SmartHLS™ Training Session 1:

Image Processing on the PolarFire® Video Kit

Training
Revision 8
Jan 30, 2024

Table of Contents

1   Revision History

The revision history describes the changes that were made to this document listed by revision.

1.1   Revision 1

First publication of the document.

1.2   Revision 2

Updated document for LegUp HLS 9.2 release.
Added new download links to updated training design files.

1.3   Revision 3

Updated document for SmartHLS™ 2021.2 release.

1.4   Revision 4

Updated document for outdated figures and for SmartHLS™ 2022.2 release.

1.5   Revision 5

Updated document for outdated figures, references and experimental results.

1.6   Revision 6

Updated document for outdated figures and for SmartHLS™ 2022.3 release.

1.7   Revision 7

Updated document for SmartHLS™ 2023.2 release. Updated sections 12.2 and 13 to use loop pipelining instead of function pipelining.

1.8   Revision 8

Updated document for outdated figures and for SmartHLS™ 2024.1 release.

2   Prerequisites

Before beginning this training, you should install the following software:
- Libero® SoC 2024.1 (or later) with ModelSim Pro
  - [Download](#)
- SmartHLS 2024.1 (or later): this is packaged with Libero
- DG0849 Video Control GUI used by the PolarFire board demo
  - [Download Link](#)

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.

You should download the training design files in advance:
- Github link to all SmartHLS trainings and examples:
  https://github.com/MicrochipTech/fpga-hls-examples
  - ZIP file: https://github.com/MicrochipTech/fpga-hls-examples/archive/refs/heads/main.zip
  - We'll use the Training1 folder for this training.
- Generate the Libero project from tcl by following the instructions here:
  https://github.com/MicrochipTech/fpga-hls-examples/tree/main/Training1/Libero
  There is also a sample .job file provided for reference in
  Training1/reference/VIDEO_KIT_TOP.job.

The following hardware is required:
- PolarFire FPGA Video and Imaging Kit ([MPF300-VIDEO-KIT](#)).
- Monitor with an HDMI input.

Make sure the following demo is working on your board: [DG0849: PolarFire FPGA Dual Camera Video Kit Demo Guide](#).

We assume you have already completed the [SmartHLS Tutorial: Sobel Filtering for Image Edge Detection](#).

We assume some knowledge of the C/C++ programming language for this training.

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

3   Overview

Time Required: 3 hours

Goals of this Training:
- Deeper dive into commonly used features of SmartHLS
- Demonstrate a SmartHLS design running on the PolarFire® board

Training Topics:
- Overview of the SmartHLS tool and design flow
- What hardware blocks to design in C++ with SmartHLS vs. RTL?
- Overview of the PolarFire board and video kit demo
- Walkthrough of image processing hardware blocks designed in C++ with SmartHLS
    - Alpha Blending
    - Color Space Conversion: RGB2YCbCr & YCbCr2RGB
    - Gaussian blur
    - Canny edge detection
- Deeper dive into SmartHLS:
    - Overview of HLS pipelining
        - What is the initiation interval?
        - What impacts the initiation interval?
    - Verification and Testing:
        - Writing a C++ testbench
        - How does co-simulation work?
        - Showing ModelSim waveforms during co-simulation
    - External top-level hardware interface
        - AXI-Stream interface (data/valid/ready)
        - Input wires (from switches)
        - RAM interface
- Deeper dive into HLS optimizations:
    - Function pipelining, loop pipelining, FIFOs for streaming
    - Canny has 4 filters streamed together using data flow
- SmartHLS C++ Library and Data Types:
    - Arbitrary precision integers (ap_int/ap_uint)
    - Fixed-point data types (ap_fixpt/ap_ufixpt)
    - FIFO
    - LineBuffer
- Export hardware blocks from SmartHLS as SmartDesign IP component
    - Integration of SmartHLS SmartDesign IP component into PolarFire Design
    - Running SmartHLS hardware on the PolarFire board

4   SmartHLS High-Level Synthesis Overview

The main reason why FPGA engineers use high-level synthesis software is to increase their productivity. Designing hardware using C++ offers a higher level of abstraction than RTL design. Higher abstraction means less code to write, less bugs and better maintainability.

In the SmartHLS high-level synthesis design flow, the engineer implements their design in C++ software and verifies the functionality with software tests. Next, they specify a top-level C++ function, which SmartHLS will compile into an equivalent Verilog hardware module. SmartHLS can run co-simulation to verify the hardware module behavior matches the software. SmartHLS uses Libero® SoC to generate the post-layout timing and resource reports for the Verilog module. Finally, SmartHLS generates a SmartDesign IP component that the engineer can instantiate into their SmartDesign system in Libero SoC. Figure 1 shows the SmartHLS high-level synthesis FPGA design flow for targeting a Microchip PolarFire FPGA.

Figure 1: High-level Synthesis FPGA Design Flow Targeting a PolarFire FPGA

When you open SmartHLS, you should find a toolbar, as shown in Figure 2, which you can use to execute the main features of the SmartHLS tool. Hover over each icon in SmartHLS to find out their meanings.

| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 |

Figure 2: SmartHLS toolbar icons

Starting from the left of Figure 3, the icons are:

1) **Add Files to Project**

Then icons for the software development flow:

2) **Compile Software with GCC**
3) **Run Software that was compiled**
4) **Debug Software with gdb**
5) **Profile Software with gprof**

The hardware development flow icons are:

6) **Compile Software to Hardware (Software to HDL)**
7) **Simulate Hardware in ModelSim with custom testbench**
8) **Software/Hardware Co-simulation**
9) **Synthesize Hardware to FPGA (HDL to hardware layout) – RTL Synthesis only for resource results**
10) **Synthesize Hardware to FPGA – RTL Synthesis, place and route for timing and resource results**
11) **Compile Software to Processor/Accelerator SoC**

With the last three icons, you can:

12) **Set HLS Constraints**
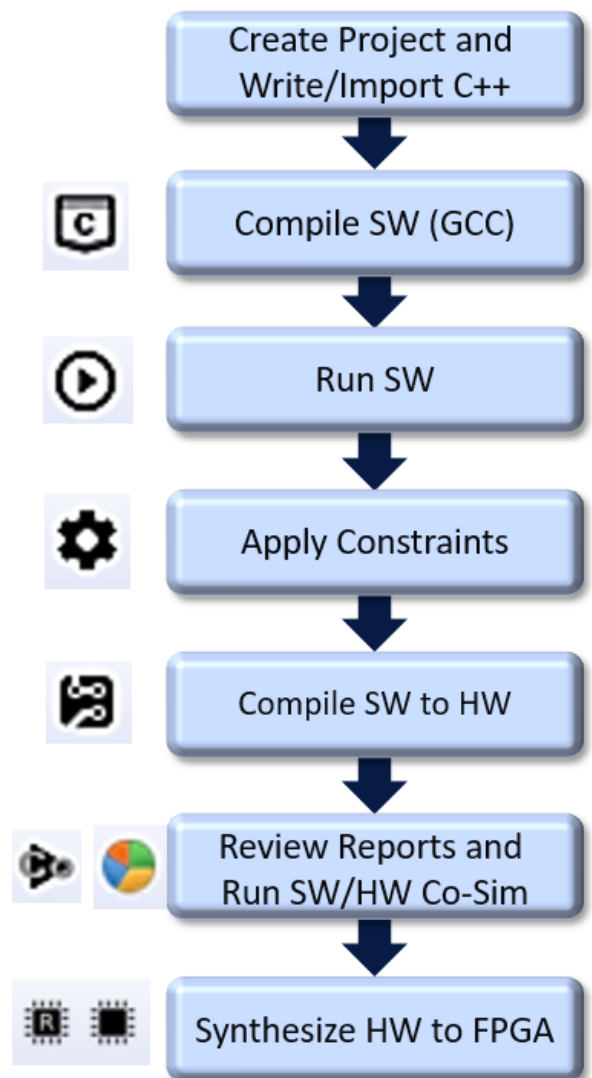13) **Launch Schedule Viewer**
14) **Clean SmartHLS Project**

These SmartHLS commands can also be run from the *SmartHLS* top bar menu. Figure 3 summarizes the SmartHLS design flow steps. We create the SmartHLS

Figure 3: SmartHLS Design Flow Steps

project and follow a standard software development flow using C++ (compile/run/debug). Then we apply HLS constraints (i.e., target clock period) and compile the software into Verilog using SmartHLS. We can review reports about the generated hardware. Then we run software/hardware co-simulation to verify the generated hardware. Finally, we can synthesize the hardware to our target FPGA to report the hardware resource usage and Fmax.

5   When to use High-Level Synthesis vs. RTL Design?

High-level Synthesis (HLS) allows you to use C++ software to describe FPGA hardware. HLS offers you much better design productivity because C++ is at a much higher level of abstraction than RTL languages like Verilog/VHDL.

High-level synthesis works very well for designers describing data-flow applications like digital signal processing or video/image processing where the mathematical algorithm can be described in C++. But for certain control heavy applications, such as a bus controller, a designer will have trouble describing the cycle-accurate behavior of the hardware in C++. For example, an AHB-Lite bus slave controller that must provide an error response after exactly 2 clock cycles. There is no way to specify a precise 2-cycle delay in C++, and RTL should be used in these and other control path cases.

C++ is better at describing the hardware at the algorithmic level. The HLS compiler can automatically add the appropriate pipelining registers to meet the specified clock period constraint. If you develop DSP applications (filters, video processing, etc.) where you start from a C++ reference implementation and manually convert to RTL, HLS will save you a lot of time. If your design is mainly control path and shuffling a few bits around, then use RTL.

Good fit for SmartHLS	Bad fit for HLS (use RTL instead)
Image processing filters (edge detect, blur, noise cancellation)	Bus controller. Reason: needs precise cycle-accurate behavior
DSP application (Viterbi Decoder)	FFT. Reason: well-known optimized hardware butterfly structure
H.264 video encoder/decoder	Push button toggle counter used in this training. Reason: debouncing logic
TCP/IP network stack	
Applications with existing C/C++ implementation (i.e., motor controller)	

HLS also has the advantage of being able to easily specify AXI interfaces in C++. For example, an AXI-slave interface to receive commands from a processor or an AXI-master interface to read/write to DDR memory. Implementing these AXI master/slave interfaces manually in RTL can be tedious and error prone. We will cover how to specify these AXI interfaces in SmartHLS in another training.

6   Programming and Running Design on the PolarFire® Kit

In this training, we target the PolarFire FPGA Video and Imaging Kit (MPF300-VIDEO-KIT). The peripherals of the board are shown in Figure 4. We will use the Dual Camera Sensor inputs, the HDMI 1.4 TX (J2) to display output on a computer monitor, and the USB-UART (J12) for bitstream programming and communication with a Video Control GUI running on the PC. For user input, we will also use the two red push buttons and 4 switches on the board.
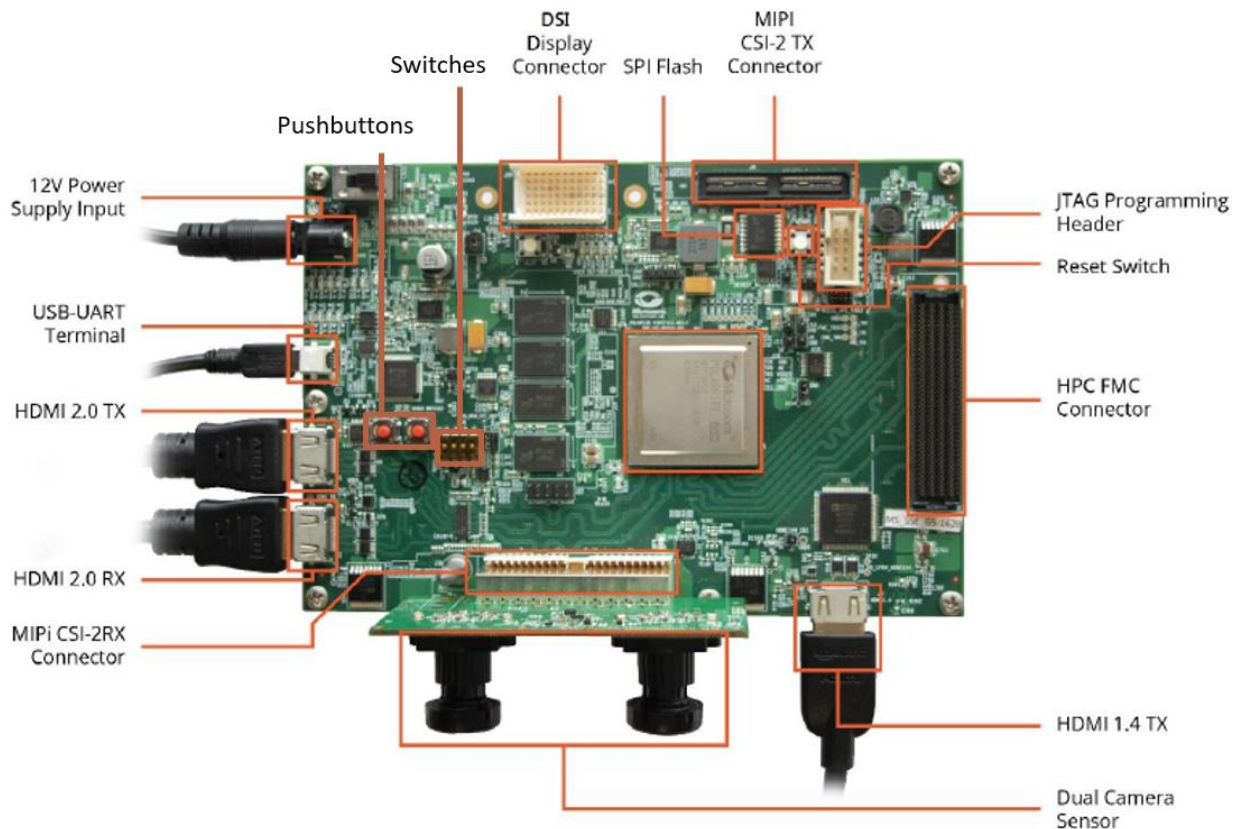

Figure 4: PolarFire Video and Imaging Kit Peripherals

During this training, we will build on top of the demo design that ships with the PolarFire Video Kit. We will use SmartHLS to design hardware blocks in C++ and generate SmartDesign IP components that we will instantiate into this design.

👆 We start by programming the final design with SmartHLS generated IP components on to the PolarFire board by following the steps below:

1. If you have not already, generate the Libero project from tcl by following the instructions here: https://github.com/MicrochipTech/fpga-hls-examples/tree/main/Training1/Libero.

2. Connect the USB cable from J12 on the PolarFire® board to your PC.

3. Connect the camera board at J5 and remove the lens caps.

4. Connect the HDMI cable from the PolarFire Video Kit (J2) to your external Monitor.

5. Refer to DG0849 for jumper settings. We use the default jumper settings shipped with the board.

6. Make sure all the DIP switches (SW6) are in the ON position.

7. Connect the AC adapter to the board and power it on (SW4).

8. Open up FlashPro Express (FPExpress v2024.1), which you can find in the Start Menu, listed under "Microchip Libero SoC v2024.1":
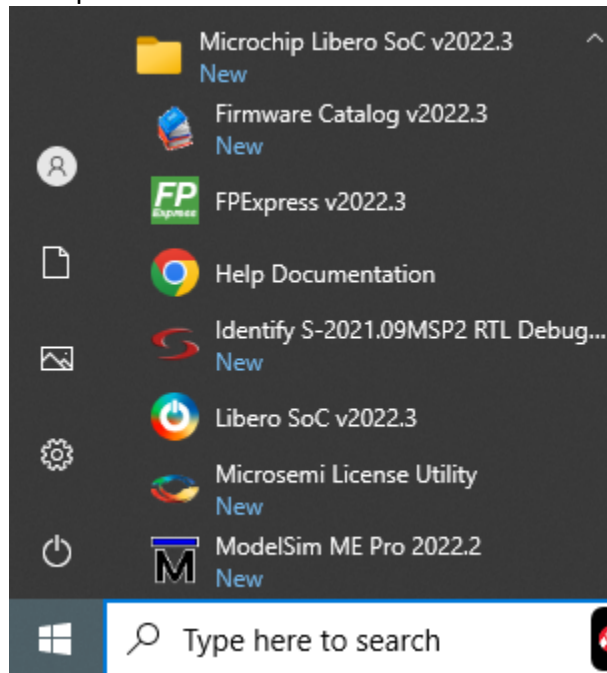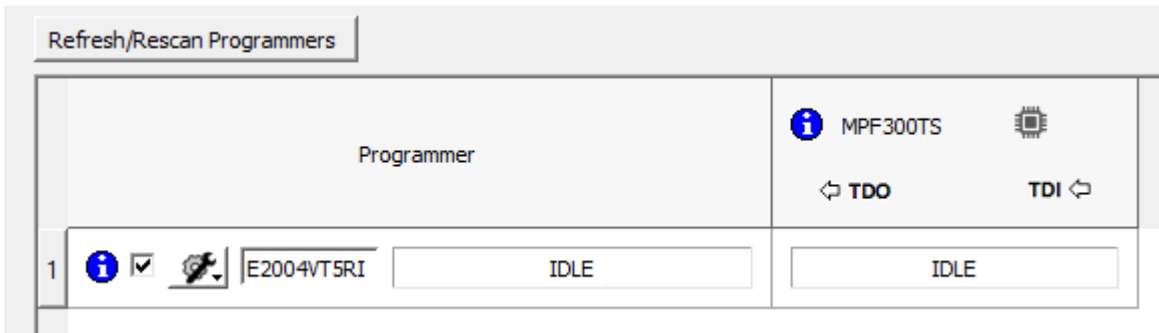

9. Select Project and New Job Project.
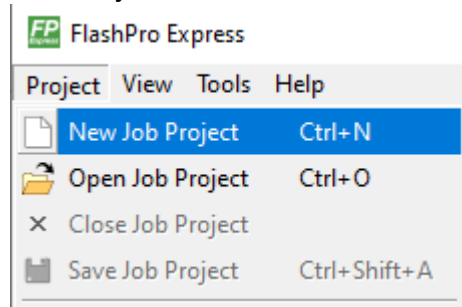
10. Now select the job file "Training1/Libero/VIDEO_KIT_TOP.job" in the folder you
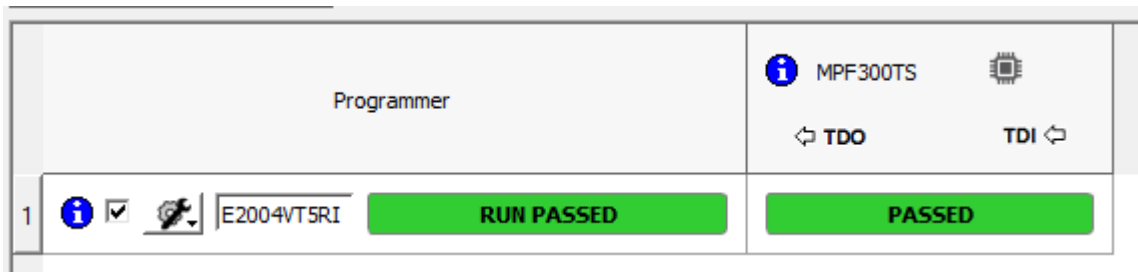
generated in step 1.

11. Enter a project location. Click OK.

12. Now the Programmer window will open. If you do not see the Programmer for the MPF300TS PolarFire® FPGA, then click Refresh/Rescan Programmers.

13. Now click the RUN button to program the FPGA.

14. After programming you should see the RUN PASSED. Now power cycle the board and close FlashPro Express.


15. Now you should see two video streams on your monitor, one in the background and then a smaller one moving around in the foreground. If the video streams look blurry, try focusing the camera by rotating the camera lens.
For example, if you hold the quick start card that comes with the PolarFire® board up to the camera:

PolarFire Video and Imaging Kit Quickstart Card

Kit Contents MPF300-VIDEO-KIT

| Quantity	Description
1	PolarFire FPGA with 300K LE MPF300TS-1FCG1152I Board
1	Dual Camera Sensor board (VIDEO-DC-DUALCAM)
1	HDMI cable
1	USB 2.0 A to Mini-B cable
1	12 V, 5 A AC power adapter and cord
1	1 Year Libero Gold Software License ($995 value)
1	Quickstart card

Then you should see the following output:
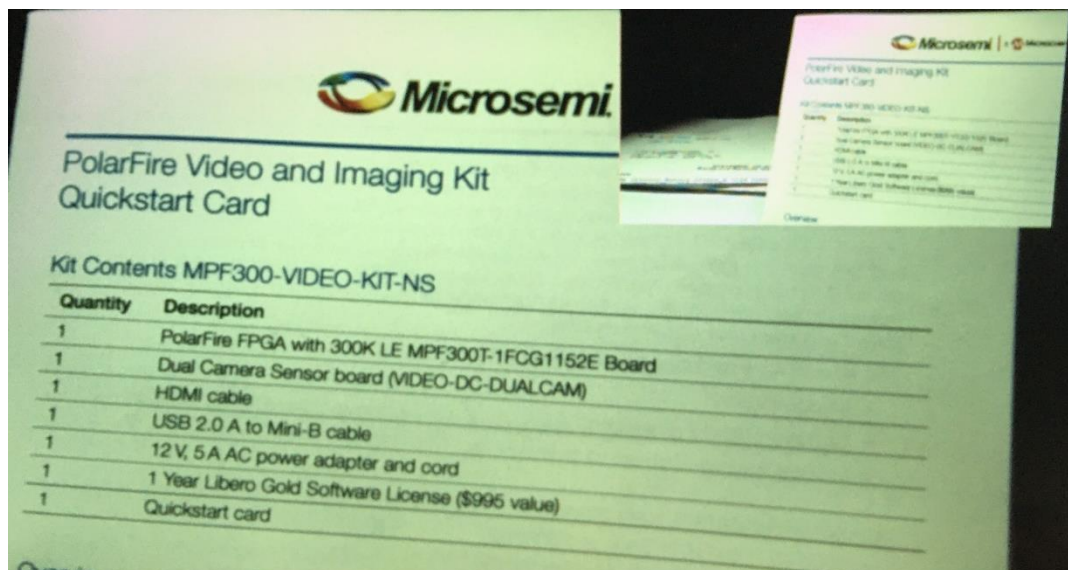
16. Launch the "Video Control GUI" from the Windows Start Menu (see prerequisites section if you do not have this program installed):

Best match

Video Control GUI
App
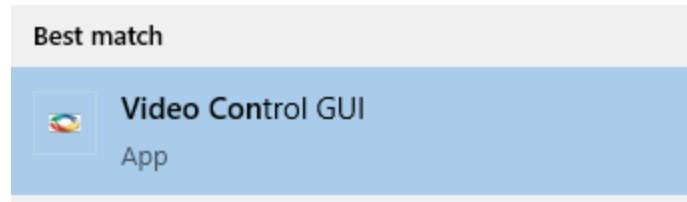
17. In the top right there is a dropdown to specify the COM port. Select the COM port (if there are multiple then choose the second highest numbered port):
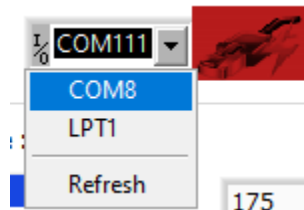
18. Now click the Red image beside the dropdown to connect to the FPGA.

19. The image should turn green to indicate the GUI is now connected to the FPGA and the smaller video feed should become fixed to the top left corner.
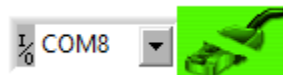
20. You can use the "Alpha" slider to test the SmartHLS generated alpha blend core. Changing the alpha affects the transparency of the smaller video feed.

21. Now select the "Edge" checkbox to enable the SmartHLS edge detection filters. The main video feed should turn to grayscale, which has a purple tint due to the default Color Balance settings.
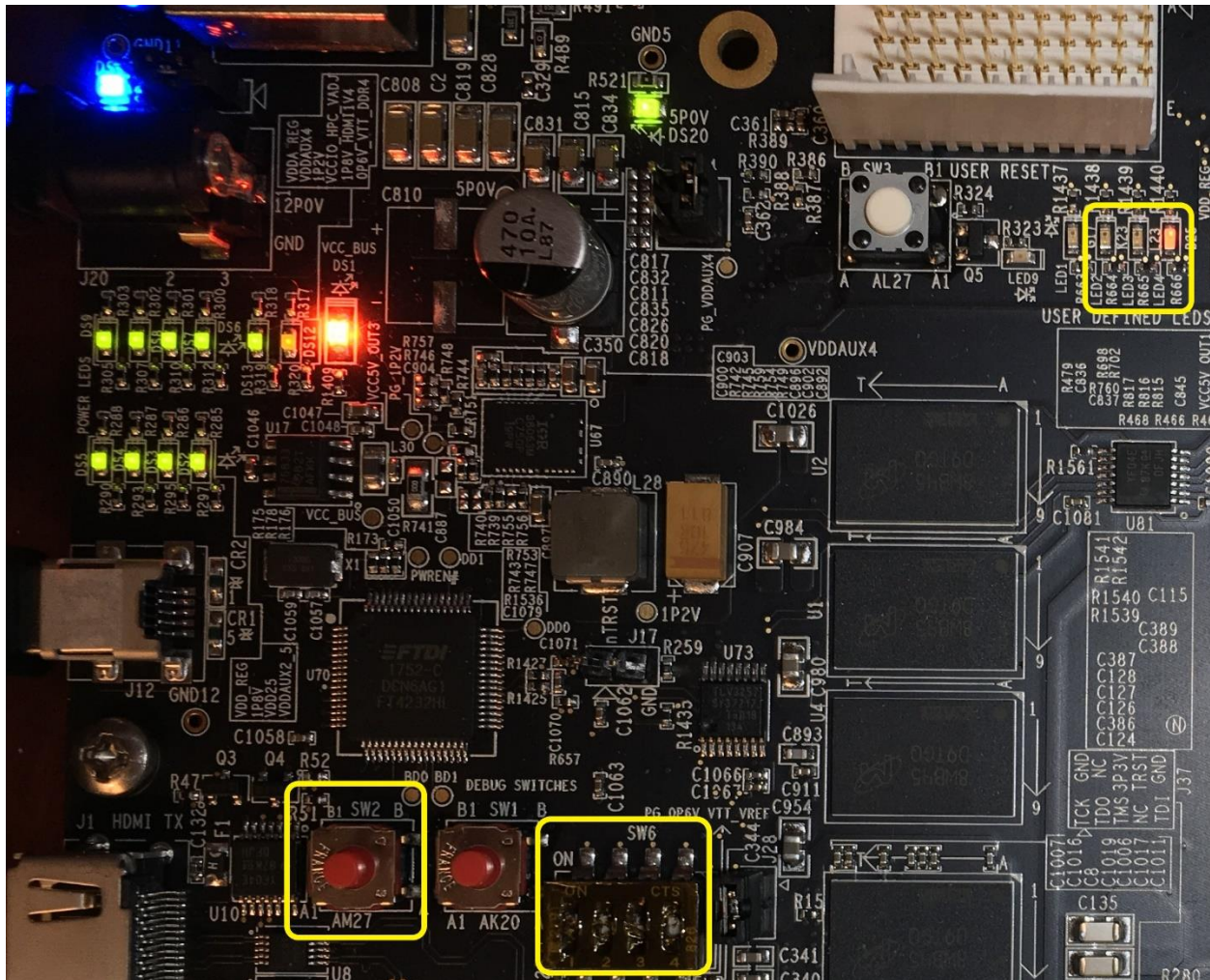
22. Click the push button (SW2) to toggle between 3 modes. The current mode will be displayed on the user defined LED2-4. LED1 should be flashing and shows that the Mi-V is communicating with the FPGA fabric.

> LED1 flashing: Mi-V is communicating with the FPGA.
> LED2 on: Grayscale image.
> LED3 on: Gaussian blur. Note: blurring effect is very subtle and only noticeable for sharp edges and details.
> LED4 on: Canny edge detection.

23. You can turn on/off each of the 4 filters in the Canny edge detection using the 4 switches (SW6). You will only see the effect of the switches when the Canny edge detection (LED4) or Gaussian blur (LED3) is on. The 4 switches maps to the 4 filters of Canny (LED4 on) and can be turned on and off individually. The first switch also turns on and off the Gaussian blur filter (LED3 on). Tip: use a pen to flip the switches, you may need to break the tape covering them first.

1) Gaussian blur
2) Sobel filter

17

3) Non-maximum suppression
4) Hysteresis

When you hold the same quick start card up to the camera, you should see the Canny Edge detection running on the monitor:
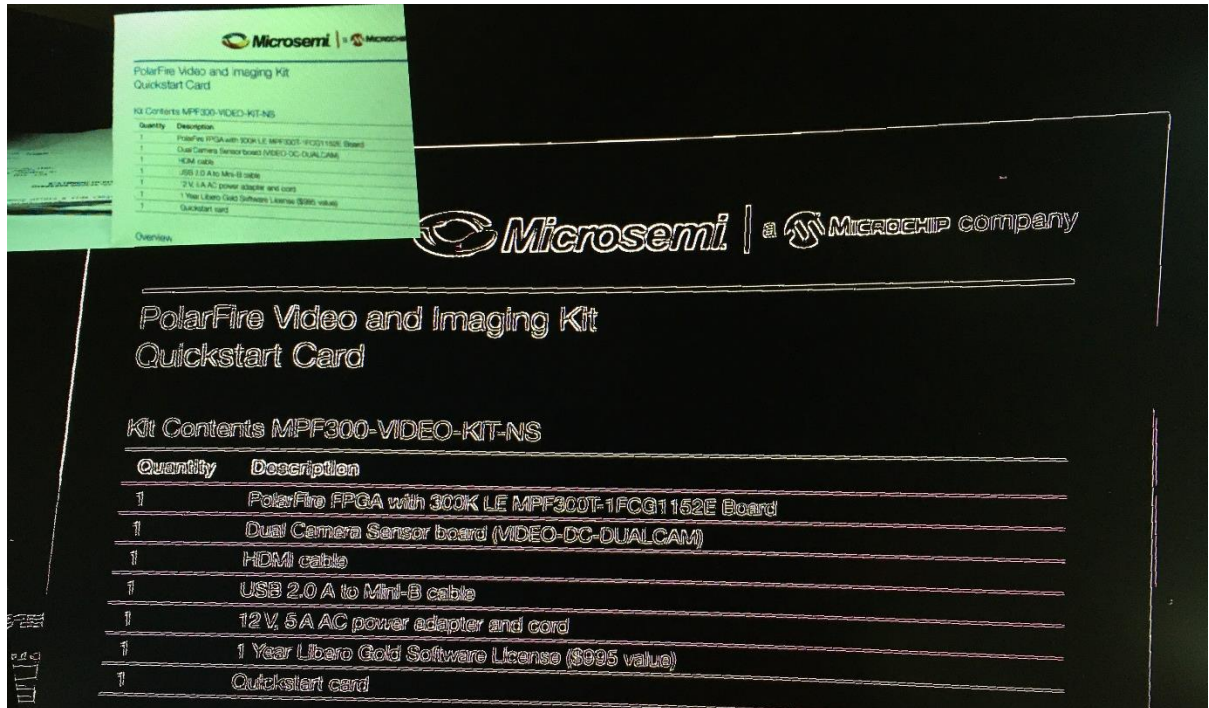
The design receives two 4K video inputs (3840x2160 30Hz resolution) from the dual Camera Sensors. On the monitor, the design shows a picture-in-picture (PIP) where the main display shows one camera output and the smaller inset image shows the other camera output. The camera source can be selected in the PIP Menu of the Video Control GUI as shown in Figure 5.
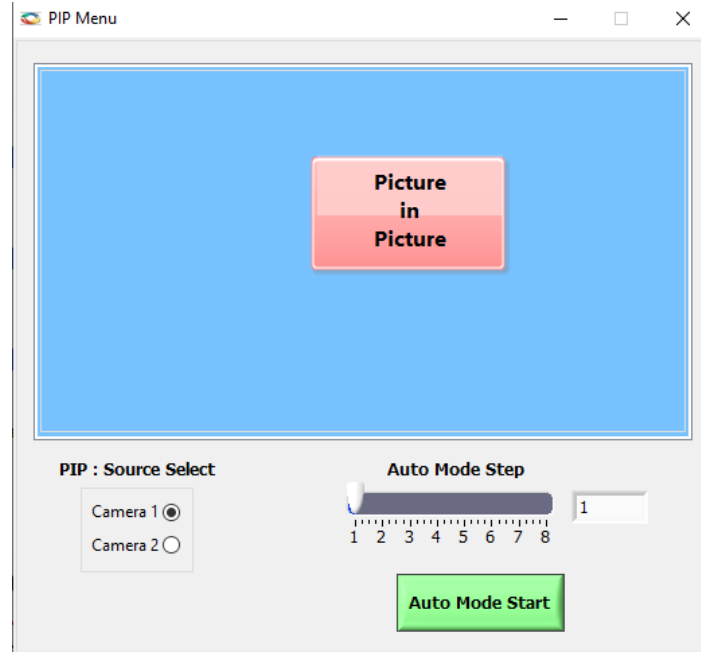
Figure 5: Picture-in-picture Menu

From the 4K video input, the design extracts a window of full HD output (1920x1080 60Hz resolution). You can use the Panning Menu in the Video Control GUI to change the location of this window as shown in Figure 6.
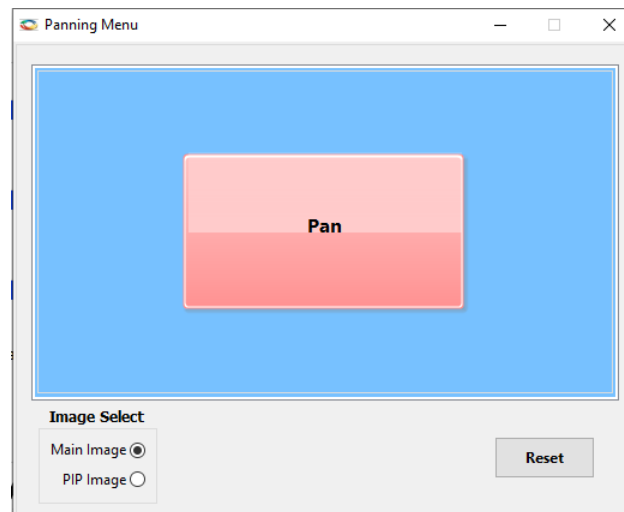
Figure 6: Panning Menu In Video Control GUI

24. Now close the Video Control GUI.

7 Design Architecture

We will give an overview of the design. For more details see AC469 Application Note PolarFire FPGA, which is a similar demo for the PolarFire® Eval Kit. Figure 7 shows the high-level hardware blocks in the design. The *Sensor Interface* block deserializes and decodes data from the dual camera sensors and then writes the 4K frames into DDR memory. Based on the location of the Panning, an HD video stream (1920x1080) is read out from DDR. This frame gets sent to the *Image/Video Processing* block to perform edge detection, alpha, brightness, contrast, and other filtering. The *Mi-V* soft processor receives configuration from the Video Control GUI running on the PC via the USB-UART. The Mi-V uses this configuration to control the Image/Video Processing block. The filtered video stream outputs from both cameras are combined in a picture-in-picture format. The *Display Controller* block sends the pixels and video control signals to the monitor via HDMI.
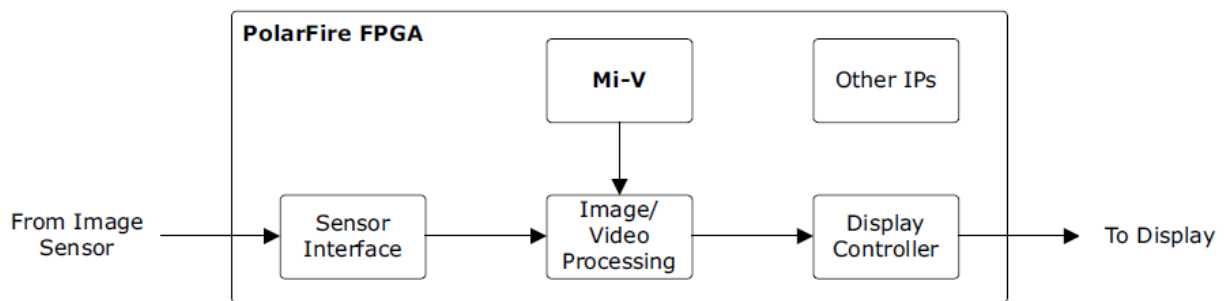
Figure 7: High-Level Design Architecture
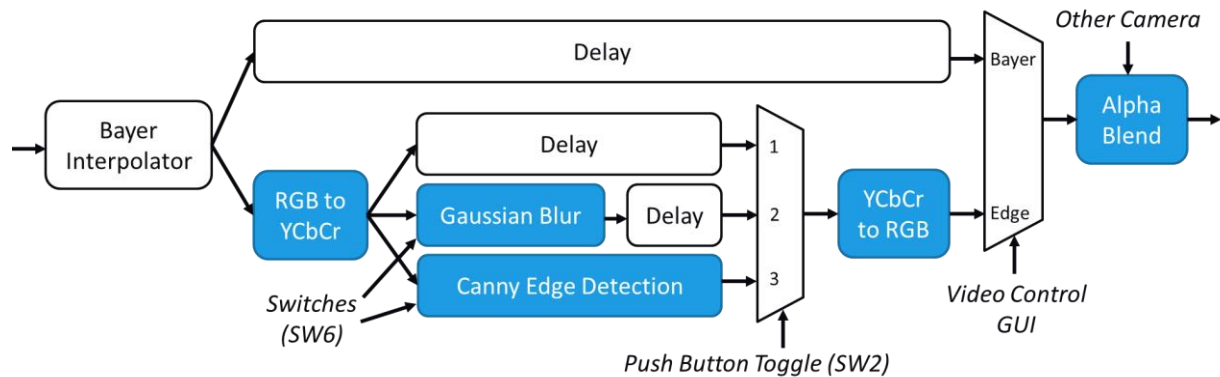
Figure 8: Image/Video Processing Block Diagram. SmartHLS cores in blue.

Figure 8 shows the *Image/Video Processing* block. The connections indicate pixel data passing between hardware blocks using a 1-bit valid signal to indicate the pixel is valid during a clock cycle. We start by converting the 8-bit raw camera inputs into 24-bit RGB format using a *Bayer Interpolator*.

If the Video Control GUI is in "Bayer" mode, then we skip any edge detection and pass the pixels into a *Delay* block to align the pixels with the other filters. The Delay block allows for variable delay by queuing pixels in a video FIFO until the FIFO read enable is triggered by the output valid of the core that we want to align the pixels to (with the longest latency).

If the Video Control GUI is in "Edge" mode, then we first use the *RGB2YCbCr* block to convert the 24-bit RGB pixels into 8-bit grayscale pixels. Then depending on the push button toggle state, we either 1) pass the grayscale pixels without filtering, or 2) perform a *Gaussian Blur*, or 3) perform *Canny Edge Detection*. User switches can also turn off/on the Gaussian blur and Canny edge detection individual filters. We take the 8-bit grayscale output pixels and convert them back to 24-bit RGB with the *YCbCr2RGB* block.

Finally, we pass the RGB pixels to the *Alpha Blend* core to blend this image frame with the other camera sensor input for the picture-in-picture effect. The alpha blend output goes to some image processing blocks (contrast, sharpening, color correction) before being displayed on the monitor.

The cores highlighted in blue are generated with SmartHLS and their design implementation will be covered in this training session.

8   Import the SmartHLS Projects into SmartHLS

We will start by importing all 9 SmartHLS projects used in this training into our SmartHLS workspace. Follow the directions below.

1. Download the zip file from github if you have not already (see Prerequisites). Extract the contents of the zip file. We will use the Training1 folder of the extracted content for this training.

2. Open SmartHLS 2024.1 and choose a workspace.
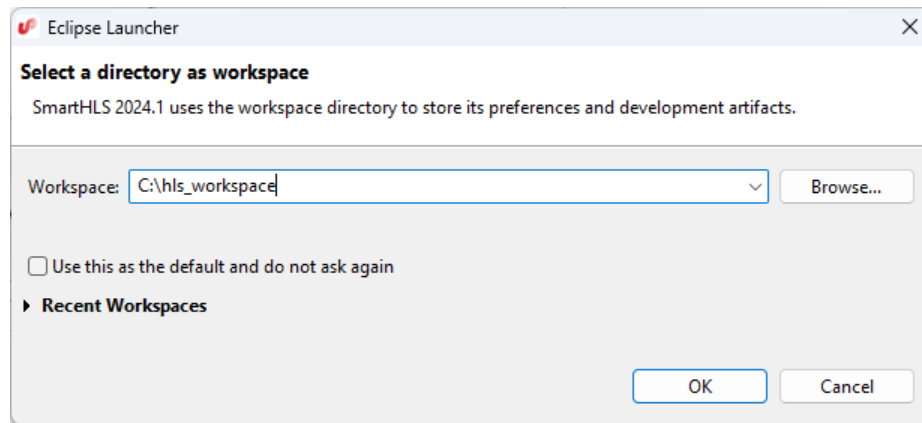
You may want to select a new folder so you can have a blank workspace for this training. **Warning:** Make sure there are no spaces in your workspace path. Otherwise, there will be an error when running synthesis (either one of) from SmartHLS.

3. Select File -> Import...
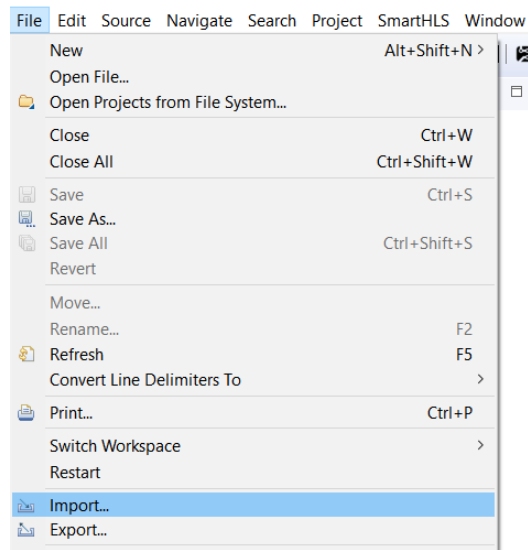
4. In the Import window, select General->Existing Projects into Workspace and then click Next.
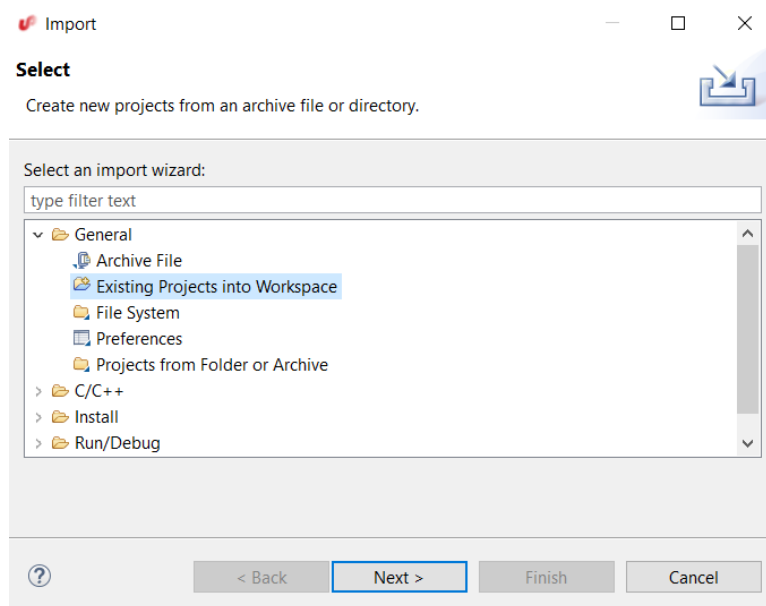
5. In the next step, check off "Copy projects into workspace" and then select "Select root directory" and then click Browse... In the popup window browse to the Training1 directory (after you extracted the zip file from github) and click OK.
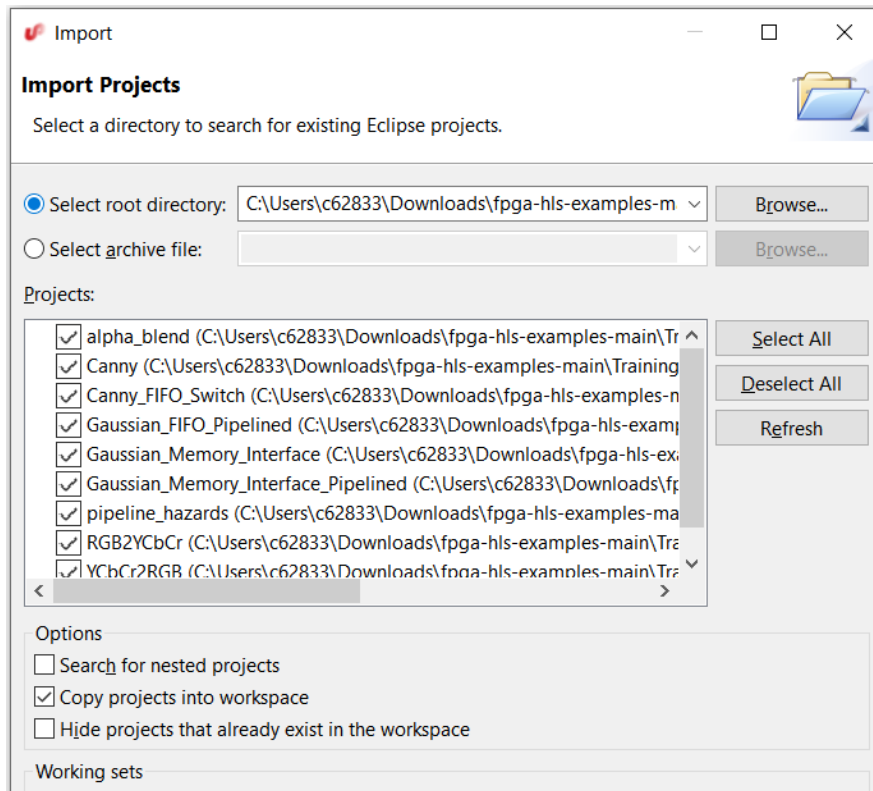
6. Now in the Projects box you should see that all 9 SmartHLS projects have been selected. Note: SmartHLS knows where the projects are by looking for Eclipse ".project" files in the subdirectories. Click Finish to import.
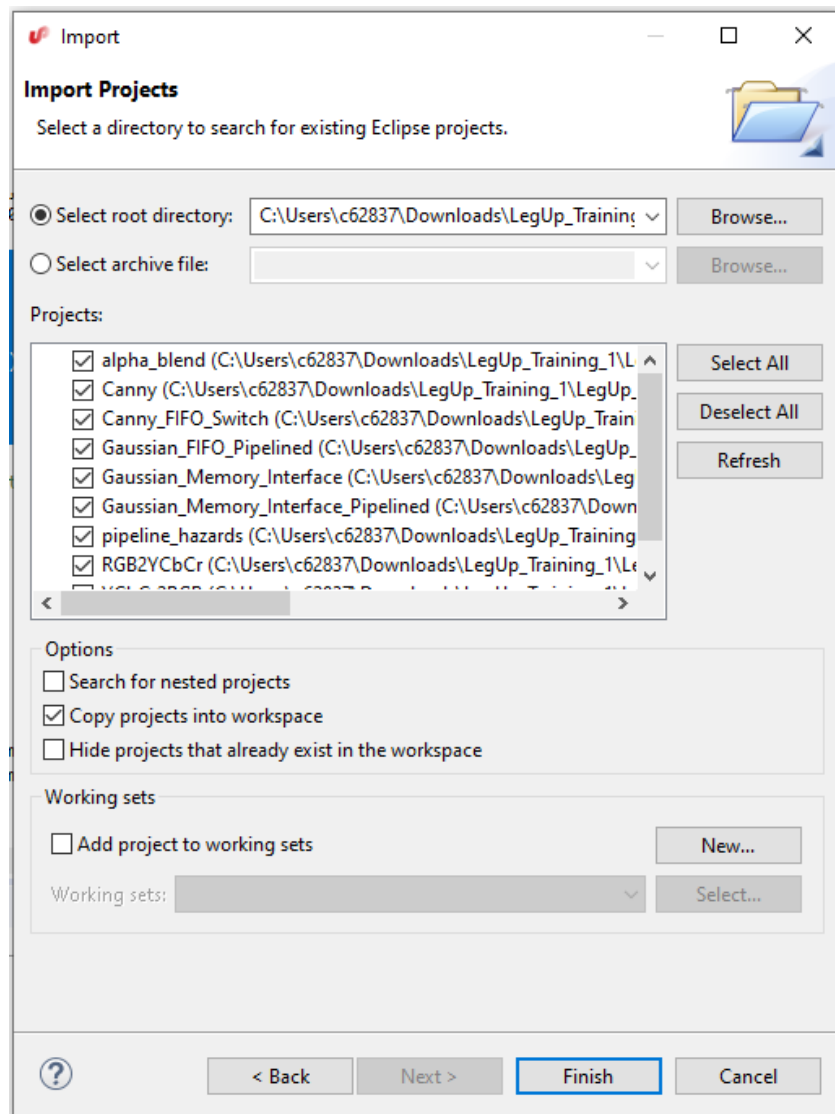
7. After importing you should see all 9 projects in the Project Explorer on the left.
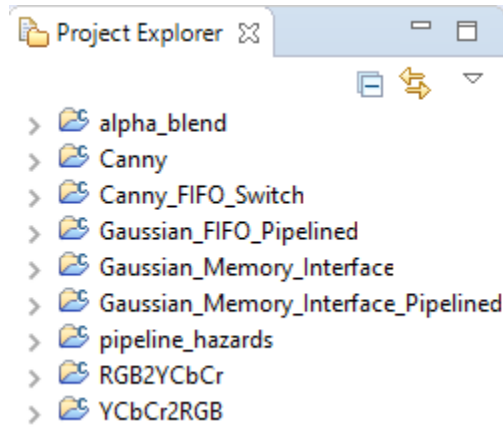
After importing projects into SmartHLS you may see red underlines on function calls with the message that the function could not be resolved, similar to what happens in Figure 9.
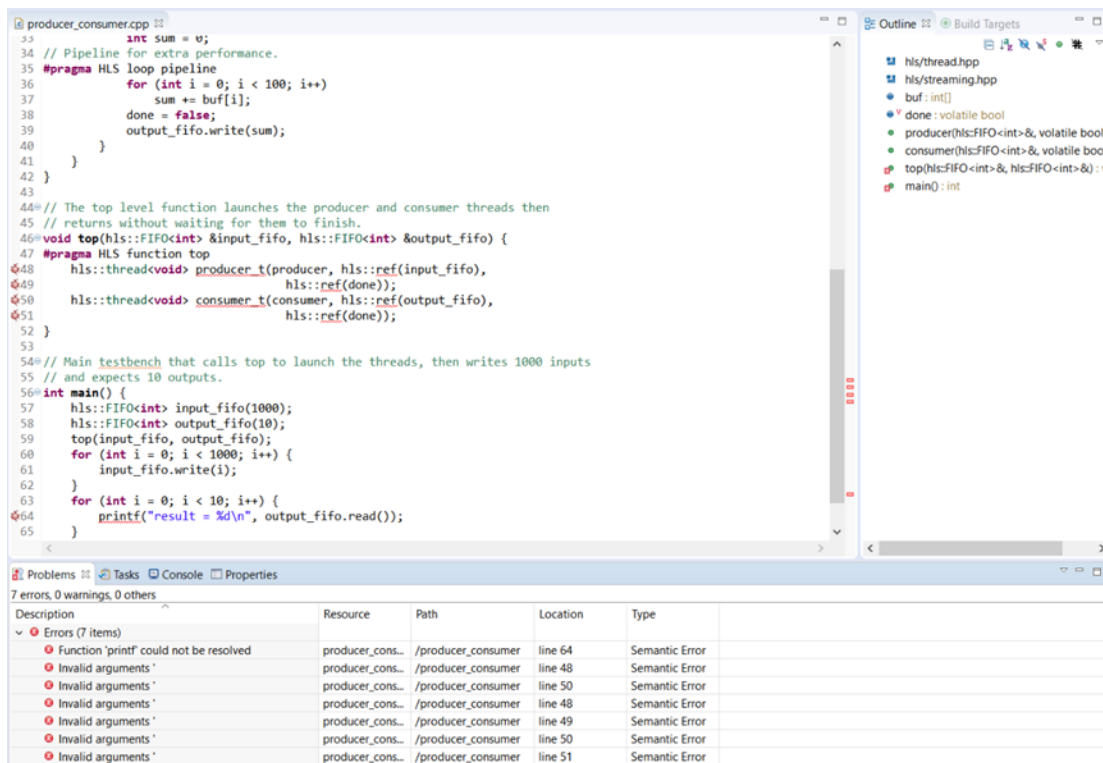
Figure 9: Eclipse indexing error causing function calls to be underlined in red

To fix these red underlines, you can go to the Project drop down menu and select C/C++ Index-> Rebuild. This will fix any Eclipse indexing issues which results in library functions being underlined in red.
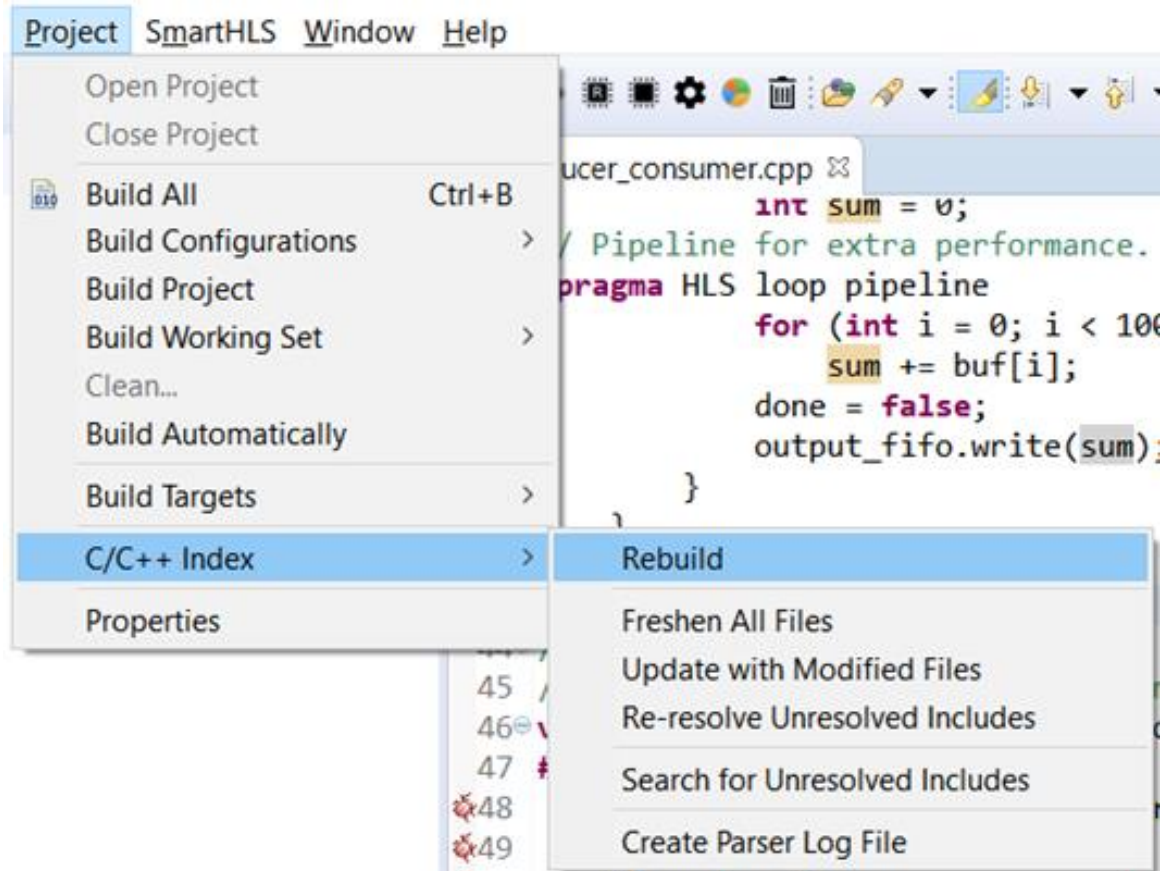
In this training, we will not be creating SmartHLS projects from scratch, please see the SmartHLS Sobel Tutorial for instructions on creating a fresh SmartHLS project.

9   Alpha Blend Block

We will start by looking at the alpha blending block. Alpha blending is the process of combining a foreground image with a background image, giving the appearance of transparency as shown in Figure 10.
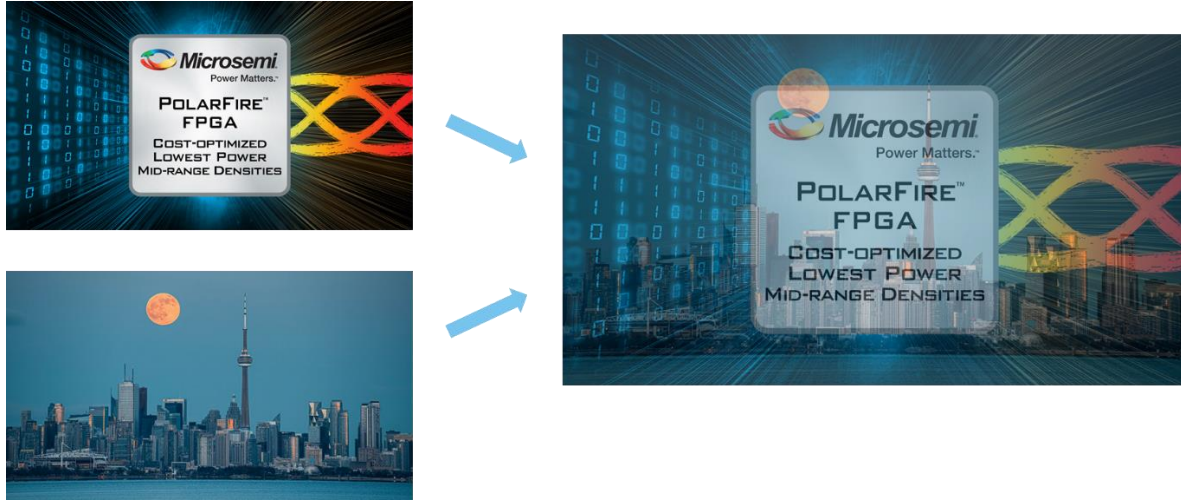
 Figure 10: Microsemi PolarFire® banner alpha blended with Toronto skyline in the background.

The degree of translucency when combining the foreground and background images is given by an alpha input coefficient. Given an input pixel with a red, green, blue (RGB) value, then the alpha blended output for each color (RGB) is given by the equation below:

$$R_{out} = R_{channel1} \times (1 - alpha) + R_{channel2} \times alpha$$
$$G_{out} = G_{channel1} \times (1 - alpha) + G_{channel2} \times alpha$$
$$B_{out} = B_{channel1} \times (1 - alpha) + B_{channel2} \times alpha$$

In the equations above, alpha ranges from 0 to 1. But in hardware the alpha input is represented by an 8-bit value that ranges from 0 to 255. An alpha input of 0 means that the image in channel 2 is completely transparent while an alpha of 255 indicates the image in channel 2 is completely opaque. For example, in Figure 10 the foreground is 50% transparent, alpha is 0.5 which is represented by the 8-bit value 127 in hardware.

For this demo we created an Alpha Blending block in SmartHLS. Our goal was to use the SmartHLS generated SmartDesign IP component as a drop-in replacement for the Alpha Blending SolutionCore previously used in the PolarFire® Video Kit demo design, see UG0641 User Guide Alpha Blending.

The block diagram of the Alpha Blending SolutionCore is shown in Figure 11 and the input and output interface are described in Table 1. Our SmartHLS-generated SmartDesign IP has an RTL interface that is compatible with the SolutionCore but not identical, since SmartHLS will

generate a few extra unused control signals (start, finish, ready, etc.).
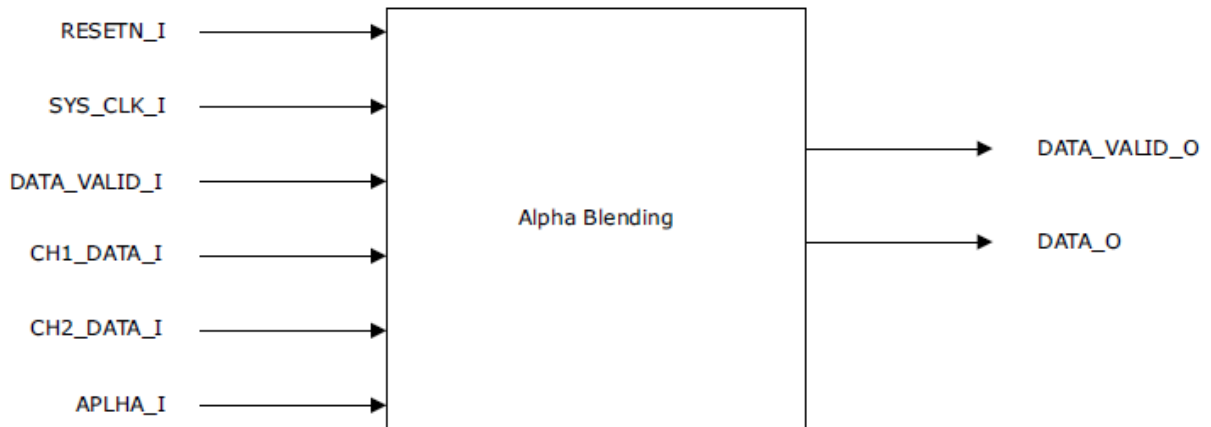
Figure 11: Block Diagram of Alpha Blending SolutionCore IP

Table 1: Alpha Blending SolutionCore IP Interface

Signal Name	Direction	Width	Description
RESETN_I	Input	1-bit	Active low async reset
SYS_CLK_I	Input	1-bit	System Clock
DATA_VALID_I	Input	1-bit	Input data valid
CH1_DATA_I	Input	24-bits	Channel 1 input data Three 8-bit pixels: 23:16 Red 15:8 Green 7:0 Blue
CH2_DATA_I	Input	24-bits	Channel 2 input data (RGB)
ALPHA_I	Input	8-bits	Alpha inputs (0-255)
DATA_VALID_O	Output	1-bit	Output data valid
DATA_O	Output	24-bits	Output data (RGB)

In SmartHLS, we open the alpha_blend project in the Project Explorer and double click on the alpha_blend.cpp C++ source file:
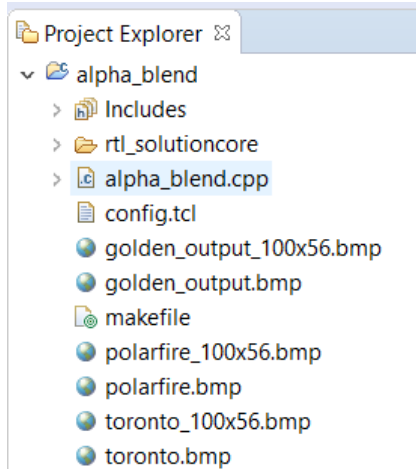
Now we click the SmartHLS Compile Software to Hardware button ( ). SmartHLS will compile the included C++ source files into the equivalent logic in Verilog. Figure 12 shows the output files and directories generated by SmartHLS after compiling to hardware.

1. Directory holding the initialization .mem files for RAMs.

2. Directory holding reports about the hardware.
   a. *dot_graphs* directory holds dot files used by the Schedule Viewer.
   b. *hls.log* has the Console output of the last SmartHLS command executed.
   c. *pipelining.hls.rpt* has pipeline scheduling information used by Scheduler Viewer.
   d. *scheduling.hls.rpt* has scheduling information used by the Scheduler Viewer.
   e. *summary.hls.alpha_blend_smarthls.rpt* has a summary of the other reports as well as interface and RAM information.

3. Generated Verilog design.

4. Generated VHDL wrapper for Verilog design.

5. TCL script to import Verilog design into SmartDesign.

6. ModelSim script to display module ports in a hierarchy.

- ∨ 📁 alpha_blend
  - › 📁 Includes
  - ∨ 📁 hls_output
    - ∨ 📁 reports | 2 |
      - › 📁 dot_graphs
      - 📄 hls.log
      - 📄 pipelining.hls.rpt
      - 📄 scheduling.hls.rpt
      - 📄 summary.hls.alpha_blend_smarthls.rpt
      - 📄 summary.results.rpt
    - ∨ 📁 rtl
      - › 📁 mem_init | 1 |
      - 📄 alpha_blend_alpha_blend_smarthls.v | 3 |
      - 📄 alpha_blend_alpha_blend_smarthls.vhd | 4 |
    - ∨ 📁 scripts
      - 📄 create_hdl_plus_alpha_blend_smarthls_top.tcl
      - 📄 create_hdl_plus.tcl | 5 |
      - 📄 hierarchy.tcl | 6 |
    - › 📁 simulation
    - › 📁 synthesis
  - › 📁 rtl_solutioncore
  - › 📄 alpha_blend.cpp
  - 📄 config.tcl
  - 🌐 golden_output_100x56.bmp
  - 🌐 golden_output.bmp
  - 📄 makefile
  - 🌐 output.bmp
  - 🌐 polarfire_100x56.bmp
  - 🌐 polarfire.bmp
  - 🌐 toronto_100x56.bmp
  - 🌐 toronto.bmp

Figure 12: SmartHLS Output Files

The SmartHLS summary.hls.alpha_blend_smarthls.rpt report file should open automatically (this can also be found under the reports directory in the Project Explorer). We can see the RTL interface of the generated SmartHLS Alpha blending block by scrolling down to Section 1:

```
+----------------------------------------------------------------------------------------+
| RTL Interface Generated by SmartHLS                                                    |
+-------------+------------------+-------------------+-----------------+------------------+
| C++ Name    | Interface Type   | Signal Name       | Signal Bit-width| Signal Direction |
+-------------+------------------+-------------------+-----------------+------------------+
|             | Clock & Reset    | clk (positive edge)| 1              | input            |
|             |                  | reset             | 1               | input            |
+-------------+------------------+-------------------+-----------------+------------------+
|             | Control          | finish            | 1               | output           |
|             |                  | ready             | 1               | output           |
|             |                  | start             | 1               | input            |
+-------------+------------------+-------------------+-----------------+------------------+
| output_fifo | Output AXI Stream | output_fifo_ready| 1               | input            |
|             |                  | output_fifo_valid | 1               | output           |
|             |                  | output_fifo       | 24              | output           |
+-------------+------------------+-------------------+-----------------+------------------+
| input_fifo  | FIFO             | input_fifo_ready  | 1               | output           |
|             |                  | input_fifo_valid  | 1               | input            |
|             |                  | input_fifo_channel2| 24             | input            |
|             |                  | input_fifo_alpha  | 8               | input            |
|             |                  | input_fifo_channel1| 24             | input            |
+-------------+------------------+-------------------+-----------------+------------------+
```
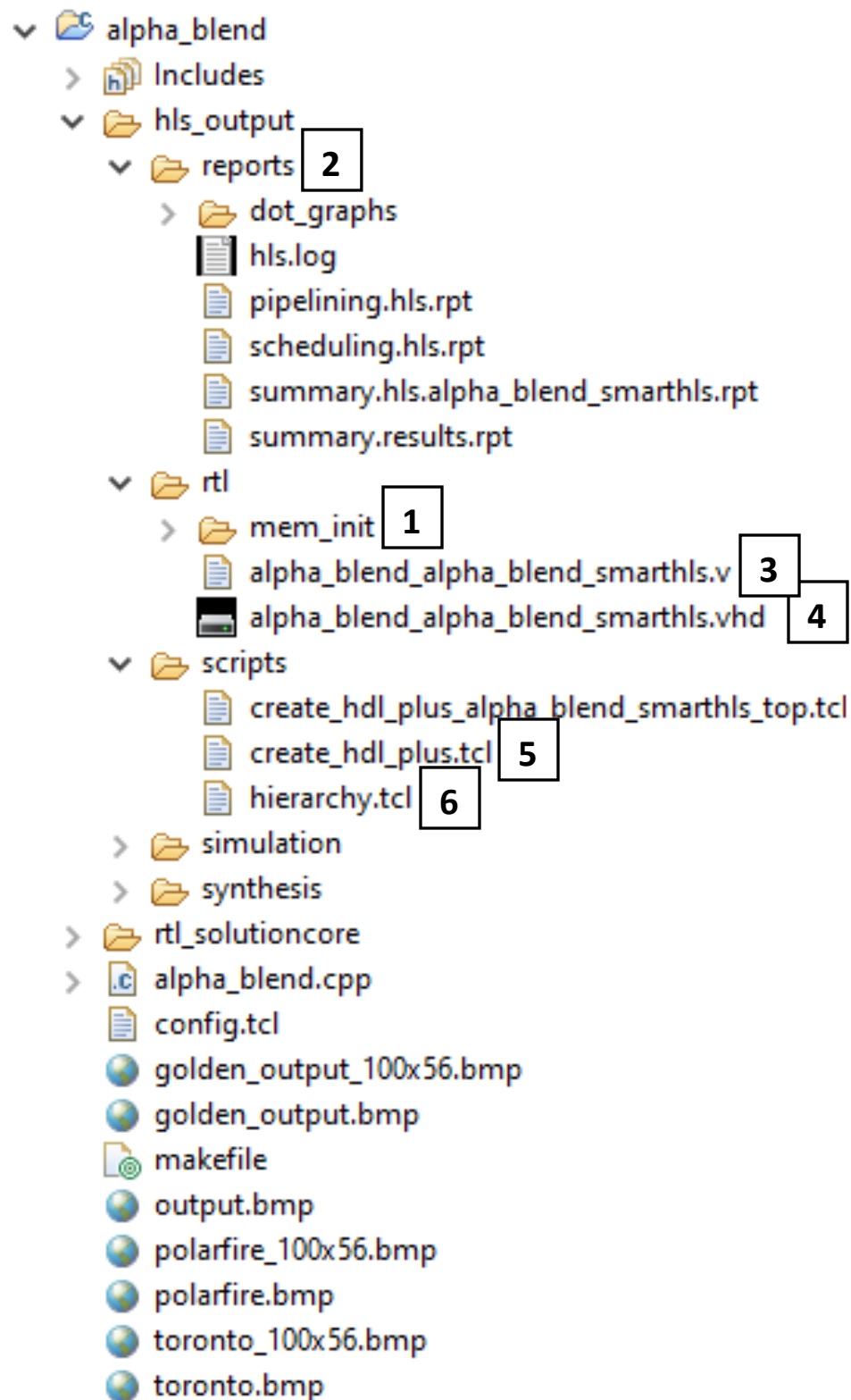
RTL interfaces reported by SmartHLS are grouped into different interface types. The first interface includes the clock and reset signals, which match the SolutionCore IP. Note, the reset signal in the SolutionCore IP is active low but the reset signal on SmartHLS generated blocks is always synchronous active high. We will compensate for this by inverting the reset port in SmartDesign. The second interface type in the table called "Control" includes an input port and two output ports: *start, finish*, and *ready*. We will tie the *start* input to high and mark the *finish* output as unused in SmartDesign since the module should always be running. The *ready* output is also unused because there is no backpressure in this design.

The remaining SmartHLS interfaces, output_fifo and input_fifo, match with the SolutionCore input/output data and data valid ports from Table 1 except for the extra *fifo_ready* signals. We will also mark these as unused because there is no backpressure in this design.

Therefore, based on the SmartHLS interface report, we can use the SmartHLS generated Alpha Blend SmartDesign IP block as a drop-in replacement for the Alpha Blending SolutionCore IP.

You can also check the top-level module interface of the SmartHLS-generated Verilog file in "alpha_blend_alpha_blend_smarthls.v" found in the alpha_blend/hls_output/rtl/ directory.

```
module alpha_blend_smarthls_top
(
    clk,
    reset,
    start,
```

32

```
        ready,
        finish,
        input_fifo_channel1,
        input_fifo_ready,
        input_fifo_valid,
        input_fifo_channel2,
        input_fifo_alpha,
        output_fifo,
        output_fifo_ready,
        output_fifo_valid
    );
```

Now we will look at the implementation of the alpha blending hardware block in SmartHLS.

Go to alpha_blend.cpp.

We start by looking at the C++ function arguments that SmartHLS will turn into the interface in RTL that we described previously. Look at the function signature of the top-level function on line 102:

```
void alpha_blend_smarthls(hls::FIFO<input_t> &input_fifo,
                          hls::FIFO<rgb_t>   &output_fifo) {
#pragma HLS function top
```

The top-level C++ function will be compiled by SmartHLS into the top-level Verilog module. You can tell that this is the top-level by the SmartHLS pragma: "function top".

The RTL interface generated by SmartHLS depends on the C++ arguments of the top-level function.
We start with the simpler second argument "output_fifo" which has the type:
```
        hls::FIFO<rgb_t>
```

The < > brackets surround the C++ template argument which defines the data type stored in the FIFO. In this case the FIFO holds `rgb_t` data. You can mouse over the `rgb_t` to display the type definition:

```
void alpha_blend_smarthls(hls::FIFO<input_t> &input_fifo,
                          hls::FIFO<rgb_t>   &output_fifo) {

#pragma HLS function top                // 24-bit RGB
#pragma HLS function pipeline           typedef ap_uint<3*W> rgb_t;
                                                    Press 'F2' for focus
```

If you scroll to the top of the C++ file you can find the data type defined as an arbitrary unsigned integer `ap_uint` with a bitwidth of 3*W=24:
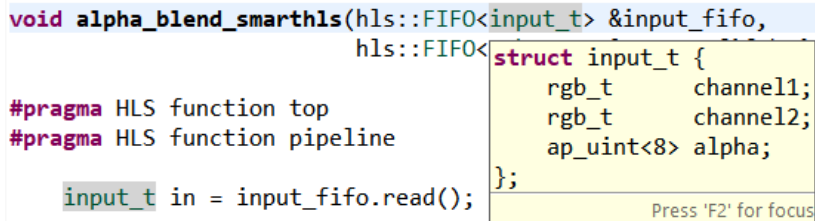
```
// bit width of a pixel
const int W = 8;
// 24-bit RGB
typedef ap_uint<3*W> rgb_t;
```

33

In SmartHLS, the hls::FIFO type will generate an RTL interface with a data, 1-bit valid and 1-bit ready interface. The output_fifo interface will have a 24-bit output data corresponding to the rgb_t type. This will create the output_fifo interface we saw in the report file.

Going back to the top-level function, the first argument "input_fifo" has the type hls::FIFO<input_t>. You can mouse over the input_t type to see the definition (this can also be found at the top of the C++ file):

```
void alpha_blend_smarthls(hls::FIFO<input_t> &input_fifo,
                          hls::FIFO< struct input_t {
                              rgb_t       channel1;
                              rgb_t       channel2;
#pragma HLS function top       ap_uint<8> alpha;
#pragma HLS function pipeline  };
                                          Press 'F2' for focus
    input_t in = input_fifo.read();
```

When a FIFO holds a struct type, then SmartHLS will split each struct element as a separate RTL data port, but all the ports will share the same 1-bit ready and 1-bit valid control signals. In this case, we will have a 24-bit channel1, a 24-bit channel2, and an 8-bit alpha. This will create the input_fifo interface we saw in the report file.

Next, we will look at the internal implementation of the Alpha Blending block by looking at the function body.

```
void alpha_blend_smarthls(hls::FIFO<input_t> &input_fifo,
                          hls::FIFO<rgb_t>   &output_fifo) {

#pragma HLS function top
#pragma HLS function pipeline

    input_t in = input_fifo.read();

    // alpha ranges from 0 to 255
    ap_uint<16> alpha = 1 + in.alpha;
    rgb_t out;

    // red
    out(R1, R2) = (in.channel1(R1, R2) * (256 - alpha) + in.channel2(R1, R2) * alpha) >> 8;
    // green
    out(G1, G2) = (in.channel1(G1, G2) * (256 - alpha) + in.channel2(G1, G2) * alpha) >> 8;
    // blue
    out(B1, B2) = (in.channel1(B1, B2) * (256 - alpha) + in.channel2(B1, B2) * alpha) >> 8;

    output_fifo.write(out);
}
```

You will see that we added the "function pipeline" pragma on line 106 to ensure the hardware generated by SmartHLS is pipelined. This is necessary to replicate the behavior of the SolutionCore IP. The Alpha Blending SolutionCore IP can accept input every cycle (initiation interval of 1) so we want to make sure the SmartHLS block can as well.

Open the summary.hls.alpha_blend_smarthls.rpt file again and scroll to section 2: Function and Loop Scheduling Results . Verify the initiation interval of the *alpha_blend_smarthls* hardware block is 1 by scrolling to the right.

```
====== 2. Function and Loop Scheduling Results ======

+------------------------------------------------------+
| Function: alpha_blend_smarthls takes 4 cycles II = 1 |
+------------------------------------------------------+
```

The initiation interval is 1, meaning that this hardware block can accept a new input value every clock cycle. We will cover SmartHLS pipelining in more detail in Section 10 including cases where the pipeline initiation interval must be greater than 1. The pipeline depth of this block is 2, meaning we will get the first output 2 cycles after the first input, after which time we will get the next output every clock cycle.

In the body of the function, we read from the input FIFO, perform some computations, and write to the output FIFO.
```
        input_t in = input_fifo.read();
        ...
        output_fifo.write(out);
```

Now we look at the calculation of the 8-bit output red pixel. The alpha input is represented by an 8-bit value (0 to 255). We could divide alpha by 255 to map this to the 0 to 1 floating point value but we want to avoid any floating-point math. Instead, we can add 1 to alpha to make the maximum alpha value 256, then multiply alpha by the 8-bit pixel values and afterwards we divide by 256, which is equivalent to right shifting by 8.

```
        ap_uint<16> alpha = 1 + in.alpha;
        rgb_t out;
        // red
        out(R1, R2) = (in.channel1(R1, R2) * (256 - alpha) + in.channel2(R1, R2) * alpha) >> 8;
```

The ap_uint syntax out(R1, R2) is used to write a specific range of bits into the "out" 24-bit. In this case, we are writing 8 bits to the range of bits from 23:16 corresponding to the red pixel. Where R1/R2 are defined as (R2=16, R1=23):
```
        // 23:16 red
        const int R2 = 2*W;
        const int R1 = R2 + W-1;
```

Similarly, the in.channel1(R1, R2) syntax reads the 8-bit red pixel value (23:16) from the 24-bit channel1 input.

9.1   SmartHLS Schedule Viewer

Now that we have generated the hardware with SmartHLS, we can launch the SmartHLS
Schedule Viewer (click the 🌐 button). The Schedule Viewer shows more information on how
the Alpha Blending C++ function body was converted into a hardware pipeline. In particular, the
scheduling of operations in the generated Verilog block.

The Schedule Viewer has four views: the Call Graph, Control Flow Graph, Schedule Chart and
Pipeline Viewer. The Call Graph contains the directed graph of which software functions are
called by which other software functions within the design. The Control Flow Graph shows the
control flow of execution between the blocks within each function for if/else conditionals and
loops. The Schedule Chart and Pipeline Viewer shows the scheduling of instructions within a
block or a pipeline on a cycle-by-cycle basis.

9.1.1   Background: LLVM Internal Representation used by SmartHLS

The instructions displayed in the Schedule Viewer are from the LLVM compiler that SmartHLS is
built on. These assembly-like instructions are called LLVM intermediate representation (IR).
Some understanding of the LLVM IR is beneficial.

For example, given the 32-bit C++ code:
>   result = a + b − 5

This C++ code could be represented as instructions in LLVM IR as:
```
%0 = add i32 %a, %b
%result = sub i32 %0, 5
```

In LLVM IR, intermediate variables are prefixed with a "%". Each operation (add/sub) includes
the bitwidth "i32" indicating 32-bit integer. The add operands are %a + %b and the result is
stored in a temporary 32-bit variable %0. The subtract operands are %0 − 5 and the result is
stored in the variable %result.

Basic blocks are also important concepts in LLVM IR. A basic block is a group of instructions that
always run together with a single entry point at the beginning and a single exit point at the end.
A basic block in LLVM IR always has a label at the beginning and a branching instruction at the
end (br, ret, etc.). Here the body.0 basic block performs some operations and then branches
unconditionally to another basic block labeled body.1. Control flow occurs between basic
blocks.
```
body.0:
  %0 = add i32 %a, %b
  %result = sub i32 %0, 5
  br label %body.1
```

All of the basic blocks and instructions shown in the Scheduler Viewer are directly from the LLVM IR optimized by SmartHLS before being compiled into Verilog.

9.1.2   Call Graph

When you open the Schedule Viewer, the default view is of the Call Graph as shown in Figure 13. You can also click on the "Call Graph" tab at the top. The Call Graph shows the top-level C++ function and all of the sub-functions that are called. Since there are no function calls within the alpha_blend_smarthls function, there is only one bubble in the Call Graph.
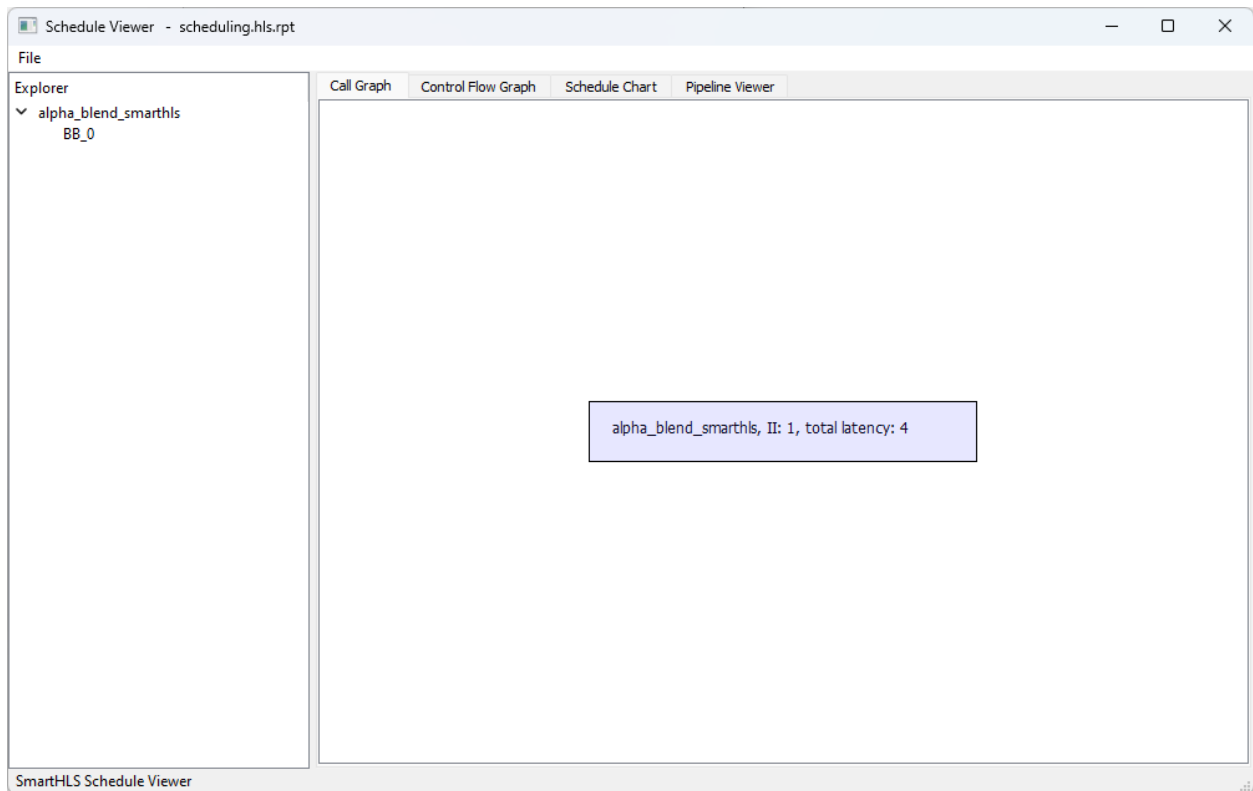

Figure 13: SmartHLS Schedule Viewer: Call Graph

9.1.3 Control Flow Graph

In the Schedule Viewer GUI, find the Explorer tab on the left. The Explorer tab holds all of the functions generated in hardware and their basic blocks.

Next, click on the the alpha_blend_smarthls function in the Explorer tab. This will open the Control Flow Graph viewer as shown in Figure 14. The long and unreadable or "mangled" name is the name of the basic block. This name is generated by LLVM to avoid name conflicts. SmartHLS normally "demangles" names to be readable but SmartHLS cannot support all cases (to be fixed in a future release). The Control Flow Graph shows the connections between basic blocks and shows which basic blocks can branch to which other basic blocks. Since there is only one basic block within the alpha_blend_smarthls function, there is only one bubble in the Control Flow Graph.
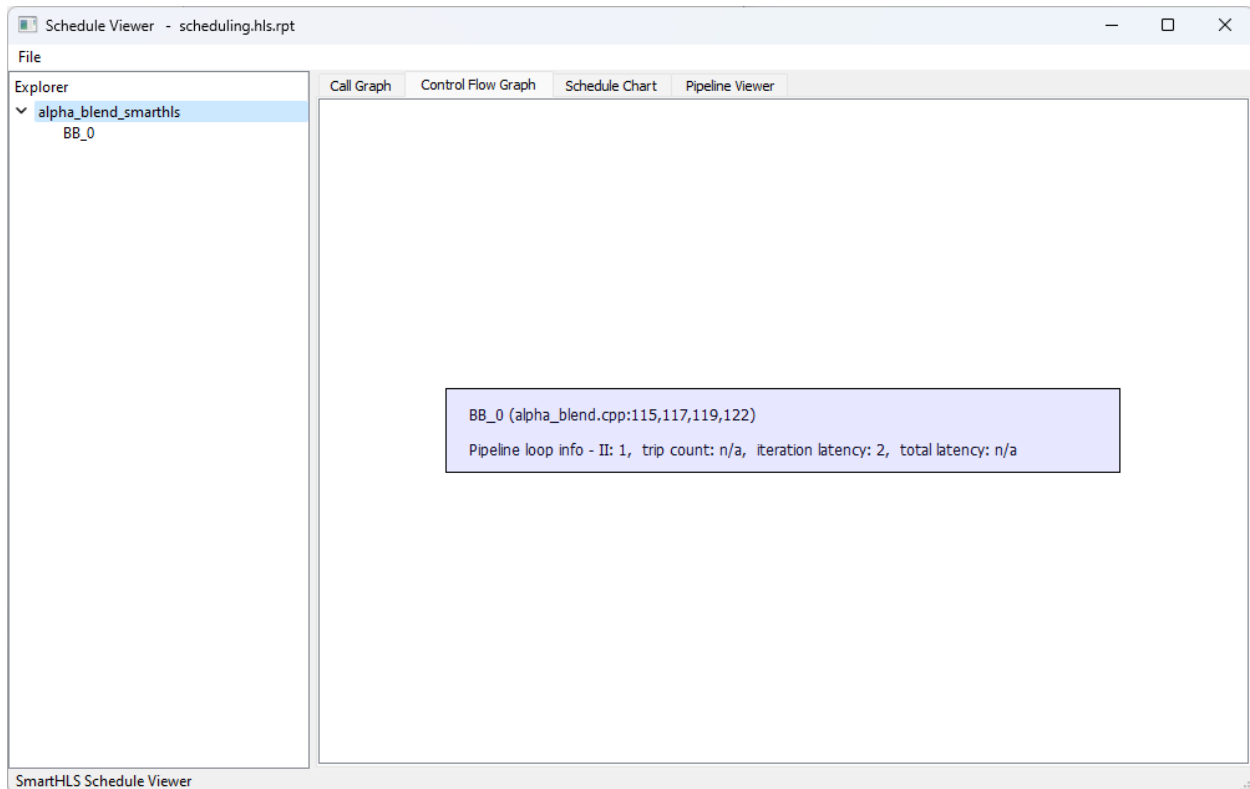

Figure 14: SmartHLS Schedule Viewer: Control Flow Graph

9.1.4 Pipeline Viewer

Now click on the basic block below the alpha_blend_smarthls function in the Explorer pane to bring up the Pipeline Viewer in Figure 15. At the top of this view, we find the initiation interval for the function.

The column headings in the first row show the clock cycle and pipeline stages for each column. The remaining rows show the instructions that run in the pipeline at each stage. The leftmost column indicates the loop iteration for the instructions in the row starting (from Iteration 0). For function pipelines, Iteration 0 corresponds to the first input.
If you hold you mouse over an instruction you will see more details about the operation type.
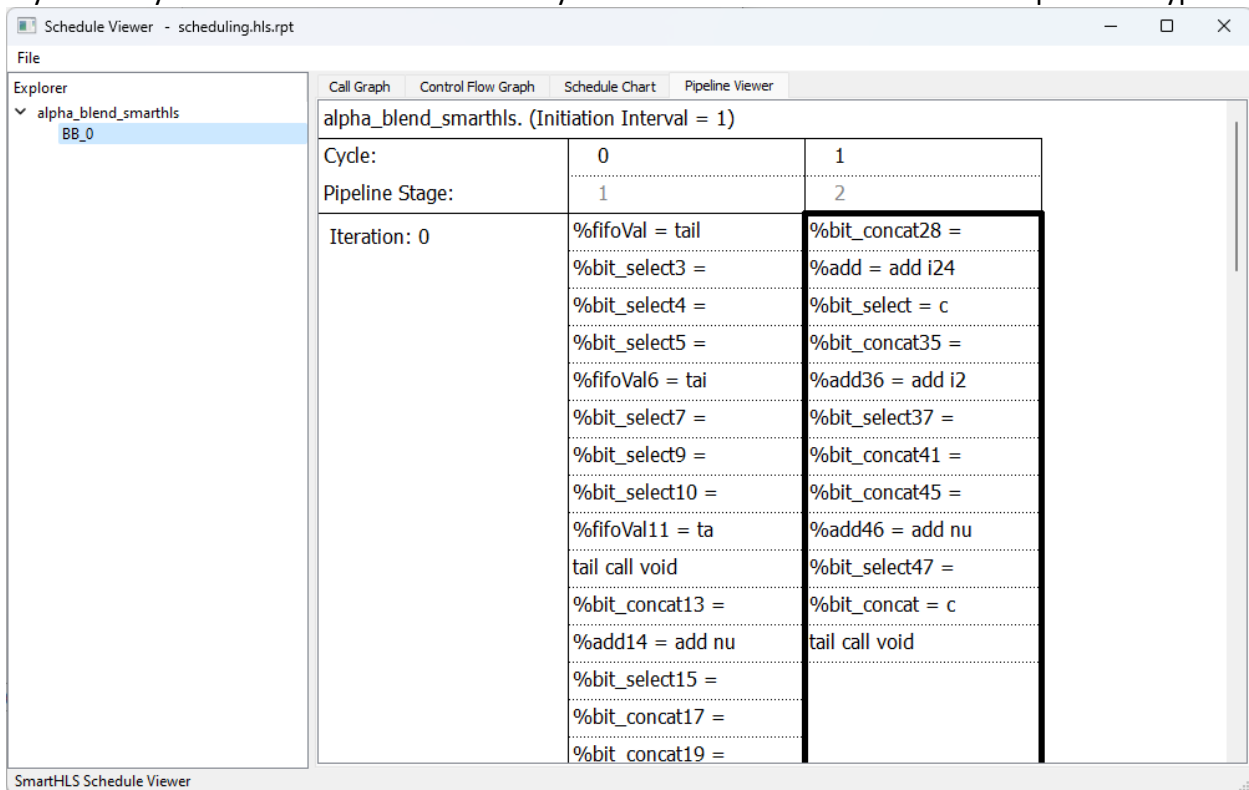

Figure 15: SmartHLS Schedule Viewer: Pipeline Viewer

In the pipeline viewer, the right-most column is highlighted with a thick black box and shows behavior of the pipeline in steady state as shown above. In this case, the steady state behavior is shown in pipeline stage 2.

From the pipeline viewer, we can see many instructions run in pipeline stage 1 and then the remaining instructions run in pipeline stage 2. The multiply operations are scheduled in pipeline stage 1 along with all of the other instructions that have no dependencies. The multiply operations take 1 cycle to finish. Any instructions that depend on the result of the multipy operations performed in pipeline stage 1 are scheduled in pipeline stage 2. These instructions cannot be scheduled until the multiply operations finish.

Scroll down to the bottom left to find the last iteration as shown in Figure 16. The SmartHLS pipeline viewer only shows the pipeline schedule until steady state. This last iteration is the first iteration of pipeline steady state. We can see that this pipeline reaches steady state after 2 iterations (1+1, since the Iteration index starts at 0) for a loop pipeline, or 2 inputs in the case of function pipelining. You can scroll to the bottom right to see the instructions scheduled.
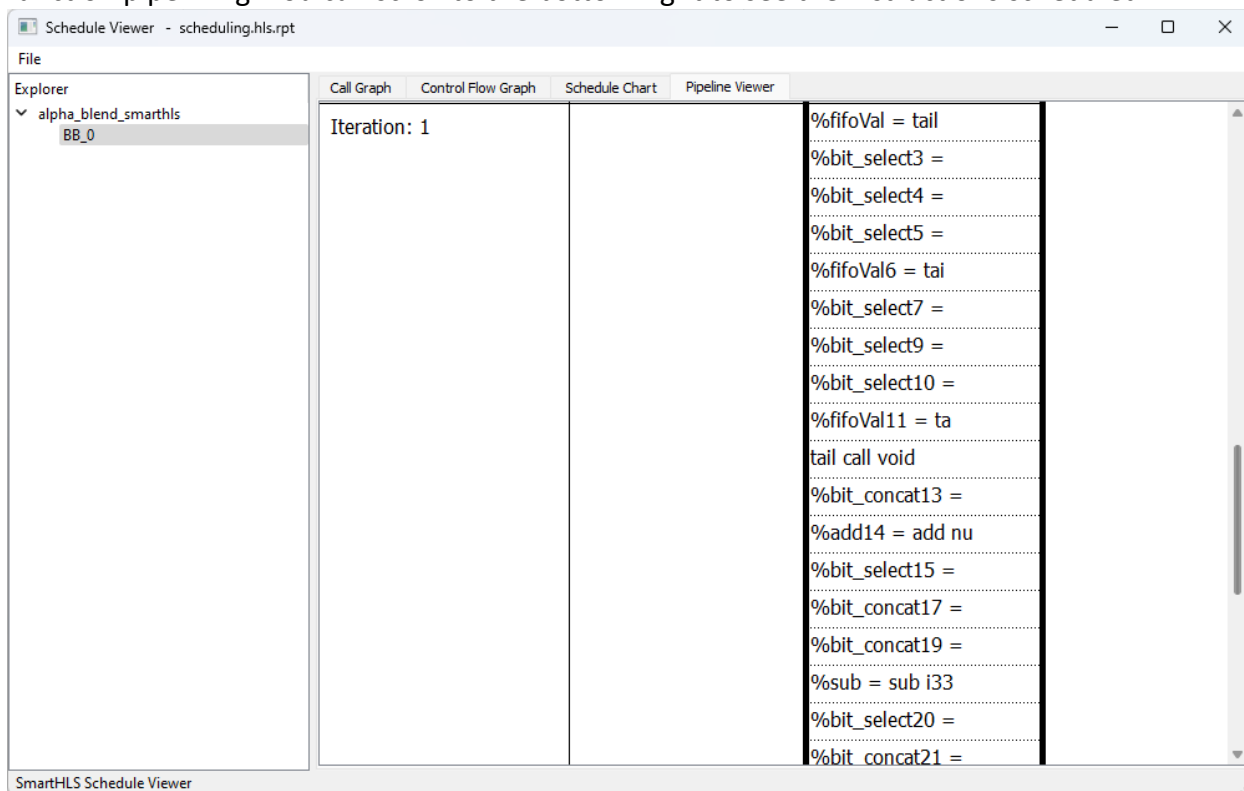

Figure 16: SmartHLS Scheduler Viewer: Pipeline Viewer. Iteration Where Steady State Reached.

The 2 iterations/inputs until steady state corresponds to the Pipeline Depth from the SmartHLS report file summary.hls.alpha_blend_smarthls.rpt file we saw previously:

```
====== 2. Function and Loop Scheduling Results ======

+--------------------------------------------------------+
| Function: alpha_blend_smarthls takes 4 cycles II = 1 |
```

+- +

9.1.5   Schedule Chart

Finally, click on the "Schedule Chart" tab at the top of the schedule viewer. The Schedule Chart shows the instructions of the non-pipelined basic block selected in the Explorer tab and which cycle they are scheduled for in hardware after the *start* signal is asserted. However, recall the SmartHLS pragma "function pipeline" on line 106 of alpha_blend.cpp. Because this function is pipelined, the Schedule Chart view is empty and you should refer to the Pipeline Viewer instead.
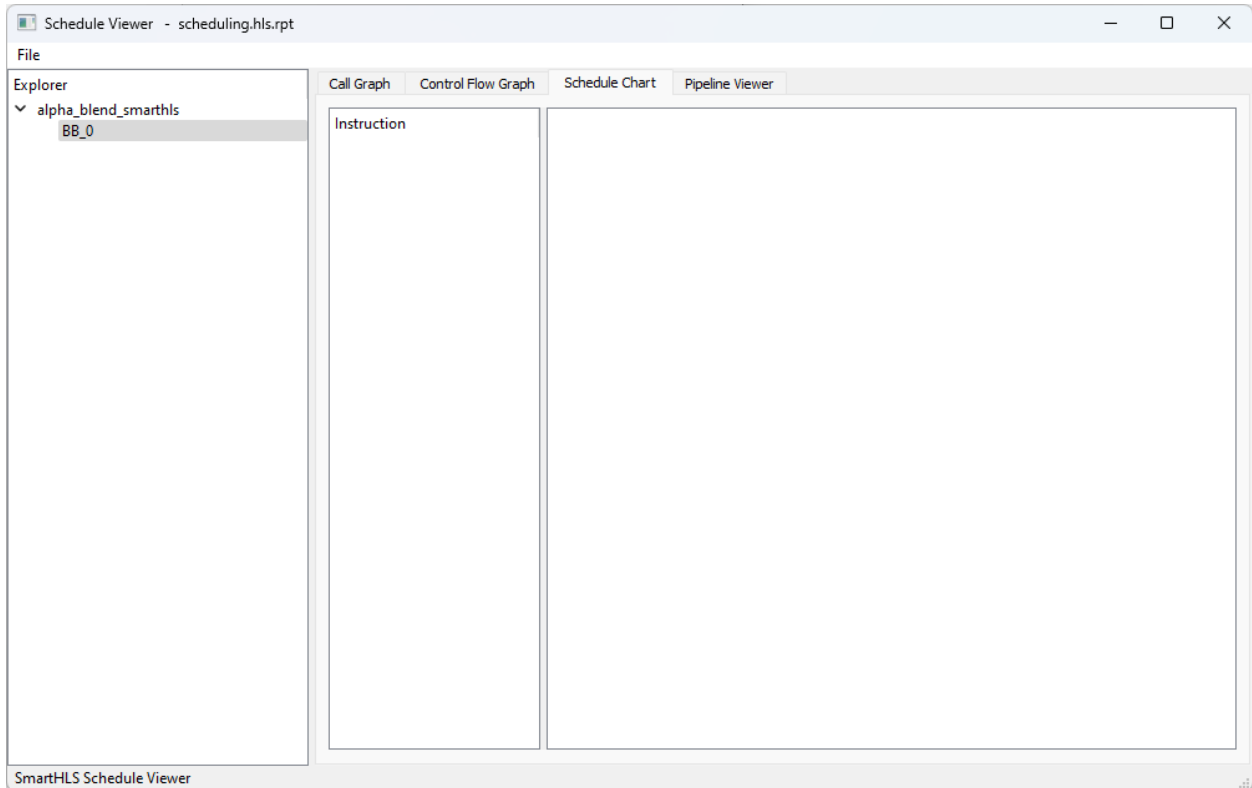

Figure 17: Empty Pipeline Viewer

Now close the Schedule Viewer window.

9.2   Design Verification: Software Testing

Now we will explain how we perform testing and verification on the Alpha Blend SmartHLS design. In the Project Explorer, double click the input image files: toronto.bmp and polarfire.bmp. These are the inputs to our alpha blend testbench: the C++ main() function.

We always recommend first testing your C++ design in software to verify correctness before

41

running any co-simulations. The reason why is software execution is always much faster than simulation. If the software execution is incorrect then the simulation will also fail.

First, we will run the tests for this block in software to confirm the functional correctness of the design. Click the compile software button [C] on the top bar and then click the run software ⊙ button. You should see this output in the Console:

```
Alpha = 127
PASS!
```

The "PASS!" is printed by our main() testbench function on line 204 when the output image matches the golden expected output image. You can visually confirm yourself by clicking on the output image file: output.bmp in the Project Explorer. The image will open in Eclipse and the output image should look like Figure 10. The expected output image is: golden_output.bmp.

The testbench of the alpha blend block which we just ran is defined in the main() function. None of C++ code inside the main function will be turned into hardware. This testbench code is just for verifying the functionality of the top-level function. SmartHLS will only generate Verilog for the top-level function alpha_blend_smarthls().

Notice the lines near the top of the file on line 32 that define which input image file is used. The commented out FAST_COSIM define might be folded into the comment by eclipse and needs to be expanded by clicking the plus button.

`32⊕ // uncomment this line to test on a smaller image for faster co-simulation⬜`

The lines highlighted in gray are disabled (since FAST_COSIM is not defined). This means that for the software simulation we just ran, we used the 1920x1080 bmp image sets.

```
// uncomment this line to test on a smaller image for faster co-simulation
//#define FAST_COSIM

#ifdef FAST_COSIM
#define WIDTH 100
#define HEIGHT 56
#define INPUT_IMAGE1 "toronto_100x56.bmp"
#define INPUT_IMAGE2 "polarfire_100x56.bmp"
#define GOLDEN_OUTPUT "golden_output_100x56.bmp"
#else
#define WIDTH 1920
#define HEIGHT 1080
#define INPUT_IMAGE1 "toronto.bmp"
#define INPUT_IMAGE2 "polarfire.bmp"
#define GOLDEN_OUTPUT "golden_output.bmp"
#endif
#define SIZE (WIDTH*HEIGHT)
```

In the main() function on line 135, we use the helper functions read_bmp() to read the image

files from disk. The following line will read either the 1920x1080 RGB pixels values from the "toronto.bmp" or the 100x56 values from "toronto_100x56.bmp" input file depending on whether FAST_COSIM is defined or not:

```
input_channel1 = read_bmp(INPUT_IMAGE1, &input_channel1_header);
```

Same with the second input channel, which will read either "polarfire.bmp" or "polarfire_100x56.bmp":

```
input_channel2 = read_bmp(INPUT_IMAGE2, &input_channel2_header);
```

The golden expected output will read either "golden_output.bmp" or "golden_output_100x56.bmp":

```
golden_output_image = read_bmp(GOLDEN_OUTPUT, &golden_output_image_header);
```

In our C++ testbench on line 147, we first perform a sanity check test based on the waveform in the alpha blending SolutionCore documentation (UG0641 page 4) shown in Figure 18.

```
// test 1: sanity check from alpha blend IP core documentation
in.channel1 = ap_uint<24>("0x456712");
in.channel2 = ap_uint<24>("0x547698");
in.alpha    = ap_uint<8>("0x84");
...
```
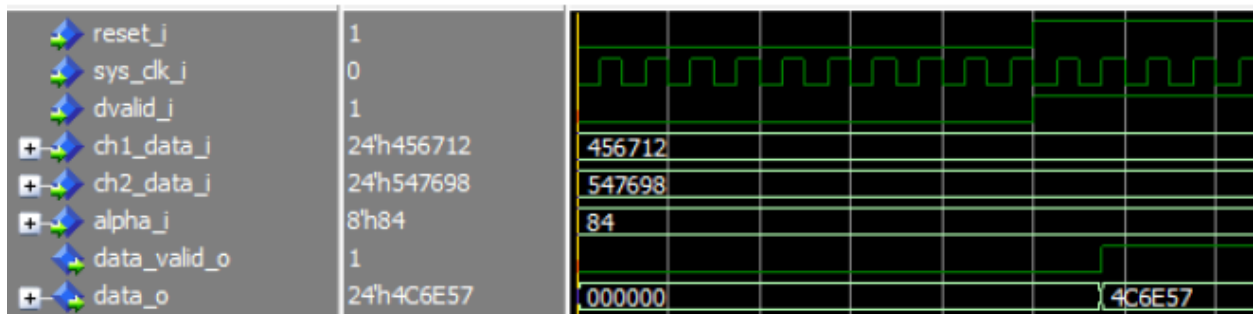

Figure 18: Alpha Blend SolutionCore Documentation Test Waveform

We initialize the input values, then write into the input_fifo, call the top-level function alpha_blend_smarthls, and read out the output from the output_fifo. Finally, we validate the output was expected. If there was a mismatch, we print out the value and then return a non-zero value from main so that the co-simulation will FAIL. Co-simulation will only pass if the main function returns zero.

```
// test 1: sanity check from alpha blend IP core documentation
in.channel1 = ap_uint<24>("0x456712");
in.channel2 = ap_uint<24>("0x547698");
in.alpha    = ap_uint<8>("0x84");
input_fifo.write(in);
alpha_blend_smarthls(input_fifo, output_fifo);
rgb_t out = output_fifo.read();
if (out != ap_uint<24>("4C6E57")) {
    std::cout << "out = " << out.to_string() << std::endl;
    std::cout << "FAIL!" << std::endl;
```

43

```
        return 1;
    }
```

Next, starting from line 160, we run alpha blending on the two input image files. We specify the input alpha value of 50%, which is represented by the 8-bit value 127:

```
    in.alpha = (int)(255 * 0.5);
```

We loop over each pixel (WIDTH x HEIGHT) of the input images. When reading from a BMP image file, consecutive pixels in the same row of the image are stored next to each other (row-major order). Therefore, the outer loop is over the image HEIGHT and the inner loop is over the WIDTH of the image:

```
    for (int i = 0; i < HEIGHT; i++) {
        for (int j = 0; j < WIDTH; j++) {
```

Note: this loop order does not matter in this example since we do not use the i or j indexes inside the loop body. At the end of the loop, we increment all the pointers for each of the images to the next pixel in the image.

In the loop body, we use the ap_uint concatenation operator "(R, G, B)" to assign the 24-bit input channels. The red pixel will be the most-significant 8 bits of the 24-bit input channel and the blue pixel will be the least-significant 8 bits.

```
            // concatenation operator
            in.channel1 = (ap_uint<8>(input_channel1->r),
                           ap_uint<8>(input_channel1->g),
                           ap_uint<8>(input_channel1->b));
```

After we write to the input_fifo we call the top-level function alpha_blend_smarthls, and then we read the output from the output_fifo. We extract out the 8-bit RGB values from the 24-bit output:

```
            rgb_t rgb = output_fifo.read();
            output_image_ptr->r = rgb(R1, R2);
            output_image_ptr->g = rgb(G1, G2);
            output_image_ptr->b = rgb(B1, B2);
```

Then we verify the output pixel matches the expected pixel. We return 1 from main if there is a mismatch.

At the end of the main function we write the alpha blended image to the "output.bmp" file:

```
    write_bmp("output.bmp", &input_channel1_header, output_image);
```

We reuse the same BMP header data (image properties like width and height) as the input channel 1 image.

And we print a message and return 0 from the main function to indicate to co-simulation that the testbench passed.

```
    printf("PASS!\n");
    return 0;
```

9.3 Design Verification: Software/Hardware Co-Simulation

Now we have confirmed that the software implementation is correct. We can now verify that the generated RTL is functionally correct with a co-simulation (co-sim) with ModelSim.

First uncomment the line defining FAST_COSIM and then save the file. Again, the commented out FAST_COSIM define might be folded into the comment by eclipse and needs to be expanded by clicking the plus button.

```
32 // uncomment this line to test on a smaller image for faster co-simulation
```

The FAST_COSIM define will change the input image to be 100x56 bmp files (instead of 1080p images). This change will speed up the co-simulation time considerably (from 20 min to 2 min):

```
// uncomment this line to test on a smaller image for faster co-simulation
#define FAST_COSIM
```

Since the code changed, we should recompile (🅒) and rerun (▶) the software verify that the software still passes on this new input:

```
Alpha = 127
PASS!
```

If you open the output.bmp image, you will notice the dimensions are now much smaller.

Since the code changed, we also need to rerun SmartHLS (🔧) to regenerate the hardware. Now, we start co-simulation (⏩) which will take a few minutes to finish. You should verify that the following results appear in the Console:

```
PASS!
+--------------------------+----------------+--------------------------+…+
| Top-Level Name           | Number of calls | Simulation time (cycles) |…|
+--------------------------+----------------+--------------------------+…+
| alpha_blend_smarthls_top | 5,601          | 5,605                    |…|
+--------------------------+----------------+--------------------------+…+
Simulation time (cycles): 5,605
SW/HW co-simulation: PASS

09:58:12 Build Finished (took 15s.387ms)
```

The "SW/HW co-simulation: PASS" indicates that the simulation was successful and the main() testbench function returned 0.

The SmartHLS co-simulation flow performs the following 3 steps automatically:
1. SmartHLS runs your main() testbench function in software. All inputs to the top-level function are saved in input test vector files.

45

2. SmartHLS generates an RTL testbench that will read the input test vector files from step 1. SmartHLS uses ModelSim to simulate the RTL testbench and SmartHLS-generated Verilog. The module outputs are saved into output simulation files.
3. SmartHLS reruns your main() testbench function in software but replaces the top-level function calls with the return value from the output simulation files from step 2. If the hardware outputs are correct then the main() function will still return 0 (PASS).

The co-simulation flow is useful to run as a sanity check that the SmartHLS generated hardware is correct and to report the number of clock cycles taken to run the testbench.

Now that we have validated that the hardware functionality is correct in simulation, we would like to know the FPGA resource usage and clock frequency of the IP block.
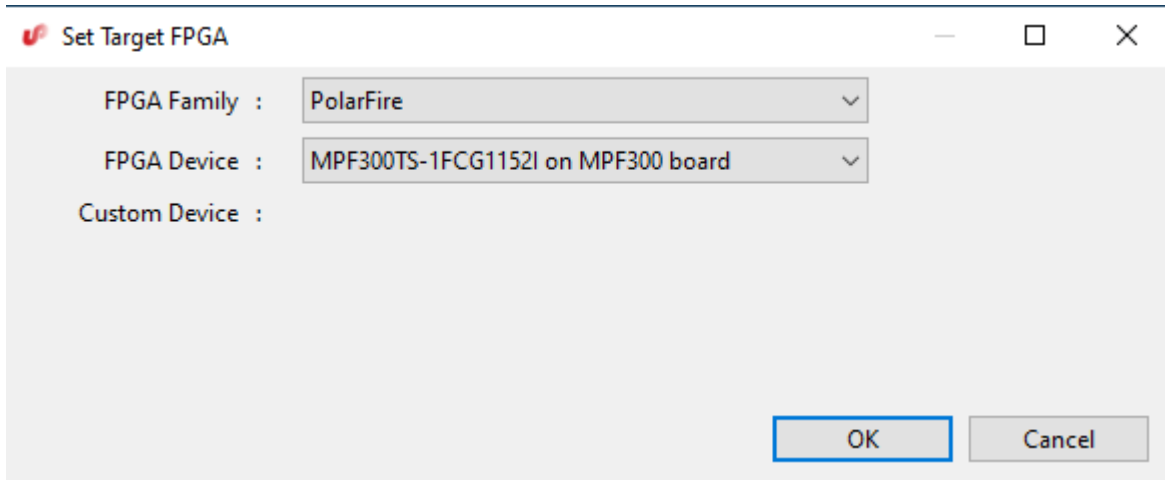
9.4   Target FPGA Device

We can open the SmartHLS -> Target FPGA Settings to confirm the target FPGA device of this project:
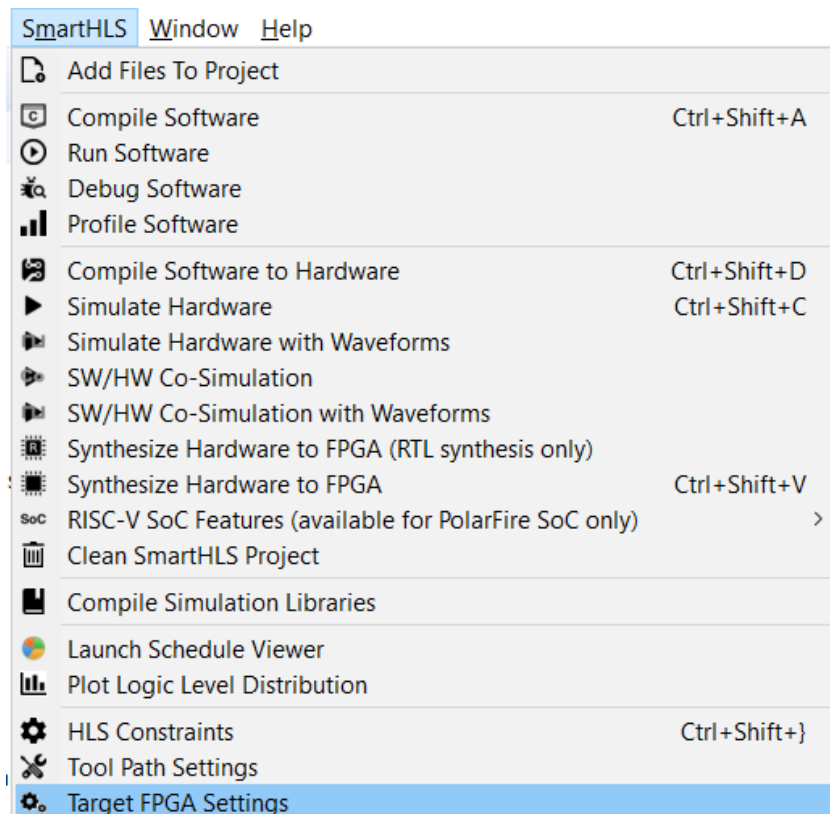
We are targeting PolarFire® MPF300TS device. Click OK:

The SmartHLS project device setting does a few things:
1) Sets up internal operator delay models for the target family. These delay models are used by SmartHLS to decide how much pipelining to add in the circuit to meet the Fmax constraint.
2) Passes the part number to Libero® SoC when running FPGA synthesis, place, and route ( ) to get resource/Fmax results.
3) Account for FPGA family-specific issues – for example SmartFusion2 RAMs do not support power-up initialization.

9.5   Design FPGA Implementation: Resources and Timing

     We can synthesize the design to target the PolarFire FPGA device by clicking on the . This will launch Libero SoC in the background and run synthesis, place, and route. This should take 5 minutes and generate the "synthesis" directory which holds the Libero SoC project directory, and the summary.results.rpt file. This will give both the timing and the resource results. Note that if you only want the resource result, you can click on , which will run synthesis only with no place and route.

We check the summary.results.rpt report file for timing and resource usage:

```
====== 2. Timing Result of HLS-generated IP Core (top-level module: alpha_blend_smarthls_top) ======

+--------------+---------------+-------------+-------------+----------+-------------+
| Clock Domain | Target Period | Target Fmax | Worst Slack | Period   | Fmax        |
+--------------+---------------+-------------+-------------+----------+-------------+
| clk          | 10.000 ns     | 100.000 MHz | 6.930 ns    | 3.070 ns | 325.733 MHz |
+--------------+---------------+-------------+-------------+----------+-------------+

The reported Fmax is for the HLS core in isolation (from Libero's post-place-and-route timing analysis).
When the HLS core is integrated into a larger system, the system Fmax may be lower depending on the
critical path of the system.

====== 3. Resource Usage of HLS-generated IP Core (top-level module: alpha_blend_smarthls_top) ======

+-------------------------+-----------------+--------+------------+
| Resource Type           | Used            | Total  | Percentage |
+-------------------------+-----------------+--------+------------+
| Fabric + Interface 4LUT*| 172 + 216 = 388 | 299544 | 0.13       |
| Fabric + Interface DFF* | 61 + 216 = 277  | 299544 | 0.09       |
| I/O Register            | 0               | 1536   | 0.00       |
| User I/O                | 0               | 512    | 0.00       |
| uSRAM                   | 0               | 2772   | 0.00       |
| LSRAM                   | 0               | 952    | 0.00       |
| Math                    | 6               | 924    | 0.65       |
+-------------------------+-----------------+--------+------------+

* Interface 4LUTs and DFFs are occupied due to the uses of LSRAM, Math, and uSRAM.
  Number of interface 4LUTs/DFFs = (36 * #.LSRAM) + (36 * #.Math) + (12 * #.uSRAM) = (36 * 0) + (36 * 6) +
(12 * 0) = 216.
```

The demo design we want to integrate this block into has a required clock period of 6.734 ns.
This means the synthesized period of the Alpha Blending block must be at most 6.734 ns.

Clock Domain	Required Period (ns)	Required Frequency (MHz)
CCC_0/PF_CCC_C1_0/PF_CCC_C3_0/pll_inst_0/OUT0	6.734	148.500

We can see from section 2 of summary.result.rpt that the minimum period for the synthesized
block is 3.070ns, which is below the threshold. This means we can safely integrate this block

48

into the demo design and meet timing.

We can compare the resources utilization to the alpha blending SolutionCore IP user guide which is shown in Table 2.

Table 2: PolarFire® Fabric Resource utilization of Alpha Blending SolutionCore

Resource	Usage
DFFs	242
4-Input LUTs	273
MACC	6
RAM1Kx18	0
RAM64x18	0

How can we reduce the resources used by this SmartHLS design?

We can start by opening the Libero SoC project created by SmartHLS to confirm the usage.

In the SmartHLS Project Explorer, expand the "synthesis" folder and the "alpha_blend" subfolder:
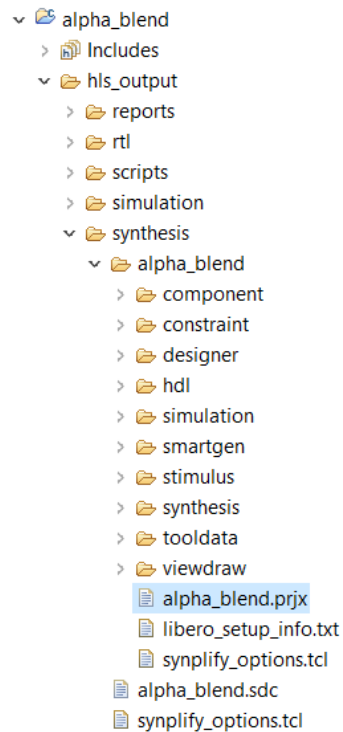
When you double click on "alpha_blend.prjx" in the Project Explorer, the contents will show in the text editor. Instead we can change the file association in SmartHLS to associate ".prjx" project files with Libero® SoC.

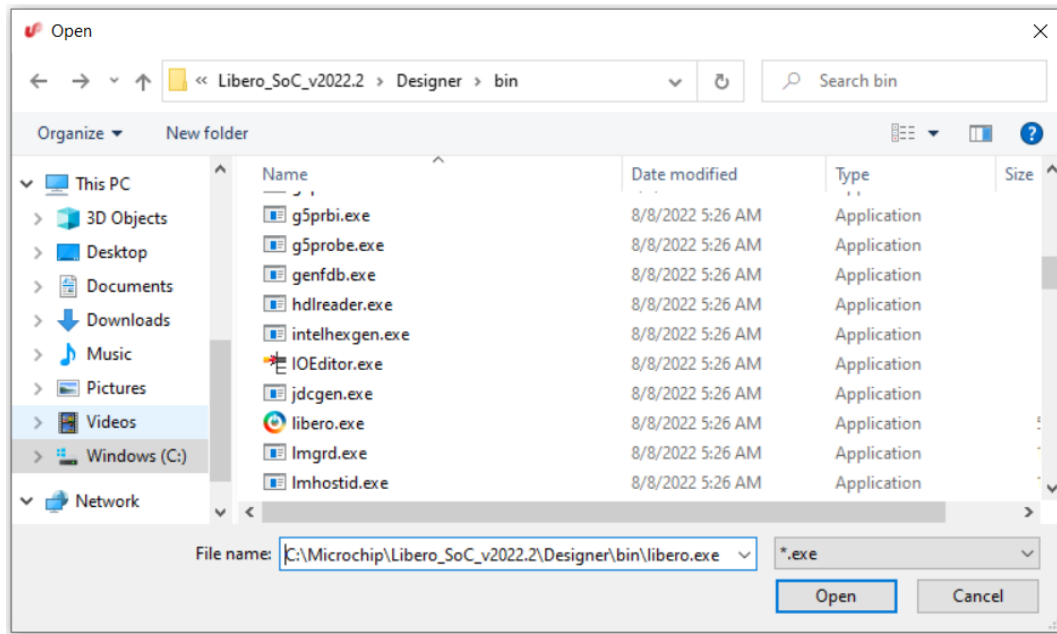Right click on "alpha_blend.prjx" and select Open With -> Other:
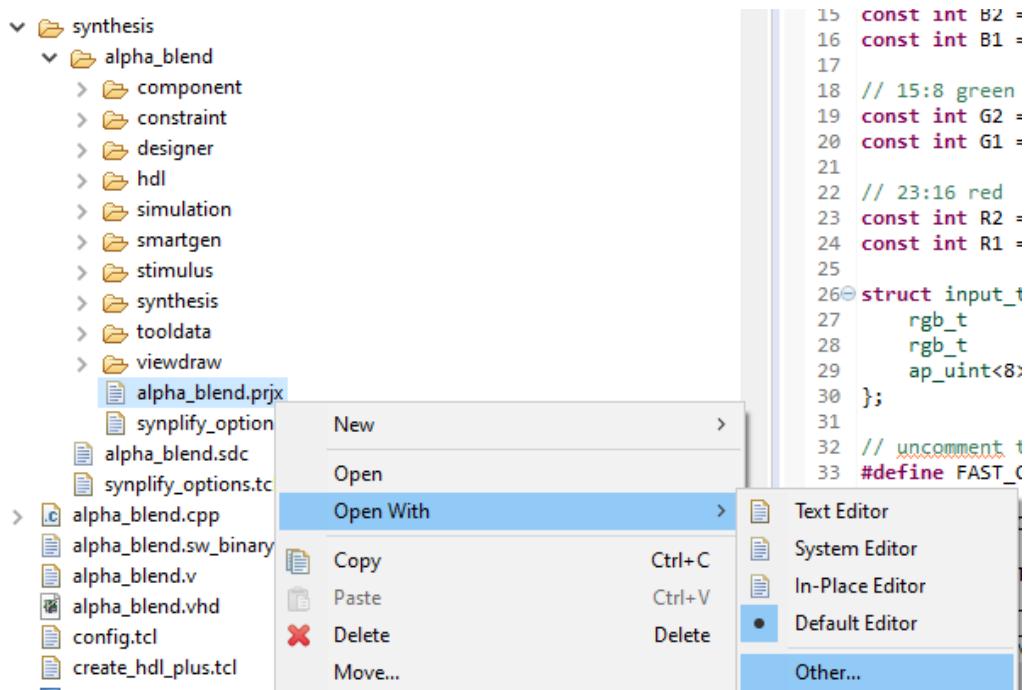
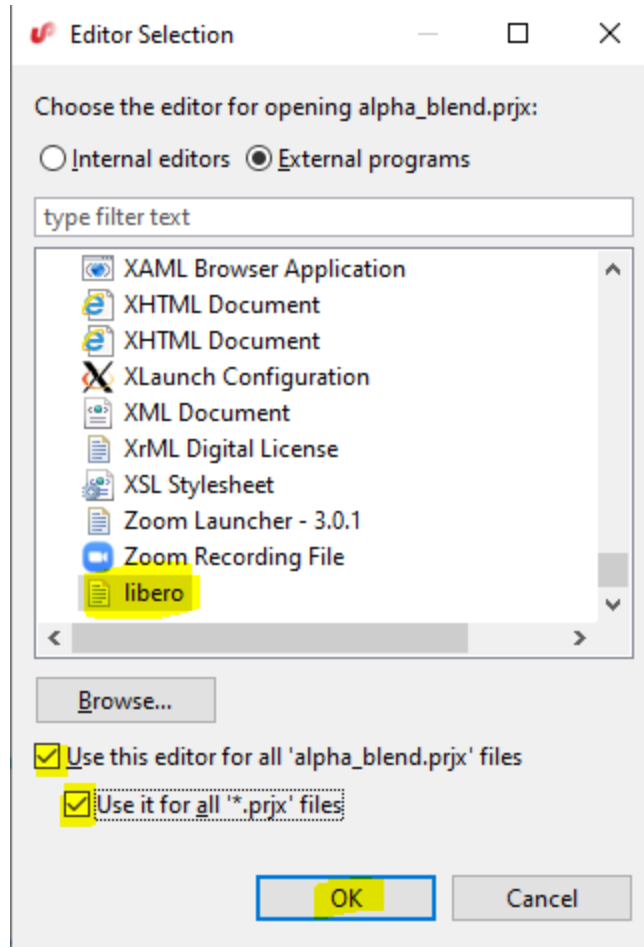In the Editor Selection pop-up, select "External programs" and click Browse.
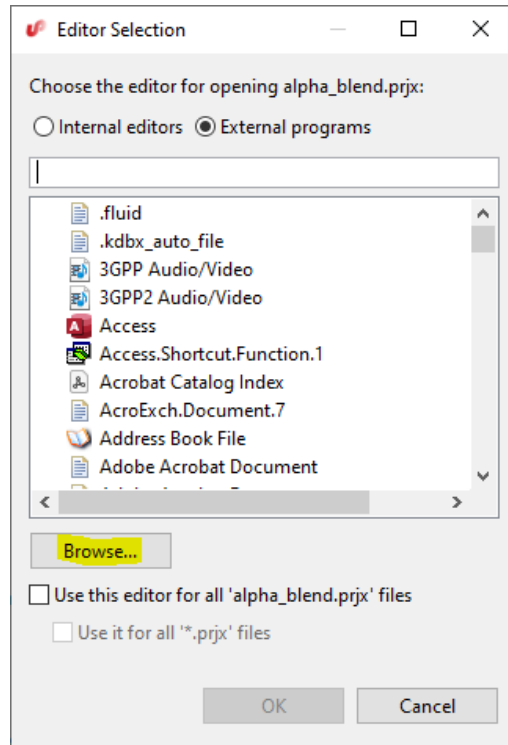
Now navigate to your Libero.exe, for example:

    C:\Microchip\Libero_SoC_2024.1\Designer\bin\libero.exe

Click OK.

Then you will see the "libero" external program has been added. Make sure "libero" is selected. Then select "Use this editor for all 'alpha_blend.prjx' files and select "Use it for all '*.prjx' files. Click OK.

You have now associated all *.prjx files in the SmartHLS Project Explorer with Libero® SoC.

In Libero® SoC go to Design -> Reports and open the
alpha_blend_smarthls_top_compile_netlist_hier_resources.csv report:

In the resource report we notice that SmartHLS is including the "Interface 4LUTs" when reporting 388 4LUTs (172 Fabric 4LUTs + 216 Interface 4LUTs). SmartHLS is also including "Interface DFFs" when reporting 277 DFFs (61 Fabric DFFs + 216 Interface DFFs). On PolarFire FPGAs, "Interface" 4LUTs/DFFs are only required by DSP blocks and RAM blocks used in the

design. All normal user logic is implemented in "Fabric" 4LUTs/DFFs.

Meanwhile the SolutionCore IP block is only reporting the Fabric 4LUTs (273) and Fabric DFFs (242) in Table 2. We want an apples-to-apples comparison, so we can update our comparison between SmartHLS and the SolutionCore to only consider Fabric resources in Table 3.

Table 3: Comparison of Fabric 4LUTs / Fabric DFFs (without MACC Interface LUTs/DFFs)

	SmartHLS Alpha Blend	SolutionCore Alpha Blend
Fabric 4LUTs	172	273
Fabric DFFs	61	242

9.6 SmartHLS Design Complexity vs SolutionCore RTL

We can now compare the complexity of the original alpha blend SolutionCore Verilog design and the SmartHLS C++ design. We have included the reference RTL code for the Alpha Blending SolutionCore IP. In the Project explorer expand the "rtl_solutioncore" folder (by clicking ⌄). Open the top-level RTL file of the SolutionCore IP in Alpha_Blending.v:
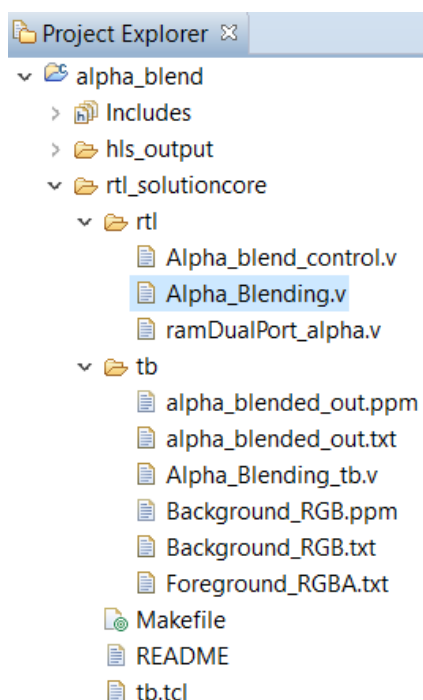
The RTL has the following top-level module interface:

```
module Alpha_Blending #(parameter g_V1_DATAWIDTH = 32,
                         parameter g_V2_DATAWIDTH = 24,
                         parameter g_OUTPUT_DATAWIDTH = 24)
                       (input SYS_CLK_I,
                        input RESET_n_I,
                        input[g_V1_DATAWIDTH - 1 : 0] V1_RDATA_i,
                        input[g_V2_DATAWIDTH - 1 : 0] V2_RDATA_i,
                        input[(g_V1_DATAWIDTH / 4) - 1 : 0] AG_i,
                        input Valid_i,
                        input Start_Alpha_blend_i,
                        output reg[(g_OUTPUT_DATAWIDTH - 1) : 0] Vout_o,
                                        output reg Vout_valid_o);
```

You will notice this RTL file is ~300 lines and the algorithm implemented is difficult to understand from the Verilog source. You can also check out the testbench file (tb/Alpha_Blending_tb.v) which is ~500 lines. There is another helper RTL file (rtl/Alpha_blend_control.v) which is ~900 lines. Now close these files.
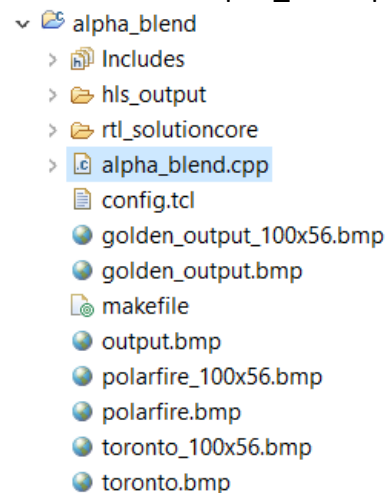
The ~200 lines of C++ in the SmartHLS project contains an equivalent design of the Alpha Blend block in SmartHLS which also includes the testbench. We can see that the implementation in the SmartHLS is much shorter and the design details are much easier to understand just by looking at the source code.

## 9.7   Integrating Alpha Blending SmartHLS Block to SmartDesign

In this section, we are going to take the SmartHLS-generated Alpha Blend block and import the IP component into SmartDesign. This will showcase the design flow for integrating SmartHLS generated Verilog Cores into Libero® SoC SmartDesign.

1. Open the alpha_blend.cpp source file in the alpha_blend project in the Project Explorer.

```
∨ 🗁 alpha_blend
    > 🔊 Includes
    > 🗁 hls_output
    > 🗁 rtl_solutioncore
    > 🖹 alpha_blend.cpp
      🖹 config.tcl
      🌐 golden_output_100x56.bmp
      🌐 golden_output.bmp
      🗎 makefile
      🌐 output.bmp
      🌐 polarfire_100x56.bmp
      🌐 polarfire.bmp
      🌐 toronto_100x56.bmp
      🌐 toronto.bmp
```

2. Click the "Compile Software to Hardware" button ![icon] on the top toolbar.

3. Launch Libero SoC 2024.1 and open the project:
   "Training1/Libero/Libero_training1/Libero_training1.prjx". The .prjx file was generated previously following the instruction: https://github.com/MicrochipTech/fpga-hls-examples/tree/main/Training1/Libero.

4. Navigate to the Design Hierarchy and search for "alpha_blend". Right click the alpha_blend_top design component and select Remove Core Definition, then right click again and Unlink. We want to avoid any duplicate blocks when importing the new alpha_blend_top HDL+ block from SmartHLS.
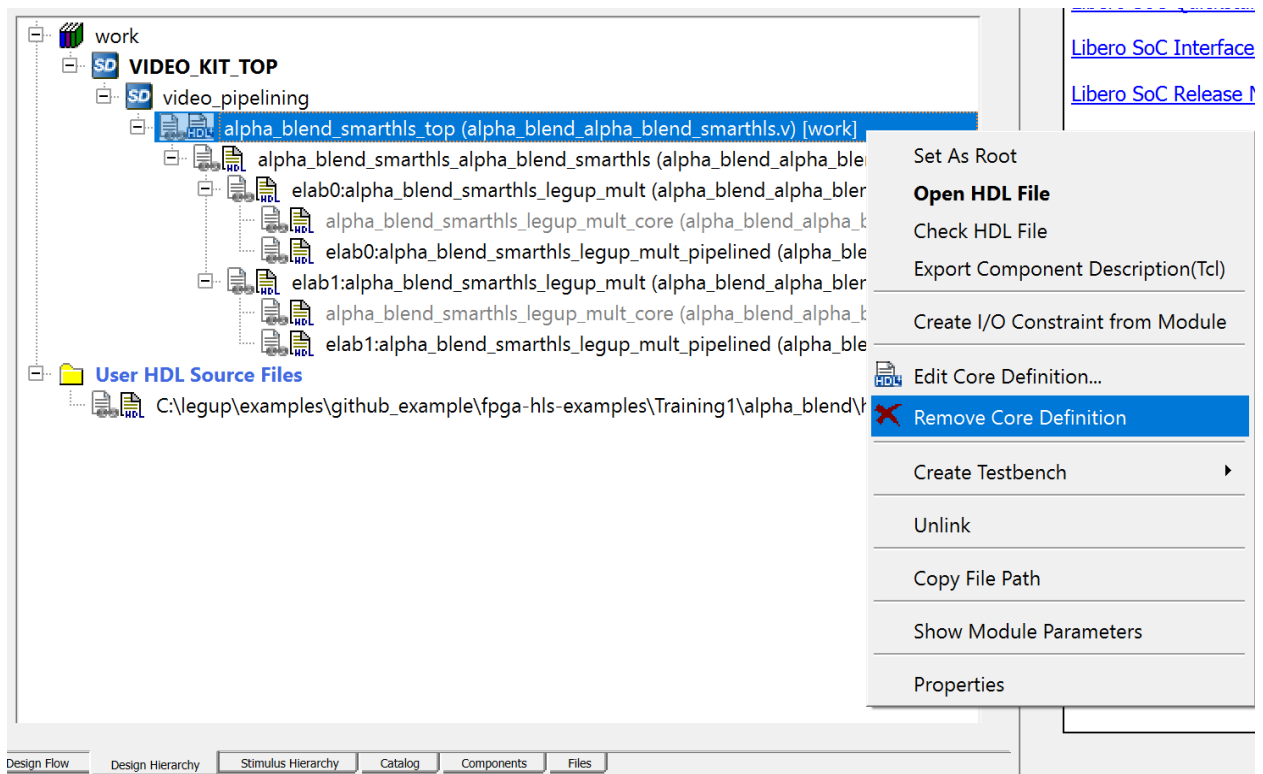
5. Without clearing the search, double click the video_pipelining SmartDesign file to open the video_pipelining SmartDesign Canvas.
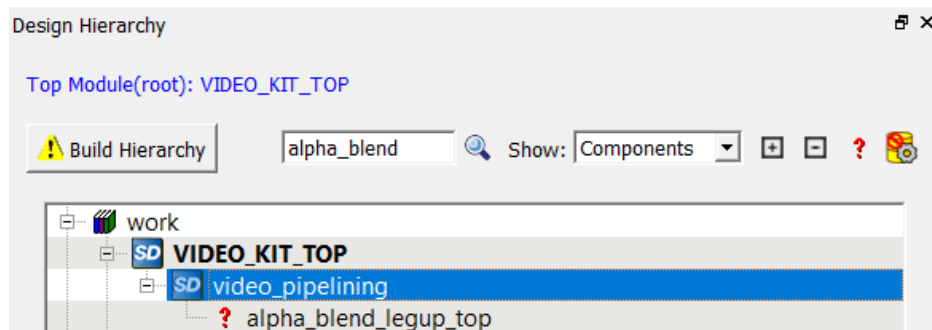
6. Find the alpha_blend_top module which should now be red.

alpha_blend_smarthls_top_0

alpha_blend_smarthls_top

7. On the top toolbar, click Project->Execute Script... and run the create_hdl_plus.tcl file in the alpha_blend\hls_output\scripts\ SmartHLS project directory. SmartDesign will open a report window when it finishes. Make sure the script executed successfully and close the report window.

The tcl script may replace the component and the module will no longer be red.
If the block is still red, follow step 8.

8. Right click the red alpha_blend_top_0 block and select Replace Component… to replace the block with the newly imported alpha_blend_top.
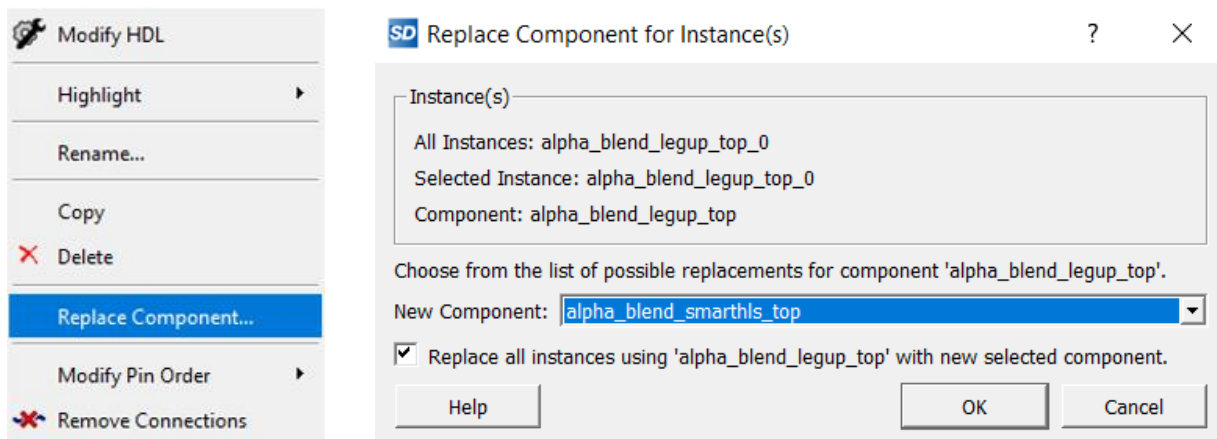
If this component is not in the list, you can instantiate it manually from the Design Hierarchy:
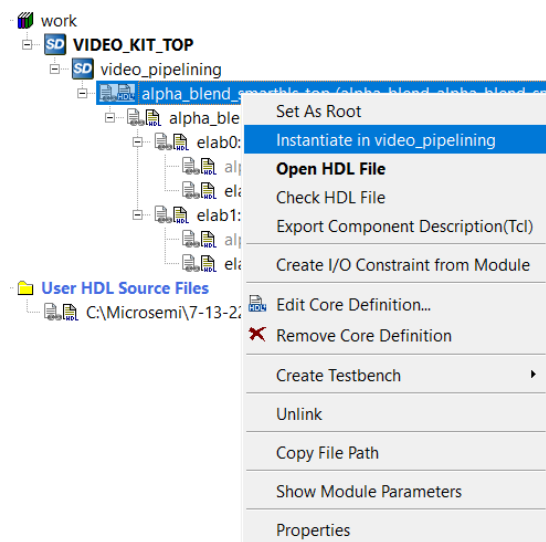
And then connect it manually to replace the red module.

9. Click the "Generate Component" ( ) button in the SmartDesign toolbar for video_pipelining and its parent component VIDEO_KIT_TOP.

10. The alpha_blend block has now been integrated and the project is ready for synthesis, place, and route. We skip this step for now since this will take 1-2 hours.

Now close Libero® SoC and all the files opened for this project in SmartHLS.

10   SmartHLS Optimization Concepts: Pipelining

10.1 SmartHLS Pipelining Background

Pipelining is a common HLS optimization used to increase hardware throughput and to better utilize FPGA hardware resources. We also covered the concept of loop pipelining in the SmartHLS Sobel Filter Tutorial. In Figure 19a) shows a loop to be scheduled with 3 single-cycle operations: Load, Comp, Store. We show a comparison of the cycle-by-cycle operations when hardware operations in a loop are implemented b) sequentially (default) or c) pipelined (with SmartHLS "pipeline" pragma). The sequential schedule takes 9 cycles to finish and in many cycles the hardware resources that perform "Comp" are idle. In the pipeline schedule, the circuit can finish in 5 cycles and starts a new load every clock cycle. On cycle 3, the pipelined circuit is executing a Load, Comp, and Store from three different loop iterations in parallel, fully utilizing the FPGA hardware resources.

```
for(i=0; i<2; i++) {
    load_data();
    compute();
    store_data();
}
```

(a) Loop nest to be scheduled

(b) Sequential Schedule

(c) Pipelined Schedule
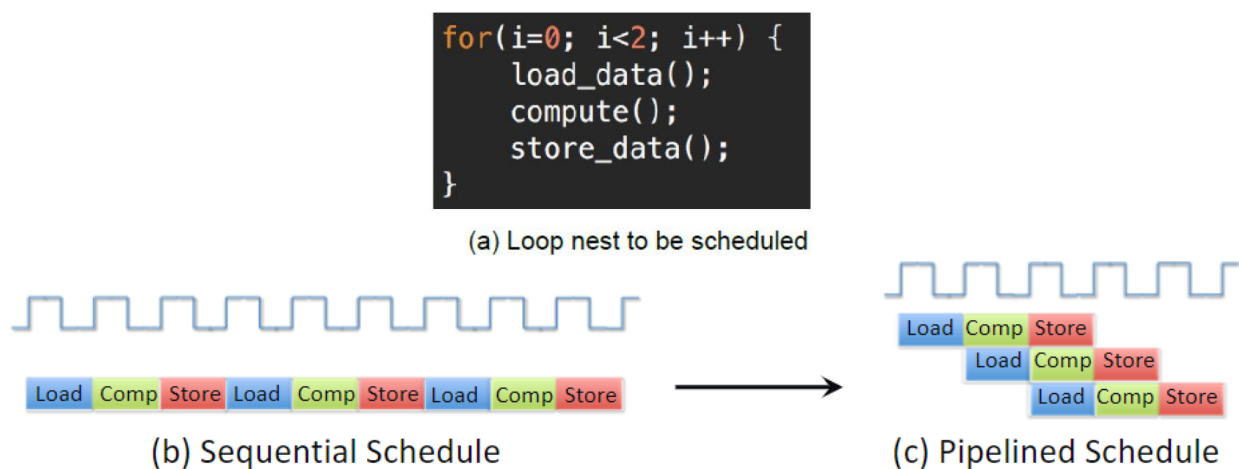
Figure 19: Comparing sequential versus pipelined hardware operations.

When pipelining, SmartHLS will automatically analyze dependencies and partition operations into pipeline stages to minimize the **initiation interval**. The initiation interval specifies how many cycles are needed between inputs to the pipeline. We typically always want to achieve an initiation interval of 1, meaning we can feed a new input into the pipeline every clock cycle.

Loop pipelining can be achieved in SmartHLS with the loop pipeline pragma or the function pipeline pragma:

```
#pragma HLS loop pipeline
#pragma HLS function pipeline
```

Loop pipelining only applies to a specific loop in a C++ function. Meanwhile, function pipelining is applied to an entire C++ function and SmartHLS will automatically unrolls all loops in that function.

10.2 SmartHLS Pipelining Hazards: Why Initiation Interval Cannot Always Be 1

In some cases, a pipeline initiation interval of 1 cannot be achieved by SmartHLS. This can happen when there are cross-iteration dependencies or resource contentions. To showcase these cases, we have included the project pipeline_hazards which includes SmartHLS source code with three examples of pipelines where the initiation interval cannot be 1.

In the Project Explorer tab, click the project pipeline_hazards and open pipeline_hazards.cpp.

```
v 🔩 pipeline_hazards
   >  📲 Includes
   >  📄 pipeline_hazards.cpp
      📄 config.tcl
      📄 makefile
```
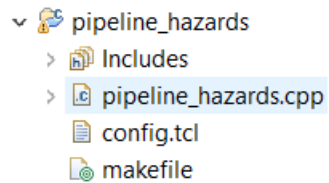
There are three functions in this file showcasing three examples of pipelines where the II is greater than 1. Before we look at the functions, compile the project to hardware 🔳 to verify that the pipelines generated have II greater than 1. Near the bottom of the Console output, you should find the following:

```
Info: Generating pipeline for loop on line 10 of pipeline_hazards.cpp with
label "for_loop_pipeline_hazards_cpp_10_2".
      Pipeline initiation interval = 3.
Info: Generating pipeline for loop on line 18 of pipeline_hazards.cpp with
label "for_loop_pipeline_hazards_cpp_18_2".
      Pipeline initiation interval = 2.
Info: Generating pipeline for loop on line 28 of pipeline_hazards.cpp with
label "for_loop_pipeline_hazards_cpp_28_2".
      Pipeline initiation interval = 2.
```

SmartHLS prints out pipelining information for each loop in the Console. This confirms that the three pipelines in the three examples have II greater than 1.

SmartHLS also prints this information to the summary.hls.main.rpt file found in the reports directory.

```
v 📂 hls_output
   v 📂 reports
      >  📂 dot_graphs
         📄 hls.log
         📄 pipelining.hls.rpt
         📄 scheduling.hls.rpt
         📄 summary.hls.main.rpt
```
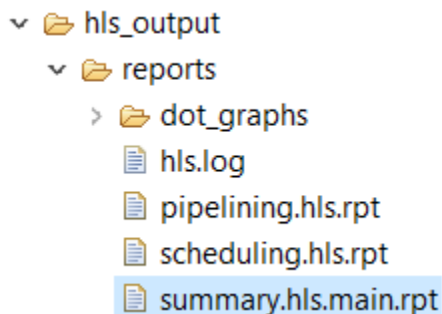
Double click summary.hls.main.rpt to open it and then scroll down to section 2: Function and Loop Scheduling Results. Scroll to the right to see the same loop pipelining information. Notice there is more information here than in the Console output, such as the pipeline length. Now close the file.

```
====== 2. Function and Loop Scheduling Results ======

+---------------------------------------------------------+
| Function: main takes 106 cycles                         |
+----------------------------------+…+----+---------------+
| Loop                             |…| II | Total Latency |
+----------------------------------+…+----+---------------+
| for.loop:pipeline_hazards.cpp:10:2 |…| 3 | 28          |
| for.loop:pipeline_hazards.cpp:18:2 |…| 2 | 22          |
| for.loop:pipeline_hazards.cpp:28:2 |…| 2 | 19          |
+----------------------------------+…+----+---------------+
```

10.3 SmartHLS Pipelining Hazards: Cross-Iteration Dependencies

Back in pipeline_hazards.cpp, scroll to line 7 and look at the cross_iteration_dependency() function. This shows an example of a cross-iteration dependency in a C++ loop that will prevent an initiation interval of 1. In the loop body we store to an array element, array[i+1], that will be loaded in the next loop iteration. But we cannot compute array[i+1] in the current iteration before the previous iteration is done computing array[i]. Therefore, there is a *recurrence* where the current loop iteration is waiting for the previous iteration, but the next iteration is also waiting for the current iteration. We cannot parallelize any operations along this recurrence, so we need to wait 1 cycle for the load, 1 cycle for the multiply, and 1 cycle for the store before starting every loop iteration. Therefore, the pipeline initiation interval is 3 cycles (1 + 1 + 1). A diagram of how the pipeline schedule would look is presented in Figure 20.

```cpp
void cross_iteration_dependency( volatile int array[N] ) {
#pragma HLS loop unroll factor(1)
#pragma HLS loop pipeline
    for (int i = 0; i < N - 1; i++) {
        array[i + 1] = array[i] * coeff1;
    }
}
```
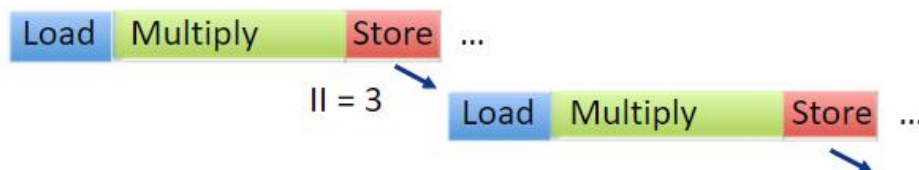

Figure 20: Example of initiation interval of 3 due to cross-iteration dependency.

In the Console output, find the messages generated from compiling the project to hardware in the previous step. Near the bottom of the Console there is the following output. You might need to scroll up a bit to see it.

```
Info: Cross-iteration dependency does not allow initiation interval (II) of 1.
    Dependency (distance = 1) from 'store' operation for array 'array' (at line 11 of pipeline_hazards.cpp) to
'load' (32b) operation for array 'array' (at line 11 of pipeline_hazards.cpp)
    Recurrence path:
    +---------------------------------------+------------------------------+---------------+-------------+
    | Operation                             | Location                     | Cycle Latency | Delay [ns] |
    +---------------------------------------+------------------------------+---------------+-------------+
    | 'load' (32b) operation for array 'array' | line 11 of pipeline_hazards.cpp | 1          | 0.00       |
    | 'mul' (32b) operation                 | line 11 of pipeline_hazards.cpp | 1          | 3.70       |
    | 'store' operation for array 'array'   | line 11 of pipeline_hazards.cpp | 1          | 0.00       |
    +---------------------------------------+------------------------------+---------------+-------------+
    |                                       | Total                        | 3             | 3.70       |
    +---------------------------------------+------------------------------+---------------+-------------+


    Total required latency = 3. Maximum allowed latency = distance x II = 1 x 1 = 1.
    Total required latency > Maximum allowed latency, we must increase II.
```
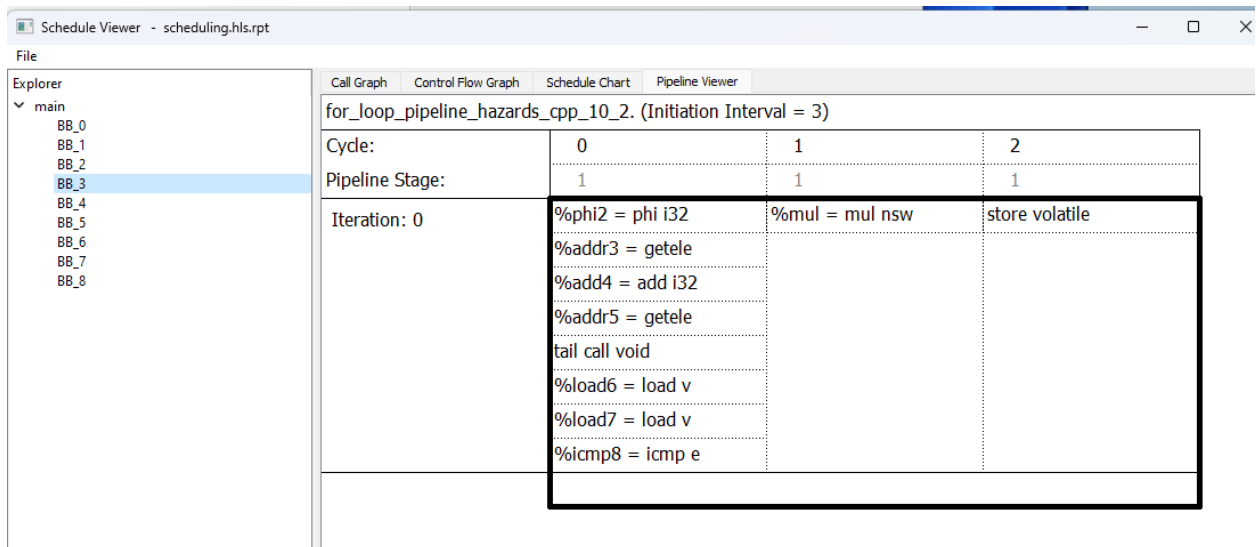
SmartHLS automatically prints out a table specifying which instructions are causing a cross-iteration dependency or resource contention when a pipeline fails to achieve initiation interval of 1. This table shows the instructions in the pipeline that caused a recurrence. The first instruction (load) depends on the last instruction (store) finishing in the previous iteration before it can start.

You can open the schedule viewer and click on "BB_3" in the Explorer on the left-hand side to see the cross_iteration_dependency() loop pipeline schedule:

The pipeline steady state is highlighted in black, there is no actual pipeline parallelism

(overlapping iterations).

10.4 SmartHLS Pipelining Hazards: Resource Contentions

Now scroll to line 15 and look at the function functional_unit_contention(). This example shows a C++ loop which is an example of resource contention assuming we have specified a SmartHLS user constraint to only generate one multiplier in hardware. The loop contains two multiply operations, but in hardware we can only perform one multiply operation per cycle. The first loop iteration must use the multiplier for two cycles. Therefore, we cannot start the next loop iteration (next input) until two cycles later. The pipeline initiation interval must be 2 due to resource contention on the single multiplier. In the schedule of Figure 21, there is only one multiply operation in any clock cycle (column). A diagram of how the pipeline would look like is presented in Figure 21.

```
Void functional_unit_contention( volatile int array[N] ) {
#pragma HLS loop unroll factor(1)
#pragma HLS loop pipeline
    for (int I = 0; i < N; i++) {
        int mult1 = coeff1 * coeff1;
        int mult2 = coeff2 * coeff2;
        array[i] = mult1 + mult2;
    }
}
```


Figure 21: Example of functional unit contention in a loop pipeline

In the Console output, find the messages about resource constraints generated for this pipeline. This should be above the messages generated for the pipeline in the previous example.

```
Info: Resource constraint limits initiation interval to 2.
      Resource 'signed_multiply_32' has 2 uses per cycle but only 1 units available.
      +----------------------+-------------------------------+--------------------+
      | Operation            | Location                      | Competing Use Count |
      +----------------------+-------------------------------+--------------------+
      | 'mul' (32b) operation | line 19 of pipeline_hazards.cpp | 1                  |
      | 'mul' (32b) operation | line 20 of pipeline_hazards.cpp | 2                  |
      +----------------------+-------------------------------+--------------------+
      |                      | Total # of Competing Uses     | 2                  |
      +----------------------+-------------------------------+--------------------+
```

This table shows the operations that caused resource contention in the pipeline. SmartHLS

mentions that there are 2 uses of the functional unit "signed_multiply_32" but only one unit available.

You can open the Schedule Viewer and click on "BB_5" in the Explorer on the left-hand side to see the functional_unit_contention() loop pipeline schedule:
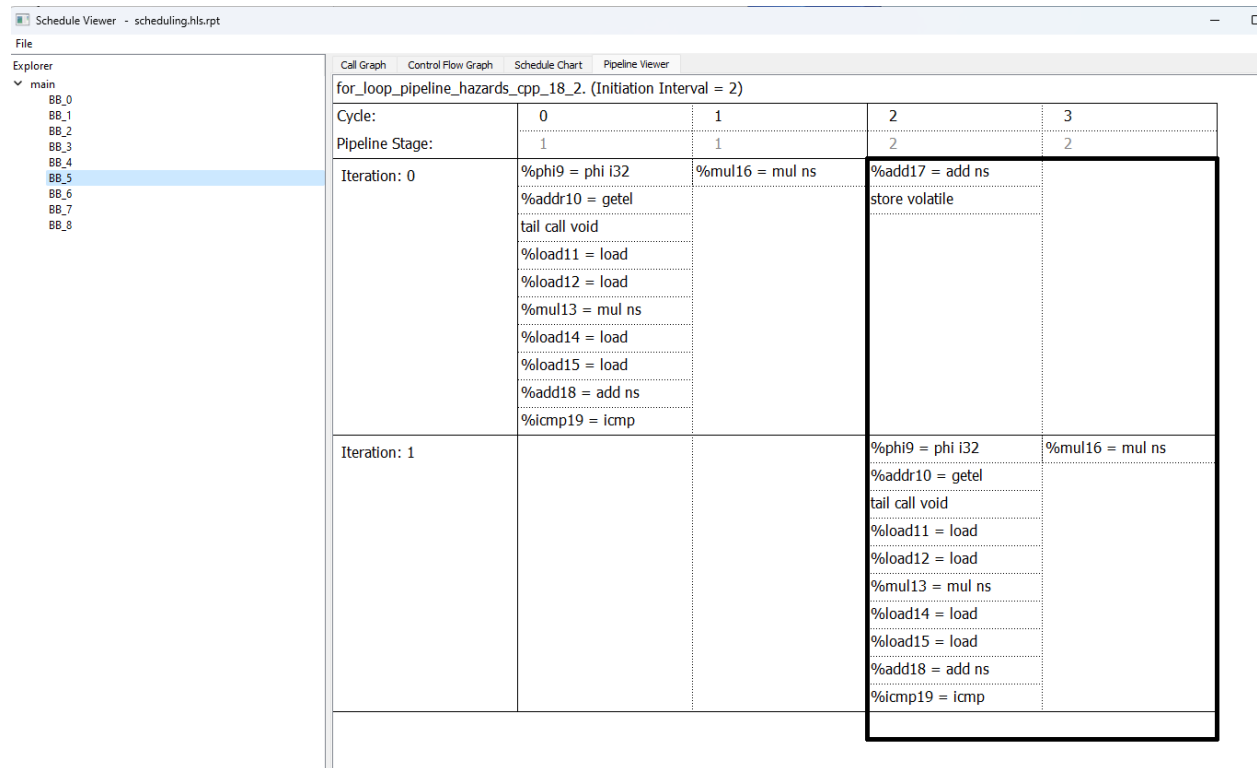
The pipeline steady state is highlighted in black. In the column for cycle 2, one multiply operation occurs (%mul.i1 = mul) and in the column for cycle 3 another multiply operation occurs (%mul1.i = mul), showing the resource contention.

Now scroll to line 25 and look at the function memory_contention(). This example also shows a C++ loop which is an example of resource contention. The loop contains two loads and one store to the same memory per iteration, but only two read/write ports exist on each RAM in hardware. The first loop iteration must use the RAM ports for two cycles. Therefore, we cannot start the next loop iteration (next input) until two cycles later. The pipeline initiation interval must be 2 due to resource contention on the read/write ports. In the schedule of Figure 22 there is only one iteration performing memory operation in any clock cycle (column).

```
void memory_contention( volatile int array[N] ) {
#pragma HLS loop unroll factor(1)
#pragma HLS loop pipeline
        for (int i = 0; i < N - 1; i++) {
                array[i] = array[i] + array[i + 1];
```

64

```
        }
    }
```
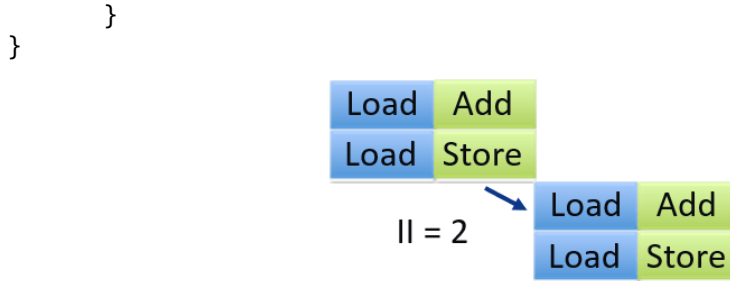

Figure 22: Example of memory contention in a loop pipeline. Two loads happen in the first cycle then an add and store happens in the second cycle.

This kind of memory port data contention can happen independently on all memories used in a pipeline. The memory with the largest number of uses will then dictate the II of the entire pipeline.

In the Console output, find the messages about resource constraints generated for this pipeline. This should be above the messages generated for the pipeline in the previous example.

```
Info: Resource constraint limits initiation interval to 2.
      Resource '@main_entry_array@_local_memory_port' has 3 uses per cycle but only 2
units available.
    +----------------------------------------+--------------------------------+--------------------+
    | Operation                              | Location                       | Competing Use Count |
    +----------------------------------------+--------------------------------+--------------------+
    | 'load' (32b) operation for array 'array' | line 29 of pipeline_hazards.cpp | 1                  |
    | 'load' (32b) operation for array 'array' | line 29 of pipeline_hazards.cpp | 2                  |
    | 'store' operation for array 'array'      | line 29 of pipeline_hazards.cpp | 3                  |
    +----------------------------------------+--------------------------------+--------------------+
    |                                        | Total # of Competing Uses      | 3                  |
    +----------------------------------------+--------------------------------+--------------------+
```

This table shows the operations that caused resource contention in the pipeline. SmartHLS mentions that there are 3 accesses to the memory "@main_entry_array@_local_memory_port" per iteration but only two ports available.

Now close the pipeline_hazards.cpp source file.

You can open the Schedule Viewer 🌀 and click on "BB_7" in the Explorer on the left-hand side to see the memory_contention() loop pipeline schedule:

The pipeline steady state is highlighted in black. In the column for cycle 0, two loads occur in parallel (dual-port memory), and in the column for cycle 1 a single store occurs.

Now close the Schedule Viewer.

The initiation interval (II) is the key metric we use to understand the performance of a pipeline. Again, we typically always aim to have an II of 1, sometimes even at the cost of Fmax, because in general this gives the generated circuit higher throughput.

11   Color Space Conversion Blocks

The color space conversion blocks convert an image from RGB (red, green, blue) color space to the YCbCr color space. Y is the luma (brightness) component and Cb and Cr are the blue-difference and red-difference chroma (color) components.

You can see an example in Figure 23, which shows the original image (top) decomposed into RGB (middle) and YCbCr (bottom) color spaces. Notice that the Y luma component (bottom left) is a grayscale version of the original image. We can use the 8-bit Y luma value as the input to the Canny Edge detection filter which works on grayscale images. The Cb and Cr images are the magnitudes of the blue-difference and the red-difference, which is represented in grayscale (white is higher intensity, black is lower intensity). For example, in the Cr image (bottom right) the moon is white indicating the red color.

Figure 23: Original image (top) decomposed into RGB (middle) and YCbCr (bottom) color spaces.

There are two hardware blocks for converting between the RGB and YCbCr color spaces: the *RGB2YCbCr* Block and the *YCbCr2RGB* block.

The computations required for converting between color spaces for 24-bit RGB and 24-bit YCbCr values (each of the three components is 8-bits) are given in Equation 1 and Equation 2 (from Wikipedia).

Equation 1: Conversion from RGB to YCbCr color space

$$Y' = \quad 16+ \quad \frac{65.738 \cdot R'_D}{256} + \quad \frac{129.057 \cdot G'_D}{256} + \quad \frac{25.064 \cdot B'_D}{256}$$

$$C_B = \quad 128- \quad \frac{37.945 \cdot R'_D}{256} - \quad \frac{74.494 \cdot G'_D}{256} + \quad \frac{112.439 \cdot B'_D}{256}$$

$$C_R = \quad 128+ \quad \frac{112.439 \cdot R'_D}{256} - \quad \frac{94.154 \cdot G'_D}{256} - \quad \frac{18.285 \cdot B'_D}{256}$$

Equation 2: conversion from YCbCr to RGB color space

$$R'_D = \quad \frac{298.082 \cdot Y'}{256} \quad\quad + \quad \frac{408.583 \cdot C_R}{256} - \quad 222.921$$

$$G'_D = \quad \frac{298.082 \cdot Y'}{256} - \quad \frac{100.291 \cdot C_B}{256} - \quad \frac{208.120 \cdot C_R}{256} + \quad 135.576$$

$$B'_D = \quad \frac{298.082 \cdot Y'}{256} + \quad \frac{516.412 \cdot C_B}{256} \quad\quad - \quad 276.836$$

Similar to Alpha Blending, we designed the two SmartHLS blocks to have the same interface as the Color Conversion SolutionCore IPs, see UG0639 User Guide Color Space Conversion. Our goal is to use the SmartHLS generated SmartDesign IP component as a drop-in replacement for the previous SolutionCore blocks. The block diagram of the SolutionCore blocks are shown in and the input and output interface for the RGB2YCbCr is described in Table 1.
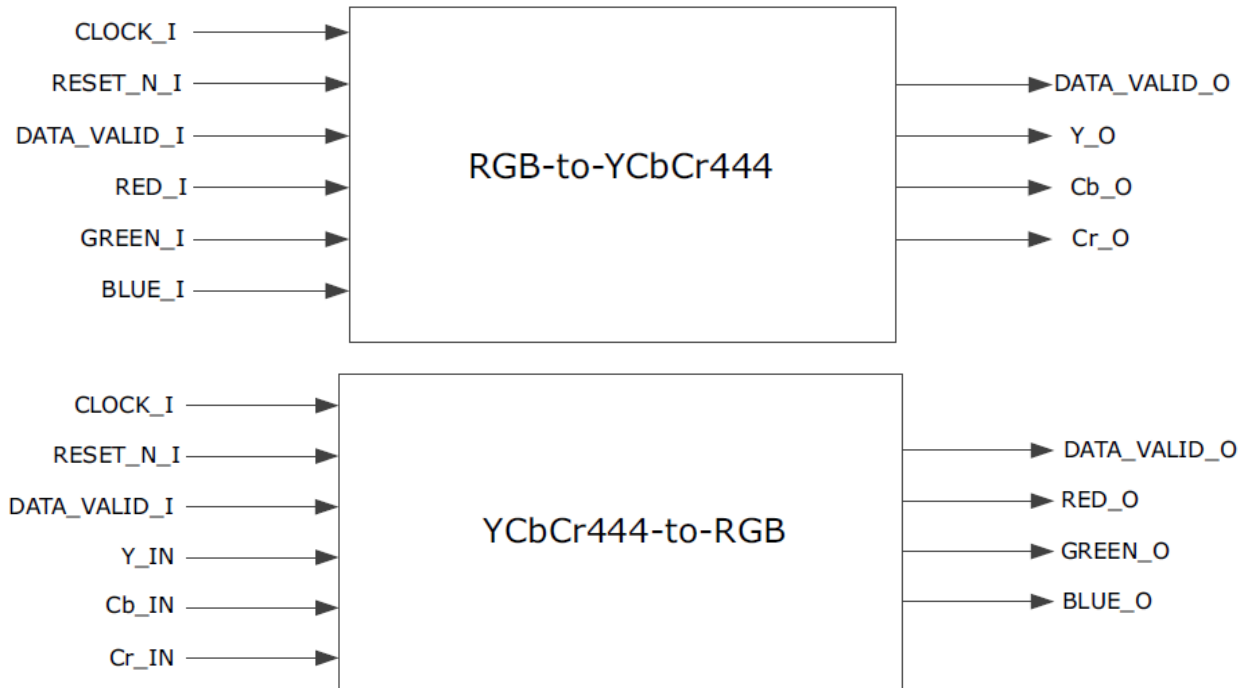
Table 4: RGB2YCbCr SolutionCore IP Interface

Signal Name	Direction	Width	Description
RESETN_I	Input	1-bit	Active low async reset
SYS_CLK_I	Input	1-bit	System Clock
RED_I	Input	8-bits	Red input pixel
GREEN_I	Input	8-bits	Green input pixel
BLUE_I	Input	8-bits	Blue input pixel
DATA_VALID_I	Input	1-bit	Input data valid
Y_OUT_O	Output	8-bits	Y luma output
Cb_OUT_O	Output	8-bits	Cb chroma output
Cr_OUT_O	Output	8-bits	Cr chroma output
DATA_VALID_O	Output	1-bit	Output data valid

The desired RTL interface splits up the input red, green, blue values into three separate 8-bit inputs sharing a data valid. In contrast to the Alpha Blend module which combined the RGB 8-bits values into a single 24-bit input.

11.1 RGB2YCbCr Block

In the SmartHLS project explorer, double click the "RGB2YCbCr" project and open up the RGB2YCbCr.cpp file.

~ 📂 RGB2YCbCr
    > 📘 Includes
    > 📂 rtl_solutioncore
    > 📄 RGB2YCbCr.cpp
    📄 config.tcl
    📄 makefile
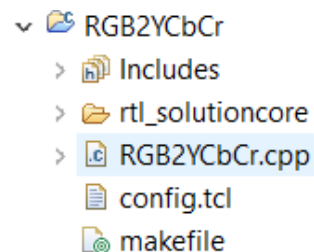
Now run SmartHLS Compile Software to Hardware (click the 🖥 button) and look at the summary.hls.RGB2YCbCr_smarthls.rpt in section 1 for the RTL interface:

```
+------------------------------------------------------------------------------------+
| RTL Interface Generated by SmartHLS                                                |
+-------------+---------------+-------------------+-------------------+----------------+
| C++ Name    | Interface Type | Signal Name      | Signal Bit-width  | Signal Direction |
+-------------+---------------+-------------------+-------------------+----------------+
|             | Clock & Reset | clk               | 1                 | input          |
|             |               | reset             | 1                 | input          |
+-------------+---------------+-------------------+-------------------+----------------+
|             | Control       | finish            | 1                 | output         |
|             |               | ready             | 1                 | output         |
|             |               | start             | 1                 | input          |
+-------------+---------------+-------------------+-------------------+----------------+
| input_fifo  | FIFO          | input_fifo_ready  | 1                 | output         |
|             |               | input_fifo_valid  | 1                 | input          |
|             |               | input_fifo_R      | 8                 | input          |
|             |               | input_fifo_B      | 8                 | input          |
|             |               | input_fifo_G      | 8                 | input          |
+-------------+---------------+-------------------+-------------------+----------------+
| output_fifo | FIFO          | output_fifo_ready | 1                 | input          |
|             |               | output_fifo_valid | 1                 | output         |
|             |               | output_fifo_Y     | 8                 | output         |
|             |               | output_fifo_Cb    | 8                 | output         |
|             |               | output_fifo_Cr    | 8                 | output         |
+-------------+---------------+-------------------+-------------------+----------------+
```

The SmartHLS generated top-level interface matches our desired RTL interface from Table 4.

Now go back to RGB2YCbCr.cpp and scroll down to the top-level function "RGB2YCbCr_smarthls" on line 27 to see the function signature that gets generated into the above interface. This function is also pipelined and has two arguments:

```cpp
void RGB2YCbCr_smarthls(hls::FIFO<RGB>   &input_fifo,
                        hls::FIFO<YCbCr> &output_fifo) {
#pragma HLS function top
#pragma HLS function pipeline
```

The input_fifo argument is of type hls::FIFO<RGB>. With the RGB type is defined above as struct with three 8-bit RGB values:

```cpp
const int RGB_BITWIDTH = 8;
struct RGB {
    ap_uint<RGB_BITWIDTH> R;
    ap_uint<RGB_BITWIDTH> G;
    ap_uint<RGB_BITWIDTH> B;
};
```

The output_fifo argument is of type hls::FIFO<YCbCr>. With the YCbCr type is defined above as struct with three 8-bit YCbCr values:

```cpp
const int YCBCR_BITWIDTH = 8;
struct YCbCr {
    ap_uint<YCBCR_BITWIDTH> Y;
    ap_uint<YCBCR_BITWIDTH> Cb;
    ap_uint<YCBCR_BITWIDTH> Cr;
};
```

70

Similar to Alpha Blending, when you use a struct inside a FIFO as a top-level function argument, SmartHLS will expose the elements of the struct, and all elements will share the same 1-bit valid/ready signals.

Now if we look in the body of the top-level function RGB2YCbCr, the line calculating the Y (luma) component corresponds to Equation 1:

```
ycbcr.Y = fixpt_t(16) +
        ((fixpt_t( 65.738)*in.R + fixpt_t(129.057)*in.G + fixpt_t( 25.064)*in.B)
         >> 8) + fixpt_t(0.5);
```

The right shift by 8 corresponds to the divide by 256 in Equation 1. The final addition of 0.5 is for rounding since C/C++ will always round down to the nearest integer.

For this computation we are using a 18-bit fixed-point type with 10 integer bits and 8 fractional bits (Q10.8) as defined below using the `ap_fixpt` SmartHLS arbitrary precision fixed-point data type (see SmartHLS documentation):

```
typedef ap_fixpt<18, 10> fixpt_t;
```

Now we will quickly simulate the design in software (© and ▶) to verify its functionality. You should see the following output in the Console meaning that the software simulation has passed:

```
Expected: Y=16 Cb=128 Cr=128
Actual: Y=16 Cb=128 Cr=128
Expected: Y=23 Cb=130 Cr=126
Actual: Y=23 Cb=130 Cr=126
Expected: Y=49 Cb=138 Cr=118
Actual: Y=49 Cb=138 Cr=118
Expected: Y=94 Cb=166 Cr=119
Actual: Y=94 Cb=166 Cr=119
Expected: Y=131 Cb=137 Cr=119
Actual: Y=131 Cb=137 Cr=119
PASS

22:43:07 Build Finished (took 29s.345ms)
```

Using SmartHLS fixed-point data types can improve productivity by avoiding error prone RTL code that requires the designer to manually keep track of the decimal place location after various operations. SmartHLS will handle the conversion between a floating-point initialization and the underlying fixed-point representation.

For example, we can print the fixed point representation of `fixpt_t(65.738)` by adding this code in the main function on line 104 after the test case validation loop:

```
std::cout << fixpt_t( 65.738).to_fixpt_string(10) << std::endl;
std::cout << "= " << fixpt_t( 65.738).to_double() << std::endl;
```

Now recompile (📋) and rerun (▶) the software. The Console will print out the fixed-point underlying 18-bit decimal value of 16,828 which represents right before it prints PASS:

```
16828 * 2^-8
= 65.7344
```

👆 By default, `ap_fixpt` will truncate bits to bring the result closer to negative infinity. If you add AP_RND to the `fixpt_t` typedef on line 25:

```
typedef ap_fixpt<18, 10, AP_RND> fixpt_t;
```

Then save, recompile and rerun software simulation. You will find the fixed-point representation will get closer to the desired 65.738 value:

```
16829 * 2^-8
= 65.7383
```

For this hardware block, more precise rounding is not necessary so remove this change and save.

👆 Undo the above changes, and run the Co-simulation to verify that the generated RTL is correct, you should see this output in the Console:

```
+-----------------------+----------------+-------------------------+...
| Top-Level Name        | Number of calls | Simulation time (cycles) |...
+-----------------------+----------------+-------------------------+...
| RGB2YCbCr_smarthls_top | 5             | 9                       |...
+-----------------------+----------------+-------------------------+...
Simulation time (cycles): 9
SW/HW co-simulation: PASS

15:54:48 Build Finished (took 16s.619ms)
```

The other columns and messages in the simulation log report other statistics, e.g. that the call latency ranges from 3-4 cycles and the initiation interval (II) is 1 cycle (in the first iteration, it is 2 cycles).

👆 We can also run co-simulation and look at the waveforms by choosing the "SW/HW Co-Simulation with Waveforms" option from the SmartHLS menu:
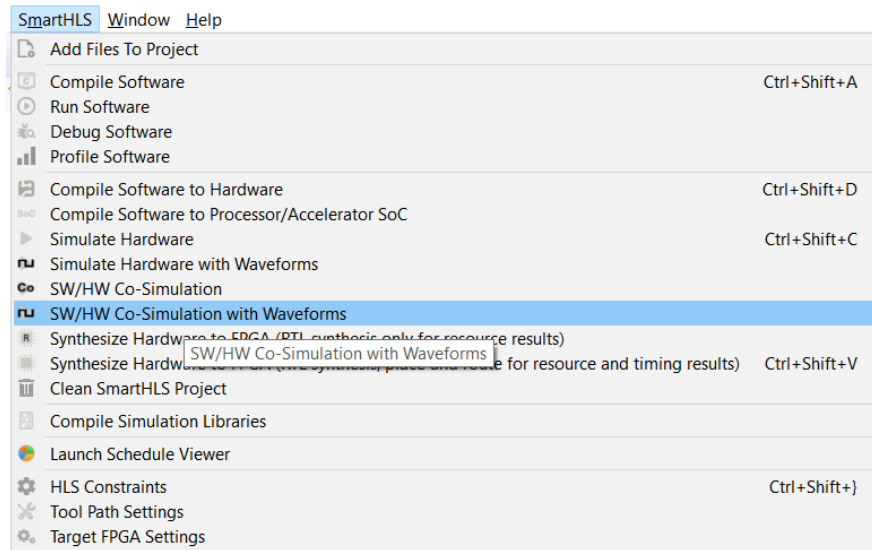
Make sure to say "No" when ModelSim prompts you to finish:
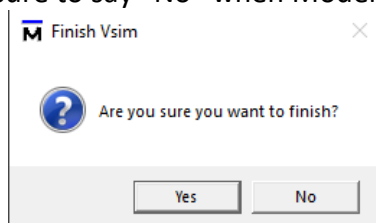
Expand the "RGB2YCbCr_smarthls_top_tb_inst" and look at the waveforms:
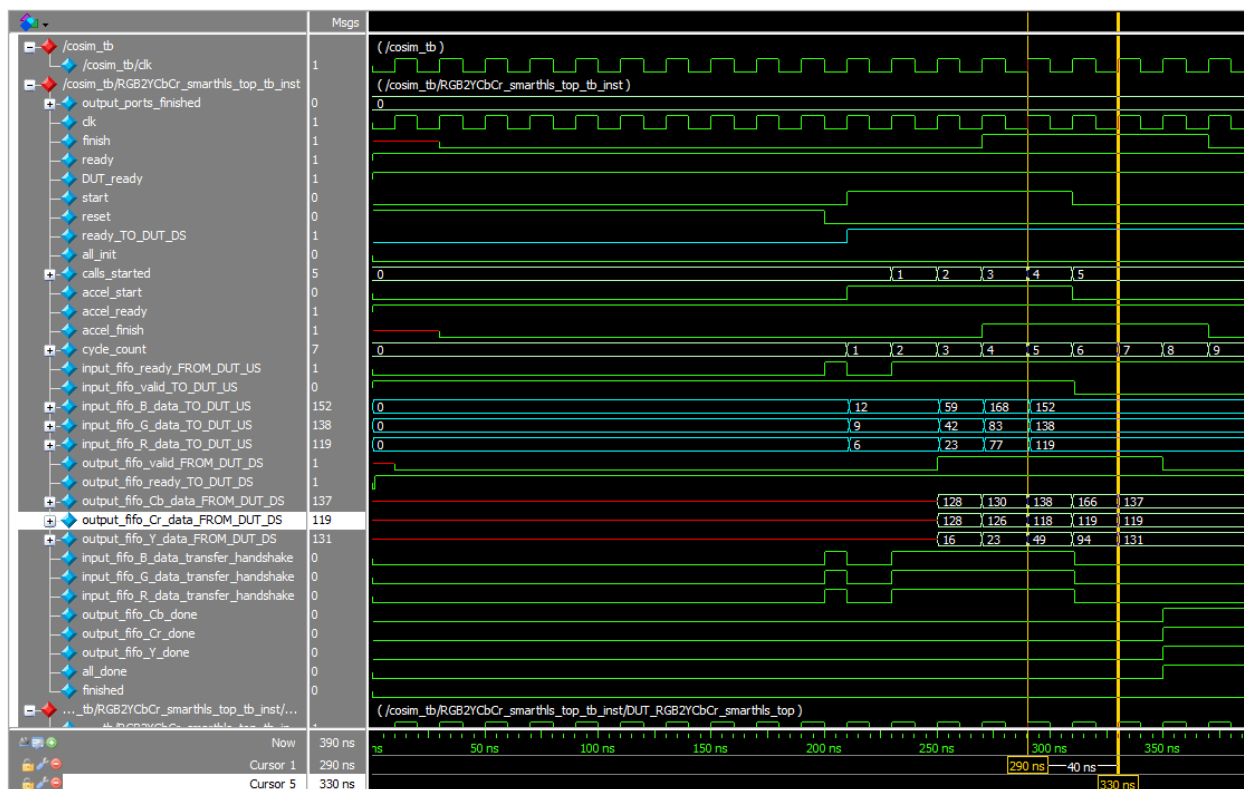
Figure 24: SW/HW Co-Simulation with Waveforms for RGB2YCbCr SmartHLS Core

We can look in the C++ main function for the input test vectors, for example on line 80, the 5th test input and expected output is given below:

```
// test 5
in.R = 119; in.G = 138; in.B = 152;
input_fifo.write(in);
expected.Y = 131; expected.Cb = 137; expected.Cr = 119;
expected_fifo.write(expected);
```

In the waveforms in Figure 24, the first cursor highlights when the 5th test vector is input to the design under test (DUT) on clock cycle 5 (see cycle_count signal). The correct output is received on clock cycle 7 as highlighted by the second cursor. Therefore, the hardware pipeline has a latency of 2 clock cycles (7 − 5 = 2). You can also see from the waveform that the hardware is receiving a new input every clock cycle, indicating a pipeline initiation interval of 1.

We can compare our observations to summary.hls.RGB2YCbCr_smarthls.rpt. Scrolling down to section 2, the latency reported is 4 cycles. The various reports may differ by 1-2 cycles due to handshaking start/finish signals with the instantiating module.

```
====== 2. Function and Loop Scheduling Results ======


+----------------------------------------------------+
| Function: RGB2YCbCr_smarthls takes 4 cycles II = 1 |
+----------------------------------------------------+
```

74

Now we can synthesize the design to target the PolarFire® FPGA device (click the button). This should take about 5 minutes. We check the summary.results.rpt report file afterwards:

```
====== 2. Timing Result of HLS-generated IP Core (top-level module: RGB2YCbCr_smarthls_top) ======

+--------------+---------------+-------------+-------------+----------+-------------+
| Clock Domain | Target Period | Target Fmax | Worst Slack | Period   | Fmax        |
+--------------+---------------+-------------+-------------+----------+-------------+
| clk          | 10.000 ns     | 100.000 MHz | 7.229 ns    | 2.771 ns | 360.881 MHz |
+--------------+---------------+-------------+-------------+----------+-------------+

The reported Fmax is for the HLS core in isolation (from Libero's post-place-and-route timing analysis).
When the HLS core is integrated into a larger system, the system Fmax may be lower depending on the
critical path of the system.

====== 3. Resource Usage of HLS-generated IP Core (top-level module: RGB2YCbCr_smarthls_top) ======

+-------------------------+-----------------+--------+------------+
| Resource Type           | Used            | Total  | Percentage |
+-------------------------+-----------------+--------+------------+
| Fabric + Interface 4LUT*| 489 + 144 = 633 | 299544 | 0.21       |
| Fabric + Interface DFF* | 124 + 144 = 268 | 299544 | 0.09       |
| I/O Register            | 0               | 1536   | 0.00       |
| User I/O                | 0               | 512    | 0.00       |
| uSRAM                   | 0               | 2772   | 0.00       |
| LSRAM                   | 0               | 952    | 0.00       |
| Math                    | 4               | 924    | 0.43       |
+-------------------------+-----------------+--------+------------+

* Interface 4LUTs and DFFs are occupied due to the uses of LSRAM, Math, and uSRAM.
  Number of interface 4LUTs/DFFs = (36 * #.LSRAM) + (36 * #.Math) + (12 * #.uSRAM) = (36 * 0) + (36 * 4) +
(12 * 0) = 144.
```

We can see from section 2 of summary.result.rpt that the minimum period for the synthesized block is 2.771 ns, which is below the threshold of 6.734 ns from the demo design. This means we can safely integrate this block into the demo design and meet timing.

We can compare the SmartHLS core resource utilization to the documentation of the RGB to YCbCr44 SolutionCore on PolarFire® shown in Table 5.

Table 5: Fabric Resource Utilization of RGB to YCbCr444 SolutionCore.

Resource	Usage
DFFs	51
4-input LUTs	86
MACC	9
RAM1kx18	0
RAM64x18	0

75

Why are the resources so different?

Like with AlphaBlend, the SolutionCore documentation only reports Fabric LUTs/DFFs and not interface LUTs/DFFs. While SmartHLS reports the combination of both Fabric and Interface LUTs/DFFs. But for this hardware block, we will only focus on the MACC math blocks.
We would expect there to be 9 MACC math blocks in the final circuit since there are 9 18x18 multiply operations in the design. However, SmartHLS has a strength reduction optimization that can lower a multiply by constant into adds with shifts-by-constant. In this case, we can save 5 multipliers as shown in Table 6.

Table 6: SmartHLS Strength Reduction Optimization

Multiply by Constant	Fixed Point Representation	Equivalent shifts-by-constant and adds
129.057	$33,038 \times 2^{-8}$	- (1 << 1) + (1 << 4) + (1 << 8) + (1 << 15)
25.064	$6,416 \times 2^{-8}$	+ (1 << 4) + (1 << 8) + (1 << 11) + (1 << 12)
112.439 (used twice)	$28,784 \times 2^{-8}$	- (1 << 4) + (1 << 7) - (1 << 12) + (1 << 15)
18.285	$4,680 \times 2^{-8}$	+ (1 << 3) + (1 << 6) + (1 << 9) + (1 << 12)

We can turn off the SmartHLS strength reduction pass to see the difference in resources. The SmartHLS strength reduction optimization is an advanced setting that is not listed in the SmartHLS IDE. Therefore, to turn off this setting we will need to create a custom constraints Tcl file.

Open to the SmartHLS Constraints Menu (⚙). Select "Set custom config file" from the dropdown and enter the Constraint Value as "custom_config.tcl". Then click Add:

Now click OK:
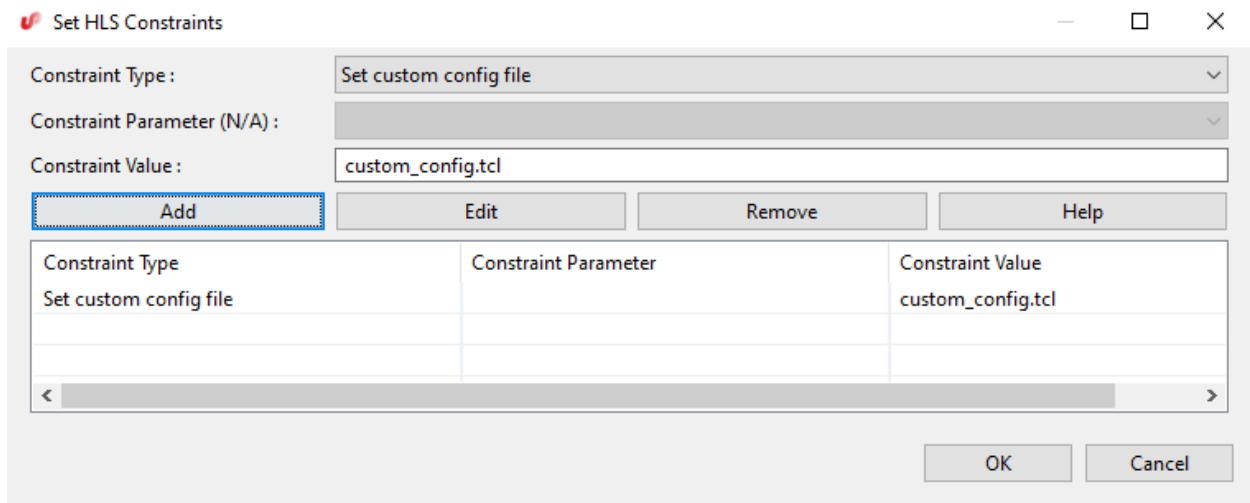
Now in the Project Explorer, right click and select New -> File:
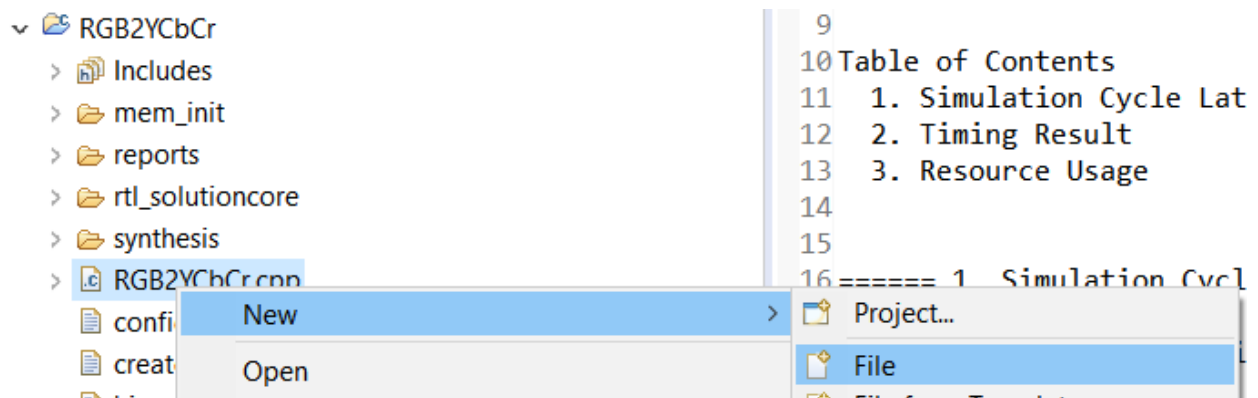
Enter the file name of "custom_config.tcl". This should match the file name entered in the Set HLS Constraints previously. Click Finish:
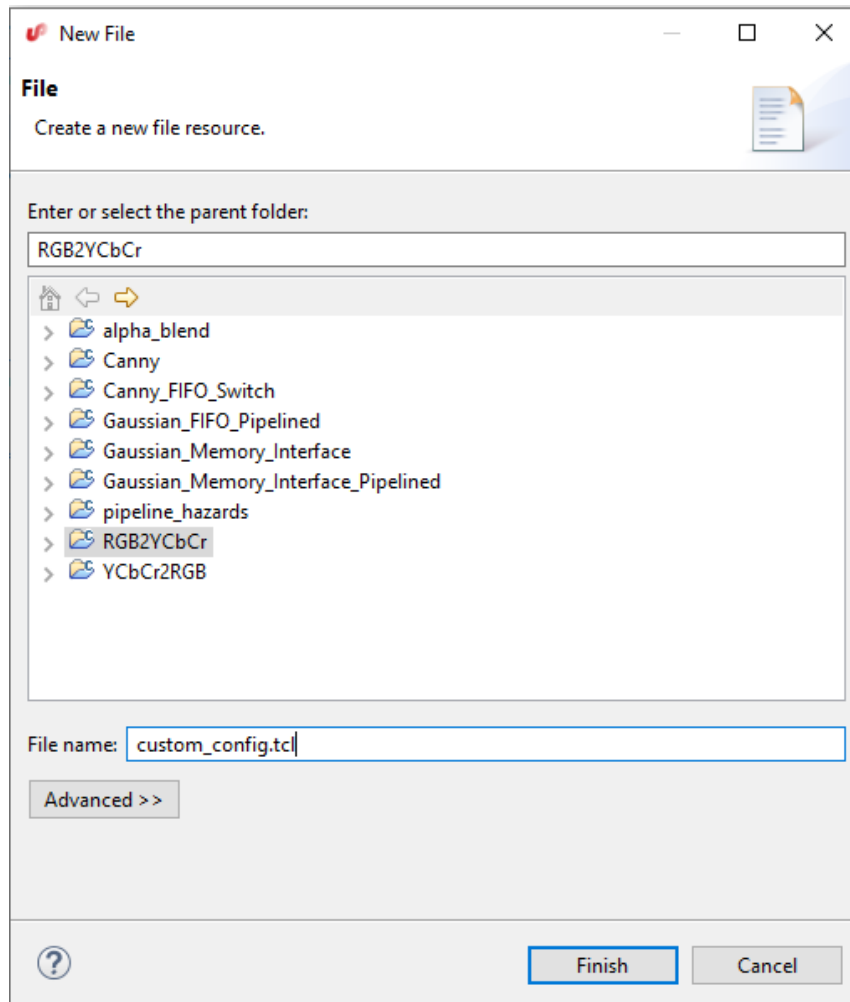
The custom Tcl file allows us to enter advanced SmartHLS Tcl constraints. In the Tcl file, enter the SmartHLS Tcl command "set_parameter STRENGTH_REDUCTION 0" and press Ctrl-S to save the changes. This will turn off (0) the SmartHLS strength reduction (STRENGTH_REDUCTION) optimization:
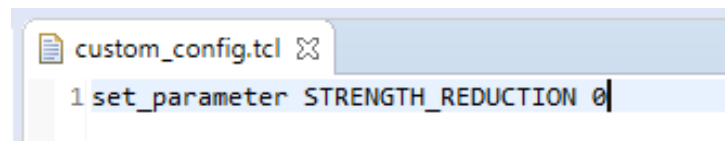
Now rerun compile software to hardware (🗲). Then rerun FPGA synthesis (▦). The new resources should be:

```
====== 2. Timing Result of HLS-generated IP Core (top-level module:
RGB2YCbCr_smarthls_top) ======


+--------------+---------------+-------------+-------------+----------+-------------+
| Clock Domain | Target Period | Target Fmax | Worst Slack | Period   | Fmax        |
+--------------+---------------+-------------+-------------+----------+-------------+
| clk          | 10.000 ns     | 100.000 MHz | 7.195 ns    | 2.805 ns | 356.506 MHz |
+--------------+---------------+-------------+-------------+----------+-------------+
```

The reported Fmax is for the HLS core in isolation (from Libero's post-place-and-route timing analysis).
When the HLS core is integrated into a larger system, the system Fmax may be lower depending on the critical path of the system.

```
====== 3. Resource Usage of HLS-generated IP Core (top-level module:
RGB2YCbCr_smarthls_top) ======


+--------------------------+-----------------+--------+------------+
| Resource Type            | Used            | Total  | Percentage |
+--------------------------+-----------------+--------+------------+
| Fabric + Interface 4LUT* | 489 + 144 = 633 | 299544 | 0.21       |
| Fabric + Interface DFF*  | 124 + 144 = 268 | 299544 | 0.09       |
| I/O Register             | 0               | 1536   | 0.00       |
| User I/O                 | 0               | 512    | 0.00       |
| uSRAM                    | 0               | 2772   | 0.00       |
| LSRAM                    | 0               | 952    | 0.00       |
| Math                     | 4               | 924    | 0.43       |
+--------------------------+-----------------+--------+------------+
```

```
* Interface 4LUTs and DFFs are occupied due to the uses of LSRAM, Math, and uSRAM.
  Number of interface 4LUTs/DFFs = (36 * #.LSRAM) + (36 * #.Math) + (12 * #.uSRAM) =
(36 * 0) + (36 * 4) + (12 * 0) = 144.
```

Now close all project files.

11.2 YCbCr2RGB Block

The YCbCr2RGB block design is very similar to the RGB2YCbCr block that we just explained. We leave investigating this design in more detail as an exercise for the reader.

In the SmartHLS project explorer, double click the "YCbCr2RGB" project and open up the YCbCr2RGB.cpp file.

The top-level function is YCbCr2RGB_smarthls() and implements Equation 2 in fixed-point math:

```
// change divide by 256 to right shift by 8, add 0.5 for rounding
fixpt_t R = fixpt_t(-222.921) + (( fixpt_t(298.082)*in.Y + fixpt_t(408.583)*in.Cr                              ) >> 8) + fixpt_t(0.5);
fixpt_t G = fixpt_t( 135.576) + (( fixpt_t(298.082)*in.Y - fixpt_t(100.291)*in.Cb - fixpt_t(208.120)*in.Cr ) >> 8) + fixpt_t(0.5);
fixpt_t B = fixpt_t(-276.836) + (( fixpt_t(298.082)*in.Y + fixpt_t(516.412)*in.Cb                              ) >> 8) + fixpt_t(0.5);
```

In this design, the fixed-point type needed 11 integer bits (vs 10 integer bits for RGB2YCbCr).

```
// Fixed point type: Q11.7
// 11 integer bits and 7 fractional bits
typedef ap_fixpt<18, 11> fixpt_t;
```

Why was this change needed? To avoid overflow caused by larger numbers in the equations.

We also need to perform saturation, which converts negative values to 0, and values greater than 255 to 255. We can do this using an 8-bit unsigned ap_ufixpt type with the AP_SAT option:

```
// saturate values to [0, 255] range
rgb.R = ap_ufixpt<8, 8, AP_TRN, AP_SAT>(R);
rgb.G = ap_ufixpt<8, 8, AP_TRN, AP_SAT>(G);
rgb.B = ap_ufixpt<8, 8, AP_TRN, AP_SAT>(B);
```

From the SmartHLS user guide, the AP_SAT option means that on positive and negative overflow, saturate the result to the maximum or minimum value in the range respectively.

Compile (ⓒ) and run (⊙) the software to verify software correctness. You should see "PASS" printed in the Console:

```
Expected: R=0 G=136 B=0
Actual: R=0 G=136 B=0
Expected: R=98 G=149 B=200
Actual: R=98 G=149 B=200
Expected: R=119 G=160 B=192
Actual: R=119 G=160 B=192
Expected: R=219 G=49 B=141
Actual: R=219 G=49 B=141
Expected: R=124 G=170 B=136
Actual: R=124 G=170 B=136
Expected: R=255 G=125 B=255
Actual: R=255 G=125 B=255
Expected: R=255 G=255 B=255
Actual: R=255 G=255 B=255
PASS
```

After compiling software to hardware (💾) the following RTL interface should be shown in the summary.hls.YCbCr2RGB_smarthls.rpt file:

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

+---------------------------------------------------------------------------------------+
| RTL Interface Generated by SmartHLS                                                   |
+-------------+---------------+------------------+------------------+-------------------+
| C++ Name    | Interface Type| Signal Name      | Signal Bit-width | Signal Direction  |
+-------------+---------------+------------------+------------------+-------------------+
|             | Clock & Reset | clk              | 1                | input             |
|             |               | reset            | 1                | input             |
+-------------+---------------+------------------+------------------+-------------------+
|             | Control       | finish           | 1                | output            |
|             |               | ready            | 1                | output            |
|             |               | start            | 1                | input             |
+-------------+---------------+------------------+------------------+-------------------+
| input_fifo  | FIFO          | input_fifo_ready | 1                | output            |
|             |               | input_fifo_valid | 1                | input             |
|             |               | input_fifo_Y     | 8                | input             |
|             |               | input_fifo_Cb    | 8                | input             |
|             |               | input_fifo_Cr    | 8                | input             |
+-------------+---------------+------------------+------------------+-------------------+
| output_fifo | FIFO          | output_fifo_ready| 1                | input             |
|             |               | output_fifo_valid| 1                | output            |
|             |               | output_fifo_R    | 8                | output            |
|             |               | output_fifo_B    | 8                | output            |
|             |               | output_fifo_G    | 8                | output            |
+-------------+---------------+------------------+------------------+-------------------+
```

After running SmartHLS co-simulation (▶) you should see the hardware passes all tests with the following output in the Console:

```
+------------------------+----------------+--------------------------+
| Top-Level Name         | Number of calls | Simulation time (cycles) |
+------------------------+----------------+--------------------------+
| YCbCr2RGB_smarthls_top | 7              | 11                       |
+------------------------+----------------+--------------------------+
Simulation time (cycles): 11
SW/HW co-simulation: PASS
```

81

Finally, if you run FPGA synthesis ( ) you should see the following expected output in summary.results.rpt:

```
====== 2. Timing Result of HLS-generated IP Core (top-level module: YCbCr2RGB_smarthls_top) ======

+--------------+---------------+-------------+-------------+----------+-------------+
| Clock Domain | Target Period | Target Fmax | Worst Slack | Period   | Fmax        |
+--------------+---------------+-------------+-------------+----------+-------------+
| clk          | 10.000 ns     | 100.000 MHz | 7.249 ns    | 2.751 ns | 363.504 MHz |
+--------------+---------------+-------------+-------------+----------+-------------+
```

The reported Fmax is for the HLS core in isolation (from Libero's post-place-and-route timing analysis). When the HLS core is integrated into a larger system, the system Fmax may be lower depending on the critical path of the system.

```
====== 3. Resource Usage of HLS-generated IP Core (top-level module: YCbCr2RGB_smarthls_top) ======

+-------------------------+-----------------+--------+------------+
| Resource Type           | Used            | Total  | Percentage |
+-------------------------+-----------------+--------+------------+
| Fabric + Interface 4LUT*| 254 + 180 = 434 | 299544 | 0.14       |
| Fabric + Interface DFF* | 29 + 180 = 209  | 299544 | 0.07       |
| I/O Register            | 0               | 1536   | 0.00       |
| User I/O                | 0               | 512    | 0.00       |
| uSRAM                   | 0               | 2772   | 0.00       |
| LSRAM                   | 0               | 952    | 0.00       |
| Math                    | 5               | 924    | 0.54       |
+-------------------------+-----------------+--------+------------+
```

* Interface 4LUTs and DFFs are occupied due to the uses of LSRAM, Math, and uSRAM.
  Number of interface 4LUTs/DFFs = (36 * #.LSRAM) + (36 * #.Math) + (12 * #.uSRAM) = (36 * 0) + (36 * 5) + (12 * 0) = 180.

Now close all project files.

12 Gaussian Blur Filter Block

Gaussian blur is widely used in image processing for blurring or smoothing an input image to remove noise and reduce detail from an image. An example is shown Figure 25, where the left image is the original and the right image is after applying the Gaussian Blur Filter. The Gaussian blur is also the first filter stage of the Canny Edge Filter.
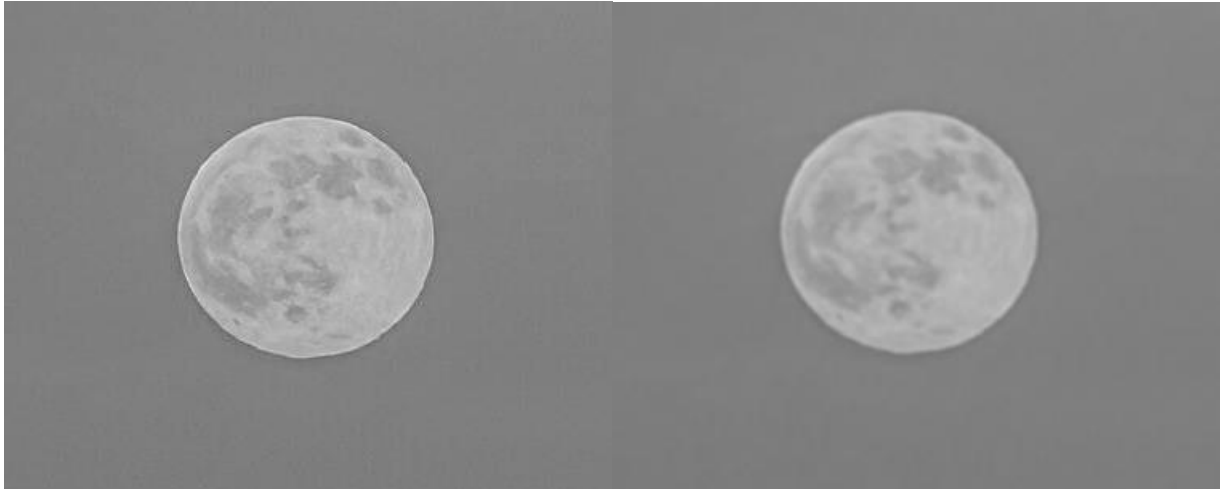
Figure 25: Side-by-side of original grayscale image (left) and Gaussian Blurred image (right)

In this section, we will describe how to design the Gaussian Blur Filter in C++ using SmartHLS and optimize the filter for hardware. This will be similar to the SmartHLS Sobel Filter Tutorial, but we will go in depth about what each optimization is for and what the impact is on the generated circuit.

Gaussian Blur filtering combines the values of the pixels surrounding the current pixel to create smaller changes between adjacent values resulting in a smoother image. The amount of blur is controlled by the size and coefficients of the filter. Gaussian Blur filtering uses a 2D Gaussian distribution as a filter. For each pixel in the image, the values of the surrounding pixels are multiplied with their corresponding filter coefficient and summed together.

To see this, open the Gaussian_Memory_Interface project and then open the gaussian_filter.cpp source file.
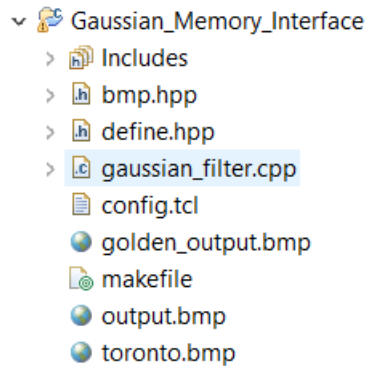
We can see on line 7 that the size of the filter used in this implementation is 5x5 with 25 *coefficients* in total. The coefficients correspond to a Gaussian distribution centered at the middle element (2,2) which has the value 12. The DIVISOR is then used to normalize the sum back to a value between 0 and 255. The values of the filter are specifically chosen so that the DIVISOR is a power of 2, making the hardware implementation of the divide a right-shift instead of a divide.

```cpp
const unsigned int KERNEL_SIZE = 5;

// A Simpler Gaussian Filter for hardware (the divisor is a power of 2)
const unsigned int GAUSSIAN[KERNEL_SIZE][KERNEL_SIZE] = {{1, 3, 4, 3, 1},
                                                         {3, 8, 10, 8, 3},
                                                         {4, 10, 12, 10, 4},
                                                         {3, 8, 10, 8, 3},
                                                         {1, 3, 4, 3, 1}};
const unsigned int DIVISOR = 128;
```

12.1 Gaussian Filter with Memory Interface

We will start with a basic implementation of the Gaussian Blur Filter. Scroll down to the gaussian_filter_memory() function on line 25. Notice this function is marked as the top-level function by the function top pragma:

```cpp
// Gaussian Filter.
Void gaussian_filter_memory(hls::ap_uint<1> on,
                            unsigned char input_buffer[][WIDTH],
                            unsigned char output_buffer[][WIDTH]) {
#pragma HLS function top
#pragma HLS interface argument(input_buffer) type(memory) num_elements(SIZE)
#pragma HLS interface argument(output_buffer) type(memory) num_elements(SIZE)
```

There are two array arguments to the top-level function which represents the input image and the filtered output image:

```cpp
unsigned char input_buffer[][WIDTH],
unsigned char output_buffer[][WIDTH]
```

84

There are two SmartHLS "interface" pragmas which are needed here to specify that these two array arguments of "memory" type interface have a certain depth. The depth of the memory must also be specified for the co-simulation, since our C++ testbench in main() does not use arrays with static size.

```
#pragma HLS interface argument(input_buffer) type(memory) num_elements(SIZE)
#pragma HLS interface argument(output_buffer) type(memory) num_elements(SIZE)
```

There is also a third input called "on" which is an unsigned int of size 1.
```
Hls::ap_uint<1> on,
```

This input will be connected to DIP switch 1 (SW6) in the demo design and turns on or off the Gaussian Blur Filter. On line 38, if the switch is turned off (!on) then we will pass the input directly to the output:

```
if (!on || out_of_bounds) {
    output_buffer[i][j] = input_buffer[i][j];
    continue;
}
```

The filtering algorithm can be seen in the main loop on line 43. The 5x5 area around the current pixel under consideration is multiplied with its corresponding Gaussian coefficient. The result is summed, normalized then stored in the output array.

```
unsigned int sum = 0;
for (unsigned int m = 0; m < KERNEL_SIZE; m++) {
    for (unsigned int n = 0; n < KERNEL_SIZE; n++) {
        sum += ((unsigned int)input_buffer[I + m-center]
                                          [j + n-center]) *
            GAUSSIAN[m][n];
    }
}

sum /= DIVISOR;

output_buffer[i][j] = (unsigned char)sum;
```

Now click the compile software button (🄲) on the top bar and then click the run software (▶) button. You should see the output in the Console stating that it passed:
```
Result: 2073600
RESULT: PASS
```

The testbench for this design is found in the main() function on line 59. This is very similar to the testbench of the Alpha Blending design where a 1920x1080 bmp image is read as input. There is also a golden output bmp image used to compare with the pixels generated by the filter implementation gaussian_filter_memory().

85

```
gaussian_filter_memory(on, input_image, output_image_gaussian);

// output validation
for (i = 0; i < HEIGHT; i++) {
    for (j = 0; j < WIDTH; j++) {
        unsigned char gold = golden_output_image->r;
        unsigned char hw = output_image_gaussian[i][j];
        output_image_ptr->r = hw;
        output_image_ptr->g = hw;
        output_image_ptr->b = hw;

        if (hw != gold) {
            printf("ERROR: ");
            printf("i = %d j = %d gold = %d hw = %d\n", i, j, gold, hw);
        } else {
            matching++;
        }

        output_image_ptr++;
        golden_output_image++;
    }
}
```

The golden output is generated by running the software version of the filter first and manually checking that the result is satisfactory. You can verify this yourself by opening the "toronto.bmp" and the "output.bmp" generated during the test. The bmp "toronto.bmp" is not in grayscale but the level of detail of each image can still be compared by zooming in on the moon in each bmp and verifying that they look something like the images shown in Figure 25: Side-by-side of original grayscale image (left) and Gaussian Blurred image (right). You can often verify that the output generated by software is satisfactory manually and then use that output as the golden output for the hardware implementation.

Now we can verify that the generated RTL is functionally correct with a co-simulation. Like Alpha Blending, to speed up the co-simulation for the purposes of this training we will run with a smaller image.

Open define.hpp and uncomment FAST_COSIM defined on line 5 and then save the file. The commented out FAST_COSIM define might be folded into the comment by eclipse and needs to be expanded by clicking the plus button.

```
// uncomment this line to test on a smaller image for faster co-simulation
#define FAST_COSIM
```

This will change the main() C++ testbench to use a much smaller 100 x 56 input image (toronto_100x56.bmp). After finishing this part of the training, be careful to **comment out this line again** before generating the hardware to be exported to SmartDesign, otherwise the generated hardware will be for the incorrect input size. This change is necessary as the function depends on the image sizes in the for-loops on line 34.

```
for (int i = 0; i < HEIGHT; i++) {
    for (int j = 0; j < WIDTH; j++) {
```

You can recompile (⊙) and rerun (▶) the software to verify correctness on the new input:

```
Result: 5600
RESULT: PASS
21:58:06 Build Finished (took 5s.849ms)
```

Now rerun SmartHLS to generate the hardware ( ) and then run co-simulation with ModelSim (click the button). You should see the following output in the Console stating that the co-sim has passed:

```
+---------------------------+----------------+--------------------------+
| Top-Level Name            | Number of calls | Simulation time (cycles) |
+---------------------------+----------------+--------------------------+
| gaussian_filter_memory_top | 1              | 152,917                  |
+---------------------------+----------------+--------------------------+
Simulation time (cycles): 152,917
SW/HW co-simulation: PASS
```

16:26:53 Build Finished (took 15s.436ms)

This version of the Gaussian filter is very similar to a software implementation of a Gaussian filter. However, there are multiple ways to improve C++ code to get better hardware performance.

As this Gaussian filter performs straight-line sequential processing for each pixel, we can get the total latency for each pixel of the input memory by finding the latency of processing a single pixel and then multiplying by the number of pixels in an image as each pixel of the input array is processed serially. Processing an entire array will take LATENCY * HEIGHT * WIDTH cycles. Because there is repeated work done in a loop, we can gain performance by generating a pipeline to run some of the computations in parallel. Note the cycle latency in the co-sim output above (177,877) for processing the 100x56 input images. This will be our baseline to compare to.

In the report, notice that both the input and output array pointer arguments have been generated as memory interfaces in RTL. SmartHLS expects the memories from array and pointer arguments to be external to the SmartHLS block itself and only provides the control signals to read and write to the memory based on the loads and stores from inside the function. Also

notice that the ap_uint argument becomes a single input wire at the interface:

```
+------------------------------------------------------------------------------------+
| RTL Interface Generated by SmartHLS                                                |
+---------------+-----------------+---------------------------+-----------------+-----------------+
| C++ Name      | Interface Type  | Signal Name               | Signal Bit-width | Signal Direction |
+---------------+-----------------+---------------------------+-----------------+-----------------+
|               | Clock & Reset   | clk                       | 1               | input           |
|               |                 | reset                     | 1               | input           |
+---------------+-----------------+---------------------------+-----------------+-----------------+
|               | Control         | finish                    | 1               | output          |
|               |                 | ready                     | 1               | output          |
|               |                 | start                     | 1               | input           |
+---------------+-----------------+---------------------------+-----------------+-----------------+
| on            | Scalar Argument | on                        | 1               | input           |
+---------------+-----------------+---------------------------+-----------------+-----------------+
| input_buffer  | Memory          | input_buffer_address_a    | 13              | output          |
|               |                 | input_buffer_address_b    | 13              | output          |
|               |                 | input_buffer_clken        | 1               | output          |
|               |                 | input_buffer_read_data_a  | 8               | input           |
|               |                 | input_buffer_read_data_b  | 8               | input           |
|               |                 | input_buffer_read_en_a    | 1               | output          |
|               |                 | input_buffer_read_en_b    | 1               | output          |
+---------------+-----------------+---------------------------+-----------------+-----------------+
| output_buffer | Memory          | output_buffer_address_a   | 13              | output          |
|               |                 | output_buffer_address_b   | 13              | output          |
|               |                 | output_buffer_clken       | 1               | output          |
|               |                 | output_buffer_write_data_a | 8              | output          |
|               |                 | output_buffer_write_data_b | 8              | output          |
|               |                 | output_buffer_write_en_a  | 1               | output          |
|               |                 | output_buffer_write_en_b  | 1               | output          |
+---------------+-----------------+---------------------------+-----------------+-----------------+
```

12.1.1 When Can SmartHLS Co-Simulation Fail?

Now that we have tested a few designs in SmartHLS we wanted to briefly cover some cases where software execution can pass but SmartHLS co-simulation fails. Note that this is quite a rare occurrence, you should normally expect SmartHLS co-simulation to always match software execution.

A simple case is if your main function ever returns a non-zero value in software. For example, change the main() function to always return 1 on line 129 in gaussian_filter.cpp:

```
//return result_incorrect;
return 1;
```

Now run co-simulation and you will see the output:
Error: Running C testbench failed. Make sure main() returns 0.

Now undo the change.

88

Another time that co-simulation could fail is if the user specifies an incorrect value in a SmartHLS pragma. For example, specifying an incorrect depth on a memory interface such as the following on line 29:

```
#pragma HLS interface argument(input_buffer) type(memory) num_elements(SIZE)
```

For example, we can try changing the correct SIZE array depth to a wrong value like 10:

```
#pragma HLS interface argument(input_buffer) type(memory) num_elements(10)
```

Now we rerun SmartHLS to generate the hardware ( ):
```
Error: Expect the specified depth (10) for argument 'input_buffer' to be a
multiple of the combined depth of the inner dimensions (100). Please change
the specified depth to a multiple of the combined inner dimension depth (100).
```

We were not able to get to the co-simulation stage, since SmartHLS was able to detect that the depth was not a multiple of the WIDTH (which is 100):
```
unsigned char input_buffer[][WIDTH],
```

We can try another wrong array depth which is a multiple of 100 to avoid this SmartHLS check:
```
#pragma HLS interface argument(input_buffer) type(memory) num_elements(100)
```

Now rerun SmartHLS to generate the hardware ( ). Since SmartHLS relies on the user to set the correct depth value, SmartHLS does not realize the depth is wrong and will not give an error message.

Now when we rerun co-simulation ( ) we will see that co-simulation fails:

```
ERROR: i = 55 j = 99 gold = 96 hw = 84
Result: 101
RESULT: FAIL
...
Simulation time (cycles): 177,877
SW/HW co-simulation: FAIL

16:31:51 Build Finished (took 15s.374ms)
```

In this case, the generated circuit is still correct, but SmartHLS's automatically generated co-simulation testbench is incorrect. Because we specified the wrong depth, the co-simulation testbench is now missing some of the expected results. Note that if we recompile and rerun the software everything will still pass in software, since the SmartHLS pragmas only affect SmartHLS hardware generation and co-simulation.

If SmartHLS co-simulation fails unexpectedly then this can also indicate a bug with the hardware generated by SmartHLS. Technical support is available through the website at: www.microchip.com/support.

Now undo all the changes and then close all project files.

12.1.2 Gaussian Filter with Loop Pipelining

We will continue trying to improve the base Gaussian Filter design.

Open the Gaussian_Memory_Interface_Pipelined project and then open the gaussian_filter.cpp source file.

```
∨  Gaussian_Memory_Interface_Pipelined
   >  Includes
   >  bmp.hpp
   >  define.hpp
   >  gaussian_filter.cpp
      config.tcl
      golden_output.bmp
      makefile
      output.bmp
      toronto.bmp
```
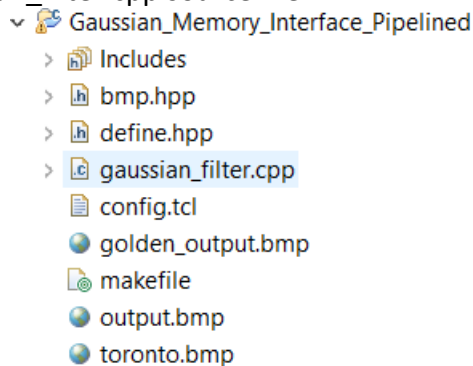
Scroll down to the main body loop on line 35. Like Sobel, we can pipeline the main loop to gain performance. With the loop pipelining pragma, the loop body will automatically be partitioned into pipeline stages. The module will also only run the pipeline for the number of iterations of the loop before requiring the start signal to be re-asserted. This optimization should increase throughput considerably.

```
#pragma HLS loop pipeline
```

Note, loop pipelining will flatten the loop body by inlining any functions and unrolling any loops. This is to make sure the loop body can be properly analyzed and partitioned into pipeline stages. As the pipeline pragma is applied to the outside for-loop, the inside j for-loop will be completely unrolled. This creates many copies of the j loop body. Not only would this use a massive amount of resources, it will also slow down compilation considerably, both of which we want to avoid. To work around this, the double for loop can be collapsed into a single for loop so that no loop unrolling needs to occur.

```
#pragma HLS loop pipeline
    for (int i = 0; i < (HEIGHT * WIDTH); i++) {
        unsigned int pos_i = i / WIDTH;
        unsigned int pos_j = i % WIDTH;
```

 Now run "Compile Software to Hardware" (click the button).

Look in the Console to find the message about loop pipelining. This message states that the initiation interval of the pipeline is 13 and the number of stages is 23.

```
Info: Done pipelining the loop on line 35 of gaussian_filter.cpp with label
"for_loop_gaussian_filter_cpp_35_5".
     Pipeline Initiation Interval (II) = 13. Pipeline length = 17.
```

We can see that there is memory contention within the loop pipeline that prevents the initiation interval from becoming 1 in the SmartHLS Info message:

```
Info: Pipelining the loop on line 35 of gaussian_filter.cpp with label "for.loop:gaussian_filter.cpp:35:5".
Info: Assigning new label to the loop on line 35 of gaussian_filter.cpp with label
"for_loop_gaussian_filter_cpp_35_5"
Info: Resource constraint limits initiation interval to 13.
     Resource '@input_buffer@_external_memory_port' has 25 uses per cycle but only 2 units available.
```

Operation	Location	Competing Use Count
'load' (8b) operation for array 'input_buffer'	line 48 of gaussian_filter.cpp	1
'load' (8b) operation for array 'input_buffer'	line 41 of gaussian_filter.cpp	1
'load' (8b) operation for array 'input_buffer'	line 48 of gaussian_filter.cpp	2
'load' (8b) operation for array 'input_buffer'	line 48 of gaussian_filter.cpp	3
'load' (8b) operation for array 'input_buffer'	line 48 of gaussian_filter.cpp	4
'load' (8b) operation for array 'input_buffer'	line 48 of gaussian_filter.cpp	5
'load' (8b) operation for array 'input_buffer'	line 48 of gaussian_filter.cpp	6
'load' (8b) operation for array 'input_buffer'	line 48 of gaussian_filter.cpp	7
'load' (8b) operation for array 'input_buffer'	line 48 of gaussian_filter.cpp	8
'load' (8b) operation for array 'input_buffer'	line 48 of gaussian_filter.cpp	9
'load' (8b) operation for array 'input_buffer'	line 48 of gaussian_filter.cpp	10
'load' (8b) operation for array 'input_buffer'	line 48 of gaussian_filter.cpp	11
'load' (8b) operation for array 'input_buffer'	line 48 of gaussian_filter.cpp	12
'load' (8b) operation for array 'input_buffer'	line 48 of gaussian_filter.cpp	13
'load' (8b) operation for array 'input_buffer'	line 48 of gaussian_filter.cpp	14
'load' (8b) operation for array 'input_buffer'	line 48 of gaussian_filter.cpp	15
'load' (8b) operation for array 'input_buffer'	line 48 of gaussian_filter.cpp	16
'load' (8b) operation for array 'input_buffer'	line 48 of gaussian_filter.cpp	17
'load' (8b) operation for array 'input_buffer'	line 48 of gaussian_filter.cpp	18
'load' (8b) operation for array 'input_buffer'	line 48 of gaussian_filter.cpp	19
'load' (8b) operation for array 'input_buffer'	line 48 of gaussian_filter.cpp	20
'load' (8b) operation for array 'input_buffer'	line 48 of gaussian_filter.cpp	21
'load' (8b) operation for array 'input_buffer'	line 48 of gaussian_filter.cpp	22
'load' (8b) operation for array 'input_buffer'	line 48 of gaussian_filter.cpp	23
'load' (8b) operation for array 'input_buffer'	line 48 of gaussian_filter.cpp	24
'load' (8b) operation for array 'input_buffer'	line 48 of gaussian_filter.cpp	25
	Total # of Competing Uses	25

In this case, the resource contention is the use of the RAM "@input_buffer@_external_memory_port" which has 25 loads all from line 48 of gaussian_filter.cpp but there are only 2 memory ports to use (dual-port RAM in FPGA). If we look at line 48 of gaussian_filter.cpp we find that all the loads come from the image values read from input_buffer used in calculating the new filtered value.

```
for (unsigned int m = 0; m < KERNEL_SIZE; m++) {
    for (unsigned int n = 0; n < KERNEL_SIZE; n++) {
        sum += ((unsigned int)input_buffer[pos_i + m - center]
                                           [pos_j + n - center]) *
            GAUSSIAN[m][n];
```

```
        }
    }
```

Why is there no memory contention for the GAUSSIAN 5x5 array which is also accessed every iteration? Because SmartHLS unrolls the loops and realizes that GAUSSIAN is a constant array. Therefore, SmartHLS can automatically replace GAUSSIAN array accesses with constant values, becoming equivalent to the following:

```
unsigned int sum = 0;
sum += ((unsigned int)input_buffer[pos_i + 0 - center]
                                   [pos_j + 0 - center]) *
        1; // GAUSSIAN[0][0]
sum += ((unsigned int)input_buffer[pos_i + 0 - center]
                                   [pos_j + 1 - center]) *
        3; // GAUSSIAN[0][1]
...
sum += ((unsigned int)input_buffer[pos_i + 4 - center]
                                   [pos_j + 4 - center]) *
        1; // GAUSSIAN[4][4]
```

The same pipelining information can be found in summary.hls.gaussian_filter_memory_pipelined.rpt. The tables on where the contention is coming from can be found in the Console output or in the report file hls.log. Open summary.hls.gaussian_filter_memory_pipelined.rpt and scroll down to section 2 and then scroll right to find the pipelining information. Note, the Iteration Count and Latency are much larger than the ones we saw when running co-sim in the design without pipelining, but this is due to the design being generated for the full 1920x1080 input while the co-sim we ran used the reduced 100x56 input.

```
====== 2. Function and Loop Scheduling Results ======

+---------------------------------------------------------------------------+
| Function: gaussian_filter_memory_pipelined takes 26956808 cycles          |
+--------------------------------+…+-------------------+----+--------------+
| Loop                           |…| Iteration Latency | II | Total Latency |
+--------------------------------+…+-------------------+----+--------------+
| for.loop:gaussian_filter.cpp:35:5 |…| 17              | 13 | 26956804      |
+--------------------------------+…+-------------------+----+--------------+
```

   Now uncomment FAST_COSIM in define.hpp line 5 like before, save, then rerun SmartHLS to generate the hardware ( ) and then run co-simulation with ModelSim (click the button).

```
// uncomment this line to test on a smaller image for faster co-simulation
#define FAST_COSIM
```

Once co-sim finishes, you should see the following output in the Console:

```
+--------------------------------------+----------------+--------------------------+
| Top-Level Name                       | Number of calls | Simulation time (cycles) |
+--------------------------------------+----------------+--------------------------+
| gaussian_filter_memory_pipelined_top | 1              | 72,818                   |
+--------------------------------------+----------------+--------------------------+
Simulation time (cycles): 72,818
SW/HW co-simulation: PASS

09:14:48 Build Finished (took 35s.896ms)
```

With this pipeline optimization and pipeline length reduced to 16, the time to process one frame becomes approximately HEIGHT * WIDTH * 13 + LATENCY (100*56*13+16 = 72,818), which is a significant improvement over the previous design. We can see that this has reduced the cycle latency from 177,877 to 72,818 which is a 59% reduction in latency.

Now close all project files.

12.2 Gaussian Filter with FIFO and LineBuffer

Although we now have a design that is better than the unpipelined version, there is still a lot of room for improvement. In almost all cases, we want to get the initiation interval to as close to 1 as possible to have the best performance.

In this case, the large initiation interval was caused by memory contention. In general, we can reduce memory contention by **partitioning** the memory into smaller pieces. This allows more elements to be accessed in parallel as each partition can have two accesses instead of the entire memory having two accesses. SmartHLS allows memories to be automatically partitioned based on analysis of the memory access patterns. SmartHLS also has pragmas to force partitioning of a memory into a specific format. However, the input array is 1920x1080 which is too large to be fully partitioned by SmartHLS and still fit on the PolarFire FPGA.

If we look at the previous hardware blocks in this demo design, like alpha blend, they all use FIFOs as top-level function inputs and outputs. This is because in hardware the video stream arrives 1 pixel every clock cycle. We can also change the Gaussian filter to use FIFOs on the top-level interface which forces the filter to only read a single pixel per iteration. This change will prevent the memory contention we saw previously for the input_buffer but is also inconvenient because we need previous pixels and future pixels to perform the filtering. SmartHLS provides a special data structure called a LineBuffer implemented in "*image_processing.hpp*" to easily keep track of these extra pixels.

Figure 26: SmartHLS Line Buffer Data Structure

Figure 26 shows an example of a 3x3 pixel window moving across the input image (Gaussian filter has a 5x5 window). The LineBuffer window represents the section of the image being used to calculate the current filter output. The filter output pixel coordinates correspond to the middle of the window. You will notice that as new pixels arrive (on the bottom right), we only need to retain the previous two rows of the image to update the LineBuffer window.
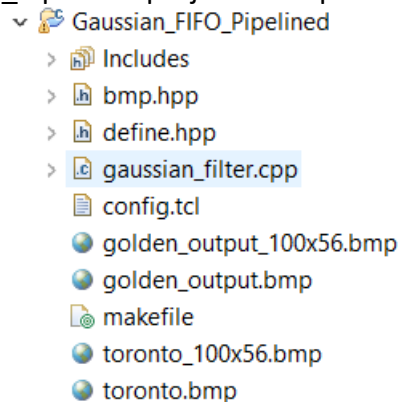
The SmartHLS LineBuffer data structure holds an internal array that has enough image rows to update the current window as new input pixels arrive. We shift a new pixel into the LineBuffer every iteration, corresponding to the last future pixel needed to process the current pixel and shift out the last unneeded pixel. In the Gaussian filter, we prefill the LineBuffer with the first few rows of the image so that all necessary previous and future pixels are available for processing immediately.

The reason why we get a benefit from using the LineBuffer, is that the LineBuffer window is **small enough** to be completely partitioned. Since SmartHLS unrolls the Gaussian filter loop, SmartHLS will also automatically partition the LineBuffer window into individual registers. Registers are not limited by two ports like memories, so the LineBuffer window registers can all be accessed in parallel every single cycle. Therefore, the LineBuffer removes all memory contention and the pipeline initiation interval is now 1.

Note that if SmartHLS did not partition the LineBuffer window memory into registers, then the Gaussian filter would still have 25 accesses to the LineBuffer window stored in dual-port memory and we would still get a pipeline initiation interval of 13.

Open the Gaussian_FIFO_Pipelined project and open the gaussian_filter.cpp source file.

```
∨ 🐾 Gaussian_FIFO_Pipelined
    > 📦 Includes
    > 🄷 bmp.hpp
    > 🄷 define.hpp
    > 🄬 gaussian_filter.cpp
        📄 config.tcl
        🌐 golden_output_100x56.bmp
        🌐 golden_output.bmp
        📑 makefile
        🌐 toronto_100x56.bmp
        🌐 toronto.bmp
```

Scroll down to the gaussian_filter_pipelined() top-level function on line 45. Both the input_fifo and output_fifo function arguments are now FIFO interfaces.

```cpp
void gaussian_filter_pipelined(hls::ap_uint<1> on_switch,
                               hls::FIFO<unsigned char> &input_fifo,
                               hls::FIFO<unsigned char> &output_fifo) {
```

Now scroll down to line 58. The loop within gaussian_filter_pipelined() is still loop pipelined.

```
#pragma HLS loop pipeline
    for (unsigned int i = 0; i < (HEIGHT * WIDTH + LineBufferFillCount); i++) {
```

On line 52, the declaration of the LineBuffer takes as C++ template arguments: the data type, the width of the image processed and the size of the filter. These arguments to tell the LineBuffer how much memory to allocate for the internal buffer.

```
    hls::LineBuffer<unsigned char, WIDTH, KERNEL_SIZE> line_buffer;
```

Every iteration of the loop, there will be a new pixel that gets shifted into the internal array of the LineBuffer. We want to pre-fill the line buffer to have all the necessary pixels to filter the first image pixel before we start the filtering.

```
    line_buffer.ShiftInPixel(input_pixel);

    // keep track of how many pixels we have shifted into the line_buffer to
    // tell when it is filled
    if (!is_filled(KERNEL_SIZE, i)) {
        continue;
    }
```

Once we fill the LineBuffer, we filter the image as normal on line 84 by using the *window* member of the LineBuffer which provides the pixels in the window of the pixel currently being processed.

```
    unsigned int sum = 0;
    for (unsigned int m = 0; m < KERNEL_SIZE; m++) {
        for (unsigned int n = 0; n < KERNEL_SIZE; n++) {
            sum += ((unsigned int)line_buffer.window[m][n]) * GAUSSIAN[m][n];
        }
    }

    sum /= DIVISOR;
    output_fifo.write((unsigned char)sum);
```

Using FIFOs and the LineBuffer data structure, we can reduce the initiation interval of the pipeline to 1 and process one pixel every single cycle. To see this, compile the design to hardware ( ).

Upon successful pipelining, you should find the following message in the Console output stating that the pipeline initiation interval is 1:

```
    Info: Generating pipeline for loop on line 59 of gaussian_filter.cpp with
    label "for_loop_gaussian_filter_cpp_59_5".
            Pipeline initiation interval = 1.
```

This result can also be found in the summary.hls.gaussian_filter_pipelined.rpt under section 2. Find *gaussian_filter_pipelined* and scroll to the right to see the pipeline result information.

```
+------------------------------------------------------------------------------+
| Function: gaussian_filter_pipelined takes 2077448 cycles                     |
|+--------------------------------+---+-----------+-----------------+----+--------------+
| Loop                           | … | Trip Count | Iteration Latency | II | Total Latency |
+--------------------------------+---+-----------+-----------------+----+--------------+
| for.loop:gaussian_filter.cpp:59:5 | … | 2077442   | 3               | 1  | 2077444      |
+--------------------------------+---+-----------+-----------------+----+--------------+
```

Also note, further up in the Console output you can find a console message stating that a LineBuffer memory has been partitioned.

        Info: Partitioning memory: line_buffer into 30 partitions.

Look at summary.hls.gaussian_filter_pipelined.rpt scroll to section 4. There are additional partitioned memories that can be found here that are not mentioned in the Console.

```
+------------------------------------------------------------------------------------+
| Local Memories                                                                     |
+------------------------------------------------------+---+------+------------+------------+-------+------------+
| Name                                                 | … | Type | Size [Bits] | Data Width | Depth | Read Latency |
+------------------------------------------------------+---+------+------------+------------+-------+------------+
| gaussian_filter_pipelined_entry_line_buffer_prev_ro   | … | RAM  | 15360      | 8          | 1920  | 1          |
| gaussian_filter_pipelined_entry_line_buffer_prev_ro_0 | … | RAM  | 15360      | 8          | 1920  | 1          |
| gaussian_filter_pipelined_entry_line_buffer_prev_ro_1 | … | RAM  | 15360      | 8          | 1920  | 1          |
| gaussian_filter_pipelined_entry_line_buffer_prev_ro_2 | … | RAM  | 15360      | 8          | 1920  | 1          |
+------------------------------------------------------+---+------+------------+------------+-------+------------+
```

See section 1 of the reports to verify the interface ports which have now changed to FIFOs.

```
+----------------------------------------------------------------------------------------------+
| RTL Interface Generated by SmartHLS                                                          |
+-------------+-------------------+-------------------+-----------------+------------------+
| C++ Name    | Interface Type    | Signal Name       | Signal Bit-width | Signal Direction |
+-------------+-------------------+-------------------+-----------------+------------------+
|             | Clock & Reset     | clk               | 1               | input            |
|             |                   | reset             | 1               | input            |
+-------------+-------------------+-------------------+-----------------+------------------+
|             | Control           | finish            | 1               | output           |
|             |                   | ready             | 1               | output           |
|             |                   | start             | 1               | input            |
+-------------+-------------------+-------------------+-----------------+------------------+
| input_fifo  | Input AXI Stream  | input_fifo_ready  | 1               | output           |
|             |                   | input_fifo_valid  | 1               | input            |
|             |                   | input_fifo        | 8               | input            |
+-------------+-------------------+-------------------+-----------------+------------------+
| output_fifo | Output AXI Stream | output_fifo_ready | 1               | input            |
|             |                   | output_fifo_valid | 1               | output           |
|             |                   | output_fifo       | 8               | output           |
+-------------+-------------------+-------------------+-----------------+------------------+
| on_switch   | Scalar Argument   | on_switch         | 1               | input            |
+-------------+-------------------+-------------------+-----------------+------------------+
```

Again, uncomment FAST_COSIM in define.hpp line 5, save, then rerun SmartHLS to generate the hardware ( ) and then run co-simulation with ModelSim (click the button).

```
// uncomment this line to test on a smaller image for faster co-simulation
#define FAST_COSIM
```

Once co-sim finishes, you should see the following output in the Console:

```
Retrieving hardware outputs from RTL simulation for gaussian_filter_pipelined
function call 1.
Result: 5600
RESULT: PASS
...
Simulation time (cycles): 5,809
SW/HW co-simulation: PASS
```

Notice that the cycle latency has been further reduced to 5,809. This can be found roughly by HEIGHT * WIDTH + LATENCY (100*56+6=5606). This is a reduction from the version without LineBuffer (72,818) by 92% and the version without pipelining (208,437) by 97%.

Now re-comment FAST_COSIM in define.hpp, save, then rerun SmartHLS ( ) to regenerate the hardware for 1920x1080 inputs.

```
// uncomment this line to test on a smaller image for faster co-simulation
//#define FAST_COSIM
```

Synthesize to design to FPGA ( ) and check the FMAX and resource usage in the summary.results.rpt file.

```
====== 2. Timing Result of HLS-generated IP Core (top-level module: gaussian_filter_pipelined_top) ======

+--------------+---------------+-------------+-------------+----------+-------------+
| Clock Domain | Target Period | Target Fmax | Worst Slack | Period   | Fmax        |
+--------------+---------------+-------------+-------------+----------+-------------+
| clk          | 10.000 ns     | 100.000 MHz | 5.365 ns    | 4.635 ns | 215.750 MHz |
+--------------+---------------+-------------+-------------+----------+-------------+

The reported Fmax is for the HLS core in isolation (from Libero's post-place-and-route timing analysis).
When the HLS core is integrated into a larger system, the system Fmax may be lower depending on the
critical path of the system.

====== 3. Resource Usage of HLS-generated IP Core (top-level module: gaussian_filter_pipelined_top) ======

+-------------------------+-------------------+--------+------------+
| Resource Type           | Used              | Total  | Percentage |
+-------------------------+-------------------+--------+------------+
| Fabric + Interface 4LUT*| 1263 + 144 = 1407 | 299544 | 0.47       |
| Fabric + Interface DFF* | 676 + 144 = 820   | 299544 | 0.27       |
| I/O Register            | 0                 | 1536   | 0.00       |
| User I/O                | 0                 | 512    | 0.00       |
| uSRAM                   | 0                 | 2772   | 0.00       |
| LSRAM                   | 4                 | 952    | 0.42       |
| Math                    | 0                 | 924    | 0.00       |
+-------------------------+-------------------+--------+------------+

* Interface 4LUTs and DFFs are occupied due to the uses of LSRAM, Math, and uSRAM.
  Number of interface 4LUTs/DFFs = (36 * #.LSRAM) + (36 * #.Math) + (12 * #.uSRAM) = (36 * 4) + (36 * 0) +
(12 * 0) = 144.
```

We can see from section 2 of summary.result.rpt that the minimum period for the synthesized block is 4.635 ns, which is below the threshold of 6.353 ns from the demo design. This means we can safely integrate this block into the demo design and meet timing. SmartHLS 2024.1 also reports the usage for fabric and interface 4LUTs and DFFs separately.

Now close all project files.

13 Canny Edge Detection Block

Canny Edge detection is an image processing filter to get the edges of an image, as shown in Figure 27: Side-by-side comparison of original (left) and Canny Edge Filtered (right) image. The left image is the original, and the right image is after running the Canny edge detection filter.

Figure 27: Side-by-side comparison of original (left) and Canny Edge Filtered (right) images

The Canny edge detection algorithm consists of 4 cascading filters shown in Figure 28. The first filter is the Gaussian Blur which we covered in the previous section. The second filter is a Sobel filter which we covered in the previous SmartHLS tutorial. The next filter is non-maximum suppression which thins out the edges produced by the Sobel filter. Lastly, the Hysteresis filter sharpens the relevant edges and throws away irrelevant ones.
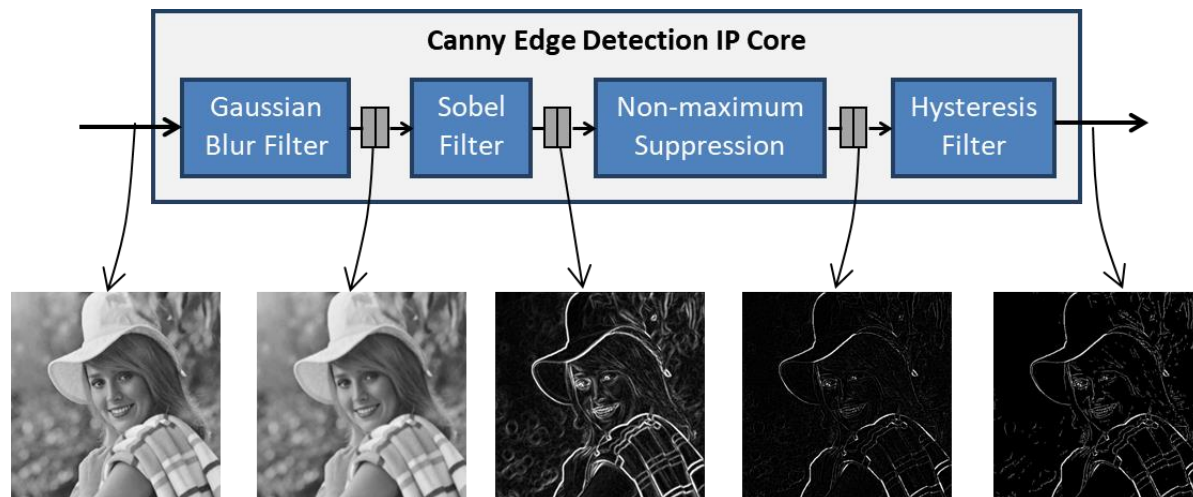

Figure 28: Canny Edge Detection Filter Block Diagram

We will not go into too much detail about each individual filter as the Gaussian and Sobel filters have already been covered and the techniques used for the rest of the filters are very similar.

Just note that each filter is loop pipelined with the use of input and output FIFOs and the LineBuffer class. All the filters also have a pipeline initiation interval (II) of 1.

Open *canny.cpp* in the Canny project.

- ∨ 🗁 Canny
  - › 📘 Includes
  - › 📄 bmp.hpp
  - › 📄 canny.cpp
  - › 📄 define.hpp
  - › 📄 gaussian_filter.cpp
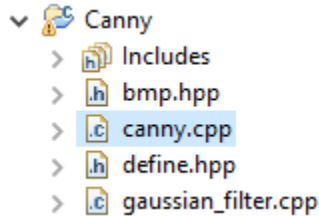
On line 6, we can see the top-level function is called canny, which calls each of the four loop pipelined filters in sequence with FIFOs passed between them. This function also has the dataflow pragma. The dataflow pragma causes the four sub-functions to overlap their execution and is ideal for generating a design where multiple functions are connected to operate as a single pipeline. To learn more about the dataflow pragma, see the SmartHLS Documentation.

```cpp
void canny(hls::FIFO<unsigned char> &input_fifo,
           hls::FIFO<unsigned char> &output_fifo) {
#pragma HLS function top
#pragma HLS function dataflow

#ifndef __SYNTHESIS__
    // For software, the fifo depth has to be larger.
    hls::FIFO<unsigned char> output_fifo_gf(WIDTH * HEIGHT * 2);
    hls::FIFO<unsigned short> output_fifo_sf(WIDTH * HEIGHT * 2);
    hls::FIFO<unsigned char> output_fifo_nm(WIDTH * HEIGHT * 2);
#else
    hls::FIFO<unsigned char> output_fifo_gf(/* depth = */ 2);
    hls::FIFO<unsigned short> output_fifo_sf(/* depth = */ 2);
    hls::FIFO<unsigned char> output_fifo_nm(/* depth = */ 2);
#endif
    gaussian_filter(input_fifo, output_fifo_gf);
    sobel_filter(output_fifo_gf, output_fifo_sf);
    nonmaximum_suppression(output_fifo_sf, output_fifo_nm);
    hysteresis_filter(output_fifo_nm, output_fifo);
}
```

The testbench for the Canny design on line 108 is similar to the Gaussian Filter testbench, however this design has an extra software implementation to compare against the hardware optimized version. The testbench checks that the software output, hardware output and golden output are all equal during co-simulation.

```cpp
// output validation
for (i = 0; i < HEIGHT; i++) {
    for (j = 0; j < WIDTH; j++) {
```

```
            unsigned char sw = hysteresis_output_golden[i][j];
            unsigned char gold = golden_output_image->r;
            assert(sw == gold);

            unsigned char hw = output_fifo.read();
            output_image_ptr->r = hw;
            output_image_ptr->g = hw;
            output_image_ptr->b = hw;

            if (sw != hw) {
                printf("ERROR: ");
                printf("i = %d j = %d sw = %d hw = %d\n", i, j, sw, hw);
            } else {
                matching++;
            }

            output_image_ptr++;
            golden_output_image++;
        }
    }
```

Now generate the hardware ( ) and then open the summary.hls.canny.rpt file, go to section 2 and scroll to the right to verify that all of the four filter functions have an initiation interval of 1. As every filter in the top level function has an initiation interval of 1, the entire pipeline then has an initiation interval of 1 as well.

```
+-----------------------------------------------------------------------------------------------+
| Function: nonmaximum_suppression takes 2075527 cycles                                         |
+-----------------------------------------+---+------------+-------------------+----+---------------+
| Loop                                    | … | Trip Count | Iteration Latency | II | Total Latency |
+-----------------------------------------+---+------------+-------------------+----+---------------+
| for.loop:nonmaximum_suppression.cpp:13:5 | … | 2075521    | 3                 | 1  | 2075523       |
+-----------------------------------------+---+------------+-------------------+----+---------------+


+-----------------------------------------------------------------------------------------------+
| Function: hysteresis_filter takes 2075527 cycles                                              |
+-----------------------------------------+---+------------+-------------------+----+---------------+
| Loop                                    | …| Trip Count | Iteration Latency | II | Total Latency |
+-----------------------------------------+---+------------+-------------------+----+---------------+
| for.loop:hysteresis_filter.cpp:14:5     | … | 2075521    | 3                 | 1  | 2075523       |
+-----------------------------------------+---+------------+-------------------+----+---------------+


+-----------------------------------------------------------------------------------------------+
| Function: gaussian_filter takes 2077448 cycles                                                |
+-----------------------------------------+---+------------+-------------------+----+---------------+
| Loop                                    | … | Trip Count | Iteration Latency | II | Total Latency |
+-----------------------------------------+---+------------+-------------------+----+---------------+
| for.loop:gaussian_filter.cpp:23:5       | … | 2077442    | 3                 | 1  | 2077444       |
+-----------------------------------------+---+------------+-------------------+----+---------------+


+-----------------------------------------------------------------------------------------------+
| Function: sobel_filter takes 2075527 cycles                                                   |
+-----------------------------------------+---+------------+-------------------+----+---------------+
| Loop                                    | … | Trip Count | Iteration Latency | II | Total Latency |
+-----------------------------------------+---+------------+-------------------+----+---------------+
| for.loop:sobel_filter.cpp:19:5          | … | 2075521    | 3                 | 1  | 2075523       |
+-----------------------------------------+---+------------+-------------------+----+---------------+
```

Now we uncomment FAST_COSIM in define.hpp, save, then rerun SmartHLS to generate the hardware (⚙) and then run co-simulation with ModelSim (▶). You should see the following output in the Console:

```
Retrieving hardware outputs from RTL simulation for canny function call 1.
Result: 5600
RESULT: PASS
...
Simulation time (cycles): 6,120
SW/HW co-simulation: PASS
```

Notice that although the pipeline is longer for Canny, the cycle latency of the simulation is about the same as that of the pipelined Gaussian design. This is because extra latency in a pipeline with initiation interval equal 1 is additive to the total cycle latency instead of multiplicative if the design was not pipelined.

Now close all project files.

13.1 Adding Inputs to a Series of Loop Pipelines

We want to add switch inputs to each filter in the Canny Edge Detection like the switch input we had for the Gaussian Filter. We can add 4 bool arguments to the top-level function and pass one to each filter.

Open the Canny_FIFO_Switch project double click the canny.cpp source file.

```
∨ 🖿 Canny_FIFO_Switch
    > 🗊 Includes
    > 🗎 bmp.hpp
    > 🗎 canny.cpp
    > 🗎 define.hpp
    > 🗎 gaussian_filter.cpp
```
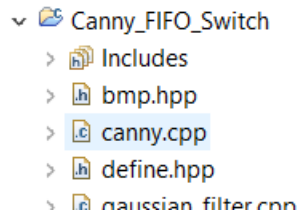
Notice on line 6 that canny now has 4 additional scalar inputs which represents the switch input that in turn goes to each filter.

```cpp
void canny(bool switch_0,
           bool switch_1,
           bool switch_2,
           bool switch_3,
           hls::FIFO<unsigned char> &input_fifo,
           hls::FIFO<unsigned char> &output_fifo) {
```

```
#pragma HLS function top
#pragma HLS function dataflow

#ifndef __SYNTHESIS__
        // For software, the fifo depth has to be larger.
        hls::FIFO<unsigned char> output_fifo_gf(WIDTH * HEIGHT * 2);
        hls::FIFO<unsigned short> output_fifo_sf(WIDTH * HEIGHT * 2);
        hls::FIFO<unsigned char> output_fifo_nm(WIDTH * HEIGHT * 2);
#else
        hls::FIFO<unsigned char> output_fifo_gf(/* depth = */ 2);
        hls::FIFO<unsigned short> output_fifo_sf(/* depth = */ 2);
        hls::FIFO<unsigned char> output_fifo_nm(/* depth = */ 2);
#endif
        gaussian_filter(switch_0, input_fifo, output_fifo_gf);
        sobel_filter(switch_1, output_fifo_gf, output_fifo_sf);
        nonmaximum_suppression(switch_2, output_fifo_sf, output_fifo_nm);
        hysteresis_filter(switch_3, output_fifo_nm, output_fifo);
}
```

Inside of the functions, for example on line 37 of hysteresis_filter.cpp, this switch is used to decide whether to pass through the pixel or apply filtering.

```
// if filter is off, pass pixel through
if (!on_switch) {
    output_fifo.write(current_pixel);
    continue;
}
```

Run "Compile Software to Hardware" (click the ⬚ button). Open the summary.hls.canny.rpt file and verify that there are now four more scalar interfaces for each of the switches in section 1.

```
+-------------+--------------------+-------------+-----------------+------------------+
| switch_0    | Scalar Argument    | switch_0    | 1               | input            |
+-------------+--------------------+-------------+-----------------+------------------+
| switch_1    | Scalar Argument    | switch_1    | 1               | input            |
+-------------+--------------------+-------------+-----------------+------------------+
| switch_2    | Scalar Argument    | switch_2    | 1               | input            |
+-------------+--------------------+-------------+-----------------+------------------+
| switch_3    | Scalar Argument    | switch_3    | 1               | input            |
+-------------+--------------------+-------------+-----------------+------------------+
```

Synthesize to FPGA (▣) and check the Fmax and resource usage.

```
====== 2. Timing Result of HLS-generated IP Core (top-level module: canny_top) ======

+--------------+---------------+-------------+-------------+----------+-------------+
| Clock Domain | Target Period | Target Fmax | Worst Slack | Period   | Fmax        |
+--------------+---------------+-------------+-------------+----------+-------------+
| clk          | 10.000 ns     | 100.000 MHz | 3.885 ns    | 6.115 ns | 163.532 MHz |
+--------------+---------------+-------------+-------------+----------+-------------+

The reported Fmax is for the HLS core in isolation (from Libero's post-place-and-route timing analysis).
```

When the HLS core is integrated into a larger system, the system Fmax may be lower depending on the critical path of the system.

====== 3. Resource Usage of HLS-generated IP Core (top-level module: canny_top) ======

```
+-------------------------+-------------------+--------+------------+
| Resource Type           | Used              | Total  | Percentage |
+-------------------------+-------------------+--------+------------+
| Fabric + Interface 4LUT*| 3947 + 720 = 4667 | 299544 | 1.56       |
| Fabric + Interface DFF* | 2107 + 720 = 2827 | 299544 | 0.94       |
| I/O Register            | 0                 | 1536   | 0.00       |
| User I/O                | 0                 | 512    | 0.00       |
| uSRAM                   | 33                | 2772   | 1.19       |
| LSRAM                   | 9                 | 952    | 0.95       |
| Math                    | 0                 | 924    | 0.00       |
+-------------------------+-------------------+--------+------------+
```

* Interface 4LUTs and DFFs are occupied due to the uses of LSRAM, Math, and uSRAM.
  Number of interface 4LUTs/DFFs = (36 * #.LSRAM) + (36 * #.Math) + (12 * #.uSRAM) = (36 * 9) + (36 * 0) + (12 * 33) = 720.

We can see from section 2 of summary.result.rpt that the minimum period for the synthesized block is 6.115 ns, which is below the threshold of 6.734 ns from the demo design. This means we can safely integrate this block into the demo design and meet timing.

# 14 Integrating Canny Edge Detection into SmartDesign and Generating a Bitstream

In this section, we are going to take the SmartHLS generated Canny Edge Detection block and import it into SmartDesign. This will showcase the flow for integrating SmartHLS generated Verilog Cores into Libero® SoC SmartDesign.

1. Open define.hpp in the Canny_FIFO_Switch project in the Project Explorer and check that FAST_COSIM is commented out. The functionality of this hardware block depends on knowing the WIDTH and HEIGHT of the input image.

   - Canny_FIFO_Switch
     - Includes
     - bmp.hpp
     - canny.cpp
     - define.hpp
     - gaussian_filter.cpp
     - gf_sw.cpp

   ```
   // uncomment this line to test on a smaller image for faster co-simulation
   //#define FAST_COSIM
   ```

2. Click the "Compile Software to Hardware" button 🖼 on the top toolbar.

3. Launch Libero® SoC 2024.1 and open the project: "Training1/Libero/Libero_training1/Libero_training1.prjx". The .prjx file was generated previously following the instruction: https://github.com/MicrochipTech/fpga-hls-examples/tree/main/Training1/Libero.

4. Navigate to the Design Hierarchy and search for "canny". Right click the canny_top design component and select Delete. This is to make sure there are no duplicated blocks before importing the new canny_top HDL+ block from SmartHLS.
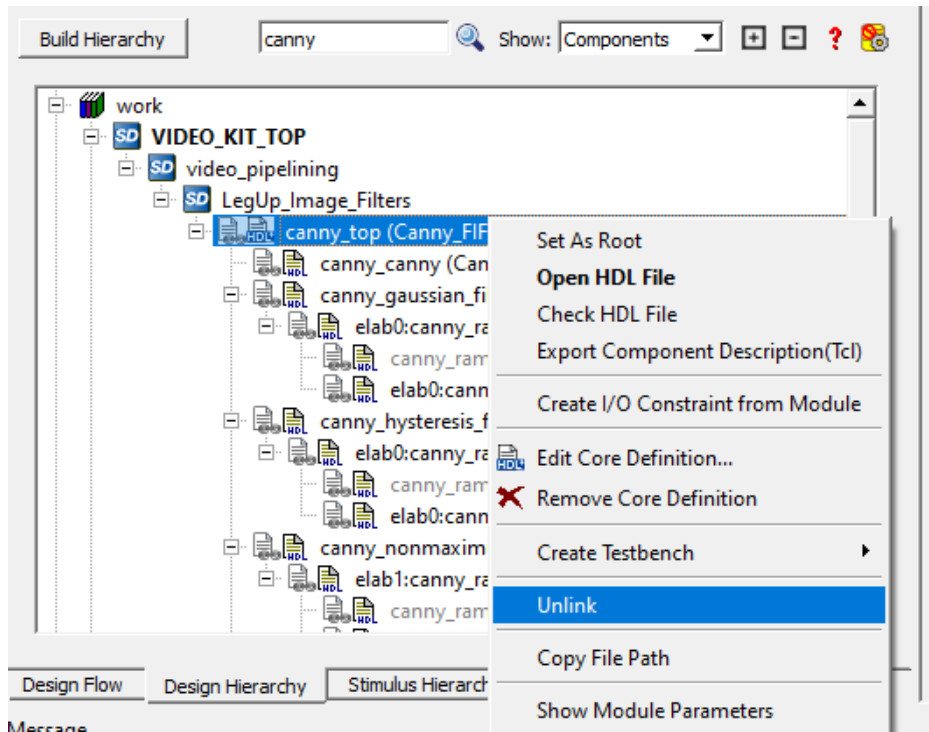
5. Without clearing the search, double click the LegUp_Image_Filters SmartDesign file to open it in the SmartDesign Canvas. Then find the canny_top_0 block which should now be missing and colored red.
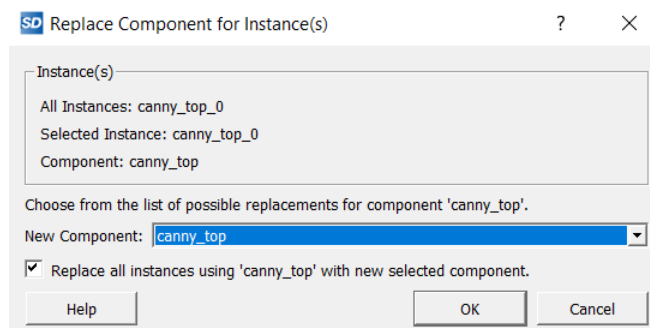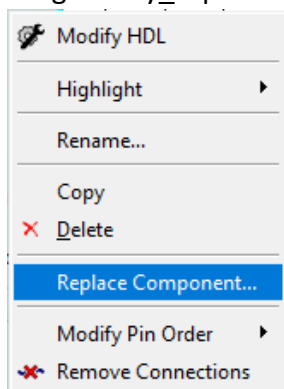
canny_top_0
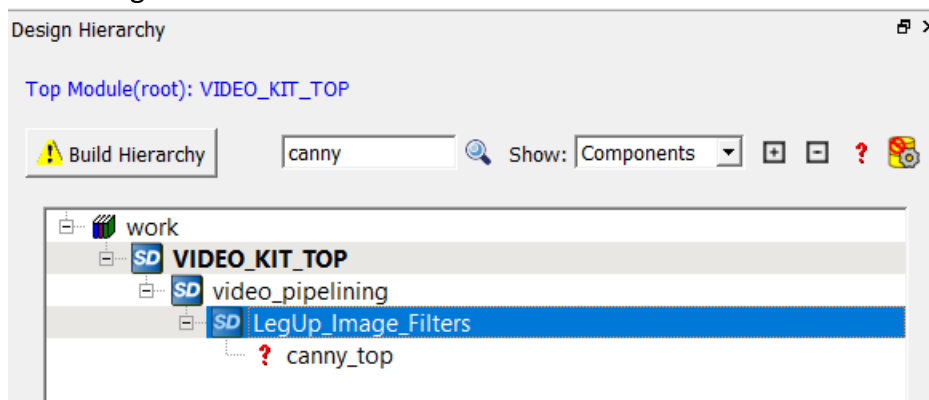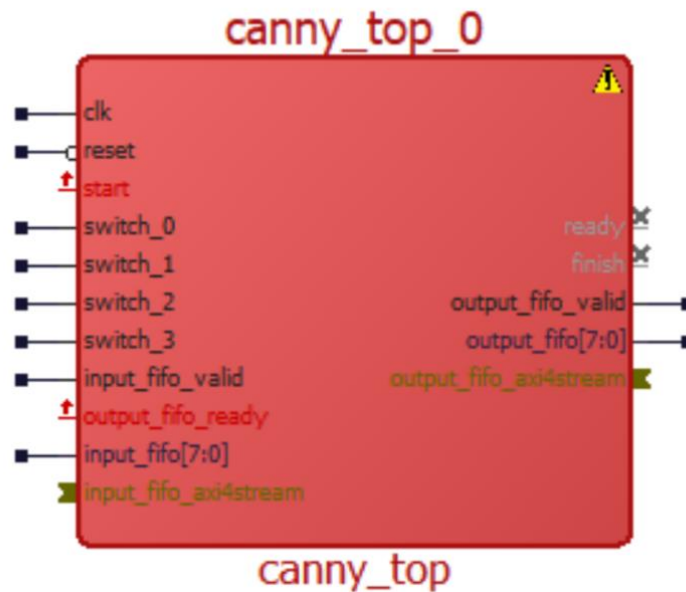
canny_top

6. On the top toolbar, click Project->Execute Script... and run the create_hdl_plus.tcl file from the Canny_FIFO_Switch SmartHLS project directory which will import the new canny_top into the design hierarchy. This will open a report window when it finishes. Make sure there are no errors and close the report window.
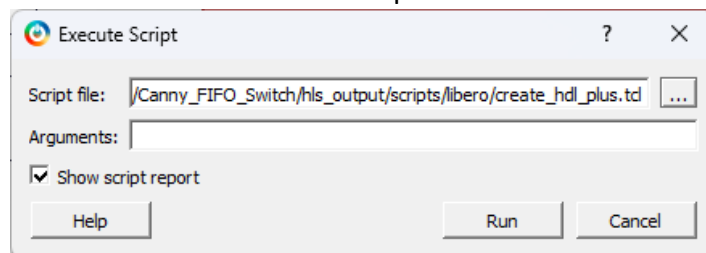

7. Right click on the canny_top_0 component, select "Replace Component..." and then replace it with the newly imported canny_top. Make sure to check "Replace all instances using 'canny_top' with new selected component."

8. After replacing the SmartDesign component, canny_top should no longer be red as shown below.
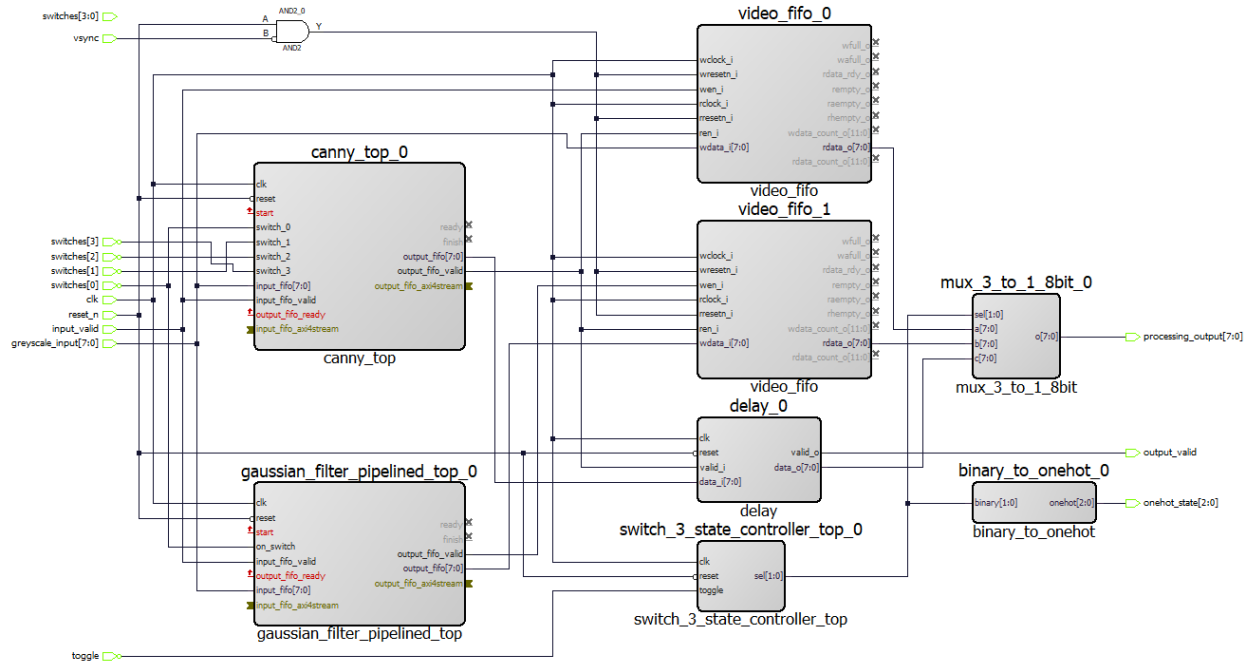
9. Click the "Generate Component" ( ) button in the SmartDesign toolbar for LegUp_Image_Filters and each parent component (video_pipelining, VIDEO_KIT_TOP).

10. Go to the Design Flow tab and double click Generate FPGA Array Data. This should take 1-2h to finish running.

11. The Mi-V soft processor receives configuration from the Video Control GUI running on the PC via the USB-UART. The Mi-V uses this configuration to control the Image/Video Processing block. To program the executable that runs on the Mi-V, double click "Configure Design Initialization Data and Memories" in the Design Flow tab. Libero should bring you to the "Design and Memory Initialization" tab. Go to the "Fabric RAMs" tab on the right:
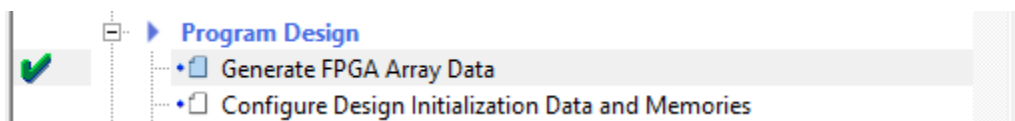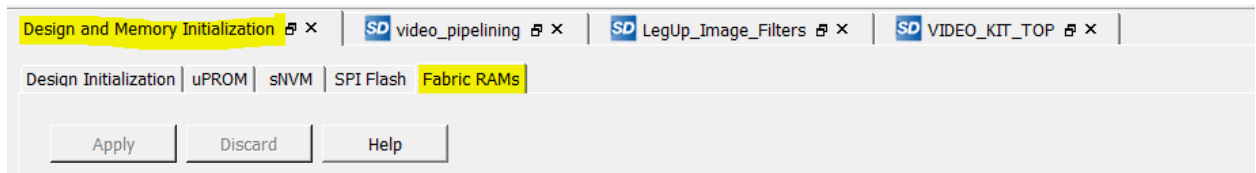
12. Check the "Filter out Inferred RAMs" checkbox and look for
PROC_SUBSYSTEM_0/PF_SRAM_AHBL_AXI_C0_0. It should be Logical Instance 8.
Double click on it.
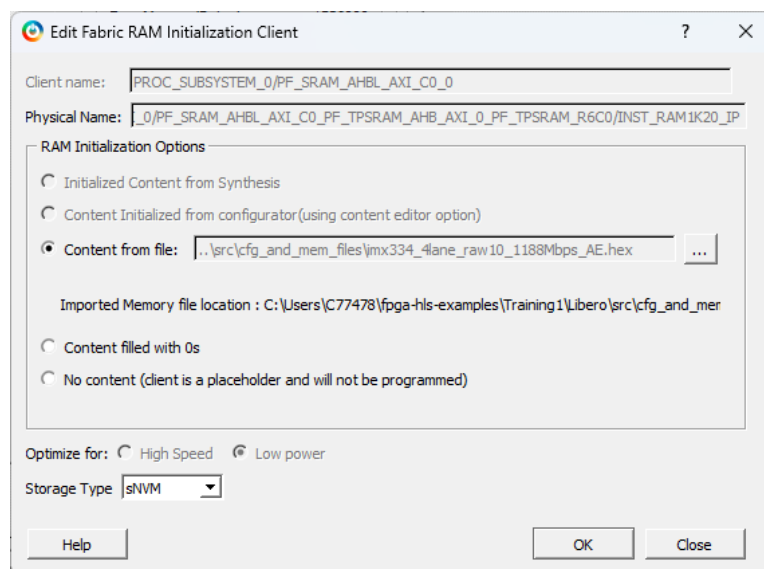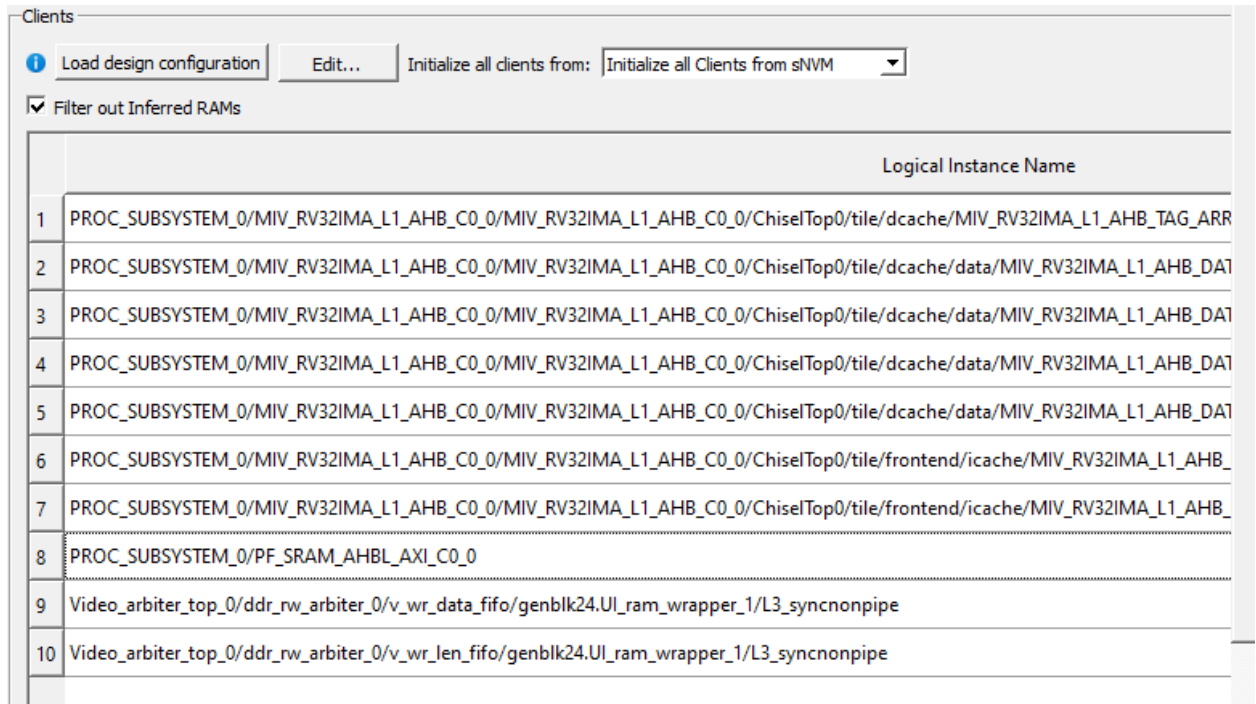
13. Click the Content from file and add the hex file as shown above. The hex file came from SoftConsole and contains the program executes on the Mi-V soft processor.

The file path may be different depending on whether you downloaded the Libero project as a .zip file or re-generated it using tcl.

Remember to select the "Use relative path" option when browsing to the memory file:
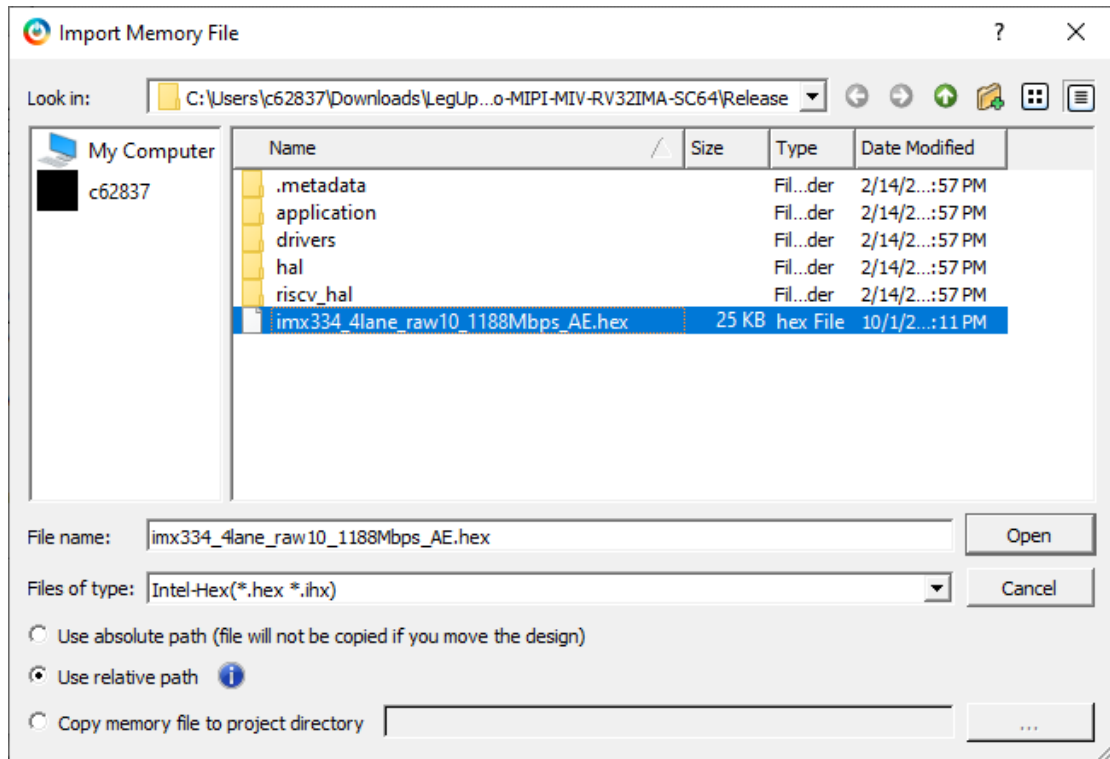
14. Click OK, then click "Apply" in the Design and Memory Initialization tab.

15. Under "Design Flow" double-click "Generate Bitstream".

16. With the same setup as Programming and Running Design on the PolarFire® Kit, double click "Run PROGRAM Action" to program the board.

17. You can also double-click "Export FlashPro Express job" to create an updated .job file.

Now the same design as presented in Section 6: Programming and Running Design on the PolarFire® Kit should now be programmed onto the board.

15   Conclusion

In this training we have described how to implement a variety of image processing hardware blocks in C++ using SmartHLS. We have compared the C++ HLS designs to equivalent RTL designs, and we found that the C++ code was much shorter and easier to understand than RTL. We have given an overview of the SmartHLS IDE and design flow steps such as software compile/run/debug, compile to hardware with SmartHLS, co-simulation with ModelSim, and finally synthesis, place, and route with Libero® SoC. We have covered SmartHLS reports and the schedule viewer for better understanding the hardware generated by SmartHLS. We have also covered SmartHLS optimization techniques like loop/function pipelining and dataflow, and given examples of the SmartHLS C++ library data types like FIFOs for streaming interfaces, ap_int, ap_fixpt, and LineBuffers. We have shown how hardware blocks designed with SmartHLS can be instantiated into a SmartDesign system. Finally, we have demonstrated that the SmartHLS-generated IP cores are functionally correct when running on the PolarFire® Video Kit board.