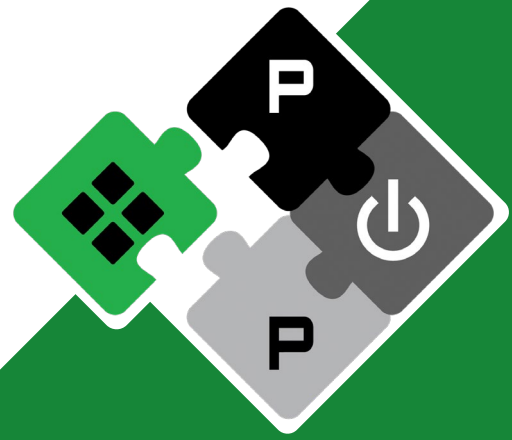
ETH zürich

ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

Heterogeneous FPGA acceleration using Xilinx Vitis Development Environment

Energy Efficient Parallel Computing Systems for Data Analytics

Arpan Suravi Prasad prasadar@iis.ee.ethz.ch
Federico Villani villanif@iis.ee.ethz.ch



Integrated Systems Laboratory (ETH Zürich)

PULP Platform

Open Source Hardware, the way it should be!

@pulp_platform

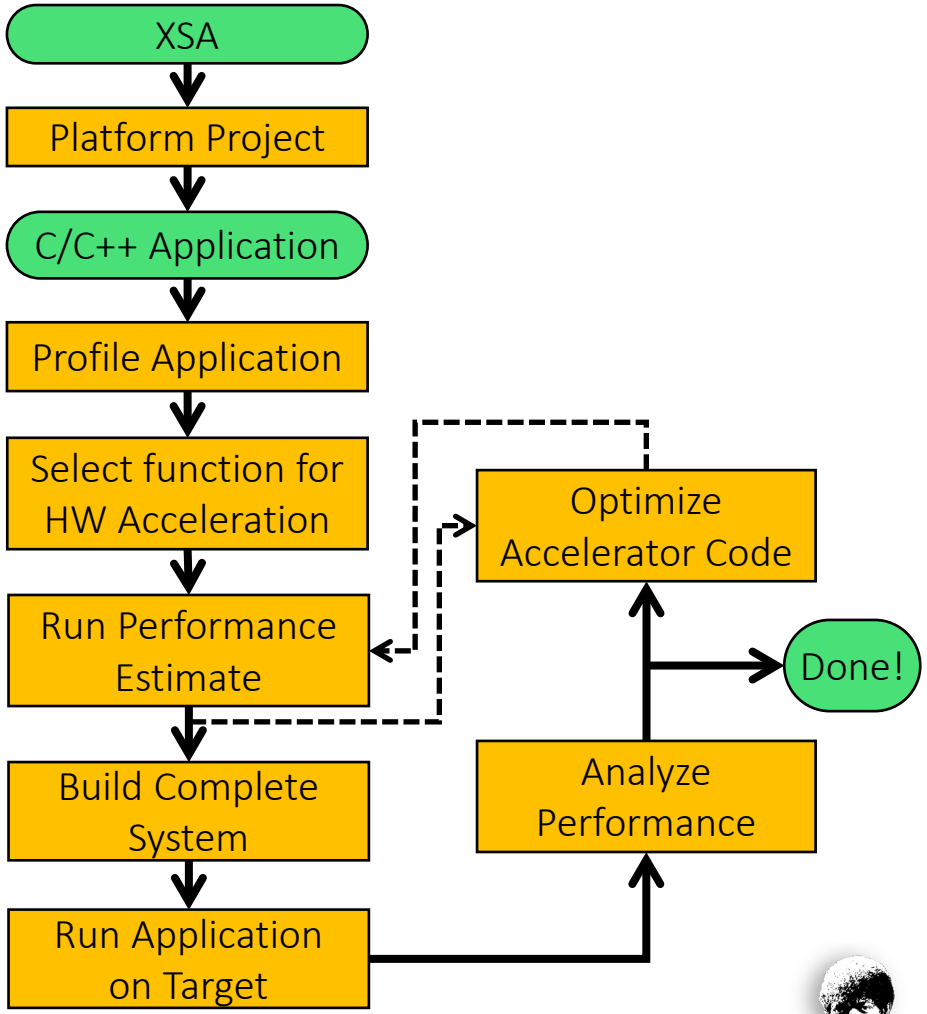
pulp-platform.org

youtube.com/pulp_platform

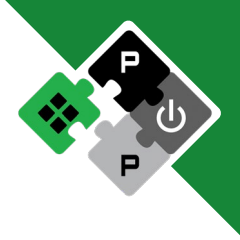
What will you do in this Exercise ?

Work through an entire system design:

- Start with a Matlab implementation of the algorithm
- Obtain a C++ implementation (simplifications, fixed-point)
- Move (partial) implementation to embedded platform (ZedBoard)
- Profile the performance of the embedded system
- Decide on what to accelerate & estimate possible speedups
- Implement and optimize the hardware accelerator
- Measure the final system performance



What should you Learn in this Exercise ?



- ❑ Learn:
  - ➢ How applications can be accelerated on heterogeneous FPGA
  - ➢ How you can tackle a system design problem systematically:
    - o Profile → Estimate → Decide
  - ➢ What are possible pitfall and bottlenecks

- ❑ Experience a very advanced high-level design flow
  - ➢ Xilinx Vitis Flow
  - ➢ Get a glimpse of how such an advanced flow works

A heterogeneous system design is cool since it is very multidisciplinary!

Installation

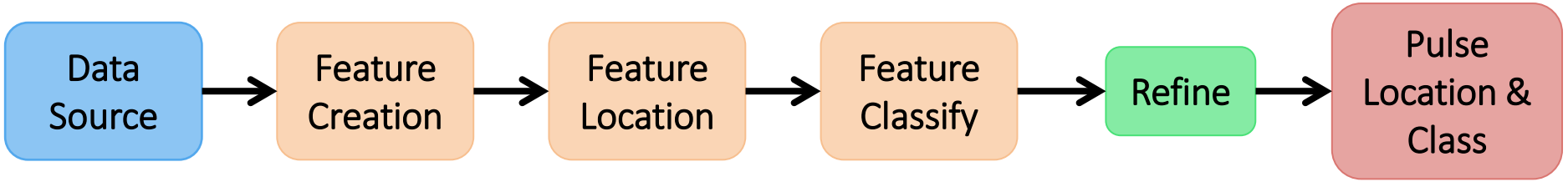
- ❑ Run install script
  - ➢ Run the command: `/home/soc_master/ex_hetero_vitis/install.sh`
  - ➢ It copies all required files to the local scratch: `/scratch/ex_hetero_vitis_$USER`
  - ➢ Please delete the created folder again after the exercise!!
- ❑ Switch to the exercise directory
  - ➢ `cd /scratch/ex_hetero_vitis_$USER`
- ❑ Have a look at the folder structure:
  - ➢ `tree -L 4`
  - ➢ ./src contains the source file (c, matlab, input data, platform)
  - ➢ ./doc is where you can find various manuals & documentation
  - ➢ ./mat contains **these slides** and the **spreadsheet** you need later

Task 1 – The Algorithm

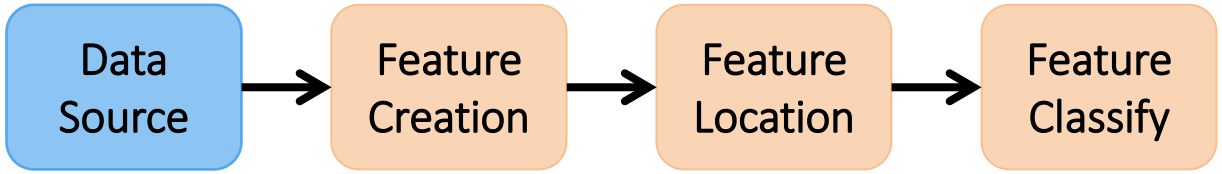
- Launch Matlab
  - `source run_matlab.sh`
  - Source files are located in src/matlab folder
- Run the `.m` script. (Right-click on `run.m` and click on **Run**)
- Function `run`
  - The script loads an ECG signal trace and detects pulses which are classified into two different types: Typ A (green), Typ B (red)
- Task:
  - Have a quick look at the 4 functions in the processing section of the run script
    Where do you expect the greatest potential for hardware acceleration?
    [only spend 2 min on this]

C++ Implementation Overview

- The algorithm has the following steps



- For the continuation of this exercise, we will only look at the first part, which is the computationally intensive part:



- We start with a pure software implementation in C++
- For this C++ application we switch to a fixed-point implementation since float/double precision is not required and would results in overly complicated HW
- The orange part will process around 22k int16 data items and output again 22k int16 values

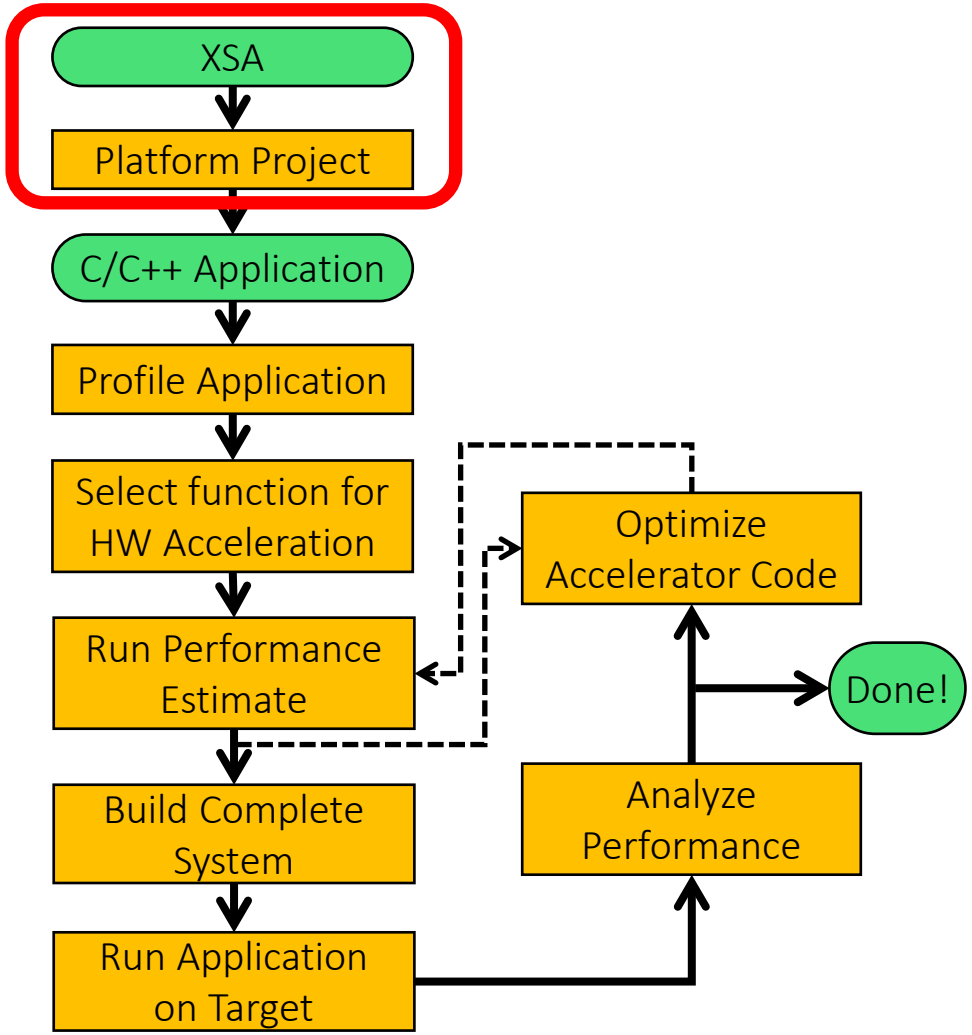
C++ Implementation

- We have already implemented the C++ code that will be executed on the ARM called c_processing.cpp in src/c folder
  - Run `verify_c.m`
  - This script runs the C++ code to verify that the latter produces the same results as the Matlab code. It also generates a couple of include files for the C++ application e.g., for the filter coefficients.

You will notice that we are tolerating some differences in the `verify_c.m` script to accommodate the fix point losses and minor implementation simplifications.

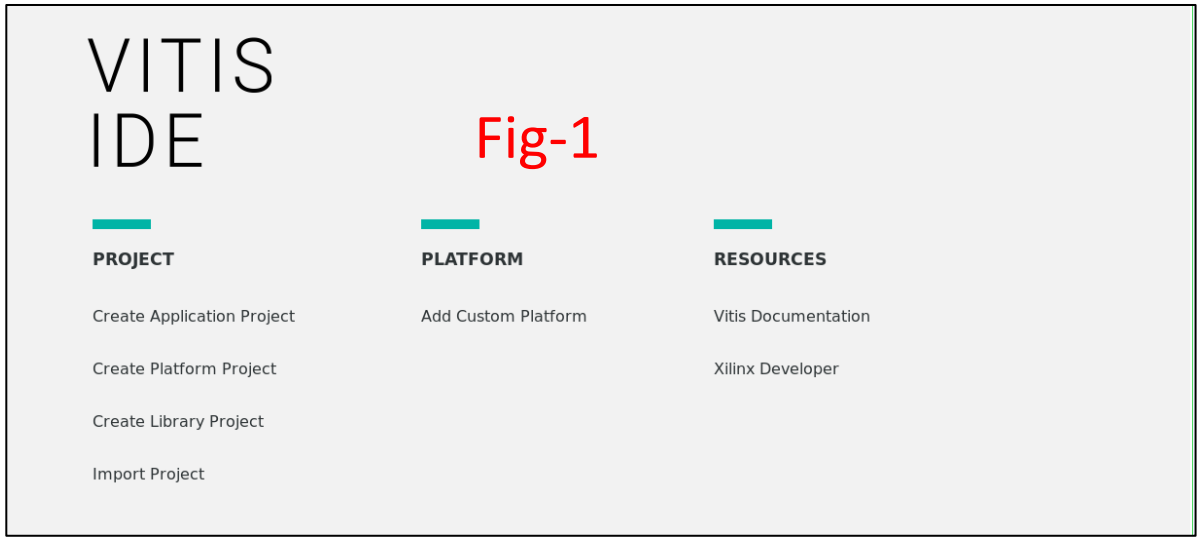
- Have a quick look at the code of `c_processing.cpp` located in the `src/c` folder

Task 2 – Preparations



Launch Vitis Workspace

- Make sure you are in **`/scratch/ex_hetero_vitis_{USER}`**
- Start the Vitis tool by following the steps below
  - Start the DZ environment **`start-dz-env`**
  - **`Singularity >`** console will appear
  - Run the command: **`vitis-2022.1 vitis –workspace ./vitis_workspace`**
  - Wait for the Vitis GUI to open(Fig-1)



Create Platform Project

To Create a Platform Project

- Click on **`File → New → Platform Project`** (Fig-1)
- A GUI to create a platform project will appear. (Fig-2)
- Give the **Platform project name** as **`zedboard_basic`** and click on **`Next`** (Fig-2)

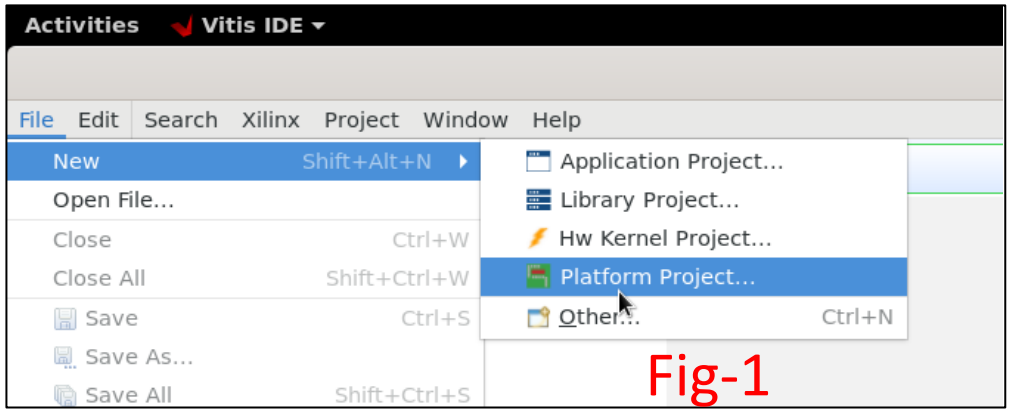


Fig-1

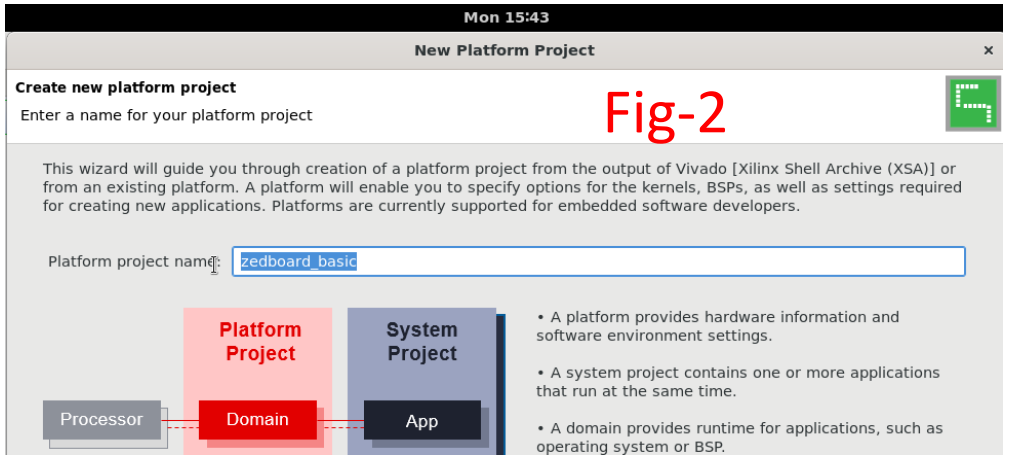


Fig-2

Create Platform Project

- In the Hardware Specification XSA location(Fig-1), add **`design_1_wrapper.xsa`** by following the below steps
  - Click on Browse. It should open the filesystem.
  - Goto **`/scratch/ex_hetero_vitis_$USER/src/platform/basic`** folder. Click on **`design_1_wrapper.xsa`** file and click on **`Finish`**
- After the platform is created, the GUI will look like Fig-2

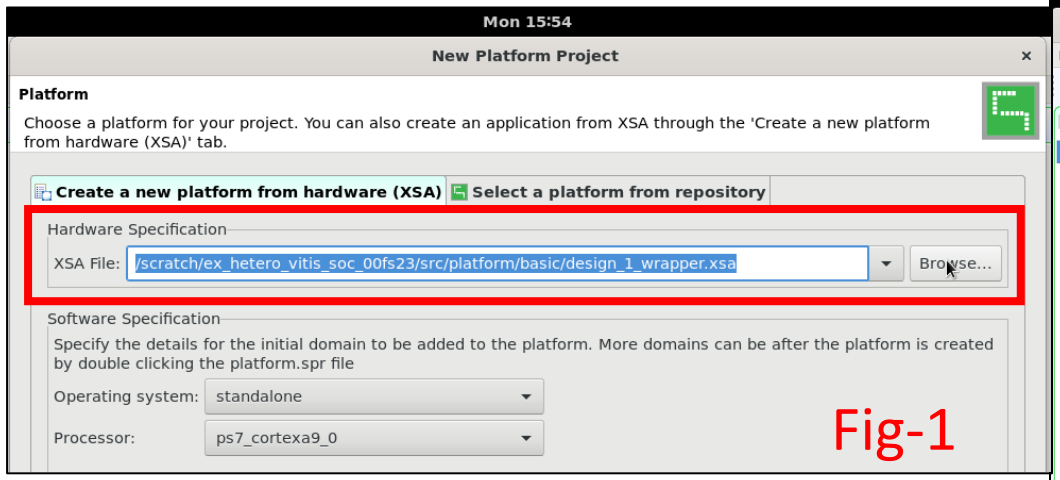


Fig-1

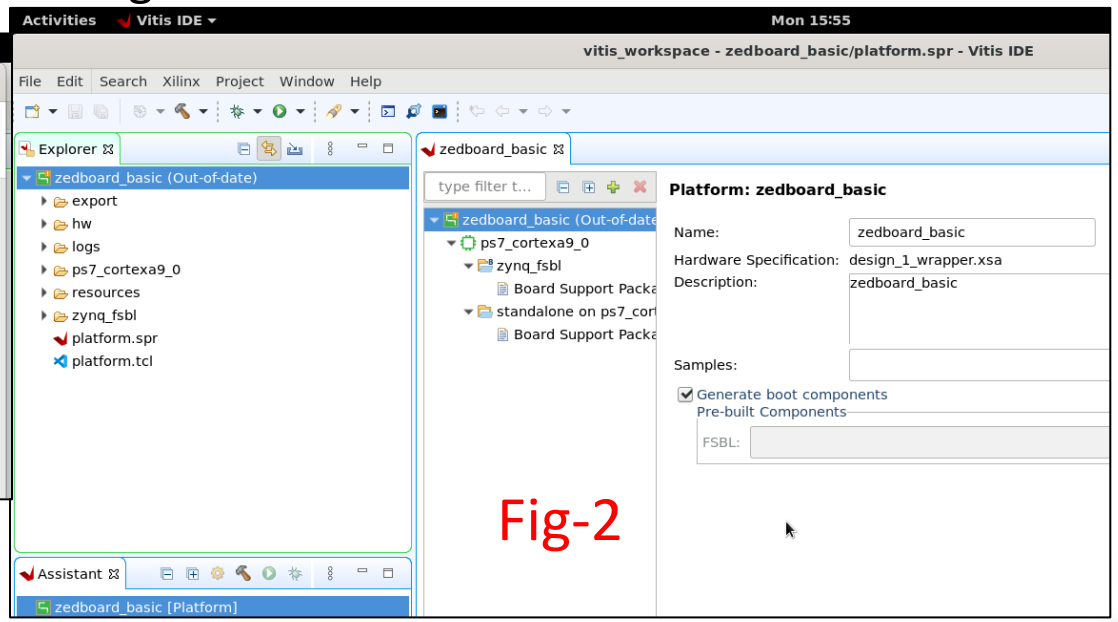
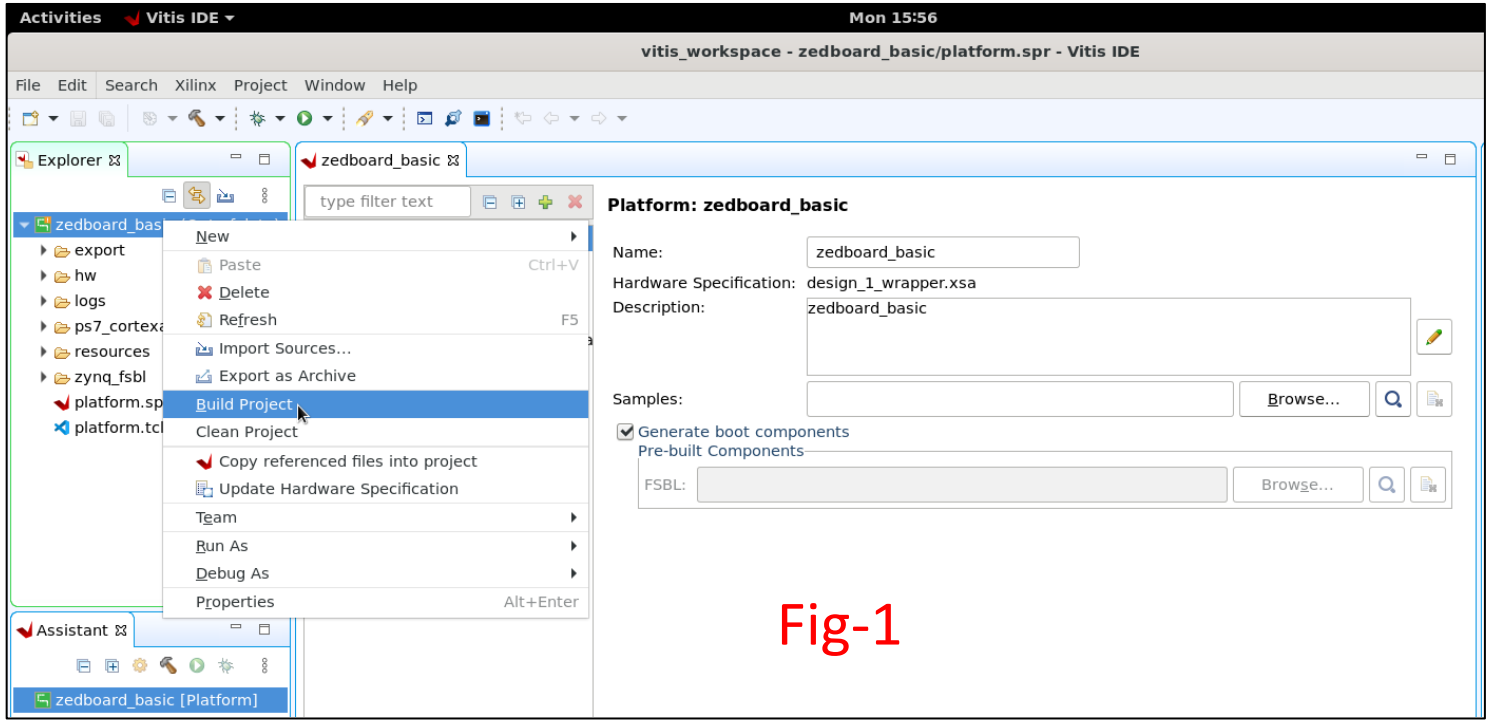


Fig-2

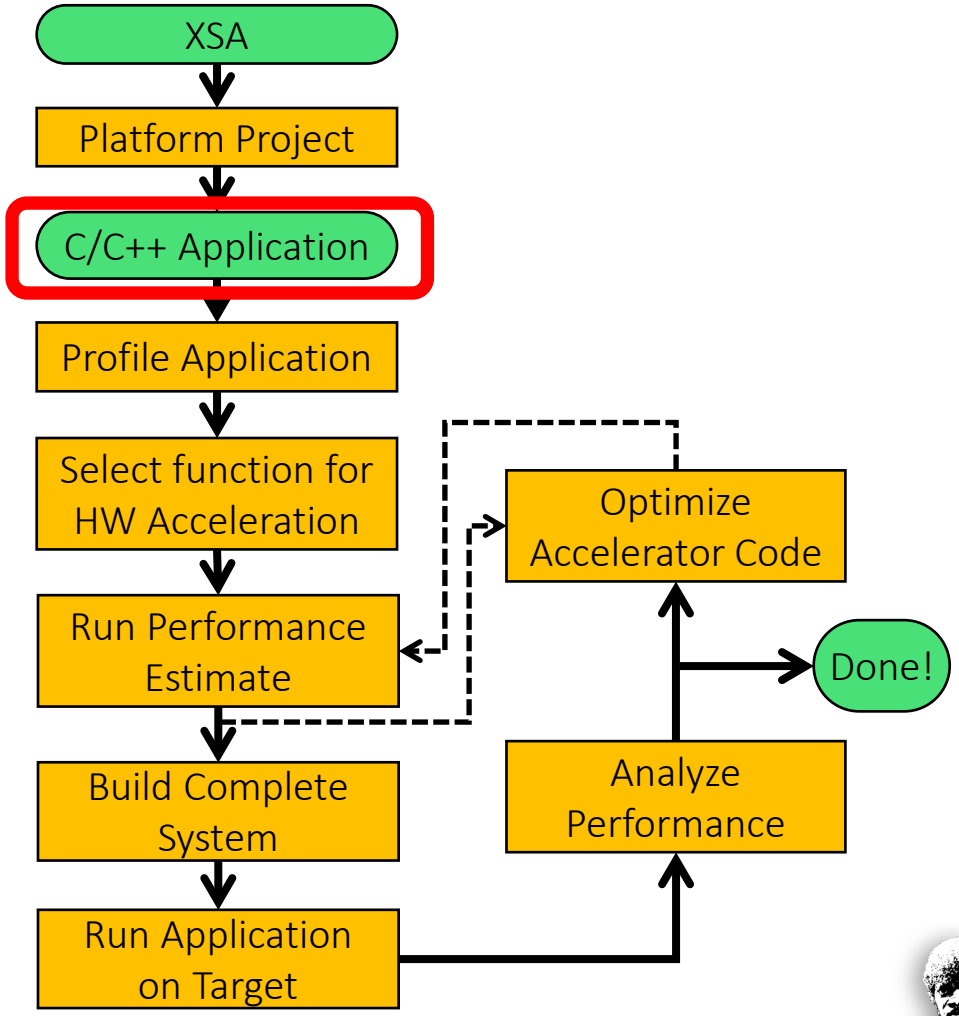
Build Platform Project

- To build the platform project
  - Right-click on **`zedboard_basic`** project → **`Build project`** (Fig-1)
  - Wait for the build to finish.



Where are we?

- We created the basic platform project.
- Platform project serves as a foundation on which other projects are built
- `zedboard_basic` platform will be fixed throughout the exercise
- Next we will create another Platform project with name `zedboard_sw` on which we will build software application. Follow the same steps to create `zedboard_sw` using `src/platform/basic/design_1_wrapper.xsa`. Also don't forget to build `zedboard_sw`
- In the following slides, we will build a project to compile C++ Application



Create application project for Zynq

To create an application project

- Click on File → **New**→**Application Project**(Fig-1)
- Select **zedboard_sw** from the list of platform(Fig-2) and click on **Next**
- Put the **Application project name** “application”. Click on **Next** → **Next** until reaching the Templates screen
- From **Templates**, select **Empty C++ Application** and click on **Finish**(Fig-3)

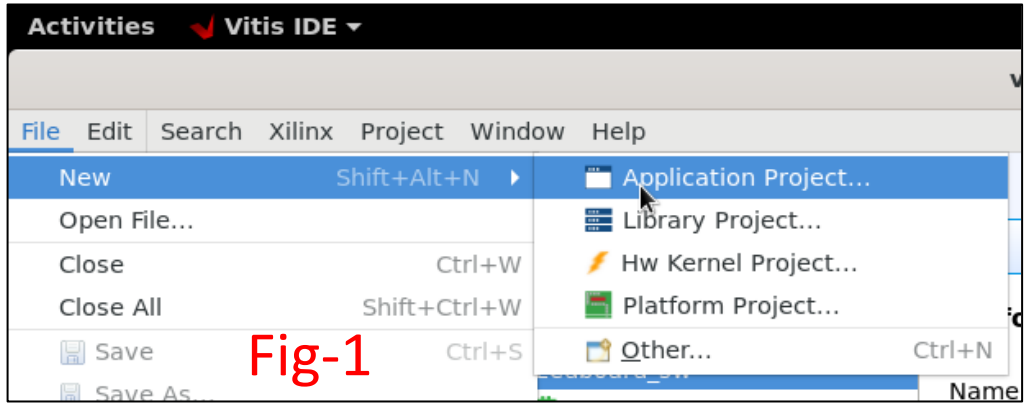


Fig-1

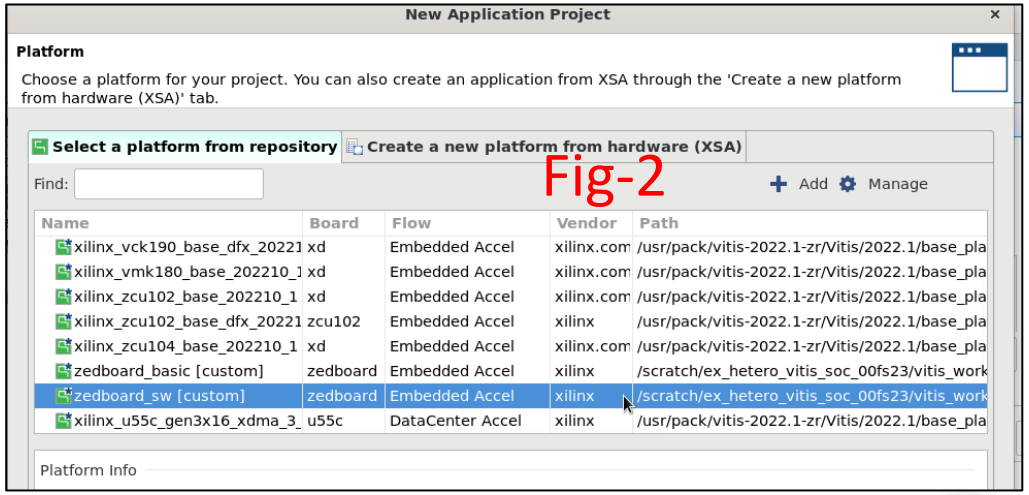


Fig-2

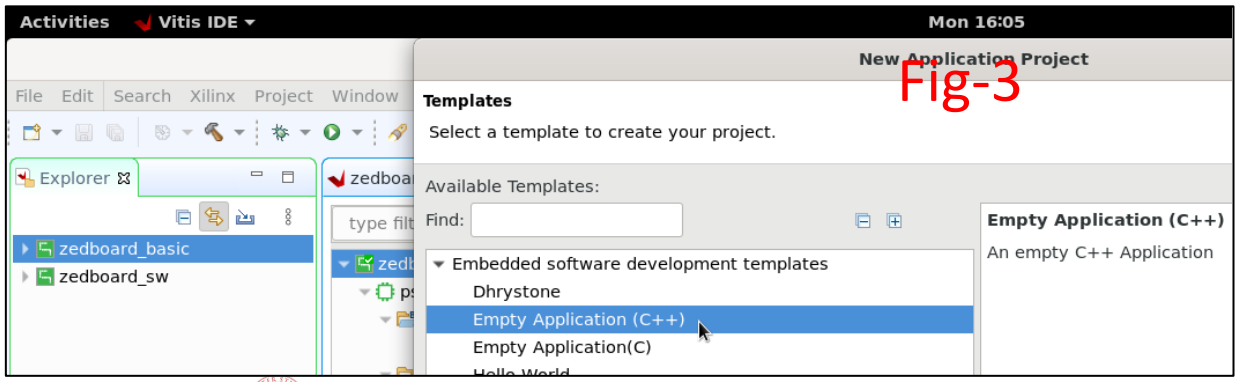
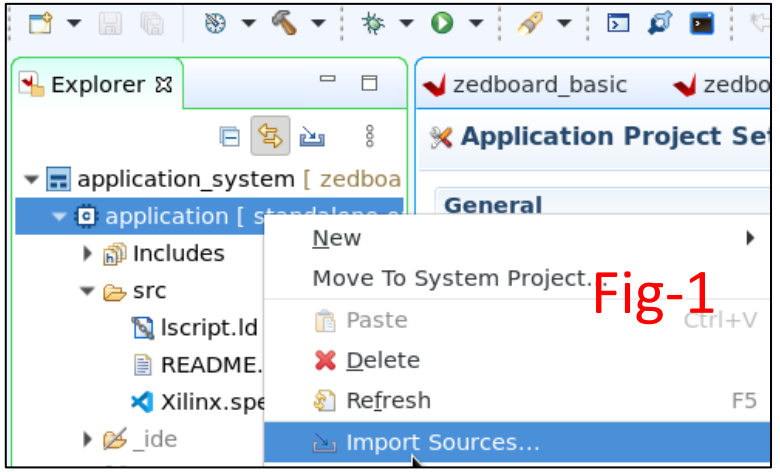


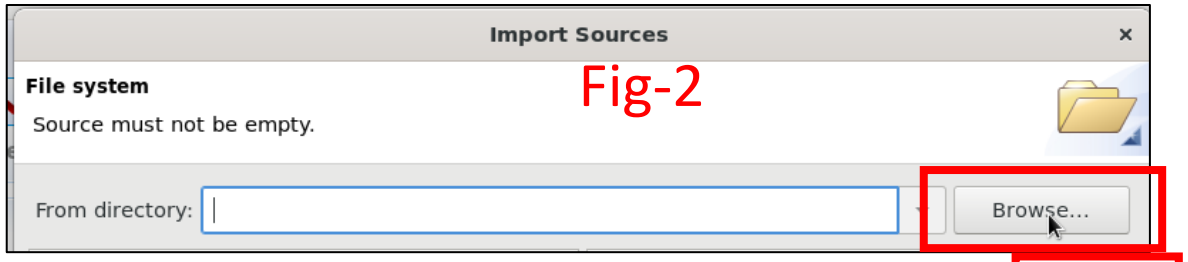
Fig-3

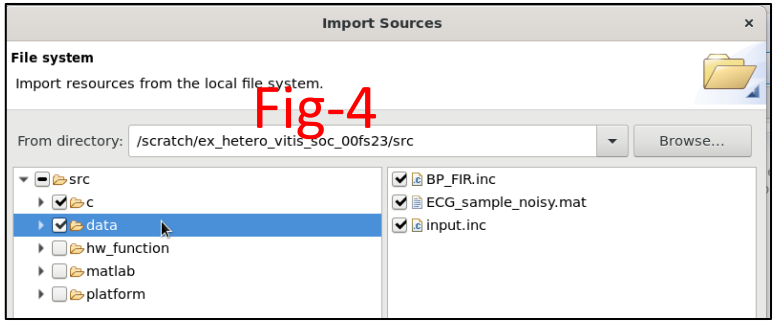
Create application project for Zynq

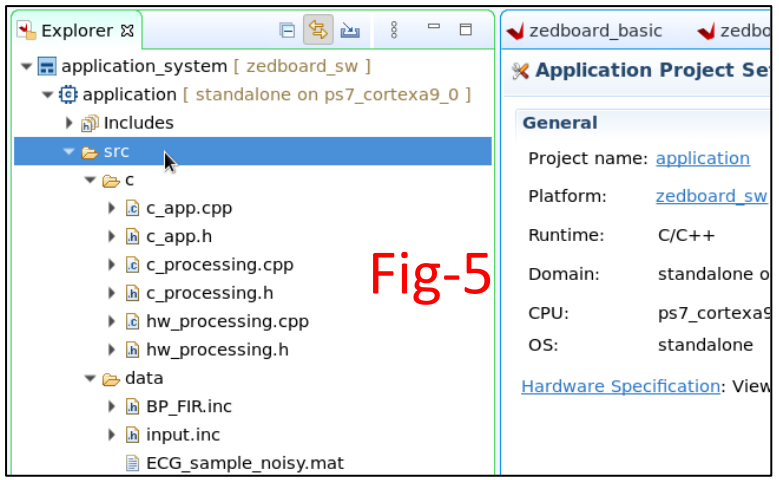
Add sources to `application` project

- Right click on "**application**" project → **Import sources** (Fig-1)
- Click on Browse (Fig-2)
- Select the src folder and click on Open (Fig-3)
- Expand the src folder(Fig-4)
  - select **`c`** and **`data`** folders → Finish. The GUI will look like Fig-5









Build application project for Zynq

- To build application project(Fig-1)
  - Right-click on the **`application → Build Project`**
- If the build is successful a binary named(Fig-2) **`application.elf`** will be produced in the **`application/Binaries`** folder
  - **`application.elf`** is the executable to run on ARM. Until now we have not mapped anything to the hardware.

Fig-1

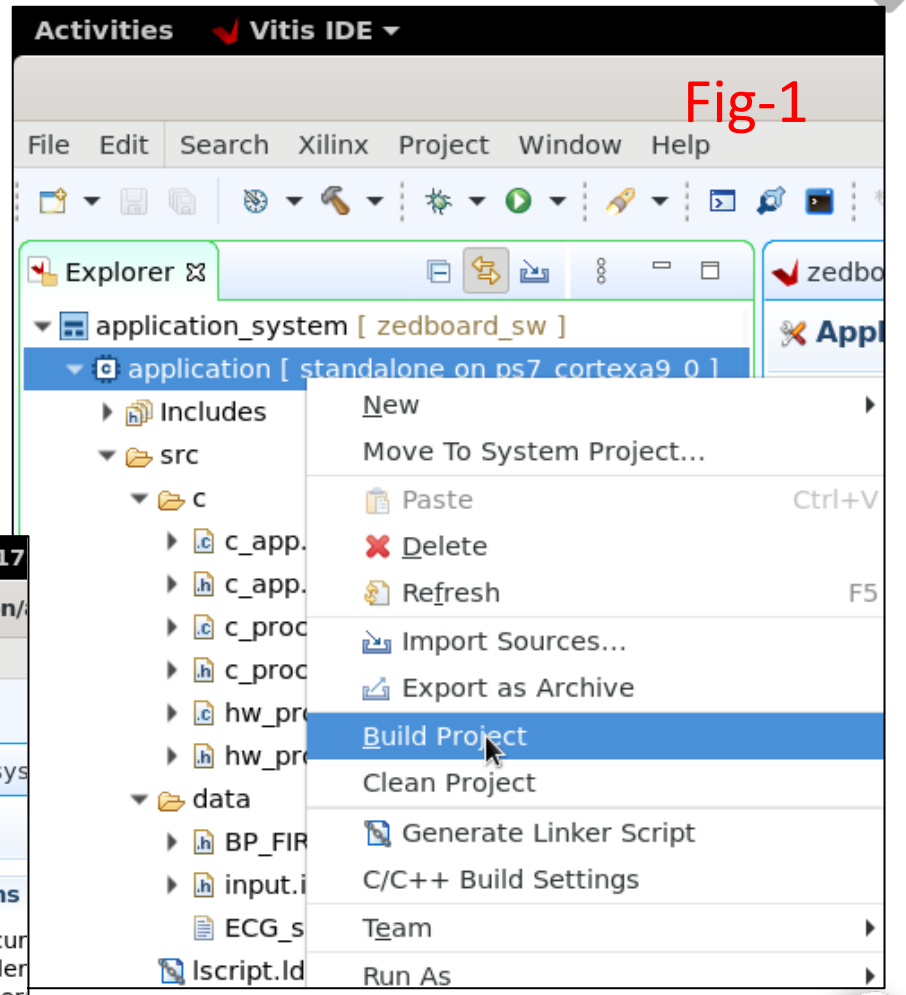
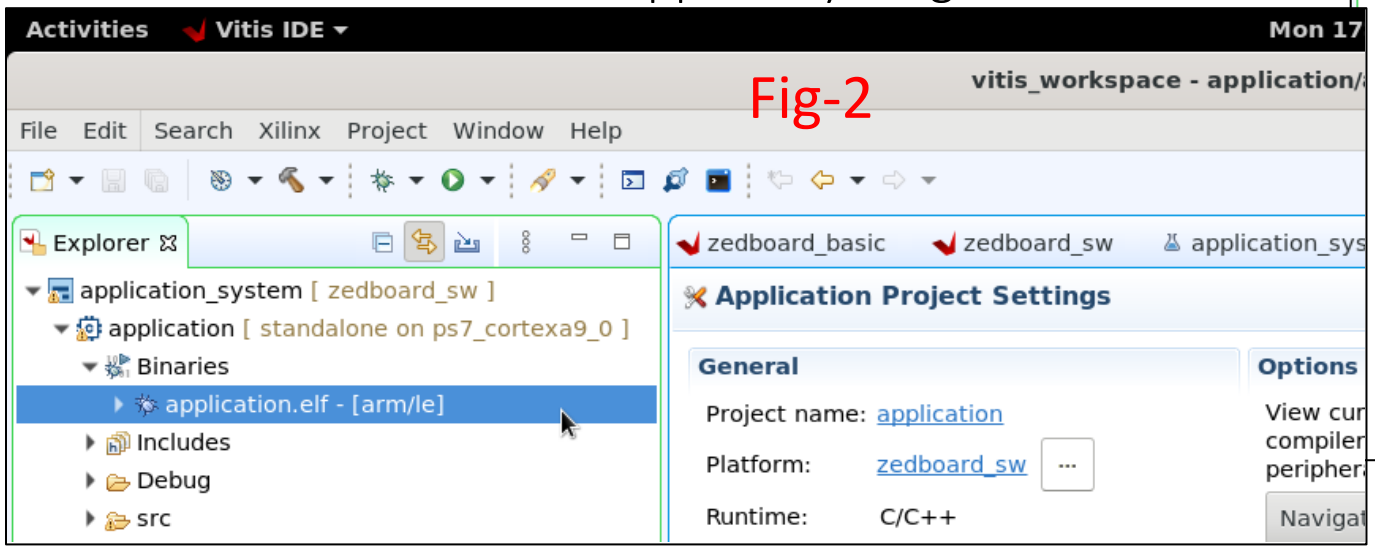
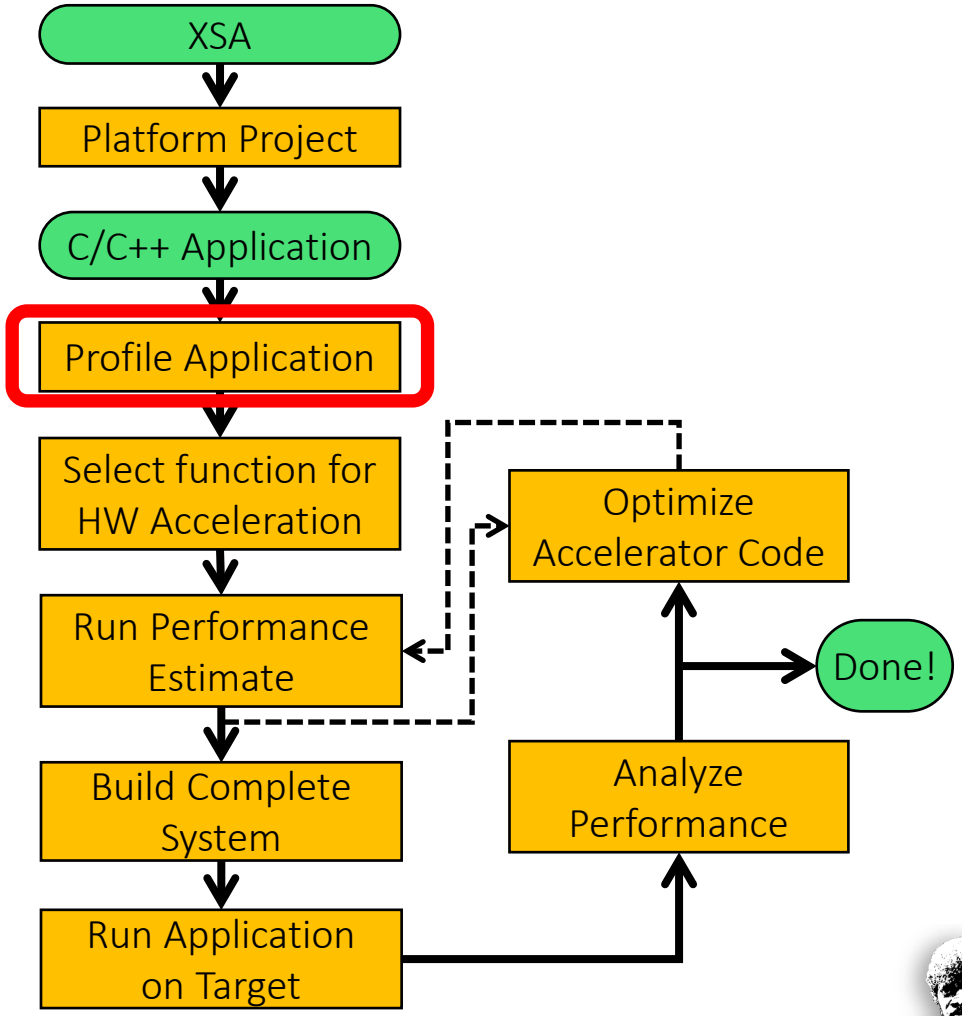


Fig-2

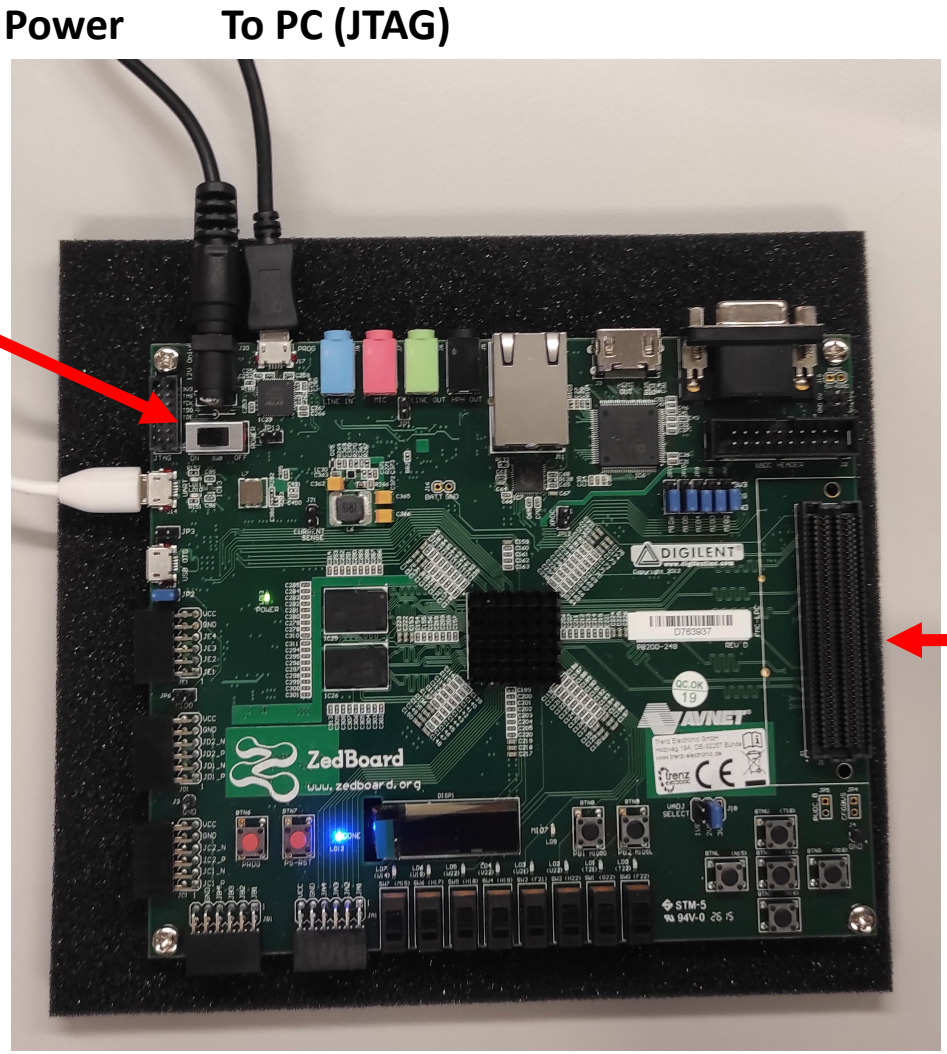


Where are we?

- We created software only application i.e.; **everything to be executed on the ARM A9 processor**
- Next task is to profile the application to extract the bottlenecks in the computation pipeline
- We will run the binary `application.elf` on the Zedboard and capture the time taken per sub-function. We will use Xilinx provided TCF profiler for this purpose
- Before we move to execute the binary, we have to setup the hardware which is explained in the following slides



Setup the Zedboard



Open UART Terminal

- **Ensure the board is on and connected**
- In the Vitis GUI, click on **`Window`** → **`Show view`**(Fig-1)
- Search for **`Serial`** in the search bar of Show View(Fig-2). Click on "**`Vitis Serial Terminal`**" and Click on **`Open`**
- The **`Vitis Serial Terminal`** will look like Fig-3
- Click on the **"+"** sign on **`Vitis Serial Terminal`** (Fig-3), select the **`Port`**(Fig-4) with the ACM prefix, and click **`OK`**.

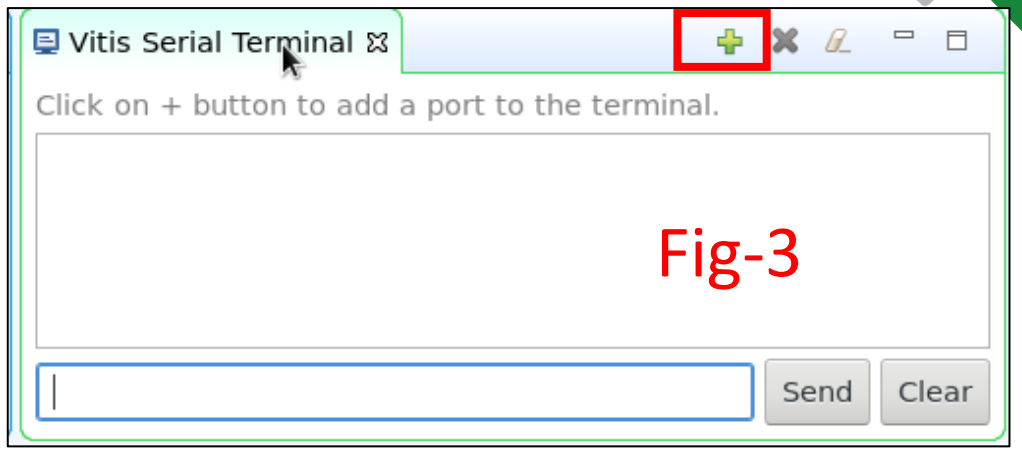


Fig-3

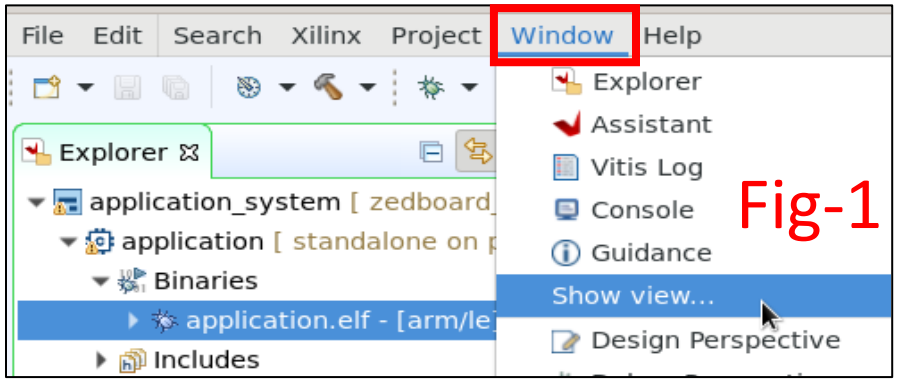


Fig-1

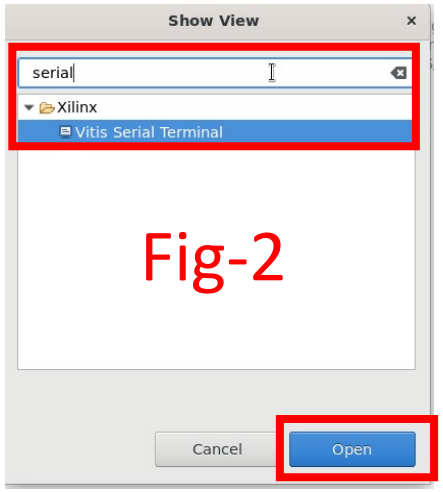


Fig-2

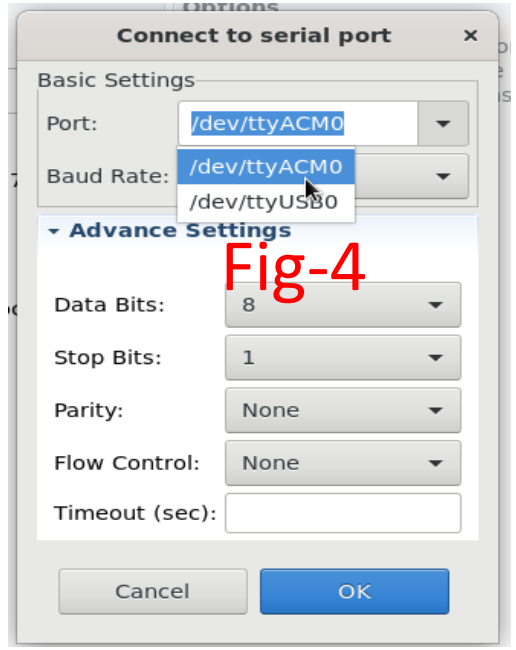
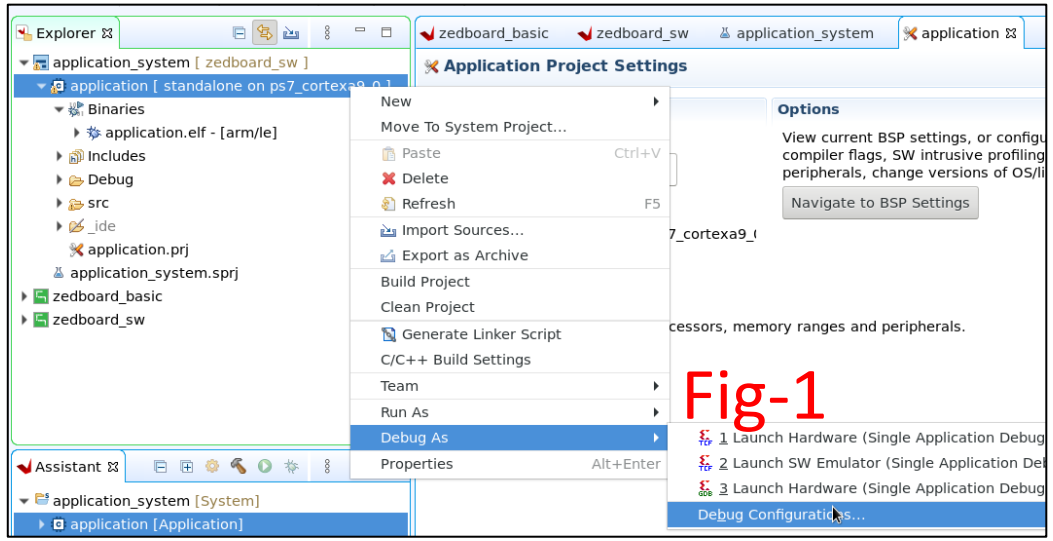


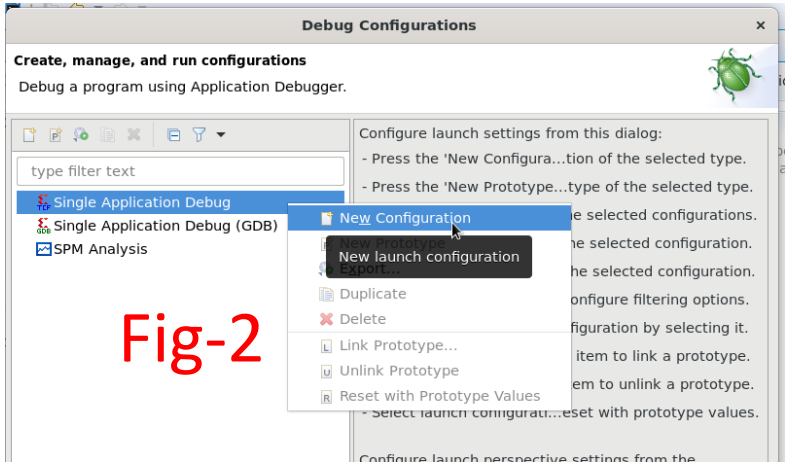
Fig-4

Task 3: Debug the C++ Application

To Debug the application on the hardware

- In Explorer window Right-click on **`application →Debug As → Debug Configurations`** (Fig-1)
- Right Click Single **`Application Debug → New Configuration`**(Fig-2)
- Click on **`Debug`**
- Confirm the view switch(**if it appears**) and the launching. You can later change the view in the upper right corner.
- The application execution will start, and the debugger will halt at the main function.





Profile the C++ Application

- Activate the TCF profiler:
  - **`Window -> Show View →`** Search for tcf(Fig-1,2)
  - Select the **`TCF Profiler`** (Fig-2) in the Debug viewer(Fig-3) and click **open**.
  - Start the profiler, which will open the configuration window (Fig-4)
- TCF Profile setup
  - Enable the stack tracing and set the max frame count to 200
  - Reduce the update interval to 100ms, click **OK**

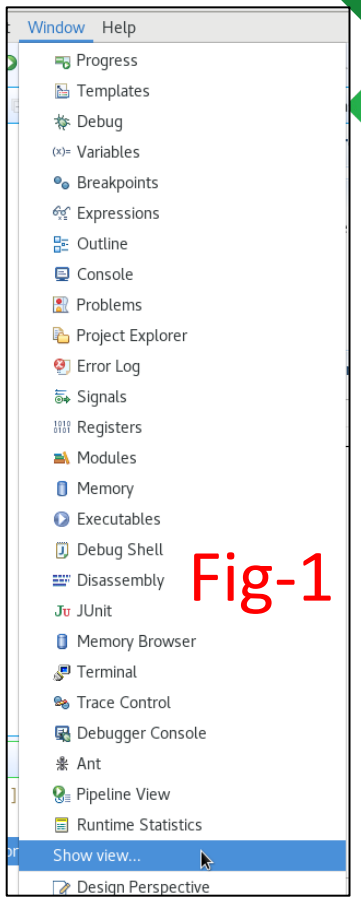


Fig-1

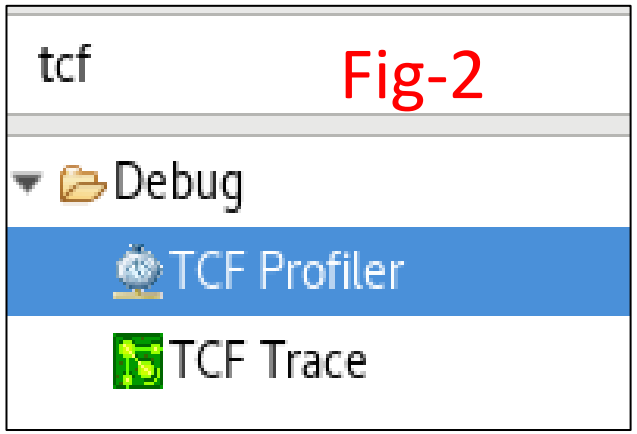


Fig-2

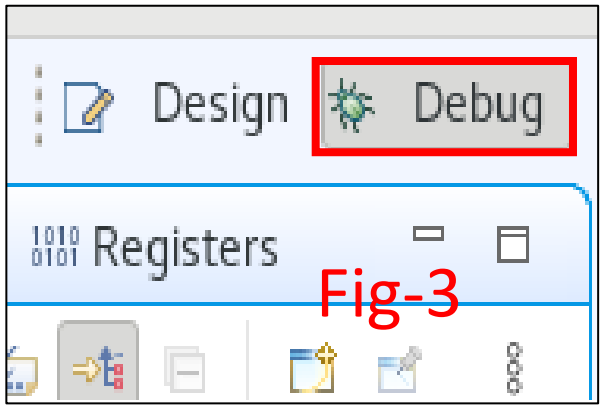


Fig-3

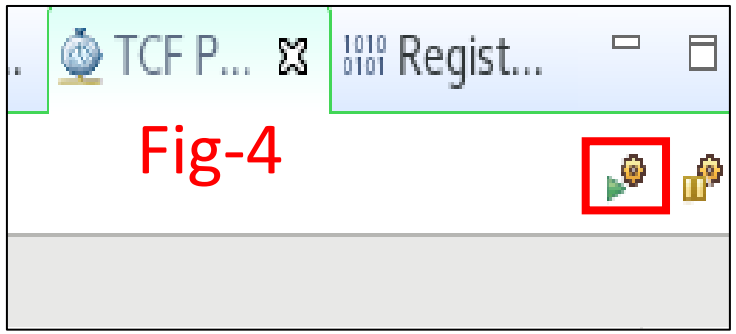


Fig-4

Profile the C++ Application

Resume the application by pressing (F8) on keyboard

- The execution will continue, and the debugger will halt again once the application is complete.
- Have a look at the profiler in the TCF Profiler window. Which are the **four** functions the program spends most of the time in? (Look at the '% Exclusive' column, if you don't understand the profiling output ask an assistant)
- **Check the reported run times on the `Vitis Serial Terminal`**(Fig-3) **console and fill it into the XLSX file(mat/perf.xlsx) to the Basic Row(Fig-2)**
- What do you observe and what do you conclude?

Implementations	SW ONLY				WITH HW ACCELERATION			
	Total	feat_creation	feat_creation/filt_bp	feat_location	Total	feat_creation	feat_creation/filt_bp	feat_location
Basic								
Hardware Acceleration(un-optimized)								
Hardware Acceleration(optimized)								

Fig-2

Fig-1

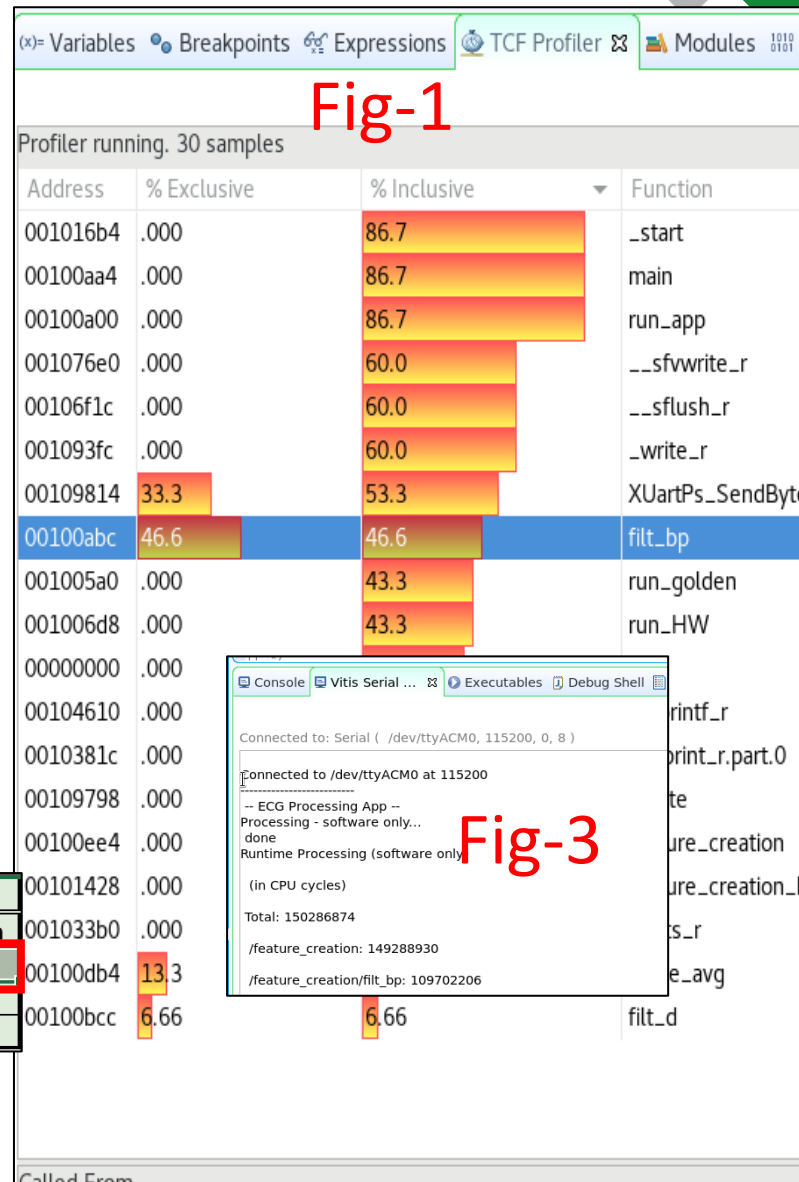
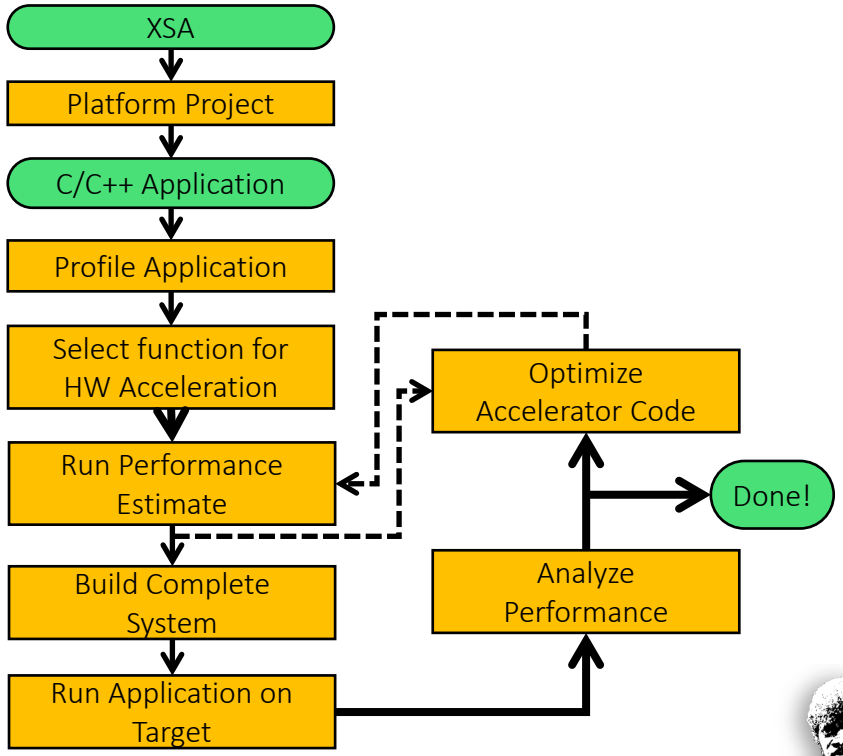


Fig-3

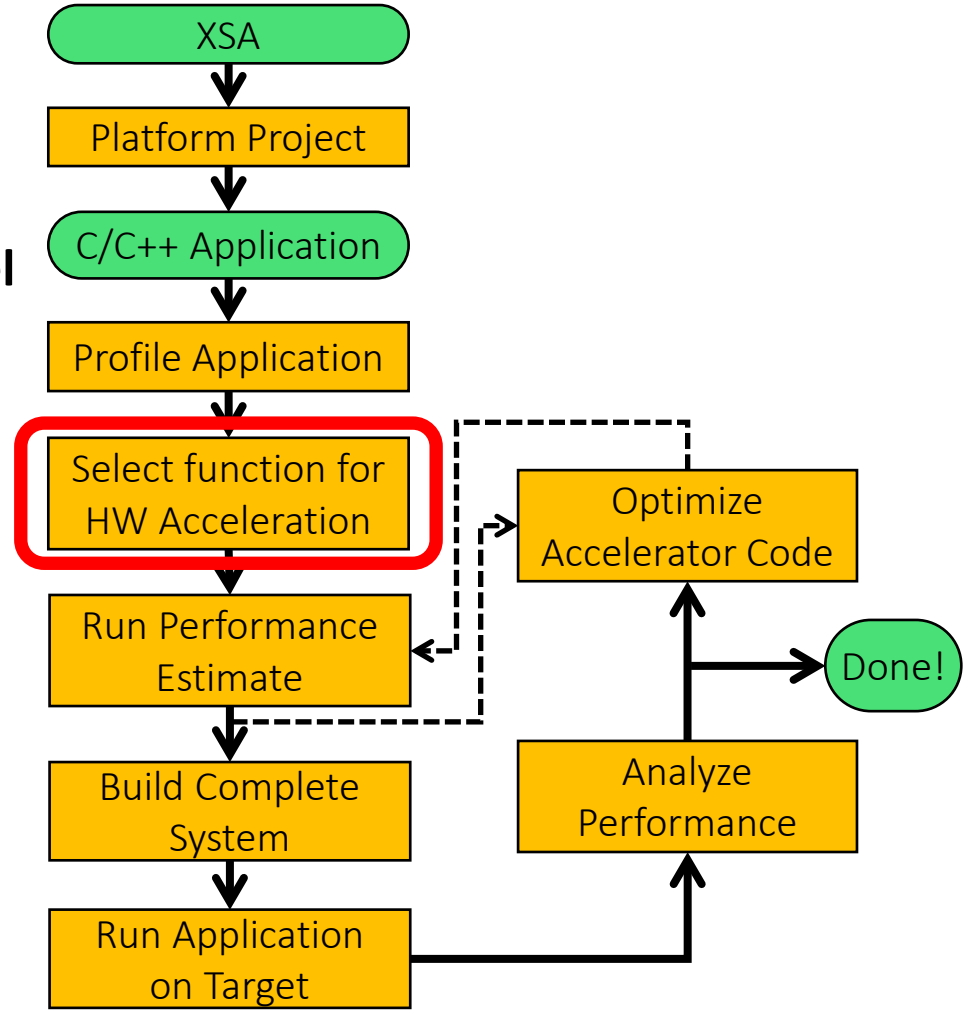
Take some time to think

- Given the profiling results (TCF output) and the measured performance from the previous step, which function(s) would you accelerate?
- Give an upper bound for the overall speedup you can achieve!
- Discuss your answer with an assistant



Where are We?

- We figured out the function suitable for Hardware Acceleration
- The next step is to accelerate the function
- This can be achieved by creating a **Hardware Kernel Project** (Steps in the following slides)



Task-4 Hardware Acceleration

- Go to design view by clicking the Design button(Fig-1) on the top right
- Hardware acceleration can be achieved using **`Hardware Kernel Project.`** Click on **`File → New → Hw Kernel project`** (Fig-2)
- If a **`Create a New Hw Kernel Project`** GUI appears(Fig-3) click **`Next`**
- Select **`zedboard_basic`** as Platform and click **`Next`** (Fig-4)

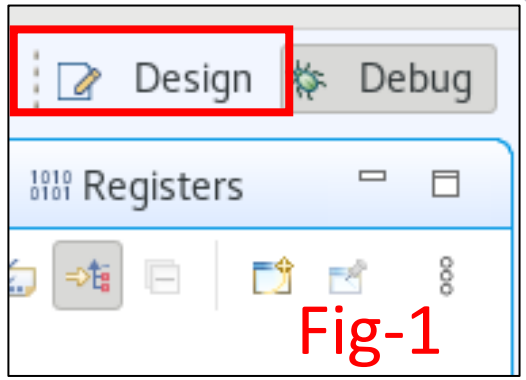


Fig-1

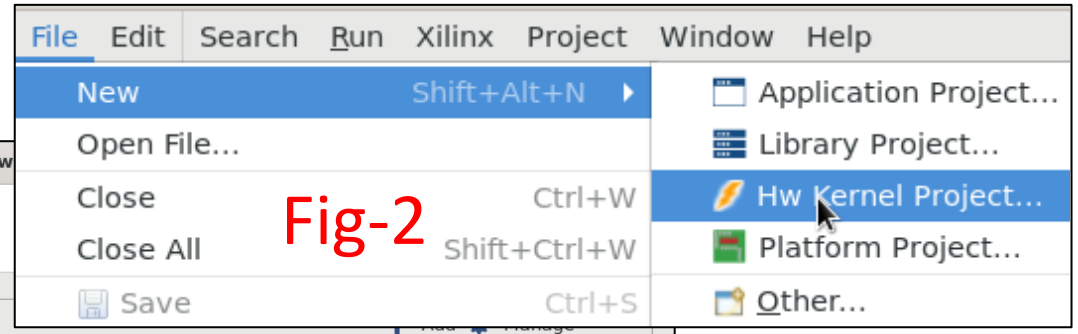


Fig-2

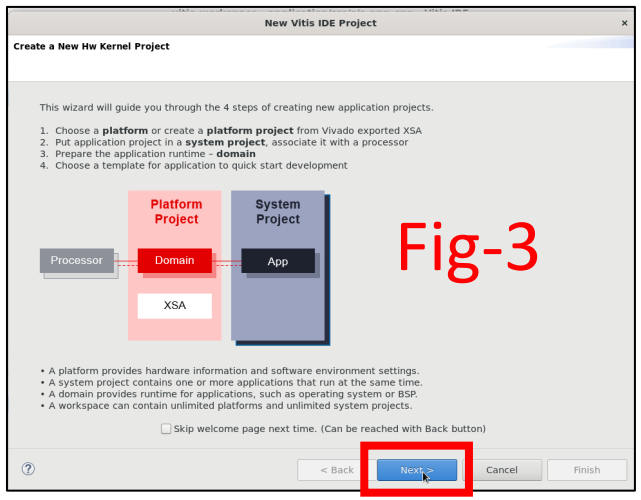


Fig-3

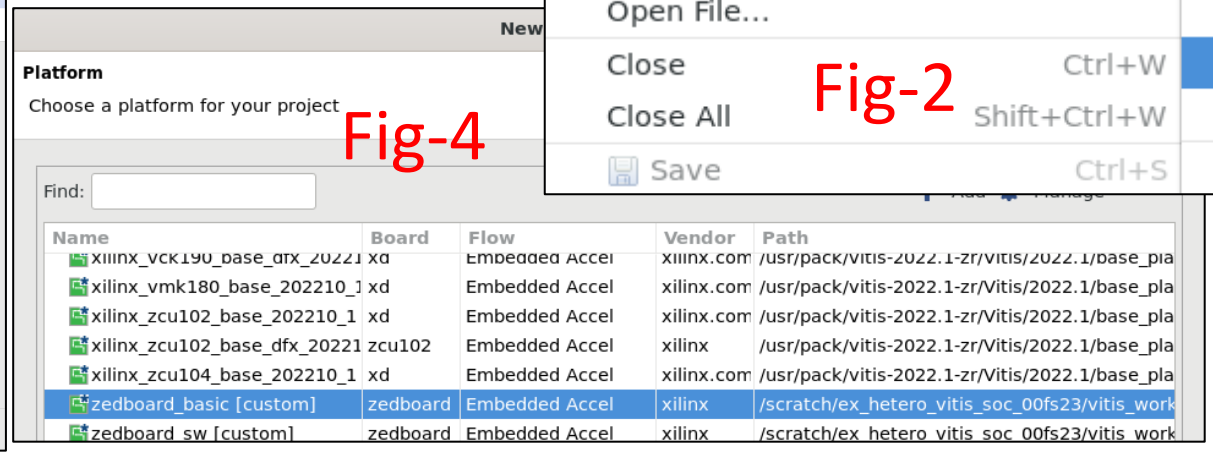


Fig-4

Task-4 Hardware Acceleration

Hw Kernel project name as `filt_bp_hw` (Fig-1) and click on **`Finish`**

This step should create two projects(Fig-2)

- `filt_bp_hw` – To accelerate subfunction
- `filt_bp_hw_system_hw_link` – Link hardware-accelerated IP with the existing platform project(`zedboard_basic`)

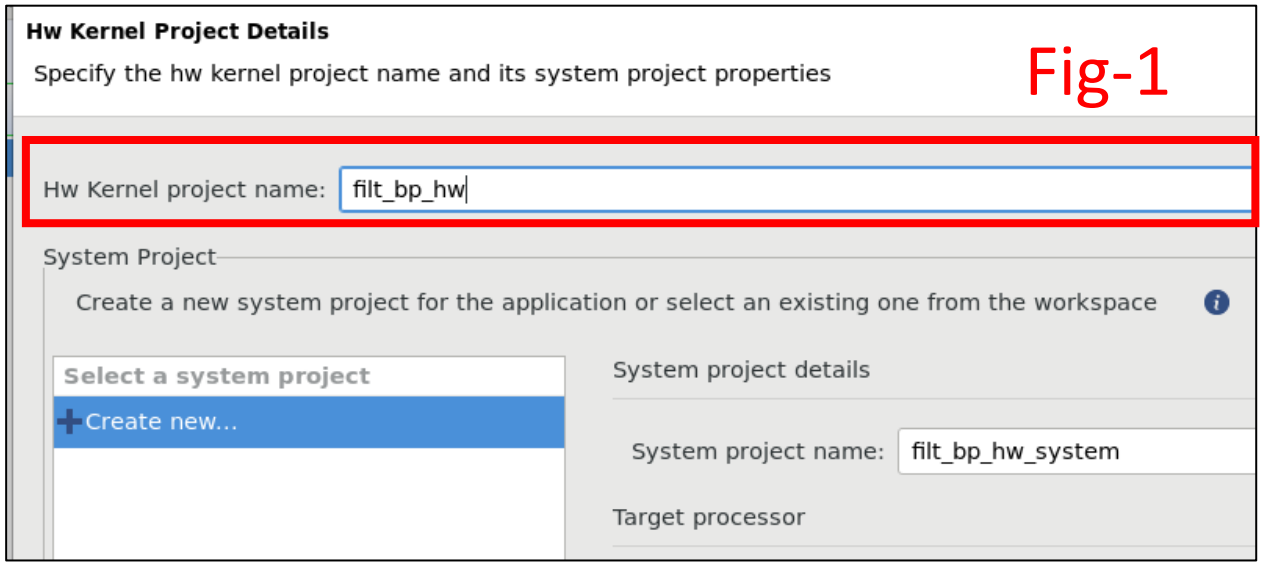


Fig-1

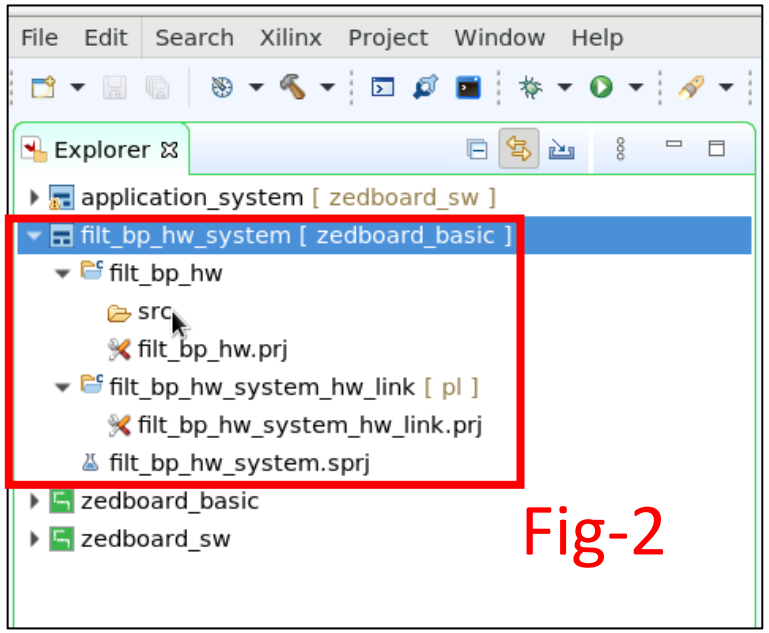
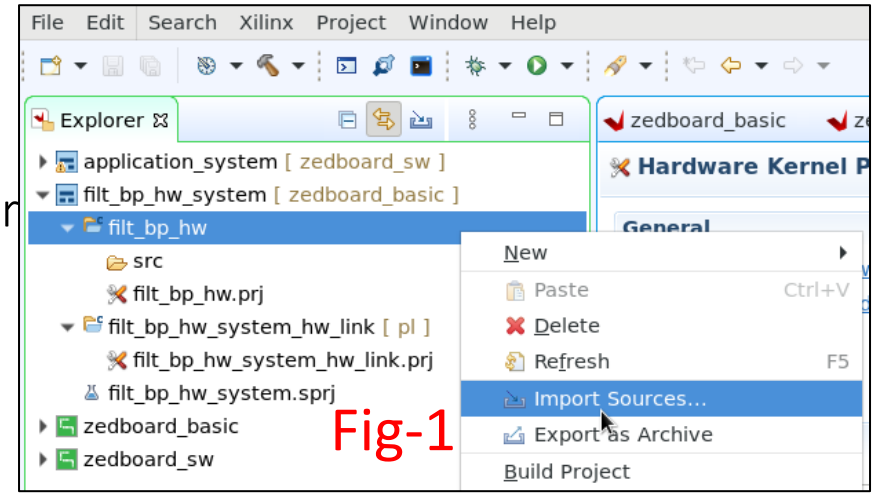


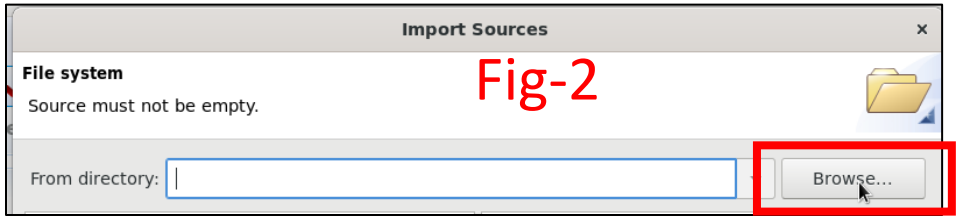
Fig-2

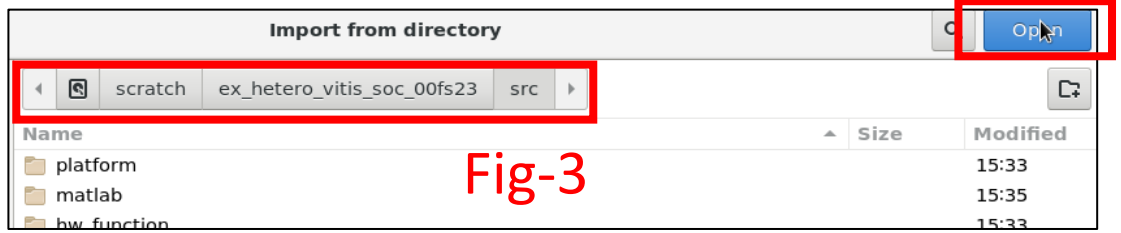
Hardware Acceleration

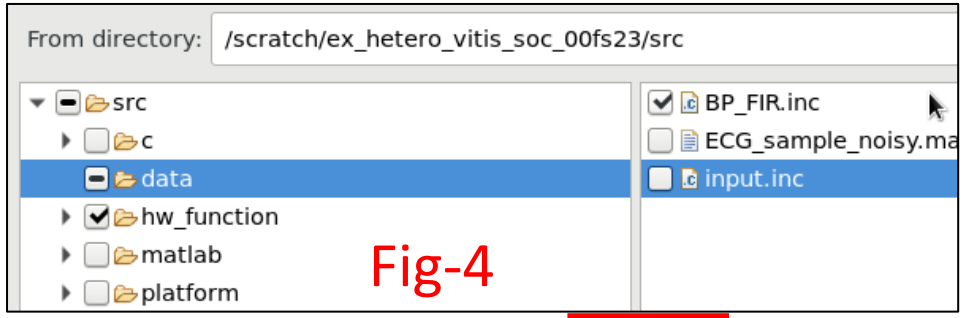
Add sources for the hardware acceleration.

- To accelerate `filt_bp_hw`, the necessary files are
  - Filter coefficients `BP_FIR.inc` located in `src/data` folder
  - Source file `hw_filt_bw.cpp` located in the `src/hw_function` folder
- `filt_bp_hw` → `Import Sources (Fig-1)`
- Click on `Browse(Fig-2)` → `Open "src"(fig-3)` folder
  - Select `src/data/BP_FIR.inc`
  - Select `src/hw_function/hw_filt_bw.cpp`
  - Click on `Finish`









Hardware Acceleration

- Double click on `filt_bp_hw_system.sprj`.
  Set the Active Build Configuration to Hardware (Fig-1)

Select the hardware function to Accelerate

- Double-click on `filt_bp_hw.prj` inside `filt_bp_hw` project(Fig-2)
- Click on `Add Hardware Function` (Fig-2) → Select `filt_bp_hw` (Fig-3)
- `Finish`

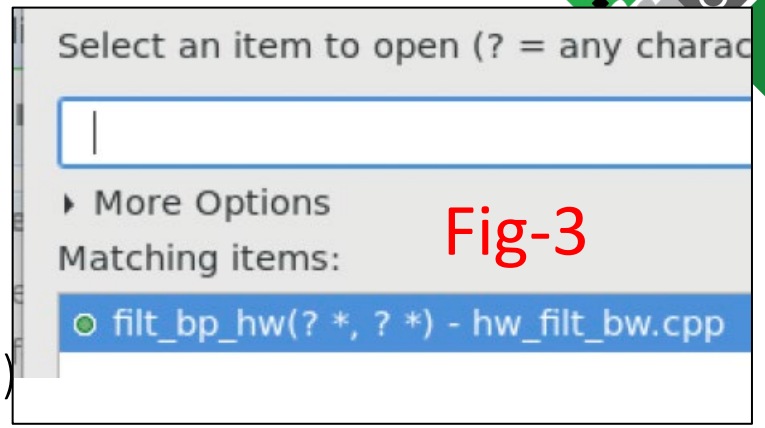
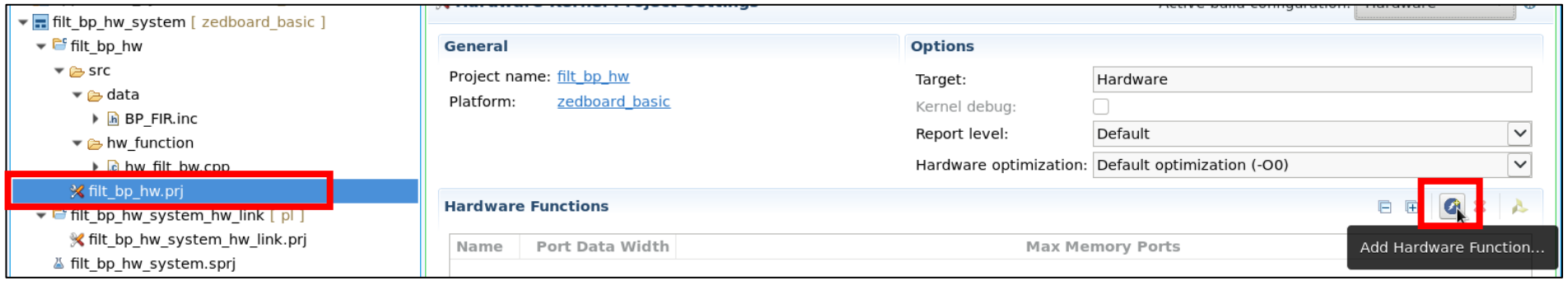


Fig-3



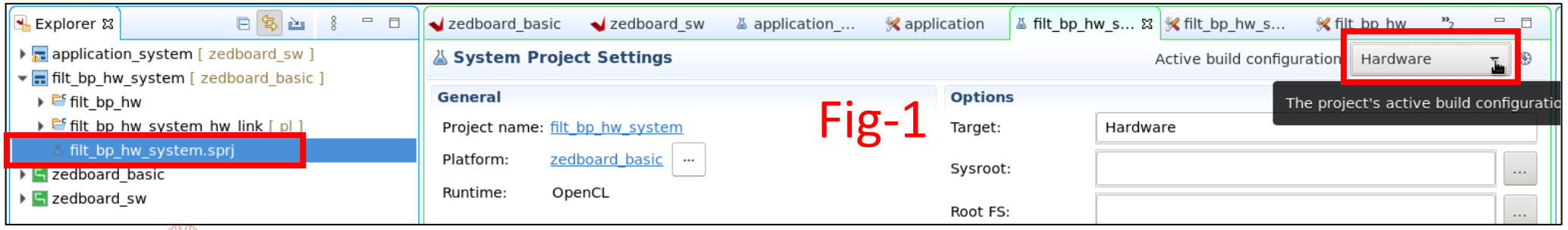
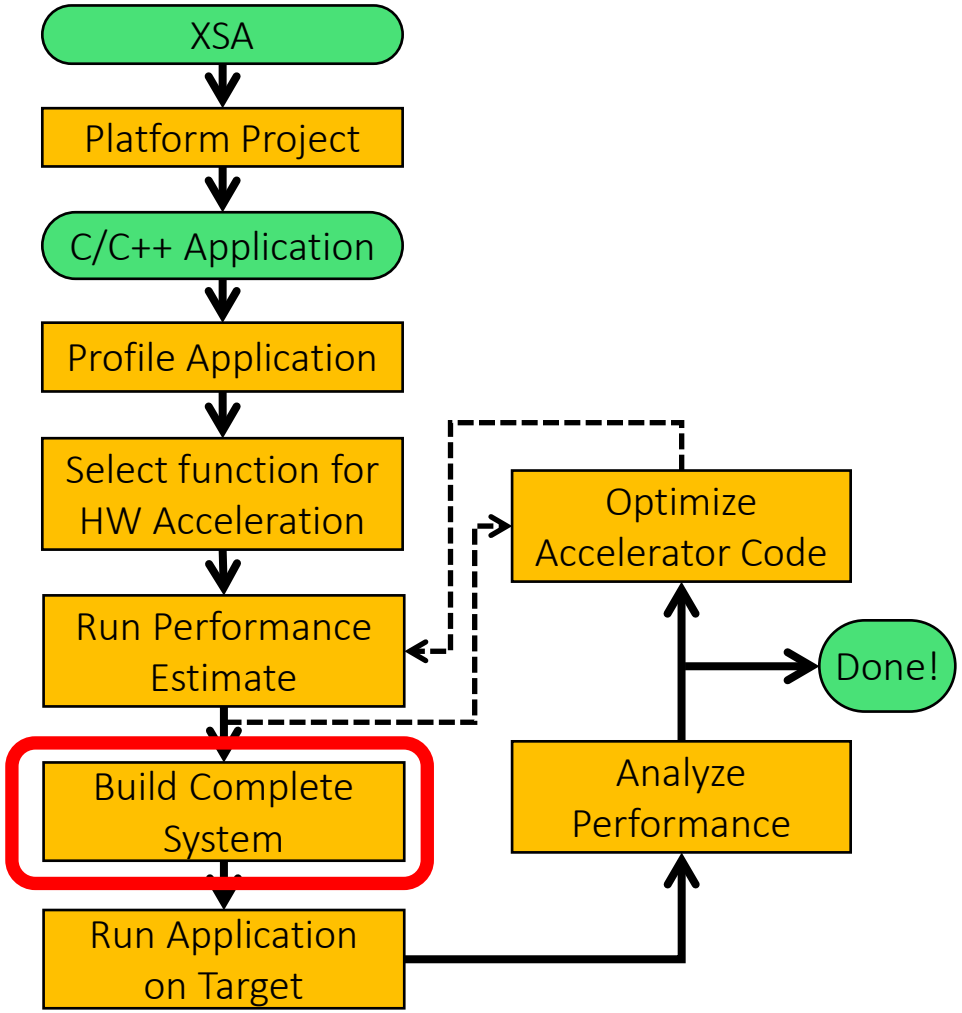


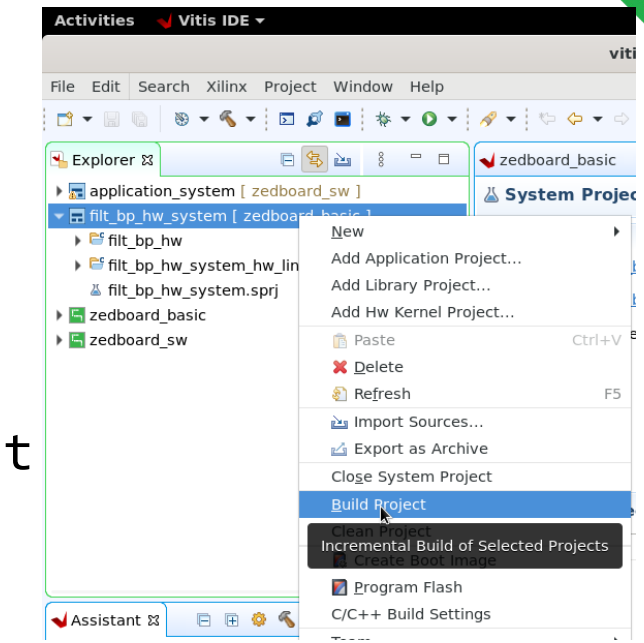
Fig-1

Where are we?


XSA
Platform Project
C/C++ Application
Profile Application
Select function for HW Acceleration
Run Performance Estimate
Build Complete System
Run Application on Target
Optimize Accelerator Code
Analyze Performance
Done!

Hardware Acceleration

- Enable **`Export hardware(XSA)`**(Fig-1)
  - Double-click on **`filt_bp_hw_system_hw_link.prj`**
  - Click to tick on the **`Export Hardware(XSA)`**
- Build the system Project
  - Right click **`filt_bp_hw_system_hw_link`** → **`Build Project`**
  - This will take some time
- Move to next slide



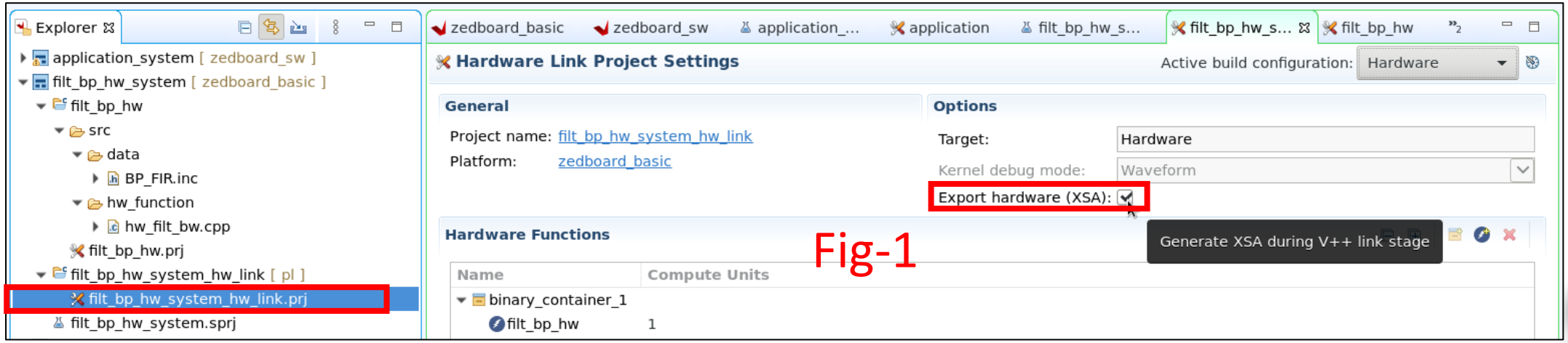


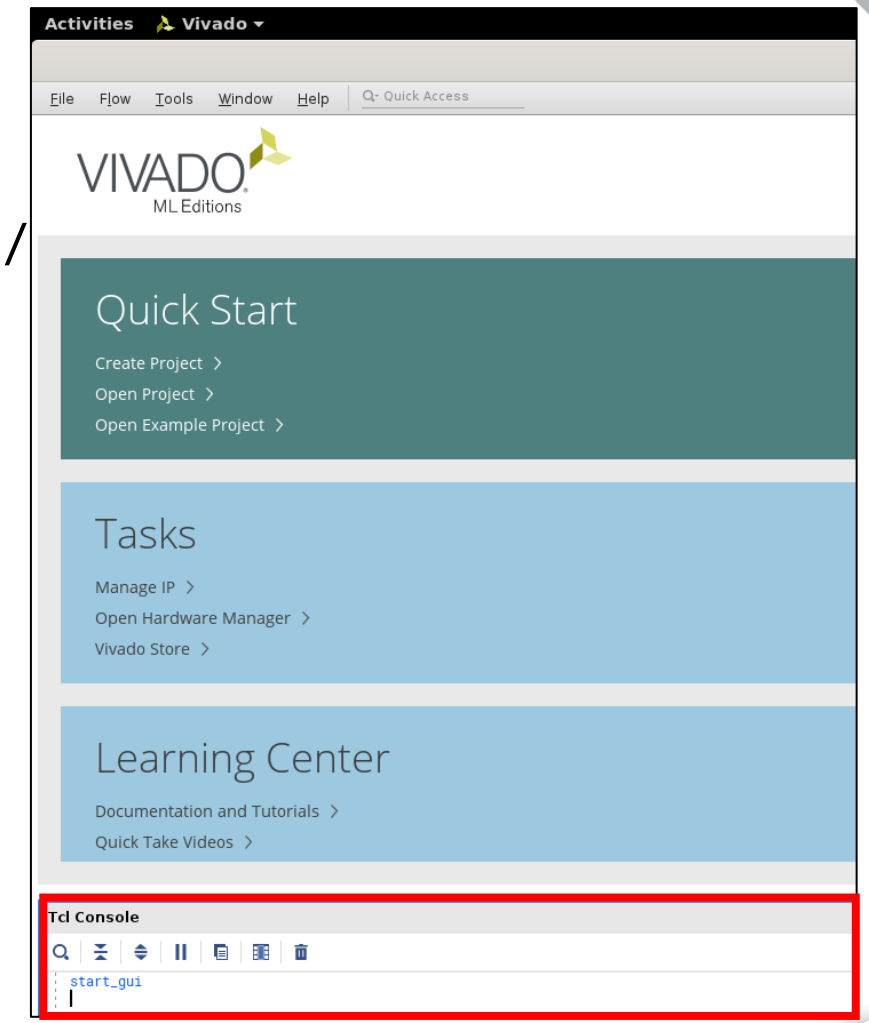
Fig-1

Hardware Acceleration (Insights)

- Under the hood
  1. Vitis HLS is called to create a hardware IP of the `filt_bp_hw` function
  2. The hardware IP generated in step-1 is linked to the zedboard_basic platform
  3. Vitis Vivado is called for the backend design flow that generates bitstream
  4. Lastly, XSA is generated from the new Vivado project containing HLS-generated IP `filt_bp_hw`
- While we wait for the Build to finish, let's look at the existing XSA(design_1_wrapper.xsa). The steps are explained in the next slide

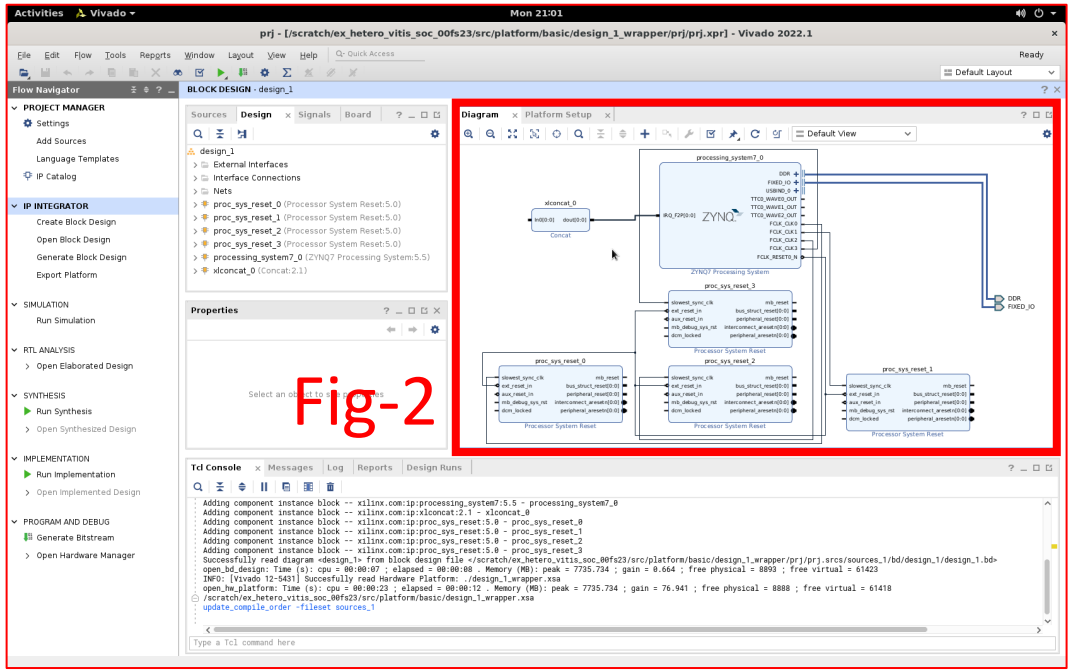
Open XSA file

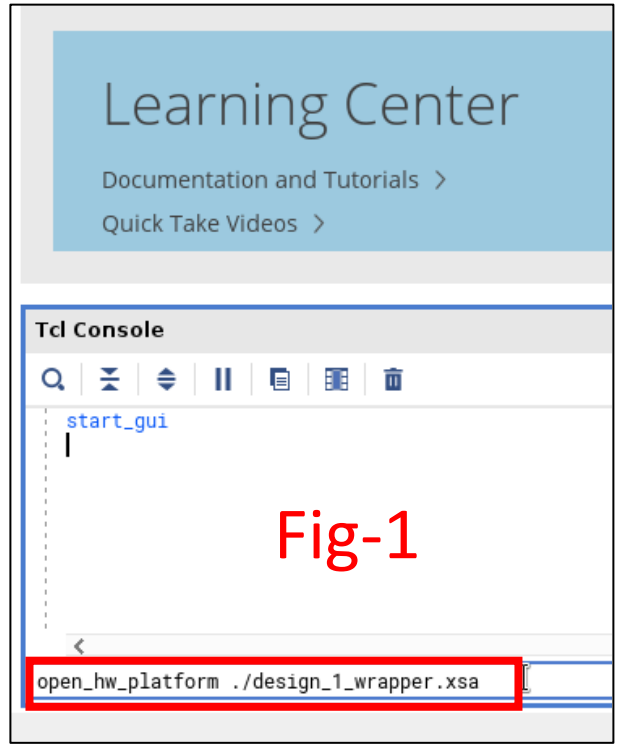
- Open a new terminal
- Navigate to folder `cd /scratch/ex_hetero_vitis_{USER}/src/platform/basic`
- `start-dz-env`
- Launch Vivado using the command
  - `vitis-2022.1 vivado &`
- After the Vivado GUI is up, you will find Tcl Console in the bottom left.



Open XSA file

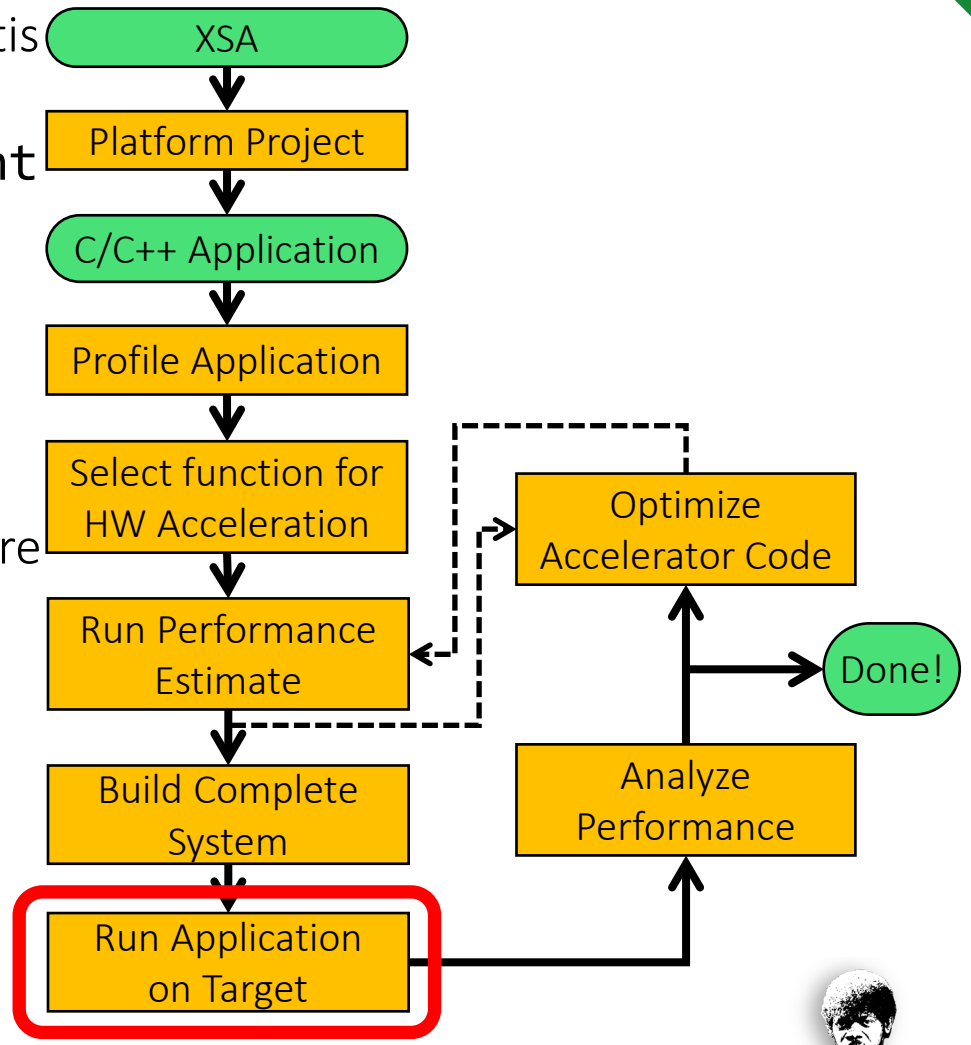
- Use Tcl command open_hw_platform ./design_1_wrapper.xsa
- It will launch the associated Vivado project. You can see the Block Diagram to get an idea about the platform(Fig-2).
- Close the Vivado Project and have a look at the Vitis GUI





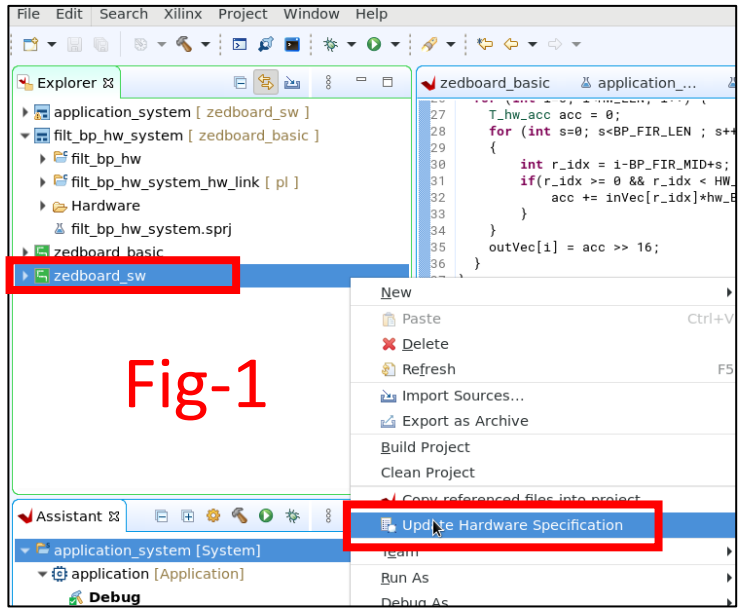
Where are we?

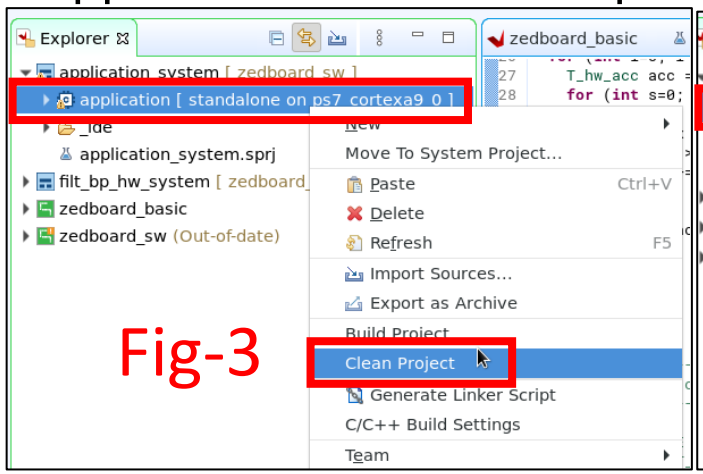
- After the build is finished, the updated XSA produced from Vitis is located in `vitis_workspace/filt_bp_hw_system_hw_link/Hardware/binary_container_1.xsa`. This XSA contains the hardware-accelerated IP generated by Vitis HLS
- Copy the `binary_container_1.xsa` to `src/platform/hw_unopt` folder.
- To take advantage of the new XSA, we will update the hardware description for the `zedboard_sw` platform project. Steps in the next slide.

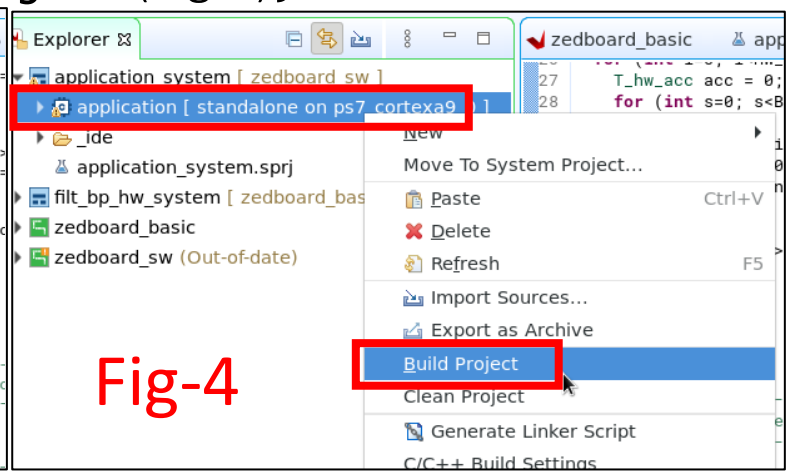


Task-5 Retarget Application

- Right-click on **`zedboard_sw`** → **`Update hardware specification`**(Fig-1). **`Browse`** the New XSA you copied in the **`src/platform/hw_unopt`**(Fig-2) folder and click **`OK`**.
- Right-click **`zedboard_sw`** project → **`Build Project`** to update the contents from the new XSA
- Update the application, right-click on **`application`** → **`clean project`**(Fig-3), then **`build`** the project(Fig-4)







Run Application on Zedboard

- Run the application by Right click on **`application → Run As → Run Configurations`**.
- Right-click on the Existing Configuration file under **`Single Application Debug`** and **`Delete`**(Fig-1)
- Right-click on **`Single Application Debug → New Configuration →`** Click on **`Run`**
- Check the output on **`Vitis serial terminal`**

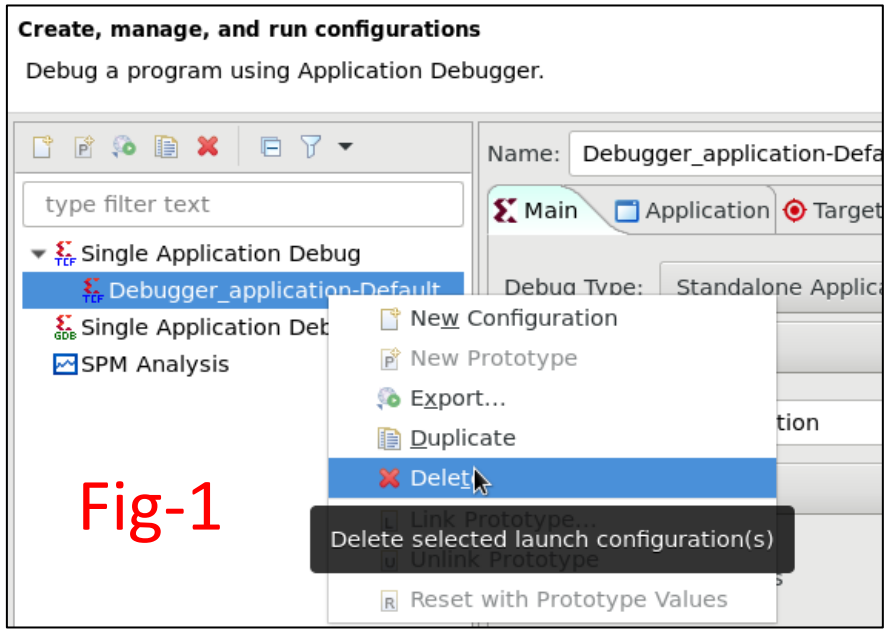


Fig-1

Run Application on Zedboard

- Do you see any difference compared to previous results?
  - Do you know why? (Hint – We have added print in the code to check if the `bp_filt` is run on Software or Hardware ☺)
- What's next?
  - Run the `bp_filt` on the hardware.
  - Uncommenting `HW_KERNEL` define would enable `bp_filt` on hardware. How?
    - Open `hw_processing.cpp` and `hw_processing.h` in `application/src/c` . The `HW_KERNEL` define enables `call_filt_bp_hw`, which in turn calls HLS-generated HW IP

Some work to make it work ☺

- In the `call_filt_bp_hw` function (in `src/c/hw_processing.cpp`), there are 3 placeholders to get you acquainted with the interface
- Task – 5.1
  - Move data from the Host to the shared memory from which the accelerator IP can fetch the data. (Hint – We achieve this by **`memcpy`** to transfer the data from **`inVec`** to **`hw_in`**)
- Task – 5.2
  - Start the accelerator IP block. We have added a hint for you in the code ☺
- Task – 5.3
  - After the accelerator IP processes the data, we move the data back to the shared memory. Again we use **`memcpy`** to transfer from **`h_out`** to **`outVec`**

Now let's test the results on the Zedboard

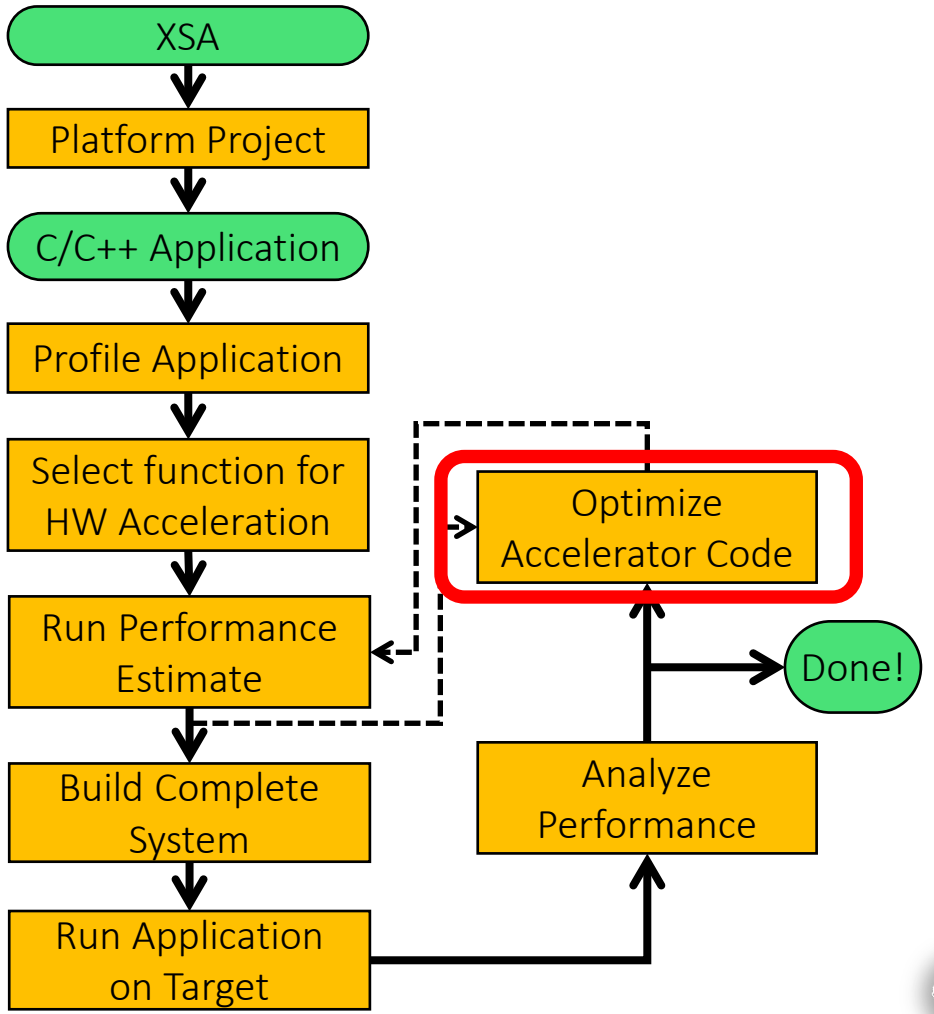
HW Accelerator on hardware

- Build the application
- Run the application on Zedboard
- How much speedup did we obtain?
  - Congratulations: we have built the first hardware deaccelerator **which is perfectly normal if your accelerator is not optimised from the HLS side**
  - Fill the results into Hardware Acceleration(un-optimized) row `src/mat/perf.xlsx`

Implementations	SW ONLY				WITH HW ACCELERATION			
	Total	feat_creation	feat_creation/filt_bp	feat_location	Total	feat_creation	feat_creation/filt_bp	feat_location
Basic								
Hardware Acceleration(un-optimized)								
Hardware Acceleration(optimized)								

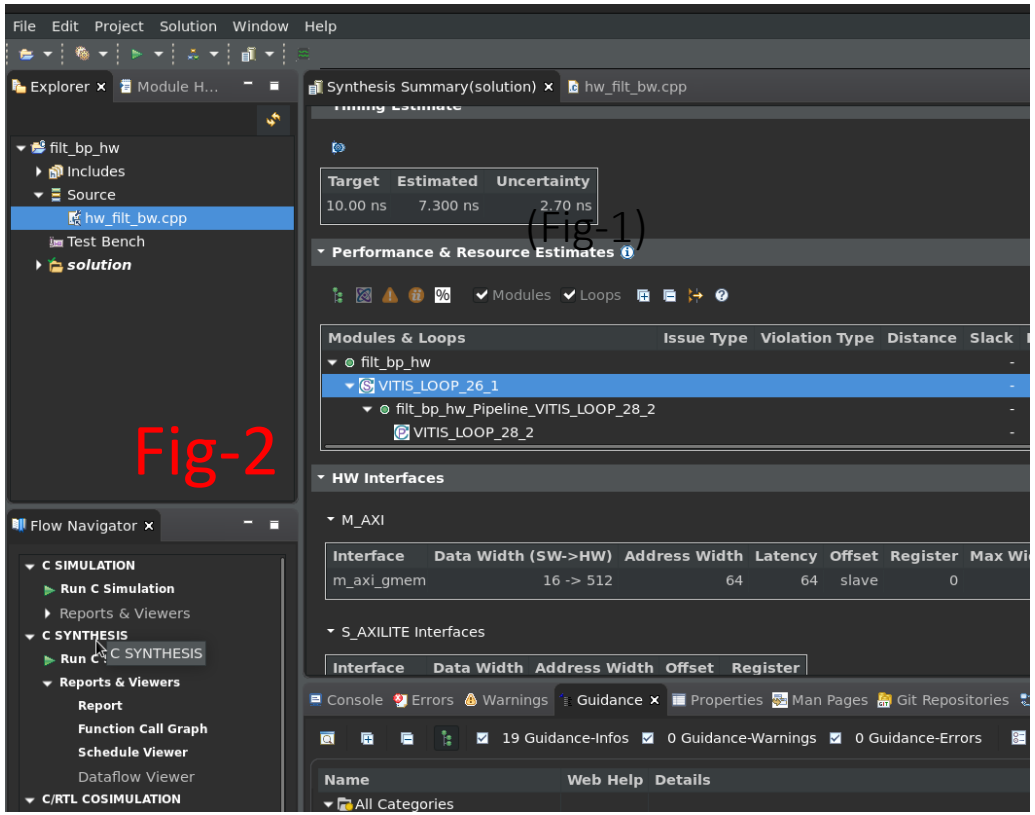
Where are we?

- We have analyzed the basic hardware acceleration on the hardware
- Next let's work on Optimizing the Accelerator Code
- To get a first estimate we can make use of the HLS reports generated by the tool

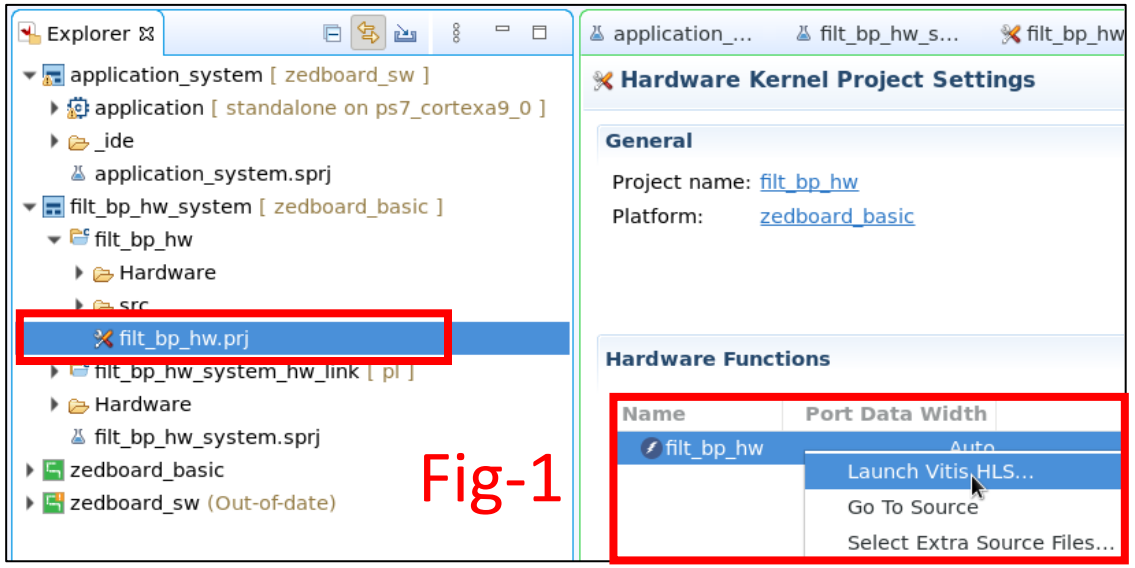


Diving into HLS

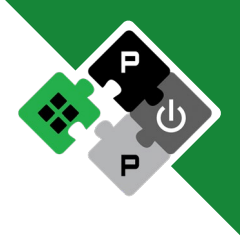
- Let's open the respective HLS project. Double-click on **`filt_bp_hw.prj`** → right click on **`filt_bp_hw`** → click on **`Launch Vitis HLS`** → **`OK`** (Fig-1)
- Vitis HLS GUI will open (Fig-2).



(Fig-1)



Hardware Accelerator – HLS point of view

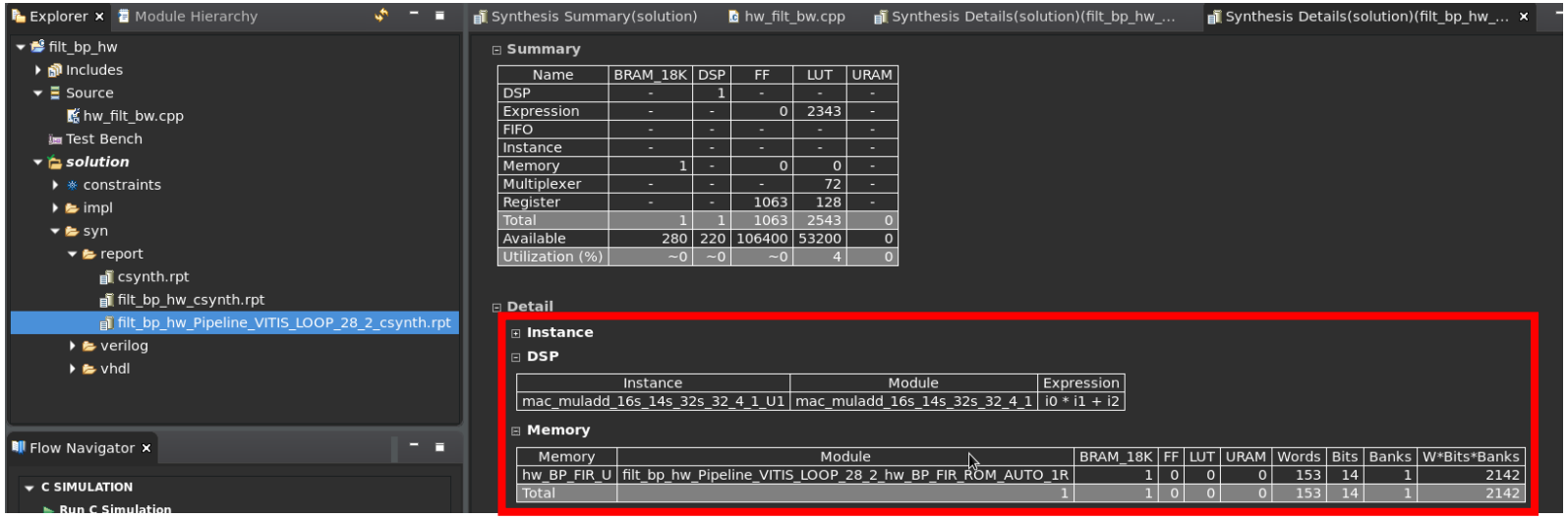


Let's have a look at the code `filt_bp_hw.cpp`

- You can see two arrays (which by default are memories in HLS) for the filter coefficients and the shift buffer (`outBuf`)
- Then we have the outer loop, the main loop over the data stream. By default, `for loops` describe sequential behaviour in HLS. Within the loop we have the following:
  - A for loop(sequential)that shifts the buffer elements forward.
  - A gated input read
  - A for loop(sequential) that updates the elements in the buffer with the contribution of the new input value
  - A gated output
- Note the following:
  - The read and writes to `inVec` and outVec are purely sequential.
  - Note the loop boundaries of the outer for loop. Together with the gated input and output this assures transient behaviour according to our needs
  - Let's see what HLS does with this code

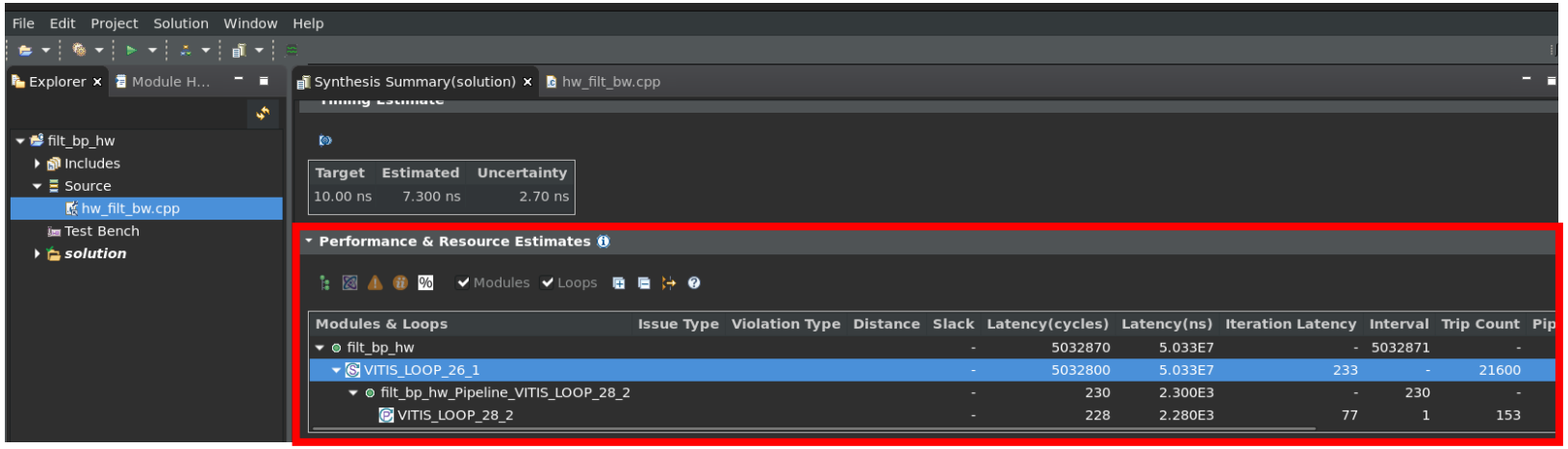
Hardware Accelerator – HLS point of view(Insights)

- The multiplication + addition operation is mapped to a single **DSP** instance
- The FIR filter coefficients **hw_bp_FIR** are mapped to the **BRAM** instance



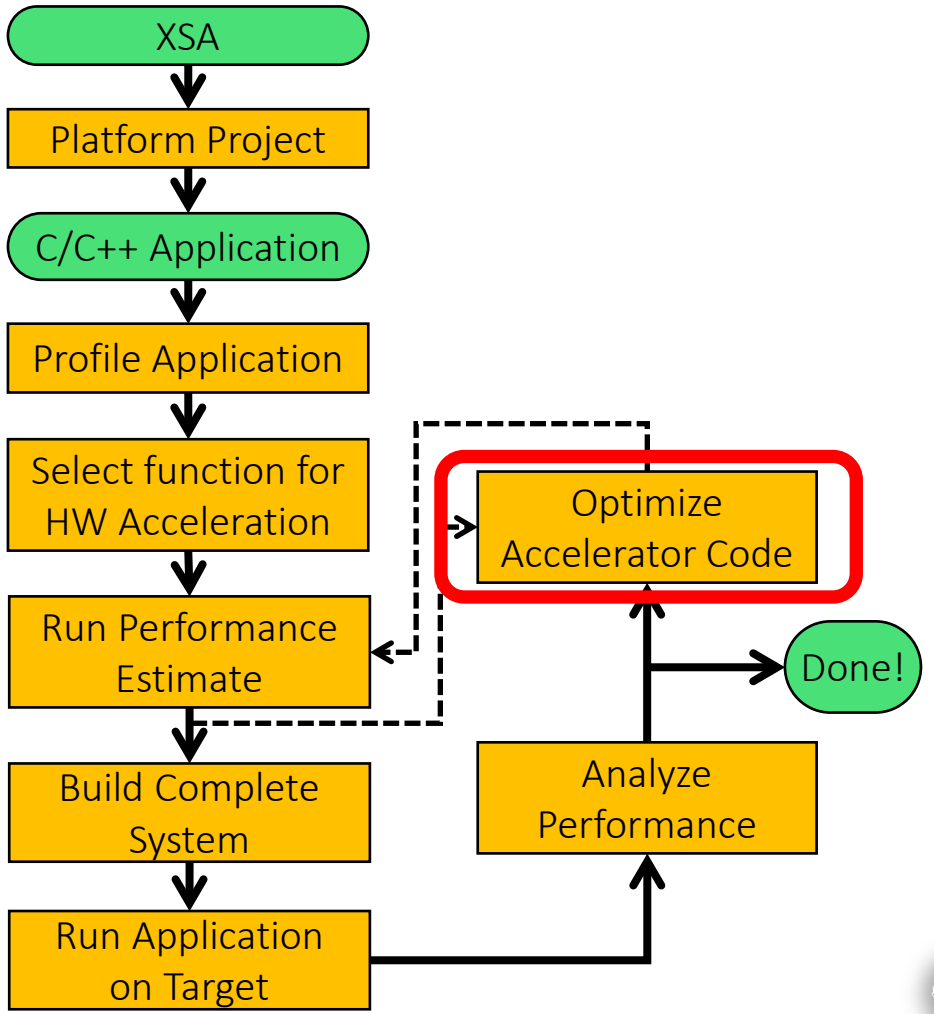
Hardware Accelerator – HLS point of view(Insights)

- The nested loop's trip count(number of iterations) (`VITIS_LOOP_28_2`) is 153, as are `BP_FIR_LEN`=153 iterations executed sequentially through a single DSP module. Similarly, the top loop has a trip count of `HW_LEN`=21600
- As `VITIS_LOOP_28_2` is pipelined, the total latency for the loop is approximately `Iteration Latency +Interval*(Trip Count - 1).` A similar analogy can be made for the top loop.
- Open `src/mat/HLS.xlsx` and fill data in `"Basic-1"` Row
-



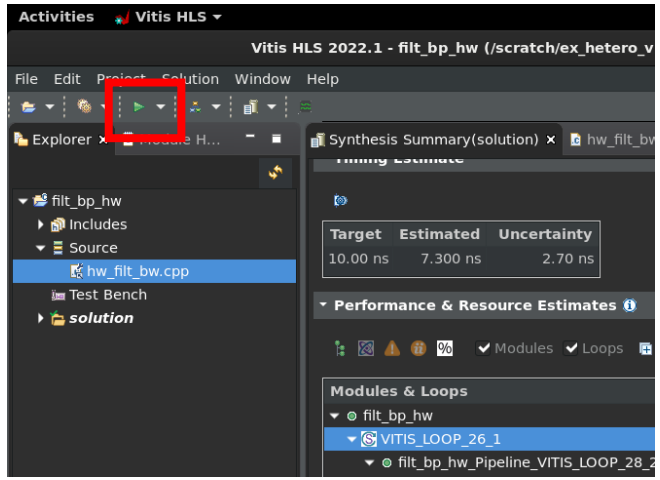
Hardware Accelerator - Optimization

- Since writing code for HLS with a good QoR (Quality of Result) is not trivial and far beyond the scope of this exercise, we provide an optimised implementation.
- Again: HLS does **NOT** create hardware from what your C/C++ code does; Rather, **YOU** have to describe a sensible hardware architecture in C/C++. Good HLS code is usually terrible as software, and vice versa!
- Open `hw_filt_bw.cpp` in Vitis HLS GUI
  - Comment out the current implementation of `filt_bp_hw` and remove the comments around the implementation below.
  - Run C Synthesis . Open `src/mat/HLS.xlsx` and fill data in **"Basic-2"** Row



Hardware Accelerator - Optimization

- Let's assume we want to achieve a throughput close to one filter output every clock cycle (i.e., every 10 ns). Currently, we are only getting one every ~462 cycles!
- Naturally, we will only get this if we compute all multiplications required for a filter output in parallel. To achieve this, we do the following:
- We completely unroll the filter calculation loop to place sufficient computing resources to compute all multiplications in parallel.
- Add in for loop *`#pragma HLS UNROLL`*
- Run `C Synthesis` and check the C Synthesis report
- Open `src/mat/HLS.xlsx` and fill data in `"OPT-1"` Row



Observations from the C Synthesis report

- Did you observe any changes to the number of DSPs?
  - How many DSPs do you expect? Is it matching your expectation?

- Did unrolling the loop help to gain a significant latency?

Please go to the next slide to check the correct answer.

Observations from the C Synthesis report

- Did you observe any changes to the number of DSPs?

  >> Yes, the number of DSPs increased to 59 from 1.

- How many DSPs do you expect? Is it matching your expectation?

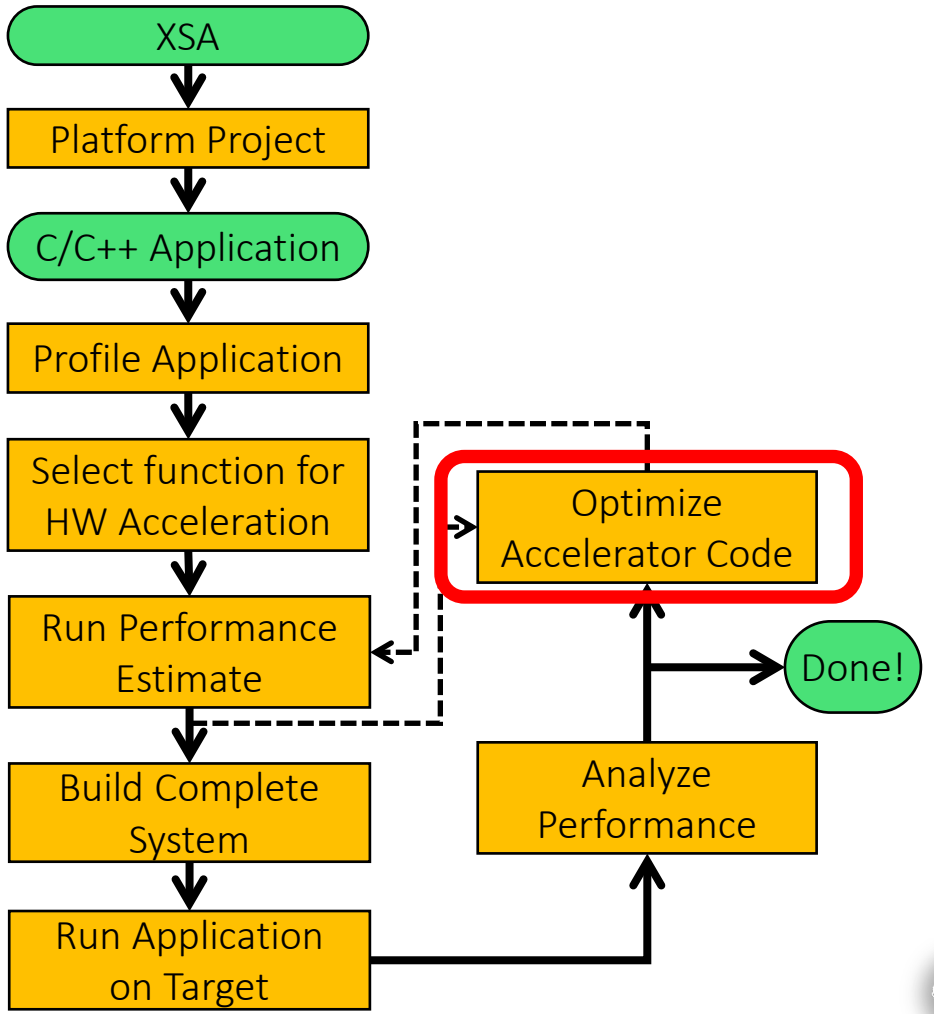
  >> Ideally, there should be 153 DSP as we unrolled the loop with BP_FIR_LEN=153 iterations. But we only use 59 DSP slices even though our filter order is 153. This is because our filter coefficients are static, and some coefficients are easier implemented without a hard-wired multiplier (e.g., * 8 with a shift)

- Did unrolling the loop help to gain a significant latency?

  >>No, we don't get a significant gain in latency despite unrolling the filter calculation loop. This is because we still use a single memory for the shift buffer. And since we only have a limited number of memory ports (2), there is a limited gain in replicating the processing part without adapting the memory (sequential) part.

Where are we?

- We still need to make further optimisation to the Accelerator code



Hardware Accelerator - Optimization

- We will increase the number of ports for `outBuf`. This can be done by mapping the `outBuf` to Flipflops instead of BRAM.
- We complete partition `outBuf` to replace the memory structure with flip-flops:(add the pragmas after the variable declarations) *`#pragma HLS ARRAY_PARTITION variable=outBuf complete dim=1`*
- Rerun the synthesis and check your observations.
  - Which resource should change?
  - Open `src/mat/HLS.xlsx` and fill data in `“OPT-2”` Row

Hardware Accelerator - Optimization

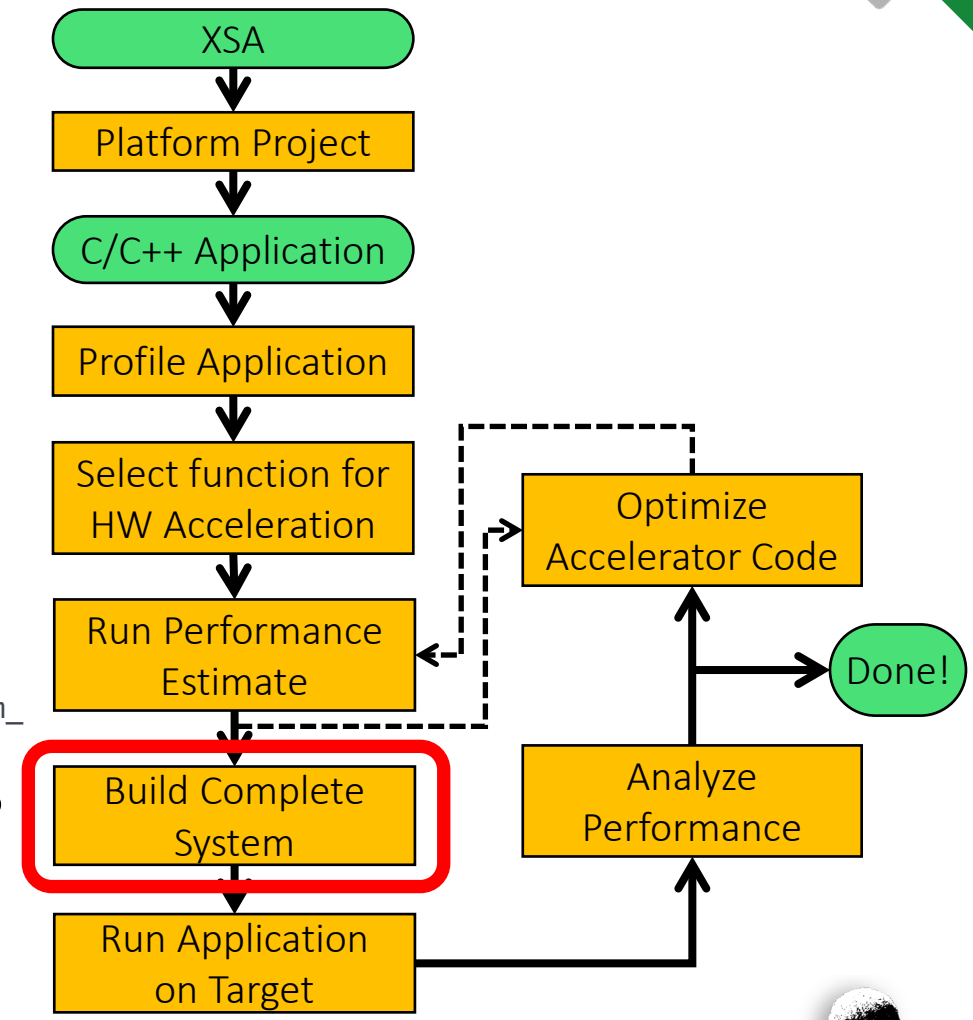
- We will increase the number of ports for `outBuf`. This can be done by mapping the `outBuf` to Flipflops instead of BRAM.
- We complete partition `outBuf` to replace the memory structure with flip-flops:(add the pragmas after the variable declarations) *`#pragma HLS ARRAY_PARTITION variable=outBuf complete dim=1`*
- Rerun the C synthesis and check your observations. Which resource should change?
  - `outBuf` is mapped to FlipFlops. So, we observe BRAM utilisation goes to 0, and FF utilisation goes to 15953.
- But still, the Interval for the loop is way bigger compared to 1. This is because the shifter loop is still sequential. We can make it parallel by unrolling it. Unroll the shift buffer advancement by placing *`#pragma HLS UNROLL`*
- Rerun the C synthesis and check your observations. Open `src/mat/HLS.xlsx` and fill data in `"OPT-3"` Row

Where are we?

- When the HW is optimized we can build the new system.
- Close Vitis HLS GUI
- Right-click on `filt_bp_hw_system_hw_link` →`Build Project`
- Wait until the bitstream is generated
- While we wait for the bitstream to complete, we can already look at the project linked by vivado:
  - In Vivado tcl console:

    ```
    open_project
    /scratch/ex_hetero_vitis_$::env(USER)/vitis_workspace/filt_bp_hw_system_
    hw_link/Hardware/binary_container_1.build/link/vivado/vpl/prj/prj.xpr
    ```

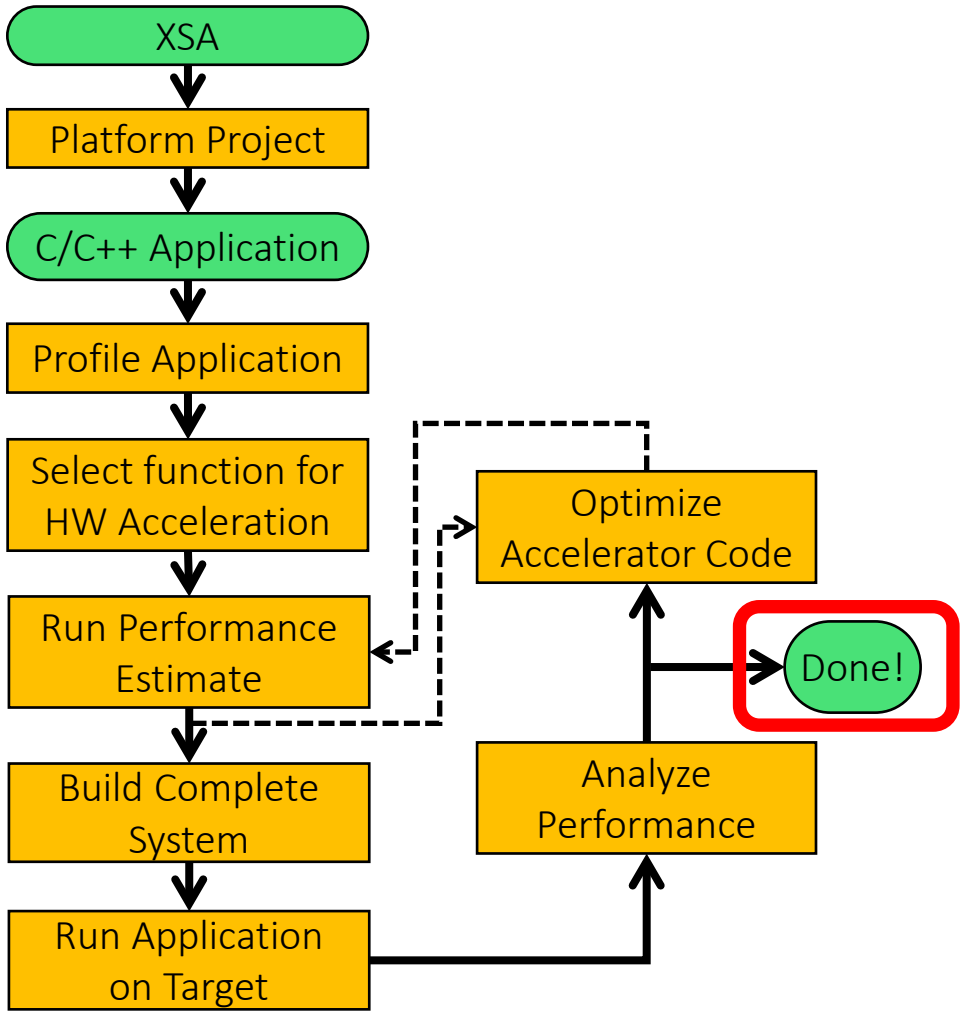
  - What is the difference you see compared to the basic XSA?
  - Close the Vivado GUI and get back to Vitis Workspace GUI
- After the build is complete, we move the generated XSA to `src/platform/hw_opt` folder.



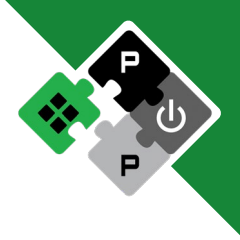
Retarget the application(Optimized Hardware Accelerator)

- Follow the same steps as we followed before.
- Use the XSA at the `src/platform/hw_opt` to update `zedboard_sw` hardware specification.
- Then we build the `zedboard_sw`
- Clean and build the application project. Followed by running the application on board.
- What do you observe?
  - Fill the `Hardware Acceleration Optimized` row in the `src/mat/perf.xlsx` file
  - We got quite some speedup.
  - Congratulations!

Where are we?


XSA
Platform Project
C/C++ Application
Profile Application
Select function for HW Acceleration
Run Performance Estimate
Build Complete System
Run Application on Target
Optimize Accelerator Code
Done!
Analyze Performance

What should you Learn in this Exercise ?



- ❑ Learn:
  - ➢ How applications can be accelerated on heterogeneous FPGA
  - ➢ How you can tackle a system design problem systematically:
    - o Profile → Estimate → Decide
  - ➢ What are possible pitfall and bottlenecks

- ❑ Experience a very advanced high-level design flow
  - ➢ Xilinx Vitis Flow
  - ➢ Get a glimpse of how such an advanced flow works

Heterogeneous system design is cool since it is very multidisciplinary!

WHEN YOU ARE DONE, PLEASE REMOVE THE EXERCISE DIRECTORY!