EPFL

EMBEDDED SYSTEMS LABORATORY

Lab on HW-SW digital systems codesign
SEL B.Sc. Curriculum (Spring Semester)
Embedded Systems Laboratory
EPFL-STI-IEL

Exercises Session 2
Out: 2025-02-27

Introduction to co-design with high-level synthesis (HLS)

1 Project flow and organization

In this session we are going to create our first IP core designs specified with high-level synthesis using Vitis HLS, we are going to integrate them in a complete system-on-chip (SoC) using Xilinx Vivado and, finally, we are going to use them from a Linux application written in C++. We will implement three systems:

- A simple combinational adder that can be accessed by the processors in the SoC using an AXI4 slave interface.
- A more complex IP core that will add the elements of two vectors into a third one. This accelerator will expose a set of configuration registers to the processors via an AXI4 slave interface, and will use another AXI4 master interface to access directly the main memory of the system (the 512 MiB of DDR SDRAM) without intervention of the processors.
- An accelerator to compute the moving average over a vector.

Since we are not yet using interrupts, the processors will check the status of the peripherals using polling on the control registers over the slave interfaces of the peripherals.

In all the exercises, we will follow a similar approach:

1. IP core specification in C/C++ with Vitis HLS with functional validation in C.
2. SoC integration and bitstream generation with Vivado.
3. Integration of the accelerator in a C/C++ application in Linux and validation in the FPGA.

1.1 Use of the Xilinx tools in the laboratory computers

As we saw in the previous session, we can start Vivado from a terminal with the following command:

```
$ /scrap/users/Xilinx2022.2/Vivado/2022.2/bin/vivado &
```

The & at the end of the command opens the program in the background so that the terminal can still be used. Similarly, we can open Vitis HLS with this command:

```
$ /scrap/users/Xilinx2022.2/Vitis_HLS/2022.2/bin/vitis_hls &
```

Access to the tools can be simplified adding the following commands to the `.bashrc` file:

```
alias vivado='/scrap/users/Xilinx2022.2/Vivado/2022.2/bin/vivado'
alias vitis_hls='/scrap/users/Xilinx2022.2/Vitis_HLS/2022.2/bin/vitis_hls'
```

1.2 Project structure and components

From now on, we will have two different sets of C/C++ files: one set for the description of the functionality of our hardware IP cores in HLS, and another set to implement the complete SW application in Linux.

- **The HLS C++ files** describe the hardware functionality and are never executed. Instead, they represent the structure of a hardware and a data flow schema. The only exception happens during C simulation inside HLS, when the code is actually compiled and executed inside the testbed to verify the correctness of the description. These files are stored in the "`HLS`" folder in the project structure shown below.
- **The Linux application C++ files** are executed in the Pynq board and are compiled and assembled into an ELF format executable. Then, they are executed by the Zynq ARM Cortex-A9 cores. These files can interact with the hardware previously implemented with Vitis HLS and Vivado. These files are stored in the "`SW`" folder in the project structure shown below.

To simplify the management of the projects, we propose the following directory structure:

```
/Exercise02
* build_HLS_project.sh     --- Invokes the TCL script to re-build the HLS project
* build_Vivado_project.sh --- Invokes the TCL script to re-build the Vivado project
|-->HDL                    --- Contains your VHDL/Verilog sources and constraint files
|-->HLS                    --- Contains your C/C++ sources for HW specification with HLS
|-->IP-catalog             --- IP repository of synthesized IP cores (generated from HLS)
|-->SW                     --- Sources for the SW project in Linux
* Exercise02_HLS.tcl       --- TCL script to re-build the HLS project
* Exercise02_Vivado.tcl    --- TCL script to re-build the Vivado project
|-->Vitis_HLS              --- Generated folder with the HLS project (don't add to git)
|-->Vivado                 --- Generated folder with the Vivado project (don't add to git)
```

If we export the project descriptions in Vitis HLS and Vivado to TCL scripts (e.g., to "`Exercise02_HLS.tcl`" and "`Exercise02_Vivado.tcl`"), then we can recreate afterwards both projects following this procedure:

1. Run "`build_HLS_project.sh`" . This will invoke Vitis HLS in batch mode to execute the script "`Exercise02_HLS.tcl`". A folder named "`Vitis_HLS`" will be created with the Vitis HLS project. The TCL script is configured to select the target FPGA model, to add the HW specification files from the folder "`HLS/`", and to add the testbed from "`HLS/main.cpp`" .
2. Start Vitis HLS, open the project (just select the folder "`Vitis_HLS`" and click "Open" without selecting any files).
3. Perform the "C Simulation" step[1] and check that the behavior of the described algorithm ("`HLS/adder.cpp`") complies with the testbed ("`HLS/main.cpp`").
4. Perform the "C Synthesis" step[1] and check in the Synthesis Summary Report window the Performance & Resource Estimates. Given a set of specification requirements, here you can check if the generated HW of your solution fulfills the desired specifications. In particular, check the latency; initiation intervals of the loops; number, type, and characteristics of the inferred AXI4 interfaces, ...
5. Perform the "Export RTL" step[1]. This will generate an RTL description of the IP core that can be integrated in a Vivado SoC project. Ensure that the *zip* file is generated in the folder "`IP-catalog/`".
6. Before importing the IP core in Vivado, uncompress the IP core *zip* package in a folder inside "`IP-catalog`". Vivado will not recognize IP cores inside *zip* files.

[1] In the Flow Navigator window

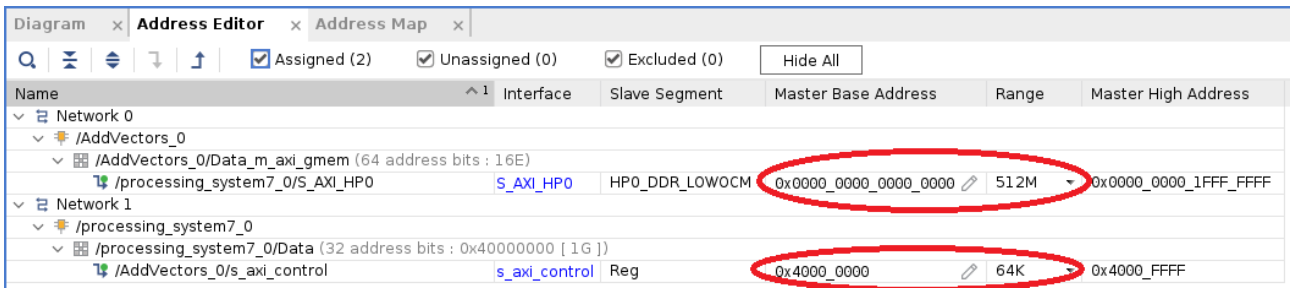


Figure 1: The address editor shows the position in the system address map that our IP core's slave registers occupy. The processors use these addresses (e.g., `0x4000-0000`) to access our peripherals. Additionally, the master ports of our IP cores can access the DDR DRAM of the system at addresses `0x0000-0000` to `0x1FFF-FFF`, which correspond to the 512 MiB of our platform.

7. Search among the files generated for "`solution1`" to locate the file named "`xaddvectors_hw.h`" .[2] This file contains the information of the registers generated in the slave interface of the peripheral, which the SW application will need to interact with it. For example, the file can be in "`Vitis_HLS/solution1/impl/ip/drivers/AddVectors_v1_0/src/`" .
8. Run "`build_Vivado_project.sh`" . This will invoke Vivado in batch mode to execute the script "`Exercise02_Vivado.tcl`". A folder named "`Vivado`" will be created with the Vivado project. The TCL script will generate a project for the correct board (i.e., not just the FPGA chip), add our user repository at "`IP-catalog`" and create a block diagram including our new IP core. This step requires that the IP core has been decompressed into the "IP-catalog" folder.
9. Synthesize the bitstream. Check the address assigned to the peripheral in the Vivado address editor. Check that the master port of the peripheral is connected to the slave port and can access the DRAM addresses (Figure 1).
10. Extract the bitstream files from the implementation folders. Use the script "`extractBitstream.sh`" as aid to know where the files are usually located. Copy the bitstream files and the contents of the "`SW/`" folder to the Pynq board using `scp`.
11. Connect to the board. Compile the program using `make`. Program the bitstream using the script "`programOverlay.py`" from the previous sessions.
12. Execute the application using `sudo`. Analyze the application output to verify the correct behavior of the peripheral.

In general, we can use TCL scripts to recreate our Vitis HLS and Vivado projects. In this way, we can keep the size of our git repositories more compact.

> **When uploading your deliverables in future sessions, always create TCL scripts to re-build your projects. Do not include the implementation files (i.e., "`Vitis_HLS`" and "`Vivado`" folders) in the repository.**

1.3 Exporting a Vitis HLS project to a TCL script

Vitis HLS stores all the commands executed within the GUI in a script file for each solution. That script file can be used to recreate the complete project from scratch if necessary. If your project employs an HLS directives file, rather than inserting them directly in the C/C++ code, include also in the repository the generated directives file. For example, for the first exercise of this session, the files should be located at:

- "`Exercise01_HLS/solution1/script.tcl`"

[2]The file name will vary according to the name of the top module in the design.

- "Exercise01_HLS/solution1/directives.tcl"

Look into the file "script.tcl" to understand how the project reconstruction works. It is also possible to comment out the lines corresponding to the simulation, synthesis and IP exporting phases, so that the script stops right after recreating the project.

The TCL script can be invoked to recreate the project with a command line similar to the following:

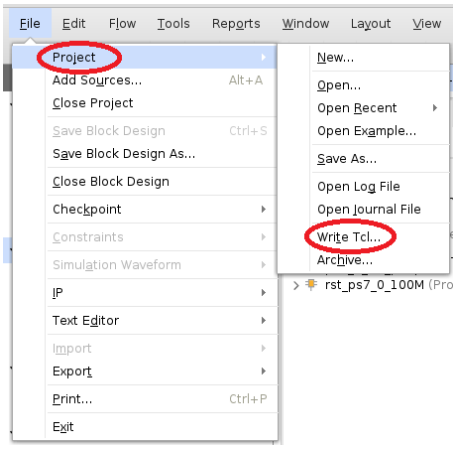
```
$ /scrap/users/Xilinx2022.2/Vitis_HLS/2022.2/bin/vitis_hls -f Exercise01_HLS.tcl
```

1.4 Exporting a Vivado project to a TCL script

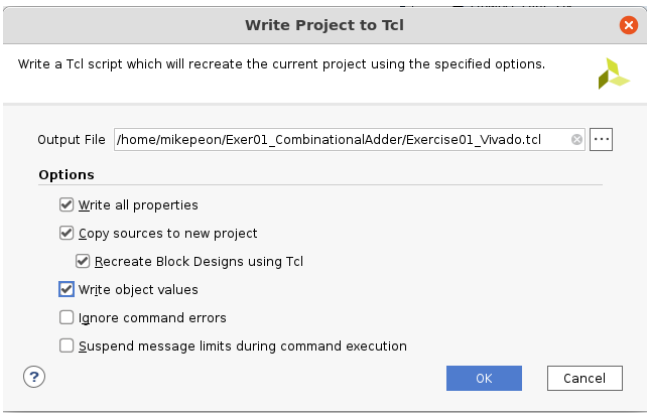
Vivado creates automatically checkpoint files (".dcp") that contain the synthesized versions of the project modules. Use the following TCL command (in Vivado's TCL console) to remove them from the project before exporting it:

```
remove_files  [get_files design_1_wrapper.dcp]
```

Then, select the Write TCL option from the menu:



In the following dialog box, select the first four options to include all the project properties and also the regeneration of the block diagrams from the TCL script. Select an output path that is *one level above* the Vivado project folder. This will facilitate the handling of relative paths when recreating the project. In this example, we create a TCL script named "Exercise01_Vivado.tcl" that will be outside the "Exercise01_Vivado/" folder:



Note that the tool will by default offer to create the output script *inside* the Vivado project folder. For example, in this case, the default path was:

```
/home/mikepeon/Exer01_CombinationalAdder/Exercise01_Vivado/Exercise01_Vivado.tcl
```

The creation of the TCL script can also be accomplished using a TCL command. Verify that the working directory of the TCL console is the current project folder (use `pwd` to check it) or change it if necessary (using `cd your_project_path`). Then, use the following command to generate the TCL script:

```
write_project_tcl -all_properties -dump_project_info {../test.tcl}
```

Before executing the TCL script to reconstruct the Vivado project, verify that all the required IP cores from Vitis HLS are available and decompressed in the corresponding IP folder (e.g., "`IP-catalog/`"). The TCL script can be invoked to recreate the project with a command line similar to the following:

```
$ /scrap/users/Xilinx2022.2/Vivado/2022.2/bin/vivado -mode batch -source
  Exercise01_Vivado.tcl -tclargs --project_name Exercise01_Vivado
```

If you are using Windows on your own computer, you can execute the script from the TCL console inside Vivado itself.

> **When you generate the TCL scripts to reconstruct your projects, verify that the scripts work correctly by reconstructing the project before you delete its files. For that, copy the original project files to a different path. Then, delete the VITIS HLS and Vivado folders and try to reconstruct the projects with the scripts. Open the projects and verify that everything has been correctly reconstructed, including the block diagrams.**

As a final consideration, the checkpoint files are useful in large projects to speed-up the implementation process using out-of-context (OOC) synthesis. Rather than deleting them, the checkpoint files can be stored in a separate repository or in a shared folder that multiple users collaborating in the same project can import locally. This can help to speed-up the implementation process of large projects by re-using the synthesis results for modules that have not been changed.

2 Exercise 1: Combinational adder in HLS

In this exercise, we are going to create a combinational adder that exposes three registers over a slave AXI4 interface: two for the input operands, one for the result. Since the adder is purely combinational, the result of the operation will be available immediately after loading any new value in the operand registers (and the result will be kept until a new operand is loaded).[3]

2.1 Description of an IP core with Vitis HLS

Open Vitis HLS and create a new project. Specify "`xc7z020clg400-1`" as the target FPGA — there is no need to specify the board at this stage; we will do it afterwards when creating the complete SoC project in Vivado. We are going to create a testbed in the file "`main.cpp`" and the implementation of our IP core in a pair of C++ files: "`adder.h`" and "`adder.cpp`" . The content of the files is available in Moodle:

```
--------------------
FILE: HLS/adder.h
--------------------
#ifndef ADDER_H
#define ADDER_H

void Adder(uint32_t a, uint32_t b, uint32_t &res);

#endif

--------------------
FILE: HLS/adder.cpp
--------------------

#include <cstdint>
#include "adder.h"
```

[3] Actually, the peripheral will have a valid signal to indicate that a valid result is available for reading.

```
void Adder(uint32_t a, uint32_t b, uint32_t &res)
{
#pragma HLS INTERFACE ap_ctrl_none port=return
#pragma HLS INTERFACE s_axilite port=a
#pragma HLS INTERFACE s_axilite port=b
#pragma HLS INTERFACE s_axilite port=res

  res = a + b;
}


--------------------
FILE: HLS/main.cpp
--------------------

#include <stdint.h>
#include <stdio.h>
#include "adder.h"

int main(int , char ** )
{
  uint32_t a = 5, b = 3, res;

  Adder(a, b, res);
  printf("Result: %u\n", res);

  return 0;
}
```

The function `Adder()` is the top level of our peripheral. Its arguments define the interface of the HW with the rest of the SoC. The pragmas in the code guide the HLS tool during the inference and implementation of the system functionality. In our example, we indicate that arguments a, b, and res should be allocated registers in a slave interface. We also indicate, with the keyword `ap_ctrl_none` that the peripheral does not have a full handshake-based control mechanism.

The project should appear as in the following figure:

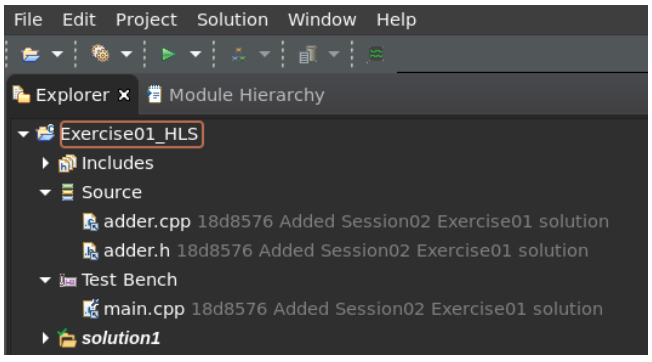


Table 1 shows the registers that are created by Vitis HLS. HLS generates an additional control register that allows the masters that use the peripheral to know if the `res` register contains a valid value. Since this register is of type clear-on-read (COR), it will present the value '1' in its least-significant bit (LSB) bit once a new result is available in `res`. Reading the control register clears the bit; hence, subsequent reads will return '0'.

2.2 Functional simulation of the HW specification in C++

Once the hardware specification and the testbed application are written, we can perform a C simulation to verify that the specified functionality complies with the specification. Go to "Project→Run C Simulation" to compile the sources and execute the testbed. This step is the only case in which

Table 1: Registers created by HLS for the combinational adder IP core. The offsets are memory addresses (as opposed to 32-bit offsets); be careful when using them with pointer arithmetic in a C++ program.

Offset	Register	Description
0x10	a	Read/Write
0x18	b	Read/Write
0x20	res	Read
0x24	res_ap_vld (bit 0)	Read/COR

the C++ files of the HW specification are actually compiled and executed in the desktop PC. In every other case, these files are used only to create a HW specification and are never executed by a processor. Indeed, it is possible to generate an IP core for an FPGA that does not contain any ARM processors — just as any other HW peripheral described in VHDL or Verilog.

2.3 Synthesis of the HW module

Once the functionality of the HW description is correct, synthesize the project to generate an RTL description of the peripheral. In this way, we will go from a C++ specification to an RTL implementation in VHDL or Verilog.

Since we are initially targeting a system frequency of 100 MHz, in the synthesis dialog box select 10 ns as desired clock period. Select also "Vivado IP Flow Target" as the flow target option.

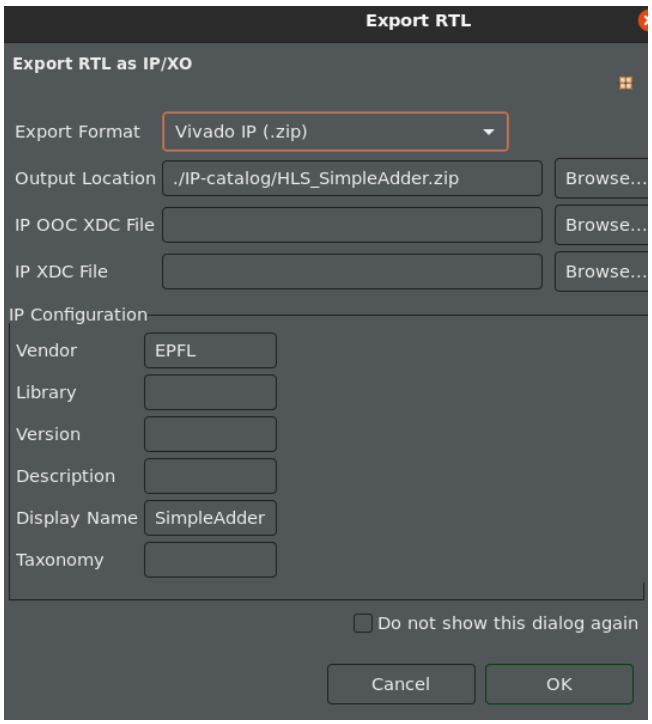
After synthesis, the tool generates a report with the estimated use of resources of the generated module. The tool cannot compute at this stage the exact use of FPGA resources of the peripheral because the subsequent place and route processes depend on the rest of the HW in the system and can affect the final implementation of this module — for example, if the FPGA has high occupation, some routes can be abnormally long and affect the propagation times of the signals in the peripheral.[4] The synthesis report contains also important information regarding the expected performance of the system in terms of latency, initiation interval of loops, dataflow planning and interfaces. Check that the tool infers the correct type of AXI4 interfaces for the ports of our design.

Read carefully the complete synthesis report and try to understand all the information.

2.4 Exporting the RTL design from Vitis HLS

Use the menu option to export the RTL design:

[4] A better estimation of FPGA resources can be obtained by performing an implementation step if required.

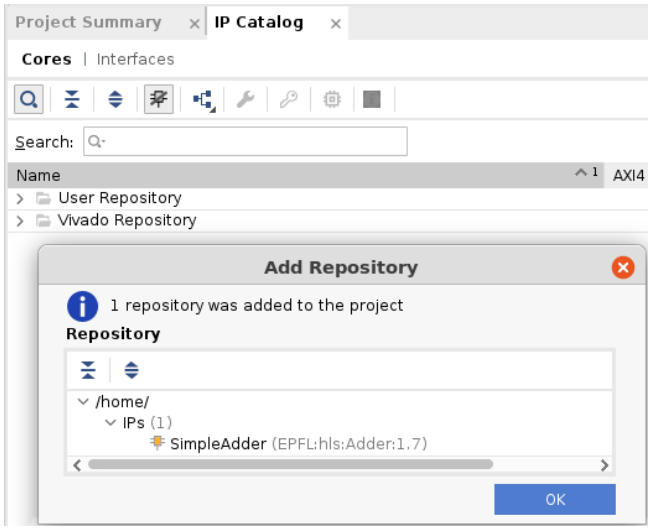


The tool outputs a *zip* file. Select the correct path to the folder that you have designated for your IP catalog. From outside Vitis HLS, uncompress the file there so that Vivado can later access the files. **Vivado will not be able to import IP cores directly from *zip* files.** From the terminal or file manager, look also for the include file that contains the definition of the register offsets in the slave AXI4 interfaces. This file is normally created inside the folder of the current "solution," with a name in the form "`xTOPFUNCNAME_hw.h`" .

Do not include the contents of the uncompressed file in the git repository, just the *zip* file itself.

2.5 Integration of the IP core in a complete SoC with Vivado

Create a new project in Vivado and add a new block diagram containing the "Zynq7 Processing System." Run the block automation wizard to finish the configuration of the processor system. To include our IP catalog, open the IP Catalog view (in the menu, "Window→IP Catalog"). Right-click on the empty space of the window and select "Add Repository". Navigate to the folder in which you have uncompressed the IP core package generated with Vitis HLS in the previous step. If the IP catalog is correctly identified, Vivado will show a dialog with the name of the catalog and a summary of the IP cores found inside:



The IP core can now be integrated in the design as any other IP core from the Xilinx catalog. Since our peripheral has an AXI4 slave interface, Vivado will offer a wizard to connect it automatically to a master and configure its clock and reset signals according to the selected AXI4 bus.

Once the peripheral is added to the SoC diagram, check, in the address editor of Vivado, the physical address at which the peripheral instance has been mapped. Generate the bitstream and extract the files from the implementation folders (use the script "`extractBitstream.sh`" as guidance).

2.6 Accessing the IP core from a Linux application running on the ARM Cortex-A9 cores

Transfer the bitstream files (the `.bit` and `.hwh` files) to the Pynq board using `scp`. Program the bitstream with the script from the previous session: `sudo ./programOverlay.py exercise01.bit`

Download the sources for the SW Linux application for the Pynq board from Moodle. The application structure is similar to the previous examples, but we have added a new class to abstract the interaction with our peripheral. This new class, named `CSimpleAdderDriver`, inherits from the original class `CAccelDriver`. It adds a single method, named `Add()`. This is the function that implements the communication with our peripheral, abstracting all the low-level details from the main application. **During the rest of the examples of this course, we will use a similar structure, where a class abstracts the low-level details of accessing to our peripheral.** In that way, if we have a software application and we want to implement one specific function with an accelerator, we just need to ensure that the driver class offers a direct replacement of the original function prototype.

The application uses the driver classes to map the peripheral registers into the address space of the application in the same way as it was done in the exercises of the previous session. However, these addresses are hidden from the main application.

Check carefully how the structure to access the register offsets is built. Since Vitis HLS generates a register file with many reserved positions, we need to add "padding" positions to ensure that the structure members have the offsets corresponding to the actual registers.

In this case, the main application repeats the same operation with random parameters, each time comparing the result obtained from the accelerator and computing in SW (just a simple addition). Notice that we should verify the status of the `res_ap_vld` register before accepting the result value, rather than reading the register directly.

3 Exercise 2: Vector adder in HLS

In this exercise, we are going to create a peripheral that adds the elements of two vectors into a third one. The functionality implemented is:

$$\text{output}[i] = \text{input1}[i] + \text{input2}[i] + \text{acc} \tag{1}$$

The peripheral interface is defined by the following C++ function prototype:

```
void AddVectors(ap_uint<32> * input1, ap_uint<32> * input2, ap_uint<32> * output,
  ap_uint<32> length, ap_uint<32> accum);
```

We use the template `ap_uint<32>` to specify the width of the ports. This enables the specification of the exact bitwidth required in the datapath for the implementation of our algorithms. This function prototype will guide Vitis HLS to generate the following interface:

- One AXI4 slave interface to provide access to a register file with:
  - A register to store the starting address of `input1`.
  - A register to store the starting address of `input2`.

  - A register to store the starting address of `output`.
  - A register to store the length of the vectors.
  - A register to store the value of the accumulator.
  - Additionally, we will ask Vitis HLS to generate registers to implement a handshake-based synchronization protocol, so that the SW running on the system processors can know the status of the peripheral (i.e., when it is idle or still performing an operation).
- One AXI4 master interface to perform the necessary accesses to the two input vectors and the output vector.

3.1 Description of the IP core with Vitis HLS

The complete HLS specification of the vector adder peripheral is as follows:

```
#include <ap_int.h>

// output[i] = input1[i] + input2[i] + accum;
void AddVectors(ap_uint<32> * input1, ap_uint<32> * input2, ap_uint<32> * output,
                ap_uint<32> length, ap_uint<32> accum)
{
#pragma HLS INTERFACE s_axilite port=length
#pragma HLS INTERFACE s_axilite port=accum
#pragma HLS INTERFACE s_axilite port=return
#pragma HLS INTERFACE m_axi depth=1024 port=input1 offset=slave
#pragma HLS INTERFACE m_axi depth=1024 port=input2 offset=slave
#pragma HLS INTERFACE m_axi depth=1024 port=output offset=slave

  if (length > 8*1024*1024)
    length = 8*1024*1024;

  for (ap_uint<32> ii = 0; ii < length; ++ ii) {
    output[ii] = input1[ii] + input2[ii] + accum;
  }
}
```

The pragma sentences indicate Vitis HLS how we want to generate our peripheral interface. For the vector parameters, we indicate that it has to generate one AXI4 interface (`m_axi`) and one register mapped on the slave interface to contain a pointer. The specification of an interface for the return values based on the AXI4 slave interface (`port=return`) also activates the default handshake-based protocol.

The handshake protocol to control the status of an AXI4 peripheral generated by Vitis HLS is based on the use of three bits in a status and control register: `ap_start`, `ap_done` and `ap_idle`. The peripheral is ready to accept a new operation when the idle bit is "1". The AXI4 master (in this case, the ARM core that is executing our SW application) can write the new parameters in the registers. The operation starts when the start bit is set to "1". The operation is completed when the done bit becomes "1".

The handshake is completed when done is "1" and the control register is read. The peripheral automatically sets done to "0" (because it is defined as COR) and completes the handshake by setting start to "0" as well. If the SW application is polling the done bit in a loop, it will read the value "1" once, since the peripheral will automatically revert its value to "0". The application can determine if the peripheral can accept a new operation (after the previous one was completed) verifying that the idle bit is "1". Figure 2 shows the default handshake behavior of the synthesized peripherals.

Vitis HLS takes care of designing the slave and master AXI4 interfaces, the dataflow required to implement the operation on the data read, and the state machines to coordinate all the elements and data movements. Table 3 details the registers generated by Vitis HLS as interface of the peripheral.

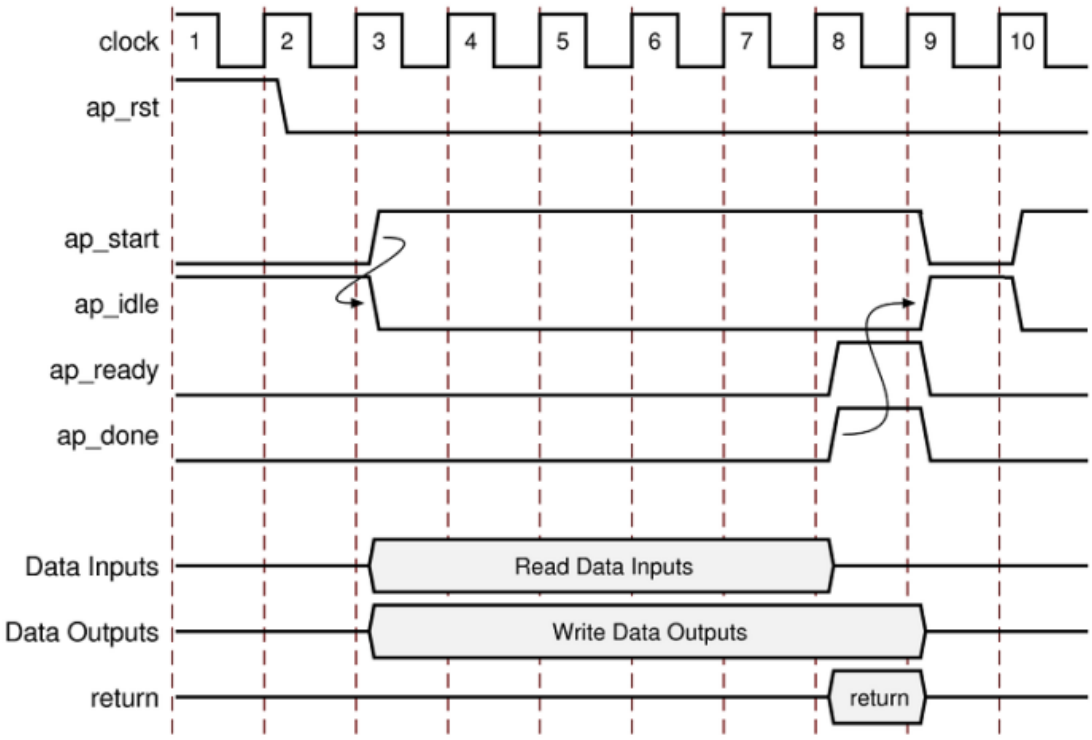


Figure 2: Default handshake protocol of the synthesized peripherals. Source: Xilinx UG1399, v 2022.2, page 198.

Table 2: Registers created by HLS for the vector adder IP core. The offsets are memory addresses. Be careful when using them with pointer arithmetic in a C++ program. COR = Clear-on-read. COH = Clear-on-handshake. Consult the generated `xaddvectors_hw.h` file for the complete specification.

Offset	Register	Description
0x00	Control & status	
	bit 0: ap_start	Read/Write (COH)
	bit 1: ap_done	Read (COR)
	bit 2: ap_idle	Read
	bit 3: ap_ready	Read (COR)
0x10	Pointer to input1	Read/Write
0x1C	Pointer to input2	Read/Write
0x28	Pointer to output	Read/Write
0x34	length	Read/Write
0x3C	accum	Read/Write

3.2 Functional simulation and synthesis of the HW

Add to the Vitis HLS project the files of the vector adder specification and the testbed.[5] Synthesize the C specification to RTL and export the RTL implementation to an IP catalog.

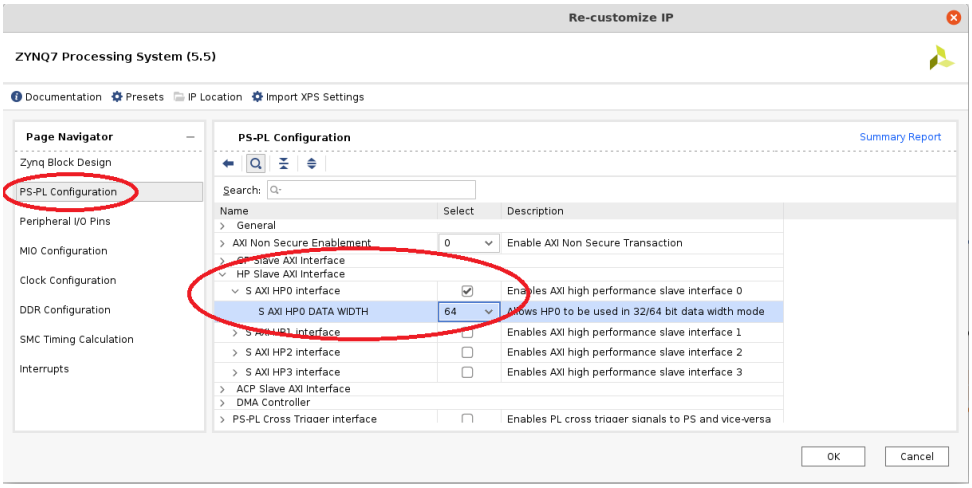
> ▶ **Question:** *Check the Synthesis Summary Report and verify the number and type of ports and interfaces generated in the HW Interfaces window. Analyze the performance report and check for any violations. Are there any performance issues in the design? Verify how many master ports are being used. How could we improve the performance of the peripheral?*

3.3 Integration of an IP core with AXI4 master interfaces in a complete SoC with Vivado

Create a new Vivado project and import the vector adder IP core. Connect the slave interface of the vector adder to the master interface of the Zynq7 using the connection automation wizard.

How can we connect the master interface of the vector adder?

In order to connect the master interfaces of our peripherals to the Zynq7 block, we need to activate the slave ports in the Zynq7 — the interfaces are deactivated by default to save energy in designs that do not require them. Double-click on the Zynq7 block to open its properties dialog. Select the tab "PS-PL Configuration" and activate one of the high-performance slave AXI interfaces (in this case, HP0 is the first port):



If one IP core implements more than one AXI4 master interface (e.g., to be able to perform parallel accesses), we can activate additional ports in the Zynq7 block (up to a maximum of 4), or we can force them to share the same bus by introducing an AXI4 interconnect with multiple slave ports and a single slave port.

Synthesize the design and generate the bitstream.

3.4 SW access to a peripheral with DMA in Linux

Our peripheral can generate streams of accesses to vectors in main memory autonomously, without intervention of the processors. Because of this, it is said to implement direct memory access (DMA). However, in an operating system such as Linux, in which multiple SW applications run concurrently with the HW peripherals, a peripheral cannot access freely any random position of memory. Instead, the applications must allocate DMA-suitable memory using specialized Linux kernel functions and use these memory areas to place the data that the peripherals will interact with.

[5] The source files can be downloaded from Moodle.

In the source code for Linux of this example, you can see how the driver class performs two different tasks at initialization: mapping the peripheral registers into the application address space, and reserving DMA-suitable memory for the vectors via the function `cma_alloc()`. Additionally, the `CAccelDriver` class calls the function `cma_get_phy_addr()` to obtain the physical address that corresponds to the allocated DMA buffers. We will explore in later sessions why this is necessary. For now, it will suffice with knowing that the application always has to use the pointers allocated via `cma_alloc()` (i.e., the virtual addresses), whereas the HW peripherals must always work with the corresponding physical addresses. The class `CAccelDriver` stores internally the correspondence between the virtual address that the application uses and the physical addresses used by the accelerators. The translation process is hidden inside the driver class functions so that the main application does not need to deal with the low-level details.

▶ **Question:** *(Additional work) Integrate a SystemILA module in the design to capture transactions in the slave and the master interfaces of the vector adder. Then, analyze the transactions and compare the addresses used in them. Does the peripheral perform burst transactions (which are more efficient)?*

The SW application implements the handshake protocol inside the function `CallAccel()`, which isolates the rest of the SW infrastructure from the specific behavior of the HW accelerator. Observe how the bits of the different registers are accessed using bit masks. It is important to understand well how this mechanism works. For example, if we want to update the start bit of the control register, which corresponds to bit 0, we have first to read the value of the register, modify only the start bit, and write back the complete register value. In this way, we will not modify other control bits in the same register.

The application also uses the standard function `clock_gettime()` to measure execution times. The `CalcTimeDiff()` function in the "`utils.cpp`" file returns the difference in ns between two measurements.

▶ **Question:** *Measure the execution time of the SW and HW versions for different vector lengths (the maximum vector length is* 8 388 608*). Is our HW module capable of accelerating the addition of the two vectors for any vector size? Reason the possible causes for your previous answers. How can we improve the performance of our accelerator?*

4 Exercise 3: Computing a moving average over a vector

In this exercise, we are going to design an accelerator that implements the moving average of a vector and stores the results into an output vector:

$$\text{output}[i] = (\text{input}[i] + \text{input}[i+1] + ... + \text{input}[i+6])/7 \tag{2}$$

The window length is the number of consecutive elements of the vector considered for the computation of each average value. To simplify our work, we will fix the length of the moving window to 7. After each calculation, the window is moved along the vector by one position and the computation is repeated. This operation continues until the end of the input vector. The result is a vector of averages.

The code to perform the computation in software is available in Moodle. Use that code as basis to start your implementation. There are two versions in Moodle for the SW implementation:

1. The first one reads 7 input elements to produce each output element.
2. The second (fast) version accumulates the first 7 elements of the input version. Then, it adds the new element and subtracts the oldest element in an accumulator. The value of the accumulator after the update is used to compute the moving average for the current output position.

Run the code in the Pynq board to measure the execution time of both SW versions.

Table 3: Registers created by Vitis HLS for the moving average IP core.

Offset	Register	Description
0x00	Control & status	
	bit 0: ap_start	Read/Write (COH)
	bit 1: ap_done	Read (COR)
	bit 2: ap_idle	Read
	bit 3: ap_ready	Read (COR)
0x10	Pointer to input	Read/Write
0x1c	Pointer to output	Read/Write
0x28	Length	Read/Write

4.1 Description of the IP core with Vitis HLS

The peripheral's interface is defined by the following C++ function prototype:

```
void MovingAvg(ap_uint<32> *input, ap_uint<32> *output, ap_uint<32> length);
```

We use the template `ap_uint<32>` to specify the width of the ports, which in this case corresponds to the `uint32_t` datatype in C++. In this way, we can easily adapt the datapath of our peripheral to any data width. This function prototype will guide Vitis HLS to generate the following interface:

- One AXI4 lite slave interface with the following registers:
  - A register to store the starting address of the input vector.
  - A register to store the starting address of the output vector.
  - A register to store the length of the input vector.
  - Control/status registers to implement a handshake-based synchronization protocol, so that the SW running on the system processors can know the status of the peripheral.
- One (or more) AXI4 master interface to perform the necessary accesses to the input and output vectors.

4.2 Functional simulation and synthesis of the HW

Write the specification of the peripheral using Vitis HLS to implement the described functionality. Then, write a `main()` function to test the implementation of the moving average functionality of the peripheral, inside Vitis HLS. Generate a few simple cases for which the output result can be easily precomputed, and some more complex cases. Once you are sure that the peripheral specification is correct, synthesize the C/C++ specification to RTL and export the RTL implementation to an IP catalog. Table 3 is an example of the register map generated by Vitis HLS.

4.3 Integration of the IP core in a complete SoC with Vivado

Create a new Vivado project and import the moving average IP core. Connect the slave and the master ports to the Zynq Processing System. Then, synthesize the design and generate the bitstream.

4.4 SW in Linux for the Pynq board

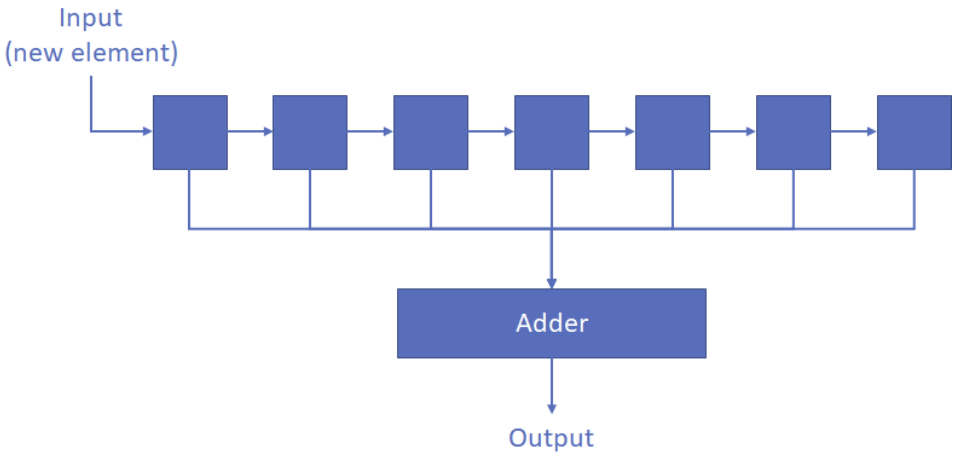
Extend the C++ application to test your moving average IP from Linux and compare its performance with both SW versions. Use the functions presented in the previous exercise (`clock_gettime()`) to measure the execution time of the function.

4.5 Optimize the number of memory accesses

Once the application is working properly, add a System ILA to the Vivado project and connect it to the AXI4 master interface of the IP core. Analyze the number of accesses to DRAM performed by the peripheral. Is it reasonable? How could we improve its behavior? Generate a new design aimed at reducing the number of memory accesses, implement it, verify its functionality (using the same testbenches than before both in Vitis HLS and in Linux) and measure its performance. Is it faster? Check the intended peripheral behavior using System ILA to inspect the bus transactions initiated by the peripheral.

4.6 TASK: Redesign your accelerator starting from a HW architecture

Rather than forcing Vitis HLS to produce an architecture for the C code we used in the SW version, design a new algorithmic implementation to reflect the following HW architecture that uses a shift register and a tree of adders:



Implement this new version and characterize its performance in comparison with your previous implementations. Use the System ILA to understand why this version has better performance than the previous one, and than the SW versions used up to now.

Hint: Look into the *Inferred Burst and Widening Missed* section of the *AXI Burst Information* of the *Synthesis Summary Report*. You may find this information about missed bursts useful: `https://docs.xilinx.com/r/2022.2-English/ug1448-hls-guidance/Burst-Inference-Failure-8`.