

SmartHLS™ Training for Microchip PolarFire® SoC Flow

Revision 3.0
January 2024



MICROCHIP

1. Revision History

Revision	Date	Changes
1.0	June 21, 2022	Initial version
1.1	June 24, 2022	<ul style="list-style-type: none">Corrected figures and runtime after correcting the source files for Section 8Fixed PolarFire® SoC registered trademark position
2.0	July 14, 2023	Updated designs for Libero 2023.2
3.0	Jan 22, 2024	Updated for Libero 2024.1

2. Table of Contents

[1 Revision History](#)

[2 Table of Contents](#)

3 Requirements

3.1 Software Requirements

3.2 Download Training Design Files

3.3 PolarFire® SoC Icicle kit Setup

4 Introduction

5 Hardware Acceleration: Software/Hardware Partitioning

6 Application Example: Vector Addition

6.1 Creating a new Project

6.2 SmartHLS IP Flow

6.2.1 Vector Add: Design Description

6.2.2 Compile Software to Hardware Reports

6.2.3 Generated Verilog Output

6.2.4 Running Co-Simulation

6.2.5 Libero Synthesis and Hardware Report

6.2.6 SmartHLS Generated Software Driver APIs

6.3 SmartHLS SoC Flow

6.3.1 SoC Data Transfer Methods

6.4 SmartHLS Memory Allocation Library

7 Running Vector-Add Reference SoC Generation on the Board

8 Integrating SmartHLS into an Existing SoC design

8.1 Motivation

8.2 Example: Integrate SmartHLS into the PolarFire® SoC Icicle Kit Reference Design

8.3 Custom Flow Integration

8.4 Simple Image Processing Example

8.5 Flashing PolarFire® SoC Icicle Kit Reference Design

8.6 Extract the Icicle Kit Reference Design Files

8.7 Compiling the hardware

8.8 Programming the FPGA bitstream

[8.8.1 Chaining HW modules using CPU shared memory \(main.simple.cpp\)](#)

[8.8.2 CPU usage \(main.cpu_usage.cpp\)](#)

[8.8.3 Non-blocking hardware execution](#)

[8.8.4 Chaining using FIFOs](#)

[8.8.5 Summary](#)

[9 Current limitations of the SoC flow](#)

3. Requirements

This section provides all the requirements needed before starting the training.

3.1 Software Requirements

You should install the following software:

- SmartHLS™ 2024.1 or later: this is packaged with Libero
- Libero® SoC 2024.1 (with Modelsim Pro 2021.3) or later
 - [Libero Download Page](#)
- A terminal emulator such as PuTTY
 - [Windows Download](#)

This document uses the Windows versions of Libero® SoC 2024.1 and SmartHLS 2024.1. Depending on the version you use, the results generated from your Libero® SoC and SmartHLS could be slightly different from that presented in this document.

3.2 Download Training Design Files

Download the training design files in advance:

- Linux image: [core-image-minimal-dev-icicle-kit-es.wic.gz](#) (216MB)
 - Github: <https://github.com/polarfire-soc/meta-polarfire-soc-yocto-bsp/releases>
 - SHA256: 4a1406ba9e764a94026fcea2ee8fbb84f91384e953e7ba6176fcb7dadcbc5522
- Training design files for Section 7 can be found on Github under [Training4/vector_add_soc](#)
- Training design files for Section 8 can be found on Github under [Training4/icicle-kit-reference-design](#)
- The pre-compiled bitstreams can be found on Github under [Training4/SmartHLS_Training4_Jobs](#)
 - Alternatively, users may regenerate the bitstreams using a .tcl script by following the instructions in section 8.7.

3.3 PolarFire® SoC Icicle kit Setup

Later parts of the training involve running steps on the Icicle kit board. The following hardware is required:

- PolarFire® SoC FPGA Icicle Kit ([MPFS-ICICLE-KIT-ES](#))
- 2 micro-USB cables for serial communication and flashing the Linux image
- Either a FlashPro6 external programmer or a micro-USB cable for the embedded FlashPro6

- Ethernet cable for network connection to the board for SSH access

This training will cover the following sections in the [SmartHLS user guide](#): [SoC Features](#), [AXI4 Initiator Interface](#), [AXI4Target Interface](#), [Driver Functions for AXI4 Target](#), and [User-defined SmartDesigns](#).



We will use this cursor symbol throughout this tutorial to indicate sections where you need to perform actions to follow along.

4. Introduction

Our previous trainings focused on using SmartHLS as an IP generator, where SmartHLS takes as input a C++ program and generates a SmartDesign IP component. The user then instantiates the generated SmartDesign IP component into their SmartDesign system in Libero before running synthesis, place and route, and ultimately programming the FPGA.

In this training, we introduce the SmartHLS SoC flow targeting a PolarFire SoC FPGA device as shown in Figure 4-1. The SoC flow will now generate C++ software drivers with APIs that can be used to control the generated IP cores from the Microprocessor Sub-System (MSS). Given the generated software drivers and the generated SmartDesign component's AXI4 interfaces, the SmartHLS IP block can be easily integrated into an existing PolarFire SoC system.

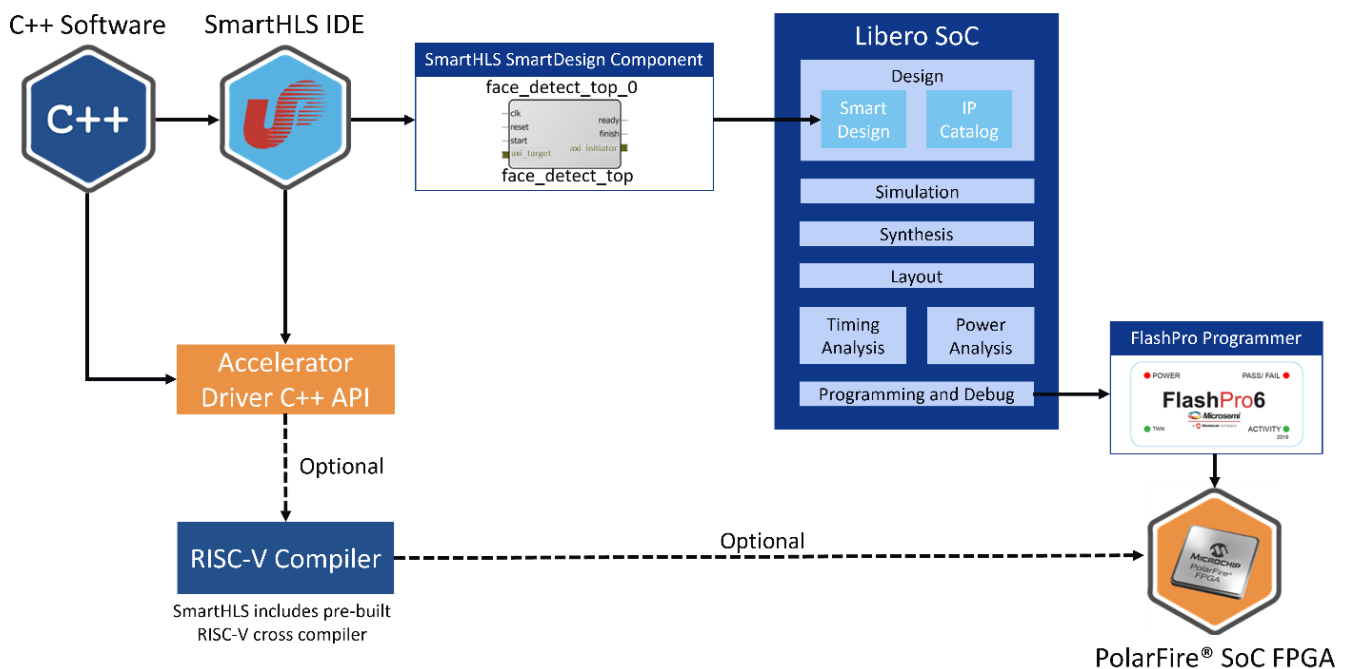


Figure 4-1 SmartHLS IP Flow from Software to Hardware on FPGA

5. Hardware Acceleration: Software/Hardware Partitioning

SmartHLS SoC flow also supports partitioning the input C++ program between software running on the MSS processor while user-specified functions are synthesized by SmartHLS into FPGA hardware cores. The SmartHLS SoC flow refers to FPGA IP cores synthesized by SmartHLS from user-specified C++ functions as *hardware accelerators*. FPGA hardware accelerators typically see a performance speedup (acceleration) compared to the original C++ software running on the MSS processor due to parallelism in the FPGA fabric.

As shown in Figure 5-1, SmartHLS takes the C++ program as input and performs user-guided hardware/software partitioning. For the software partition, SmartHLS automatically transforms the original C++ software program by replacing the user-specified functions with calls to FPGA hardware using the generated software drivers. Then SmartHLS compiles the software using the RISC-V compiler toolchain to get a RISC-V software binary to run on the MSS. For the hardware partition, SmartHLS generates the hardware accelerators for user-specified C++ functions and then connects these accelerators to the MSS via an AXI4 interconnect in a generated Reference SoC hardware system targeting the PolarFire SoC.

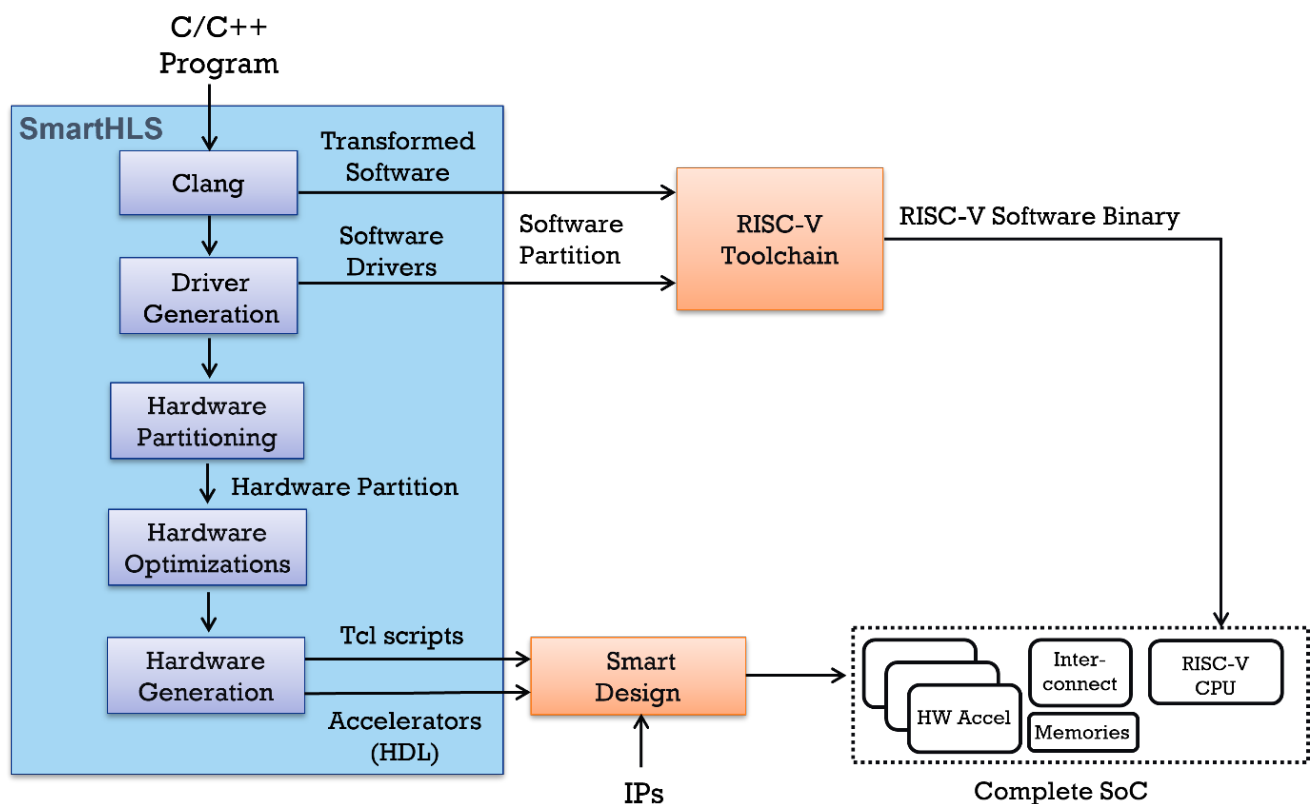


Figure 5-1 SmartHLS SoC flow Details

Alongside the accelerators, SmartHLS also produces a Tcl script for easy SmartDesign integration, and C++ accelerator driver code to control the accelerators, which can be directly called by the software program running on the MSS.

All SmartHLS-generated hardware accelerators will implement an AXI4 target interface, with memory-mapped registers for control and data transfer. Therefore, the accelerators can be instantiated into any existing AXI4-compatible SoC design. Figure 5-2 below gives a system diagram of the PolarFire SoC Reference SoC that can be generated by SmartHLS.

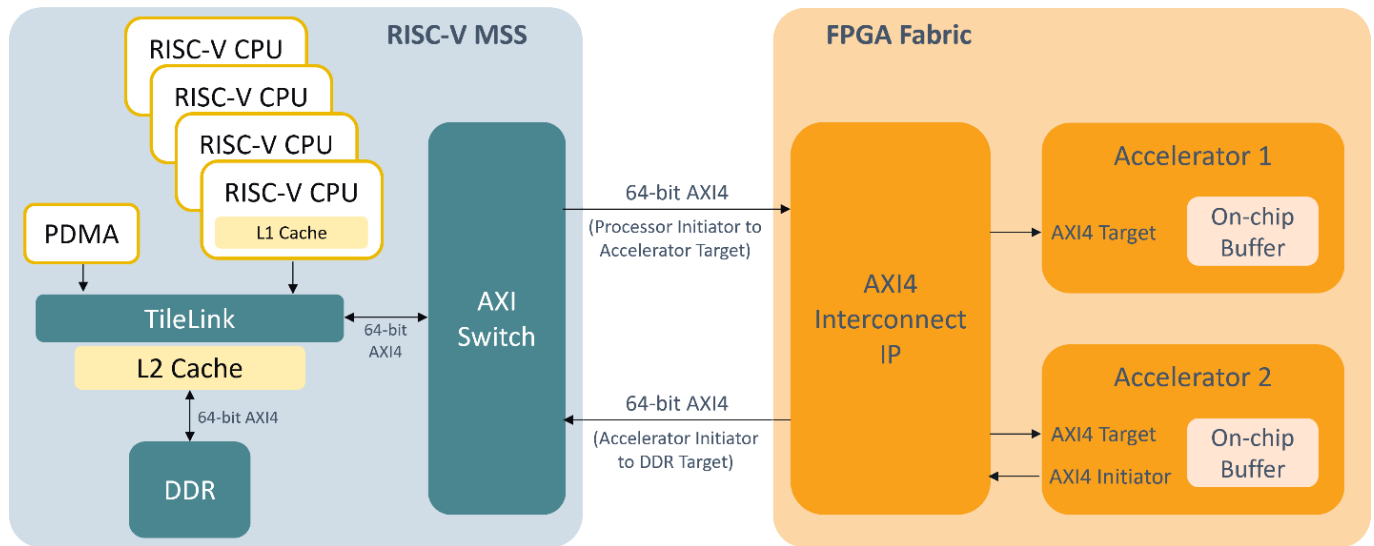


Figure 5-2 SmartHLS Generated Reference SoC Architecture Overview

On the left, we have the PolarFire SoC Microprocessor Sub-System (MSS), which contains four RISC-V processors running the user's software on Linux. On the right, we have one or more hardware accelerators. The processor communicates with the accelerators using a memory-mapped AXI interconnect. Additional hardware accelerators can be added, if there is room in the memory-map, by simply attaching them to the AXI interconnect.

6. Application Example: Vector Addition

In this section, you will use SmartHLS SoC flow to target a vector addition program written in C++ to the PolarFire SoC FPGA. The vector addition will take two input arrays, add these two arrays element-by-element, and store the sum for each element into the output array.

6.1 Creating a new Project



First start the SmartHLS IDE.

On Windows, this can be done by double-clicking on the SmartHLS shortcut either in the start menu or the desktop.

On Linux, make sure that `$(SMARTHLS_INSTALL_DIR)/SmartHLS/bin` is on your PATH and the SmartHLS IDE can be opened by running the following command:

```
> shls -g
```

You will first see a dialog box to select a workspace directory as shown in Figure 6-1 below. You can use the default workspace for all parts of this tutorial by clicking on OK.

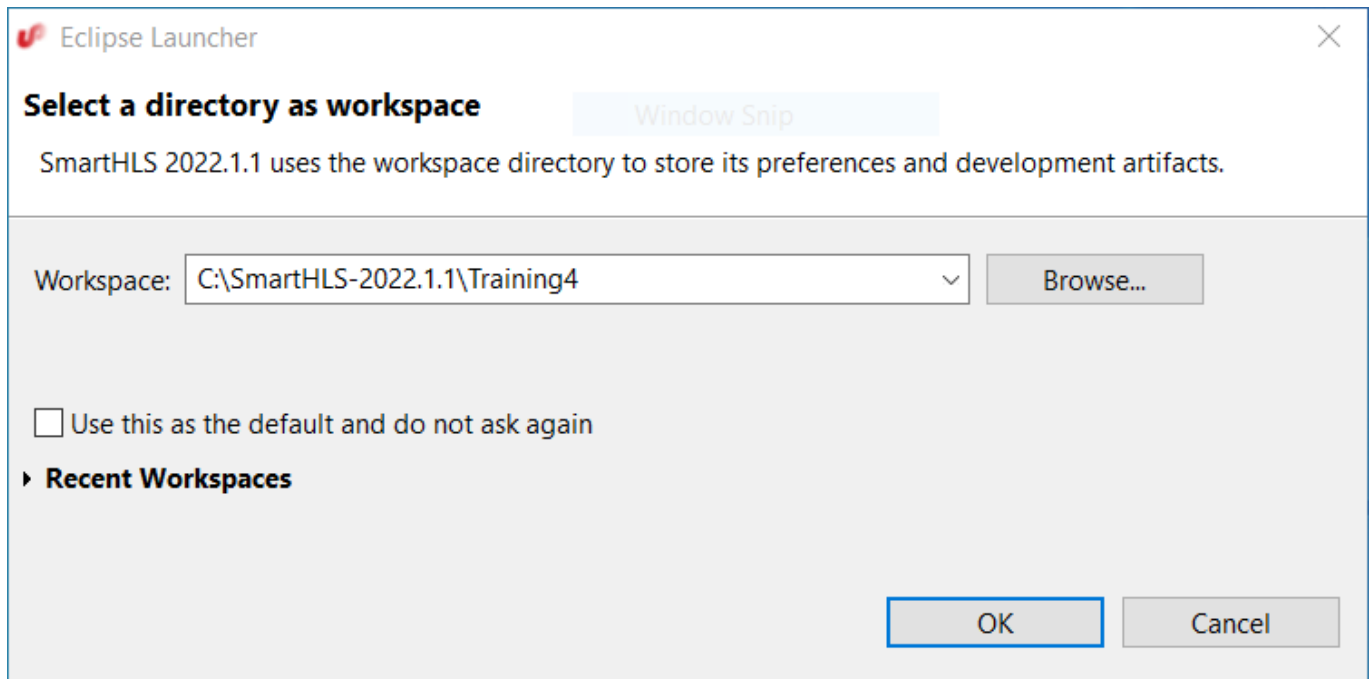


Figure 6-1 Choosing a Workspace

Warning: Make sure there are no spaces in your workspace path. Otherwise, SmartHLS will give an error when running synthesis. Also, keep the path short as there's a 90-character limit on file names on Windows.



Once the SmartHLS IDE opens, under the File menu, choose *New* and then *SmartHLS C/C++ Project* as shown below in Figure 6-2.

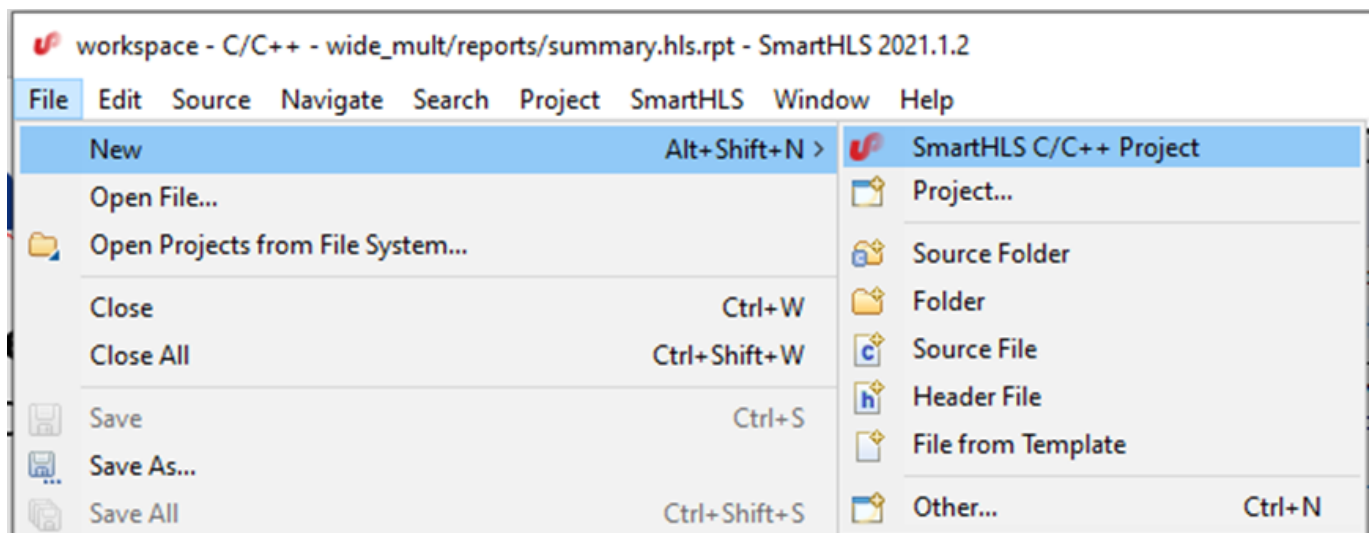


Figure 6-2 Creating a new SmartHLS C/C++ Project



For the project name, enter `vector_add_soc` and select "Example Project 5: Vector Add" from the list of example projects, as shown in Figure 6-3. Then click on *Next*.

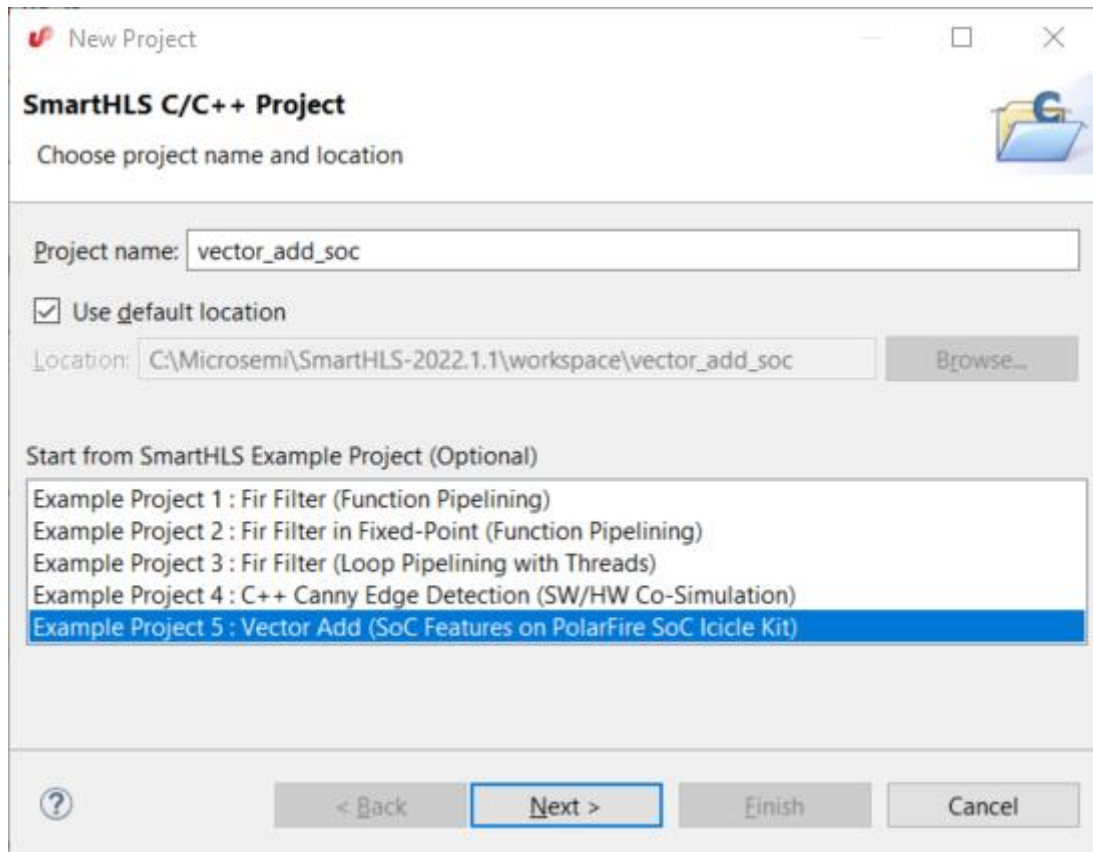


Figure 6-3 Creating Vector Add SmarthLS Project



Finally, to complete the project creation, you will choose the FPGA device you intend to target. Use the selections shown in Figure 6-4 for *FPGA Family* choose *PolarFireSoC*. For *FPGA Device*, choose *MPFS250T_ES-FCVG484E on Icicle Board*. Click on *Finish* when you are done. SmarthLS may take a few moments to create the project.

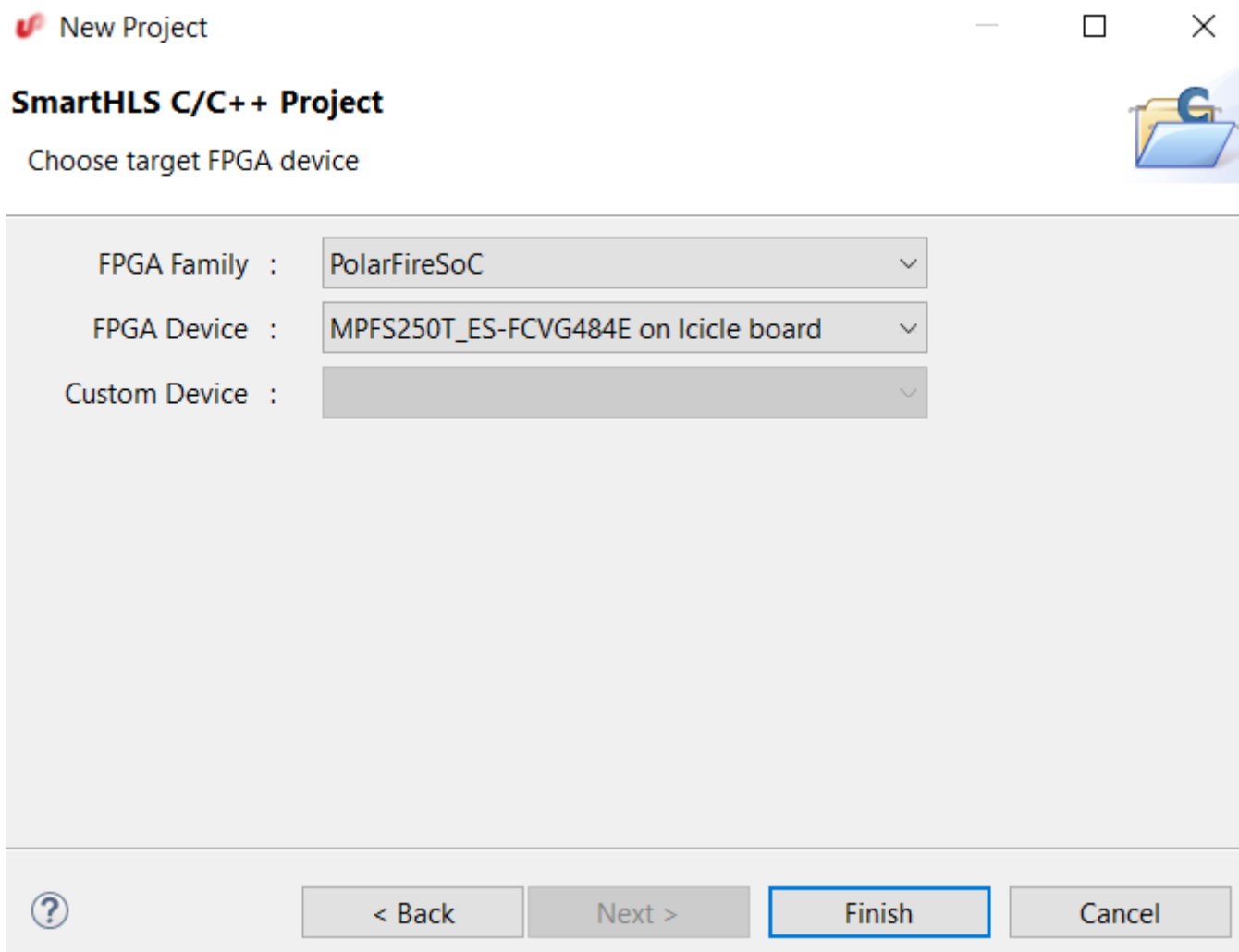


Figure 6-4 Choosing FPGA device, select SoC IP Flow



If this is the first time you are using SmartHLS, you will need to set up the paths to ModelSim (and Microsemi Libero® for later parts of this tutorial). To setup the paths, click on *SmarterHLS* on the top menu bar, then click on *Tool Path Settings*. Once the dialog opens, set the paths for *ModelSim Simulator* and *Microsemi Libero® SoC* as shown in Figure 6-5 and click *OK*.

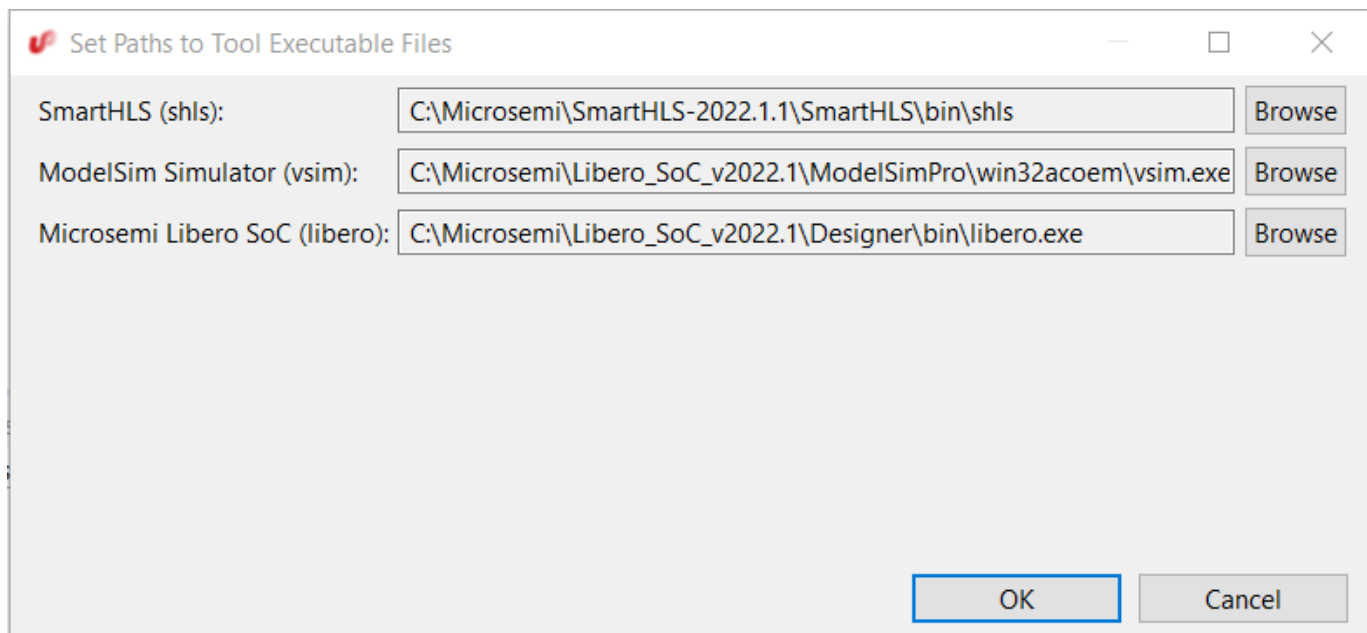


Figure 6-5 SmartHLS Tool Path Settings

An important panel of the SmartHLS IDE is the *Project Explorer* on the left side of the window as shown in Figure 6-6. We will use the project explorer throughout this tutorial to view source files and synthesis reports.



Click on the small arrow icon to expand the `vector_add_soc` project. You can now double click any of the source files, such as `vector_add_soc.cpp`, and you will see the source file appear in the main panel to the right of the *Project Explorer*.

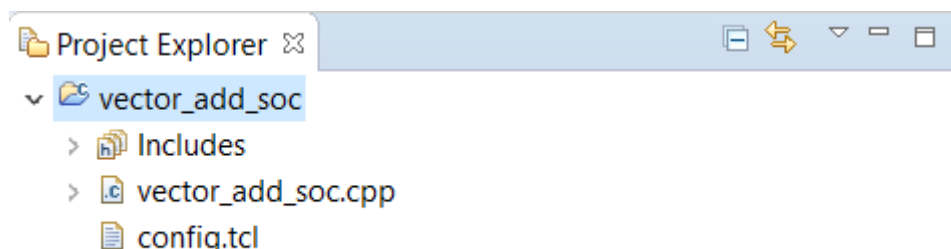


Figure 6-6 Project Explorer for browsing source files and reports

6.2 SmartHLS IP Flow

The *SmartHLS IP flow* refers to when SmartHLS generates a hardware IP core that can be integrated into a user's SmartDesign system in Libero. SmartHLS can go one step further and integrate the generated IP core, which we refer to as an accelerator, into a Reference SoC targeting PolarFire SoC. We call this flow the *SmartHLS SoC flow* (described later in Section 6.3).

Once a SmartHLS project is created, you should always open one of the source files (such as `vector_add_soc.cpp`) or double-click on the `vector_add_soc` directory in the Project Explorer pane (Figure 6-6). This will make `vector_add_soc` the active project. You can also see the active project name in the Console tab after running a SmartHLS command in Figure 6-7.

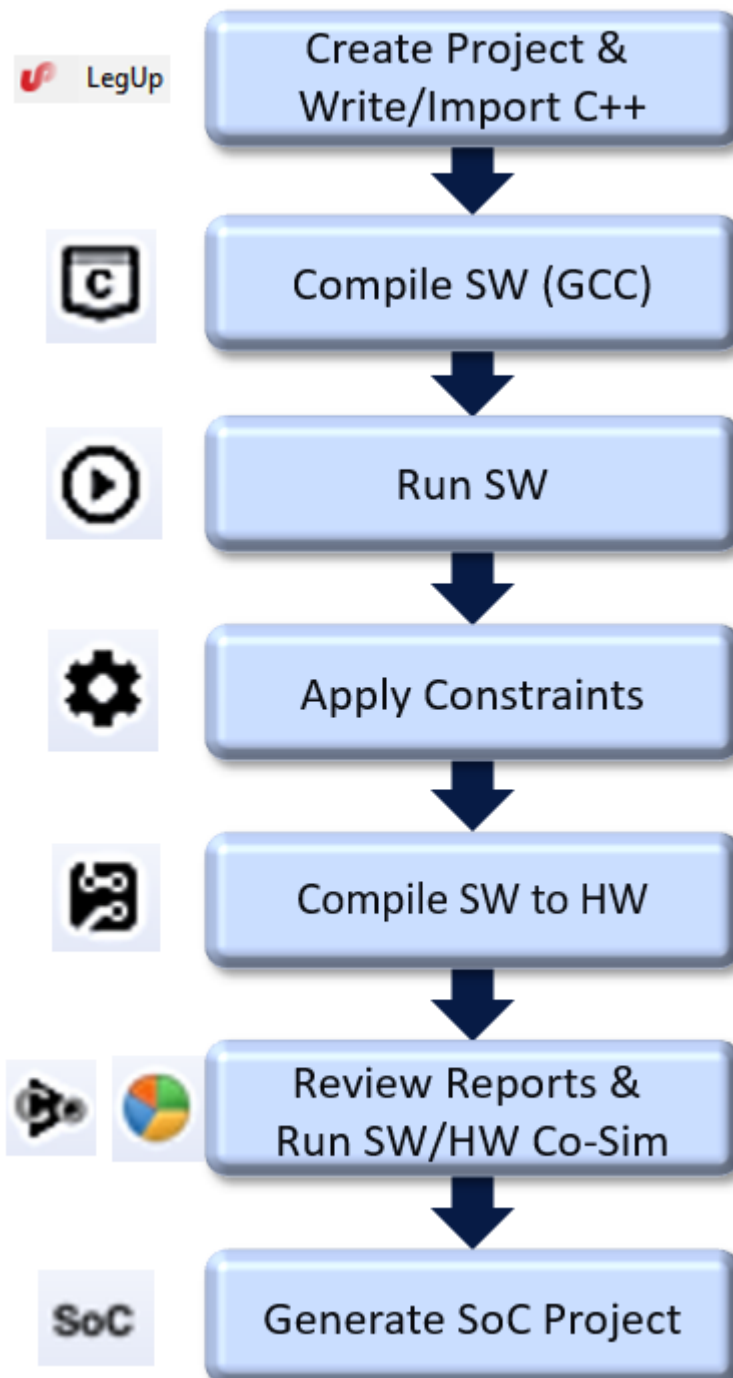


Figure 6-9 SmartHLS Workflow

Figure 6-9 summarizes the steps for the SmartHLS flow. We initially create a SmartHLS project and follow a standard software development flow on the C++ (compile/run/debug). Then we apply HLS constraints using SmartHLS C++ pragmas. These include HLS constraints covered in previous trainings such as the target clock period, loop optimizations, and memory configuration. For more details see our [optimization guide](#).

There are new SmartHLS **interface** pragmas used to specify the data transfer method for each top-level function argument. These pragmas specify how the generated hardware accelerators interface with the rest of the SoC. Figure 6-10 below contains a summary of the SmartHLS pragmas used in the vector-add example. More details on the interfaces will be covered in Section 6.3.1. For a complete pragma reference, see our [pragma guide](#).

In Figure 6-9, after specifying the argument interfaces, we can compile the software into a hardware IP core using SmartHLS, and review reports about the generated hardware. Then we run software/hardware co-simulation to verify the generated hardware. Finally, we can try synthesizing the generated IP, and integrate the IP into an existing hardware system using the output SmartDesign TCL script, software drivers, and Verilog for the FPGA hardware accelerators. The last SoC step of the workflow, “Generate SoC Project” will be covered in Section 6.3.

Pragma	Description
#pragma HLS function top	Identify the function being compiled into an accelerator
#pragma HLS interface default \ type(<axi_target simple>)	Set the default interface type, including interface type for control and arguments.
#pragma HLS interface control \ type(<axi_target simple>)	Set the default module control interface type. The control interface is used for starting the accelerator, reading completion status and retrieving return data.
#pragma HLS interface argument(<ARGUMENT_NAME>) \ type(axi_target) \ num_elements(<NUM_ARRAY_ELEMENTS>) \ dma(true false)	Set pointer argument of ARGUMENT_NAME to use axi_target as the interface. NUM_ARRAY_ELEMENTS indicates the number of elements in the array. If dma(true), DMA will be used for transferring data. More details in Section 6.3.1.2.
#pragma HLS interface argument(<ARGUMENT_NAME>) \ type(axi_initiator) \ ptr_addr_interface(<simple axi_target>) \ num_elements(<NUM_ARRAY_ELEMENTS>)	Set Pointer argument of ARGUMENT_NAME to use axi_initiator as the interface. ptr_addr_interface is the interface that receives the address from MSS. NUM_ARRAY_ELEMENTS indicates the number of elements in the array.

Figure 6-10 Summary of Pragmas Used in Vector-Add

6.2.1 Vector Add: Design Description

We can now browse through the code in `vector_add_soc.cpp` file. We will first look at line 25 of the `vector_add_sw` C++ function as shown in Figure 6-11. The function has three pointer arguments. Two input arrays: `a` and `b`, and an output array `result`. Each array is expressed in C++ as a pointer to an int (32-bit) array of size: `SIZE`. The loop on line 26 performs a vector addition of `a` and `b` and stores the sum in the `result` array.

```

24 // The core logic of this example
25 void vector_add_sw(int a, int b, int result) {

```

```

26     for (int i = 0; i < SIZE; i++) {
27         result[i] = a[i] + b[i];
28     }
29 }

```

Figure 6-11 Core Logic of Vector-Add

Now we look on line 70 at the `vector_add_axi_target_memcpy` top-level C++ function as shown in Figure 6-12.

```

70 void vector_add_axi_target_memcpy(int a, int b, int result) {
71     #pragma HLS function top
72     #pragma HLS interface control type(axi_target)
73     #pragma HLS interface argument(a) type(axi_target) num_elements(SIZE)
74     #pragma HLS interface argument(b) type(axi_target) num_elements(SIZE)
75     #pragma HLS interface argument(result) type(axi_target)
num_elements(SIZE)
76     vector_add_sw(a, b, result);
77 }

```

Figure 6-12 Accelerator Version of Vector-Add

We use SmartHLS to compile the vector addition into a hardware accelerator running on the FPGA by adding SmartHLS pragmas. Immediately following the function prototype, "`#pragma HLS function top`", on line 71 specifies that the `vector_adder_axi_target_memcpy` top-level function will be turned into a hardware accelerator by SmartHLS. The sub-functions called by top-level functions will also be compiled to hardware, for example the `vector_add_sw()` here. SmartHLS can compile multiple top-level functions, each with a "`top`" pragma, into hardware accelerators but for this example we will use a single accelerated function.

The pragmas on lines 73-75 describe the interface type of each argument to the accelerated function. On line 72, the control type is set to `axi_target` instead of the default `simple`. The control type interface must be `axi_target` if the user wishes SmartHLS to generate a Reference SoC automatically. Requiring an AXI target interface allows the generated Reference SoC to interact with the accelerator through an AXI4 interface without manually configuring the input and output wiring. If the control interface type is `simple`, the control interface will use individual wires for `clock`, `reset`, `ready`, etc., instead of an AXI target port and users will be responsible for connecting these to their system.


On lines 73-75, the interface type for arguments `a`, `b`, and `result` are all set to `axi_target`. For an `axi_target` interface, the hardware accelerator expects data to be sent to the data AXI target port and the accelerator will store the data in local memory blocks. The `num_elements` field specifies the length of the array that will be transferred for each argument. For more information on the required pragmas and tradeoffs, please see our [pragma manual](#).

In this example, we separated the core C++ algorithm into the `vector_add_sw` function. We can then call this function from multiple different SmartHLS top-level functions. We also call this function from our software test bench in `main` on line 158.


In the main function on line 141, we allocate the input arrays in contiguous physical memory using `hls_malloc` (covered in Section 6.4) and initialize the input arrays on lines 150-155. The main function calls the top-level function that will be turned into hardware on line 160, and compares the result against a software-computed golden output on line 168. Note that the main function returns 0 if the results match, which is required to run Software-Hardware Co-Simulation (Section 6.2.4). There are no restrictions on C++ code used in the main function that will not be turned into hardware, for example, file I/O can be used for your software testbench.

6.2.2 Compile Software to Hardware Reports



Click on the *Compile Software* icon  in the toolbar. This compiles the software with the GCC compiler. You will see the output from the compilation appearing in the bottom of the screen in the *Console* window of the IDE.



Now, execute the compiled software by clicking on the *Run Software* icon  in the toolbar. You should see the message *RESULT: PASS* appearing in the *Console* window, as shown in Figure 6-13.

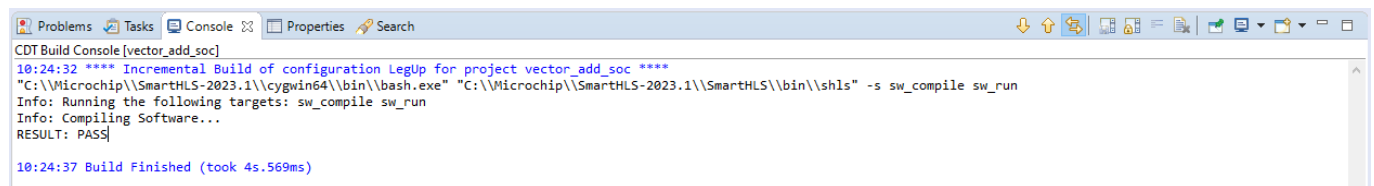



Figure 6-13 Compile Software Successful



Now we can compile the C++ software into hardware using SmartHLS by clicking on the toolbar icon  to *Compile Software to Hardware*. This command invokes SmartHLS to compile functions designated with the pragma “HLS function top” into hardware. If the top function calls descendant functions, all descendant functions are also compiled into hardware.

When the *Compile Software to Hardware* command is finished, SmartHLS will open the report file `hls_output/reports/summary.hls.vector_add_axi_target_memcpy.rpt`.

Notice that the top-level function name `vector_add_axi_target_memcpy` is specified in the report filename. There is one report generated for each top-level function.

The report shows the RTL interface of the generated Verilog module corresponding to the C++ top-level function as shown below in Figure 6-14. We can see that the generated IP's interface has input ports for clock and a single AXI4 Target port. Due to the large number of AXI4 ports in the RTL, SmartHLS uses a wildcard “`axi4target_*`” to simplify the table. The “Control AXI4 Target” indicates that start/finish control as done using the AXI target interface. Each of the function's three arguments also use the AXI target interface. The address map of the AXI target port is given later in the report.

```

+-----+
+-----+
| RTL Interface Generated by SmartHLS
|
+-----+-----+-----+-----+
+-----+-----+-----+-----+
| C++ Name | Interface Type          | Signal Name          |
Signal Bit-width | Signal Direction |
+-----+-----+-----+-----+
+-----+-----+-----+-----+
|          | Clock & Reset          | clk (positive edge)  | 1
| input    |          |          | reset (synchronous active high) | 1
| input    |          |          |
+-----+-----+-----+-----+
+-----+-----+-----+-----+
|          | Control via AXI4 Target | axi4target_*         |
|          |          |          |
+-----+-----+-----+-----+
+-----+-----+-----+-----+
| a        | AXI4 Target            | axi4target_*         |
|          |          |          |
+-----+-----+-----+-----+
+-----+-----+-----+-----+
| b        | AXI4 Target            | axi4target_*         |
|          |          |          |
+-----+-----+-----+-----+
+-----+-----+-----+-----+
| result   | AXI4 Target            | axi4target_*         |
|          |          |          |
+-----+-----+-----+-----+
+-----+-----+-----+-----+

```

Figure 6-14 RTL Interface Generated for vector_add_axi_target_memcpy

Report section “Scheduling Result” gives the number of cycles scheduled for each basic block of the function. Report section “Memory Usage” lists the memories that are used in the hardware. Any memory that is accessed by both the software testbench (parent functions of the top-level function) and hardware functions (the top-level functions and its descendants) becomes an *I/O memory*. These are any non-constant arguments for top-level function or global variables that are accessed by both the software testbench and hardware functions. I/O memories become memory interfaces of the top-level module for the generated hardware. For more information on interfaces, please refer to [Top-Level RTL Interface](#).

The “I/O Memories” table is shown in Figure 6-15 and has an entry for each top-level function argument, which each have a data width of 32-bits (int). There’s a known issue where the Size and Depth are incorrectly shown as 0, which will be fixed in a future SmartHLS version. The correct Depth should be 16 (**SIZE**), and “Size [Bits]” should be 512 bits (16x32).

I/O Memories				
Name	Accessing Function(s)	Type	Size [Bits]	Data Width
Depth	Read Latency			
a	vector_add_axi_target_memcpy	ROM	0	32
0	1			
b	vector_add_axi_target_memcpy	ROM	0	32
0	1			
result	vector_add_axi_target_memcpy	RAM	0	32
0	1			

Figure 6-15 An Example I/O Memory Usage Table

The report section on the AXI4 target interface address map, is shown below in Figure 6-16. This section first confirms that “Yes” (highlighted) this HLS accelerator is compatible with the reference SoC features (to be covered in Section 6.3). An accelerator is compatible if the control and all function arguments have an interface type of either `axi_target` or `axi_initiator`, so that the accelerator can be automatically integrated into the Reference SoC. If any of the interface types are the default of simple, the accelerator will be incompatible, and the user will not be able to generate a Reference SoC automatically. In addition, the target board must be the PolarFire SoC Icicle kit for the accelerator to be compatible with the Reference SoC features. If the accelerator is compatible, then the default base address for this accelerator when automatically integrated in the generated reference SoC is also shown: `0x70000000`.

The AXI4 Target Interface Address Map table informs the user of the address offsets, size, and direction for the Module Control (start and finish registers), and the three function arguments which are each 16 array elements (SIZE) x 4 bytes per element (int) = 64 bytes.

==== 4. AXI4 Target Interface Address Map =====

Compatibility of HLS accelerator with reference SoC features: Yes.
Default base address in reference SoC: `0x70000000`.

Accelerator Function: vector_add_axi_target_memcpy (Address Space Range: 0x100)			
Argument	Address Offset	Size [Bytes]	Direction

-----+				
	Module Control	0x008	4	inout
	a	0x040	64*	input
	b	0x080	64*	input
	result	0x0c0	64*	output
+-----+				+-----
-----+				

* On PolarFire SoC devices, it is recommended to use the PDMA engine for data transfer when the transfer size is bigger than 16KB, and use the memcpy driver functions when the transfer size is smaller than 16KB.

See memcpy and dma transfer driver functions in
[hls_output/accelerator_drivers/vector_add_soc_accelerator_driver.\[h|cpp\]](#)

Figure 6-16 Compatibility with Reference SoC Features and Address Space of Accelerator's Module Control and Arguments

6.2.3 Generated Verilog Output

You can find the generated Verilog code in

[hls_output/rtl/vector_add_soc_vector_add_axi_target_memcpy.v](#).

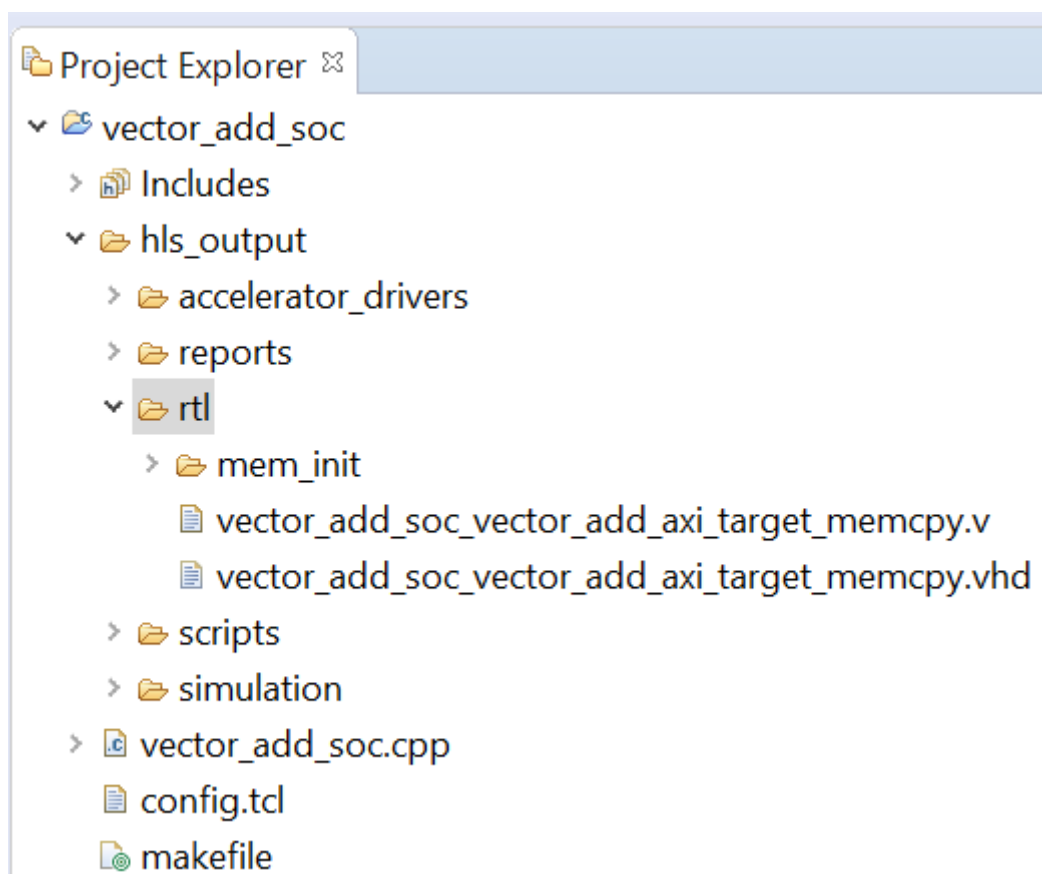


Figure 6-17 Finding the SmartHLS-Generated Verilog in the Project Explorer

If you open the Verilog file you will see the `clk`, `reset`, and AXI Target interface port (`axi4target`) as shown in Figure 6-18.


```
module vector_add_axi_target_memcpy_top # (  
    parameter ADDR_WIDTH = 8,  
    parameter AXI_DATA_WIDTH = 64,  
    parameter AXI_ID_WIDTH = 1  
) (  
    input  clk,  
    input  reset,  
  
    output                                axi4target_arready,  
    input                                axi4target_arvalid,  
    input  [ADDR_WIDTH - 1:0]           axi4target_araddr,  
    input  [AXI_ID_WIDTH - 1:0]         axi4target_arid,  
    input  [1:0]                         axi4target_arburst,  
    ...  
)
```

Figure 6-18 Snippet of `vector_add_soc_vector_add_axi_target_memcpy.v`

6.2.4 Running Co-Simulation

Now we can simulate the Verilog RTL hardware with ModelSim to find out the number of cycles needed to execute the circuit – the cycle latency.



Click on the *SW/HW Co-Simulation* icon  in the toolbar. *SW/HW co-simulation* will simulate the generated Verilog module, `vector_add_axi_target_memcpy_top`, in RTL using ModelSim, while running the rest of the program, `main`, in software. The co-simulation flow allows us to simulate and verify the SmartHLS-generated hardware without writing a custom RTL testbench.

In the *Console* window, you will see various messages printed by ModelSim related to loading simulation models for the hardware. The hardware may take a few minutes to simulate. We want to focus on the messages near the end of the simulation which will look like this:

```
# run 10000000000000000ns  
# Running SW/HW co-simulation...  
# Initializing AXI target input arguments at cycle =          0  
# AXI target initialization: Writing argument "a" at cycle =          0  
# AXI target initialization: Writing argument "b" at cycle =          0  
# Finished initializing of AXI target input arguments at cycle =  
0  
# Starting DUT using AXI target interface CSR at cycle =          0  
# --- vector_add_axi_target_memcpy_top Call          0: start at cycle =  
1  
# Polling AXI target interface CSR for finish signal at cycle =          1  
# ...  
# Received AXI target interface CSR finish signal at cycle =          56
```

```
# --- vector_add_axi_target_memcpy_top Call          0: finish at cycle =
57, total latency =          56
# Retrieving AXI target output arguments at cycle =          58
# AXI target retrieval: Reading argument "result" at cycle =          58
#          1 /          1 function calls completed.
# Finished retrieving AXI target output arguments at cycle =          58
# vector_add_axi_target_memcpy_top execution time (cycles):          56
# Number of calls:          1
# vector_add_axi_target_memcpy_top simulation time (cycles):          58
# ** Note: $finish      : cosim_tb.sv(794)
#   Time: 1265 ns  Iteration: 1  Instance: /cosim_tb
# End time: 10:31:55 on Jan 31,2024, Elapsed time: 0:00:01
# Errors: 0, Warnings: 0
Info: Verifying RTL simulation
Retrieving hardware outputs from RTL simulation for
vector_add_axi_target_memcpy function call 1.
RESULT: PASS
```

Top-Level Name	Number of calls	Simulation time (cycles)	Call Latency (min/max/avg)	Call II (min/max/avg)
vector_add_axi_target_memcpy_top	1	58	56 (single call)	N/A (single call)

```
Simulation time (cycles): 58
SW/HW co-simulation: PASS
```


Figure 6-19 Sample CoSim Results

The simulation printed “**SW/HW co-simulation: PASS**” which indicates that the RTL generated by SmartHLS matches the software model.

The co-simulation flow uses the return value from the main software function to determine whether the co-simulation has passed. If the main function returns 0, then the co-simulation will **PASS**; otherwise, a non-zero return value will **FAIL**. Please make sure that your main function always follows this convention and returns 0 if the top-level function tests are all successful.

6.2.5 Libero Synthesis and Hardware Report



Click the  icon on the toolbar to *Synthesize Hardware to FPGA*. SmartHLS will run Libero synthesis and place & route on the generated hardware accelerator.

Once the command completes, SmartHLS will open the **summary.results.rpt** report file. SmartHLS will summarize the resource usage and Fmax results reported by Libero® after place and route. You should get similar results as shown below in Figure 6-20. Your numbers may differ slightly, depending on the version of

SmartHLS and Libero® you are using. This tutorial used Libero® SoC v2022.1. The timing results and resource usage might also differ depending on the random seed used in the Libero tool flow.

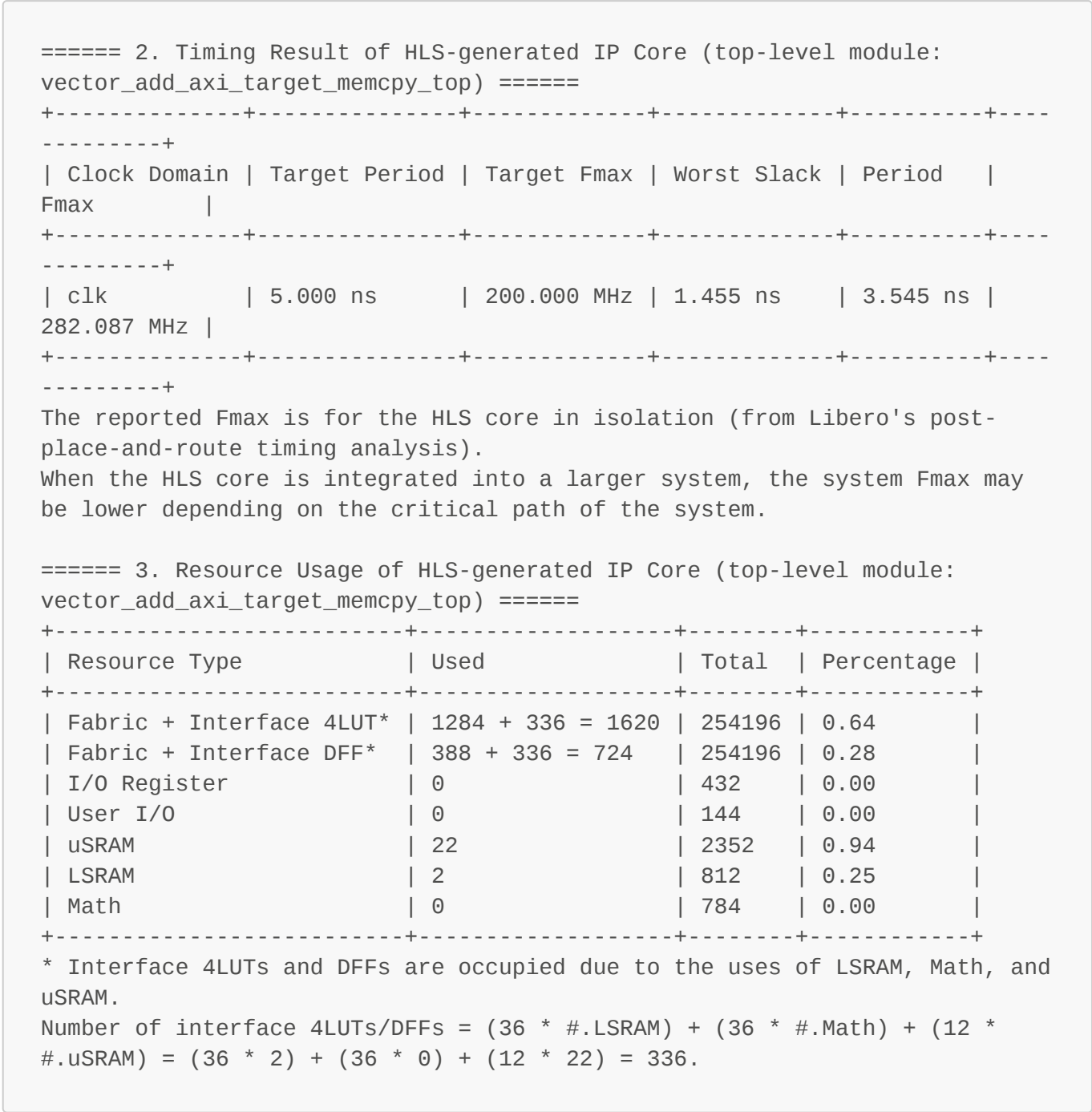


Figure 6-20 Timing and Resource Usage Results

6.2.6 SmartHLS Generated Software Driver APIs

SmartHLS generates C++ driver functions that can be used to control the generated hardware from an attached processor. This accelerator driver code can be found under `hls_output` in the `accelerator_drivers` output directory as shown in Figure 6-21.

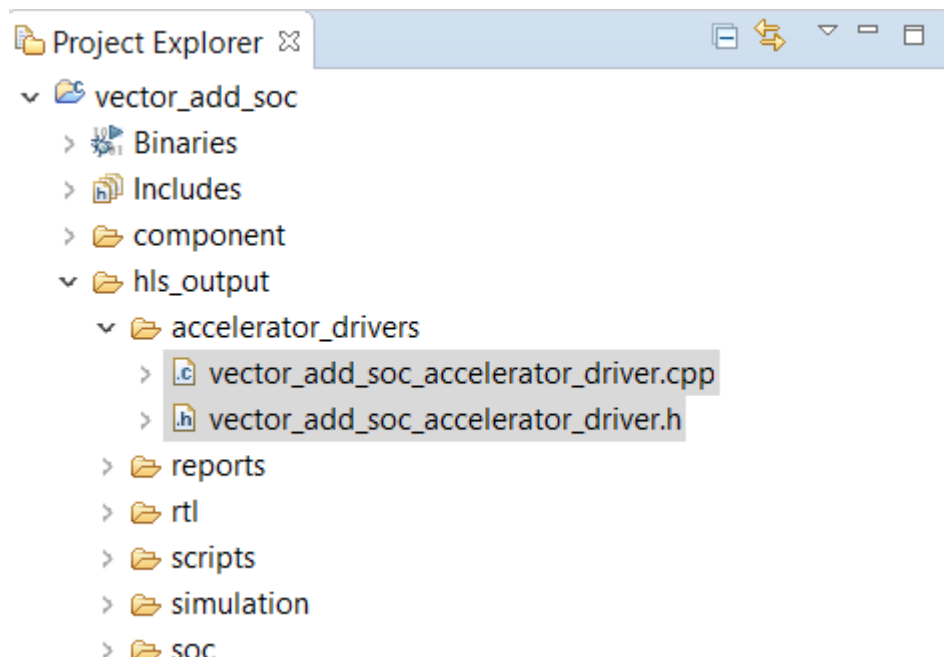


Figure 6-21 Accelerator Driver Files Location

The header file, `<PROJ_NAME>_accelerator_drivers.h`, in the directory lists the user-callable functions that can be used to control each HLS accelerator, while the `<PROJ_NAME>_accelerator_driver.cpp` file implements the driver functions. The driver functions are generated for arguments and module control if they are configured to use AXI4 target interface. Figure 6-22 summarizes the different categories of driver functions. Please visit [Driver Functions for AXI4 Target](#) section of our user guide for a more detailed explanation.

Example Function	Usage
<u>Module Control Driver Functions</u>	
int MyTopFunc_is_idle();	Returns 1 if the HLS module is idle (or has finished the last invocation).
void MyTopFunc_start();	Starts the HLS module. All other input arguments are expected to have been set when this function is called.
RETURN_TYPE MyTopFunc_join();	A blocking function that waits for the completion of the HLS module and returns the return value of the HLS module (if not void).
<u>Scalar Argument Driver Functions</u>	
void MyTopFunc_write_MyScalarArg(TYPE val);	Sets the scalar argument MyScalarArg to val.
TYPE MyTopFunc_read_MyScalarArg();	Retrieves the value of MyScalarArg, that was previously set by the write function above.

Example Function	Usage
<u>Pointer Argument Driver Functions - memcpy</u>	
void MyTopFunc_memcpy_write_MyPtrArg(void* MyPtrArg, uint64_t byte_size); void MyTopFunc_memcpy_read_MyPtrArg (void* MyPtrArg, uint64_t byte_size);	<p>The processor performs memory-mapped write/read operations (using the standard memcpy function) to copy data between the memory at MyPtrArg and the HLS accelerator. The total size to transfer is defined by the byte_size argument.</p>
<u>Pointer Argument Driver Functions - DMA</u>	
void MyTopFunc_dma_write_MyPtrArg(void* MyPtrArg, uint64_t byte_size); void MyTopFunc_dma_read_MyPtrArg (void* MyPtrArg, uint64_t byte_size);	<p>The processor offloads the data transfer to a DMA engine to move data between the memory at MyPtrArg and the HLS accelerator. The total size to transfer is defined by the byte_size argument.</p>
<u>AXI-Initiator Argument's Pointer Address Driver Function</u>	
void MyTopFunc_write_MyPtrArg_ptr_addr(void* offset);	<p>Sets pointer address for MyPtrArg</p>
<u>Top-Level Driver Functions</u>	
RETURN_TYPE MyTopFunc_hls_driver(...);	<p>This blocking function initializes all input argument data, starts the HLS module, waits for its completion, and retrieves output argument data and return value. It can be used as a direct replacement to the original top-level function. The arguments and return type are the same as the top-level function's.</p>

Example Function	Usage
void MyTopFunc_write_input_and_start(...);	This function initializes all input argument data and starts the HLS module. It starts the HLS module and resume to execute other parts of the software while the HLS module is running, then later call the <code>_join_and_read_output()</code> function below. The arguments are the input arguments of the top-level functions.
RETURN_TYPE MyTopFunc_join_and_read_output(...);	This is a blocking function that waits for the HLS module to finish the execution and retrieves output argument data and return value. This function expects <code>_start()</code> or <code>_write_input_and_start()</code> has been called prior to calling this function. The arguments are the output arguments of the top-level functions.

Figure 6-22 Summary of Driver Functions



Open `vector_add_soc_accelerator_driver.h` from the location shown in Figure 6-21. Line 10 and line 11 define the base address and the size of the address space the `vector_add` module occupies. These macro values have the same values as the values shown in the report (see Figure 6-23) and can be modified when incorporating these driver functions into your own hardware system if the accelerator base address changes.

`vector_add_soc_accelerator_driver.h`

```
10 #define VECTOR_ADD_AXI_TARGET_MEMCPY_BASE_ADDR 0x70000000
11 #define VECTOR_ADD_AXI_TARGET_MEMCPY_SPAN_ADDR 0x100
```

`summary.hls.vector_add_axi_target_memcpy.rpt`

```
===== 4. AXI4 Target Interface Address Map =====
```

```
Compatibility of HLS accelerator with reference SoC features: Yes.
Default base address in reference SoC: 0x70000000.
```

```
+-----+
+-----+
| Accelerator Function: vector_add_axi_target_memcpy (Address Space Range:
0x100)      |
+-----+-----+-----+-----+
+-----+
```


Argument	Address Offset	Size [Bytes]	Direction
Module Control	0x008	4	inout
a	0x040	64*	input
b	0x080	64*	input
result	0x0c0	64*	output

Figure 6-23 Module Base Address and Span in Header File



Open `vector_add_soc_accelerator_driver.cpp`. Line 137 to line 164 are the control module functions for the `vector_add_axi_target_memcpy` top function. `vector_add_axi_target_memcpy_start()` writes to the `vector_add_axi_target_memcpy` accelerator control register to start the accelerator. The accelerator will write a 0 to the same control register when the computation is done, and `vector_add_axi_target_memcpy_join()` runs a busy loop checking for 0 on that same control register. These functions only control the starting and waiting for the accelerator. They do NOT pass in the parameters.

```

137 vector_add_axi_target_memcpy_is_idle() {
138
139     volatile int *acc_start_addr =
140         (volatile int *)(vector_add_axi_target_memcpy_phys_map + 8); //
base+8
141
142     return *acc_start_addr == 0;
143 }
144
145 // This is a non-blocking function that starts the computation on the
accelerator.
146 // Any arguments, if any, should be written using the write functions
given.
147 // Use vector_add_axi_target_memcpy_join_and_read_output() to wait for
the accelerator to finish and return with the result.
148 void vector_add_axi_target_memcpy_start() {
149
150     // Run accelerator
151     volatile int *acc_start_addr =
152         (volatile int *)(vector_add_axi_target_memcpy_phys_map + 8); //
base+8
153
154

```

```

155  *acc_start_addr = 1;
156  }
157
158  // This is a blocking function that waits for the computation started
159  // by vector_add_axi_target_memcpy_start() to return.
160  // The return value is the result computed by the accelerator.
161  void vector_add_axi_target_memcpy_join() {
162
163  // Wait for accelerator to finish, acc_start_addr is set to 1 in the
164  // start function
165  while (!vector_add_axi_target_memcpy_is_idle()) {}
166  }

```

Figure 6-24 Control Module Function of vector_add_axi_target_memcpy Top Module



Go to line 167. `vector_add_axi_target_memcpy_hls_driver()` is the direct replacement function for the software version of `vector_add_axi_target_memcpy()`. SmartHLS will automatically replace the body of `vector_add_axi_target_memcpy()` by a single call to `vector_add_axi_target_memcpy_hls_driver()` when you click “Run software with accelerators” (covered in Section 6.3). `vector_add_axi_target_memcpy_hls_driver()` has the same parameters as `vector_add_axi_target_memcpy()`, but the parameters are casted into void pointers, as void pointers can be used to point to any data type.

```

167  // This is a blocking function that calls and waits for the
168  // accelerator to return.
169  // The return value is the result computed by the accelerator.
170  void vector_add_axi_target_memcpy_hls_driver(void* in, void* out) {
171
172  vector_add_axi_target_memcpy_write_input_and_start(in);
173  vector_add_axi_target_memcpy_join_and_read_output(out);
174  }
175
176  // This is a non-blocking function that starts the computation on the
177  // accelerator.
178  // Use vector_add_axi_target_memcpy_join() to wait for the accelerator
179  // to finish and return with the result.
180  void vector_add_axi_target_memcpy_write_input_and_start(void* in) {
181
182  // Run setup function
183  if (vector_add_axi_target_memcpy_setup() == 1) {
184  printf("Error: setup function failed for invert");
185  exit(EXIT_FAILURE);
186  }
187
188  vector_add_axi_target_memcpy_memcpy_write_a(a, 64);
189  vector_add_axi_target_memcpy_memcpy_write_b(b, 64);

```

```

188
189 vector_add_axi_target_memcpy_start();
190
191 }
192
193
194 // This is a blocking function that waits for the computation started
by vector_add_axi_target_memcpy_start() to return.
195 // The return value is the result computed by the accelerator.
196 void vector_add_axi_target_memcpy_join_and_read_output(void* out) {
197
198 vector_add_axi_target_memcpy_join();
199
200 vector_add_axi_target_memcpy_memcpy_read_result(result, 64);
201
202 }

```

Figure 6-25 Top-Level Function for vector_add_axi_target_memcpy Top Module

On line 169, `vector_add_axi_target_memcpy_hls_driver()` makes a call to the non-blocking `vector_add_axi_target_memcpy_write_input_and_start()` function that writes the input arguments and starts the accelerator. Then, `vector_add_axi_target_memcpy_hls_driver()` calls the blocking `vector_add_axi_target_memcpy_join_and_read_output()` function on line 172 to wait for and read back the accelerator's output. Users can use `vector_add_axi_target_memcpy_write_input_and_start()` to start the calculation, then execute other computations on the MSS in parallel with the hardware accelerator execution in the FPGA fabric. Later, users can use `vector_add_axi_target_memcpy_join_and_read_output()` to retrieve the results from the hardware accelerator. This is like using threads to do parallel computations.

6.3 SmartHLS SoC Flow

SmartHLS can generate a reference SoC design, with user-specified partitioning of software running on MSS and hardware accelerators running on the FPGA fabric. We refer to this as the *SmartHLS SoC flow*. No code changes are required to go from IP flow to SoC flow using the `vector_add_soc` example, since the top-level function is compatible with the Reference SoC features (see Figure 6-16). The `main` software function will run on the MSS in Linux and `main` will call the driver software APIs to run the hardware accelerator.

The SmartHLS SoC flow steps are broken down below (Figure 6-26). These steps are available to users at the click of a button in the IDE, but we first want to provide further details to give users a better understanding of SmartHLS.

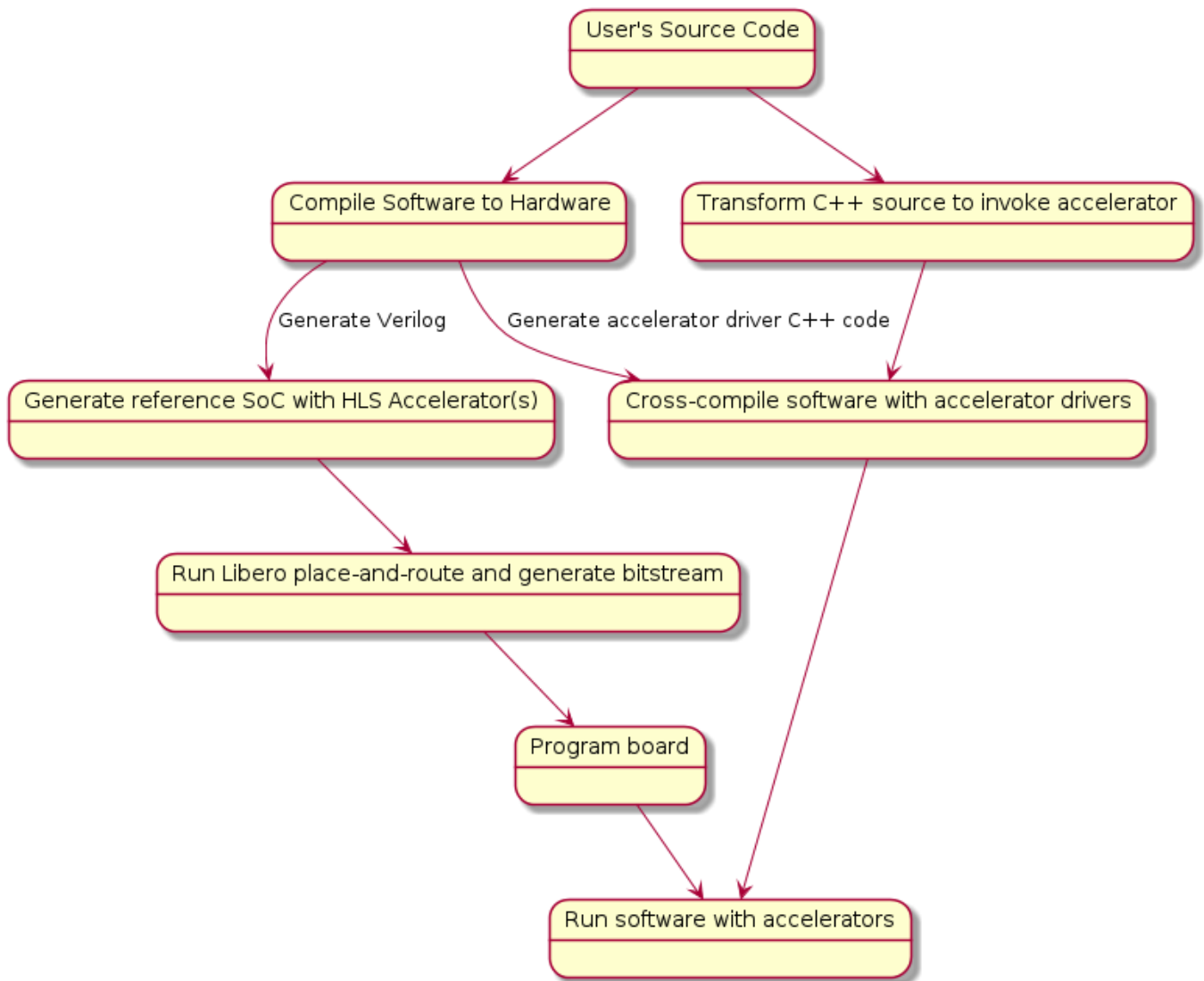


Figure 6-26 SoC Project Generation Flow

Each box in Figure 6-26 under *User's Source Code* corresponds with a compilation step. Whenever the user code is changed, *Compile Software to Hardware* and *Transform C++ source to invoke accelerator* become out of date and require a remake. Following the right-hand path of Figure 6-26, SmartHLS performs a C++ to C++ source transformation to replace the body of the functions marked by the users to invoke the FPGA accelerator instead of running the code in software. After *Compile Software to Hardware* generated the C++ driver code that runs the accelerators from software, SmartHLS cross-compiles the transformed C++ source and the driver code generated by *Compile Software to Hardware* to run on the RISC-V processor. This completes the software portion needed for running the software on the board.

Following the left-hand path of Figure 6-26 and after *Compile Software to Hardware* has completed, *Generate reference SoC with HLS Accelerator(s)* invokes SmartDesign to integrate the accelerators using the TCL scripts generated by the previous step. SmartHLS then performs the place and route and generates the bitstream. Once the bitstream is available, SmartHLS programs the board and completes all the hardware prerequisites for running the system on board. Now that the board has been programmed and the software is generated, we can copy the RISC-V software binaries to the board using SSH and run the software with accelerator on the board.

You can run the SmartHLS SoC flow from the top menu of the SmartHLS IDE, under the *SmartHLS -> RISC-V SoC Features (available for PolarFire SoC only)* as shown in Figure 6-27. You can also find the same menu

options by clicking the “SoC” button in the toolbar (previously shown in Figure 6-8).

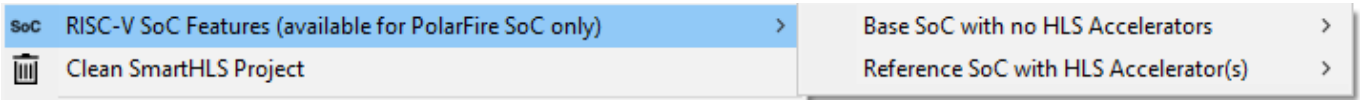


Figure 6-27: SmartHLS RISC-V SoC Features (available for PolarFire SoC only)

There are two options shown in Figure 6-27 under this menu:

Base SoC with no HLS Accelerators means that we are programming the prebuilt reference bitstream that ships with SmartHLS (Base SoC) and we are only running software on the MSS. The prebuilt FPEXpress job file for the Base SoC can be found at

```
<SMARTHLS_INSTALLATION_DIR>\SmartHLS\reference_designs\Icicle_SoC\MPFS_ICICLE_BASE_DESIGN_2023_02.job
```

Reference SoC with HLS Accelerator(s) means that SmartHLS performs hardware/software partitioning between software running on the MSS and the FPGA accelerators. SmartHLS will generate a new bitstream (Reference SoC) with the accelerator connected to the MSS over AXI.

There are 3 options under *Base SoC with no HLS Accelerators* as shown in Figure 6-28. Running later steps (further down) can depend on running previous steps. For example, clicking *Run software without accelerators* will prompt a dialog asking to run *Cross-compile software for RISC-V* and *Program board with prebuilt bitstream*.

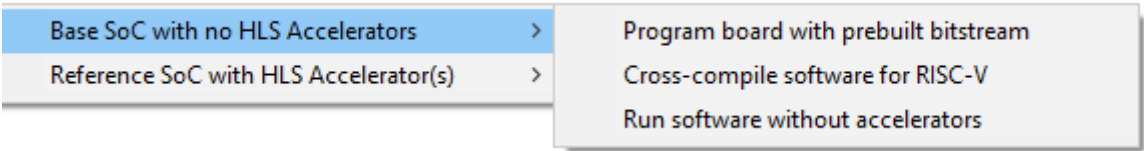


Figure 6-28: Base SoC with no HLS Accelerators menu options

There are 6 options under *Reference SoC with HLS Accelerator(s)* as shown in Figure 6-29. There are 3 more options than Figure 6-28 because the bitstream is not prebuilt like the Base SoC. We have additional steps to generate the Libero design, run RTL synthesis, and run place-and-route.

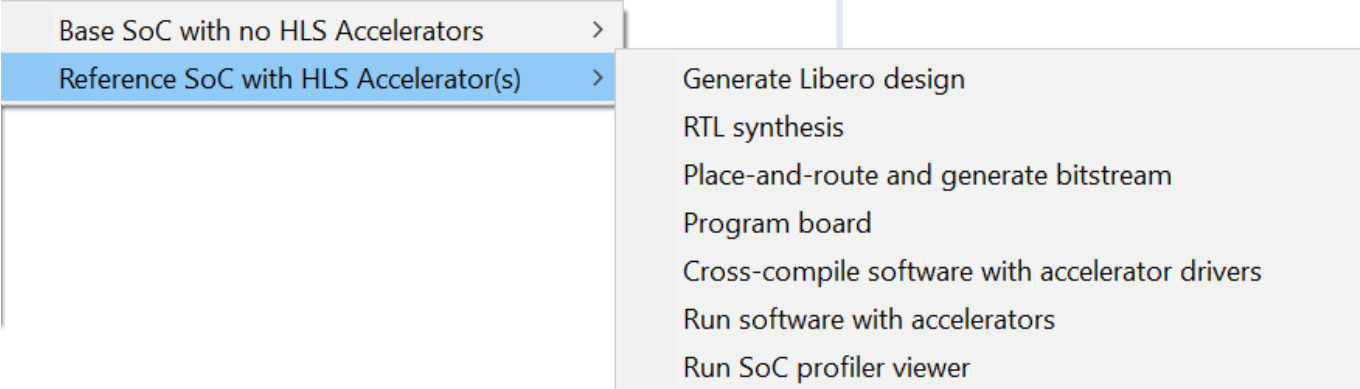


Figure 6-29: Reference SoC with HLS Accelerator(s) menu options



In the SmartHLS IDE, select *SmartHLS -> RISC-V SoC Features (available for PolarFire SoC only) -> Reference SoC with HLS Accelerator(s) -> Generate Libero design*, as seen in Figure 6-30. This command will generate a “Reference SoC” Libero project containing the Icicle kit MSS, the generated vector-add accelerator, and setup a Libero project with the vector-add accelerator connected to the MSS. The Libero project is now ready for synthesis, place-and-route and programming onto the board as you would for a regular Libero project.

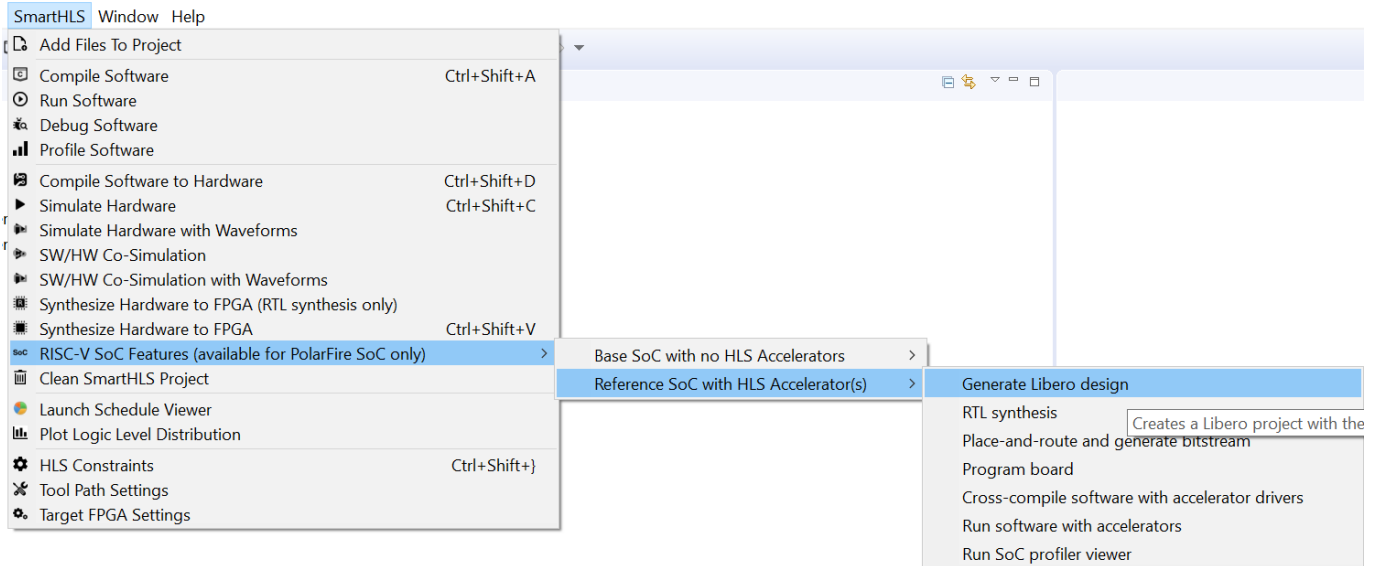


Figure 6-30 “Reference SoC with HLS Accelerator(s) -> Generate Libero Design” Menu

Generate Libero design (Figure 6-30) will create a Libero project that you can open with Libero in `hls_output/soc/Icicle_SoC.prjx`. The generated SmartDesign hardware system contains the MSS connected via AXI4 to the `vector_add_axi_target_memcpy` accelerator as shown in Figure 6-31. Note that the generated accelerator is the same as the one generated using IP flow in the previous Section 6.2.



The Reference SoC Libero design is generated in the project directory, under `hls_output/soc/Icicle_SoC.prjx` Libero project file. Open this project in Libero to view the reference SoC design. In the *Design Hierarchy* tab, double-click `FIC_0_PERIPHERALS` to open the SmartDesign project, as seen in Figure 6-31.

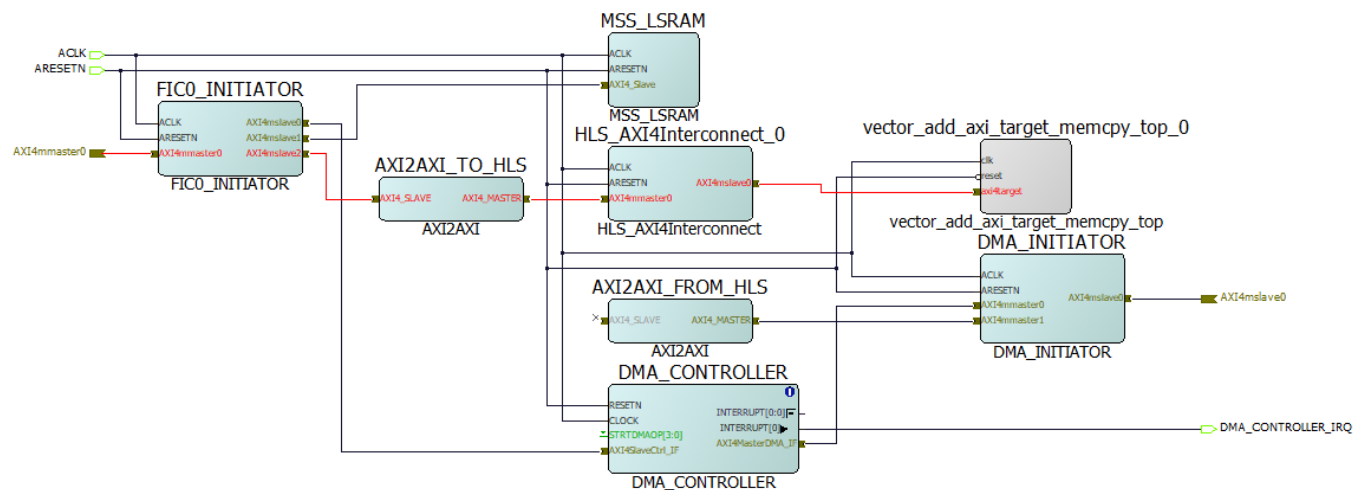
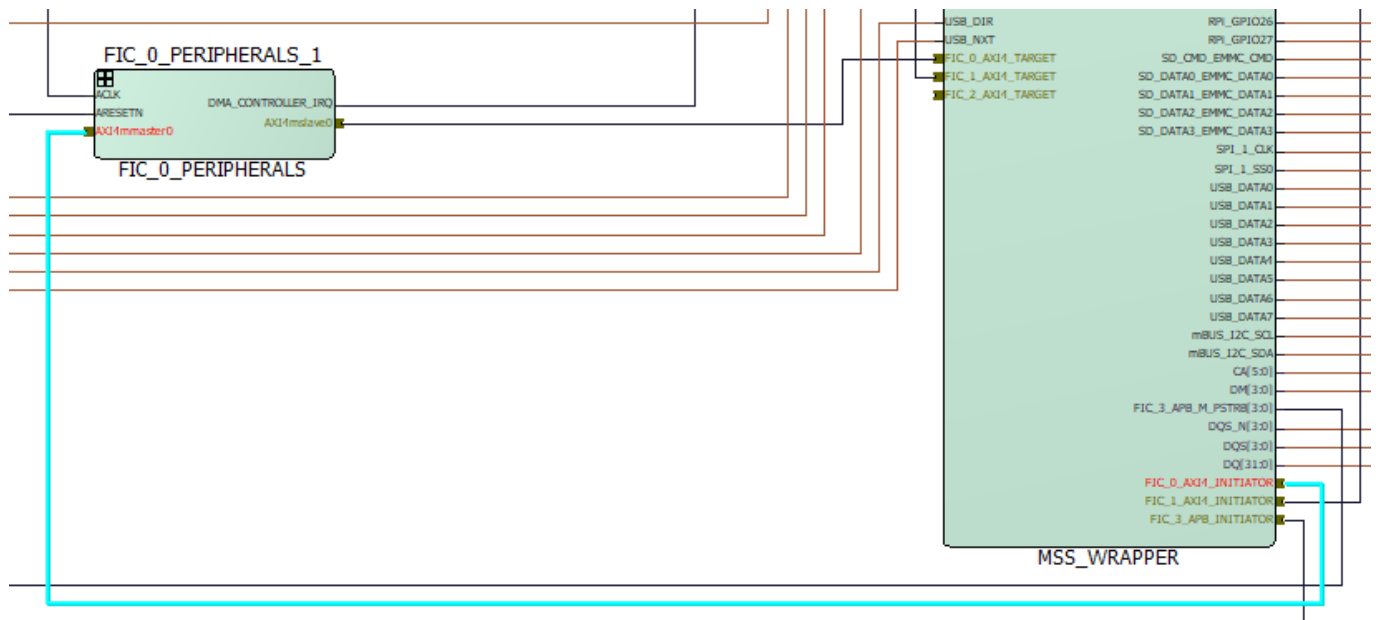




Figure 6-31 SmartDesign for Vector-Add SmartHLS Generated Reference SoC

In the SmartDesign project, the accelerator IP `vector_add_axi_target_memcpy_top` is instantiated on the right and connected through an AXI interconnect IP (center) to the MSS . This is the path through which the software main function running on the processor communicates with the accelerator IPs as well as the path for data transfers between the DDR and the accelerator. Any additional accelerators would be connected to the same AXI interconnect. For more information on the architecture of the Reference SoC, please see our [user guide](#).

We can simplify the SmartDesign visualization by clicking  Hide Nets,  Compress Layout, then dragging the HLS_AXI4Interconnect_0, and vector add modules as shown in Figure 6-32. We have highlighted the AXI4 interface connections between the MSS and the vector add accelerator.

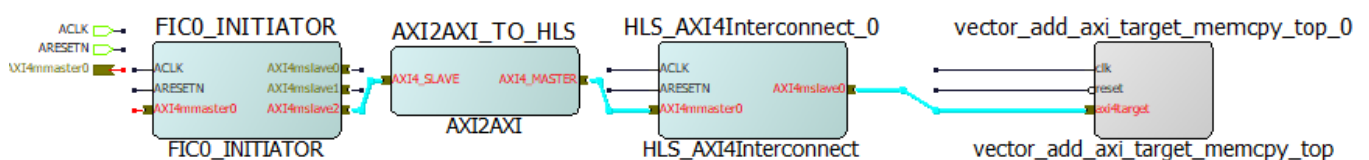


Figure 6-32 Simplified SmartDesign for Generated Reference SoC with Vector Add Accelerator



Now close the Libero project and go back to the SmartHLS IDE.

6.3.1 SoC Data Transfer Methods

In this section, we will cover the three different SoC data transfer methods supported between the MSS and the hardware accelerator: CPU Copy, DMA Copy, and Accelerator Direct Access. The transfer method is specified for each function argument using the interface type pragmas. AXI interfaces are used to send and receive data from the accelerator to/from the MSS. Each function argument can be configured to a different interface type depending on the application, for example larger arguments could use DMA transfers.

When sharing data between the MSS and the FPGA fabric, we need to transfer data from the MSS main memory in off-chip DDR memory. For each pointer argument of an accelerator, data can be copied from DDR memory to accelerator's on-chip memory buffer, or the data can be accessed directly in DDR by the accelerator. Any access to DDR, whether data is copied or accessed directly, goes through the MSS data cache to maintain cache coherency. See the [SoC Data Transfer Methods](#) user guide section for further reference.

6.3.1.1 CPU Copy: AXI Target

In CPU Copy mode, the MSS handles the transfer of data between the DDR and the accelerator. The MSS requests the data from DDR and passes the data through the AXI4 interconnect to the accelerator. The accelerator has an on-chip buffer storing the received data. This is the recommended mode when transferring data under 16 kB in size. Figure 6-33 shows how data travels between the accelerator and the DDR.

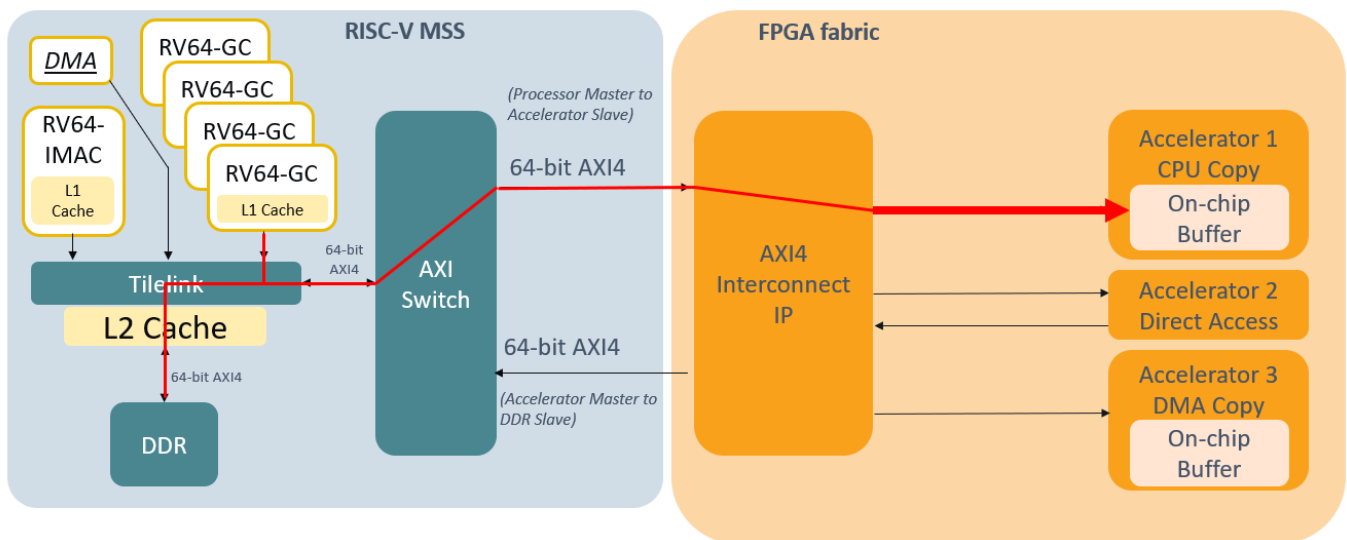


Figure 6-33 CPU Copy Data Path

CPU Copy mode occurs when a function argument interface type is AXI target, for example in the ``vector_add_axi_target_memcpy` top-level function (see code previously in Figure 6-12):

```
#pragma HLS interface argument(a) type(axi_target)
```


6.3.1.2 DMA Copy: AXI Target with DMA

In DMA Copy mode, the MSS will use the hardened DMA engine (PDMA) to transfer data between the DDR and the accelerator (Figure 6-34). This is the recommended mode when transferring data over 16 kB in size.

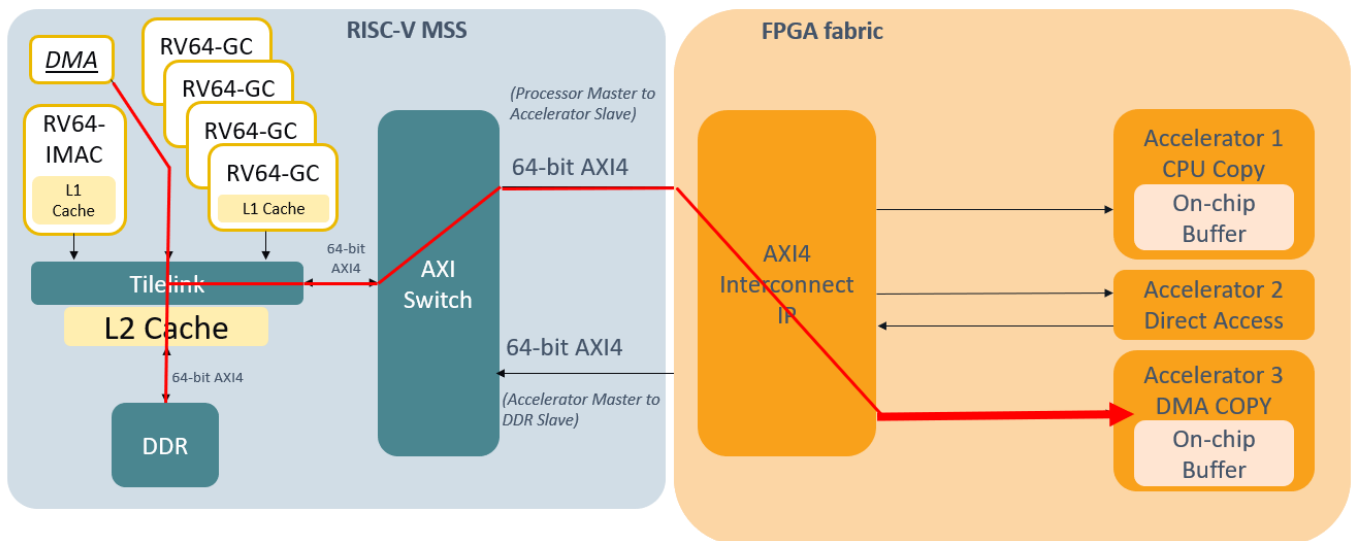


Figure 6-34 DMA Copy Data Path

DMA Copy mode occurs when a function argument interface type is AXI Target with the DMA sub-option specified. For example, in the function `vector_add_axi_target_dma` as highlighted on line 85 shown in Figure 6-35. Note that the generated RTL for this accelerator is no different from `vector_add_axi_target_memcpy`. The only difference is how the MSS transfers data to the accelerator. Since the size of the array is only 16 in this example, the data transfer time doesn't benefit from using the DMA. We wrote this function for illustrative purposes.

```
81 void vector_add_axi_target_dma(int *a, int *b, int *result) {
82 #pragma HLS function top
83 #pragma HLS interface control type(axi_target)
84 #pragma HLS interface argument(a) type(axi_target) dma(true)
  num_elements(SIZE)
85 #pragma HLS interface argument(b) type(axi_target) dma(true)
  num_elements(SIZE)
86 #pragma HLS interface argument(result) type(axi_target) dma(true)
  \
87   num_elements(SIZE)
88   vector_add_sw(a, b, result);
89 }
```

Figure 6-35 AXI Target DMA Pragma

6.3.1.3 Accelerator Direct Access: AXI Initiator

Accelerator direct access mode allows the hardware accelerator to directly read and write to DDR. Unlike the AXI Target interface, AXI Initiator interface does not receive the data directly from the processor or the DMA,

instead the accelerator will have two AXI interfaces:

1. AXI Target: The accelerator receives a pointer to where the argument is stored (e.g. address in DDR memory) through the `ptr_addr_interface` AXI Target interface from the MSS (Figure 6-36), and then
2. The accelerator using the AXI initiator interface accesses DDR memory through the MSS cache (Figure 6-37).

The memory accesses are cache coherent between the accelerator and MSS since they share the L2 cache, but L1 cache could be invalidated. Since the data is accessed directly from DDR without copying, there are no additional on-chip memory needed for the accelerator.

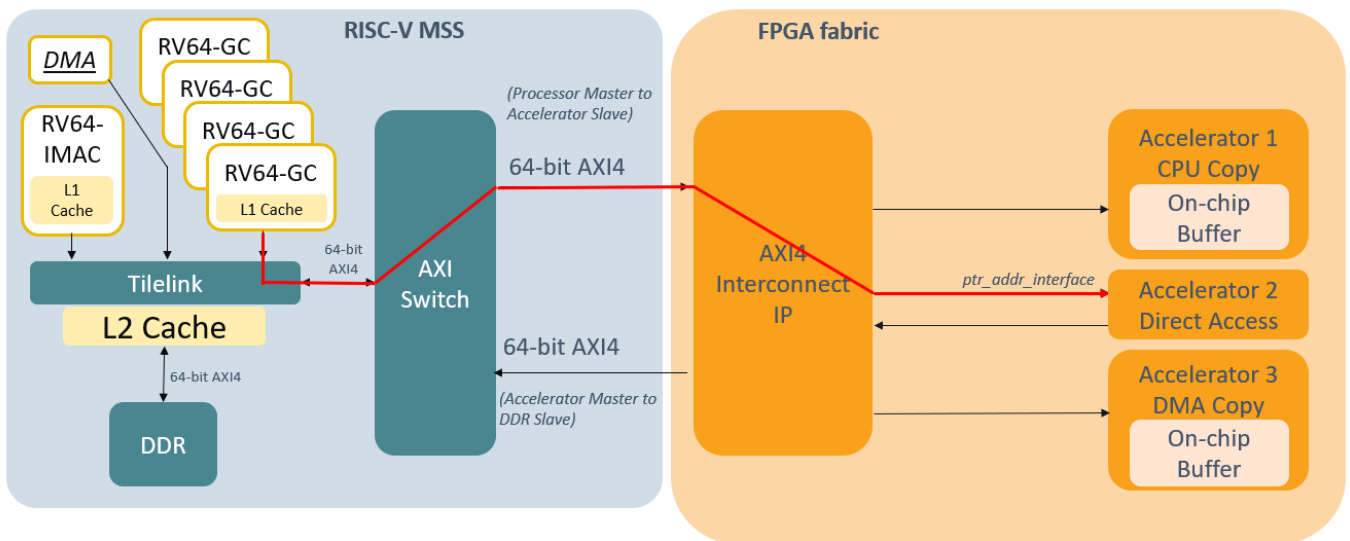


Figure 6-36 MSS Sends the Pointer to Accelerator AXI Target Interface in Direct Access Mode

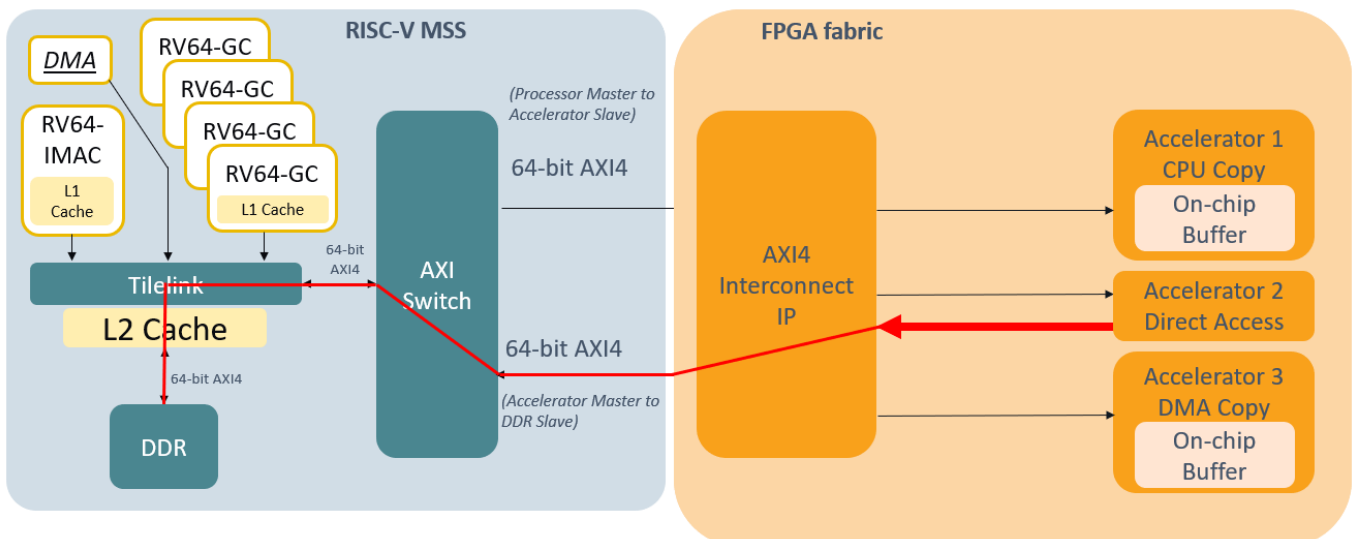


Figure 6-37 Accelerator AXI Initiator Interface Requests Data Directly from DDR in Direct Access Mode

Look at line 119 of `vector_add_soc.cpp` (Figure 6-38). We defined another top-level function `vector_add_axi_initiator` that uses AXI initiator interface for its arguments. Line 123 defined the default type for all arguments and control. If the default argument type were unspecified, the default argument type for all arguments and control is simple. If the interface for a, b and result were not defined, they would be

defaulted to `axi_target`. So, when planning to use SmartHLS SoC reference design, be sure to set the default or specify each interface to either `axi_target` or `axi_initiator`.

```
119 void vector_add_axi_initiator(int *a, int *b, int *result) {
120 // Note that both the control and ptr_addr_interface are redundant
since the
121 // default is already axi_target
122 #pragma HLS function top
123 #pragma HLS interface default type(axi_target)
124 #pragma HLS interface control type(axi_target)
125 #pragma HLS interface argument(a) type(axi_initiator)          \
126     ptr_addr_interface(axi_target) num_elements(SIZE)
127 #pragma HLS interface argument(b) type(axi_initiator)
num_elements(SIZE)
128 #pragma HLS interface argument(result) type(axi_initiator)
num_elements(SIZE)
129     vector_add_sw(a, b, result);
130 }
```

Figure 6-38 AXI Initiator Example

Line 124 specifies the default control type for `axi_target`, which is redundant since the default interface type was defined as `axi_target` on line 123. The interface type for arguments `a`, `b` and `result` is set to `axi_initiator` on lines 125-129. The `ptr_addr_interface` sub-option on line 126 specifies the type of interface that is used to receive the pointer address to access the argument. In this case, the pointer address of argument "a" will be received with the AXI target interface as shown in Figure 6-36, and this pointer address will be used to access the data for argument "a" with the AXI initiator interface as shown in Figure 6-37. If the `ptr_addr_interface` is not specified, for example for argument b, SmartHLS will use the default interface type defined on line 123 (`axi_target`). See the [AXI4 Initiator Interface](#) section of the user guide.

If users specify the `ptr_addr_interface` or any other interface type as `simple`, then the accelerator is not compatible with Reference SoC features and they would have to manually connect the input for the interface using a TCL script or in Libero.



We will now change the top-level accelerator argument interface from AXI target to AXI initiator. Go to line 21 of `vector_add_soc.cpp`, change the definition of `INTERFACE` from `AXI_TARGET_MEMCPY` (highlighted in Figure 6-39) to `AXI_INITIATOR`.

```
14 // Choose which interface to compile
15 // Possible Values: AXI_TARGET_MEMCPY, AXI_TARGET_DMA, AXI_INITIATOR
16 #define AXI_TARGET_MEMCPY 0
17 #define AXI_TARGET_DMA 1
18 #define AXI_INITIATOR 2
19
20 #ifndef INTERFACE
```


```

21 #define INTERFACE AXI_TARGET_MEMCPY
22 #endif

```

Figure 6-39 Pragma for Choosing Example's Interface Type



Go to SoC pulldown menu , select *Reference SoC with HLS Accelerator(s)* -> *Generate Libero Design*. You should see a pop-up window (Figure 6-40) asking for confirmation to run *Compile Software to Hardware*. Click *Yes* to continue. If users have made changes in the future that does not affect the generated hardware in the source code, such as adding a comment, users can choose *Skip above step(s)* to save compilation time.

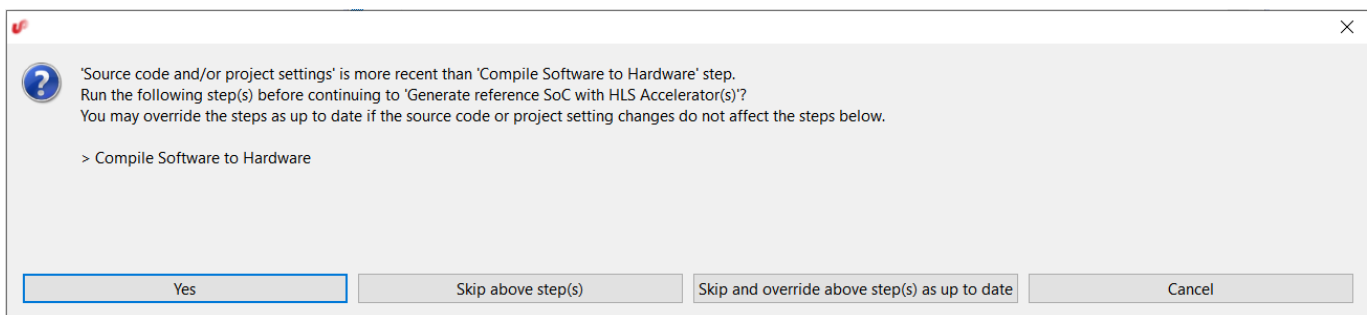


Figure 6-40 Compilation Confirmation Pop-up Window



`hls_output/reports/summary.hls.vector_add_axi_initiator.rpt` will be generated once the compilation has finished. You can see that the RTL interface summary (Figure 6-41) is drastically different from `AXI_TARGETs` (Figure 6-14). The pointer addresses of `a`, `b`, and `result` are being passed to the accelerator through the `axi_target` interface. The accelerator will then read the data from the DDR directly using the given address and write the result into the DDR memory directly.

===== 1. RTL Interface =====

```

+-----+
+-----+
| RTL Interface Generated by SmartHLS
|
+-----+-----+-----+-----+
+-----+-----+-----+-----+
| C++ Name | Interface Type | Signal Name
| Signal Bit-width | Signal Direction |
+-----+-----+-----+-----+
+-----+-----+-----+-----+
|          | Clock & Reset |          | clk (positive edge)
| 1          | input          |          |
|          |          |          | reset (synchronous
active high) | 1          | input          |
+-----+-----+-----+-----+

```

```

+-----+-----+-----+
|          | Control via AXI4 Target          | axi4target_*
|          |          |          |
+-----+-----+-----+
+-----+-----+-----+
| a        | AXI4 Initiator          | axi4initiator_*
|          |          |          |
|          | with ptr_addr_interface(axi_target) | axi4target_*
|          |          |          |
+-----+-----+-----+
+-----+-----+-----+
| b        | AXI4 Initiator          | axi4initiator_*
|          |          |          |
|          | with ptr_addr_interface(axi_target) | axi4target_*
|          |          |          |
+-----+-----+-----+
+-----+-----+-----+
| result   | AXI4 Initiator          | axi4initiator_*
|          |          |          |
|          | with ptr_addr_interface(axi_target) | axi4target_*
|          |          |          |
+-----+-----+-----+
+-----+-----+-----+

```

Figure 6-41 An Example RTL Interface Generated Table for AXI Initiator

6.4 SmartHLS Memory Allocation Library

In `vector_add_soc.cpp`, on lines of 144-147 of the main function, we used the `hls_malloc` function to allocate physically contiguous memory regions for the data passed to/from the hardware accelerator as shown in Figure 6-42.

```

143 // Allocating memory from DDR memory
144 int *a = (int *)hls_malloc(SIZE * sizeof(int));
145 int *b = (int *)hls_malloc(SIZE * sizeof(int));
146 int *result_hw = (int *)hls_malloc(SIZE * sizeof(int));
147 int *result_sw = (int *)hls_malloc(SIZE * sizeof(int));

```

Figure 6-42 Allocating Memory in the DDR for Vectors

DMA Copy mode and Accelerator Direct Access require the memory to be allocated using the `hls_malloc` function from the [SmartHLS Memory Allocation Library](#) to keep data in physically contiguous memory for the DMA engine. Using `hls_malloc` prevents splitting data across different virtual memory pages in physical memory. The accelerators and DMA engine do not perform translation from virtual to physical memory addresses.

Unlike DMA Copy mode and Accelerator Direct Access, CPU Copy mode does not require the use of `hls_malloc` for allocating the argument data. In CPU Copy mode, the MSS controls all data that is

read/written to accelerators and DDR and the MSS will automatically handle the virtual memory address translations.

7. Running Vector-Add Reference SoC Generation on the Board

This section uses a PolarFire SoC Icicle Kit. The Icicle Kit is a low-cost development platform featuring a hardened five-core RISC-V processor, capable of running Linux, a PolarFire SoC FPGA, and many peripherals. For more details on the Icicle Kit, and information on how to obtain one, please see the [product page](#).

In this part of the training, we will run the vector add application on the Icicle Kit board. We will generate the reference SoC, program the bitstream to the PolarFire SoC FPGA, and run the accelerator driver software on the MSS.

Users without an Icicle Kit can still follow along to learn about how a SoC reference project is generated.



To prepare your Icicle kit for use with SmartHLS, follow the [Icicle Setup Instructions](#) and note down the IP of the board.



Create a new file named `Makefile.user` by right clicking on `vector_add_soc` then *New -> File* (Figure 7-1).

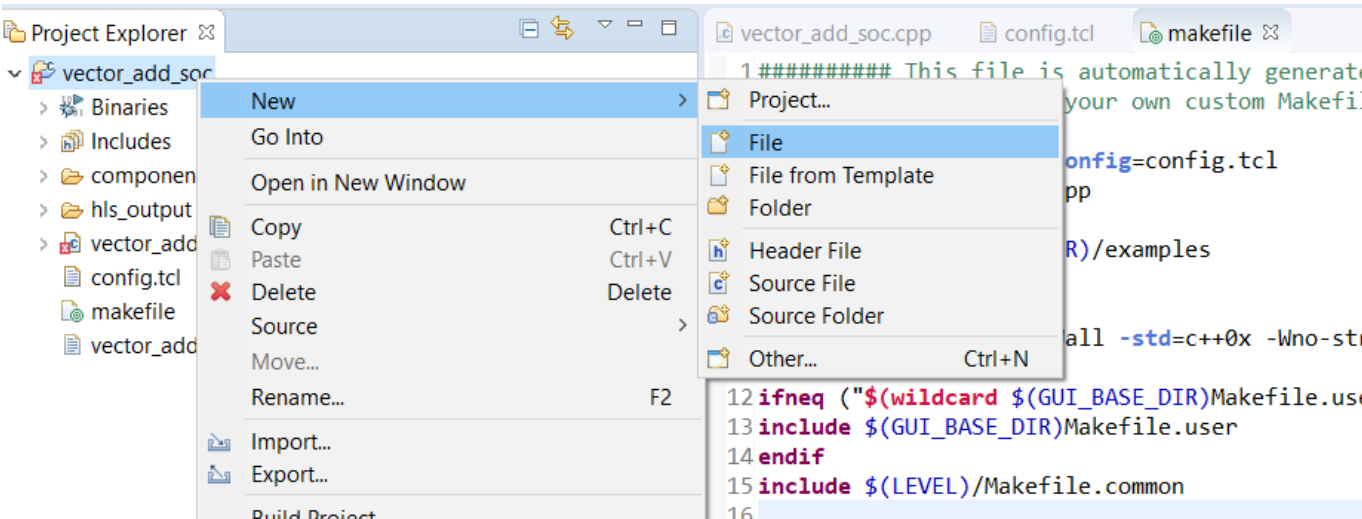
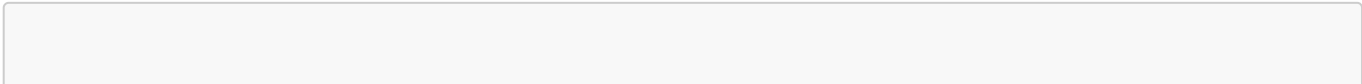


Figure 7-1 Creating a New File



Insert a line for your board's network IP address, like what is shown in Figure 7-2.



```
BOARD_IP = <Your Board IP>
```

Figure 7-2 Makefile.user's Content

Since the `makefile` is freshly regenerated every time the SmartHLS IDE compiles, users must define makefile changes in `Makefile.user` for the changes to take effect. There are several predefined user flags that SmartHLS reads in `Makefile.user` where users can define and modify options such as compiler and linker flags. For example, users can modify `USER_CXX_FLAG` to append additional C++ compilation flags for their project. Visit the [Makefile Variable](#) section of our user guide for a full list of predefined user flags and their uses.



From the SmartHLS menu, select *SmartHLS -> RISC-V SoC Features (available for PolarFire SoC only) -> Base SoC with no HLS Accelerators -> Program Board with Prebuilt Bitstream* (see Figure 7-3). SmartHLS will program the prebuilt Base SoC bitstream to the attached Icicle board. After the Icicle board has been successfully programmed, you will see the message in Figure 7-4.

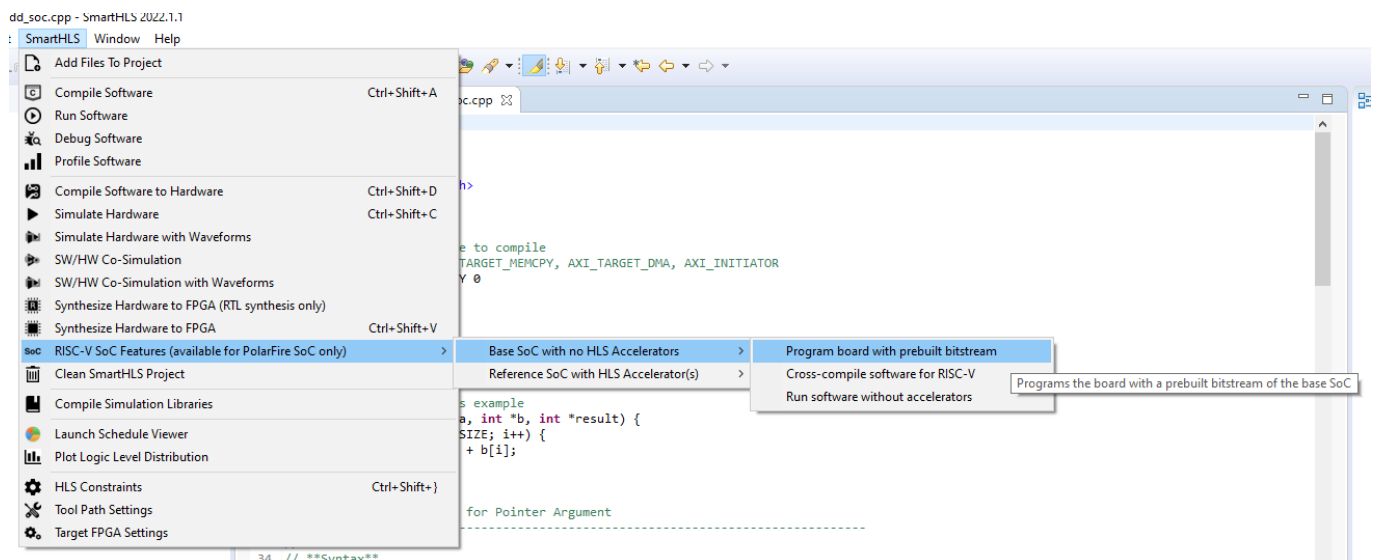


Figure 7-3 Program Board with Prebuilt Bitstream Option Menu

```
programmer '1D1A9C5C' : device 'MPFS250T_ES' : Executing action PROGRAM PASSED.
programmer '1D1A9C5C' : Chain programming PASSED.
Chain Programming Finished: Sun May 22 00:50:06 2022 (Elapsed time 00:01:01)

o - o - o - o - o - o - o

The 'run_selected_actions' command succeeded.
The Execute Script command succeeded.
Exported log file hls_output/FPEXpress_program/job.log.

00:50:07 Build Finished (took 1m:54s.77ms)
```

Figure 7-4 Program Board Successful



Users can run their program entirely in software on the MSS without calling the accelerators. This is useful for verifying the correctness of the software and the MSS system, as well as profiling the performance of the system. To run only the software on the board, go to *SmartHLS -> RISC-V SoC Features (available for PolarFire SoC only) -> Base SoC with no HLS Accelerators -> Run software without accelerators* (as shown in Figure 7-5). SmartHLS will cross-compile the source code for RISC-V, then SmartHLS will copy the RISC-V binary to the board over SSH, using the **BOARD_IP** specified in **Makefile.user**. The correct result should see **RESULT: PASS** as seen on Figure 7-6.

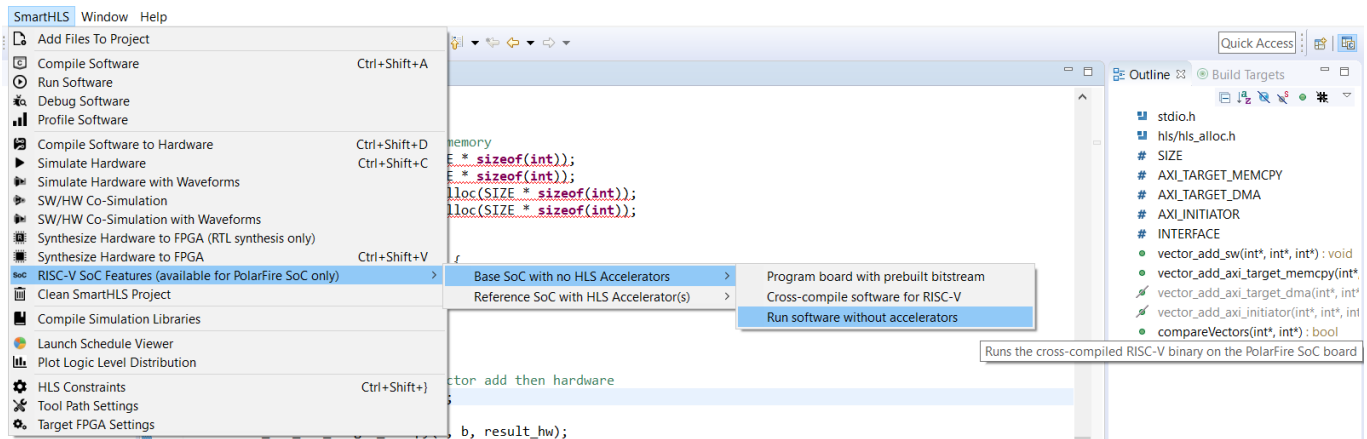


Figure 7-5 Run Software without Accelerators Option Menu

```
15:19:16 **** Incremental Build of configuration LegUp for project vector_add_soc ****
"C:\Microsemi\SmartHLS-2022.1.1\cygwin64\bin\bash.exe" "C:\Microsemi\SmartHLS-2022.1.1\SmartHLS\bin\shls" -s soc_base_proj_run
Running the following targets: soc_base_proj_run
/cygdrive/c/Microsemi/SmartHLS-2022.1.1/SmartHLS/examples/Makefile.soc:353: Warning: PROGRAMMER_ID is not set. All connected programmers will be programmed.
Info: Checking for SmartHLS_SoC feature license.
Info: SmartHLS_SoC feature license was successfully checked out.
/cygdrive/c/Microsemi/SmartHLS-2022.1.1/SmartHLS/examples/Makefile.soc:353: Warning: PROGRAMMER_ID is not set. All connected programmers will be programmed.
Waiting on board ready...
Board ready!
Copying hls_output/vector_add_soc.no_accel.elf to root@192.168.8.48:
Application starting (over ssh root@192.168.8.48)
Running: ./vector_add_soc.no_accel.elf > bin_cl_out.txt; cat bin_cl_out.txt
make[1]: Entering directory '/cygdrive/c/Work/Training4/vector_add_soc'
Application output:
RESULT: PASS
make[1]: Leaving directory '/cygdrive/c/Work/Training4/vector_add_soc'
Application finished!
Copying bin_cl_out.txt from root@192.168.8.48: to hls_output/files/

15:19:39 Build Finished (took 22s.412ms)
```

Figure 7-6 Expected Output from Running Software on Board

Now that you have verified that your software program can run correctly on your Icicle Kit, you can run the software with accelerators that SmartHLS generates. This software executable is the same as **vector_add_soc.cpp**, but with the calls to **vector_add_axi_target_memcpy** automatically replaced with driver code to control the accelerator IP on the FPGA fabric.



In the same menu as before, click *SmartHLS -> RISC-V SoC Features (available for PolarFire SoC only) -> Reference SoC with HLS Accelerator(s) -> Run software with accelerators* (Figure 7-7). SmartHLS will automatically run all the steps prior to Run software with accelerators, i.e. *Generate Libero design, RTL synthesis, Place-and-route and generate bitstream, Program board, Cross-compile software with accelerator drivers* (Figure 7-7).

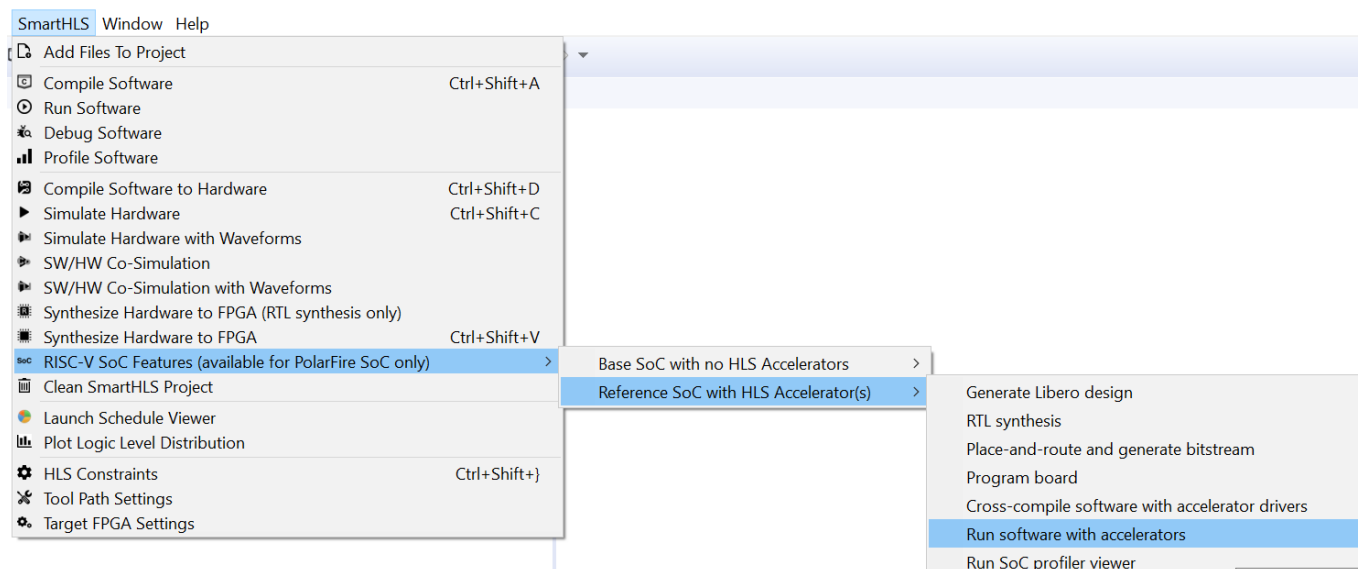


Figure 7-7 Run Software with Accelerators Option Menu

If everything works correctly, you will see the output from the executable running on the board, and the output should match Figure 7-6. In this case, the vector add computation is being performed by the hardware accelerator generated by SmartHLS.

8. Integrating SmartHLS into an Existing SoC design

8.1 Motivation

Until this point, we have been targeting the Reference SoC Libero project generated by SmartHLS. This allows users with no experience using FPGAs to port C++ code to PolarFire SoC devices and offload parts of the software to the FPGA fabric without knowing much about Libero's TCL commands, Verilog or VHDL. SmartHLS provided a *fully automated flow*.

However, users with FPGA knowledge may already have an existing Libero SoC project, which could be different from the SmartHLS reference SoC. They can also have their own Linux image because of the differences in the device tree or simply because they have different software loaded on the image. In addition to the difference in SoC design and Linux image, advanced Libero users may have their own compilation flows. For example, using scripts to run specific tasks before, during or after calling Libero, with custom steps and setting different options for synthesis, place-and-route, bitstream generation, etc.

This section will show how *SmartHLS can be used as a plugin* into a custom compilation flow, and how to integrate the SmartHLS generated hardware modules into an existing SoC design. As an example, we show how you can integrate a SmartHLS system into the *PolarFire SoC Icicle Kit Reference Design* created by the Embedded Software Systems team, which is shipped with SmartHLS.

8.2 Example: Integrate SmartHLS into the *PolarFire® SoC Icicle Kit Reference Design*

Custom SoC designs can have many different configurations. SmartHLS defines a set of TCL parameters, as shown underlined in Figure 8-1, to simplify the automatic integration of SmartHLS-generated modules into a custom SoC.

SmartHLS Subsystem and Integration Parameters

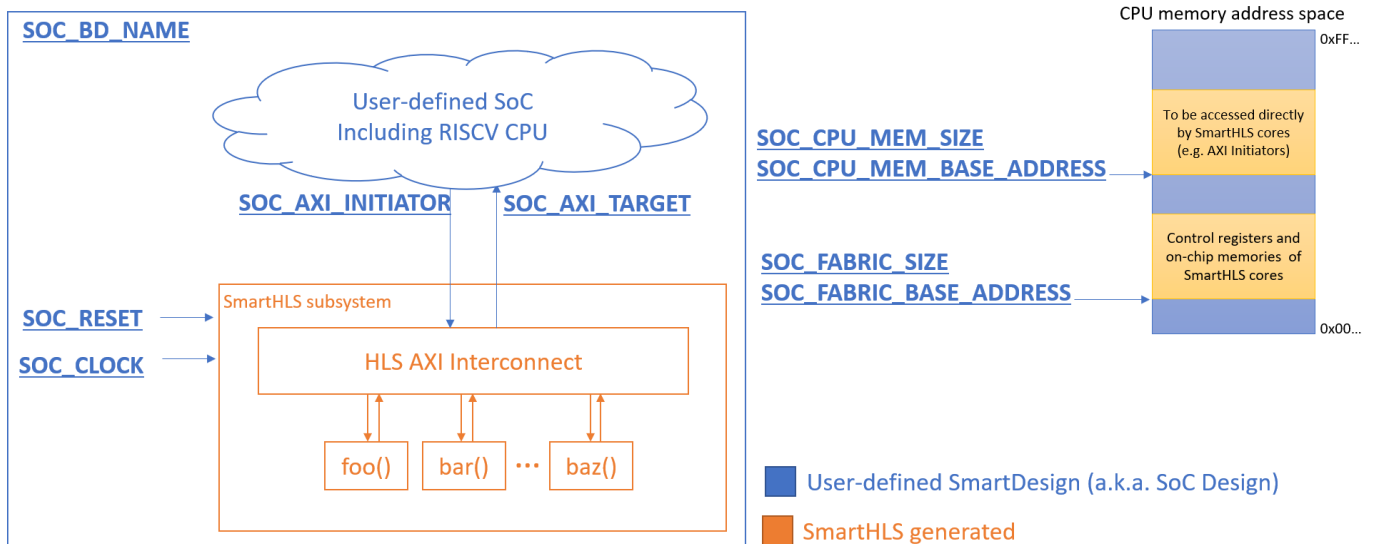


Figure 8-1 TCL Parameters for Interfacing between a custom SoC and SmartHLS Subsystem

Figure 8-2 presents a brief description of each SoC integration parameter in Figure 8-1 that needs to be passed on to SmartHLS to be able to automatically integrate the generated accelerators into a custom SoC.

TCL Parameter	Description
SOC_BD_NAME	The name of the SmartDesign project into which the SmartHLS IP modules will be integrated.
SOC_RESET	Identifies the reset signal to be used.
SOC_CLOCK	Identifies the clock to use for the SmartHLS IP modules. Currently, the same clock is used for all modules.
SOC_AXI_INITIATOR	Identifies the downstream AXI interface to use. This is used for register control and any data write and read transfers initiated by the CPU down to the SmartHLS IP modules.
SOC_AXI_TARGET	Identifies the upstream AXI interface to use. This is used for writing and reading transfer requests issued by the SmartHLS IP modules targeting the CPU memory.
SOC_CPU_MEM_SIZE	This is size of the CPU memory window used when the SmartHLS IP modules act as AXI initiators.
SOC_CPU_MEM_BASE_ADDRESS	This base address identifies the beginning of a memory window in the CPU physical memory address space that the SmartHLS IP modules can use when they are AXI Initiators. This address is used to configure the HLS AXI interconnect and allow transactions to move upstream towards the CPU's memory.

TCL Parameter	Description
SOC_FABRIC_SIZE	Determines the size of the memory window used for mapping control registers and on-chip buffers for ALL modules in each SmartHLS project instantiated on the fabric. The size can be larger than what a specific function may need. For example, a 2MB memory window could be reserved but the IP module may only use half of it, leaving the other half for future growth. Reserving a larger window does not mean more on-chip memory will be used.
SOC_FABRIC_BASE_ADDRESS	This is the base address of a memory window in the CPU memory address space that is reserved for all SmartHLS modules instantiated on the FPGA fabric. Control registers and on-chip memory buffers are allocated and mapped from this memory window. This address is also used to configure the HLS AXI interconnect to allow AXI transactions to move downstream from the CPU towards the SmartHLS IP modules.

Figure 8-2 Description of the TCL Parameters.

These parameters allow SmartHLS not just to convert C++ functions into IP cores, but also to:

- Create SmartDesign HDL+ wrappers
- Instantiate an AXI interconnect and configure its address decoding
- Attach the HDL+ cores to the AXI interconnect
- Connect the clock signal (same clock for all HW modules and interconnect)
- Connect the reset signal
- Connect to the CPU via AXI channels (Initiator & Target)

Users can perform these steps by hand in the GUI or using TCL commands. However, with SmartHLS it is very easy to add and remove functions to the system and having an automated way of doing this is very helpful.

SmartHLS uses the *Icicle Kit Reference Design* as the base design to which the accelerators are automatically attached to. For the *Reference Design*, these parameters have default values and only need to be adjusted for custom SoCs. These default parameters are specified in the SmartHLS *config.tcl* file, for example:

```
#
# Parameters used for SoC integration
#

set_parameter SOC_BD_NAME                MPFS_ICICLE_KIT_BASE_DESIGN
set_parameter SOC_AXI_INITIATOR           AXI2AXI_TO_HLS:AXI4_MASTER
set_parameter SOC_AXI_TARGET              AXI2AXI_FROM_HLS:AXI4_SLAVE
set_parameter SOC_RESET
CLOCKS_AND_RESETS:RESETN_CLK_125MHz
set_parameter SOC_CLOCK                   CLOCKS_AND_RESETS:CLK_125MHz
```

```
# Using FIC-0 Address range: 0x7000_0000 - 0x7040_0000 (4MB)

set_parameter SOC_FABRIC_BASE_ADDRESS      0x70000000
set_parameter SOC_FABRIC_SIZE              0x400000

# Starting from Cached memory base address (0x80000000) all the way up to
# just
# before FIC-1 (~1.7GB)
# NOTE. In the Icicle board not all the memory is contiguous for buffer
# allocation.
# The SW driver should know about those memory partitions. On the hardware
# side,
# it's just easier to set the max address range and rely on the software
# driver
# to not program memory accesses in invalid regions.

set_parameter SOC_CPU_MEM_BASE_ADDRESS      0x80000000
set_parameter SOC_CPU_MEM_SIZE              0x60000000
```

Figure 8-3 Default Parameter Values for Integrating SmartHLS

Users can change the default parameters by [creating a `custom_config.tcl`](#) file inside their HLS project. For example, if we wanted to change the `SOC_FABRIC_BASE_ADDRESS` to start at `0x70100000`, we would include the following in our `custom_config.tcl` file:

```
set_parameter SOC_FABRIC_BASE_ADDRESS      0x70100000
```

Figure 8-4 Custom Parameter Values for Integrating SmartHLS

Note: If the GUI is not used, users must add the following line to their `Makefile.user`:

```
LOCAL_CONFIG += -legup-config=custom_config.tcl
```

Figure 8-5 Additional Makefile Line

This change works with our current *Icicle Kit Reference Design* (though it is not needed, as the default parameters work fine.) This exercise is used to demonstrate how given a different reference design, the SoC integration parameters may be changed, as long as the changes are valid for that specific design.

8.3 Custom Flow Integration

The difference in the compilation processes between the Custom Flow and SoC Flow that we have covered in Section 6.3 resides in what tool drives the flow. In Section 6.3, we used SmartHLS GUI as the main entry point and driver of the compilation process. SmartHLS has TCL scripts to generate the HDL hardware modules from the C++ description and integrate them automatically. In this case SmartHLS calls Libero to

perform different tasks, such as synthesis, place and route, etc. The series of compilation steps are defined by SmartHLS.

In a custom flow, users are responsible for integrating SmartHLS generated subsystem into their own SoC. In this example, **our custom design** based off the PFSoc Icicle Kit Reference Design has a TCL file that drives the overall compilation process. The compilation steps are defined in a file called *MPFS_ICICLE_KIT_REFERENCE_DESIGN.tcl* and the compilation steps are shown in Figure 8-6. This script is executed by Libero, and the TCL script goes through a series of steps, and then calls SmartHLS only as an extra step to generate the HDL modules for the C++ functions. Once the HDL modules have been generated, then SmartHLS can automatically integrate them into the design. This custom flow (not SmartHLS) continues with synthesis, place-and-route, and bitstream generation.

Calling SmartHLS from a custom Libero TCL script

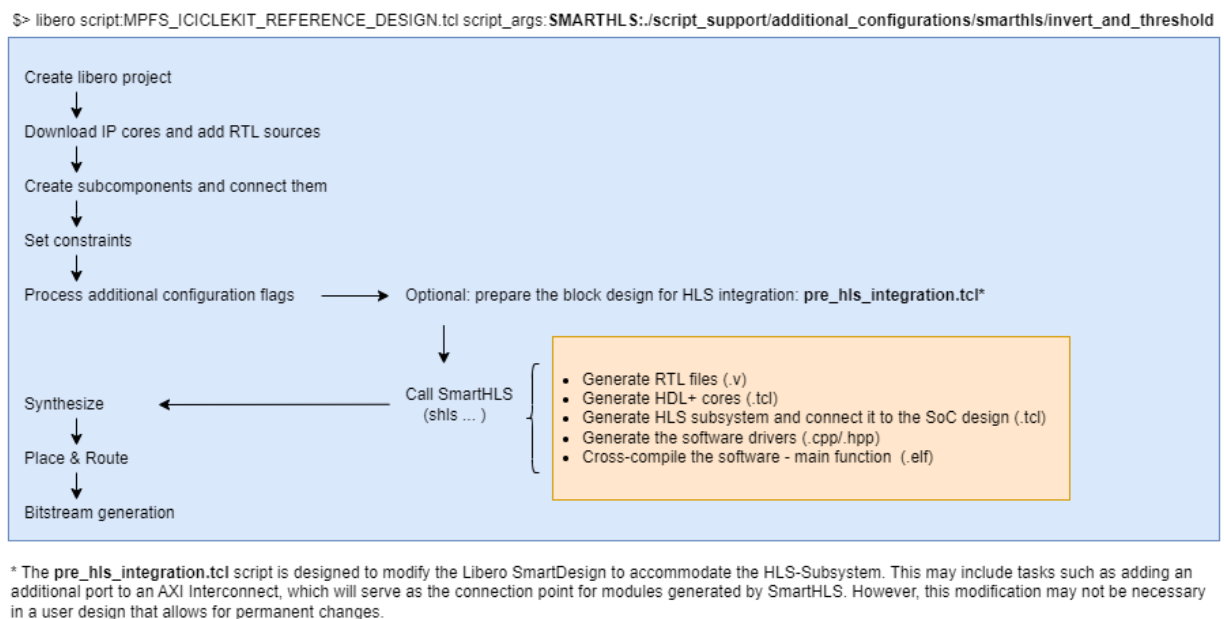


Figure 8-6 Steps for User-Defined SoC with SmartHLS Integration

Compilation of the SmartHLS modules can be done *on-the-fly* using a TCL script. An example script that users may write to call SmartHLS in their custom flow is provided in the *compile_and_integrate_shls_to_refdesign.tcl* file, which you will see in Figure 8-7.

After SmartHLS generates HDL modules from their C++ description, they can be integrated by hand in Libero's GUI or automatically by sourcing SmartHLS-generated TCL script, *shls_integrate_accels.tcl*, as shown in Figure 8-7.

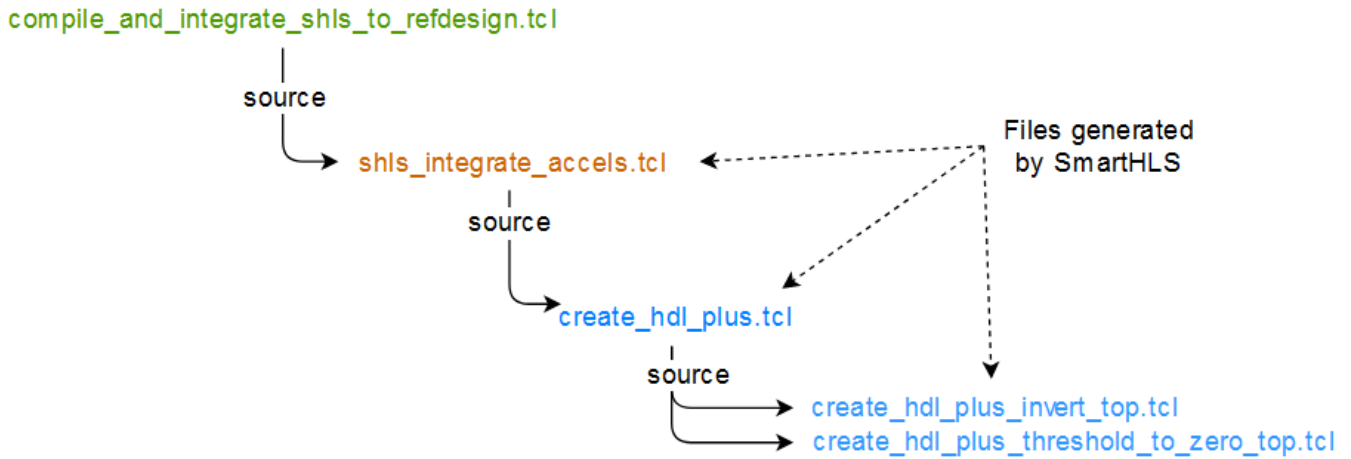


Figure 8-7 TCL Scripts Hierarchy

Figure 8-7 shows the hierarchy of the TCL scripts generated by SmartHLS. The user script, `compile_and_integrate_shls_to_refdesign.tcl`, can source the SmartHLS generated script `shls_integrate_accels.tcl`, which is responsible for generating the SmartHLS subsystem. In turn, `shls_integrate_accels.tcl` will source `create_hdl_plus.tcl`. `create_hdl_plus.tcl` is a SmartHLS-generated TCL script which can be run by Libero to automatically import the generated Verilog files into a SmartDesign HDL+ component, which can then be integrated with existing SmartDesign projects.

8.4 Simple Image Processing Example

We now introduce a simple image processing example to highlight some performance and resource aspects to keep in mind when using the SmartHLS SoC flow. These examples are kept deliberately simple to make straightforward explanations of the necessary concepts. Our objective is not to produce the fastest, most useful image filters.

We will be working with two hardware modules: a pixel value inversion (i.e. simply flip the bits of every pixel value) and a `threshold_to_zero` transformation. The latter is defined as:

$$pixOut(x, y) = \begin{cases} pixIn(x, y), & pixIn(x, y) > thresh \\ 0, & otherwise \end{cases}$$

1920x1080 RGB bitmap (toronto.bmp). Image size: 6.2 MB

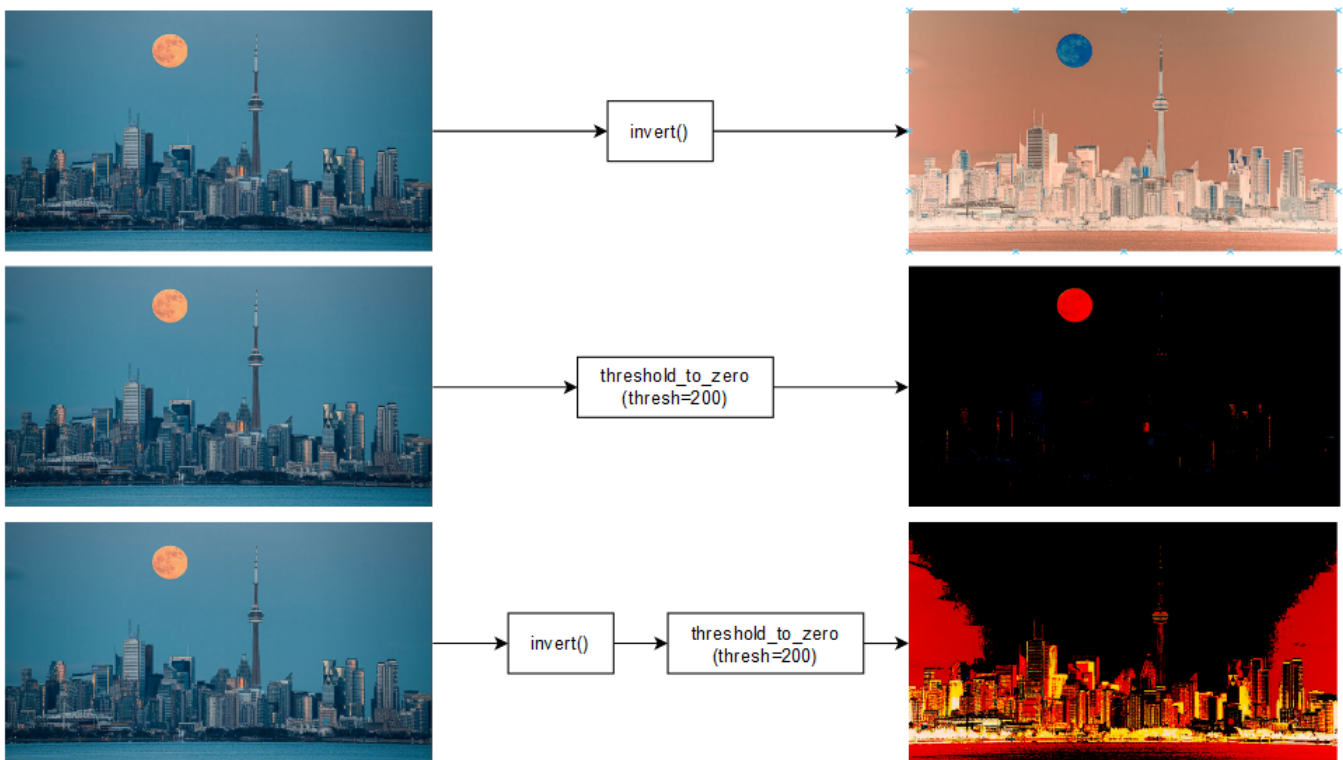


Figure 8-8 Visual Example of Invert and Threshold_to_Zero Transformations

8.5 Flashing PolarFire® SoC Icicle Kit Reference Design

As in Section 7, a Linux image needs to be flashed to the eMMC memory in the Icicle board. If users have already flashed the Linux image as described in the [Icicle Setup Guide](#), this section may be skipped, and users may move on to Section 8.6. A similar procedure can be followed for the user's own Linux image when integrating SmartHLS design into their own existing system.



If you haven't already, please download **core-image-minimal-dev-icicle-kit-es.wic.gz** (See Section 3.2: Download Training Design Files). This Linux image is the Icicle Reference Image and has the same or extended functionality compared to the pre-programmed FPGA design on the Icicle Kit. This Linux image is from [PolarFire SoC Yocto BSP \(v2023.02.1\)](#).



Follow the instructions on [Icicle Setup Guide](#) for setting the Icicle kit. As explained in the guide, when flashing the Icicle board (Step 5 in the Icicle Setup Guide), use **core-image-minimal-dev-icicle-kit-es.wic.gz** that you have downloaded in the previous step. It is important that the Linux image version, design version, Libero version, and Hart Software Services (HSS) version all match/are all compatible with each other. Failure to do this will result in unexpected behaviour. Flashing the Linux image in this step could take 15-30 minutes.

8.6 Extract the Icicle Kit Reference Design Files



Navigate to `<SMARTHLS_INSTALLATION_DIR>\SmarthLS\boards\iciclekit\ref_design`. Rename `ref_design` to `icicle-kit-reference-design`, and move it to your `C:\` drive. Open it, and you should see the following files and directories:

Name	Type	Compressed size	Password ...
.ci	File folder		
.github	File folder		
diagrams	File folder		
script_support	File folder		
XML	File folder		
.gitignore	GITIGNORE File	1 KB	No
LICENSE.md	MD File	1 KB	No
MPFS_ICICLE_KIT_REFERENCE_DESI...	TCL File	4 KB	No
Readme.md	MD File	9 KB	No

Figure 8-9 Icicle Kit Reference Design Folder

In this case, the `MPFS_ICICLE_KIT_REFERENCE_DESIGN.tcl` tcl script is used to drive the custom flow (shown previously in Figure 8-6). We run this tcl script from Libero GUI with the SMARTHLS script argument set to point to a directory where the SmarthLS project is located.

The SmarthLS project files are located under the directory: `icicle-kit-reference-design\script_support\additional_configurations\smarthls` as shown in Figure 8-10.

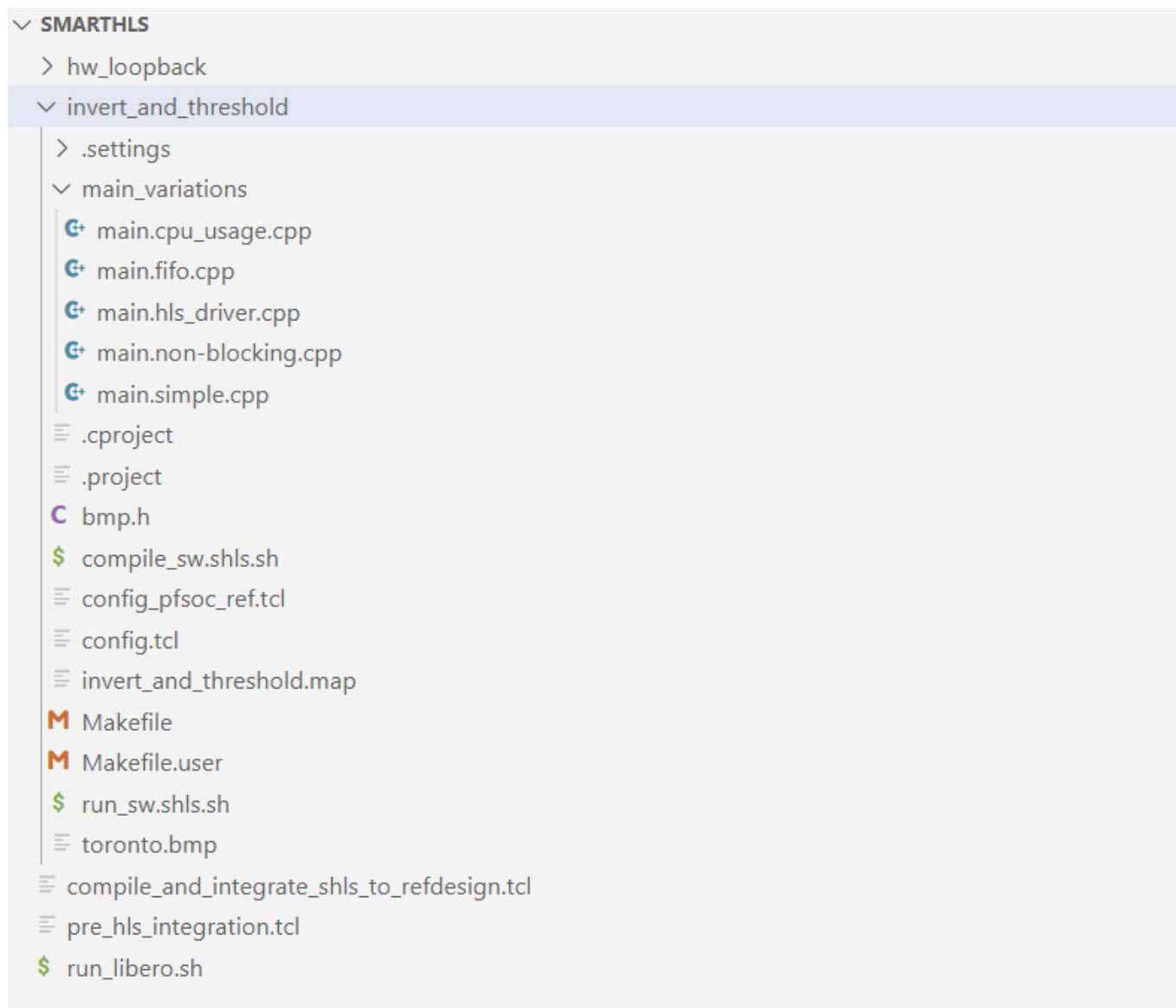


Figure 8-10 Files in SmartHLS Projects

A description of each file in `invert_and_threshold` is given in Figure 8-11.

File Name	Description
main_variations	<ul style="list-style-type: none"> Includes variations of the source code to show case different aspects of SmartHLS. Each file can be selected in the Makefile and will be described below
bmp.h	<ul style="list-style-type: none"> Bitmap read/write functions Timestamp function Constants
compile_sw.shls.sh	<ul style="list-style-type: none"> Convenience script to compile the software with and without HW module drivers

File Name	Description
config_pfsoc_ref.tcl	<ul style="list-style-type: none"> • SmartHLS configuration file including SoC integration parameters
Makefile	<ul style="list-style-type: none"> • Autogenerated makefile by SmartHLS IDE
Makefile.user	<ul style="list-style-type: none"> • SmartHLS project options, including: <ul style="list-style-type: none"> ◦ Select the application variation (i.e. source code to compile) ◦ Runtime settings, such as input arguments, and input and output files
run_sw.shls.sh	<ul style="list-style-type: none"> • Convenience script to copy files to the board and run both binaries: with and without HW support
toronto.bmp	<ul style="list-style-type: none"> • Input reference image file
compile_and_integrate_shls_to_refdesign.tcl	<ul style="list-style-type: none"> • Calls SmartHLS to generate HDL from C++ and integrate the modules into the SoC
pre_hls_integration.tcl	<ul style="list-style-type: none"> • Modifies the Icicle Kit Reference Design by adding the necessary AXI ports to integrate the generated HLS modules

Figure 8-11 Description of Various Files in SmartHLS Example

8.7 Compiling the hardware

In this section, we are going to generate a Libero project and the bitstream for the PolarFire SoC Reference Design, but with a SmartHLS subsystem that contains an invert function accelerator and a `threshold_to_zero` function accelerator connected. We have generated the bitstream in advance and stored the bitstream in the `SmartHLS_Training4_Jobs` folder downloaded in section 3.3. Users can save time by using the precompiled bitstream instead and continue onto the next section, section 8.8.



We can compile by either using the Libero GUI or by running the `run_libero.sh` script in `icicle-kit-reference-design\script_support\additional_configurations\smarthls`.

To use the Libero GUI to compile, open Libero. Press Ctrl+U in Libero to open the “Execute Script” dialog as shown in Figure 8-12. In the “Script” file field, enter the path to the `MPFS_ICICLE_KIT_REFERENCE_DESIGN.tcl` script. In the “Arguments” field enter the following:

```
SMARTHLS:C:\icicle-kit-reference-  
design\script_support\additional_configurations\smarthls\invert_and_thresho  
ld  
EXPORT_FPE:C:\icicle-kit-reference-design\MPFS_ICICLE_SMARTHLS_DEMO \  
HSS_UPDATE:1
```

The above argument assumes that you have extracted the reference folder into `C:\`. Change the path accordingly if you have extracted the folder to elsewhere. The first part of the argument, `SMARTHLS:<Path to SmarthLS Project>`, informs the script where is the SmarthLS project to be built and integrated into the Icycle Kit reference design. The second part, `EXPORT_FPE:<Path>`, specifies the location of the output .job file; the last argument, `HSS_UPDATE:1`, updates the Hart Software Services (HSS) that performs boot and system monitoring functions for PolarFire SoC.

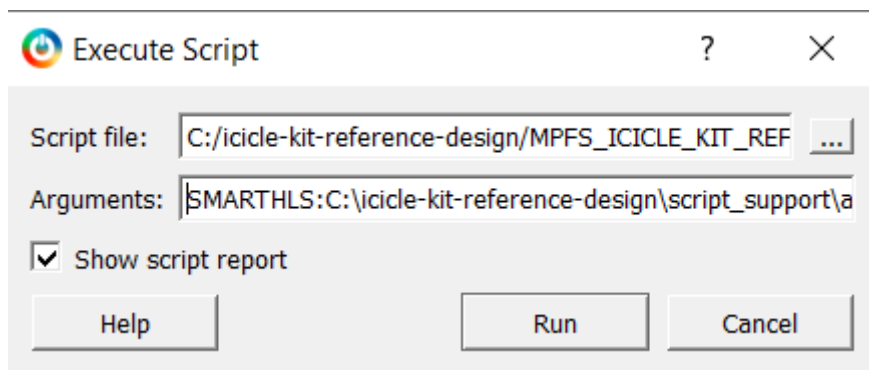


Figure 8-12 Libero's Execute Script Window

`MPFS_ICICLE_KIT_REFERENCE_DESIGN.tcl` is the main driver script that generates an Icycle kit reference demo design.

```
223 # Compile and integrate the SmarthLS code  
224 if {[info exists SMARTHLS]} {  
225     # Prepare the SmartDesign for HLS integration  
226     source  
./script_support/additional_configurations/smarthls/pre_hls_integration.tcl  
227     # Call SmarthLS tool  
228     source  
./script_support/additional_configurations/smarthls/compile_and_integrate_s  
hls_to_refdesign.tcl  
229 }
```

Figure 8-13 SmarthLS configuration of `MPFS_ICICLE_KIT_REFERENCE_DESIGN.tcl`

Figure 8-13 shows a snippet of `MPFS_ICICLE_KIT_REFERENCE_DESIGN.tcl`. When compiling with SmarthLS, `MPFS_ICICLE_KIT_REFERENCE_DESIGN.tcl` sources two scripts. The first script,

`pre_hls_integration.tcl`, modifies the Icicle Kit Reference Design by adding the necessary AXI ports to integrate the generated HLS modules. When you open `pre_hls_integration.tcl` under `icicle-kit-reference-design/additional_configurations/smarthls/`, you will see two very long `configure_core` TCL commands at the start of the script. The first `configure_core` command configures FIC0_INITIATOR to have 4 AXImslaves instead of 3. The second `configure_core` command configures PCIE_INITIATOR to have 2 AXImmasters instead of 1. The additional ports are needed to connect to the SmartHLS subsystem.

The second script, `compile_and_integrate_shls_to_refdesign.tcl`, takes in a SmartHLS project and calls SmartHLS to generate HDL from C++ and integrate the modules into the SoC. This script attempts to obtain the path to SmartHLS based on the user's PATH. If the script cannot find SmartHLS, the script will attempt to look in the default `C:\Microchip\SmartHLS-v2024.1` installation path for Windows. If SmartHLS still cannot be found, the script will give an error and users will have to manually modify the script or add SmartHLS to their PATH environment variable.

To use `run_libero.sh` to compile, you will need to be able to run `bash` scripts. If you cannot do this, you will need to use the GUI to compile.

First, open your shell and navigate to the `icicle-kit-reference-design` folder. Then run `./script_support/additional_configurations/smarthls/run_libero.sh`.

```
01  #!/bin/bash
02  #
03  # Usage:
04  #   cd icicle-kit-reference-design
05  #   ./script_support/additional_configurations/smarthls/run_libero.sh
06  #
07  set -e
08
09  prjDir=soc
10
11  HLS_PATH=./script_support/additional_configurations/smarthls/invert_and_threshold
12
13  #
14  # Start from a clean state
15  #
16  rm -rf \
17    $HLS_PATH/hls_output \
18    $prjDir
19
20  #
21  # Compile the Icicle reference design
22  #
23  target=SMARTHLS:$HLS_PATH+EXPORT_FPE:./$prjDir+HSS_UPDATE:1
24  time libero \
25    script:MPFS_ICICLE_KIT_REFERENCE_DESIGN.tcl \
26    script_args:$target \
27    logfile:$prjDir/MPFS_ICICLE_KIT_REFERENCE_DESIGN.log
```

Figure 8-14 Script: run_libero.sh

This script does essentially the same thing as what a user would do to run `MPFS_ICICLE_KIT_REFERENCE_DESIGN.tcl` using the Libero GUI (see instructions above on how to compile the hardware using the Libero GUI.)

8.8 Programming the FPGA bitstream

After generating the project, we can program the Icicle board using FlashPro Express. FlashPro Express comes packaged with the Libero installation.



Open FPEXpress. The program can be found by pressing the Windows key and searching for “FPEXpress”. Linux users can find FPEXpress under `<Libero Installation Folder>/Libero/bin/`



Click *New...*, select *Import FlashPro Express job file* radio button, and navigate to Icicle reference design folder to select the generated bitstream from Section 8.7 `<icicle-kit-reference-design>\soc\Icicle_SoC.job`

If you have skipped the previous section, you can program with the precompiled .job file in the Jobs folder `SmArthLS_Training4_Jobs\INVERT_AND_THRESHOLD_SIMPLE.job`.

Set your FPEXpress project location to wherever you please, then click *OK*.

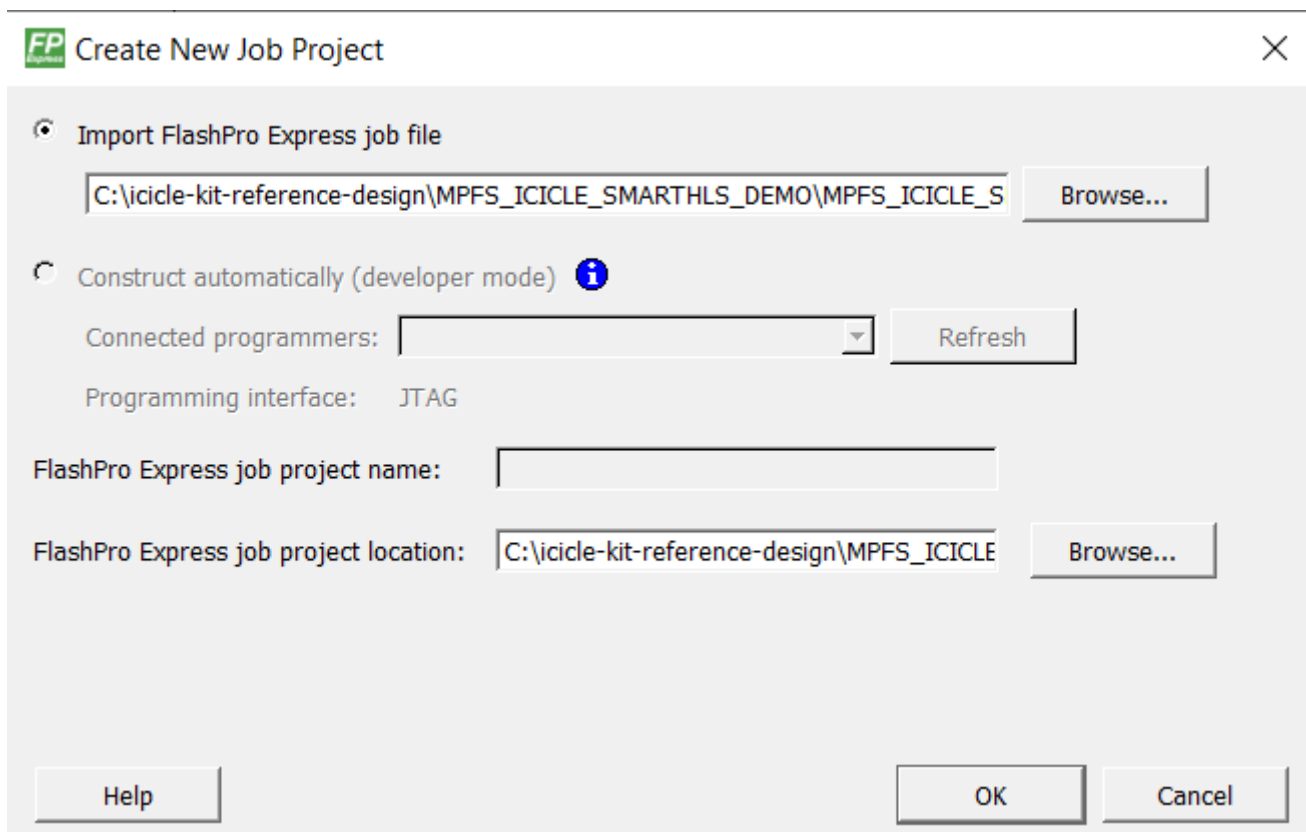


Figure 8-15 Create New Job Project Setting



From the drop-down box above the *RUN* button make sure that *PROGRAM* is selected.

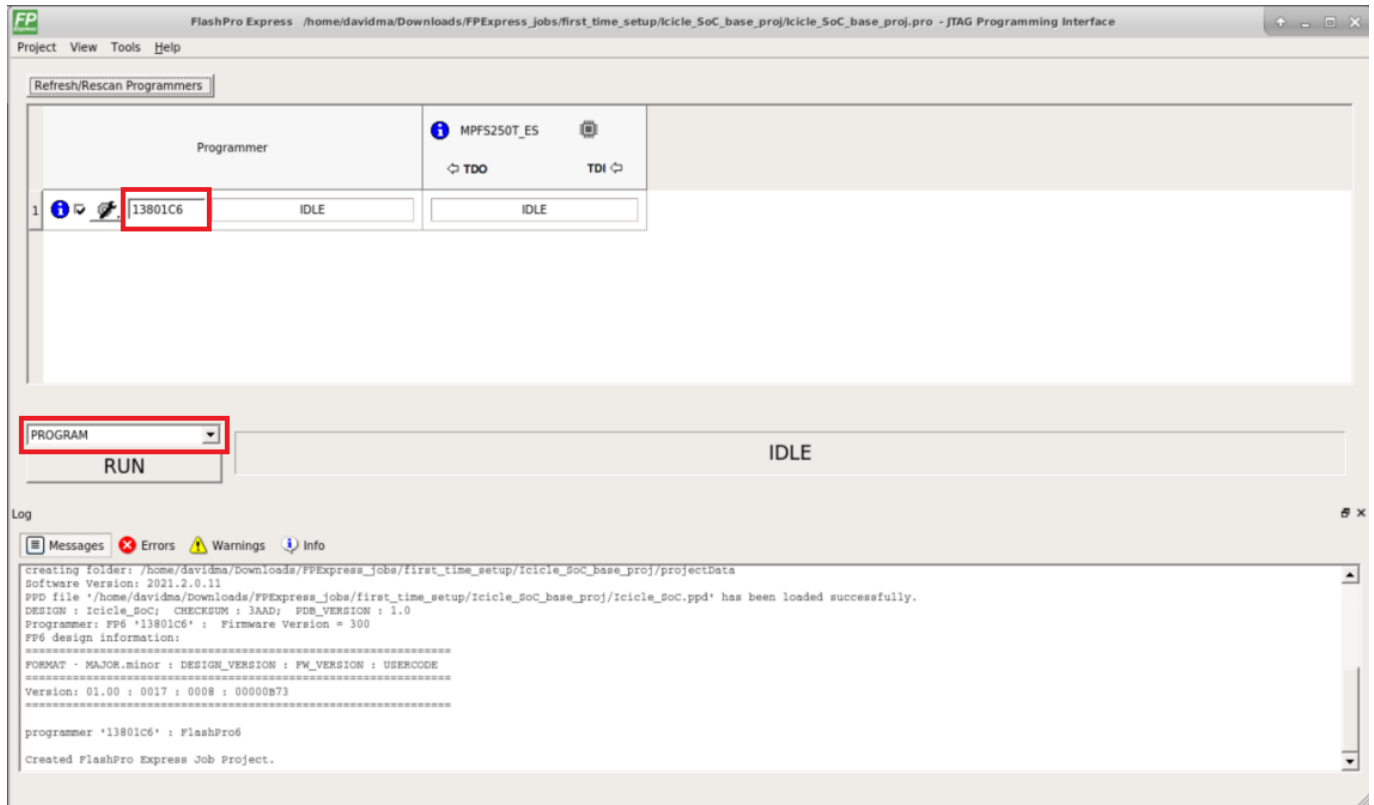


Figure 8-16 FlashPro Express Program Screen



Now press the *RUN* button, and you should see a confirmation that the programming passed:

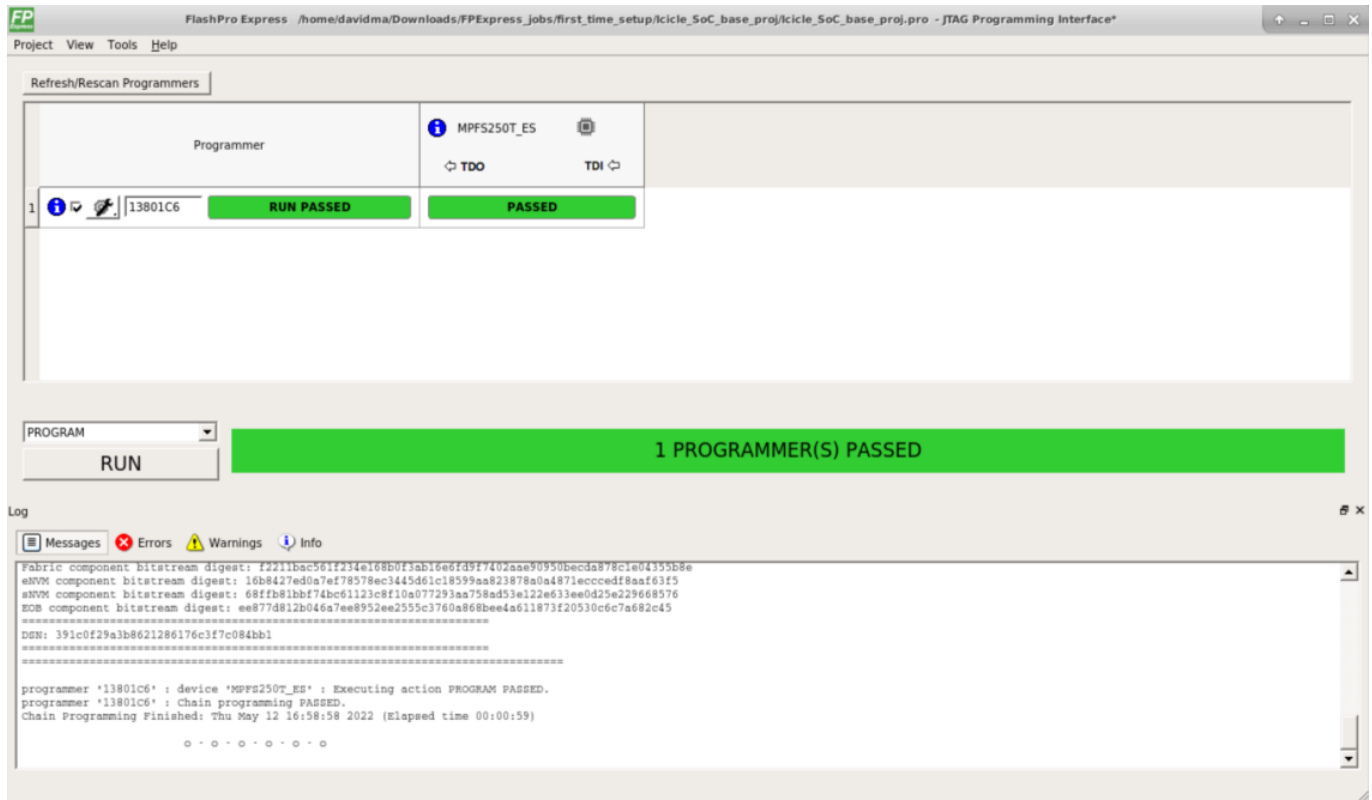


Figure 8-17 Program Successful

This will program a default bitstream to the FPGA fabric, as well as a compatible bootloader (HSS), which will allow the board to boot up with the newly added Linux image.



After the board has successfully booted, you can connect using a serial terminal. Connect in the same manner as the serial terminal used during the writing of the Linux image, except this time using channel 1 (`/dev/ttyUSB1` on Linux, and **Interface 1** on Windows), you should see a login screen:

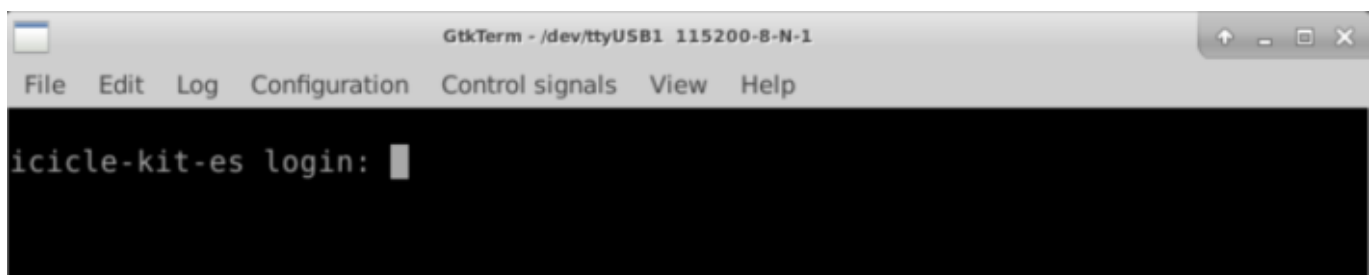


Figure 8-18 Login Screen

The login is **root**, and no password is required.



After logging in, you should be able to see a terminal. Now enter **ifconfig**, look for **inet** and take note of the IP address that should have been assigned to the Icicle Kit by the network:

```

root@icicle-kit-es:~# ifconfig
eth0: flags=4099<UP,BROADCAST,MULTICAST> mtu 1500
    ether 00:04:a3:09:cb:e8 txqueuelen 1000  (Ethernet)
    RX packets 0  bytes 0 (0.0 B)
    RX errors 0  dropped 0  overruns 0  frame 0
    TX packets 0  bytes 0 (0.0 B)
    TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0
    device interrupt 26

eth1: flags=4163<UP,BROADCAST,RUNNING,MULTICAST> mtu 1500
    inet 192.168.8.48 netmask 255.255.255.0 broadcast 192.168.8.255
    inet6 fe80::204:a3ff:fe09:cbe7 prefixlen 64 scopeid 0x20<link>
    inet6 fe80::d124:f472:74aa:b846 prefixlen 64 scopeid 0x20<link>
    ether 00:04:a3:09:cb:e7 txqueuelen 1000  (Ethernet)
    RX packets 13  bytes 1870 (1.8 KiB)
    RX errors 0  dropped 0  overruns 0  frame 0
    TX packets 40  bytes 4352 (4.2 KiB)
    TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0
    device interrupt 32  base 0x2000

lo: flags=73<UP,LOOPBACK,RUNNING> mtu 65536
    inet 127.0.0.1 netmask 255.0.0.0
    inet6 ::1 prefixlen 128 scopeid 0x10<host>
    loop txqueuelen 1000  (Local Loopback)
    RX packets 80  bytes 6080 (5.9 KiB)
    RX errors 0  dropped 0  overruns 0  frame 0
    TX packets 80  bytes 6080 (5.9 KiB)
    TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0

```

Figure 8-19 Getting IP Address from ifconfig

Now that the IP address of the board is determined, you can access it remotely over the network using SSH with the command `ssh root@[your board IP here]`.

8.8.1 Chaining HW modules using CPU shared memory (main.simple.cpp)

We will now explore different versions of the image filter introduced in Section 8.4. The goal of this exercise is demonstrating the design considerations that should be taken and the understanding how the system should work as a whole. We implemented a simple version of invert and `threshold_to_zero` functions in `main.simple.cpp`. Although we will not be using the SmartHLS IDE for compilation, we will be using the SmartHLS IDE for exploring and editing the code.



Open the SmartHLS project under your Icicle Kit Reference Design folder. Go to *File -> Open Projects from File System...* (Figure 8-20), and then in *Import Source*, open the SmartHLS project under your Icicle Kit Reference Design Folder (Figure 8-21), under `script_support/additional_configurations/smarthls/invert_and_threshold`.

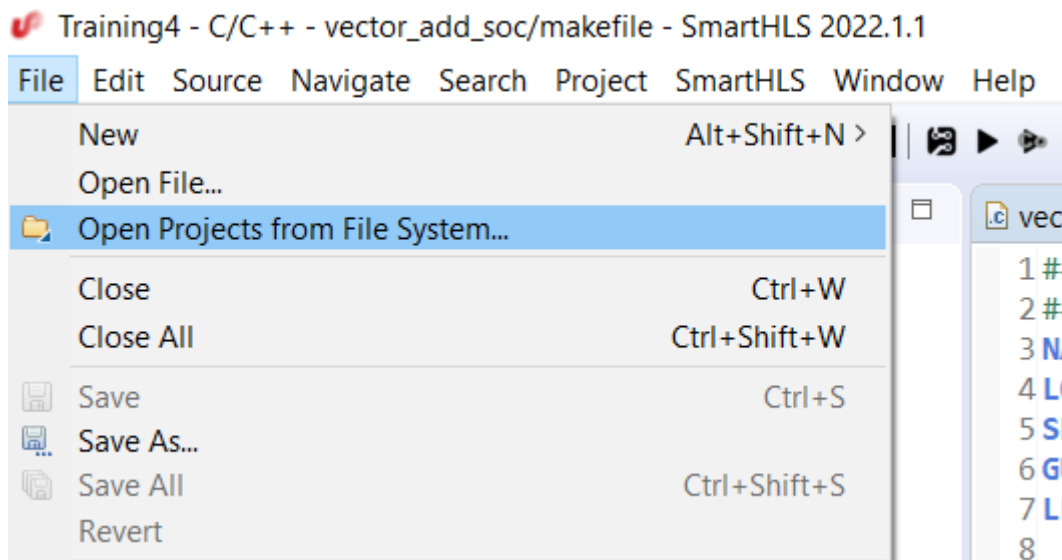


Figure 8-20 Open Projects from File Menu

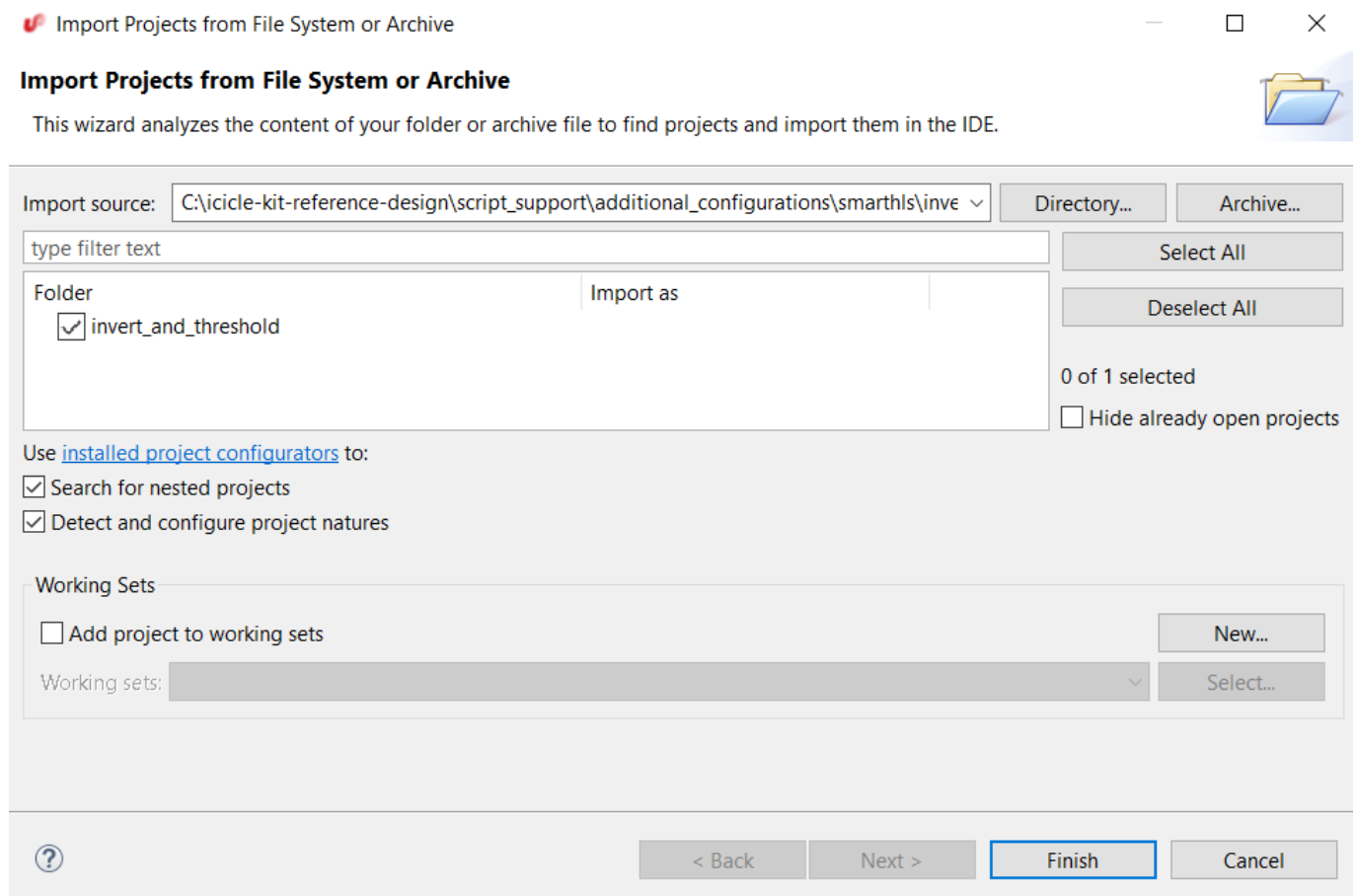


Figure 8-21 Import Projects from File System Settings



Open `main_variations/main.simple.cpp`. The file contains two *top* functions, `invert` and `threshold_to_zero`. Each *top* function is an independent hardware module connected to the AXI interconnect. There is a limit to the PolarFire SoC FPGA on-chip memory of about 2MB for the entire FPGA fabric (MPFS250T part on the Icicle kit). Thus, we have to split the large Full-HD image (1920x1080) into multiple blocks. We have set the `N_ROWS` constant in `bmp.h` to process the input image 45 rows at a time.

The program takes in two arguments. The first argument is either 0 or 1. When the first argument is 0, the program will not perform pixel inversion; otherwise, the program will. The second argument is the threshold ranging from 0 to 255. A zero-value threshold will bypass the `threshold_to_zero()` module.

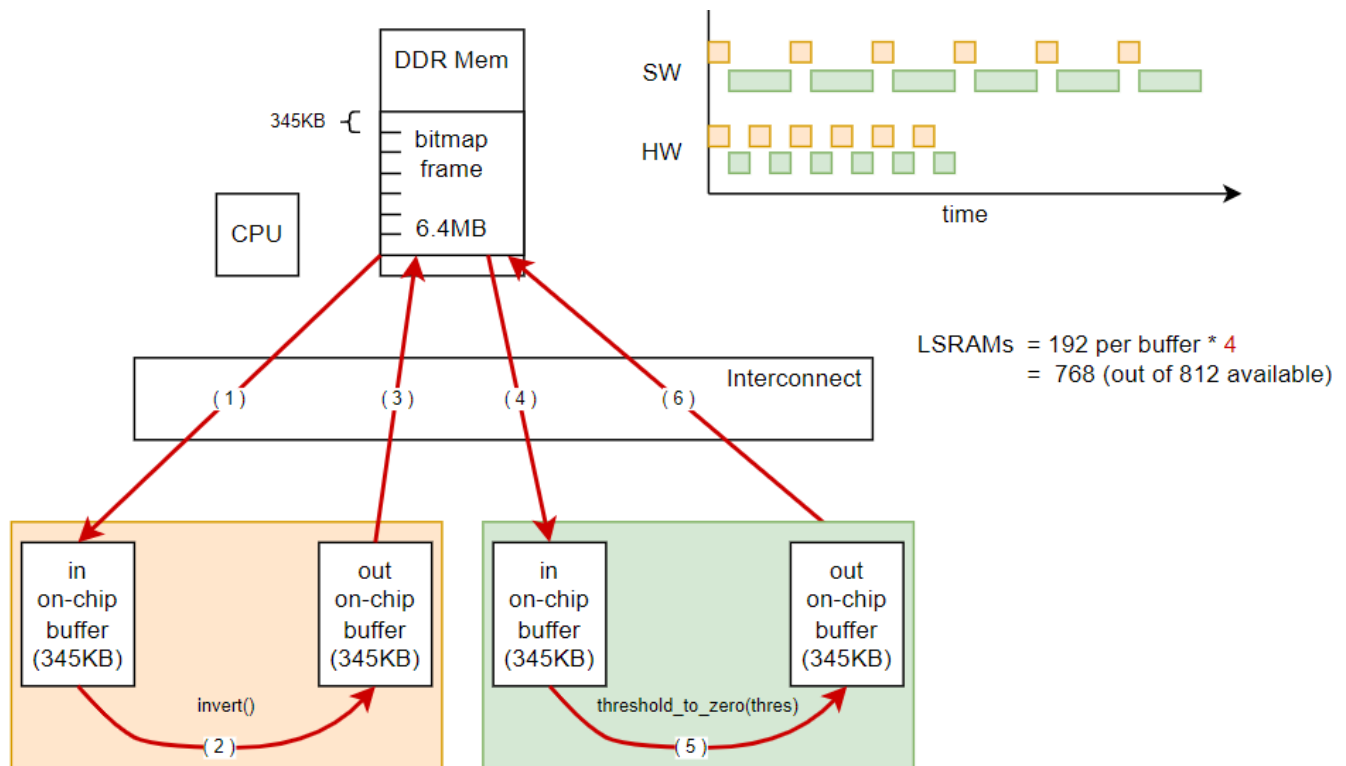


Figure 8-22 Data Movement of main.simple.cpp

Each hardware module has an input and output on-chip buffer to store the incoming and output data. Figure 8-22 shows the data movement of one block of data. First, the CPU initiates a DMA read from the DDR memory to the input on-chip buffer of the `invert()` module, performs the inversion operation and stores the result into the output on-chip buffer. After that the CPU initiates another DMA transaction to write back to the DDR memory. `threshold_to_zero()` follows a similar flow. In total there are four DMA operations.



We can run the software and hardware accelerated versions of the code for comparison. If you are using Linux, open the command line interface. If you are using Windows, open the command prompt (cmd).

Add `<SmarthLS Installation Path>/SmarthLS/bin` your PATH environment variable. If you are using Linux, enter the following command:

```
$ export PATH=<SmarthLS Installation Path>/SmarthLS/bin:$PATH
```

If you are using Windows, enter the following command:

```
> set PATH=<SmarthLS Installation Path>/SmarthLS/bin;%PATH%
```



After adding SmartHLS to the PATH environment variable, export **BOARD_IP** environment variable so that SmartHLS knows the IP address of the Icicle board. Please refer to Figure 8-19 on finding out the IP of the Icicle board. If you are using Linux, enter the following command:

```
$ export BOARD_IP=<Your Icicle Board IP>
```

If you are using Windows, enter the following command:

```
> set BOARD_IP=<Your Icicle Board IP>
```



Now that we have finished setting up the environment, we can move on compiling and running the software. Go to your Icicle Kit Reference Design folder. If you are using Linux, run the following commands:

```
$ cd script_support/additional_configurations/smarthls/invert_and_threshold
$ ./compile_sw.shls.sh
$ ./run_sw.shls.sh
```

If you are using Windows, run the following commands:

```
> cd script_support\additional_configurations\smarthls\invert_and_threshold
> compile_sw.shls.bat
> run_sw.shls.bat
```

You might see a warning about “REMOTE HOST IDENTIFICATION HAS CHANGED” because we have changed the OS image. Simply remove previous ssh info by doing a “**rm ~/.ssh**” and accept the new RSA fingerprint the next time ssh prompts.

While the code compiles, let’s look at these 2 scripts. The first script, **compile_sw_shls.sh** (Figure 8-24), simply compiles the RISC-V executables with and without accelerators. Line 10 is equivalent to *Cross-compile with accelerator drivers* in Figure 7-7 and Line 13 is equivalent to *Cross-compile software for RISC-V* in Figure 7-5 Run Software without Accelerators Option Menu. The **-a** option tells shls to build all dependencies in Figure 6-26 without prompting.

```

01  #!/bin/bash
02
03  set -eu
04
05  # Remove binaries and results from previous runs
06  ssh root@$BOARD_IP "rm -f output*.bmp *.elf"
07  shls clean
08
09  echo "Compiling w/HW module"
10  shls -a soc_sw_compile_accel
11
12  echo "Compiling SW-only"
13  shls -a soc_sw_compile_no_accel

```

Figure 8-24 Compilation Script: compile_sw.shls.sh

The second script, `run_sw_shls.sh` (Figure 8-25), runs the RISC-V executables with and without accelerators on the board. Line 7 is equivalent to *Run software without accelerators* (Figure 7-5) and line 11 is equivalent to *Run software with accelerators* (Figure 7-7). However, unlike the options chosen from the IDE, line 8 and 11 do not build any dependencies as described in Figure 6-26. The `soc_base_proj_run` and `soc_accel_proj_run` commands skip all build dependencies because we do not wish to program the board with a SmartHLS SoC, we already have programmed our Custom SoC bitstream to the FPGA in Section 8.8.

```

01  #!/bin/bash
02
03  set -eu
04
05  echo "-----"
06  echo "Run SW-only"
07  shls -s soc_base_proj_run
08
09  echo "-----"
10  echo "Run w/HW module"
11  shls -s soc_accel_proj_run

```

Figure 8-25 Run Program Script: run_sw_shls.sh

`Makefile.user` defines various options related to compiling and running the compiled program. Figure 8-26 is a snippet of `Makefile.user` containing the runtime settings. Visit the [Makefile Variable](#) section of our user guide for a full list of predefined user flags and their uses. Important: Ensure that `SRCS` is set to `main_variables/main.simple.cpp`.

```

29  #-----
30  # Runtime settings
31  #-----
32  # Specify the working directory on the board

```

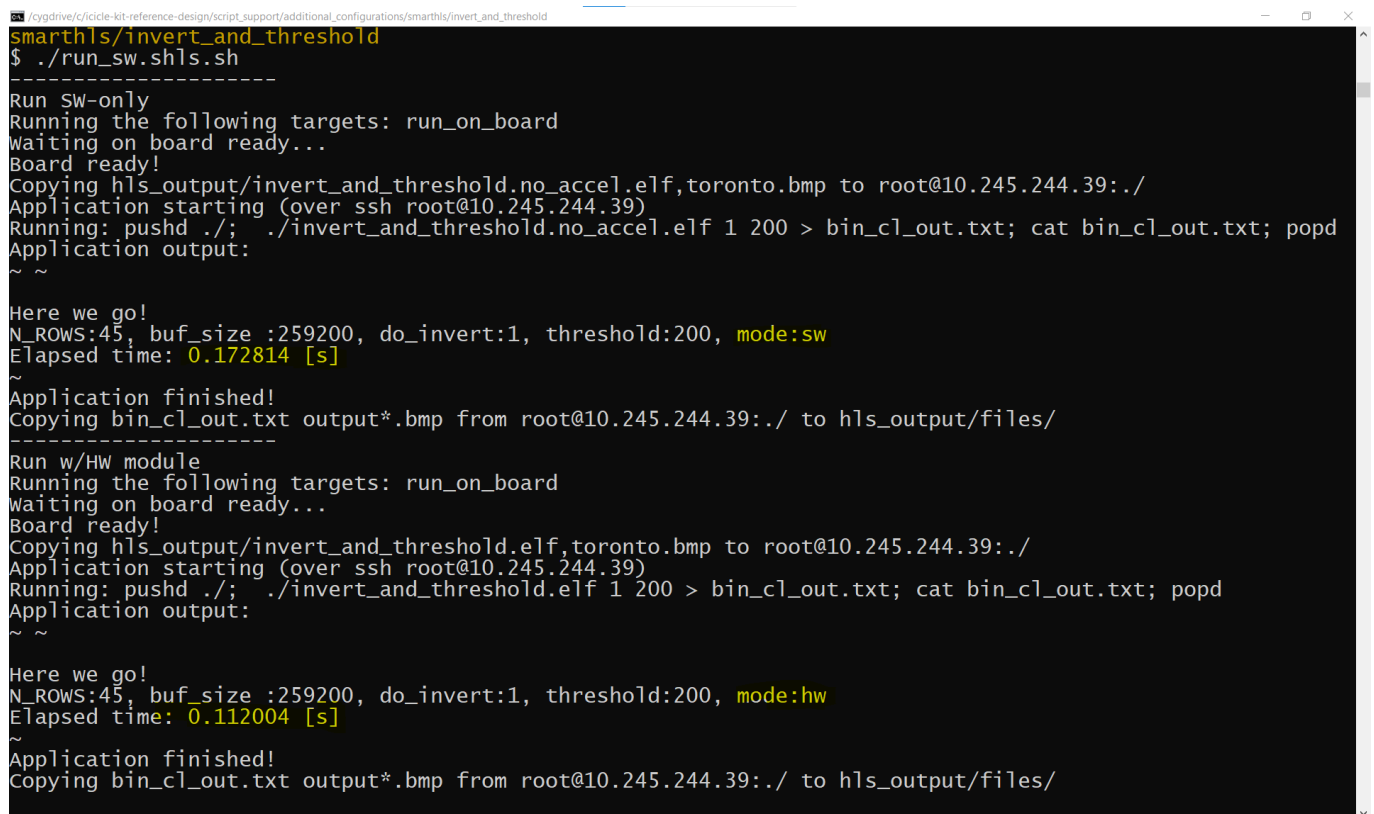
```

33 # All input, output, binaries will be based off this folder.
34 BOARD_PATH = ./
35
36 # INPUT_FILES_RISCV should use host paths.
37 # It lists the files, separated by a space, to be copied onto the board
38 INPUT_FILES_RISCV = toronto.bmp
39
40 # OUTPUT_FILES_RISCV should use on-board paths.
41 # It lists the files, separated by a space, to be copied from the board
42 OUTPUT_FILES_RISCV = output*.bmp
43
44 # Arguments to the program
45 # First argument: <0|1> 0 for skipping invert
46 # 1 for performing invert
47 # Second argument: <0..255> Threshold for not setting pixel to zero
48 PROGRAM_ARGUMENTS = 1 200

```

Figure 8-26 Runtime Settings Section of Makefile.user

If the run was successful, you should see similar output to Figure 8-27 Sample Output of Successful Run below.



```

smarthls/invert_and_threshold
$ ./run_sw.shls.sh
-----
Run SW-only
Running the following targets: run_on_board
Waiting on board ready...
Board ready!
Copying hls_output/invert_and_threshold.no_accel.elf,toronto.bmp to root@10.245.244.39:./
Application starting (over ssh root@10.245.244.39)
Running: pushd ./; ./invert_and_threshold.no_accel.elf 1 200 > bin_cl_out.txt; cat bin_cl_out.txt; popd
Application output:
~ ~

Here we go!
N_ROWS:45, buf_size :259200, do_invert:1, threshold:200, mode:sw
Elapsed time: 0.172814 [s]
~
Application finished!
Copying bin_cl_out.txt output*.bmp from root@10.245.244.39:./ to hls_output/files/
-----
Run w/HW module
Running the following targets: run_on_board
Waiting on board ready...
Board ready!
Copying hls_output/invert_and_threshold.elf,toronto.bmp to root@10.245.244.39:./
Application starting (over ssh root@10.245.244.39)
Running: pushd ./; ./invert_and_threshold.elf 1 200 > bin_cl_out.txt; cat bin_cl_out.txt; popd
Application output:
~ ~

Here we go!
N_ROWS:45, buf_size :259200, do_invert:1, threshold:200, mode:hw
Elapsed time: 0.112004 [s]
~
Application finished!
Copying bin_cl_out.txt output*.bmp from root@10.245.244.39:./ to hls_output/files/

```

Figure 8-27 Sample Output of Successful Run



We can also SSH into the board to run the binaries directly. Log onto the board by entering the following command:

```
ssh root@$BOARD_IP
```

There are two `.elf` files in the home directory. They were copied over when we ran `run_sw.shls.sh`. The exact location of where `shls soc_accel_proj_run` and `shls soc_base_proj_run` are run depends on `BOARD_PATH` defined in `Makefile.user`. We can experiment with running either program with various parameters. The accepted range for the arguments is explained in the comments of `PROGRAM_ARGUMENTS` in Figure 8-26.

```
root@icicle-kit-es:~# ./invert_and_threshold.no_accel.elf 1 0
Here we go!
N_ROWS:45, buf_size :259200, do_invert:1, threshold:0, mode:sw
Elapsed time: 0.046492 [s]
root@icicle-kit-es:~# ./invert_and_threshold.no_accel.elf 0 200
Here we go!
N_ROWS:45, buf_size :259200, do_invert:0, threshold:200, mode:sw
Elapsed time: 0.140887 [s]
root@icicle-kit-es:~# ./invert_and_threshold.no_accel.elf 1 200
Here we go!
N_ROWS:45, buf_size :259200, do_invert:1, threshold:200, mode:sw
Elapsed time: 0.172550 [s]
root@icicle-kit-es:~# ./invert_and_threshold.elf 0 200
Here we go!
N_ROWS:45, buf_size :259200, do_invert:0, threshold:200, mode:hw
Elapsed time: 0.057144 [s]
root@icicle-kit-es:~# ./invert_and_threshold.elf 1 200
Here we go!
N_ROWS:45, buf_size :259200, do_invert:1, threshold:200, mode:hw
Elapsed time: 0.111549 [s]
root@icicle-kit-es:~#
```

Figure 8-28 Sample main.simple.cpp Output

Runtime (ms)	Without Accelerators	With Accelerators
do_invert:0 threshold:0	0	0
do_invert:0 threshold:200	141	57
do_invert:1 threshold:0	46	57
do_invert:1 threshold:200	173	112

Figure 8-29 Runtime of main.simple.cpp

We summarize the runtime of the application with various program arguments in Figure 8-29. When using hardware accelerator, the execution times of either invert or threshold is roughly 55 ms (Figure 8-29). However when running only in software, inversion takes about 46 ms while the `threshold_to_zero()` function takes approximately 141 ms. So, a slight increase in complexity has a big effect on the overall software runtime. 57 ms is approximately the time the MSS needs to move data to and from the accelerator and DDR. This is, in fact, where most of the time is spent in these simple hardware accelerators as can be seen in Figure 8-30.

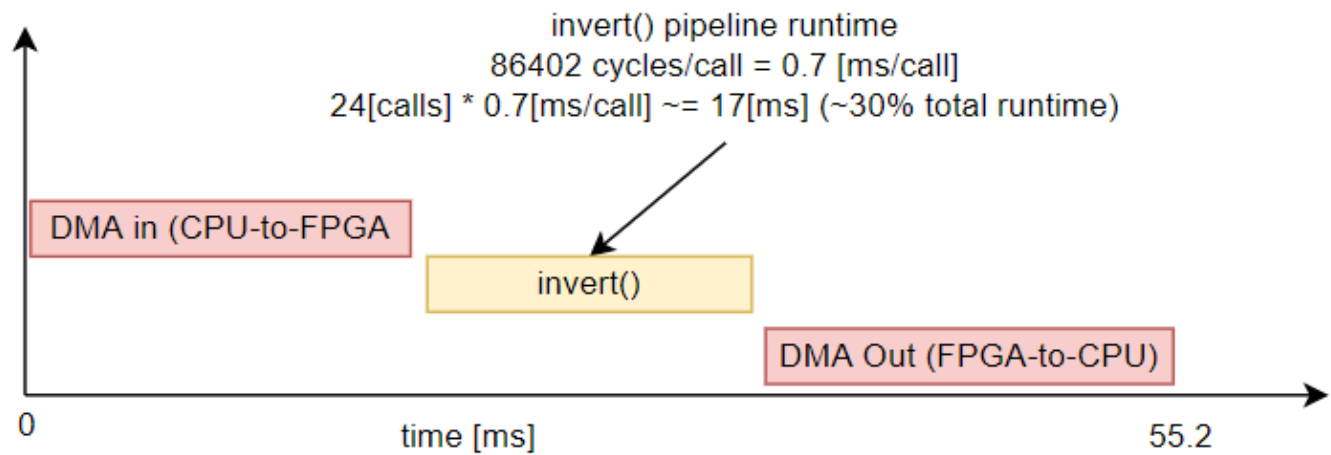


Figure 8-30 Runtime Breakdown of invert()

If we look at the `hls_output/reports/summary.hls.invert.rpt` (Figure 8-31), we can see that the invert latency is 86,402 cycles at 125MHz, which is 0.7ms. The invert function is called 24 times, which means that the invert pipeline total runtime is only about 17ms of the 55.2ms measured runtime. This represents only 30% of the total runtime, with the other 70% spent performing data transfer.

```
===== 2. Function and Loop Scheduling Results =====
+-----+-----+-----+-----+
+
| Label                                | Function | ... | Latency
+-----+-----+-----+-----+
+
| for_loop_main_variations_main_simple_cpp_11_5 | invert   | ... | 86402
+-----+-----+-----+-----+
+
```

Figure 8-31 Pipeline Result of invert()

Chaining the hardware modules introduces a data dependency because the output of the `invert()` module is the input of the `threshold_to_zero()` module. This causes a serialization in the execution as we can see in the alternation of data transfer in Figure 8-22. The hardware modules cannot run in parallel but the three channels per pixel (Red, Green, and Blue) are processed in parallel in the pipeline. When only one of the `invert()` or `threshold_to_zero()` hardware accelerators are called, the processing time is about the same (57 ms) because the data transfer time dominates the overall runtime.

8.8.2 CPU usage (main.cpu_usage.cpp)

Faster execution times is one benefit of offloading functions to the FPGA, the other benefit is leaving the CPU free to perform other tasks. They both contribute to reducing power consumption. In this case, the CPU does not have other tasks to perform, but we will see the CPU usage with and without hardware acceleration.

We introduced a new for loop with `N_ITER` iterations in `main.cpu_usage.cpp` to artificially increase the runtime and be able to see the CPU usage using the Linux `top` command. Think of this loop as a sequence of video frames, where the same processing is performed repeatedly.



Modify `Makefile.user` and select:

```
SRCS = main_variations/main.cpu_usage.cpp
```

Open a terminal on the Icicle board and run the Linux `top` command:

```
$ ssh root@$BOARD_IP
```

```
$ top
```

In another terminal, compile the software and run again. If you are using Linux, run:

```
$ shls clean

$ ./compile_sw.shls.sh

$ ./run_sw.shls.sh
```

If you are using Windows, run:

```
> shls.bat clean

> compile_sw.shls.bat

> run_sw.shls.bat
```

Running software-only causes the CPU to reach 100% usage:

```
top - 14:37:23 up 44 min, 3 users, load average: 0.39, 0.17, 0.07
Tasks: 93 total, 3 running, 90 sleeping, 0 stopped, 0 zombie
%Cpu(s): 25.3 us, 0.2 sy, 0.0 ni, 74.5 id, 0.0 wa, 0.0 hi, 0.0 si, 0.0 st
MiB Mem : 1843.9 total, 1723.4 free, 55.5 used, 64.9 buff/cache
MiB Swap: 0.0 total, 0.0 free, 0.0 used, 1757.8 avail Mem
```

PID	USER	PR	NI	VIRT	RES	SHR	S	%CPU	%MEM	TIME+	COMMAND
490	root	20	0	12192	10740	2468	R	99.7	0.6	0:12.80	invert_and_thre
408	root	20	0	4172	2140	1748	R	0.7	0.1	0:02.11	top
501	root	20	0	4280	2212	1676	R	0.7	0.1	0:00.07	top
1	root	20	0	17832	7760	5172	S	0.0	0.4	0:12.04	systemd
2	root	20	0	0	0	0	S	0.0	0.0	0:00.01	kthreadd
3	root	0	-20	0	0	0	I	0.0	0.0	0:00.00	rcu_gp
4	root	0	-20	0	0	0	I	0.0	0.0	0:00.00	rcu_par_gp
8	root	0	-20	0	0	0	I	0.0	0.0	0:00.00	mm_percpu_wq
9	root	20	0	0	0	0	S	0.0	0.0	0:00.00	rcu_tasks_trace
10	root	20	0	0	0	0	S	0.0	0.0	0:00.05	ksoftirqd/0

Figure 8-32 CPU Usage when Running Software on RISC-V Only

Running with hardware module, the CPU utilization is about 11%:

```
top - 14:38:32 up 45 min, 3 users, load average: 0.15, 0.14, 0.07
Tasks: 92 total, 2 running, 90 sleeping, 0 stopped, 0 zombie
%Cpu(s): 0.3 us, 0.3 sy, 0.0 ni, 99.5 id, 0.0 wa, 0.0 hi, 0.0 si, 0.0 st
MiB Mem : 1843.9 total, 1730.8 free, 48.1 used, 64.9 buff/cache
MiB Swap: 0.0 total, 0.0 free, 0.0 used, 1765.2 avail Mem
```

PID	USER	PR	NI	VIRT	RES	SHR	S	%CPU	%MEM	TIME+	COMMAND
504	root	20	0	300652	2664	2432	S	11.6	0.1	0:01.85	invert_and_thre
501	root	20	0	4280	2212	1676	R	1.0	0.1	0:00.61	top
408	root	20	0	4172	2140	1748	R	0.7	0.1	0:02.64	top
1	root	20	0	17832	7760	5172	S	0.0	0.4	0:12.04	systemd
2	root	20	0	0	0	0	S	0.0	0.0	0:00.02	kthreadd
3	root	0	-20	0	0	0	I	0.0	0.0	0:00.00	rcu_gp
4	root	0	-20	0	0	0	I	0.0	0.0	0:00.00	rcu_par_gp
8	root	0	-20	0	0	0	I	0.0	0.0	0:00.00	mm_percpu_wq
9	root	20	0	0	0	0	S	0.0	0.0	0:00.00	rcu_tasks_trace
10	root	20	0	0	0	0	S	0.0	0.0	0:00.05	ksoftirqd/0

Figure 8-33 CPU Usage when Running with Accelerators

SmartHLS has a TCL parameter called `SOC_POLL_DELAY` with a value specified in microseconds. This parameter is used for controlling how often the hardware driver polls the module to check for completion. Sometimes for long running tasks, the MSS only needs to check occasionally (e.g., every 1 second), instead of many thousands of times per second, which frees up the CPU to do other useful work.

8.8.3 Non-blocking hardware execution

In `main.non-blocking.cpp`, we change the objective and we no longer require chaining the two image transformations as we did before with `main.simple.cpp`. Now we want to output the inverted picture and `threshold_to_zero` picture separately into two different output files. This requires two new buffers to be allocated in memory, to hold the output data of each image transformation. With this change, we have removed the data dependency between the transformations and we can now overlap the computation and data transmission between the hardware modules. To accomplish this, we can use the non-blocking software driver API functions generated by SmartHLS. See Section 6.2.6 for explanation on the generated software driver APIs.

Instead of calling `invert()` or `threshold_to_zero()`, we used a different call in `main.non-blocking.cpp` as shown in Figure 8-34.

```
64 for(int i = 0; i < HEIGHT/N_ROWS; i++) {
65     if (do_invert) {
66         #ifdef HAS_ACCELERATOR
67             invert_write_input_and_start((uint32_t
68 *)&BitMap[i*WIDTH*N_ROWS]);
69         #else
69             invert((uint32_t *)\&BitMap[i*WIDTH*N_ROWS], (uint32_t
70 *)&OutBitMap1[i*WIDTH*N_ROWS]);
71         #endif
72     }
73     if (threshold > 0) {
74         #ifdef HAS_ACCELERATOR
75             threshold_to_zero_write_input_and_start((uint32_t
76 *)&BitMap[i*WIDTH*N_ROWS], threshold);
77         #else
77             threshold_to_zero((uint32_t *)&BitMap[i*WIDTH*N_ROWS], (uint32_t
78 *)\&OutBitMap2[i*WIDTH*N_ROWS], threshold);
79         #endif
80     }
81     #ifdef HAS_ACCELERATOR
82         if (do_invert)
83             invert_join_and_read_output((uint32_t
84 *)\&OutBitMap1[i*WIDTH*N_ROWS]);
85         if (threshold > 0)
86             threshold_to_zero_join_and_read_output((uint32_t
87 *)\&OutBitMap2[i*WIDTH*N_ROWS]);
88     #endif
89 }
```

Figure 8-34 Main Execution Loop of main.non-blocking.cpp

`HAS_ACCELERATOR` is a SmartHLS defined macro that indicates whether the program is compiled with accelerators or not. The `*_write_input_and_start()` functions send the data to the hardware accelerator and start the accelerators without waiting for their completion. We check for completion on line 83 and 86, where the `*_join_and_read_output()` functions are called. This approach is like starting a thread and the waiting for the result at synchronization. A full list of available driver functions can be found under the `hls_output/accelerator_driver` directory as described in Section 6.2.6.

Although `invert()` and `threshold_to_zero()` can run independently of each other, they still share the same physical DMA in the MSS that can only access a single DDR memory channel. Thus, their execution time do not completely overlap with each other. We will explore an alternative in the next section.

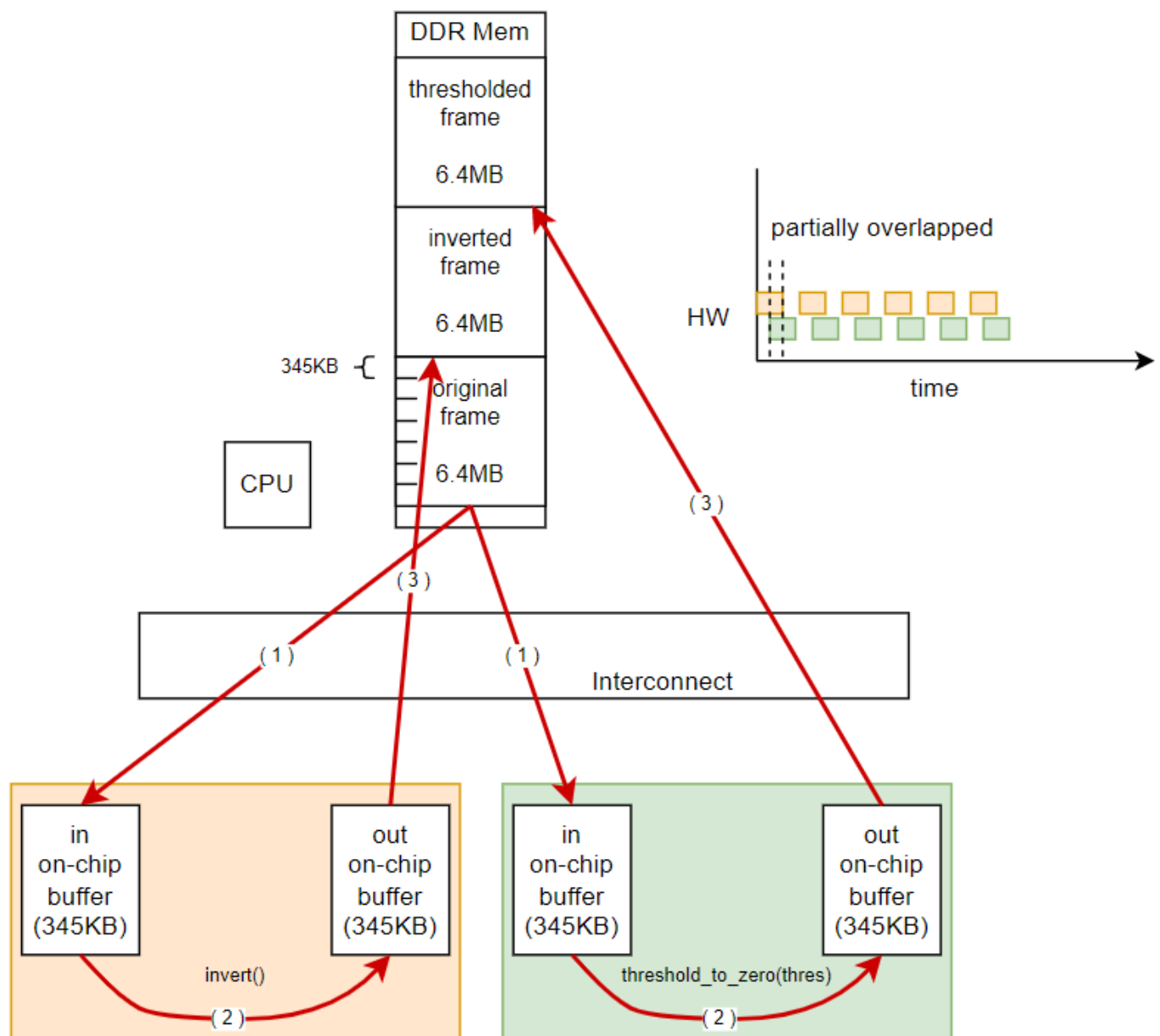


Figure 8-35 Data Movement of main.non-blocking.cpp



Modify the `Makefile.user` and select:

```
SRCS = main_variations/main.non-blocking.cpp
```

Then compile the software and run again. For Linux:

```
$ shls clean

$ ./compile_sw.shls.sh

$ ./run_sw.shls.sh
```

For Windows:

```
> shls.bat clean

> compile_sw.shls.bat

> run_sw.shls.bat
```

```
$ ./run_sw.shls.sh
-----
Run SW-only
Running the following targets: run_on_board
Waiting on board ready...
Board ready!
Copying hls_output/invert_and_threshold.no_accel.elf,toronto.bmp to root@192.168.8.48:./
Application starting (over ssh root@192.168.8.48)
Running: pushd ./; ./invert_and_threshold.no_accel.elf 1 200 > bin_cl_out.txt; cat bin_cl_out.txt; popd
Application output:
~ ~

Here we go!
N_ROWS:45, buf_size :259200, do_invert:1, threshold:200, mode:sw
Opened file toronto.bmp
function elapsed time: 0.260350 [s]
~
Application finished!
Copying bin_cl_out.txt output*.bmp from root@192.168.8.48:./ to hls_output/files/
-----
Run w/HW module
Running the following targets: run_on_board
Waiting on board ready...
Board ready!
Copying hls_output/invert_and_threshold.elf,toronto.bmp to root@192.168.8.48:./
Application starting (over ssh root@192.168.8.48)
Running: pushd ./; ./invert_and_threshold.elf 1 200 > bin_cl_out.txt; cat bin_cl_out.txt; popd
Application output:
~ ~

Here we go!
N_ROWS:45, buf_size :259200, do_invert:1, threshold:200, mode:hw
Opened file toronto.bmp
function elapsed time: 0.072135 [s]
~
Application finished!
Copying bin_cl_out.txt output*.bmp from root@192.168.8.48:./ to hls_output/files/
```

Figure 8-36 Runtime Results of main.non-blocking.cpp

Recall in Figure 8-29, performing a single `invert` or `threshold_to_zero` takes approximately 57 ms with accelerators as that is the approximate amount of time required to transfer the data from the DMA to the accelerator then back. Running `invert` and `threshold_to_zero` in parallel in this example did not

completely overlap the runtime of the two functions. They can only be partially overlapped because they share the same DMA.

8.8.4 Chaining using FIFOs

In the past sections we have only been changing the software, and have made no changes regarding the hardware. Now we will change the hardware and generate a new bitstream. Alternatively, you can use the `INVERT_AND_THRESHOLD_FIFO.job` precompiled bitstream in the `SmartHLS_Training4_Jobs` folder on Github.

In this example, we will refactor the code and merge the two functions, `invert()` and `threshold_to_zero()`, into a single top function called `invert_and_threshold_to_zero()`, which essentially calls the two functions internally as shown in Figure 8-37.

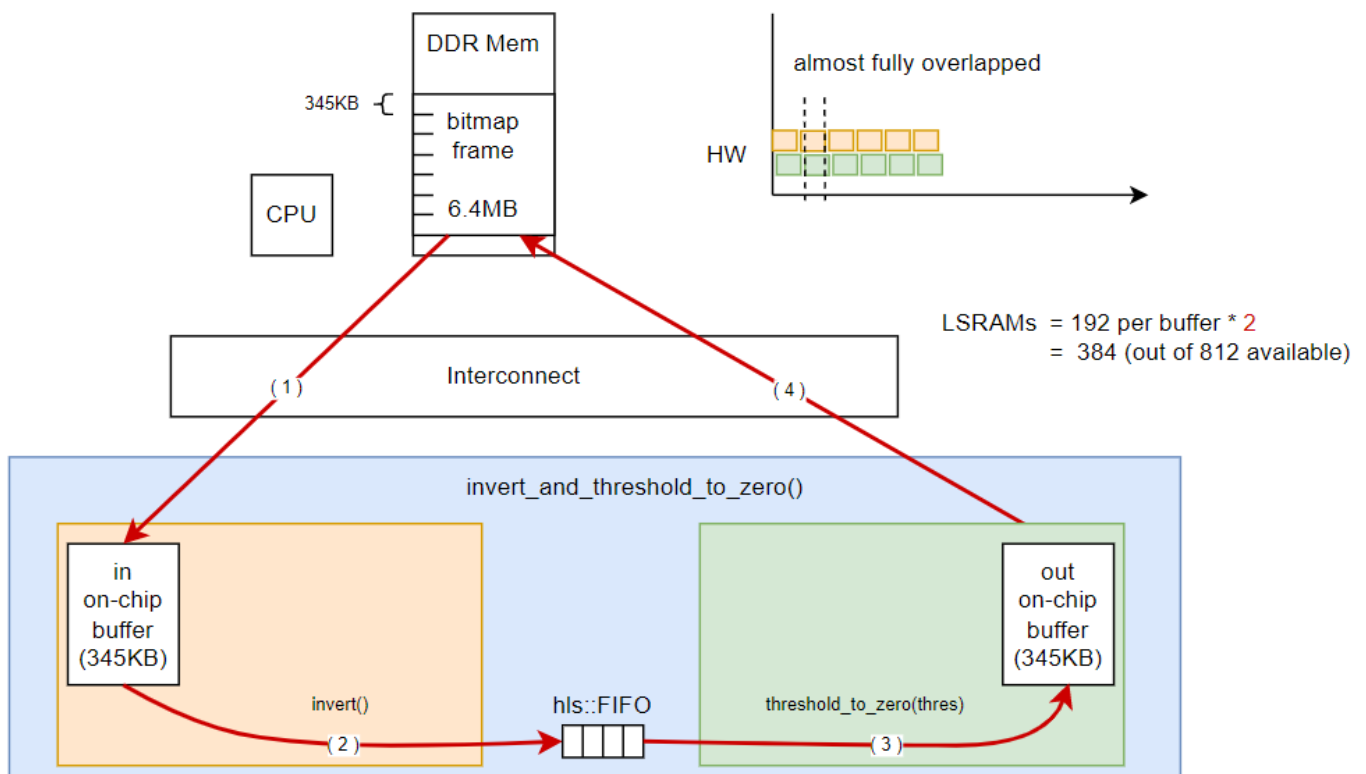


Figure 8-37 Data Movement of main.fifo.cpp

We consolidated `invert` and `threshold_to_zero` into a single top module, instead of two independent cores before. Normally, a user would have to disconnect and remove the previous two modules by hand using the GUI or via TCL commands to reconnect the new single hardware module. SmartHLS will take care of that integration now automatically. Figure 8-38 shows how this new accelerator is implemented in SmartHLS:

```

27 void invert_and_threshold_to_zero(uint32_t *in, uint32_t *out, int
do_invert, uint8_t thres) {
28     #pragma HLS function top
29     #pragma HLS interface default type(axi_target)
30     #pragma HLS interface argument(in) type(axi_target) dma(true)
num_elements(WIDTH*N_ROWS)
31     #pragma HLS interface argument(out) type(axi_target) dma(true)
num_elements(WIDTH*N_ROWS)
32

```

```

33     hls::FIFO<uint32_t> fifo(16);
34     hls::thread t1(invert, in, std::ref(fifo), do_invert);
35     hls::thread t2(threshold_to_zero, std::ref(fifo), out, thres);
36     t1.join();
37     t2.join();
38 }

```

Figure 8-38 Thread and FIFO in main.fifo.cpp

The operations performed in the `invert_and_threshold_to_zero()` function is a classic example of the producer-consumer pattern. The in data is received from the AXI target interface and passed to each stage of the computation, namely `invert` and `threshold_to_zero`. We use a thread (`hls::thread`) for each stage as each stage can be run independently as long as there are data available. The two stages are connected via a fifo between them.

By combining the 2 functions into one, we achieved the following:

- Doubled the performance and half the memory resources as we no longer need the output buffer of `invert()` and the input buffer of `threshold_to_zero()`.
- Reduced the runtime by half compared to `main.simple.cpp` because the execution of the two functions is now pipelined. The execution of the two operations almost fully overlapped except for initial DMA transfer and pipeline latency of first module.
- The amount of LSRAMs is reduced by half because we only need 2 DMA transfers instead of 4 compared to the simple configuration. The image data stays longer on the fabric increasing the amount of computation per data movement to/from the CPU.



Modify the `Makefile.user` and select:

```
SRCS = main_variations/main.fifo.cpp
```

Rerun the entire flow as described in Section 8.7: [Compiling the hardware](#) and Section 8.8: [Programming the FPGA bitstream](#) because this variation requires a hardware change.

Alternatively, you can use the `INVERT_AND_THRESHOLD_FIFO.job` precompiled bitstream in `SmartHLS_Training4_Jobs` folder on Github.

Then compile the software and run again. On Linux:

```

$ shls clean

$ ./compile_sw.shls.sh

$ ./run_sw.shls.sh

```

On Windows:

```
> shls.bat clean

> compile_sw.shls.bat

> run_sw.shls.bat
```

```
-----
Run w/HW module
Running the following targets: run_on_board
Waiting on board ready...
Board ready!
Copying hls_output/invert_and_threshold.elf,toronto.bmp to root@10.245.244.39:./
Application starting (over ssh root@10.245.244.39)
Running: pushd ./; ./invert_and_threshold.elf 1 200 > bin_cl_out.txt; cat bin_cl_out.txt; popd
Application output:
~ ~

Here we go!
N_ROWS:45, buf_size :259200, do_invert:1, threshold:200, mode:hw
Elapsed time: 0.057402 [s]
~
Application finished!
Copying bin_cl_out.txt output*.bmp from root@10.245.244.39:./ to hls_output/files/
```

Figure 8-39 main.fifo.cpp Runtime with Hardware Acceleration

As shown in Figure 8-39, the runtime is now ~57 ms for both hardware modules, which is the same runtime as running only one of the accelerators and almost half the runtime of running both accelerators (~112ms) in `main.simple.cpp` variation. Despite an increase in computation in the accelerator, we do not see any difference in runtime between a simple invert and the combined `invert` and `threshold_to_zero` function. The runtime is still dominated by DMA transfers. Thus, we can expect more saving in runtime as we increase the complexity of the accelerator function.

8.8.5 Summary

The runtime of the various implementations that we have explored in Section 8.8 are summarized below in Figure 8-40.

Main Variation	Arguments	Without Accelerators	With Accelerators
main.simple.cpp	do_invert:0 threshold:0	0 ms	0 ms
	do_invert:0 threshold:200	141 ms	57 ms
	do_invert:1 threshold:0	46 ms	57 ms
	do_invert:1 threshold:200	173 ms	112 ms
main.non-blocking.cpp	do_invert:1 threshold:200	260 ms	72 ms

Main Variation	Arguments	Without Accelerators	With Accelerators
main.fifo.cpp	do_invert:0 threshold:0	4.6 s to 5.0 s	57 ms
	do_invert:0 threshold:200	4.6 s to 5.0 s	57 ms
	do_invert:1 threshold:0	4.6 s to 5.0 s	57 ms
	do_invert:1 threshold:200	4.6 s to 5.0 s	57 ms

Figure 8-40 Runtime of Various Implementations

Several things are of note here:

1. **DMA transfers dominate the overall runtime when running with accelerators.** When `main.fifo.cpp` consolidated invert and `threshold_to_zero` into a single accelerator, the runtime is effectively halved (57 ms) compared to `main.simple.cpp`'s runtime of performing both transformations (112 ms). Regardless of the complexity of invert, `threshold_to_zero`, and the combined function, the runtime is about 55 ms for each function called.
2. **The DMA is shared and can become the bottleneck when multiple accelerators are accessing at the same time.** `main.non-blocking.cpp` produces an inverted image and a `threshold_to_zero` image in parallel. However, the execution of invert and `threshold_to_zero` functions can only be partially overlapped due to the DMA being shared amongst them. Hence, the runtime is longer (72 ms) than running only one of the transformations (57 ms), but shorter than `main.simple.cpp`'s runtime of performing both transformations (112 ms).
3. **Saving could be accomplished even for relatively simple functions despite the cost of DMA transfer.** In `main.simple.cpp`, the accelerator version (57 ms) almost breaks even with the simple invert software function (46 ms). Running the accelerator version of `threshold_to_zero` (57 ms) took less than 40% of the pure software runtime (141 ms).
4. **Threads are expensive in software but cheap in hardware.** `main.fifo.cpp` uses `hls::thread` to implement the producer-consumer behaviour. Creating and destroying threads for very simple calculations is costly. Even though the runtime for running with accelerators improved, runtime for pure software on the MSS has increased significantly.
5. **Software can be used to save computations.** `main.simple.cpp` performs a check on the argument and does not send the data to the accelerator if calculations were not required, i.e., the argument is zero. On the other hand, `main.fifo.cpp` blindly sends the data to the accelerator to compute. Hence, `main.fifo.cpp` still takes 55 microseconds to complete even when the arguments are zero, but `main.simple.cpp` saved time (0 ms) by not doing the unnecessary calculations.

We have shown how to integrate SmartHLS generated accelerators into your own SoC through the AXI interface, how to use non-blocking driver functions to parallelize computation, how the DMA affects runtime, and how the DMA can be the bottleneck in your system. We hope you take what we have shown here and incorporate SmartHLS into your own SoC designs.

9. Current limitations of the SoC flow

In the first release of the SmartHLS PolarFire SoC flow there are a few limitations:

1. No AXI streaming arguments
2. No AXI initiator arguments with burst support
3. No arbitrary bit-width types (ap_[u]int) are supported for function arguments
4. No variable length transfers

Even though the amount of memory was reduced by half in the main.fifo variation, there is still the possibility to eliminate the on-chip buffers all together. However, to achieve this we need to address the first two limitations.

The AXI streaming arguments can be used to send data to hardware modules that can consume incoming data at line rate and eliminate the need for the on-chip buffer. That means the hardware modules should not backpressure the CPU interconnect, which is achieved when the modules are fully pipelined with an initiation interval (II) of 1.

The AXI initiator with burst support optimization would also allow removing the outgoing on-chip buffer because the hardware module would be able to directly write into the CPU memory at line rate without the CPU having to initiate a DMA transfer.

The image below (Figure 9-1) shows what a full streaming configuration may look like. In this case, since there is only one DDR bank, the performance would be limited by the memory controller and interconnect.

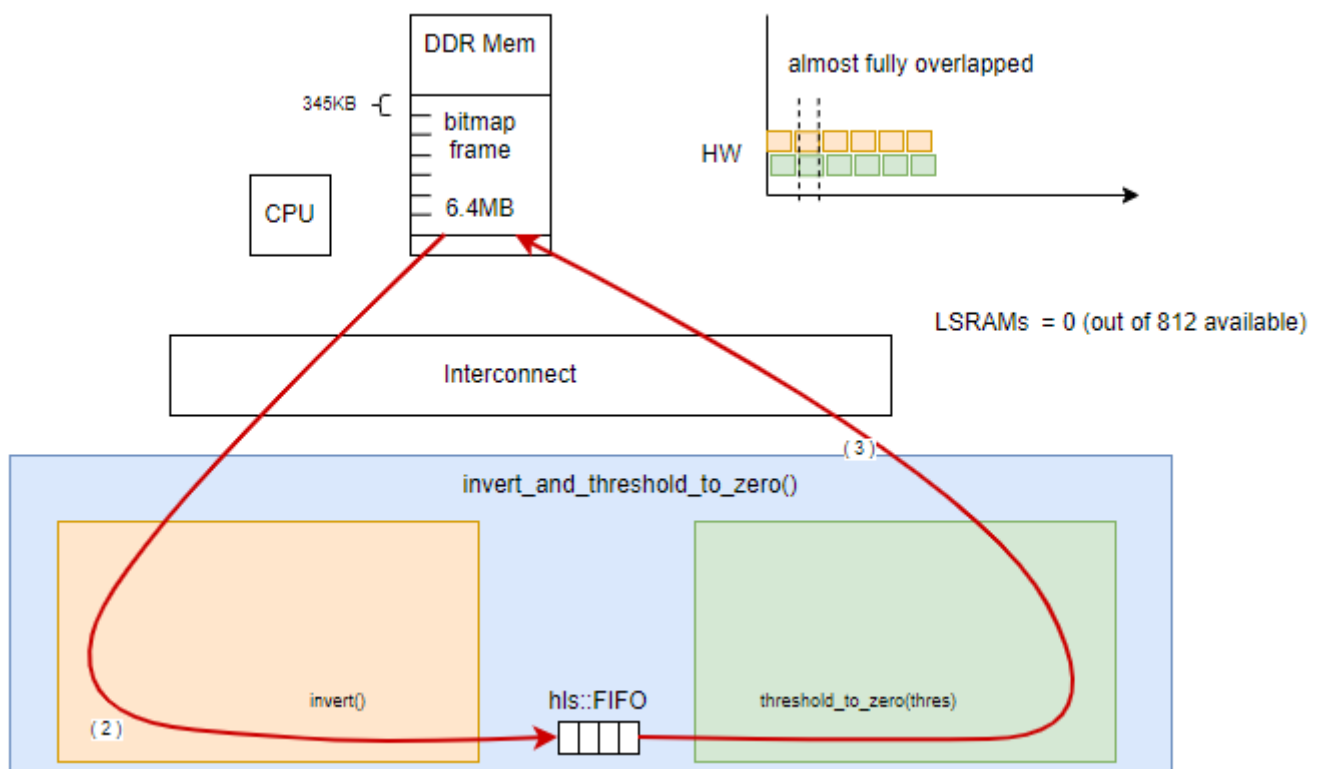


Figure 9-1 Fully Streaming Configuration

Also, SmartHLS does not currently support using arbitrary bit-width types as function arguments like this:

```
foo(hls::ap_int<24> &in)
```


The function would have to be rewritten (padded) like this:

```
foo(uint32_t &in)
```

For this reason, in Section 8.4, the 24-bit pixel format (3-channels RGB, 8-bits per channel) had to be padded with the extra 8-bit alpha channel even though the original .bmp image does not contain the alpha channel. The alpha channel is ignored when reading and writing back to the .bmp files.

Finally, the amount of data being transferred is determined by SmartHLS at compile-time via `num_elements(WIDTH*N_ROWS)` pragma option as shown in Figure 9-2. For example, if we wanted to work with two different image frame sizes HD (1280x720) and FULL-HD (1920x1080) on the `invert()` function, we would have to use the largest size (Full-HD in this case) for the value of `num_elements`, and add a function argument indicating the actual size to use. This, however, would only limit the amount of data that is processed but not the amount of data that is transferred during the DMA transactions.

```
04 void invert(uint32_t *in, uint32_t *out) {
05     #pragma HLS function top
06     #pragma HLS interface default type(axi_target)
07     #pragma HLS interface argument(in) type(axi_target) dma(true)
    num_elements(WIDTH*N_ROWS)
08     #pragma HLS interface argument(out) type(axi_target) dma(true)
    num_elements(WIDTH*N_ROWS)
09
10     #pragma HLS loop pipeline II(1)
11     for (int j = 0; j < WIDTH*N_ROWS; j++) {
12         out[j] = ~in[j];
13     }
14 }
```

Figure 9-2 Compile-Time Determination of the Number of Elements to be Processed