THE IMAGINATION UNIVERSITY PROGRAMME

SweRV EH1 Reference

Hierarchy, Modules, Signals, and Types

This document provides extra instructions on the following topics:

- Section 1: **Sigasi Studio**
- Section 2: **Configuration of the SweRV EH1 processor**
- Section 3: **RVfpga System hierarchy of modules and their most relevant signals**
- Section 4: **Main structures/types for grouping control bits**
- Section 5: **RISC-V compressed instructions**
- Section 6: **Real Benchmarks**

1. SIGASI STUDIO

Sigasi Studio improves designer productivity by helping to write, inspect, and modify digital circuit designs in the most intuitive way. This tool understands the design context. Advanced features such as intelligent autocompletes and code refactoring make VHDL, Verilog and SystemVerilog design easier and more efficient.

Sigasi Studio requires a fee for obtaining a license and being able to use it professionally. Fortunately, there is a free license for educational purposes that you can easily obtain at: https://www.sigasi.com/try-form-edu/. Once you fill in your data and your license is approved, you will receive an e-mail with the instructions and a link for downloading (https://www.sigasi.com/download/, see Figure 1), installing, and using Sigasi Studio. The software is available for Windows, Linux and MacOS.

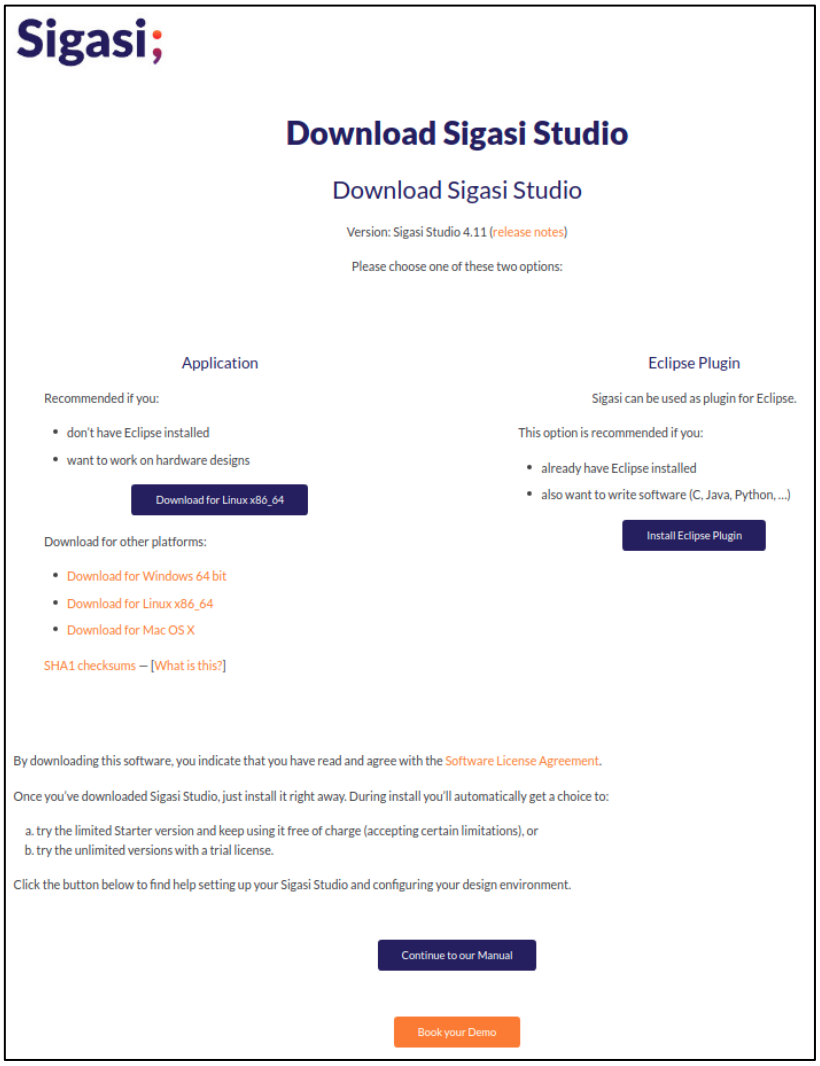

Figure 1. Link for downloading, installing, and using Sigasi Studio

Once you have installed Sigasi Studio in your system you can start using it for inspecting RVfpga. In the following link, two years ago, Hendrik Eeckhaut published instructions for creating and configuring a project for SweRV EH1: https://insights.sigasi.com/tech/swerv_riscv/. Using that information as a starting point, we next provide complete instructions for creating and configuring a project for RVfpga.

1. Create a copy of the *[RVfpgaPath]/RVfpga/src* directory and name it *[RVfpgaPath]/RVfpga/src_SigasiStudio*

2. Open Sigasi Studio by going into the downloaded directory and double-clicking on file *sigasi_internal* (see Figure 2).

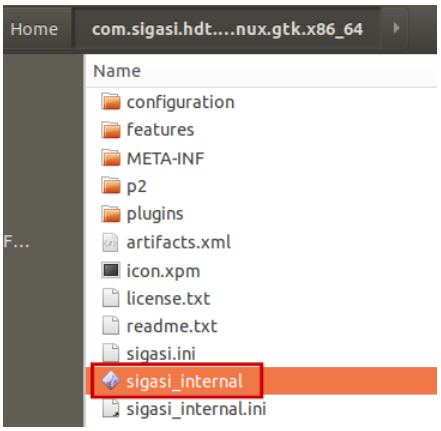

Figure 2. Open Sigasi Studio

3. On the Sigasi Studio window click on File → Import… A new window will open that asks you to select the type of project that you want to add to your system. Choose "Import a (System) Verilog project" and click next (Figure 3).

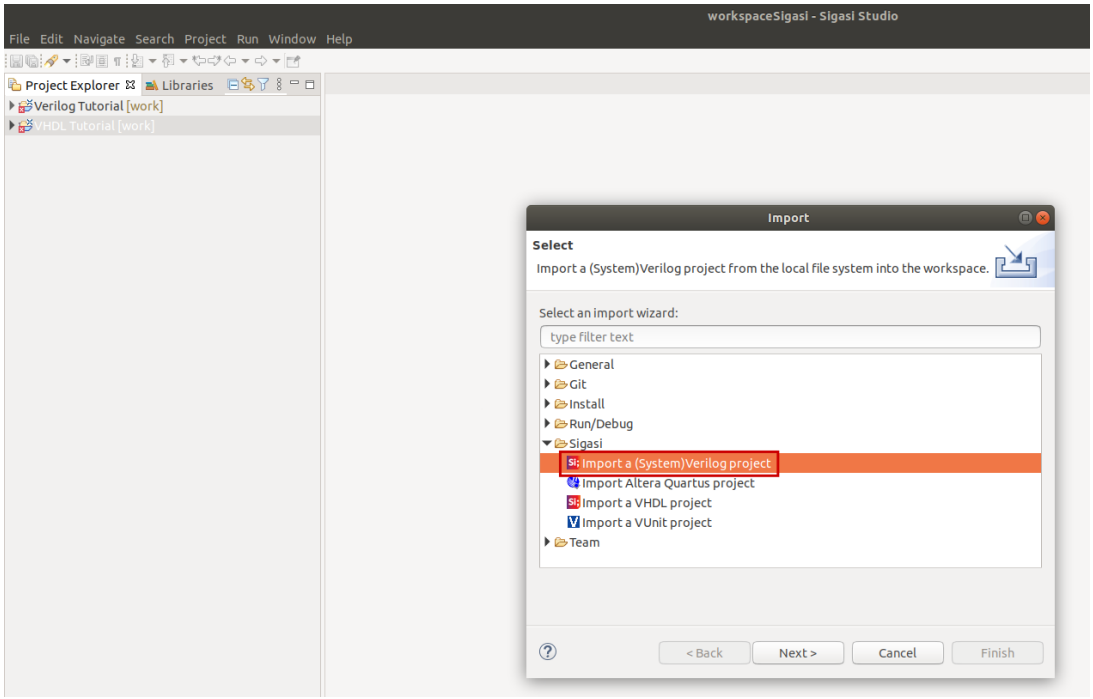

Figure 3. Import the RVfpga project

4. Now click on "Browse…" and navigate to and select the *src_SigasiStudio* directory and click Open (see Figure 4) and then click on Finish.

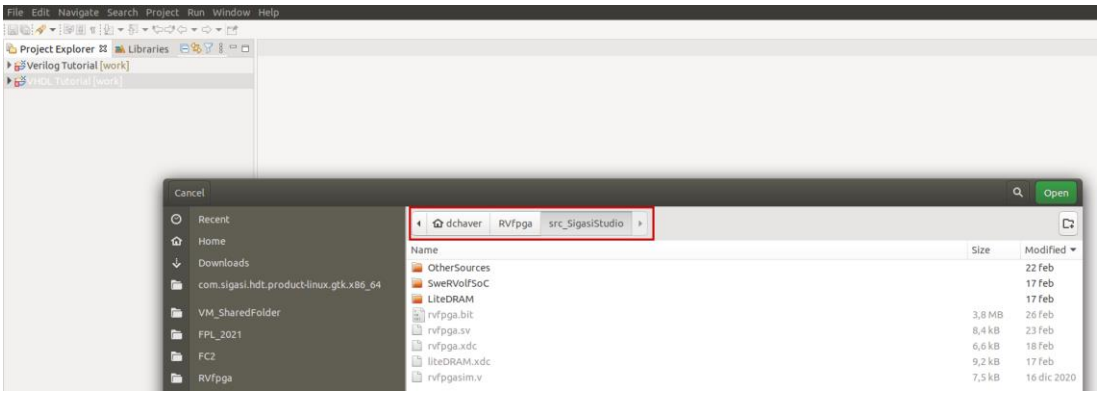

Figure 4. Open the RVfpga source directory

5. The project will open with many errors (see Figure 5), most of them due to the lack of many include files in the project configuration.

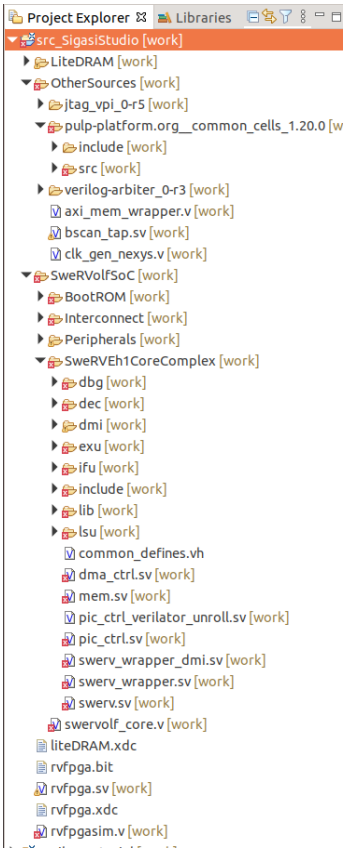

Figure 5. Initial errors in the RVfpga Sigasi Studio Project.

6. In the Project Explorer, right-click on the *src_SigasiStudio* project and open the Properties window (see Figure 6).

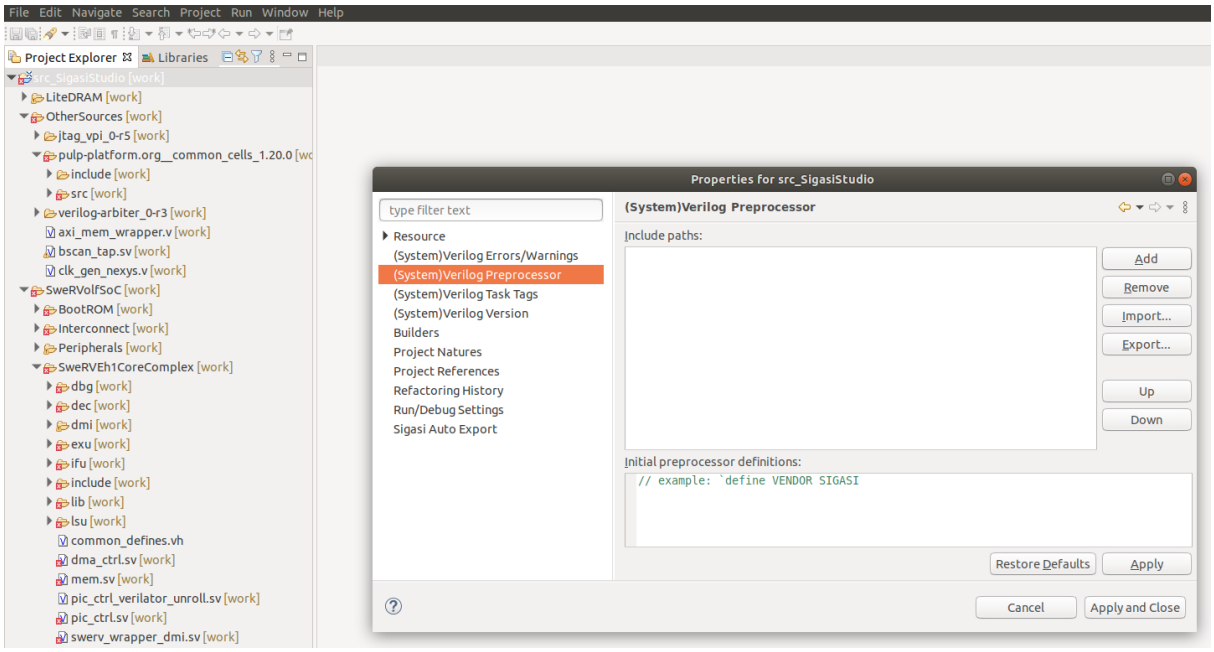

Figure 6. Project properties.

7. In the Properties window (Figure 6) select the “(System)Verilog Preprocessor” and add the following include paths (by clicking on the Add button on the right):
    - *[RVfpgaPath]/RVfpga/src_SigasiStudio/SweRVolfSoC/SweRVEh1CoreComplex/include*
    - *[RVfpgaPath]/RVfpga/src_SigasiStudio/OtherSources/pulp-platform.org__common_cells_1.20.0/include*
    - *[RVfpgaPath]/RVfpga/src_SigasiStudio/SweRVolfSoC/Interconnect/AxiInterconnect/pulp-platform.org__axi_0.25.0/include*
    - *[RVfpgaPath]/RVfpga/src_SigasiStudio/SweRVolfSoC/Interconnect/AxiInterconnect*
    - *[RVfpgaPath]/RVfpga/src_SigasiStudio/SweRVolfSoC/Interconnect/WishboneInterconnect*

    Once the five directories have been added, click on the Apply button.

    Then, in the same window, on the bottom box (Initial preprocessor definitions), enter the following line: `` `include "common_defines.vh" ``. Click on the Apply and Close button.

    Figure 7 shows the final state.

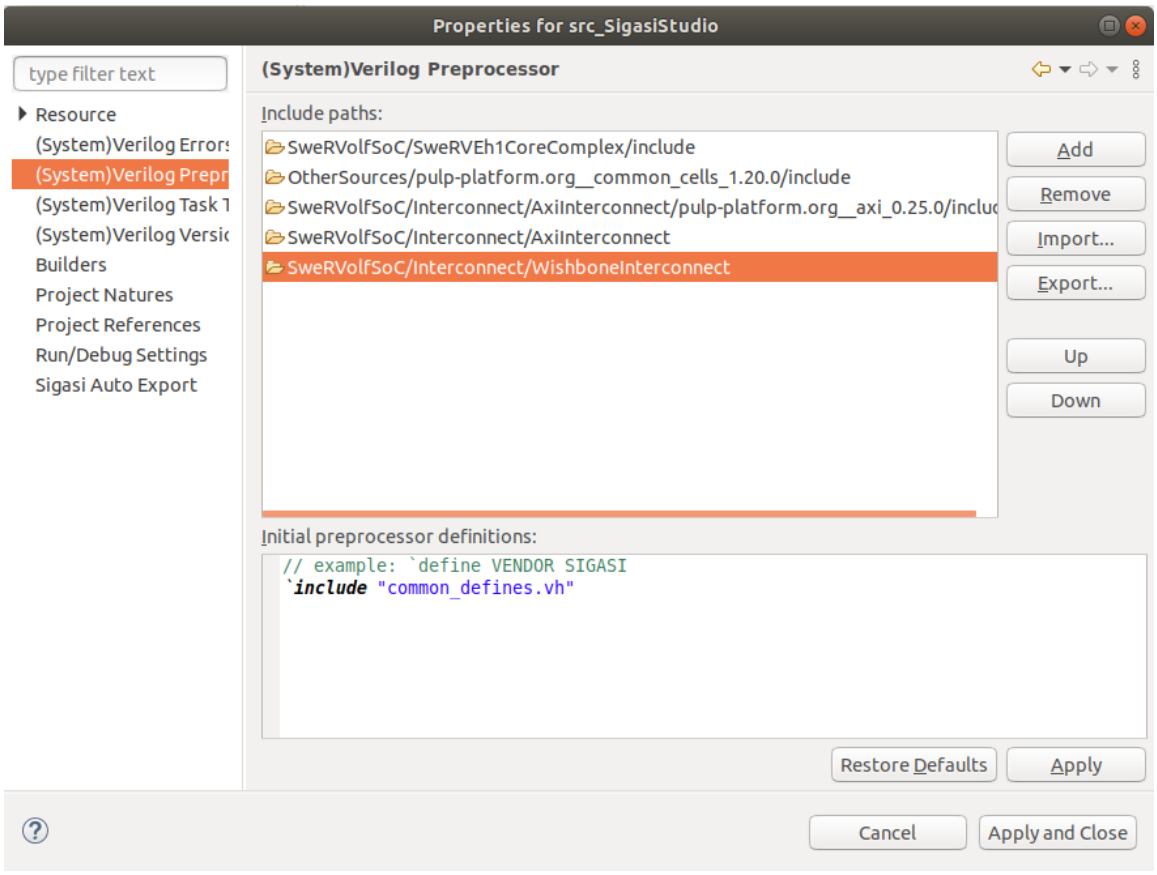

Figure 7. Include directories and files

8. Finally, delete file *[RVfpgaPath]/RVfpga/src_SigasiStudio/SweRVolfSoC/BootROM/sw/boot_main.vh*, which we do not need for our project and gives some errors. You can either delete it in your File Explorer or inside Sigasi Studio.

All the errors should have disappeared after these steps and only some warnings should remain, that you can ignore.

You can start using Sigasi Studio for inspecting the RVfpga SoC. As a test, we next show some functionalities of the tool:

1. On the top menu, open Window → Show View → Block Diagram, which opens a new window on the right part of the tool that lets you navigate graphically through the module.

2. In this lab we analyse arithmetic and logical instructions. These instructions are executed in the ALU, which is implemented inside module **exu_alu_ctl**. Open that module by double-clicking on it on the Project Explore window. You should see what we show in Figure 8.

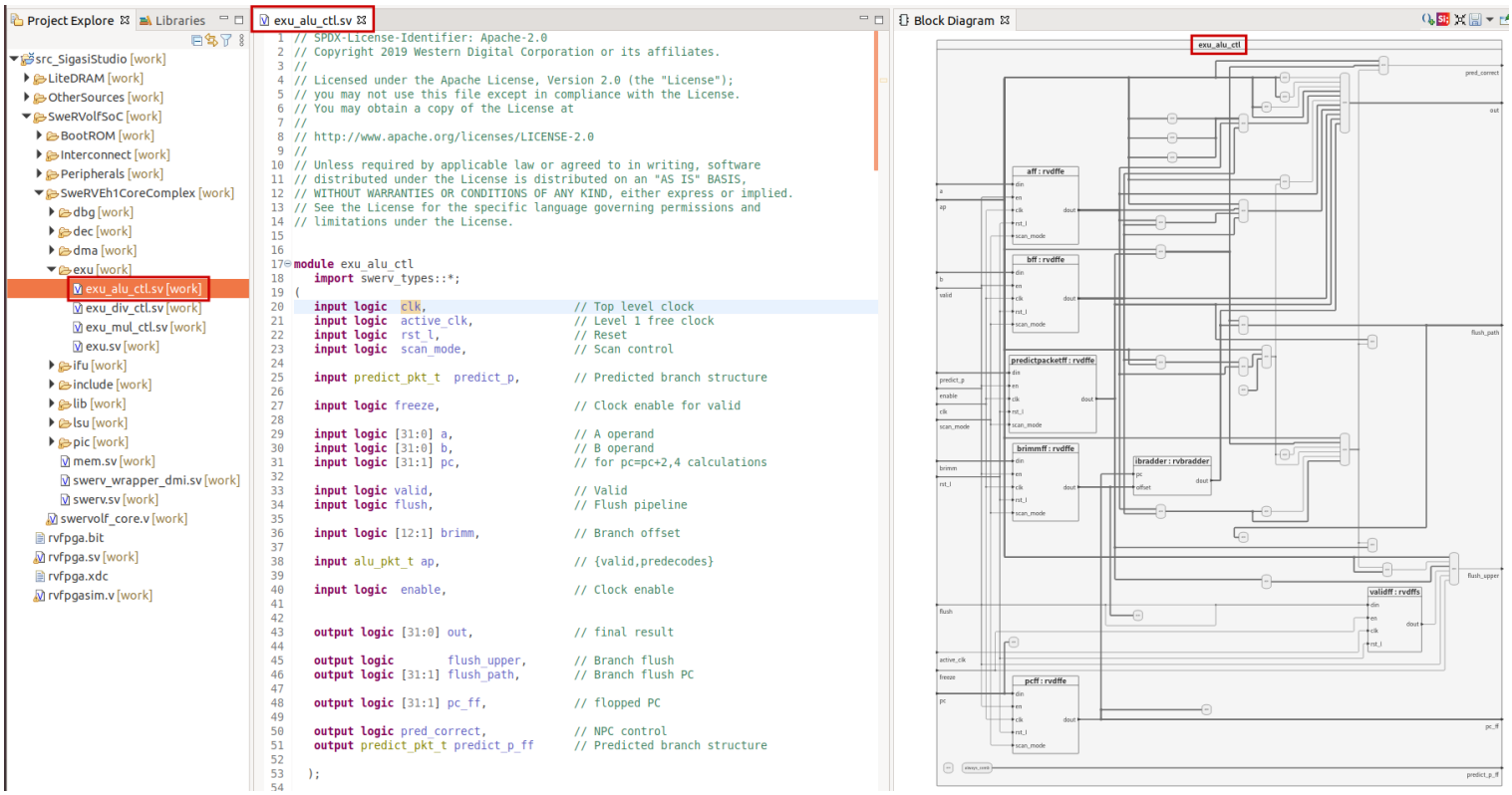

Figure 8. File *exu_alu_ctl.sv*: Verilog Code and Block Diagram

3. You can highlight a signal on the diagram by right-clicking on it in the Verilog code and selecting Show In → Block Diagram. The wires associated with the signal will highlight on the Block Diagram window, as shown in Figure 9, where the `ap` packet is highlighted.

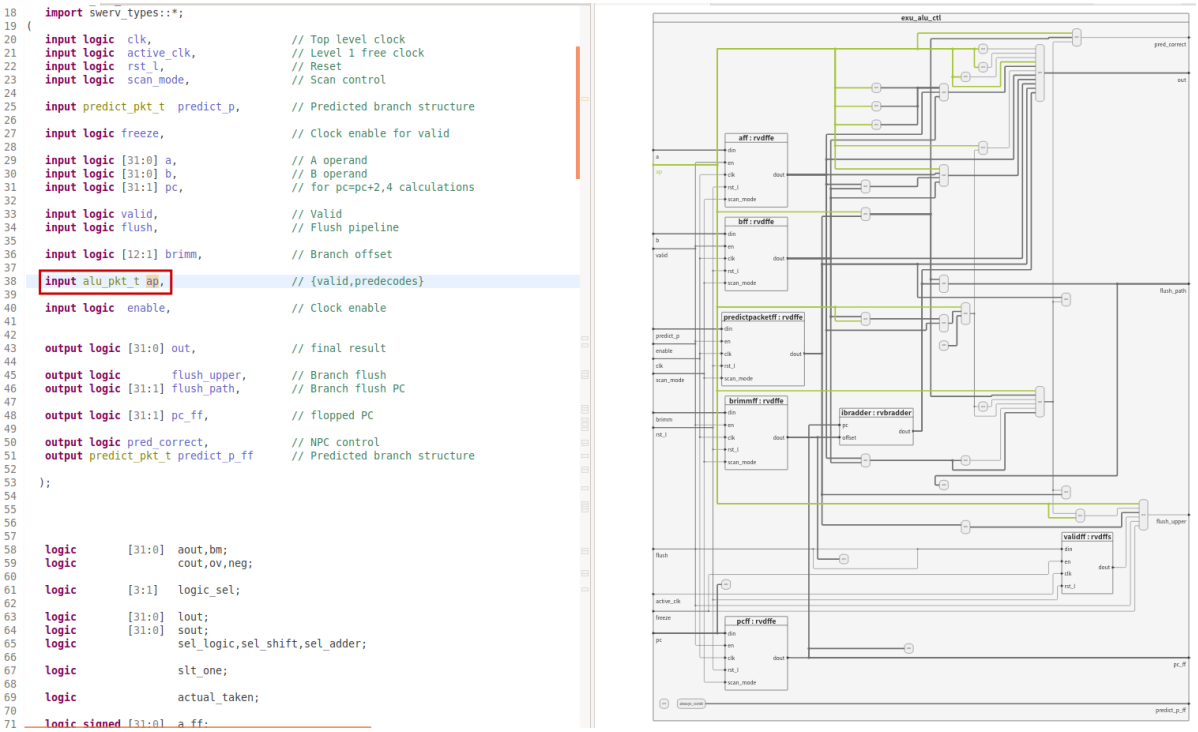

Figure 9. Highlight signal `ap`

4. You can also look for the implementation of a combinational module in the Verilog code, by double-clicking on the module in the Block Diagram. For example, in Figure 10, the module that generates signal `out` is shown.

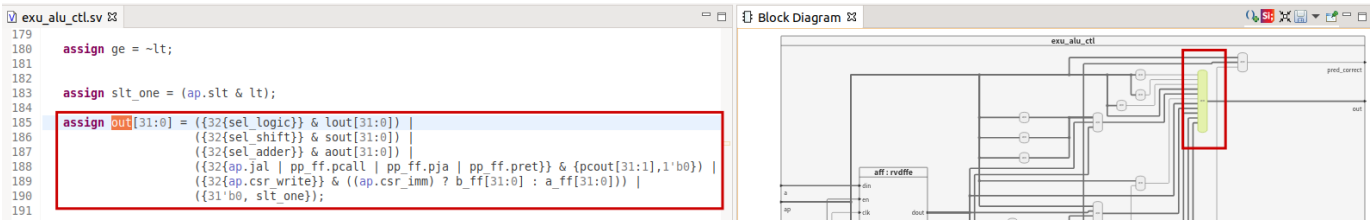

Figure 10. Highlight the Verilog code for the combinational module that generates signal `out`

5. Finally, we open a module declaration on the Block Diagram by right-clicking on the module instantiation in the Verilog code and selecting Open Declaration. Figure 11 shows module **rvdffe**, implemented in file *beh_lib.sv*.

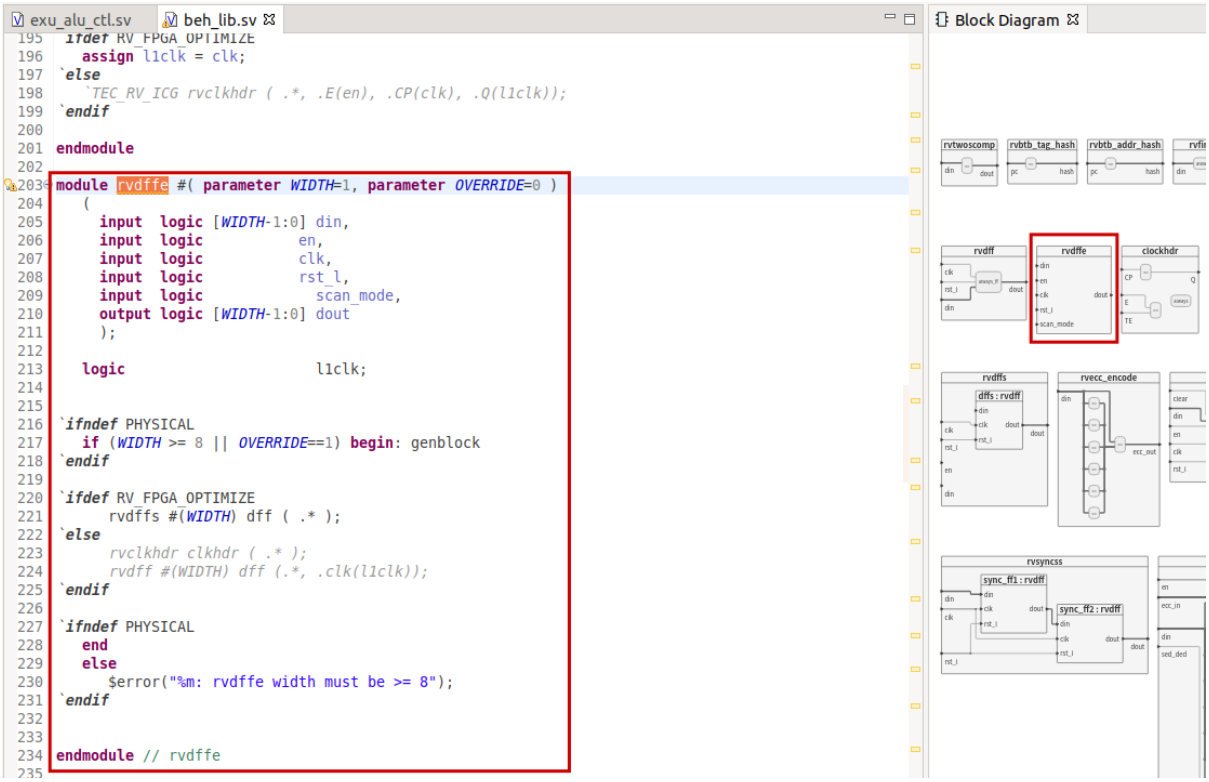

Figure 11. Module rvdffe

2. CONFIGURATION OF THE SWERV EH1 PROCESSOR

A. Configure the Core Structures

[RVfpgaPath]/RVfpga/src/SweRVolfSoC/SweRVEh1CoreComplex/include/common_defines.vh permits the user to configure many structures of the core, such as the Instruction Cache, the ICCM/DCCM, the Branch Predictor, etc. A default configuration is provided in the RVfpga System, which you can change in two different ways:

- You can manually edit the parameters in file *common_defines.vh*.

- You can use the *swerv.config* script provided by Western Digital with the SweRV EH1 package. The use of this script is described at https://github.com/chipsalliance/Cores-SweRV/tree/branch1.8
  In RVfpga you can find the *swerv.config* script at:
  [RVfpgaPath]/RVfpga/src/SweRVolfSoC/SweRVEh1CoreComplex/include/

Once you have generated the new configuration files, you can resynthesize the SoC in Vivado as explained in Lab 1 and obtain the new RVfpga System bitstream.

B. Disable the use of Compressed Instructions

In some cases, we may be interested in disabling the use of compressed instructions. For that purpose, we must make two changes to our PlatformIO project:

- Include the following new lines in file *platformio.ini*:

```
build_unflags = -Wa,-march=rv32imac -march=rv32imac
build_flags = -Wa,-march=rv32ima -march=rv32ima
extra_scripts = extra_script.py
```

- Add file *extra_script.py* to the sources of the project. This file contains the following lines:

```
Import("env")
env.Append(
  LINKFLAGS=[
      "-Wa,-march=rv32ima",
      "-march=rv32ima"
  ]
)
```

In most of the examples used in Labs 11-20 we will disable the use of compressed instructions for the sake of simplicity.

C. Enable/Disable Core Features

Table 10-1 of the SweRV EH1 Programmer's Reference Manual (https://github.com/chipsalliance/Cores-SweRV/blob/master/docs/RISC-V_SweRV_EH1_PRM.pdf) shows the *mfdc* register (at *CSR* 0x7F9) bits. This register hosts low-level core control bits to disable specific features, such as pipelined or dual-issue execution, the Branch Predictor, etc. Table 1 shows the nine core features that can be controlled by this register. Setting the proper bits of the register to 0 or 1, enables or disables each core feature. For example, you can include the following two assembly instructions in

your assembly program for disabling the dual-issue execution, the secondary ALU and the pipelined execution:

```
li t2, 0x481
csrrs t1, 0x7F9, t2
```

Table 1. Feature Disable Control Register (*mfdc*: CSR 0x7F9)

31-11	Reserved	7	0: enable secondary ALU 1: disable secondary ALU	3	0: enable branch prediction and return address stack 1: disable branch prediction and return address stack
10	0: dual issue execution 1: single issue execution	6	0: side effect stores are pipelined 1: side effect stores block all subsequent bus transactions until store response with default value received	2	0: enable Write Buffer coalescing 1: disable Write Buffer coalescing
9	Reserved	5	0: enable non-blocking loads/divides 1: disable non-blocking loads/divides	1	Reserved
8	0: ICCM/DCCM ECC checking enabled 1: ICCM/DCCM ECC checking disabled	4	0: enable fast divide 1: disable fast divide	0	0: pipelined execution 1: single instruction execution

We will use different configurations in Labs 11-20 in order to compare the performance, I$ hits/misses, Branch Predictor hits/misses, etc., of SweRV EH1 when the different core features are enabled/disabled.

3. MAIN MODULES AND SIGNALS OF THE SweRV EH1 CORE

The RVfpga System runs on the Artix-7 FPGA located on the Nexys A7 board, as shown in Figure 12. The figure details the system's hierarchy, including the names of the Verilog modules and submodules. The RVfpga System consists of the SweRVolf core (***swervolf_core***), the DRAM controller (***litedram_top***), the clock generation module (***clk_gen_nexys***), and some interface modules. The SweRVolf core, in turn, consists of the SweRV EH1 processor (***swerv_wrapper_dmi***) and additional interface modules (***wb_intercon***, ***axi_intercon***, ***uart_top***, etc.). The top module for the SweRV EH1 processor, ***swerv_wrapper_dmi***, instantiates the two main modules of the core: ***mem*** and ***swerv***. In the remainder of this document, we list the submodules and main signals of these two modules. Note that you can find the remaining signals of each module at the interface of the module. In Labs 11-20 we study these signals when analysing the operation of the different processor parts.

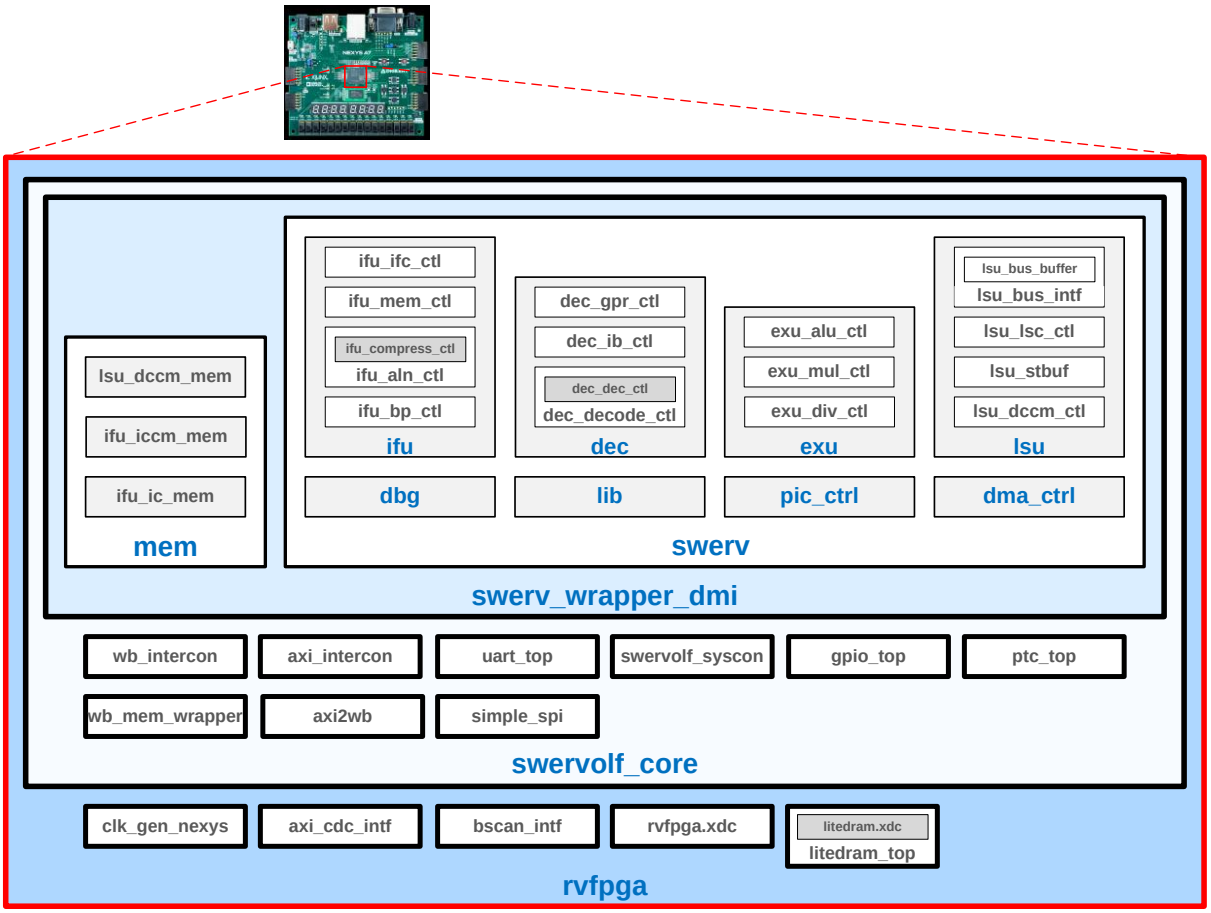

Figure 12. Hierarchy of the RVfpga System

MODULE: *mem*

FUNCTION: This module instantiates the three internal memories available in SweRV: ICCM, DCCM, and I$. Table 2 lists *mem*'s submodules and their interface signals.

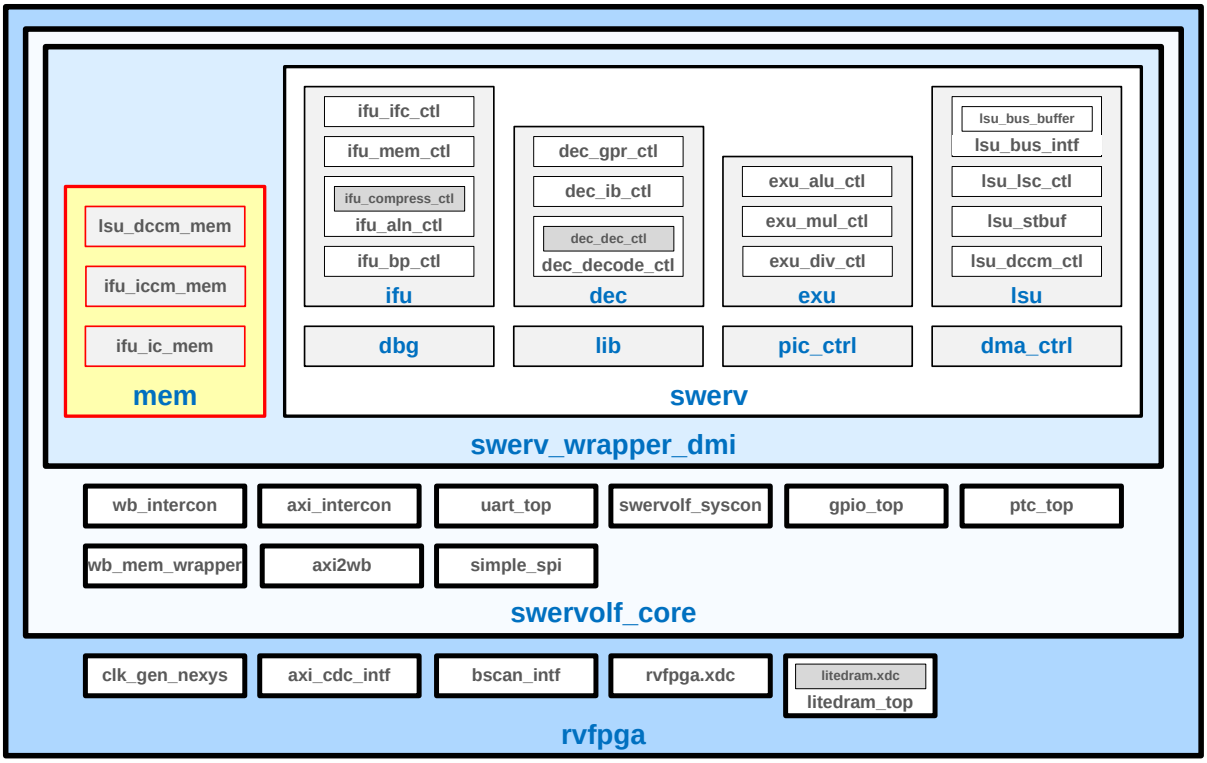

Figure 13. Module *mem* and its submodules

Table 2. *mem* submodules and I/O

Unit	I/O	Name	Description
ICCM: ***ifu_iccm_mem*** (It contains the ICCM module wrapper)	Input	`iccm_wren`	Write enable
		`iccm_rden`	Read enable
		`[`RV_ICCM_BITS-1:2] iccm_rw_addr`	Read/Write Address
		`[77:0] iccm_wr_data`	Write data
	Output	`[155:0] iccm_rd_data`	Read data
I$: ***ifu_ic_mem*** (It contains the Instruction Cache Data & Tag module wrapper)	Input	`[3:0] ic_wr_en`	Write enable
		`ic_rd_en`	Read enable
		`[31:2] ic_rw_addr`	Read/Write Address
		`[67:0] ic_wr_data`	Data to fill to the Icache. With Parity.
	Output	`[135:0] ic_rd_data`	Data read from Icache. F2 stage. With Parity.
		`[3:0] ic_rd_hit`	Hit/Miss in each way
DCCM: ***lsu_dccm_mem*** (It contains the DCCM module wrapper)	Input	`dccm_wren`	Write enable
		`dccm_rden`	Read enable
		`[`RV_DCCM_BITS-1:0] dccm_wr_addr`	Write address
		`[`RV_DCCM_BITS-1:0] dccm_rd_addr_lo`	Read address
		`[`RV_DCCM_BITS-1:0] dccm_rd_addr_hi`	Read address for the upper (high) bank when misaligned access

		`[`RV_DCCM_FDATA_WIDTH-1:0] dccm_wr_data`	Write data
	Output	`[`RV_DCCM_FDATA_WIDTH-1:0] dccm_rd_data_lo`	Read data low bank
		`[`RV_DCCM_FDATA_WIDTH-1:0] dccm_rd_data_hi`	Read data high bank

MODULE: *swerv*

FUNCTION: As shown in Figure 14, *swerv* is the top level module for the SweRV EH1 core. It instantiates the main modules of the core, most importantly: *ifu*, *dec*, *exu* and *lsu*. Table 3 – Table 6 list each of these unit's submodules and interface signals. The *swerv* module communicates with the *mem* module via the SweRV wrapper (*swerv_wrapper_dmi*).

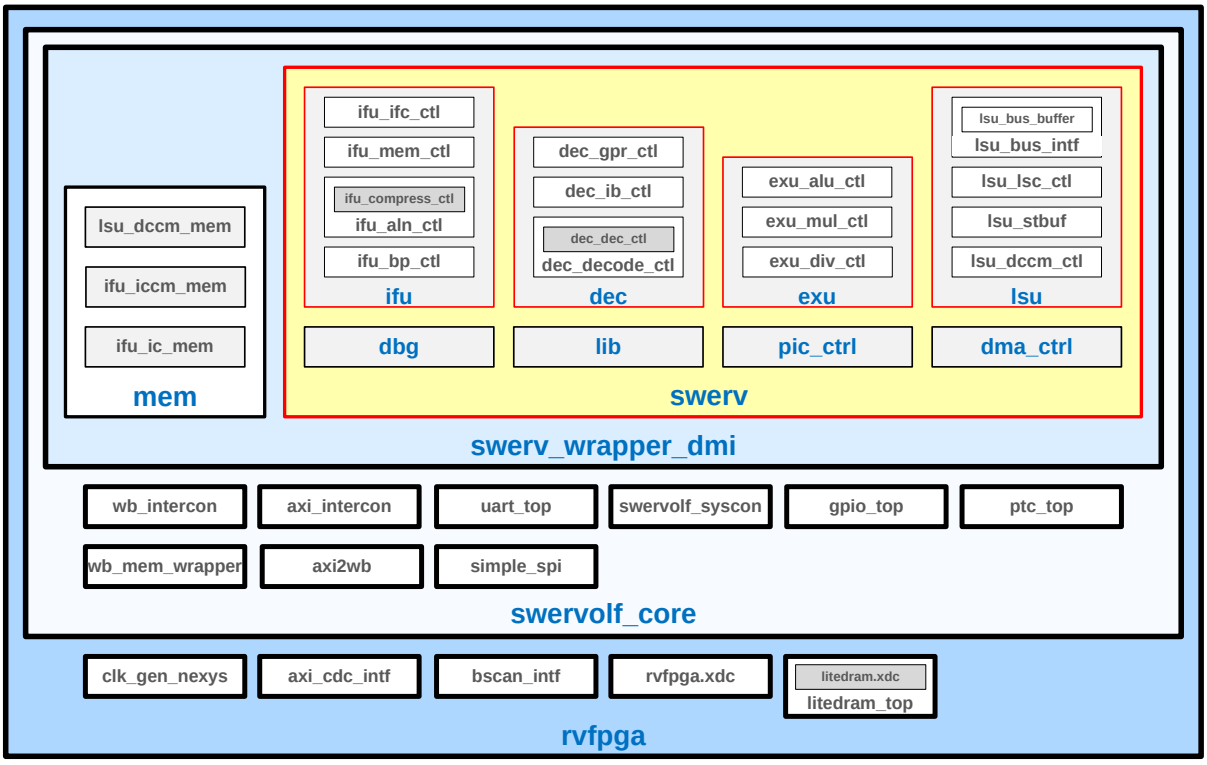

Figure 14. *swerv* and its submodules

Table 3. *ifu* (Instruction Fetch Unit) I/O and submodules (including their I/O)

Unit	I/O	Name	Description
Instruction Fetch Unit: ***ifu*** (This is the top level module for the Fetch of the	Input/Output	Several signals	ICCM ports to/from mem module
		Several signals	I$ ports to/from mem module
		Several signals	IFU AXI ports
	Input	`exu_flush_final`	Flush the pipeline
		`[31:1] exu_flush_path_final`	Flush fetch address

Instructions, the Prediction from the Branch Predictor and the Aligner)	Output	`[31:0] ifu_i0_instr`	Instruction 0. From Align to Decode
		`[31:0] ifu_i1_instr`	Instruction 1. From Align to Decode
		`[31:1] ifu_i0_pc`	Instruction 0 PC (program counter). From Align to Decode
		`[31:1] ifu_i1_pc`	Instruction 1 PC. From Align to Decode
Fetch Control: ***ifu_ifc_ctl*** (This module implements the Fetch Pipe Control. It generates the next address to fetch from the Instruction Memory.)	Input	`exu_flush_final`	Flush the pipeline
		`[31:1] ifu_bp_btb_target_f2`	Predicted target PC
		`[31:1] exu_flush_path_final`	Flush path
	Output	`output logic [31:1] ifc_fetch_addr_f1`	Fetch address at FC1
	Internal	`logic [31:1] fetch_addr_next`	Sequential address
Instruction Memory (I$ and ICCM) Control: ***ifu_mem_ctl*** (Instruction Memory Control – Icache and ICCM –)	Input	`[31:1] fetch_addr_f1`	Fetch addr at FC1 (`ifc_fetch_addr_f1` renamed)
	Output	`[127:0] ic_data_f2`	Data read at FC2 from I$ or ICCM to Align stage
Align Control: ***ifu_aln_ctl*** (Instruction Aligner)	Input	`[127:0] ifu_fetch_data`	128-bit fetch data from Fetch Stage
	Internal	`logic [127:0] q2,q1,q0`	3 Buffers
	Output	`[31:0] ifu_i0_instr`	Instruction Way 0
		`[31:0] ifu_i1_instr`	Instruction Way 1
		`[31:1] ifu_i0_pc`	Instruction Way 0 PC
		`[31:1] ifu_i1_pc`	Instruction Way 1 PC
Branch Predictor: ***ifu_bp_ctl***	Input	`[31:1] ifc_fetch_addr_f1`	Fetch address at FC1
	Output	`[31:1] ifu_bp_btb_target_f2`	Predicted target PC
		`ifu_bp_kill_next_f2`	Taken/Non-Taken branch

Table 4. *dec* (Decode Unit) I/O and submodules (including their I/O)

Unit	I/O	Name	Description

Decode Unit: ***dec*** (This is the top level module for the Decoding of the Instructions, the Dependency Scoreboard and the access to the Register File)	Input	`exu_flush_final`	flush the pipeline when 1
		`[31:0] ifu_i0_instr,` `[31:1] ifu_i1_instr`	Instructions from Align
		`[31:1] ifu_i0_pc` `[31:1] ifu_i1_pc`	PCs from Align
	Output	`alu_pkt_t i0_ap` `alu_pkt_t i1_ap`	ALU control signals
		`lsu_pkt_t lsu_p`	LSU control signals
		`mul_pkt_t mul_p`	MUL control signals
		`div_pkt_t div_p`	DIV control signals

		`predict_pkt_t`
		`i0_predict_p_d`
		`i1_predict_p_d`
			prediction signals to ALUs

		`[31:1] dec_i0_pc_d` `[31:1] dec_i1_pc_d`	Address of instructions at decode Stage

		`[31:0] gpr_i0_rs1_d`
		`[31:0] gpr_i0_rs2_d`
		`[31:0] gpr_i1_rs1_d`
		`[31:0] gpr_i1_rs2_d`
			I0/I1 rs1/rs2 data from register file

		`[31:0] dec_i0_immed_d` `[31:0] dec_i1_immed_d`	Immediate value
		`[12:1] dec_i0_br_immed_d` `[12:1] dec_i1_br_immed_d`	Branch offset

		`[31:0]`
		`i0_rs1_bypass_data_d`
		`[31:0]`
		`i0_rs2_bypass_data_d`
		`[31:0]`
		`i0_rs1_bypass_data_e2`
		`[31:0]`
		`i0_rs2_bypass_data_e2`
		`[31:0]`
		`i0_rs1_bypass_data_e3`
		`[31:0]`
		`i0_rs2_bypass_data_e3`
			I0 rs1/rs2 bypass data

		`[31:0]`
		`i1_rs1_bypass_data_d`
		`[31:0]`
		`i1_rs2_bypass_data_d`
		`[31:0]`
		`i1_rs1_bypass_data_e2`
		`[31:0]`
		`i1_rs2_bypass_data_e2`
		`[31:0]`
		`i1_rs1_bypass_data_e3`
		`[31:0]`
		`i1_rs2_bypass_data_e3`
			I1 rs1/rs2 bypass data

	Internal	`[31:0] dec_i0_instr_d` `[31:0] dec_i1_instr_d`	Instructions in Decode stage

		[31:0] dec_i0_rs1_d [31:0] dec_i0_rs2_d [31:0] dec_i1_rs1_d [31:0] dec_i1_rs2_d	rs1/rs2 data
Instructions/PC to send from Align to Decode: ***dec_ib_ctl*** (Buffers for propagating the instructions and PCs from the Aligner to the Decoder)	Input	[31:0] ifu_i0_instr [31:0] ifu_i1_instr	I0/I1 instruction from Align
		[31:1] ifu_i0_pc [31:1] ifu_i1_pc	I0/I1 PC from Align
	Output	[31:0] dec_i0_instr_d [31:0] dec_i1_instr_d	I0/I1 instruction at Decode
		[31:1] dec_i0_pc_d [31:1] dec_i1_pc_d	I0/I1 PC at Decode
Decode instruction and compute bypass values: ***dec_decode_ctl*** (Decode the 2 instructions and compute the bypass values)	Input	[31:1] dec_i0_pc_d [31:1] dec_i1_pc_d [31:0] exu_i0_result_e1	I0/I1 PC
		[31:0] dec_i0_instr_d, [31:0] dec_i1_instr_d	instruction in Decode stage
	Output	alu_pkt_t i0_a alu_pkt_t i1_ap	ALU control signals
		lsu_pkt_t lsu_p	LSU control signals
		mul_pkt_t mul_p	MUL control signals
		div_pkt_t div_p	DIV control signals
		predict_pkt_t i0_predict_p_d i1_predict_p_d	prediction signals to ALU
		[4:0] dec_i0_rs1_d [4:0] dec_i0_rs2_d [4:0] dec_i1_rs1_d [4:0] dec_i1_rs2_d	I0/I1 rs1/rs2 index
		[31:0] dec_i0_immed_d [31:0] dec_i1_immed_d	Immediate value
		[12:1] dec_i0_br_immed_d [12:1] dec_i1_br_immed_d	Branch offset
		[31:0] i0_rs1_bypass_data_d [31:0] i0_rs2_bypass_data_d [31:0] i0_rs1_bypass_data_e2 [31:0] i0_rs2_bypass_data_e2 [31:0] i0_rs1_bypass_data_e3 [31:0] i0_rs2_bypass_data_e3	I0 rs1/rs2 bypass data
		[31:0] i1_rs1_bypass_data_d [31:0] i1_rs2_bypass_data_d [31:0] i1_rs1_bypass_data_e2	I1 rs1/rs2 bypass data

		[31:0] i1_rs2_bypass_data_e2 [31:0] i1_rs1_bypass_data_e3 [31:0] i1_rs2_bypass_data_e3	
Register File: ***dec_gpr_ctl*** (Register File)	Input	[4:0] raddr0, raddr1 [4:0] raddr2, raddr3	Read addresses
		[4:0] waddr0, waddr1 [4:0] waddr2	Write addresses
		[31:0] wd0, wd1, wd2	Write data
		rden0, rden1, rden2, rden3	Read enable
		wen0, wen1, wen2	Write enable
	Output	[31:0] rd0, rd1, rd2, rd3	Read data

Table 5. *exu* (Execute Unit) I/O and submodules (including their I/O)

Unit	I/O	Name	Description
Execute Unit: ***exu*** (This is the top level module for the Execution of the A-L Instructions)	Input	alu_pkt_t i0_ap, alu_pkt_t i1_ap	ALU control
		mul_pkt_t mul_p	MUL control
		div_pkt_t div_p	DIV control
		[31:1] dec_i0_pc_d, dec_i1_pc_d	PCs from Decode
		[31:0] gpr_i0_rs1_d [31:0] gpr_i0_rs2_d [31:0] gpr_i1_rs1_d [31:0] gpr_i1_rs2_d	I0/I1 rs1/rs2
		[31:0] dec_i0_immed_d [31:0] dec_i1_immed_d	Immediate values
		[12:1] dec_i0_br_immed_d [12:1] dec_i1_br_immed_d	Branch offsets
		[31:0] i0_rs1_bypass_data_d [31:0] i0_rs2_bypass_data_d [31:0] i0_rs1_bypass_data_e2 [31:0] i0_rs2_bypass_data_e2 [31:0] i0_rs1_bypass_data_e3 [31:0] i0_rs2_bypass_data_e3	I0 rs1/rs2 bypass data
		[31:0] i1_rs1_bypass_data_d [31:0] i1_rs2_bypass_data_d [31:0] i1_rs1_bypass_data_e2 [31:0] i1_rs2_bypass_data_e2 [31:0] i1_rs1_bypass_data_e3 [31:0] i1_rs2_bypass_data_e3	I1 rs1/rs2 bypass data
	Output	exu_flush_final	flush pipeline when 1
		[31:0] exu_i0_result_e1 [31:0] exu_i1_result_e1	primary ALU result
		[31:0] exu_i0_result_e4 [31:0] exu_i1_result_e4	secondary ALU result
		[31:0] exu_mul_result_e3	MUL result
		[31:0] exu_div_result	DIV result
		[31:0] exu_lsu_rs1_d	Load/Store address

		[31:0] exu_lsu_rs2_d	store data
ALU: ***exu_alu_ctl*** (Arithmetic Logic Unit)	Input	[31:0] a	A operand
		[31:0] b	B operand
		[31:1] pc	for pcnext calculations (i.e., pc+2 or pc+4)
		[12:1] brimm	branch offset
		alu_pkt_t ap	ALU control
	Output	[31:0] out	ALU result
		flush_upper	branch flush
		[31:1] flush_path	Target PC
		[31:1] pc_ff	
Multiplier: ***exu_mul_ctl***	Input	[31:0] a	A operand
		[31:0] b	B operand
		mul_pkt_t mp	MUL control
	Output	[31:0] out	MUL result
Divider: ***exu_div_ctl***	Input	[31:0] dividend	numerator
		[31:0] divisor	denominator
		div_pkt_t dp	DIV control
	Output	[31:0] out	DIV result

Table 6. *lsu* (Load/Store Unit) I/O and submodules (including their I/O)

Unit	I/O	Name	Description
Load/Store Unit: ***lsu*** (This is the top level module for the Load/Store Unit of the Instructions)	Input/Output	Several signals	DCCM ports to/from mem module
		Several signals	DMA slave
		Several signals	LSU AXI ports
	Input	[31:0] exu_lsu_rs1_d	Load/Store address
		[31:0] exu_lsu_rs2_d	store data
		[11:0] dec_lsu_offset_d	address offset
		lsu_pkt_t lsu_p	LSU control
	Output	[31:0] lsu_result_dc3	LSU read data
Address computation: ***lsu_lsc_ctl*** (LSU Control and compute Load/Store Address)	Input	[31:0] exu_lsu_rs1_d	Load/Store address
		[31:0] exu_lsu_rs2_d	store data
		[11:0] dec_lsu_offset_d	address offset
		lsu_pkt_t lsu_p	LSU control
	Output	[31:0] lsu_addr_dc1 [31:0] end_addr_dc1	Initial/Final address
DCCM Control: ***lsu_dccm_ctl*** (DCCM Control)	Input	[`RV_DCCM_FDATA_WIDTH-1:0] dccm_rd_data_lo	read data (lo bank)
		[`RV_DCCM_FDATA_WIDTH-1:0] dccm_rd_data_hi	read data (hi bank)
	Output	dccm_wren	write enable
		dccm_rden	read enable
		[`RV_DCCM_BITS-1:0] dccm_wr_addr	write address

		`[`RV_DCCM_BITS-1:0] dccm_rd_addr_lo`	read address (lo)
		`[`RV_DCCM_BITS-1:0] dccm_rd_addr_hi`	read address (hi): needed for misaligned loads
		`[`RV_DCCM_FDATA_WIDTH-1:0] dccm_wr_data`	write data
Store Buffer: ***lsu_stbuf*** (Store Buffer)	Input	`lsu_addr_dc3`	address
		`[`RV_DCCM_DATA_WIDTH-1:0] store_ecc_datafn_hi_dc3`	write data (hi)
		`[`RV_DCCM_DATA_WIDTH-1:0] store_ecc_datafn_lo_dc3`	write data (lo)
	Output	`[`RV_LSU_SB_BITS-1:0] stbuf_addr_any`	store buffer address
		`[`RV_DCCM_DATA_WIDTH-1:0] stbuf_data_any`	store buffer data

4. STRUCTURES AND TYPES FOR GROUPING CONTROL BITS

Below is a summary of the main structure types defined in file *[RVfpgaPath]/RVfpga/src/SweRVolfSoC/SweRVEh1CoreComplex/include/swerv_types.sv* and used in the SweRV EH1 processor for grouping the control signals.

- **`dec_pkt_t`:** This is the main control structure type and it contains the processor main control signals, such as `alu` (1 if an arithmetic-logic instruction is executed, 0 otherwise), `load` (1 if a `load` instruction is executed, 0 otherwise), `legal` (1 if the instruction is legal, 0 if it is not), `rs1` (1 if the instruction obtains the first input operand from the Register File, 0 otherwise), `imm12` (1 if the instruction uses a 12-bit immediate as an input operand, 0 otherwise), etc.

  This structure type is used inside module **dec_decode_ctl** for generating many other control signals. Four signals of this type are declared (Way-0: `i0_dp_raw`, `i0_dp`. Way-1: `i1_dp_raw`, `i1_dp`) and are used for generating the control bits of other structures defined in file *swerv_types.sv*.

  These bits are assigned inside module **dec_dec_ctl**, a module that is automatically generated using open-source tools (*coredecode* and *espresso*) and that can be found at the end of file *[RVfpgaPath]/RVfpga/src/SweRVolfSoC/SweRVEh1CoreComplex/dec/dec_decode_ctl.sv*.

- **`alu_pkt_t`:** This structure type contains the control signals related with the ALU operation, such as `valid` (1 if an arithmetic-logic instruction is executed, 0 otherwise), `add` (1 if an `add` instruction is executed, 0 otherwise), `beq` (1 if a `beq` instruction is executed, 0 otherwise), etc. Two signals of this type, called `i0_ap` and `i1_ap`, are defined inside module **dec_decode_ctl**.

  These bits are assigned inside module **dec_decode_ctl** (implemented at: *[RVfpgaPath]/RVfpga/src/SweRVolfSoC/SweRVEh1CoreComplex/dec/dec_decode_ctl.sv*), based on the bits from structure `dec_pkt_t` (see lines 711-770 of **dec_decode_ctl**).

- **`reg_pkt_t`:** This structure type contains the identifiers of the two source registers (fields `rs1` and `rs2`) and the destination register (field `rd`). Two signals of this type, called `i0r` and `i1r`, are defined inside module **dec_decode_ctl**. These signals are assigned from the proper fields of the Instruction Register inside module **dec_decode_ctl** (see lines 1121-1127 of this module).

- **`dest_pkt_t`:** This structure type contains control bits used in the Write-Back stage, which we will analyse in a forthcoming section. A signal of this type, called `dd`, is defined inside module **dec_decode_ctl**.

- **`rets_pkt_t`, `br_pkt_t`, `br_tlu_pkt_t`, and `predict_pkt_t`:** These structure types are related with branch instructions and the Branch Predictor.

- **`lsu_pkt_t`:** This structure type contains the control signals related with the Load/Store Unit, such as `half` (1 if a half word is read/written, 0 otherwise), `load` (1 if a `load` instruction is executed, 0 otherwise), `valid` (1 if the instruction is valid, 0 otherwise), etc. One signal of this type, called `lsu_p`, is defined inside module **dec_decode_ctl**.

- **`mul_pkt_t`:** This structure type contains the control signals related with the Multiply Unit, such as `rs1_sign` and `rs2_sign` (that determine if the input operands are treated as signed or unsigned), `valid` (1 if the instruction is valid, 0 otherwise), etc. One signal of this type, called `mul_p`, is defined inside module **dec_decode_ctl**.

- **`div_pkt_t`:** This structure type contains the control signals related with the Divide Unit, such as `unsign` (1 if the operation is unsigned, 0 otherwise), `valid` (1 if the instruction is valid, 0 otherwise), etc. One signal of this type, called `div_p`, is defined inside module **dec_decode_ctl**.

TASK: Open file *[RVfpgaPath]/RVfpga/src/SweRVolfSoC/SweRVEh1CoreComplex/include/swerv_types.sv* and analyse it during the next descriptions of the structure types that group together the control bits.

TASK: Take a quick look at modules **dec_decode_ctl** and **dec_dec_ctl** to see how the fields of the control signals are assigned based on the 32 bits of the instruction. These two modules are very extensive and quite complex, so the idea is not to analyse them in detail. Moreover, see that module **dec_dec_ctl** is created automatically as explained in lines 2482-2495 of *dec_decode_ctl.sv*.

5. COMPRESSED INSTRUCTIONS

Even though in most of the experiments that we include in the labs we disable the use of compressed instructions for the sake of simplicity, in this section we describe and analyse RISC-V's compressed instruction extension (RVC) and the execution of compressed instructions in SweRV EH1. Obviously, you are free to enable the use of compressed instructions in the experiments and extend the analysis on your own.

NOTE: Before starting this lab, we recommend reading Section 6.6.5 of the book by S. Harris and D. Harris, "*Digital Design and Computer Architecture: RISC-V Edition*", Morgan Kaufmann [DDCARV]. Some of this section's contents are inspired by that book.

The RVC extension reduces the size of common integer and floating-point instructions to 16 bits by reducing the sizes of the control, immediate, and register fields and by taking advantage of redundant or implied registers. This reduced instruction size decreases cost, power, and required memory – all of which can be crucial for hand-held and mobile applications. Our assembly programs can use a mix of compressed and 32-bit instructions, given that SweRV EH1 includes the RVC.

In SweRV EH1 there is a module specifically devoted to uncompressing instructions: **ifu_compress_ctl**. This module receives a compressed 16-bit instruction and outputs the corresponding uncompressed 32-bit instruction. In Figure 15 we show the Align Stage with a bit more detail than in Lab 11 (we still leave some black boxes that you can analyse by yourself). Three **ifu_compress_ctl** modules are instantiated inside module **ifu_aln_ctl**, which receive a compressed instruction from signal `aligndata[63:0]` and return the corresponding uncompressed instruction in signals `uncompress0[31:0]`, `uncompress1[31:0]` and `uncompress2[31:0]`. If the instructions are already in their uncompressed format, they are directly provided from the `aligndata[63:0]` signal.

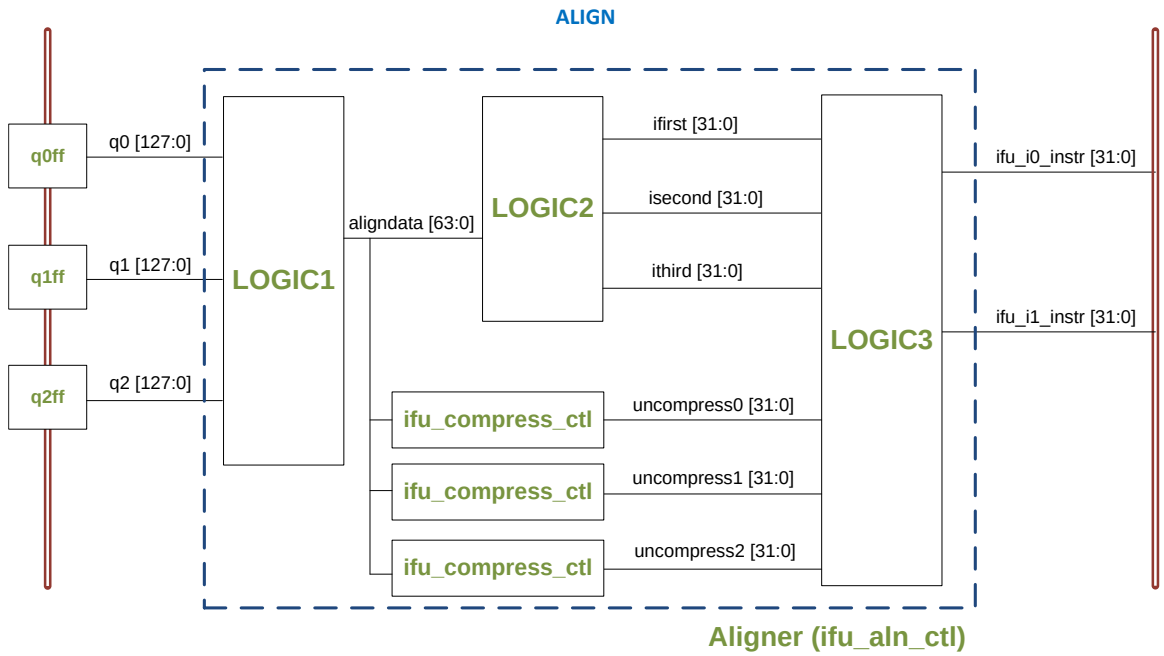

Figure 15. Align Stage

The code illustrated on the top part of Figure 16 shows the simple C program from Chapter 6 – Code Example 31 – DDCARV. The code illustrated on the bottom part of Figure 16 shows

the assembly code generated when the C program is compiled in PlatformIO with the RVC extension enabled (note that the assembly code is slightly different to the one shown in [DDCARV]). We highlight in red the instructions that make up the loop body, which are a combination of 16-bit and 32-bit instructions.

```
int scores[200];
int main(void) {
    int i;
    for (i = 0; i < 200; i = i + 1){
        scores[i] = scores[i] + 10;
    }
    return(0);
}
```

```
00000088 <main>:
  88:  6789                lui    a5,0x2
  8a:  12078793            addi   a5,a5,288 # 2120 <scores>
  8e:  32078693            addi   a3,a5,800
  92:  4398                lw     a4,0(a5)
  94:  0791                addi   a5,a5,4
  96:  0729                addi   a4,a4,10
  98:  fee7ae23            sw     a4,-4(a5)
  9c:  fed79be3            bne    a5,a3,92 <main+0xa>
  a0:  4501                li     a0,0
  a2:  8082                ret
```

Figure 16. Example of compressed instructions

Figure 17 shows the Verilator simulation of a whole iteration of the loop from Figure 16. Note that when instruction `addi a5,a5,4` is at the align stage (highlighted in red in the first cycle of the figure), the instruction is extracted from the 64-bit bundle (`aligndata[63:0]`) and decompressed from a 16-bit instruction (**0x0791**) into a 32-bit instruction (**0x00478793**). (The code is provided at *[RVfpgaPath]/RVfpga/Labs/Lab11/Compressed_C-Example* so that you can execute your own Verilator simulation.)

- In RISC-V, the opcode for the 16-bit `c.addi` instruction is (see Appendix B of [DDCARV]):

  `000 | imm(1-bit) | rd/rs1 | imm(5-bits) | 01`

  So you can easily verify that **0x0791** (0000011110010001) corresponds to: `c.addi a5,4` (remember that a5=x15).
  - Imm = 000100
  - rd = rs1 = 01111 (`x15`)

- In RISC-V, the opcode for the 32-bit `addi` instruction is (see Appendix B of [DDCARV]):

  `imm(12-bits) | rs1 | 000 | rd | 0010011`

  So you can easily verify that **0x00478793** (00000000010001111000011110010011) corresponds to: `addi a5,a5,4` (remember that a5 = x15).
  - Imm = 000000000100
  - rs1 = 01111 (`x15`)
  - rd = 01111 (`x15`)

In the second cycle shown in Figure 17, the `sw` instruction is aligned. Given that this instruction lacks the corresponding compressed version in RISC-V architecture, it needs no uncompressing and it is selected and propagated to the Decode Stage directly from the `aligndata[63:0]` signal.

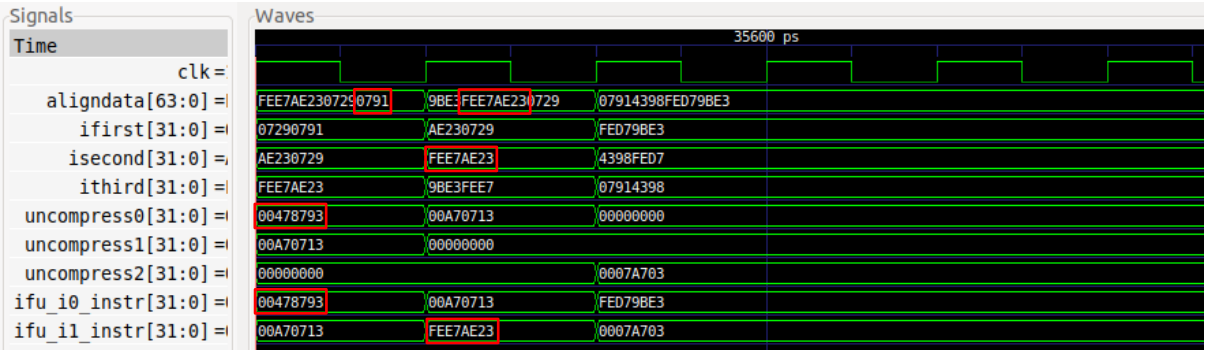

Figure 17. Simulation of the code shown in Figure 16

TASK: Analyse the remaining instructions from the loop body in terms of compressed/uncompressed instructions.

TASK: Take a look inside module **ifu_compress_ctl** and try to get an idea about how it works.

6. REAL BENCHMARKS

In folder *[RVfpgaPath]/RVfpga/Labs/Lab20/RealBenchmarks* we provide three real applications that you will use in Lab 20 for testing the different features of our SweRV EH1 processor. In that lab you can find further description about these three benchmarks and the different versions that we provide for each of them.

- **CoreMark**: In folder *[RVfpgaPath]/RVfpga/Labs/Lab20/RealBenchmarks/CoreMark_HwCounters* you can find a PlatformIO project that contains the CoreMark benchmark for running on RVfpgaNexys. We've used the sources provided at https://github.com/chipsalliance/Cores-SweRV and adapted them to our RVfpga System.

- **Dhrystone**: In folder *[RVfpgaPath]/RVfpga/Labs/Lab20/RealBenchmarks/Dhrystone_HwCounters* you can find a PlatformIO project that contains the Dhrystone benchmark for running on RVfpgaNexys. We've used the sources provided at https://github.com/chipsalliance/Cores-SweRV and adapted them to our RVfpga System.

- **Image Processing**: In folder *[RVfpgaPath]/RVfpga/Labs/Lab20/RealBenchmarks/ImageProcessing_HwCounters* you can find a PlatformIO project that contains the application that we used in Lab 5 for transforming an RGB image into grayscale.