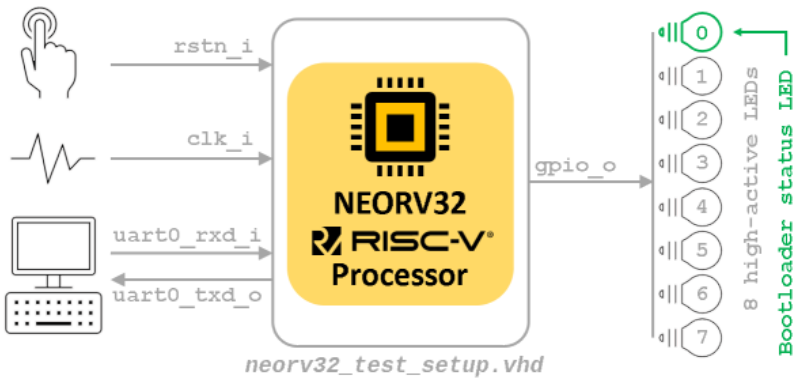
RISC-V tutorial

This tutorial will guide you to implement the RISC-V “neorv32” processor on the Zybo Z7 board and set the RISC-V compiler toolchain on Ubuntu.



You can find additional information for the NEORV-32 microarchitecture in the following links

[Datasheet] The NEORV32 RISC-V Processor

[User Guide] The NEORV32 RISC-V Processor

https://github.com/stnolting/neorv32/tree/v1.7.2 Can't find link

Hardware implementation

- First, we should download the VHDL source code of the NEORV-32.
- Open a terminal and type the following:

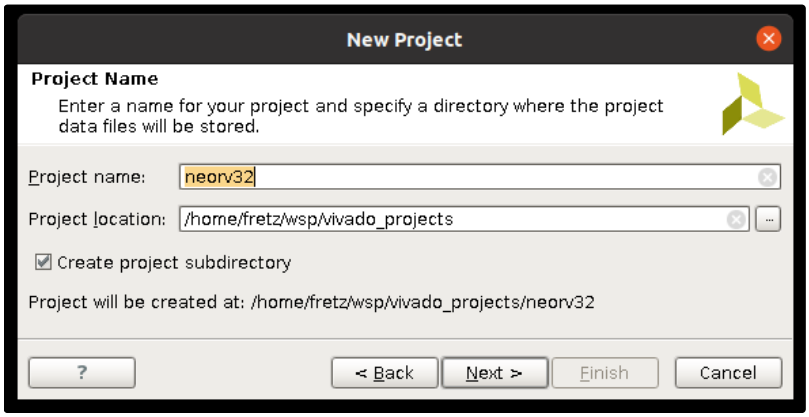
```
cd
mkdir -p wsp
cd wsp
git clone https://github.com/stnolting/neorv32.git
cd neorv32/
git checkout v1.8.0
```

- Beautiful, we now have the source code for NEORV-32. Let’s open Vivado to implement the processor
- Open Vivado

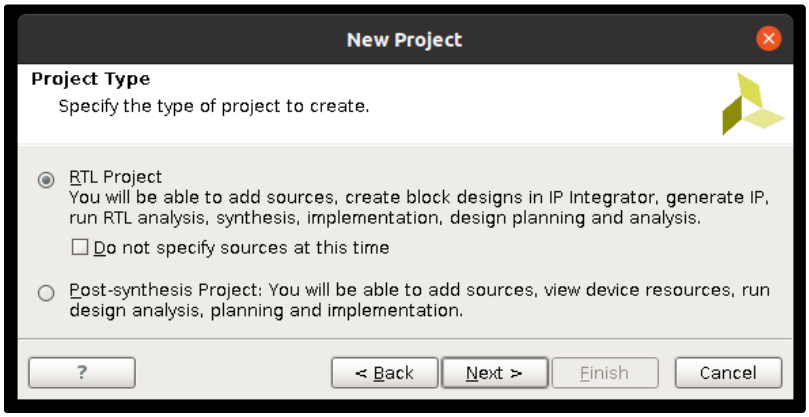
```
cd
cd wsp
mkdir -p vivado_projects
cd vivado_projects
source /opt/Xilinx/Vivado/2016.4/settings64.sh
vivado &
```

- Create a new project. Click from Vivado menu File → New project and click on the next button

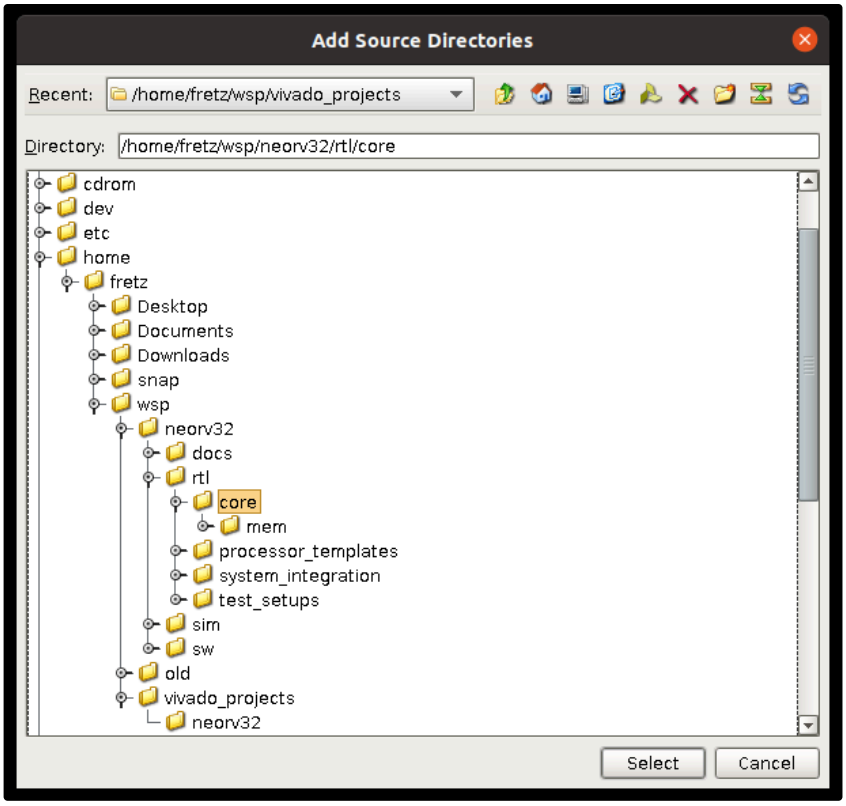
- Type neorv32 as the project name



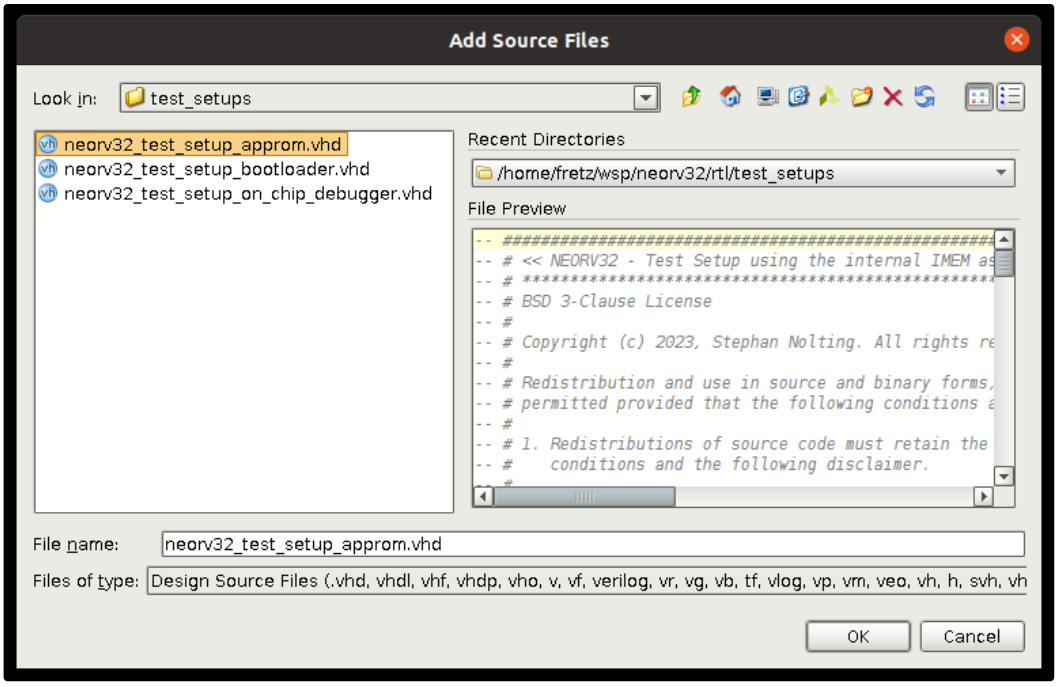
- Choose the RTL project



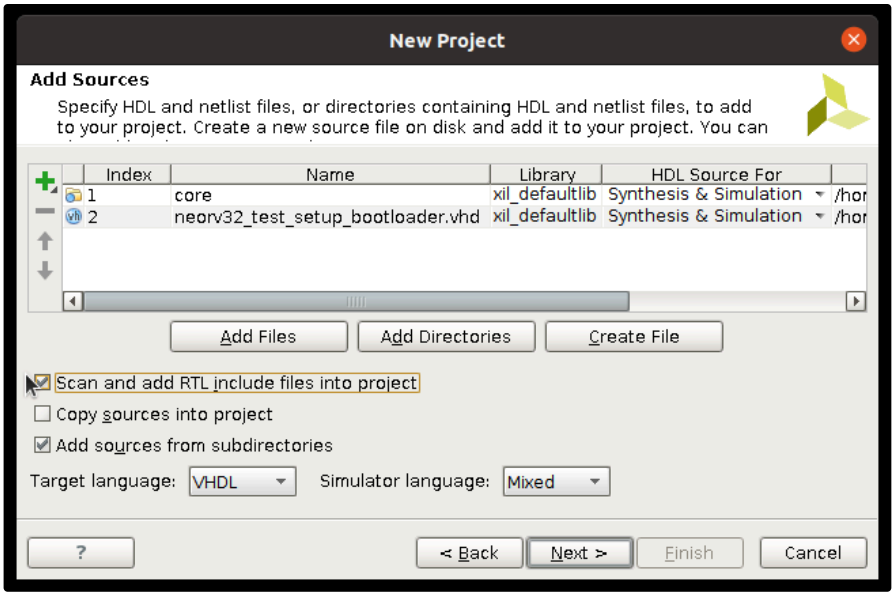
- Choose as target language **VHDL** and click on the **Add Directories** button to add the VHDL source code of the NEORV-32
- Choose the Directory → `/home/fretz/wsp/neorv32/rtl/core/`

- 

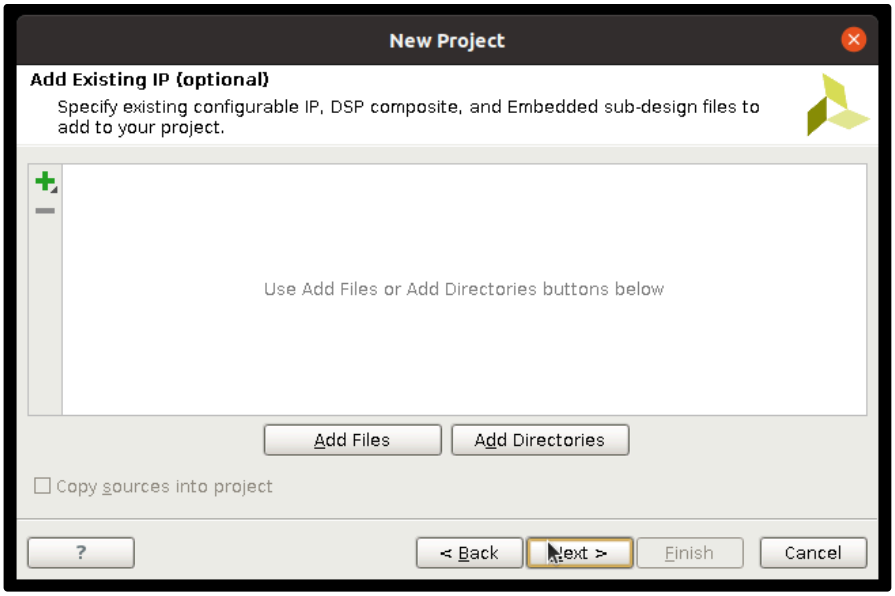
- Click on the **Add Files** button to add one more source code. Choose `/home/fretz/wsp/neorv32/rtl/test_setups/neorv32_test_setup_bootloader.vhd` and click on the OK button



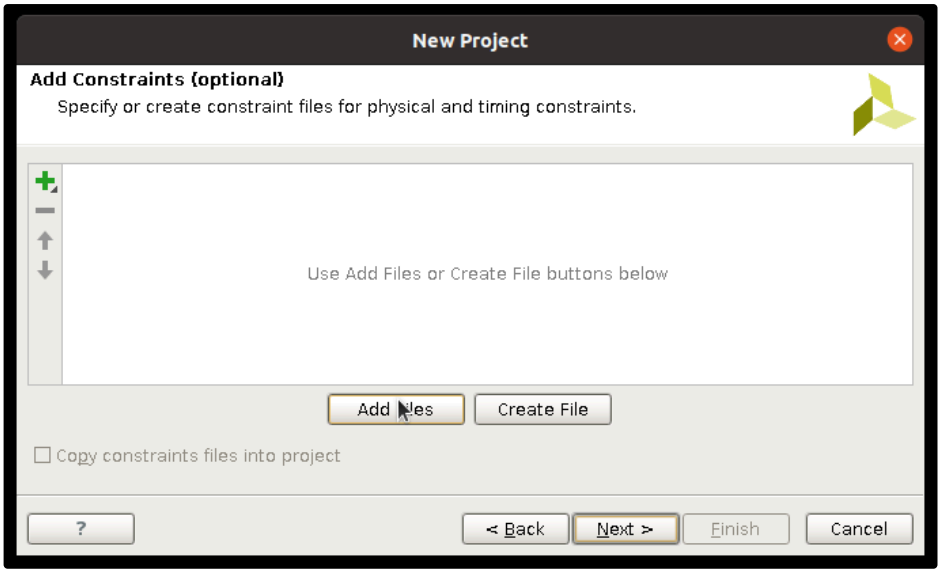
- Tick the Scan and add RTL include files into the project, and finally click on the NEXT button



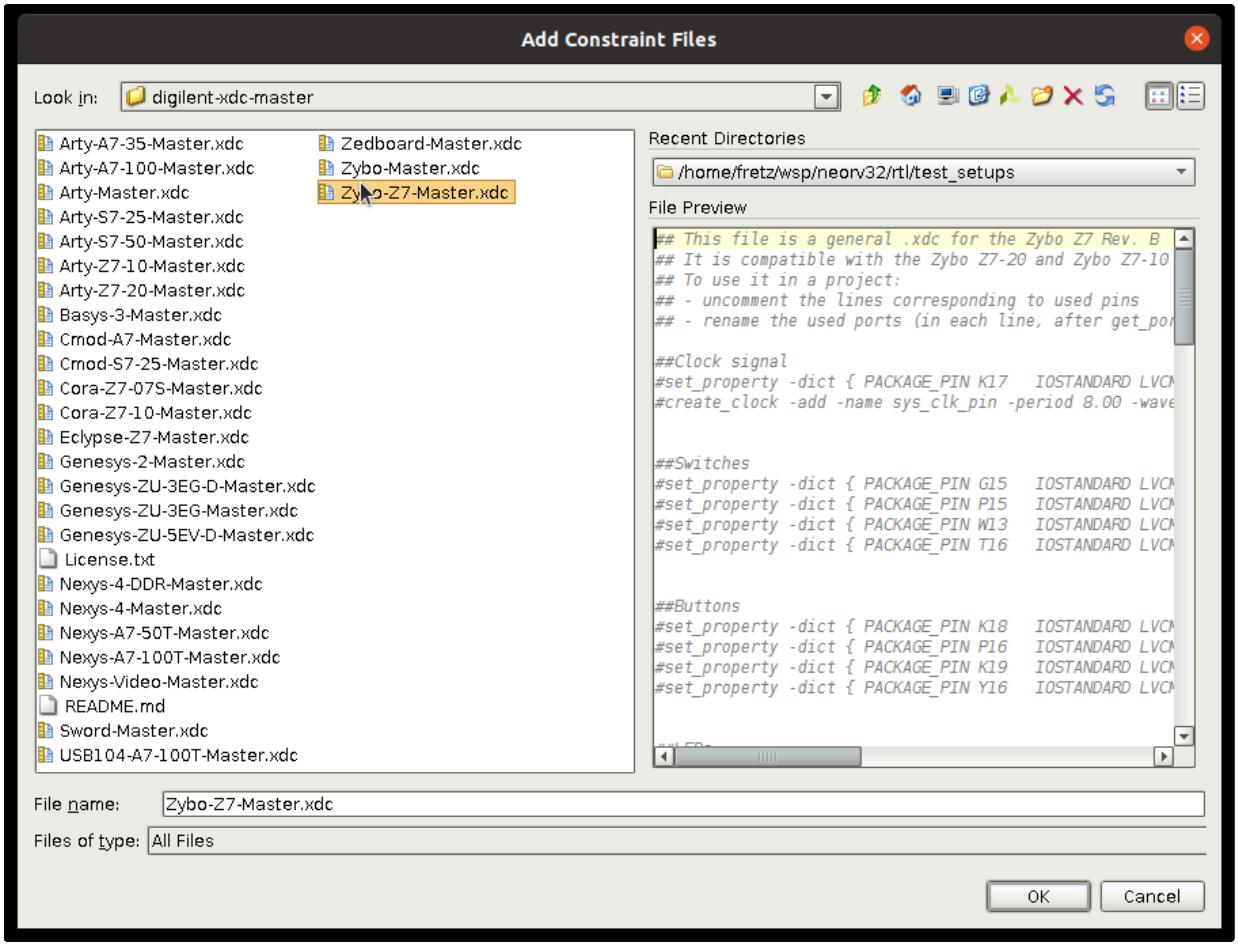
- Click next on the Add Existing IP (optional)



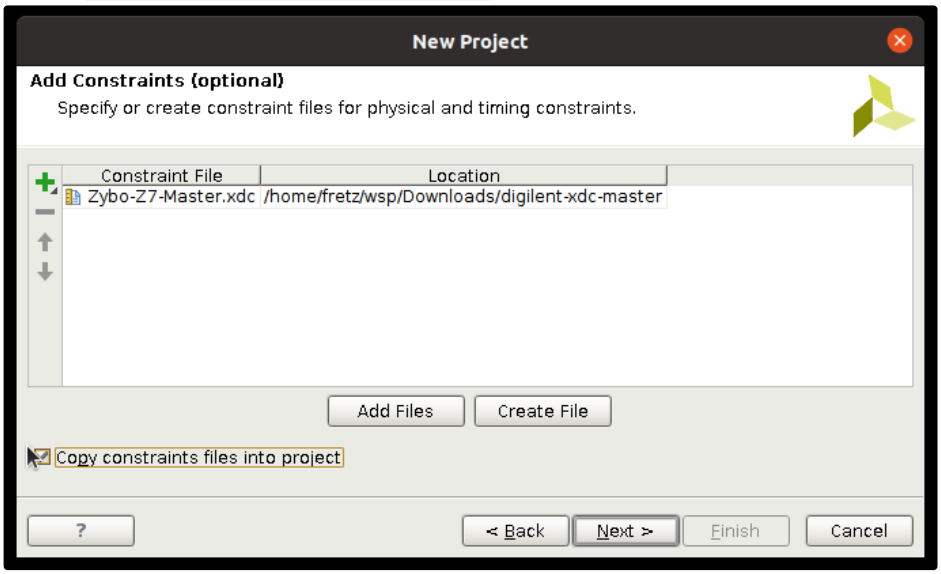
- Click on the Add Files on the Add Constraints window



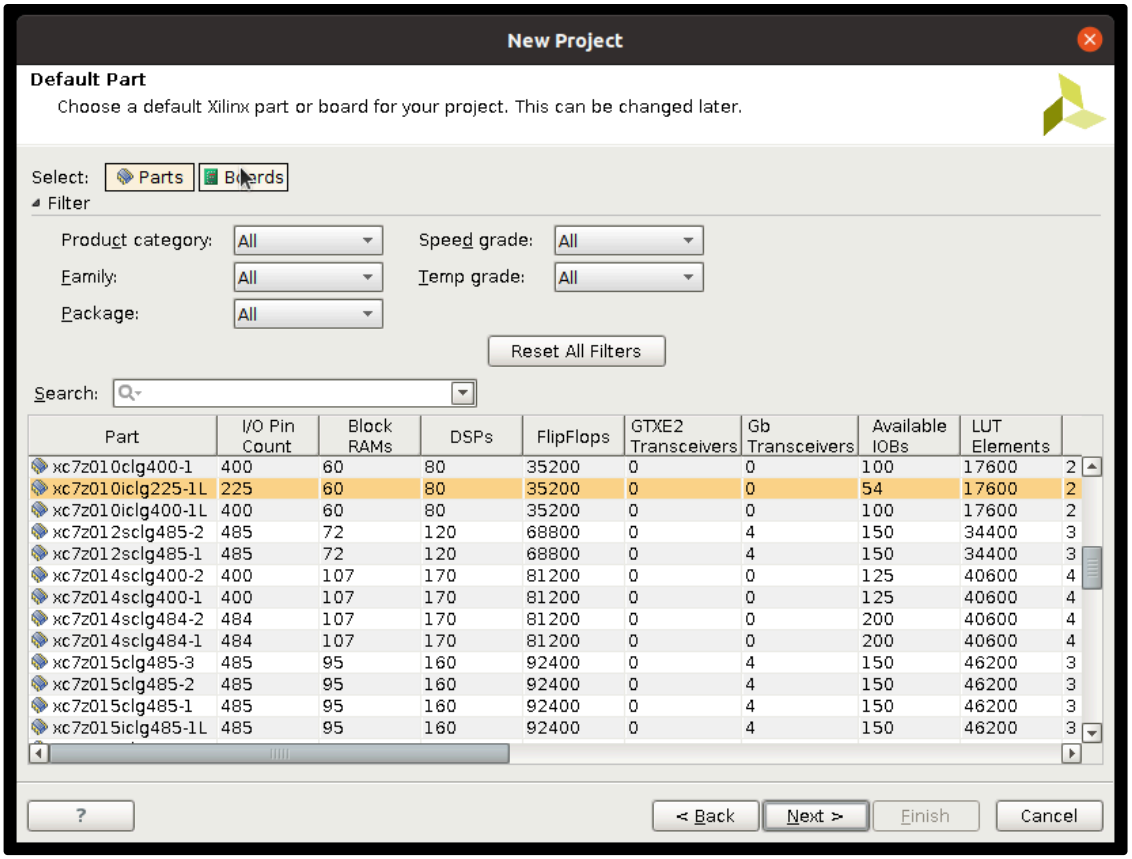
- Choose the `/home/fretz/wsp/Downloads/digilent-xdc-master/Zybo-Z7-Master.xdc` and press the OK button



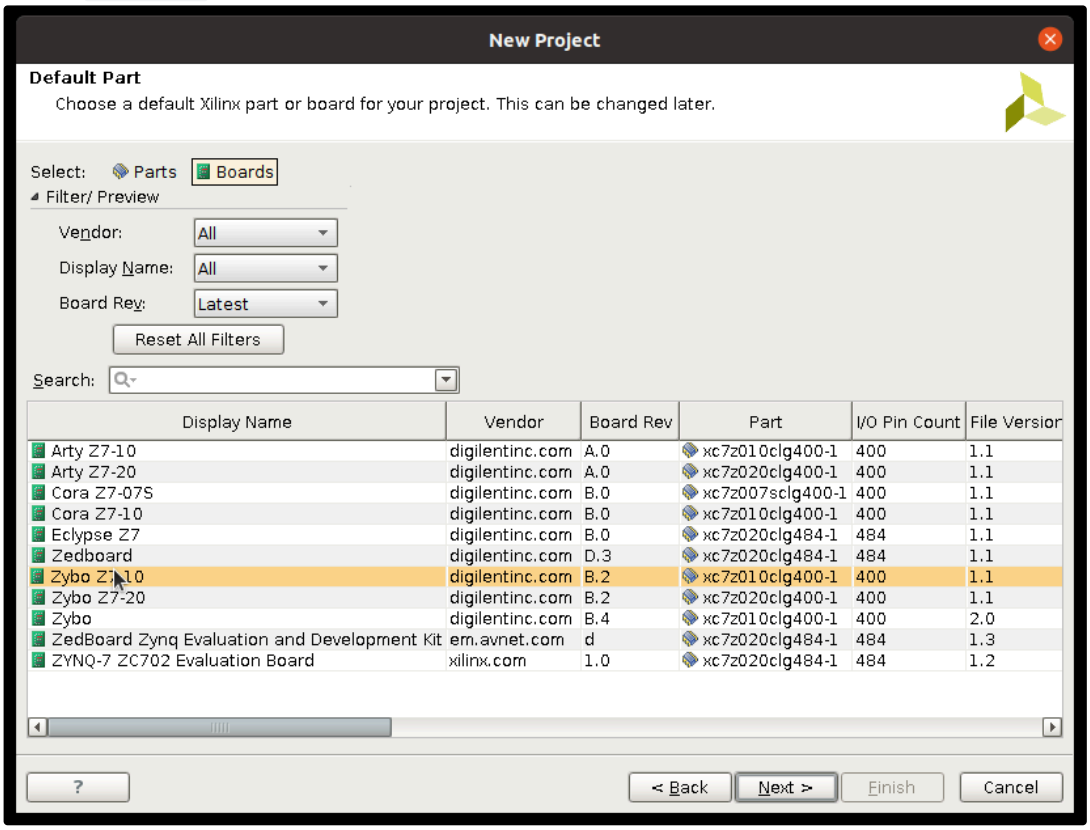
- Tick the `copy constraints files into project` and click the Next button



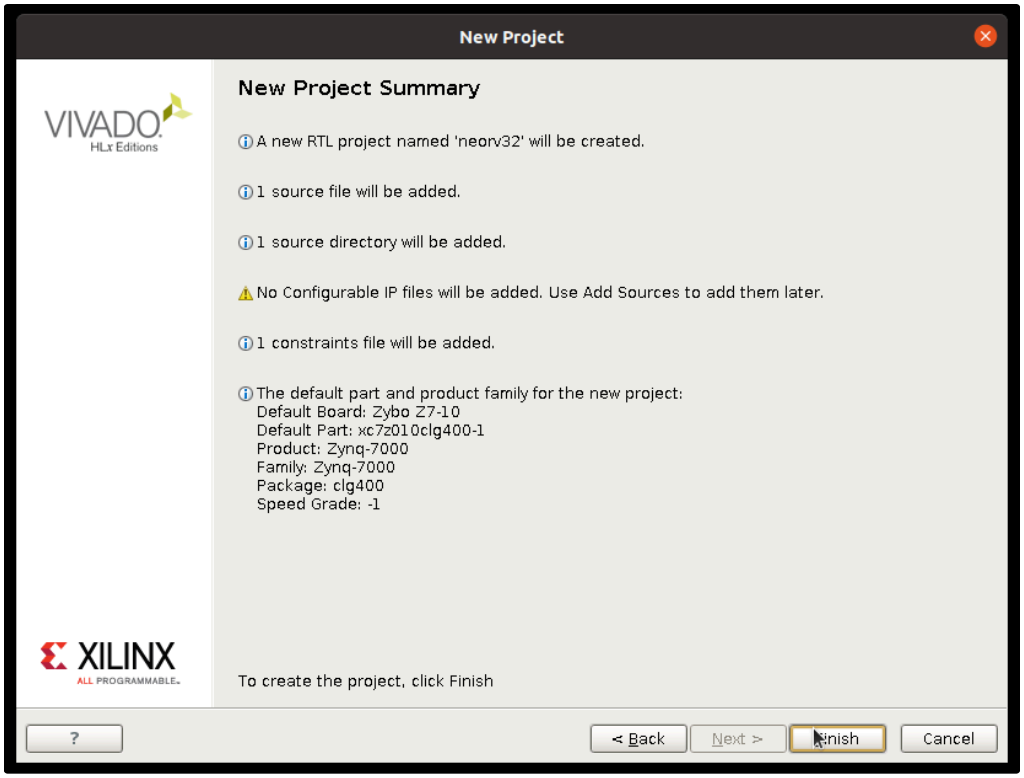
- Click on Boards



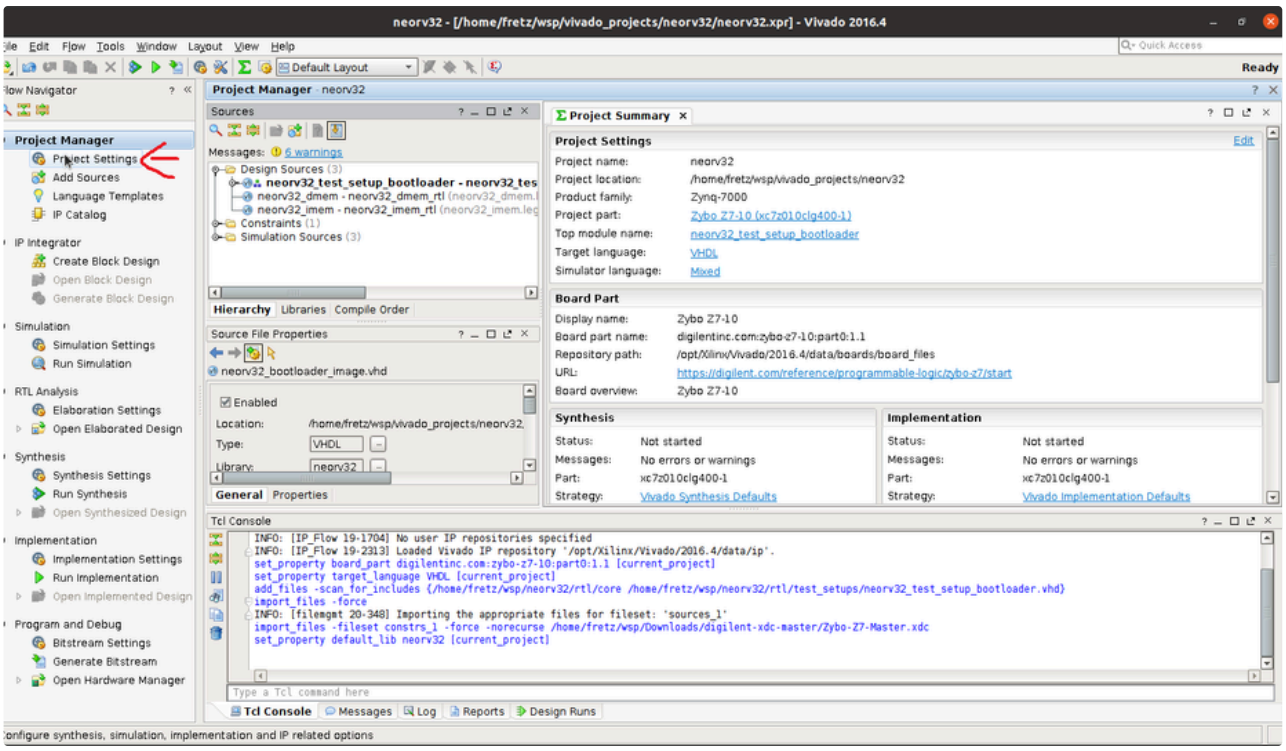
- Choose `Zybo Z7-10` and press the next button



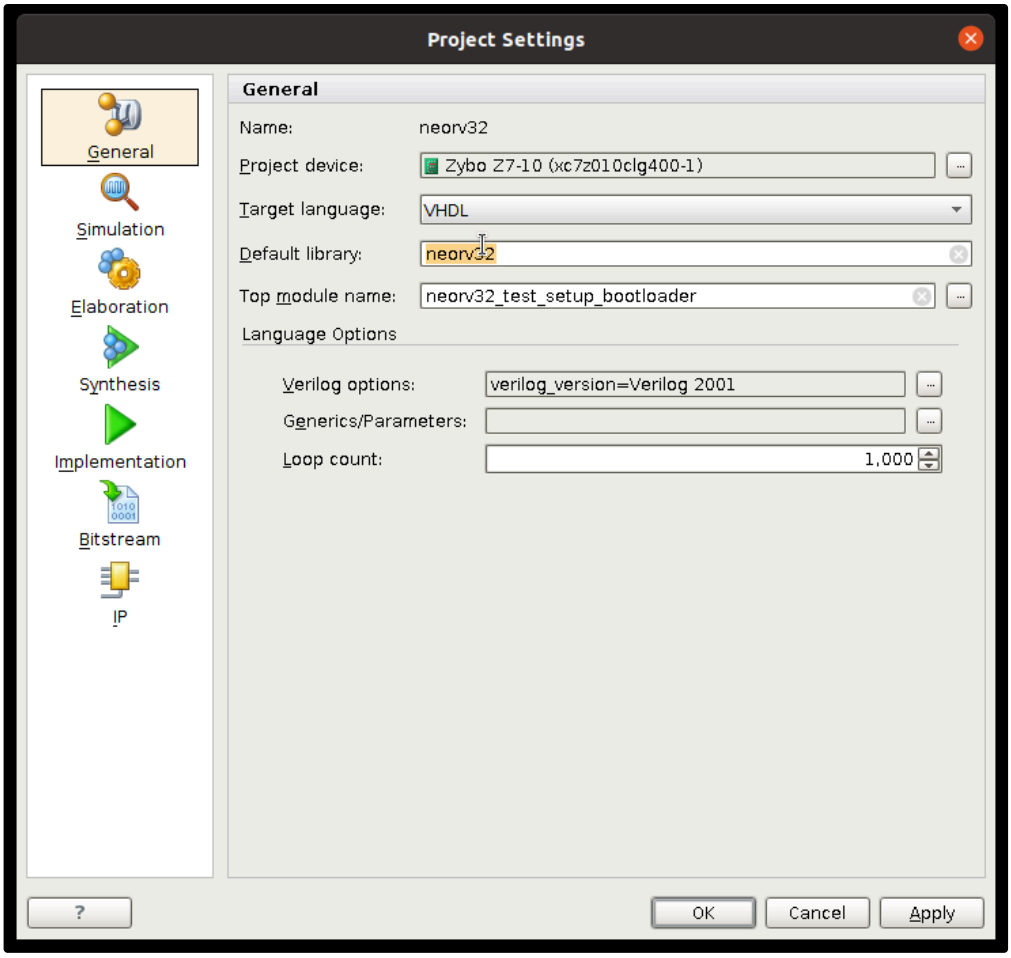
- Finally, click on the Finish button



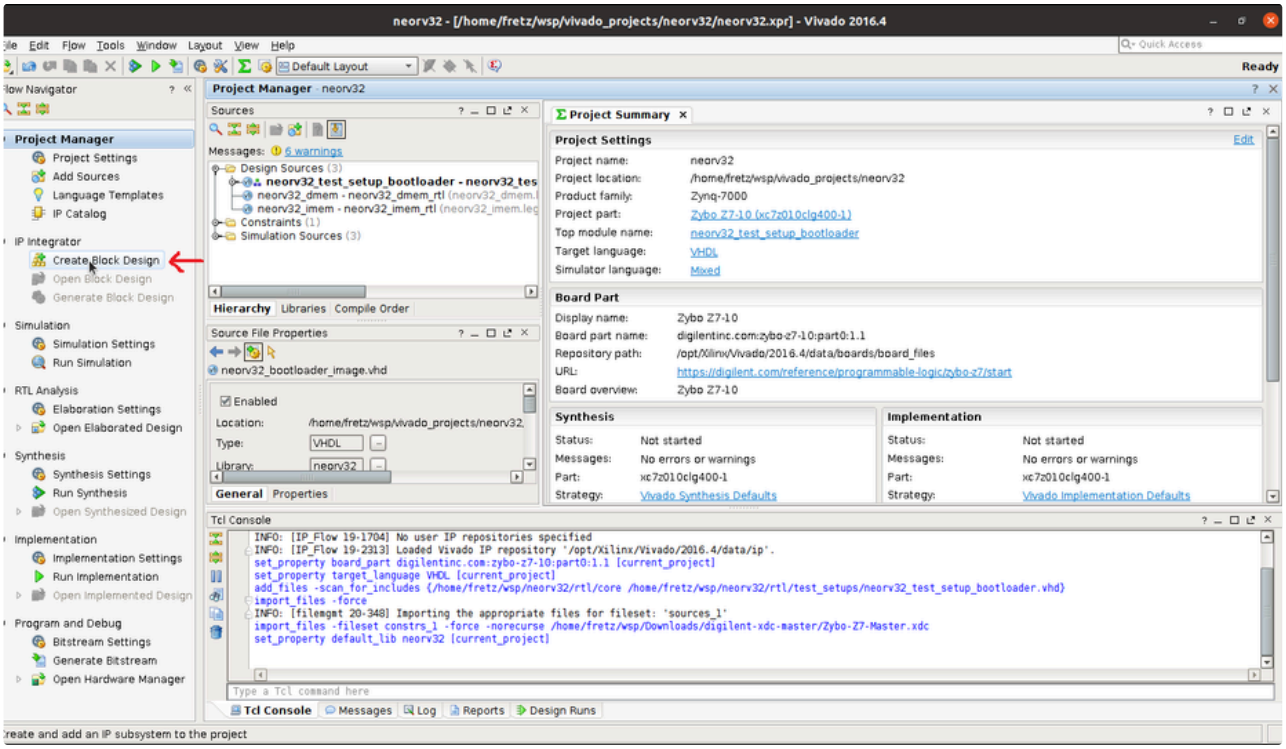
- Select the Project Settings under the Project Manager



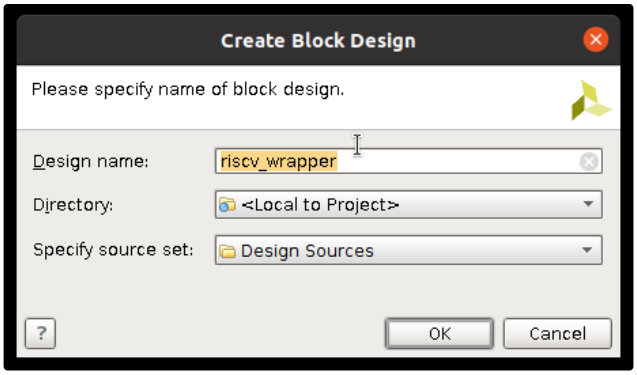
- Rename the Default library `xil_defaultlib` with `neorv32`. Close the Project Settings by pressing the Apply and OK buttons



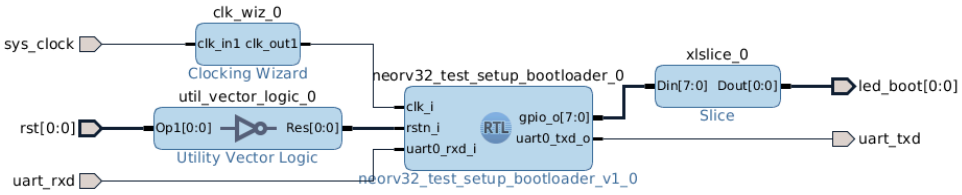
- Click on the Create Block Design



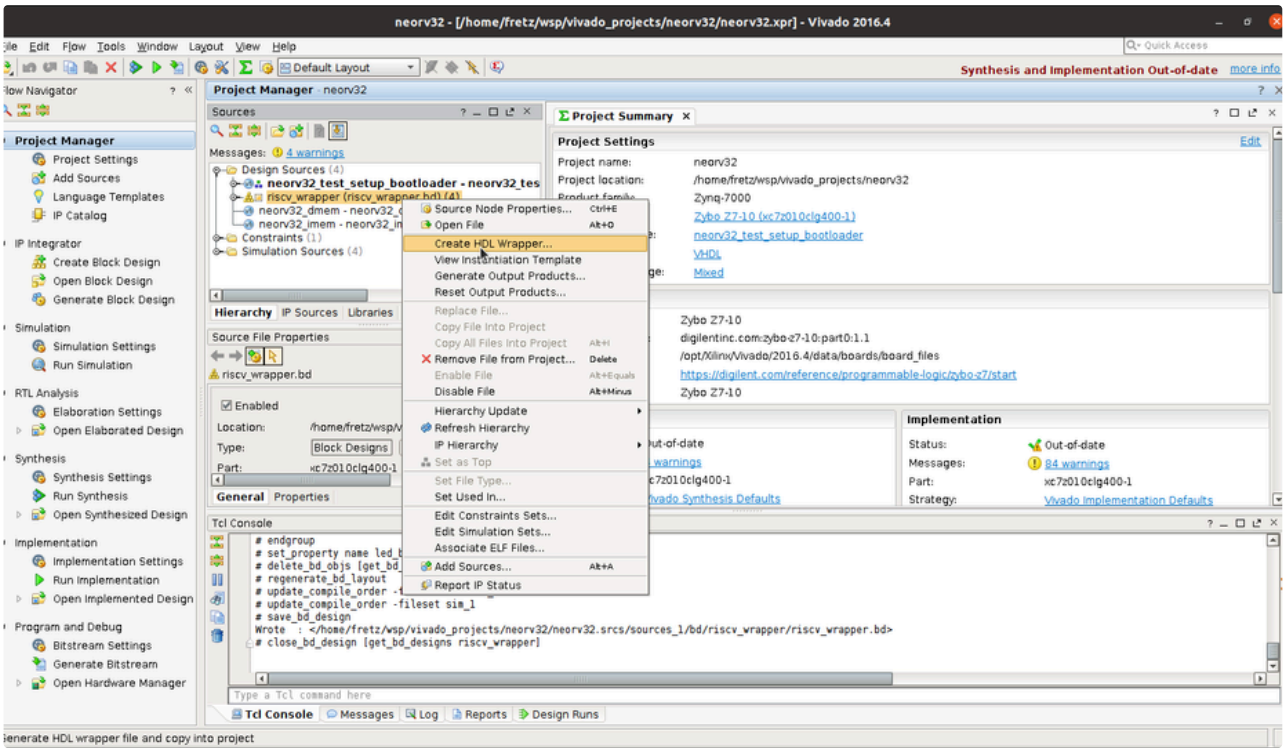
- Give the name `riscv_wrapper`. Then press the OK button



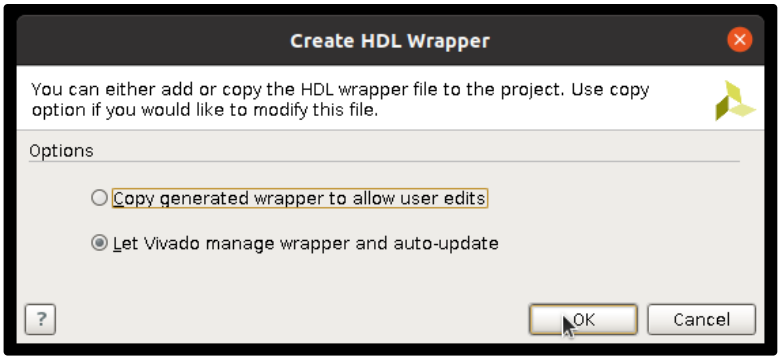
- Create the following block design



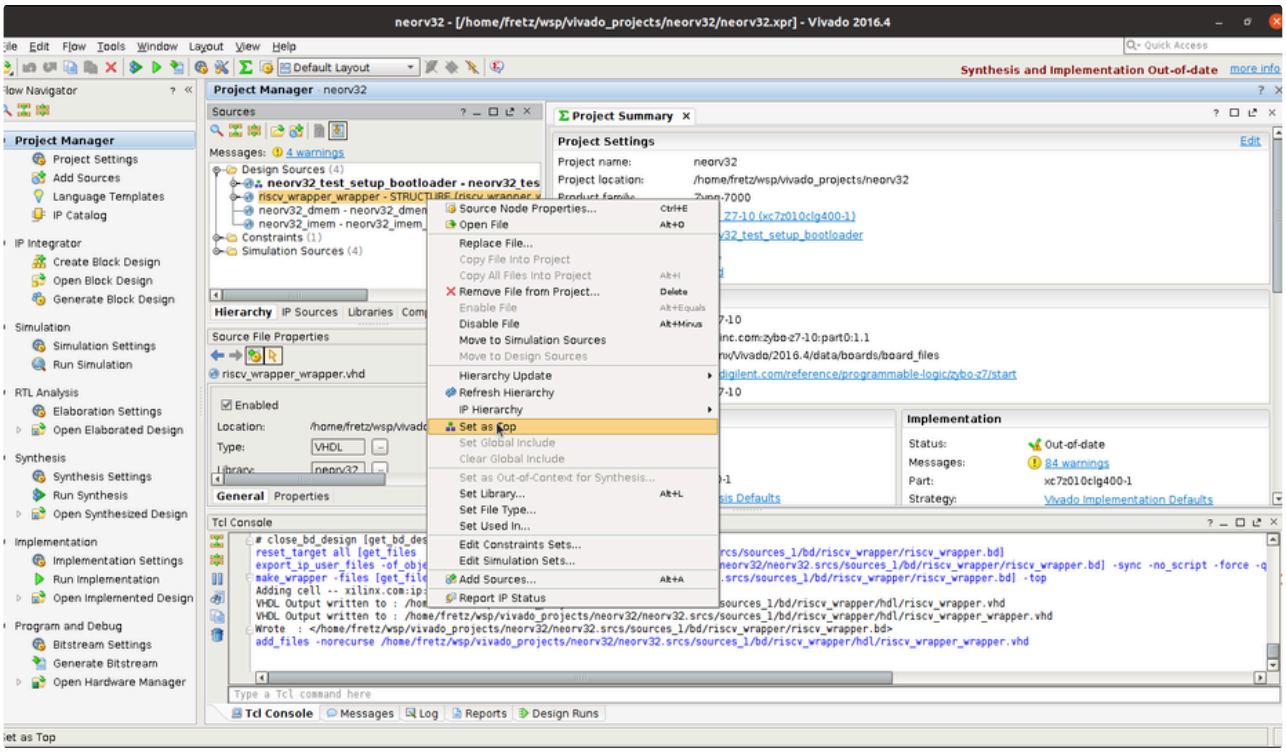
- Next, create a VHDL wrapper for the block design



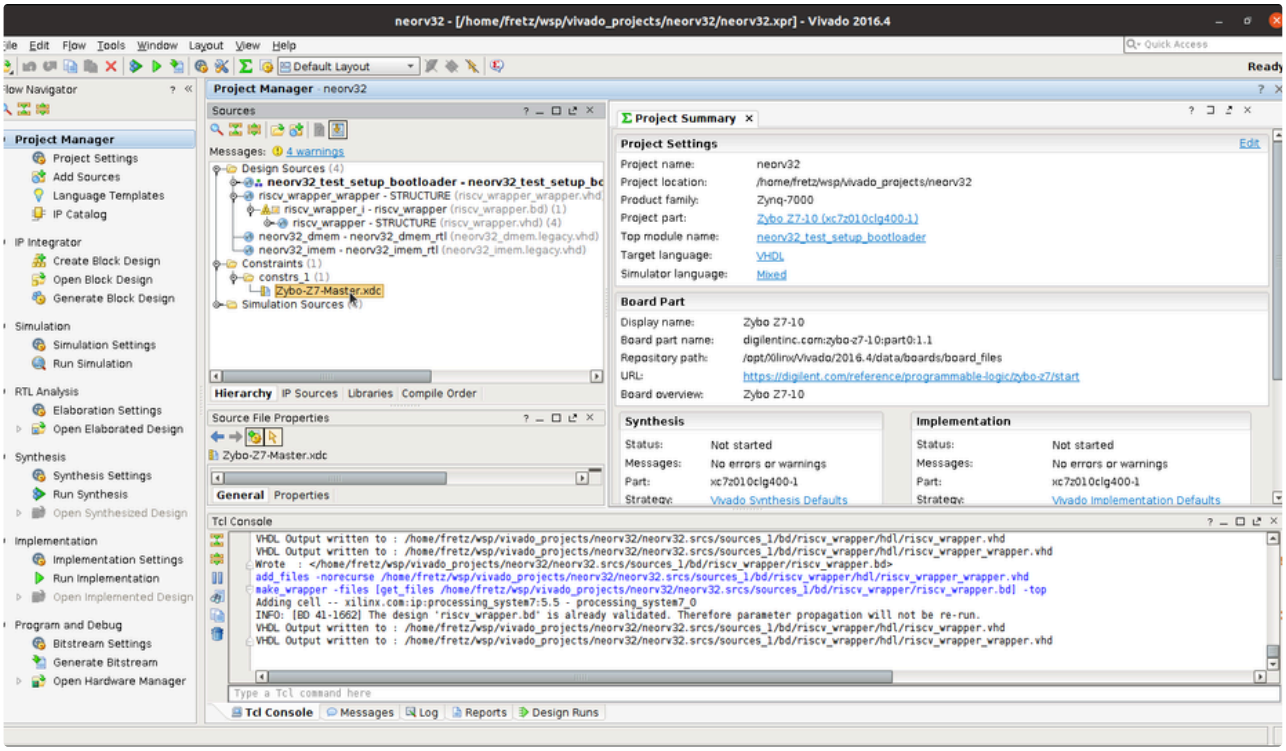
- Select the `Let Vivado manage` and click the OK button



- Next, set as the top module the wrapper you just created



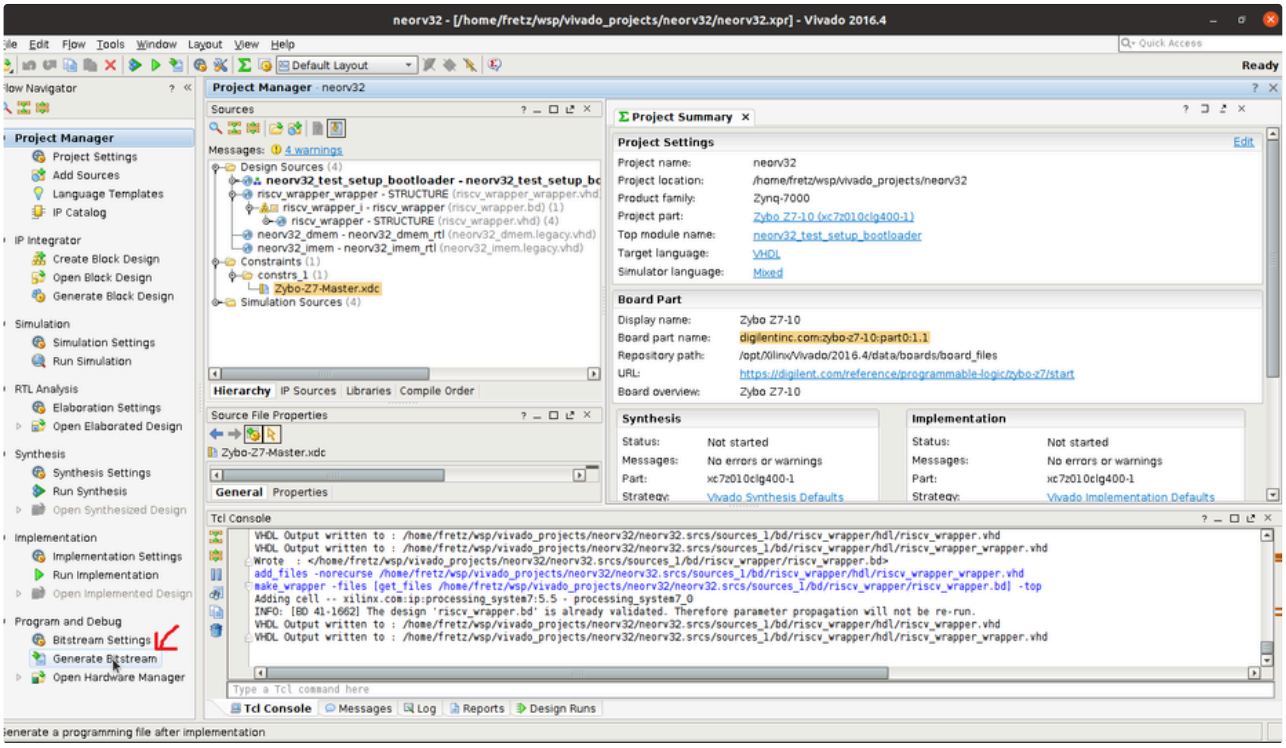
- Open the XDC file of the board to connect the NEORV32 ports to the appropriate pins of the FPGA.

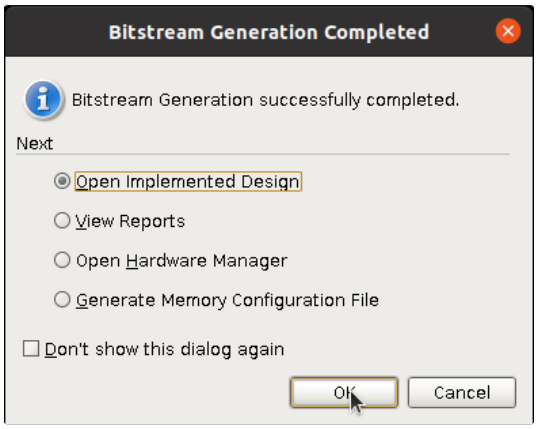


- You should modify the following pins

```
## RISC-V Reset
set_property -dict { PACKAGE_PIN K18   IOSTANDARD LVCMOS33 } [get_ports { ADD_PORT }]; #IO_L12N_T1_MRCC_35 Sch
# RISC-V LEDs
set_property -dict { PACKAGE_PIN M14   IOSTANDARD LVCMOS33 } [get_ports { ADD_PORT }]; #IO_L23P_T3_35 Sch=led[
##Pmod Header JC
set_property -dict { PACKAGE_PIN V15   IOSTANDARD LVCMOS33 } [get_ports { ADD_PORT }]; #IO_L10P_T1_34 Sch=JC1_
set_property -dict { PACKAGE_PIN W15   IOSTANDARD LVCMOS33 } [get_ports { ADD_PORT}]; #IO_L10N_T1_34 Sch=JC1_N
```

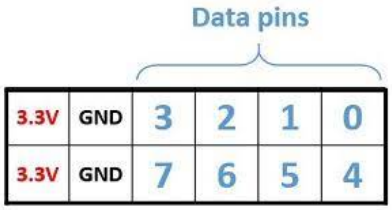
- Save the XDC file and press the Generate Bitstream button. This will generate the bitstream after synthesis and implementation are successfully finished.

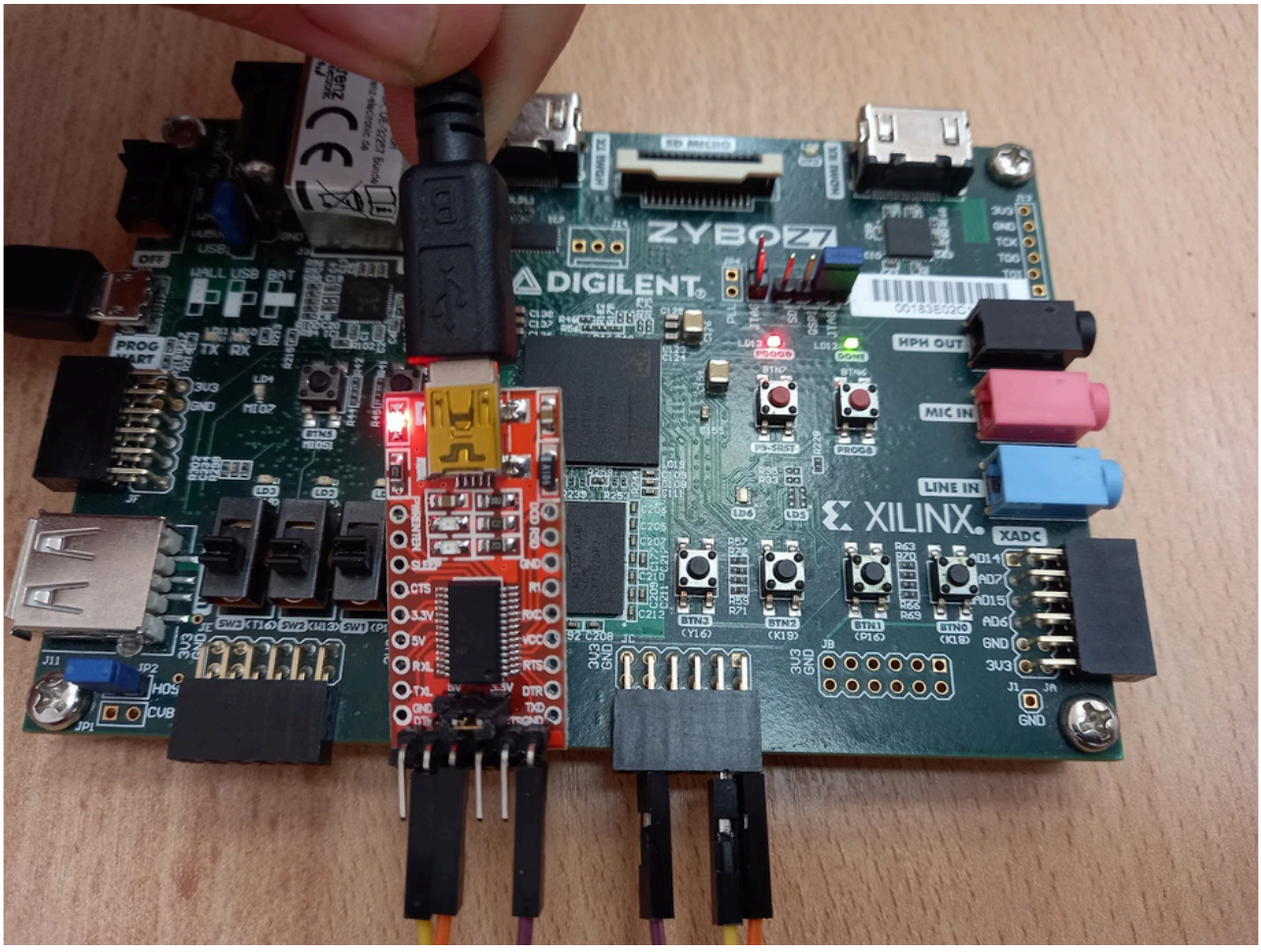


- When the bitstream generation finishes, open to see the implemented design
- 

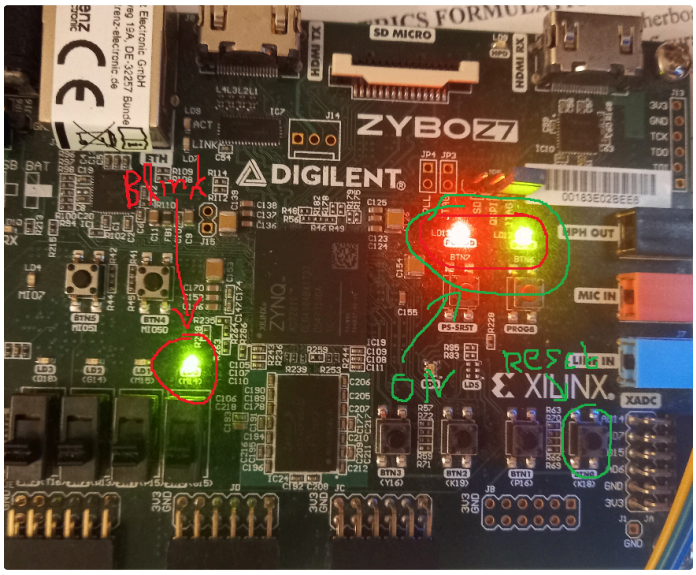
Board setup and run hello world software on the NEORV-32

- In order to get UART access on NEORV-32, we need to connect a USB/TTL UART external board on the PMOD header JC.

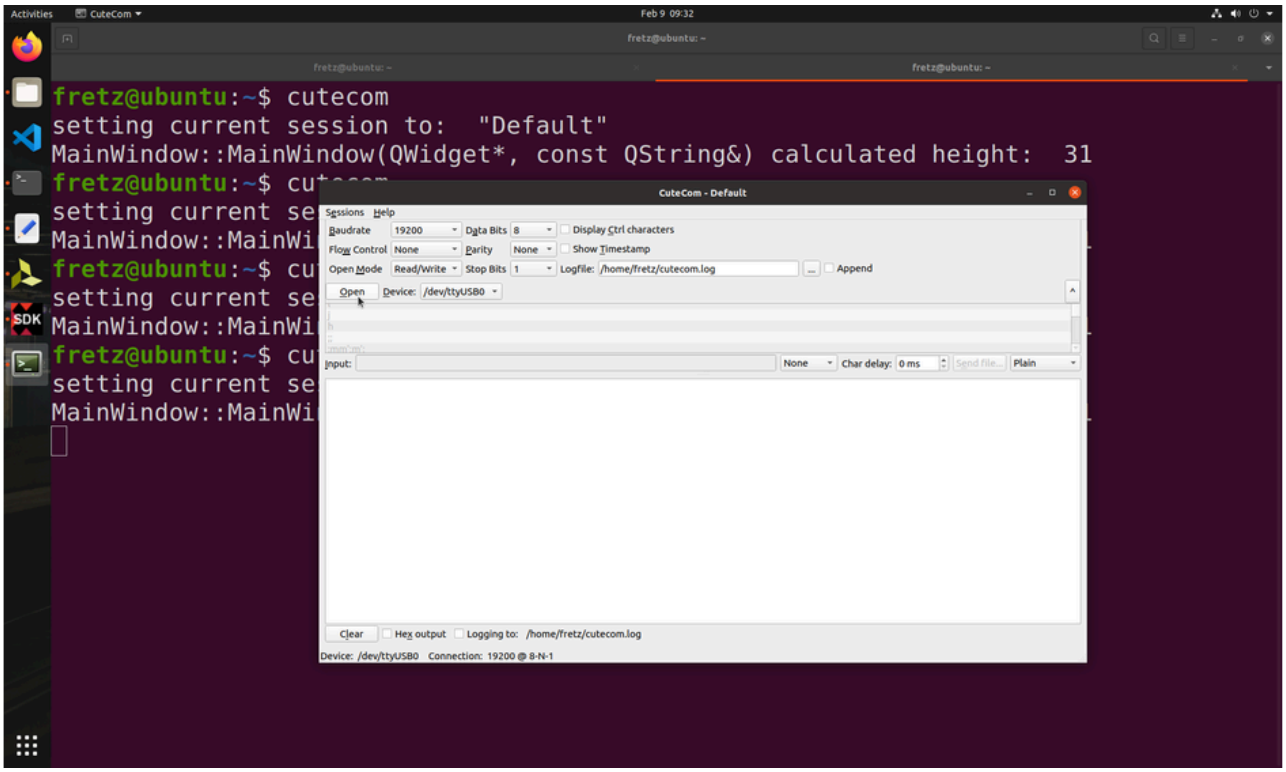




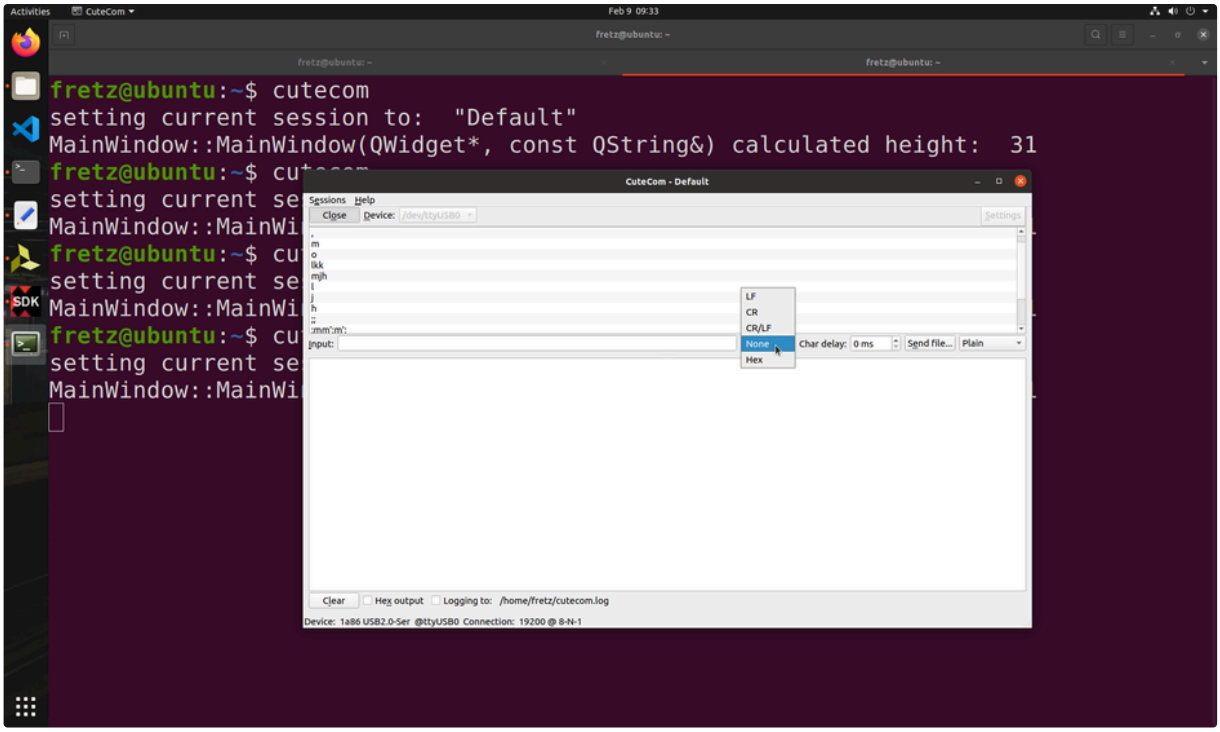
-
- Now, Programm the FPGA
- Once you program the FPGA you should see that the LD0 blinks (left side of the photo) after you press the reset button (right side of the photo). Also, the PGOOD and DONE leds should be ON.



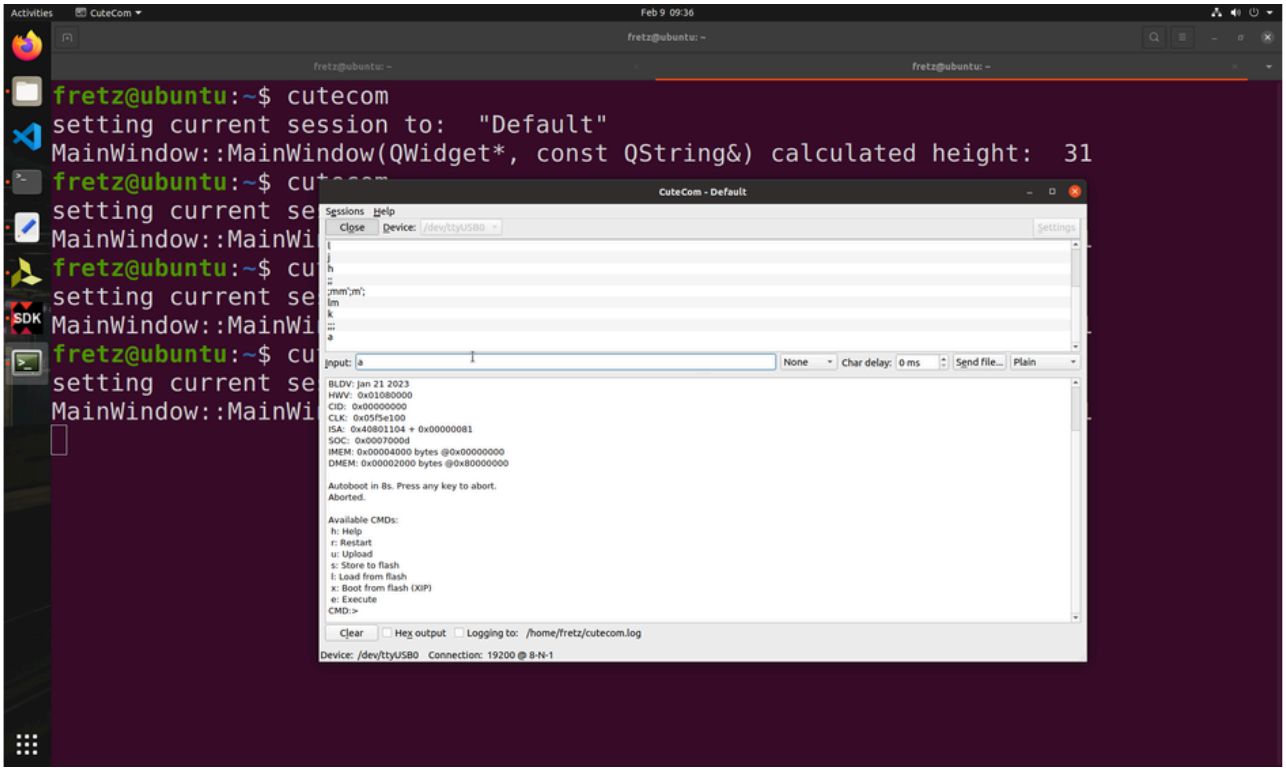
- Open a terminal and press `cutecom`
- Click on settings and configure as follows and then click on open



- Select None



- In the input, add the character `a`, press the reset button of NEORV32-V on the board and then press enter on cutecom



Setup RISC-V compiler

Skip these steps if you have the Fretz virtual machine.

- Open the Fretz VM and install `cutecom`, and the RISC-V compile flow
- In a terminal type, the following

```
$ sudo apt install cutecom -y
$ cd ~/wsp/Downloads/
```

```
$ wget https://github.com/stnolting/riscv-gcc-prebuilt/releases/download/rv32i-2.0.0/riscv32-unknown-elf.gcc-10.2
```

Create a folder where you want to install the toolchain, for example `/opt/riscv` (you will need `sudo` rights to create this folder and copy data to it).

```
$ sudo mkdir /opt/riscv
```

Navigate to the download folder. Decompress your toolchain (replace `TOOLCHAIN` with your toolchain archive of choice). Again, you might have to use `sudo` if your target directory is protected.

```
$ sudo tar xzfv riscv32-unknown-elf.gcc-10.2.0.rv32i.ilp32.newlib.tar.gz -C /opt/riscv/
```

Now add the toolchain's `bin` folder to your system's `PATH` environment variable (or add this line to your `.bashrc` if applicable):

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

Test the toolchain:

```
$ riscv32-unknown-elf-gcc -v
```

Continue from here: Compile your first hello world example!!!!

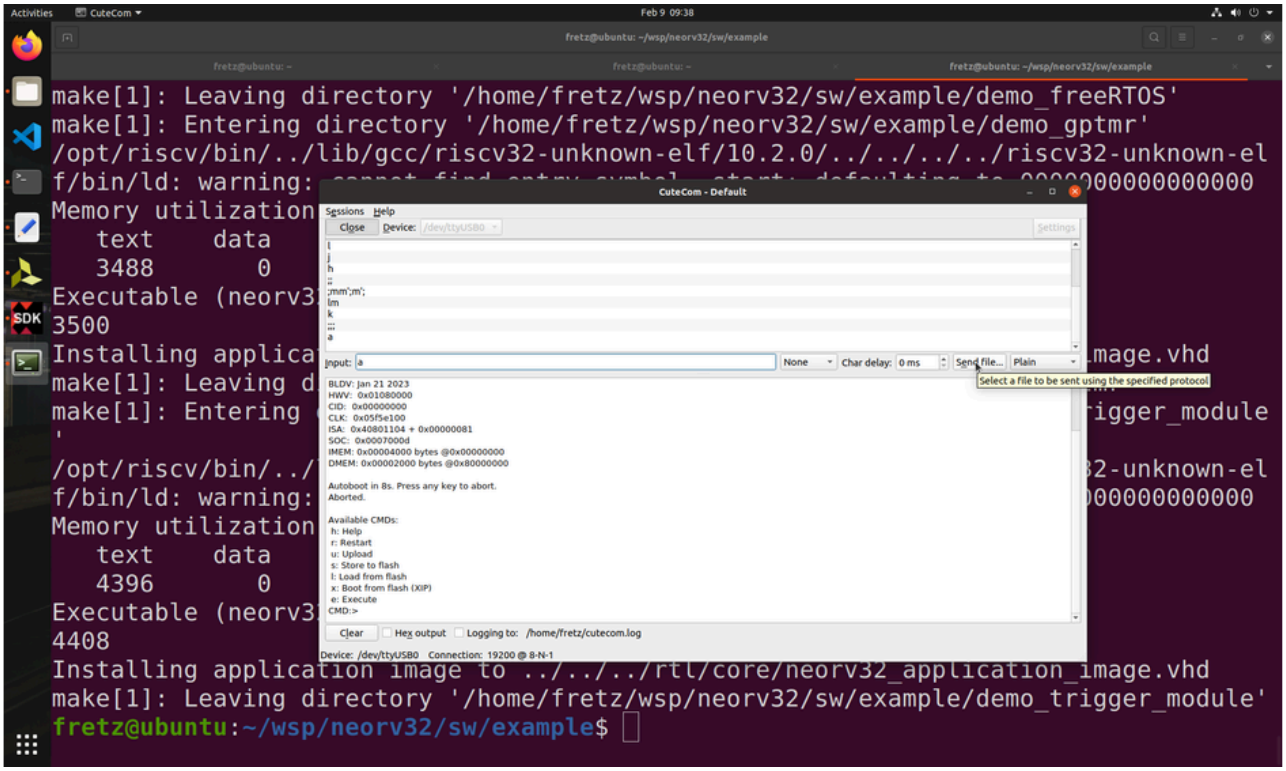
- Let's download some application examples. Open a terminal and type

```
cd ~/wsp/neorv32/sw/example/hello_world
make clean_all
make
```

- On cutecome enter the character `u` and press enter

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

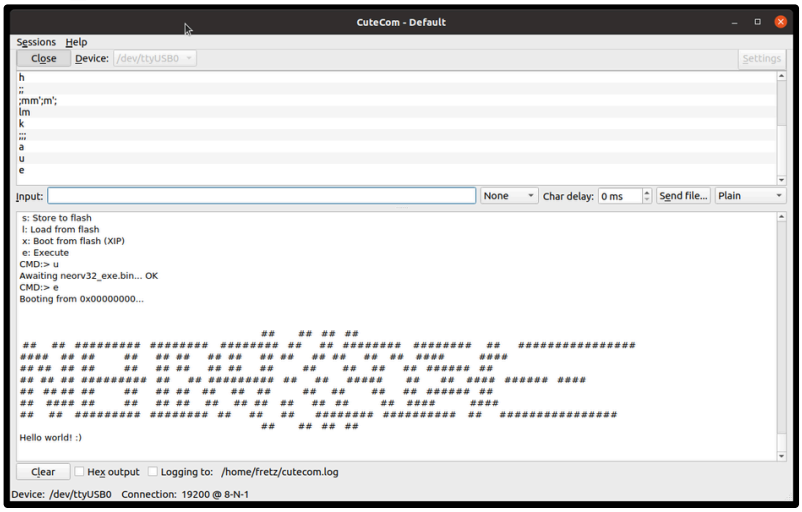
- On cutecome click the send file button



- If everything goes fine, OK will appear in your terminal:

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

- The executable is now in the instruction memory of the processor. To execute the program right now, run the "Execute" command by typing `e` in cutecome and press the Enter on your keyboard:



- Read [User Guide] The NEORV32 RISC-V Processor section 6
-

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 app
```