THE IMAGINATION UNIVERSITY PROGRAMME

RVfpga-SoC Lab 2
Running Software on
RVfpga-SoC

Table 1. RVfpga Terms

Name	Description
Courses	
RVfpga	A course that shows how to use RVfpgaNexys and RVfpgaSim, RISC-V system-on-chips (SoCs), to run programs and extend the system by adding peripherals (RVfpga Labs 1-10), and explore the core and memory system by running simulations, measuring performance, adding instructions, and modifying the memory system (RVfpga Labs 11-20). Throughout the course, users are also shown how to use the RISC-V toolchain (compilers and debuggers) and simulators, the Verilator HDL simulator, and Western Digital's Whisper instruction set simulator (ISS).
RVfpga-SoC	A course that shows how to build a subset SweRVolfX SoC from scratch using building blocks such as the SweRV core, memories, and peripherals. The course also shows how to load the Zephyr real-time operating system (RTOS) onto SweRVolf and run programs including Tensorflow Lite's hello world example on top of the operating system.
Cores and SoCs	
SweRV EH1 Core	Open-source commercial RISC-V core developed by Western Digital (https://github.com/chipsalliance/Cores-SweRV).
SweRV EH1 Core Complex	SweRV EH1 core with added memory (ICCM, DCCM, and instruction cache), programmable interrupt controller (PIC), bus interfaces, and debug unit (https://github.com/chipsalliance/Cores-SweRV).
SweRVolfX	The System on Chip that we use in the RVfpga course. It is an extension of SweRVolf. **SweRVolf** (https://github.com/chipsalliance/Cores-SweRVolf): An open-source SoC built around the SweRV EH1 Core Complex. It adds a boot ROM, UART interface, system controller, interconnect (AXI Interconnect, Wishbone Interconnect, and AXI-to-Wishbone bridge), and an SPI controller. **SweRVolfX**: It adds four new peripherals to SweRVolf: a GPIO, a PTC, an additional SPI, and a controller for the 8 Digit 7-Segment Displays.
RVfpgaNexys	The SweRVolfX SoC targeted to the Nexys A7 board and its peripherals. It adds a DDR2 interface, CDC (clock domain crossing) unit, BSCAN logic (for the JTAG interface), and clock generator. RVfpgaNexys is the same as SweRVolf Nexys (https://github.com/chipsalliance/Cores-SweRVolf), except that the latter is based on SweRVolf.
RVfpgaSim	The SweRVolfX SoC with a testbench wrapper and AXI memory intended for simulation. RVfpgaSim is the same as SweRVolf Sim, (https://github.com/chipsalliance/Cores-SweRVolf), except that the latter is based on SweRVolf.

1. Introduction

This Lab shows how to run programs written in C or Assembly language on the SweRVolfX subset we created in Lab 1 using the Vivado Block design tool. You may choose to simulate the design using Verilator 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 Block Design'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 simulation binaries 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 **RVfpgaNexys**, 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.

2. Requirements

To complete this lab, you will need to install the following tools:
- VSCode   (Refer to Installation Guide (Page No.06))
- PlatformIO   (Refer to Installation Guide (Page No.06))
- GTKWave   (Refer to Installation Guide (Page No.08))
- Verilator   (Refer to Installation Guide (Page No.08))
- Cygwin (For Windows User only)   (Refer to Installation Guide (Page No.13))

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

For example, if you have not already, install VScode 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.

3. Running the SoC Created in Block Design

In the first Lab, we created a subset of the SweRVolfX SoC 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 Block Design module as a whole. In our case, it was the "**BD.v**" file.

Now we have two options and pathways to run the Block Design's SoC
- Run Block Design's SoC on the Nexys A7 100T board.
- Run Block Design's SoC on the Verilator simulator.

The Block Design's SoC has two top-level modules that exist for each of those targets: RVfpgaSim (rvfpgasim) and RVfpga Nexys (rvfpga), as described below:

1. **RVfpgaSim** (rvfpgasim.v)

The RVfpgaSim module is used as the top module of the RVfpga system for **Simulation**. We use Verilator (a hardware description language (HDL) simulator that simulates the Verilog that defines RVfpga) for simulating the RVfpga system. Running the SoC in simulation allows us to analyze the system's internal signals in depth. Later in this Lab, when we generate 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.
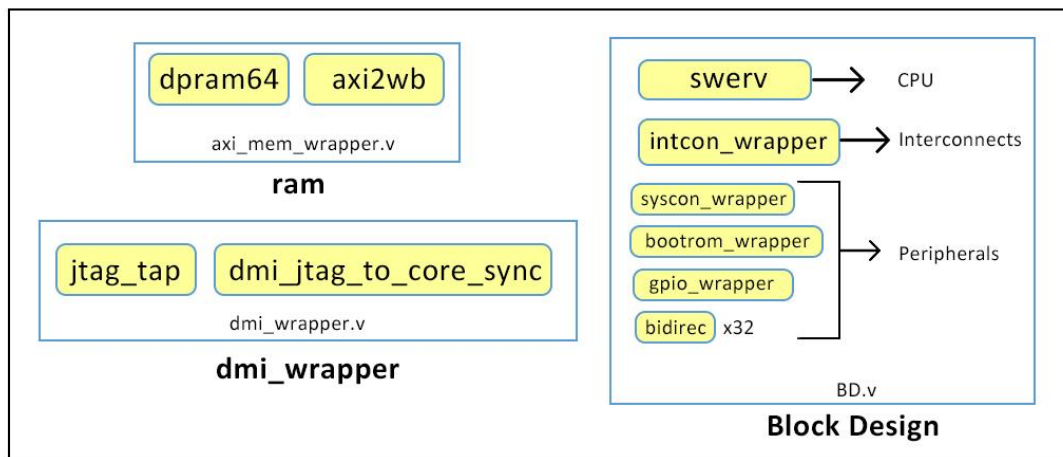
Figure 1. RVfpgaSim

It includes three modules:
- Block Design (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 several external pin connections. These external connections of the **"Block Design"** module are connected in the top module "**rvfpgasim**" with other modules. For instance, the "**DMI**" external connections in the "**Block Design**" module are connected with the "**dmi_wrapper**" module, and the "**RAM**" external connections of the "**Block Design**" module are connected with the "**ram**" module.

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

The RVfpga Nexys module is used as the top module of the RVfpga system 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.

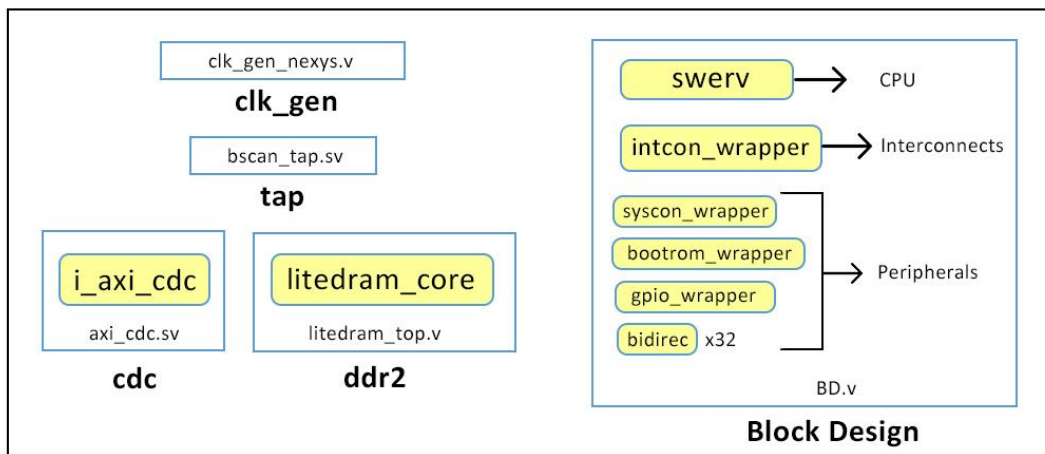**Figure 2. RVfpga Nexys**

RVfpga Nexus includes five modules :
- Block Design(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
- cdc (axi_cdc.sv)
  - This is the Clock Domain Crossing module.

In the "**rvfpga.sv**" top module, the "**RAM**" external connections of the "**Block Design**" module are connected with the "**ddr2**" module. The "**DMI**" external connections of "**Block Design**" are connected with the "**bscan_tap.sv**" module. The "**clk**" external connection is connected with the "**clk_gen**" module (see Figure 3).
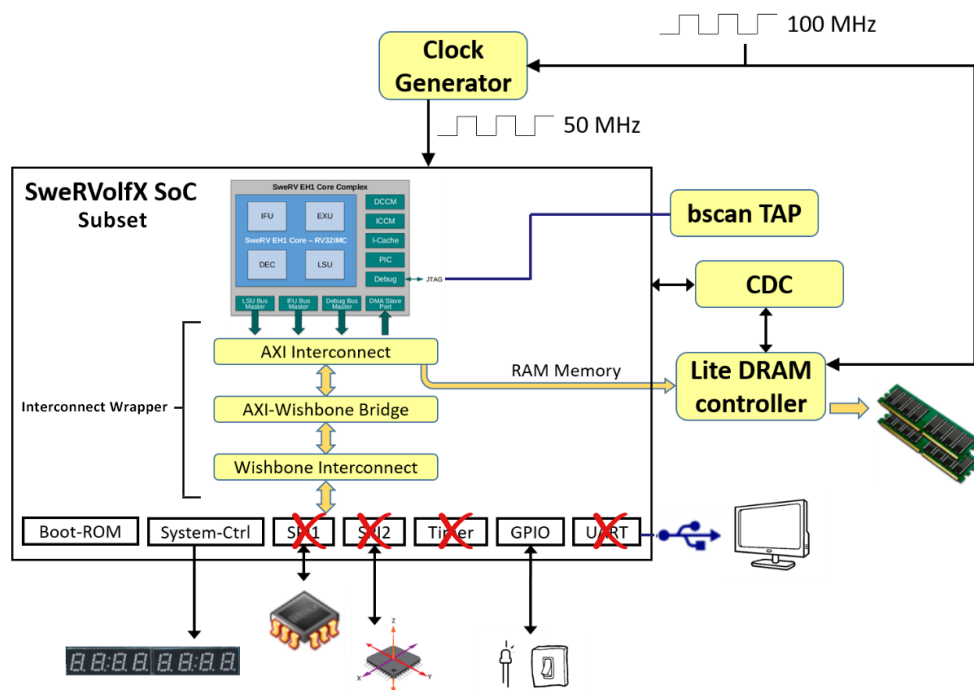
Figure 3. RVfpga Nexys (SweRVolfX SoC Subset)

4. 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.

> **Note**: All RVfpga Verilog source modules need to be prefixed with "BD_" to work with the Block Design generated "BD.v" file. This "BD.v" file is used by the top module "rvfpgasim" for simulation on verilator. The source modules are instantiated in the "BD.v" file and it requires all the modules used to be prefixed with "BD_". We have already done that for you in a separate "src" folder at the following path :
> `[RVfpgaSoCPath]/RVfpgaSoC/Labs/LabResources/Lab2/src.`

First, we will need to move our "**Block Design**" 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 previous lab, we were provided with its full path (see Figure 4). We will need to navigate this path and then copy the file.

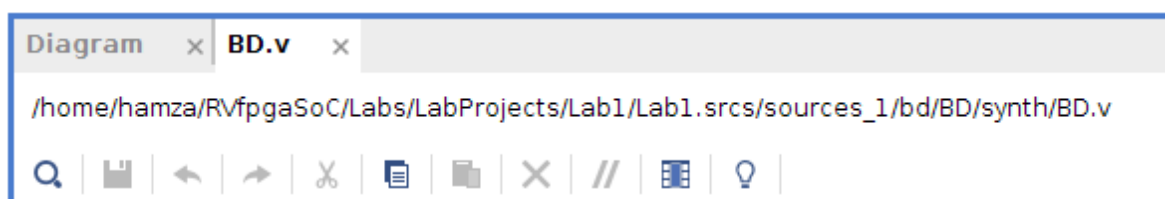`[RVfpgaSoCPath]/RVfpgaSoC/Labs/LabProjects/Lab1/Lab1.srcs/sources_`
`1/bd/BD/synth/BD.v`

Figure 4. The path of the "BD.v" Verilog file of "Block Design" module

Step 1. Copy the "BD.v" file from the path given in (Figure 4) and paste the "**BD.v**" file to the following path (see Figure 5):

```
[RVfpgaSoCPath]/RVfpgaSoC/Labs/LabResources/Lab2/src/SweRVolfSoC/
```
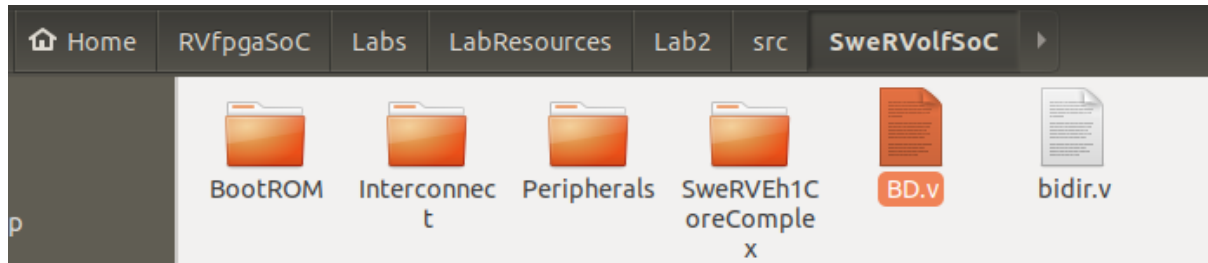

Figure 5. "BD.v" Pasted in the "SweRVolfSoC" directory.

Step 2. Open the "BD.v" file and make sure the following module's name ends with "**_0_0**" (see Figure 6).
- BD_bootrom_wrapper_0_0
- BD_gpio_wrapper_0_0
- BD_intcon_wrapper_bd_0_0
- BD_swerv_wrapper_verilog_0_0
- BD_syscon_wrapper_0_0

Note: This is done to keep consistency in the naming of the modules for simulation. If they are not consistent then we will receive an error while generating the simulation binary for RvfpgaSim in the next step.

```
BD_bootrom_wrapper_0_0 bootrom_wrapper_0
    (.i_clk(clk_0_1),
     .i_rst(rst_0_1),
     .i_wb_adr(intcon_wrapper_bd_0_wb_rom_adr_o),
     .i_wb_cyc(intcon_wrapper_bd_0_wb_rom_cyc_o),
     .i_wb_dat(intcon_wrapper_bd_0_wb_rom_dat_o),
     .i_wb_sel(intcon_wrapper_bd_0_wb_rom_sel_o),
     .i_wb_stb(intcon_wrapper_bd_0_wb_rom_stb_o),
     .i_wb_we(intcon_wrapper_bd_0_wb_rom_we_o),
     .o_wb_ack(bootrom_wrapper_0_o_wb_ack),
     .o_wb_rdt(bootrom_wrapper_0_o_wb_rdt));
```

Figure 6. "BD.v"

Now we will start with the process of running the *AL_Operations* Program on our Block Design's 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 3. 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/src*.

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

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

> ➢ ```
> cd
> [RVfpgaSoCPath]/RVfpgaSoC/Labs/LabResources/Lab2/verilat
> orSIM
> ```
> ➢ ```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 4. 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 7), and browse into directory *[RVfpgaSoCPath]/RVfpgaSoC/Labs/LabResources/Lab2/examples/*
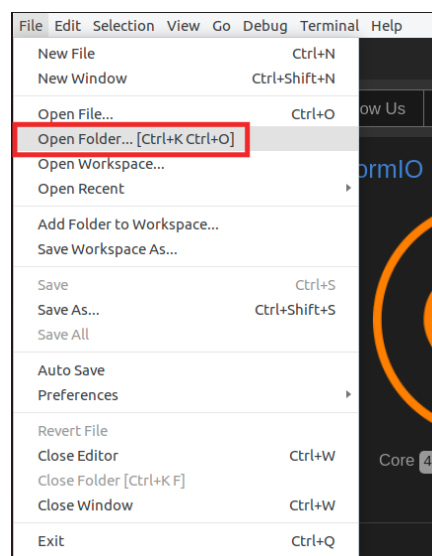

Figure 7. Open the AL_Operations.S example

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

4. 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 8).

```
board_debug.verilator.binary =
[RVfpgaSoCPath]/RVfpgaSoC/Labs/LabResources/Lab2/verilatorSIM/Vrvfpgasim
```
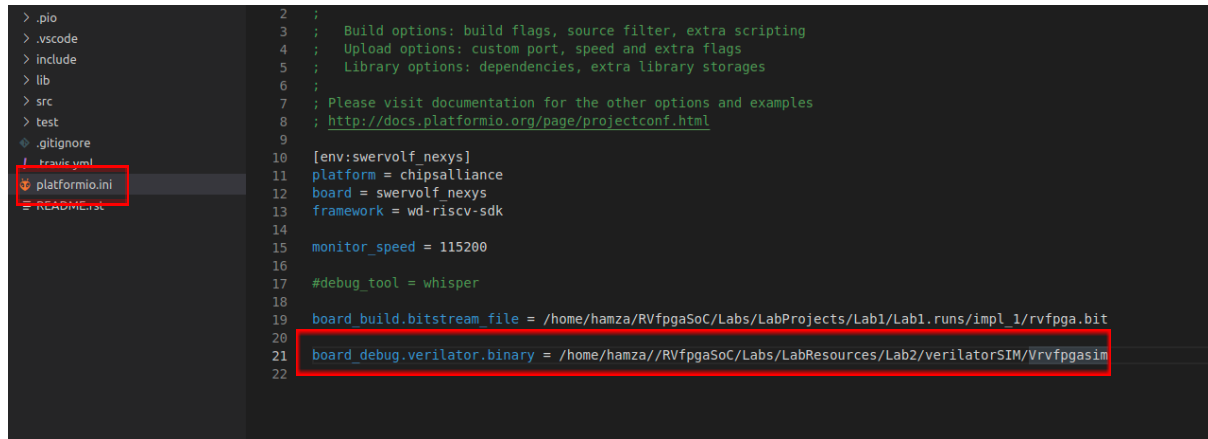

Figure 8. PlatformIO initialization file: platformio.ini

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

```
 board_debug.verilator.binary =
[RVfpgaSoCPath]\RVfpgaSoC\Labs\LabResources\Lab2\verilatorSIM\Vrvfpgasim.exe
```

5. 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 9.

Figure 9. Generating trace from Verilator

You can generate the trace from a PlatformIO terminal window as an alternative. For that purpose, click on the [image] 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`

6. A few seconds after the previous step, file *trace.vcd* should have been generated inside *[RVfpgaSoCPath]/RVfpgaSoC/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 5. 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_0 → swerv_eh1_2 → swerv,** and click on module **ifu** (it will highlight as shown in Figure 10),

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.
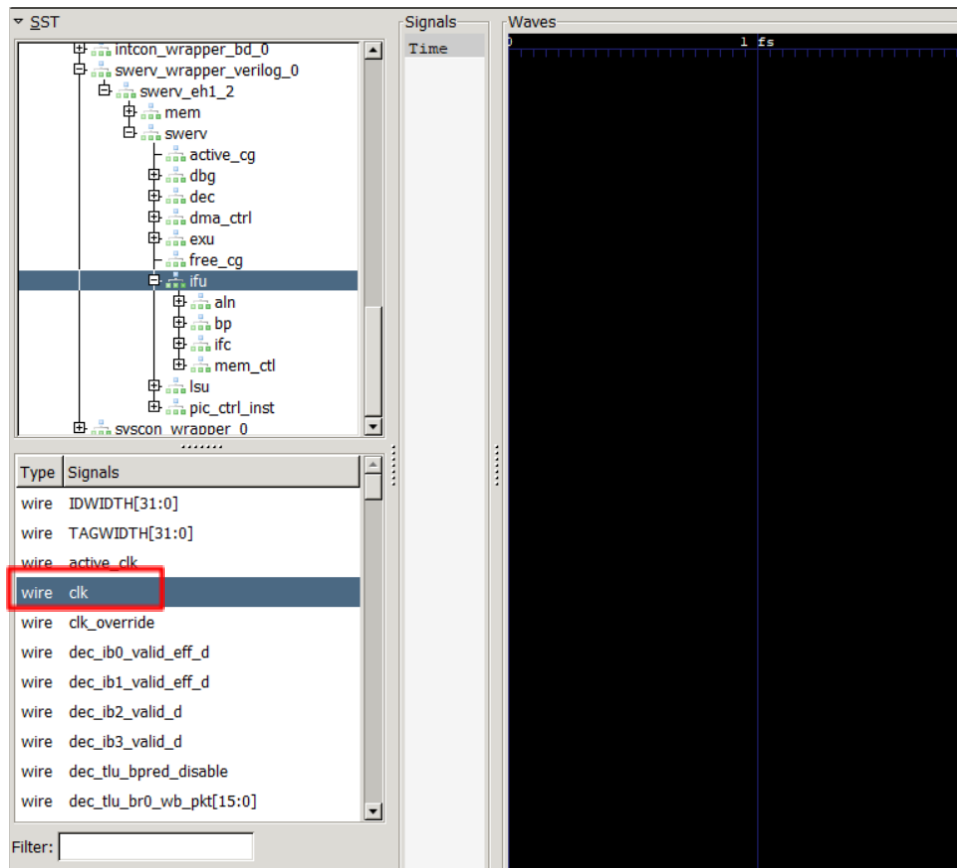

Figure 10. Add signal *clk* to the graph

2. Do a Zoom Fit and then Zoom in several times so that you can view the clock signal change (Figure 11).
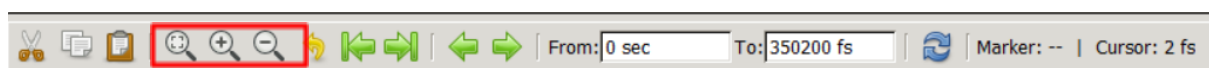

Figure 10. Zoom in

3. 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 12), 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.
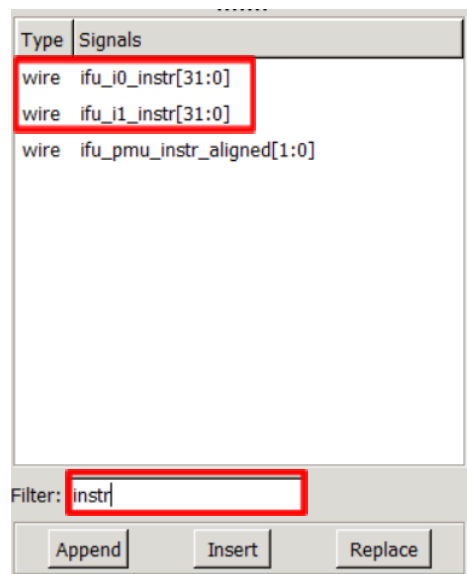4. You can use the search filter to find the signals quickly (see Figure 12).

Figure 12. Add signals *ifu_i0_instr[31:0]* and *ifu_i1_instr[31:0]* to the timing waveform

5. 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 13).

Figure 13. Add signal *dout[31:0]* to the graph

6. Another way of showing signals in GTKWave is to use a *.tcl* file. File *gtkwave_signals.tcl* is provided at *[RVfpgaSoCPath]/RVfpgaSoC/Labs/LabResources/Lab2/*. Open that file and analyze it. In each line, you will see the path and the name of each signal that we want to show in the graph.

```
gtkwave::addSignalsFromList rvfpgasim.clk
gtkwave::addSignalsFromList
rvfpgasim.swervolf.swerv_wrapper_verilog_0.swerv_eh1_2.swerv.ifu.ifu_i0_instr
gtkwave::addSignalsFromList
rvfpgasim.swervolf.swerv_wrapper_verilog_0.swerv_eh1_2.swerv.ifu.ifu_i1_instr
gtkwave::addSignalsFromList
rvfpgasim.swervolf.swerv_wrapper_verilog_0.swerv_eh1_2.swerv.dec.arf.gpr_banks(0).gpr(28).gprff.dout
```

For using the *.tcl* file on GTKWave, you can simply click on *File – Read Tcl Script File* and select the *RVfpgaSoCPath]/RVfpgaSoC/Labs/LabResources/Lab2/gtkwave+signals.tcl* file.

Figure 14 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
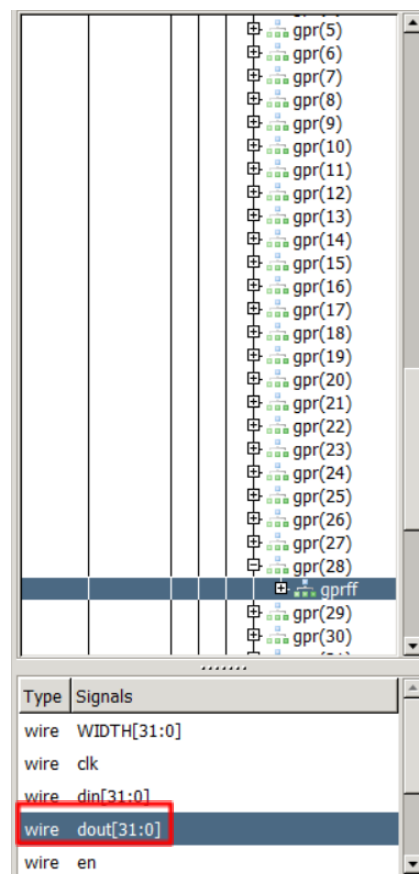
REPEAT:
    addi t3, t3, 6          # t3 = t3 + 6             # 0x006E0E13
    addi t3, t3, -1         # t3 = t3 - 1             # 0xFFFE0E13
    andi t3, t3, 3          # t3 = t3 AND 3           # 0x003E7E13

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

Figure 14. 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 15 as the program runs:

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

Figure 15. Instruction flow and values of register `t3` (`x28`) during AL_Operations execution

7. 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 16). 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 16. Execution of the three Arithmetic-Logic instructions from the example

5. Running A Program on Nexys A7 Board

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

Follow the following steps :
Step 1. Connect the Nexys A7 board to your computer.

Step 2. Turn on the Nexys A7 board using the switch at the top left. (see Figure 17)

Figure 17. Nexys A7 Board ON/OFF Button

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

Step 4. On the top menu bar, click on *File → Open Folder* (see Figure 18) and browse into directory *[RVfpgaSoCPath]/RVfpgaSoC/Labs/LabResources/Lab2/examples/*
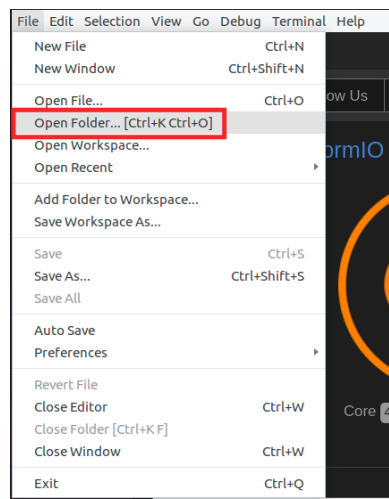
Figure 18. Open Folder

Step 5. Select the directory *Blinky* (do not open it, but just select it and click OK at the top of the window. PlatformIO will now open the example.

Step 6. Open file *platformio.ini* by clicking on *platformio.ini* in the left sidebar (see Figure 19). Establish the path to the RVfpga bitstream in your system by editing the following line (see Figure 19).
Step 7. The "rvfpga.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 19. 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/.

Step 8. Click on the PlatformIO icon in the left menu ribbon (see Figure 20).

Figure 20. PlatformIO icon

In case the Project Tasks window is empty (Figure 21), you must refresh the Project Tasks first by clicking on . This can take several minutes.
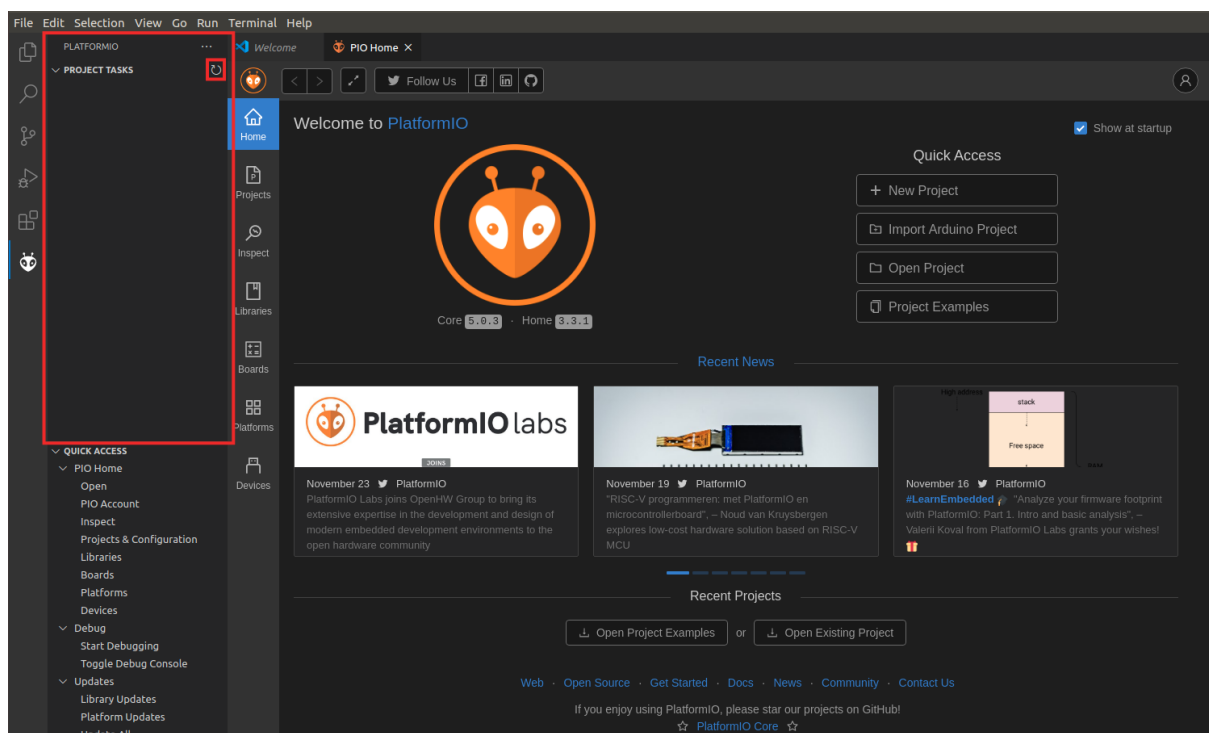

Figure 21. PROJECT TASKS window empty – Refresh

Expand Project Tasks → env:swervolf_nexys → Platform and click on Upload Bitstream, as shown in Figure 22. **After one or two seconds, the FPGA will be programmed with our Block Design SoC** (the 7-Segment Displays available on the board should output 8 zeros).
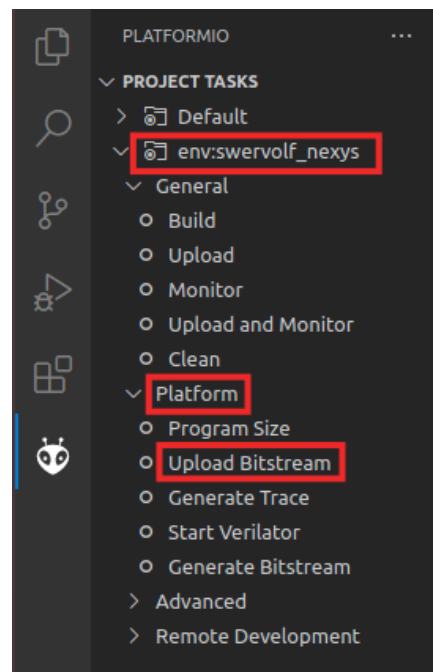
Figure 22. Upload Bitstream

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

Figure 23. blinky.S in PlatformIO

Step 9. Click on [icon] to run and debug the program; then start debugging by clicking on the play button [RUN ▷ PIO Debug ⌄]. PlatformIO sets a temporary breakpoint at the beginning of the main function. So, click on the Continue button [icon] to run the program.

Step 10. On the board, you will see the right-most LED start to blink.
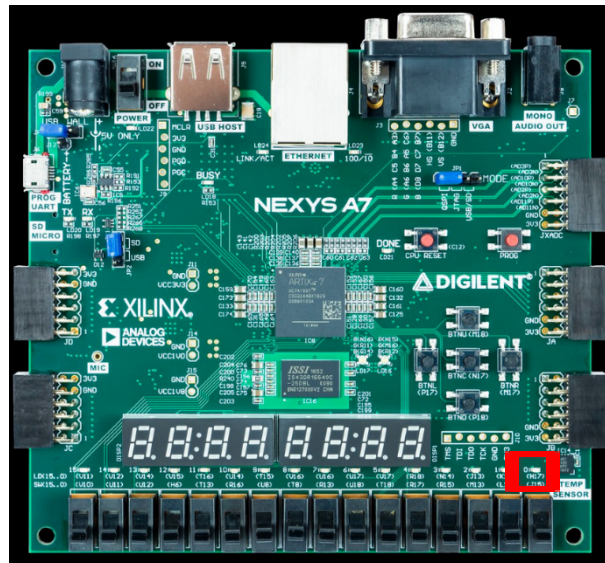

Figure 24. rightmost LED Blinking

Step 11. Pause the execution by clicking on the pause button

. The execution will stop somewhere inside the infinite loop (probably, inside the `time1` delay loop).

Step 12. 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 25).
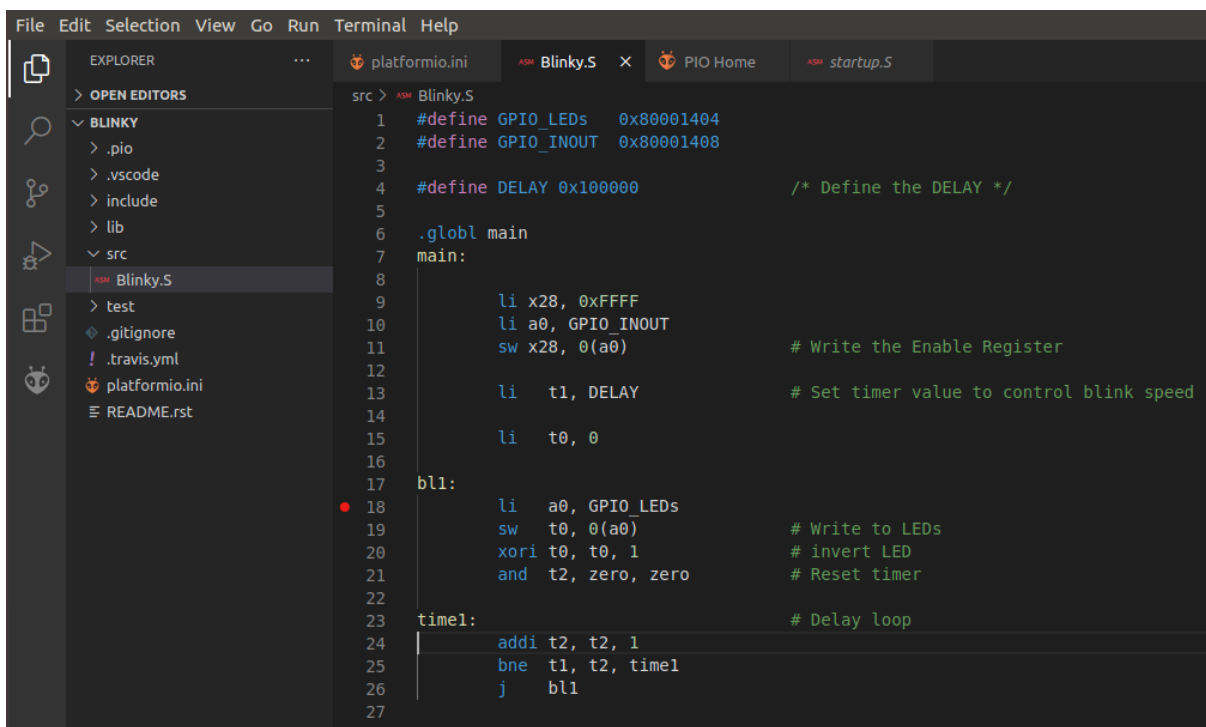

Figure 25. Setting a breakpoint in blinky.S

Step 13. 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.

Step 14. Continue execution several times; you will see that the value-driven to the right-most LED changes each time.

Step 15. Stop debugging and go back to the Explorer

window by clicking on . Close the program by selecting *File → Close Folder.*

So we have successfully run the example programs on the RVfpgaSIM and RVfpgaNexys using the Block Design module that we had created in lab 1.