THE IMAGINATION UNIVERSITY PROGRAMME

RVfpga
Getting Started Guide

Acknowledgements

AUTHORS
Prof. Sarah Harris
Prof. Daniel Chaver
Zubair Kakakhel
M. Hamza Liaqat

ADVISER
Prof. David Patterson

CONTRIBUTORS
Robert Owen
Olof Kindgren
Prof. Luis Piñuel
Ivan Kravets
Valerii Koval
Ted Marena
Prof. Roy Kravitz

ASSOCIATES
Prof. José Ignacio Gómez
Prof. Christian Tenllado
Prof. Daniel León
Prof. Katzalin Olcoz
Prof. Alberto del Barrio
Prof. Fernando Castro
Prof. Manuel Prieto
Prof. Francisco Tirado
Prof. Román Hermida
Prof. Julio Villalba
Prof. Ataur Patwary
Cathal McCabe
Dan Hugo
Braden Harwood
Prof. David Burnett
Gage Elerding
Prof. Brian Cruickshank
Deepen Parmar
Thong Doan
Oliver Rew
Niko Nikolay
Guanyang He
Prof. Peng Liu

Sponsors and Supporters

AUTHORS
- Prof. Sarah Harris (https://www.linkedin.com/in/sarah-harris-12720697/)
- Prof. Daniel Chaver (https://www.linkedin.com/in/daniel-chaver-a5056a156/)
- Zubair Kakakhel (https://www.linkedin.com/in/zubairlk/)
- M. Hamza Liaqat https://www.linkedin.com/in/muhammad-hamza-liaqat-ab73a0195/)

ADVISER
- Prof. David Patterson (https://www.linkedin.com/in/dave-patterson-408225/)

CONTRIBUTORS
- Robert Owen (https://www.linkedin.com/in/robert-owen-4335931/)
- Olof Kindgren (https://www.linkedin.com/in/olofkindgren/)
- Prof. Luis Piñuel ()
- Ivan Kravets (https://www.linkedin.com/in/ivankravets/)
- Valerii Koval (https://www.linkedin.com/in/valeros/)
- Ted Marena (https://www.linkedin.com/in/tedmarena/)
- Prof. Roy Kravitz (https://www.linkedin.com/in/roy-kravitz-4725963/)

ASSOCIATES
- Prof. José Ignacio Gómez (https://www.linkedin.com/in/jos%C3%A9-ignacio-gomez-182b981/)
- Prof. Christian Tenllado (https://www.linkedin.com/in/christian-tenllado-31578659/)
- Prof. Daniel León (www.linkedin.com/in/danileon-ufv)
- Prof. Katzalin Olcoz (https://www.linkedin.com/in/katzalin-olcoz-herrero-5724b0200/)
- Prof. Alberto del Barrio (https://www.linkedin.com/in/alberto-antonio-del-barrio-garc%C3%ADa-1a85586a/)
- Prof. Fernando Castro (https://www.linkedin.com/in/fernando-castro-5993103a/)
- Prof. Manuel Prieto (https://www.linkedin.com/in/manuel-prieto-matias-02470b8b/)
- Prof. Francisco Tirado (https://www.linkedin.com/in/francisco-tirado-fern%C3%A1ndez-40a45570/)
- Prof. Román Hermida (https://www.linkedin.com/in/roman-hermida-correa-a4175645/)
- Prof. Julio Villalba (https://www.linkedin.com/in/julio-villalba-moreno-97474824)
- Cathal McCabe (https://www.linkedin.com/in/cathalmccabe/)
- Dan Hugo (https://www.linkedin.com/in/danhugo/)
- Braden Harwood (https://www.linkedin.com/in/braden-harwood/)
- David Burnett (https://www.linkedin.com/in/david-burnett-3b03778/)
- Gage Elerding (https://www.linkedin.com/in/gage-elerding-052b16106/)
- Brian Cruickshank (https://www.linkedin.com/in/bcruiksh/)
- Deepen Parmar (https://www.linkedin.com/in/deepen-parmar/)
- Thong Doan (https://www.linkedin.com/in/thong-doan/)
- Oliver Rew (https://www.linkedin.com/in/oliver-rew/)
- Niko Nikolay (https://www.linkedin.com/in/roy-kravitz-4725963/)
- Guanyang He (https://www.linkedin.com/in/guanyang-he-5775ba109/)
- Prof. Ataur Patwary (https://www.linkedin.com/in/ataurpatwary/)
- Prof. Peng Liu (https://person.zju.edu.cn/liupeng)

Table of Contents

0. PREFACE

This RVfpga course in Computer Architecture provides hands-on understanding of a commercial RISC-V processor, RISC-V SoC, and the RISC-V ecosystem. The course provides an understanding of the system from the underlying digital design and signals to the instruction set architecture and processor to the programming environment, boot code, and compiler. The fact that RVfpga users walk away with this top to bottom understanding of the RISC-V system is remarkable. They not only have a working RISC-V SoC and ecosystem, but they know how to use and expand the RISC-V processor and system for future projects and research.

Professor David Patterson, who shared the ACM A.M. Turing Award with John Hennessy for their contribution to RISC, says, “RISC-V is transforming processor design and software/hardware co-design. RISC-V is an open architecture, which enables open-source hardware implementations. This new option means that software development can occur alongside hardware development, accelerating the design path. The RVfpga course enhances the understanding of not only RISC-V processors but also the RISC-V ecosystem and RISC-V SoCs. This course provides a deep understanding of an industrial-strength processor architecture and system of increasing popularity, which will prove useful throughout their academic and industry careers.”

1. INTRODUCTION

RISC-V FPGA, also written RVfpga, is a package that includes instructions, tools, and labs for targeting a commercial RISC-V processor to a field programmable gate array (FPGA) and to a simulator, and then using and expanding it to learn about computer architecture, digital design, embedded systems, and programming.

This RVfpga Getting Started Guide has the following sections, as described briefly below:

- **Quick Start Guide** (Section 2)
- **Background and Overview**
  - **RISC-V Architecture** (Section 3)
  - **The RVfpga System** (Section 4)
- **Using the RVfpga System in Hardware**
  - **Installing Software Tools** (Section 5)
  - **Running and Programming the RVfpga System** (Section 6)
- **Simulating the RVfpga System**
  - **Using Verilator**, an HDL Simulator (Section 7)
  - **Using Whisper**, Western Digital's Instruction Set Simulator (Section 8)
- **Appendices**
  - **Using the native RISC-V toolchain and OpenOCD** (Appendix A)
  - **Installing drivers in Windows to use PlatformIO** (Appendix B)
  - **Installing Verilator and GTKWave in Windows** (Appendix C)
  - **Installing Verilator and GTKWave in macOS** (Appendix D)
  - **Using Vivado to download the RVfpga System onto an FPGA** (Appendix E)
  - **Example: Using RVfpga in an industrial IoT application** (Appendix F)

The Quick Start Guide (Section 2) describes the minimal software installation needed for RVfpga and then shows how to download and execute a simple example program on the RVfpga System. To understand RVfpga more fully, skip Section 2 and start with the complete guide that starts in Section 3.

Section 3 gives a brief introduction to the RISC-V computer architecture. Section 4 describes the RVfpga System (Section 4.A – 4.C) and the organization of the Verilog files that make up the system (Section 4.D). The RVfpga System is based on the SweRVolf SoC (https://github.com/chipsalliance/Cores-SweRVolf) which, in turn, uses Western Digital's (WD's) open-source RISC-V SweRV EH1 Core (https://github.com/chipsalliance/Cores-SweRV). Figure 1 and Table 1 illustrate the hierarchical organization of the RVfpga System, from the SweRV EH1 Core up to RVfpgaNexys and RVfpgaSim.

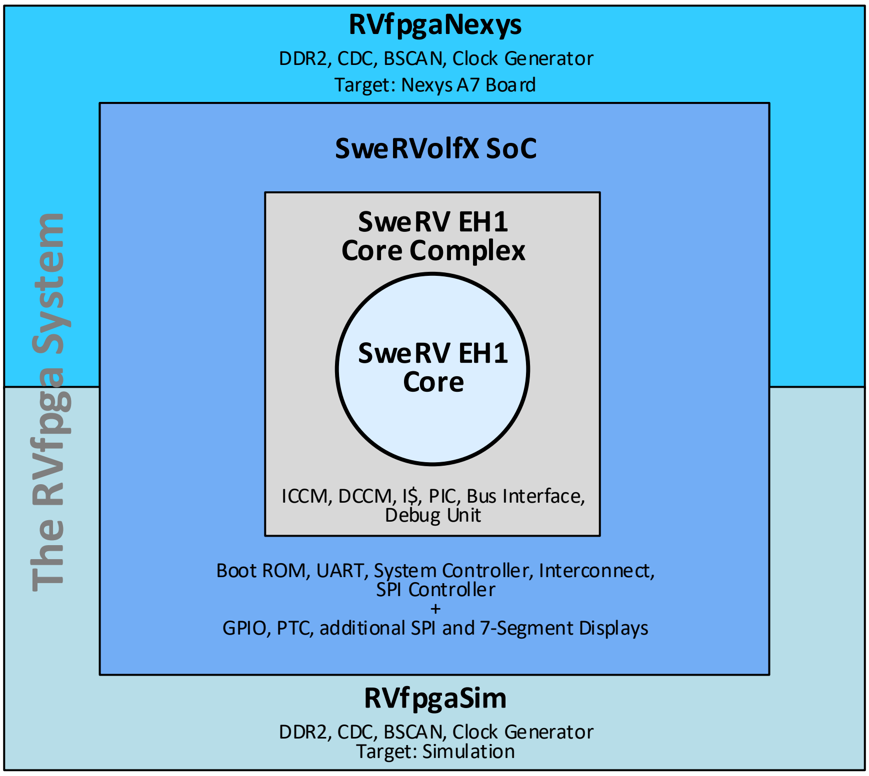


Figure 1. RVfpga System Hierarchy

Table 1. RVfpga System Hierarchy

Name	Description
SweRV EH1 Core	Open-source commercial RISC-V core developed be Western Digital (https://github.com/chipsalliance/Cores-SweRV).
SweRV EH1 Core Complex	SweRV EH1 core with added memory (ICCM, DCCM, and instruction cache), programmable interrupt controller (PIC), bus interfaces, and debug unit (https://github.com/chipsalliance/Cores-SweRV).
SweRVolfX (Extended SweRVolf)	The System on Chip that we use in the RVfpga course. It is an extension of SweRVolf. SweRVolf (https://github.com/chipsalliance/Cores-SweRVolf): An open-source SoC built around the SweRV EH1 Core Complex. It adds a boot ROM, UART interface, system controller, interconnect (AXI Interconnect, Wishbone Interconnect, and AXI-to-Wishbone bridge), and an SPI controller. SweRVolfX: It adds 4 new peripherals to SweRVolf: a GPIO, a PTC, an additional SPI and a controller for the 8 digit 7-Segment Displays.
RVfpgaNexys	The SweRVolfX SoC targeted to the Nexys A7 board and its peripherals. It adds a DDR2 interface, CDC (clock domain crossing) unit, BSCAN logic (for the JTAG interface), and clock generator. RVfpgaNexys is the same as SweRVolf Nexys (https://github.com/chipsalliance/Cores-SweRVolf), except that the latter is based on SweRVolf.
RVfpgaSim	The SweRVolfX SoC with a testbench wrapper and AXI memory intended for simulation. RVfpgaSim is the same as SweRVolf sim, (https://github.com/chipsalliance/Cores-SweRVolf), except that the latter is based on SweRVolf.

The remaining sections show how to use the RVfpga System in both hardware (RVfpgaNexys) and simulation (RVfpgaSim). Section 5 shows how to install the software tools needed to use RVfpga. Section 6 shows how to use PlatformIO to both download RVfpgaNexys onto the Nexys A7 FPGA board (Section 6.A) and download and run several example programs on it (Section 6.B-6.H). Sections 7 and 8 show how to simulate RVfpgaSim using Verilator (Section 7), an open-source HDL simulator, and how to use Whisper (Section 8), Western Digital's RISC-V Instruction Set Simulator (ISS).

Finally, the appendices show how to use RVfpga at the command prompt in Linux (Appendix A), how to install needed drivers and software on Windows and macOS machines (Appendices B-D), and how to use Vivado to download RVfpgaNexys onto an FPGA using Vivado (Appendix E). The last appendix, Appendix F, shows how to use RVfpga in an industrial IoT application (Appendix F).

Table 2 lists the software and hardware needed for RVfpga. This guide shows how to install and use these tools and hardware on the Ubuntu 18.04 operating system (OS). Other operating systems (such as Windows or macOS), follow similar (if not exactly the same) steps. When instructions differ, we insert specific instructions for **Windows** and **macOS** using this highlighting.

Note: if you do not have access to the Nexys A7 FPGA board, the labs can still be completed in simulation using Whisper, Western Digital's instruction set simulator (ISS), and Verilator, an open-source HDL simulator. In this case, you do not need to install Vivado (Section 5.A); you need only install VSCode/PlatformIO (as explained in Section 2.A) and Verilator/GTKWave (as explained in Section 5.C).

Table 2. Required Software and Hardware for RVfpga

Software		
Name	**Website**	**Cost**
Vivado 2019.2 WebPACK	https://www.xilinx.com/support/download/index.html/content/xilinx/en/downloadNav/vivado-design-tools/2019-2.html	free
VSCode	https://code.visualstudio.com/Download	free
PlatformIO	https://platformio.org/ Installed within VSCode	free
Verilator (an HDL simulator) and GTKWave	https://github.com/verilator/verilator http://gtkwave.sourceforge.net/	free
Whisper (Western Digital's RISC-V Instruction Set Simulator)	https://github.com/chipsalliance/SweRV-ISS Installed within PlatformIO	free
RISC-V Toolchain and OpenOCD	https://github.com/riscv/riscv-gnu-toolchain https://github.com/riscv/riscv-openocd Installed within PlatformIO	free
Hardware		
Name	**Website**	**Cost**
Nexys A7 FPGA Board*	https://store.digilentinc.com/nexys-a7-fpga-trainer-board-recommended-for-ece-curriculum/	$265 (academic price: $199)
RISC-V Core and System-on-Chip (SoC)**		

Name	Website	Cost
Western Digital's SweRV EH1 Core	https://github.com/chipsalliance/Cores-SweRV	free
SweRVolf	https://github.com/chipsalliance/Cores-SweRVolf	free

* All of the steps described in this guide also work on Digilent's Nexys4 DDR FPGA board.
** Provided with the RVfpga download from Imagination Technologies

Expected Prior Knowledge:

Before completing this RVfpga course, which includes this RVfpga Getting Started Guide and RVfpga Labs, it is expected that users have at least a fundamental understanding of the following topics:

- Digital logic design
- High-level programming (preferably C)
- Assembly programming
- Instruction set architecture
- Processor microarchitecture
- Memory systems

These topics are covered in the textbook *Digital Design and Computer Architecture: RISC-V Edition*, Harris & Harris, © Morgan Kaufmann 2021. Other textbooks, including *Computer Organization and Design RISC-V Edition*, Patterson & Hennessy, © Morgan Kaufmann 2017, cover some of these topics.

2. QUICK START GUIDE

This section shows how to install the minimal tools needed to use RVfpga and then shows how to use PlatformIO to both download RVfpgaNexys onto the Nexys A7 FPGA board and then run a program on RVfpgaNexys. You will need to purchase the FPGA board (see Table 2). These steps also work for the Nexys4-DDR FPGA board, an earlier version of the board.

A. **Minimal installation: VSCode, PlatformIO and Nexys A7 board drivers**
B. **Download RVfpgaNexys onto FPGA and run programs on it**

The instructions below are for an Ubuntu 18.04 system. They also work for Windows 10 and macOS operating systems – when instructions differ from Ubuntu, we insert boxes with specific instructions for **Windows** and **macOS**. If you are using Ubuntu, you can just ignore those boxes. Paths are written as Linux paths using forward slashes (/), but Windows paths are typically the same but with backward slashes (\).

A. Minimal installation: VSCode, PlatformIO and Nexys A7 board drivers

In this step, you will install the minimum software and drivers needed to use RVfpga. First, you will install the programming environment, and then you will install the drivers for the Nexys A7 FPGA board.

VSCode and PlatformIO Installation: You will use PlatformIO, an integrated development environment (IDE) to download RVfpgaNexys onto the Nexys A7 board and also to download and run programs on RVfpgaNexys. PlatformIO is built as an extension of Microsoft's Visual Studio Code (VSCode). PlatformIO is cross-platform and includes a built-in debugger.

Follow these steps to install both VSCode and PlatformIO:

1. Install VSCode:
   a. Download the installation file from the following link: https://code.visualstudio.com/Download

   b. Open a terminal, and install and execute VSCode:
      ```
      cd ~/Downloads
      sudo dpkg -i code*.deb
      code
      ```

Windows / macOS: VSCode packages are also available for Windows (.exe file) and macOS (.zip file) at https://code.visualstudio.com/Download. Follow the usual steps used for installing and executing an application in these operating systems.

2. Install PlatformIO on top of VSCode:
   a. Install python3 utilities by typing the following in a terminal:
      ```
      sudo apt install -y python3-distutils python3-venv
      ```

Windows / macOS: this step (2.a) is not required in Windows. As for macOS, you can use *homebrew* to install python3: `brew install python3`

   b. If not yet open, start VSCode by selecting the Start button and typing "VSCode" in the search menu, then select VSCode, or type `code` in an Ubuntu terminal.

c. In VSCode, click on the Extensions icon located on the left side bar of VSCode (see Figure 2).

Figure 2. VSCode's Extensions icon

d. Type *PlatformIO* in the search box and install the PlatformIO *IDE* by clicking on the install button next to it (see Figure 3).

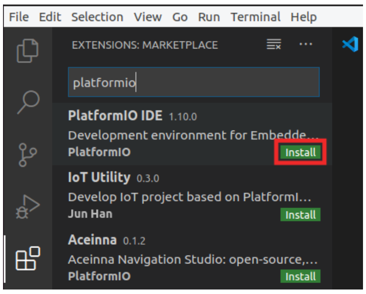


Figure 3. PlatformIO IDE Extension

e. The OUTPUT window on the bottom will inform you about the installation process. Once finished, click "Reload Now" on the bottom right side window, and PlatformIO will finish installing inside VSCode (see Figure 4).

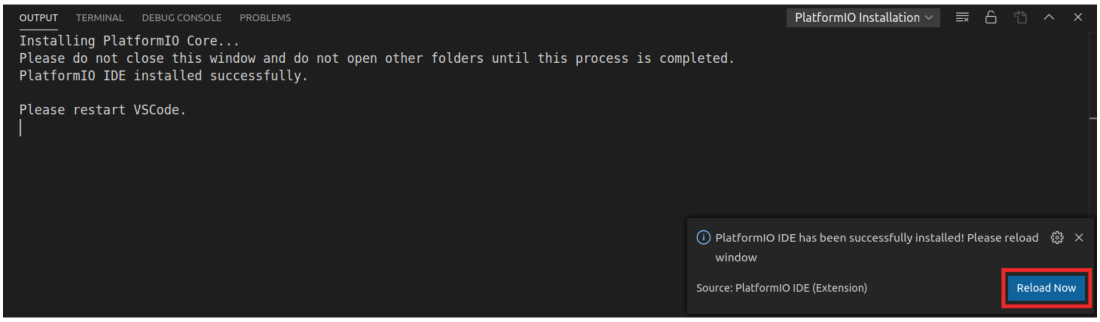


Figure 4. Reload Now after PlatformIO installs

<u>**Nexys A7 cable drivers installation**</u>**:** you need to manually install the drivers for the Nexys A7 board.

- Open a terminal.
- Go into directory *[RVfpgaPath]/RVfpga/driversLinux_NexysA7*. (For simplicity, we provide these drivers inside the RVfpga folder. When you install Vivado in Section 5

of this guide, you can also find these drivers inside the downloaded package as described in that section.)

- Run the installation script:

```
chmod 777 *
sudo ./install_drivers
```

- Unplug the Nexys A7 board from your computer and restart the computer for the changes to have effect.

Windows: follow the instructions provided in Appendix B for installing the drivers for the Nexys A7 board.

macOS: it is not necessary to install any additional drivers.

B. Download RVfpgaNexys onto FPGA and run programs on RVfpgaNexys

Now you will download RVfpgaNexys, the RISC-V system targeted to an FPGA, to the Nexys A7 FPGA board. Although we will not modify it in this Getting Started Guide, the Verilog for the RVfpga System is available in *[RVfpgaPath]/RVfpga/src*. We will describe the source code for the RVfpga system in Section 4 of this GSG and in more detail in RVfpga Labs 6-20. You will also modify the RVfpga system in some of the exercises for those labs.

Run RVfpgaNexys on the Nexys A7 FPGA board by completing the following steps:

Step 1. Connect Nexys A7 FPGA board to computer and turn the board on
Step 2. Open PlatformIO and C program
Step 3. Download RVfpgaNexys to Nexys A7 board
Step 4. Download and run program on RVfpgaNexys

Step 1. Connect Nexys A7 FPGA board to computer and turn the board on

Connect the Nexys A7 board to your computer using the provided USB cable. Figure 5 shows the physical locations of the LEDs and switches on the Nexys A7 FPGA board as well as the USB connector, on switch, pushbuttons, and 7-segment displays. Connect a cable between the USB connector port on the Nexys A7 board and turn on the board.

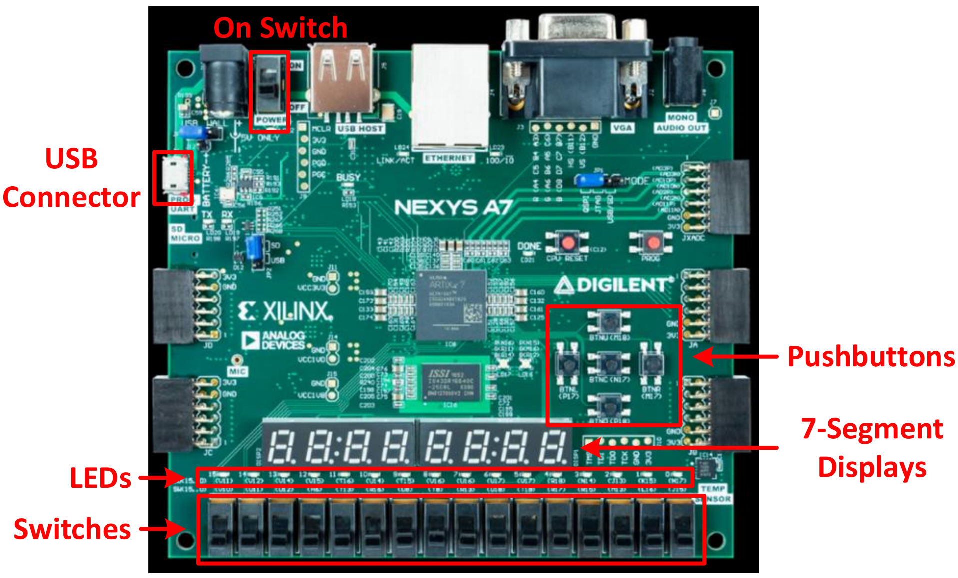


Figure 5. Digilent's Nexys A7 FPGA board's I/O interfaces
(figure of board from https://reference.digilentinc.com/)

Step 2. Open PlatformIO and C program

Now open Visual Studio Code (VSCode) by typing VSCode in the Start Menu (see Figure 6) or by typing `code` in a terminal.

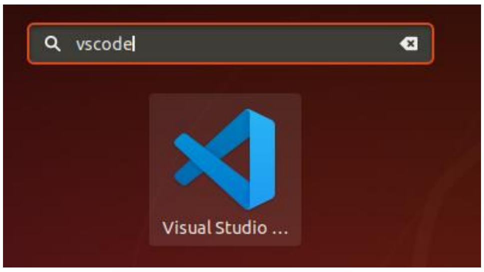


Figure 6. Open VSCode

If the PlatformIO Home (PIO Home) window does not automatically open, click on the PlatformIO icon in the left ribbon menu: . Then expand PIO Home and click on Open. Now PIO Home will open to the Welcome window (see Figure 7).

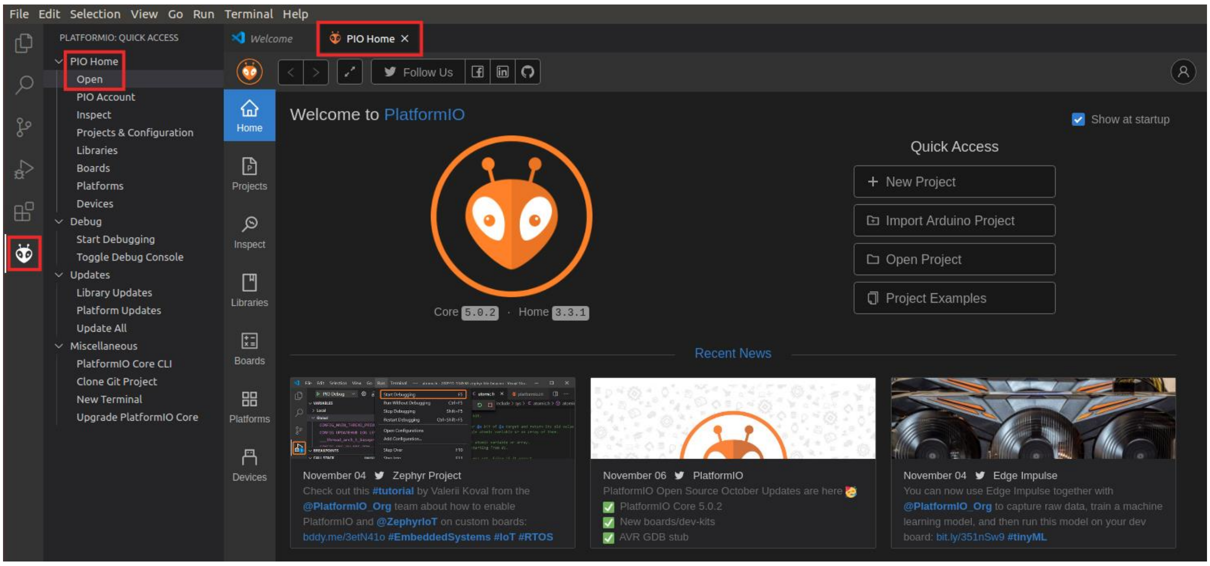


Figure 7. Open PIO Home

Now click on File → Open Folder from the top file menu and select:

[RVfpgaPath]\RVfpga\examples\LedsSwitches_C-Lang

Select the folder, but do not open it (see Figure 8). PlatformIO will now open this program, LedsSwitches_C-Lang, that reads the switch values on the Nexys A7 board and writes their value onto the LEDs on the board.

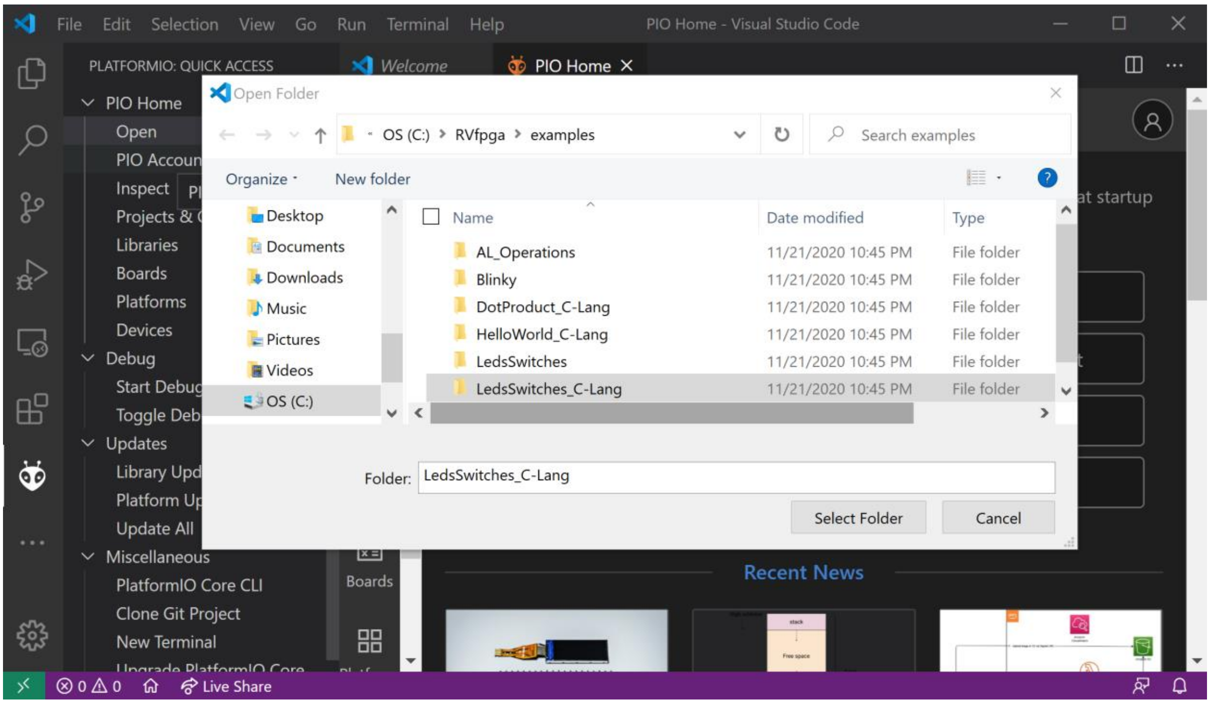


Figure 8. Open LedsSwitches_C-Lang example

You can view the LedsSwitches_C-Lang program by expanding the *src* folder and double-clicking on LedsSwitches_C-Lang.c (Figure 9). We discuss this program in detail later in this Getting Started Guide. For this Quick Start Guide, we will simply download this program onto RVfpgaNexys, which will be running on the Nexys A7 board.

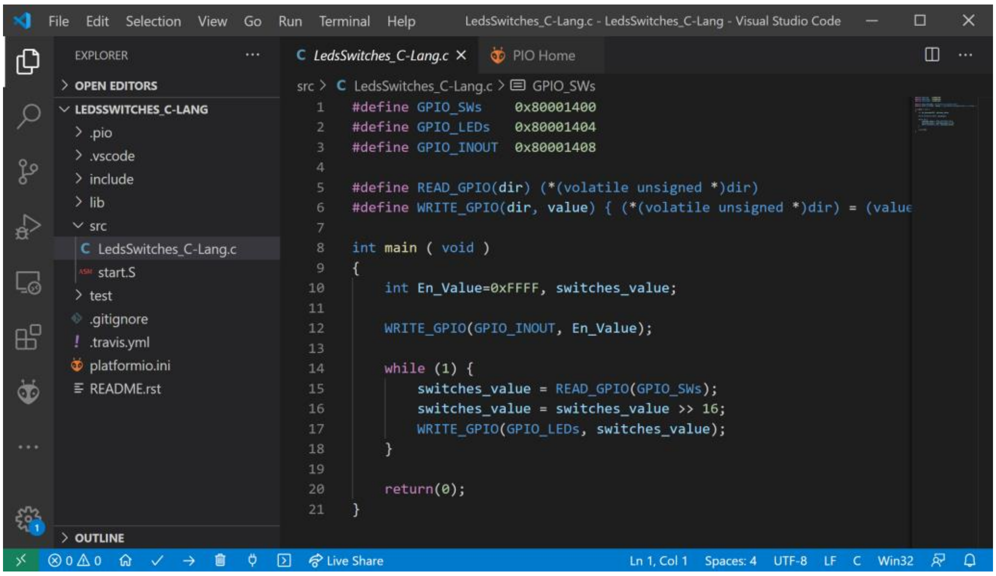


Figure 9. LedsSwitches_C-Lang.c program

Note that the first time that an RVfpga example is opened in PlatformIO, the Chips Alliance platform gets automatically installed (you can view it inside PIO Home → Platforms, as shown in Figure 10). This platform includes several tools that you will use later, such as the pre-built RISC-V toolchain, OpenOCD for RISC-V, an RVfpgaNexys bitfile and RVfpgaSim, JavaScript and Python scripts, and several examples. If, for any reason, the Chips Alliance platform did not get installed automatically, you could install it manually, as will be explained in Section 6.A.

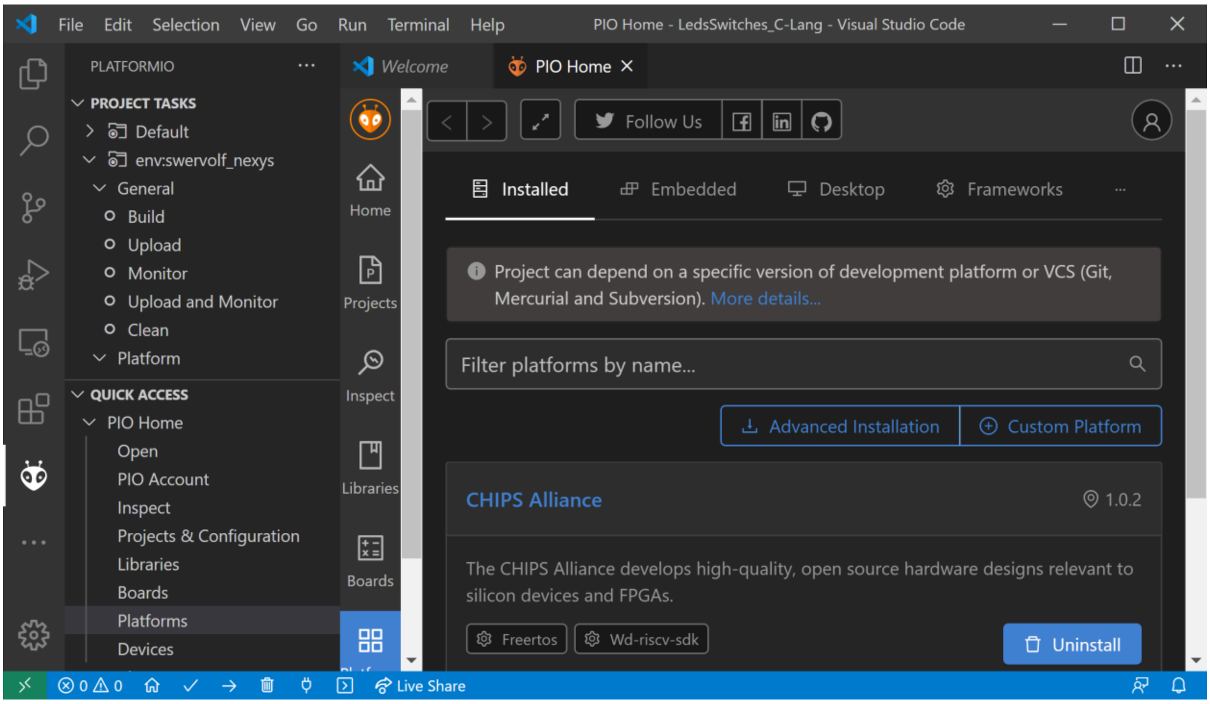


Figure 10. Chips Alliance platform installed in PlatformIO

Step 3. Download RVfpgaNexys to Nexys A7 board

You are now ready to download RVfpgaNexys, the RISC-V SoC that includes a RISC-V processor with support for peripherals. Open the platformio.ini (PlatformIO initialization file) by double-clicking on it in the EXPLORER window, as shown in Figure 11. (If the Explorer window is not already open, open it by clicking on  in the left ribbon menu.) Now, add the path for the location of the bitfile that defines RVfpgaNexys by replacing the board_build.bitstream_file path with your own path (see Figure 11):

board_build.bitstream_file = [RVfpgaPath]/RVfpga/src/rvfpganexys.bit

Save the platformio.ini file by pressing Ctrl-s.

Many commands exist for the Project Configuration File (*platformio.ini*); more information about these options are available at: https://docs.platformio.org/en/latest/projectconf/.

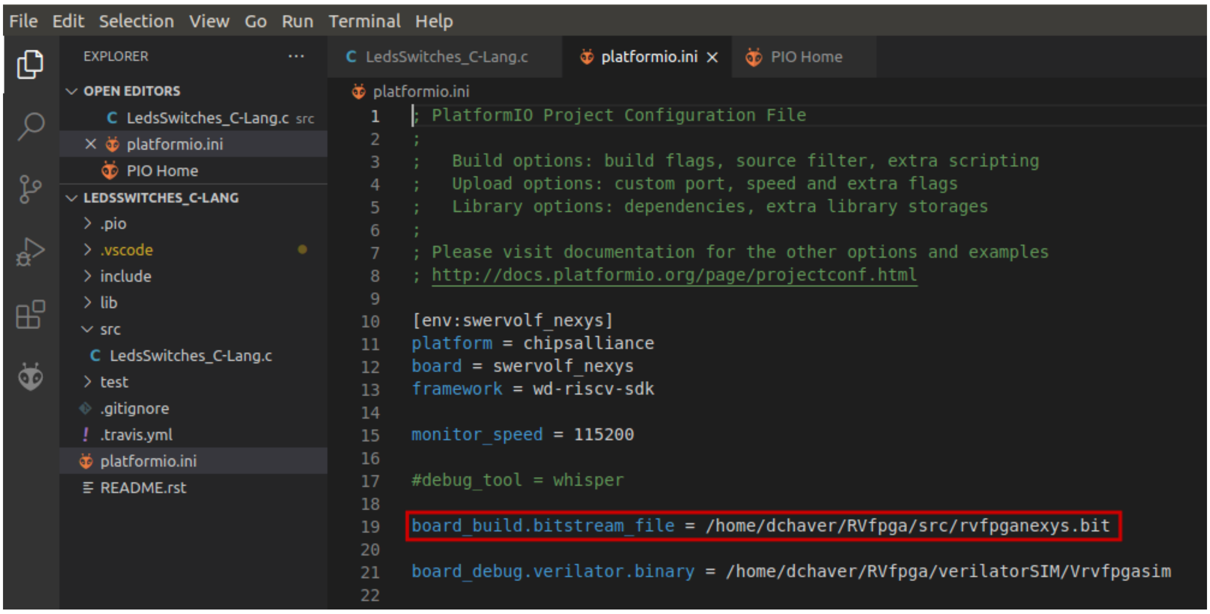


Figure 11. Add path to RVfpgaNexys bitfile

Download RVfpgaNexys (as defined by this bitfile) onto the Nexys A7 board:

- Click on the PlatformIO icon  in the left menu ribbon (see Figure 12).

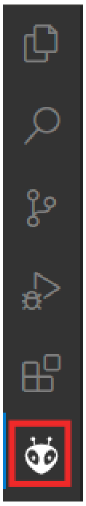


Figure 12. PlatformIO icon

- In case the Project Tasks window is empty (Figure 13), you must refresh the Project Tasks first by clicking on . This can take several minutes.

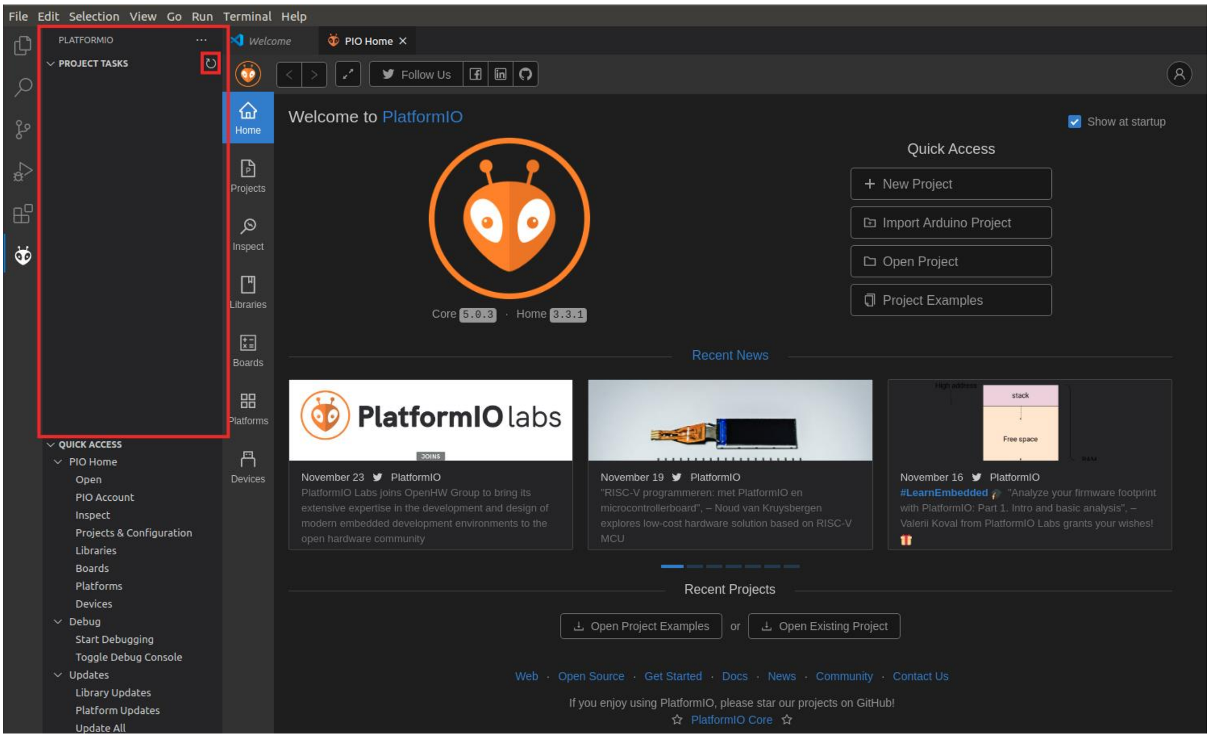


Figure 13. PROJECT TASKS window empty – Refresh

- Then expand Project Tasks → env:swervolf_nexys → Platform and click on Upload Bitstream, as shown in Figure 14. **After one or two seconds, the FPGA will be programmed with the RVfpgaNexys SoC**.

  By default, the processor starts fetching instructions at address 0x80000000, where the Boot ROM is placed in our SoC (see Table 6). The Boot ROM is initialized with a program (*boot_main.mem*) that blinks the LEDs and the 7-Segment Displays four times and then turns off all the LEDs, writes 0s to the 8 7-Segment Displays and stays in an empty loop. You can find this program in folder: *[RVfpgaPath]/RVfpga/src/SweRVolfSoC/BootROM/sw*. If you want to change and recompile it, do it as explained in Appendix A – Section III (note that file *boot_main.mem* is simply a copy of file *boot_main.vh*). In Lab 5, we will show how the Boot ROM is initialized with this program when creating the bitstream.

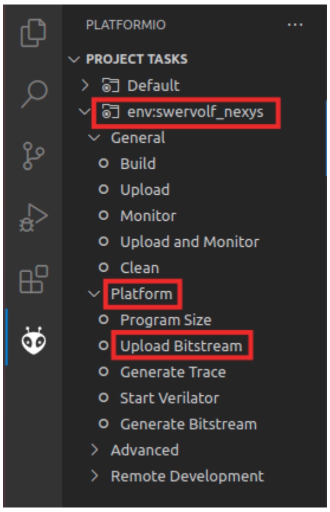


Figure 14. Upload Bitstream

Step 4. Download and run program on RVfpgaNexys

Now that RVfpgaNexys is downloaded and running on the Nexys A7 board, you will download the program into the memory of RVfpgaNexys and run/debug the program. Click on the “Run and Debug” button:  , which is available in the left side bar. Start the debugger by clicking on the play button 

(make sure that the ”PIO Debug” option is selected). You can find this button near the top of the window (see Figure 15). The program will first compile and then debugging will start.

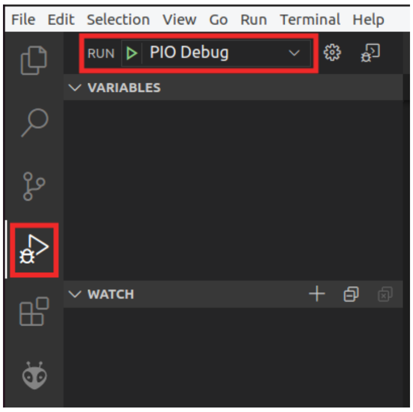


Figure 15. Compile and download the program and start the debugger

To control your debugging session, you can use the debugging toolbar which appears near the top of the editor (see Figure 16). We will describe and test all the options later in this Getting Started Guide.

Figure 16. Debugging tools

PlatformIO sets a temporary breakpoint at the beginning of the main function. So, click on the Continue button  to run the program. Now toggle the switches on the Nexys A7 FPGA board and view as the corresponding LEDs light up.

3. RISC-V ARCHITECTURE OVERVIEW

RISC-V is an Instruction Set Architecture (ISA) that was created in 2011 in the Par Lab at the University of California, Berkeley. The goal was for RISC-V to become a “Universal ISA” for processors used for the entire range of applications, from small, constrained, low-resource IoT devices to supercomputers. RISC-V architects established five principles for the architecture to achieve this goal:

- It must be compatible with a wide range of software packages and programming languages.
- Its implementation must be feasible in all technology options, from FPGAs to ASICs (application specific integrated circuits) as well as emerging technologies.
- It must be efficient in the various microarchitecture scenarios, including those implementing microcode or hardwired control, in-order or out-of-order pipelines, various types of parallelism, etc.
- It must be able to be tailored to specific tasks to achieve the required maximum performance without drawbacks imposed by the ISA itself.
- Its base instruction set must be stable and long-lasting, offering a common and solid framework for developers.

RISC-V is an open standard, in fact, the specification is public domain, and it has been managed since 2015 by the **RISC-V Foundation**, now called **RISC-V International**, a non-profit organization promoting the development of hardware and software for RISC-V architectures. In 2018, the RISC-V Foundation began an ongoing collaboration with the Linux Foundation, and in March 2020 the RISC-V Foundation became RISC-V International headquartered in Switzerland. This transition dissipated any concern the community might have had about future openness of the standard. As of 2020, RISC-V International is supported by more than 200 key players from research, academia, and industry, including Microchip, NXP, Samsung, Qualcomm, Micron, Google, Alibaba, Hitachi, Nvidia, Huawei, Western Digital, ETH Zurich, KU Leuven, UNLV, and UCM.

RISC-V is one of the few, and probably the only, globally relevant ISAs created in the past 10-20 years because of it being an open standard and modular, instead of incremental. Its modularity makes it both flexible and sleek. Processors implement the base ISA and only those extensions that are used. This modular approach differs from traditional ISAs, such as x86 or ARM, that have incremental architectures, where previous ISAs are expanded and each new processor must implement all instructions, even those that are tagged as “obsolete”, to ensure compatibility with older software programs. As an example, x86, that started with 80 instructions, has now over 1300, or 3600 if you consider all different opcodes available in machine code. This large number of instructions and the requirement of backward compatibility result in large, power-hungry processors that must support long instructions, because most of the short opcodes, or small instructions, are already in use.

RISC-V has four base ISA options: two 32-bit versions (integer and embedded versions, RV32I and RV32E) and 64- and 128-bit versions (RV64I and RV128I), as shown in Table 3. The ISA modules marked Ratified have been ratified at this time. The modules marked Frozen are not expected to change significantly before being put up for ratification. The modules marked Draft are expected to change before ratification. The ability to build small processors is a particularly key requirement for cost-, space-, and energy-constrained devices. Instruction extensions can be added on top of these base ISAs to enable specific tasks, for example floating point operations, multiplication and division, and vector operations. These specialized hardware extensions are also included in the standard and known by the compilers, so enabling the desired options in a compiler will allow for a targeted binary code generation. Each of these extensions is identified by a letter that must be added to the core ISA to represent the hardware capabilities of the implementation, as shown in Table 4. For example, RVM is the multiply/divide extension, RVF is the floating-point extension, and so on.

Table 3. RISC-V base ISAs
(table from https://riscv.org/technical/specifications/)

Base	Version	Status
RVWMO	2.0	**Ratified**
RV32I	**2.1**	**Ratified**
RV64I	**2.1**	**Ratified**
RV32E	*1.9*	*Draft*
RV128I	*1.7*	*Draft*

Table 4. RISC-V standard ISA extensions
(table from https://riscv.org/technical/specifications/)

Extension	Version	Status
Zifencei	**2.0**	**Ratified**
Zicsr	**2.0**	**Ratified**
M	**2.0**	**Ratified**
A	*2.0*	Frozen
F	**2.2**	**Ratified**
D	**2.2**	**Ratified**
Q	**2.2**	**Ratified**
C	**2.0**	**Ratified**
Ztso	*0.1*	*Frozen*
Counters	*2.0*	*Draft*
L	*0.0*	*Draft*
B	*0.0*	*Draft*
J	*0.0*	*Draft*
T	*0.0*	*Draft*
P	*0.2*	*Draft*
V	*0.7*	*Draft*
N	*1.1*	*Draft*
Zam	*0.1*	*Draft*

The letter G, that denotes "general", is used to denote the inclusion of all MAFD extensions. Note that a company or an individual may develop proprietary extensions using opcodes that are guaranteed to be unused in the standard modules. This allows third-party implementations to be developed in a faster time-to-market.

For example, a 64-bit RISC-V implementation, including all four general ISA extensions plus *Bit Manipulation* and *User Level Interrupts*, is referred to as an RV64GBN ISA. All these modules are covered in the unprivileged or user specification. RISC-V International also covers a set of requirements and instructions for privileged operations required for running general-purpose operating systems.

4. RVFPGA SYSTEM OVERVIEW

In this section we describe the entire RVfpga system from the core up to the FPGA board interface. Figure 17 illustrates the typical hierarchical organization of an embedded system starting with the processor core, then the SoC built around the core, and finally the system and board interface.

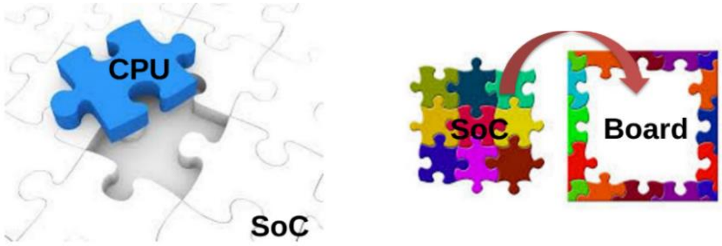


Figure 17. Embedded System organization

Figure 1 and Table 1 show the hierarchical organization of our system, from the SweRV EH1 Core up to RVfpgaNexys and RVfpgaSim. In the following sections, we start by describing the processor core (**Western Digital's SweRV EH1 Core**), which executes the RISC-V instructions; then, in Section B, we describe the **SweRVolfX SoC**, which integrates the system's hardware components (core, memory, and input/output), and the extensions performed for using it within RVfpga; in Section C we describe the SweRVolfX SoC implemented on the Nexys A7 FPGA board (**RVfpgaNexys**) and also describe the SweRVolfX SoC used in simulation (**RVfpgaSim**). Finally, we explain the file structure of the whole RVfpga System in Section D.

A. SweRV EH1 Core and SweRV EH1 Core Complex

Western Digital developed three RISC-V cores over the past few years: **SweRV EH1** (the core used in the RVfpga System), SweRV EH2, and SweRV EL2 (future versions of RVfpga may include these cores). Each core has an Apache 2.0 license. The SweRV EH1 Core is a 32-bit, 2-way superscalar, 9-stage pipeline core. The SweRV Core EH2 builds on and expands the EH1 Core to add dual threaded capability for additional performance. The SweRV Core EL2 is a smaller core with moderate performance. The RISC-V page at https://www.westerndigital.com/company/innovations/risc-v outlines the three available cores, whose main features are given in Table 5.

Table 5. Main features of the three WD RISC-V Cores

(table from https://www.westerndigital.com/company/innovations/risc-v)

Core Name	RISC-V Type	Pipeline Stages	Threads	Size @ TSMC	CoreMarks/Mhz
SweRV Core EH1	RV32IMC	9- dual issue	Single	.11mm @ 28nm	4.9
SweRV Core EH2	RV32IMC	9- dual issue	Dual	.067 @ 16nm	6.3
SweRV Core EL2	RV32IMC	4- single issue	Single	.023 @ 16nm	3.6

Out of the three cores, the **SweRV EH1 Core** (provided with the RVfpga package and also available from https://github.com/chipsalliance/Cores-SweRV) is preferred for its high performance/MHz and its simple thread structure. Moreover, Chips Alliance, a group committed to providing open-source hardware, provides a complete and verified SoC, called SweRVolf (provided with the RVfpga package and also available from https://github.com/chipsalliance/Cores-SweRVolf). The RVfpga System uses an extension of the SweRVolf SoC that, in turn, uses Western Digital's **SweRV EH1 Core** version **1.8**.

The **SweRV EH1 Core** is a machine-mode (M-mode) only, 32-bit CPU core which supports RISC-V's integer (I), compressed instruction (C), and integer multiplication and division (M) extensions. The Programmer's Reference Manual (https://github.com/chipsalliance/Cores-SweRV/blob/master/docs/RISC-V_SweRV_EH1_PRM.pdf) describes in detail all aspects of the core, from its structure to timing information and memory maps. SweRV EH1 is a superscalar core, with a dual-issue 9-stage pipeline (see Figure 18) that supports four arithmetic logic units (ALUs), labelled EX1 to EX4 in two pipelines, I0 and I1. Both ways of the pipeline support ALU operations. One way of the pipeline supports loads/stores and the other way has a 3-cycle latency multiplier. The processor also has one out-of-pipeline 34-cycle latency divider. Four stall points exist in the pipeline: 'Fetch 1', 'Align', 'Decode', and 'Commit'. The 'Fetch 1' stage includes a Gshare branch predictor. In the 'Align' stage, instructions are retrieved from three fetch buffers. In the 'Decode' stage, up to two instructions from four instruction buffers are decoded. In the 'Commit' stage, up to two instructions per cycle are committed. Finally, in the 'Writeback' stage, the architectural registers are updated.

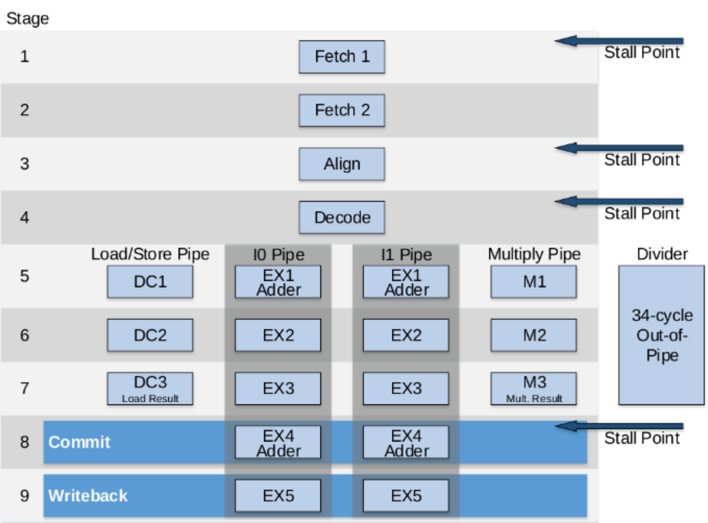


Figure 18. SweRV EH1 core microarchitecture

(figure from https://github.com/chipsalliance/Cores-SweRV/blob/master/docs/RISC-V_SweRV_EH1_PRM.pdf)

Figure 19 shows a comparison of different current cores and processors. The SweRV EH1 Core performance per MHz is impressively high at 4.9 CM/MHz (CoreMark per MHz): it is twice as fast as the ARM Cortex A8 and its performance even surpasses the ARM Cortex A15 performance.

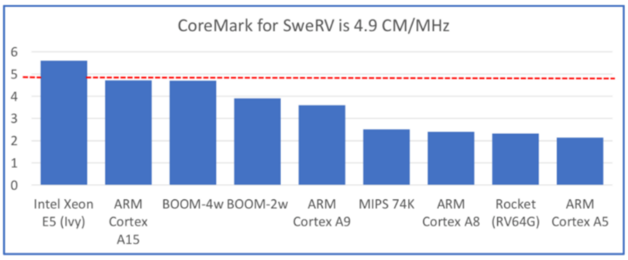


Figure 19. Benchmark comparison per thread and MHz

(figure from https://content.riscv.org/wp-content/uploads/2019/12/12.11-14.20a3-Bandic-WD_SweRV_Cores_Roadmap_v4SCR.pdf)

Western Digital also provides an extension to the SweRV EH1 Core called the **SweRV EH1 Core Complex** (see Figure 20), which adds the following elements to the EH1 Core described above and coloured in blue in the figure:

- Two dedicated memories, one for instructions (ICCM) and the other for data (DCCM), which are tightly coupled to the core. These memories provide low-latency access and SECDED ECC (single-error correction and double-error detection error correcting codes) protection. Each of the memories can be configured as 4, 8, 16, 32, 48, 64, 128, 256, or 512KB.
- An optional 4-way set-associative instruction cache with parity or ECC protection.
- An optional Programmable Interrupt Controller (PIC), that supports up to 255 external interrupts.
- Four system bus interfaces for instruction fetch (IFU Bus Master), data accesses (LSU Bus Master), debug accesses (Debug Bus Master), and external DMA accesses (DMA Slave Port) to closely coupled memories (configurable as 64-bit AXI4 or AHB-Lite buses).
- Core Debug Unit compliant with the RISC-V Debug specification.

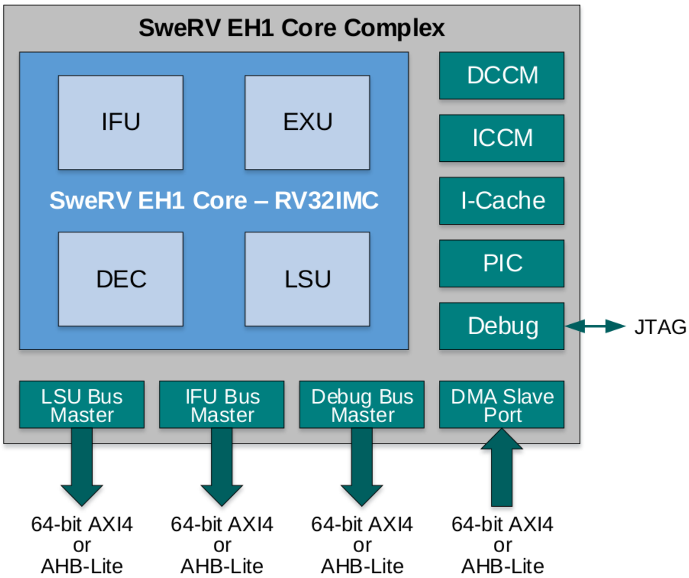


Figure 20. SweRV EH1 Core Complex

(figure from https://github.com/chipsalliance/Cores-SweRV/blob/master/docs/RISC-V_SweRV_EH1_PRM.pdf)

B. SweRVolfX SoC

The System on Chip (SoC) used in this RVfpga package, called SweRVolfX and illustrated in Figure 21, is based on SweRVolf version 0.7.3 (https://github.com/chipsalliance/Cores-SweRVolf/releases/tag/v0.7.3), which is built on top of the SweRV EH1 Core Complex. In addition to the SweRV EH1 Core Complex (see Figure 20), the SweRVolf SoC also includes a Boot ROM, a UART, a System Controller and an SPI controller (Figure 21 shows these elements in white). Given that the SweRV EH1 Core uses an AXI bus and the peripherals use a Wishbone bus, the SoC also has an AXI-Wishbone Bridge.

In RVfpga we extend the SweRVolf SoC with some more functionality, such as another SPI controller (SPI2), a GPIO (General Purpose Input/Output) controller, a PTC (PWM/Timer/Counter) module and a controller for interfacing with 8-digit 7-Segment Displays. Figure 21 shows these new peripherals in red, except for the 7-Segment Displays controller, which is included in the System Controller. We call this System on Chip **SweRVolfX** (the X stands for eXtended).

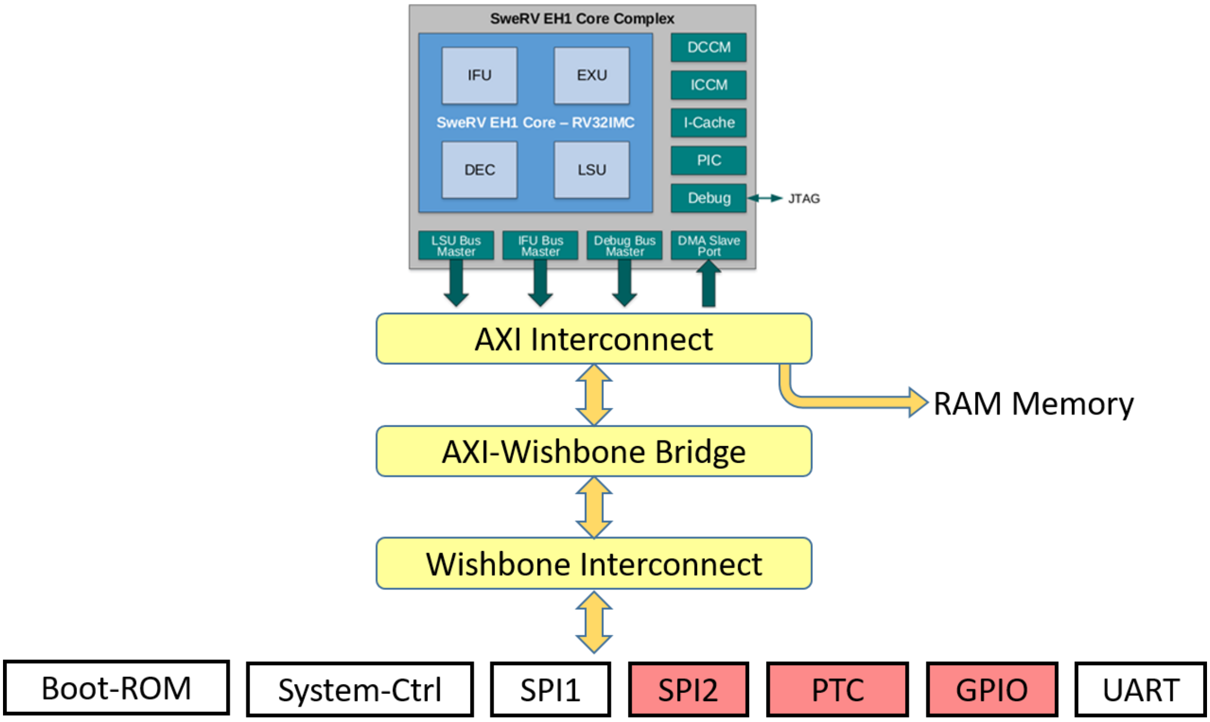


Figure 21. SweRVolfX (SweRVolf eXtended with 4 new peripherals) System on Chip

Table 6 shows the memory-mapped addresses of the peripherals connected to the core via the Wishbone interconnect.

Table 6. Memory-mapped addresses of Extended SweRVolfX SoC peripherals

System	Address
Boot ROM	0x80000000 - 0x80000FFF
System Controller	0x80001000 - 0x8000103F
SPI1	0x80001040 - 0x8000107F
SPI2	0x80001100 - 0x8000113F
PTC	0x80001200 - 0x8000123F
GPIO	0x80001400 - 0x8000143F
UART	0x80002000 - 0x80002FFF

i. Input/Output

The SweRVolfX SoC uses two kinds of hardware controllers for communicating with the peripherals: custom controllers written in Verilog and open-source controllers from OpenCores [https://opencores.org/], an online community for the development of gateware IP (Intellectual Properties) cores in the spirit of free and open source collaboration. The SweRVolfX SoC that we use in this course includes the I/O interfaces listed below, which we will use, explain in detail and even extend in RVfpga Labs 6-20.

- **System Controller:** the system controller contains common system functionality such as keeping register with the SoC version information, RAM initialization status and the RISC-V machine timer. At https://github.com/chipsalliance/Cores-SweRVolf you can find the complete memory map. We have modified this module as follows:

- We have included a new controller for communicating with the 8-digit 7-Segment Displays available on the Nexys A7 board, called **SevSegDisplays_Controller**, and we have included two new registers for this controller mapped in addresses 0x80001038 and 0x8000103C.
- We have added two 1-bit registers for handling interrupts from the GPIO and the PTC, mapped in address 0x80001018.
- We have removed the simple GPIO registers provided by SweRVolf and mapped in addresses 0x80001010–0x80001017. Note that we have added a more complete GPIO controller as described below.

- **SPI:** two open-source SPI controllers (obtained from https://opencores.org/projects/simple_spi and named SPI1 and SPI2) are implemented in SweRVolfX. Their exposed registers (SPI_SPCR, SPI_SPSR, SPI_SPDR, SPI_SPER, SPI_SPSS) are mapped between addresses 0x80001040 and 0x8000107F (for SPI1) and between addresses 0x80001100 and 0x8000113F (for SPI2).

- **PTC:** We use the timer module from https://opencores.org/projects/ptc. Its registers are mapped in the address range 0x80001200 to 0x800012FF.

- **GPIO:** We use the GPIO controller from https://opencores.org/projects/gpio. It includes 32 I/O ports mapped in the address range 0x80001400 to 0x800014FF. Each pin is connected with a tristate buffer, so it can be configured as input or output.

- **UART:** an open-source UART controller (obtained from https://opencores.org/projects/uart16550) is available in SweRVolfX. Its exposed registers are mapped between addresses 0x80002000 and 0x80002FFF.

ii. Memory

The SweRVolfX SoC includes a Boot ROM memory and the necessary hardware to enable the user to include RAM and SPI Flash memories.

- **Boot ROM:** a Boot ROM contains a first-stage bootloader. After system reset, the SweRVolfX SoC will start fetching the initial instructions from this area, which occupies addresses 0x80000000 to 0x80000FFF.

- **RAM:** the SweRVolfX SoC does not include a memory controller, but it reserves the first 128MiB of its memory map (0x00000000-0x07FFFFFF) and exposes the AXI bus, so that the user can access RAM memory by using a memory controller.

- **SPI Flash:** an SPI Flash memory can also be included using the SPI1 controller described in the previous section (address range: 0x80001040-0x8000107F).

iii. Interconnection

The SweRV EH1 Core uses an AXI4 bus to connect the core and memory. The bus could also be configured as an AHB-Lite bus, but we will not use that option in these materials. All of the peripherals (I/O devices) are connected to a Wishbone bus, an open source bus that is heavily used in OpenCore CPU's and peripherals. The system includes an AXI to Wishbone Bridge (as shown in Figure 21) to connect the core to the peripherals.

In this section, we briefly describe the operation of an AXI4 bus and a Wishbone bus. If you are interested in extending your knowledge about the specification of these buses, you can use the references provided below.

The AXI4 Bus

The SweRV EH1 Core Complex uses an AXI4 Interconnect for communicating with the outside world (see Figure 20). The Advanced eXtensible Interface (AXI) is a common bus used by many processors and it is part of the ARM Advanced Microcontroller Bus Architecture on-chip interconnect specification.

In the following subsections, we briefly explain some of the main aspects of the AXI4 interconnect. You can find the whole AXI specification in the following document: https://static.docs.arm.com/ihi0022/e/IHI0022E_amba_axi_and_ace_protocol_spec.pdf

- **AXI Bus Main Features**

The main features of the AXI bus technology are as follows:

- It is suitable for both high-bandwidth and low-latency designs
- It provides high-frequency operation without using complex bridges
- It can meet the interface requirements of a wide range of components
- It is suitable for memory controllers with high initial access latency
- It provides flexibility in the implementation of interconnect architectures
- It is backward compatible with existing AHB and APB interfaces
- It provides separate address/control and data phases
- It includes support for unaligned data transfers (using byte strobes)
- It allows burst-based transactions with only the start address issued
- It provides separate read and write data channels, which can allow low-cost DMA
- It allows address information to be issued ahead of the actual data transfer
- It provides support for issuing multiple outstanding addresses and out-of-order transaction completion
- It allows easy addition of register stages to provide timing closure

- **AXI Architecture**

The AXI protocol defines the following independent transaction channels:

- Read address
- Read data
- Write address
- Write data
- Write response

Figure 22 shows how a read transaction uses the read address and read data channels. First the address and control bits are sent from the master device, then the slave device responds with the data on the read data channel.

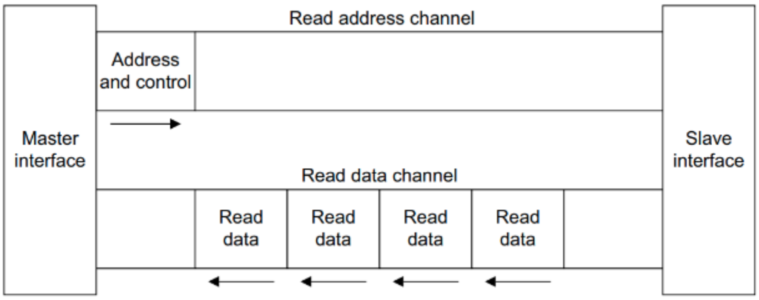


Figure 22. Channel architecture of reads
(figure from https://static.docs.arm.com/ihi0022/e/IHI0022E_amba_axi_and_ace_protocol_spec.pdf)

Figure 23 shows how a write transaction uses the write address, write data, and write response channels. Similar to a read, the master device sends the address and control bits. Then the master device sends the data on the write data channel and the slave device sends a response.

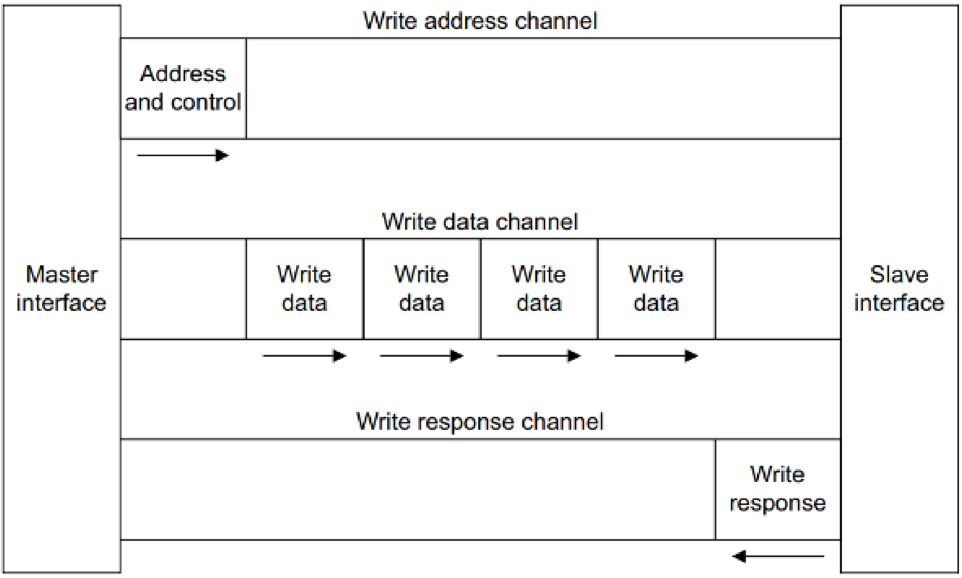


Figure 23. Channel architecture of writes
(figure from https://static.docs.arm.com/ihi0022/e/IHI0022E_amba_axi_and_ace_protocol_spec.pdf)

The AXI address channel carries addresses and control information that describes the nature of the data to be transferred. The data is transferred between the master and slave using either:

- A read data channel to transfer data from the slave to the master (Figure 22).
- A write data channel to transfer data from the master to the slave (Figure 23). In a write transaction, the slave uses the write response channel to signal the completion of the transfer to the master (Figure 23).

- **AXI Signals**

Table 7. shows the main signals used in the AXI bus and a brief description of each of them. The signals are organized in five groups, which correspond to the five channels described in the previous section:

- **Write address channel** signals, whose names start with **AW**
- **Write data channel** signals, whose names start with **W**

- **Write response channel** signals, whose names start with **B**
- **Read address channel** signals, whose names start with **AR**
- **Read data channel** signals, whose names start with **R**

Table 7. AXI Signals

(table from https://static.docs.arm.com/ihi0022/e/IHI0022E_amba_axi_and_ace_protocol_spec.pdf)

Signal	Source: master/ slave	Input/ Output	Description
Aclk	Global	Input	Global clock signal.
AResetn	Global	Input	Global reset signal
AWID[3:0]	Master	Input	Write address ID.
AWADDR[31:0]	Master	Input	Write address.
AWLEN[3:0]	Master	Input	Write burst length.
AWSIZE[2:0]	Master	Input	Write burst size.
AWBURST[1:0]	Master	Input	Write burst type.
AWLOCK[1:0]	Master	Input	Write lock type.
AWCACHE[3:0]	Master	Input	Write cache type.
AWPROT[2:0]	Master	Input	Write protection type.
WDATA[31:0]	Master	Input	Write data.
ARID[3:0]	Master	Input	Read address ID.
ARADDR[31:0]	Master	Input	Read address.
ARLEN[3:0]	Master	Input	Read Burst length.
ARSIZE[2:0]	Master	Input	Read Burst size.
ARLOCK[1:0]	Master	Input	Read Lock type.
ARCACHE[3:0]	Master	Input	Read Cache type.
ARPROT[2:0]	Master	Input	Read Protection type.
RDATA[31:0]	Master	Input	Read data.
WLAST	Master	Input	Write last.
RLAST	Slave	Output	Read last.
AWVALID	Master	Output	Write address valid.
AWREADY	Slave	Output	Write address ready.
WVALID	Master	Output	Write valid.
RAVLID	Slave	Output	Read valid.
WREADY	Slave	Output	Write ready.
BID[3:0]	Slave	Output	Write Response ID.
RID[3:0]	Slave	Output	Read response ID.
BRESP[1:0]	Slave	Output	Write response.
RRESP[1:0]	Slave	Output	Read response.
BVALID	Slave	Output	Write response valid.

The Wishbone bus

The SweRVolfX peripherals use the Wishbone System-on-Chip (SoC) Interconnection Architecture for Portable IP Cores (https://opencores.org/howto/wishbone). The main purpose of this bus is to foster design reuse by alleviating System-on-Chip integration problems. Previously, IP cores used non-standard interconnection schemes that made them difficult to integrate. These non-standard interconnects required the creation of custom glue logic to connect each of the cores together. By adopting a standard interconnection scheme such as the Wishbone bus, cores can be integrated more quickly and easily by the end user.

- **Wishbone main features**

The main features of this Wishbone bus technology are as follows:

- It supports structured design methodologies used by large project teams.
- It includes a full set of popular data transfer bus protocols including:
  1. READ/WRITE cycles
  2. BLOCK transfer cycles

iii. READ/MODIFY/WRITE cycles

- It provides modular data bus widths and operand sizes up to 64-bits.
- It supports both BIG ENDIAN and LITTLE ENDIAN data ordering.
- It supports various core interconnection methods including point-to-point, shared bus, crossbar switch, and switched fabric interconnections.
- It includes handshaking protocols that allow each IP core to throttle its data transfer speed.
- It supports single clock data transfers.
- It supports normal cycle termination, retry termination, and termination due to error.
- It includes modular address widths.
- It provides a partial address decoding scheme for slaves. This facilitates high speed address decoding, uses less redundant logic, and supports variable address sizing and interconnection methods.
- It provides user-defined tags. These are useful for applying information to an address or data bus or a bus cycle. User-defined tags are especially helpful when modifying a bus cycle to identify information such as:
    i. Data transfers
    ii. Parity or error correction bits
    iii. Interrupt vectors
    iv. Cache control operations
- It includes a Master/Slave architecture for flexible system designs.
- It has multiprocessing (multi-MASTER) capabilities. This allows for a wide variety of SoC configurations
- It includes an arbitration methodology that can be defined by the end user (priority arbiter, round-robin arbiter, etc.)

- **Wishbone Architecture and Signals**

Figure 24 illustrates the standard connection between a master (in our case, the SweRV EH1 Core) and a slave (in our case, a peripheral such as the GPIO, the SPI…) through a Wishbone bus. The Wishbone bus is much simpler than the AXI4 bus and, as shown in Table 8, it uses fewer signals.

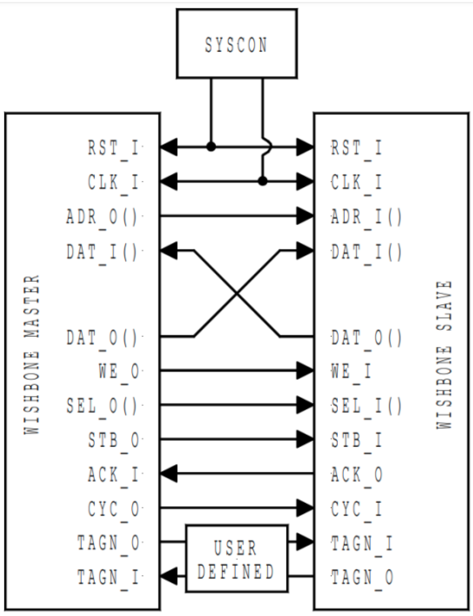


Figure 24. Wishbone Architecture
(figure from https://opencores.org/howto/wishbone)

Table 8. Wishbone Signals
(table from https://opencores.org/howto/wishbone)

Signal name	description
CLK_O	It coordinates all activities for the internal logic within the WISHBONE interconnect. The INTERCON module connects the [CLK_O] output to the [CLK_I] input on MASTER and SLAVE
RST_O	It forces all WISHBONE interfaces to restart. All internal self-starting state machines are forced into an initial state. The INTERCON connects the [RST_O] output to the [RST_I] input on MASTER and SLAVE

Signal name	Description
CLK_I	All WISHBONE output signals are registered at the rising edge of [CLK_I]. All WISHBONE input signals are stable before the rising edge of [CLK_I].
DAT_I()	The data input array [DAT_I()] is used to pass binary data. The array boundaries are determined by the port size, with a maximum port size of 64-bits (e.g. [DAT_I(63..0)]).
DAT_O()	The data output array [DAT_O()] is used to pass binary data. The array boundaries are determined by the port size, with a maximum port size of 64-bits (e.g. [DAT_I(63..0)]).
RST_I()	The reset input [RST_I] forces the WISHBONE interface to restart
TGD_I()	Data tag type [TGD_I()] is used on MASTER and SLAVE interfaces. It contains information that is associated with the data input array [DAT_I()], and is qualified by signal [STB_I].
TGD_O()	Data tag type [TGD_O()] is used on MASTER and SLAVE interfaces. It contains information that is associated with the data output array [DAT_O()], and is qualified by signal [STB_O]

Signal name	Description
ACK_I	The acknowledge input [ACK_I], when asserted, indicates the normal termination of a bus cycle
CYC_O	The cycle output [CYC_O], when asserted, indicates that a valid bus cycle is in progress
STALL_I	The pipeline stall input [STALL_I] indicates that current slave is not able to accept the transfer in the transaction queue
ERR_I	The error input [ERR_I] indicates an abnormal cycle termination
RTY_I	The retry input [RTY_I] indicates that the interface is not ready to accept or send data, and that the cycle should be retried
STB_O	The strobe output [STB_O] indicates a valid data transfer cycle
WE_O	The write enable output [WE_O] indicates whether the current local bus cycle is a READ or WRITE cycle

Signal name	Description
ACK_O	The acknowledge output [ACK_O], when asserted, indicates the termination of a normal bus cycle
CYC_I	The cycle input [CYC_I], when asserted, indicates that a valid bus cycle is in progress
STALL_O	The pipeline stall signal [STALL_O] indicates that the slave can not accept additional transactions in its queue
ERR_O	The error output [ERR_O] indicates an abnormal cycle termination
RTY_O	The retry output [RTY_O] indicates that the indicates that the interface is not ready to accept or send data, and that the cycle should be retried
STB_I	The strobe input [STB_I], when asserted, indicates that the SLAVE is selected. A SLAVE shall respond to other WISHBONE signals only when this [STB_I] is asserted
WE_I	The write enable input [WE_I] indicates whether the current local bus cycle is a READ or WRITE cycle

C. SweRVolfX SoC on the Nexys A7 FPGA Board and in Simulation

The SweRVolfX SoC (Figure 21) can run either (1) on the Nexys A7 (or Nexys4 DDR) FPGA board, which configuration is referred to as **RVfpgaNexys** in this course**,** or (2) in simulation, which is referred to as **RVfpgaSim** in this course.

i. RVfpgaNexys

RVfpgaNexys is the SweRVolfX SoC targeted to the Digilent Nexys A7 FPGA board (Figure 25). RVfpgaNexys is the same as SweRVolf Nexys (https://github.com/chipsalliance/Cores-SweRVolf), except that the latter is based on SweRVolf. The main elements used by **RVfpgaNexys** are illustrated in Figure 25:

- Hardware programmed onto the FPGA:
    - **SweRVolfX SoC** (illustrated in Figure 21)
    - **Lite DRAM controller**
    - **Clock Generator:** the Nexys A7 board includes a single **100 MHz** crystal oscillator that is used by the **Lite DRAM controller**. The frequency of this clock is scaled down to **50 MHz** to use in the **SweRVolfX SoC**.
    - **Clock Domain Crossing module:** connection of 2 clock domains: SweRVolfX SoC and Lite DRAM.
    - **BSCAN logic for the JTAG:** you can find more information about this

module at https://github.com/chipsalliance/Cores-SweRVolf/issues/29.

- Memory/Peripherals used in RVfpgaNexys from the Nexys A7 (or Nexys4 DDR) FPGA board:
  - **DDR2 memory** (accessed through the Lite DRAM controller mentioned above)
  - **USB** connection
  - **SPI Flash memory**
  - **SPI Accelerometer**
  - **16 LEDs** and **16 Switches**
  - **8-digit 7-Segment Displays**

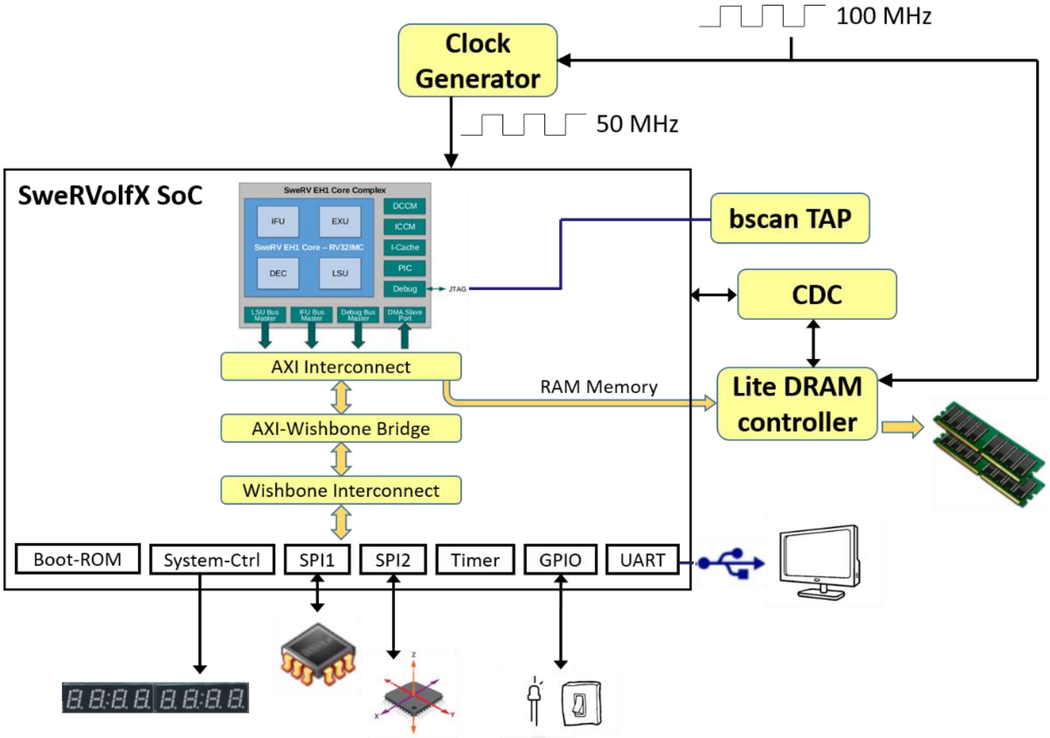


Figure 25. RVfpgaNexys

The Nexys A7 board (Figure 26) is a recommended trainer board for electrical and computer engineering curricula. This board costs $265 (or a discounted price of $198.75 with academic pricing – sign up for a Digilent account with a .edu email address). Digilent provides an extensive reference manual of the Nexys A7 board at: https://reference.digilentinc.com/_media/reference/programmable-logic/nexys-a7/nexys-a7_rm.pdf. This board may be powered from a 5V wall wart (not provided with the board) or from a PC via the microUSB connector on the board. A Microchip PIC24 microcontroller manages the loading process onto the FPGA, making this board a user-friendly option. The board is programmable using Xilinx's Vivado Design Suite or OpenOCD. The desired configuration can be downloaded to the FPGA using one of four different sources: a FAT32 formatted MicroSD card, a FAT32 formatted USB pendrive, the internal flash memory, or a JTAG interface.

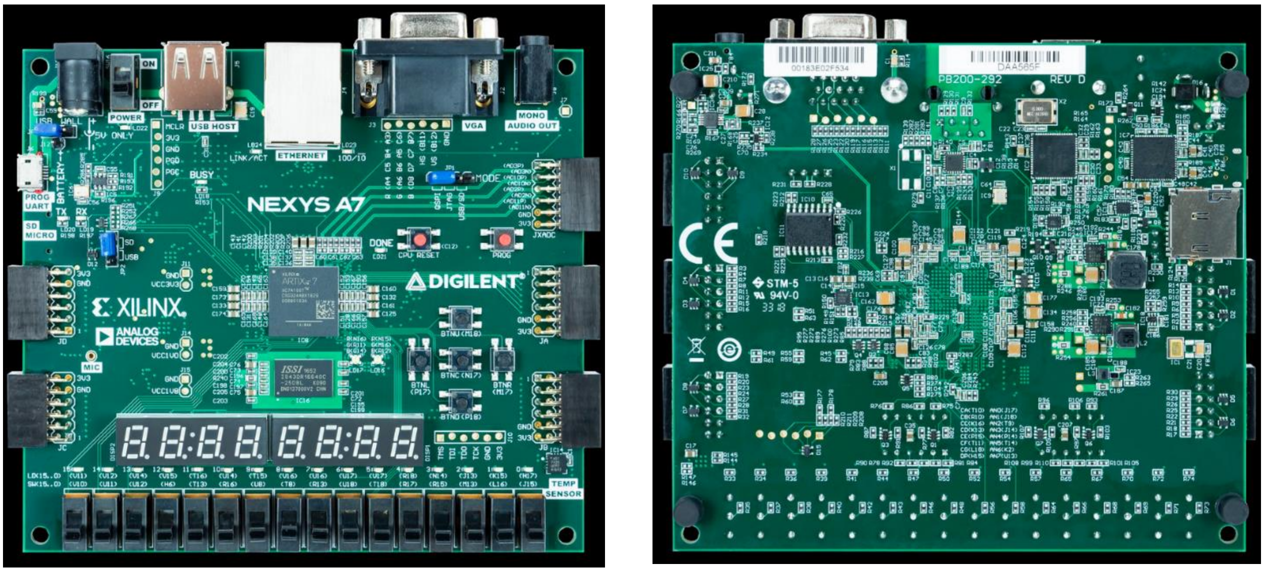


Figure 26. Digilent's Nexys A7 FPGA board
(figure from https://reference.digilentinc.com/)

The Nexys A7-100T FPGA board includes the following interfaces and devices:

- 128 MiB DDR RAM
- 128 Mibit SPI Flash Memory
- 8-digit 7-Segment Displays
- 16 Switches
- 16 LEDs
- Sensors and connectors, including a microphone, audio jack, VGA 25 port, USB host port, RGB-LEDs, I2C temperature sensor, SPI accelerometer, among other.
- Xilinx Artix-7 FPGA, which has the following features:
  - 15.850 Logic slices of four 6-input LUTs and 8 flip-flops.
  - 4.860 Kibits of total block RAM
  - 6 clock management tiles (CMTs)
  - 170 I/O pins
  - 450 MHz internal clock frequency

ii. RVfpgaSim

The SweRVolfX SoC (Figure 21) can also include a Verilog wrapper to enable simulation. **RVfpgaSim** is the SweRVolfX SoC wrapped in a testbench to be used by HDL simulators. RVfpgaSim is the same as SweRVolf sim (https://github.com/chipsalliance/Cores-SweRVolf), except that the latter is based on SweRVolf.

Although many open-source HDL simulators exist, we use Verilator (https://www.veripool.org/wiki/verilator). This open and free HDL simulator accepts synthesizable Verilog or SystemVerilog and it claims to be the fastest Verilog/SystemVerilog simulator. It is widely used in industry and academia; it provides out-of-the-box support from ARM and RISC-V vendor IPs; and it is guided by Chips Alliance and the Linux Foundation.

D. File Structure

In the previous sections we have shown the high-level organization of the system that we use in these materials, from the **SweRV EH1 Core Complex** (Figure 20), to the **SweRVolfX SoC** (Figure 21) and, finally, to **RVfpgaNexys** (Figure 25) and **RVfpgaSim** implementations.

In this section, we describe the file structure of the whole system. While reading these explanations, open the files and view them on your PC. The files are available at ***[RVfpgaPath]/RVfpga/src***.

i. SweRV EH1 Core Complex

Figure 27 shows the file structure of the **SweRV EH1 Core Complex** (Figure 20). The core is organized into three main blocks: a SweRV wrapper (highlighted in grey) that includes the SweRV EH1 Core (highlighted in green) and some other elements (such as the Interrupt Controller or the Debug Unit), and the Data/Instruction memories and Instruction Cache (highlighted in red).

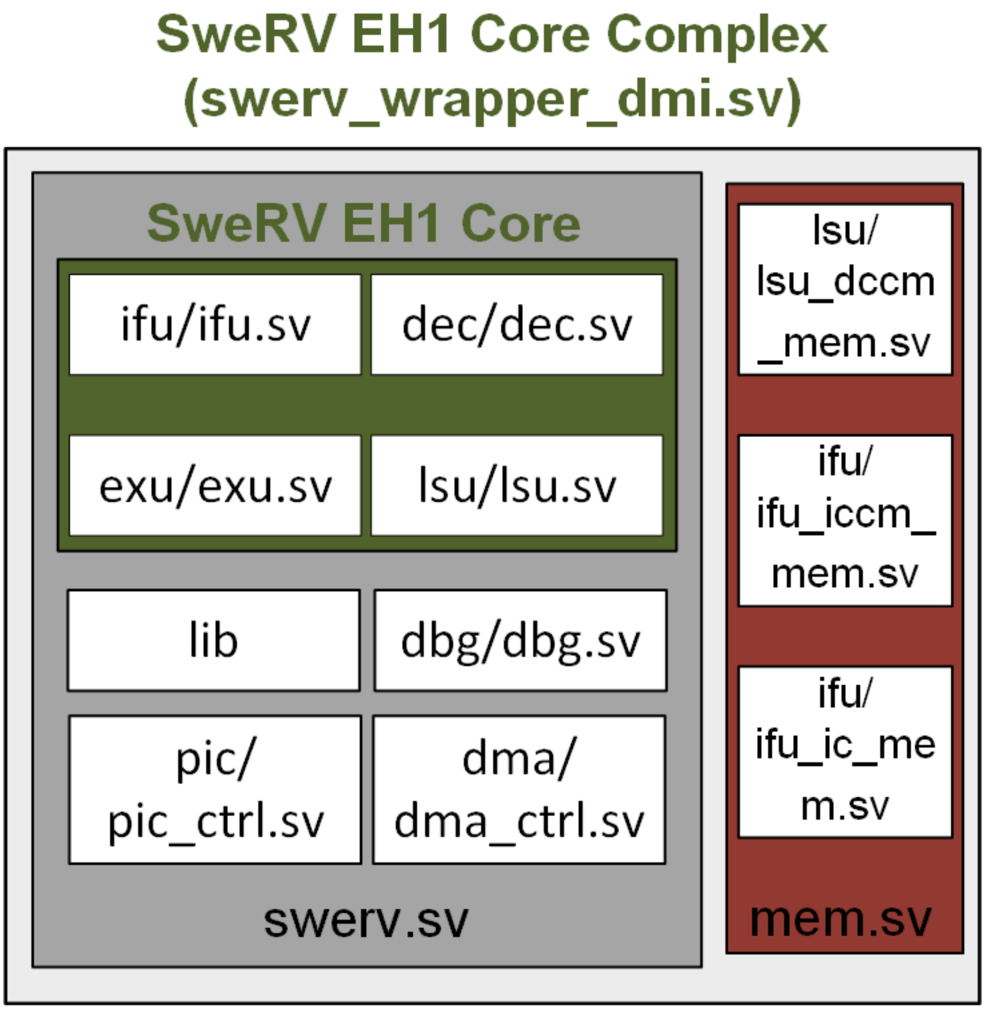


Figure 27. SweRV EH1 Core Complex

The Verilog files for the SweRV EH1 Core Complex are available in this folder:
`[RVfpgaPath]/RVfpga/src/SweRVolfSoC/SweRVEh1CoreComplex`

Find that directory on your PC to view the files as we refer to them in this section.

The top file for the SweRV EH1 Core Complex is in the file: *swerv_wrapper.sv*; the top module is called **swerv_wrapper**, and it instantiates two modules that correspond to the two blocks highlighted in grey and red in Figure 27:

- **mem** (implemented inside *mem.sv*): this module instantiates the modules for the implementation of the DCCM (**lsu_dccm_mem**, implemented in file *lsu/lsu_dccm_mem.sv*), the ICCM (**ifu_iccm_mem**, implemented in file *ifu/ifu_iccm_mem.sv*) and the Instruction Cache (**ifu_ic_mem**, implemented in file *ifu/ifu_ic_mem.sv*).

- **swerv** (implemented inside *swerv.sv*): this module instantiates the units that comprise the core.

  The SweRV EH1 Core (highlighted in green in Figure 27) consists of the following four units:
  - Folder ***ifu*** (Instruction Fetch Unit): this folder includes the Verilog files (top module available inside *ifu.sv*) for the Icache (instruction cache), Fetch, Branch Predictor and Aligner.
  - Folder ***dec*** (Decode Unit): this folder includes the Verilog files (top module available inside *dec.sv*) for the Instruction Decoding, the Dependency Scoreboard, and the Register File.
  - Folder ***exu*** (Execution Unit): this folder includes the Verilog files (top module available inside *exu.sv*) for the arithmetic/logical units available in the core: two pipelined ALUs, one pipelined Multiplier and one out-of-pipeline Divider.
  - Folder ***lsu*** (Load Store Unit): this folder includes the Verilog files (top module available inside *lsu.sv*) for the pipelined Load/Store Unit.

  Other units included in this module are:
  - Folder ***dbg*** (Debug Unit): this folder includes the Verilog files (top module available inside *dbg.sv*) of the Debug Unit, which is responsible to put the rest of the core in quiescent mode, send the commands/address, send write data and receive read data, and then resume the core to do the normal mode.
  - Folder ***lib***: this folder includes the Verilog files for the AXI and AHB-Lite Buses.
  - Folder **pic**: this folder includes file *pic_ctrl.sv*, which implements the Programmable Interrupt Controller in module **pic_ctrl**.
  - Folder **dma**: this folder includes file *dma_ctrl.sv*, which implements the Direct Memory Access in module **dma_ctrl**.

ii. SweRVolfX SoC

Figure 28 shows the file structure for the **SweRVolfX SoC** shown in Figure 21. The SoC is organized as the modules that correspond to the blocks shown in Figure 21.

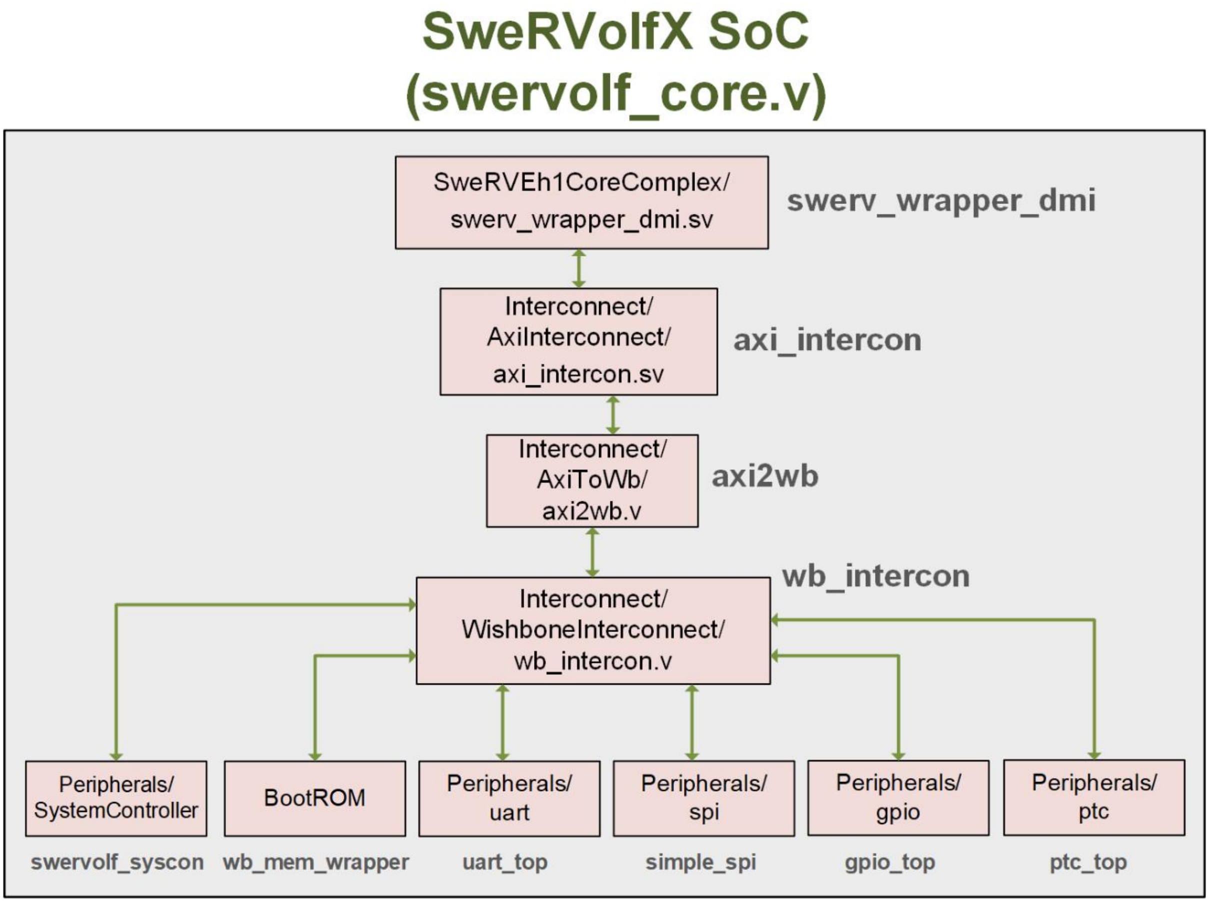


Figure 28. SweRVolfX SoC

The files for the SweRVolfX SoC are in:
[RVfpgaPath]/RVfpga/src/SweRVolfSoC

Find that directory on your PC to view the files as we refer to them in this section.

The top module for the **SweRVolfX SoC** is available at:
[RVfpgaPath]/RVfpga/src/SweRVolfSoC/swervolf_core.v. Open that file, and notice that it includes the modules contained within the SweRVolfX SoC (Figure 21), specifically:

- **axi_intercon** (available inside *Interconnect/AxiInterconnect/axi_intercon.v*): this module is included through another file at line 100 (`` `include "axi_intercon.vh" ``). It connects the SweRV EH1 Core Complex with the AXI-to-Wishbone Bridge.
- **axi2wb** (available inside *Interconnect/AxiToWb/axi2wb.v*): this module, which is instantiated in line 153 of *swervolf_core.v*, is the AXI-to-Wishbone Bridge that allows communication between the AXI based EH1 Core and the Wishbone-based peripherals.
- **wb_intercon** (available inside *Interconnect/WishboneInterconnect/wb_intercon.v*): this module is included through another file at line 145 (`` `include "wb_intercon.vh" ``). It connects the AXI-to-Wishbone Bridge with the different peripherals through a multiplexer that we will analyse and modify later.
- **wb_mem_wrapper** (available inside *BootROM/wb_mem_wrapper.v*): the wrapper for the Boot Memory described above is instantiated at line 197 of *swervolf_core.v*. It

instantiates the **dpram64** module (available inside *BootROM/dpram64.v*), which is a basic RAM module.

- **swervolf_syscon** (available inside *Peripherals/SystemController/swervolf_syscon.v*): this module, which is instantiated at line 215 of *swervolf_core.v*, defines the System Controller.
- **simple_spi**: SPI controller obtained from OpenCores and available inside *Peripherals/spi/simple_spi_top.v*. It is instantiated at lines 246 (spi) and 387 (spi2) of *swervolf_core.v*.
- **uart_top**: UART controller obtained from OpenCores and available inside *Peripherals/uart/uart_top.v*. It is instantiated at line 272 of *swervolf_core.v*.
- **gpio_top**: GPIO controller obtained from OpenCores and available inside *Peripherals/gpio/gpio_top.v*. It is instantiated at line 338 of *swervolf_core.v*.
- **ptc_top**: PTC controller obtained from OpenCores and available inside *Peripherals/ptc/ptc_top.v*. It is instantiated at line 361 of *swervolf_core.v*.
- **swerv_wrapper_dmi** (available inside *SweRVEh1CoreComplex/swerv_wrapper_dmi.v*): instantiation (line 407 of *swervolf_core.v*) of Western Digital's SweRV EH1 Core Complex, described in the previous section (Figure 27).

iii. Wrappers for on-board execution and simulation

SIMULATION:
RVfpgaSim is a simulation target that wraps the **SweRVolfX SoC** (Figure 21) in a testbench that is used by HDL simulators. It is available at *[RVfpgaPath]/RVfpga/src/rvfpgasim.v*.

ON BOARD EXECUTION:
RVfpgaNexys (available at: *[RVfpgaPath]/RVfpga/src/rvfpganexys.v*) wraps the **SweRVolfX SoC** (Figure 21) in a wrapper that targets it to the Nexys A7 FPGA board and its peripherals (see Figure 25). This module instantiates, in addition to some other modules (such as a *clock generator* module, **clk_gen_nexys**, a *clock domain crossing* module, **axi_cdc_intf**, or a BSCAN module for the JTAG port, **bscan_tap**), the two main SoC structures:

- **swervolf_core**: instantiation of the **SweRVolfX SoC** described in the previous subsection (Figure 28). This also requires a constraints file called *rvfpganexys.xdc* (available at *[RVfpgaPath]/RVfpga/src/*), which defines the connections between the SoC and the board.
- **litedram_top**: wrapper for LiteDRAM DDR2 Controller, which connects the SweRVolfX SoC with the DDR2 Memory, and which is implemented in file *[RVfpgaPath]/RVfpga/src/LiteDRAM/litedram_top.v.* This also requires a constraints file called *litedram.xdc* (available inside *[RVfpgaPath]/RVfpga/src/LiteDRAM*), which defines the connections between the Memory Controller and the on-board DDR2 Memory.

As a summary, Figure 29 shows the hierarchy for the whole RVfpgaNexys implementation on the Nexys A7 FPGA board.

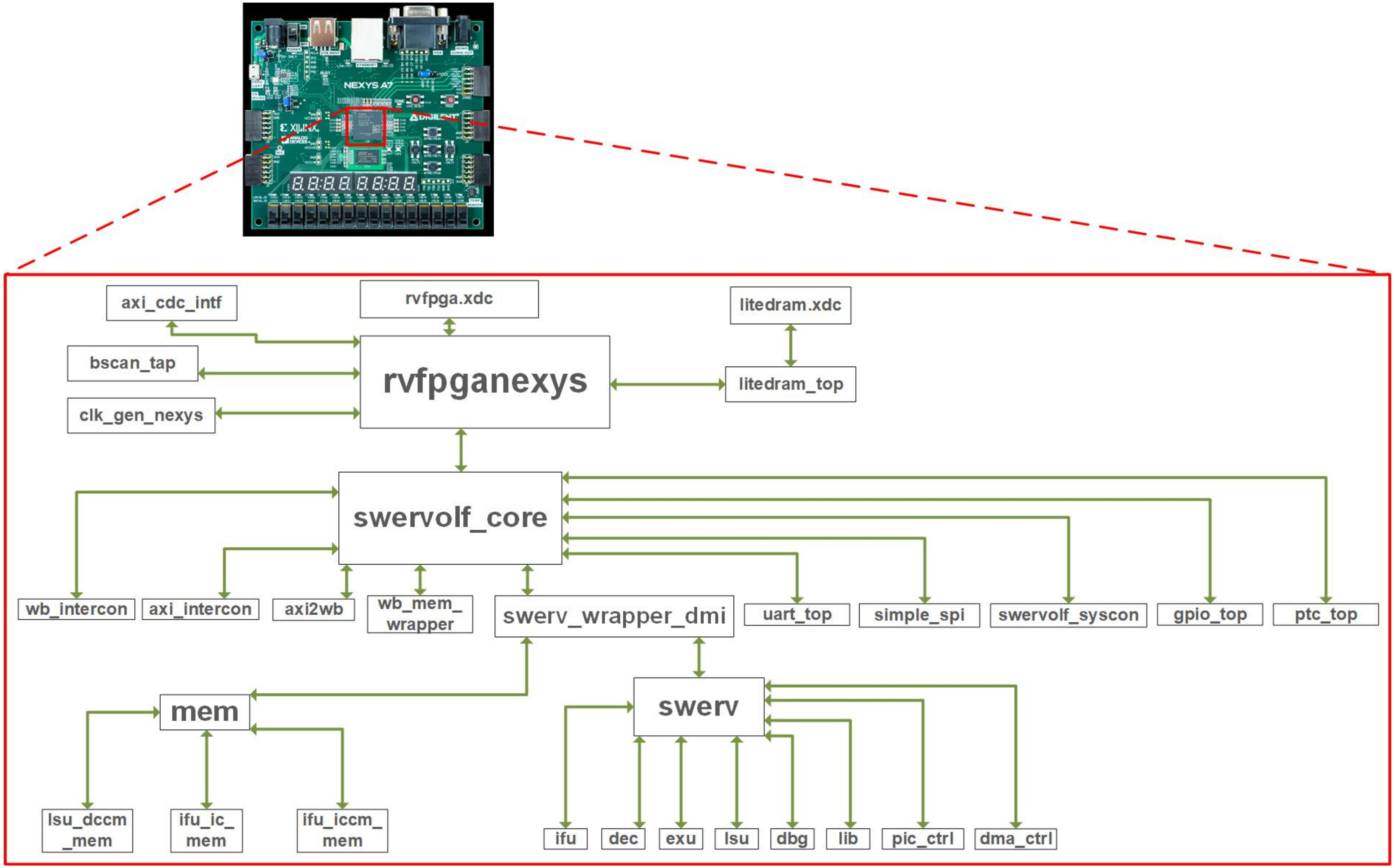


Figure 29. Modules Hierarchy for the Nexys A7 FPGA board implementation

5. INSTALLING SOFTWARE TOOLS

The instructions below are for an Ubuntu 18.04 OS, but other Linux operating systems, as well as Windows or macOS, follow similar (if not exactly the same) steps. In some cases, we insert boxes with specific instructions for those different OSs. If you are using Ubuntu, you can just ignore those boxes.

The instructions show you how to install the following tools:

- **A. Vivado**: required for resynthesizing the System on Chip. This is something that you will mainly do in Labs 6-20, where different features will be included to the baseline SoC.
- **B. VSCode (Visual Studio Code) and PlatformIO**: these are the main tools used in the GSG and in the Labs. They are used for programming the FPGA and for running/debugging programs on it.
- **C. Verilator and GTKWave**: required for simulating the SoC and analysing the different signals. Again, you will mainly use these tools in Labs 6-20.

Note that, for most things that you will do in this GSG and in the Labs, installing VSCode and PlatformIO would be enough. However, we recommend you to install the other tools now as well (Vivado, Verilator and GTKWave), so that no more installations are required later.

This process can take several hours (or more, depending on your download speed), but most of the time is spent waiting while the programs are downloaded and installed.

A. Install Vivado

Vivado is a Xilinx tool for viewing, modifying, and synthesizing the Verilog code for RISC-V FPGA. You will use it extensively in later labs. The installation instructions are available at https://reference.digilentinc.com/vivado/installing-vivado/start and are summarized below.

> **Windows:** the webpage referenced above (https://reference.digilentinc.com/vivado/installing-vivado/start) also includes detailed instructions for installing Vivado in Windows. Below we insert boxes when specific instructions are required for Windows.

> **macOS:** Vivado is not supported in macOS; thus, you need a Linux/Windows Virtual Machine for running Vivado in this OS.

1. Navigate to https://reference.digilentinc.com/vivado/installing-vivado/start

2. You will be guided to the Xilinx download page: https://www.xilinx.com/support/download.html

3. It is recommended that you install the “Self Extracting Web Installer”. At the time of writing this document, it is at this link on the download page: Xilinx Unified Installer 2019.2: Linux Self Extracting Web Installer

> **WINDOWS:** At the time of writing this document, the “Self Extracting Web Installer” for Windows is at this link on the download page: Xilinx Unified Installer 2019.2: Windows Self Extracting Web Installer

4. You will be asked to log in to your Xilinx account before you can download the installer. If you don’t already have an account, you will need to create one.

5. Execute the binary file. Open a terminal and make it root (type “sudo su”). Then drag the binary file (Xilinx_Unified_2019.2_1106_2127_Lin64.bin) into the terminal. If it prompts you to make the file executable and run it, select OK.

- **Troubleshooting:** If the terminal says permission denied, type the following in the terminal (in the same directory as the binary file):
  > `sudo chmod +x ./Xilinx_Unified_2019.2_1106_2127_Lin64.bin`
  > `sudo ./Xilinx_Unified_2019.2_1106_2127_Lin64.bin`

WINDOWS: In Windows you can simply execute the .exe file that you downloaded in steps 3 and 4 by double-clicking on it.

6. The Vivado installer will walk you through the installation process. Important notes:
- Select **Vivado** (*not* Vitis) as the Product to install.
- Select Vivado HL **Webpack** (*not* Vivado HL System Edition); Webpack is free.
- Otherwise, defaults should be selected.

Hint: If you changed the installation directory of Vivado, you will need to modify the path appropriately in the following steps.

WINDOWS: Steps 7 and 8, are not necessary in Windows. You can simply ignore these two steps and go directly to step 9.

7. After Vivado has installed, you need to set up the environment. Open a terminal and type:
```
source /tools/Xilinx/Vivado/2019.2/settings64.sh
```

Add that line (`source /tools/Xilinx/Vivado/2019.2/settings64.sh)` to your `~/.bashrc` file so that it runs each time you launch a terminal.

8. Test Vivado by typing the following in a terminal:

```
vivado
```

Troubleshooting:
- If your system cannot find that executable, you'll need to add the following to your path:

```
/tools/Xilinx/DocNav
/tools/Xilinx/Vivado/2019.2/bin
```

- If you get an error such as “application-specific initalization failed…”, type the following at a terminal:

```
sudo ln -s /lib/x86_64-linux-gnu/libtinfo.so.6 /lib/x86_64-linux-gnu/libtinfo.so.5
```

9. You will need to **manually install the cable drivers for the Nexys A7 FPGA board**. Type the following at a terminal window:
```
cd
/tools/Xilinx/Vivado/2019.2/data/xicom/cable_drivers/lin64/install_script/install_drivers/
```

```
sudo ./install_drivers
```

WINDOWS: Vivado installation in Windows automatically installs drivers for the Nexys A7 board which are not compatible with PlatformIO. Thus, if you are using Windows, **you must update the drivers as explained in Appendix B.** You must **do this even if you already did it in the Quick Start Guide section because the drivers were overwritten by the Vivado installation**.

10. You will also need to manually install the Digilent Board Files.
    - Download the archive of the vivado-boards from the Github repository and extract it.
    - Open the folder extracted from the archive and navigate to its *new/board_files* directory. Select all folders within this directory and copy them.
    - Open the folder that Vivado was installed into (*/tools/Xilinx/Vivado* by default). Under this folder, navigate to its *<version>/data/boards/board_files* directory, then paste the board files into this directory.
    - You can also use the terminal, by going into the *new/board_files* directory and typing:

```
sudo cp -r *
/tools/Xilinx/Vivado/2019.2/data/boards/board_files
```

WINDOWS: copy/paste the downloaded folders as explained in Step 10. In Windows, you can find Vivado's *board_files* folder at: *C:\Xilinx\Vivado\2019.2\data\boards\board_files*

B. Install VSCode and PlatformIO

Now you will install VSCode and PlatformIO. **If you already did this in the Quick Start Guide – Section 1 – you do not need to repeat the process here again and you can directly go to Section C**.

PlatformIO is an integrated development environment (IDE) for embedded systems that is built on top of Microsoft's Visual Studio (VS) Code. It allows you to program the RISC-V processor (that is located on the FPGA) using C or assembly. PlatformIO is cross-platform and includes a built-in debugger.

Follow these steps to install both VSCode and PlatformIO:

LINUX command-line: although using VSCode+PlatformIO is the recommended method, Appendix A provides instructions for anyone who is interested in installing and using the native RISC-V toolchain and OpenOCD in Linux and use them in place of PlatformIO. If you are going to use PlatformIO, just ignore Appendix A.

1. Install VSCode:

Follow these steps to install VSCode:

a. Download the .deb file from the following link: https://code.visualstudio.com/Download

b. Open a terminal, and install and execute VSCode by typing the following in the terminal:

```
cd ~/Downloads
```

```
sudo dpkg -i code*.deb
code
```

Windows / macOS: VSCode packages are also available for Windows (.exe file) and macOS (.zip file) at https://code.visualstudio.com/Download. Follow the common steps used for installing and executing an application in these operating systems.

2. Install PlatformIO on top of VSCode:

Follow these steps to install PlatformIO:

a. Install python3 utilities by typing the following in a terminal:
`sudo apt install -y python3-distutils python3-venv`

Windows / macOS: this step (2.a) is not required in Windows. As for macOS, you can use homebrew to install python3: `brew install python3`

b. If not yet open, start VSCode by selecting the Start button and typing “VSCode” in the search menu, then select VSCode, or by typing `code` in a terminal.

c. In VSCode, click on the Extensions icon  located on the left side bar of VSCode (see Figure 30).

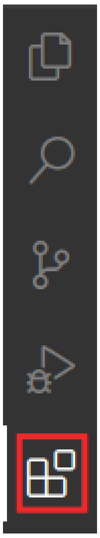


Figure 30. VSCode’s Extensions icon

d. Type *PlatformIO* in the search box and install the PlatformIO *IDE* by clicking on the install button next to it (see Figure 31).

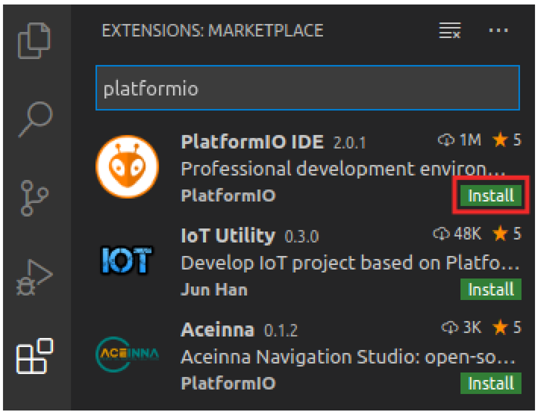


Figure 31. PlatformIO IDE Extension

e. The OUTPUT window on the bottom will inform you about the installation process. Once finished, click "Reload Now" on the bottom right side window, and PlatformIO will be installed inside VSCode (see Figure 32).

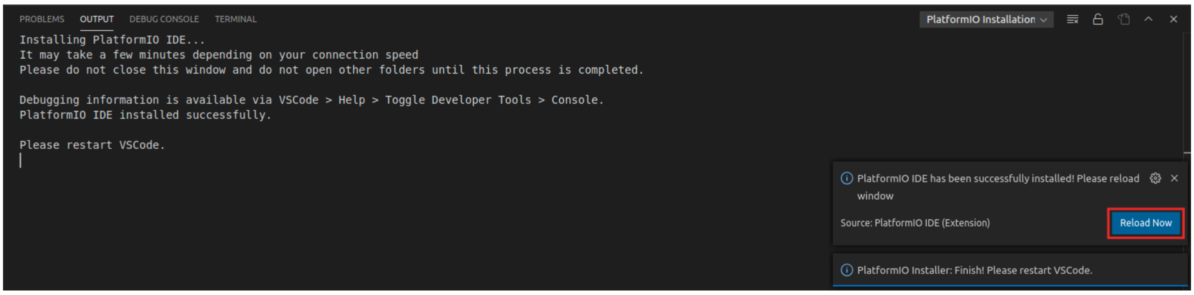


Figure 32. Reload Now after PlatformIO installs

C. Install Verilator and GTKWave in Ubuntu 18.04

The instructions in this section are valid for Linux systems only.

Windows: use Appendix C instead of the instructions provided in this section.

macOS: use Appendix D instead of the instructions provided in this section.

Follow the next steps to install Verilator (instructions are available at: https://www.veripool.org/projects/verilator/wiki/Installing but are also summarized below) and GTKWave in your Ubuntu 18.04 Linux system. This process takes a long time.

- `sudo apt-get install git make autoconf g++ flex bison libfl2 libfl-dev`
- `sudo apt-get install -y gtkwave`
- `git clone https://git.veripool.org/git/verilator`
- `cd verilator`
- `git pull`
- `git checkout v4.106`
- `autoconf`
- `./configure`
- `make` (alternatively you can use `make -j$(nproc)` to make it go faster)
- `sudo make install`

- `export PATH=$PATH:/usr/local/bin` (change the path in your system)

To add */usr/local/bin* permanently to your path, add the last line to your ~/.bashrc file.

6. RUNNING AND PROGRAMMING RVfpgaNexys

In this section, we show how to run seven simple programs on RVfpgaNexys (see Figure 25).

LINUX / Windows / macOS: All the instructions described in this section should work for the three operating systems, assuming that all the required tools and drivers were installed correctly as explained in Section 5. In some cases, you may need to modify some minor details, such as the slash, used in Linux, for a backslash, used in Windows.

We demonstrate how to use RVfpgaNexys by showing how to run the seven example programs listed in Table 9. The first three programs are written in RISC-V assembly language and the last four programs are written in C. Directions for running each of the programs on RVfpgaNexys are described below.

Table 9. RVfpgaNexys Example Programs

Program Name	Description	Language
AL_Operations	exercises arithmetic and logical operations	RISC-V assembly
Blinky	blinks an LED on the Nexys A7 board	RISC-V assembly
LedsSwitches	reads switch values on Nexys A7 board and writes that value to the LEDs	RISC-V assembly
LedsSwitches_C-Lang	reads switch values on Nexys A7 board and writes that value to the LEDs	C
HelloWorld_C-Lang	prints a short message to a shell through the serial port	C
VectorSorting_C-Lang	sorts a vector from largest to smallest	C
DotProduct_C-Lang	computes the dot product of two vectors	C

Note that, before being able to execute any of these seven examples, **you must program the FPGA with RVfpgaNexys**, as explained in the following section.

A. Program the FPGA with RVfpgaNexys

In this section, we explain the recommended method for programming the FPGA with RVfpgaNexys, which uses PlatformIO. Follow the next steps for programming the FPGA with RVfpgaNexys:

(If you are interested in using Vivado for programming the FPGA, you can follow the instructions provided at Appendix E of this guide instead of the following instructions below. However, the method described there is only possible for Linux and Windows systems (*not* macOS) – and, overall, the method of using Vivado to download RVfpgaNexys onto the FPGA is not recommended. Instead, it is recommended that you follow the instructions below and ignore Appendix E.)

a. Connect the Nexys A7 board to your computer.
b. Turn on the Nexys A7 board using the switch at the top left.
c. Open VSCode and PlatformIO if it is not already open.
d. On the top menu bar, click on *File → Open Folder* (see Figure 33) and browse into directory *[RVfpgaPath]/RVfpga/examples/*

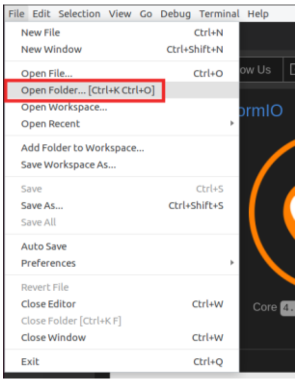


Figure 33. Open Folder

e. Select the PlatformIO project that you are going to use. In this section, as an example, we use AL_Operations, the first example mentioned in Table 9, that you will debug in the next section, but you could follow the same steps with any other example. Thus, select directory *AL_Operations* (do not open it, but just select it – see Figure 34) and click OK at the top of the window. PlatformIO will now open the example.

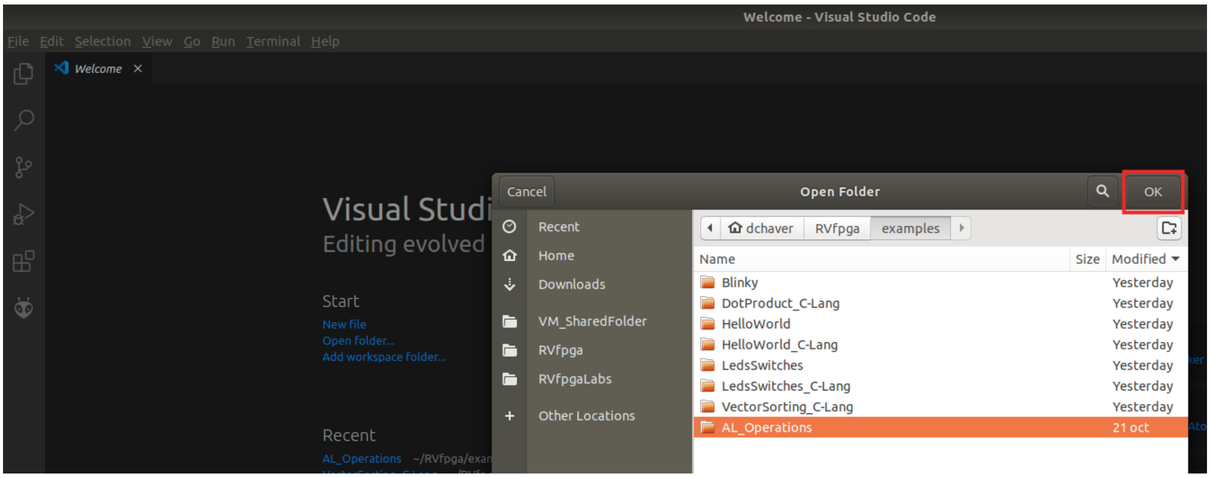


Figure 34. Open AL_Operations folder

f. Open file *platformio.ini*, by clicking on *platformio.ini* in the left sidebar (see Figure 35). Establish the path to the RVfpgaNexys bitstream in your system by editing the following line (see Figure 35). Note that a pre-synthesized bitstream of RVfpgaNexys is provided in the RVfpga folder at: *[RVfpgaPath]/RVfpga/src/rvfpganexys.bit*.

```
board_build.bitstream_file = [RVfpgaPath]/RVfpga/src/rvfpganexys.bit
```

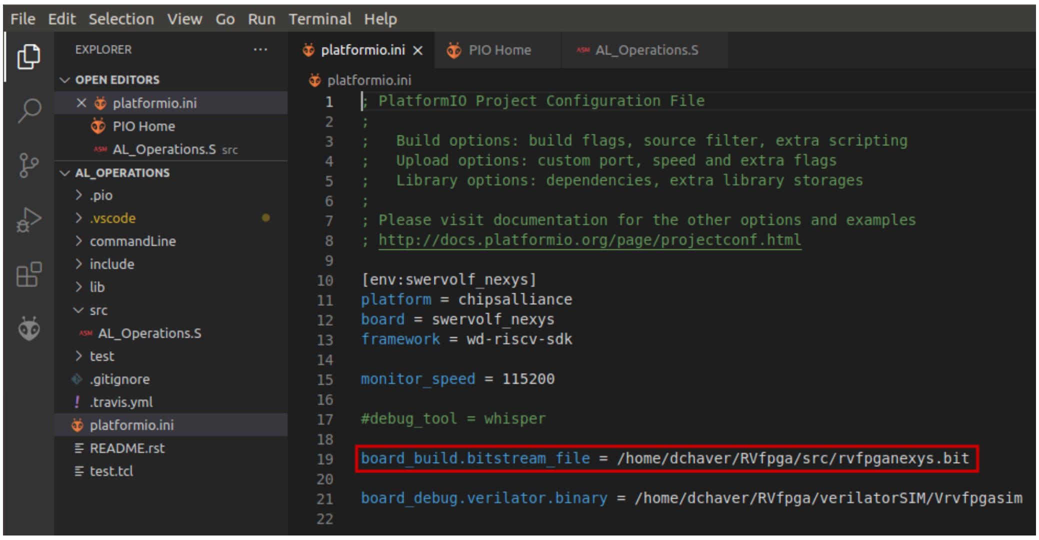


Figure 35. Platformio initialization file: platformio.ini

There are many different commands that you can use in the Project Configuration File (*platformio.ini*), and for which you can find information at: https://docs.platformio.org/en/latest/projectconf/.

g. Click on the PlatformIO icon  in the left menu ribbon (see Figure 36).

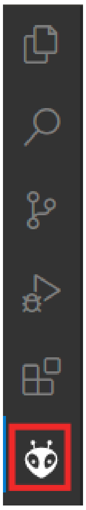


Figure 36. PlatformIO icon

In case the Project Tasks window is empty (Figure 37), you must refresh the Project Tasks first by clicking on . This can take several minutes.

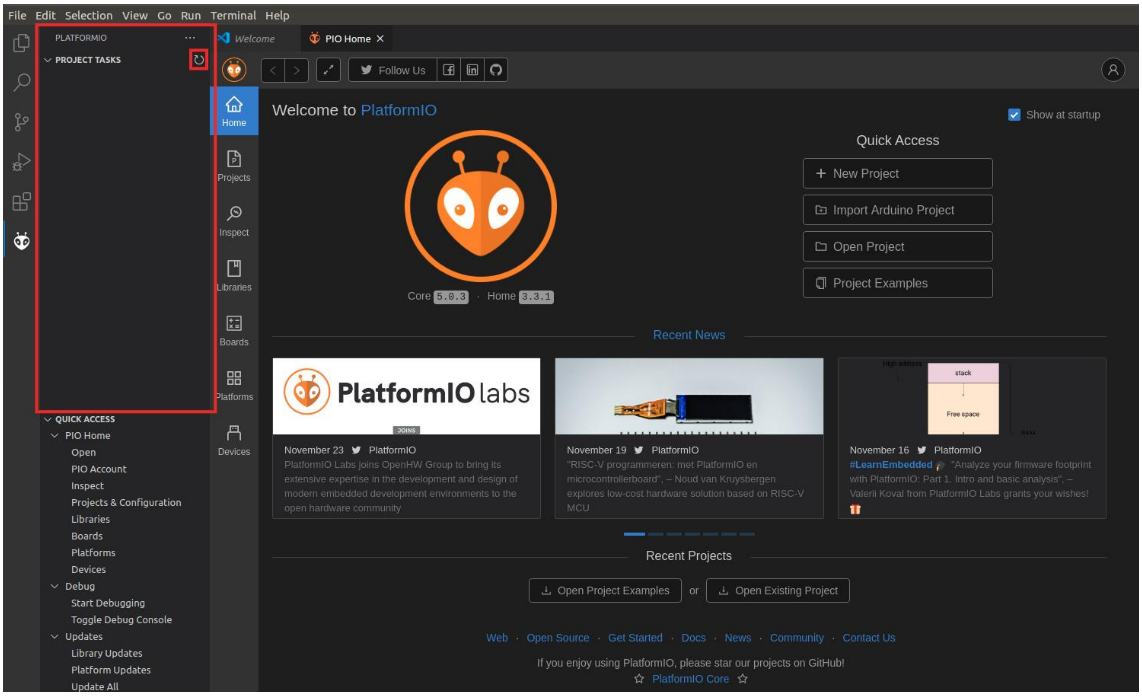


Figure 37. PROJECT TASKS window empty – Refresh

Then expand Project Tasks → env:swervolf_nexys → Platform and click on Upload Bitstream, as shown in Figure 38. **After one or two seconds, the FPGA will be programmed with RVfpgaNexys**.

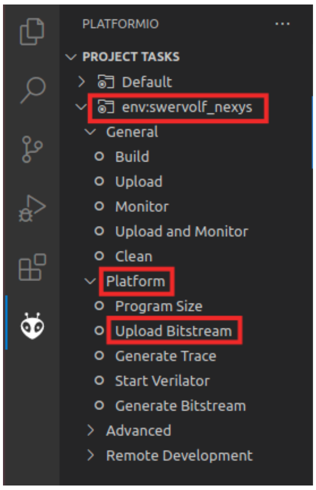


Figure 38. Upload Bitstream

By default, the processor starts fetching instructions at address 0x80000000, where the Boot ROM is placed in our SoC (see Table 6). The Boot ROM is initialized with a program (*boot_main.mem*) that blinks the LEDs and the 7-Segment Displays four times and then turns off all the LEDs, writes 0s to the 8 7-Segment Displays and stays in an empty loop. You can find this program in folder: *[RVfpgaPath]/RVfpga/src/SweRVolfSoC/BootROM/sw*.

Pressing the CPU Reset button on the Nexys A7 board (Figure 26) makes this program to execute again.

If you want to change and recompile this program, do it as explained in Appendix A –

Section III (note that file *boot_main.mem* is simply a copy of file *boot_main.vh*). In Lab 5, we will show how the Boot ROM is initialized with this program when creating the bitstream.

h. As an alternative to the previous step (step g), you can download RVfpgaNexys from a PlatformIO terminal window as shown in Figure 39. Click on the button (PlatformIO: New Terminal button) at the bottom of the PlatformIO window for opening a new terminal window, and then type (or copy) the following command into the PlatformIO terminal:

```
pio run -t program_fpga
```

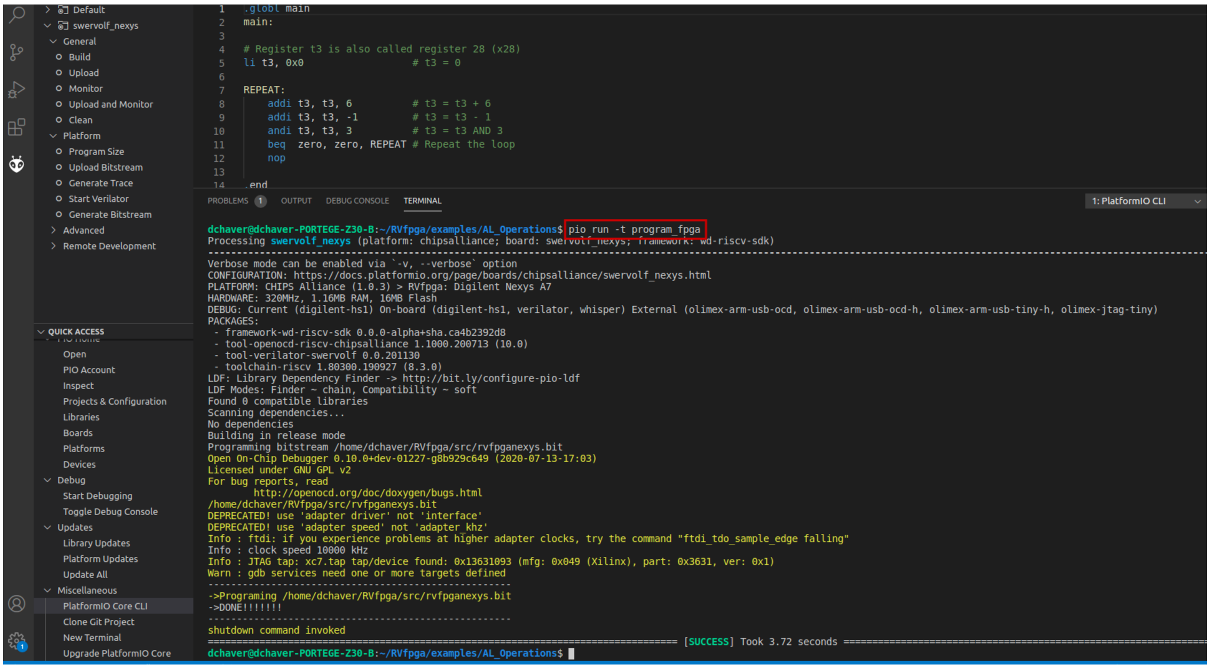


Figure 39. Upload RVfpgaNexys onto Nexys A7 FPGA Board using PlatformIO

Note that the first time that an example is opened in PlatformIO, the Chips Alliance platform gets automatically installed (you can view it inside the PIO Home, as shown in Figure 40). This platform includes several tools that you will use later, such as the pre-built RISC-V toolchain, OpenOCD for RISC-V, an RVfpgaNexys bitfile and RVfpgaSim, JavaScript and Python scripts, and several examples.

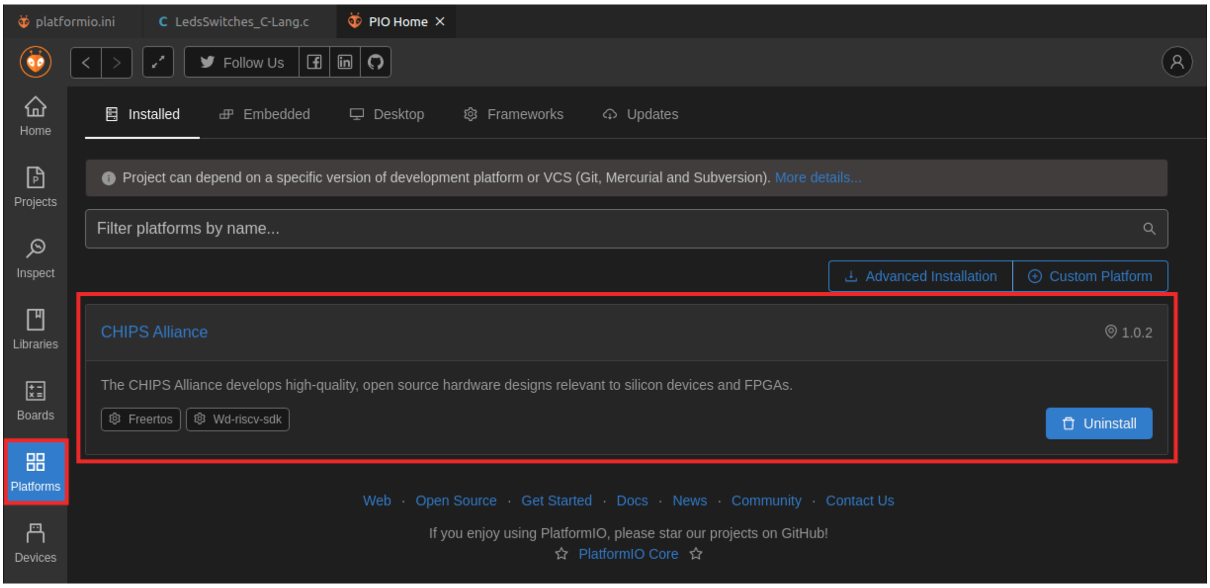


Figure 40. Chips Alliance platform installed in PlatformIO

If, for any reason, the Chips Alliance platform did not install automatically, you can install it manually following the next steps (normally, you can simply skip this procedure and continue with Section B):

- View the Quick Access menu by clicking on the  button, located in the left side bar (see Figure 41). Then, in the PIO Home, click on the Platforms button and then on the Embedded tab (Figure 41). Look for **Chipsalliance** (the platform that we use in RVfpga) and open it by clicking on the CHIPS Alliance button (Figure 41).

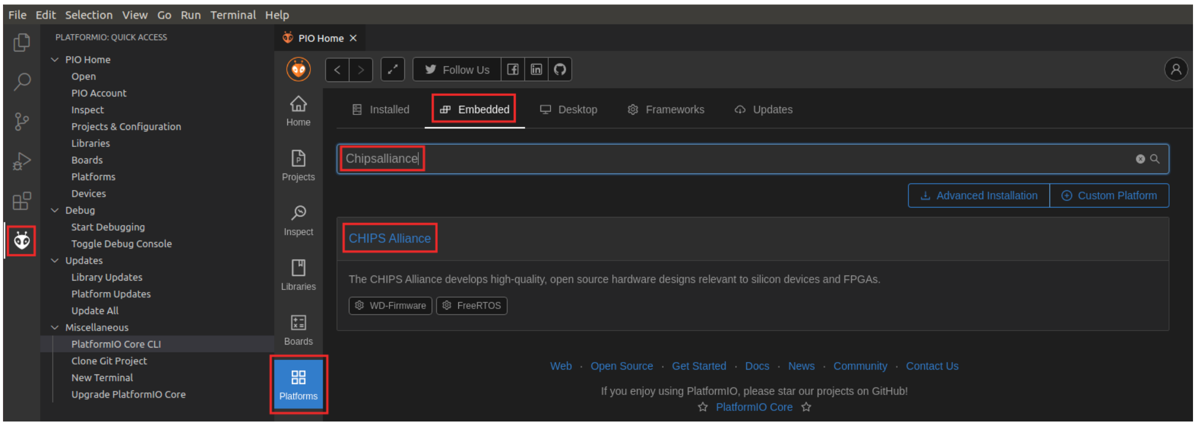


Figure 41. Selecting the CHIPS Alliance Platform

- After clicking on the CHIPS Alliance button, you will see the details of the Chips Alliance platform (as in Figure 42). Install it by clicking on the Install button (Figure 42).

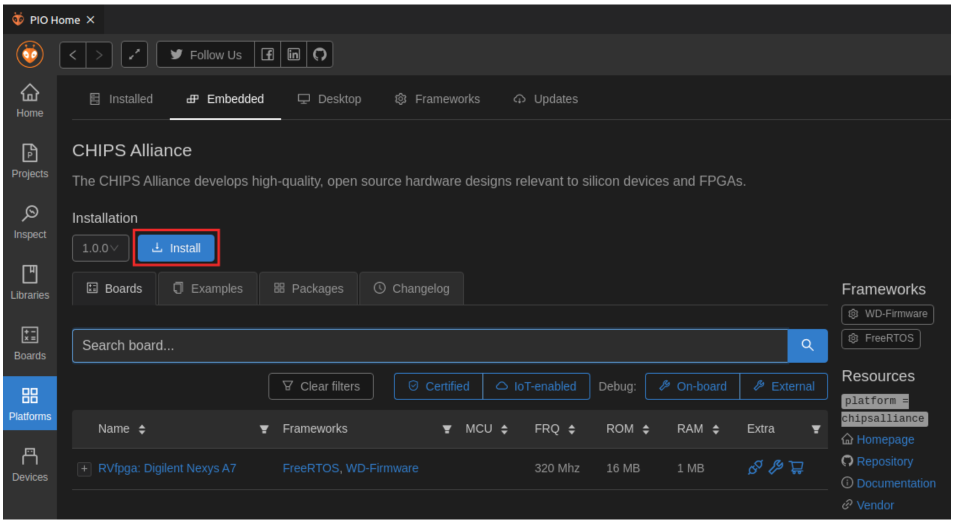
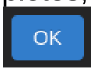


Figure 42. Installing the CHIPS Alliance Platform

- Once installation completes, a summary of the tools that have been installed is shown, as in Figure 43. Click 

to close that window.

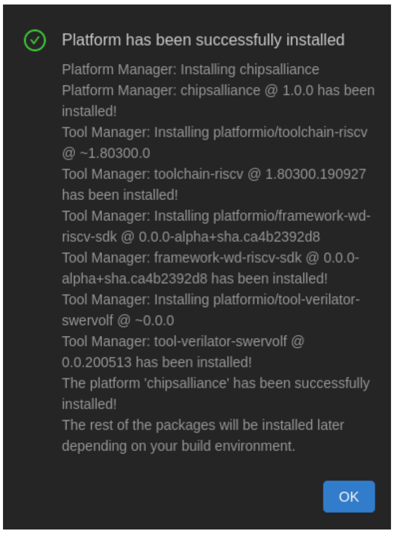


Figure 43. Successful installation of CHIPS Alliance Platform

B. AL_Operations program

The first example program, AL_Operations.s (see Figure 44), is an assembly program that performs three arithmetic-logic instructions (addition, subtraction, and logical and) on the same register, `t3` (also called `x28`), within an infinite loop.

```
.globl main
main:

# Register t3 is also called register 28 (x28)
li t3, 0x0                  # t3 = 0

REPEAT:
    addi t3, t3, 6          # t3 = t3 + 6
    addi t3, t3, -1         # t3 = t3 - 1
    andi t3, t3, 3          # t3 = t3 AND 3
    beq  zero, zero, REPEAT # Repeat the loop
```

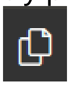
```
    nop

.end
```

Figure 44. AL_Operations program: AL_Operations.S

Follow these steps to run and debug this code on the Nexys A7 FPGA board using PlatformIO:

1. Program the FPGA as explained in the previous section. Note that you already have the *AL_Operations* project opened in PlatformIO.

2. Open the assembly program, AL_Operations.S, by clicking on the Explorer icon in the left menu ribbon , expanding *src* under *AL_OPERATIONS* in the left sidebar and clicking on AL_Operations.S (see Figure 45).

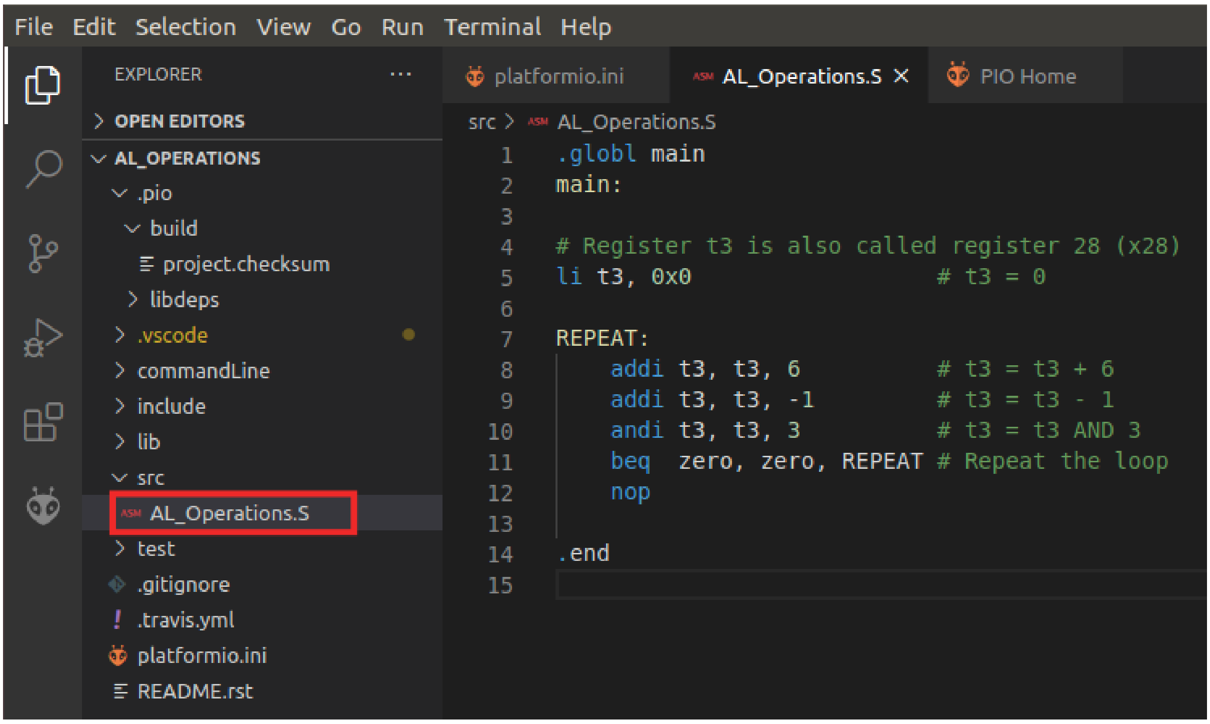
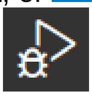
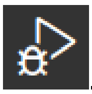


Figure 45. View assembly file AL_Operations.S

3. VSCode and PlatformIO provide different ways of compiling, cleaning and debugging the program. In the bottom part of VSCode, you can find some buttons that provide useful functionalities: . For example,  can be used to build the project, or  can be used to clean it. In the left side bar (see Figure 30), the "Run" button  can be used to compile the program and then open the debugger.

4. Click on the "Run" button . Start the debugger by clicking on the play button 

(make sure that the "PIO Debug" option is selected). You can find this button near the top of the window (see Figure 46). The program will first compile and then debugging will start. PlatformIO sets a temporary breakpoint at the beginning of the main function, so the execution will stop there.

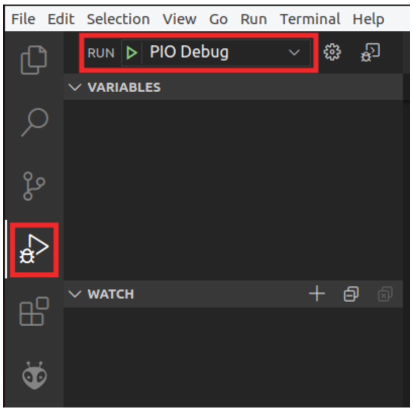
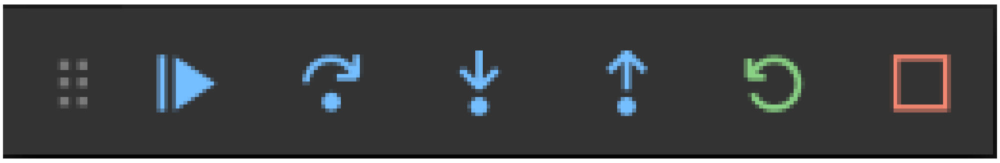


Figure 46. Start debugger

5. To control your debugging session, you can use the debugging toolbar that appears near the top of the editor (see Figure 47). Below are the options:
   - **Continue** executes the program until the next breakpoint.
   - **Breakpoints** can be added by clicking to the left of the line number in the editor.
   - **Step Over** executes the current line and then stop.
   - **Step Into** executes the current line and if the current line includes a function call, it will jump into that function and stop.
   - **Step Out** executes all of the code in the function you are in and then stops once that function returns.
   - **Restart** restarts the debugging session from the beginning of the program.
   - **Stop** stops the debugging session and returns to normal editing mode.
   - **Pause** pauses execution. When the program is running, the Continue button is replaced by the Pause button.

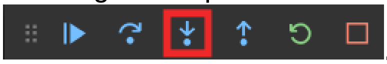


CONTINUE STEP-OVER STEP-INTO STEP-OUT RESTART STOP

Figure 47. Debugging tools

6. On the left sidebar, you can view the Debugger options. The following options are available:
   - **Variables:** lists local, global, and static variables present in your program along with their values.
   - **Call Stack:** shows you the current function being run, the calling function (if any), and the location of the current instruction in memory.
   - **Breakpoints:** show any set breakpoints and highlight their line number. Breakpoints can be managed in this section. Breakpoints can also be temporarily deactivated without removing them by toggling the checkbox.

- **Peripherals**: shows the status of the registers of the memory-mapped peripherals of the device (we will cover these in more detail in the RVfpga Labs).
- **Registers:** lists the current values present in each of the registers of the processor.
- **Memory:** displays the contents of a specific address of memory.
- **Disassembly:** shows the assembly code for a specific function – for higher-level code such as C, this allows you to view the assembly for debugging the instructions one-by-one.

7. Expand the Registers option in the Debugger Side Bar and continue the execution step-by-step . You will observe that register `x28` (also called `t3`, as shown in the REGISTERS section) stores the results of the three arithmetic-logic operations: *addition*, *subtraction,* and *logical AND*. See Figure 48.

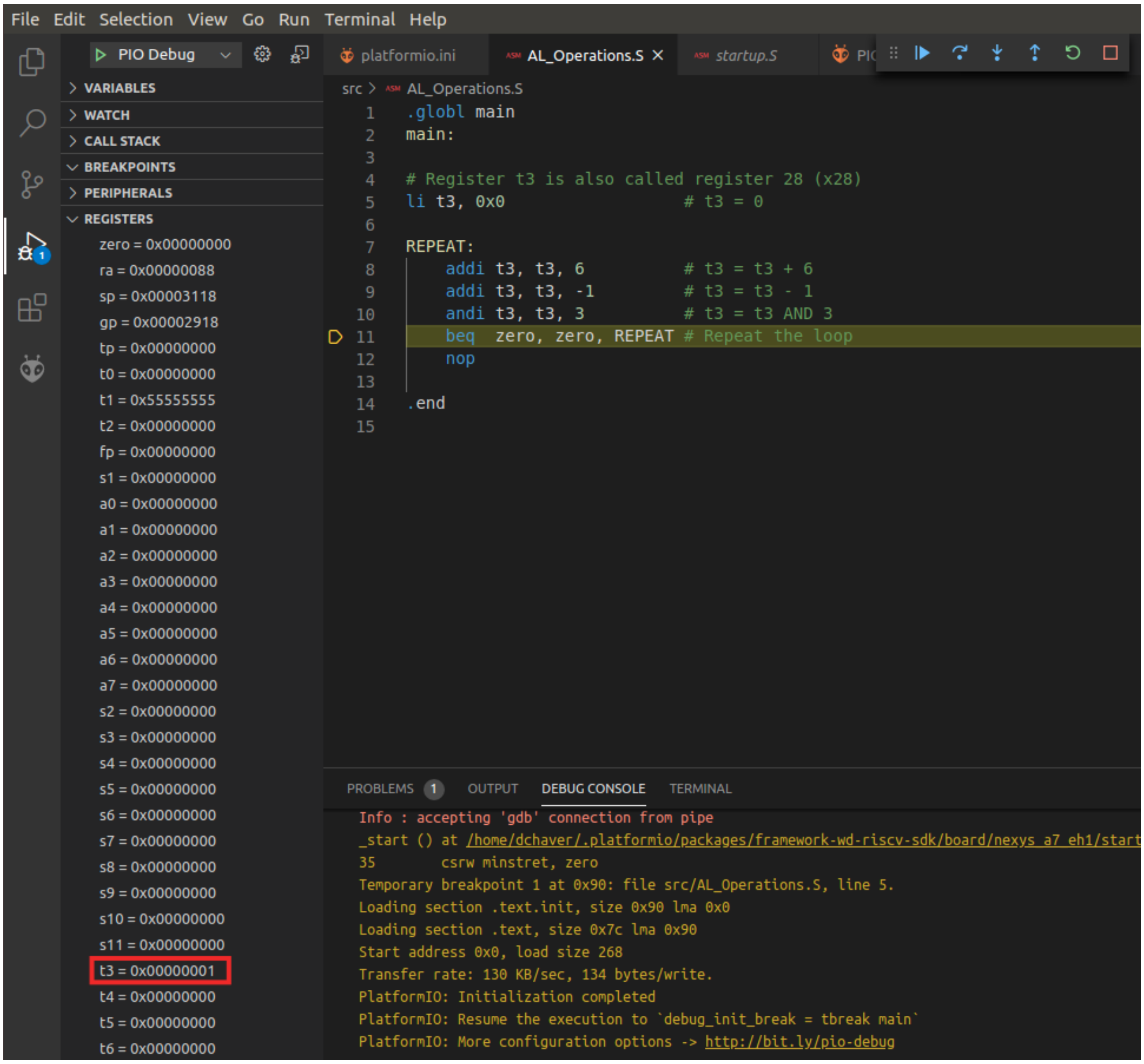


Figure 48. Viewing register contents

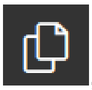
8. Before calling the *main* function, a start-up file, provided by Western Digital at *~/.platformio/packages/framework-wd-riscv-sdk/board/nexys_a7_eh1/startup.S*, is executed. This file configures the core: Instruction Cache set-up, registers initialization

(such as *sp* or *gp*), etc. When debugging is launched, this file opens in the main window (see Figure 48), and you can inspect it there.

Windows: The *.platformio* folder is located inside your user folder (C:\Users\<USER>). Note that you may need to enable the system for viewing hidden files/folders.

macOS: Like in Linux, the *.platformio* folder is located inside your home folder (*~/.platformio*).

9. We should also highlight that, in the same directory (*~/.platformio/packages/framework-wd-riscv-sdk/board/nexys_a7_eh1/*), file *link.lds* is provided, which constitutes the linker script that we will use in all our projects. This file determines the placement of the assembly sections (*text*, *data*, *bss*…) in memory.

10. Finally, stop debugging  (which will make the Boot ROM program to execute again) and go back to the Explorer window by clicking on , which you can find in the top of the left-most side bar. On the top menu bar, click on *File → Close Folder*.

C. Blinky program

The second example program, *blinky.S*, is an assembly program that makes the Nexys A7 board's right-most LED blink (see Figure 49). The program repeatedly inverts the value connected to the right-most LED with a delay between each inversion.

```
#define GPIO_LEDs   0x80001404
#define GPIO_INOUT  0x80001408

#define DELAY 0x100000                /* Define the DELAY */

.globl main
main:

       li x28, 0xFFFF
       li a0, GPIO_INOUT
       sw x28, 0(a0)                # Write the Enable Register

       li   t1, DELAY               # Set timer value to control blink speed

       li   t0, 0

bl1:
       li   a0, GPIO_LEDs
       sb   t0, 0(a0)               # Write to LEDs
       xori t0, t0, 1               # invert LED
       and  t2, zero, zero          # Reset timer

time1:                              # Delay loop
       addi t2, t2, 1
       bne  t1, t2, time1
       j    bl1
```

Figure 49. blinky.S

Follow the next steps to run and debug this code on RVfpgaNexys, the RISC-V SoC loaded onto the FPGA board:

1. RVfpgaNexys is already programmed on the FPGA board if you executed the first example (*AL_Operations*), so you should not need to program it again. However, if you do need to reprogram RVfpgaNexys onto the board again, do it as explained in Section A, using the Blinky example instead of the AL_Operations example.

2. On the top bar, click on *File → Open Folder*, and browse into directory *[RVfpgaPath]/RVfpga/examples/*

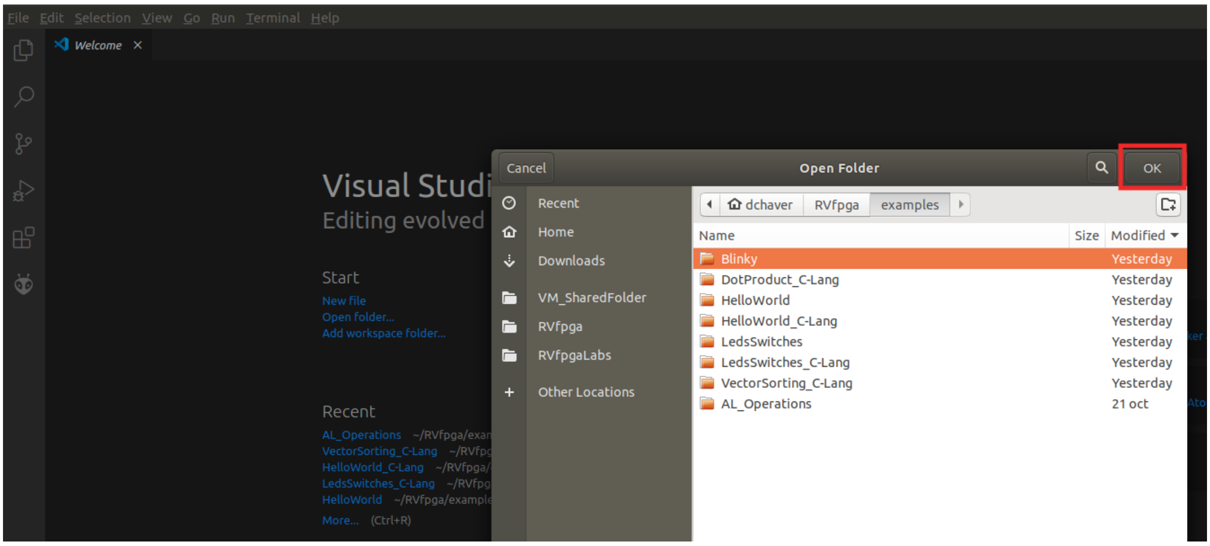


Figure 50. Blinky program folder

3. Select directory *Blinky* and click OK (Figure 50).

4. Open the assembly code of the example, file *blinky.S,* in the editor, by clicking on it (Figure 51).

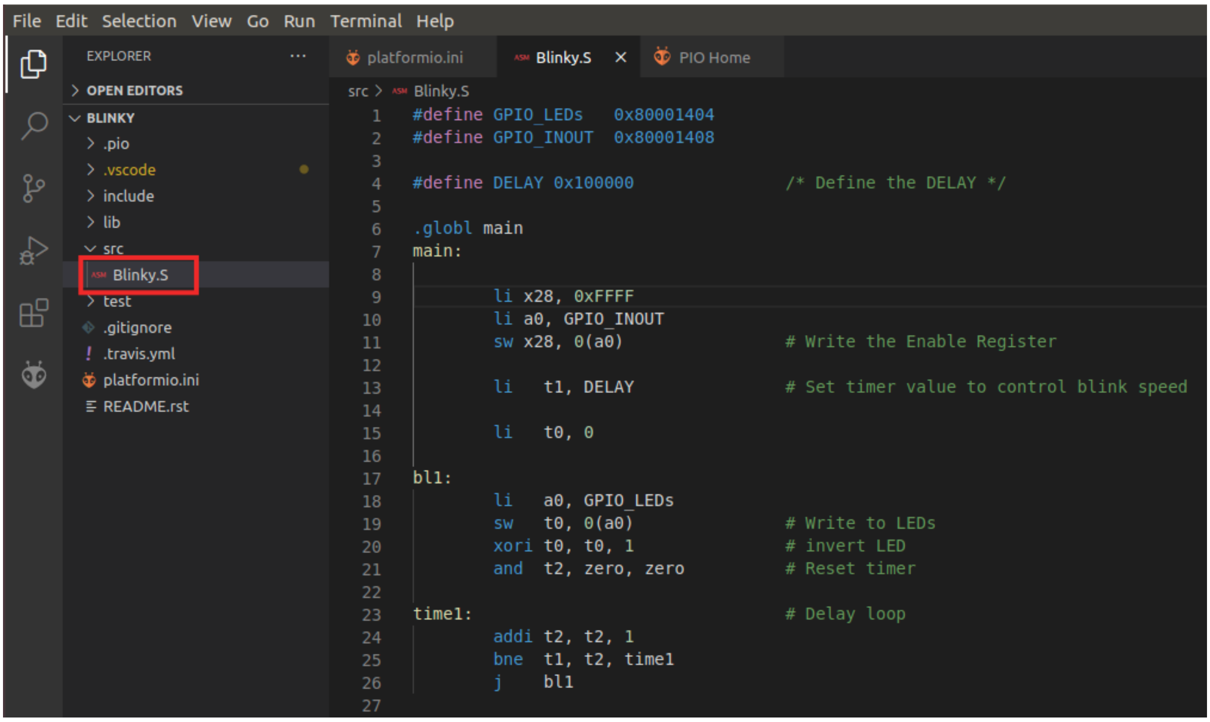
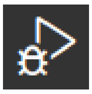
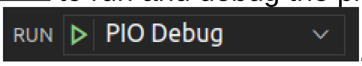
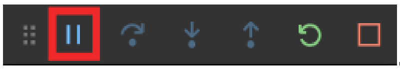


Figure 51. blinky.S in PlatformIO

5. Click on  to run and debug the program; then start debugging by clicking on the play button 

. PlatformIO sets a temporary breakpoint at the beginning of the main function. So, click on the Continue button  to run the program.

6. On the board, you will see the right-most LED start to blink.

7. Pause the execution by clicking on the pause button  . The execution will stop somewhere inside the infinite loop (probably, inside the `time1` delay loop).

8. Establish a breakpoint by clicking to the left of line number 18. A red dot will appear and the breakpoint will be added to the BREAKPOINTS tab (see Figure 52).

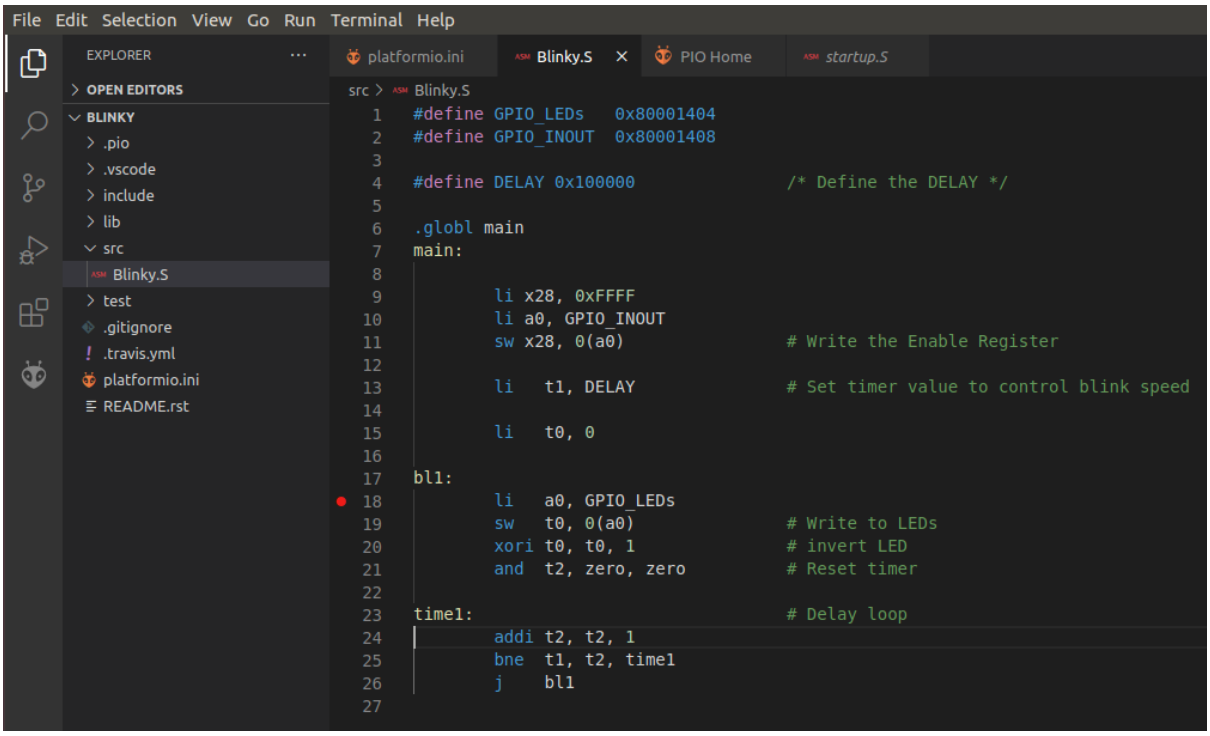
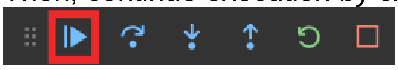
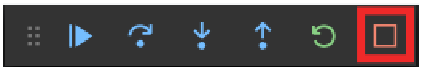
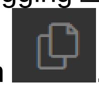


Figure 52. Setting a breakpoint in blinky.S

9. Then, continue execution by clicking on the Continue button . Execution will continue and it will stop after the store word (`sw`) instruction, which writes 1 (or 0) to the right-most LED.

10. Continue execution several times; you will see that the value driven to the right-most LED changes each time.

11. Stop debugging  and go back to the Explorer window by clicking on . Close the program by selecting *File → Close Folder.*

D. LedsSwitches program

The third assembly example communicates with the LEDs and the switches available on the board (see Figure 53).

```
#define GPIO_SWs    0x80001400
#define GPIO_LEDs   0x80001404
#define GPIO_INOUT  0x80001408

.globl main
main:

li x28, 0xFFFF
li x29, GPIO_INOUT
sw x28, 0(x29)                  # Write the Enable Register

next:
   li  a1, GPIO_SWs            # Read the Switches
   lw  t0, 0(a1)

```

```
    li  a0, GPIO_LEDs
    srl t0, t0, 16
    sw  t0, 0(a0)              # Write the LEDs

    beq zero, zero, next
.end
```

Figure 53. LedsSwitches.S

Follow the next steps for running and debugging this code on the FPGA board:

1. RVfpgaNexys is already programmed on the FPGA board if you executed the previous examples, so you should not need to program it again. However, if you do need to reprogram RVfpgaNexys onto the board again, do it as explained in Section A, using the LedsSwitches example instead of the AL_Operations example.

2. On the top bar, click on *File → Open Folder*, and browse to directory *[RVfpgaPath]/RVfpga/examples/.* Select directory *LedsSwitches* and click OK.

3. The program *LedsSwitches.S* has an infinite loop where the switches are read and then their state is shown on the LEDs (Figure 54).

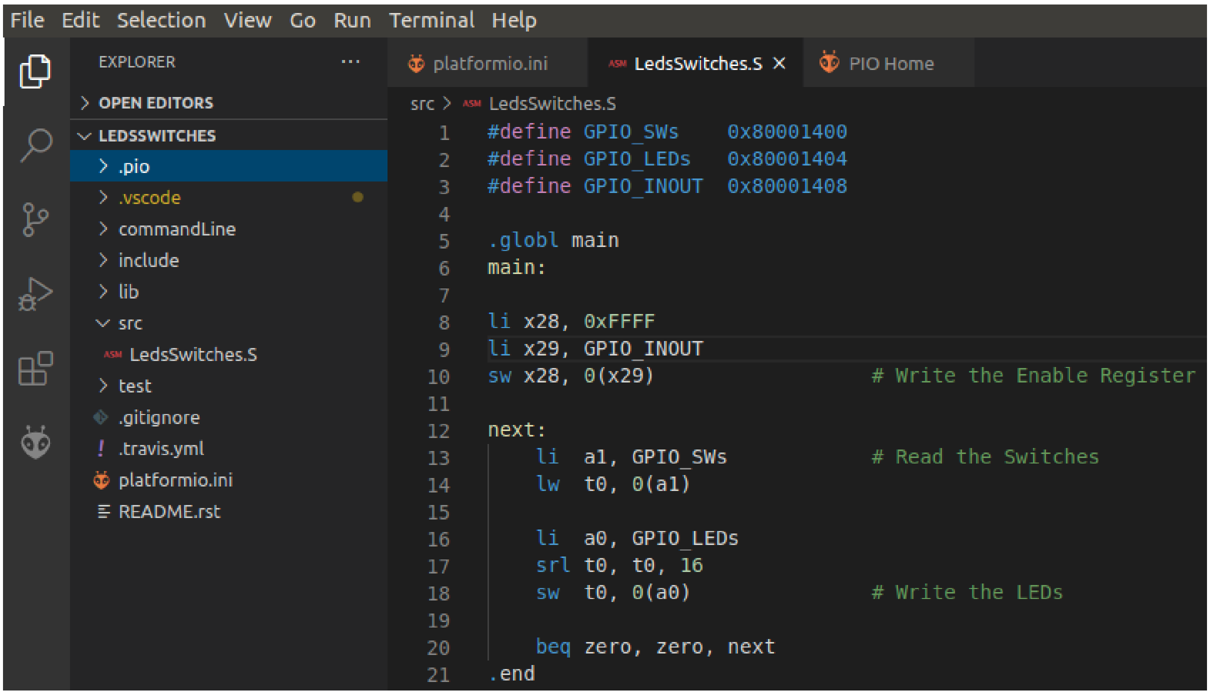


Figure 54. LedsSwitches.S in PlatformIO

4. After launching the debugger as explained for prior programs, the program starts to run. PlatformIO sets a temporary breakpoint at the beginning of the main function. So, click on the Continue button to run the program.

5. Toggle the switches on the bottom of the Nexys A7 board. You will immediately see on the board that the LEDs show the new value of the switches. You can pause the execution, run step-by-step and inspect the registers as explained above. When you are finished, close the project by clicking on *File → Close Folder.*

6. Sometimes, it can be very useful to inspect the values stored in memory. For that purpose, PlatformIO provides a Memory Display.

a. Pause the execution and step until the beginning of the *next* loop. Expand the Memory Display on the left part of the window (see Figure 55) and click on *Enter address…*

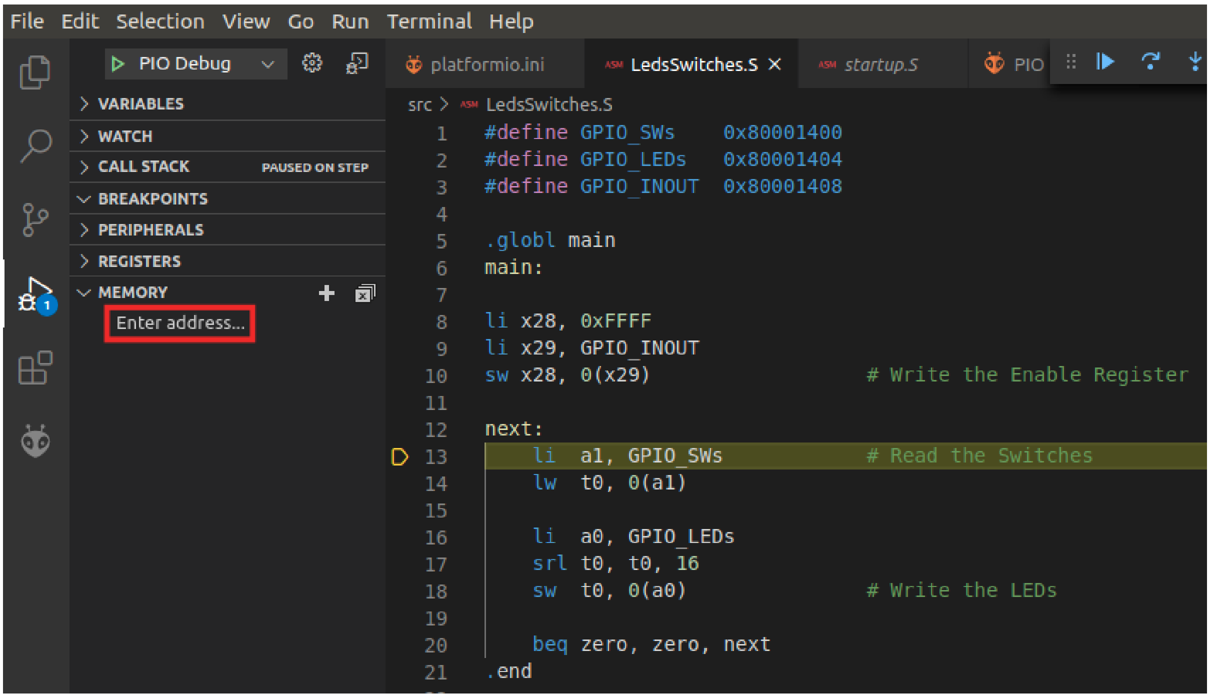


Figure 55. Memory Display

b. The initial memory address will be requested (see Figure 56). Insert the initial address where the Switches are mapped, in our case 0x80001400.

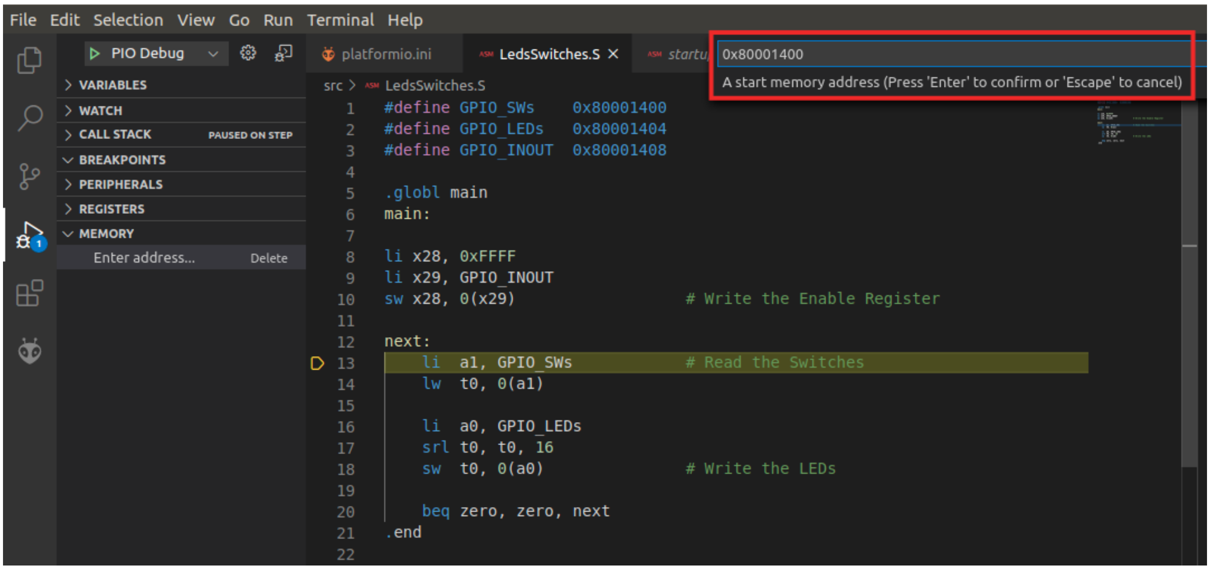


Figure 56. Initial memory address to show

c. Then, the number of bytes that you want to inspect is requested (see Figure 57), so insert a value of 0xc (we want to inspect three 4-byte I/O registers, thus we need 12 bytes).

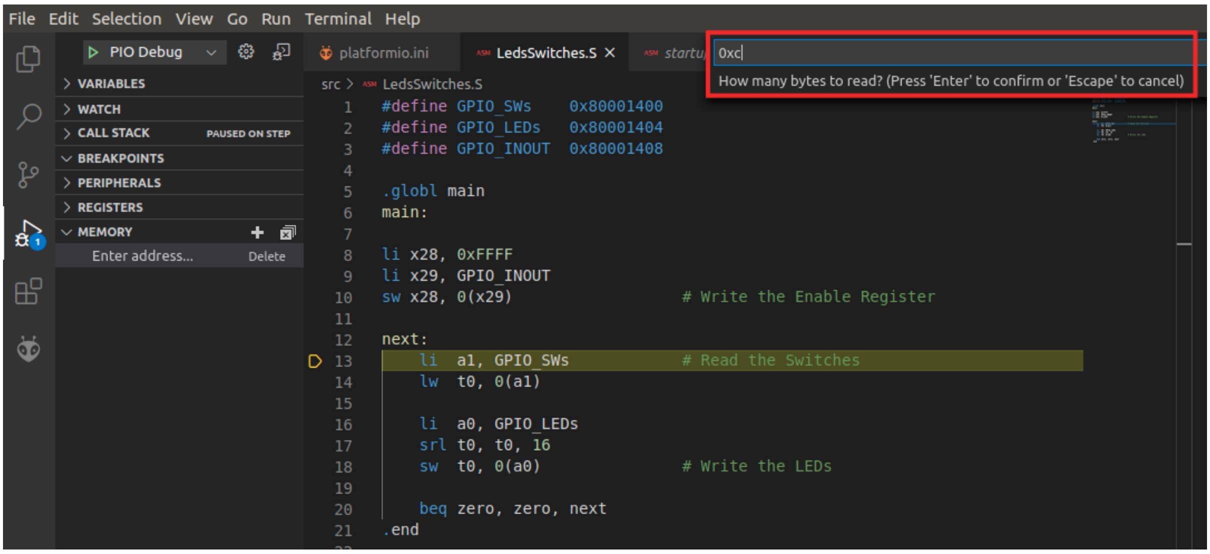


Figure 57. Number of bytes to show

d. The Memory Display will open to the right, showing the 12 bytes that we have requested (see Figure 58). The value that we have in the 16 switches is 0x123C (see bytes at addresses 0x80001402 and 0x80001403). Taking into account that RISC-V architecture is little endian, the value shown in the figure is coherent with that. The 16 LEDs (stored at addresses 0x80001404 and 0x80001405) show the same value.

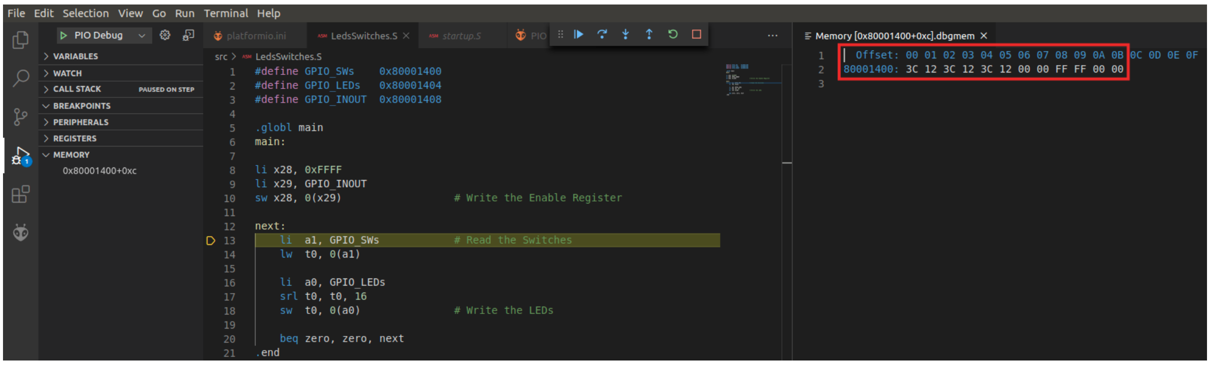


Figure 58. Memory addresses 0x80001400-0x8000140B

e. Change the values of the switches on the board, for example to 0x5555, and execute one more iteration of the loop step-by-step. The value of the switches in memory should change immediately after executing the first instruction (Figure 59, top), and the value of the LEDs should change accordingly after executing the `sw` instruction (Figure 59, bottom).

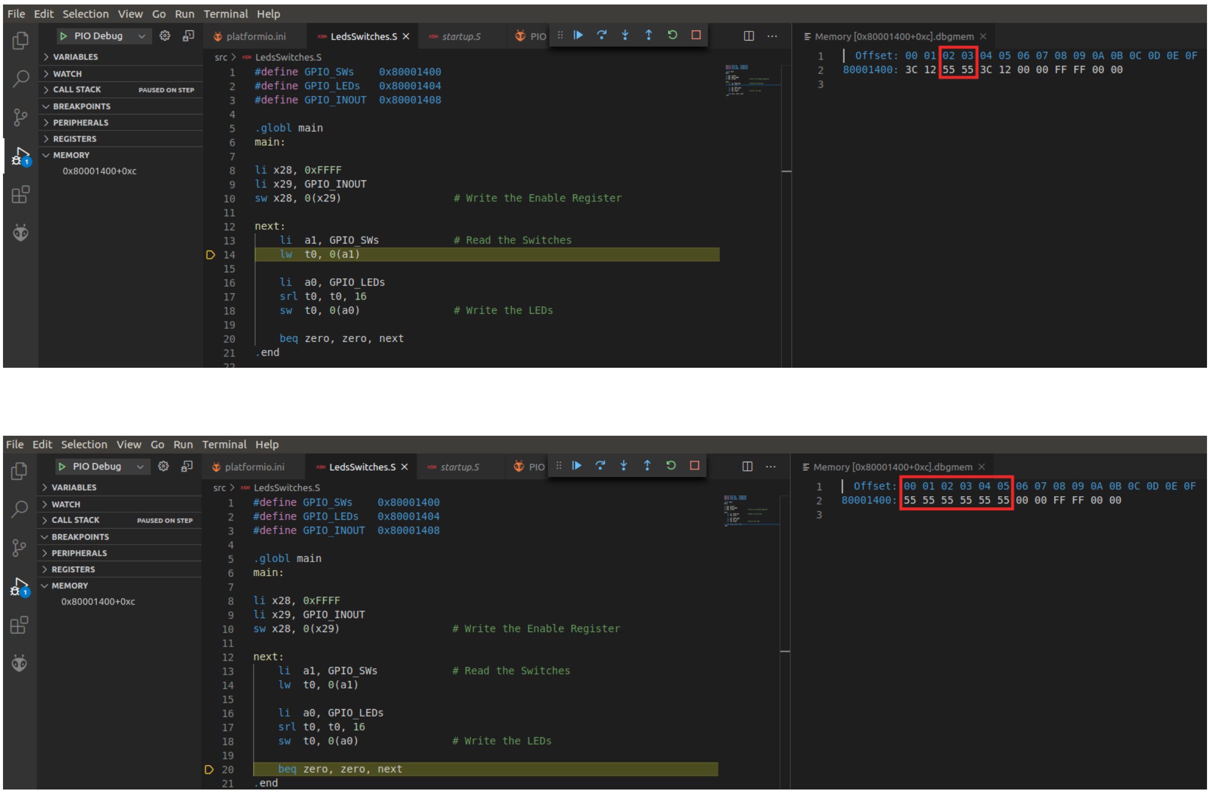


Figure 59. Change of the Switches and LEDs

f. You can also view other memory locations, such as the RAM addresses that store the machine instructions of your program. Open another memory range starting at 0x0 (initial address assigned to the RAM memory) and occupying 0x100 bytes (Figure 60). You will see the instructions from the LedsSwitches program stored in the address range 0x90-0xC4, right after the startup program (Startup.S).

```
≡ Memory [0x0+0x100].dbgmem ×
 Offset: 00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F
00000000: 73 10 20 B0 73 10 20 B8 81 40 01 41 81 41 01 42
00000010: 81 42 01 43 81 43 01 44 81 44 01 45 81 45 01 46
00000020: 81 46 01 47 81 47 01 48 81 48 01 49 81 49 01 4A
00000030: 81 4A 01 4B 81 4B 01 4C 81 4C 01 4D 81 4D 01 4E
00000040: 81 4E 01 4F 81 4F 37 53 55 55 13 03 53 55 73 10
00000050: 03 7C 97 31 00 00 93 81 E1 8E 17 31 00 00 13 01
00000060: 61 0E 17 25 00 00 13 05 E5 0D 97 25 00 00 93 85
00000070: 65 0D 63 77 B5 00 23 20 05 00 11 05 E3 6D B5 FE
00000080: 91 20 01 45 81 45 29 20 01 A0 00 00 00 00 00 00
00000090: 37 0E 01 00 13 0E FE FF B7 1E 00 80 93 8E 8E 40
000000a0: 23 A0 CE 01 B7 15 00 80 93 85 05 40 83 A2 05 00
000000b0: 37 15 00 80 13 05 45 40 93 D2 02 01 23 20 55 00
000000c0: E3 02 00 FE 41 11 22 C4 4A C0 17 04 00 00 13 04
000000d0: 64 F3 17 09 00 00 13 09 E9 F2 33 09 89 40 06 C6
000000e0: 26 C2 13 59 29 40 63 09 09 00 81 44 1C 40 85 04
000000f0: 11 04 82 97 E3 1C 99 FE 17 04 00 00 13 04 84 F0
```

Figure 60. Memory addresses 0x0 to 0x100

g. You can view the machine code for the program's instructions by opening the disassembly of the program available at: *[RVfpgaPath]/RVfpga/examples/LedsSwitches/.pio/build/swervolf_nexys/firmware.dis* (see Figure 61). Compare the two figures and try to identify the instructions of the program.

```
Disassembly of section .text:

00000090 <main>:
  90: 00010e37          lui t3,0x10
  94: fffe0e13          addi  t3,t3,-1 # ffff <_sp+0xcebf>
  98: 80001eb7          lui t4,0x80001
  9c: 408e8e93          addi  t4,t4,1032 # 80001408 <OVERLAY_END_OF_OVERLAYS+0xa0001408>
  a0: 01cea023          sw  t3,0(t4)

000000a4 <next>:
  a4: 800015b7          lui a1,0x80001
  a8: 40058593          addi  a1,a1,1024 # 80001400 <OVERLAY_END_OF_OVERLAYS+0xa0001400>
  ac: 0005a283          lw  t0,0(a1)
  b0: 80001537          lui a0,0x80001
  b4: 40450513          addi  a0,a0,1028 # 80001404 <OVERLAY_END_OF_OVERLAYS+0xa0001404>
  b8: 0102d293          srli  t0,t0,0x10
  bc: 00552023          sw  t0,0(a0)
  c0: fe0002e3          beqz  zero,a4 <next>
```

Figure 61. Disassembly version of the LedsSwitches program

E. LedsSwitches_C-Lang program

Program LedsSwitches_C-Lang.c (Figure 62) does the same as the LedsSwitches.s program shown previously (Figure 53) but it is written in C instead of assembly.

```
1  #define GPIO_SWs    0x80001400
2  #define GPIO_LEDs   0x80001404
3  #define GPIO_INOUT  0x80001408
4
5  #define READ_GPIO(dir) (*(volatile unsigned *)dir)
6  #define WRITE_GPIO(dir, value) { (*(volatile unsigned *)dir) = (value); }
7
8  int main ( void )
9  {
10    int En_Value=0xFFFF, switches_value;
11
12    WRITE_GPIO(GPIO_INOUT, En_Value);
13
14    while (1) {
15        switches_value = READ_GPIO(GPIO_SWs);
16        switches_value = switches_value >> 16;
17        WRITE_GPIO(GPIO_LEDs, switches_value);
18    }
19
20    return(0);
21 }
```

Figure 62. LedsSwitches_C-Lang.c

Follow the next steps for running and debugging this program on the FPGA board:

1. RVfpgaNexys is already programmed on the FPGA board if you executed the previous examples, so you should not need to program it again. However, if you do need to reprogram RVfpgaNexys onto the board again, do it as explained in Section A, using the LedsSwitches_C-Lang example instead of the AL_Operations example.

2. On the top menu bar, click on *File → Open Folder*, and browse into directory *[RVfpgaPath]/RVfpga/examples/.* Select directory *LedsSwitches_C-Lang* and click OK.

3. Before calling the debugger, set a breakpoint at line 15 in the C Code.

4. Then, start debugging. The program will start executing and will stop at the breakpoint (Figure 63).

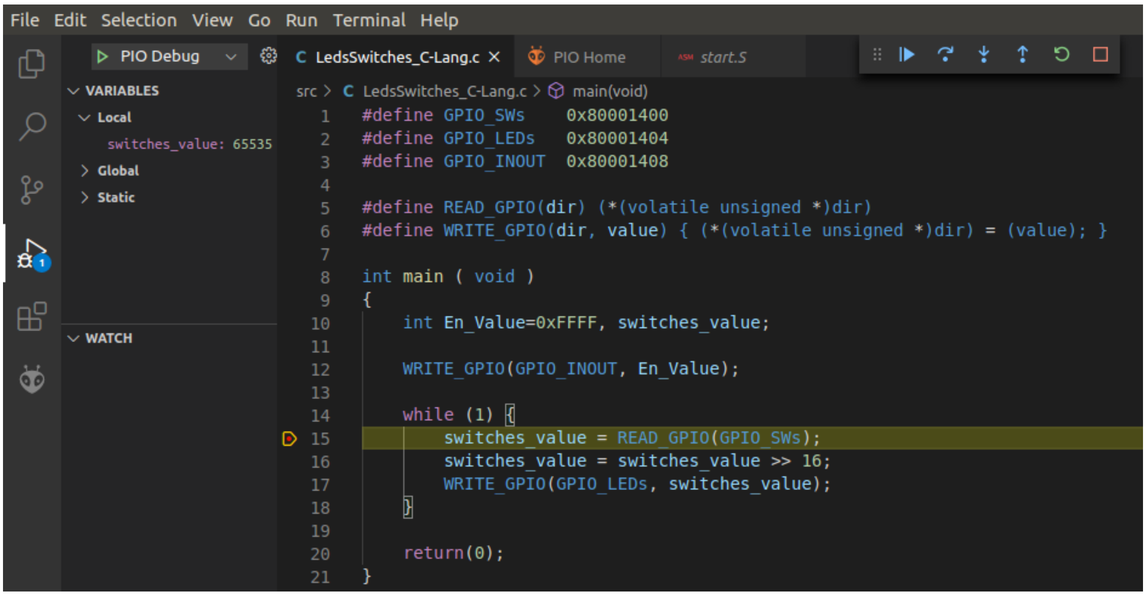


Figure 63. Execution stopped at the breakpoint

5. Make the program continue execution  several times, but change the switches in between each click. The LEDs should display the value of the switches.

6. You can view the execution of the program in C as above or you can view the execution of the assembly program generated by the compiler, by clicking on Switch to assembly highlighted in Figure 64.

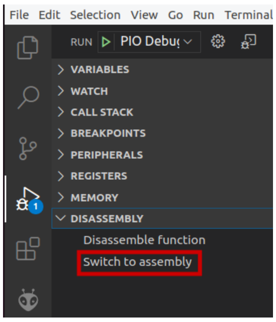


Figure 64. Switch to assembly

7. The program in assembly (Figure 65) first reads the value in the Switches with a load instruction (`lw a5,1024(a4)`) and then writes it to the LEDs with a store instruction (`sw a5,1028(a4)`). Execute it step by step, change the switches and verify that the LEDs change to reflect the new switch values.

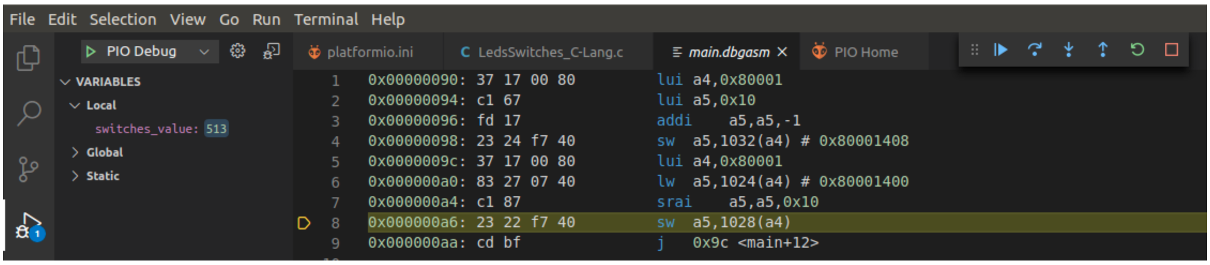


Figure 65. Assembly program

F. HelloWorld_C-Lang program

The second C example prints a short message to your shell through the serial port. To view this message, you could use any terminal emulator such as *gtkterm*, *minicom*, etc.; however, PlatformIO provides its own serial monitor, so here we show how to use this monitor.

For configuring PlatformIO serial monitor some parameters must be configured; specifically, the data rate (in bits per second, or bauds) for serial data transmission must be established, which we can do by using the *monitor_speed* parameter in file *platformio.ini* (note that this file is part of your PlatformIO projects). See Figure 66.

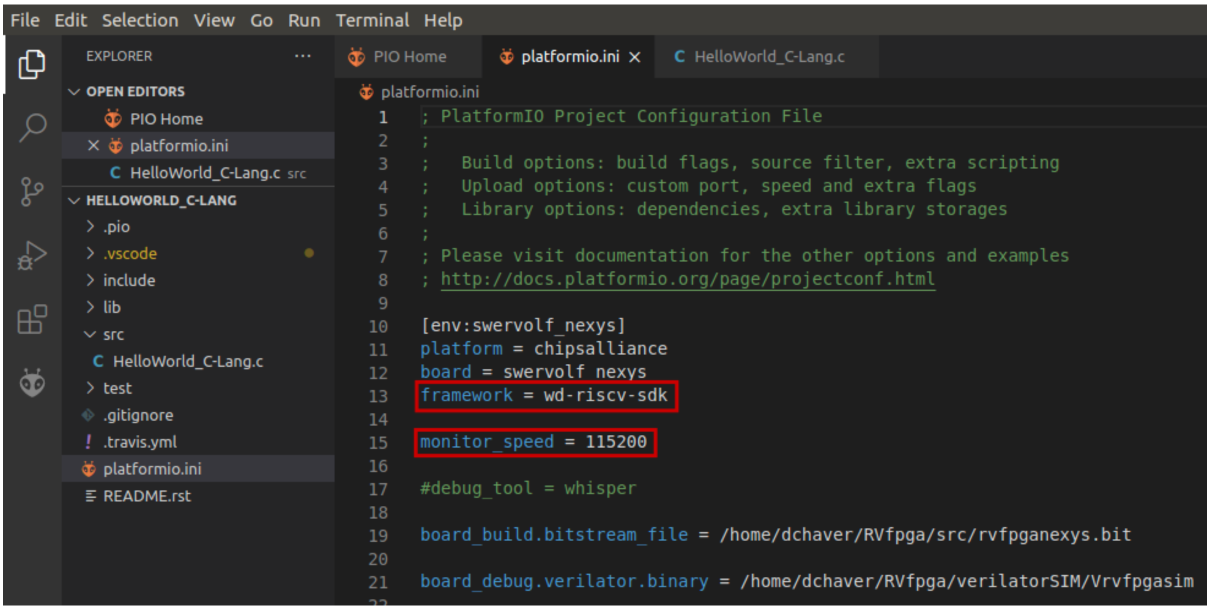


Figure 66. Serial monitor configuration

In addition, you need to add yourself to the `dialout`, `tty` and `uucp` groups by typing the following commands in a terminal:

```
sudo usermod -a -G dialout $USER
sudo usermod -a -G tty $USER
sudo usermod -a -G uucp $USER
```

After the three commands, restart your computer so that the changes in groups can take effect.

Windows/macOS: Windows and macOS users do not need to complete the above step.

Furthermore, this program uses the Processor Support Package (PSP) and Board Support Package (BSP) provided by WD within its Firmware Package (https://github.com/chipsalliance/riscv-fw-infrastructure/). These libraries are included in the project using a specific command in platformio.ini (`framework = wd-riscv-sdk`), as shown in Figure 66, and by including the proper files at the beginning of the C program, as shown in Figure 67. You can find the complete libraries in your system in the following paths:

- PSP: `~/.platformio/packages/framework-wd-riscv-sdk/psp/`

- BSP: `~/.platformio/packages/framework-wd-riscv-sdk/board/nexys_a7_eh1/bsp/`

These libraries provide many functions and macros that allow you do many things such as using interrupts, printing a string, reading/writing individual registers… In this example, we will use the `printfNexys` function for printing a message on the serial monitor. In subsequent examples and in the labs we will show how to use other functions and macros for different purposes.

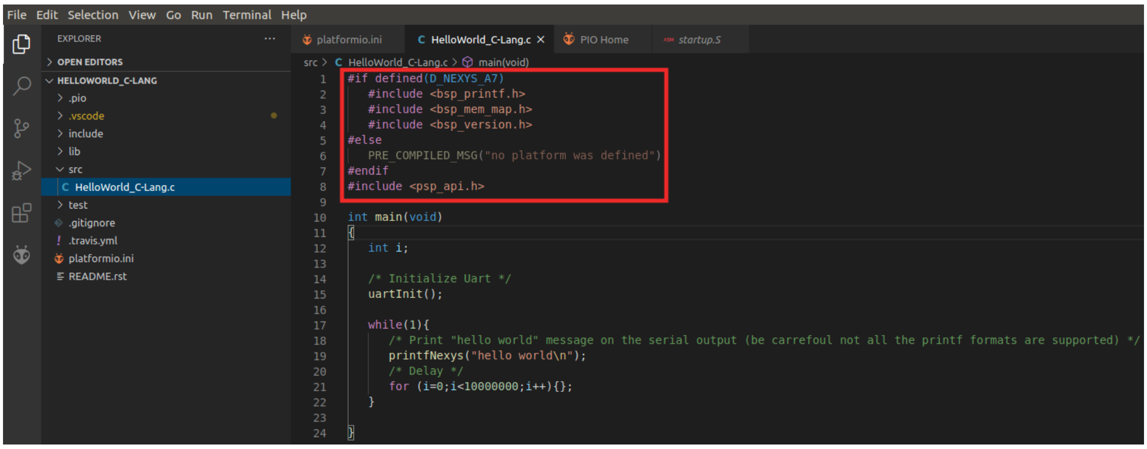


Figure 67. Include *.h* files in *HelloWorld_C-Lang.c*

Follow the next steps for running and debugging this code on the FPGA board:

1. RVfpgaNexys is already programmed on the FPGA board if you executed the previous examples, so you should not need to program it again. However, if you do need to reprogram RVfpgaNexys onto the board again, do it as explained in Section A, using the HelloWorld_C-Lang example instead of the AL_Operations example.

2. Open VSCode. PlatformIO should automatically open within VSCode when you open VSCode. On the top bar, click on File → Open Folder, and browse to directory *[RVfpgaPath]/RVfpga/examples/*. Select the *HelloWorld_C-Lang* folder and click OK.

3. The program *HelloWorld_C-Lang.C* (Figure 68) initializes the UART (function **uartInit**) and then sends the string through the serial port, using function **printfNexys** (you can find the implementation of these functions in file *~/.platformio/packages/framework-wd-riscv-sdk/board/nexys_a7_eh1/bsp/bsp_printf.c*). It then delays some time before going back to the beginning of the loop.

```c
#if defined(D_NEXYS_A7)
   #include <bsp_printf.h>
   #include <bsp_mem_map.h>
   #include <bsp_version.h>
#else
   PRE_COMPILED_MSG("no platform was defined")
#endif
#include <psp_api.h>

int main(void)
{
   int i;

   /* Initialize Uart */
   uartInit();

   while(1){
      /* Print "hello world" message on the serial output (be carrefoul not all the printf formats are supported) */
      printfNexys("hello world\n");
      /* Delay */
      for (i=0;i<10000000;i++){};
   }

}
```

Figure 68. HelloWorld_C-Lang.C main function

4. Launch the debugger in PlatformIO. When the program starts to run, open the serial monitor, by clicking on the *plug* button available on the bottom of VS Code (Figure 69).

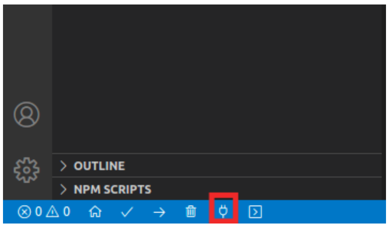


Figure 69. Open serial terminal

5. The serial monitor repeatedly prints the message “HELLO WORLD !!!”, as shown in Figure 70.

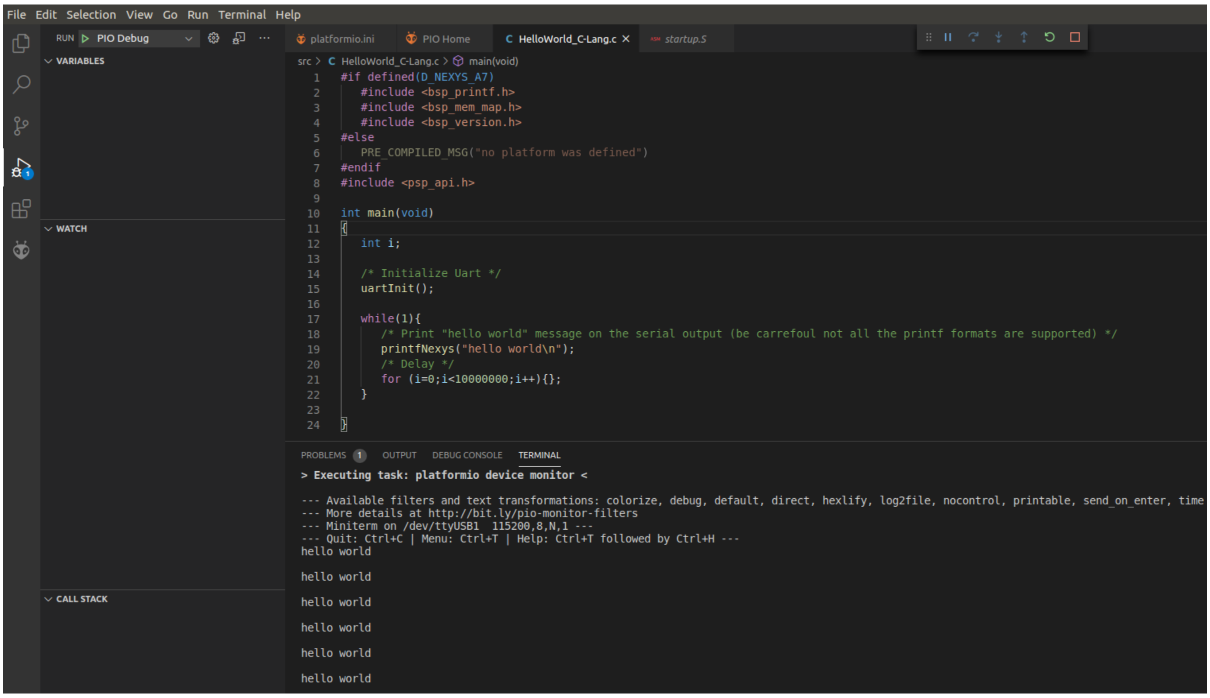


Figure 70. Execution of the program

G. VectorSorting_C-Lang program

Finally, we show another C program that sorts the elements of a vector, A, from largest to smallest and places the sorted values in a second vector, B. Vector A values are replaced with zeroes. Figure 71 shows the program.

```
#define N 8

int A[N]={7,3,25,4,75,2,1,1};
int B[N];

int main ( void )
{
    int max, ind, i, j;

    for(j=0; j<N; j++){
        max=0;
        for(i=0; i<N; i++){
            if(A[i]>max){
                max=A[i];
                ind=i;
            }
```

```
17          }
18          B[j]=A[ind];
19          A[ind]=0;
20      }
21
22      while(1);
23  }
```

Figure 71. VectorSorting_C-Lang.c

Follow the next steps for running and debugging this program on the FPGA board:

1. RVfpgaNexys is already programmed on the FPGA board if you executed the previous examples, so you should not need to program it again. However, if you do need to reprogram RVfpgaNexys onto the board again, do it as explained in Section A, using the VectorSorting_C-Lang example instead of the AL_Operations example.

2. On the top menu bar, click on *File → Open Folder*, and browse into directory *[RVfpgaPath]/RVfpga/examples/.* Select the *VectorSorting_C-Lang* folder and click OK.

3. Place a breakpoint at line 10 and start debugging. The execution will stop at the beginning of the `for` loop (Figure 72). Expand the VARIABLES section in the Debugger Side Bar and analyse the values of the A and B arrays (highlighted in red in Figure 72).

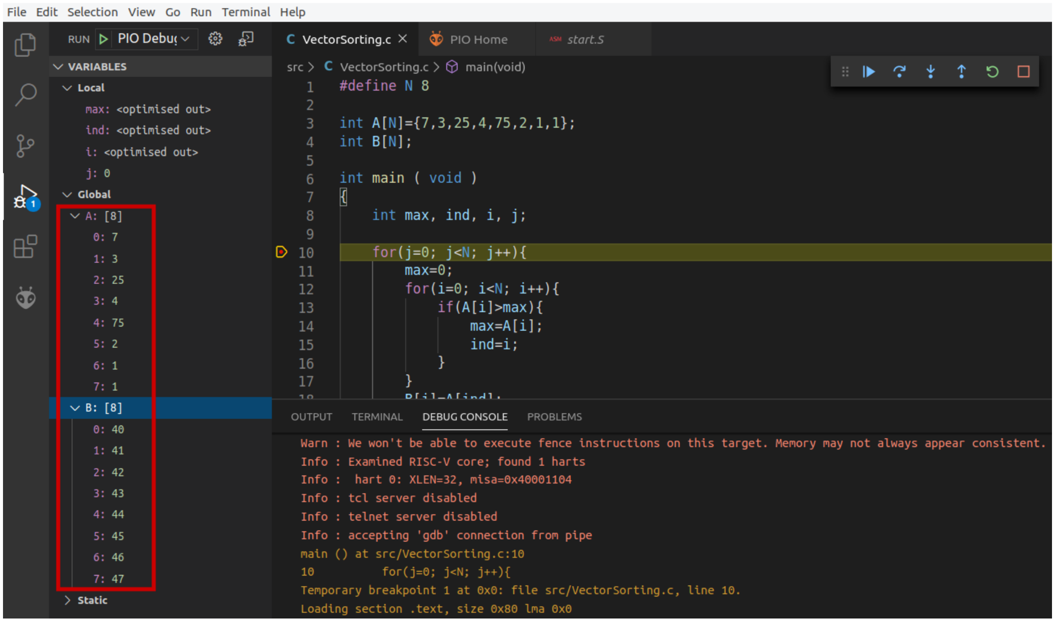


Figure 72. Execution stopped at the beginning of the program

4. Now place another breakpoint at line 18 and continue execution by clicking on  (see Figure 73). Open the Memory Display (as explained for the LedsSwitches program, Figure 55) and show 0x50 bytes starting from address 0x2148 (see Figure 73), which is the address where vector A is stored in memory for this program. You can view the initial values of vectors A (in the range 0x2148-0x2167) and B (in the range 0x2178-0x2197).

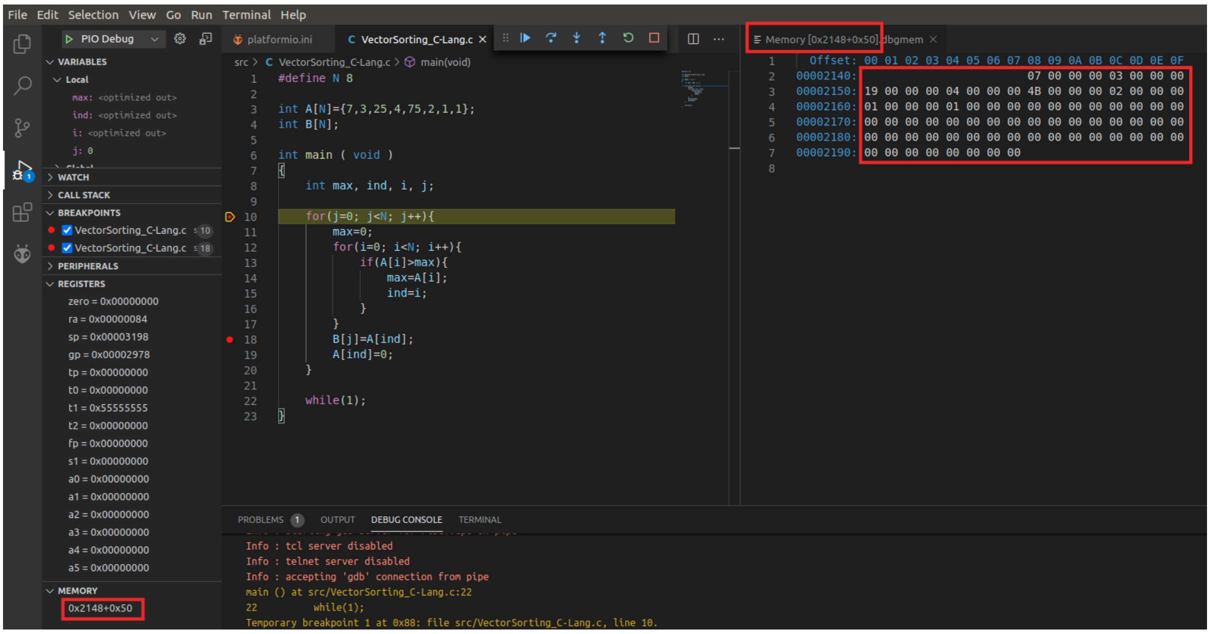


Figure 73. Memory Display for arrays A and B – Initial state.

Note that you can easily find out the address where vectors A and B are stored in memory by switching to assembly, as explained in Figure 64, and analysing any of the instructions that access these vectors (Figure 74). As you see in the figure, in most cases the comments provide this information; however, you could also step up to those instructions and see the value that is stored in the register.

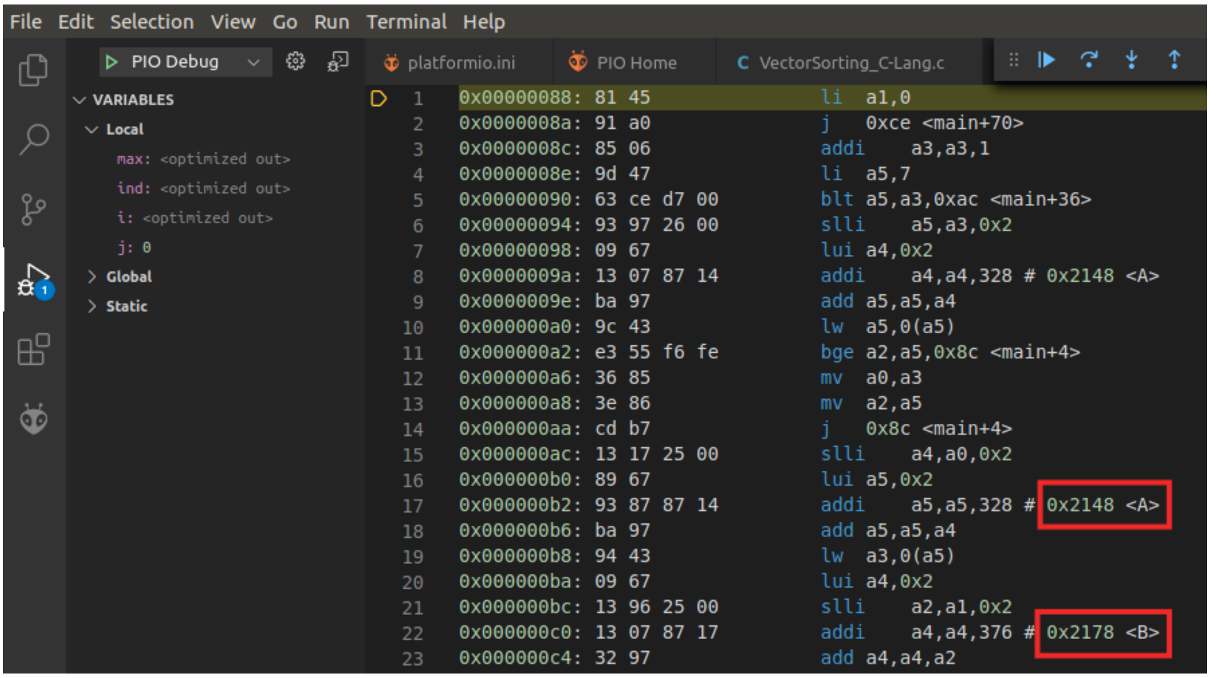


Figure 74. Address where A and B are stored in memory.

Click twice on the Step Over button (), and you will see the first component of B stored in memory and the corresponding value in A set to 0 (see Figure 75).

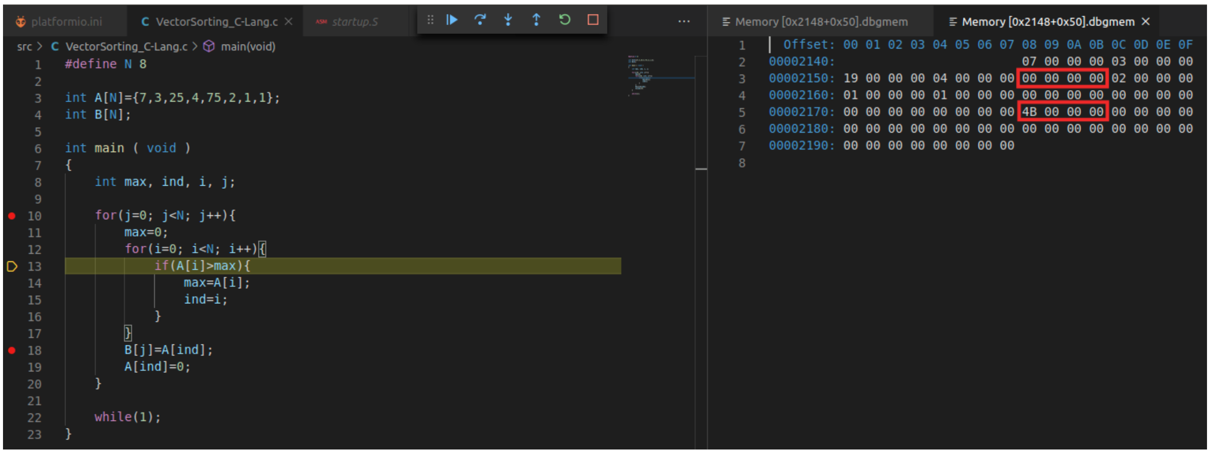


Figure 75. Memory Display for arrays A and B – Store the first component of B and reset corresponding component in A.

5. Remove all breakpoints, continue execution and pause it after several seconds – at which point the program will have finished executing. Again, analyse the values stored in the A and B arrays. As shown in Figure 76, vector B holds the values from the original vector A sorted from largest to smallest and vector A holds all zeroes (you can see this both at the variables list on the left and at the memory console on the right).

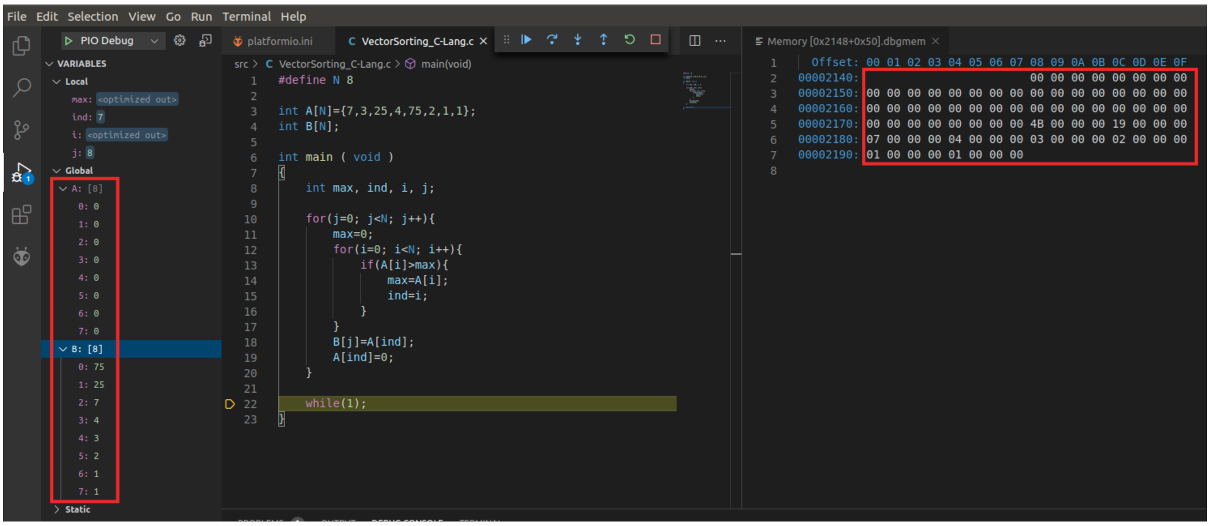


Figure 76. Execution stopped at the end of the program

H. DotProduct_C-Lang program

The last example program, DotProduct_C-Lang.c (Figure 77), computes the dot product of two vectors. The program has two functions: *main* and *dotproduct*. The first function invokes the second one with three input arguments: vector size, and the initial addresses of two vectors. Then, the *dotproduct* function computes the dot product of the two vectors and returns the result.

```
1  #define DIM 3
2
3  double dot;
4
5  double dotproduct(int n, double a[], double b[]){
6    volatile int i;
7    double sum=0;
8
9    for (i=0; i<n; i++) {
10        sum += a[i]*b[i];
11   }
12   return sum;
13 }
14
15 void main(void) {
16    double x[DIM] = {3.1, 4.3, 5.9};            // x is an array of size 3(DIM)
17    double y[DIM] = {1.4, 2.2, 3.7};            // same as x
18
19    dot = dotproduct(DIM, x, y);
20
21    return;
22 }
```

Figure 77. DotProduct_C-Lang.c

In this example we operate with real numbers (note that the data type for the variables `x`, `y` and `dot`, is `double`). However, the SweRV EH1 processor does not include floating-point support. Thus, the example uses floating point emulation through the software floating point library provided by *gcc* (https://gcc.gnu.org/onlinedocs/gccint/Soft-float-library-routines.html). This library is used whenever `-msoft-float` is included to disable generation of floating point instructions.

Follow the next steps for running and debugging this code on the FPGA board:

1. RVfpgaNexys is already programmed on the FPGA board if you executed the previous examples, so you should not need to program it again. However, if you do need to reprogram RVfpgaNexys onto the board again, do it as explained in Section A, using the DotProduct_C-Lang example instead of the AL_Operations example.

2. On the top menu bar, click on *File → Open Folder*, and browse into directory *[RVfpgaPath]/RVfpga/examples/.* Select directory *DotProduct_C-Lang* and click OK.

3. Before calling the debugger, set a breakpoint at line 10 and another one at line 19 (see Figure 78).

4. Then, start debugging. The program will start executing; stop it at the first breakpoint (see Figure 78).

5. On the Debugger sidebar, expand the Variables section (see Figure 78). The two vectors contain the initial values assigned in *main*. The *dot* variable is initialized to 0.

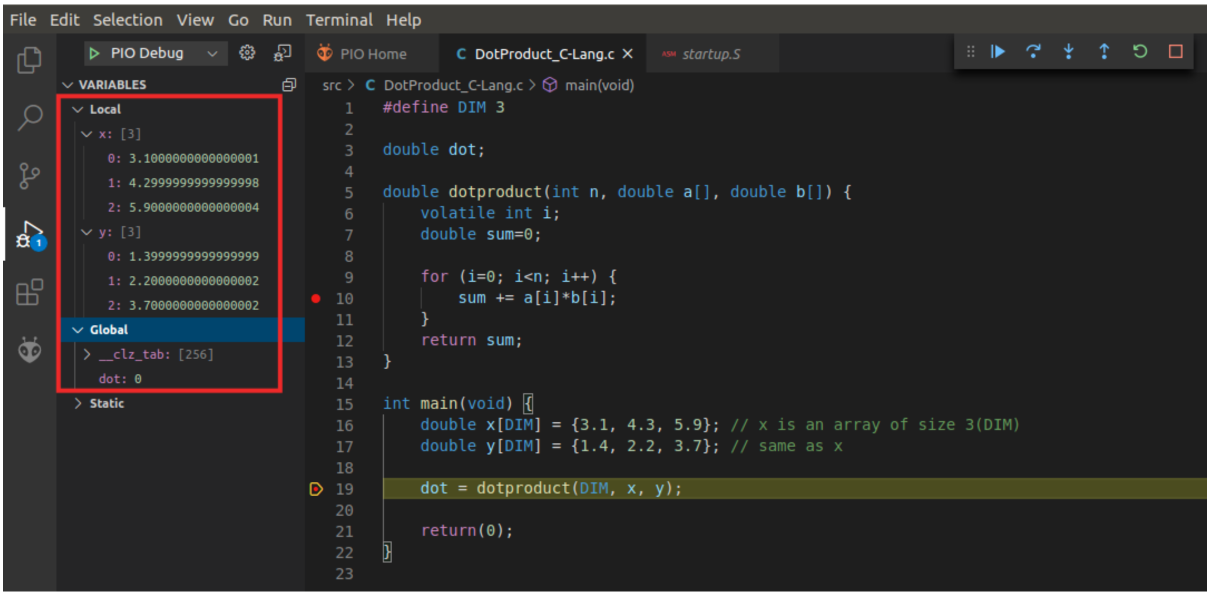


Figure 78. DotProduct_C-Lang program: values of the variables at the first breakpoint

6. Make the program continue execution . The program stops at the second breakpoint (line 10).

7. Switch to assembly (as you did in Figure 64). You can see the floating point emulation routines and analyse them in detail by stepping into them (see Figure 79).

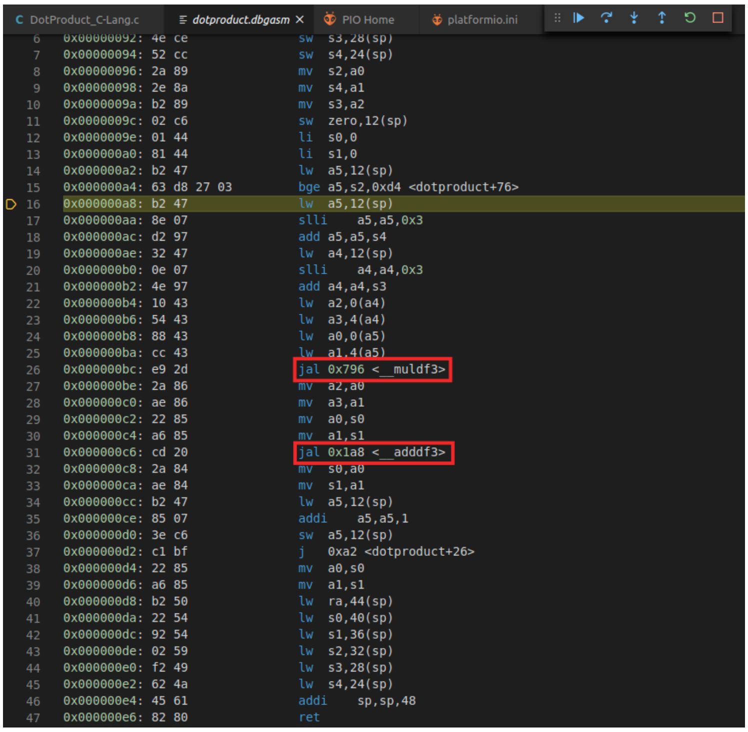


Figure 79. DotProduct_C-Lang program: assembly code at the second breakpoint

8. Switch back to C and delete the two breakpoints. Continue execution and pause it. You will see that the value of variable *dot* will change to the dot product of the two vectors (Figure 80).

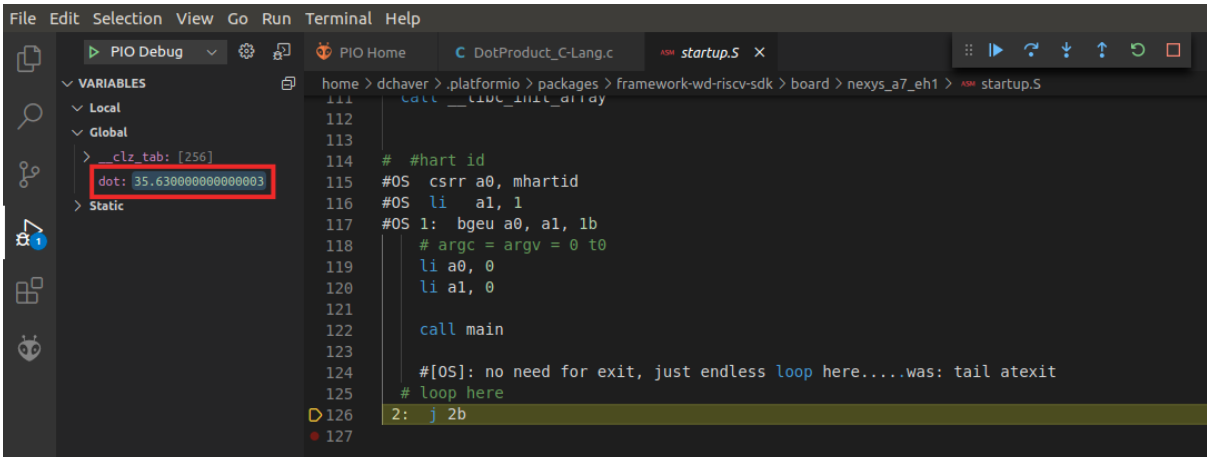


Figure 80. DotProduct_C-Lang program: result of the dot product

9. Once you are finished exploring this program, close the project by clicking on *File → Close Folder.*

7. SIMULATION IN VERILATOR

In this section, you will run the first program used in the previous section (*AL_Operations*) on RVfpgaSim using Verilator. Verilator is a hardware description language (HDL) Simulator that simulates the Verilog that defines the SoC (available at *[RVfpgaPath]/RVfpga/src*). This way of running the SoC allows you to analyse the internal signals of the system, which is especially useful for future labs and exercises where we add internal operations or new hardware to the SoC.

Here we show how to use Verilator to view the cycle-by-cycle instructions and register values of the *AL_Operations*, the first simple assembly program that you executed and debugged in Section 6 (Figure 44). You will generate the simulation trace using PlatformIO and then add the clock, instructions for both ways of the super-scalar processor, and register `x28` (i.e., register `t3`) signals to the simulation waveform, and view with GTKWave the instruction and register signals change as the program executes.

GENERATE THE SIMULATION BINARY, Vrvfpgasim:

Directory *[RVfpgaPath]/RVfpga/verilatorSIM* contains the *Makefile* and the *script* (*swervolf_0.7.vc*) for generating the simulator binary for RVfpgaSim. The *script* contains information for Verilator to know, among other things, where to find the sources for the SoC, which in our case are available at *[RVfpgaPath]/RVfpga/src*. We next show how you can generate the binary for RVfpgaSim, that later will be used for creating the simulation trace of program *AL-Operations* running on RVfpgaSim.

1. In a terminal window, generate the simulator binary by executing the following commands:

   ```
   cd [RVfpgaPath]/RVfpga/verilatorSIM
   make clean
   make
   ```

   File ***Vrvfpgasim*** (the RVfpgaSim simulation binary), should be generated inside directory *[RVfpgaPath]/RVfpga/verilatorSIM*.

> **Windows:** if you are using Windows, you must do these same steps inside the Cygwin terminal (refer to Appendix C for the detailed instructions). Note that the *C:* Windows folder can be found inside Cygwin at: */cygdrive/c*. All the other instructions from this section are the same as those described for Linux.

> **macOS:** Refer to Appendix D for the detailed instructions.

GENERATE THE SIMULATION TRACE FROM PLATFORMIO, USING Vrvfpgasim:

Once the simulator binary (*Vrvfpgasim*) has been generated, you will use it inside PlatformIO for generating the simulation trace (*trace.vcd*) of program AL_Operations.

2. Open VSCode and then PlatformIO in your computer.

3. On the top bar, click on *File→Open Folder...* (Figure 81), and browse into directory *[RVfpgaPath]/RVfpga/examples/*

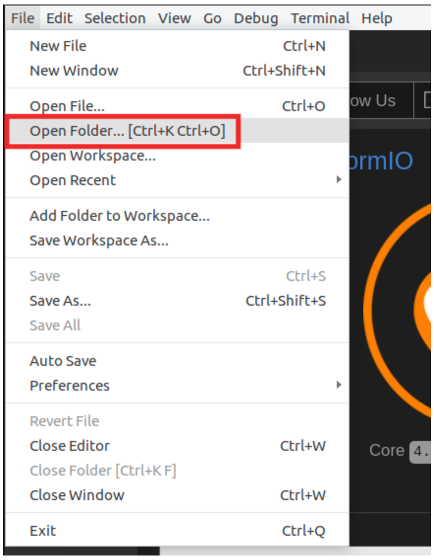


Figure 81. Open the AL_Operations.S example

4. Select directory *AL_Operations* (do not open it, but just select it) and click OK. The example will open in PlatformIO.

5. Open file *platformio.ini*. Establish the path to the RVfpgaSim simulation binary generated in the first step (*Vrvfpgasim*) by editing the following line (see Figure 82).

```
board_debug.verilator.binary =
[RVfpgaPath]/RVfpga/verilatorSIM/Vrvfpgasim
```

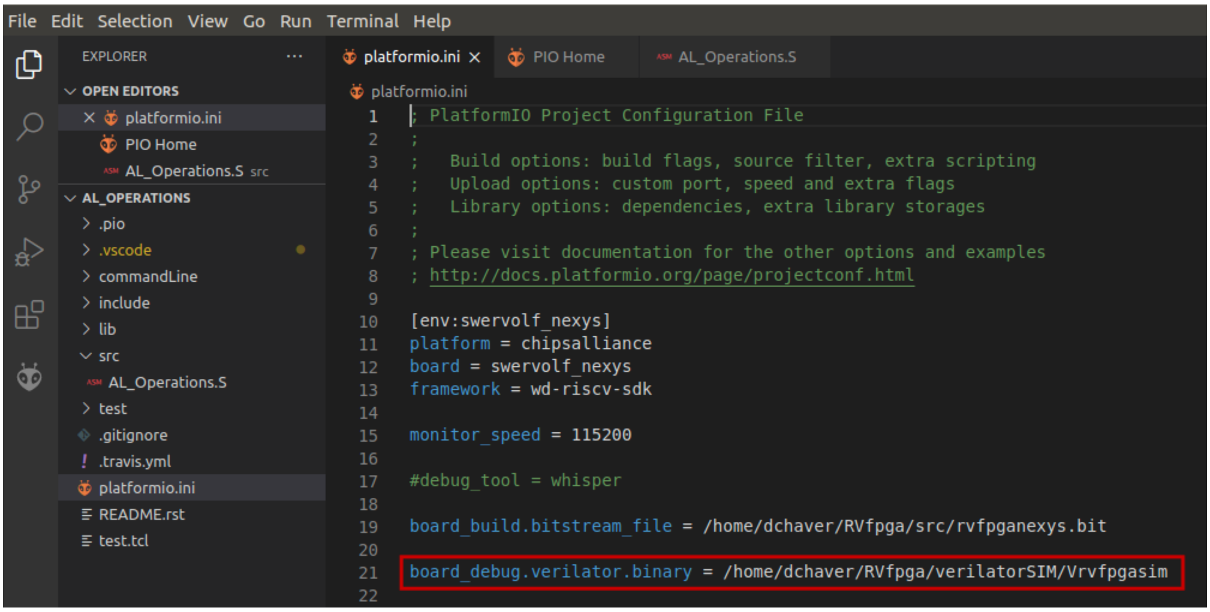


Figure 82. PlatformIO initialization file: platformio.ini

Windows: in Windows, the RVfpgaSim simulation executable is called `Vrvfpgasim.exe`. Thus:

```
board_debug.verilator.binary = [RVfpgaPath]\RVfpga\verilatorSIM\Vrvfpgasim.exe
```

6. Run the simulation by clicking on the PlatformIO icon in the left menu ribbon , then expand Project Tasks → env:swervolf_nexys → Platform and click on Generate Trace, as shown in Figure 83.

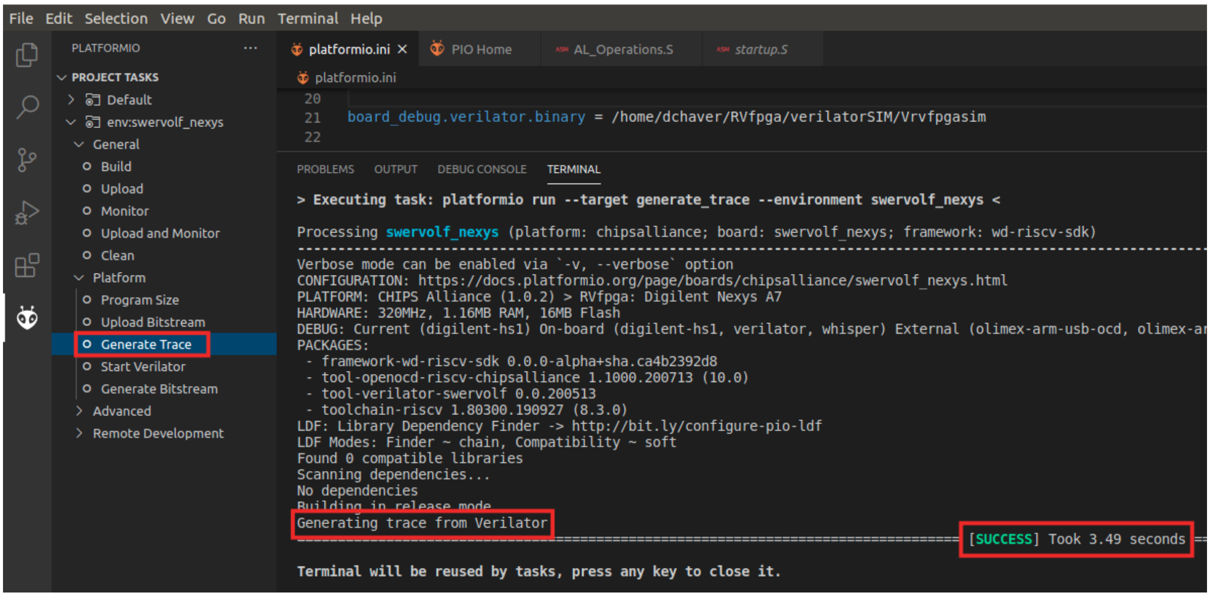


Figure 83. Generating trace from Verilator

As an alternative, you can generate the trace from a PlatformIO terminal window. For that purpose, click on the button (PlatformIO: New Terminal button) at the bottom of the PlatformIO window for opening a new terminal window, and then type (or copy) the following command into the PlatformIO terminal: `pio run --target generate_trace`

7. A few seconds after the previous step, file *trace.vcd* should have been generated inside *[RVfpgaPath]/RVfpga/examples/AL_Operations/.pio/build/swervolf_nexys*, and you can open it with *GTKWave*.

```
gtkwave [RVfpgaPath]/RVfpga/examples/AL_Operations/.pio/build/swervolf_nexys/trace.vcd
```

WINDOWS: folder *gtkwave64* that you downloaded, includes an application called *gtkwave.exe* inside the *bin* folder. Launch GTKWave by double clicking on that application. On the top part of the application, click on **File – Open New Tab**, and open the trace.vcd file generated in folder *[RVfpgaPath]/RVfpga/examples/AL_Operations/.pio/build/swervolf_nexys*.

ANALYSE THE SIMULATION TRACE IN GTKWAVE:

8. Now you will add clock, instruction, and register signals. On the top left pane of *GTKWave*, expand the hierarchy of the SoC so that you can add signals to the graph. Expand the hierarchy into **TOP → rvfpgasim → swervolf → swerv_eh1 → swerv,** and click on module **ifu** (it will highlight as shown in the Figure 84), select signal *clk* (which is

the clock used for the core) and drag it into the white Signals pane or the black Waves pane on the right.

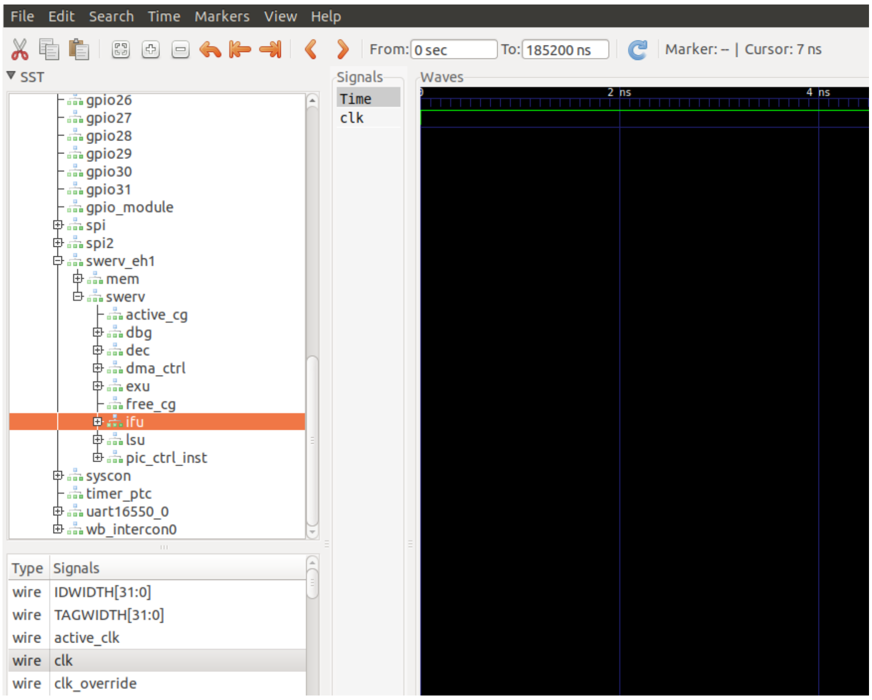


Figure 84. Add signal *clk* to the graph

9. Zoom in several times so that you can view the clock signal change (Figure 85).

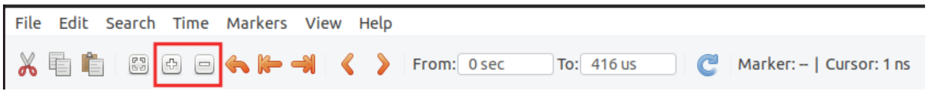


Figure 85. Zoom in

10. Now add the signals that show the instructions that execute in each way of the two-way superscalar RISC-V core. In the same module (**ifu**) look for signals *ifu_i0_instr[31:0]* and *ifu_i1_instr[31:0]* (Figure 86), and drag them into the black Waves pane. The prefix *ifu* indicates the instruction fetch unit, *i0* indicates superscalar way 0 and *i1* indicates superscalar way 1; *instr*[31:0] indicates the 32-bit instruction.

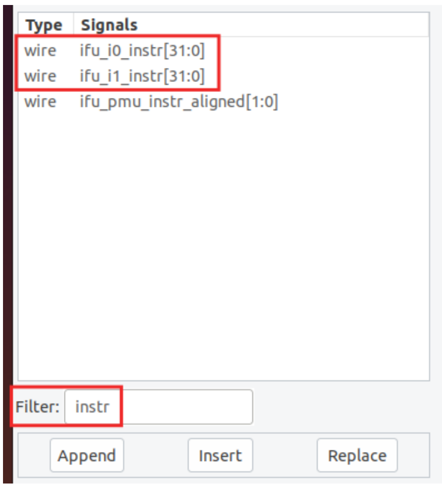


Figure 86. Add signals *ifu_i0_instr[31:0]* and *ifu_i1_instr[31:0]* to the timing waveform

11. Now add the signal that holds the value of register `t3` (i.e., register number 28, `x28`). Expand the hierarchy under **swerv** into **dec** → **arf** → **gpr_banks(0)** → **gpr(28)** and click on module **gprff** (it will highlight as shown in the following figure), select signal *dout[31:0]* (which shows the contents of register `x28`, used in the *AL_Operations.S* example) and drag it into the black Waves pane (Figure 87).

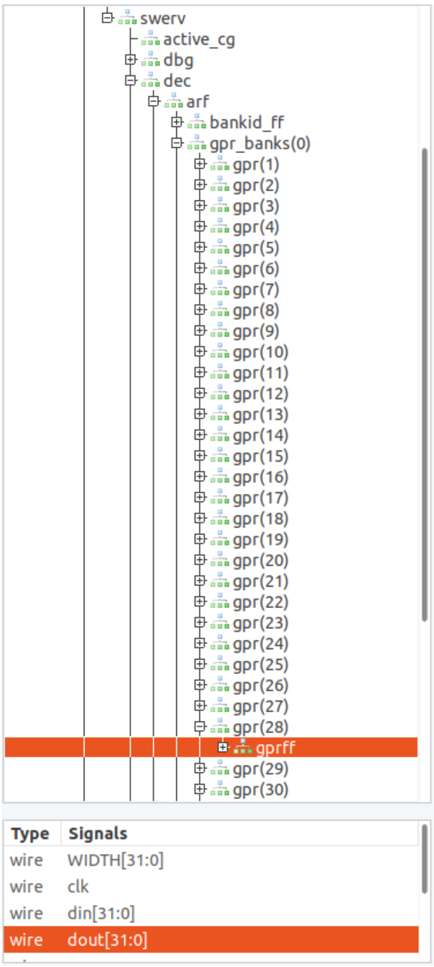


Figure 87. Add signal *dout[31:0]* to the graph

12. Another way of showing signals in GTKWave is to use a *.tcl* file. File *test.tcl* is provided at *[RVfpgaPath]/RVfpga/examples/AL_Operations*. Open that file and analyse it. In each line, you will see the path and the name of each signal that we want to show in the graph.

```
gtkwave::addSignalsFromList rvfpgasim.clk
gtkwave::addSignalsFromList rvfpgasim.swervolf.swerv_eh1.swerv.ifu.ifu_i0_instr
gtkwave::addSignalsFromList rvfpgasim.swervolf.swerv_eh1.swerv.ifu.ifu_i1_instr
gtkwave::addSignalsFromList rvfpgasim.swervolf.swerv_eh1.swerv.dec.arf.gpr_banks(0).gpr(28).gprff.dout
```

For using the *.tcl* file on GTKWave, you can simply click on *File – Read Tcl Script File* and select the *[RVfpgaPath]/RVfpga/examples/AL_Operations/test.tcl* file.

13. Figure 88 shows the AL_Operations.S program and its equivalent machine instructions.

```
# RISC-V assembly                 # comment (t3 = x28)        # machine code
    li    t3, 0x0                 # t3 = 0                    # 0x00000E13
```

```
REPEAT:
    addi t3, t3, 6              # t3 = t3 + 6           # 0x006E0E13
    addi t3, t3, -1             # t3 = t3 – 1           # 0xFFFE0E13
    andi t3, t3, 3              # t3 = t3 AND 3         # 0x003E7E13

    beq  zero, zero, REPEAT     # Repeat the loop       # 0xFE000CE3
    nop                         # nop                   # 0x00000013
```

Figure 88. AL_Operations.S with equivalent machine code

Now view the signals change as the program executes. We expect the instructions and `t3` (register `x28`) to become the values shown in Figure 89 as the program runs:

```
            li   t3, 0x0            # t3 = 0            # 0x00000E13
REPEAT:     addi t3, t3, 6          # t3 = 0 + 6 = 6    # 0x006E0E13
            addi t3, t3, -1         # t3 = 5            # 0xFFFE0E13
            andi t3, t3, 3          # t3 = 5 & 3 = 1    # 0x003E7E13
            beq  zero, zero, REPEAT # Repeat the loop   # 0xFE000CE3
            nop                     # nop               # 0x00000013
REPEAT:     addi t3, t3, 6          # t3 = 1 + 6 = 7    # 0x006E0E13
            addi t3, t3, -1         # t3 = 7 – 1 = 6    # 0xFFFE0E13
            andi t3, t3, 3          # t3 = 6 & 3 = 2    # 0x003E7E13
            beq  zero, zero, REPEAT # Repeat the loop   # 0xFE000CE3
            ...
```

Figure 89. Instruction flow and values of register `t3` (`x28`) during AL_Operations execution

14. Zoom in around 10100 ps, where you will analyse the execution of the three arithmetic-logic instructions of the first and second iterations of the loop (Figure 90). The first two instructions (`li t3, 0x0` = `0x00000E13` and `addi t3, t3, 6` = `0x006E0E13`) are fetched first, one in each way of the superscalar RISC-V processor as shown on signals *ifu_i0_instr*[31:0] and *ifu_i1_instr*[31:0]. The next two instructions (`addi t3, t3, -1` = `0xFFFE0E13` and `andi t3, t3, 3` = `0x003E7E13`) are fetched in the next cycle. The last two instructions are fetched (`beq zero, zero, REPEAT` = `0xFE000CE3` and `nop` = `0x00000013`) in the next cycle.

    Because of the SweRV core's 9-stage pipelined processor and dependencies, the effects of the instructions are seen eight or more cycles after the instructions are fetched. Eight cycles after the first and second instructions are fetched, `x28` (`t3`) becomes 0 (which it was already) because of the first instruction: `li t3, 0x0` (`0x00000E13`). One cycle later, `x28` is updated to 0x6 because of the next instruction: `addi t3, t3, 6` (`0x006E0E13`). Next, `x28` updates to 5 because of the next instruction: `addi t3, t3, -1` (`0xFFFE0E13`). Finally, `x28` updates to 1 because of the next instruction: `andi t3, t3, 3` (`0x003E7E13`). Then the next two instructions are fetched: `beq zero, zero, REPEAT` (`0xFE000CE3`) and `nop` (`0x00000013`), the branch is taken and the loop repeats. This is as predicted in Figure 89. Using a similar reasoning, you can analyse the second iteration, which is also highlighted in Figure 90 and predicted in Figure 89.

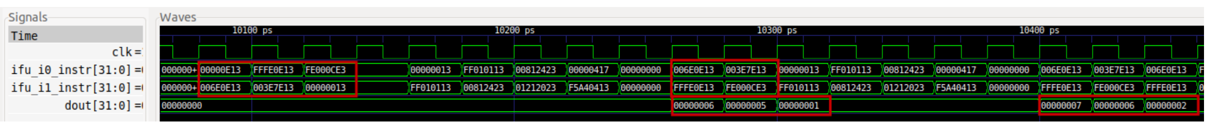


Figure 90. Execution of the three Arithmetic-Logic instructions from the example

8. SIMULATION IN WHISPER

Whisper (https://github.com/chipsalliance/SweRV-ISS) is a RISC-V instruction set simulator (ISS) developed by Western Digital for the verification of the SweRV micro-controller. It allows the user to run RISC-V code without requiring underlying RISC-V hardware. Using Whisper, you can test, run, and debug C or assembly programs using PlatformIO without requiring the Nexys A7 FPGA board.

Windows: All the instructions described in this section should work for Windows (we'd like to thank Jean-François Monestier, who was the first to port Whisper to Windows: https://jean-francois.monestier.me/porting-western-digital-swerv-iss-to-windows/). Note that a pop-up window may ask you to allow Whisper through the Windows firewall.

macOS: All the instructions described in this section also work for macOS.

Whisper can be executed both using the command line or using an IDE (integrated development environment) such as Eclipse or PlatformIO. In this section we demonstrate one example to show how to simulate a program with Whisper in PlatformIO. You can then use the same steps as the ones described here to simulate other programs.

We start by using the Whisper ISS to simulate *AL_Operations*, the first simple assembly program that you executed and debugged in Section 6 (see Figure 44). Follow the next steps for running and debugging this code on Whisper:

1. Open VSCode (and PlatformIO). On the top menu bar, click on *File → Open Folder* and browse into directory *[RVfpgaPath]/RVfpga/examples/*, select (but do not open) directory *AL_Operations* and then click OK.

2. Click on *File → Open File* and double-click on *[RVfpgaPath]/RVfpga/examples/AL_Operations/platformio.ini*, and set **whisper** as the debug tool by uncommenting line 17 (see Figure 91). Save the file (press Ctrl-s).

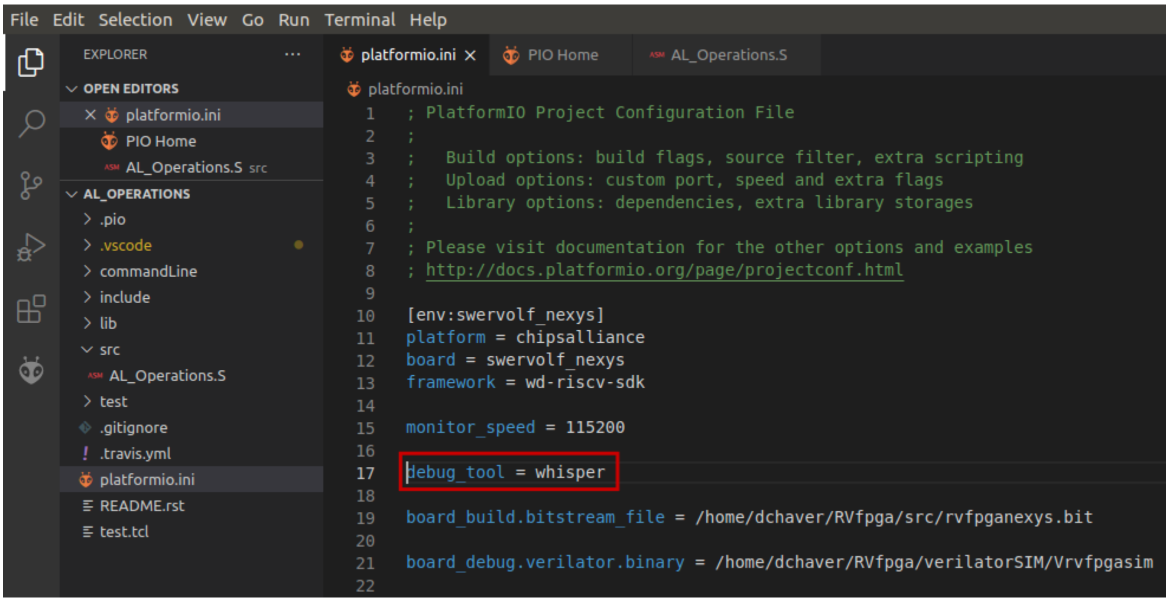


Figure 91. Uncomment line 17.

3. Launch the debugger as usual, by clicking on  and then on 

4. You can now debug the program exactly as you did in Section 6.B, but this time the program is running in simulation on Whisper instead of on the Nexys A7 FPGA board.

5. If a program uses the *printfNexys* function in Whisper, such as the HelloWorld_C-Lang example (Section 6.F), you should not open the PlatformIO serial monitor, as messages are shown in the DEBUG console instead (see Figure 92).

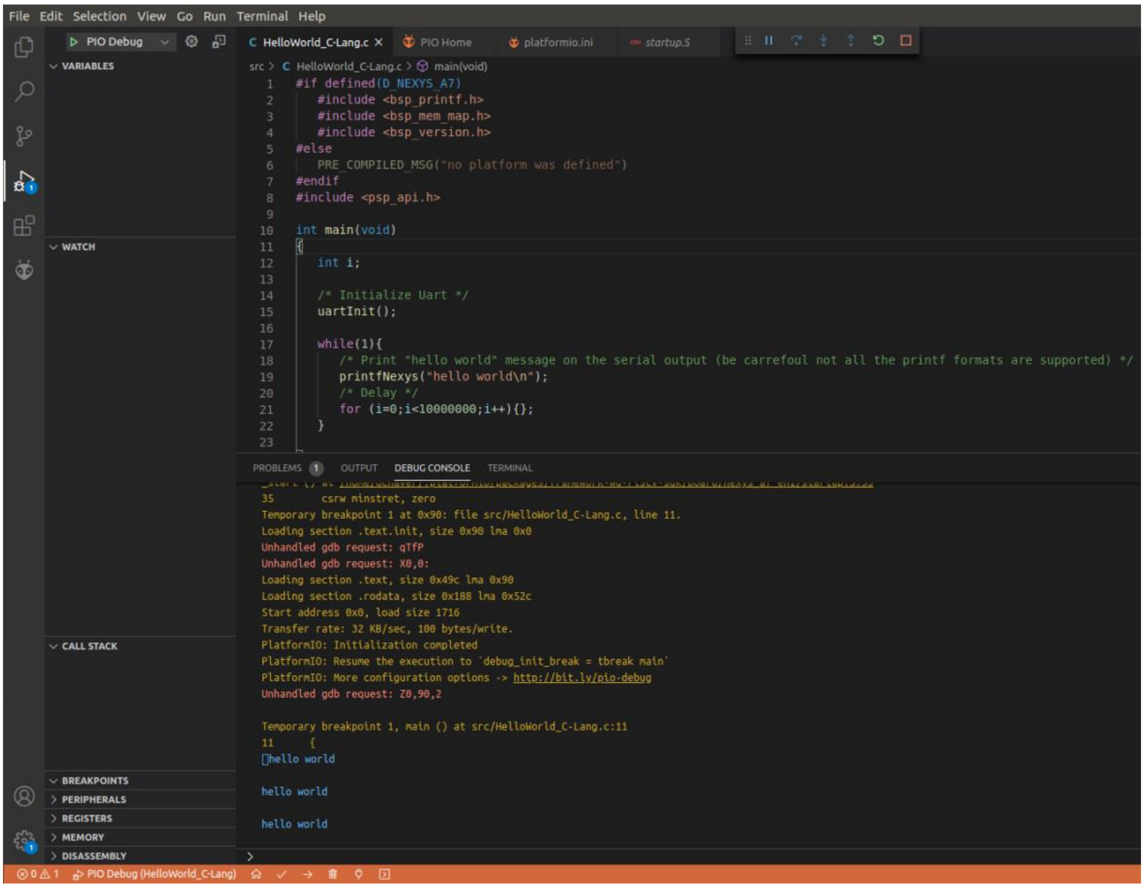


Figure 92. Execution of the HelloWorld_C-Lang example in Whisper

9. APPENDICES

The following appendices show how to use the native RISC-V toolchain and OpenOCD (instead of PlatformIO) in Linux, how to install in Windows the drivers to download the bitstream using PlatformIO, how to install Verilator and GTKWave on Windows and Mac OS machines, and how to program RVfpgaNexys using Vivado. Table 10 lists all of the appendices available in this RVfpga Getting Started Guide.

Table 10. List of Appendices

Appendix	Description	Operating System
A	Using the Native RISC-V Toolchain and OpenOCD for RVfpga in Ubuntu 18.04	Linux
B	Installing drivers in Windows to use PlatformIO	**Windows**
C	Installing Verilator and GTKWave in Windows	**Windows**
D	Installing Verilator and GTKWave in macOS	**macOS**
E	Using Vivado to download RVfpgaNexys onto an FPGA	**Windows** and Linux
F	Using RVfpga in an industrial IoT application	All

Appendix A should be used by those who want to natively compile and run/debug programs using the native gcc/gdb tools and OpenOCD. However, it is **recommended that RVfpga users use PlatformIO** instead, as described in this Getting Started Guide.

Windows users **must follow instructions in Appendices B and C**. Instructions in Appendix B show how to download drivers so that Windows systems can use PlatformIO to both download programs and download RVfpgaNexys onto the Nexys A7 FPGA board. Appendix C shows how to install Verilator and GTKWave so that Windows users can simulate the RVfpgaSim.

macOS users **must follow instructions in Appendix D** in order to simulate the RVfpgaSim using Verilator and GTKWave.

It is recommended to use PlatformIO to download the RVfpgaNexys system (as defined by the bitfile, rvfpganexys.bit) onto the Nexys A7 FPGA board. This bitfile (rvfpganexys.bit) can be generated by Vivado or PlatformIO. It is also possible to use Vivado to download the RVfpgaNexys system onto the Nexys A7 FPGA board, as described in Appendix E. However, using Vivado to download RVfpgaNexys onto the board is **not recommended** – especially for **Windows** users, as it would require continually swapping drivers.

Appendix A: Using the Native RISC-V Toolchain and OpenOCD in Ubuntu 18.04

Although we recommend the use of PlatformIO, in this section we show how to install, run, and use the native RISC-V toolchain and use OpenOCD to download RVfpgaNexys onto the Nexys A7 FPGA board and gdb to run and debug programs on RVfpgaNexys. The toolchain consists of a gnu compiler, debugger, assembler, etc. We show how to install the RISC-V toolchain and OpenOCD on an Ubuntu 18.04 operating system (OS), but this process should also work for other Linux distributions as well. These instructions assume a fresh Ubuntu system.

The following steps are not needed if you are using PlatformIO, as described earlier in this guide. Using PlatformIO, Vivado, and Verilator or Whisper is the recommended method for running, debugging, and simulating RISC-V programs, but the following instructions are provided for anyone who is interested in using the native RISC-V toolchain and OpenOCD in place of PlatformIO and the Vivado Hardware Manager.

I. Native installation on a Linux Ubuntu OS

In this section we describe how to install natively in your Ubuntu 18.04 machine the RISC-V toolchain, OpenOCD and Whisper. These tools only substitute PlatformIO; installing Vivado and Verilator is still required as explained in Section 5 of this GSG.

RISC-V Toolchain

Here we show how to install the complete RISC-V Toolchain – i.e., gnu compiler, debugger, etc. – on your computer. Installation instructions are provided by RISC-V International at: https://github.com/riscv/riscv-gnu-toolchain. These instructions are summarized below.

NOTE: Installing the RISC-V toolchain and OpenOCD could take several hours – mostly waiting while the toolchain downloads, compiles, and installs

At a terminal, type the following (the process can take more than an hour, but most of the time is spent waiting while the programs are downloaded and installed):

- `sudo apt-get install git autoconf automake autotools-dev curl libmpc-dev libmpfr-dev libgmp-dev gawk build-essential bison flex texinfo gperf libtool patchutils bc zlib1g-dev libexpat-dev`
- `git clone --recursive https://github.com/riscv/riscv-gnu-toolchain`
- `cd riscv-gnu-toolchain/`
- `./configure --prefix=/opt/riscv --with-arch=rv32imc`
- `sudo make` (If possible use `sudo make -j$(nproc)` as it significantly decreases compile time)
- `export PATH=$PATH:/opt/riscv/bin` (change the path in your system)

OpenOCD

OpenOCD is an open on-chip debugger that allows users to program and debug embedded target devices. Follow the next steps to install RISC-V OpenOCD onto your computer:

- `sudo apt-get install libusb-1.*`
- `sudo apt-get install pkg-config`
- `git clone https://github.com/riscv/riscv-openocd.git`
- `cd riscv-openocd/`
- `./bootstrap`
- `./configure --prefix=/opt/riscv --program-prefix=riscv- --enable-ftdi`

--enable-jtag_vpi
- make
- sudo make install

Whisper

Follow the next steps to install Whisper onto your computer (instructions are available at: https://github.com/chipsalliance/SweRV-ISS but are also summarized below):

- apt-cache policy libboost-all-dev
- sudo apt-get install libboost-all-dev
- cd [RVfpgaPath]
- git clone https://github.com/chipsalliance/SweRV-ISS
- cd SweRV-ISS
- make BOOST_DIR=/usr/include/boost
- export PATH=$PATH:[RVfpgaPath]/SweRV-ISS/build-Linux (replace [RVfpgaPath] as required).

II. Executing a program on RVfpgaNexys using the Nexys A7 FPGA board using OpenOCD

Step A. Download RVfpgaNexys (Figure 25) into the Nexys A7

1. Go into the project directory that contains the bitfile for RVfpgaNexys:
```
cd [RVfpgaPath]/RVfpga/src
```

2. Download RVfpgaNexys into the board using OpenOCD:
```
riscv-openocd -c "set BITFILE rvfpganexys.bit" -f
OtherSources/ConfigFiles/swervolf_nexys_program.cfg
```

Step B. Execute LedsSwitches, the program that reads the Switches and prints their state on the LEDs

3. Go into the LedsSwitches/commandLine directory:
```
cd [RVfpgaPath]/RVfpga/examples/LedsSwitches/commandLine
```

   In that directory you will find the Makefile for compiling the sources, the link script, a python script, and the *LedsSwitches.S* program.

4. Build the .elf file:
```
make clean
make LedsSwitches.elf
```

5. Connect OpenOCD to the SoC:
```
riscv-openocd -f
../../../src/OtherSources/ConfigFiles/swervolf_nexys_debug.cfg
```

   Once OpenOCD starts running, you will see several messages including one that says:
```
Info : Listening on port 4444 for telnet connections
```

6. Open a new terminal, and go into the program directory (cd [RVfpgaPath]/RVfpga/examples/LedsSwitches/commandLine) and run the following command:

```
telnet localhost 4444
```

Then, inside the telnet connection, type:

```
load_image LedsSwitches.elf
reg pc 0
resume
```

These three commands (1) load the LedsSwitches.elf program onto RVfpgaNexys, (2) set the program counter (PC) to 0 (the address location of the program's first instruction), and (3) resume execution.

The program will start to run on RVfpgaNexys, the RISC-V SweRVolfX SoC that was already downloaded onto the Nexys A7 FPGA board in Step 2. The program makes the LEDs show the state of the switches. As you toggle the switches, the LEDs should immediately change to reflect the value of the switches.

Step C. Debug the AL_Operations_CommandLine program that executes simple arithmetic-logic operations

Now we show how to debug another program (AL_Operations_CommandLine) using OpenOCD and gdb.

7. Keep the OpenOCD connection open (see Step 5).

8. In the other terminal where telnet is running (from Step 6), exit the telnet connection by typing:
   ```
   exit
   ```

9. Change to the project directory that contains AL_Operations/commandLine:
   ```
   cd ../../AL_Operations/commandLine
   ```

   In that directory you will find the Makefile for compiling the sources, the link script, a python script, and the *AL_Operations.S* program.

10. Build the .elf file:
    ```
    make clean
    make AL_Operations.elf
    ```

11. Then, in this terminal, start gdb by typing:
    ```
    riscv32-unknown-elf-gdb AL_Operations.elf
    ```

12. Inside the gdb console, type:
    ```
    target remote localhost:3333
    load
    ```

    This will connect to OpenOCD and load the *AL_Operations.elf* program into memory.

13. You should now be able to debug the program. Type the following sequence and analyse the outputs:

    i. `disas 0,20`

       This shows the assembly code from address 0 to 20 (not including address 20). See Figure 93.

```
(gdb) disas 0,20
Dump of assembler code from 0x0 to 0x14:
=> 0x00000000 <_start+0>:       li      t3,0
   0x00000004 <REPEAT+0>:       addi    t3,t3,6
   0x00000008 <REPEAT+4>:       addi    t3,t3,-1
   0x0000000c <REPEAT+8>:       andi    t3,t3,3
   0x00000010 <REPEAT+12>:      beqz    zero,0x4 <REPEAT>
End of assembler dump.
```

Figure 93. View the assembly program

ii. `i r t3`

This displays the contents of register `t3`. Alternately, you could type the longer version: `info reg t3`. See Figure 94.

```
(gdb) i r t3
t3             0x0      0
```

Figure 94. Print the value contained in register `t3`

iii. `i r pc`

This displays the contents of the program counter (`pc`). See Figure 95.

```
(gdb) i r pc
pc             0x0      0x0 <_start>
```

Figure 95. Print the value contained in register PC, that points to the first instruction

iv.
```
stepi
i r t3
stepi
i r t3
stepi
i r t3
stepi
i r t3
```

`stepi` causes the program to execute one instruction. `i r t3` then displays the contents of register `t3`. See Figure 96.

```
(gdb) stepi
0x00000004 in REPEAT ()
(gdb) i r t3
t3             0x0      0
(gdb) stepi
0x00000008 in REPEAT ()
(gdb) i r t3
t3             0x6      6
(gdb) stepi
0x0000000c in REPEAT ()
(gdb) i r t3
t3             0x5      5
(gdb) stepi
0x00000010 in REPEAT ()
(gdb) i r t3
t3             0x1      1
```

Figure 96. Execute several instructions one by one and view the `t3` register

Once you are finished debugging and exploring the program and registers using gdb, exit gdb by typing **quit** in the gdb terminal and exit OpenOCD by typing **^C** in the OpenOCD terminal.

III. Simulating a program on RVfpgaSim using Verilator

1. Open a terminal in Ubuntu

2. In a terminal window, generate the simulator binary by executing the following commands:

```
cd [RVfpgaPath]/RVfpga/verilatorSIM
make clean
make
```

   File *Vrvfpgasim* (the RVfpgaSim simulation binary), should be generated inside directory *[RVfpgaPath]/RVfpga/verilatorSIM*.

3. Go into the folder that contains the example program:

```
cd [RVfpgaPath]/RVfpga/examples/AL_Operations/commandLine
```

4. Create the hexadecimal program for simulation.

```
make clean
make AL_Operations.elf
make AL_Operations.bin
make AL_Operations.vh
```

5. Execute the simulator.

```
../../../verilatorSIM/Vrvfpgasim
+ram_init_file=AL_Operations.vh +vcd=1
```

   After a few seconds, stop the simulation by entering *^C* in the terminal. File *trace.vcd* should have been generated, and you can open it with `GTKWave`.

```
gtkwave trace.vcd
```

6. Follow the instructions provided in steps 8 to 12 of Section 7 for adding signals to the graph and analysing them.

IV. Simulating a program on Whisper

1. Open a terminal in Ubuntu

2. Go into the folder that contains the example program:

```
cd [RVfpgaPath]/RVfpga/examples/AL_Operations/commandLine
```

3. Create the disassembly program.

```
make AL_Operations.dis
```

4. Open *AL_Operations.dis* in an editor. This is what you should see:

```
<_start>:
   0: 00000e13              li    t3,0
```

```
<REPEAT>:
   4: 006e0e13                addi  t3,t3,6
   8: fffe0e13                addi  t3,t3,-1
   c: 003e7e13                andi  t3,t3,3
  10: fe000ae3                beqz  zero,4 <REPEAT>
  14: 00000013                nop
```

5. Execute the simulator in interactive mode.

```
whisper --interactive AL_Operations.elf
```

6. Debug the program.

```
whisper> step
  #1 0 00000000 00000e13 r 1c        00000000  addi     x28, x0, 0x0

whisper> peek r x28
  0x00000000

whisper> step
  #2 0 00000004 006e0e13 r 1c        00000006  addi     x28, x28, 0x6

whisper> peek r x28
  0x00000006

whisper> step
  #3 0 00000008 fffe0e13 r 1c        00000005  addi     x28, x28, -0x1

whisper> peek r x28
  0x00000005

whisper> step
  #4 0 0000000c 003e7e13 r 1c        00000001  andi     x28, x28, 0x3

whisper> peek r x28
  0x00000001
```

Once you are finished debugging and exploring the program and registers using whisper, exit by typing **quit** in the terminal.

Appendix B: Installing drivers in Windows to use PlatformIO

To download the Zadig executable, browse to the following website (see Figure 97):

https://zadig.akeo.ie/

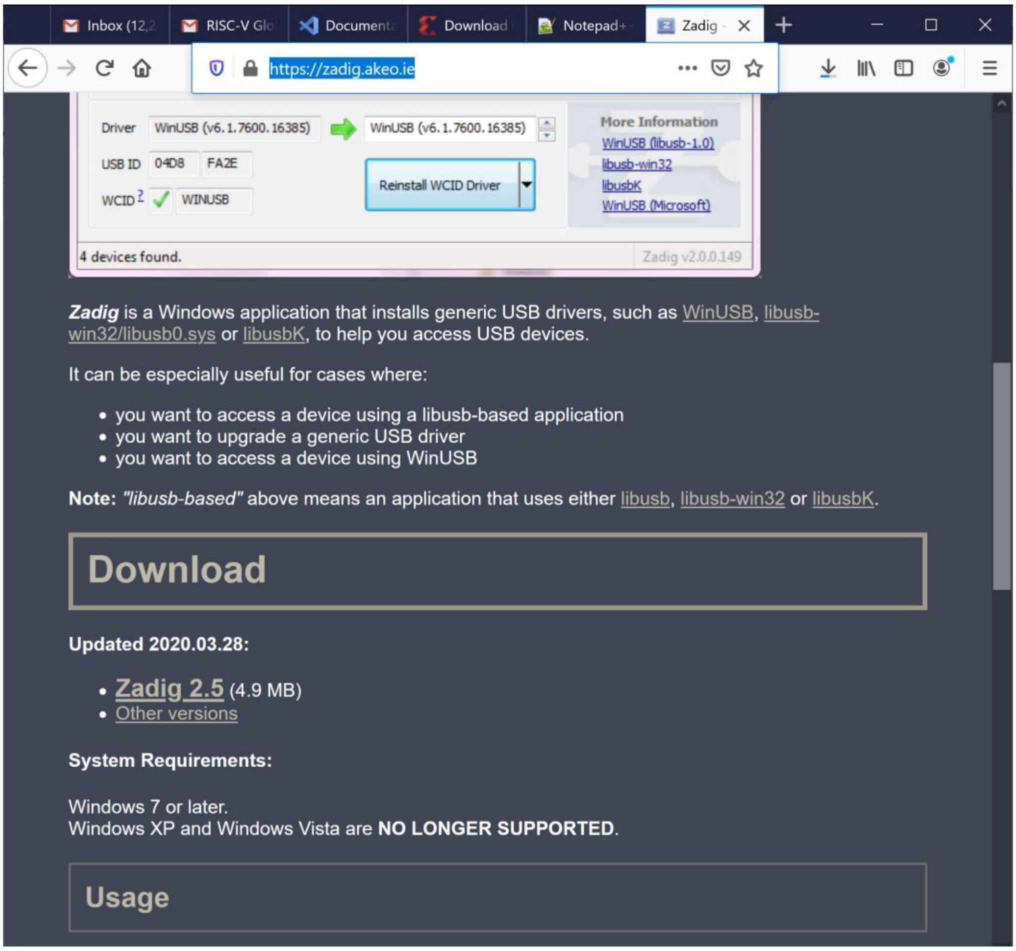


Figure 97. Install Nexys A7 board driver used by PlatformIO

Click on Zadig 2.5 and save the executable. Then run it (zadig-2.5.exe), which is located where you downloaded it. You can also type zadig into the Start menu to find it. You will probably be asked if you want to allow Zadig to make changes to your computer and if you will let it check for updates. Click Yes both times.

Connect the Nexys A7 Board to your computer and switch it on. In Zadig, click on Options → List All Devices (see Figure 98).

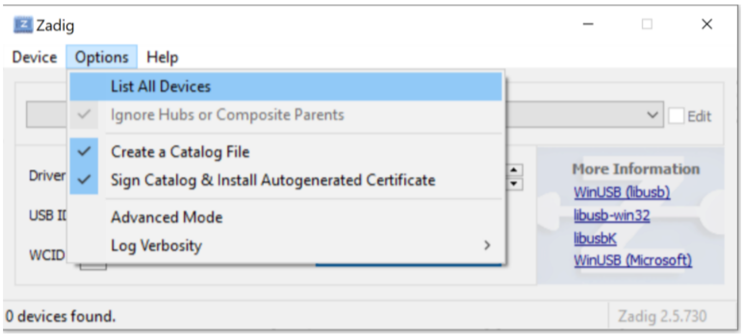


Figure 98. List all devices in Zadig

If you click on the drop-down menu, you will see Digilent USB Device (Interface 0) and Digilent USB Device (Interface 1) listed. You will install new drivers for only Digilent USB Device (Interface 0) (see Figure 99).

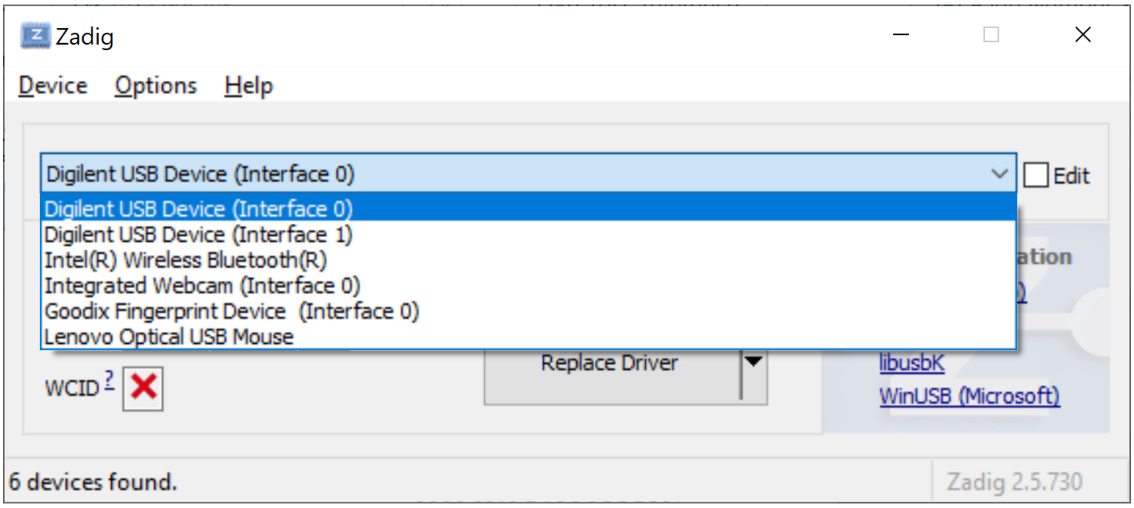


Figure 99. Install WinUSB driver for Digilent USB Device (Interface 0)

You will now replace the FTDI driver with the WinUSB driver, as shown in Figure 100. Click on Replace Driver (or Install Driver) for Digilent USB Device (Interface 0). You are installing the driver for the Nexys A7 board or, if you previously installed Vivado, you are replacing the FTDI driver used by Vivado with the WinUSB driver used by PlatformIO.

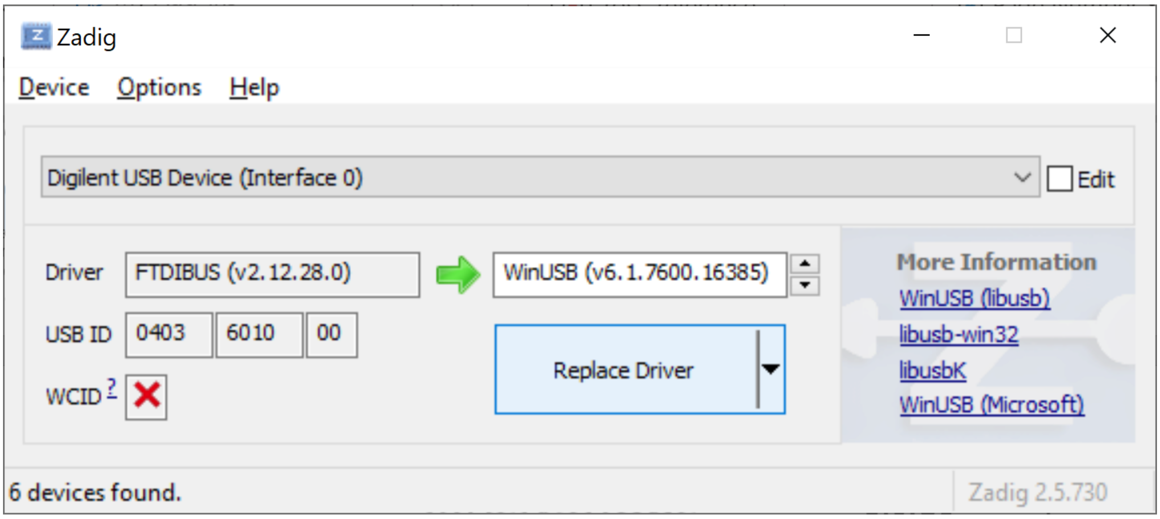


Figure 100. Replace driver for Nexys A7 board

After some time, typically several minutes, Zadig will indicate the driver was installed correctly. Click Close and then close the Zadig window.

Next time you use PlatformIO you do not need to re-install the driver. However, note that **this driver is not compatible with Vivado in Windows**. So you can no longer use Vivado to download bitfiles to the FPGA board. If you wanted to use Vivado to download bitfiles (not recommended) you would need to revert the driver back to the original driver installed with Vivado, as described in Appendix E.

Appendix C: Installing Verilator and GTKWave in Windows

In this section, we explain how to install Verilator and GTKWave in Windows 10. In Windows, you must use Cygwin to install Verilator, so we first explain how to install this programming/runtime environment.

Cygwin installation:

As described on its webpage (https://www.cygwin.com), Cygwin consists of GNU and Open Source tools which provide functionality on Windows similar to that of a Linux distribution. Follow the next steps to install Cygwin on Windows 10.

1. Navigate to the installation webpage (https://cygwin.com/install.html) and download the installation file, called *setup-x86_64.exe* (Figure 101).

Figure 101. Cygwin installation webpage

2. Execute the setup file in your machine by double-clicking on it (Figure 102). Click **Next** several times, maintaining the default options. The installer will ask you to **Choose a Download Site** (Figure 103), you can choose any one of them.

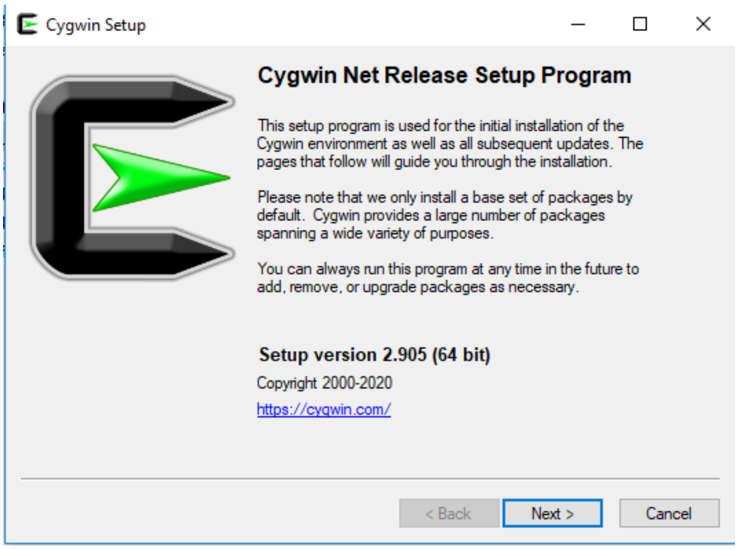


Figure 102. Cygwin installation window

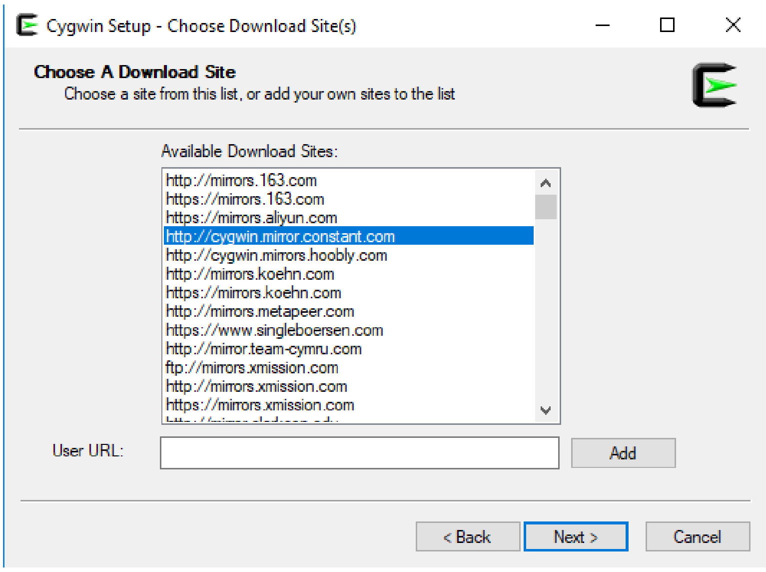


Figure 103. Choose Download Site

3. After several steps, you will reach the **Select Packages** window (Figure 104). Select the **Full** view, as shown in Figure 104.

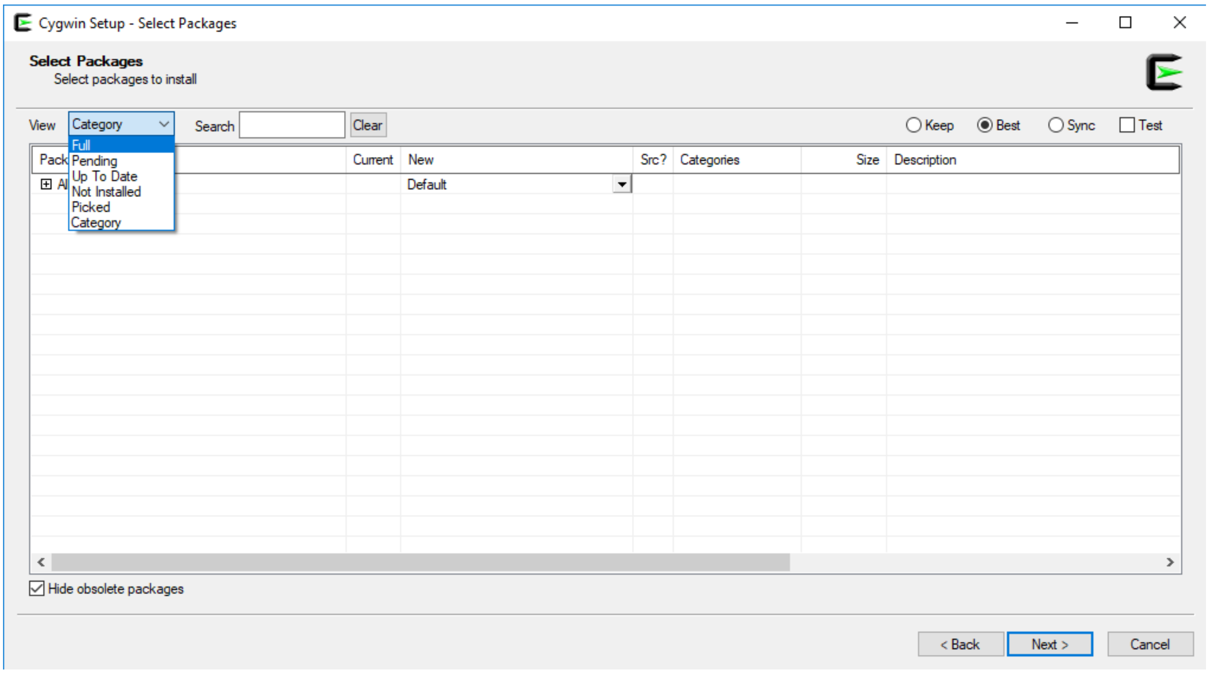


Figure 104. Select Packages window

4. The complete list of packages that you can install will appear (Figure 105). In the **Search** box, select the specific packages that you want to install.

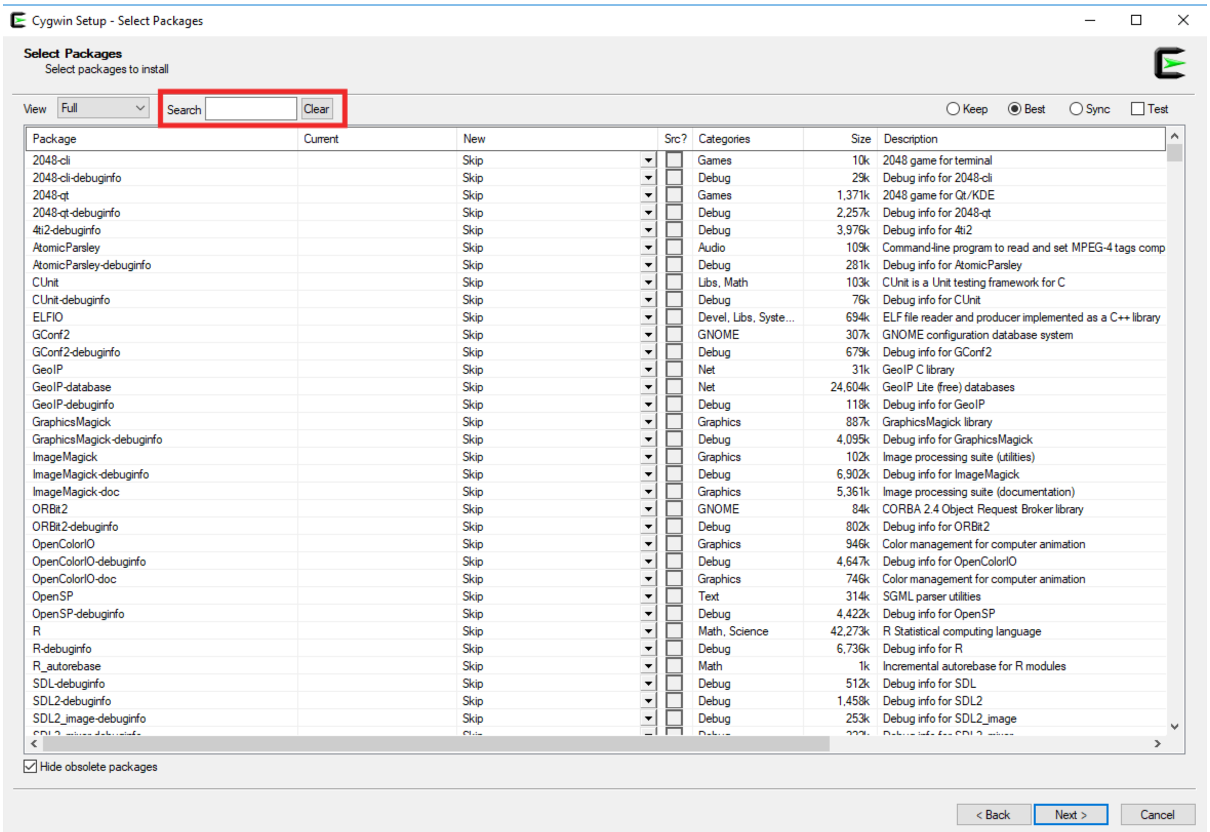


Figure 105. Select Packages window – Full view

To be able to compile Verilator and generate a new simulator binary, you need to install the following packages:

- `git`
- `make`
- `autoconf`
- `gcc-core`
- `gcc-g++`
- `flex`
- `bison`
- `perl`
- `libargp-devel`

Include at least these packages in your Cygwin installation. Select them one-by-one following the steps below (we only show the detailed steps for the first package in the list, `git`; the process is the same for the other packages):

- Look for the `git` package in the **Search** box (Figure 106).

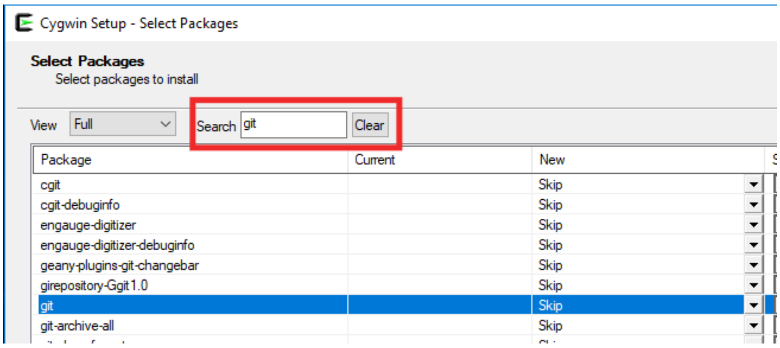


Figure 106. Look for the `git` package

- Select the most up-to-date version in the dropdown menu **and** tick the box (Figure 107).

Figure 107. Select the most up-to-date version and tick the box

- Do the same for the remaining packages in the above list. In most cases, you can use the most up-to-date version, but for packages `gcc-core` and `gcc-g++` you should use version 10.2.0, since the most up-to-date version, which at the moment of writing this document was 11.2.0, presents some conflicts with Verilator.

5. Once you have selected the nine packages, click **Next** in the subsequent windows to include these packages in your Cygwin installation (the installation process, see Figure 108, may take several minutes) and finalize the installation by clicking Finish (Figure 109).

Imagination university programme

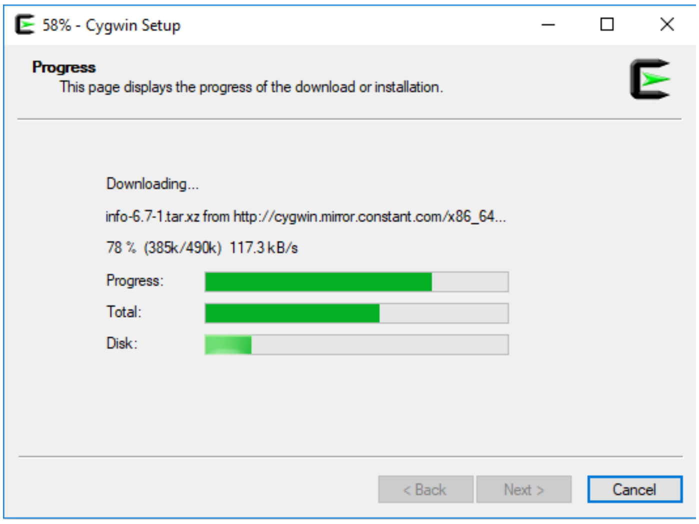


Figure 108. Cygwin setup

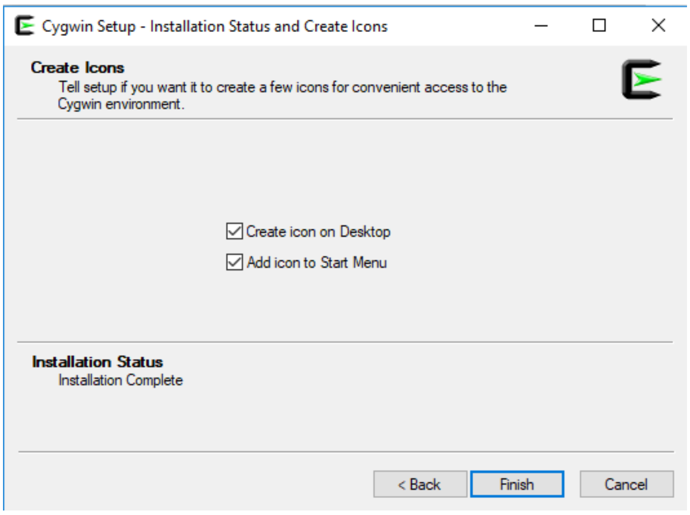


Figure 109. Finish the installation

6. If you need to add a package to your Cygwin installation, repeat steps 2-5 for that package.

Verilator installation:

Follow the next steps to install Verilator on Windows 10.

1. Open the Cygwin terminal (Figure 110), available on your Windows Desktop or from the Start menu.

Figure 110. Cygwin terminal

2. Build and install Verilator by following these steps. This may take some time (even hours), depending on the speed of your computer:

   - `git clone https://git.veripool.org/git/verilator`
   - `cd verilator`
   - `git pull`
   - `git checkout v4.106`
   - `autoconf`
   - `./configure`
   - `make`
   - `make install`

   Note that Verilator v4.106 can also be installed by downloading it directly from GitHub: https://github.com/verilator/verilator/releases/tag/v4.106. In this case, the four initial steps would simply be avoided.

GTKWave installation:

GTKWave can be downloaded as a precompiled package from https://sourceforge.net/projects/gtkwave/files/. Look for the most recent Windows package (at the time this document was written, it was called ***gtkwave-3.3.100-bin-win64***), and download and unzip (uncompress) it. You can find an executable file called *gtkwave* inside folder *bin*, which you can execute and use in your Windows machine.

Appendix D: Installing Verilator and GTKWave in macOS

In this section, we explain how to install Verilator and GTKWave in a macOS. The instructions are tested with macOS Catalina 10.15.6 but are expected to work in other versions of the OS. Homebrew (https://brew.sh/) package manager is used for the installation. Similar steps may be found for MacPorts, the other widely used package manager in macOS (https://www.macports.org/).

gcc installation:

In order to build a new simulator using Verilator, a compiler toolchain needs to be installed in the system. There are many ways to install a valid compiler toolchain. We cite two of them below:

1. Install the XCode Command Line Tools. Note that this will install LLVM, but a *gcc* command will be anyhow available after installation. To do so, type the following command in a Terminal window:

   - `xcode-select -install`

2. Install *gcc* using Homebrew. Use the following recipe:

   - `brew install gcc@9`

Verilator installation:

Installing Verilator with Homebrew is as simple as typing the following command in an open Terminal:

- `brew install verilator`

gtkwave installation:

Once again, we will use Homebrew to install *gtkwave*. But this time we need to use *cask* because it is a GUI macOS application. Type the following commands in an open Terminal:

- `brew tap homebrew/cask`
- `brew cask install xquartz`
- `brew cask install gtkwave`

After the installation, an icon for *gtkwave.app* should appear in the Application folder. In order to use it from the command line, you may need to install Perl's Switch module:

- `cpan install Switch`

Appendix E: Using Vivado to Download RVfpgaNexys onto an FPGA

Follow the next steps for programming the FPGA with RVfpgaNexys using Vivado:

WINDOWS: Before following the next steps, in Windows you need to revert the drivers back to the ones used by Vivado as explained at the end of this Appendix (Appendix E).

a. Connect the Nexys A7 board to your computer.
b. Turn on the Nexys A7 board using the switch at the top left.
c. Open Vivado 2019.2.
d. Open the *Hardware Manager* available in Vivado and highlighted in Figure 111.

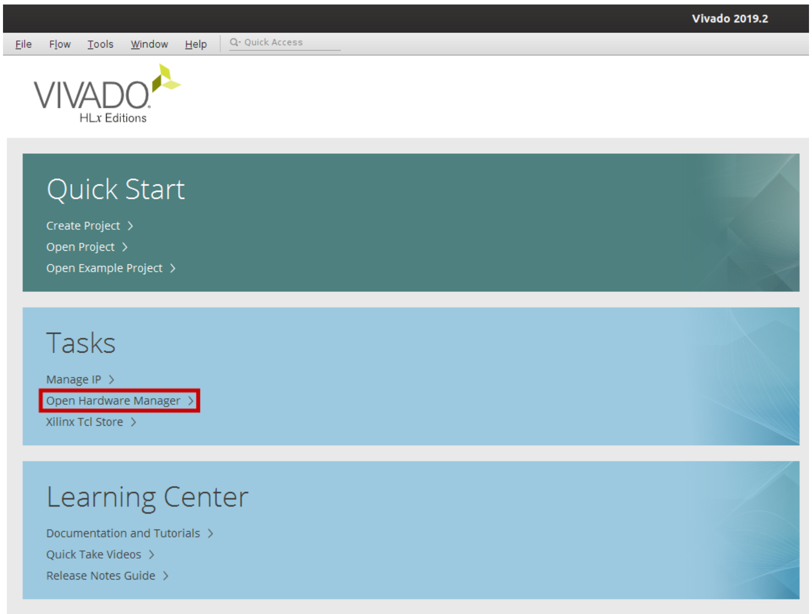


Figure 111. Open Hardware Manager

e. The Hardware Manager opens and informs you that no hardware target is open. Open the target by clicking on *Open target – Auto connect* (Figure 112).

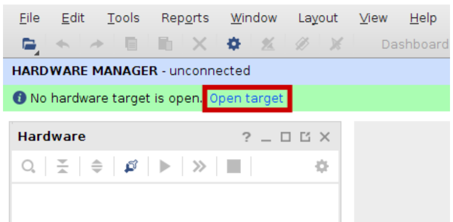


Figure 112. Open target

f. Select *Program device* as shown in Figure 113. You will now load RVfpgaNexys onto the FPGA. In the new window, select the *Bitstream file* from *[RVfpgaPath]/RVfpga/src/rvfpganexys.bit*. Click *Program*.

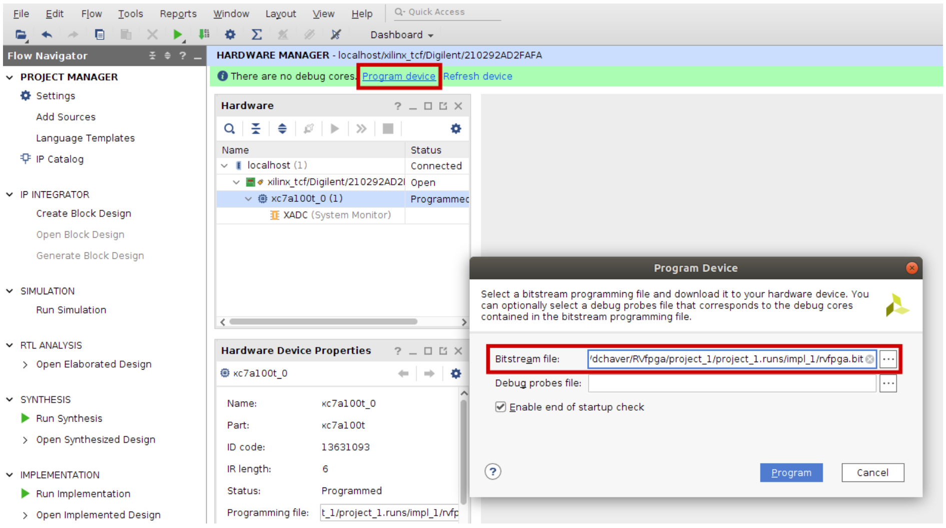


Figure 113. Program device

g. After a few seconds, the FPGA will be programmed with RVfpgaNexys, the **SweRVolfX SoC targeted to an FPGA** (see Figure 25).
h. Finally, **close the Hardware Manager** by clicking on the X button on the top right of the Hardware Manager pane in Vivado (Figure 114), so that Vivado releases the board.

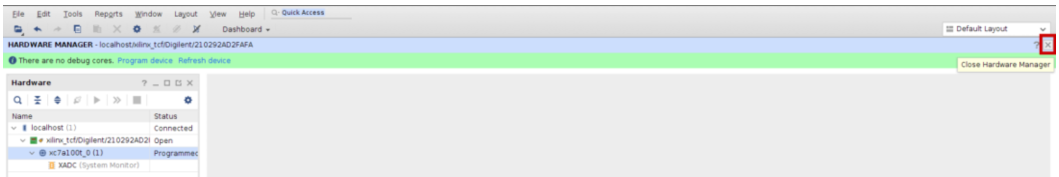


Figure 114. Close the Hardware Manager

How to revert the drivers back to the ones used by Vivado in Windows

Unfortunately, in Windows, the drivers for the Nexys A7 FPGA board differ for Vivado and PlatformIO. It is **strongly recommended that you use PlatformIO to program the FPGA, as explained in Section 5.A of this GSG**. However, if you want to use Vivado to download bitfiles, you must revert the drivers that you installed in Appendix B to the Vivado (FTDI) drivers for the Nexys A7 FPGA board. To do so, open the Device Manager by clicking on the Start menu, typing device manager in the search box, and clicking on Device Manager (see Figure 115).

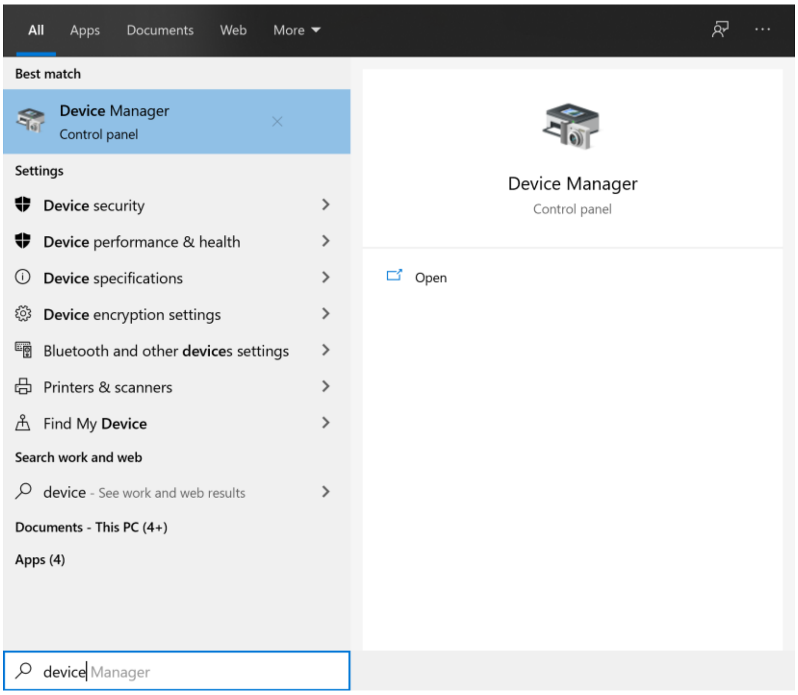


Figure 115. Open Device Manager

Next, expand Universal Serial Bus Devices, right-click on Digilent USB Device, and select Properties (see Figure 116).

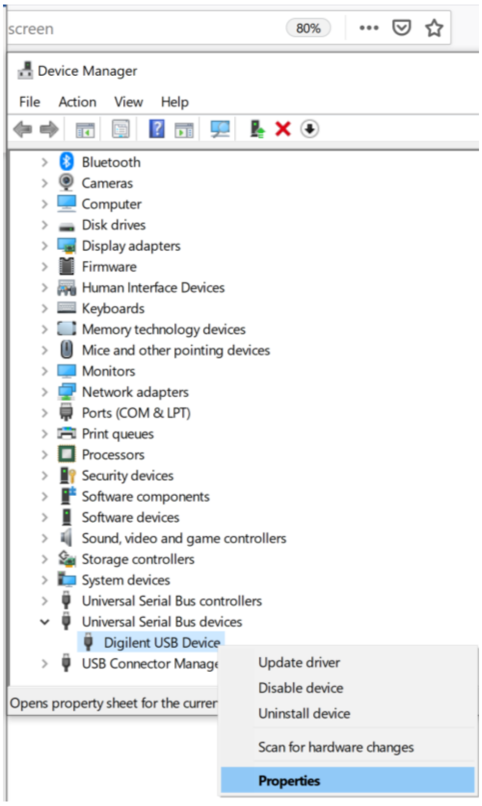


Figure 116. Open driver properties for Digilent's Nexys A7 FPGA board

In the Properties window, click on the Driver tab and select Roll Back Driver (see Figure 117).

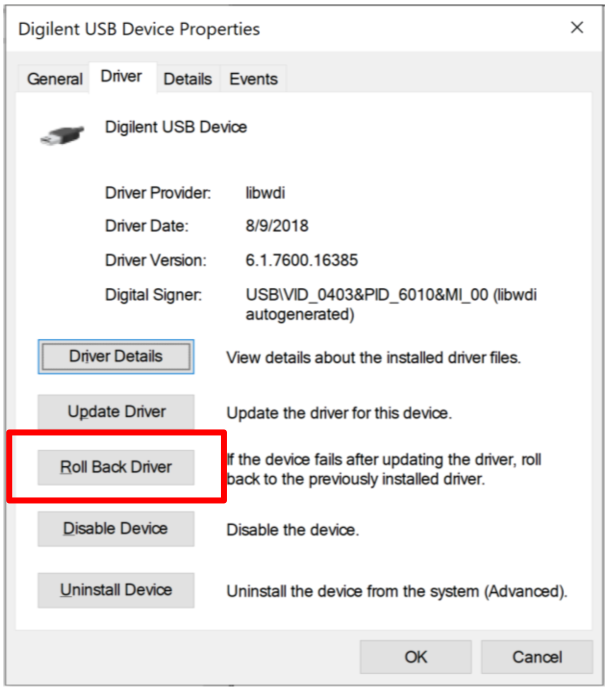


Figure 117. Roll back driver

A window will pop up asking why you are rolling back the driver. Select a reason and click Yes (see Figure 118).

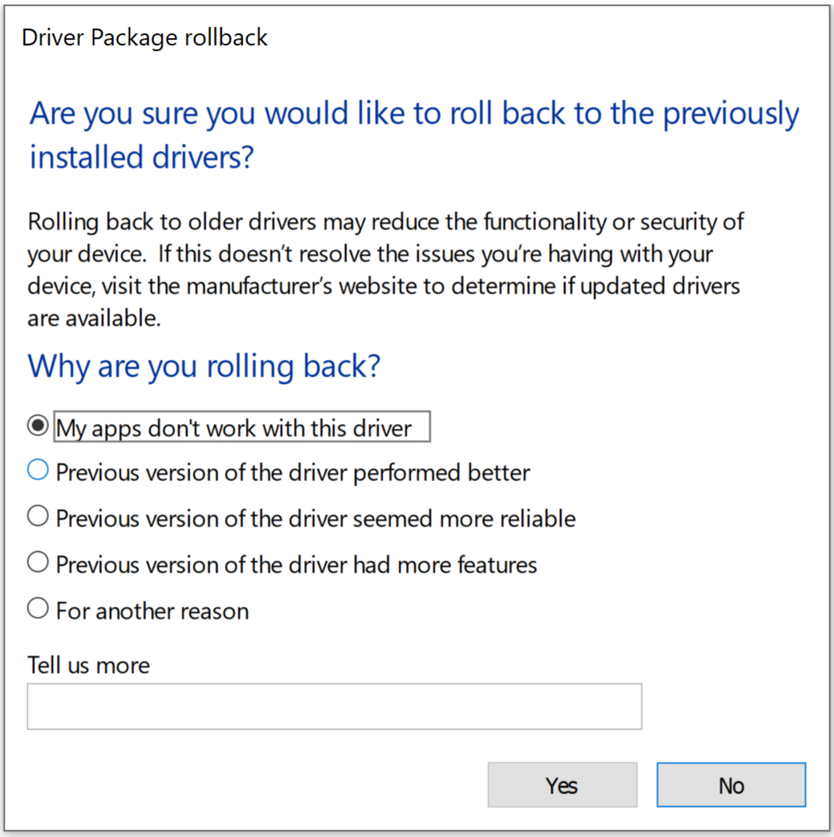


Figure 118. Confirm roll back

After the driver reverts back to the previous driver, the Driver Provider should be listed as FTDI (see Figure 119).

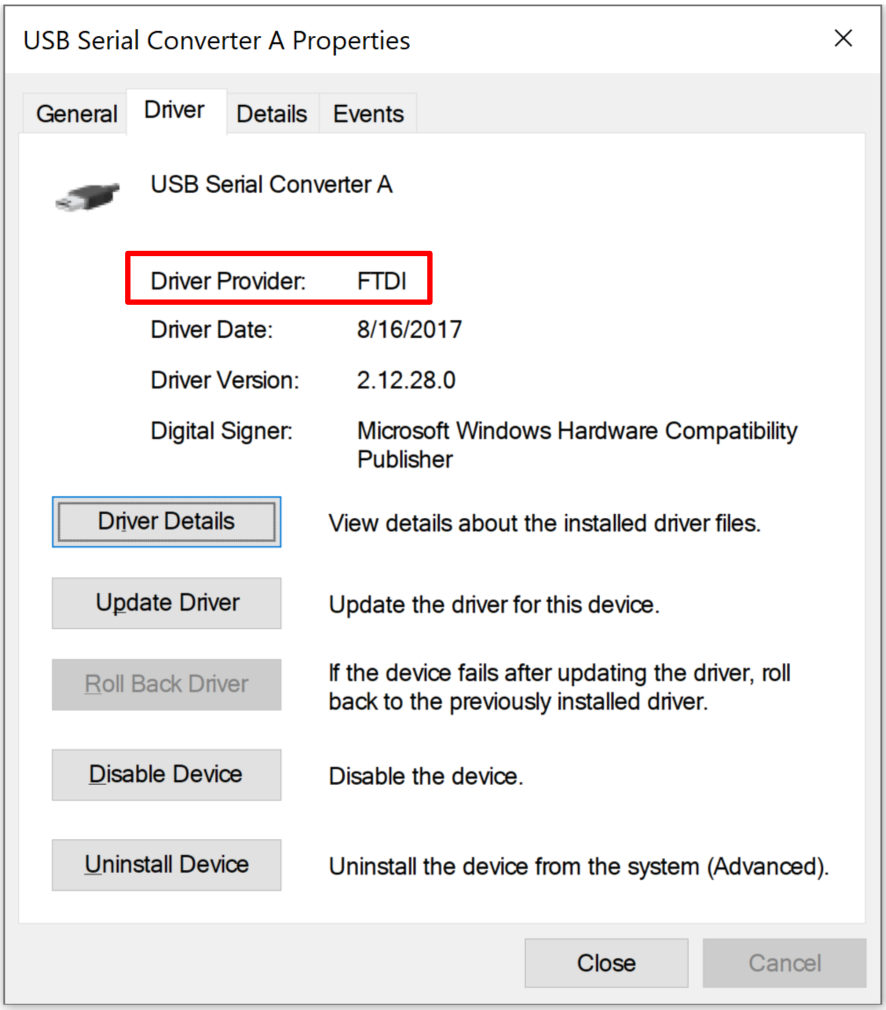


Figure 119. FTDI driver shown as driver provided

Now you can load bitfiles onto the FPGA board using Vivado. However, you will still need to use Zadig to replace the Nexys A7 board's driver, so that PlatformIO can download the program onto RVfpgaNexys. Thus, it is recommended that you use PlatformIO to download bitfiles as well (instead of using Vivado) – this will keep you from continually having to swap drivers.

Appendix F: Using RVfpga in an industrial IoT application

In July 2020, Daniel León González, a master's student at the University Complutense of Madrid completed his master's thesis titled "FPGA implementation of an ad-hoc RISC-V system-on-chip for industrial IoT". This work shows the use of RVfpga in a real industrial IoT application. We provide the project abstract below, and the complete thesis is available at: https://eprints.ucm.es/62106/1/DANIEL_LEON_GONZALEZ_DL_-_FPGA_Implementation_of_an_ad-hoc_RISC-V_SoC_for_Industrial_IoT__Graded__4286351_962908330.pdf.

FPGA implementation of an ad-hoc RISC-V system-on-chip for industrial IoT

Abstract: Node devices for IoT need to be energy efficient and cost effective, but they do not require a high computing power in a large number of scenarios. This changes substantially in an Industrial IoT environment, where massive sensor utilization and the fast pace of events require more processing power. A custom developed node, using an efficient processor and a high performance and feature-full operating system, may balance these requirements and offer an optimal solution. This project addresses the hardware implementation, using an Artix-7 FPGA, of a prototype IoT node based on the RISC-V processor architecture. The project presents the implemented custom SoC and the development of the necessary Zephyr OS drivers to support a proof-of-concept application, which is deployed in a star network around a custom border router. End-to-end messages can be sent and received between the node and the ThingSpeak cloud platform. This thesis includes an analysis of the existing RISC-V processor implementations, a description of the required elements, and a detailed guide to environment configuration and project design.