x1c8 <neorv32_cpu...

```
Name : neorv32_cpu_delay_ms
    Details:{void (uint32_t)} 0x1c8 <neorv32_cpu_delay_ms>
    Default:{void (uint32_t)} 0x1c8 <neorv32_cpu_delay_ms>
    Decimal:19
    Hex:0x13
    Binary:10011
    Octal:023
```

Register	Address	Value
GPTMR	0xFFFFF100	
CTRL	0xFFFFF100	0x00000000
THRES	0xFFFFF104	0x00000000
COUNT	0xFFFFF108	0x00000000
CAPTURE	0xFFFFF10C	0x00000000

```
          main:
00000178:   add     sp,sp,-16
 8          neorv32_gpio_port_set(0);
0000017c:   li      a0,0
00000180:   li      a1,0
 5        int main() {
00000184:   sw      ra,12(sp)
00000188:   sw      s0,8(sp)
 8          neorv32_gpio_port_set(0);
0000018c:   jal     0x2a8 <neorv32_gpio_port_set>
10          int cnt = 0;
00000190:   li      a0,0
13            neorv32_gpio_port_set(cnt++ & 0xFF); // increment counter and mask for lowest 8 bit
00000194:   add     s0,a0,1
00000198:   li      a1,0
0000019c:   zext.b  a0,a0
000001a0:   jal     0x2a8 <neorv32_gpio_port_set>
```

```
<< NEORV32 Bootloader >>

BLDV: Jun 14 2024
HWV:  0x01100001
CLK:  0x05f5e100
MISA: 0x40901107
XISA: 0xc0000fc3
SOC:  0xffefc71f
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
 x: Boot from flash (XIP)
 e: Execute
CMD:>
```

```
eclipse_example Default [GDB OpenOCD Debugging]
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.
Started by GNU MCU Eclipse
Info : Listening on port 6666 for tcl connections
Info : Listening on port 4444 for telnet connections
Info : accepting 'gdb' connection on tcp/3333
Warn : Prefer GDB command "target extended-remote :3333" instead of "target remote :3333"
Info : JTAG tap: neorv32.cpu tap/device found: 0x00000001 (mfg: 0x000 (<invalid>), part: 0x0000, ver: 0x0)
Info : JTAG tap: neorv32.cpu tap/device found: 0x00000001 (mfg: 0x000 (<invalid>), part: 0x0000, ver: 0x0)
Info : Disabling abstract command writes to CSRs.
```

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

15.1. 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

Some NEORV32 makefile targets 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`.

15.2. 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

15.3. 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.

15.3.1. 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**.

15.3.2. 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.

15.3.3. 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

15.3.4. 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**.

15.3.5. 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**.

15.3.6. 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**.

15.3.7. 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 he 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 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**.

15.3.8. 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.

15.4. 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
- Eclipse OpenOCD Plug-In: https://eclipse-embed-cdt.github.io/debug/openocd/

Chapter 16. Legal

About

The NEORV32 RISC-V Processor
https://github.com/stnolting/neorv32
Stephan Nolting, M.Sc.
🇪🇺 European Union
stnolting@gmail.com

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

License

SPDX Identifier

`SPDX-License-Identifier: BSD-3-Clause`

Proprietary Notice

Disclaimer

Limitation of Liability for External Links

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.