The NEORV32 RISC-V Processor

User Guide

by Stephan Nolting (M.Sc.)

Version v1.8.4-r0-g7aa10cf4

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

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
- configure RISC-V ISA extensions and optimizing the core for your application
- add custom hardware extensions and customizing the bootloader
- program an external SPI flash for persistent application storage
- generate a Xilinx 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

Chapter 1. Software Toolchain Setup

To compile (and debug) executables for the NEORV32 a RISC-V toolchain is required. There are two possibilities to get this:

1. Download and *build* the official RISC-V GNU toolchain yourself.
2. Download and install a prebuilt version of the toolchain; this might also done via the package manager / app store of your OS

The default toolchain prefix (`RISCV_PREFIX` variable) for this project is **`riscv32-unknown-elf-`**. Of course you can use any other RISC-V toolchain (like `riscv64-unknown-elf-`) that is capable to emit code for a `rv32` architecture. Just change `RISCV_PREFIX` according to your needs.

1.1. Building the Toolchain from Scratch

To build the toolchain by yourself you can follow the guide from the official https://github.com/riscv-collab/riscv-gnu-toolchain GitHub page. You need to make sure the generated toolchain fits the architecture of the NEORV32 core. To get a toolchain that even supports minimal ISA extension configurations, it is recommend to compile for `rv32i` only. Please note that this minimal ISA also provides further ISA extensions like `m` or `c`. Of course you can use a *multilib* approach to generate toolchains for several target ISAs at once.

Listing 1. Configuring GCC build for `rv32i` (minimal ISA)

```
riscv-gnu-toolchain$ ./configure --prefix=/opt/riscv --with-arch=rv32i --with
-abi=ilp32
riscv-gnu-toolchain$ make
```

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

Make sure to use "newlib" as C standard library as the NEORV32 has no native support for Linus-like operating systems.

1.2. Downloading and Installing a Prebuilt Toolchain

Alternatively, you can download a prebuilt toolchain.

1.2.1. Use The Toolchain I have Build

I have compiled a GCC toolchain on a 64-bit x86 Ubuntu (Ubuntu on Windows, actually) and uploaded it to GitHub. You can directly download the according toolchain archive as single *zip-file* within a packed release from https://github.com/stnolting/riscv-gcc-prebuilt.

Unpack the downloaded toolchain archive and copy the content to a location in your file system (e.g. `/opt/riscv`). More information about downloading and installing my prebuilt toolchains can be found in the repository's README.

1.2.2. Use a Third Party Toolchain

Of course you can also use any other prebuilt version of the toolchain. There are a lot RISC-V GCC packages out there - even for Windows. On Linux system you might even be able to fetch a toolchain via your distribution's package manager.

Make sure the toolchain can (also) emit code for a `rv32i` architecture, uses the `ilp32` or `ilp32e` ABI and **was not build** using CPU extensions that are not supported by the NEORV32 (like `D`).

1.3. Installation

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

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

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

1.4. Testing the Installation

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

```
neorv32/sw/example/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.

Chapter 2. General Hardware Setup

This guide shows the basics of setting up a NEORV32 project for FPGA implementation (or simulation only) *from scratch*. It uses a *simplified* test "SoC" setup of the processor to keeps things simple at the beginning. This simple setup is intended for evaluation or as "hello world" project to check out the NEORV32 on *your* FPGA board.

If you want to use a more sophisticated pre-defined setup to start with, check out the `setups` folder, which provides example setups for various FPGA, boards and toolchains.

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. The main difference between the setups is the processor boot concept - so how to get a software executable *into* the processor:

- `rtl/test_setups/neorv32_testsetup_approm.vhd`: this setup does not require a connection via UART. The software executable is "installed" into the bitstream to initialize a read-only memory. Use this setup if your FPGA board does *not* provide a UART interface.
- `rtl/test_setups/neorv32_testsetup_bootloader.vhd`: this setups uses the UART and the default NEORV32 bootloader to upload new software executables. Use this setup if your board *does* provide a UART interface.
- `rtl/test_setups/neorv32_testsetup_on_chip_debugger.vhd`: besides the UARt bootloader, this setups uses on-chip debugger to upload and inspect new software executables. Use this setup if your board *does* provide a JTAG interface (the UART is optional).

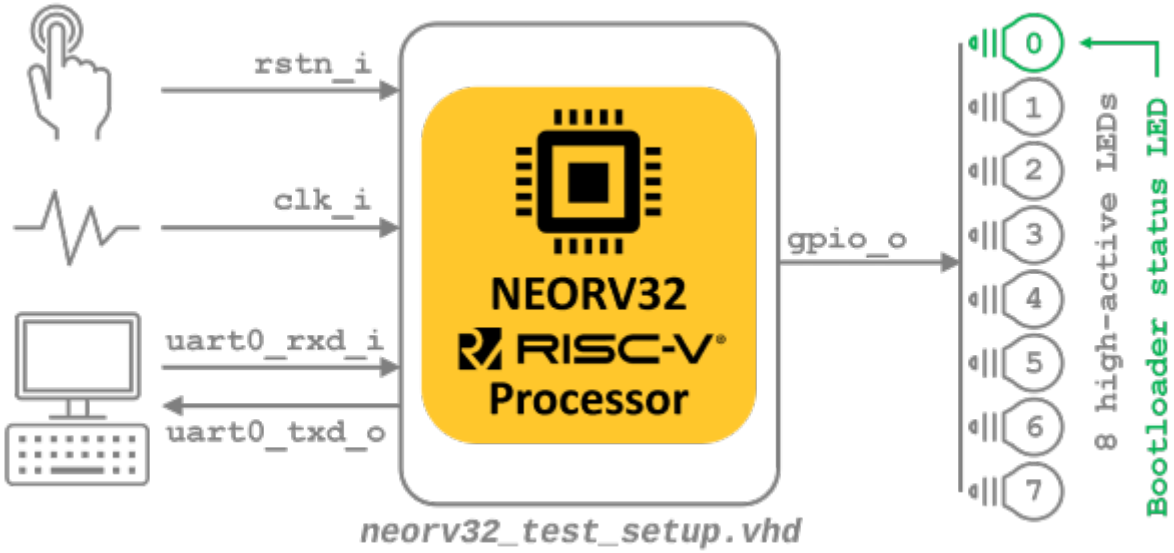

Figure 1. NEORV32 "hello world" test setup (`rtl/test_setups/neorv32_testsetup_bootloader.vhd`)

External Clock Source

These test setups are intended to be directly used as **design top entity**. Of course

you can also instantiate them into another design unit. If your FPGA board only provides *very fast* external clock sources (like on the FOMU board) you might need to add clock management components (PLLs, DCMs, MMCMs, ...) to the test setup or to the according top entity if you instantiate one of the test setups.

1. Create a new project with your FPGA EDA tool of choice.
2. Add all VHDL files from the project's `rtl/core` folder to your project.

Internal Memories

For a *general* first setup (technology-independent) use the `*.default.vhd` memory architectures for the internal memories (IMEM and DMEM). These are located in `rtl/core/mem` so make sure to add the files to your project, too.

If synthesis cannot efficiently map those default memory descriptions to the available memory resources, you can later replace the default memory architectures by optimized platform-specific memory architectures. **Example:** The `neorv32-setups/radiant/UPduino_v3` example setup uses optimized memory primitives. Hence, it does not include the default memory architectures from `rtl/core/mem` as these are replaced by device-specific implementations. However, it still has to include the entity definitions from `rtl/core`.

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

Compile order

Some tools (like Lattice Radiant) might require a *manual compile order* of the VHDL source files to identify the dependencies. The package file `neorv32_package.vhd` should be analyzed first followed by the memory image files (`neorv32_application_imagevhd` and `neorv32_bootloader_image.vhd`) and the entity-only files (`neorv32_*mem.entity.vhd`).

4. The `rtl/core/neorv32_top.vhd` VHDL file is the top entity of the NEORV32 processor, which can be instantiated into the "real" 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.

5. Add the pre-defined test setup of choice to the project, too, and select it as *top entity*.
6. The entity of both test setups provide 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

7. If you feel like it - or if your FPGA does not provide sufficient resources - you can modify the *memory sizes* (`MEM_INT_IMEM_SIZE` and `MEM_INT_DMEM_SIZE` - marked with notes "2" and "3"). But as mentioned above, let's keep things simple at first and use the standard configuration for now.
8. 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).

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.

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

Listing 3. Entity signals of `neorv32_testsetup_approm.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
);
```

Listing 4. Entity signals of `neorv32_testsetup_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 can add `not` gates. Example: The reset signal `rstn_i` is low-active by default; the LEDs connected to `gpio_o` high-active by default. You can do this in your board top if you instantiate the test setup, or *inside* the test setup if this is your top entity (low-active LEDs example: `gpio_o <= NOT con_gpio_o(7 downto 0);`).

10. 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.
11. If possible, connected *at least* bit `0` of the GPIO output port `gpio_o` to a LED (see "Signal Polarity" note above).
12. 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. USB-UART converter).
13. If your are using the on-chip debugger setup connect the processor's JTAG signal `jtag_*` to a suitable JTAG adapter.
14. Perform the project HDL compilation (synthesis, mapping, bitstream generation).
15. Program the generated bitstream into your FPGA and press the button connected to the reset signal.
16. Done! The LED(s) connected to `gpio_o` should be flashing now.

Going Further

Now that the hardware is ready, you can advance to one of these chapters to learn how to get a software executable into your processor setup (setup the GCC toolchain before; next section General Software Framework Setup):

`neorv32_testsetup_approm.vhd`: Installing an Executable Directly Into Memory
`neorv32_testsetup_bootloader.vhd`: Uploading and Starting of a Binary Executable Image via UART
`neorv32_testsetup_on_chip_debugger.vhd`: Debugging using the On-Chip Debugger

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

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 5. 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 the `neorv32_ram_size` variable needs to 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

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 to make per-project memory configuration without changing the code base.

The RAM and ROM sizes from Modifying the Linker Script (as well as the 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

Listing 6. Example: override default RAM size while invoking make

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

The `-Wl` will pass the following commands/flags to the linker. `--defsym` will define a symbol for the linker. `neorv32_rom_size` is the variable that will be defined and `16k` is the value assigned to it (= 16*1024 bytes). As a result, this command will set the RAM region to a size of 16kB.

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.

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. 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. We are using the `clean_all` target to make sure everything is re-build.
4. This will compile and link the application sources together with all the included libraries. At the end, your application is transformed into an ELF file (`main.elf`). The *NEORV32 image generator* (in `sw/image_gen`) takes this file and creates a final executable. 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
```

Make sure the size of the `text` segment (3176 bytes here) does not overflow the size of the processor's IMEM (if used at all) - otherwise there will be an error during synthesis or during bootloader upload.

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.

The compilation process will also create a `main.asm` assembly listing file in the current folder, which shows the actual assembly code of the application.

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
   - receiver (host computer) newline on `\r\n` (carriage return & newline)
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).
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 7. 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 bytes @0x00000000
DMEM: 0x00002000 bytes @0x80000000

Autoboot in 8s. 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:>
```

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:

```
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.

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 8. 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 9. 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. Enabling RISC-V CPU Extensions

Whenever you enable/disable a RISC-V CPU extensions via the according `CPU_EXTENSION_RISCV_x` generic, you need to adapt the toolchain configuration so the compiler can actually generate according code for it.

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

```
# CPU architecture and ABI
MARCH ?= rv32i ①
MABI  ?= ilp32 ②
```

① MARCH = Machine architecture ("ISA string")

② MABI = Machine binary interface

For example, if you enable the RISC-V `C` extension (16-bit compressed instructions) via the `CPU_EXTENSION_RISCV_C` generic (set `true`) you need to add the `c` extension also to the `MARCH` ISA string in order to make the compiler emit compressed instructions.

Privileged Architecture Extensions

Privileged architecture extensions like `Zicsr` or `Zifencei` are "used" *implicitly* by the compiler. Hence, according instruction will only be generated when "encoded" via inline assembly or when linking according libraries. In this case, these instruction will *always* be emitted (even if the according extension is not specified in `MARCH`).
I recommend to *not* specify any privileged architecture extensions in `MARCH`.

ISA extension enabled in hardware can be a superset of the extensions enabled in software, but not the other way around. For example generating compressed instructions for a CPU configuration that has the `c` extension disabled will cause *illegal instruction exceptions* at runtime.

You can also override the default `MARCH` and `MABI` configurations from the makefile when invoking the makefile:

```
$ make MARCH=rv32ic clean_all all
```

The RISC-V ISA string for `MARCH` follows a certain *canonical* structure: `rev32[i/e][m][a][f][d][g][q][c][b][v][n]⋯` For example `rv32imac` is valid while `rv32icma` is not.

Chapter 9. Application-Specific Processor Configuration

Due to the processor's configuration options, which are mainly defined via the top entity VHDL generics, the SoC can be tailored 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).

9.1. 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 ⇒ true` to use DSP slices for multiplications, `FAST_SHIFT_EN ⇒ true` use a fast barrel shifter for shift operations.
- Implement the instruction cache: `ICACHE_EN ⇒ true`
- Use as many *internal* memory as possible to reduce memory access latency: `MEM_INT_IMEM_EN ⇒ true` and `MEM_INT_DMEM_EN ⇒ true`, maximize `MEM_INT_IMEM_SIZE` and `MEM_INT_DMEM_SIZE`
- Increase the CPU's instruction prefetch buffer size: if **no** instruction cache is implemented `CPU_IPB_ENTRIES` should be quite large
- *To be continued...*

9.2. 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 `IO_PWM_NUM_CH` or the external interrupt controller `XIRQ_NUM_CH`).
- Disable the instruction cache (`ICACHE_EN ⇒ 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).
- Reduce the CPU's prefetch buffer size (`CPU_IPB_ENTRIES`) to its minimum (=1).
- Map CPU shift operations to a small and iterative shifter unit (`FAST_SHIFT_EN ⇒ 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 ⇒ 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...*

9.3. 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.

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 ⇒ true`) reducing LUT usage and critical path impact while also increasing overall performance.
- Use the synchronous (registered) RX path configuration of the external memory interface (`MEM_EXT_ASYNC_RX ⇒ 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.

9.4. 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. Disable all processor modules that are not actually used (exclude them from synthesis if the will be *never* used; disable the module via it's control register if the module is not *currently* used). When is sleep mode, you can keep a timer module running (MTIME or the watch dog) to wake up the CPU again. Since the wake up is triggered by *any* interrupt, the external interrupt controller can also be used to wake up the CPU again. By this, all timers (and all other modules) can be deactivated as well.

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 10. 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.

10.1. 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. Device-specific interrupt capabilities could be added using the External Interrupt Controller (XIRQ).

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.

10.2. 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.

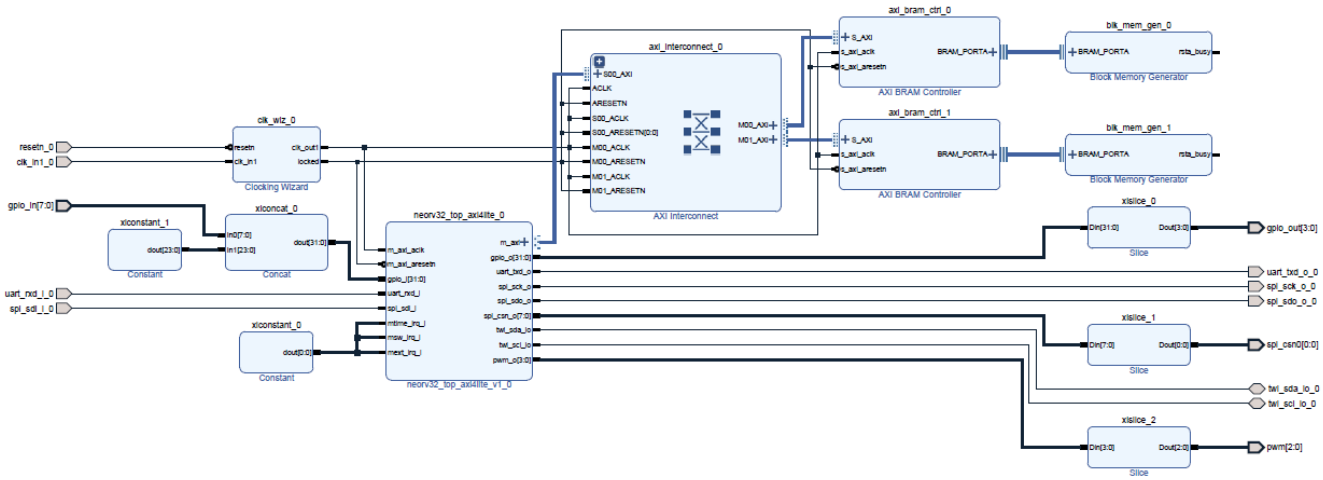

Figure 2. Example AXI SoC using Xilinx Vivado

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.

10.3. 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).

10.4. 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.

10.5. 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

	Custom Functions Unit	Custom Functions Subsystem	External Bus Interface
SoC 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
Arbitrary accesses	yes	yes	yes
Access latency	minimal	low	medium to high
Buffered access (e.g. FIFO)	no	user-defined	user-defined
External IO interfaces	no	yes, but limited	yes
Interrupt-capable	no	yes	user-defined

Chapter 11. 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 to 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 32kB and should be compiled using the minimal base + privileged ISA `rv32i_zicsr` only to ensure it can work independently of the actual CPU configuration.

Table 2. Bootloader configuration parameters

Parameter	Default	Legal values	Description
		Serial console interface	
`UART_EN`	`1`	`0`, `1`	Set to `0` to disable UART0 (no serial console at all)
`UART_BAUD`	`19200`	*any*	Baud rate of UART0
`UART_HW_HANDSHAKE_EN`	`0`	`0`, `1`	Set to `1` to enable UART0 hardware flow control
		Status LED	
`STATUS_LED_EN`	`1`	`0`, `1`	Enable bootloader status led ("heart beat") at `GPIO` output port pin #`STATUS_LED_PIN` when `1`
`STATUS_LED_PIN`	`0`	`0` ... `31`	`GPIO` output pin used for the high-active status LED
		Auto-boot configuration	
`AUTO_BOOT_TIMEOUT`	`8`	*any*	Time in seconds after the auto-boot sequence starts (if there is no UART input by the user); set to 0 to disabled auto-boot sequence
		SPI configuration	
`SPI_EN`	`1`	`0`, `1`	Set `1` to enable the usage of the SPI module (including load/store executables from/to SPI flash options)
`SPI_FLASH_CS`	`0`	`0` ... `7`	SPI chip select output (`spi_csn_o`) for selecting flash
`SPI_FLASH_ADDR_BYTES`	`3`	`2`, `3`, `4`	SPI flash address size in number of bytes (2=16-bit, 3=24-bit, 4=32-bit)
`SPI_FLASH_SECTOR_SIZE`	`65536`	*any*	SPI flash sector size in bytes

Parameter	Default	Legal values	Description
SPI_FLASH_CLK_PRSC	CLK_PRSC_8	CLK_PRSC_2 CLK_PRSC_4 CLK_PRSC_8 CLK_PRSC_64 CLK_PRSC_128 CLK_PRSC_1024 CLK_PRSC_2024 CLK_PRSC_4096	SPI clock pre-scaler (dividing main processor clock)
SPI_BOOT_BASE_ADDR	0x00400000	*any* 32-bit value	Defines the *base* address of the executable in external flash
XIP configuration			
XIP_EN	0	0, 1	Set 1 to enable the XIP options
XIP_PAGE_BASE_ADDR	0x40000000	*any* 32-bit value	Defines the page *base* address where the XP flash will be mapped to

The XIP options re-use the "SPI configuration" options for configuring the XIP's SPI connection.

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.

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 10. 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`).

11.1. 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 12. 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).

12.1. 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 p to trigger the programming of the flash (do not execute the image via the e command as this might corrupt the image):

```
CMD:> u
Awaiting neorv32_exe.bin... OK
CMD:> p
Write 0x000013FC bytes to SPI flash @ 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:> p
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 13. Packaging the Processor as IP block for Xilinx Vivado Block Designer

1. Import all the core files from `rtl/core` (including default internal memory architectures from `rtl/core/mem`) and assign them to a *new* design library `neorv32`.
2. Instantiate the `rtl/system_integration/neorv32_top_axi4lite.vhd` module.
3. Then either directly use that module in a new block-design ("Create Block Design", right-click → "Add Module", thats easier for a first try) or package it ("Tools", "Create and Package new IP") for the use in other projects.
4. Connect your AXI-peripheral directly to the core's AXI4-Interface if you only have one, or to an AXI-Interconnect (from the IP-catalog) if you have multiple peripherals.
5. Connect ALL the `ACLK` and `ARESETN` pins of all peripherals and interconnects to the processor's clock and reset signals to have a *unified* clock and reset domain (easier for a first setup).
6. Open the "Address Editor" tab and let Vivado assign the base-addresses for the AXI-peripherals (you can modify them according to your needs).
7. For all FPGA-external signals (like UART signals) make all the connections you need "external" (right-click on the signal/pin → "Make External").
8. Save everything, let VIVADO create a HDL-Wrapper for the block-design and choose this as your *Top Level Design*.
9. Define your constraints and generate your bitstream.

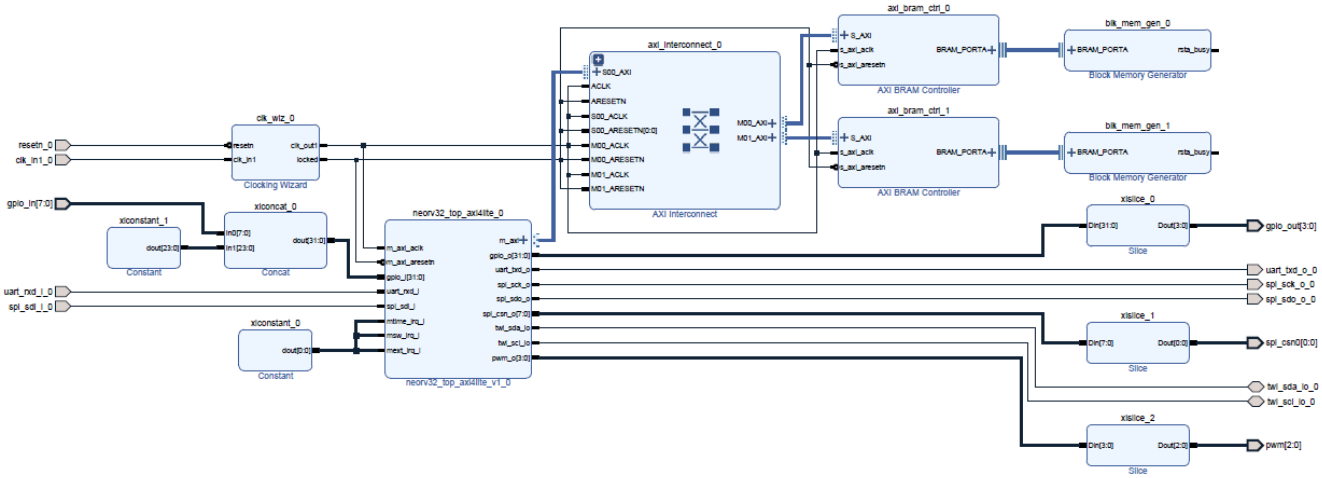

Figure 3. Example AXI SoC using Xilinx Vivado

True Random Number Generator

The NEORV32 TRNG peripheral is enabled by default in the `neorv32_top_axi4lite` AXI wrapper. Otherwise, Vivado cannot insert the wrapper into a block design (see https://github.com/stnolting/neorv32/issues/227.). [1] If the TRNG is not needed, you can disable it by double-clicking on the module's block and de-selecting

> "IO_TRNG_EN" after inserting the module.

Combinatorial Loops DRC error 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:

```
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
```

Guide provided by GitHub user `AWenzel83` (see https://github.com/stnolting/neorv32/discussions/52#discussioncomment-819013). □□

[1] Seems like Vivado has problem evaluating design source files that have more than two in-file sub-entities.

Chapter 14. Simulating the Processor

The NEORV32 project includes a core CPU, built-in peripherals in the Processor Subsystem, and additional peripherals in the templates and examples. Therefore, 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).

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

Xilinx Vivado / ISIM

When using Xilinx Vivado (ISIM for simulation) make sure to **turn of** "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

The processor can check if it is being *simulated* by checking the SYSINFO *SYSINFO_SOC_IS_SIM* flag (see https://stnolting.github.io/neorv32/#_system_configuration_information_memory_sysinfo). Note that this flag is not guaranteed to be set correctly (depending on the HDL toolchain's pragma support).

14.1. Testbench

A plain-VHDL (no third-party libraries) testbench (`sim/simple/neorv32_tb.simple.vhd`) can be used for simulating and testing the processor. This testbench features a 100MHz clock and enables all optional peripheral and CPU extensions except for the `E`.

True Random Number Generator

The NEORV32 TRNG will be set to "simulation mode" when enabled for simulation (replacing the ring-oscillators by pseudo-random LFSRs). See the neoTRNG documentation for more information.

The simulation setup is configured via the "User Configuration" section located right at the beginning of the testbench's architecture. Each configuration constant provides comments to explain the functionality.

Besides the actual NEORV32 Processor, the testbench also simulates "external" components that are connected to the processor's external bus/memory interface. These components are:

- an external instruction memory (that also allows booting from it)
- an external data memory

- an external memory to simulate "external IO devices"
- a memory-mapped registers to trigger the processor's interrupt signals

The following table shows the base addresses of these four components and their default configuration and properties:

Attributes:

- `r` = read
- `w` = write
- `e` = execute
- `8` = byte-accessible
- `16` = half-word-accessible
- `32` = word-accessible

Table 3. Testbench: processor-external memories

Base address	Size	Attributes	Description
`0x00000000`	`imem_size_c`	`r/w/e 8/16/32`	external IMEM (initialized with application image)
`0x80000000`	`dmem_size_c`	`r/w/e 8/16/32`	external DMEM
`0xf0000000`	64 bytes	`r/w/e 8/16/32`	external "IO" memory
`0xff000000`	4 bytes	`-/w/- -/-/32`	memory-mapped register to trigger "machine external", "machine software" and "SoC Fast Interrupt" interrupts

The simulated NEORV32 does not use the bootloader and *directly boots* the current application image (from the `rtl/core/neorv32_application_image.vhd` image file).

UART output during simulation

Data written to the NEORV32 UART0 / UART1 transmitter is send to a virtual UART receiver implemented as part of the testbench. Received chars are send to the simulator console and are also stored to a log file (`neorv32.testbench_uart0.out` for UART0, `neorv32.testbench_uart1.out` for UART1) inside the simulation'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.

14.2. Faster Simulation Console Output

When printing data via the 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 Documentation: 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 execution directory:

- `neorv32.uart?.sim_mode.text.out`: ASCII data.
- `neorv32.uart?.sim_mode.data.out`: all written 32-bit dumped as 8-char hexadecimal values.

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. To enable the simulation mode just compile and install your application and add *UART?_SIM_MODE* to the compiler's *USER_FLAGS* variable (do not forget the `-D` suffix flag):

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

The provided define will change the default UART0/UART1 setup function in order to set the simulation mode flag in the according UART's control register.

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

14.3. Simulation using a shell script (with GHDL)

To simulate the processor using *GHDL* navigate to the `sim/simple/` folder and run the provided shell script. Any arguments that are provided while executing this script are passed to GHDL. For example the simulation time can be set to 20ms using `--stop-time=20ms` as argument.

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

14.4. Simulation using Application Makefiles (In-Console with GHDL)

To directly compile and run a program in the console (using the default testbench and GHDL as simulator) you can use the `sim` makefile target. Make sure to use the UART simulation mode (`USER_FLAGS+=-DUART0_SIM_MODE` and/or `USER_FLAGS+=-DUART1_SIM_MODE`) to get faster / direct-to-console UART output.

```
sw/example/demo_blink_led$ make USER_FLAGS+=-DUART0_SIM_MODE clean_all sim
[...]
Blinking LED demo program
```


14.4.1. Hello World!

To do a quick test of the NEORV32 make sure to have GHDL and a RISC-V gcc toolchain installed. Navigate to the project's `sw/example/hello_world` folder and run `make USER_FLAGS+=-DUART0_SIM_MODE MARCH=rv32imc clean_all sim`:

The simulator will output some *sanity check* notes (and warnings or even errors if something is ill-configured) right at the beginning of the simulation to give a brief overview of the actual NEORV32 SoC and CPU configurations.

```
stnolting@Einstein:/mnt/n/Projects/neorv32/sw/example/hello_world$ make USER_FLAGS+=-
DUART0_SIM_MODE MARCH=rv32imc clean_all sim
../../../sw/lib/source/neorv32_uart.c: In function 'neorv32_uart0_setup':
../../../sw/lib/source/neorv32_uart.c:301:4: 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 simulations only! [-Wcpp]
  301 |   #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 simulations
only! ①
      |    ^~~~~~~
Memory utilization:
   text    data     bss     dec     hex filename
   4612       0     120    4732    127c main.elf ②
Compiling ../../../sw/image_gen/image_gen
Installing application image to ../../../rtl/core/neorv32_application_image.vhd ③
Simulating neorv32_application_image.vhd...
Tip: Compile application with USER_FLAGS+=-DUART[0/1]_SIM_MODE to auto-enable
UART[0/1]'s simulation mode (redirect UART output to simulator console). ④
Using simulation runtime args: --stop-time=10ms ⑤
../rtl/core/neorv32_top.vhd:347:3:@0ms:(assertion note): NEORV32 PROCESSOR IO
Configuration: GPIO MTIME UART0 UART1 SPI TWI PWM WDT CFS NEOLED XIRQ ⑥
../rtl/core/neorv32_top.vhd:370:3:@0ms:(assertion note): NEORV32 PROCESSOR CONFIG
NOTE: Boot configuration: Direct boot from memory (processor-internal IMEM).
../rtl/core/neorv32_top.vhd:394:3:@0ms:(assertion note): NEORV32 PROCESSOR CONFIG
NOTE: Implementing on-chip debugger (OCD).
../rtl/core/neorv32_cpu.vhd:169:3:@0ms:(assertion note): NEORV32 CPU ISA Configuration
(MARCH): RV32IMCU_Zbb_Zicsr_Zifencei_Zfinx_Debug
../rtl/core/neorv32_imem.vhd:107:3:@0ms:(assertion note): NEORV32 PROCESSOR CONFIG
NOTE: Implementing processor-internal IMEM as ROM (16384 bytes), pre-initialized with
application (4612 bytes).
../rtl/core/neorv32_dmem.vhd:89:3:@0ms:(assertion note): NEORV32 PROCESSOR CONFIG
NOTE: Implementing processor-internal DMEM (RAM, 8192 bytes).
../rtl/core/neorv32_wishbone.vhd:136:3:@0ms:(assertion note): NEORV32 PROCESSOR CONFIG
NOTE: External Bus Interface - Implementing STANDARD Wishbone protocol.
../rtl/core/neorv32_wishbone.vhd:140:3:@0ms:(assertion note): NEORV32 PROCESSOR CONFIG
NOTE: External Bus Interface - Implementing auto-timeout (255 cycles).
../rtl/core/neorv32_wishbone.vhd:144:3:@0ms:(assertion note): NEORV32 PROCESSOR CONFIG
NOTE: External Bus Interface - Implementing LITTLE-endian byte order.
```

```
../rtl/core/neorv32_wishbone.vhd:148:3:@0ms:(assertion note): NEORV32 PROCESSOR CONFIG
NOTE: External Bus Interface - Implementing registered RX path.
⑦

##

##         ##   ##   ##
 ##     ##   #########   ########    ########   ##      ##   ########    ########
##       #################
####    ##  ##          ##      ##  ##      ##  ##      ##  ##      ##  ##      ##
##     ####           ####
## ##   ##  ##          ##      ##  ##      ##  ##      ##          ##         ##
##       ##   ######   ##
##  ##  ##  #########   ##      ##  #########   ##      ##      #####        ##
##     ####   ######   ####
##   ## ##  ##          ##      ##  ##    ##     ##    ##          ##     ##
##       ##   ######   ##
##    ####  ##          ##      ##  ##     ##     ##  ##    ##      ##   ##
##     ####           ####
##     ##    #########   ########   ##      ##      ##       ########   ##########
##       #################

##         ##   ##   ##

##
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 A note regarding UART "simulation mode", but we have already enabled that.

⑤ List of (default) arguments that were send to the simulator. Here: maximum simulation time (10ms).

⑥ "Sanity checks" from the core's VHDL files. These reports give some brief information about the SoC/CPU configuration (→ generics). If there are problems with the current configuration, an ERROR will appear.

⑦ Execution of the actual program starts.

14.5. Advanced Simulation using VUnit

VUnit is an open source unit testing framework for VHDL/SystemVerilog. It allows continuous and automated testing of HDL code by complementing traditional testing methodologies. The motto of VUnit is *"testing early and often"* through automation.

VUnit is composed by a Python interface and multiple optional VHDL libraries. The Python interface allows declaring sources and simulation options, and it handles the compilation, execution and gathering of the results regardless of the simulator used. That allows having a single `run.py` script to be used with GHDL, ModelSim/QuestaSim, Riviera PRO, etc. On the other hand, the VUnit's VHDL libraries provide utilities for assertions, logging, having virtual queues, handling CSV files, etc. The Verification Component Library uses those features for abstracting away bit-toggling when verifying standard interfaces such as Wishbone, AXI, Avalon, UARTs, etc.

Testbench sources in `sim` (such as `sim/neorv32_tb.vhd` and `sim/uart_rx*.vhd`) use VUnit's VHDL libraries for testing NEORV32 and peripherals. The entry-point for executing the tests is `sim/run.py`.

```
# ./sim/run.py -l
neorv32.neorv32_tb.all
Listed 1 tests

# ./sim/run.py -v
Compiling into neorv32:   rtl/core/neorv32_uart.vhd
passed
Compiling into neorv32:   rtl/core/neorv32_twi.vhd
passed
Compiling into neorv32:   rtl/core/neorv32_trng.vhd
passed
...
```

See VUnit: User Guide and VUnit: Command Line Interface for further info about VUnit's features.

Chapter 15. VHDL Development Environment

To navigate and develop the NEORV32 processor VHDL code you can use the free and open source VHDL-LS language server. The easiest way to get started is to install the VHDL-LS VSCode extension. The VHDL-LS server requires a `vhdl_ls.toml` file which is automatically generated by the `sim/run.py` script. See Simulate the processor for further information.

1. Run `sim/run.py` to create the library mapping file
2. Install the VHDL-LS VSCode extension
3. Open the root folder of the NEORV32 repository in VSCode
4. Open any VHDL file

Chapter 16. 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 11. Example: Generate HTML documentation (data sheet) using `asciidoctor`

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

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

Chapter 17. Zephyr RTOS Support

The NEORV32 processor is supported by upstream Zephyr RTOS: https://docs.zephyrproject.org/latest/boards/riscv/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 18. FreeRTOS Support

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

Chapter 19. 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 Wishbone interface is used to with the 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

19.1. 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 netlist 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.

19.2. 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 20. Debugging using the On-Chip Debugger

The NEORV32 on-chip debugger 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`.

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

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 to be implemented (top generics *CPU_EXTENSION_RISCV_Zicsr* = *true* and *CPU_EXTENSION_RISCV_Zifencei* = *true*).

Segger Embedded Studio can also be used to develop and debug applications for the NEORV32 using the on-chip debugger.

20.1. 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 4. JTAG pin mapping

NEORV32 top signal	JTAG signal	FTDI port
`jtag_tck_i`	TCK	D0
`jtag_tdi_i`	TDI	D1
`jtag_tdo_o`	TDO	D2
`jtag_tms_i`	TMS	D3
`jtag_trst_i`	TRST	D4

The low-active JTAG tap reset `jtag_trst_i` signals is *optional* as a reset can also be triggered via the TAP controller issuing special commands. If `jtag_trst_i` is not connected make sure to pull the signal *high*.

20.2. 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 12. 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 fot `gdb` to connect via port 3333.

20.3. Debugging with GDB

The GNU debugger is part of the RISC-V GCC toolchain (see Software Toolchain

Setup).

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 13. 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 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 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 `mail.elf` by GDB and uploaded via the debugger's indirect memory access.

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)
```

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

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))`.

20.3.1. 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 16. 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 17. 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 18. 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

20.3.2. 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 19. 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.

20.4. 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 21. 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 an unoptimized plain-Verilog netlist 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 20. 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 netlist has been tested with Icarus Verilog (simulation) and Xilinx Vivado (simulation and synthesis).

For detailed information check out the `neorv32-verilog` repository at https://github.com/stnolting/neorv32-verilog.

Chapter 22. Legal

License

BSD 3-Clause License

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.

The NEORV32 RISC-V Processor
HQ: https://github.com/stnolting/neorv32
By Dipl.-Ing. (M.Sc.) Stephan Nolting
European Union, Germany
Contact: stnolting@gmail.com

Proprietary Notice

- "GitHub" is a Subsidiary of Microsoft Corporation.
- "Vivado" and "Artix" are trademarks of Xilinx Inc.
- "AXI", "AXI4-Lite" and "AXI4-Stream" are trademarks of Arm Holdings plc.
- "ModelSim" is a trademark of Mentor Graphics – A Siemens Business.
- "Quartus Prime" and "Cyclone" are trademarks of Intel Corporation.
- "iCE40", "UltraPlus" and "Radiant" are trademarks of Lattice Semiconductor Corporation.
- "Windows" is a trademark of Microsoft Corporation.
- "Tera Term" copyright by T. Teranishi.
- "NeoPixel" is a trademark of Adafruit Industries.
- Images/figures made with *Microsoft Power Point*.
- Timing diagrams made with *WaveDrom Editor*.
- Documentation proudly made with `asciidoctor`.
- "Segger Embedded Studio" and "J-Link" are trademarks of Segger Microcontroller Systems GmbH.
- All further/unreferenced products belong to their according copyright holders.

PDF icons from https://www.flaticon.com and made by Freepik, Good Ware, Pixel perfect, Vectors Market

Disclaimer

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

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 *

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

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

Listing 21. BibTeX

```
@misc{nolting22,
  author       = {Nolting, S. and ...},
  title        = {The NEORV32 RISC-V Processor},
  year         = {2022},
  publisher    = {GitHub},
  journal      = {GitHub repository},
  howpublished = {\url{https://github.com/stnolting/neorv32}}
}
```

DOI

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

Acknowledgments

**A big shout-out to the community and all contributors, who helped improving this project! **

RISC-V - instruction sets want to be free!

Continuous integration provided by GitHub Actions and powered by GHDL.