EFCL Winter School on Open Source IC Design and ML Acceleration

Track 2 - FPGA implementations of RISC-V based micro-controllers with accelerators

Hands-on

Leveraging the open-source PULPissimo design

Accelerating FIR with Custom Hardware

Yvan Tortorella - *Fondazione CHIPS-IT (Bologna, IT)*
Riccardo Gandolfi - *Fondazione CHIPS-IT (Bologna, IT)*
Gabriele Lodi - *Fondazione CHIPS-IT (Bologna, IT)*
Bowen Wang - *PULP TEAM (ETH Zurich, CH)*
Hong Pang - *PULP TEAM (ETH Zurich, CH)*

ETH Zurich, Zurich, Switzerland
February 9th - 12th, 2026

1 Introduction

In this exercise, we will demonstrate the hardware acceleration capability offered by the PULPISSIMO platform. We will demonstrate it by using two dedicated Finite Impulse Response (FIR) acceleration strategies: an instruction set extension based on X-Interface (XIF), and a cooperative dedicated Hardware Processing Engine (HWPE), both integrated into the PULPissimo platform. This process involves several steps: integrating the FIR Intellectual Property (IP), simulating the design with Questasim-2022.3, implementing it on the Vitis-2022.1, generating the bitstream, and finally, verifying the output on ZyboZ7.

For this hands on session we will use only open-source IPs from the PULP Platform GitHub page.

2 Notation

Text-Based Command Inputs (Shell Commands): In general the exercises of this course assume a reasonable proficiency of the students in handling UNIX shells.

Throughout the exercises of this course, different types of *consoles* will be used.

```
cp /unspecified/shell.cmd .
echo "We make this command very long to illustrate a command spanning\
    multiple lines escaping the newline in the task description"
```

Figure 1: Console command representation used throughout this exercise.

Files and Directories: File paths and directories, stated within this exercise, are always given *relative* to the root path of the PULPissimo project, unless otherwise specified.

Student Task: Parts of the exercise where you are required to take action will be explained in a shaded box, just like this paragraph.

3 Acknowledgment

We thank Bowen Wang and Hong Pang for their valuable help in debugging these assignments!

4 Tasks

4.1 Introduction to PULPissimo

In the first hands-on session, we will set up our basic environment, run some initial tests, and do some initial hardware modifications in PULPissimo to learn about the design and how to easily modify it.

To get started, log into the workstation in the computer room with the password provided by the course assistants. We will use these computers, as they have commercial tools for simulation and debugging, as well as Field-Programmable Gate Array (FPGA) implementation, already pre-installed.

4.1.1 Getting started with PULPissimo

As discussed in the lecture, PULPissimo is a small microcontroller host system. While most of the Intellectual Propertys (IPs) inside the design are confined to dependencies, the PULPissimo top-level repository contains the wrapping part of the system, shown in Figure 2, and provides the infrastructure to collect the needed dependencies. Furthermore, it contains the necessary testbench infrastructure, shown in Figure 3, as well as FPGA setups for testing. Follow Student Task 1 to get started.

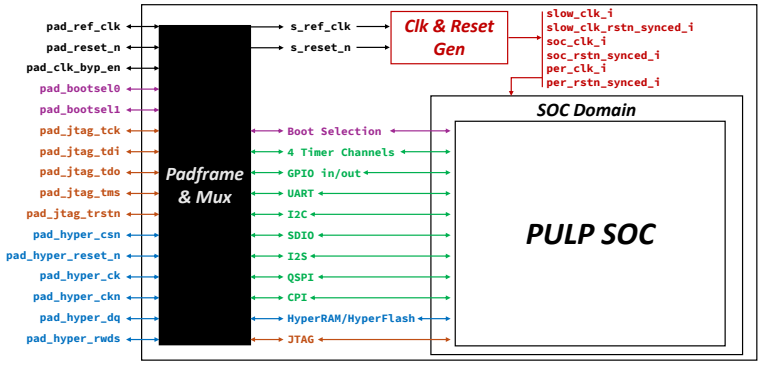

Figure 2: Block diagram of the PULPissimo architecture in the top-level repository.

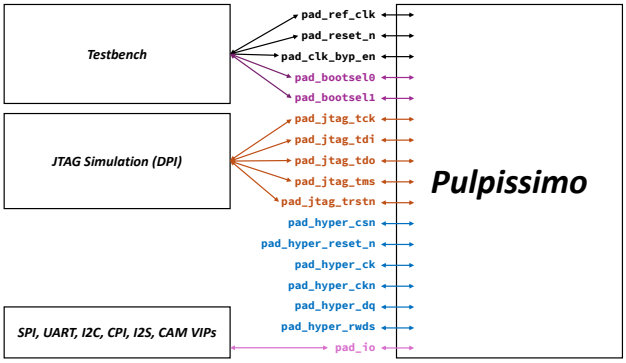

Figure 3: Block diagram of the PULPissimo testbench.

Student Task 1 (Clone PULPissimo):

- Once logged in to the Compute server, open a Terminal window (available applications for this are `Konsole`, `Terminal`, as well as a few others).
- Move to the `/scratch` partition, create a folder for you user (`mkdir <YOUR_USER_NAME\>`) and move there (`cd <YOUR_USER_NAME>`).
- Clone the PULPissimo repository into your home directory using the following command:

```
git clone https://github.com/FondazioneChipsIT/pulpissimo.git
```

- For this course, we have prepared a dedicated branch with some custom modifications to simplify the exercise. Please navigate to the pulpissimo directory and check out the correct branch using the following commands:

```
cd pulpissimo
git checkout EfclExercise
```

- By default, not all sources and software will be present yet in the repository. Please run the following commands to ensure submodule checkout:

```
git submodule update --init --recursive
make checkout
```

Then you need to create the python environment with all required packages.

- Step 1: Go to the directory "utils" and make install_conda.sh executable with:

```
cd utils
chmod +x install_conda.sh
```

- Step 2: Install conda at scratch space:

```
install_conda.sh /scratch/<YOUR_USER>
```

- Step 3: In order to automatically initialize conda for future shell sessions, type command:

```
echo '[[ -f /scratch/<YOUR_USER_NAME>/conda/bin/conda ]] && \
   eval "$(/scratch/<YOUR_USER>/conda/bin/conda shell.bash \
   hook)"' >> /home/<YOUR_USER>/.bashrc
```

  Then you need to open a new terminal for the .bashrc update to work.

- Step 4: Create conda environment (pick a name of your choice), and install numpy package by:

```
conda create -n <YOUR_ENV_NAME> python=3.13.11 numpy
```

- Step 5: Activate conda environment with:

```
conda activate <YOUR_ENV_NAME>
```

NOTE: you need to repeat this last command to activate the conda environment every time you open a new terminal.

The PULPissimo repository is organized in a rather complex structure, managed by a tool called `Bender`[1] that helps solve dependencies between the various IPs used to compose the system. After the checkout performed in the previous task, which hides several operations performed by Bender, the repository will contain the following hierarchy of folders (incomplete – some are less relevant for now):

```
pulpissimo
|-- .bender             # hidden folder containing all Git repos & \
   RTL code of dependencies managed by Bender.
|   |-- git
|       |-- checkouts   # checked out RTL code of dependencies
|       |-- db          # Git repos -- do not hamper with this!
|-- .github             # CI configuration
|   |-- workflows
|-- doc                 # documentation
|   |-- datasheet
|-- hw                  # PULPissimo top-level and system RTL
|   |-- includes
|   |-- padframe
|   |-- vendored_ips
|-- sw
|   |-- bootcode         # boot code example for ROM compilation
|   |-- pulp-runtime     # PULPissimo runtime environment
|   |-- regression_tests # set of tests designed for PULP systems
|   |-- efcl_ss          # exercises for the EFCL Summer School
|-- target
|   |-- fpga             # FPGA-related top-level and scripts
|   |-- lint
|   |-- sim              # QuestaSim-related top-level and scripts
|-- utils
    |-- bin
```

Now we will proceed with building the simulation platform, which relies on the Siemens QuestaSim Register Transfer Level (RTL) simulator, and run a simple *hello world* application.

Student Task 2 (Software application): Let's get an initial simulation running with a hello-world application. For this, we will need to set up some environment variables to get everything working properly.

- **build simulation**: from the `pulpissimo` root folder, build the simulation environment using

  ```
  make build
  ```

  You should see a long log of all the IP compilations, terminating with an output similar to this one:

  ```
  # Optimized design name is vopt_tb
  ```

[1] Available open-source on Github (`https://github.com/pulp-platform/bender`)

```
# End time: 00:11:18 on May 24,2024, Elapsed time: 0:00:02
# Errors: 0, Warnings: 1
#  quit
Finished building design 'tb_pulp'. The optimized design has been stored in a unit called ''vopt_tb''.

To run a simulation directly with the PULP runtime or SDK `make run` commands execute the following:

    export VSIM_PATH=/home/<YOUR_USER_NAME>/pulpissimo/build/questasim
```

- **setup simulation folder**: follow the prompt of the simulation build by exporting the environment variables. To fix the QuestaSim version, **it is recommended** to slightly modify them in this way:

```
export VSIM_PATH=$PWD/build/questasim
```

  The simulation environment is now correctly set up. Please also check that VSIM variable is empty.

- **software environment**: to complete the software environment of your workstation, you need to install PyElftools, by cloning it into your home directory:

```
cd /home/<YOUR_USER>
git clone https://github.com/eliben/pyelftools.git
```

  Then, in every new terminal you should set the following exports to access this library and the RISC-V C compiler we use (LLVM 18)

```
  export PYTHONPATH=/home/<YOUR_USER>/pyelftools:\$PYTHONPATH
  export PULP_RISCV_GCC_TOOLCHAIN=/usr/pack/riscv-1.0-kgf/\
     efclschool-llvm-18.1.6
```

  Note 1: You can add both commands at the end of `.bashrc` or in a stand-alone `sourceme.sh` file for convenience.

- **source software runtime environment**: back to the `pulpissimo` root folder, the software runtime environment, located in `sw/pulp-runtime`, contains all the hardware abstraction layer files and `Makefile` rules necessary to build a simple bare-metal PULPissimo application. To make it available, you must source a configuration file in every new terminal you open:

```
source sw/pulp-runtime/configs/pulpissimo_cv32e40x.sh
```

  This particular configuration is tuned for our work in the school as it builds code suited for the CV32E40X core, which without additional extensions is a RV32IMC Instruction Set Architecture (ISA) implementation.

  Note 2: You can add also this command at the end of `.bashrc` or in a stand-alone `sourceme.sh` file for convenience.

- **build hello-world**: you are now ready to simulate the “hello world” application. Navigate to `sw/efcl2026/task2_hello` and build the application:

```
cd sw/efcl2026/task2_hello
make all
```

Note 3: In the following tasks, when you build the software multiple times it is always safer to run

```
make clean all
```

to make sure you always have a clean database.

No error should arise. The outputs of the build will be found under the `build` folder.

Note 4: You can also inspect the application using an editor, e.g., Visual Studio Code:

```
code .
```

As you can observe, it is an extremely simple C program in the `test.c` source.

- **run hello-world in Command Line Interface (CLI)**: to run the application in the CLI, simply execute the following command

```
make run
```

The command will launch QuestaSim in CLI mode.

Note 5: If you get an error, you most likely did not export `VSIM` correctly and you are using the wrong version of QuestaSim. Check back!

If everything goes smoothly, you should see a long output ending with something similar to this:

```
# [STDOUT-CL31_PE0, 3851772ns] Hello !
# [TB  ] 3864001ns - retrying debug reg access
# [TB  ] 3908001ns - Waiting for end of computation
# [TB  ] 4002101ns - Received status core: 0x00000000
# ** Note: stop    : /home/<YOUR_USER_NAME>/pulpissimo/target/sim/tb/tb_pulp.sv(821)
#    Time: 4002101 ns  Iteration: 0  Instance: /tb_pulp
# Break in Module tb_pulp at /home/<YOUR_USER_NAME>/pulpissimo/target/sim/tb/tb_pulp.sv line 821
# Stopped at /home/<YOUR_USER_NAME>/pulpissimo/target/sim/tb/tb_pulp.sv line 821
# End time: 00:44:07 on May 24,2024, Elapsed time: 0:00:25
# Errors: 0, Warnings: 8
```

The testbench output `Received status core: 0x00000000` indicates that the testbench exited with the core reporting no error through its return code. Often, you will simply return the number of errors from your `int main()` function as a way to quickly signal that your execution is correct. Also, notice the standard output line a few lines earlier, signaled by `[STDOUT-CL31_PE0, <TIME>] \ Hello !`.

4.1.2 Debugging PULPissimo Simulations

Software traces & Disassembly: Traces and disassembly are fundamental instruments to understand your code. The simulation procedure illustrated above generates both; you can open them (e.g., with VSCode), in `build/trace_core.log` and `build/test/test.dis` respectively. It usually makes sense to open them in a split window so you can look at both at the same time. Traces contain the sequence of all instructions executed by the CV32E40X core, together with relevant register values and memory addresses. For example:

```
ns          | pc         | rs1 (   data   ) | rs2 (   data   ) | rd  (   data   ) | memaddr    | rdata      | wdata      ||  Assembly
=============================================================================================================================================
52220     ns | 0x1a000080 | x0  (0x00000000) | x0  (0x00000000) | x0  (0x00000000) | 0x1a000084 | 0x00000000 | 0x00000000 ||  NA
52391     ns | 0x1a001016 | x0  (0x00000000) | x0  (0x00000000) | x3  (0x1c000016) | 0x1c000016 | 0x00000000 | 0x00000000 ||  NA
```

To find more interesting information, skip directly to Program Counter (PC) `0x1c008080`, which contains the first "useful" instruction as explained during the lecture. You can use the disassembly to understand the meaning of each line.

Simulation Waves to gather more information (and debug potential issues) you will need to use the QuestaSim Graphical User Interface (GUI). You can execute the following command:

```
make run gui=1
```

This will open a window similar to this:

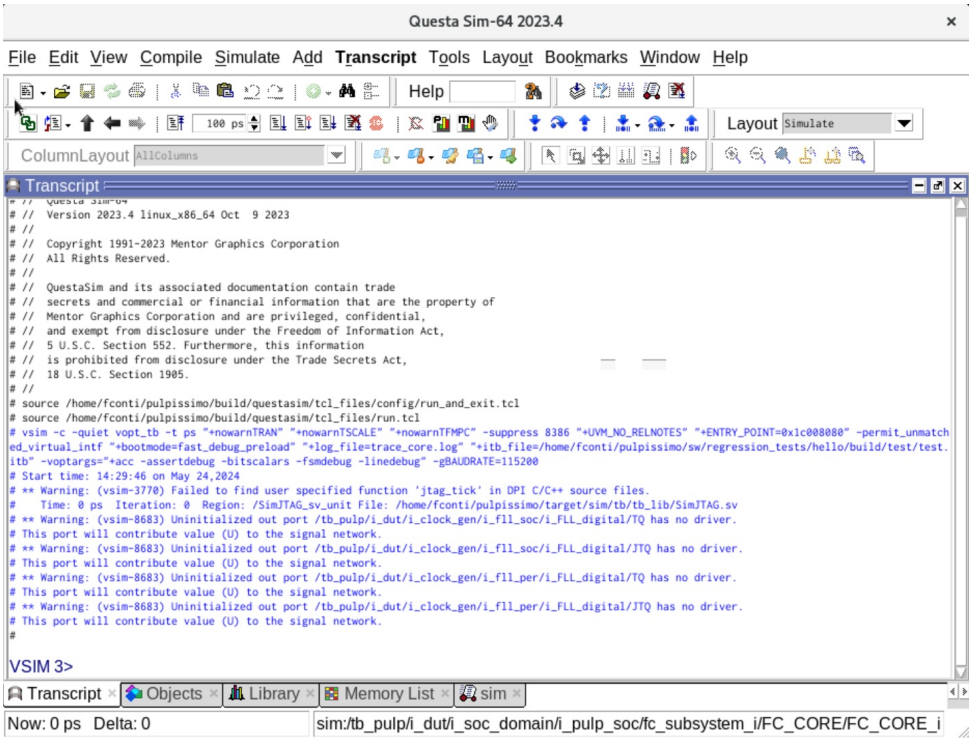

Figure 4: QuestaSim landing window.

We generally assume some familiarity with QuestaSim or equivalent tools, but here are a couple of minimal hints that could be useful. The *Transcript* is a command-line window for QuestaSim. The *sim* tab includes the hierarchy of instances of the simulated design. You can use the search bar to look for a specific instance, e.g., `FC_CORE` (the fabric controller core):

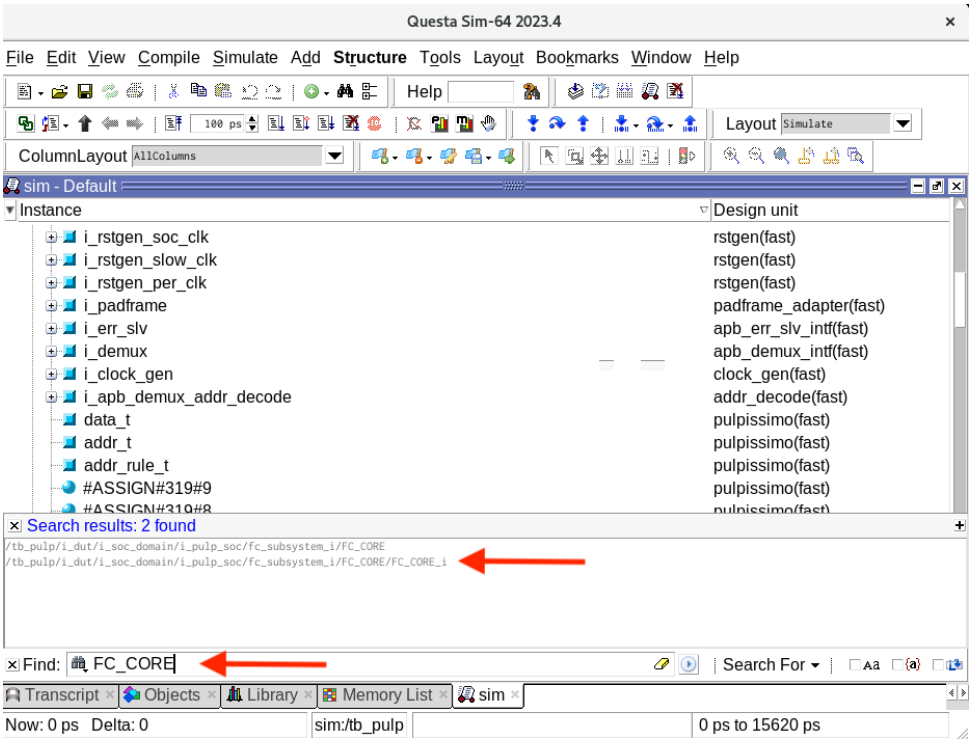

Figure 5: QuestaSim instance search.

which can then added to the wave by right-clicking on it and selecting *Add to Wave*:

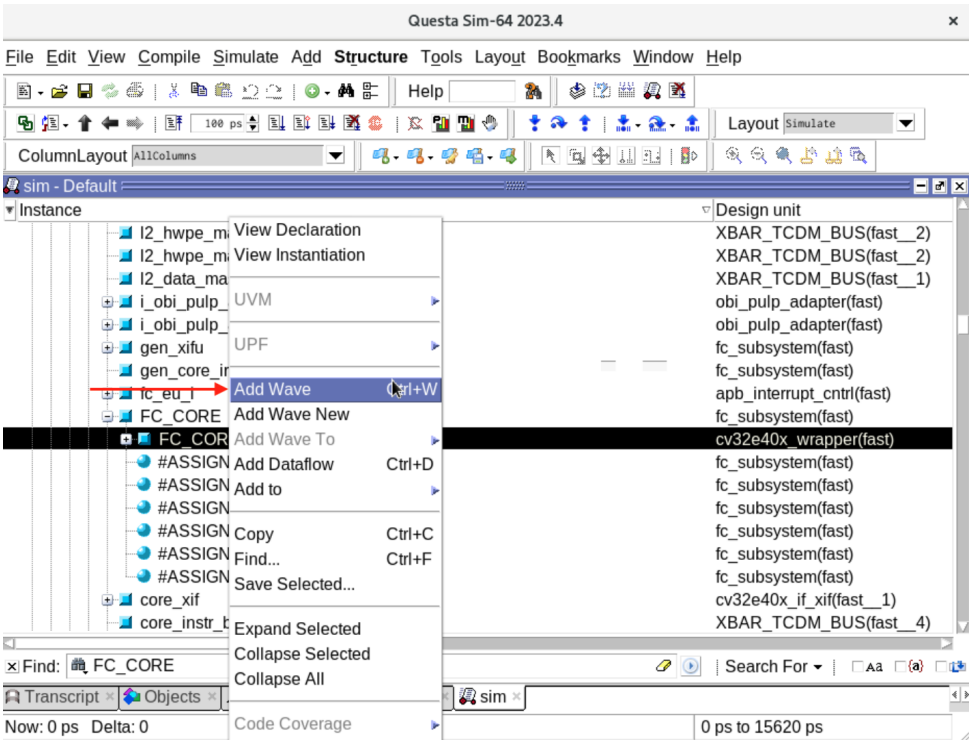

Figure 6: QuestaSim add waves.

To actually run the simulation, go to the *Transcript* and use the following command:

```
run -all
```

The *Wave* window will fill with waves for all signals that have been selected. The simulation is set up so that all signals are logged, so if you do not remember to log a signal, you can also add it later after the simulation has been run. Be sure to familiarize yourself with all of the commands of the waveform viewer, especially the ones visible here:

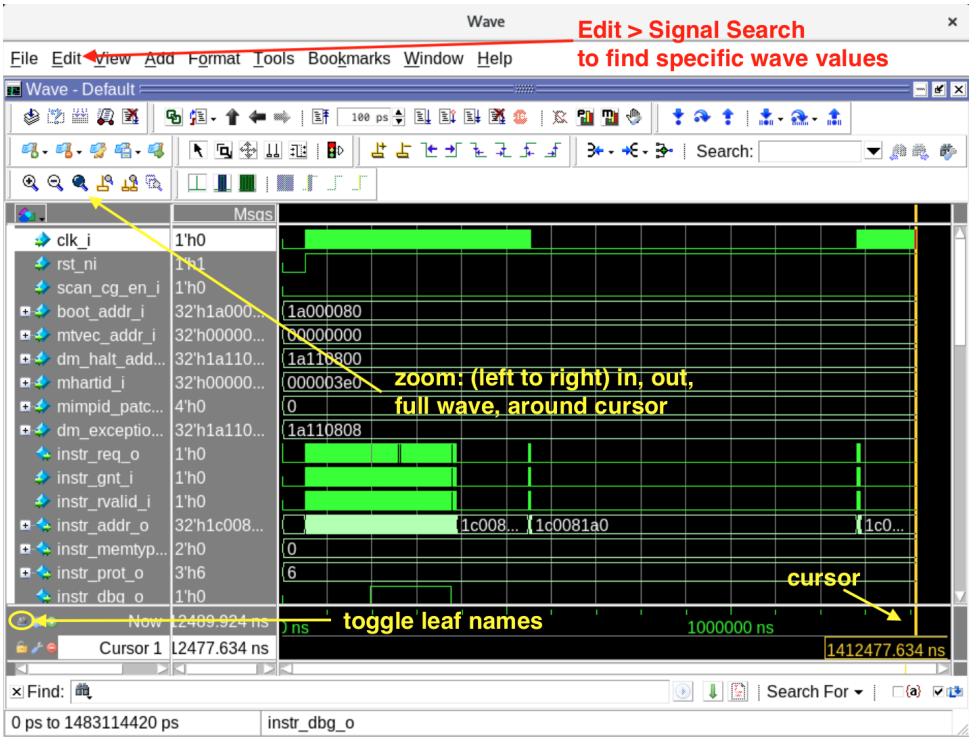

Figure 7: QuestaSim *Wave* view.

Student Task 3 (Debugging a Simulation):

- Inspect the traces and disassembly of the hello world application.
- Start the simulation in the GUI following the instructions above. Add the waves to view the `FC_CORE`.
- Try to sync a view on 1) simulation, 2) core trace, and 3) disassembly on the following instruction, which is the first one of the `puts` function.

```
1c0082d6: 03 46 05 00    lbu     a2, 0x0(a0)
```

> **Note 6:** The specific address `1c0082d6` could be slightly different on your machine; however it will be in the general vicinity. You can also simply search for `puts` in the disassembly.

- keep the simulation open for the following task!

This setup is sufficient to enable simple experiments on PULPissimo, but to dig deeper into the Hardware Description Language (HDL) source code (and possibly modify the system), you need to go inside the Bender-managed IPs, which constitutes 95% of the design. These are inside the `.\bender` folder, where they are not assumed to be tampered with but used "as is". Modifying them here is dangerous, as any update may overwrite your changes, losing any progress or updates you may have made. Fortunately, bender offers the `clone` command to assist in editing existing dependencies. The entry point, as highlighted during the lecture, is the `pulp_soc` module. In the following task, you will fetch a local working copy of `pulp_soc` and dig into it.

PULPissimo SoC architecture: For some help in tracing the elements, Figure 8 contains high-level overview and introduction into the PULPissimo System-on-Chip (SoC) architecture within the `pulp_soc` dependency. The SoC part connects the Fabric Controller (FC) subsystem to a variety of memory banks and peripherals through the SoC interconnect. Feel free to explore these elements in the code and in Student Task 4.

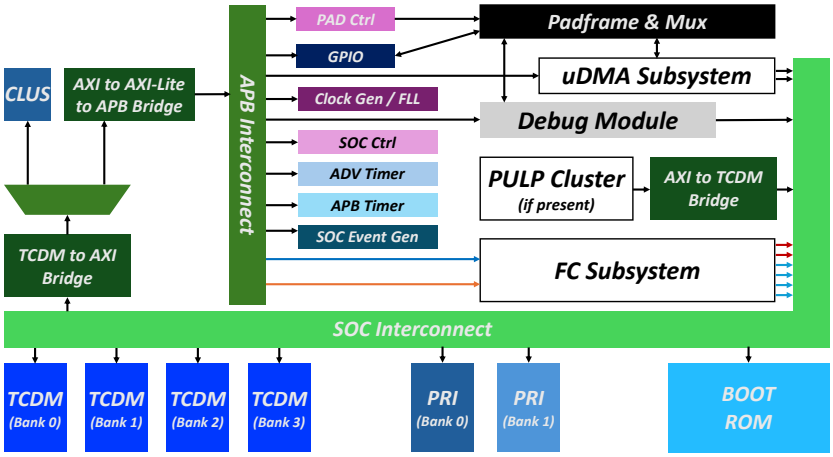

Figure 8: Block-level schematic of the PULPissimo SoC

Student Task 4 (Working with dependencies): In future exercises, we will be modifying PULPissimo's core components. To understand the hardware structure, we will have a look at the `pulp_soc` module.

- Fetch a `pulp_soc` copy into `working_dir`. Within the `pulpissimo` root folder, type:

```
oseda -2025.07 bender clone pulp_soc
```

  You will find the cloned folder into "pulpissimo/working_dir/pulp_soc"

- Open design with an editor. A variety of editors are installed for you, such as VScode (`code`), Sublime Text (`sublime_text`), Emacs (`emacs`), and others.

  Note 7: You may need to install packages supporting for syntax highlighting of SystemVerilog or other formats.

- Navigate the code and see if you can find different PULPissimo components in the RTL code and in the simulation window: `FC_CORE_i`, `i_soc_interconnect`, `soc_peripherals_i`

4.1.3 Software Application: FIR filter

We will now progress to the development of a simple, pure-SW baseline application that will be useful as a baseline for our next steps that go towards the acceleration / modification of the HW platform. The baseline application is based on a simple Finite Impulse Response (FIR) filter. As a reminder, a FIR filter with N taps operates on a time-series input x_n in the following way:

$$y_n = \sum_{k=0}^{N-1} h_k \cdot x_{n+k} \tag{1}$$

where we assume x_0 is the most recent sample and h_k coefficients are properly ordered. Here we will focus on a purely software-centric FIR implementation, with x and y allocated in L2 memory.

Student Task 5 (Software Development & Testing): Let's develop a simple FIR test and benchmark it. This will provide us with a performance baseline for our next acceleration steps.

- Open a browser and go to `https://github.com/FondazioneChipsIT/efcl_fir_genstim.git` - it contains a Jupyter Notebook that you can run on Colab (requires a Google account) or Binder (no requirements). Follow through the notebook down to the generation of a set of stimuli, which you should download (NOTE: generated stimuli files should be found by clicking on the folder icon on the top-left corner of the page)
- Go to the folder `sw/efcl2026/task5_baseline_fir_filter/`. Navigate it and observe it contains a baseline `Makefile`, a `fir_test.c` containing the "main" of the test, and a placeholder `fir.c` with a fake FIR function. The `fir_test.c` includes setup functions to "pad" the input with a set of zeros, as well as to measure performance.
- Plug in the stimuli you got from the Jupyter Notebook into the test "include" folder, then try to build and run the test as-is. What happens?

  Note 8: You can kill/stop the simulation with `CTRL+C` in CLI mode or by clicking the "stop" button in GUI mode if it takes too long.

- Keeping in mind the definition of a FIR filter, complete the `fir.c` placeholder with a suitable function.
- Test and debug your software, exploiting all the capabilities discussed: waves, `printf`'s, traces. At the end, the test should work and we expect an output similar to the following:

```
# [STDOUT-CL31_PE0, 9484203ns] Test ended in <NUMBER> cycles
# [TB  ] 9708152ns - Received status core: 0x00000000
# ** Note: $stop    : /home/<YOUR_USER_NAME>/pulpissimo/target/sim/tb/tb_pulp.sv(821)
#    Time: 9708151716 ps  Iteration: 0  Instance: /tb_pulp
# Break in Module tb_pulp at /home/<YOUR_USER_NAME>/pulpissimo/target/sim/tb/tb_pulp.sv line 821
# Stopped at /home/<YOUR_USER_NAME>/pulpissimo/target/sim/tb/tb_pulp.sv line 821
# End time: 12:30:26 on May 27,2024, Elapsed time: 0:02:08
# Errors: 0, Warnings: 10
```

- How fast is your implementation? Do you think you can improve it to make it faster?

For future parts of the exercise, or modifications to the PULPissimo architecture, it may be relevant to have an initial understanding of the memory map within the SoC. Figure 9 and Figure 10 illustrate the address space mapped in the PULPissimo system.

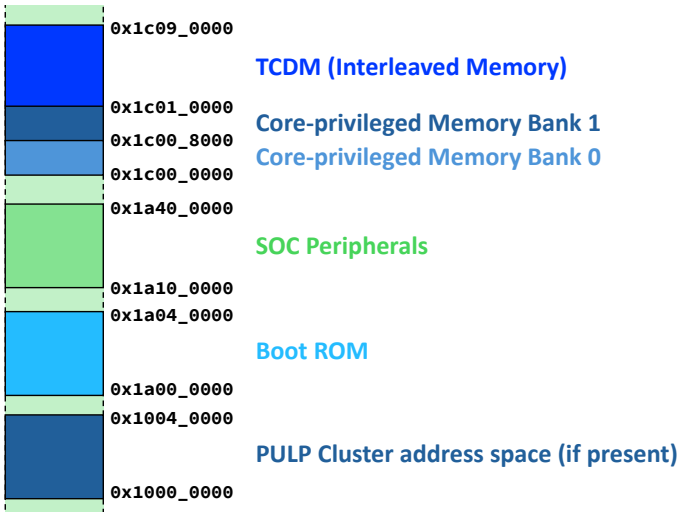

Figure 9: Address map for the PULPissimo design.

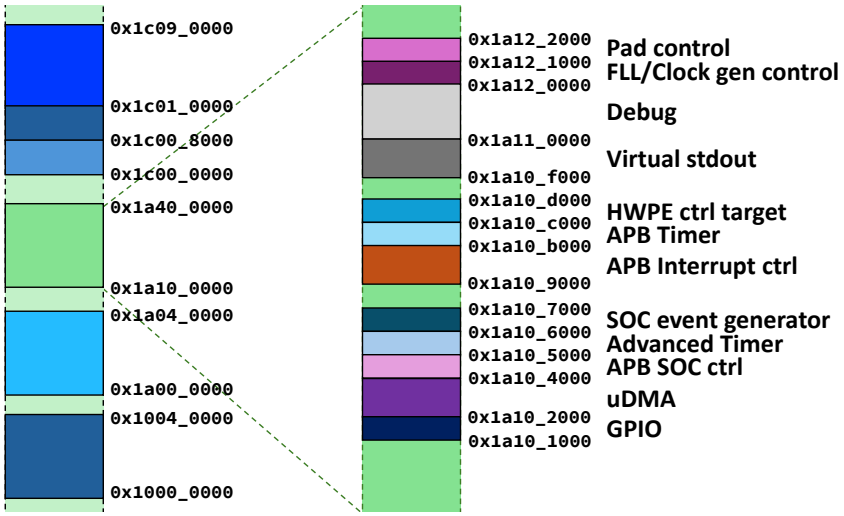

Figure 10: Address map for the peripheral space in the PULPissimo design.

You are done with this exercise.
Discuss your results and any questions you might have with an instructor.

4.2 RISC-V ISA extension in hardware

In this hands-on session you will deal with the development of some Instruction Set Architecture (ISA) extensions to accelerate a typical Digital Signal Processing (DSP) application: the Finite Impulse Response (FIR) filter.

Two of the possible approaches to implement an ISA extension are:

- a direct modification to the pipeline of the baseline processor
- the implementation of a co-processor tightly coupled to the processor through a dedicated interface

For the sake of simplicity, you will develop the FIR ISA extension adopting the second approach and using the CORE-V *X-Interface (XIF)*. Specifically, we focus on the XIF as defined in the CV32E40X docs[2]. To the best of our understanding, this corresponds to `v0.2.0` of the CV-XIF interface[3]. Later XIF revisions remove useful functionality such as memory requests from the XIF units. Also notice that our version of CV32E40X is lightly patched to better support the XIF interface[4].

In order to tackle the acceleration of the FIR application through ISA extensions, you will have to introduce 3 new instructions:

- a **dot-product (DOTP)** instruction called `xfirdotp`. `xfirdotp` follows the R-format of RISC-V instructions, e.g.,

  ```
  xfirdotp xrd, xrs1, xrs2
  ```

  where `xrd`, `xrs1`, `xrs2` are registers in the X-Interface Functional Unit (XIFU) register file. `xfirdotp` will take two 16-bit 2-element vectors out of the source registers `xrs1, xrs2` and perform the dot-product, accumulating the value into `xrd` used as an accumulator:

  ```
  xrd <- xrd + xrs1[15:0] * xrs2[15:0] + xrs1[31:16] * xrs2\
     [31:16]
  ```

  where all operands are sign-extended to 32-bits.

- a custom **load** instruction with **post-increment** to load data from memory to the dedicated Register File of the co-processor, called `xfirlw`. `xfirlw` uses the I-format of RISC-V instructions, e.g.,

  ```
  xfirlw xrd, Imm(rs1)
  ```

  `xfirlw` copies the content of memory at the address given by a (word-aligned) pointer in the `rs1` register (in the architectural register file of the core) into the XIFU register `xrd`. At the same time, it auto-increments `rs1` by the immediate value `Imm`:

  ```
  xrd <- Mem[rs1]
  rs1 <- rs1 + Imm
  ```

  where the immediate is sign-extended.

[2] https://cv32e40x-user-manual.readthedocs.io/en/latest/x_ext.html
[3] https://docs.openhwgroup.org/projects/openhw-group-core-v-xif/en/v0.2.0
[4] https://github.com/pulp-platform/cv32e40x/tree/fc/xif-fixes

- a custom **store** instruction to store data from the custom Register File of the co-processor to the memory, called `xfirsw`. `xfirsw` uses the S-format of RISC-V instructions, e.g.,

```
xfirsw Imm(rs1), xrs2
```

`xfirsw` right-shifts the content of the XIFU register `xrs2` by the 5 least significant bits of the immediate; then it copies the result into memory at the address given by a (word-aligned) pointer in the `rs1` register (in the architectural register file of the core). At the same time, it auto-increments `rs1` by the 7 most significant bits of the immediate `Imm`:

```
Mem[rs1] <- xrs2 >> Imm[4:0]
rs1 <- rs1 + Imm[11:5]
```

where the immediate is sign-extended.

We will use a top-down approach, starting from connecting the "empty shell" of the XIFU and gradually completing all the blocks we need.

4.2.1 Instantiate and correct FIR XIFU placeholder

To make the design of this complex ISA extension unit possible in the limited hands-on time, we developed a stub for the XIFU that you can already plug inside PULPissimo. The stub can be built but, of course, it will not be functional before deep modifications are made. The architecture of the XIFU, explained during the lecture, is visible in Fig. 11:

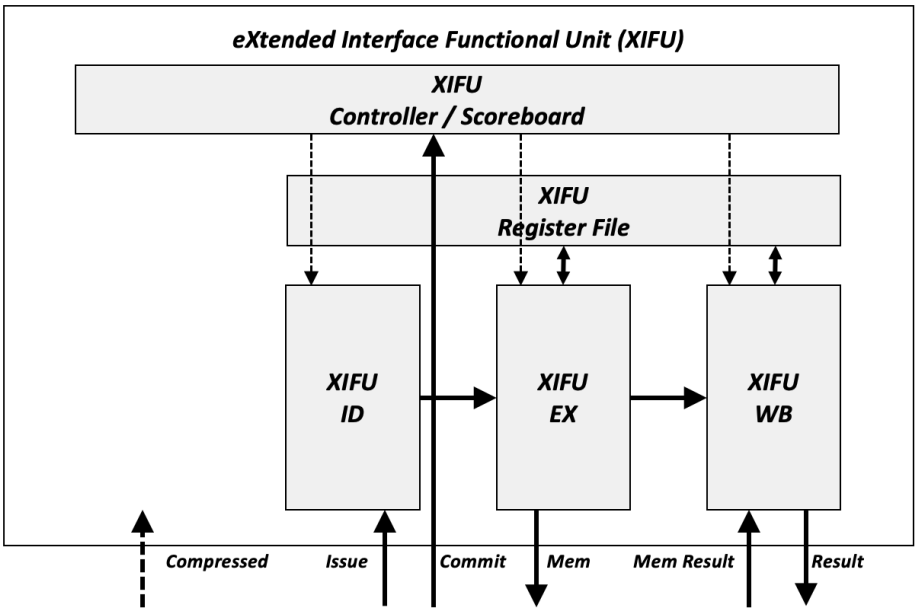

Figure 11: XIFU stub architecture.

Student Task 6 (Adding a FIR XIFU stub): Let's create a new Intellectual Property (IP) for the FIR XIFU stub.

- Clone the XIFU stub in `working_dir`

```
cd working_dir
git clone https://github.com/FondazioneChipsIT/fir-xifu-stub.\
   git
```

  This stub (on-purpose) does not have a Bender.yml manifest.

- Initialize the bender package by running

```
cd fir-xifu-stub
oseda -2025.07 bender init
```

- Update the Bender.yml, listing IP name, authors, and source files (add all the ones in the `rtl` folder):

```
sources:
  - foo.sv
  - bar.sv
```

  It is required to start hierarchically from bottom up (e.g., first the SystemVerilog package, then the leaf modules, with the top module last).

  Note 9: Take a look at the `Bender.yml` manifest of `pulp_soc` in `working_dir/\ pulp_soc/Bender.yml` and at the top-level of PULPissimo for inspiration.

  Observe that each module corresponds 1:1 with the modules shown in Fig. 11.

- Add the correct CV32E40X dependencies:

```
dependencies:
  cv32e40x: { git: "https://github.com/pulp-platform/cv32e40x.\
     git", rev: xifu-v0.1.0 }
```

- Link the dependency into `pulp_soc` by adding the following line in the dependencies section of the `Bender.yml` of `working_dir/pulp_soc`:

```
dependencies:
  fir-xifu: {path: "../fir-xifu-stub"}
```

  Note 10: This links the dependency through the path. Common procedure for open-source projects is to link through git dependencies instead of paths.

- Force the regeneration of scripts including the newly added IP. In the root of PULPissimo:

```
rm -rf .bender
oseda -2025.07 bender update
make checkout scripts
```

- Connect the FIR XIFU stub inside PULPissimo's peripheral domain in `pulp_soc`. Look for *"placeholder"* inside the `fc_subsystem.sv` and plug in the XIFU instantiation (look in the top-level file `fir_xifu_top.sv` to see what is the exposed interface). Set the number of registers in the XIFU to 32 by overriding the `NB_REGS` parameter.

  Note 11: The "clear_i" input port of the xifu stub interface is not used, you can connect it to 0.

- Finally, you need to enable the `USE_XIFU` parameter by overriding its value to 1 inside `pulp_soc.sv` where the `fc_subsystem` is instantiated.

Take a few minutes to explore the architecture of the XIFU starting from `fir_xifu_top.sv`. Modules are connected to one another using SystemVerilog `struct packed`, and with the XIF interface using SystemVerilog interfaces defined by CV32E40X.

4.2.2 Define the instruction encoding

Now we will decide the instruction encoding step-by-step, and then introduce it in the XIFU model. The RISC-V available encoding space for instructions is shown in Fig. 12:

inst[4:2] inst[6:5]	000	001	010	011	100	101	110	111 (> 32*b*)
00	LOAD	LOAD-FP	*custom-0*	MISC-MEM	OP-IMM	AUIPC	OP-IMM-32	48*b*
01	STORE	STORE-FP	*custom-1*	AMO	OP	LUI	OP-32	64*b*
10	MADD	MSUB	NMSUB	NMADD	OP-FP	*reserved*	*custom-2/rv128*	48*b*
11	BRANCH	JALR	*reserved*	JAL	SYSTEM	*reserved*	*custom-3/rv128*	≥ 80*b*

Figure 12: RISC-V available instruction encoding space.

For this exercise, we will use the *custom-2/rv128* available space, meaning that our opcode is defined in the following way:

```
inst[4:2] = 3'b110
inst[6:5] = 2'b10
```

Moreover, we will stick to 32-bit instructions only (CV32E40X does not currently support compressed instruction offload to XIF anyways!), so the last two bits are set to 1 by RISC-V specifications:

```
inst[1:0] = 2'b11
```

The opcode defines the overall area of the encoding space our extension will use, but not the encoding of specific instructions. That is defined by choosing the encoding formats for each instruction, which are reported in Fig. 13:

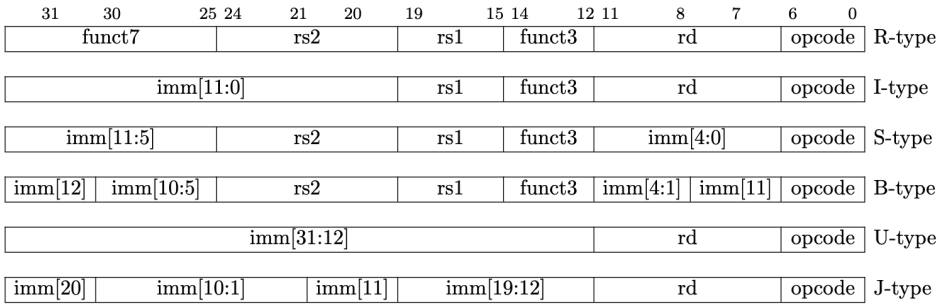

Figure 13: RISC-V instruction encoding formats.

For our instructions, we will follow encodings similar to the standard instructions they are inspired with. `xfirdotp` is a register-register instruction (albeit on the XIFU regfile instead of the CV32E40X one!) and therefore the most suited encoding format is as an *R-type* instruction, like most ALU instructions in RISC-V. For `xfirlw` we follow the convention of other load instructions, and therefore we opt for an *I-type* encoding. Finally, `xfirsw` follows the convention of other store instructions using an *S-type* encoding. In all these formats, the specific instruction is defined by the `funct3` field (and potentially by `funct7` for R-type). We will set the following encodings:

```
xfirlw:   funct3 = 3'b000
xfirsw:   funct3 = 3'b001
xfirdotp: funct3 = 3'b010, funct7 = 7'b0 (funct7 is ignored in XIFU\
   )
```

Note 12: You are free to choose your own encoding if you really want to, but beware: if you do so, you must adapt **all your code** later on (including the compiler at the end of today's afternoon lecture).

Student Task 7 (Introduce encoding in XIFU):

- Once the encoding has been defined, it can be directly introduced inside the XIFU. Open the `fir-xifu` folder with an editor, then edit `rtl/fir_xifu_pkg.sv`.

  Note 13: Look for the "placeholder" keyword, which is always around areas of code you need to change.

- Replace the instruction opcode with the correct string as defined above. This is shared between all three instructions.
- Replace the funct3 for all three supported instructions.
- From the PULPissimo root folder, rebuild the simulation platform with

  ```
  make build
  ```

4.2.3 Minimal SW and testing infrastructure

In the following task, we will design an extremely minimal C application designed to test the functionality of the XIFU you are designing today. You will be able to use this as a go-to test early in your design to check for minimal functionality. The task will

1. define three simple words as "test vectors" for the XIFU dot-product instruction (two inputs, one output);
2. load pointers to them in the CV32E40X register file through inline assembly instructions: for a register-immediate instruction, an example is

```
asm volatile("add <RISC-V reg>, zero, %0" ::"r"(<C pointer>));
```

3. use inline assembly with raw bits (quite literally "hard-core" coding) to perform two XIFU loads, one dot-product, and one store to test the XIFU functionality. This kind of inline assembly lets you literally define the code arbitrarily:

```
asm volatile(".word 0xdeadbeef\n");
```

   Of course *with great power comes great responsibility...*

4. check the expected result.

Regular inline assemblies generally use the ABI names of RISC-V registers, but for `.word` with raw-bits we need to directly indicate registers by their number (`x0-x31`). For reference, you can find here in Fig. 14 the RISC-V calling convention table with all register ABI names and numbers:

Register	ABI Name	Description	Saver
x0	zero	Hard-wired zero	—
x1	ra	Return address	Caller
x2	sp	Stack pointer	Callee
x3	gp	Global pointer	—
x4	tp	Thread pointer	—
x5–7	t0–2	Temporaries	Caller
x8	s0/fp	Saved register/frame pointer	Callee
x9	s1	Saved register	Callee
x10–11	a0–1	Function arguments/return values	Caller
x12–17	a2–7	Function arguments	Caller
x18–27	s2–11	Saved registers	Callee
x28–31	t3–6	Temporaries	Caller
f0–7	ft0–7	FP temporaries	Caller
f8–9	fs0–1	FP saved registers	Callee
f10–11	fa0–1	FP arguments/return values	Caller
f12–17	fa2–7	FP arguments	Caller
f18–27	fs2–11	FP saved registers	Callee
f28–31	ft8–11	FP temporaries	Caller

Figure 14: RISC-V calling convention and register names.

Student Task 8 (Minimal software test):

- create a new folder `sw/efcl2026/task8_xifu_minimal` (you can copy the

`task2_hello` if you are lazy)

- define three `uint32_t` for the two inputs and the output (e.g., `x, y, z`); set the inputs to values chosen by you, and the outputs to 0.
- define three pointers to the variables defined above, casted to `uint32_t` (e.g., `px, py, \ pz`).
- load the pointer values `px, py, pz` to three RISC-V registers of your choice with inline assembly.

> **Note 14:** You can use for example a `li` or `addi` instruction. Temporary registers are probably a good choice for the registers to be used. The compiler expects these registers named according to the RISC-V calling convention; take note of the "raw" base-32 register ID however, you will need it for the raw-bits instructions!

- use four inline assembly with raw bits to perform two Loads, one DotProduct, and one store. You will need to remember the instruction encoding chosen previously! Also refer to Fig. 13 for the correct encoding. For now, do not test the auto-increment function of `xfirlw` and `xfirsw` and the right-shift of the latter instruction.

> **Note 15:** The inline assembly words can be expressed in terms of bitfields with `|\` (bitwise or) and `<<` (left-shift) operations; moreover you can indicate bitfields both in hexadecimal format (`0x16`) or in binary format (`0b00010110`). You can also add comments with the plain C style (*`/* comment */`*).

- space out the instructions putting a no-operation between them, e.g.,

```
// xfir instruction
asm volatile("xor zero, zero, zero");
// other xfir instruction
```

- calculate by hand the expected output and check that the value pointed by `pz` (i.e., `z`) corresponds to the expected output, returning 0 if so and 1 if not.
- build the test and try to run it (with a GUI): it should trigger an illegal instruction exception!

```
# 1347151ns: Illegal instruction (core 0) at PC 0x1c008164:
# [TB  ] 1351966ns - retrying debug reg access
# [TB  ] 1395966ns - Waiting for end of computation
# [TB  ] 1490066ns - Received status core: 0x1c000830
# ** Note: $stop    : /home/<YOUR_USER_NAME>/pulpissimo/target/sim/tb/tb_pulp.sv(821)
#    Time: 1490065604 ps  Iteration: 0  Instance: /tb_pulp
```

- Open the waves. Add the XIF interface in the XIFU stub, you can do that manually or with the command

```
add wave sim:/tb_pulp/i_dut/i_soc_domain/i_pulp_soc/\
   fc_subsystem_i/gen_xifu/i_fir_xifu_top/xif_issue_i/*
```

- At the timestamp of the illegal instructions, something is offloaded through the X interface (visible by the fact that `issue_valid=1'b1`): check the `issue_req.instr` corresponds to the expected raw-bits inline assembly. Also check that the XIFU does not accept these instructions yet (`issue_resp.accept=1'b0`).
- Note down the timestamp where "interesting things" happen, you'll need it later.

As you build the XIFU in the following tasks, this test will provide you the first (and before we develop the compiler, the only) way to check that what you are doing makes sense.

4.2.4 ID stage

Now we will start to progressively fill the various stages of the XIFU with the logic necessary to process the XIF. We will start with the ID stage, which needs to be changed in two main parts:

1. the actual internal XIFU "decoder", which decides whether an instruction is accepted or not at the XIF interface and forwards the instruction internally.
2. the ID/EX pipe, which defines what information should be forwarded to the EX pipe stage.

Student Task 9 (Modify the Instruction Decode stage (ID stage)):

- Open `fir_xifu_id.sv` with your editor and look for the "placeholder" keyword, which indicates code to be replaced.
- Go to the instruction decoding combinational block (line 70 and following). The `unique\ case` statement checks for some bits in the instruction coming from XIF to detect which instruction it is: use this information to add the actual cases to check.
- The XIF interface expects three bits in its response: `accept` (whether or not the instruction will be executed in the XIFU), `writeback` (whether or not the instruction will write back to the CV32E40X register file), and `loadstore` (whether or not the instruction will use the LSU of CV32E40X). Adapt them to our instructions.
- The rest of the stage also needs to set `valid_instr` and `instr` in a reasonable way, do that.
- Go to the pipe stage (line 113 in the original stub). You need to correctly fill the placeholders for `id2ex_d.rs1, rs2, rd, offset`. The latter in particular requires some care: follow the instruction in the comment.
- With these changes, rebuild the simulation platform from the PULPissimo root:

```
make build
```

- Run the `task8_xifu_minimal` test again and observe the outcome, looking specifically at the XIF signals around the timestamp you noted down from the previous task. Can you make an hypothesis of what is going wrong? Discuss this with a TA.

4.2.5 EX stage

The execution stage is the heart of the implementation of our ISA extension. Like the rest of the XIFU pipeline, it is modeled over the existing CV32E40X pipeline, meaning that it contains both the actual hardware executing arithmetic operations and the triggering of the LSU. The EX stage also communicates with the register file by selecting specific registers (`ex2regfile_o`) and using the data returned (`regfile2ex_i`). The current stub needs to be changed in all of these aspects:

1. the `xfirdotp` datapath, which should follow the specifications described earlier:

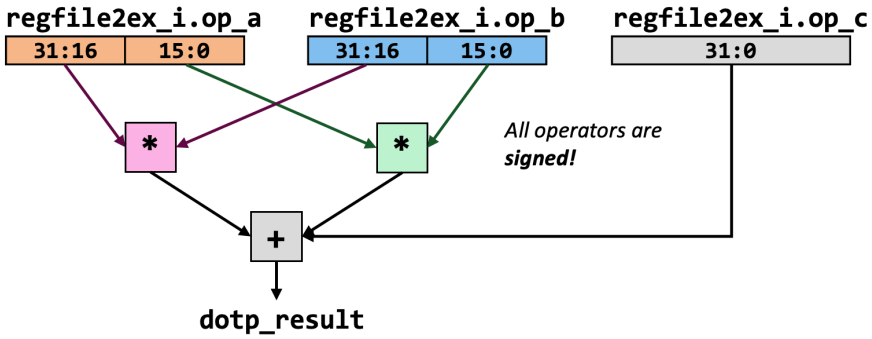

Figure 15: FIR XIFU `xfirdotp` unit.

2. the autoincrement datapath for `xfirlw` and `xfirsw`.
3. the right-shift datapath used in `xfirsw`.
4. the memory request through XIF (which is forwarded to CV32E40X's LSU).
5. the EX/WB pipe, which defines what information should be forwarded to the WB pipe stage.
6. the interface with the register file.

Student Task 10 (Modify the Execution Stage (EX stage)):

- Open `fir_xifu_ex.sv` with your editor and look for the "placeholder" keyword, which indicates code to be replaced.
- Lines 88–118 of the original stub file contain a few suggestions on how to design the dot product unit. Consider the dot-product architecture of Fig. 15 and implement it.
- The auto-increment datapath should be added around line 50 of the original stub.
- The `always_comb` block at line 63 of the original stub contains a placeholder for the memory requests to be offloaded back to memory; fill it with the information necessary to process the correct memory request.
- Finally, consider also the EX/WB pipe stage and the interface to the register file, which is needed to extract the correct data.
- With these changes, rebuild the simulation platform from the PULPissimo root:

```
make build
```

- Run the `task8_xifu_minimal` test again and observe the outcome, looking specifically at the XIF signals around the timestamp you noted down from the previous tasks. Also look what happens inside the EX stage.

4.2.6 WB stage

The Write-Back Stage (WB stage) is correspondent to CV32E40X's WB stage. Here we need to modify the stub so that results are actually propagated to the register files of both the XIFU (through the internal signals) and CV32E40X (through the XIF).

Student Task 11 (Modify the WB stage):

- Open `fir_xifu_wb.sv` with your editor and look for the "placeholder" keyword, which indicates code to be replaced.
- Lines 50–56 of the original stub file need to be changed to account for actual write-back information towards the XIFU register file.
- Lines 101–104 refer to the write-back towards the CV32E40X register file, as well as other response information towards CV32E40X. Fix these to report back the correct output.
- With these changes, rebuild the simulation platform from the PULPissimo root:

```
make build
```

- Run the `task8_xifu_minimal` test again and observe the outcome, looking specifically at the XIF signals around the timestamp you noted down from the previous tasks. Also look what happens inside the WB stage.
- Is the test working?

4.2.7 (Optional) A little faster, please

Note 16: If you opt not to perform this optional task, discuss with a TA where to get a fixed `fir_xifu_regfile.sv` for the following tasks.

Are we really done? Not yet, but we are close...

Student Task 12 (Remove no-operations from the minimal test): Remove all no-operations from the minimal test and re-run the test. Does it work?

There is one small missing point in our design: to enable some of the bypassing between stages! Part of it is already in the baseline XIFU stub to cover a few corner cases, but the main need is to bypass the register file when the WB stage destination register and one of the EX stage source registers are the same.

Student Task 13 (Activate bypassing):

1. Open `fir_xifu_regfile.sv` with your editor and look for the "placeholder" keyword, which indicates code to be replaced.

2. Modify the `regfile2ex_o` assigns so that they take into account bypassing. The requirement is that if the destination register `rd` from the WB stage is equivalent to the source register `rs1/rs2/rd` from the EX stage, then we should simply forward the WB stage `result`.
3. Run the `task8_xifu_minimal` test one final time (almost!) and observe the outcome, looking specifically at the XIF signals around the timestamp you noted down from the previous tasks. Also look what happens inside the register file.
4. Does it work?

$\mathcal{E}$ **You are done with this exercise.**
Discuss your results and any questions you might have with an instructor. $\mathcal{E}$

4.3 Cooperative Hardware Processing Engine

4.3.1 Setup and Introduction

On the first day, you learned how to implement a baseline Finite Impulse Response (FIR) filter through software and how to implement an Instruction Set Architecture (ISA) extension to accelerate it. Today, you will accelerate the FIR filter workload using a dedicated cooperative hardware accelerator, commonly referred to within the PULP Platform project as a Hardware Processing Engine (HWPE). You will reuse the same golden model written in Python to generate stimuli and verify the output. Let's begin by downloading the codebase and familiarizing ourselves with the folder structure.

Student Task 14 (Downloading the `fir-hwpe` skeleton):

- Add `fir-hwpe` as a dependency to `working_dir/pulp_soc/Bender.yml` by adding an entry

```
fir-hwpe: { git: "https://github.com/FondazioneChipsIT/fir-\
   hwpe.git", version: =2.0.3 }
```

- Create a working copy of `fir-hwpe` using the following commands from pulpissimo top-level:

```
sh> oseda -2025.07 bender update
sh> oseda -2025.07 bender clone fir-hwpe
```

- Use the following command to display the top-level folder structure of the repository.

```
sh> cd working_dir/fir-hwpe
sh> tree -L 2
```

The folder structure of the fir-hwpe is given below with the explanations to the relevant directories

```
fir-hwpe
├── rtl/
├── tb/
├── stimuli/
└── sw/
    └── inc/
```

- `fir-hwpe/`: Top folder structure.
- `rtl/`: Contains the hardware description files including simulation sources.
- `tb/`: Contains the simulation testbench sources.
- `stimuli/`: Stimuli to be copied from the golden model.
- `sw/`: Software infrastructure to test the standalone fir-hwpe.
- `sw/inc/`: Software stimuli to be copied from the golden model.

A design choice for this accelerator is that at the interface of modules only a few categories of signals are exposed:

- global signals (`clk_i`, `rst_ni`)
- HWPE-Stream interfaces (`x`, `h`, `y`)
- an input control packed struct (`ctrl_i`) and an output flags packed struct (`flags_o`)

The `flags_o` packed struct encapsulates all of the information about the internal state of the module that must be exposed to the controller, and the `ctrl_i` all the control information necessary for configuring the current module. In this way, it is possible to make significant changes to the control interface (which can typically propagate through a large hierarchy of modules) without manually modifying the interface in all modules; it is sufficient to change the packed struct definition in the package where it is defined. **Packed structs are essentially bit vectors where bit fields have a name, and as such are easily synthesizable and much more readable than Verilog-2001-ish code.**

Looking at the code, you can notice that the HWPE streams are unrolled into `logic` bit vectors for convenience: they're split in `*_data` and `*_valid` signals. We identify the **inputs** with **`x`**, the **coefficients** with **`h`**, and the **outputs** with **`y`**, keeping the notations consistent with Equation. 1.

Student Task 15 (Downloading the dependencies):

```
sh> cd working_dir/fir-hwpe
sh> make update-ips
```

This command will checkout all the dependent Intellectual Property (IP) needed for the `fir-hwpe`.

4.3.2 Deep dive to the datapath

Student Task 16 (Fixing the datapath): We have provided the datapath of the HWPE in `fir-hwpe/rtl/fir_datapath.sv`. There are placeholders to be completed for this task.

1. **Ordering the Inputs** - The first task is to delay the input signal. Note that the first input is not delayed, while the others need to be delayed. You should use a shift register to delay the input.
   Hint: Relevant signal is `x_delay_data_q`.
2. **Accelerating element-wise multiplication** - Perform element-wise multiplication of `h_data` and `x_delay_data_q`. We will prioritize faster latency at the expense of using more resources. Implement your solution in the provided placeholder.
   Hint: Assign the product to the `prod_d` variable and ensure typecasting to produce a 64-bit signed output.
3. **Block Diagram** - After we fixed the datapath, let's sketch out a block diagram of the datapath in the space provided. You can make use of the elements such as adder tree and shift registers.

Datapath

Now our datapath for the FIR filter is ready for verification. Now let's create the relevant stimuli to verify the correctness of our datapath. We will reuse the golden model that we have used in Section-4.1.3.

Student Task 17 (Generating stimuli): Run the golden model available at `https://github.com//FondazioneChipsIT/efcl_fir_genstim.git`. The golden model will generate the following files: `x_stim.txt` for input, `h_stim.txt` for filter coefficients, and `y_gold.txt` for the expected output. Download these files and move them to the `fir-hwpe/stimuli` folder. Additionally, move the `x_stim32.h`, `h_stim32.h`, and `y_gold32.h` files to the `fir-hwpe/sw/inc` folder. Note that while we will use the **`.txt`** files for this part of the exercise, the **`.h`** files will be used later in the exercise.

Next, we will start our verification by testing the datapath implemented in the previous tasks. The testbench files are located in the `tb` folder, and the relevant file for this exercise is `tb_fir_datapath.sv`.

Student Task 18 (Familiarizing with the testbench): Open the `tb_fir_datapath.sv` file and familiarize yourself with the code. You will notice `fir_datapath` is instantiated in the testbench. Additionally, there are `hwpe_stream_traffic_gen` and `hwpe_stream_traffic_recv` modules. Can you guess what purpose these modules served?

Note 17: Only spend 2 minutes on this.

Also, note that the `STIM_FILE_X`, `STIM_FILE_H` and `STIM_FILE_Y` are the *.txt files in the stimuli folder. We will overwrite the path to point to the correct folder while running the simulation. Let's move towards building the simulation target.

Student Task 19 (Compile your RTL): Set the correct version of Questasim to compile your code.

```
sh>  export QUESTA=questa-2022.3
```

From the `fir-hwpe/` directory, run:

```
sh>  make hw-all TESTBENCH=tb_fir_datapath
```

This command will compile the RTL and the testbench for the datapath of the FIR filter.

Check the compilation report to verify that in your code there are no syntax errors.

Note 18: If you need to fix bugs in your code, it is strongly recommended to clean the libraries created by QuestaSim during the previous compilation. This prevents the tool to re-use pre-compiled code that, in some cases, may lead to not seeing your modifications reflected in the new compiled code. To clean the generated libraries, run:

```
sh> make hw-clean
```

Once your code compiles correctly, you can proceed with the simulation. Since the testbench does not automatically check if the outputs are correct, you'll need to look at the waveforms with the QuestaSim GUI.

Student Task 20 (Simulate your FIR datapath): To launch the simulation in GUI mode, hit the following command:

```
sh> make run TESTBENCH=tb_fir_datapath gui=1
```

Then, add the useful waveforms you want to monitor and enter the following command from QuestaSim terminal:

```
vsim> run -all
```

Look at the generated waveforms and check that the behavior of your datapath reflects the specifications.

Note 19: Click on No when asked to Finish Vsim.

Student Task 21 (Analyzing the Result): You will observe in the Questasim Transcript "ERROR: data mismatch expected = —-, actual = —-". Advance the simulation by hitting `run -a`. Can you guess the reason behind this mismatch?
Hint: Observe the actual and expected values and make an educated guess.

Student Task 22 (Correcting the Parameter): Go to the golden model Jupyter Notebook, extract the correct right shift value, and update the **`parameter`** `RIGHT_SHIFT` with the correct value in `tb/tb_fir_datapath.sv`. Then recompile and run the simulation using the steps from Task 19 and Task 20. You should not see any further mismatches.

The filter-coefficients `h` are fixed, whereas the input `x` is varying over time. Our current datapath is implementing the FIR as a data-flow graph in which the `h` are streamed in parallel continuously. Thanks to the streaming assumption, we do not care where they come from – memory, I/O, another buffer? If we consider that 1) the taps are fixed, 2) streaming in parallel from memory would require a huge bandwidth, the latter option (i.e., a buffer) is evidently the most effective one. In the FIR HWPE, this role is taken by the *tap buffer*, a module that first streams in the taps serially (at the start of the computation), then provides them in parallel to the *datapath*. In the next step we simulate the design with the h values coming from the `fir_tap_buffer` located in `fir-hwpe/rtl` folder.

Student Task 23 (Compile and Simulate datapath with tap buffer): Have a look at the `fir-hwpe/tb/tb_fir_buffer_datapath.sv`.

1. Extend the block diagram Task-16.3, and add relevant connections to the tap buffer.

   Datapath with Tap Buffer

2. From the `fir-hwpe/` directory, enter:

   ```
   sh>  make hw-all run TESTBENCH=tb_fir_buffer_datapath gui=1
   ```

   The above test should also pass without any error.

4.3.3 Compiling the top module

In the above tasks, the control functionality is managed by the testbench. However, in a real-life scenario, you would want a host, such as a RISC-V core, to program the HWPE, allowing it to autonomously execute tasks and notify the RISC-V core upon completion. In the next task, we will simulate such a design. A block diagram of the HWPE is shown in Figure 16. Compared to the previous design, the `FIR_TOP` now includes:

- `FIR_CTRL`: This module contains the configuration registers that can be programmed by the host through the `periph` interface. It generates control signals for the streamer and datapath, and, using status flags and an internal FSM, ensures different execution stages.

- `FIR_STREAMER`: This module interacts with memory via the `tcdm` port. Guided by control signals from the `FIR_CTRL`, the `FIR_STREAMER` demultiplexes among `h_o`, `x_o`, and `y_i` to interact with memory as dictated by the `ctrl_i` signal.

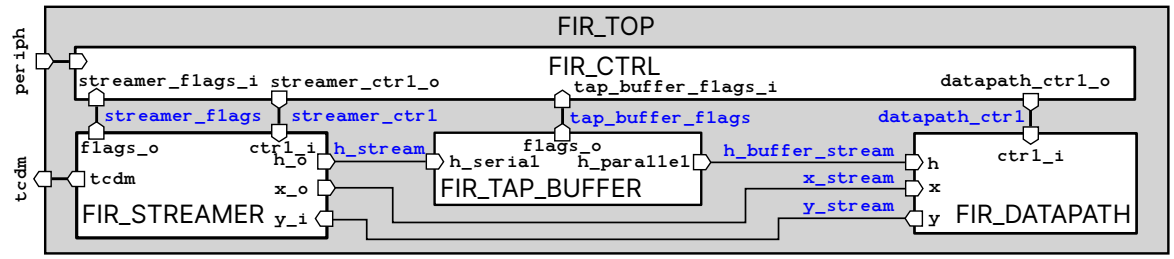

Figure 16: Block diagram of `fir_top`. Commonly used signals are removed for cleanliness.

Student Task 24 (Complete the missing connection): Open the `rtl/fir_top.sv` file.

1. There are missing connections for the streamer, control signals, and status flags. Follow

the inline instructions to make these connections.

2. There are missing connections for the data from the streamer. Route the data to the appropriate modules.

Hint: Refer to Figure 16 for the appropriate signal names.

After connecting the appropriate data signals, let's verify our design.

Student Task 25 (Familiarize with the top testbench module): Examine the testbench file located at `fir-hwpe/tb/tb_fir_top.sv`.

Unlike the previous testbench, this one includes an instance of the RISC-V core `zeroriscy_core`. The data port of the `zeroriscy_core` is connected to the `periph` port of the `fir_top_wrap` module. In this task, we will use the software infrastructure to simulate the design. The `sw` folder contains the relevant files. The main kernel to be executed is located in `sw/tb_fir.c`. Additionally, there is a `hal_fir.h` file containing information about the relevant configuration registers for the HWPE.

Student Task 26 (Generate the stimuli): Follow these steps to generate the stimuli. From the `fir-hwpe/` directory, execute:

```
sh>  riscv -pulp-gcc-1.0.16 make sw-all
```

This will generate two stimulus files, `stim_instr.txt` and `stim_data.txt`, in the `sim` folder.

Note 20: The `riscv -pulp-gcc-1.0.16` is necessary as the HWPE testbench used here is not yet compatible with a recent LLVM-based RISC-V compiler.

The `stim_instr.txt` file contains the instructions to be executed by the RISC-V core, while the `stim_data.txt` file contains the relevant stimuli needed for the HWPE, including the stimuli we copied to the `sw/inc` folder in Task 17. Next, we will move towards simulating the entire design.

Student Task 27 (Compile and simulate the top module): To compile and simulate the design, from the `fir-hwpe/` directory, execute:

```
sh>  make hw-all run TESTBENCH=tb_fir_top gui=1
```

```
vsim>  run -all
```

What do you observe? Is the simulation coming to an end? Inspect the `state_q` signal from the `fir-hwpe` controller by adding it to the waveform. You can also use the following command:

```
vsim>  add wave -position insertpoint sim:/tb_fir_top/\
          i_hwpe_top_wrap/i_fir_top/i_ctrl/state_q
```

Can you determine what is happening here?

There seems to be an issue with the Control Finite State Machine (FSM) as it is not coming to an end. The FSM transitions through the `FSM_IDLE`, `FSM_TAP_BUFFER`, and `FSM_COMPUTE\` states. The FSM switches from `FSM_IDLE` to `FSM_TAP_BUFFER` to load the filter coefficients into the `FILTER_TAP_BUFFER` module. Once all the coefficients are loaded, the FSM switches to the `FSM_COMPUTE` state, where the streamer loads input data `x`, computes the output `y` and saves to the memory through `tcdm`. After writing all the outputs, the FSM should return to `FSM_IDLE`. However, the FSM is not transitioning back to the `FSM_IDLE` state. In the next task, we will fix this issue.

Student Task 28 (Fix the problem and re-run the simulation):

- Open the `rtl/fir_ctrl.sv` file. You will find an FSM named `main_fsm_comb`. Complete the FSM by following the inline instructions.
  Hint: Use `done` signal of `y_sink_flags` field of `streamer_flags_i`
- Re-run the simulation by compiling the source code as done in Task 27.

The simulation should not result in an infinite loop anymore. But are the outputs correct?
Hint: Look at the transcript of the vsim window.

Observe the simulation output. Do you see an issue similar to the one in Task 21? Can you determine where the `right_shift` value is coming from? Hint: Check the `sw/tb_fir.c` file.

Student Task 29 (Fix the right shift value and rerun the simulation): Fix the right shift value in the register configuration in the `sw/tb_fir.c` file. Regenerate the stimuli with the corrected application by executing:

```
sh> riscv-pulp-gcc-1.0.16 make sw-all
```

Then rerun the simulation using:

```
sh> make hw-all run TESTBENCH=tb_fir_top gui=1
```

You should not see any more errors. Voila! Now you have a standalone working HWPE!

4.3.4 Integration with pulpissimo

In the previous task we implemented parts of the HWPE and verified it standalone, and it is time to integrate it with the PULPISSIMO infrastructure.

Student Task 30 (Enable HWPE in Pulpissimo simulation): Set the `USE_HWPE` parameter to 1 in the `pulpissimo` instantiation located in `pulpissimo/target/sim/tb/tb_pulp.sv`

In the successive tasks, we will make changes to the `pulp_soc` module.

Student Task 31 (Instantiate FIR-HWPE in the `pulp_soc`): Open the `fc_hwpe.sv` file located in `pulpissimo/working_dir/pulp_soc/rtl/fc` and add the instantiation of the FIR-HWPE in the placeholder provided.
Hint: You can take inspiration from the FIR-HWPE instantiation available in the `fir-hwpe/tb/tb_fir_top.sv` file. Assign `.ID ( ID_WIDTH )` and keep other parameters unchanged.

Note 21: Watch out the signal names. The `wen` must be negated in the port map.

Student Task 32 (Add FIR-HWPE to the memory map): The peripheral memory maps are described in `pulpissimo/working_dir/pulp_soc/rtl/pulp_soc/soc_peripherals.sv`. To add a HWPE, we need to make the following changes:

- Increase the number of APB slaves to 11 by adjusting the `NumAPBSlaves` parameter.
- In the `APBAddrRanges`, add an entry at index 10 for the HWPE. The start and end address values are defined in `pulpissimo/hw/includes/soc_mem_map.svh`.
  Hint: For syntax and define names, refer to an existing entry such as `SOC_MEM_MAP_GPIO_START_ADDR`.
- Create an APB slave interface named `hwpe` using the `SOC_PERIPHERALS_CREATE_SLAVE` define. This creates an `APB` interface with name `s_hwpe_slave`.
- Assign `s_hwpe_slave` to `apb_hwpe_master` propagating outside.
  Hint: You can use the `APB_ASSIGN` define.
- Compile the pulpissimo system

```
make build
```

This concludes the integration of the HWPE into the Pulpissimo infrastructure. Now it's time to verify the integration.

Student Task 33 (Verify Integration): We will use the `pulpissimo/sw/efcl2026/taskY_hwpe_fir_filter` directory for verification. We will follow the steps below to complete the software testing for the HWPE integration:

1. Copy the contents from `pulpissimo/working_dir/fir-hwpe/sw/inc/` to `pulpissimo/sw/efcl2026/taskY_hwpe_fir_filter/include`. These files are the stimuli headers generated by the golden Python model.
2. Then copy the test "fir_test.c" from Task5:

```
sh>  cd sw/efcl2026
sh>  cp task5_baseline_fir_filter/src/fir_test.c \
        taskY_hwpe_fir_filter/src
```

3. Examine the `pulpissimo/sw/efcl2026/taskY_hwpe_fir_filter/src/fir.c` file. What similarities and differences do you notice compared to `pulpissimo/sw/efcl2026/task5_baseline_fir_filter/src/fir.c`?

The `fir16` function signatures is the same, but the implementation differs. The implementation of HWPE involves configuring the respective registers and depends on the `include/fir.h` file. There are placeholders in this file that need to be completed.

Student Task 34 (Complete the hardware abstraction layer): Open the `pulpissimo/sw/efcl2026/taskY_hwpe_fir_filter/include/fir.h` file.

1. Update the `FIR_ADDR_BASE` and `FIR_ADDR_SPACE` with the correct addresses.

Hint: You can refer to the address values used in Task 32, located in `pulpissimo/hw/includes/soc_mem_map.svh`.

2. Complete the drivers for setting the `x`, `y`, and `h` addresses.
   Hint: You can use the `HWPE_WRITE` *defines.*
3. Implement the function to update the `FIR_REG_SHIFT_LENGTH` register.
   Hint: The length occupies the 16 MSB bits, and the shift occupies the 5 LSB bits.

Now let's move to the fir.c source file.

Student Task 35 (Fix the simulation source files):

1. Open `taskY_hwpe_fir_filter/src/fir.c` and fill the placeholders using the appropriate function arguments.
2. Next we move to the function instantiation in `fir_test.c` file. Modify the function call to `fir16` in `test_run()` with appropriate input.
3. Include `common.h` in the stimuli headers x, y, h in the include folder.
4. Add `__sram` attribute to the `x_stim`, `h_stim`, `y_stim` arrays.
   Hint: the `h_stim32.h` *looks like the following*:

```
#include <common.h> __attribute__((aligned(16)))uint32_t h_stim[] \
__sram=
```

Student Task 36 (Test the simulation): In the `taskY_hwpe_fir_filter` directory, execute the following command

```
sh>  make all run gui=1
```

Then run the simulation till the end. You should see the simulation end without errors.

You are done with this exercise.
Discuss your results and any questions you might have with an instructor.

4.4 Implementing on FPGA

In this last hands-on session, we will implement our PULPissimo design on a Field-Programmable Gate Array (FPGA) and test the software application.

4.4.1 FPGA implementation

FPGA support is fortunately already set up for us in the PULPissimo repository. While a variety of FPGAs are supported, we will be using the ZyboZ7 FPGA. As with the simulation, bender will set up the scripts for us, and the make commands are already set up to run FPGA elaboration, synthesis, and implementation in Vivado.

If you want to know more about how the design is set up for the ZyboZ7, feel free to check out `target/fpga/pulpissimo-zyboz7`, where the `rtl/xilinx_pulpissimo.v` and `constraints/zyboz7.xdc` will likely be the most interesting.

Student Task 37 (Bitstream generation): Use PULPissimo's make targets to implement your design for the ZyboZ7 FPGA. From pulpissimo directory, run:

```
vitis-2022.1 make zyboz7
```

Debug any errors that occur during parsing, synthesis, and implementation.

If you prefer, you can use the Graphical User Interface (GUI) visualization by navigating to `target/fpga/pulpissimo-zyboz7` and running:

```
vitis-2022.1 make gui
```

If you don't know how to fix an error, ask an assistant for help.

4.4.2 Debugging Software on FPGA

To test PULPissimo on FPGA, the design's JTAG pins and some GPIOs configured for UART are exposed on the PMOD JE of the ZyboZ7 FPGA, shown in Figure 17. To test the design, we will need to program the FPGA, and connect a debugger and UART to USB converter to the design. For debugging, we use a Digilent HS2 JTAG programmer, which contains an FTDI chip inside. The UART-USB converter also contains an FTDI chip, with the datasheet explaining the connections as shown in Figure 18.

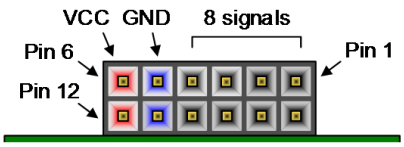

Figure 17: Pin description of the PMOD header

The pinout for the different FPGA configurations can be checked in the corresponding *constraints* directory. The pinout on the ZyboZ7 JE pin header is as described in Table 1.

Pin number	Function
Pin 1	JTAG TMS
Pin 2	JTAG TDI
Pin 3	JTAG TDO
Pin 4	JTAG TCK
Pin 7	UART RX (GPIO 1) (connect to FTDI UART TX)
Pin 8	UART TX (GPIO 0) (connect to FTDI UART RX)
Pin 9	GPIO 3
Pin 10	GPIO 2

Table 1: ZyboZ7 PMOD JE pinout (see Figure 17)

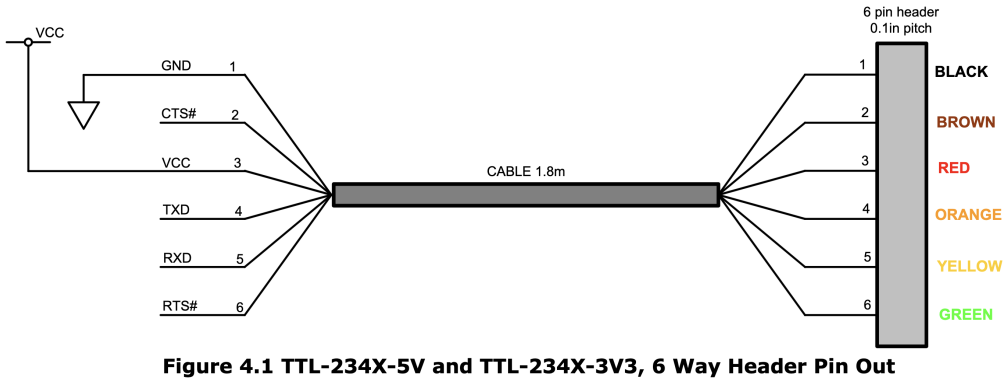

Figure 18: USB UART pinout

Student Task 38 (Flash FPGA): While the implementation is running, collect the devices you need for FPGA programming. Once the bitstream is generated, flash the FPGA and connect the debugger with the following steps.

- Connect the FPGA USB connector to your workstation with a USB cable (don't connect the other cables yet to ensure proper detection of the FPGA).
- Once the FPGA is connected, flash the PULPissimo design bitstream onto the FPGA. In the PULPissimo root directory, run

```
vitis-2022.1 make -C target/fpga/pulpissimo-zyboz7 download
```

  Or navigate to the `target/fpga/pulpissimo-zyboz7` directory and execute

```
vitis-2022.1 make download
```

- After this you need to kill the process "hw_server" created by Vivado when loading the bitstream into the FPGA. You can find the process ID with the following command and then kill it

```
> ps aux | grep hw_server
> kill -9 ID
```

- Connect the digilent HS2 debugger to the PMOD JE pin header. Consult the above pinout description and the labels on the debugger for reference.
- Launch the openOCD debugger in a new Terminal window connected to PULPissimo and check if the device is accessible. Then run

```
openocd -f target/fpga/pulpissimo-zyboz7/openocd-zyboz7-\
   digilent-jtag-hs2.cfg
```

As you have several software tests implemented now, testing basic functionality and the various additions we made in the course, let's test these on the FPGA as well.

To properly debug the device we connect OpenOCD (Open On-Chip Debugger) to the debug module. OpenOCD is a software designed to interface directly with debug modules on a variety of System-on-Chips (SoCs) with a variety of debug adapters, and in our case, it directly supports RISC-V and the FTDI chip in the Digilent HS2.

To program the chip, we use GNU Debugger (GDB). This tool allows us to read and program a binary, and, if needed, also to step through the individual instructions.

Student Task 39 (Debugging Software): Let's start by programming our emulated PULPissimo with the simple hello-world program in `task2_hello`.

- Connect the USB to UART cable to the PMOD JE pin header on the FPGA, following the pinout described above.
- Open a serial connection to the UART pins of the PULPissimo on the FPGA in a new Terminal window by connecting to the proper device. Feel free to check which device is used for the USB UART serial device (likely `/dev/ttyUSB0`) by listing all serial devices (`ls -l /dev/serial/by-id`) and plugging in and out the UART device. Use the following command to open a serial connection (ensure the correct device)

```
screen /dev/serial/by-id/usb-FTDI_TTL_... 115200
```

  To close GNU Screen again, use C-a k.
- To run the application on FPGA, make sure you are compiling for FPGA. To do this, append `target=fpga` to your `make all` command. Navigate to the `sw/efcl2026/task2_hello` directory and execute this command (make sure in your terminal you have the setup sourced in Task 2):

```
make clean all platform=fpga
```

- To program you application on PULPissimo, launch a RISC-V GDB with your application using the following commands to create a python environment:

```
> conda create -n py27lib -y python=2.7
> conda activate py27lib
```

  Then run these commands to move to the application folder (e.g. task2_hello) and launch GDB:

```
> cd sw/efcl2026/task2_hello
> env LD_LIBRARY_PATH="$CONDA_PREFIX/lib:$LD_LIBRARY_PATH" \
   riscv riscv32-unknown-elf-gdb build/test/test
```

- In GDB, connect to the target (ensure the port matches what is reported by OpenOCD):

```
target extended-remote :3333
```

- Halt the processor core, program the application, and resume:

```
> monitor reset halt
> load
> continue
```

- Check the serial output window for your expected output.
- How does the runtime compare to simulation? Is this expected?
- Try running other applications on the FPGA following the same steps as above.

Student Task 40 (Optional extensions):

- Design further software tests to run on the FPGA with the custom hardware we developed - any other application?
- Extend the custom hardware extensions

$\mathcal{E}$ **You are done with this exercise.**
Discuss your results and any questions you might have with an instructor. $\mathcal{E}$

Acronyms

CLI	Command Line Interface. 7
DSP	Digital Signal Processing. 15
EX stage	Execution Stage. 23
FC	Fabric Controller. 12
FIR	Finite Impulse Response. 2, 13, 15, 26, 28
FPGA	Field-Programmable Gate Array. 3, 35–38
FSM	Finite State Machine. 32
GDB	GNU Debugger. 37, 38
GUI	Graphical User Interface. 8, 11, 35
HDL	Hardware Description Language. 12
HWPE	Hardware Processing Engine. 2, 26, 27, 31–33
ID stage	Instruction Decode stage. 22
IP	Intellectual Property. 2, 3, 5, 12, 17, 27
ISA	Instruction Set Architecture. 6, 15, 26
PC	Program Counter. 8
RTL	Register Transfer Level. 5, 12
SoC	System-on-Chip. 12, 14, 37
WB stage	Write-Back Stage. 24
XIF	X-Interface. 2, 15, 18
XIFU	X-Interface Functional Unit. 15, 16, 20