THE IMAGINATION UNIVERSITY PROGRAMME

RVfpgaEL2
Getting Started Guide

Acknowledgements

AUTHORS
Prof. Daniel Chaver
Prof. Sarah Harris

ADVISER
Prof. David Patterson

CONTRIBUTORS
Robert Owen
Olof Kindgren
Prof. Luis Piñuel
Prof. Roy Kravitz
Chris Owen
Zubair Kakakhel
M. Hamza Liaqat
Prof. Alexander Grinshpun

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
Ivan Kravets
Valerii Koval
Ted Marena
Prof. Chathuranga Hettiarachchi
Jingyang Liu

Sponsors and Supporters

Table of Contents

1. Introduction

RISC-V FPGA EL2, also written RVfpgaEL2, 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 simulators, and then using and expanding it to learn about computer architecture, digital design, embedded systems, and programming.

This RVfpgaEL2 Getting Started Guide has the following sections, as described briefly below:

- **Background and Overview**
  - **RISC-V Architecture** (Section 2)
  - **The RVfpgaEL2 System** (Section 3)
- **Installing Software Tools** (Section 4)
- **Executing the RVfpgaEL2 System** on the board (Section 5)
- **Simulating the RVfpgaEL2 System**
  - **Using RVfpgaEL2-Trace** (Section 6)
  - **Using RVfpgaEL2-ViDBo** (Section 7)
  - **Using RVfpgaEL2-Pipeline** (Section 8)
  - **Using RVfpgaEL2-Whisper** (Section 9)
- **Appendices**
  - **Installing JTAG drivers in Windows** (Appendix A)
  - **Installing tools used by the simulators in Windows** (Appendix B)
  - **Installing tools used by the simulators in MacOS** (BETA) (Appendix C)
  - **Using the simulators in Windows** (Appendix D)
  - **Using RVfpgaEL2 with PlatformIO** (Appendix E)

Section 2 gives a brief introduction to the RISC-V computer architecture. Section 3 describes the RVfpgaEL2 System (Section 3.A – 3.C) and the organization of the Verilog files that make up the system (Section 3.D). The RVfpgaEL2 System is based on the VeeRwolf SoC (https://github.com/chipsalliance/VeeRwolf) which, in turn, uses open-source RISC-V VeeR EL2 Core (https://github.com/chipsalliance/Cores-VeeR-EL2). Figure 1 and Table 1 illustrate the hierarchical organization of the RVfpgaEL2 System, from the VeeR EL2 Core up to the hardware (RVfpgaEL2-Basys3) and simulators (RVfpgaEL2-Trace, RVfpgaEL2-ViDBo, RVfpgaEL2-Pipeline, and Whisper).

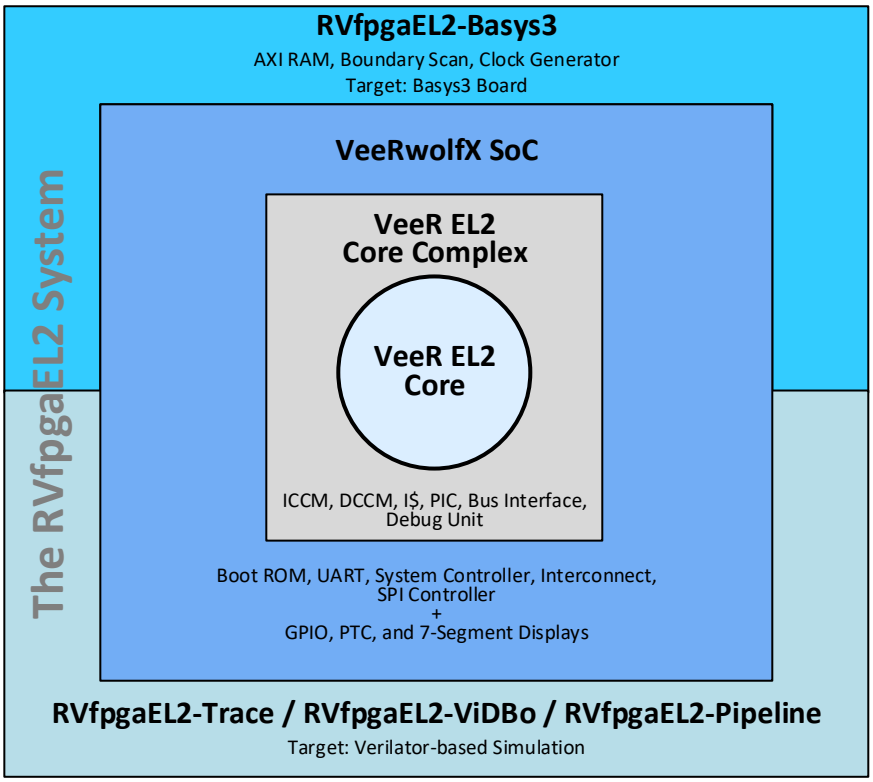


Figure 1. RVfpga System Hierarchy

Table 1. RVfpga System Hierarchy

Name	Description
VeeR EL2 Core	Open-source commercial RISC-V core (https://github.com/chipsalliance/Cores-VeeR-EL2).
VeeR EL2 Core Complex	VeeR EL2 core with added memory (ICCM, DCCM, and instruction cache), programmable interrupt controller (PIC), bus interfaces, and debug unit (https://github.com/chipsalliance/Cores-VeeR-EL2).
VeeRwolfX (Extended VeeRwolf)	The System on Chip that we use in the RVfpgaEL2 course. It is an extension of VeeRwolf. VeeRwolf (https://github.com/chipsalliance/VeeRwolf): An open-source SoC built around the VeeR EL2 Core Complex. It adds a boot ROM, UART interface, system controller, SPI controller and interconnect (AXI Interconnect, Wishbone Interconnect, and AXI-to-Wishbone bridge). VeeRwolfX: It adds new peripherals to VeeRwolf: a GPIO, a PTC, and a controller for the 4-digit 7-Segment Displays.
RVfpgaEL2-Basys3	The VeeRwolfX SoC targeted to the Basys3 board and its peripherals. It adds an AXI RAM, BSCAN logic (for the JTAG interface) and clock generator.
RVfpgaEL2-Trace, RVfpgaEL2-ViDBo, RVfpgaEL2-Pipeline	The VeeRwolfX SoC with a testbench wrapper and AXI memory intended for simulation in Verilator.

The remaining sections show how to use the RVfpga System in both hardware (RVfpgaEL2-Basys3) and simulation (RVfpgaEL2-Trace, RVfpgaEL2-ViDBo, RVfpgaEL2-Pipeline, and Whisper). Section 4 shows how to install the software tools needed to use RVfpgaEL2. Section 5 shows how to use Catapult Studio, an SDK (software development kit), to both

download RVfpgaEL2-Basys3 onto the Basys 3 FPGA board (Section 5.A) and download and run several example programs on it (Section 5.B-5.H). Sections 6 through 9 show how to simulate the RVfpga System using Verilator (Sections 6-8), an open-source HDL simulator, and using Whisper (Section 9), a RISC-V Instruction Set Simulator (ISS).

The appendices show how to install needed drivers and software, and how to use the simulators on Windows and macOS machines. The final appendix, Appendix E, also shows how to use RVfpgaEL2 with PlatformIO (https://platformio.org/).

Table 2 lists the software and hardware used in this RVfpgaEL2 course. The software is freely available, and the hardware (FPGA boards) is optional. All of the tutorials in this guide as well as the labs can be completed in simulation only.

This guide shows how to install and use these tools and hardware on the Ubuntu 22.04 operating system (OS), which is the recommended OS. Other operating systems (such as Windows or macOS), follow the same or similar steps. When instructions differ, we insert specific instructions for **Windows** and **macOS** using this highlighting. Almost everything works correctly in **Windows** but only some things work in **macOS**.

We provide a Ubuntu 22.04 Virtual Machine (user and password: "rvfpga") with everything installed on it, so that you can easily use all tools in a Linux environment (independently of the OS that you use in your computer) out-of-the-box. This Ubuntu 22.04 Virtual Machine was created in Virtual Box 6.1 and exported to a .ova file that uses Open Virtualization Format 1.0. It can be executed in typical virtualization software such as VirtualBox or VMWare. It is configured with 2 CPUs, 8GB of memory and more than 100GB of dynamically allocated disk. If you use the Virtual Machine for completing the course, you don't need to perform the installation of the RVfpgaEL2 tools. In the Virtual Machine, you can find the RVfpga_EL2 folder in the home directory, so you don't need to download it either.

Note: if you do not have access to one of the FPGA boards (Nexys A7, Boolean Board, or Basys 3), the labs can all be completed in simulation using Whisper, an instruction set simulator (ISS) and Verilator, an open-source HDL simulator. In this case, you do not need to install all the tools, as we explain in Section 4.

Table 2. Software and Hardware for RVfpga

Software		
Name	**Website**	**Cost**
Vivado 2022.2 WebPACK*	https://www.xilinx.com/support/download/index.html/content/xilinx/en/downloadNav/vivado-design-tools/2022-2.html	free
Catapult SDK	https://developer.imaginationtech.com	free
Verilator (an HDL simulator)	https://github.com/verilator/verilator	free
GTKWave	http://gtkwave.sourceforge.net/	free
LibWebSockets	https://libwebsockets.org/	free
Whisper (RISC-V Instruction Set Simulator) **	https://github.com/chipsalliance/VeeR-ISS	free
Optional Hardware		
Name	**Website**	**Cost**

Nexys A7 FPGA Board ***	https://digilent.com/shop/nexys-a7-fpga-trainer-board-recommended-for-ece-curriculum/	$349 (academic price at https://digilent.com/shop/academic/academic-price-list/: $261.75)
Boolean Board	https://www.realdigital.org/hardware/boolean	$105 (academic: $87 or $74)
Basys 3 FPGA Board	https://digilent.com/shop/basys-3-artix-7-fpga-trainer-board-recommended-for-introductory-users/	$165 (academic: $124)
RISC-V Core and System-on-Chip (SoC)**		
Name	**Website**	**Cost**
VeeR EL2 Core **	https://github.com/chipsalliance/Cores-VeeR-EL2	free
VeeRwolf **	https://github.com/chipsalliance/VeeRwolf	free

* Vivado WebPACK is optional and is only needed when using the optional hardware.
** Already provided with the RVfpgaEL2 download from Imagination Technologies
*** The course also works on Digilent's Nexys4 DDR FPGA board.

Expected Prior Knowledge:

Before completing this RVfpgaEL2 course, which includes this RVfpgaEL2 Getting Started Guide and RVfpgaEL2 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. 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 Imagination Technologies, 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	2.0	Ratified
RV64E	2.0	Ratified
RV128I	*1.7*	*Draft*

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

Extension	Version	Status
M	2.0	Ratified
A	2.1	Ratified
F	2.2	Ratified
D	2.2	Ratified
Q	2.2	Ratified
C	2.0	Ratified
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	*1.0*	*Frozen*
Zicsr	2.0	Ratified
Zifencei	2.0	Ratified
Zihintpause	2.0	Ratified
Zihintntl	*0.3*	*Frozen*
Zam	*0.1*	*Draft*
Zfh	1.0	Ratified
Zfhmin	1.0	Ratified
Zfinx	1.0	Ratified
Zdinx	1.0	Ratified
Zhinx	1.0	Ratified
Zhinxmin	1.0	Ratified
Zmmul	1.0	Ratified
Ztso	1.0	Ratified

The letter G, that denotes “general”, indicates 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.

3. RVfpgaEL2 System Overview

In this section we describe the entire RVfpgaEL2 system from the core up to the FPGA board interface. Figure 2 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.

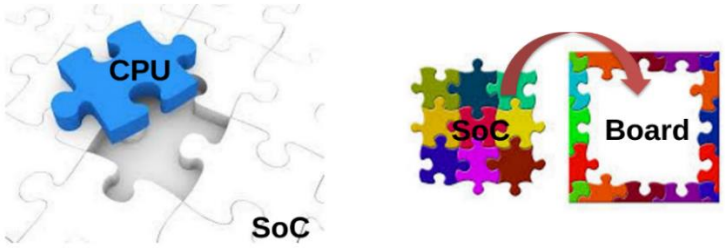


Figure 2. Embedded system organization

Figure 1 and Table 1 show the hierarchical organization of our system, from the VeeR EL2 Core up to RVfpgaEL2-Basys3 and RVfpgaEL2-Trace / RVfpgaEL2-ViDBo / RVfpgaEL2-Pipeline. In the following sections, we start by describing the processor core (**VeeR EL2 Core**), which executes the RISC-V instructions; then, in Section B, we describe the **VeeRwolfX SoC**, which integrates the system's hardware components (core, memory, and input/output), and the extensions performed for using it within RVfpgaEL2; in Section C we describe the VeeRwolfX SoC implemented on the Basys3 FPGA board (**RVfpgaEL2-Basys3**) and also describe the VeeRwolfX SoC used in simulation (**RVfpgaEL2-Trace, RVfpgaEL2-ViDBo, and RVfpgaEL2-Pipeline**). Finally, we explain the file structure of the whole RVfpgaEL2 System in Section D.

A. VeeR EL2 Core and VeeR EL2 Core Complex

Western Digital developed three RISC-V cores over the past few years: VeeR EH1 (formerly called SweRV), VeeR EH2 (or SweRV EH2 previously), and VeeR EL2 (the core we use in this course, formerly called SweRV EL2). Each core has an Apache 2.0 license. The VeeR EH1 Core (https://github.com/chipsalliance/Cores-VeeR-EH1) is a 32-bit, 2-way superscalar, 9-stage pipeline core. The VeeR EH2 Core (https://github.com/chipsalliance/Cores-VeeR-EH2) builds on and expands the EH1 Core to add dual threaded capability for additional performance. The VeeR EL2 Core (https://github.com/chipsalliance/Cores-VeeR-EL2) is a smaller core with moderate performance (see Figure 3) is a scalar 4-stage core with one execution pipeline, one load/store pipeline, one multiplier pipeline, and one out-of-pipeline divider. There are two stall points in the pipeline: 'Fetch' and 'Decode'. The figure also shows how VeeR EL2's logic stages have been shifted up with respect to VeeR EH1 and merged into 4 stages named Fetch (F), Decode (D), Execute/Memory (X/M), and Retire (R). Also shown is additional logic such as a new branch adder in the D stage, not included in VeeR EH1. The branch mispredict penalty is either 1 or 2 cycles in VeeR EL2. The merged F stage performs the program counter calculation and the I-cache/ICCM memory access in parallel. The load pipeline has been moved up so that the DC1 memory address generation (AGU) logic is now combined with align and decode logic to enable a DCCM memory access to start at the beginning of the M stage. The design supports a load-to-use of 1 cycle for smaller memories and a load-to-use of 2 cycles for larger memories. For 1-cycle load-to-use, the memory is accessed and the load data aligned and formatted for the register file and forwarding paths, all in the single-cycle M stage. For 2-cycle load-to-use, almost the entire M stage is allocated to the memory access, and the DC3/DC4 logic combined into the R stage is used to perform the load align and formatting for the register file and forwarding paths.

EX3 and EX4/WB are combined into the R stage and primarily used for commit and writeback to update the architectural registers.

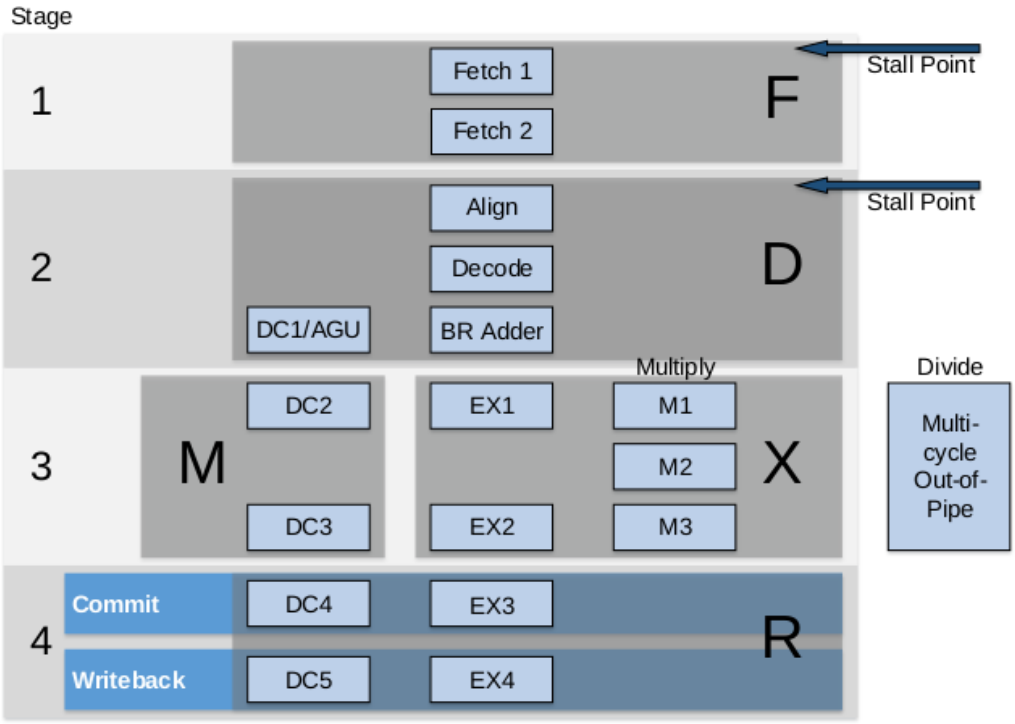


Figure 3. VeeR EL2 core microarchitecture

(figure from https://github.com/chipsalliance/Cores-VeeR-EL2/blob/main/docs/RISC-V_VeeR_EL2_PRM.pdf)

ChipsAlliance also provides an extension to the VeeR EL2 Core called the **VeeR EL2 Core Complex** (see Figure 4), which adds the following elements to the EL2 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.

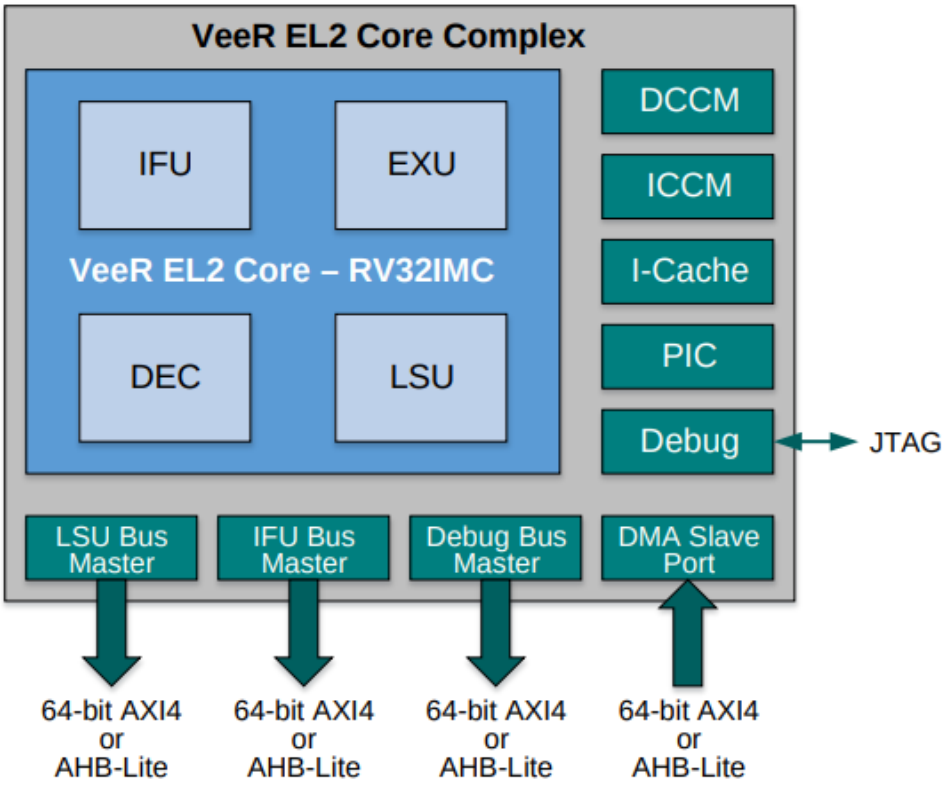


Figure 4. VeeR EL2 Core Complex

(figure from https://github.com/chipsalliance/Cores-VeeR-EL2/blob/main/docs/RISC-V_VeeR_EL2_PRM.pdf)

B. VeeRwolfX SoC

The System on Chip (SoC) used in this RVfpgaEL2 package, called VeeRwolfX and illustrated in Figure 5, is based on VeeRwolf (https://github.com/chipsalliance/VeeRwolf), which is built on top of the VeeR EL2 Core Complex. In addition to the Core Complex (Figure 4), the VeeRwolf SoC also includes a Boot ROM, a UART, an SPI controller and a System Controller (Figure 5 shows these elements in white). Given that the core uses an AXI bus and the peripherals use a Wishbone bus, the SoC also has an AXI-Wishbone Bridge.

In RVfpgaEL2 we extend the VeeRwolf SoC with some more functionality, such as a GPIO (General Purpose Input/Output) controller, a PTC (PWM/Timer/Counter) module and a controller for interfacing with 4-digit 7-Segment Displays. Figure 5 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 **VeeRwolfX** (the X stands for eXtended).

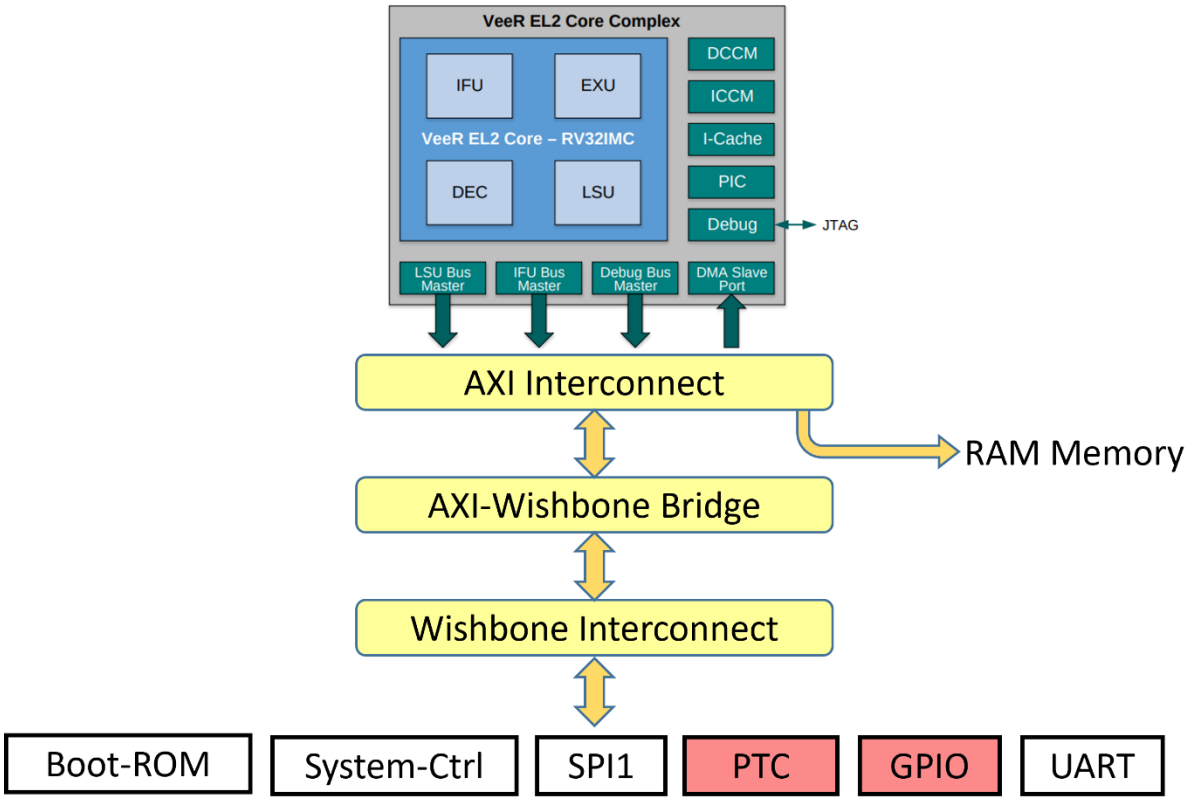


Figure 5. VeeRwolfX (VeeRwolf eXtended with new peripherals) System on Chip

Table 5 shows the memory-mapped addresses of the peripherals connected to the core via the Wishbone interconnect.

Table 5. Memory-mapped addresses of Extended VeeRwolfX SoC peripherals

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

i. Input/Output

The VeeRwolfX 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 VeeRwolfX 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 the RVfpgaEL2 labs.

- **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/VeeRwolf you can find the complete memory map. We have modified this module as follows:
    - We have included a new controller for communicating with the 4-digit 7-Segment Displays available on the Basys3 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 VeeRwolf and mapped in addresses 0x80001010–0x8000101F. Note that we have added a more complete GPIO controller as described below.

- **SPI:** an open-source SPI controllers (obtained from https://opencores.org/projects/simple_spi and named SPI1) is implemented in VeeRwolfX. Their exposed registers (SPI_SPCR, SPI_SPSR, SPI_SPDR, SPI_SPER, SPI_SPSS) are mapped between addresses 0x80001040 and 0x8000107F (for SPI1). The addresses for each of the registers can be found in the document of the controller (provided in folder *[RVfpgaBasysPath]/src/VeeRwolf/Peripherals/spi/docs*).

- **PTC:** We use the timer module from https://opencores.org/projects/ptc. Its registers are mapped in the address range 0x80001200 to 0x800012FF. The addresses for each of the registers can be found in the document of the controller (provided in folder *[RVfpgaBasysPath]/src/VeeRwolf/Peripherals/ptc/docs*) and in Lab 8.

- **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. The addresses for each of the registers can be found in the document of the controller (provided in folder *[RVfpgaBasysPath]/src/VeeRwolf/Peripherals/gpio/docs*) and in Lab 6. 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 VeeRwolfX. Its exposed registers are mapped between addresses 0x80002000 and 0x80002FFF. The addresses for each of the registers can be found in the document of the controller (provided in folder *[RVfpgaBasysPath]/src/VeeRwolf/Peripherals/uart/docs*).

ii. Memory

The VeeRwolfX 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 VeeRwolfX SoC will start fetching the initial instructions from this area, which occupies addresses 0x80000000 to 0x80000FFF.

- **RAM:** the VeeRwolfX 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 VeeR EL2 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 5) 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 VeeR EL2 Core Complex uses an AXI4 Interconnect for communicating with the outside world (see Figure 4). 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 6 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.

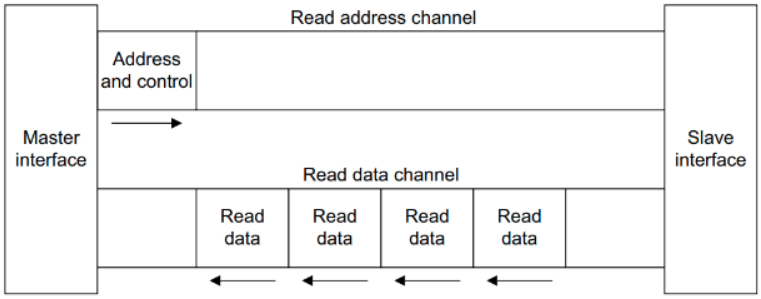


Figure 6. Channel architecture of reads
(figure from https://static.docs.arm.com/ihi0022/e/IHI0022E_amba_axi_and_ace_protocol_spec.pdf)

Figure 7 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.

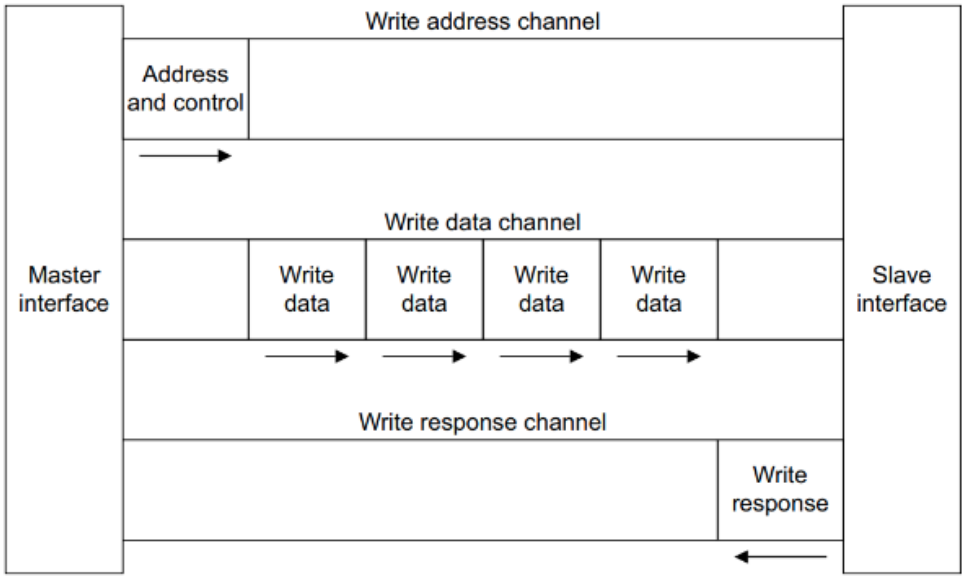


Figure 7. Channel architecture of writes
(figure from https://static.docs.arm.com/ihi0022/e/IHI0022E_amba_axi_and_ace_protocol_spec.pdf)

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 6).
- A write data channel to transfer data from the master to the slave (Figure 7). In a write transaction, the slave uses the write response channel to signal the completion of the transfer to the master (Figure 7).

- **AXI Signals**

Table 6. 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 6. 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 VeeRwolfX 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:
  i. READ/WRITE cycles
  ii. 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 8 illustrates the standard connection between a master (in our case, the VeeR EL2 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 7, it uses fewer signals.

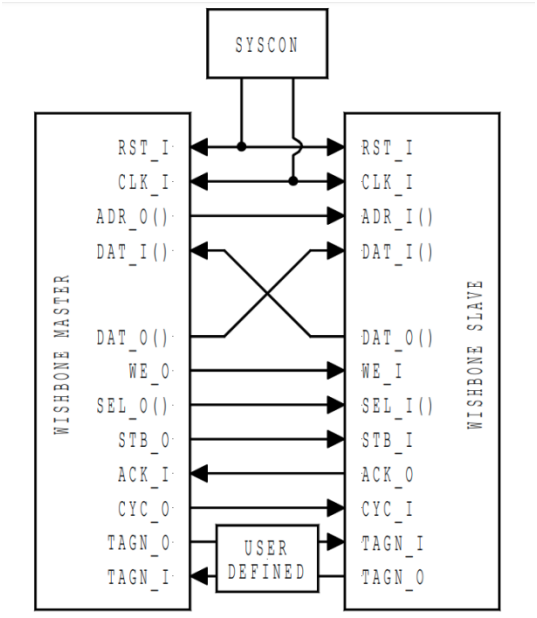


Figure 8. Wishbone Architecture
(figure from https://opencores.org/howto/wishbone)

Table 7. 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. RVfpgaEL2 tools: VeeRwolfX SoC on the Basys3 FPGA Board and in Simulation

In this course, you will use the following tools:

- Execution in hardware (requires a Basys3 physical board):
  - **`RVfpgaEL2-Basys3`**
- Simulation:
  - Verilator-based * (cycle accurate simulation):
    - **`RVfpgaEL2-ViDBo`**
    - **`RVfpgaEL2-Trace`**
    - **`RVfpgaEL2-Pipeline`**
  - Instruction Set Simulator (functional simulation)
    - **`RVfpgaEL2-Whisper`**

Verilator *(https://verilator.org/guide/latest/index.html) converts Verilog and SystemVerilog HDL designs into a C++ or SystemC model that, after compiling, can be executed.*

The last tool (`RVfpgaEL2-Whisper`) is based on the Whisper instruction set simulator (ISS), which is also referred to as the VeeR-ISS (https://github.com/chipsalliance/VeeR-ISS).

The first four tools (`RVfpgaEL2-ViDBo`, `RVfpgaEL2-Trace`, `RVfpgaEL2-Pipeline`, `RVfpgaEL2-Basys3`) share a common **Back-End** but have a different **Front-End** (Figure 9).

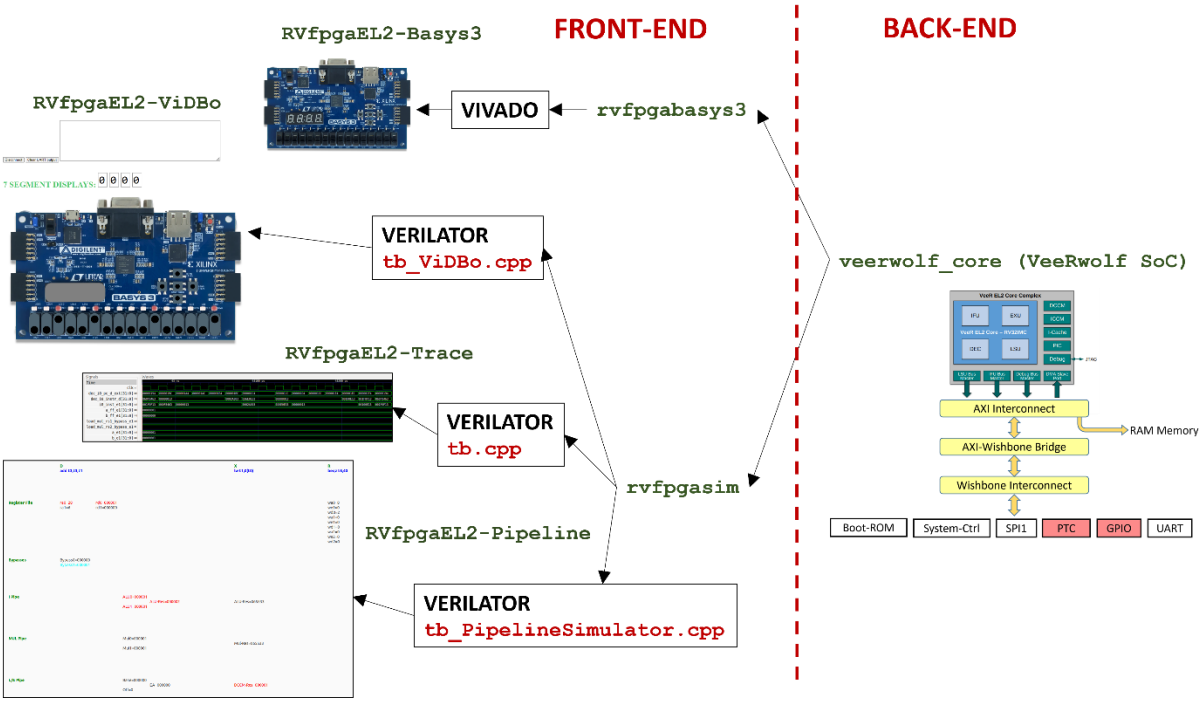


Figure 9. RVfpgaEL2 tools

The **Back-End** is based on the Verilog VeeRwolfX SoC described in the previous section. The **Front-End** is different in each RVfpgaEL2 tool. In the following subsections we briefly describe each of them.

i. Optional Hardware

The RVfpgaEL2 System can be run in simulation only, but users may optionally run the system in hardware on various FPGA boards. In this course, we show how to run the system on three different FPGA boards: Nexys A7, Boolean Board, and Basys 3. The Boolean Board is the least expensive ($74-$105), and the Nexys A7 board is the most expensive ($262-$349). Each of the hardware implementations of the RVfpgaEL2 system is named by the FPGA board it targets: RVfpgaEL2-Nexys (Nexys A7 board), RVfpgaEL2-Boolean (Boolean Board), and RVfpgaEL2-Basys3 (Basys 3 board). This guide shows how to use the Basys3 board, but separate sets of labs show how to use each of the three boards.

RVfpgaEL2-Basys3

The Basys3 FPGA board (Figure 10) is a recommended trainer board for electrical and computer engineering curricula. Digilent provides an extensive reference manual of the board at: https://digilent.com/shop/basys-3-artix-7-fpga-trainer-board-recommended-for-introductory-users/.

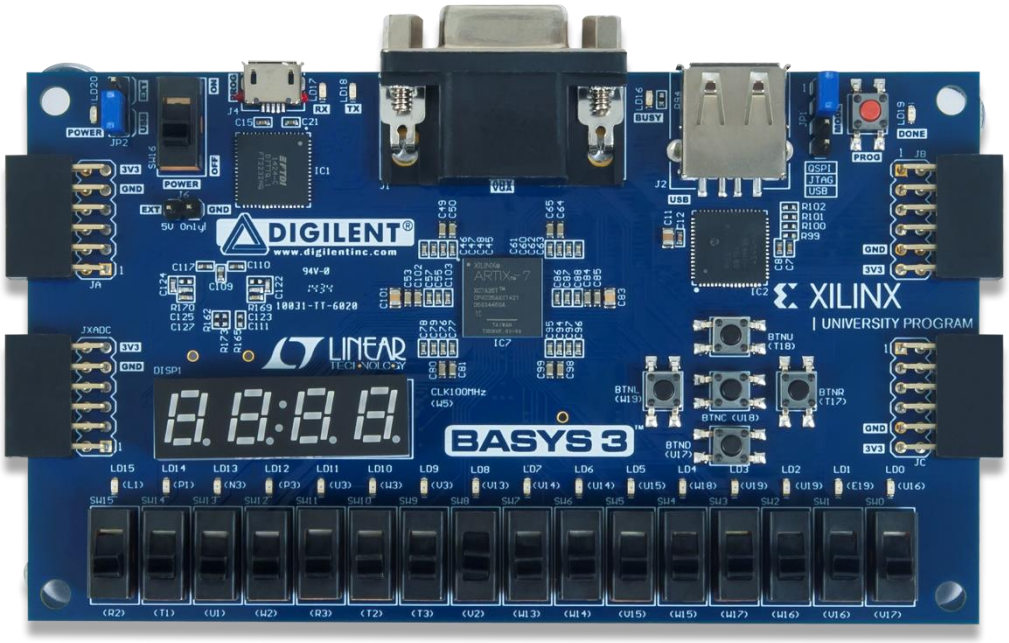


Figure 10. Digilent Basys3 FPGA board
(figure provided by Digilent)

The Basys3 board includes the following interfaces and devices:

- Features the Xilinx Artix-7 FPGA: XC7A35T-1CPG236C
- 33,280 logic cells in 5200 slices (each slice contains four 6-input LUTs and 8 flip-flops)
- 1,800 Kbits of fast block RAM
- Five clock management tiles, each with a phase-locked loop (PLL)
- 90 DSP slices
- Internal clock speeds exceeding 450 MHz
- On-chip analog-to-digital converter (XADC)
- Digilent USB-JTAG port for FPGA programming and communication
- Serial Flash
- USB-UART Bridge
- 12-bit VGA output
- USB HID Host for mice, keyboards and memory sticks
- 16 user switches
- 16 user LEDs
- 5 user pushbuttons
- 4-digit 7-segment display
- 4 Pmod ports: 3 Standard 12-pin Pmod ports, 1 dual purpose XADC signal / standard Pmod port

RVfpgaEL2-Basys3 is the VeeRwolfX SoC targeted to the Basys3 FPGA board. The main elements used by **RVfpgaEL2-Basys3** are illustrated in Figure 11:

- Hardware programmed onto the FPGA:
  - **VeeRwolfX SoC** (illustrated in Figure 5)
  - **Clock Generator:** the Basys3 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 **12.5 MHz** to use in the **VeeRwolfX SoC**.

- **BSCAN logic for the JTAG:** you can find more information about this module at https://github.com/chipsalliance/VeeRwolf/issues/29.

- Memory/Peripherals used in RVfpgaEL2-Basys3 from the Basys3 FPGA board:
  - **USB** connection for **UART**
  - **SPI Flash memory**
  - **16 LEDs** and **16 Switches**
  - **4-digit 7-Segment Displays**

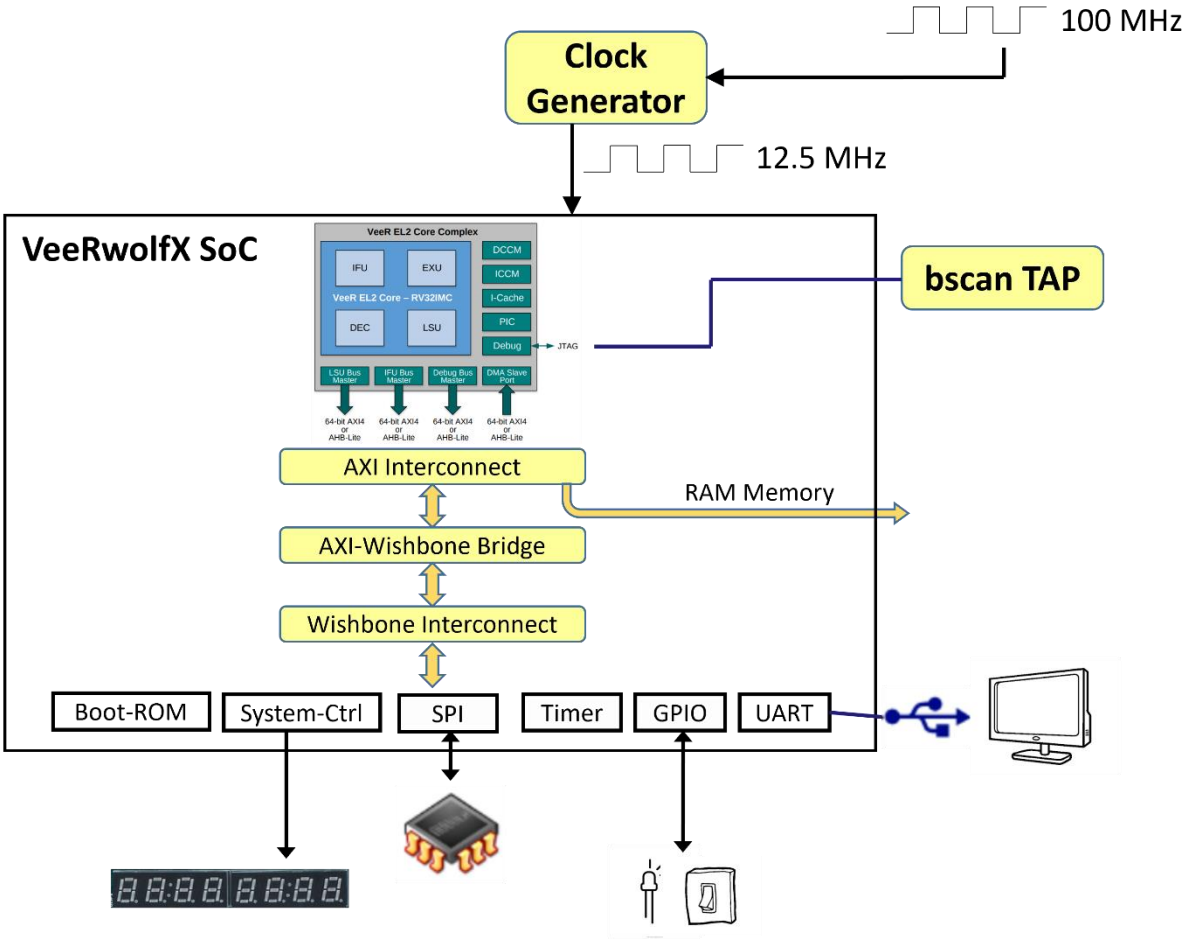


Figure 11. RVfpgaEL2-Basys3

ii. Simulators

In this course we use one instruction set simulator (ISS) called Whisper and three Verilator-based simulators for simulating the RVfpgaEL2 hardware:

- **`RVfpgaEL2-ViDBo`**
- **`RVfpgaEL2-Trace`**
- **`RVfpgaEL2-Pipeline`**

In the following subsections, we describe the main characteristics of the Verilator-based simulators, and in sections 6-8 we show examples of these simulation tools. Section 9 shows how to use the Whisper instruction set simulator.

a. `RVfpgaEL2-ViDBo`

It uses a Virtual Basys3 Board to perform a simulation of the RVfpgaEL2 System and communicate with the peripherals of the simulated board. The following peripherals are supported in RVfpgaEL2-ViDBo: 16 switches, 16 LEDs, UART, 4 pushbuttons, 8 7-Segment Displays, and a tricolor LED. Figure 12 illustrates RVfpgaEL2-ViDBo while executing an example program that shows the value of the switches on the LEDs.

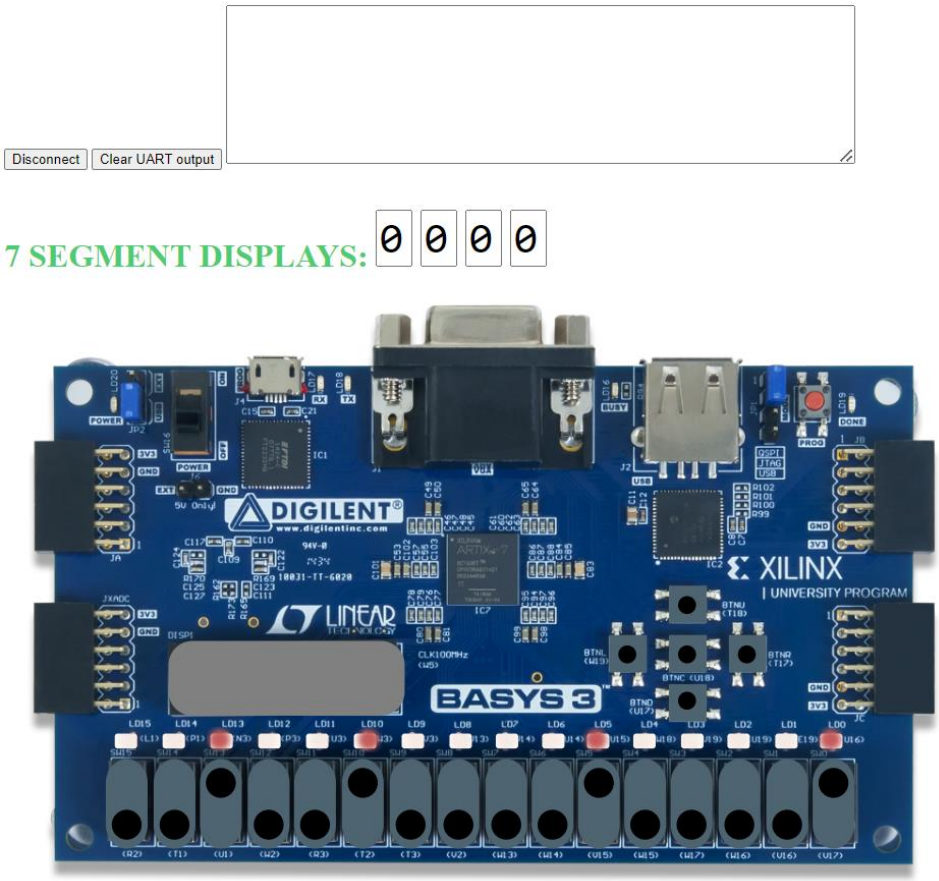


Figure 12. RVfpgaEL2-ViDBo

b. `RVfpgaEL2-Trace`

It generates a trace of the different internal signals of the SoC while executing a given program and then uses GTKWave (https://gtkwave.sourceforge.net/) for visualizing the waveform of this trace. Figure 13 illustrates the waveform of an example program that includes a `mul` and an `add` instruction.

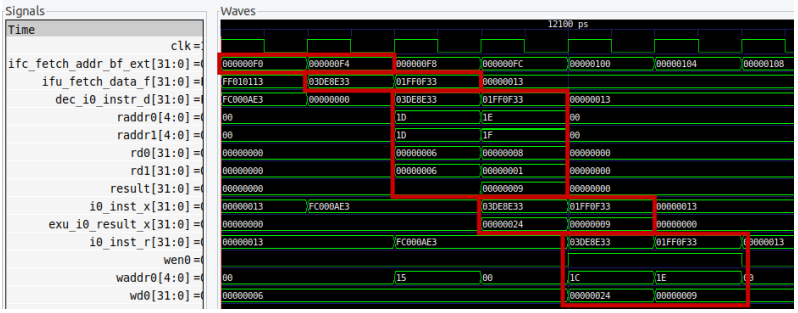


Figure 13. RVfpgaEL2-Trace

c. RVfpgaEL2-Pipeline

It uses a VeeR EL2 Pipeline Simulator for analyzing the evolution of the instructions through the pipeline. This is very useful for analysing the VeeR EL2 core, as we will do in the second half of labs.

Figure 14 illustrates the RVfpgaEL2-Pipeline while executing an example program that includes several Arithmetic-Logic instructions.

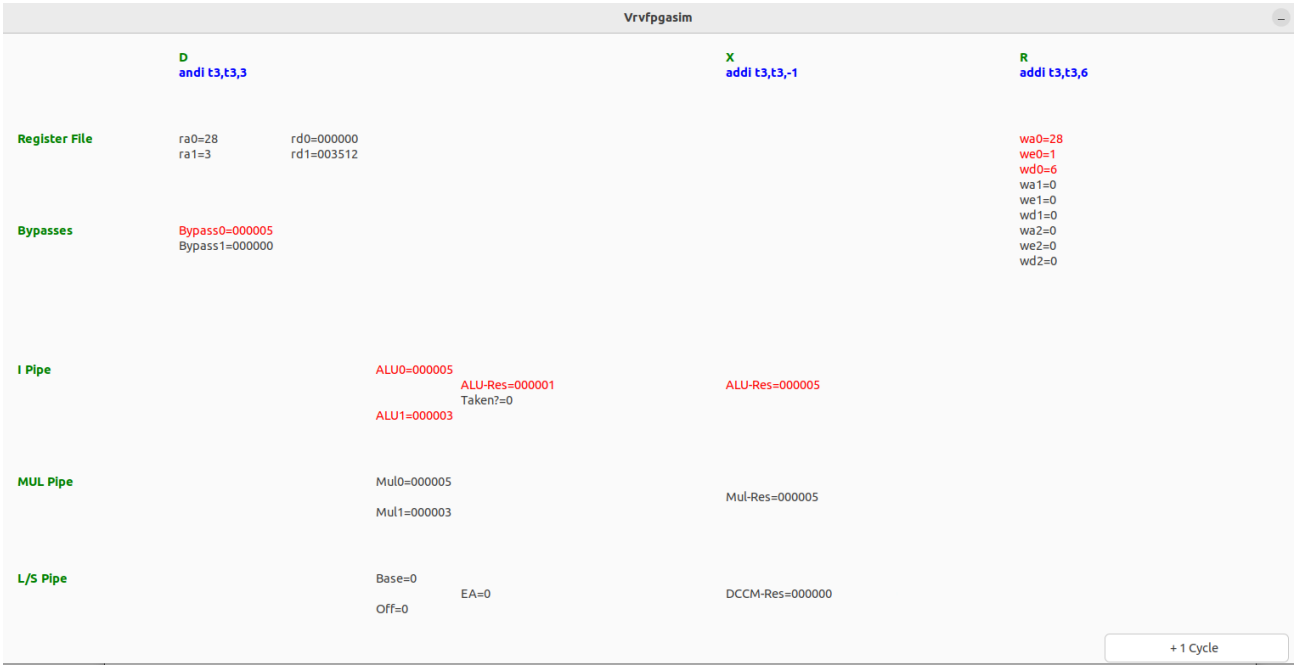


Figure 14. RVfpgaEL2-Pipeline

D. File Structure

In the previous sections we showed the high-level organization of the RVfpgaEL2 System, from the **VeeR EL2 Core Complex** (Figure 4), to the **VeeRwolfX SoC** (Figure 5) and, finally, to the optional hardware (**RVfpgaEL2-Basys3** (Figure 11)) and Verilator-based simulators (**RVfpgaEL2-ViDBo, RVfpgaEL2-Trace, and RVfpgaEL2-Pipeline** (Figure 12, Figure 13, and Figure 14)).

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 ***[RVfpgaBasysPath]/src*** and ***[RVfpgaBasysPath]/Simulators***).

i. VeeR EL2 Core Complex

Figure 15 shows the file structure of the **VeeR EL2 Core Complex** (Figure 4). The core is organized into three main blocks: a VeeR wrapper (highlighted in grey) that includes the VeeR EL2 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).

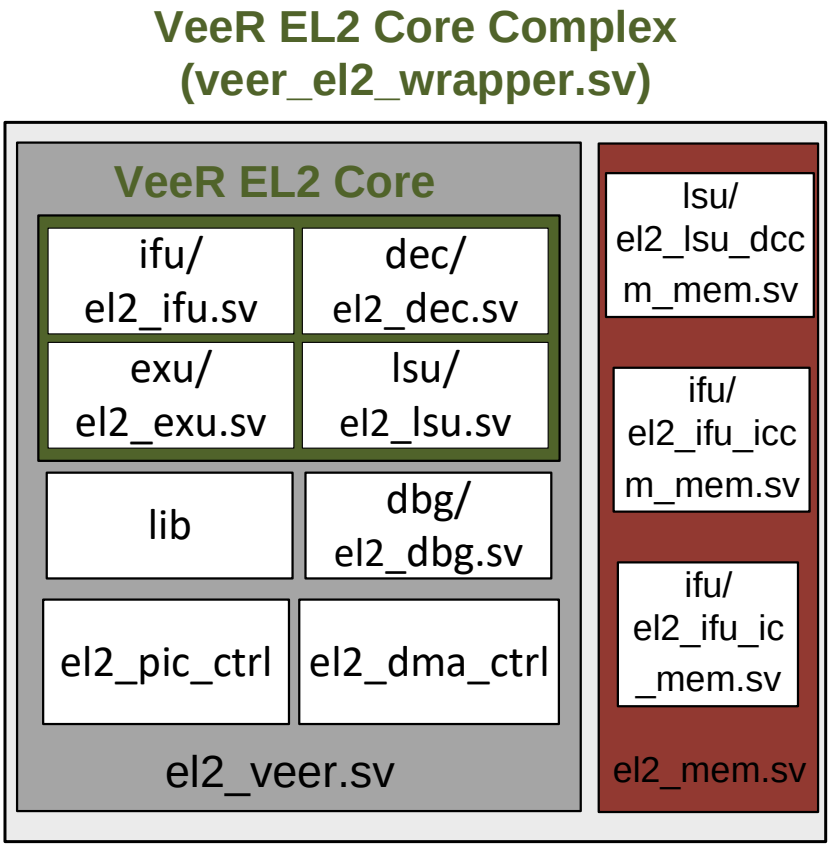


Figure 15. VeeR EL2 Core Complex

The Verilog files for the VeeR EL2 Core Complex are available in this folder: *[RVfpgaBasysPath]/src/VeeRwolf/VeeR_EL2CoreComplex*

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

The top file for the VeeR EL2 Core Complex is in the file: *veer_el2_wrapper.sv*; the top module is called **`veer_el2_wrapper`**, and it instantiates two modules that correspond to the two blocks highlighted in grey and red in Figure 15:

- **`el2_mem`** (implemented inside *el2_mem.sv*): this module instantiates the modules for the implementation of the DCCM (**`el2_lsu_dccm_mem`**, implemented in file *lsu/el2_lsu_dccm_mem.sv*), the ICCM (**`el2_ifu_iccm_mem`**, implemented in file *ifu/el2_ifu_iccm_mem.sv*) and the Instruction Cache (**`el2_ifu_ic_mem`**, implemented in file *ifu/el2_ifu_ic_mem.sv*).

- **`el2_veer`** (implemented inside *el2_veer.sv*): this module instantiates the units that comprise the core.

The VeeR EL2 Core (highlighted in green in Figure 15) consists of the following four units:

- Folder ***ifu*** (Instruction Fetch Unit): this folder includes the Verilog files (top module available inside *el2_ifu.sv*) for the I$ (instruction cache), Fetch, Branch Predictor, and Aligner.
- Folder ***dec*** (Decode Unit): this folder includes the Verilog files (top module available inside *el2_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 *el2_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 *el2_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 *el2_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.
- File *el2_pic_ctrl.sv*, which implements the Programmable Interrupt Controller in module **el2_pic_ctrl**.
- File *el2_dma_ctrl.sv*, which implements the Direct Memory Access in module **el2_dma_ctrl**.

ii. VeeRwolfX SoC

Figure 16 shows the file structure for the **VeeRwolfX SoC** shown in Figure 5. The SoC is organized as the modules that correspond to the blocks shown in Figure 5.

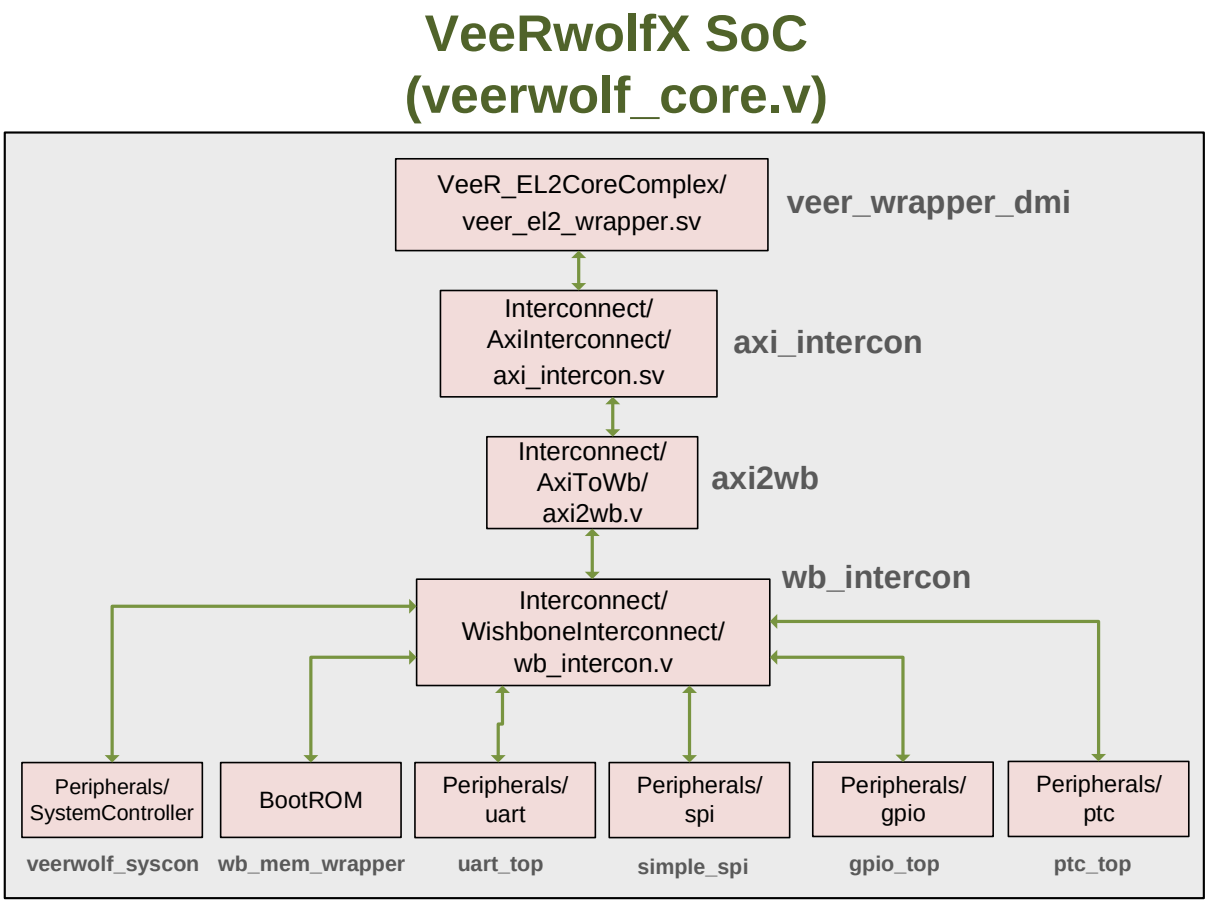


Figure 16. VeeRwolfX SoC

The files for the VeeRwolfX SoC are in:
[RVfpgaBasysPath]/src/VeeRwolf

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

The top module for the **VeeRwolfX SoC** is available at:
[RVfpgaBasysPath]/src/VeeRwolf/VeeRwolf_core.v. Open that file, and notice that it includes the modules contained within the VeeRwolfX SoC (Figure 5), specifically:

- **`axi_intercon`** (available inside *Interconnect/AxiInterconnect/axi_intercon.v*): this module is included through another file (`` `include "axi_intercon.vh" ``). It connects the VeeR EL2 Core Complex with the AXI-to-Wishbone Bridge.
- **`axi2wb`** (available inside *Interconnect/AxiToWb/axi2wb.v*): this module, is the AXI-to-Wishbone Bridge that allows communication between the AXI based EL2 Core and the Wishbone-based peripherals.
- **`wb_intercon`** (available inside *Interconnect/WishboneInterconnect/wb_intercon.v*): this module is included through another file (`` `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 *VeeRwolf_core.v*. It instantiates the **`dpram64`** module (available inside *BootROM/dpram64.v*), which is a basic RAM module.

- **`VeeRwolf_syscon`**: this module defines the System Controller (available inside *Peripherals/SystemController/VeeRwolf_syscon.v*):.
- **`simple_spi`**: SPI controller obtained from OpenCores and available inside *Peripherals/spi/simple_spi_top.v*. It is instantiated twice in *VeeRwolf_core.v*.
- **`uart_top`**: UART controller obtained from OpenCores and available inside *Peripherals/uart/uart_top.v*.
- **`gpio_top`**: GPIO controller obtained from OpenCores and available inside *Peripherals/gpio/gpio_top.v*.
- **`ptc_top`**: PTC controller obtained from OpenCores and available inside *Peripherals/ptc/ptc_top.v*.
- **`veer_wrapper_dmi`** (available inside *VeeR_EL2CoreComplex/veer_el2_wrapper.v*): instantiation of VeeR EL2 Core Complex, described in the previous section (Figure 15).

iii. Wrappers for on-board execution and simulation

ON BOARD EXECUTION:

Module `rvfpgabasys3` (that can be found at *[RVfpgaBasysPath]/src/rvfpgabasys3.sv*), instantiates the VeeRwolf SoC (module `veerwolf_core`) and some other elements (see Figure 11), such as an AXI RAM or a clock generator, that are necessary to make the SoC work on the Basys3 board. It also specifies the interface of input/output signals used to communicate the SoC and the board (for example, `i_sw` provides the values of the switches from the board to the SoC, and `o_led` provides the values of the LEDs from the SoC to the board).

File `rvfpgabasys3.xdc` (that can be found at *[RVfpgaBasysPath]/src/rvfpgabasys3.xdc*) is the constraints file, which provides the information of what physical pins on the FPGA are used by the VeeRwolf SoC (LEDs, Switches, UART, etc.).

The Verilog files and the constraints files are provided to Vivado for creating a bitstream, which can then be loaded into the Basys3 FPGA and used for running programs on the SoC.

As a summary, Figure 17 shows the hierarchy for the whole RVfpgaEL2-Basys3 implementation on the Basys3 FPGA board.

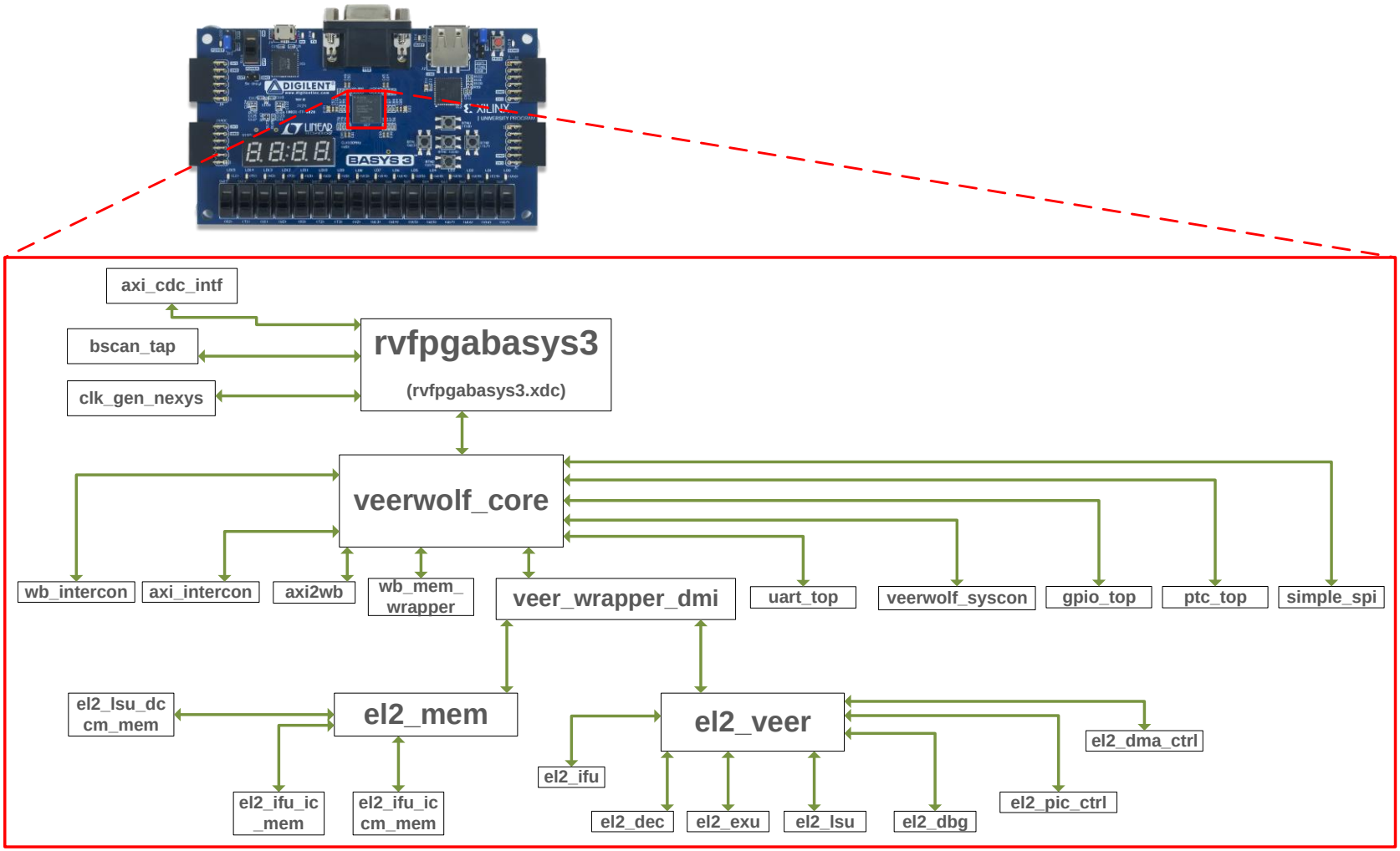


Figure 17. Modules hierarchy for the Basys3 FPGA board implementation of RVfpgaEL2

SIMULATION:

Module `rvfpgasim` (that can be found at *[RVfpgaBasysPath]/Simulators/SimulationSources/rvfpgasim.v*), instantiates the VeeRwolf SoC (module `veerwolf_core`) and some other elements (such as a simulated memory) that are necessary to make the SoC work on simulation. It also specifies the interface of input/output signals used to communicate the SoC and the simulator (note that the interface is different depending on the simulation tool that we use: *RVfpgaEL2-ViDBo*, *RVfpgaEL2-Trace*, *RVfpgaEL2-Pipeline*).

RVfpgaEL2-ViDBo:

File *tb_ViDBo.cpp* (that can be found at *[RVfpgaBasysPath]/Simulators/verilatorSIM_ViDBo*), contains the main loop for the simulation. Each iteration of the loop simulates half a cycle. The program acts as a bridge between the VeeRwolf SoC and the Virtual Board:

- It receives signals from the SoC (for example `top->o_led`, which is connected with `output o_led` of module `rvfpgasim` and contains the values for the LEDs provided by the VeeR EL2 core) and sends them to the Virtual Board (where the LEDs show the value provided in signal `o_led`)

- It receives signals from the Virtual Board (for example `top->i_sw`, which is connected with the switches of the board) and sends them to the SoC (where the signal is received in `input i_sw` and provided to the VeeR EL2 Core).

The virtual board is implemented in folder *[RVfpgaBasysPath]/Simulators/verilatorSIM_ViDBo/Basys3Board*. This folder includes an HTML program (*basys3.html*) that draws the board and its peripherals and communicates with the Verilator simulated SoC. For performing this communication, files *vidbo.c* and *vidbo.h* (available at *[RVfpgaBasysPath]/Simulators/verilatorSIM_ViDBo/*) define several functions that send/receive signals based on the LibWebSockets C library (https://github.com/warmcat/libwebsockets).

RVfpgaEL2-Trace:

File *tb.cpp* (that can be found at *[RVfpgaBasysPath]/Simulators/verilatorSIM_Trace*), contains the main loop for the simulation. Each iteration of the loop simulates half a cycle. The loop generates a file (*trace.vcd*) that contains information for all SoC internal signals.

The trace file is then visualized in GTKWave (https://gtkwave.sourceforge.net/), which allows it to analyze the evolution in time of the internal SoC signals.

RVfpgaEL2-Pipeline:

File *tb_PipelineSimulator.cpp* (that can be found at *[RVfpgaBasysPath]/Simulators/verilatorSIM_Pipeline*), contains the program that illustrates the core internal signals in the VeeR EL2 pipeline. It uses gtk (https://github.com/GNOME/gtk) to draw the VeeR EL2 pipeline diagram. In order to disassemble the instructions, the RVfpgaEL2-Pipeline simulator makes use of the RISCV disassembler developed by Michael Clark, which can be obtained from: https://github.com/michaeljclark/riscv-disassembler/tree/master.

4. Installing Software Tools

The instructions below are for an Ubuntu 22.04 system. They also work for older Ubuntu versions, for other Linux distributions, for Windows 10 and for 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.

Virtual Machine: if you are using the provided Ubuntu 22 Virtual Machine you can skip this section, as everything is already installed except for Vivado, which you can install yourself if you want to use it.

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 the advanced labs, where different features will be included to the baseline SoC.
- B. **Catapult Studio**: this IDE is used in the GSG and in the Labs. It is used for programming the FPGA, for compiling the programs and for running/debugging the programs on the board.
- **C. Drivers for the Basys3 board**: required for your computer to be able to use the board.
- **D. Tools for the simulators**: required for simulating the SoC in the different simulators.

The whole installation can take around a couple of hours (or more, depending on your download speed), but most of the time is spent waiting while the programs are downloaded and installed.

If you do not have the hardware (FPGA board), you do not need to install Vivado or the board drivers. In that case, skip Section 4.A and 4.C and complete only sections 4.B and 4.D. Then, skip Section 5 (Execution on the board) and complete sections 6-9 (Simulation using the various tools).

If you want to use the physical board but you are not going to rebuild the SoC (which is only done in a few advanced exercises where the baseline SoC is extended), you do not need Vivado. In this case, skip Section 4.A and complete only sections 4.B, 4.C and 4.D.

A. Install Vivado

This is an optional step for users who want to implement RVfpgaEL2 in hardware on an FPGA board. 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.

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://www.xilinx.com/support/download/index.html/content/xilinx/en/downloadNav/vivado-design-tools/2022-2.html

2. Click on *Xilinx Unified Installer 2022.2: Linux Self Extracting Web Installer*.

WINDOWS: Click on *Xilinx Unified Installer 2022.2: Windows Self Extracting Web Installer*.

3. 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.

4. Execute the binary file with root permissions. You may need to give execution rights to the installation file. For example:

```
chmod +x ./Xilinx_Unified_2022.2_1014_8888_Lin64.bin

sudo ./Xilinx_Unified_2022.2_1014_8888_Lin64.bin
```

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

5. The Vivado installer will walk you through the installation process. Important notes:
   - Select **Download and Install Now**.
   - Select **Vivado** (*not* Vitis) as the Product to install.
   - Select **Vivado ML Standard** (*not* Vivado ML Enterprise), as this is the no-cost version.
   - Set all the **I Agree** boxes.
   - Otherwise, defaults should be selected.

WINDOWS: Steps 6 and 7 are not necessary in Windows. You can simply ignore these steps.

6. After Vivado has installed, you need to set up the environment. Open a terminal and type:

```
source /tools/Xilinx/Vivado/2022.2/settings64.sh
```

Add that line (`source /tools/Xilinx/Vivado/2022.2/settings64.sh`) to your `~/.bashrc` file so that it runs each time you launch a terminal.

Hint: If you changed the installation directory of Vivado, you will need to modify the path appropriately in Step 6.

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

```
vivado
```

B. Install Catapult SDK

Catapult SDK, a free RISC-V software development kit from Imagination Technologies, contains a toolchain for building your C code, debugging tools such as gdb and OpenOCD, and **Catapult Studio**, an Integrated Development Environment (IDE). You will use Catapult Studio to download RVfpgaEL2-Basys3 onto the Basys3 board and also to download and run programs on RVfpgaEL2-Basys3.

Follow these steps to install Catapult SDK:

1. Download the installer for your platform from https://github.com/imgtec-riscv/catapult-sdk Install the latest 1.X.X version (when this document was written it was 1.9.1). Whenever the Catapult version is referenced in a command (such as in item 3 below), make sure

that you use the one that corresponds to the version that you have installed.

2. Open a terminal and run the installer:
   ```
   cd ~/Downloads
   sudo apt install ./catapult-sdk*.deb
   ```
   This installs the tools under the `/opt/imgtec` folder and also installs some example projects and documentation in the ~/catapult-sdk_examples folder.
3. If you wish to use the tools on the command line as well as within Catapult Studio, you should add the bin directory of the SDK to your path. You can do this by running the following (but substitute the correct SDK version number in the directory path):
   ```
   cd ~/catapult-sdk_examples/1.9.1
   source ./set_sdk_path.sh
   ```
4. Catapult Studio should now appear in your application menu.

Windows / macOS: Catapult SDK packages are also available for Windows (.exe file) and macOS (.pkg file) at the download location above. Follow the usual steps used for installing and executing an application in these operating systems.

On macOS, if you have not already done so, you may need first to go to the Security and Privacy section of System Preferences, General tab, and enable "Allow apps downloaded from Apple and selected developers" in order to permit the install.

C. Install the drivers for the Basys3 board

You need to manually install the drivers for your computer to be able to use the Basys3 board. Follow the next steps:

- Plug the Basys3 board in your computer
- Open a terminal.
- Go into directory *[RVfpgaBasysPath]/driversLinux*. For simplicity, we provide these drivers inside the RVfpgaEL2 folder; however, you can also find these drivers inside the Vivado package, at:
  /tools/Xilinx/Vivado/2022.2/data/xicom/cable_drivers/lin64/install_script/install_drivers
- Run the installation script:
  - `chmod 777 *`
  - `sudo ./install_drivers`
- Unplug the Basys3 board from your computer and restart the computer for the changes to have effect.

Windows: follow the instructions provided in Appendix A for installing the drivers for the Basys3 board.

D. Install tools used by the simulators

For being able to use the different Verilator-based simulators that we include with this course, you need to install some tools first. Before carrying out the next steps, install the following packages:

```
sudo apt-get update
sudo apt-get install git make autoconf g++ flex bison libfl2 libfl-dev
```

Windows: The instructions for installing the tools required to build and run the simulators in Windows are provided in Appendix B.

macOS: The instructions for installing the tools required to build and run the simulators in macOS are provided in Appendix C.

NOTE: We provide pre-compiled binaries for the simulators in the *OriginalBinaries* folder included in each simulator folder. So, for example, at *[RVfpgaBasysPath]/Simulators/verilatorSIM_Trace/OriginalBinaries* you can find the binaries for the RVfpgaEL2-Trace simulator. If you do not want to regenerate the binaries, you can avoid Verilator installation.

Verilator installation:

Verilator allows you to regenerate the simulators if you make any changes. In Ubuntu you can install Verilator by simply using the pre-compiled package:

```
sudo apt-get update
sudo apt-get install verilator
```

The version installed in Ubuntu 22.04 is v4, which should work correctly. In some systems v5 can be installed by default, which may cause some problems. In those cases we recommend to downgrade to v4.

You can also compile the sources. In this case, follow the next steps (instructions are available at: https://www.veripool.org/projects/verilator/wiki/Installing but are also summarized below). This process can take a long time.

- `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.

RVfpgaEL2-Trace:

GTKWave enables us to visualize the traces generated with RVfpgaEL2-Trace. In Ubuntu you can simply install it with the following command:

```
sudo apt-get update
sudo apt-get install gtkwave
```

RVfpgaEL2-ViDBo:

LibWebSockets enables us to communicate the SoC with the Virtual Board.

In Ubuntu you can install LibWebSockets by simply using the pre-compiled package:

```
sudo apt-get update
sudo apt-get install libwebsockets-dev
```

In Utuntu, Python is already installed by default. If you don't have it, you can install it using the `apt-get` command.

RVfpgaEL2-Pipeline:

You need to install some packages for using this simulator.

- `sudo apt-get update`
- `sudo apt install libcairo2-dev`
- `sudo apt-get install libgtk-3-dev`

5. Execution in RVfpgaEL2-Basys3

In this section, we show how to run seven simple programs on RVfpgaEL2-Basys3 (see Figure 11).

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 4. 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 RVfpgaEL2-Basys3 by showing how to run the seven example programs listed in Table 8. 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 RVfpgaEL2-Basys3 are described below.

Table 8. RVfpgaEL2-Basys3 example programs

Program Name	Description	Language
AL_Operations	exercises arithmetic and logical operations	RISC-V assembly
Blinky	blinks an LED on the Basys3 board	RISC-V assembly
LedsSwitches	reads switch values on Basys3 board and writes that value to the LEDs	RISC-V assembly
LedsSwitches_C-Lang	reads switch values on Basys3 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 in hardware, **you must program the FPGA with RVfpgaEL2-Basys3**, as explained in the following section.

A. Program the FPGA with RVfpgaEL2-Basys3

In this optional section, we explain the recommended method for programming the FPGA with RVfpgaEL2-Basys3, which uses Catapult Studio. Follow the next steps:

Step 1. Connect Basys3 FPGA board to computer and turn the board on
Connect the Basys3 board to your computer using the provided USB cable.

Virtual Machine: If you are using the VM, you need to select the board on the VM (see the following picture):

Figure 18 shows the physical locations of the LEDs and switches on the Basys3 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 Basys3 board and turn on the board.

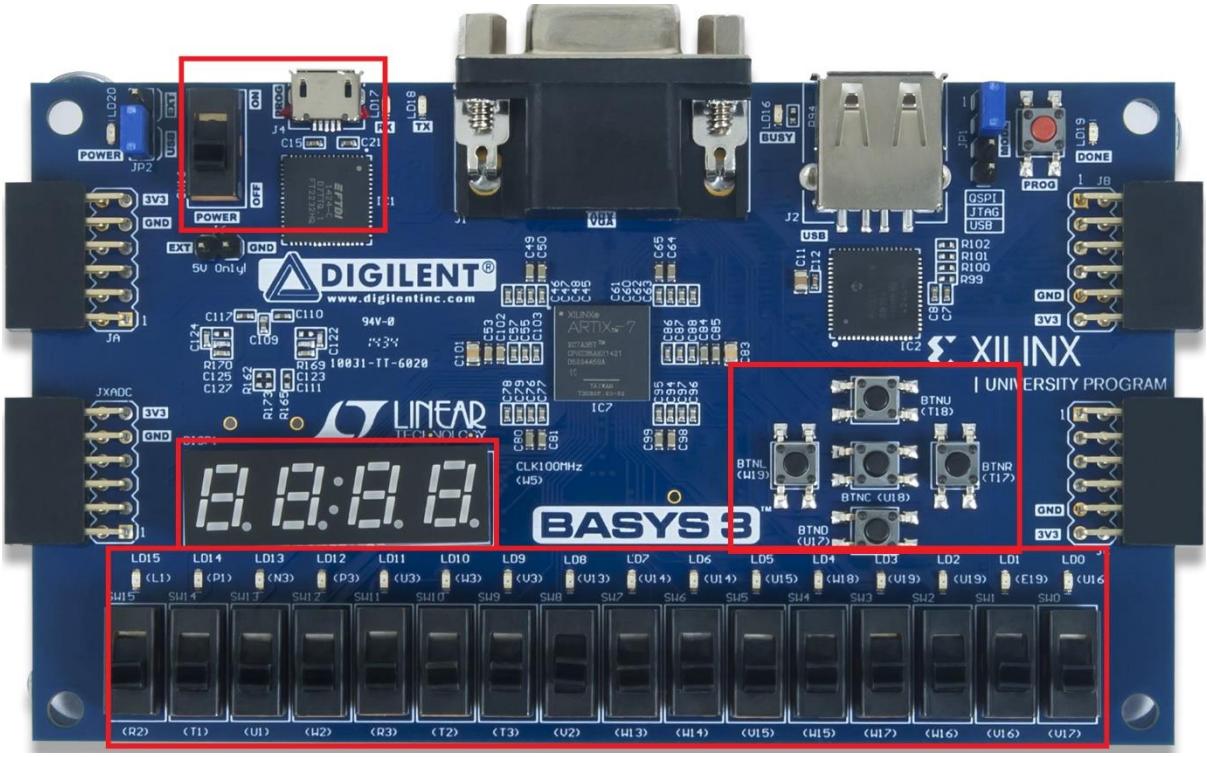


Figure 18. Basys3 FPGA board's I/O interfaces
(figure provided by Digilent)

Step 2. Open Catapult Studio and C project

Now open Catapult Studio by typing Catapult in the Start Menu (see Figure 19).

Figure 19. Open Catapult Studio

On the top menu bar, click on *File → Open Folder* (see Figure 20) and browse into directory *[RVfpgaBasysPath]/examples/*

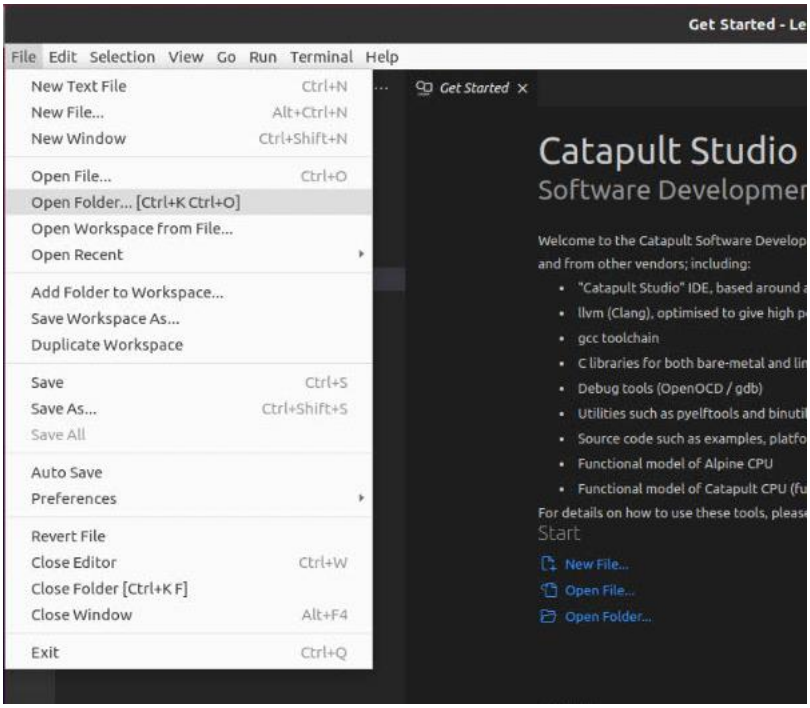


Figure 20. Open folder

Select the project that you are going to use. In this section, as an example, we use *AL_Operations*, the first example mentioned in Table 8, 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 21) and click OK at the top of the window. Catapult Studio will now open the example.

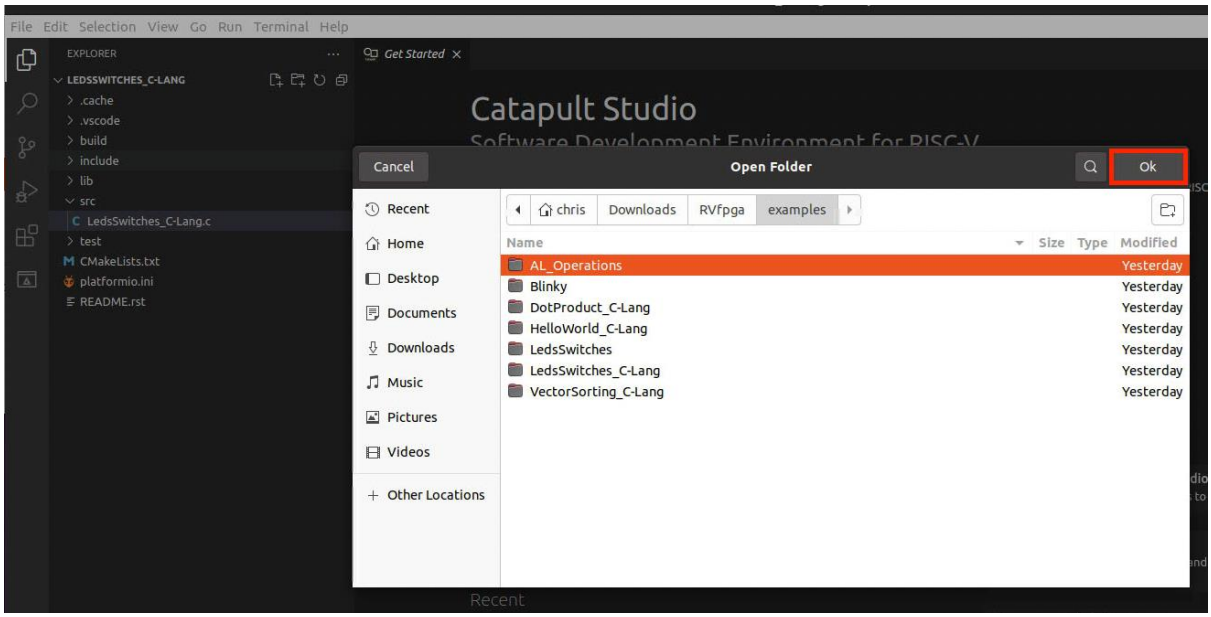


Figure 21. Open AL_Operations folder

If asked if you trust the folder and its parent, say yes (see Figure 22).

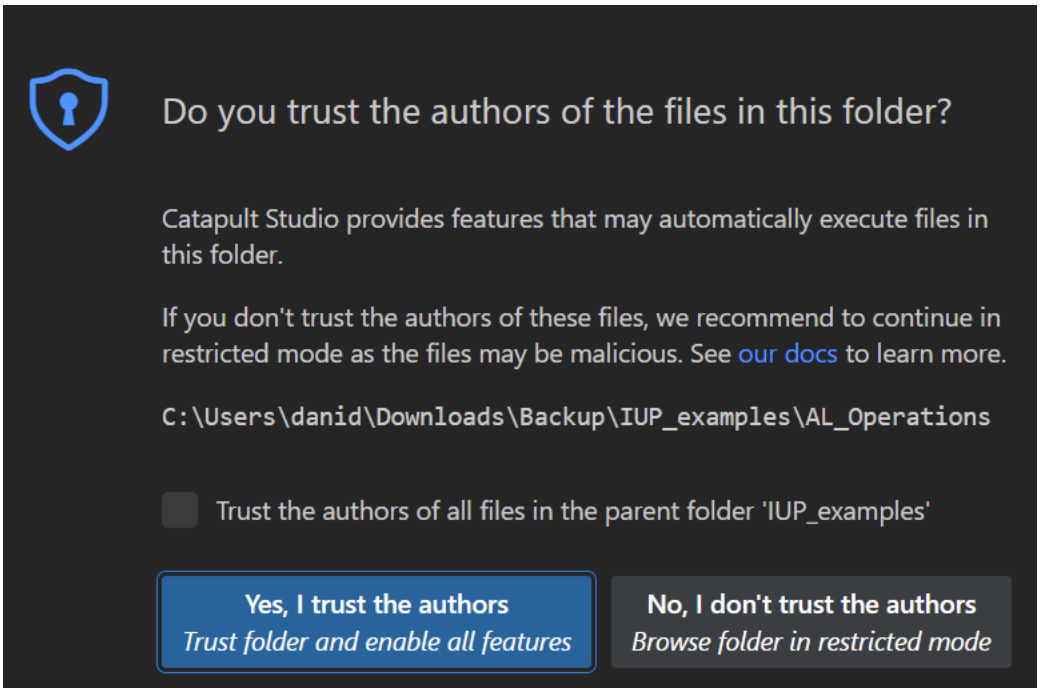


Figure 22. Opening AL_Operations example

You may see a dialog box saying that Catapult Studio has detected the project as a CMake project and offering to configure it (Figure 23). Click Yes to this.

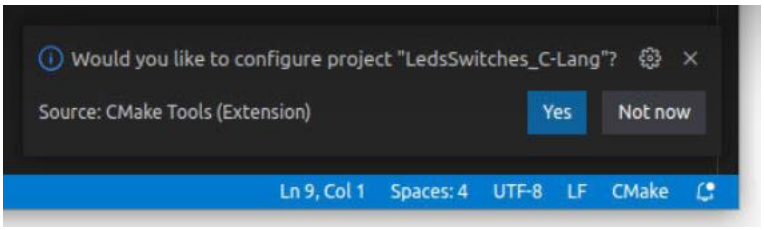


Figure 23: CMake project dialog

You may also have to select a kit for the project, as shown in Figure 24. Select “Catapult SDK gcc-elf”.

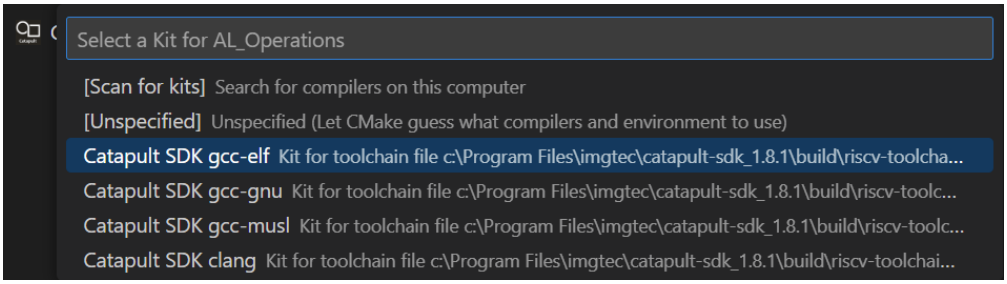


Figure 24: Select a kit

You may also see a warning that the project does not have *tasks.json* and *launch.json* files, as shown in Figure 25). Dismiss the warning.

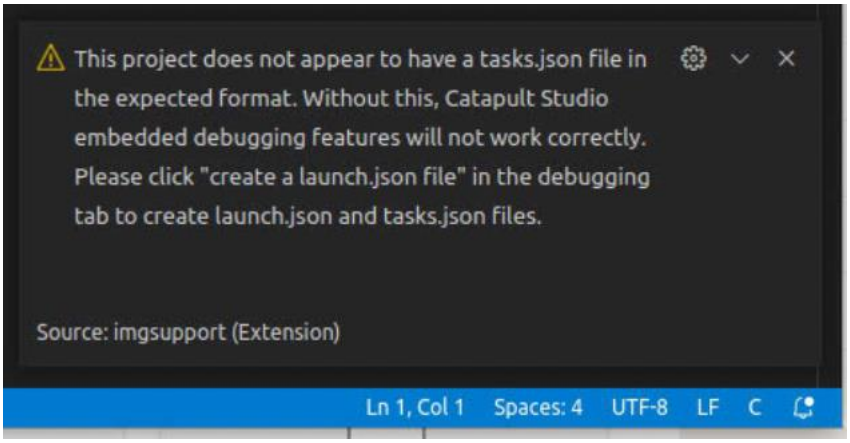


Figure 25. Prompt to create tasks.json and launch.json files

If these files (*tasks.json* and *launch.json*) have not been created, do it by clicking on the Run/Debug tab on the left hand side, then clicking “create a launch.json file” (see Figure 26).

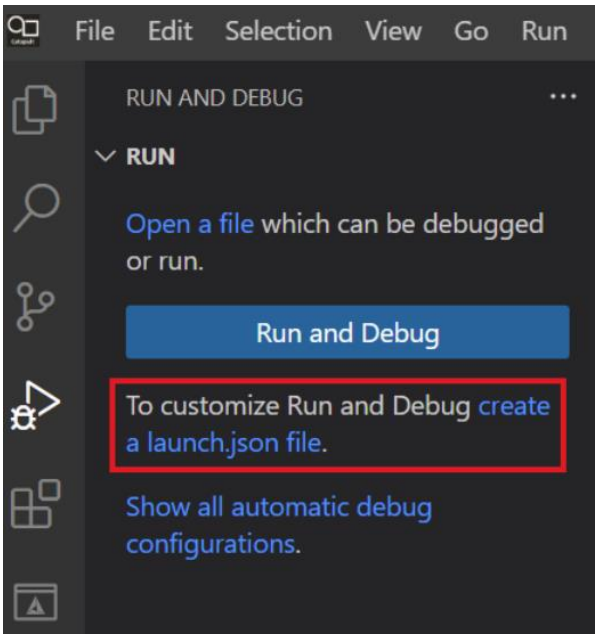


Figure 26: Creating a launch.json file

This will produce a dialog box as illustrated in Figure 27: select “C/C++ Embedded” as the debugger type. This only has to be done once when opening a new project.

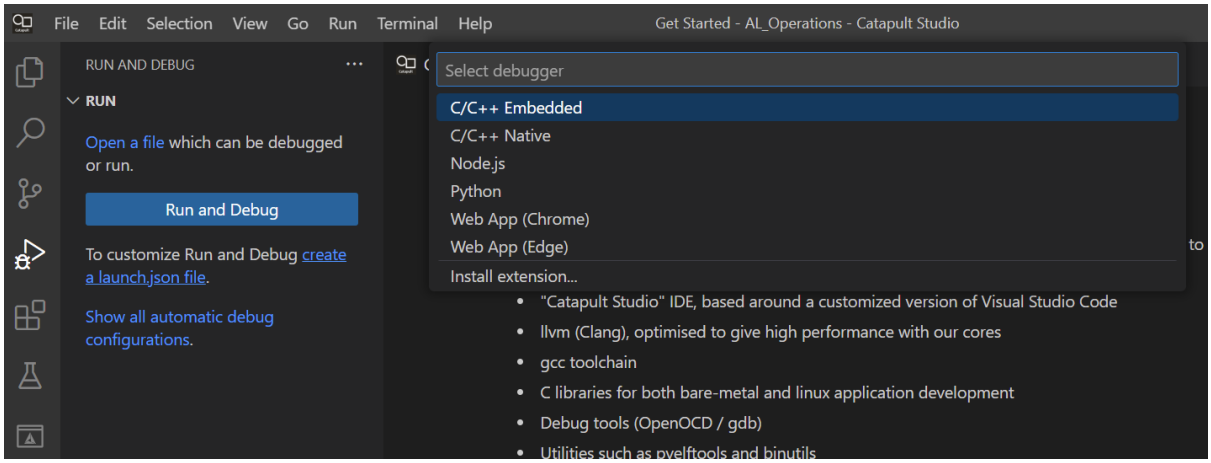


Figure 27: Selection of C/C++ Embedded debugger type

Finally, click the Platform selection on the bar at the bottom of the screen, and select “RVfpga Basys3 via OpenOCD” for the platform (Figure 28).

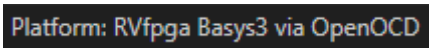


Figure 28: Platform selection

Normally, for speed, the CMake build system will only re-build files that have changed. In some circumstances, such as after changing the selected platform, you need to clean the build and re-start it from scratch. To do this, select the CMake tab and select “Clean reconfigure all projects” from the “…” menu (Figure 29).

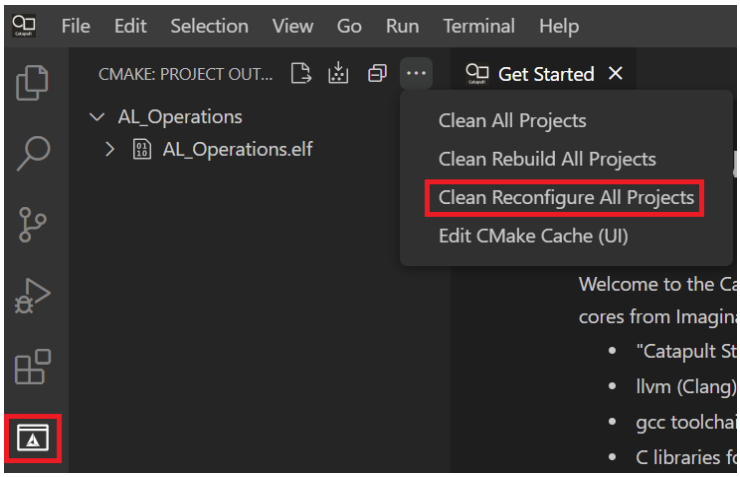


Figure 29: Cleaning the CMake build

Step 3. Build the C Program

The program builds using CMake, a popular build framework which is well supported by Catapult Studio (see https://cmake.org). The build is controlled by the file CMakeLists.txt in the top-level folder.

Assuming you accepted to allow Catapult Studio to configure the project as a CMake project in step 2, you should see a number of CMake controls in the bar at the bottom (Figure 30).

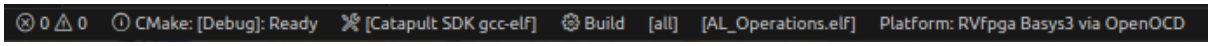


Figure 30: CMake controls

- CMake: [Debug]: Ready this selects the build configuration (e.g. Debug or Release): select Debug.
- [Catapult SDK gcc-elf] this selects a toolchain for the build: select Catapult SDK gcc-elf.
- Build finally hit this button to run the build.

You will see the build output in the built-in terminal within Catapult Studio. If successful, it should end with “Build finished with exit code 0”.

Step 4. Download RVfpgaEL2-Basys3 to Basys3 board

You are now ready to download RVfpgaEL2-Basys3, the RISC-V SoC that includes a RISC-V processor with support for peripherals.

Click the Platform selector in the bar at the bottom, and select “Configure Platforms” (alternatively, you can bring up the command palette with CTRL-Shift-P, and select “Catapult: Configure Platforms”). See Figure 31.

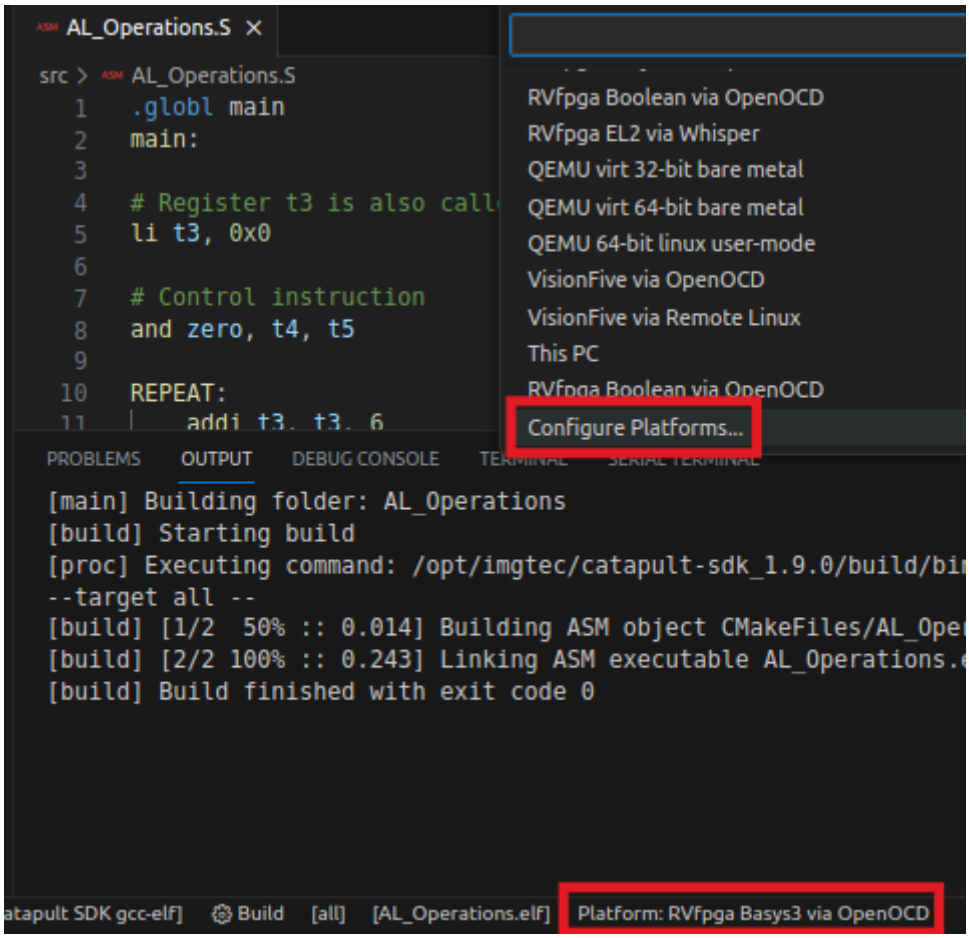


Figure 31: Configure Platforms

This opens the Configure Platforms window illustrated in Figure 32:

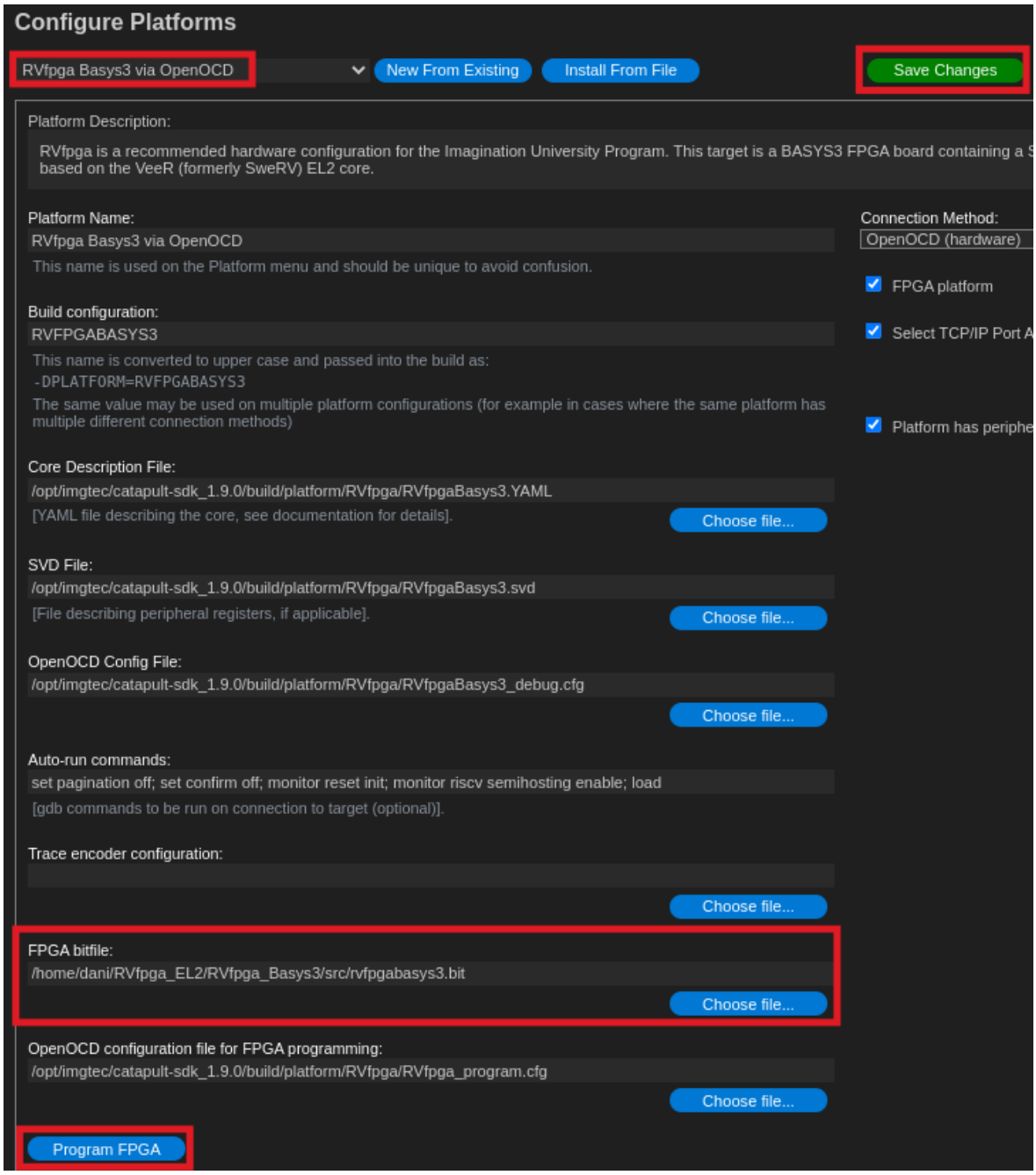


Figure 32: Configure Platforms window

Ensure that “RVfpga Basys3 via OpenOCD” is selected at the top. Now, enter the path for the location of the bitfile that defines RVfpgaEL2-Basys3 into the box labelled “FPGA bitfile”. This path should be:

[RVfpgaBasysPath]/src/rvfpgabasys3.bit

Once you have modified the form, a green “Save Changes” button will appear. Click this button, then click the “Program FPGA” button at the bottom. You should see the output of the programming operation in the Catapult Studio terminal window, which looks like Figure 33.

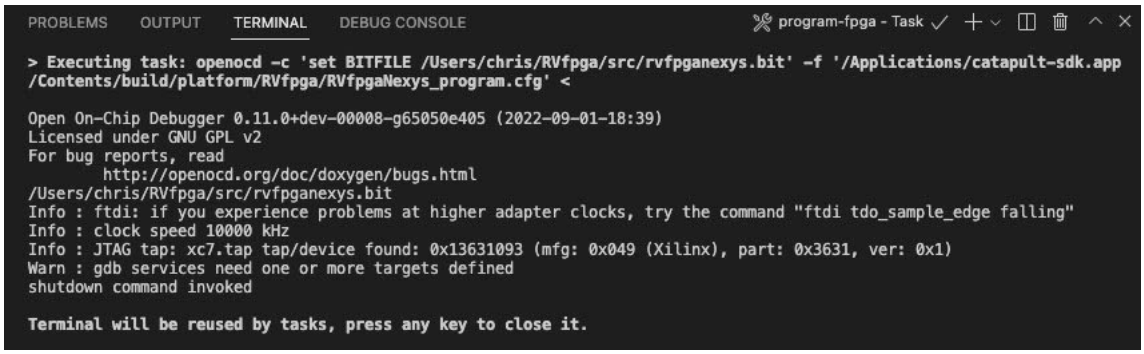


Figure 33: Output from Program FPGA operation

By default, the processor starts fetching instructions at address 0x80000000, where the Boot ROM is placed in our SoC (see Table 5). 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: *[RVfpgaBasysPath]/src/VeeRwolf/BootROM/sw*. In Lab 5, we will show how the Boot ROM is initialized with this program when creating the bitstream.

B. AL_Operations program

The first example program, AL_Operations.S (see Figure 34), 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.

```
1  .globl main
2  main:
3
4  # Register t3 is also called register 28 (x28)
5  li t3, 0x0                    # t3 = 0
6
7  REPEAT:
8      addi t3, t3, 6            # t3 = t3 + 6
9      addi t3, t3, -1           # t3 = t3 - 1
10     andi t3, t3, 3            # t3 = t3 AND 3
11     beq  zero, zero, REPEAT   # Repeat the loop
12     nop
13
14 .end
```

Figure 34. AL_Operations program: AL_Operations.S

Follow these steps to run and debug this code on the Basys3 FPGA board using Catapult Studio:

1. Program the FPGA as explained in the previous section.

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 35).

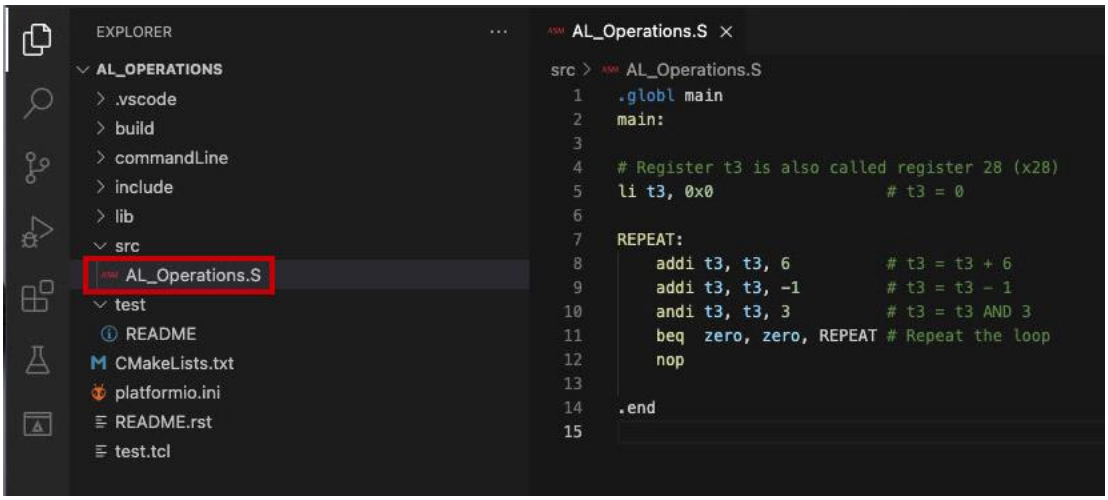


Figure 35. View assembly file AL_Operations.S

3. The program is designed to build using the CMake build system, so CMake controls should be visible in the bar at the bottom (Figure 36). Click build; you should see the build output appearing in the terminal.

Figure 36: Cmake controls

4. Click on the “Run and Debug” button  . Go to the LAUNCH-SELECT section and select the .elf file which you just built (or select “follow cmake target”). See Figure 37.

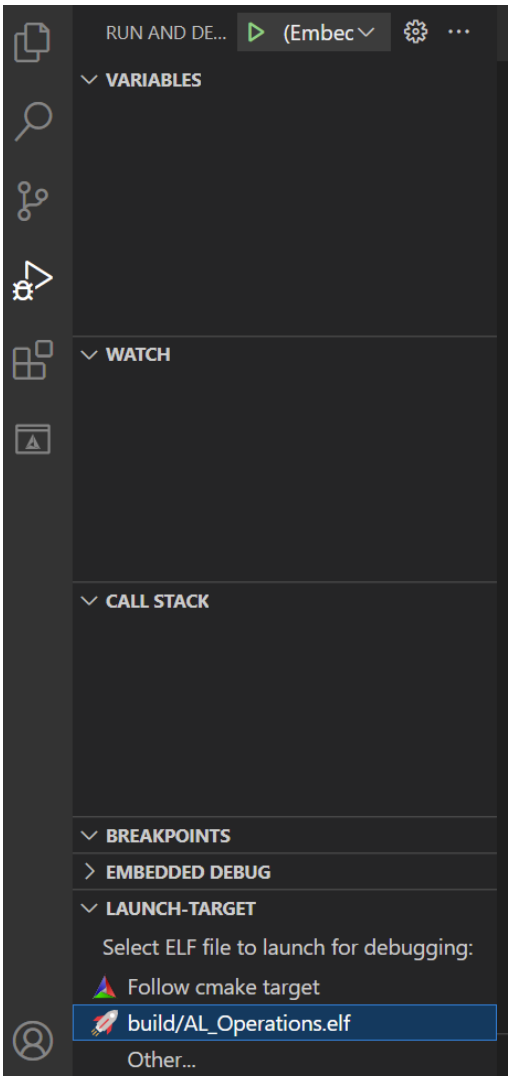


Figure 37. Select the .elf file

5. Start the debugger by clicking on the play button 

(make sure that the ”(Embedded) Attach” option is selected). You can find this button near the top of the window (see Figure 38).

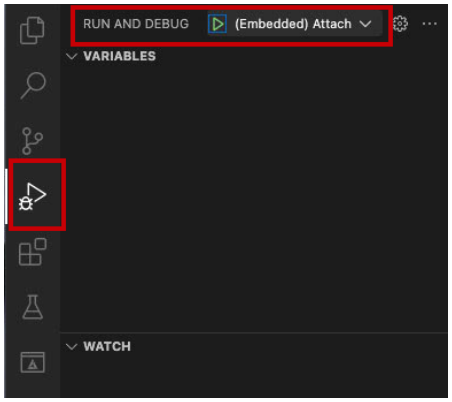


Figure 38. Start debugger

6. Debugging will start. Catapult Studio sets a temporary breakpoint at the beginning of the

main function, so the execution will stop there. To control your debugging session, you can use the debugging toolbar that appears near the top of the editor (see Figure 39). Below are the options:

1. **Continue** executes the program until the next breakpoint.
2. **Breakpoints** can be added by clicking to the left of the line number in the editor.
3. **Step Over** executes the current line and then stop.
4. **Step Into** executes the current line and if the current line includes a function call, it will jump into that function and stop.
5. **Step Out** executes all of the code in the function you are in and then stops once that function returns.
6. **Stop** stops the debugging session and returns to normal editing mode.
7. **Pause** pauses execution. When the program is running, the Continue button is replaced by the Pause button.

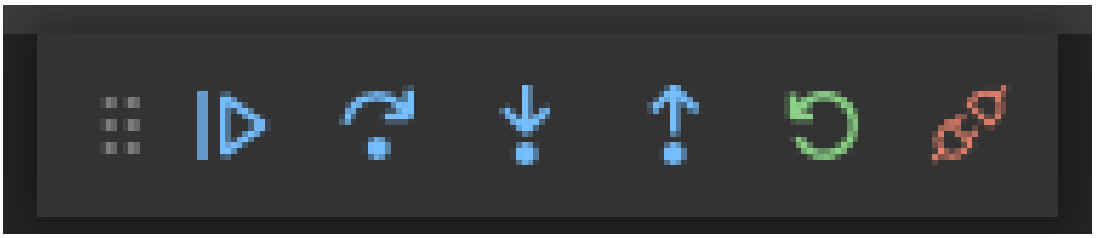


Figure 39. Debugging tools

Note: do not attempt to use the RESTART button. Connection to the target involves a hardware reset, and code will normally expect to start with the hardware in this state. RESTART may not apply this reset. So, it is better to stop, then start a new debugging session, each time you want to re-start your code.

7. 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. Register values are also displayed in this section: both CPU registers and CSRs (Control and Status Registers).
   - **Watch:** in this section you can enter the names of your own variables to watch, or expressions to be evaluated
   - **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.
   - **Embedded Debug:** this section allows you to launch viewers for the target system's memory and peripheral registers. There is also a Vector Plot window which allows you to plot the contents of arrays or other regions of memory.
   - In this section you will also find the **Launch-Target** section which allows you to choose an .elf file for launch and debug.

8. 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 40.

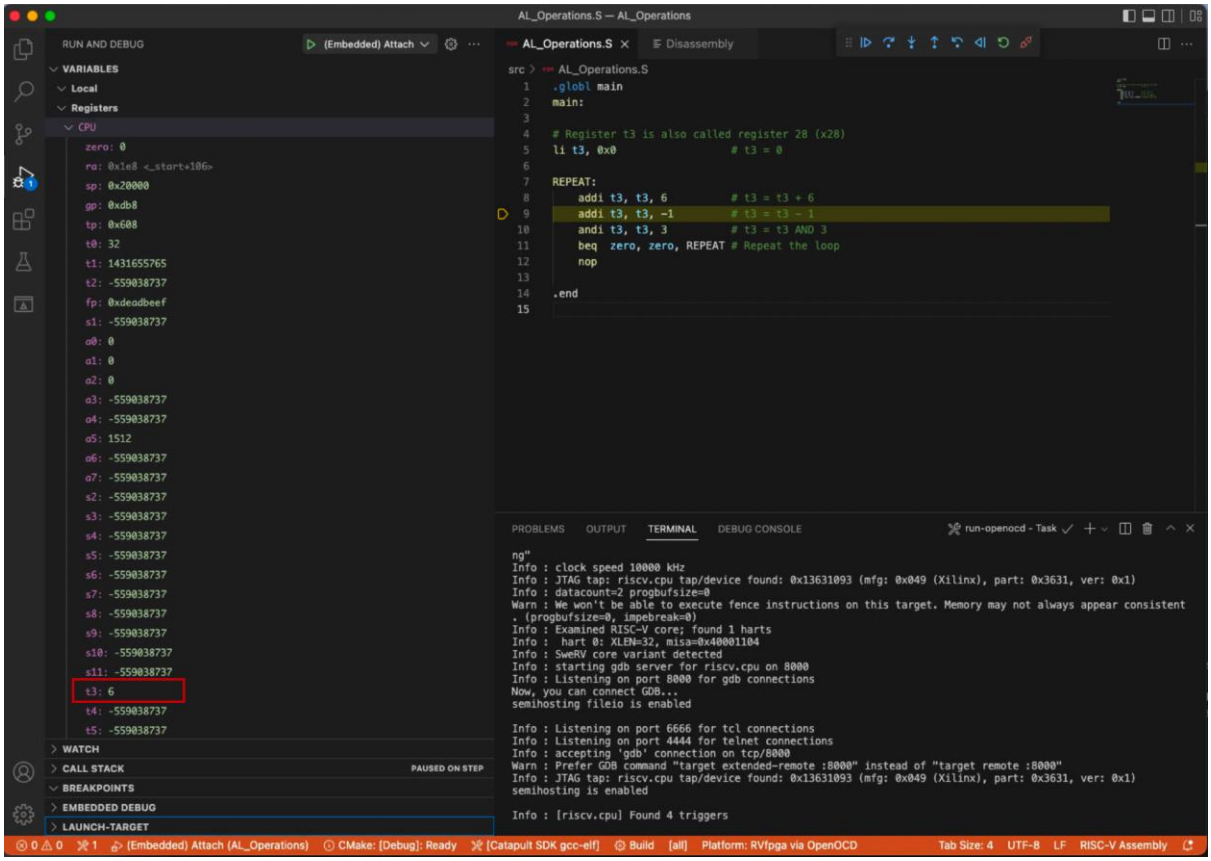


Figure 40. Viewing register contents

9. Before calling the *main* function, start-up code is executed. This is supplied in binary form as part of the C libraries within the SDK. You can examine this code within two files (**rvinit.S** and **crt0.S**) which form part of the picolibc C runtime library code. This code is supplied for reference in the */opt/imgtec/catapult-sdk_1.9.1/share/lib-source/riscv-sdk-picolibc/picocrt/machine/riscv* folder structure within your Catapult SDK installation. This code configures the core: Instruction Cache set-up, registers initialization (such as *sp* or *gp*), etc.

10. We should also highlight that all our projects use a *linker script* to determine the placement of the assembly sections (*text*, *data*, *bss*…) in memory. Catapult SDK contains a program called **ldgen** which is able to auto-generate linker script files, and this is run as part of the build. ldgen is documented in the SDK documentation. You can examine the generated linker script by looking for a file with extension .ld in the build output directory (in the case of this project it is AL_Operations.elf.ld).

11. Finally, stop debugging  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 Basys3 board's right-most LED blink (see Figure 41). The program repeatedly inverts the value connected to the right-most LED with a delay between each inversion.

```
1  #define GPIO_LEDs   0x80001404
2  #define GPIO_INOUT  0x80001408
3
4  #define DELAY 0x100000              /* Define the DELAY */
5
6  .globl main
7  main:
8
9         li x28, 0xFFFF
10        li a0, GPIO_INOUT
11        sw x28, 0(a0)               # Write the Enable Register
12
13        li   t1, DELAY              # Set timer value to control blink speed
14
15        li   t0, 0
16
17 bl1:
18        li   a0, GPIO_LEDs
19        sb   t0, 0(a0)              # Write to LEDs
20        xori t0, t0, 1              # invert LED
21        and  t2, zero, zero         # Reset timer
22
23 time1:                             # Delay loop
24        addi t2, t2, 1
25        bne  t1, t2, time1
26        j    bl1
```

Figure 41. blinky.S

Follow the next steps to run and debug this code on RVfpgaEL2-Basys3, the RISC-V SoC loaded onto the FPGA board:

1. RVfpgaEL2-Basys3 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 RVfpgaEL2-Basys3 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 *[RVfpgaBasysPath]/examples/*

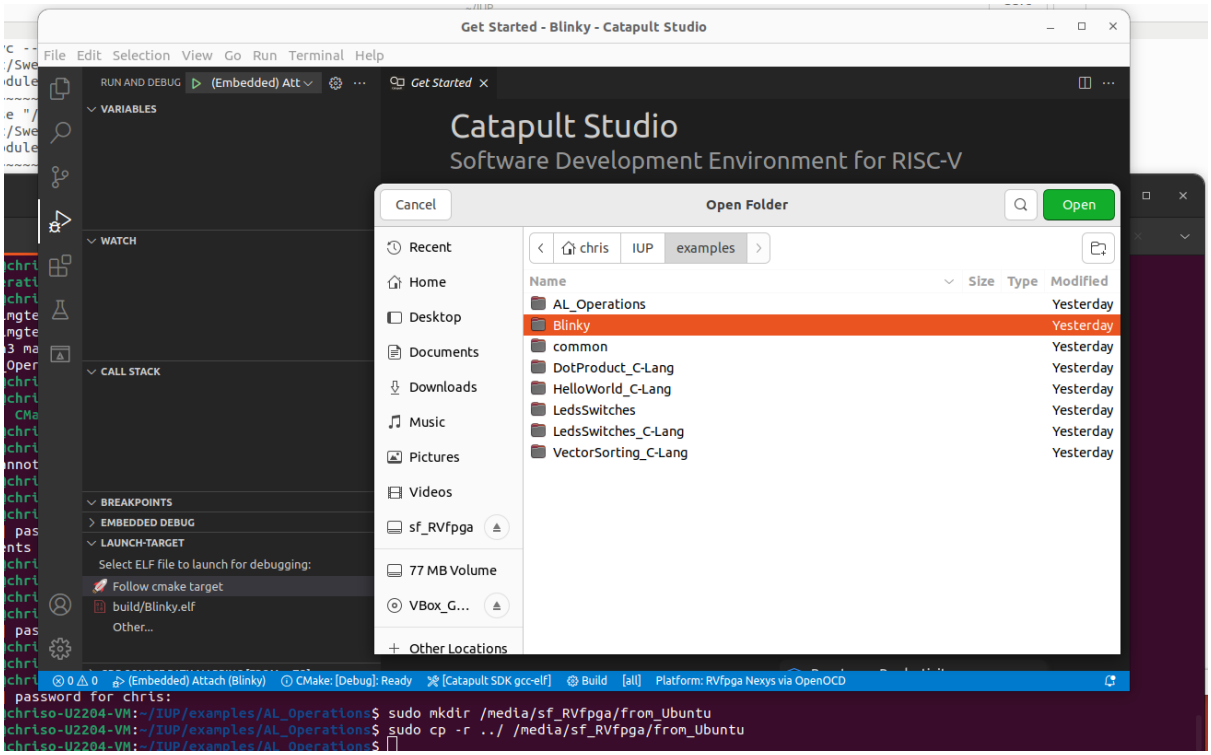


Figure 42. Blinky program folder

3. Select directory *Blinky* and click OK (Figure 42).

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

5. Select “RVfpga Basys3 via OpenOCD” as the platform and “Catapult SDK gcc-elf” for the toolchain. Select “Follow cmake target” in the LAUNCH-TARGET box in the Run and Debug panel (Figure 43).

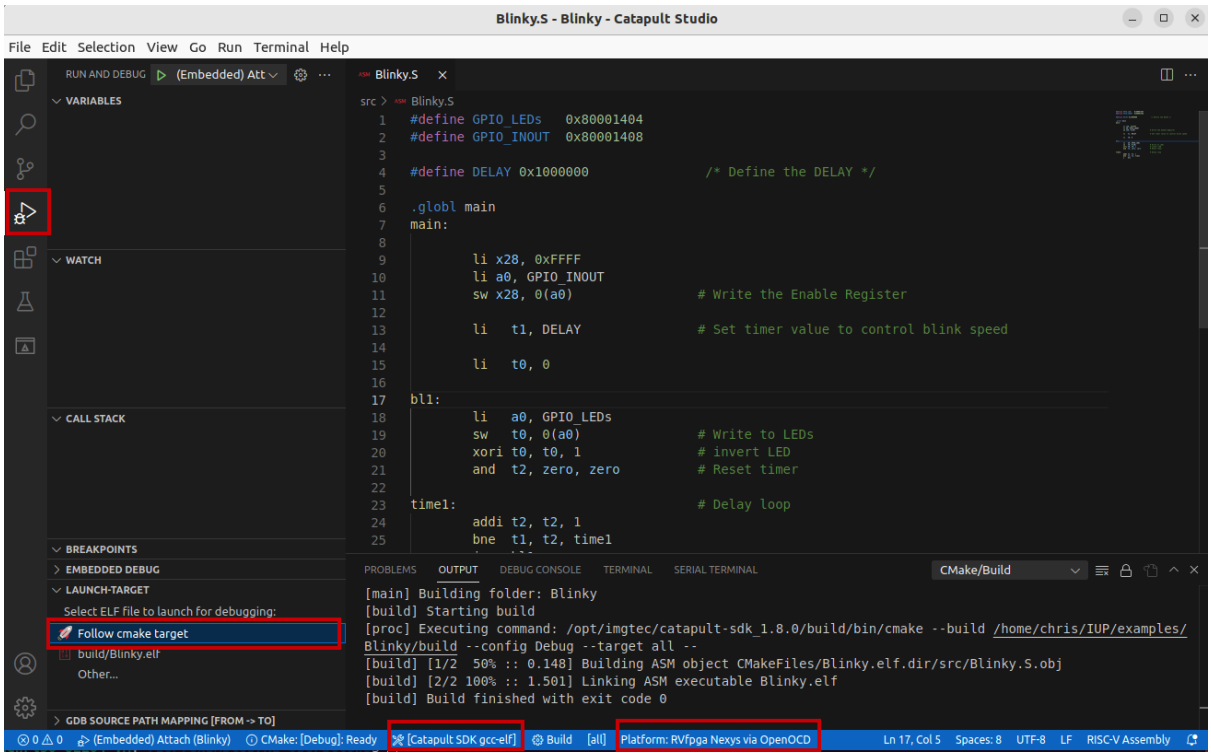


Figure 43. blinky.S in Catapult Studio

6. Click Build on the bottom toolbar and check for successful build output in the terminal window.

7. Click on  to run and debug the program; then start debugging by clicking on the play button  . Catapult Studio sets a temporary breakpoint at the beginning of the main function. So, click on the Continue button  to run the program.

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

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

10. 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 44).

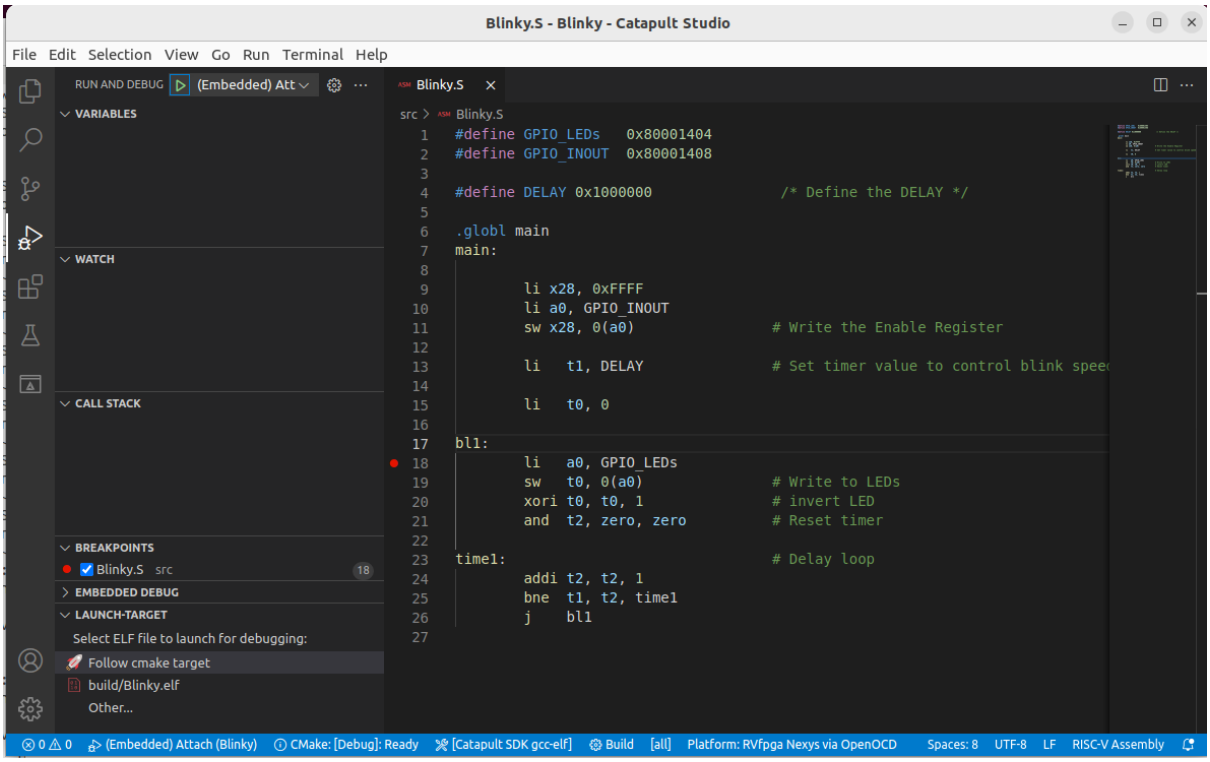


Figure 44. Setting a breakpoint in blinky.S

11. 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.

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

13. 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 45).

```
1  #define GPIO_SWs    0x80001400
2  #define GPIO_LEDs   0x80001404
3  #define GPIO_INOUT  0x80001408
4
5  .globl main
6  main:
7
8  li x28, 0xFFFF
9  li x29, GPIO_INOUT
10 sw x28, 0(x29)                  # Write the Enable Register
11
12 next:
13    li  a1, GPIO_SWs             # Read the Switches
```

```
14    lw  t0, 0(a1)
15
16    li  a0, GPIO_LEDs
17    srl t0, t0, 16
18    sw  t0, 0(a0)               # Write the LEDs
19
20    beq zero, zero, next
21 .end
```

Figure 45. LedsSwitches.S

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

1. RVfpgaEL2-Basys3 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 RVfpgaEL2-Basys3 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 *[RVfpgaBasysPath]/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 46).

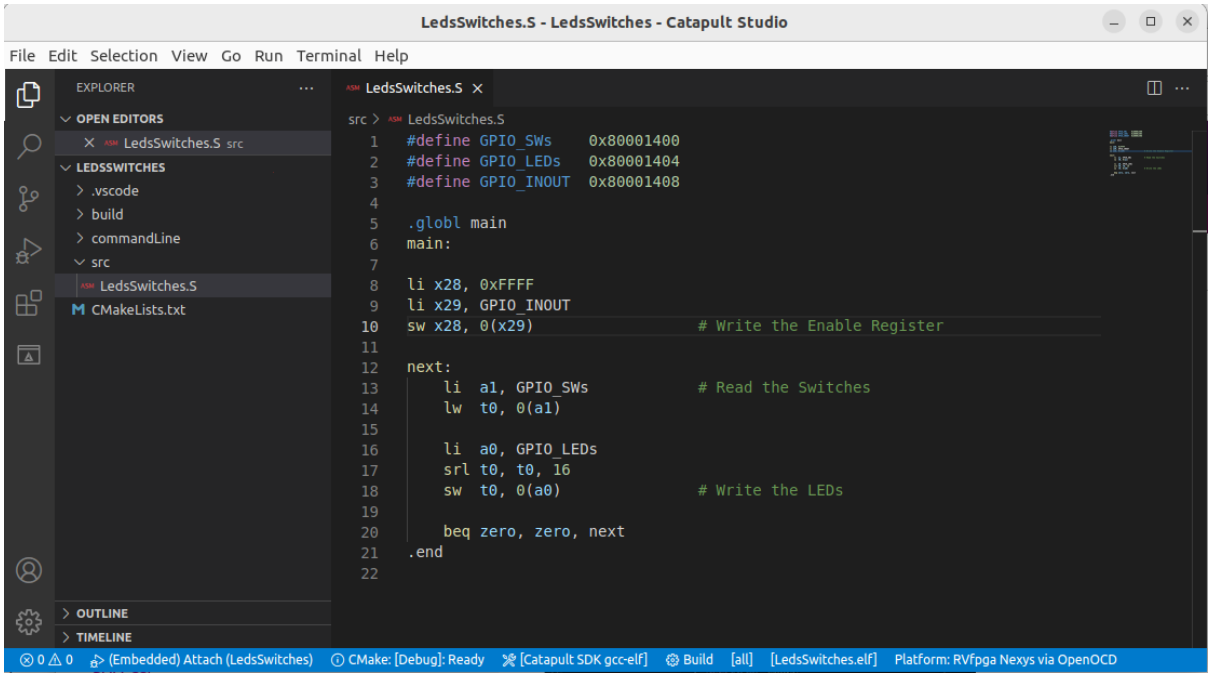


Figure 46. LedsSwitches.S in Catapult Studio

4. After building the program and launching the debugger as explained for prior programs, the program starts to run. Catapult Studio 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 Basys3 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 of memory-mapped peripheral

registers. For that purpose, Catapult Studio provides a Peripheral Registers Display.

a. Pause the execution and step until the beginning of the *next* loop.
b. Press CTRL-Shift-P (CMD-P on Mac) to bring up the Catapult Studio command palette. (Note that the P must be upper-case). Start typing "Catapult" to get a list of Catapult-specific commands, then select "Catapult: New Peripheral Registers Window" (Figure 47).

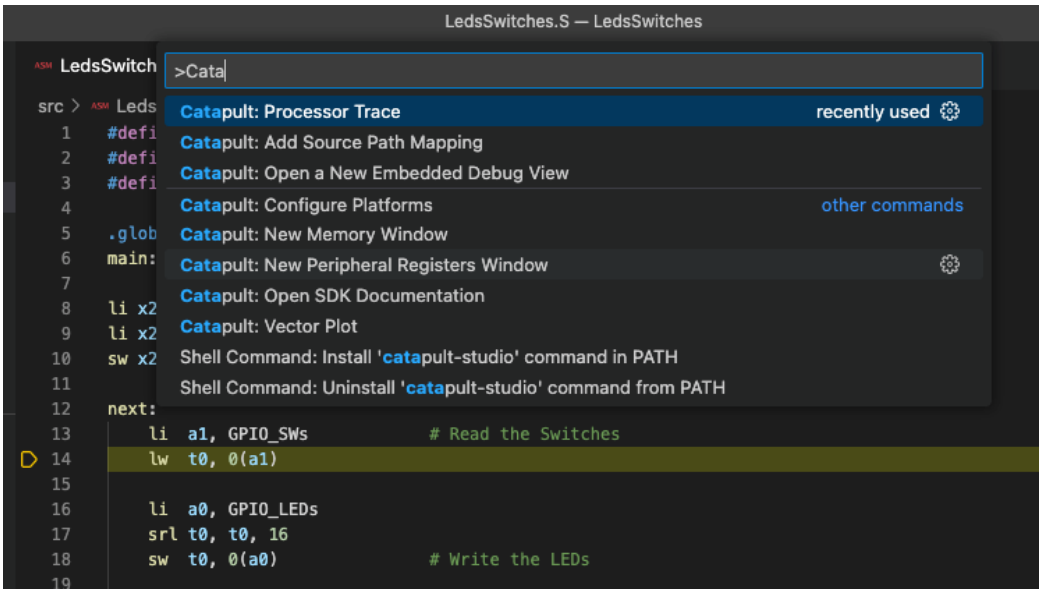


Figure 47: Using the command palette to access embedded debug features

c. In the Peripheral Registers View, open the GPIO section. GPIO stands for "General Purpose Input / Output". This is the section of memory where the switch inputs are mapped (Figure 48).
d. Make the window periodically refresh with a period of 0.5s, by moving the Refresh Period slider.
e. Now toggle some switches. You will see the corresponding values in the 16 most significant bits of the GPIO input data register change.

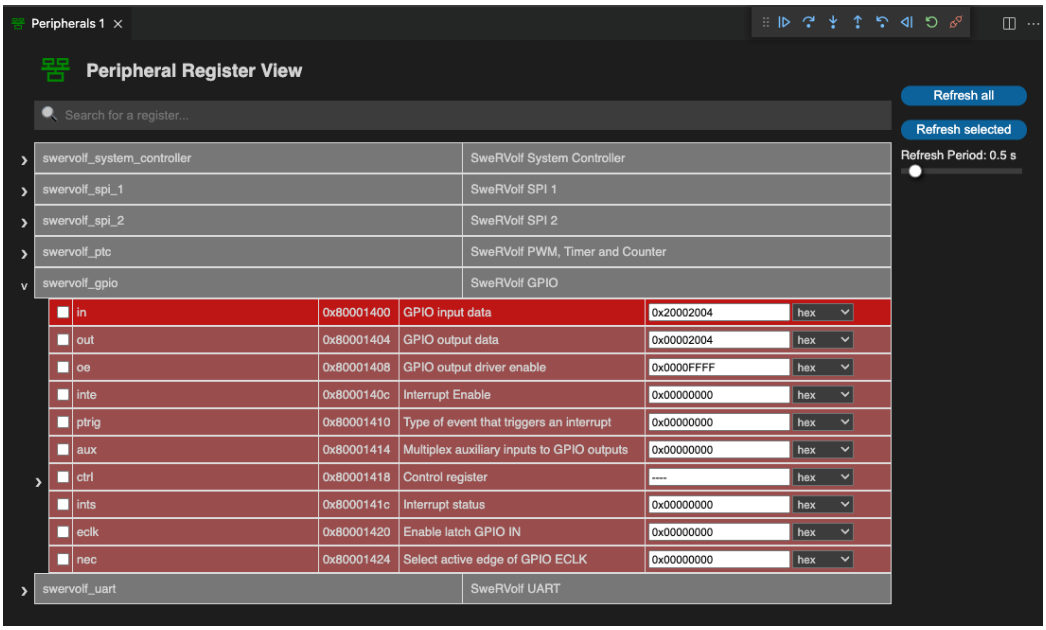


Figure 48: Catapult Studio Peripheral Registers View

f. Run the program one further iteration of the loop. You will see the 16 least significant bits (LSBs) of the GPIO output data change as they are written by the program (this value is also reflected back in the LSBs of the GPIO input data).

2. Catapult Studio also provides two other ways of viewing memory: a general-purpose Memory Window, and a Disassembler view which allows you to view the memory associated with your code.

   a. Go back to the window that shows the code for your program, right-click, and select “Open Disassembly View” (Figure 49).

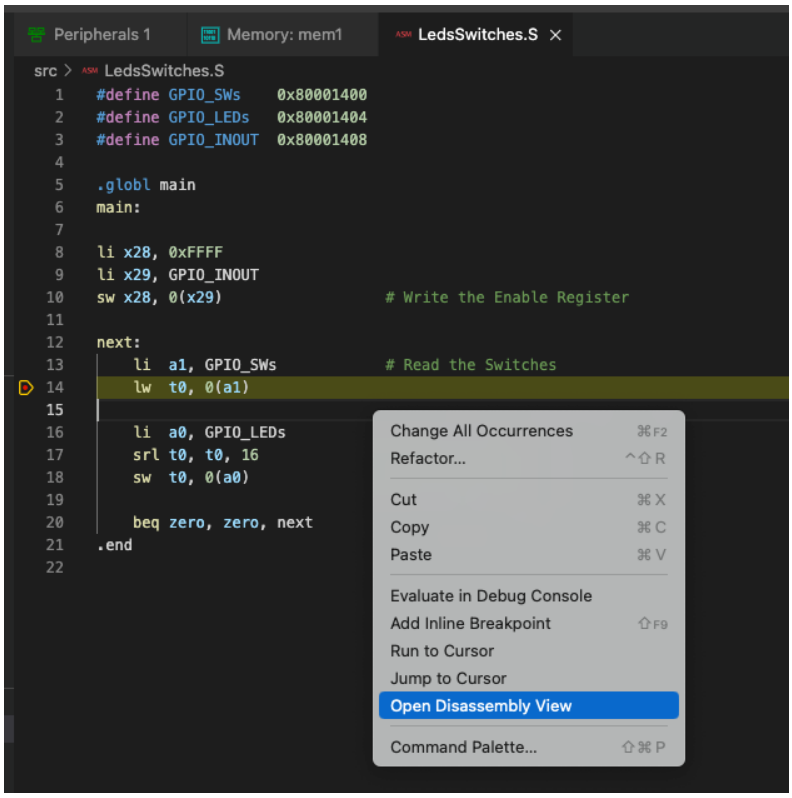


Figure 49: Opening Disassembly View

b. The disassembly window shows the lines of your code in blue, interspersed with the associated binary instructions and, on the left, the addresses where those instructions are stored (Figure 50). Note that you can also set breakpoints in the disassembly window and step through the individual instructions.

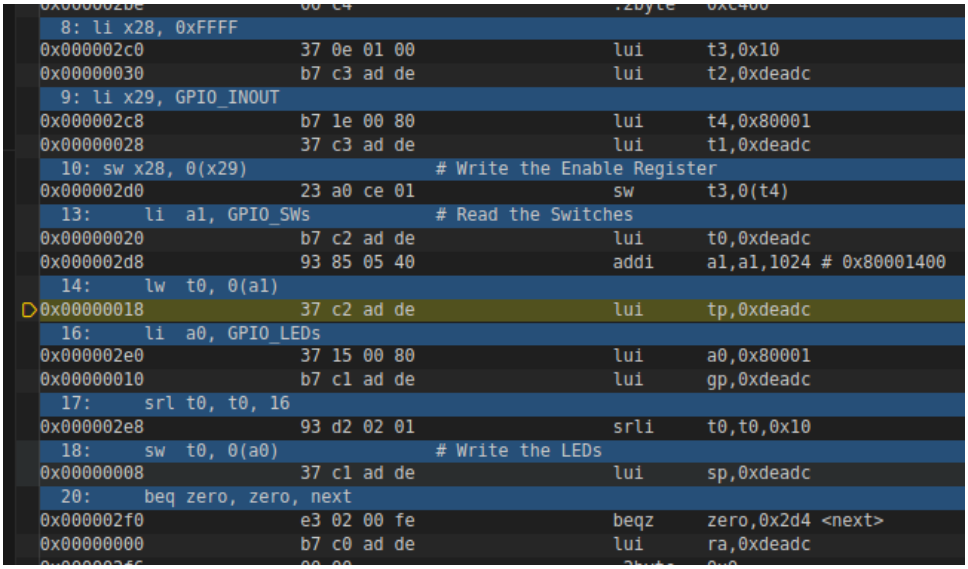


Figure 50: Catapult Studio Disassembly Window

c. From the command palette, select “Catapult: New Memory Window”

d. Choose the address of one of the instructions in your code, as displayed in the Disassembly window, and type it into the Start Address box of the memory window. You should see the same data as shown in the Disassembly Window (Figure 51).

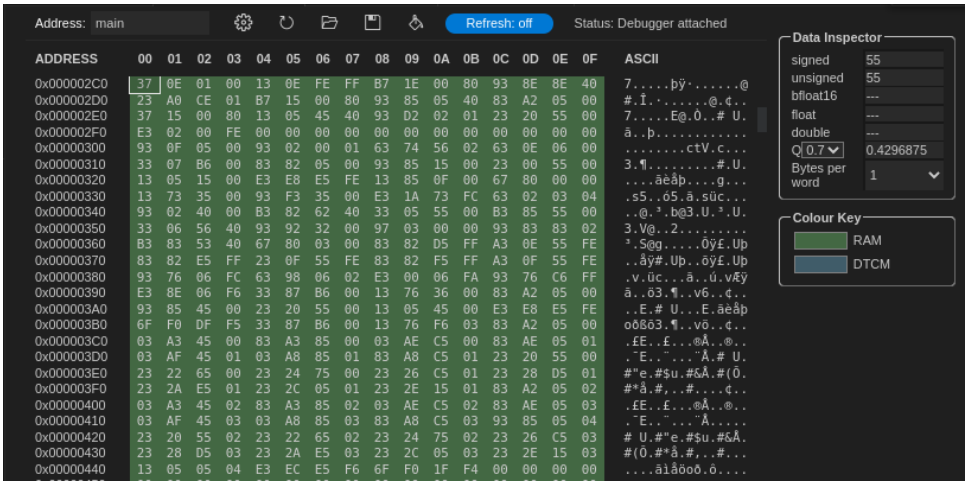


Figure 51: Catapult Studio Memory Window

e. The memory window allows editing of memory contents, and also includes a "Data Inspector" which can interpret memory contents containing common C number formats such as floating point, integer etc. The memory display is colour-coded according to the type of memory (RAM, flash, tightly-coupled memories etc.).

E. LedsSwitches_C-Lang program

Program LedsSwitches_C-Lang.c (Figure 52) does the same as the LedsSwitches.S program shown previously (Figure 45) 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 52. LedsSwitches_C-Lang.c

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

1. RVfpgaEL2-Basys3 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 RVfpgaEL2-Basys3 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 *[RVfpgaBasysPath]/examples/.* Select directory *LedsSwitches_C-Lang* and click OK.

3. Select “RVfpga Basys3 via OpenOCD” as the platform, select “Catapult SDK gcc-elf” for the toolchain, and build the code as for the previous examples. Select “Follow cmake target” in LAUNCH-TARGET.

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

5. Then, start debugging. The program will start executing and will stop at the breakpoint (Figure 53).

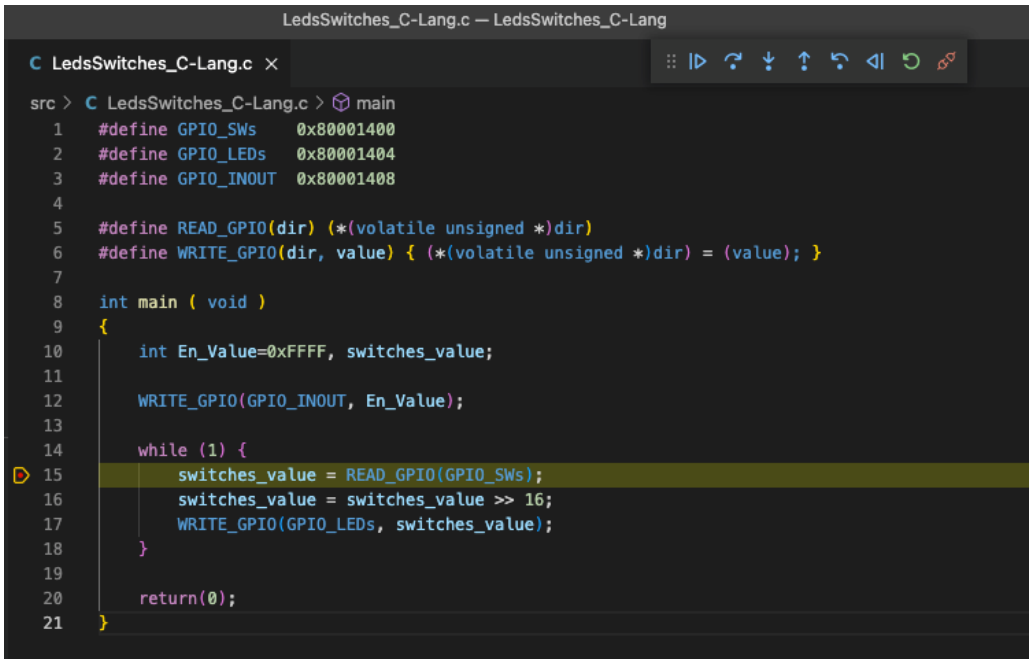


Figure 53. Execution stopped at the breakpoint

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

7. 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 right-clicking on the source file and selecting “Open Disassembly View” (Figure 54).

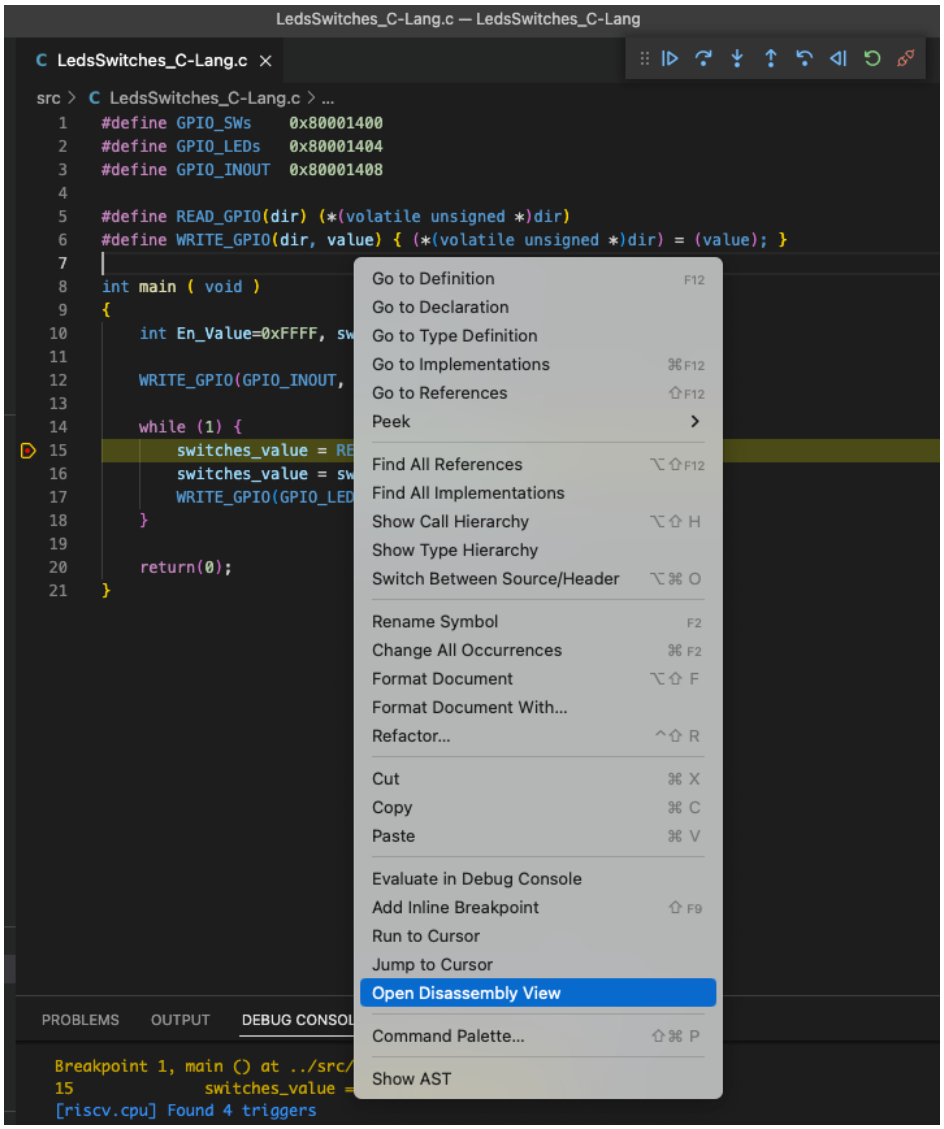


Figure 54. Switch to assembly

8. The program in assembly (Figure 55) first reads the value in the Switches with a load instruction (`lw a5,0(a5)`) and then writes it to the LEDs with a store instruction (`sw a4,0(a5)`). Execute it step by step, change the switches and verify that the LEDs change to reflect the new switch values.

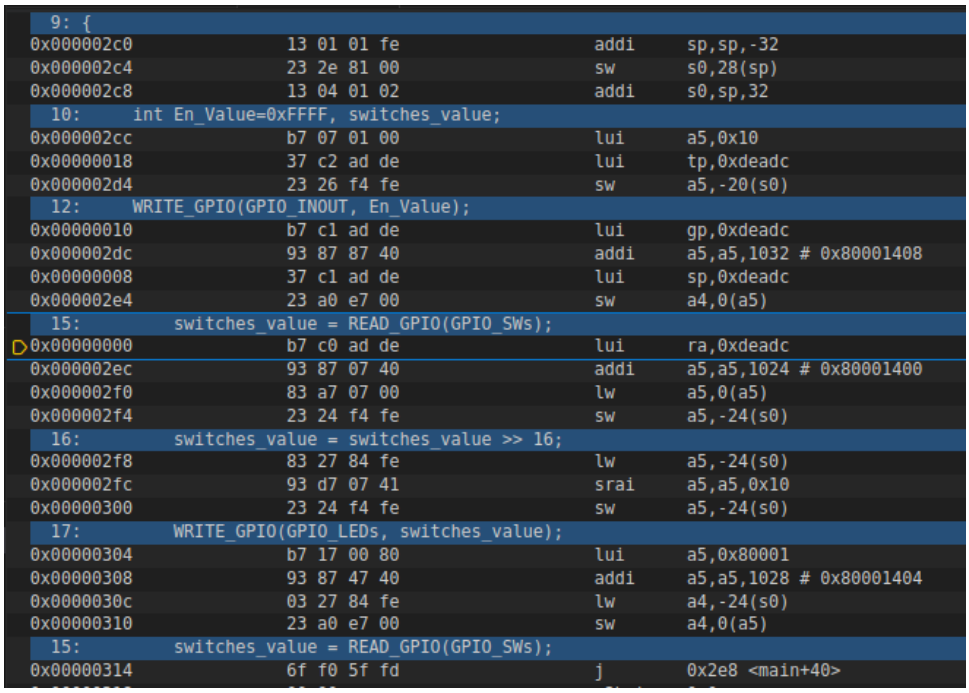


Figure 55. Assembly program

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, Catapult Studio provides its own serial monitor, so here we show how to use this monitor.

First, 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.

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

1. RVfpgaEL2-Basys3 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 RVfpgaEL2-Basys3 onto the board again, do it as explained in Section A, using the HelloWorld_C-Lang example instead of the AL_Operations example.

2. Ensure that the Basys3 board is connected via the micro-USB cable, and open Catapult Studio.

3. Click the SERIAL TERMINAL tab in the terminal section of Catapult Studio, then click the connect icon (Figure 56). This leads you through a set of configuration options to connect to the Basys3 serial port. Firstly you must select the correct serial port from a

drop-down list of serial ports on your system. Then you should select baudRate 115200, dataBits 8, parity none, stopBits 1. This should result in a “Connected Success” message in the terminal window.

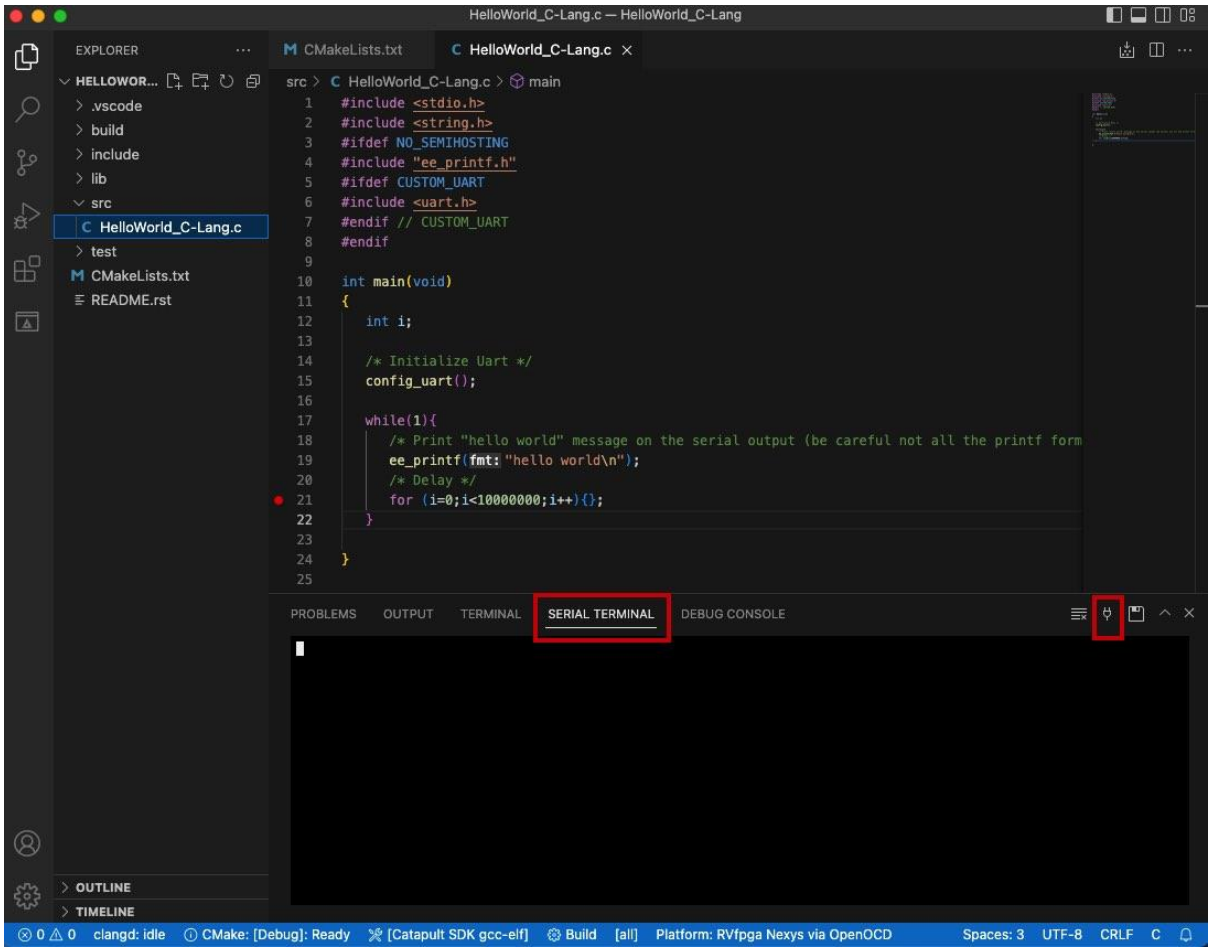


Figure 56: Serial terminal connect icon

4. On the top bar, click on File → Open Folder, and browse to directory *[RVfpgaBasysPath]/examples/*. Select the *HelloWorld_C-Lang* folder and click OK.

5. The program *HelloWorld_C-Lang.C* (Figure 57) initializes the UART (function **config_uart**) and then sends the string through the serial port, using function **ee_printf** (you can find the implementation of these functions in file *[RVfpgaBasysPath]/examples/common/drivers* and *[RVfpgaBasysPath]/examples/common/ee_printf*). It then delays some time before going back to the beginning of the loop.

```c
#include <stdio.h>
#include <string.h>
#ifdef NO_SEMIHOSTING
#include "ee_printf.h"
#ifdef CUSTOM_UART
#include <uart.h>
#endif // CUSTOM_UART
#endif

int main(void)
{
    int i;

    /* Initialize Uart */
    config_uart();

    while(1){
        /* Print "hello world" message on the serial output (be careful not all the printf form
        ee_printf(fmt: "hello world\n");
        /* Delay */
        for (i=0;i<10000000;i++){};
    }

}
```

Figure 57. HelloWorld_C-Lang.C main function

6. Ensure that “RVfpga Basys3 via OpenOCD” is selected for Platform, and “Catapult SDK gcc-elf” is selected for the toolchain. Hit build and check that the program builds successfully.

7. Click on the “Run and Debug” button  . Go to the LAUNCH-SELECT section, and select the .elf file which you just built, inside the build folder (or select “follow cmake target”).

8. Start the debugger by clicking on the play button 

(make sure that the ”(Embedded) Attach” option is selected). You can find this button near the top of the window (see Figure 38). The program will first compile and then debugging will start. Catapult Studio sets a temporary breakpoint at the beginning of the main function, so the execution will stop there.

9. Continue execution by clicking on the Continue button  . The serial monitor repeatedly prints the message *hello world*, as shown in Figure 58.

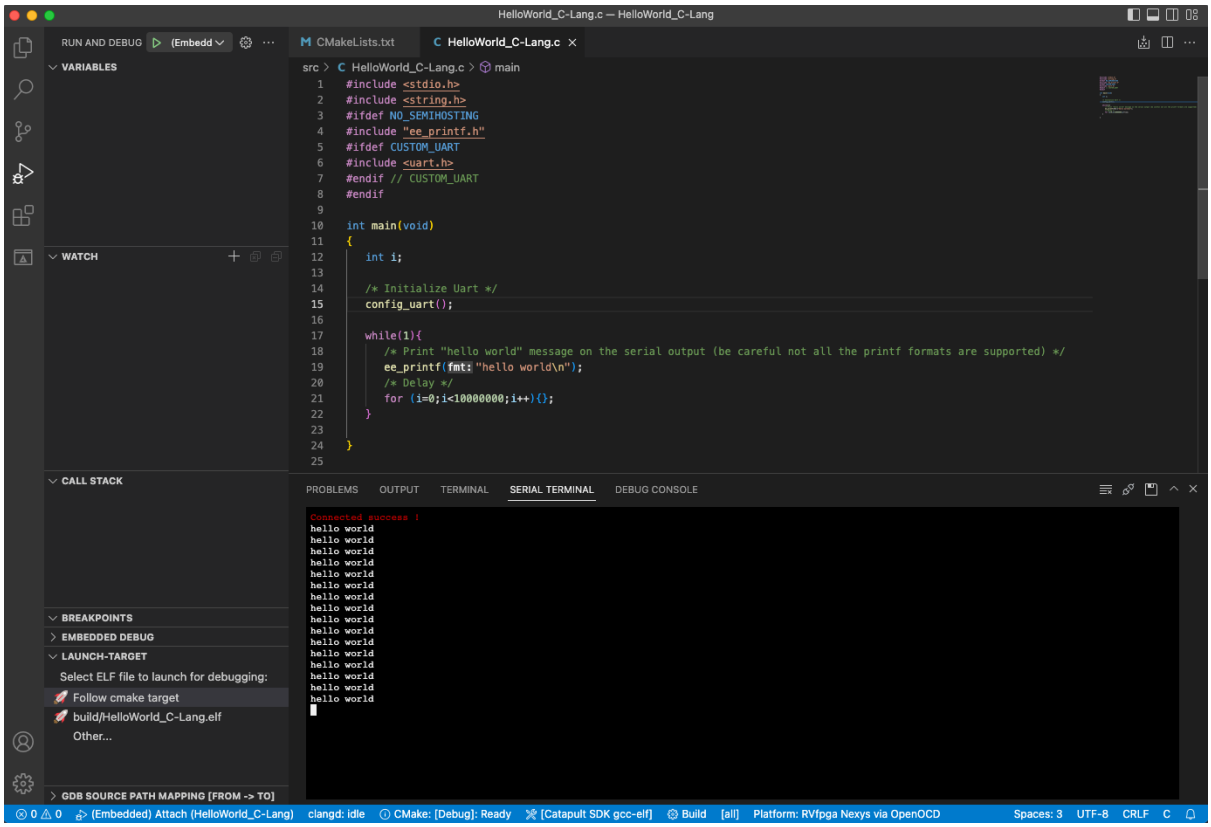


Figure 58. Execution of the program

G. VectorSorting_C-Lang program

In this section 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 59 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;
            }
        }
        B[j]=A[ind];
        A[ind]=0;
    }

    while(1);
}
```

Figure 59. VectorSorting_C-Lang.c

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

1. RVfpgaEL2-Basys3 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 RVfpgaEL2-Basys3 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 *[RVfpgaBasysPath]/examples/.* Select the *VectorSorting_C-Lang* folder and click OK.

3. Ensure that “RVfpga Basys3 via OpenOCD” is selected for Platform, and “Catapult SDK gcc-elf” is selected for the toolchain. Hit "Build” and check that the program builds successfully.

4. Click on the “Run and Debug” button  . Go to the LAUNCH-SELECT section, and select the .elf file which you just built, inside the build folder (or select “follow cmake target”).

5. Place a breakpoint at line 10 and start debugging. The execution will stop at the beginning of the `for` loop (Figure 60). In the WATCH section in the Debugger Side Bar, click the + icon and add the variable A. Do the same for variable B. Expand these to analyse the values of the A and B arrays (highlighted in red in Figure 60).

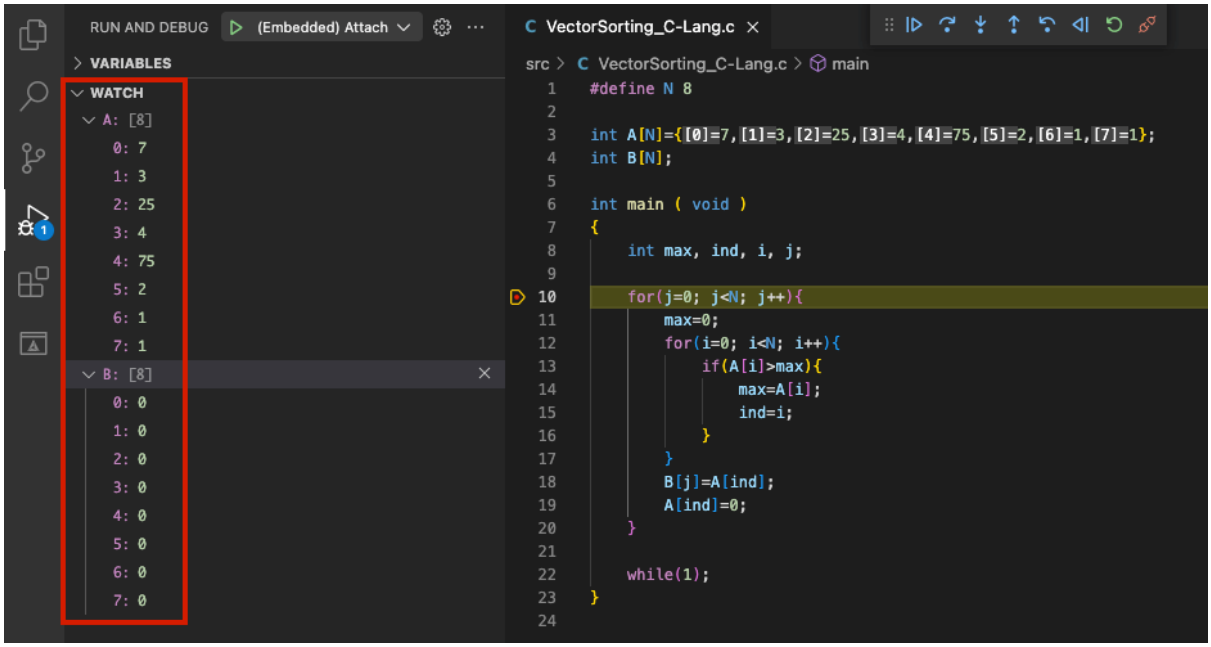


Figure 60. Execution stopped at the beginning of the program

6. Now place another breakpoint at line 18 and continue execution by clicking on  (see Figure 61). Open the Memory Display (as explained for the LedsSwitches program) and type &A (which means “address of A”) into the Start Address box. Select 0.5s refresh rate for the memory window, by dragging the slider. If you wish you can also drag the windows around to see both source and memory windows at the same time (see Figure 61). You can view the initial values of vector A and B.

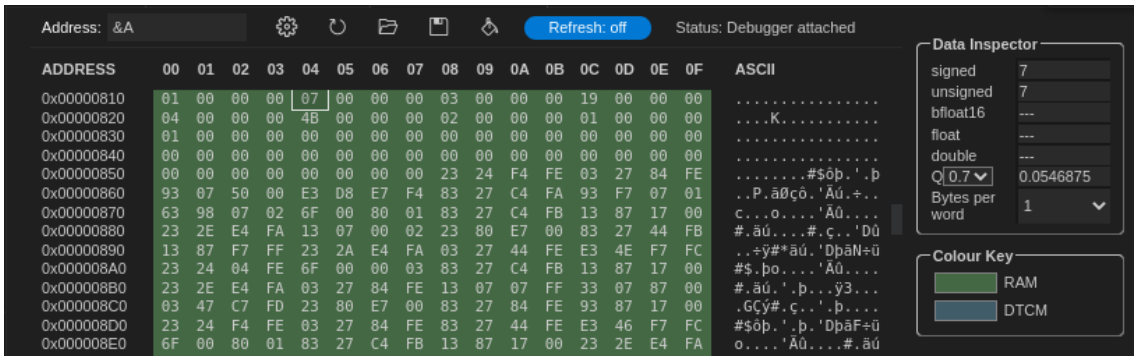


Figure 61. Memory Display for arrays A and B – Initial state.

Click  twice and you will see the first component of B stored in memory. The changed values are highlighted in red in the memory window.

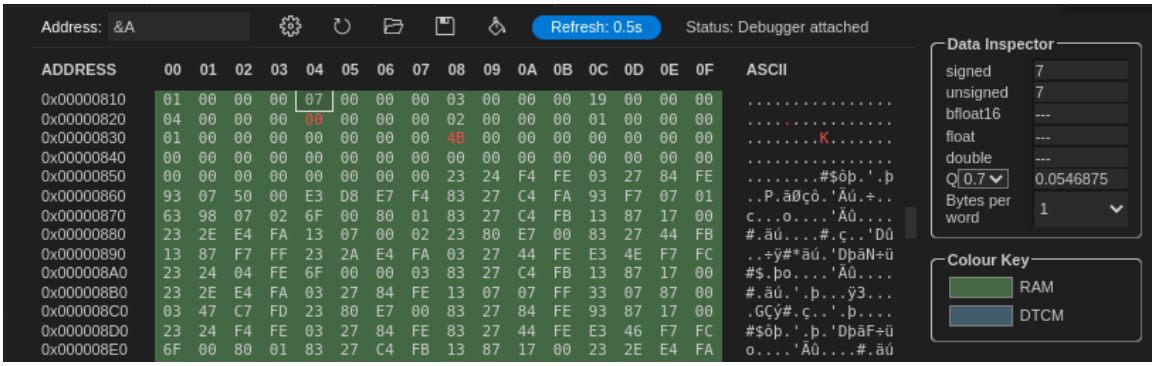


Figure 62. Store the first component of B.

7. 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 63, 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).

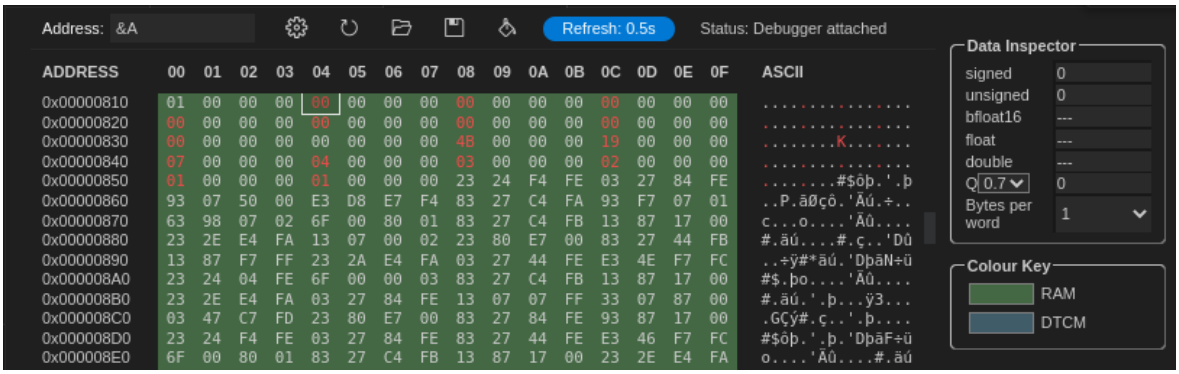


Figure 63. Execution stopped at the end of the program

H. DotProduct_C-Lang program

The last example program, DotProduct_C-Lang.c (Figure 64), 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 64. 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 VeeR EL2 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. RVfpgaEL2-Basys3 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 RVfpgaEL2-Basys3 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 *[RVfpgaBasysPath]/examples/.* Select directory *DotProduct_C-Lang* and click OK.

3. Ensure that “RVfpga Basys3 via OpenOCD” is selected for Platform, and “Catapult SDK gcc-elf” is selected for the toolchain. Hit "Build” and check that the program builds successfully.

4. Click on the “Run and Debug” button  . Go to the LAUNCH-SELECT section, and select the .elf file which you just built, inside the build folder (or select “follow cmake target”).

5. Before calling the debugger, set a breakpoint at line 10 and another one at line 19 (see Figure 65).

6. Then, start debugging. The program will start executing; stop it at the first breakpoint (see Figure 65).

7. On the Debugger sidebar, expand the values of x and y in the VARIABLES section (see Figure 65). The two vectors contain the initial values assigned in *main*. Add dot to the WATCH section. The *dot* variable is initialized to 0.

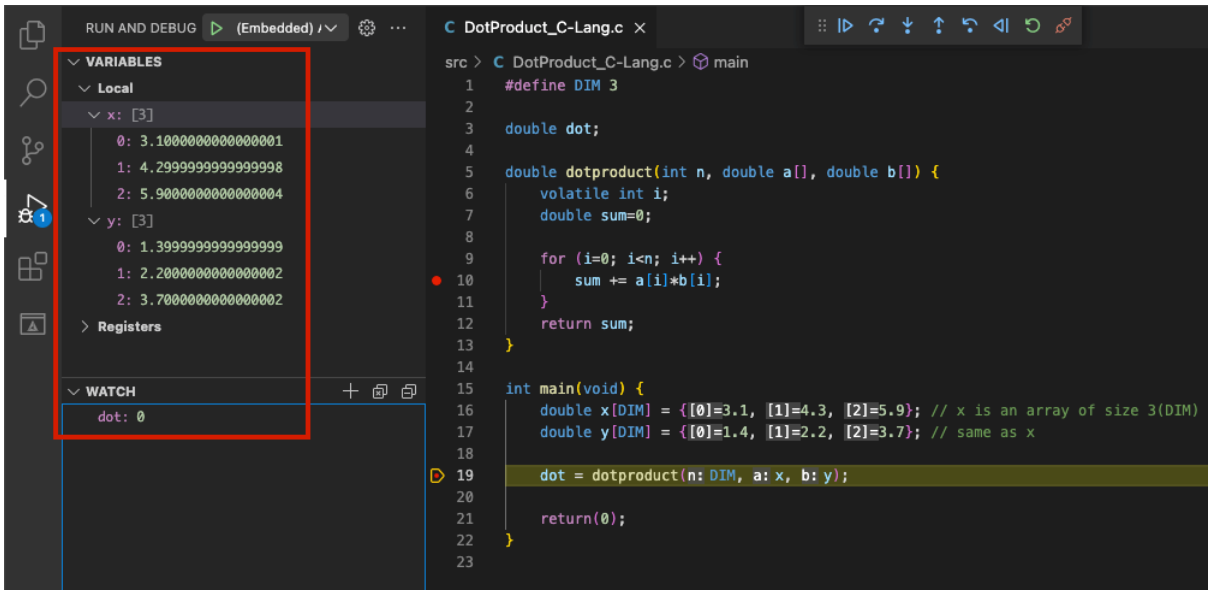


Figure 65. DotProduct_C-Lang program: values of the variables at the first breakpoint

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

9. Switch to assembly (as you did in Figure 54). You can see the floating point emulation routines and analyse them in detail by stepping into them (see Figure 66).

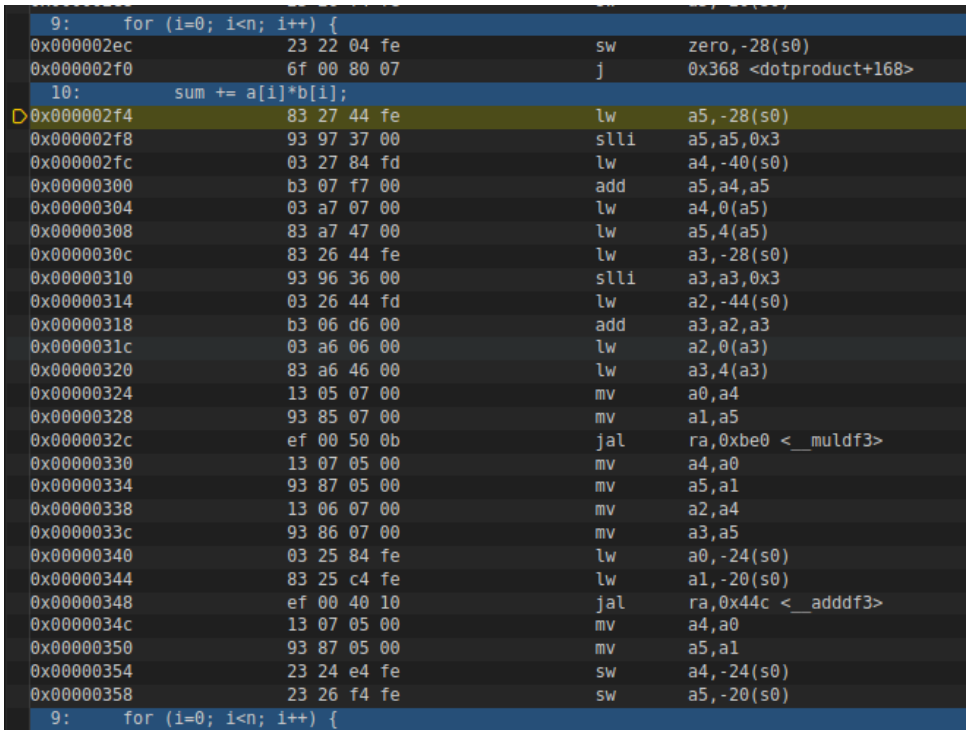


Figure 66. DotProduct_C-Lang program: assembly code of the operation

10. 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 67).

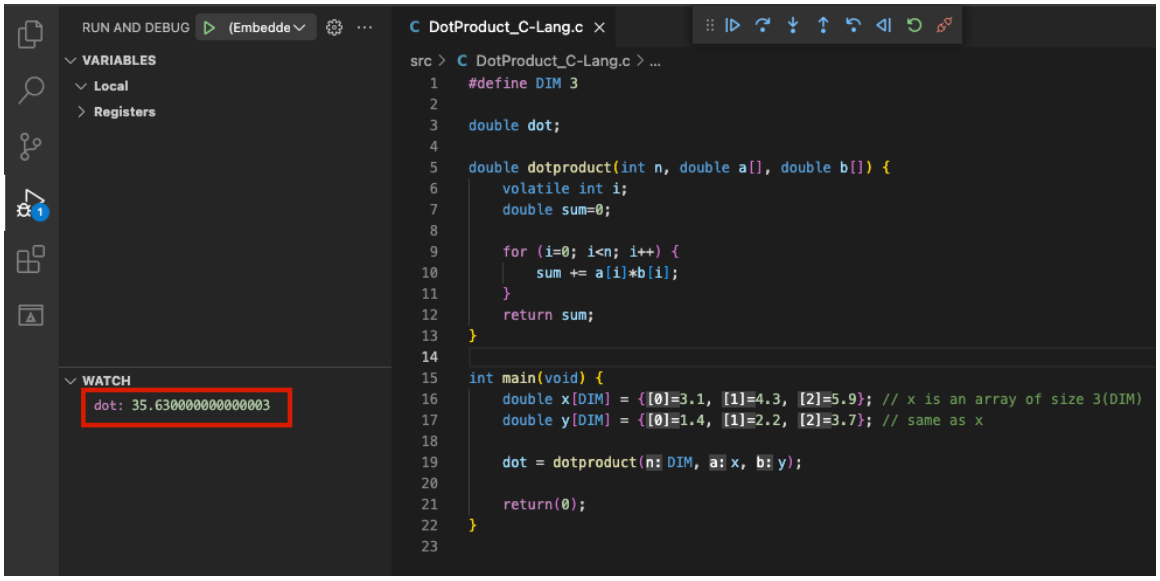


Figure 67. DotProduct_C-Lang program: result of the dot product

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

6. Simulation in RVfpgaEL2-Trace

In this section, you will run the first program used in the previous section (*AL_Operations*) on RVfpgaEL2-Trace using Verilator. Verilator is a hardware description language (HDL) Simulator that simulates the Verilog that defines the SoC (available at *[RVfpgaBasysPath]/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 5 (Figure 34). You will generate the simulation trace 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.

WINDOWS: for running the RVfpgaEL2-Trace simulator in Windows, see Appendix D.

macOS: follow the same instructions as for a Ubuntu OS.

GENERATE THE SIMULATION BINARY, Vrvfpgasim:

NOTE: you can avoid this step and use instead simply use the binary provided at: *[RVfpgaBasysPath]/Simulators/verilatorSIM_Trace/OriginalBinaries*.

Directory *[RVfpgaBasysPath]/Simulators/verilatorSIM_Trace* contains the *Makefile* and the *script* (*rvfpgasim.vc*) for generating the simulator binary for RVfpgaEL2-Trace. 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 *[RVfpgaBasysPath]/src* and at *[RVfpgaBasysPath]/Simulators/SimulationSources*.

We next show how you can generate the binary for RVfpgaEL2-Trace, that later will be used for creating the simulation trace of program *AL-Operations* running on RVfpgaEL2-Trace (note that we also provide pre-built binaries at *[RVfpgaBasysPath]/Simulators/verilatorSIM_Trace/OriginalBinaries*).

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

```
cd [RVfpgaBasysPath]/Simulators/verilatorSIM_Trace/
make clean
make
```

File ***Vrvfpgasim*** (the RVfpgaEL2-Trace simulation binary), should be generated inside directory *[RVfpgaBasysPath]/Simulators/verilatorSIM_Trace/*.

GENERATE THE SIMULATION TRACE, USING Vrvfpgasim:

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

1. Open the *AL_Operations* example in Catapult and build the program as explained in previous sections. This will generate the *.elf* file in the *build* directory.

2. Open a terminal window and go into the *AL_Operations* example:

```
cd [RVfpgaBasysPath]/examples/AL_Operations/commandLine
```

3. Generate a .vh file from your compiled program binary, by executing the following commands:

NOTE: You need to use the RISC-V toolchain. The easiest way is to use the one installed by Catapult and available at: */opt/imgtec/catapult-sdk_1.9.1/bin*.

IMPORTANT: The Makefile uses an environment variable called `CATAPULT_SDK_TOPDIR` to know the location of the toolchain in Catapult. In Linux, you can set that variable by using the following command:

```
export CATAPULT_SDK_TOPDIR="/opt/imgtec/catapult-sdk_1.9.1"
```

Note that you can execute the command every time you want to use RVfpgaEL2 or you can define the variable permanently by editing any of the Linux configuration files (*/etc/profile*, */etc/environment*, *~/.profile*, *~/.bashrc*, etc). Note also that you must use the Catapult SDK directory (`catapult-sdk_1.X.X`) that corresponds to the version that you have installed.

```
make clean
export CATAPULT_SDK_TOPDIR="/opt/imgtec/catapult-sdk_1.9.1"
make AL_Operations.vh
```

This file contains the binary data of the program, in text form, that can be loaded into the simulation memory.

It can also be useful to generate the disassembly file, which you can easily do by running:

```
make AL_Operations.dis
```

4. From the same folder, execute the following command (if you want to use the binary provided inside folder *[RVfpgaBasysPath]/Simulators/verilatorSIM_Trace/OriginalBinaries*, modify the path accordingly):

```
../../../Simulators/verilatorSIM_Trace/Vrvfpgasim
+ram_init_file=AL_Operations.vh +vcd=1
```

This will start running the simulation of the program in RVfpgaEL2-Trace. After a few seconds, quit using CTRL-C. It should have produced a trace file trace.vcd.

5. Now you can open the trace file using *GTKWave*.

```
gtkwave trace.vcd
```

ANALYSE THE SIMULATION TRACE IN GTKWAVE:

1. 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`** → **`VeeRwolf`** → **`rvtop`** → **`veer`**, and click on module **`ifu`** (it will highlight as shown in the Figure 68), 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.

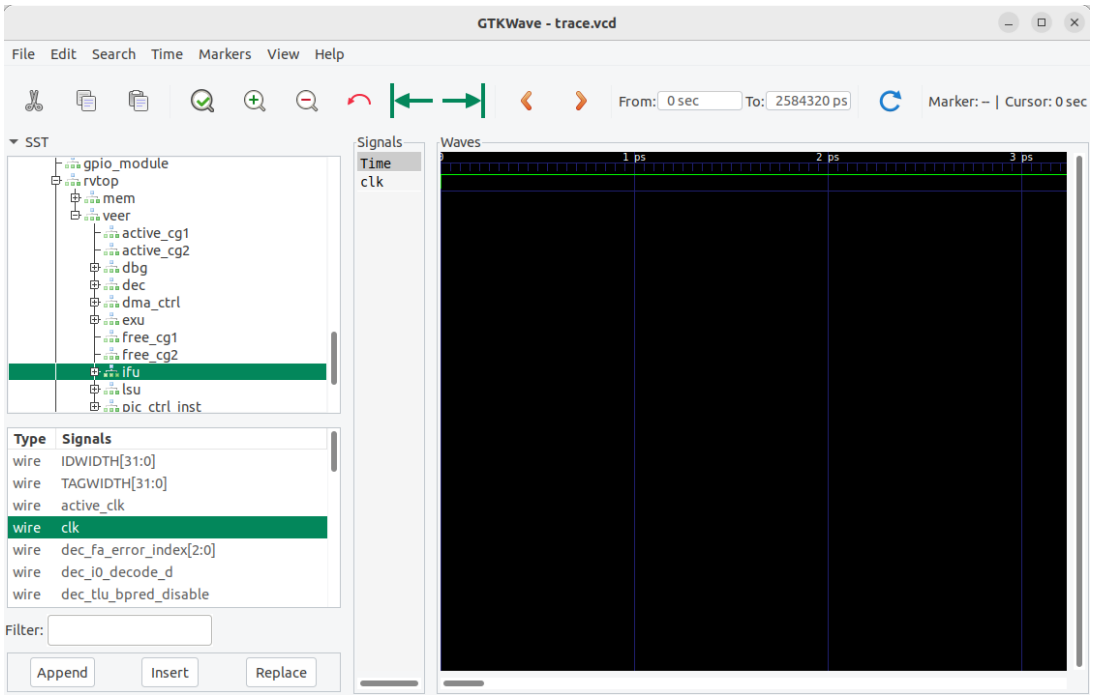


Figure 68. Add signal *clk* to the graph

2. Zoom out several times so that you can view the clock signal change (Figure 69).

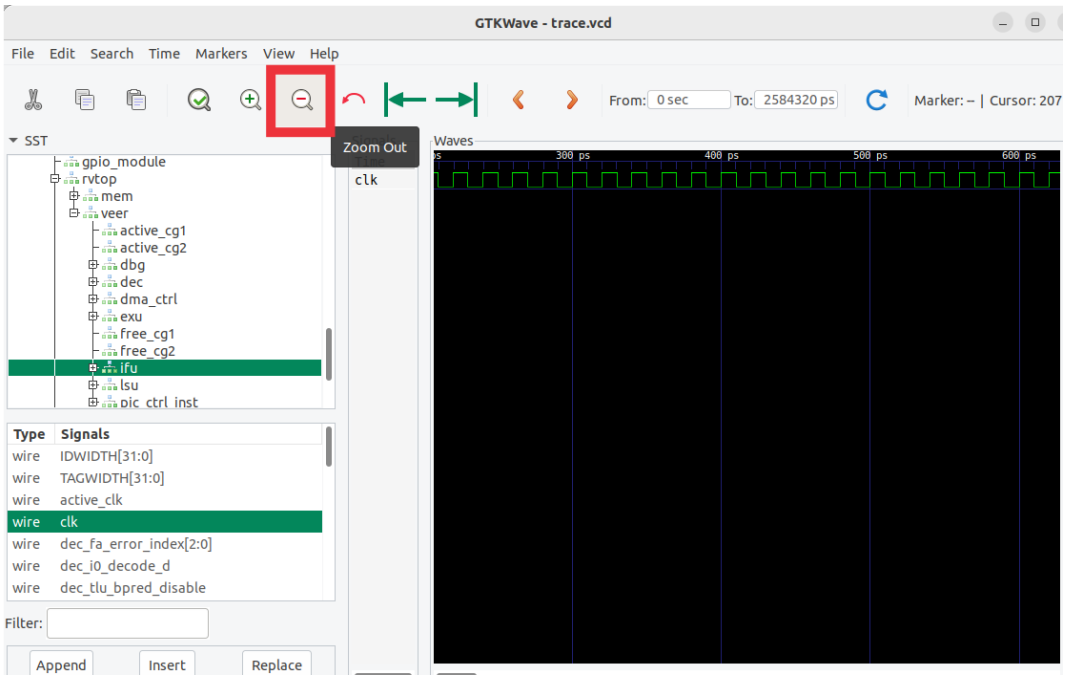


Figure 69. Zoom out

3. Now add the signals that show the instruction that executes in the RISC-V core. In the same module (**`ifu`**) look for signal `ifu_i0_instr[31:0]` (Figure 70), and drag it into the black Waves pane. The prefix `ifu` indicates the instruction fetch unit and `instr[31:0]` indicates the 32-bit instruction.

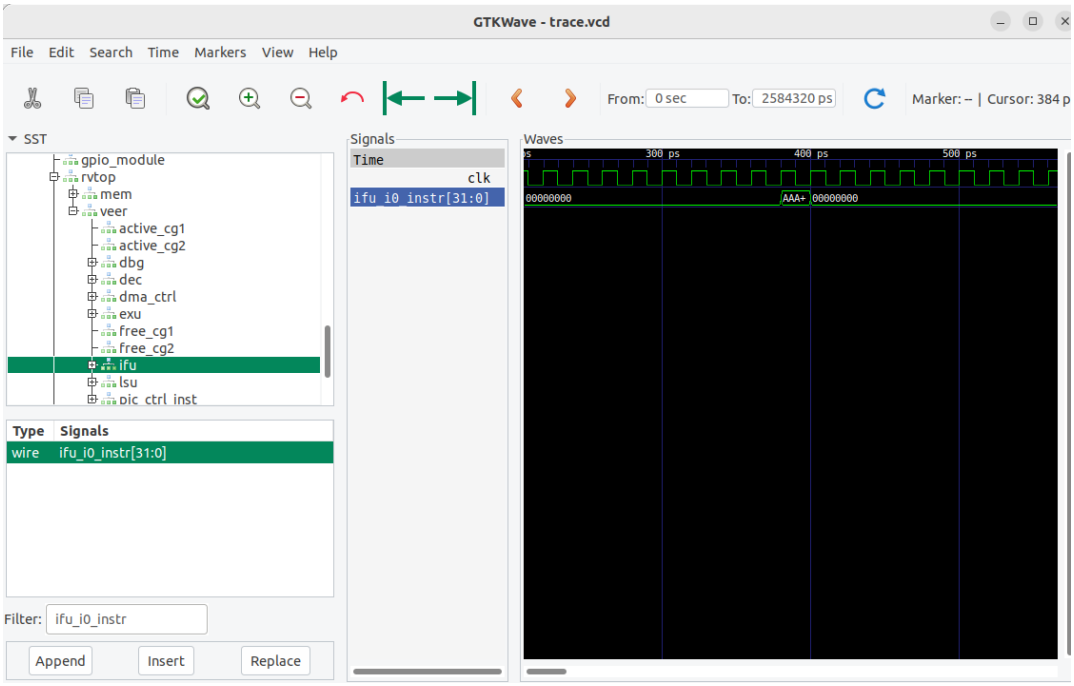


Figure 70. Add signal `ifu_i0_instr[31:0]` to the timing waveform

4. Now add the signal that holds the value of register `t3` (i.e., register number 28, `x28`). Expand the hierarchy under **`veer`** into **`dec`** → **`arf`** → **`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 71).

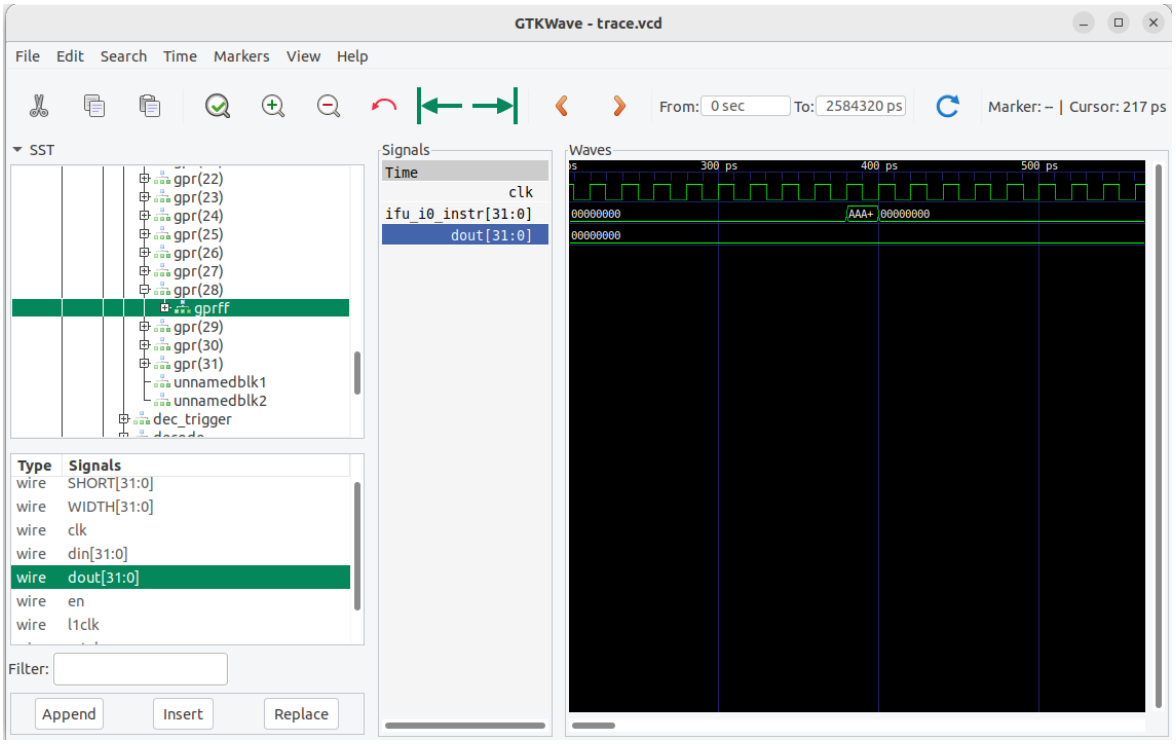


Figure 71. Add signal `dout[31:0]` to the graph

5. Another way of showing signals in GTKWave is to use a *.tcl* file. File *test.tcl* is provided at *[RVfpgaBasysPath]/examples/AL_Operations/commandLine*. 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. See Figure 72.

```
test.tcl
1  gtkwave::addSignalsFromList rvfpgasim.clk
2
3  gtkwave::addSignalsFromList rvfpgasim.veerwolf.rvtop.veer.ifu.ifu_i0_instr
4
5  gtkwave::addSignalsFromList rvfpgasim.veerwolf.rvtop.veer.dec.arf.gpr(28).gprff.dout
```

Figure 72. File *test.tcl*

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

6. Figure 73 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           # 0xFE000AE3
    nop                            # nop                       # 0x00000013
```

Figure 73. 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 74 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   # 0xFE000AE3
            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   # 0xFE000AE3
            ...
```

Figure 74. Instruction flow and values of register `t3` (`x28`) during AL_Operations execution

7. Move to any iteration of the loop and analyse the execution of two random iterations that show the behaviour described in Figure 74. See Figure 75.

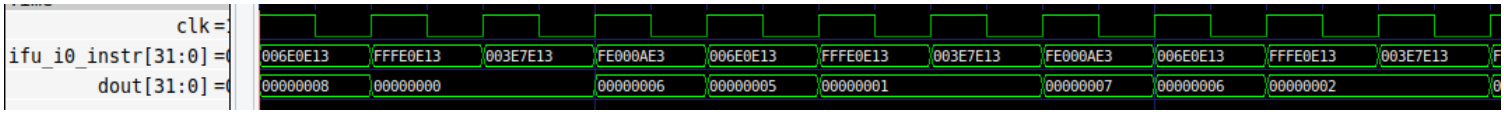


Figure 75. Execution of the three Arithmetic-Logic instructions from the example

7. Simulation in RVfpgaEL2-ViDBo

In this section, you will run the fourth program used in the previous section (*LedsSwitches_C-Lang*) on RVfpgaEL2-ViDBo using Verilator as in the previous section. Verilator is a hardware description language (HDL) Simulator that simulates the Verilog that defines the SoC (available at *[RVfpgaBasysPath]/src*). This way of running the SoC allows you to communicate with the peripherals available in a Virtual Development Board called ViDBo and provided at: https://github.com/olofk/vidbo. In this course we extend the virtual board and adapt it to the RVfpgaEL2 System.

WINDOWS: for running the RVfpgaEL2-ViDBo simulator in Windows, see Appendix D.

macOS: follow the same instructions as for a Ubuntu OS. We must highlight that this simulator may have problems with the libwebsockets library.

GENERATE THE SIMULATION BINARY, Vrvfpgasim:

NOTE: you can avoid this stage and use instead the binary provided at: *[RVfpgaBasysPath]/Simulators/verilatorSIM_ViDBo/OriginalBinaries*.

Directory *[RVfpgaBasysPath]/Simulators/verilatorSIM_ViDBo* contains the *Makefile* and the *script* (*rvfpgasim.vc*) for generating the simulator binary for RVfpgaEL2-ViDBo. 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 *[RVfpgaBasysPath]/src* and at *[RVfpgaBasysPath]/Simulators/SimulationSources*.

We next show how you can generate the binary for RVfpgaEL2-ViDBo, that later will be used for performing the simulation using the Virtual Board (note that we also provide pre-built binaries at *[RVfpgaBasysPath]/Simulators/verilatorSIM_ViDBo/OriginalBinaries*).

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

```
cd [RVfpgaBasysPath]/Simulators/verilatorSIM_ViDBo/
make clean
make
```

   File ***Vrvfpgasim*** (the RVfpgaEL2-ViDBo simulation binary), should be generated inside directory *[RVfpgaBasysPath]/Simulators/verilatorSIM_ViDBo/*.

SIMULATE A PROGRAM USING Vrvfpgasim

Once the simulator binary (*Vrvfpgasim*) has been generated, you will use it for simulating a program with RVfpgaEL2-ViDBo and communicate with some of the peripherals on the board.

1. Open the *LedsSwitches* example in Catapult and build the program as explained in previous sections. This will generate the *.elf* file in the *build* directory.

IMPORTANT: Any program should execute equally in RVfpgaEL2-Basys3 and RVfpgaEL2-ViDBo. There is only something that you must take into account: Given that RVfpgaEL2-Basys3 performs real execution whereas RVfpgaEL2-ViDBo performs only simulation, the latter is much slower than the former. Therefore, when using delays in your programs, you must reduce them significantly for running the programs in RVfpgaEL2-ViDBo.

2. Open a terminal window and go into the *LedsSwitches* example:

```
cd [RVfpgaBasysPath]/examples/LedsSwitches/commandLine
```

3. Generate a .vh file from your compiled program binary, by executing the following commands:

NOTE: You need to use the RISC-V toolchain. The easiest way is to use the one installed by Catapult and available at: */opt/imgtec/catapult-sdk_1.9.1/bin*.

IMPORTANT: The Makefile uses an environment variable called `CATAPULT_SDK_TOPDIR` to know the location of the toolchain in Catapult. In Linux, you can set that variable by using the following command:

```
export CATAPULT_SDK_TOPDIR="/opt/imgtec/catapult-sdk_1.9.1"
```

Note that you can execute the command every time you want to use RVfpgaEL2 or you can define the variable permanently by editing any of the Linux configuration files (*/etc/profile*, */etc/environment*, *~/.profile*, *~/.bashrc*, etc). Note also that you must use the Catapult SDK directory (`catapult-sdk_1.X.X`) that corresponds to the version that you have installed.

```
make clean
export CATAPULT_SDK_TOPDIR="/opt/imgtec/catapult-sdk_1.9.1"
make LedsSwitches.vh
```

This file contains the binary data of the program, in text form, that can be loaded into the simulation memory.

It can also be useful to generate the disassembly file, which you can easily do by running:

```
make LedsSwitches.dis
```

4. From the same folder, execute the following command (if you want to use the binary provided inside folder *[RVfpgaBasysPath]/Simulators/verilatorSIM_ViDBo/OriginalBinaries*, modify the path accordingly):

```
../../../Simulators/verilatorSIM_ViDBo/Vrvfpgasim
+ram_init_file=LedsSwitches.vh
```

This will start running the simulation of the program in RVfpgaEL2-ViDBo.

5. Open a new terminal window, and execute the python server by running the following commands:

   - `cd [RVfpgaBasysPath]/Simulators/verilatorSIM_ViDBo`

- `python3 -m http.server --directory Basys3Board/`

6. Finally, open a browser and connect to http://localhost:8000/basys3.html. Click on Connect to the board and test the program. In the executed program, when you click on any of the 16 switches, its state will invert as well as the state of the corresponding LED (see Figure 76). As we will see later, we can execute any other program in the course.

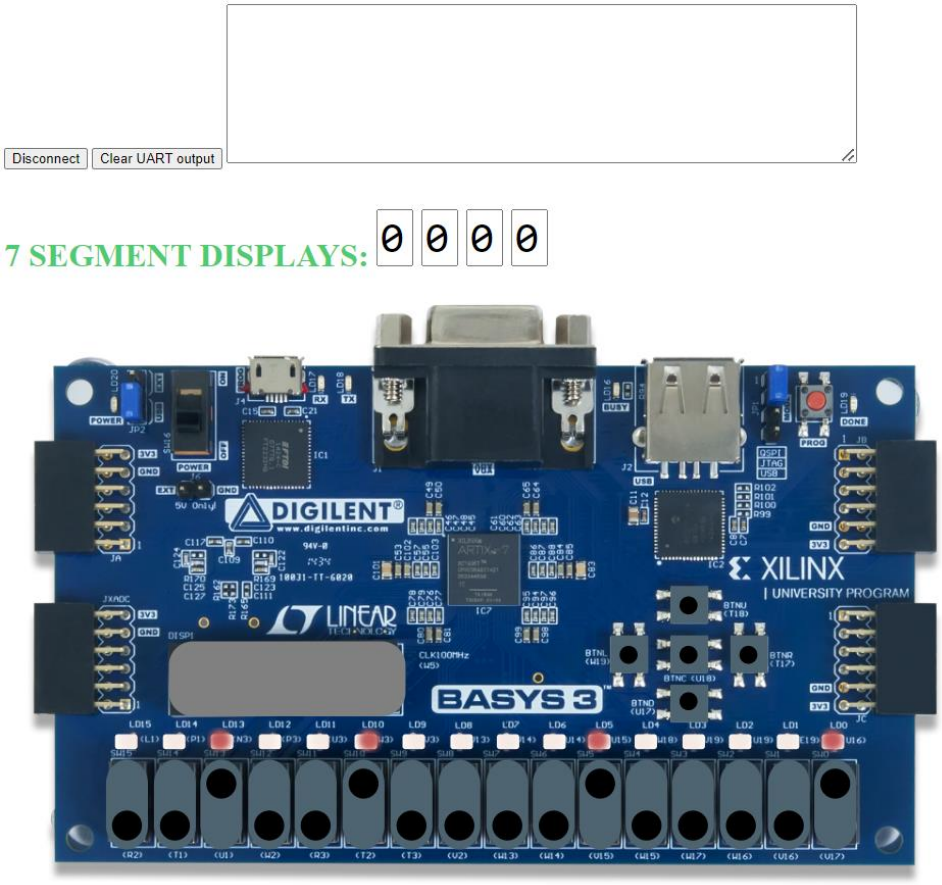


Figure 76. RVfpgaEL2-ViDBo executing the LedsSwitches program

NOTE: If you want to modify the size of the board shown in the browser, you can do it very easily by changing the width and height values in file:

[RVfpgaBasysPath]/Simulators/verilatorSIM_ViDBo/Basys3Board/basys3.html

See the following figure:

```
    <object id="basys3-svg" data="basys3-export.svg" type="image/svg+xml" width="80%" height="80%" object-fill=contain>
</body>
```

8. Simulation in RVfpgaEL2-Pipeline

In this section, you will run the first program used in the previous section (*AL_Operations*) on RVfpgaEL2-Pipeline using Verilator as in the previous section. Verilator is a hardware description language (HDL) Simulator that simulates the Verilog that defines the SoC (available at *[RVfpgaBasysPath]/src*). This way of running the SoC allows you to visualize the VeeR EL2 pipeline while executing instructions.

WINDOWS: for running the RVfpgaEL2-Pipeline simulator in Windows, see Appendix D.

macOS: follow the same instructions as for a Ubuntu OS.

GENERATE THE SIMULATION BINARY, Vrvfpgasim:

NOTE: you can avoid this stage and use instead the binary provided at: *[RVfpgaBasysPath]/Simulators/verilatorSIM_Pipeline/OriginalBinaries*.

Directory *[RVfpgaBasysPath]/Simulators/verilatorSIM_Pipeline* contains the *Makefile* and the *script* (*rvfpgasim.vc*) for generating the simulator binary for RVfpgaEL2-Pipeline. 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 *[RVfpgaBasysPath]/src* and at *[RVfpgaBasysPath]/Simulators/SimulationSources*.

We next show how you can generate the binary for RVfpgaEL2-Pipeline, that later will be used for performing the simulation using the Virtual Board (note that we also provide pre-built binaries at *[RVfpgaBasysPath]/Simulators/verilatorSIM_* Pipeline*/OriginalBinaries*).

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

```
cd [RVfpgaBasysPath]/Simulators/verilatorSIM_Pipeline/
make clean
make
```

   File ***Vrvfpgasim*** (the RVfpgaEL2-Pipeline simulation binary), should be generated inside directory *[RVfpgaBasysPath]/Simulators/verilatorSIM_*Pipeline/.

SIMULATE A PROGRAM USING Vrvfpgasim

Once the simulator binary (*Vrvfpgasim*) has been generated, you will use it for simulating a program with RVfpgaEL2-Pipeline and communicate with some of the peripherals on the board.

1. This simulator executes the AL_Operations program (or any other program that we want to analyse) and stops execution at the point where it reaches a control instruction, which in our simulator is set to instruction `and zero, t4, t5` (note that this instruction does not change the architectural state of the processor, as the destination register, register zero, is hard-wired to the integer value 0 in the RISC-V architecture). From that point we can simulate the program cycle by cycle and analyze some selected core internal signals

as instructions progress through the VeeR EL2 pipeline.
Insert the control instruction in the program by editing file *src/AL_Operations.S*. Place the control instruction right before the end of the loop as shown in Figure 77.

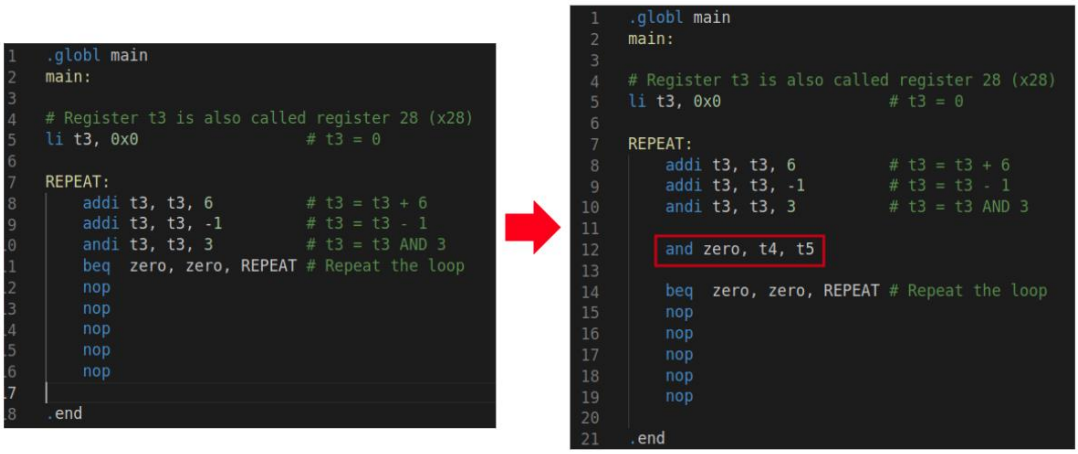


Figure 77. Insert control instruction

2. Open the *AL_Operations* example in Catapult and build the program as explained in previous sections. This will generate the *.elf* file in the *build* directory.

3. Open a terminal window and go into the *AL_Operations* example:

```
cd [RVfpgaBasysPath]/examples/AL_Operations/commandLine
```

4. Generate a .vh file from your compiled program binary, by executing the following commands:

NOTE: You need to use the RISC-V toolchain. The easiest way is to use the one installed by Catapult and available at: */opt/imgtec/catapult-sdk_1.9.1/bin*.

IMPORTANT: The Makefile uses an environment variable called `CATAPULT_SDK_TOPDIR` to know the location of the toolchain in Catapult. In Linux, you can set that variable by using the following command:

```
export CATAPULT_SDK_TOPDIR="/opt/imgtec/catapult-sdk_1.9.1"
```

Note that you can execute the command every time you want to use RVfpgaEL2 or you can define the variable permanently by editing any of the Linux configuration files (*/etc/profile*, */etc/environment*, *~/.profile*, *~/.bashrc*, etc). Note also that you must use the Catapult SDK directory (`catapult-sdk_1.X.X`) that corresponds to the version that you have installed.

```
make clean
export CATAPULT_SDK_TOPDIR="/opt/imgtec/catapult-sdk_1.9.1"
make AL_Operations.vh
```

This file contains the binary data of the program, in text form, that can be loaded into the simulation memory.

It can also be useful to generate the disassembly file, which you can easily do by running:

```
make AL_Operations.dis
```

5. From the same folder, execute the following command (if you want to use the binary provided inside folder *[RVfpgaBasysPath]/Simulators/verilatorSIM_Pipeline/OriginalBinaries*, modify the path accordingly):

```
../../../Simulators/verilatorSIM_Pipeline/Vrvfpgasim
+ram_init_file=AL_Operations.vh
```

This will start running the simulation of the program in RVfpgaEL2-Pipeline. A new window will open that shows the VeeR EL2 pipeline with a selection of signals for each of the 4 stages (see Figure 78). Click several times on the + 1 Cycle" button in the bottom right corner until you reach the second iteration of the loop, and then analyze the execution of the AL_Operations program.

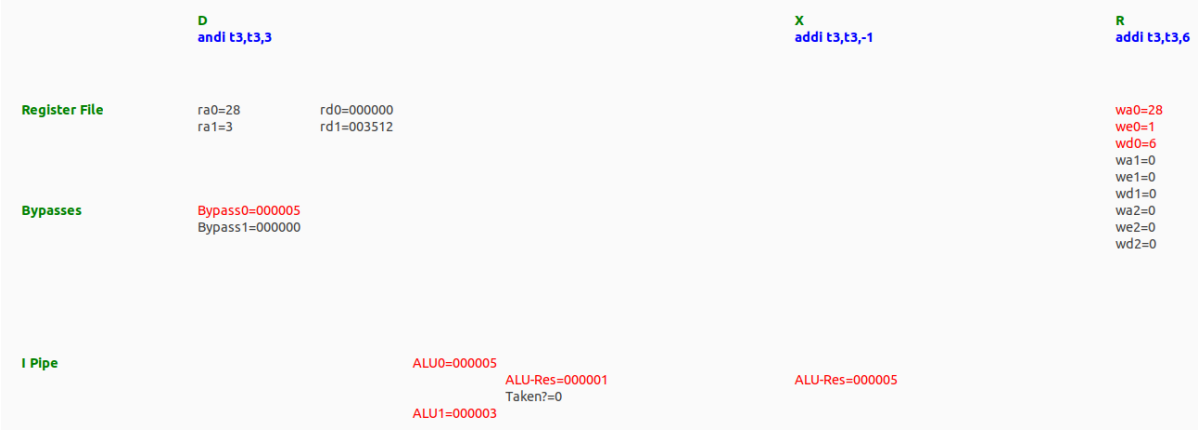


Figure 78. RVfpgaEL2-Pipeline executing AL-Operations

9. 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 their cores. 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 Catapult Studio without requiring the Basys3 FPGA board.

Windows and **macOS:** All the instructions described in this section should work.

Whisper can be executed both using the command line or using an IDE (integrated development environment) such as Eclipse or Catapult Studio. In this section we demonstrate one example to show how to simulate a program with Whisper in Catapult Studio. 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 5 (see Figure 34). Follow the next steps for running and debugging this code on Whisper:

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

2. *[RVfpgaBasysPath]* Ensure that “RVfpga EL2 via Whisper” is selected for Platform, and “Catapult SDK gcc-elf” is selected for the toolchain. Hit build and check that the program builds successfully.

3. Click on the “Run and Debug” button  . Go to the LAUNCH-SELECT section, and select the .elf file which you just built (or select “follow cmake target”).

4. Start the debugger by clicking on the play button 

(make sure that the ”(Embedded) Attach” option is selected). You can find this button near the top of the window (see Figure 38). The program will first compile and then debugging will start. Catapult Studio sets a temporary breakpoint at the beginning of the main function, so the execution will stop there.

5. You can now debug the program exactly as you did in Section 5.B, but this time the program is running in simulation on Whisper instead of on the Basys3 FPGA board.

6. If a program uses the *ee_printf* function in Whisper, such as the HelloWorld_C-Lang example (Section 5.F), you should not open the Catapult Studio serial monitor, as messages are shown in the Terminal console instead (see Figure 79).

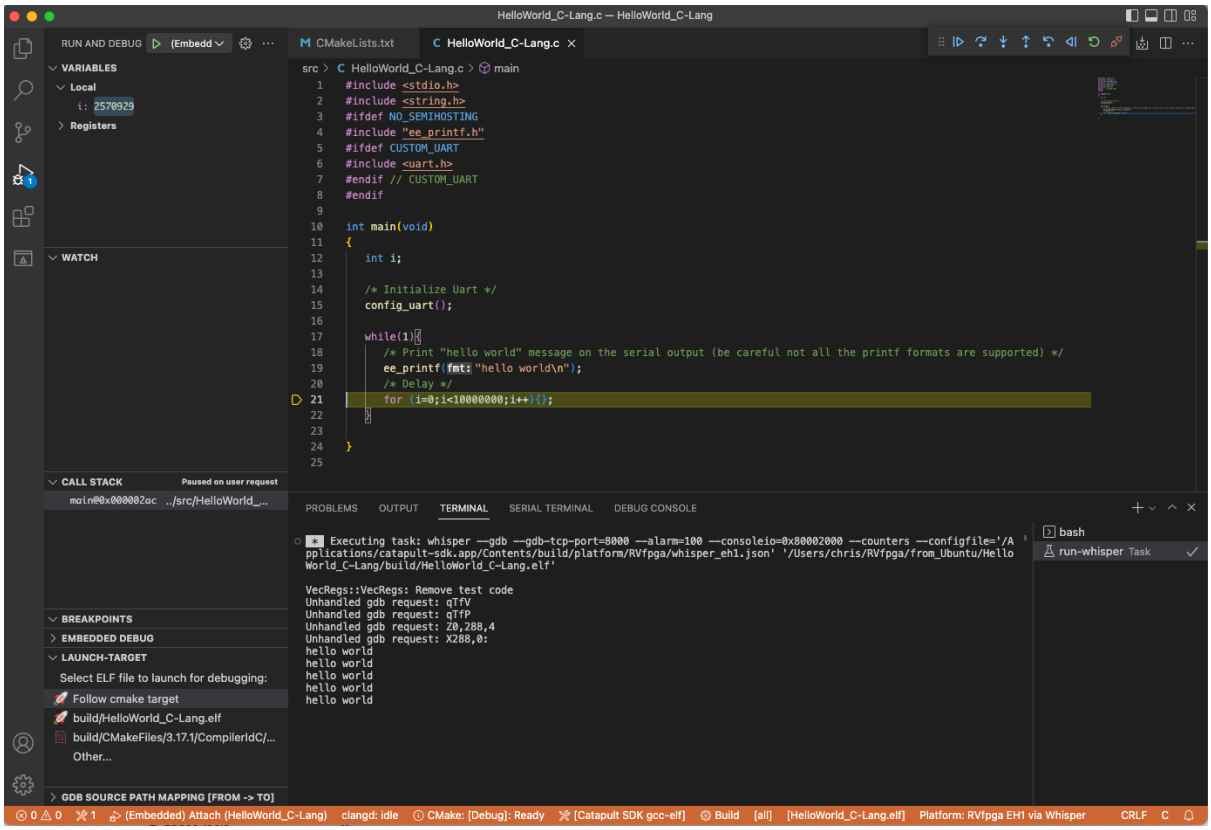


Figure 79. Execution of the HelloWorld_C-Lang example in Whisper

10. Appendices

Table 9 lists all of the appendices available in this RVfpgaEL2 Getting Started Guide.

Table 9. List of Appendices

Appendix	Description	Operating System
A	Installing JTAG drivers in Windows	**Windows**
B	Installing tools used by the simulators in Windows	**Windows**
C	Installing tools used by the simulators in MacOS (BETA)	**MacOS**
D	Using the simulators in Windows	**Windows**
E	Using RVfpgaEL2 with PlatformIO	All

Appendix A: Installing JTAG drivers in Windows

To download the Zadig executable, browse to the following website (see Figure 80):

https://zadig.akeo.ie/

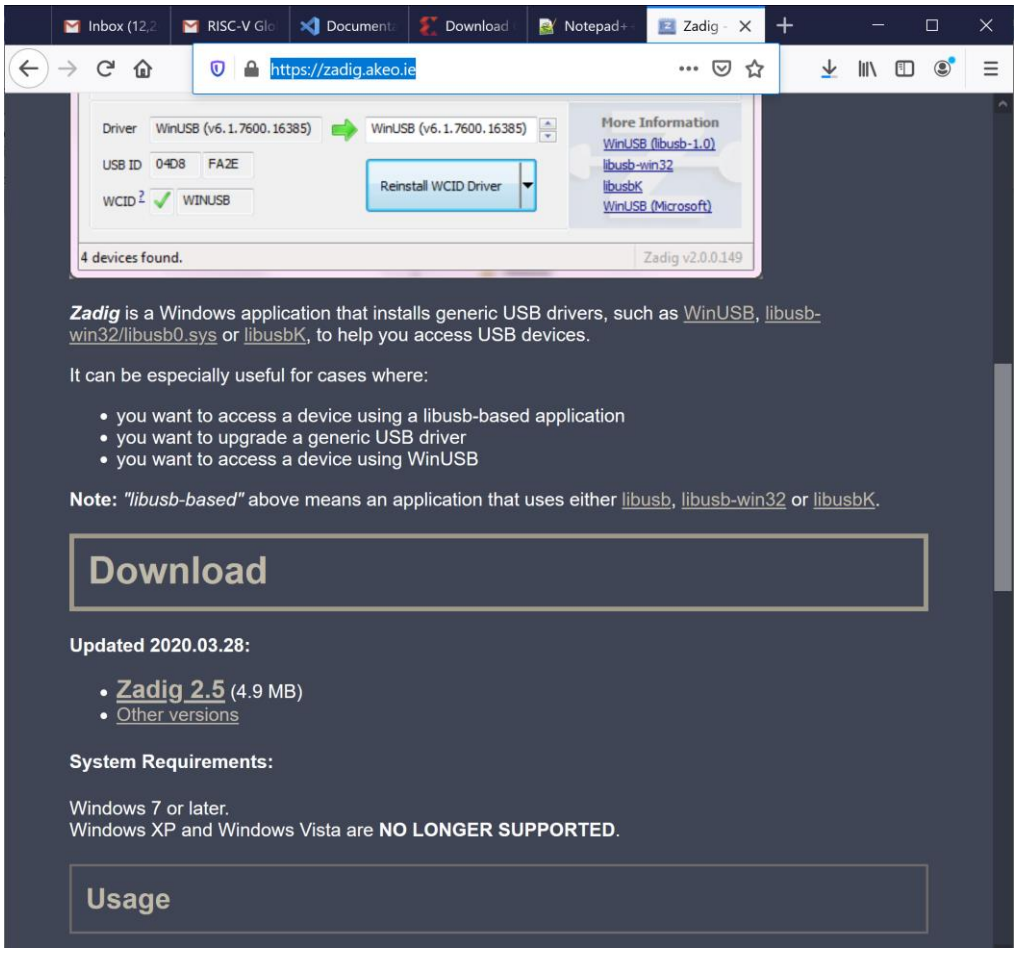


Figure 80. Install Basys3 board driver

Click on Zadig 2.5 (or the newest available version) 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 Basys3 Board to your computer and switch it on. In Zadig, click on Options → List All Devices (see Figure 81).

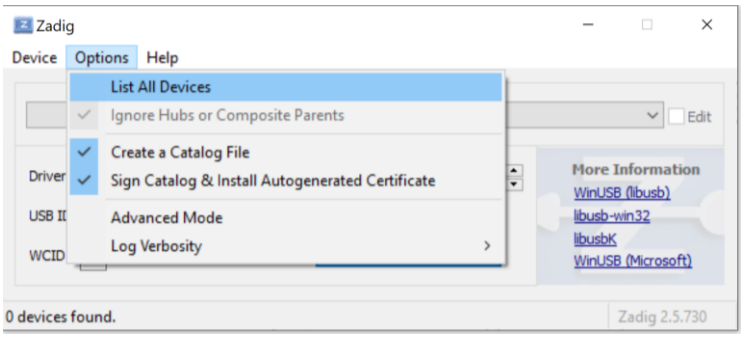


Figure 81. List all devices in Zadig

If you click on the drop-down menu, you will see Digilent USB Device (Interface 0) and (Interface 1) listed. You will now replace the FTDI driver with the WinUSB driver, as shown in Figure 82. Click on Replace Driver (or Install Driver) for Digilent USB Device (Interface 0). You are installing the driver for the Basys3 board or, if you previously installed Vivado, you are replacing the FTDI driver used by Vivado with the WinUSB driver used by Catapult Studio.

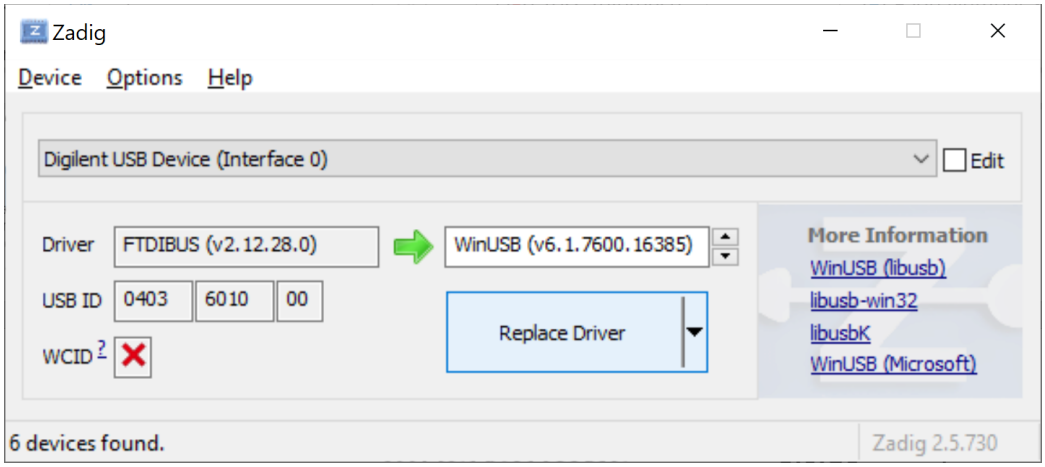


Figure 82. Replace driver for Basys3 board

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

NOTE: If the board does not work initially, try to reinstall the drivers with Zadig.

Appendix B: Installing tools used by the simulators in Windows

In this Appendix we describe in detail how to install the tools required to build and run the simulators in Windows.

1. Install Cygwin

You will use Cygwin for using the simulators in Windows, thus, before installing the simulation tools, you first need to install Cygwin. 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 83).

Figure 83. Cygwin installation webpage

2. Execute the setup file in your machine by double-clicking on it (Figure 84).

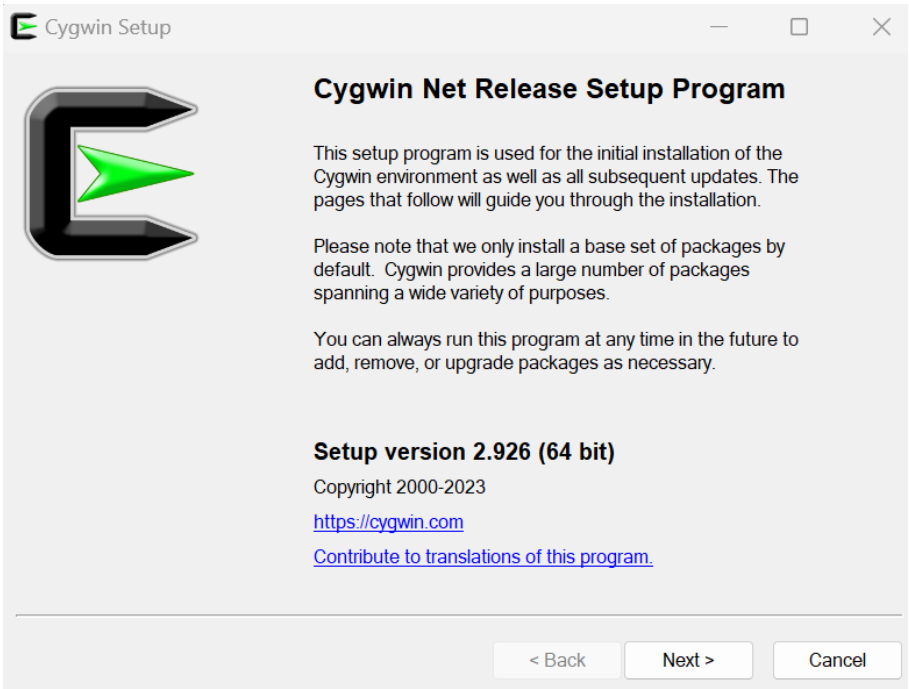


Figure 84. Cygwin installation window

Click **Next** several times, maintaining the default options. The installer will ask you to **Choose a Download Site** (Figure 85), you can choose any one of them.

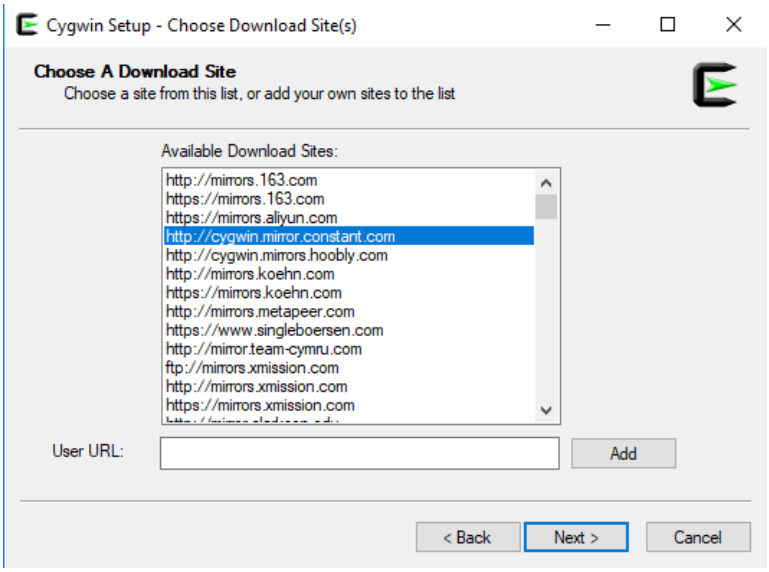


Figure 85. Choose Download Site

3. After several steps, you will reach the **Select Packages** window (Figure 86). Select the **Full** view, as shown in Figure 86.

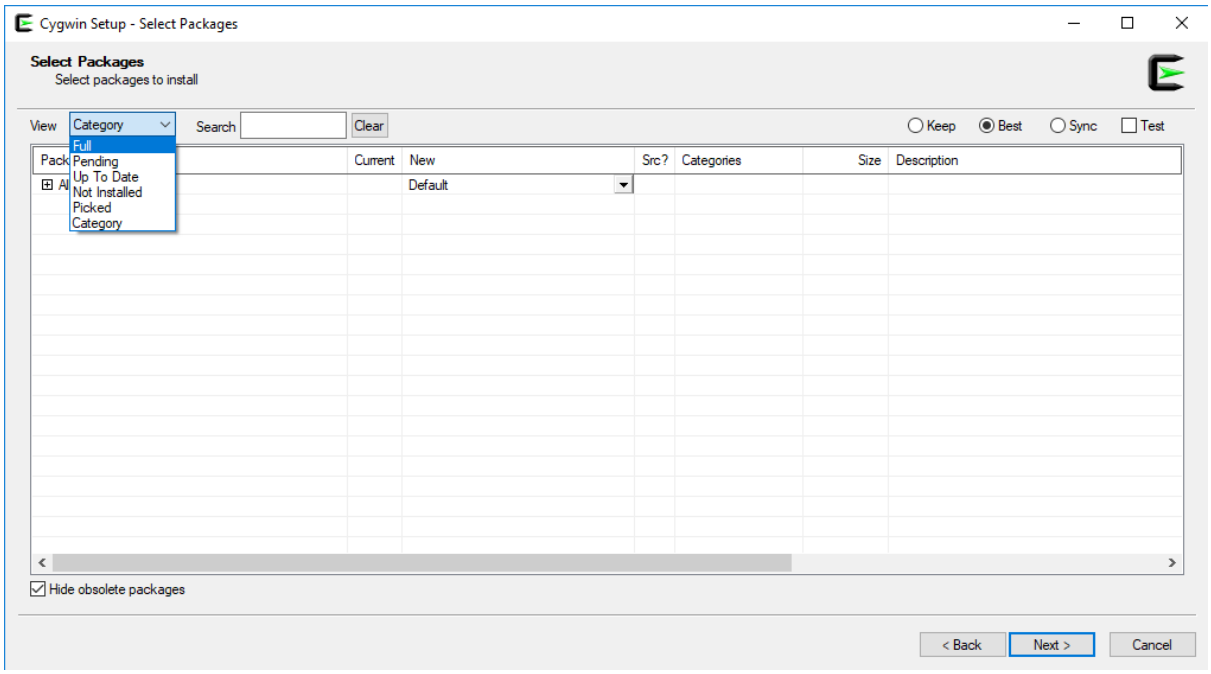


Figure 86. Select Packages window

4. The complete list of packages that you can install will appear (Figure 87). In the **Search** box, select the specific packages that you want to install.

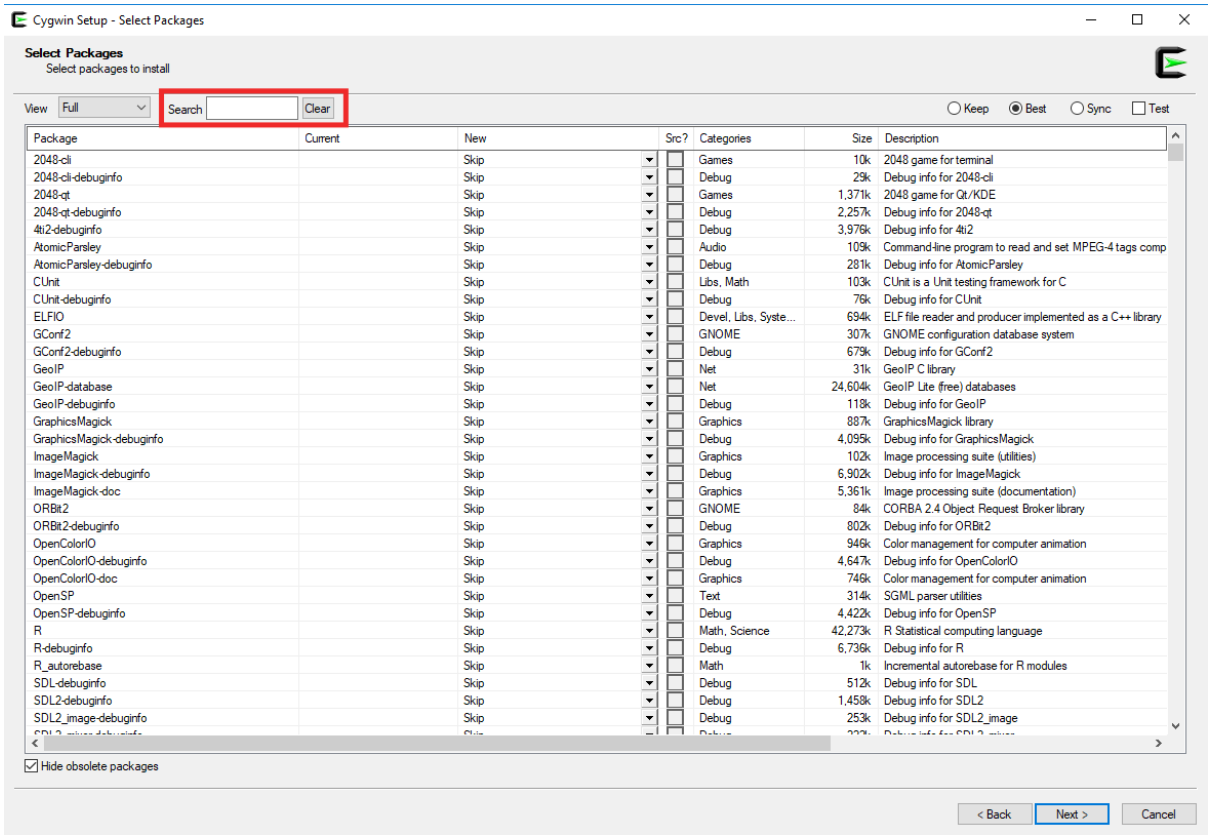


Figure 87. Select Packages window – Full view

To be able to use the RVfpgaEL2 simulators and generate new simulator binaries, you need to install the following packages:

- `git`
- `make`
- `autoconf`
- `gcc-core`
- `gcc-g++`
- `flex`
- `bison`
- `perl`
- `libargp-devel`
- `python3`
- `xorg-server`
- `xinit`
- `libgtk3-devel`
- `libcairo-devel`
- `libcairo2`
- `libssl-devel`
- `cmake`

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 88).

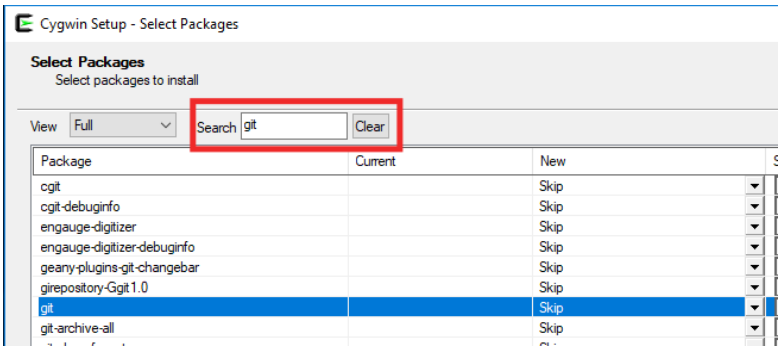


Figure 88. Look for the `git` package

- Select the most up-to-date stable version in the dropdown menu **and** tick the box (Figure 89).

Figure 89. Select the most up-to-date version and tick the box

- Do the same for the remaining packages in the above list.

5. Once you have selected all the packages, click **Next** in the subsequent windows to include these packages in your Cygwin installation (the installation process, see Figure 90, may take several minutes).

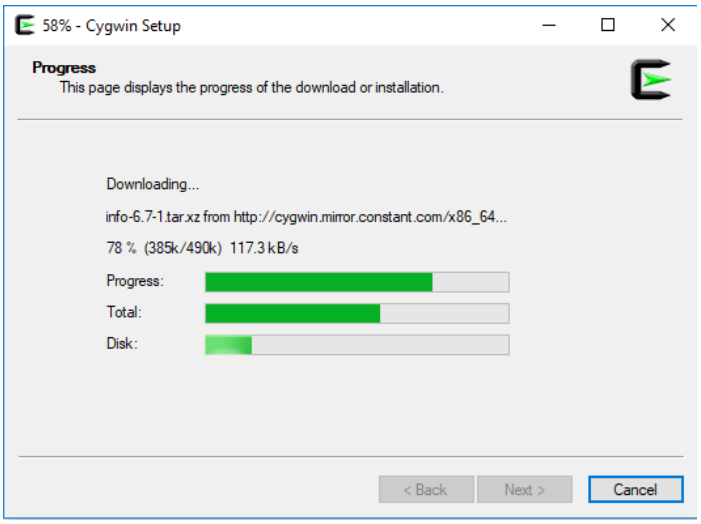


Figure 90. Cygwin setup

Finalize the installation by clicking Finish (Figure 91).

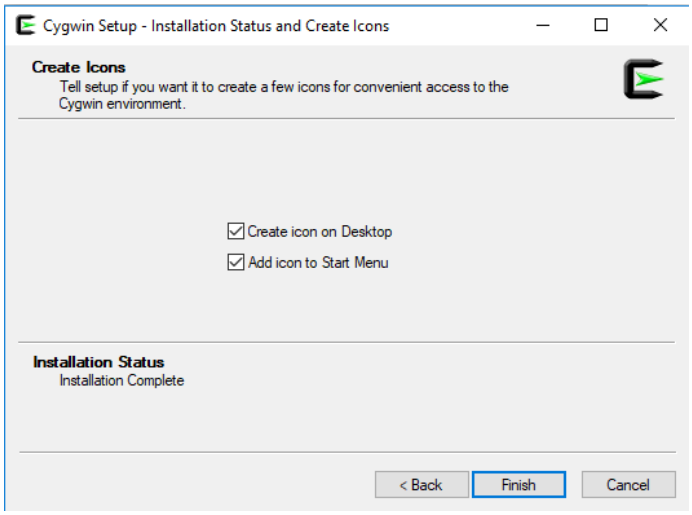


Figure 91. Finish the installation

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

2. Verilator

Verilator allows you to regenerate the simulators if you make any changes. Open a Cygwin terminal, go into a folder of your Windows user where you want to download the sources (note that the *C:* Windows folder can be found inside Cygwin at: */cygdrive/c*; for example: `cd /cygdrive/c/Users/dani/Desktop`) and follow the next steps:

- `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.

3. Tools for RVfpgaEL2-Trace

GTKWave can be downloaded as a precompiled package from https://sourceforge.net/projects/gtkwave/files/. Look for the most recent Windows package and download and unzip (uncompress) it (the package should include a text like "win32" or "win64" in its name, such as "gtkwave-3.3.100-bin-win64"). You can find an executable file called gtkwave inside the folder bin, which you can execute and use in your Windows machine.

4. Tools for RVfpgaEL2-ViDBo

The websockets library for Windows is only provided in the RVfpga_EL2 package for the NexysA7-DDR configuration (you can find it at *[RVfpgaEL2NexysA7DDRPath]/Simulators/verilatorSIM_ViDBo/libwebsockets*). You need to carry out some configurations related with it:

1. Copy the *libwebsockets* library from *[RVfpgaEL2NexysA7DDRPath]/Simulators/verilatorSIM_ViDBo/libwebsockets* to *[RVfpgaBasysPath]/Simulators/verilatorSIM_ViDBo/libwebsockets*

2. At the beginning of file *[RVfpgaBasysPath]/Simulators/verilatorSIM_ViDBo/rvfpgasim.vc*, include the following lines, for including the libwebsockets library:

   - `+incdir+./libwebsockets/build/include`
   - `-CFLAGS -I./libwebsockets/build/include`

   See the following screenshot:

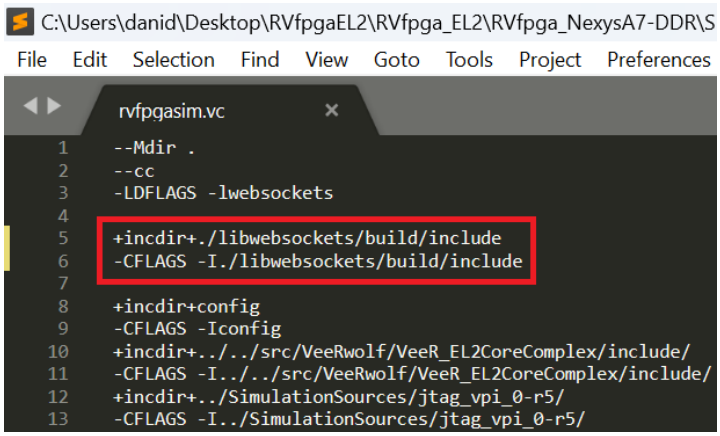


Figure 92. Include libwebsockets library in file *rvfpgasim.vc*

> **IMPORTANT:** This path assumes that the `libwebsockets` folder is located inside the *verilatorSIM_ViDBo* folder. In some exercises you will be asked to modify the SoC and thus you will have to regenerate the simulator binaries. If you use a copy of the simulator folder, make sure that the path points to the correct location.

3. Add to the PATH environment system variable the route to folder *[RVfpgaBasysPath]\Simulators\verilatorSIM_ViDBo\libwebsockets\build\bin*.

   See the following screenshot:

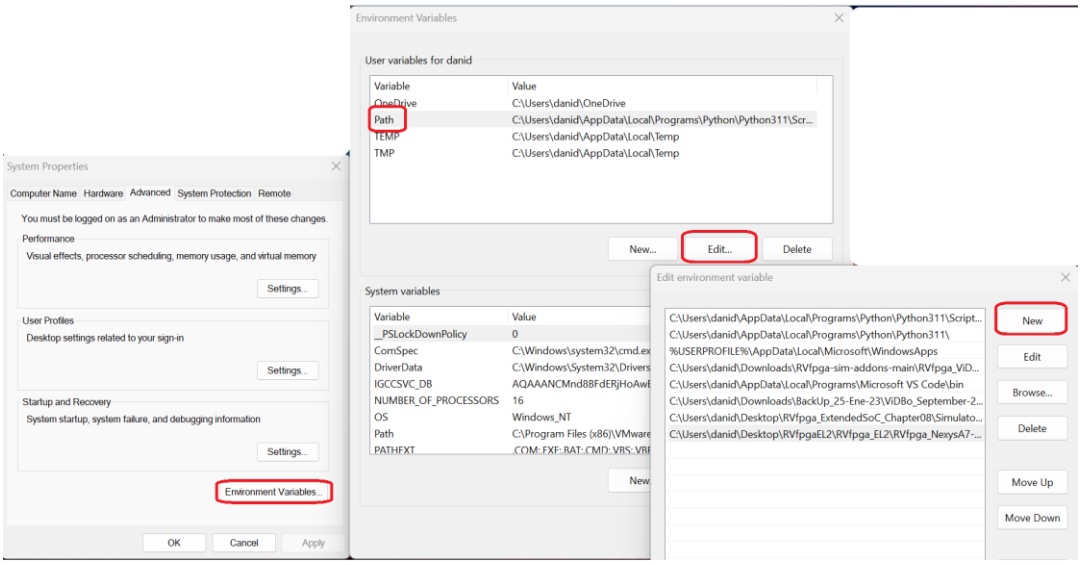


Figure 93. Include path to libwebsockets

4. Open a Windows File Explorer, go into *[RVfpgaBasysPath]/Simulators/verilatorSIM_ViDBo/libwebsockets* and copy the following two library files:

   - `build/lib/libwebsockets.dll.a`
   - `build/lib/libwebsockets.a`

   to the following Cygwin folder: *C:\cygwin64\lib\gcc\x86_64-pc-cygwin\11*

   See the following screenshot:

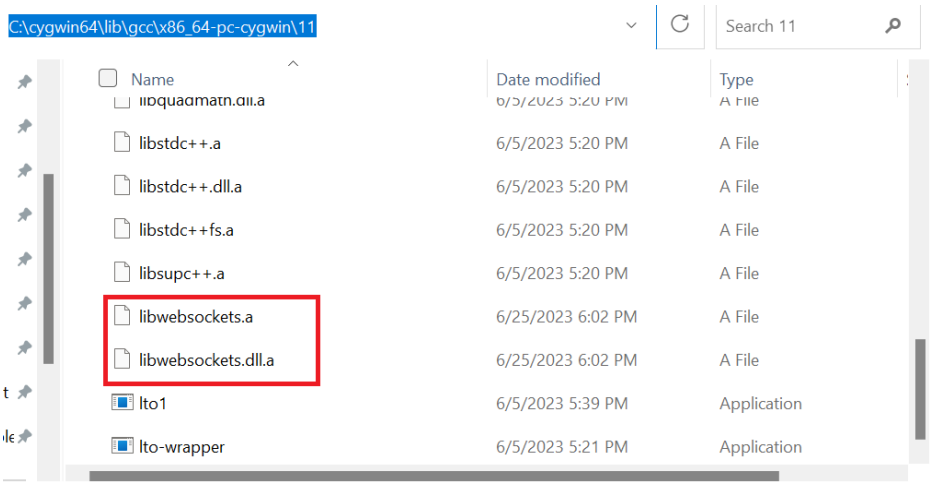


Figure 94. Copy libwebsockets files to Cygwin

5. Finally, consider that, if you use a prebuilt binary for RVfpga-ViDBo, you need to place it in the same directory where the libwebsockets folder is located at.

5. Tools for RVfpgaEL2-Pipeline

You already installed all the required packages when you installed Cygwin.

Appendix C: Installing tools used by the simulators in MacOS (BETA)

In this Appendix we describe in detail how to install the tools required to build and run the simulators in MacOS.

1. Install Verilator

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
```

2. Tools for RVfpgaEL2-Trace

The simplest way is to use either Homebrew or MacPorts package managers to install the package gtkwave. Below are the instructions for Homebrew:

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

After the installation, an icon for *gtkwave.app* should appear in the Application folder. Note that if you don't have an X server already installed (Xquartz) you should also install it:

```
brew install --cask xquartz
```

3. Tools for RVfpgaEL2-ViDBo

Follow the same instructions as for an Ubuntu OS. Install *libwebsockets* using the MacPorts package manager.

4. Tools for RVfpgaEL2-Pipeline

Follow the same instructions as for an Ubuntu OS. Install gtk+3 using a package manager (e.g.: `brew install gtk+3`).

Appendix D: Using the simulators in Windows

For using the three RVfpgaEL2 Verilator-based simulation tools in Windows you must follow the same steps as in Ubuntu from a Cygwin terminal. However, you must take into account the following important facts:

- You probably installed in Cygwin `gcc` and `g++` versions newer than version 10, as version 11 is chosen by default. In that case, when you try to generate the simulator binary, you will obtain the following error:

  ```
  error: ‘numeric_limits’ is not a member of ‘std’
  ```

  You can resolve this error by adding the following line:

  ```
  #include <limits>
  ```

  in file */usr/local/share/verilator/include/verilated.cpp*, which you can find in your windows machine, in a File Explorer, at: *C:\cygwin64\usr\local\share\verilator\include*.

  See the following screenshot:

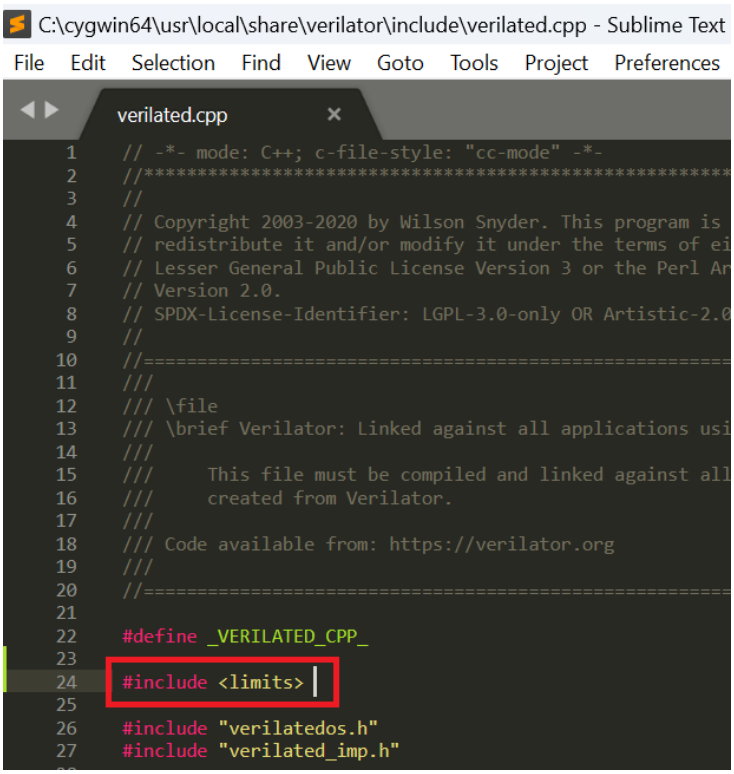


Figure 95. Include header file in file *verilated.cpp*

- When generating a .vh file for the example that you want to simulate, you must take into account that the RISC-V toolchain installed by Catapult is available at: *C:\Program Files\imgtec\catapult-sdk_1.9.1\bin*. As in Linux, you must set the `CATAPULT_SDK_TOPDIR` environment variable appropriately. In Cygwin you can do this using the following command:

```
export CATAPULT_SDK_TOPDIR="C:/Program\ Files/imgtec/catapult-sdk_1.9.1"
```

Note that you can execute the command every time you want to use RVfpgaEL2 or you can define the variable permanently by editing any of the Cygwin configuration files (for example, file */etc/profile*, which can be found at *C:**cygwin64**etc*). Note also that you must use the `catapult-sdk_1.X.X` directory that corresponds to the Catapult version that you have installed.

- RVfpgaEL2-Trace: When opening the trace file with *GTKWave*, note that folder *gtkwave64* that you downloaded in the installation section, 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.

- RVfpgaEL2-Pipeline: When running the compiled RVfpgaEL2-Pipeline simulator, you need to use an X Server from inside Cygwin. For example, for the AL_Operations example, run the command as follows (note that the path to the simulator must be absolute and not relative):

```
startx [AbsolutePathToSimulator]/Vrvfpgasim
+ram_init_file=AL_Operations.vh
```

Appendix E: Using RVfpgaEL2 with PlatformIO

As you have already seen in this GSG, RVfpgaEL2 uses Catapult as its main Software Development Kit. However, the RVfpga version based on VeeR EH1 uses PlatformIO as its SDK. In the new RVfpgaEL2 version we also provide support for this widespread SDK.

In this appendix we show how to install and use RVfpgaEL2 with PlatformIO using some examples. If you need more details, you can find them at the RVfpgaEH1 package.

NOTE: We provide PlatformIO projects for the GSG examples and for some of the Labs, respectively, at:
- *[RVfpgaBasysPath]/examples/examples_PlatformIO*
- *[RVfpgaBasysPath]/Labs/RVfpgaLabsPlatformIO*

1. VSCode, PlatformIO and ChipsAlliance Platform installation

Follow these steps to install both VSCode, PlatformIO and the ChipsAlliance platform:

Virtual Machine: if you are using the provided Ubuntu 22 Virtual Machine you can skip this installation section, as everything is already installed. However, you do need to perform the final step (3.4. "If you are going to use PlatformIO […]") for replacing the current EH1 files/folders with the EL2 ones.

1. **Install VSCode:**

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

    2. 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:**

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

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

    2. 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.

3. In VSCode, click on the Extensions icon located on the left side bar of VSCode (see Figure 96).

Figure 96. VSCode's Extensions icon

4. Type *PlatformIO* in the search box and install the PlatformIO *IDE* by clicking on the install button next to it (see Figure 97).

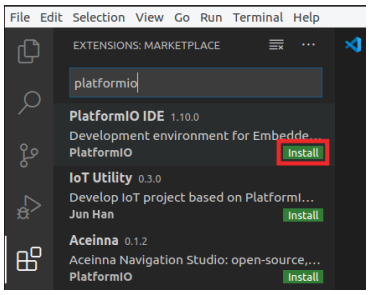


Figure 97. PlatformIO IDE Extension

5. 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 98).

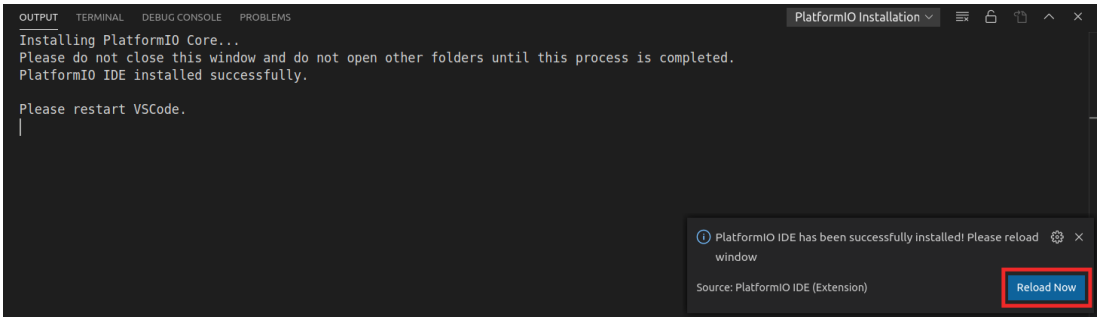


Figure 98. Reload Now after PlatformIO installs

3. Install ChipsAlliance Platform:

1. View the Quick Access menu by clicking on the button, located in the left side bar (see Figure 99). Then, in the PIO Home, click on the Platforms button and then on the Embedded tab (Figure 99). Look for **Chipsalliance** (the platform that we use in

RVfpga) and open it by clicking on the CHIPS Alliance button (Figure 99).

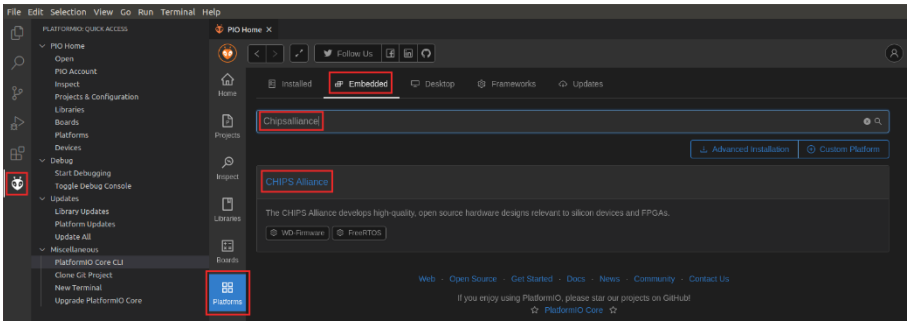


Figure 99. Selecting the CHIPS Alliance Platform

2. After clicking on the CHIPS Alliance button, you will see the details of the Chips Alliance platform (as in Figure 100). Install it by clicking on the Install button (Figure 100).

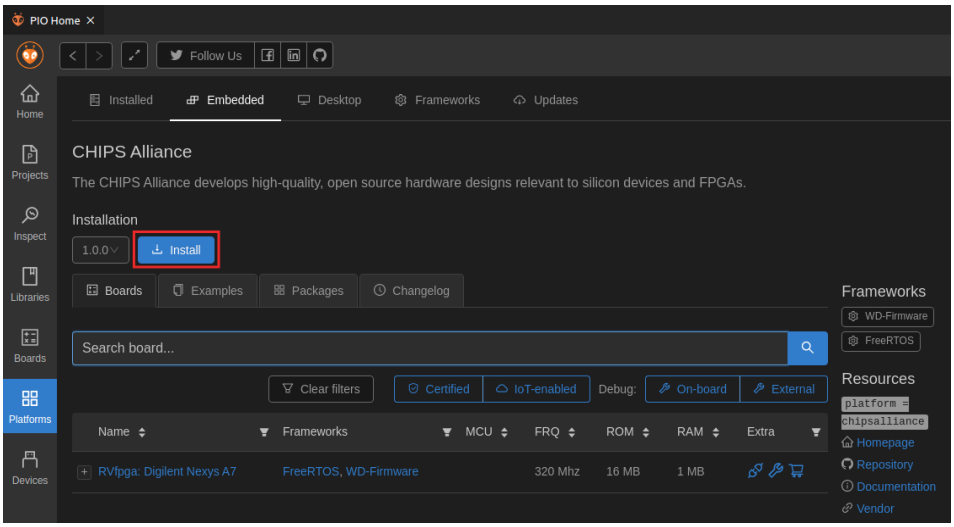


Figure 100. Installing the CHIPS Alliance Platform

3. Once installation completes, a summary of the tools that have been installed is shown, as in Figure 101. Click OK to close that window.

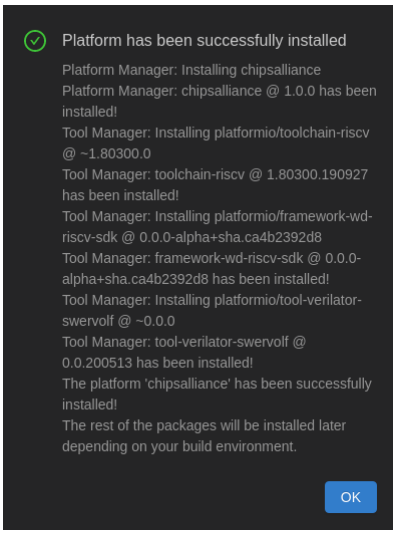


Figure 101. Successful installation of CHIPS Alliance Platform

4. If you are going to use PlatformIO with the VeeR EH1 core, as in RVfpgaEH1, leave the Chips Alliance Platform as it is now and skip this item. However, if you want to use PlatformIO with the VeeR EL2 core, as in RVfpgaEL2, you have to do some previous adaptation of the current platform, as described next. Specifically, a folder called `.platformio` has been created in your home directory, and you must perform the following changes:

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

      with folder
      `[RVfpgaBasysPath]/PlatformIO_ChipsAlliance_Files/psp`

    - Replace file:
      `~/.platformio/platforms/chipsalliance/builder/frameworks/wd-riscv-sdk.py`

      with file:
      `[RVfpgaBasysPath]/PlatformIO_ChipsAlliance_Files/wd-riscv-sdk.py`

    - Replace file:
      `~/.platformio/platforms/chipsalliance/boards/swervolf_nexys.json`

      with file:
      `[RVfpgaBasysPath]/PlatformIO_ChipsAlliance_Files/swervolf_nexys.json`

2. Program the FPGA with RVfpgaEL2-Basys3

In this section, we explain how to program the FPGA with RVfpgaEL2-Basys3 using PlatformIO. Follow the next steps for programming the FPGA with RVfpgaEL2-Basys3:

1. Connect the Basys3 board to your computer.

2. Turn on the Basys3 board using the switch at the top left.

3. Open VSCode and PlatformIO if it is not already open.

4. On the top menu bar, click on *File → Open Folder* (see Figure 102) and browse into directory *[RVfpgaBasysPath]/examples/examples_PlatformIO*

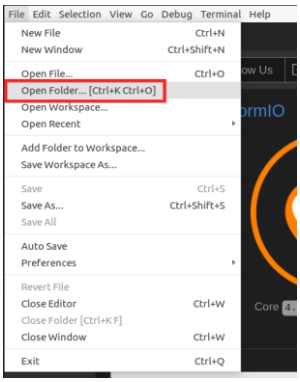


Figure 102. Open Folder

5. 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 103) and click OK at the top of the window. PlatformIO will now open the example.

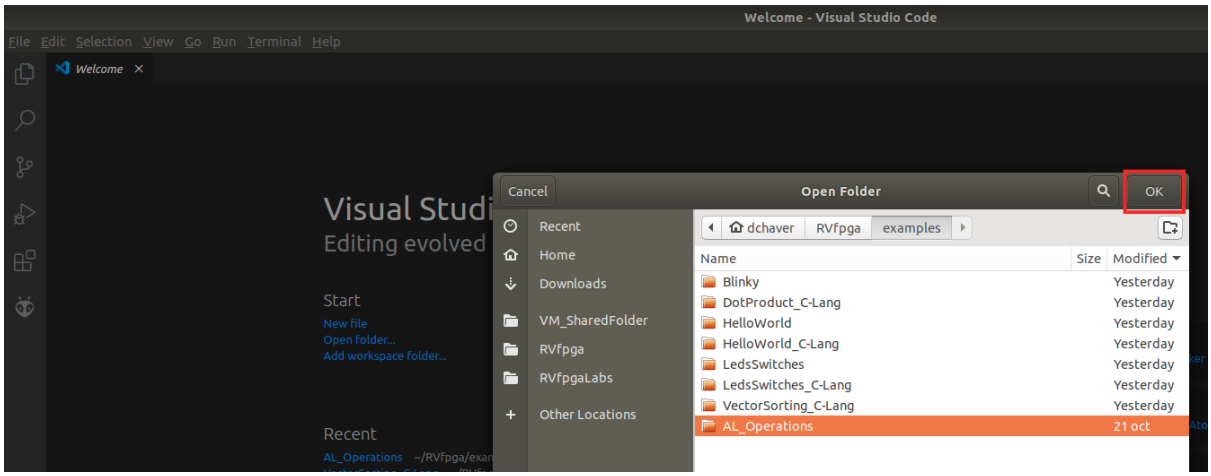


Figure 103. Open AL_Operations folder

6. Open file *platformio.ini*, by clicking on *platformio.ini* in the left sidebar (see Figure 104). Establish the path to the RVfpgaEL2-Basys3 bitstream in your system by editing the following line (see Figure 104). Note that a pre-synthesized bitstream of RVfpgaEL2-Basys3 is provided at: *[RVfpgaBasysPath]/src/rvfpgabasys3.bit*

```
board_build.bitstream_file = [RVfpgaBasysPath]/src/rvfpgabasys3.bit
```

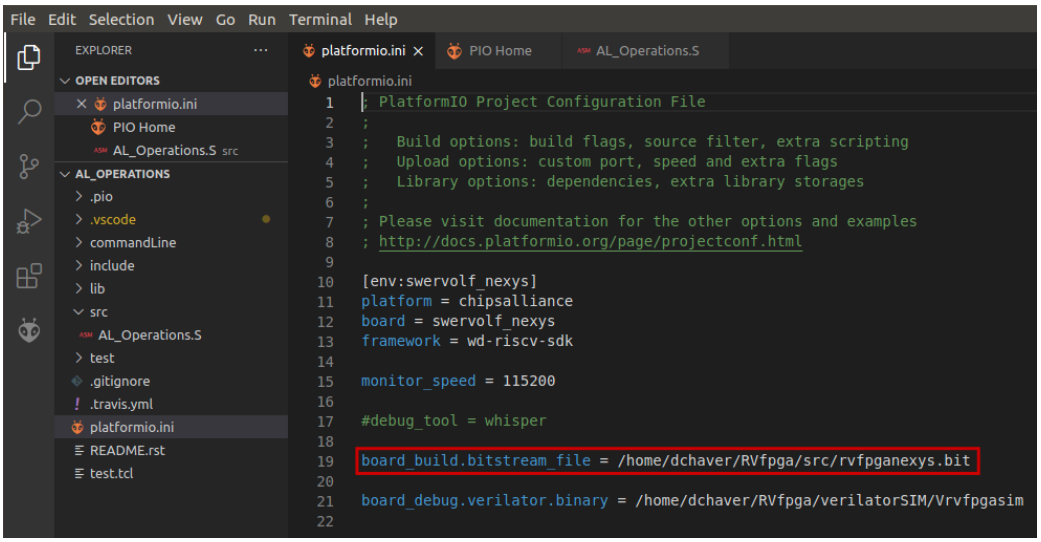


Figure 104. Platformio initialization file: platformio.ini

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/.

7. Click on the PlatformIO icon in the left menu ribbon (see Figure 105).

Figure 105. PlatformIO icon

In case the Project Tasks window is empty (Figure 106), you must refresh the Project Tasks first by clicking on . This can take several minutes.

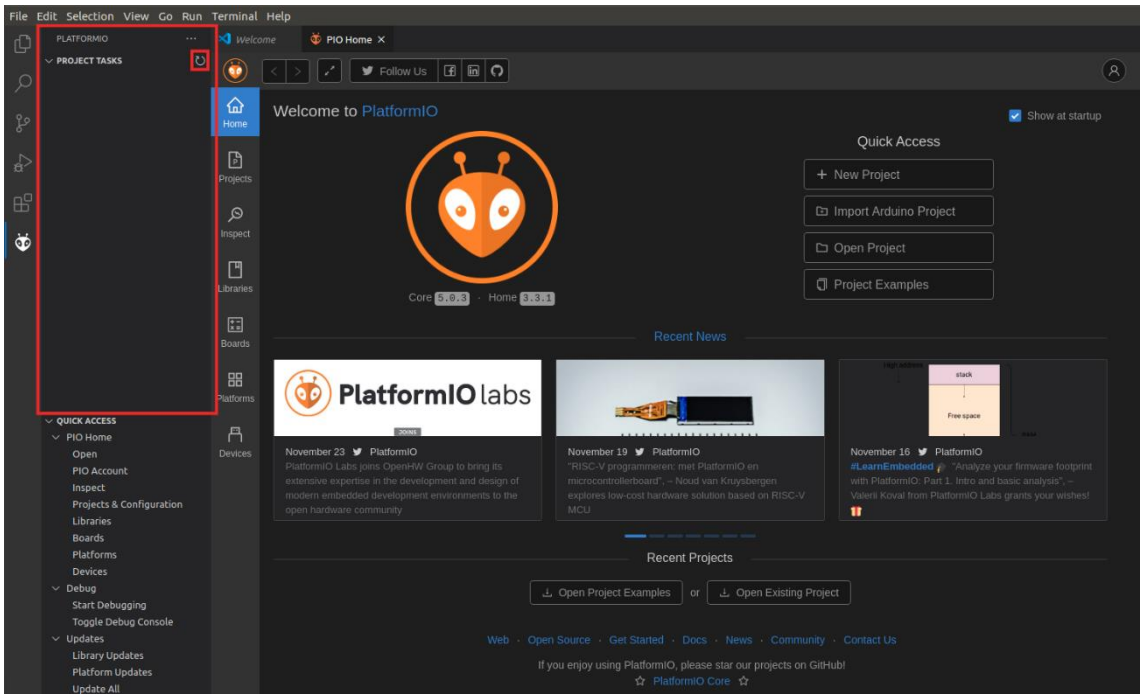


Figure 106. PROJECT TASKS window empty – Refresh

Then expand Project Tasks → env:swervolf_nexys → Platform and click on Upload Bitstream, as shown in Figure 107. **After one or two seconds, the FPGA will be programmed with RVfpgaEL2-Basys3**.

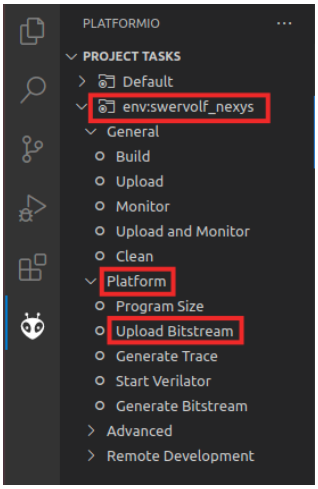


Figure 107. Upload Bitstream

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: *[RVfpgaBasysPath]/src/VeeRwolf/BootROM/sw*.

3. Executing AL_Operations on RVfpgaEL2-Basys3

The AL_Operations program (see Figure 108) 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.

```
1  .globl main
2  main:
3
4  # Register t3 is also called register 28 (x28)
5  li t3, 0x0                    # t3 = 0
6
7  REPEAT:
8      addi t3, t3, 6            # t3 = t3 + 6
9      addi t3, t3, -1           # t3 = t3 - 1
10     andi t3, t3, 3            # t3 = t3 AND 3
11     beq  zero, zero, REPEAT   # Repeat the loop
12     nop
13
14 .end
```

Figure 108. AL_Operations program: AL_Operations.S

Follow these steps to run and debug this code on the Basys3 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 109).

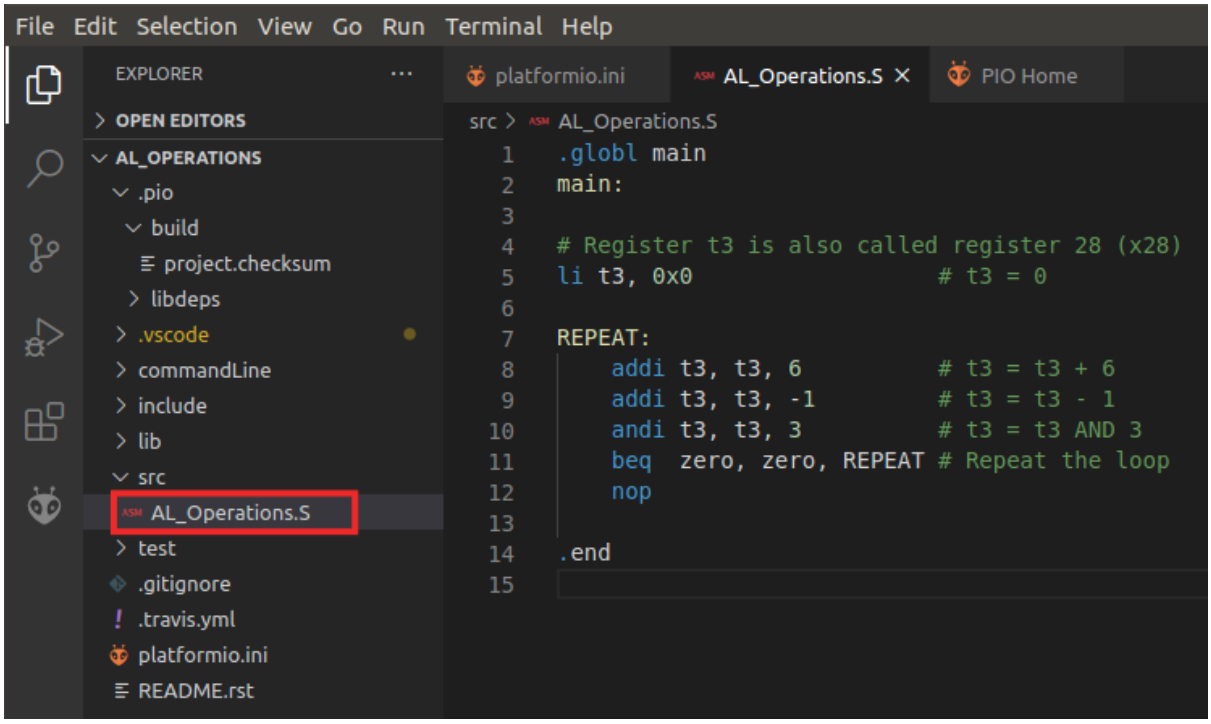
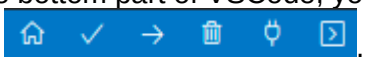


Figure 109. View assembly file AL_Operations.S

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, 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 110). 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.

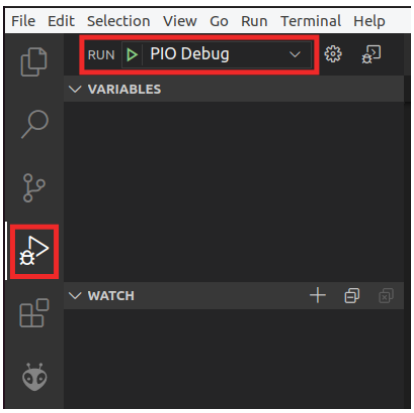


Figure 110. Start debugger

5. To control your debugging session, you can use the debugging toolbar that appears near the top of the editor (see Figure 111). 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.

Figure 111. 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 112.

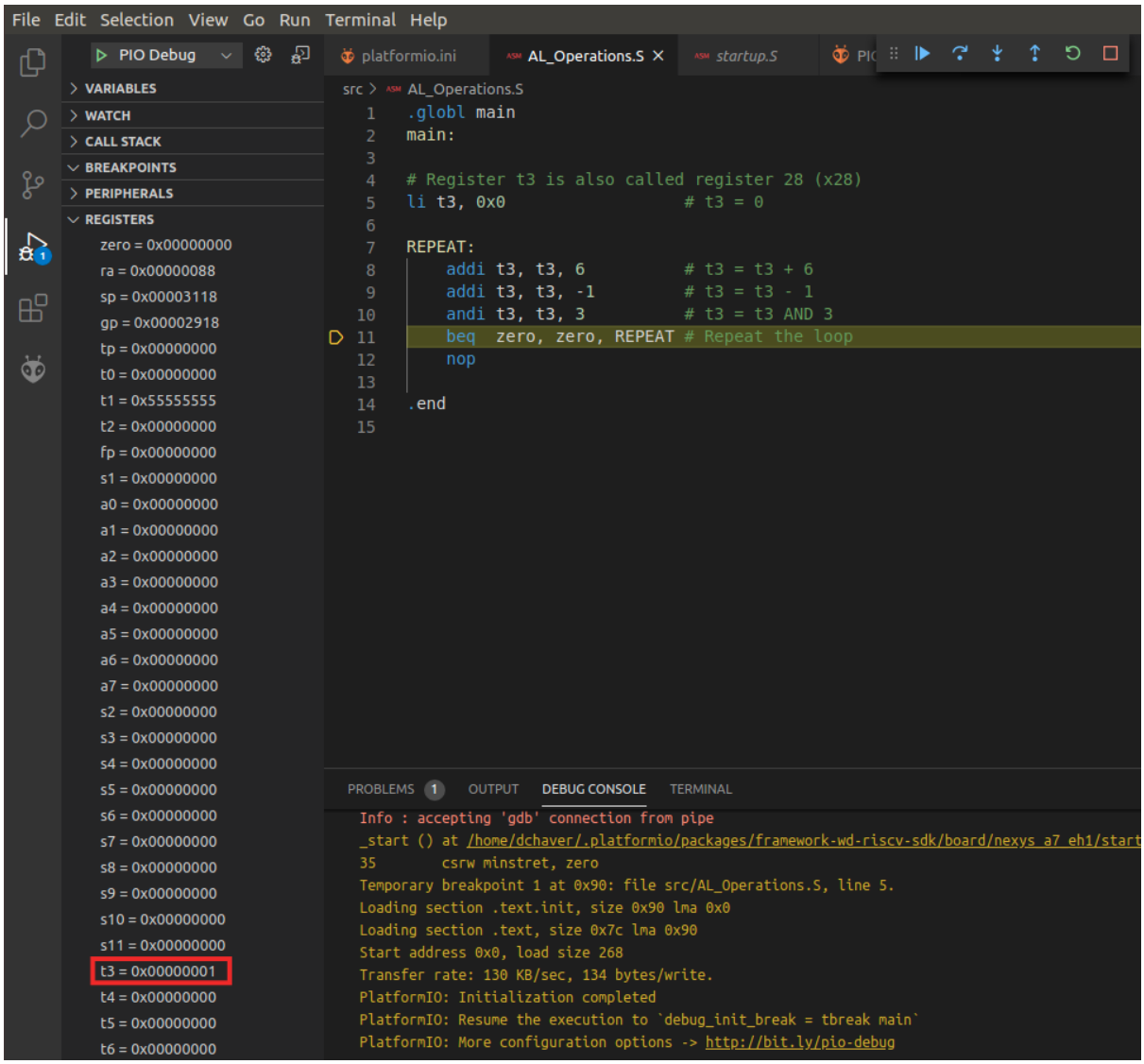


Figure 112. Viewing register contents

8. Before calling the *main* function, a start-up file, provided 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, 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*.

4. Simulating AL_Operations on RVfpgaEL2-Trace

Follow these steps to run and debug this code on RVfpgaEL2-Trace using PlatformIO:

1. Open Visual Studio Code (VSCode) by typing "VSCode" in the Start Menu or by typing "code" in a terminal.

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

3. Then, open the specific project you want to work with (in this case *AL_Operations*), by selecting the folder of the project and clicking on the Open button at the top-right corner. PlatformIO will now open this program, which includes three assembly arithmetic-logic instructions (addition, subtraction, and logical and) on the same register, t3 (also called x28), within an infinite loop. You can view the program by expanding the src folder and double-clicking on *AL_Operations.S*.

4. Open file *platformio.ini*. Establish the path to the provided RVfpgaEL2-Trace simulator binary by editing the following line:

   ```
   board_debug.verilator.binary =
   [RVfpgaBasysPath]/Simulators/verilatorSIM_Trace/Vrvfpgasim
   ```

   Note that you must use the absolute path to the simulator binary (a relative path does not work in this case). Note also that you can use the precompiled binaries provided at `[RVfpgaBasysPath]/Simulators/verilatorSIM_Trace/OriginalBinaries`

5. Execute the RVfpgaEL2-Trace simulator from PlatformIO:

   a. Click on the PlatformIO icon in the left menu ribbon:  .
   b. In case the PROJECT TASKS window is empty, you must refresh the Project Tasks first by clicking on  . This can take several minutes.
   c. Then expand *PROJECT TASKS* > *env:swervolf_nexys* > *Platform* and click on **Generate Trace** (see Figure 113). This first compiles the program and then launches the Verilator simulation of the RVfpga SoC running this program.

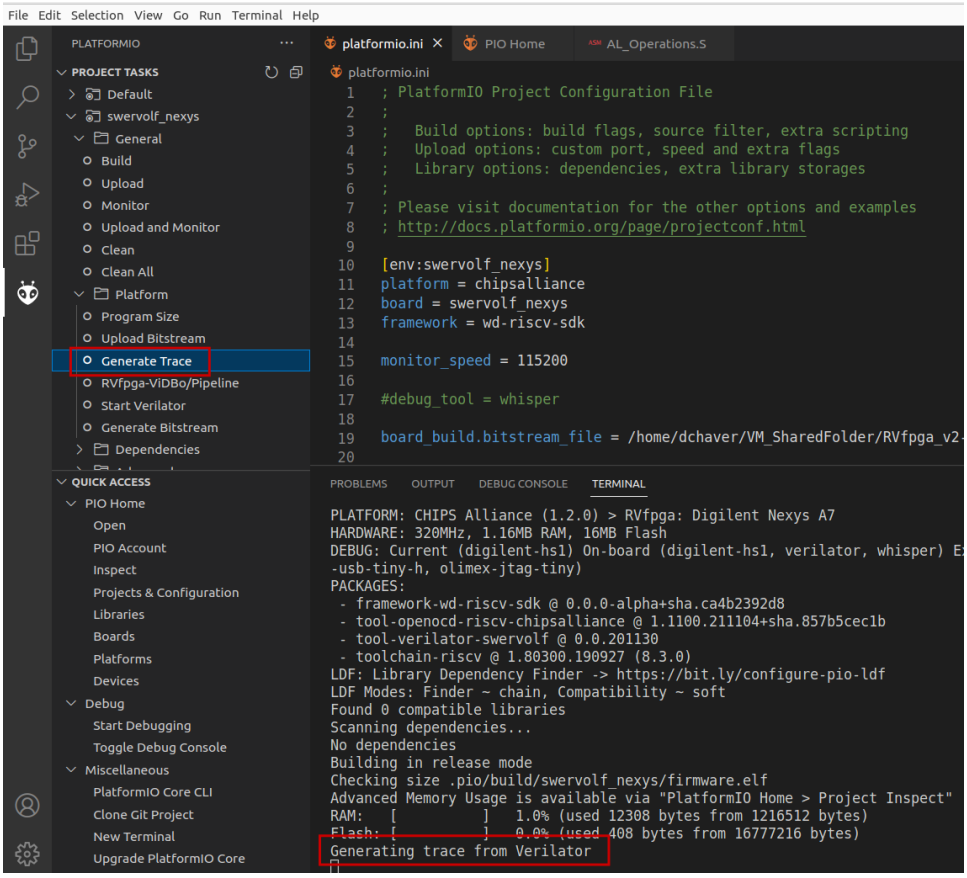


Figure 113. Generating Trace from Verilator

6. A few seconds after the previous step, the *AL_Operations* program is compiled and file *trace.vcd* is generated inside folder *[RVfpgaBasysPath]/examples/examples_PlatformIO/AL_Operations/.pio/build/swervolf_nexys.* For analyzing the trace in the next step it may be useful to visualize the disassembly program that has been generated at: *[RVfpgaBasysPath]/examples/examples_PlatformIO/AL_Operations/.pio/build/swervolf_nexys/firmware.dis*.

7. Finally, analyze the trace as explained in Section 6.

5. Simulating LedsSwitches on RVfpgaEL2-ViDBo

Follow these steps to run and debug this code on RVfpgaEL2-ViDBo using PlatformIO:

1. Open Visual Studio Code (VSCode) by typing "VSCode" in the Start Menu or by typing "code" in a terminal.

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

3. Then, open the specific project you want to work with (in this case *LedsSwitches*), by selecting the folder of the project and clicking on the Open button at the top-right corner. PlatformIO will now open this program. You can view the program by expanding the src folder and double-clicking on *LedsSwitches*.

4. Open file *platformio.ini*. Establish the path to the provided RVfpgaEL2-ViDBo simulator binary by editing the following line:

```
board_debug.verilator.binary =
[RVfpgaBasysPath]/Simulators/verilatorSIM_ViDBo/Vrvfpgasim
```

Note that you must use the absolute path to the simulator binary (a relative path does not work in this case). Note also that you can use the precompiled binaries provided at `[RVfpgaBasysPath]/Simulators/verilatorSIM_ViDBo/OriginalBinaries`

5. Execute the RVfpgaEL2-ViDBo simulator from PlatformIO:

   a. Click on the PlatformIO icon in the left menu ribbon:  .
   b. In case the PROJECT TASKS window is empty, you must refresh the Project Tasks first by clicking on  . This can take several minutes.
   c. Then expand *PROJECT TASKS* > *env:swervolf_nexys* > *Platform* and click on **RVfpgaEL2-ViDBo/Pipeline** (see Figure 114). This first compiles the program and then launches the Verilator simulation of the RVfpga SoC running this program.

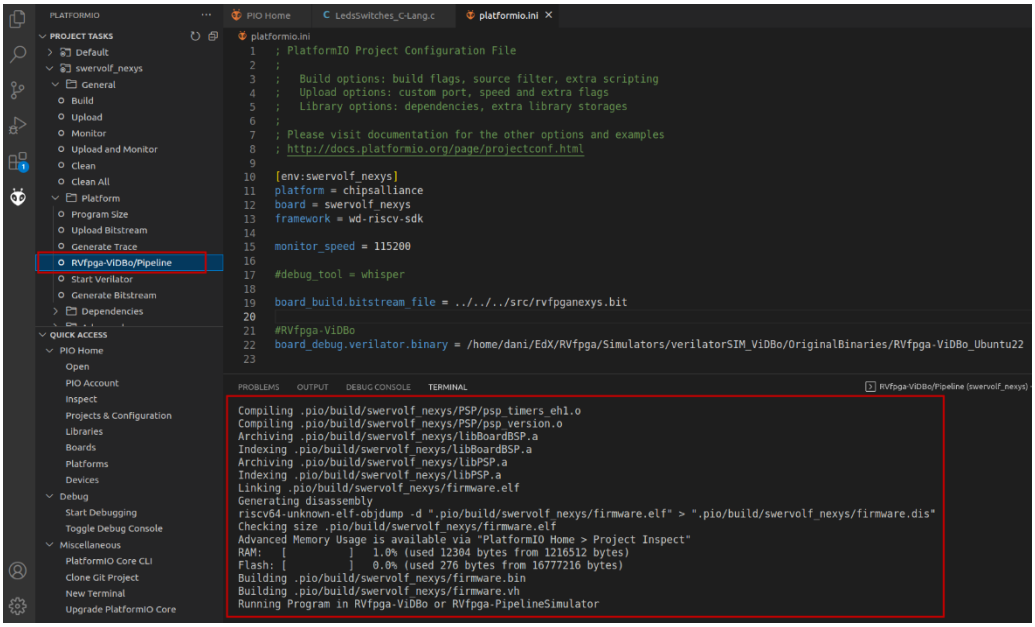


Figure 114. Launching RVfpgaEL2-ViDBo

6. Once the RVfpgaEL2-ViDBo is executing, you must launch from a terminal the python server simulating the Basys3 board (`python3 -m http.server --directory Basys3Board/`) and connect to http://localhost:8000/basys3.html inside a browser.

7. Finally, analyze the simulation as explained in Section 7.

6. Simulating AL_Operations on RVfpgaEL2-Pipeline

Follow these steps to run and debug this code on RVfpgaEL2-Pipeline using PlatformIO:

1. Open Visual Studio Code (VSCode) by typing "VSCode" in the Start Menu or by typing "code" in a terminal.

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

3. Then, open the specific project you want to work with (in this case *AL_Operations*), by selecting the folder of the project and clicking on the Open button at the top-right corner. PlatformIO will now open this program, which includes three assembly arithmetic-logic instructions (addition, subtraction, and logical and) on the same register, t3 (also called x28), within an infinite loop. You can view the program by expanding the src folder and double-clicking on *AL_Operations*.

4. Open file *platformio.ini*. Establish the path to the provided RVfpgaEL2-Pipeline simulator binary by editing the following line:

```
board_debug.verilator.binary =
[RVfpgaBasysPath]/Simulators/verilatorSIM_Pipeline/Vrvfpgasim
```

   Note that you must use the absolute path to the simulator binary (a relative path does not work in this case). Note also that you can use the precompiled binaries provided at `[RVfpgaBasysPath]/Simulators/verilatorSIM_Pipeline/OriginalBinaries`.

5. As explained in Section 8, this simulator executes the *AL_Operations* program (or any other program that we open in PlatformIO) and stops execution at the point where it reaches a control instruction, which in our simulator is set to instruction `and zero, t4, t5`. From that point we can simulate the program cycle by cycle and analyze some selected core internal signals as instructions progress through the VeeR EL2 pipeline. Insert the control instruction in the program by editing file *src/AL_Operations.S*. Place the control instruction right before the end of the loop.

6. Execute the RVfpgaEL2-Pipeline simulator from PlatformIO:

   a. Click on the PlatformIO icon in the left menu ribbon:  .
   b. In case the PROJECT TASKS window is empty, you must refresh the Project Tasks first by clicking on  . This can take several minutes.
   c. Then expand *PROJECT TASKS* > *env:swervolf_nexys* > *Platform* and click on **RVfpgaEL2-ViDBo/Pipeline** (see Figure 115). This first compiles the program and then launches the Verilator simulation of the RVfpga SoC running this program.

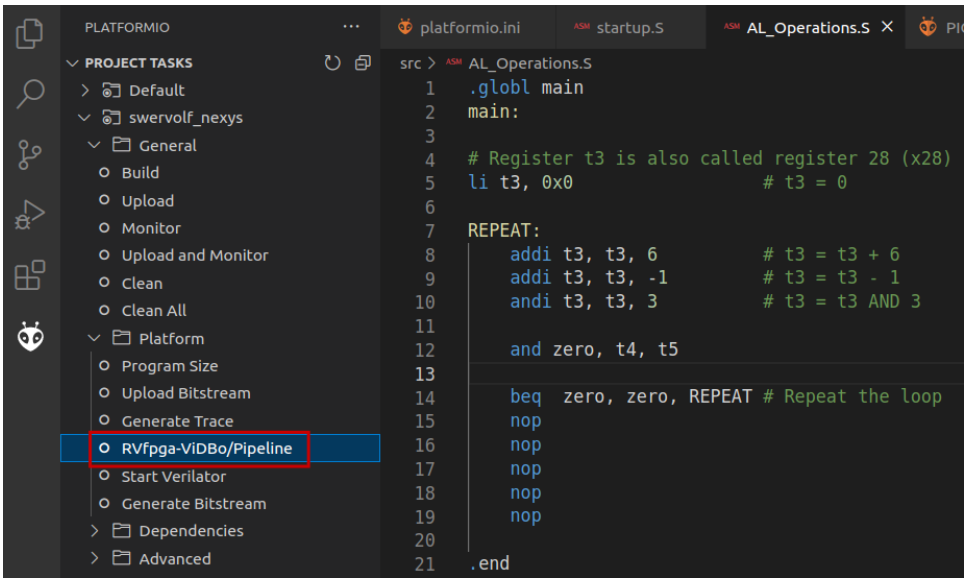


Figure 115. Launching RVfpgaEL2-Pipeline

7. Finally, analyze the simulation as explained in Section 8.

7. Simulating AL_Operations on RVfpgaEL2-Whisper

Follow these steps to run and debug this code on RVfpgaEL2-Whisper using PlatformIO:

1. Open Visual Studio Code (VSCode) by typing "VSCode" in the Start Menu or by typing "code" in a terminal.

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

3. Then, open the specific project you want to work with (in this case *AL_Operations*), by selecting the folder of the project and clicking on the Open button at the top-right corner. PlatformIO will now open this program, which includes three assembly arithmetic-logic instructions (addition, subtraction, and logical and) on the same register, t3 (also called x28), within an infinite loop. You can view the program by expanding the src folder and double-clicking on *AL_Operations*.

4. Open file *platformio.ini* and set whisper as the debug tool by uncommenting line 17 (see Figure 116). Save the file (press Ctrl-s).

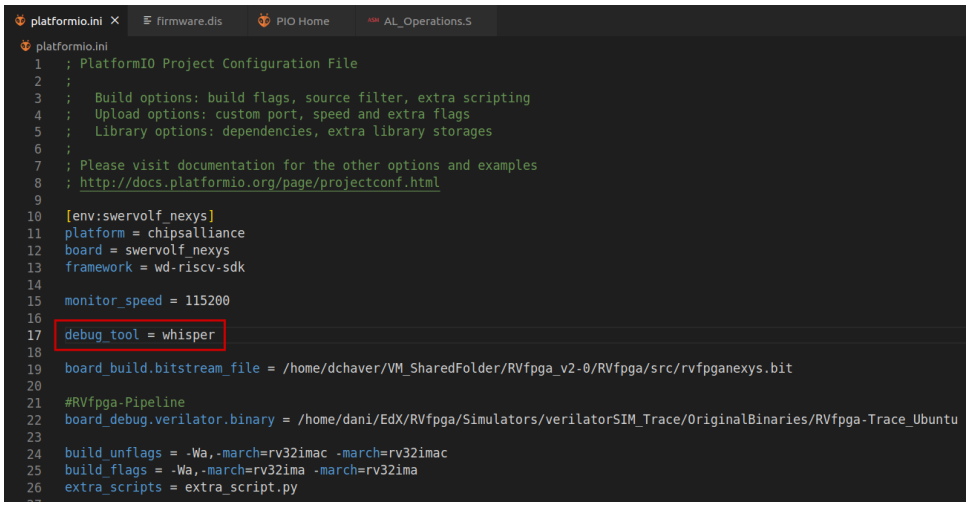
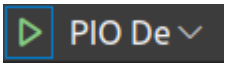


Figure 116. Set Whisper as the Simulation Tool

5. Click on the RUN AND DEBUG button  , which is available in the bar on the left-hand side. Start the debugger by clicking on the Play button 

(make sure that the "PIO Debug" option is selected). The program will first compile and then debugging will start. To control your debugging session, you can use the debugging toolbar which appears near the top of the editor.