

**THE IMAGINATION UNIVERSITY PROGRAMME**

**RVfpga-SoC Lab 2**

**Running Software on RVfpga-SoC**

# Introduction

This lab shows how to run programs written in C or Assembly language on the RVfpga-SoC design we created in Lab1. You may choose to simulate the design using Verilator and/or run the design on the Nexys A7 board. If you do not have access to an FPGA board, this lab may be completed only in simulation using Verilator. To complete this lab, you will use the SweRVolf’s “BD.v” Verilog file and the “rvfpga.bit” bit file that was generated in Lab 1 using Vivado’s Block Design.

In this Lab, we will show how to generate the binary file for RVfpgaSIM, which will be used later for creating the simulation trace of an example program. We will also analyze the simulation trace using GTKWave.

As an optional step, we will show how to download the RVfpga-SoC, as defined by the bitstream that we created in Lab 1, onto our Nexys A7 board using PlatformIO and then debug an example program using PlatformIO. This step is optional but recommended.

# Requirements :

To complete this lab, you will need to install the following tools:

* VSCode (Refer to Installation Guide (Page No.07))
* PlatformIO (Refer to Installation Guide (Page No.07))
* GTKWave (Refer to Installation Guide (Page No.13))
* Verilator (Refer to Installation Guide (Page No.13))
* Cygwin (For Windows User only) (Refer to Installation Guide (Page No.09))

**IMPORTANT:** Before starting RVfpga-SoC Labs, we highly recommend completing the RVfpga-SoC Installation Guide.

For example, if you have not already, install Xilinx’s Vivado and Verilator following the instructions in the RVfpga-SoC Installation Guide. Make sure that you have copied the RVfpga-SoC folder that you downloaded from Imagination’s University Programme to your machine.

# Running RVfpga-SoC :

In the first Lab, we created RVfpga-SoC (SweRVolf) by connecting the processor core, interconnects, and peripherals with each other using Vivado's Block Design tool. The Block Design then generates a Verilog file of that SweRVolf module as a whole. In our case, it was the “**BD.v**'' file, which we later called the “**swervolf**” module file.

Now we have two options and pathways to run RVfpga-SoC

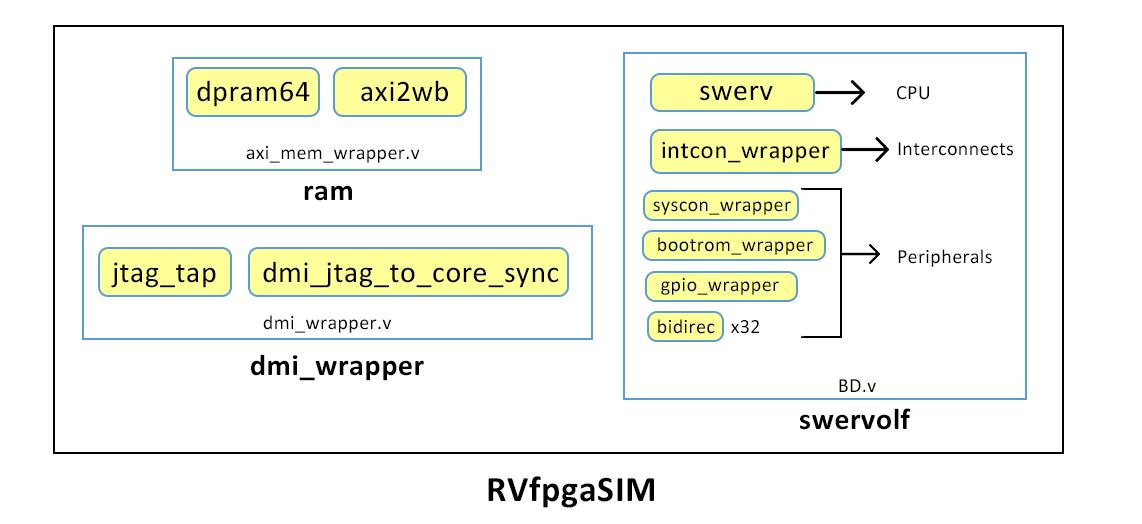
* Run RVfpga-SoC on the Nexys A7 100T board.
* Run RVfpga-SoC on the Verilator simulator.

RVfpga-SoC has two top-level modules that exist for each of those targets: RVfpga SIM (rvfpgasim) and RVfpga (rvfpga), as described below.:

1. **RVfpgaSIM** (rvfpgasim.v)

The RVfpga SIM module is used as the top module of RVfpga for **Simulation**. We use Verilator (a hardware description language (HDL) simulator that simulates the Verilog that defines RVfpga) for simulating RVfpga-SoC. Running the SoC in simulation allows us to analyze the system’s internal signals in depth. Later in this Lab, when we create the simulation binary for RVfpgaSIM, we will use “**rvfpgasim.v**” as the top module file.

The top module “rvfpgasim” structure is illustrated in Figure 1.



**Figure 1. RVfpgaSIM**

It includes three modules:

* swervolf (BD.v)

This is the SoC module that we have created using Vivado’s Block Design.

* ram (axi\_mem\_wrapper.v)

This is the memory module.

* dmi\_wrapper (dmi\_wrapper.v)

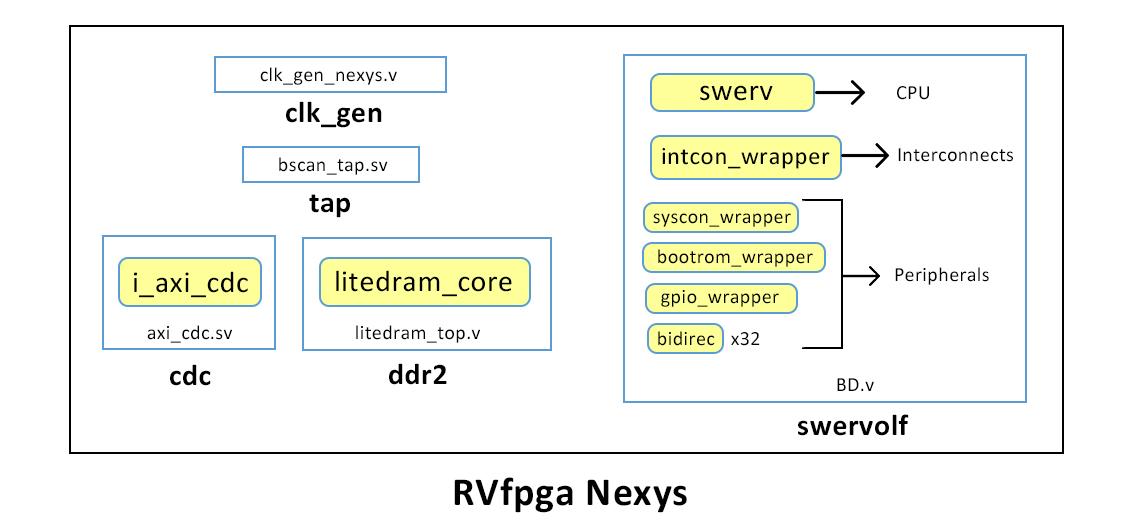
This is the Debugging module Interface.

In Lab 1, while connecting pins using Vivado’s Block Design, we made some of the pin connections external. These external connections of the **“swervolf” module** are connected in the top module “**rvfpgasim**” with other modules. For instance, the “**DMI**” external connections in the “**swervolf**” module are connected with the “**dmi\_wrapper**” module, and the “**RAM**” external connections of the “**swervolf**” module are connected with the “**ram**” module.

1. **RVfpga Nexys** (rvfpga.sv)

The RVfpga Nexys module is used as the top module of RVfpga for the Hardware (On-Board Implementation), which is targeted to the Digilent Nexys A7 board (or, interchangeably, the older Nexys 4 DDR board).

The top module “rvfpga.sv” structure is illustrated in Figure 2.



**Figure 2. RVfpgaNexys**

RVfpga Nexys includes five modules :

* swervolf (BD.v)

This is the SoC module that we have created using Block Design.

* ddr2 (litedram\_top.v)

This is the DDR memory controller module.

* clk\_gen (clk\_gen\_nexys.v)

This is the Clock generator module.

* tap (bscan\_tap.sv)

This is the jtag debug module. For more information, see this [link](https://github.com/chipsalliance/Cores-SweRVolf/issues/29)

* cdc (axi\_cdc.sv)

This is the Clock Domain Crossing module.

In the “**rvfpga.sv**” top module, the “**RAM**” external connections of the “**swervolf**” module are connected with the “**ddr2**” module. The “**DMI**” external connections of “**swervolf**” are connected with the “**bscan\_tap.sv**” module.

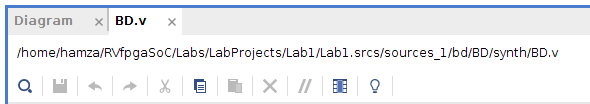
# Running A Program on Verilator

This section will take you through the process of how to run your first program (*AL\_Operations*) on RVfpgaSIM using Verilator.

First, we will need to move our “**swervolf**” module file, **BD.v**, to the folder containing all of the other source files, including the top module file “rvfpgasim.v”.

When we created the HDL wrapper in the last Lab, we were provided with its full path (See Figure 3). We will need to navigate this path and then copy the file.

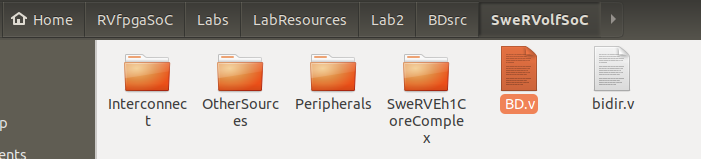
[RVfpgaSoCPath]/RVfpgaSoC/Labs/LabProjects/Lab1/Lab1.srcs/sources\_1/bd/BD/synth/BD.v

****

**Figure 3. The path of the “BD.v” Verilog file of Swervolf Core**

Copy the “BD.v” file from the path given in (Figure 3) and paste the “**BD.v**” file to the following path (See Figure 4) :

[RVfpgaSoCPath]/RVfpgaSoC/Labs/LabResources/Lab2/BDsrc/SweRVolfSoC/

**

**Figure 4. “BD.v” Pasted in the “SweRVolfSoC” directory.**

Now we will start with the process of running the *AL\_Operations* Program on our RVfpga-SoC.

First, we will generate the simulation trace using PlatformIO and then add the clock, instructions for both ways of the superscalar processor, and register x28 (i.e., register t3) signals to the simulation waveform, and view with GTKWave of the instruction and register signals change as the program executes.

To do so, complete the following steps :

**Step 1. Generate The Simulation Binary for RvfpgaSIM**

The directory *[RVfpgaSoCPath]/RVfpgaSoC/Labs/LabResources/Lab2/verilatorSIM* contains the *Makefile* and the *script* (*swervolf\_0.7.vc*) for generating the simulator binary for RVfpgaSIM. The *script* contains information for Verilator to know, among other things, where to find the sources for the SoC, which in our case are available at : *[RVfpgaSoCPath]/RVfpgaSoC/Labs/LabResources/Lab2/BDsrc*.

Next, generate the binary for RVfpgaSIM, which will later be used to create the simulation trace of program *AL-Operations* running on RVfpga.

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

cd [RVfpgaSoCPath]/RVfpgaSoC/Labs/LabResources/Lab2/verilatorSIM

make clean

make

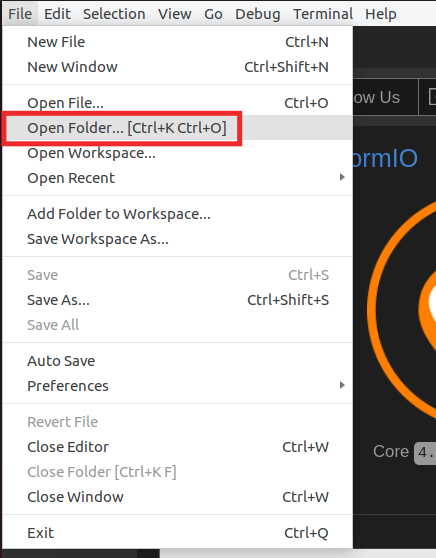
File ***Vrvfpgasim*** (the RVfpga simulation binary) should be generated inside the directory *[RVfpgaSoCPath]/RVfpgaSoC/Labs/LabResources/Lab2/verilatorSIM*.

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

**Step 2. Generate The Simulation Trace From PlatformIO**

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

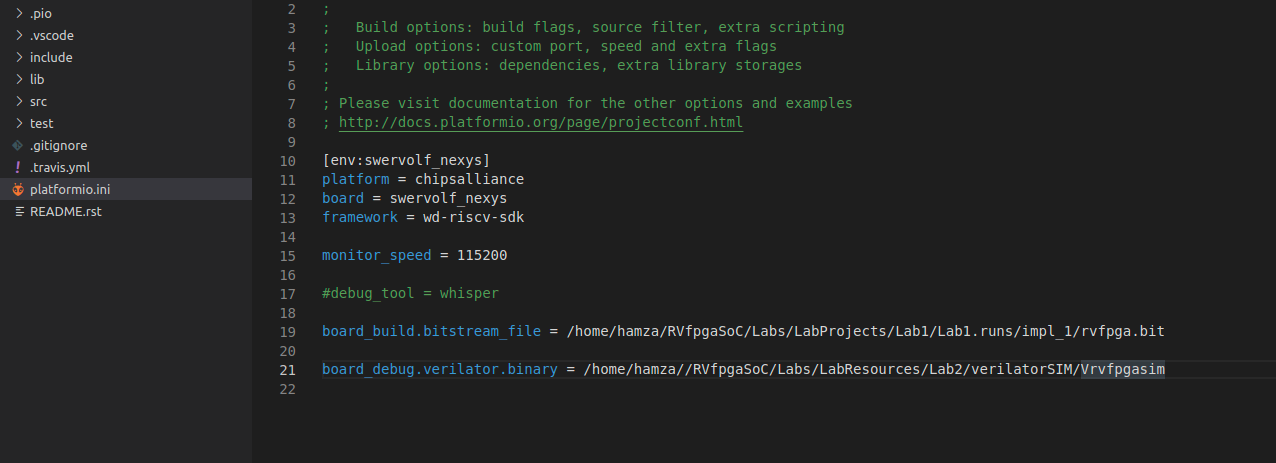
1. Open VSCode and then PlatformIO on your computer.
2. On the top bar, click on *File*→*Open Folder* (Figure 3), and browse into directory *[RVfpga-SoCPath]/RVfpga-SoC/Labs/LabResources/Lab2/examples/*



**Figure 3. Open the AL\_Operations.S example**

1. Select directory *AL\_Operations* (do not open it, but just select it)and click OK. The example will open in PlatformIO.
2. Open file *platformio.ini*. Establish the path to the RVfpga simulation binary generated in the first step (*Vrvfpgasim*) by editing the following line (see Figure 4).

board\_debug.verilator.binary = [RVfpgaSoCPath]/RVfpgaSoC/Labs/LabResources/Lab2/verilatorSIM/Vrvfpgasim

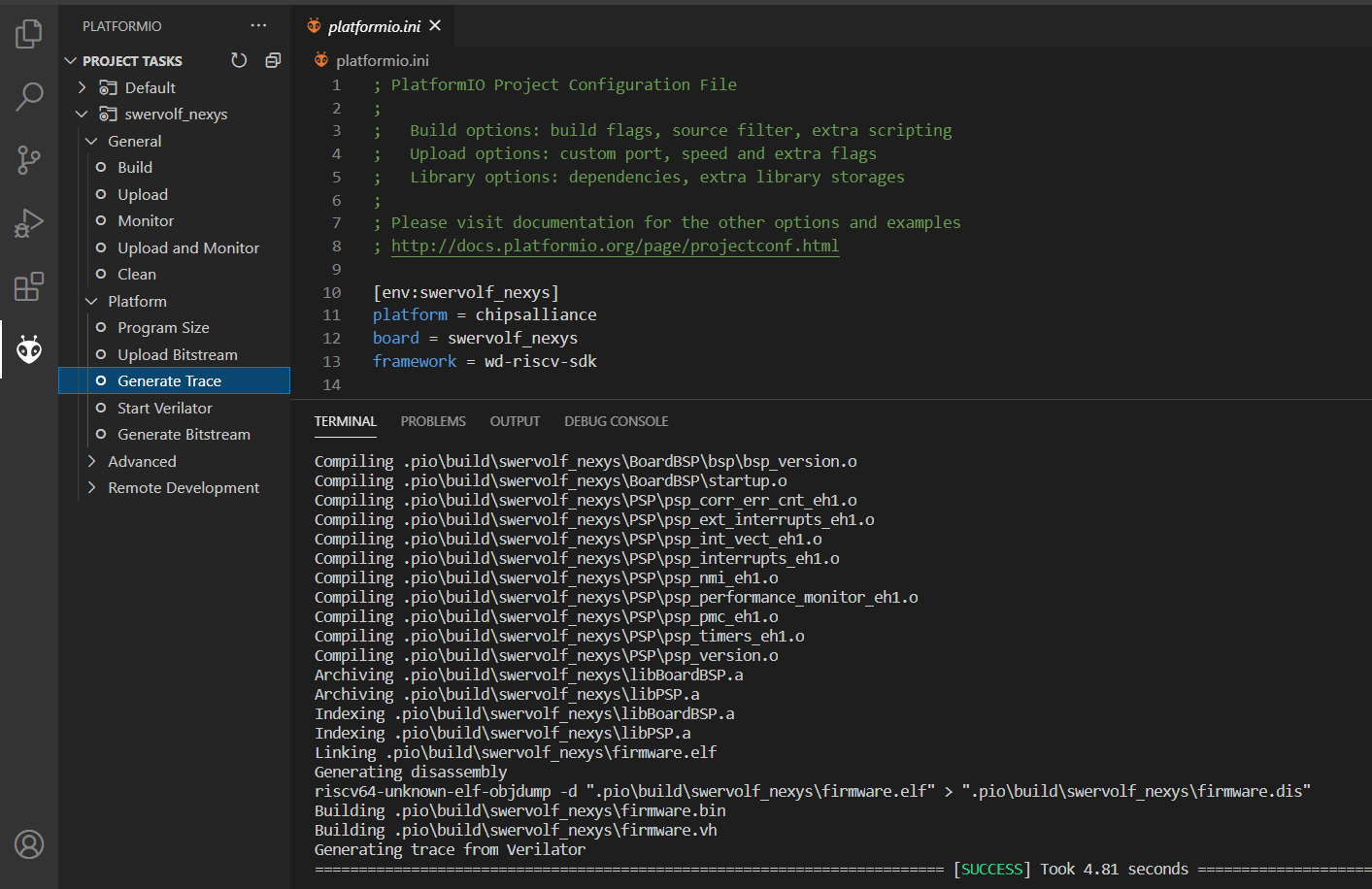
****

**Figure 4. PlatformIO initialization file: platformio.ini**

**Windows:** in Windows, the RVfpga simulation executable is called Vrvfpgasim.exe. Thus:

board\_debug.verilator.binary = [RVfpgaSoCPath]\RVfpgaSoC\Labs\LabResources\Lab2\verilatorSIM\Vrvfpgasim.exe

1. Run the simulation by clicking on the PlatformIO icon in the left menu ribbon , then expand Project Tasks → env:swervolf\_nexys → Platform and click on Generate Trace, as shown in Figure 5.



**Figure 5. Generating trace from Verilator**

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

1. A few seconds after the previous step, file *trace.vcd* should have been generated inside *[RVfpga-SoCPath]/RVfpga-SoC/Labs/LabResources/Lab2/examples/AL\_Operations/.pio/build/swervolf\_nexys*, and you can open it with *GTKWave*. Open Ubuntu terminal and type :

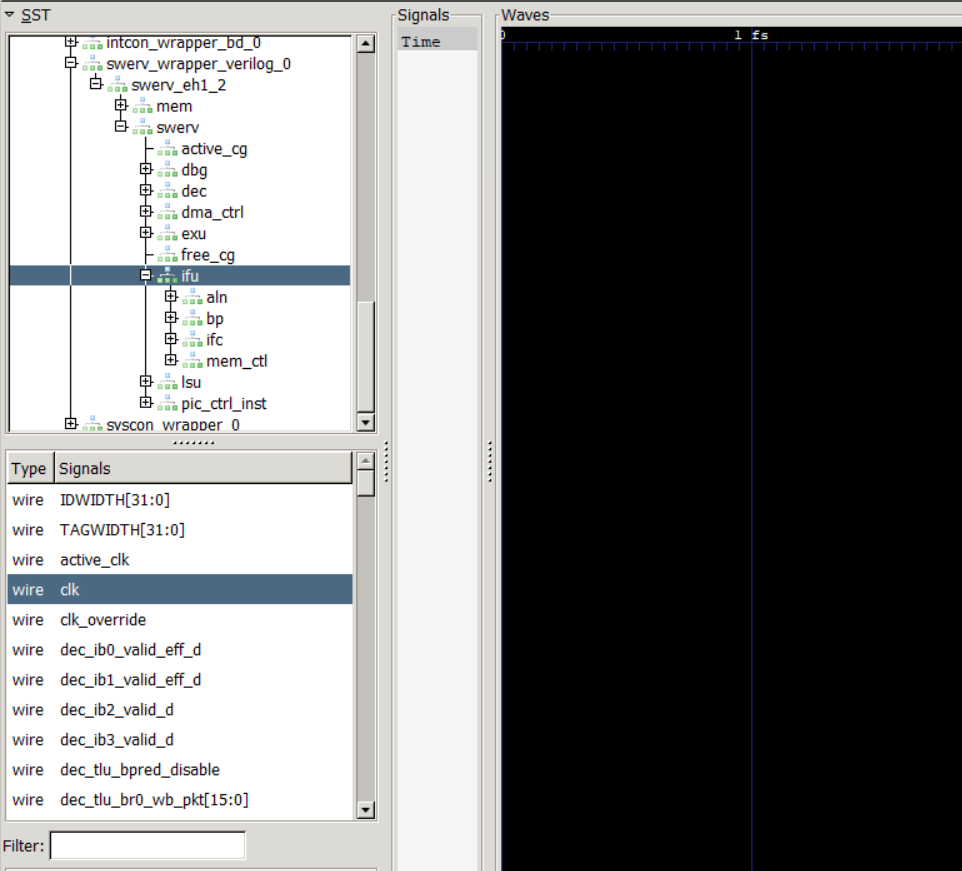
gtkwave [RVfpgaSoCPath]/RVfpgaSoC/Labs/LabResources/Lab2/examples/AL\_Operations/.pio/build/swervolf\_nexys/trace.vcd

**WINDOWS:** folder *gtkwave64* that you downloaded includes an application called *gtkwave.exe* inside the *bin* folder. Launch GTKWave by double-clicking on that application. On the top part of the application, click on **File – Open New Tab**, and open the trace.vcd file generated in the folder *[RVfpgaSoCPath]/RVfpgaSoC/Labs/LabResources/Lab2/examples/AL\_Operations*

*/.pio/build/swervolf\_nexys*.

**Step 3. Analyze the simulation in GTKWave**

1. Now we will add a clock, instruction, and register signals. On the top left pane of *GTKWave*, expand the SoC hierarchy so that you can add signals to the graph. Expand the hierarchy into **TOP → rvfpgasim → swervolf → swerv\_wrapper\_verilog → swerv\_eh1\_2 → swerv,** and click on module **ifu** (it will highlight as shown in Figure 6), 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.



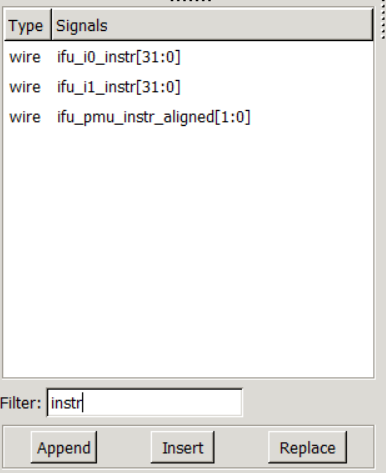
**Figure 6. Add signal *clk* to the graph**

1. Do a Zoom Fit and then Zoom in several times so that you can view the clock signal change (Figure 7).



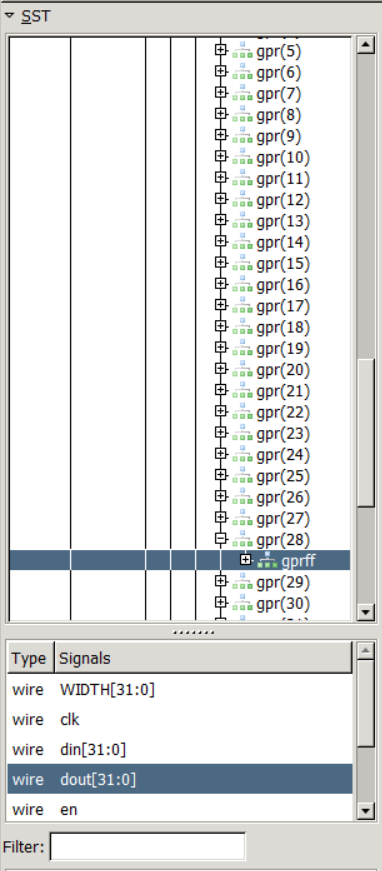
**Figure 7. Zoom in**

1. Now add the signals that show the instructions that execute each way of the two-way superscalar RISC-V core. In the same module (**ifu**) look for signals *ifu\_i0\_instr[31:0]* and *ifu\_i1\_instr[31:0]* (Figure 8), and drag them into the black Waves pane. The prefix *ifu* indicates the instruction fetch unit, *i0* indicates superscalar way 0, and *i1* indicates superscalar way 1; *instr*[31:0] indicates the 32-bit instruction.



**Figure 8. Add signals *ifu\_i0\_instr[31:0]* and *ifu\_i1\_instr[31:0]* to the timing waveform**

1. Now add the signal that holds the value of register t3 (i.e., register number 28, x28). Expand the hierarchy under **swerv** into **dec → arf → gpr\_banks(0) → gpr(28)** and click on module **gprff** (it will highlight as shown in the following figure), select signal *dout[31:0]* (which shows the contents of register x28, used in the *AL\_Operations.S* example) and drag it into the black Waves pane (Figure 9).



**Figure 9. Add signal *dout[31:0]* to the graph**

Figure 10 shows the AL\_Operations.S program and its equivalent machine instructions.

**# RISC-V assembly # comment (t3 = x28) # machine code**

**li t3, 0x0 # t3 = 0 # 0x00000E13**

**REPEAT:**

**addi t3, t3, 6 # t3 = t3 + 6 # 0x006E0E13**

**addi t3, t3, -1 # t3 = t3 – 1 # 0xFFFE0E13**

**andi t3, t3, 3 # t3 = t3 AND 3 # 0x003E7E13**

**beq zero, zero, REPEAT # Repeat the loop # 0xFE000CE3**

**nop # nop # 0x00000013**

**Figure 10. 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 11as the program runs:

li t3, 0x0 # t3 = **0** # 0x00000E13

REPEAT: addi t3, t3, 6# t3 = 0 + 6 = **6** # 0x006E0E13

addi t3, t3, -1# t3 = **5** # 0xFFFE0E13

andi t3, t3, 3 # t3 = 5 & 3 = **1** # 0x003E7E13

beq zero, zero, REPEAT # Repeat the loop# 0xFE000CE3

nop # nop# 0x00000013

REPEAT: addi t3, t3, 6 # t3 = 1 + 6 = **7** # 0x006E0E13

addi t3, t3, -1# t3 = 7 – 1 = **6** # 0xFFFE0E13

andi t3, t3, 3# t3 = 6 & 3 = **2** # 0x003E7E13

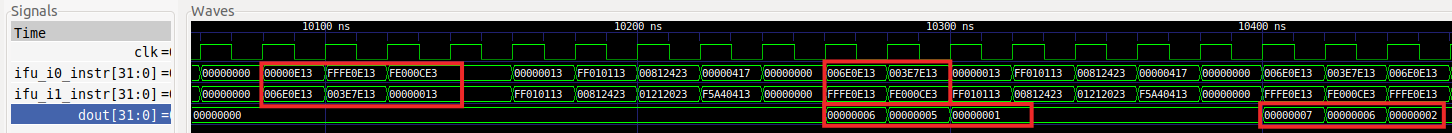
beq zero, zero, REPEAT# Repeat the loop # 0xFE000CE3

...

**Figure 11. Instruction flow and values of register t3 (x28) during AL\_Operations execution**

1. Zoom in around 10,100 ns, where you will analyse the execution of the three arithmetic-logic instructions of the first and second iterations of the loop (Figure 12). The first two instructions (li t3, 0x0 = 0x00000E13 and addi t3, t3, 6 = 0x006E0E13) are fetched first, one in each way of the superscalar RISC-V processor as shown on signals *ifu\_i0\_instr*[31:0] and *ifu\_i1\_instr*[31:0]. The next two instructions (addi t3, t3, -1 = 0xFFFE0E13 and and.i t3, t3, 3 = 0x003E7E13) are fetched in the next cycle. The last two instructions are fetched (beq zero, zero, REPEAT = 0xFE000CE3 and nop = 0x00000013) in the next cycle.

Because of the SweRV core’s 9-stage pipelined processor and dependencies, the instructions’ effects are seen eight or more cycles after the instructions are fetched. Eight cycles after the first and second instructions are fetched, x28 (t3) becomes 0 (which it was already) because of the first instruction: li t3, 0x0 (0x00000E13). One cycle later, x28 is updated to 0x6 because of the next instruction: addi t3, t3, 6 (0x006E0E13). Next, x28 updates to 5 because of the next instruction: addi t3, t3, -1 (0xFFFE0E13). Finally, x28 updates to 1 because of the next instruction: andi t3, t3, 3 (0x003E7E13). The next two instructions are fetched: beq zero, zero,REPEAT (0xFE000CE3), and nop (0x00000013); the branch is taken, and the loop repeats.



**Figure 12. Execution of the three Arithmetic-Logic instructions from the example**

# Running A Program on Nexys A7 Board

This section will program the FPGA with the RVfpga-SoC, which uses PlatformIO. Follow the next steps for programming the FPGA with the RVfpga-SoC:

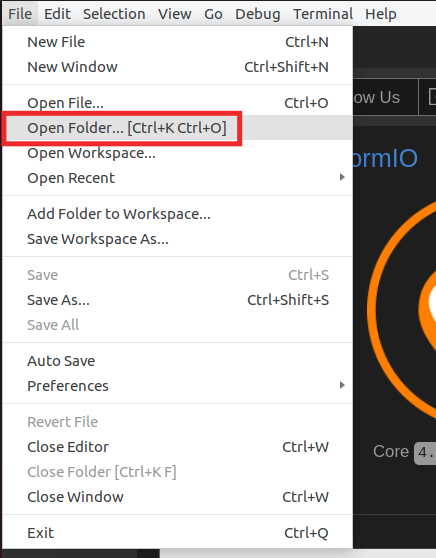
Follow the following steps :

1. Connect the Nexys A7 board to your computer.
2. Turn on the Nexys A7 board using the switch at the top left. (see Figure 13)



**Figure 13. Nexys A7 Board ON/OFF Button**

1. Open VSCode and PlatformIO if it is not already open.
2. On the top menu bar, click on *File* → *Open Folder* (see Figure 14) and browse into directory *[RVfpgaSoCPath]/RVfpgaSoC/Labs/LabResources/Lab2/examples/*

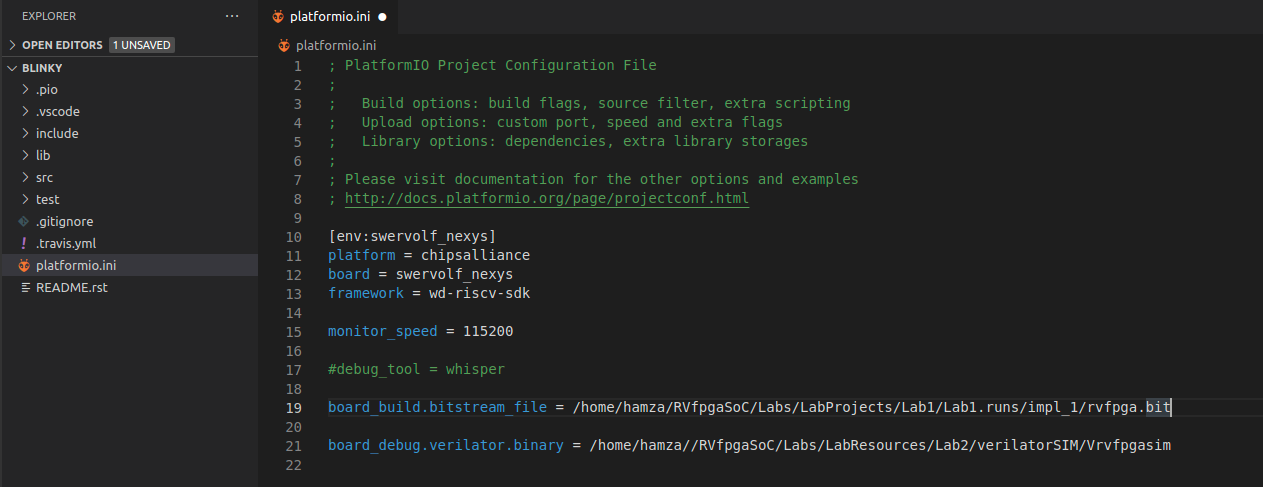


**Figure 14. Open Folder**

1. Select the directory *Blinky* (do not open it, but just select itand click OK at the top of the window. PlatformIO will now open the example.
2. Open file *platformio.ini* by clicking on *platformio.ini* in the left sidebar (see Figure 15). Establish the path to the RVfpga bitstream in your system by editing the following line (see Figure 15).
3. The bit file created using Vivado Block Design is at the following path :

board\_build.bitstream\_file = [RVfpgaSoCPath]/RVfpgaSoC/Labs/LabProjects/Lab1/Lab1.runs/impl\_1/

rvfpga.bit

****

**Figure 15. Platformio initialization file: platformio.ini**

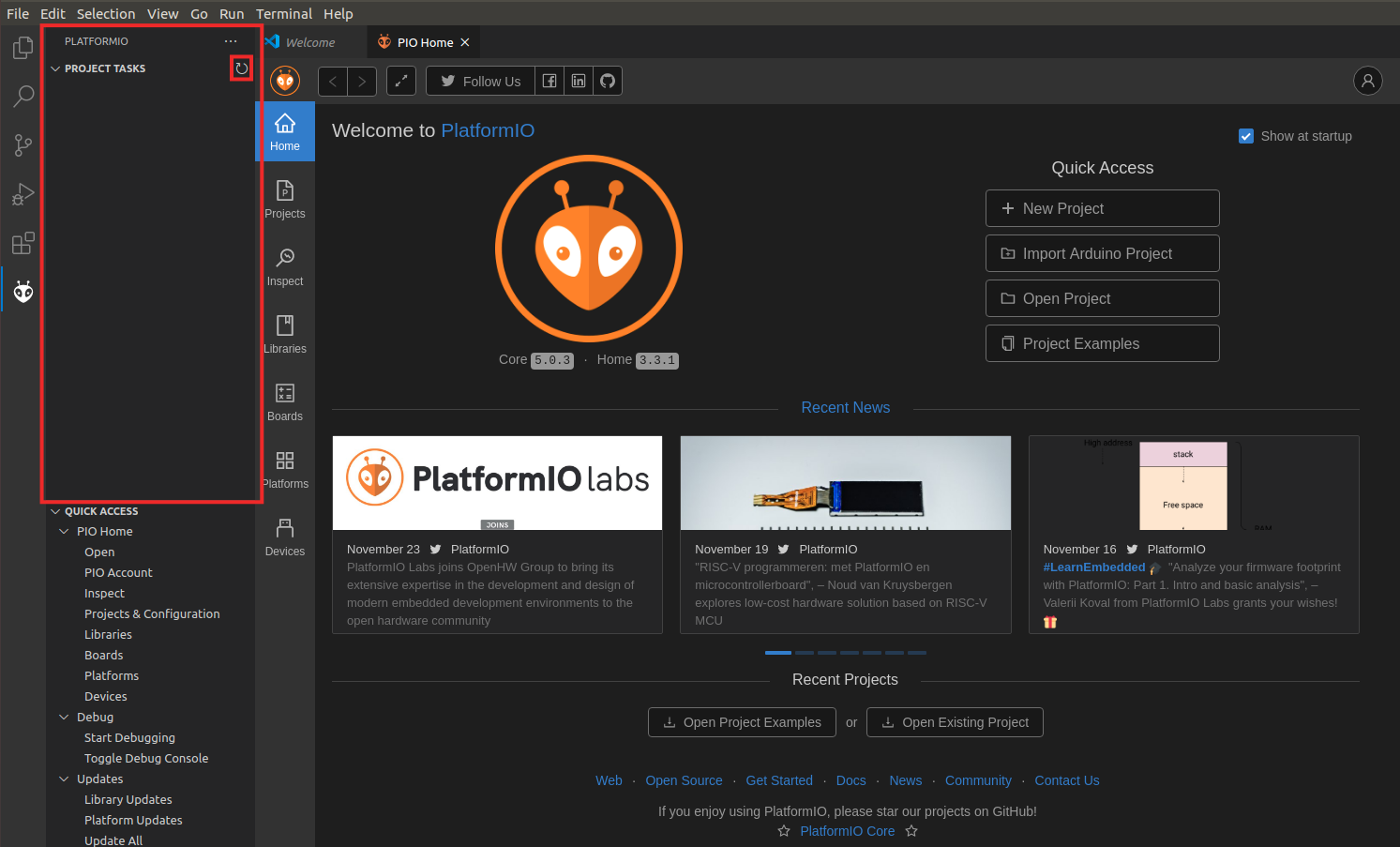
There are many different commands that you can use in the Project Configuration File (*platformio.ini*), and for which you can find information at <https://docs.platformio.org/en/latest/projectconf/>.

1. Click on the PlatformIO icon  in the left menu ribbon (see Figure 16).



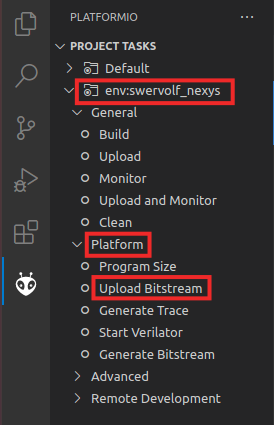
**Figure 16. PlatformIO icon**

In case the Project Tasks window is empty (Figure 17), you must refresh the Project Tasks first by clicking on . This can take several minutes.



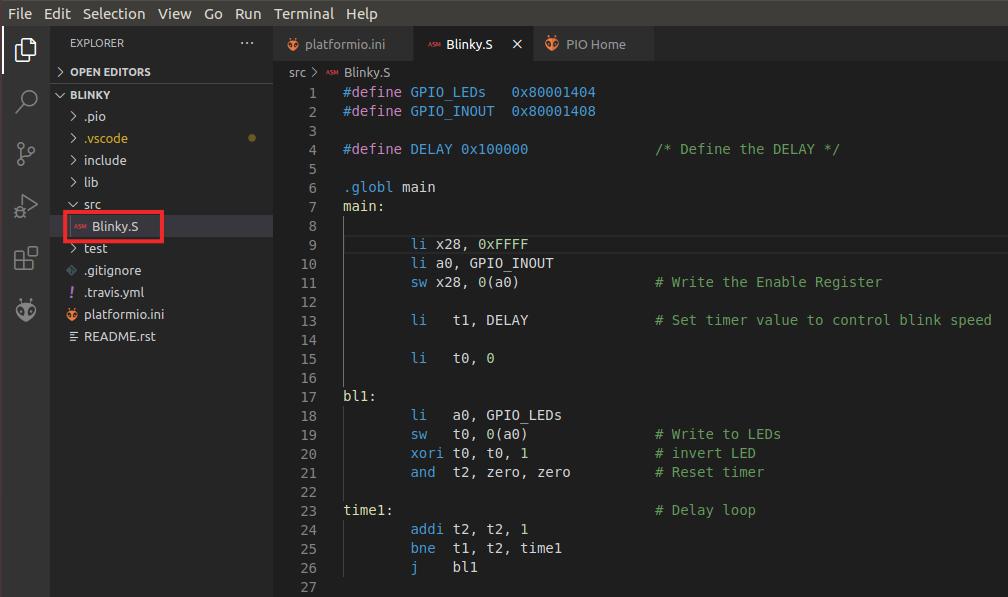
**Figure 17. PROJECT TASKS window empty – Refresh**

Expand Project Tasks → env:swervolf\_nexys → Platform and click on Upload Bitstream, as shown in Figure 18. **After one or two seconds, the FPGA will be programmed with the RVfpga-SoC** (the 7-Segment Displays available on the board should output 8 zeros).



**Figure 18. Upload Bitstream**

Now that the bitstream has been uploaded, we will start the debugging process.

****

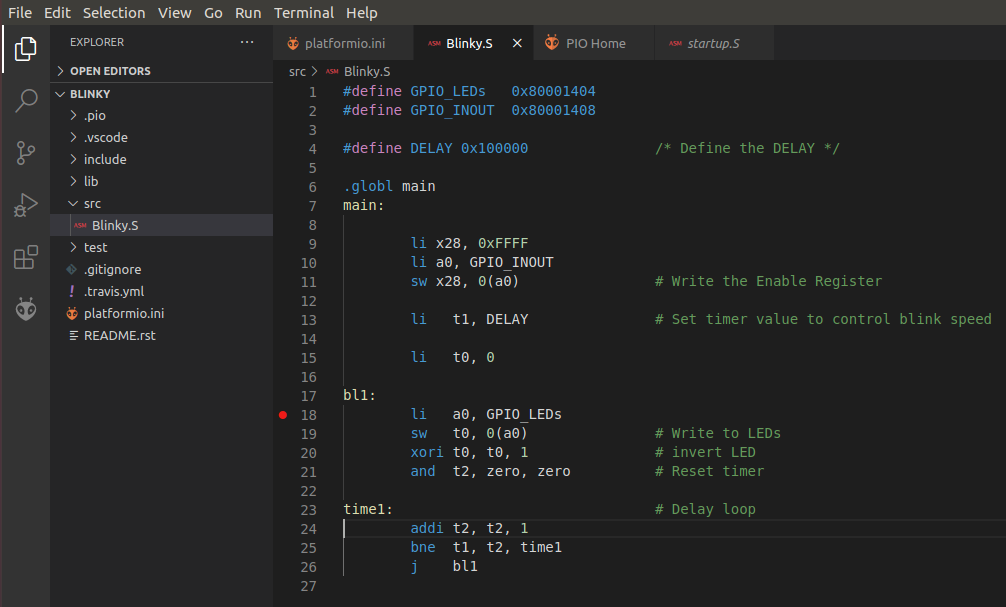
**Figure 23. blinky.S in PlatformIO**

1. Click on  to run and debug the program; then start debugging by clicking on the play button . PlatformIO sets a temporary breakpoint at the beginning of the main function. So, click on the Continue button  to run the program.
2. On the board, you will see the right-most LED start to blink.



**Figure 24. rightmost LED Blinking**

1. Pause the execution by clicking on the pause button . The execution will stop somewhere inside the infinite loop (probably, inside the time1 delay loop).
2. Create 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 24).

****

**Figure 25. Setting a breakpoint in blinky.S**

1. Then, continue execution by clicking on the Continue button . Execution will continue, and it will stop after the store byte (sb) instruction, which writes 1 (or 0) to the right-most LED.
2. Continue execution several times; you will see that the value-driven to the right-most LED changes each time.
3. Stop debugging  and go back to the Explorer window by clicking on . Close the program by selecting *File* → *Close Folder.*