

SmartHLS™ Training Session 3: AXI Interfaces to DDR & Mi-V Soft Processor on the PolarFire® Video Kit

Training
September 20, 2022
Revision 4.0



1 Table of Contents

1	Table of Contents	2
2	Revision History	3
2.1	Revision 1.0	3
2.2	Revision 2.0	3
2.3	Revision 3.0	3
2.4	Revision 4.0	3
3	Overview	4
4	Smart High-Level Synthesis (SmartHLS™) Overview	5
5	Prerequisites	6
6	Setting up SmartHLS	7
7	Implementing Wide Multiply in SmartHLS	12
7.1	Design Overview	12
7.2	SmartHLS Implementation of Wide Multiply	13
7.2.1	Design Strategy	14
7.2.2	Helpful SmartHLS Features	14
8	AXI4 Protocol Primer	16
9	SmartHLS Accelerators on the PolarFire® Video Kit	19
9.1	Design Overview	19
9.2	Implementing AXI4 Target and AXI4 Initiator Using SmartHLS	20
9.2.1	Implementing AXI4 Target in SmartHLS	20
9.2.2	AXI4 Initiator Implementation	21
10	Programming and Running the Design on the PolarFire® Video Kit	29
11	Reference A: Wide Multiply Implementation	36
11.1	Simulation and Generated Hardware	38
11.2	Quality of Results	40
12	Reference B: AXI4 Target Implementation	43
12.1	Simulation and Generated Hardware	44
13	Appendix A: Integrating with SmartDesign and SoftConsole	47
13.1	Replacing Existing Component in SmartDesign	47
13.2	SmartDesign AXI4 Connection	50
13.3	Code Running on the Mi-V Soft Processor	56
14	Appendix B: Rotozoom and Texture Mapper	61

14.1	Code Running on the Mi-V Soft Processor	64
15	Appendix C: Rebuilding the SoftConsole Project.....	66

2 Revision History

The revision history describes the changes that were made to this document listed by revision, starting with the most recent publication.

2.1 Revision 1.0

First publication of the document.

2.2 Revision 2.0

Updated document for SmartHLS™ 2021.2 release.

2.3 Revision 3.0

Updated document for minor wordings and outdated figures.

2.4 Revision 4.0

Added Appendix C on Rebuilding the SoftConsole Project

3 Overview

Time Required: 4 hours

Goals of this Training:

- Get hands-on experience designing in SmartHLS™.
- Introduce the AXI protocol and explain how to generate AXI interfaces using SmartHLS.
- Show a SmartHLS design that has processor/accelerator communication.

Training Topics:

- Wide Multiply accelerator design:
 - Hands-on design with SmartHLS using fixed-point types to generate an arithmetic hardware block with a wide datapath and compare to an [RTL reference design](#).
 - Referring to previous SmartHLS trainings.
 - Referring to the [SmartHLS User Guide](#).
 - Referring to the [SmartHLS Github examples repository](#).
- AXI protocol and SmartHLS:
 - SmartHLS AXI target (also called AXI subordinate or slave) interface support and integration with a soft Mi-V processor.
 - SmartHLS AXI initiator (also called AXI manager or master) interface support and integration with DDR4 off-chip memory.
 - AXI initiator interface support for burst reads and burst writes to/from DDR4.
 - Integration into SmartDesign using CoreAXI4Interconnect.
 - Mi-V processor SoftConsole code for communicating with the SmartHLS hardware block through the AXI target interface.

4 Smart High-Level Synthesis (SmartHLS™) 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 design flow (formerly LegUp HLS), 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's 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.

The C++ code input to the SmartHLS Synthesis tool describes the behavior of one specific hardware functional block that exists in the overall FPGA design. A hardware engineer writes the C++ in SmartHLS to describe the hardware logic, not a software engineer (who would typically write C/C++ code for a processor to execute). The hardware engineer will need to handle advanced hardware issues outside of the SmartHLS tool such as the clock network, clock domain crossing, Triple Modular Redundancy (TMR) settings, etc.

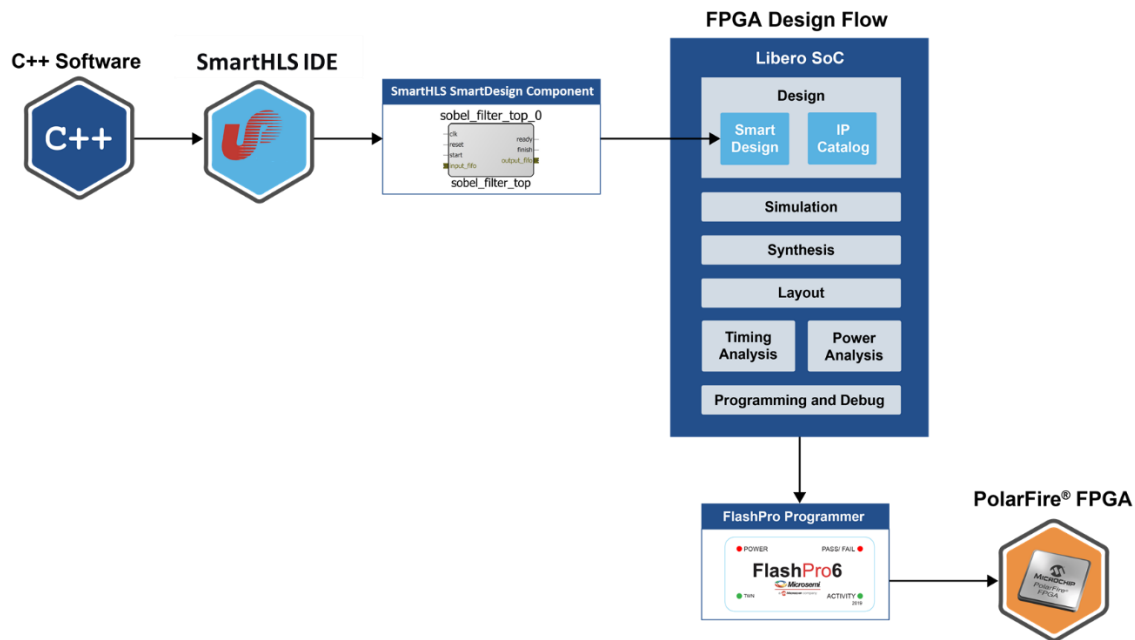


Figure 1: Smart High-level Synthesis Design Flow Targeting a PolarFire FPGA

5 Prerequisites

Before beginning this training, you should install the following software:

- Libero® SoC 2021.2 with ModelSim Pro
 - [Windows Download](#)
 - [Linux Download](#)
- SmartHLS 2021.2
 - [Windows Download](#)
 - [Linux Download](#)
- A terminal emulator such as PuTTY
 - [Windows Download](#)

This document uses the Windows versions of Libero SoC and SmartHLS.

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 Training3 folder for this training.
- Download the file [SmartHLS Training3 Libero.zip](#) (809MB)
MD5SUM: ae41d674b118851907f0c3aca43cbcd1

In Libero SoC 2021.2 you should download the following IPs in the IP Catalog as their configurators will be used in this training:

- Mi-V RV32 version 3.0.100
- CoreAXI4Interconnect version 2.8.103

The following hardware is required:

- PolarFire FPGA Video and Imaging Kit ([MPF300-VIDEO-KIT](#)).
- Monitor with an HDMI input.

In the [SmartHLS user guide](#), you should read [Section 1.13: SmartHLS C++ Library](#) up to and including [Section 1.13.6: Supported Operations in ap_int/ap_fixed](#), and [Section 1.15.3.3: AXI4 Slave Type](#). This knowledge will be directly applied in this training.

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.

6 Setting up SmartHLS

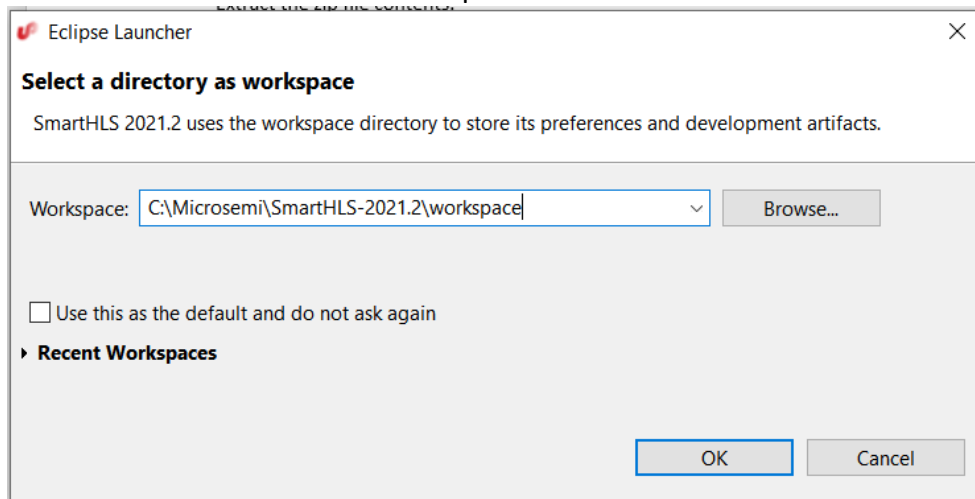
In this section we will set up the workspace and projects for SmartHLS 2021.2 used in this tutorial, as well as tool paths. We assume that SmartHLS 2021.2 and Libero® SoC 2021.2 have already been installed prior to this training. A Libero SoC install is needed for SmartHLS to run synthesis.



To set up SmartHLS 2021.2:

Download the zip file from github if you have not already (see Prerequisites). Extract the contents of the zip file. We will use the Training3 folder of the extracted content for this training.

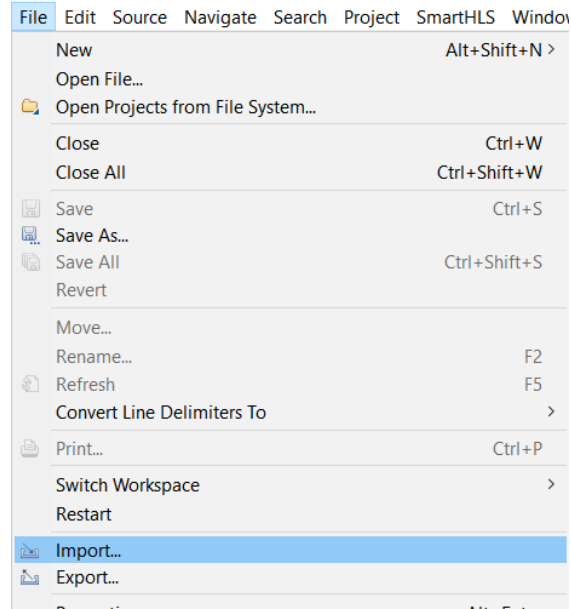
1. Open SmartHLS 2021.2 and choose a workspace.



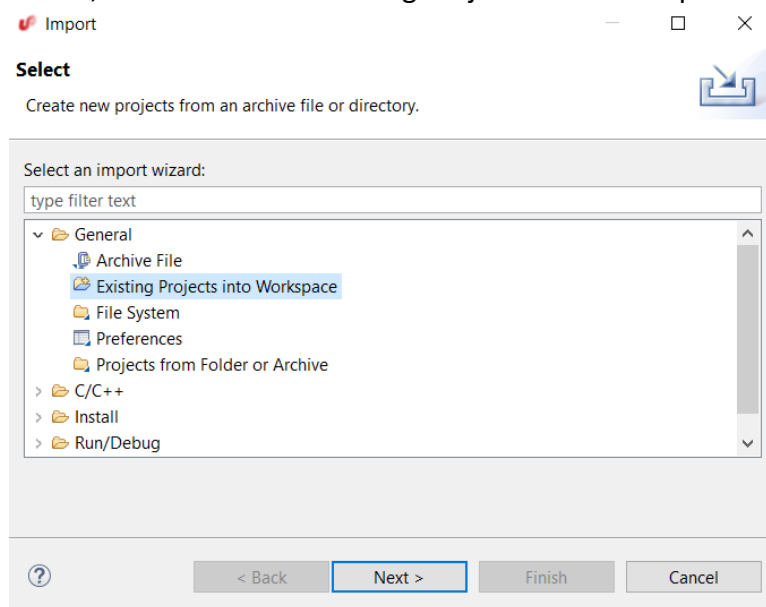
You may want to select a new folder so you can have a blank workspace for this training.

Warning: Make sure there are no spaces in your workspace path. Otherwise, SmartHLS will error out when running synthesis.

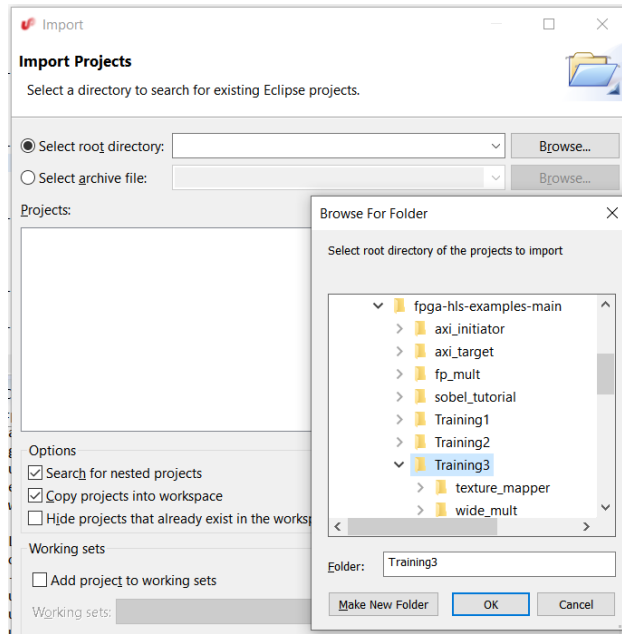
2. Select File -> Import...



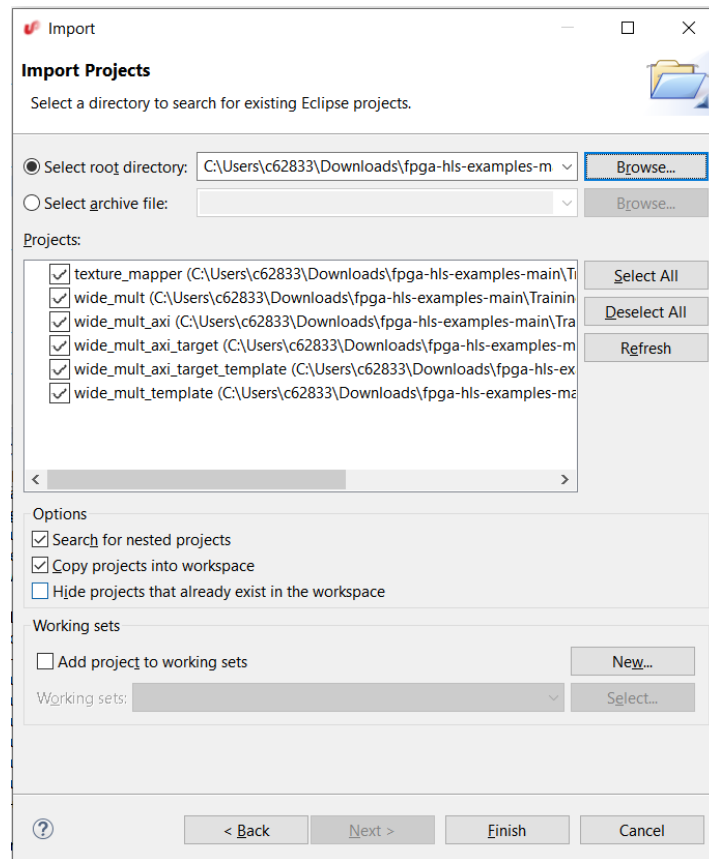
3. In the Import window, select General->Existing Projects into Workspace and then click Next.



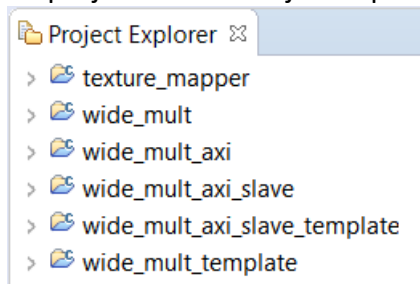
4. In the next step, select "Select root directory" and then click Browse... In the popup window browse to the Training3 directory and click OK.



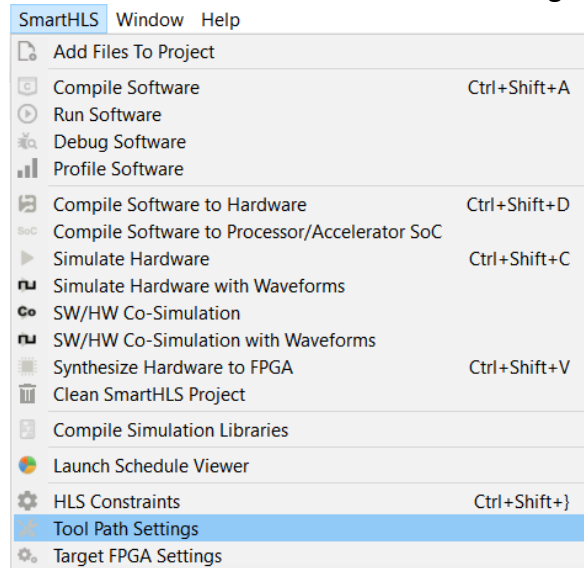
- Now in the Projects box you should see that all SmartHLS projects have been selected. Click Finish to import the projects.



6. After importing you should see 6 projects in the Project Explorer on the left.

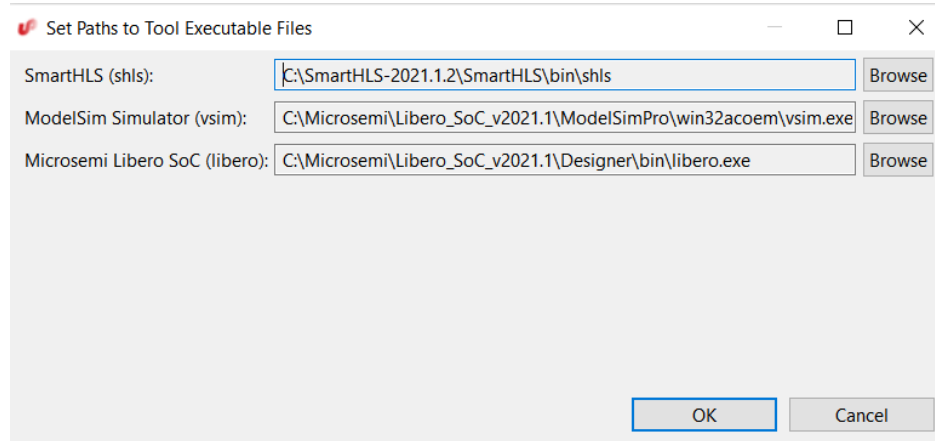


7. Next go to the top toolbar and select SmartHLS->Tool Path Settings



8. Make sure the tool paths to vsim.exe and libero.exe are properly set to:
C:\Microsemi\Libero_SoC_v2021.1\ModelSimPro\win32acoem\vsim.exe
C:\Microsemi\Libero_SoC_v2021.1\Designer\bin\libero.exe

Note: update these tool paths if your Libero is installed in a different location.



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.

7 Implementing Wide Multiply in SmartHLS

In this section, we will introduce the “wide multiply” design created by one of our FAEs using RTL. The wide multiply design performs a series of arithmetic operations on five fixed-point and integer inputs. We will recreate the same wide multiply design using SmartHLS, to apply what we learned in the previous trainings. We will then go over a reference SmartHLS implementation of the wide multiply.

Goals of this section:

1. Apply learning from previous trainings by designing with SmartHLS.
2. Read and familiarize yourself with the SmartHLS User Guide.
3. Implement and optimize a design based on requirements.

7.1 Design Overview

The wide multiply [RTL reference design](#) (WideMult) is designed by one of our FAEs for a real customer use case. The design’s requirements and design process can be found in this [design document](#). The design takes 128-bit fixed point integers and 64-bit integers and performs a fixed set of arithmetic operations.

The WideMult block implements the function $A + B + C * (D + 0.5 * (C * E))$ where the signal types and widths are:

- A is a 128-bit signed: 1 sign bit, 63 integer bits, and 64 fractional bits.
- B is a 64-bit unsigned: 64 integer bits, and no fractional bits.
- C is a 64-bit signed: 1 sign bit, 63 integer bits, and no fractional bits.
- D is a 128-bit signed: 1 sign bit, 63 integer bits, and 64 fractional bits.
- E is a 128-bit signed: 1 sign bit, 63 integer bits, and 64 fractional bits.
- Result (dout) is a 256-bit signed: 1 sign bit, 191 integer bits, and 64 fractional bits.

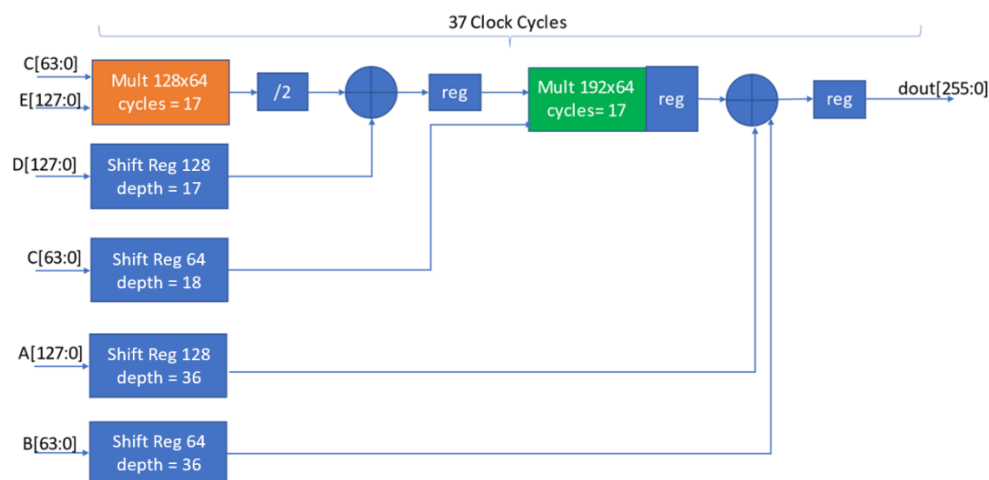


Figure 2: High-Level Architecture of RTL Wide Multiply Block

Figure 2 shows the high-level architecture of the RTL block. The only requirement of this design is to meet a Fmax target of 156.25 MHz. The 128x64 and 192x64 multiply functions are parameterized and pipelined. The RTL implementation has a pipeline initiation interval of 1 and a latency of 37 cycles, meaning we can receive a new set of inputs every clock cycle and the first output will occur 37 cycles after the inputs arrive. The Shift Register blocks, comprised of uSRAMs, are used as delay elements to line up the inputs to the pipeline stages where they are used. Using uSRAM saves logic resources. Registers are used between operations to improve Fmax. The divide-by-two after the 128x64 multiply is an arithmetic right shift-by-one.

Table 1 shows the resources used by the RTL implementation.

Table 1: Resource Usage of RTL Implementation

Fabric + Interface 4LUT	Fabric + Interface DFF	uSRAM 1K	MACC 18x18
1958 + 3288 (5246)	10839 + 3288 (14127)	34	80

7.2 SmartHLS Implementation of Wide Multiply



In this section, we want you to implement the same design using SmartHLS. We will aim to implement a pipelined HLS design with initiation interval of 1, with the same MACC usage and similar 4LUT, DFF, and uSRAM usage as presented in Table 1.

In SmartHLS you can find a base project in the *wide_mult_template* project:

```

v wide_mult_template
  > Includes
  > wide_mult.cpp
  > config.tcl
  > makefile
  > WideMultTopPf_Tb.sv

```

This project includes the *wide_mult.cpp* C++ source file defining an empty *wide_mult* top-level HLS function for you to fill in. The main function contains test cases to create a software testbench that can be used for Verification using co-simulation. The project also contains a SystemVerilog testbench to verify your hardware implementation. The testbench will check your implementation has generated the expected hardware interface and has a pipeline initiation interval of 1 and a latency of 43.

Your implementation of the *wide_mult* top-level HLS function should take around 10 lines of C++ code.

7.2.1 Design Strategy

The recommended design process for this design is to implement the operations using the arbitrary precision SmartHLS C++ library due to its support for wide fixed-point values. We recommend writing the code as you would in software using arbitrary precision values and verifying your code using the test cases provided in the main function of *wide_mult.cpp* via co-simulation. You can then further check the correctness by using the provided SystemVerilog testbench to verify the generated hardware interface and pipeline latency. Then you can check to see if the quality of results is already acceptable before trying to optimize. You may find that SmartHLS has automatically optimized your design such that you do not need to perform further optimizations.

The constraint to set WideMultTopPf_Tb.sv as the testbench should already exist. You can check by opening the configuration menu (⚙️). If it is not there, then add the testbench file and WideMultTopPf_Tb as the custom testbench using the method shown in the Custom Testbench section of the [SmartHLS Training 2 document](#) as shown in Figure 3.

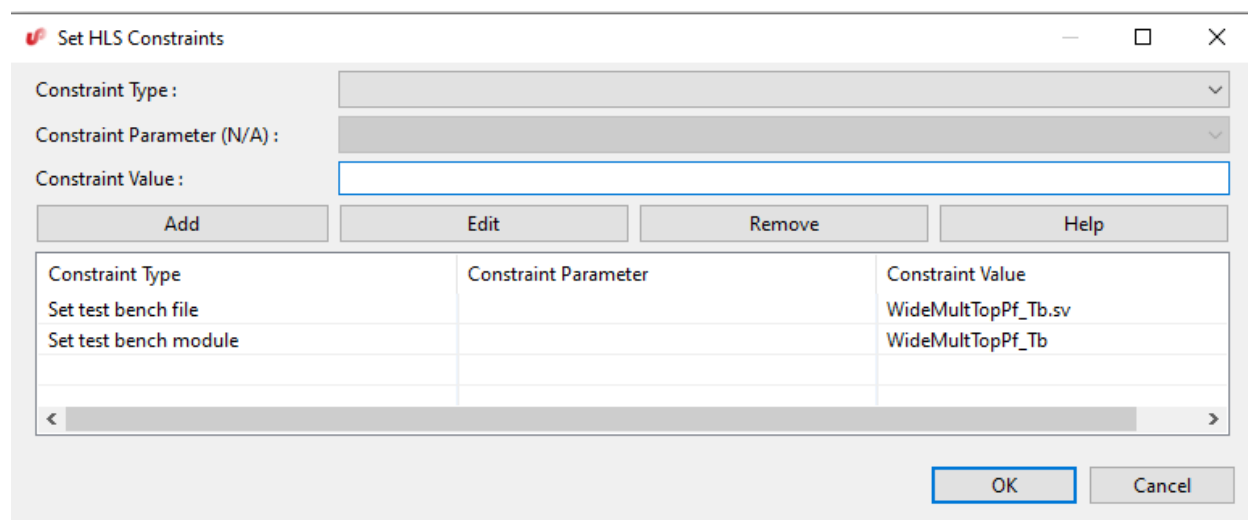


Figure 3: Adding custom SystemVerilog testbench.

7.2.2 Helpful SmartHLS Features

Many methods and libraries used in previous trainings can be applied in creating this wide multiply block. Some topics include the C++ arbitrary precision library, how top-level function interfaces map to hardware, and pipelining. Refer to the [SmartHLS User Guide](#) or [previous trainings](#) for more details on specific topics.

All of the necessary operators to implement the wide multiply operations are provided in the arbitrary precision library API. Look at [Section 1.13.2 through 1.13.6](#) in the SmartHLS User Guide if you have not already. This section provides information about the arbitrary precision library. Keep in mind that the operation performed in the divide-by-two operation in the

original RTL is implemented as an arithmetic right shift-by-one. The “>>” operator in C++ is a logical right shift and not an arithmetic right shift. You must use an arithmetic right shift operation to handle signed division properly. See: [Section 1.13.6: Supported Operations in ap_int/ap_fixed](#).

When you use the SmartHLS C++ arbitrary precision library, SmartHLS automatically handles sign extension, padding, and shifting when operating on types of different sizes. For example, if a variable B that has 64 integer bits and 0 fractional bits is added with a variable q with 192 integer bits and 64 fractional bits, SmartHLS will automatically shift and pad B to correctly add it to q. A representation of this addition is shown in Figure 4.

	q255	...	q128	q127	q126	...	q64	.	q63	...	q0
+	B63	...	B63	B63	B62	...	B0	.	0	...	0

Figure 4: Padded and Shifted Addition between *q* and *B*

Since there is wasted logic in having those extra padding bits in hardware, SmartHLS also automatically removes the padding when generating the hardware operations.

Because of the abstraction offered by the SmartHLS arbitrary precision library, the implementation of the wide multiply block should take no more than 10 short lines of C++ code.

You can set the [USE_FIFO_FOR_PIPELINE_REG](#) parameters to 1 in a custom configuration file in order to enable using uSRAMs for pipeline register. This will allow the resource usage to match the original design more closely. The steps to do so is similar to setting the STRENGTH_REDUCTION parameter as seen in the RGB2YCbCr section of the [Training 1 document](#).

A reference design exists in Section 11 that includes analysis of performance and resource usage compared to the original RTL. We will go over this implementation once you have tried to create this design on your own.

8 AXI4 Protocol Primer

In this section, we will talk about the AMBA AXI4 protocols, which we will use to allow the wide multiply core to communicate with the Mi-V soft processor present in the design. This is to allow the wide multiply core to accelerate this operation for the Mi-V. We recommend using an AXI4 interface for all processor to SmartHLS core communication because AXI is widely implemented, performant, and easy to use from the processor as a memory-mapped interface. AXI4 is also the interface we use in the upcoming SoC HLS flow.

The AXI specification is part of ARM AMBA, a family of micro controller buses first introduced in 1996. The first version of AXI (AXI3) was first included in [AMBA 3.0](#), released in 2003. [AMBA 4.0](#), released in 2010, includes the second version of AXI (AXI4).

In the past, the AMBA specifications have used the terms AXI master and AXI slave to define concepts in the AXI protocol. ARM has recently changed their terminology to AXI manager as AXI master and AXI subordinate as AXI slave. However, we will be using the following terminology: AXI master is now called AXI initiator and AXI slave is now called AXI target.

There are three types of AXI4 interfaces:

- AXI4 (or AXI4-Full): for high-performance memory-mapped requirements.
- AXI4-Lite: for simple, low-throughput memory-mapped communication (for example, to and from control and status registers).
- AXI4-Stream: for high-speed streaming data. For example, video pixels.

In this training, we will refer to the family of protocols as the AXI4 protocols, and the high-performance version of the protocol as AXI4, AXI4-Full, or the AXI4 protocol.

In this training, we will be using the AXI4 protocol and briefly touch on AXI4-Lite. We will not be covering AXI4-Stream in this training. If you are interested in AXI4-Stream, you can find more information in [AXI4-Stream](#) section of the AMBA 4.0 specification.

Some key features of the AXI4 protocols are:

- Separate address/control and data phases.
- Support for unaligned data transfers, using byte strobes.
- Uses burst-based transactions with only the start address issued.
- Separate read and write data channels that can provide low-cost Direct Memory Access.
- Support for issuing multiple outstanding addresses.
- Support for out-of-order transaction completion.
- Permits easy addition of register stages to provide timing closure.

The AXI4 protocol is burst-based and defines the following 5 independent transaction channels: read address, read data, write address, write data, and write response.

The address channel carries control information that describes the nature of the data to be transferred. The data is transferred between initiator (master) and target (slave) using either the write data channel when transferring data from the initiator to the target, or the read data channel to transfer data from the target to the initiator. In a write transaction, the target uses the write response channel to signal the completion of the transfer to the initiator.

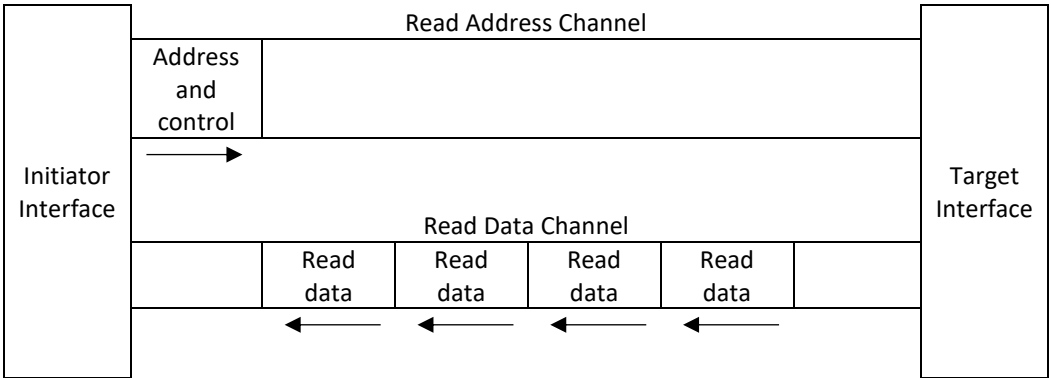


Figure 5: Read Channel Architecture

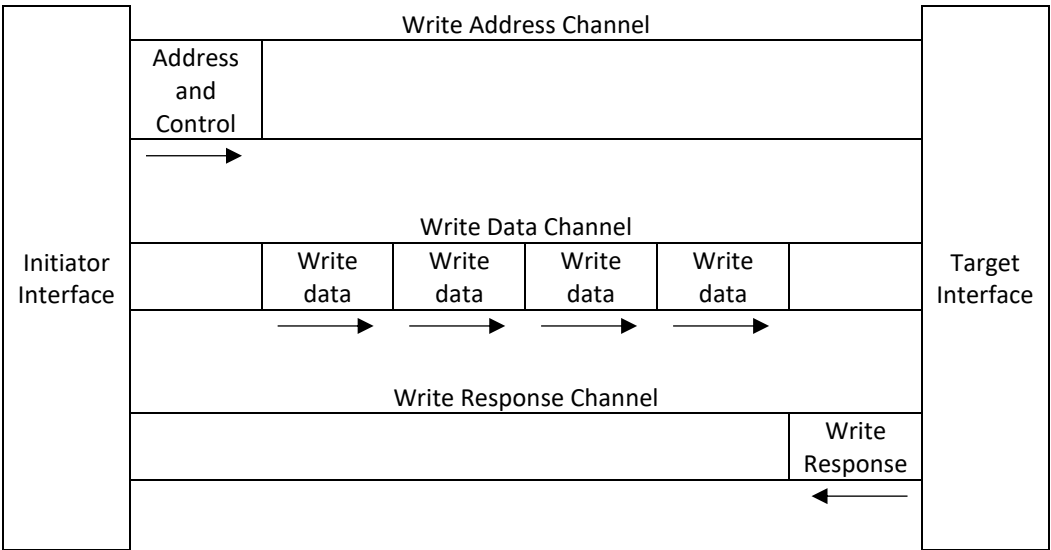


Figure 6: Write Channel Architecture

Figure 5 and Figure 6 show a typical read and write transaction using AXI4. AXI4 provides separate data and address connections for reads and writes, which allows simultaneous, bidirectional data transfer. Transfers are initiated by the Initiator Interface with a single address and can burst up to 256 words of data. At a hardware level, AXI4 allows a different clock for each AXI initiator-target pair. In addition, the AXI protocol allows the insertion of register slices (often called pipeline stages) to aid in timing closure.

Each type of AXI channels (read address, read data, write address, write data, write response)

has its own set of VALID and READY signals that provide a two-way handshake mechanism. These AXI channels can be modeled with the FIFO interfaces that we have seen in past trainings. Both the read data channel and the write data channel also include a LAST control signal to indicate the transfer of the final data item in a burst transaction.

The read data channel carries both the read data and the read response information from the target to the initiator, and includes the data bus, that can be 8, 16, 32, 64, 128, 256, 512, or 1024 bits wide.

The write data channel carries the write data from the initiator to the target, and includes the data bus, that can be 8, 16, 32, 64, 128, 256, 512, or 1024 bits wide, and a byte lane strobe signal indicating which bytes of the data are valid. There is one write strobe bit for each byte of the write data bus, with the lower bits of the strobe signal specifying the validity of the lower bytes of the data bus. Write data channel information is always treated as buffered, so that the initiator can perform write transactions without target acknowledgement of previous write transactions. A target uses the write response channel to respond to write transactions. All write transactions require completion signaling on the write response channel. The completion is signaled only for a complete transaction, not for each data transfer in a transaction.

There may be other control signals present in each of these channels, but the ones mentioned in the preceding paragraphs are enough to understand the high-level use of the protocol.

The AXI protocol requires the following relationships to be maintained:

1. A write response from the target to the initiator must always follow the last write transfer in the write transaction of which it is a part.
2. Read data must always follow the address to which the data relates.
3. Channel handshakes must conform to its dependencies.

Otherwise, the protocol does not define any relationship between the channels.

The AXI protocol provides response signaling for both read and write transactions.

The response codes are:

- OKAY: Normal access success. Indicates that a normal access has been successful. Can also indicate an exclusive access has failed.
- EXOKAY: Exclusive access okay. Indicates that either the read or write portion of an exclusive access has been successful.
- SLVERR: Target error. Used when the access has reached the target successfully, but the target wishes to return an error condition to the originating initiator.
- DECERR: Decode error. Generated, typically by an interconnect component, to indicate that there is no target at the transaction address.

AXI4-Lite is a protocol with a subset of AXI4's features. The most important difference is that AXI4-Lite has less control signals and does not support burst transfers.

9 SmartHLS Accelerators on the PolarFire® Video Kit

In this section, we present how a SmartHLS module can communicate with the Mi-V soft processor using the AXI4 protocol. We will modify the wide multiply design to create an AXI4 target enabled accelerator that can be used by the Mi-V processor to accelerate this wide multiply operation. We then show how we can use the same wide multiply accelerator to connect to DDR through an AXI4 initiator interface so that the accelerator can burst read inputs written into the DDR by the Mi-V and burst write results to DDR for the Mi-V to read.

Goals of this section:

1. How to create an HLS accelerator for a processor.
2. How to create AXI4 initiator and AXI4 target interfaces in SmartHLS.
3. More hands-on experience designing with SmartHLS.
4. How to burst read and write using an AXI4 initiator interface in SmartHLS.

9.1 Design Overview

Figure 7 shows the high-level hardware blocks used for the design. The blue arrows indicate AXI4 interface connections, where the arrow points from the AXI Initiator to the AXI Target. The blocks in blue are implemented in SmartHLS. We will only cover the Wide Multiply block in this training. More information on the Texture Mapper block can be found in Appendix B.

The Mi-V communicates with the host PC through a UART interface. The Mi-V processor can initiate AXI transactions to 3 targets: DDR, Wide Multiply and Texture Mapper. We connected the Mi-V processor to the two SmartHLS cores and a second AXI Interconnect using a 1-to-3 AXI Interconnect. The DDR is a target of two initiators, namely the Mi-V and the Wide Multiply, which is why we need a second interconnect. The second interconnect is a 2-to-1 AXI Interconnect connecting the Mi-V initiator interface from the first AXI Interconnect and the Wide Multiplier SmartHLS core to the DDR4 controller. The interconnects also provide clock domain crossing logic from the 50 MHz processor clock to the 100 MHz system clock, and from the 100 MHz system clock to the 200 MHz DDR clock.

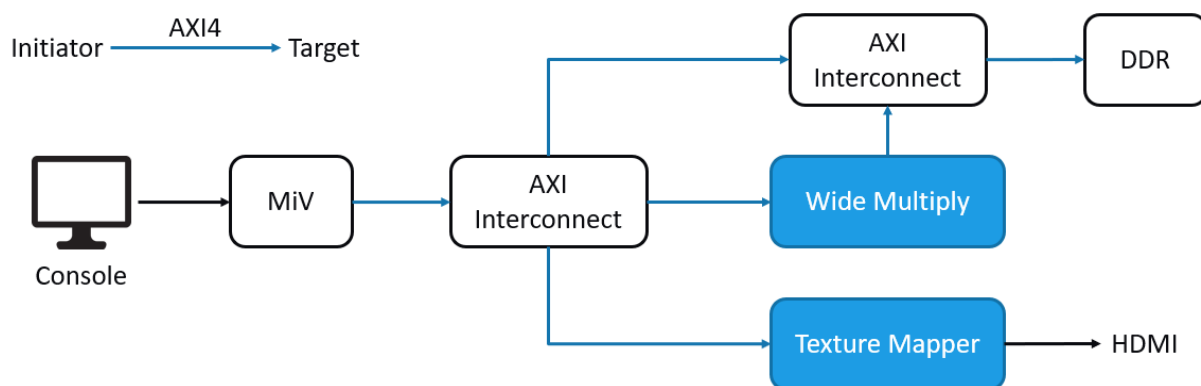


Figure 7: SmartHLS Accelerator System Diagram

9.2 Implementing AXI4 Target and AXI4 Initiator Using SmartHLS

We can use an AXI4 interface to allow the Mi-V to communicate with a SmartHLS module and use the SmartHLS core to accelerate slow processor operations. We can easily implement an AXI4 target interface using SmartHLS. When we want to run many inputs at once, we may want to use DDR to store many inputs and run them all at once in the accelerator. To accomplish this, we can use a AXI4 initiator interface from the SmartHLS accelerator to read from and write to DDR once the processor has filled the DDR with inputs.

Note, the AXI4 interfaces generated by SmartHLS is an enhanced version of AXI4-Lite that supports burst transactions. This burst enabled AXI4-Lite interface can properly interface with both AXI4 and AXI4-Lite in most FPGA use cases.

9.2.1 Implementing AXI4 Target in SmartHLS



In this section, we want you to implement an AXI target interface in SmartHLS to communicate with the Mi-V processor.

To start, you should see [Section 1.15.3.3](#) in the SmartHLS User Guide if you haven't already. This section includes information on how to create a AXI target interface in SmartHLS. You can also start with an example found in the SmartHLS Github [examples repository](#). Note, these resources may refer to the AXI4 initiator as the AXI4 master and the AXI4 target as the AXI4 slave.

You can find the base project in the *wide_mult_axi_target_template* project imported in Section 6. This project includes a C++ source file *wide_mult.cpp* with the same test cases from the original SystemVerilog test bench. It also includes a header file *axi4_target.h* that holds an empty struct that you should populate. This struct will be used to construct a AXI4 target channel by SmartHLS. Note, SmartHLS does not support using C++ arbitrary precision types inside of an AXI4 target struct. However, C++ arbitrary precision types should still be used for the wide multiply computations. This can be achieved by assigning the simple types in the target struct to the corresponding slices of the arbitrary precision type. See: [Section 1.13.4.1: Selecting and Assigning to a Range of Bits](#).

Next you will find the *initiator_layout.txt* file in the template project. This gives you the processor side memory map layout for communicating with the SmartHLS target. You should use the same layout between the processor and the SmartHLS target in order for the data transfer addressing to line up.

The run field found in the processor side layout is used to control the execution of the target. The run field will be set to 1 by the processor to signal to the SmartHLS core to begin computation on the inputs. When the SmartHLS core is done processing, it should set run to 0.

The processor will poll run until the run value becomes 0, then read the SmartHLS outputs.

SmartHLS will automatically generate a burst enabled 64-bit AXI4-Lite target interface and the required control logic when a global variable struct is declared as an AXI target interface. Again, this burst enabled AXI4-Lite interface is enough for most applications and can work with both AXI4-Full and AXI4-Lite interfaces in most FPGA use cases.

With the addition of an AXI4 target interface, the implementation function body should be roughly 50 lines, where the majority of the code is related to reading from and writing to the AXI4 target struct.

A reference design exists in Section 12. We will go over this implementation once you have tried to create this design on your own.

9.2.2 AXI4 Initiator Implementation

Now that we have seen an AXI4 target implementation that can communicate with the Mi-V in Section 12, we have a working accelerator. However, in the case where we do not want to manually enter inputs, we can write pre-set argument values to DDR via a second AXI initiator interface from the Mi-V, then tell the wide multiply core to fetch the values from DDR and write the result back to DDR for the Mi-V to read. To achieve this, we need to implement an AXI initiator interface from the wide multiply core to read and write to the DDR.

This can be done using the SmartHLS AXI interface library. Using this library, we need to manually implement the AXI4 initiator interface by specifying read and write requests with our desired addresses and control signals. We also need to manually read and check response codes when we complete a write transaction. More information on the AXI4 protocol can be found in Section 8.



Open the *wide_mult_axi* project and open the *axi4_target.h* file.

- ▼ wide_mult_axi
 - > Includes
 - > **axi4_target.h**
 - > wide_mult_axi.cpp
 - config.tcl
 - initiator_layout.txt
 - sc_main_c

This AXI target struct is very similar to the one we have seen before, except for two new control fields. The *base_addr* field is used by the Mi-V to pass in the DDR address information to the core. The *error* field is used by the core to send back an error code in case it reads any errors in the response from the DDR controller. This is also the exact same struct as found in *initiator_layout.txt*.

```

struct TargetLayout {
    uint32_t run;
    uint32_t base_addr;
    uint32_t error;
    uint32_t result0;
    ...
    uint32_t result7;
    uint32_t A0;
    ...
    uint32_t E3;
    uint32_t padding;
};

```



Next open the *wide_mult_axi.cpp* source file. On line 4, we have included the AXI initiator interface library.

```
#include <hls/axi_interface.hpp>
```

On line 21, we have moved the wide multiply operation back into its own function and added a `noinline` pragma. This allows sharing of the wide multiply between the “single” and “burst” mode circuits.

```

ap_fixpt<256, 192> wide_mult(ap_fixpt<128, 64> A, ap_uint<64> B, ap_int<64> C,
                             ap_fixpt<128, 64> D, ap_fixpt<128, 64> E) {
    #pragma HLS function noinline
    // return A + B + C * (D + 0.5 * (C * E));

    ap_fixpt<192, 128> m, n, o;
    ap_fixpt<256, 192> p, q;

    m = C * E;
    n = m.ashr(1);
    o = D + n;
    p = C * o;
    q = A + p;
    return B + q;
}

```

Starting on line 37, we have the new top-level function for this design. This function has an `AxiInterface` object passed into it by reference. This object in the function signature will generate a burst enabled AXI4-Lite initiator interface in hardware. As template arguments, we provide the address width, data bus width, and strobe width as `ap_uints`.

```

void wide_mult_axi(AxiInterface<ap_uint<32>, ap_uint<32>, ap_uint<4>>
&initiator) {
    #pragma HLS function top

```

In the function body, we have repurposed the `run` field to also signify when the module should run in “single” or “burst” mode. When `run` is 0, the module is finished running or idle. When the module is run in single mode, the module runs the same operations as we have seen in Section 12.

```
if (data.run == 0) // idle
```

```

    return;

if (data.run == 1) { // single
    A(31, 0) = data.A0;
    ...
    E(127, 96) = data.E3;

    r = wide_mult(A, B, C, D, E);

    data.result0 = r(31, 0);
    ...
    data.result7 = r(255, 224);

    data.run = 0;

```

For any other value of run, the value is taken as the burst size and that number of runs are performed by reading and writing values to and from the DDR through the AXI initiator interface.

```

    } else { // burst
        // get data from DDR
        const uint32_t BURST_SIZE = data.run;
        const uint32_t NUM_INPUT_WORDS = 16;
        const uint32_t NUM_OUTPUT_WORDS = 8;
        const uint32_t WORD_SIZE = 4;
        uint32 addr = data.base_addr;

        ap_uint<2> bresp;
        ap_uint<2> error = 0;

        for (uint32_t i = 0; i < BURST_SIZE; i++) {

```

The SmartHLS core reads input from DDR by first issuing a single read request in the read address channel by using the `axi_m_read_req` function. This function takes the same address width, data bus width, and strobe width template parameters as the initiator object it operates on. As arguments, we provide the initiator interface object, the starting address to read from, and the number of data words to burst read, which corresponds to sixteen for inputs A-E. The address must be the same type as the provided address width template parameter, and the word size will match the data bus width template parameter. We then increment the address by the byte size of the data read in this transaction. We can now read the sixteen words from the read data channel by making a call to `axi_m_read_data`. We provide the same address width and data bus width parameters and provide the initiator interface object as an argument. This will read a single word from the data read channel per call and return it. For the sixteen-word burst we requested, we will call `axi_m_read_data` sixteen times and store the values in our input variables A-E.

```

        axi_m_read_req<ap_uint<32>, ap_uint<32>, ap_uint<4>>(
            initiator, addr, ap_uint<9>(NUM_INPUT_WORDS));
        addr += NUM_INPUT_WORDS * WORD_SIZE;

        A(31, 0) = axi_m_read_data<ap_uint<32>, ap_uint<32>>(initiator);
        ...

```

```
E(127, 96) = axi_m_read_data<ap_uint<32>, ap_uint<32>>(initiator);
```

We then run the wide multiply with inputs fetched from DDR.

```
r = wide_mult(A, B, C, D, E);
```

To write the result back into DDR, first we need to send a write request in the write address channel. This is done by calling the `axi_m_write_req` function. Again, it takes the same address width, data bus width, and strobe width template parameters as the initiator object it operates on. The address and number of words to write are provided as arguments. We then increment the address by the byte size of the data read in this transaction. Now we write the wide multiply result as eight 32-bit words to the write data channel by making a call to `axi_m_write_data`. As arguments, we provide the initiator interface object, the data, the strobe signal for each byte of the data sent. Lastly, we provide a flag signaling if the requested transaction has ended. For burst transactions of eight words as requested, the last call should set the flag to 1 while all others have it set to 0.

```
axi_m_write_req<ap_uint<32>, ap_uint<32>, ap_uint<4>>(
    initiator, addr, ap_uint<9>(NUM_OUTPUT_WORDS));
addr += NUM_OUTPUT_WORDS * WORD_SIZE;

axi_m_write_data<ap_uint<32>, ap_uint<32>, ap_uint<4>>(
    initiator, ap_uint<32>(r(31, 0)), ap_uint<4>(0xF), /*is_last=*/0);
...
axi_m_write_data<ap_uint<32>, ap_uint<32>, ap_uint<4>>(
    initiator, ap_uint<32>(r(255, 224)), ap_uint<4>(0xF), /*is_last=*/1);
```

After a full write transaction, we manually collect the write response by making a call to `axi_m_write_response` and aggregate it in an error variable.

```
bresp = axi_m_write_resp(initiator);
error |= bresp;
```

Make sure to read and write the correct number of data words or responses as specified in the read and write requests, otherwise the channel might fill, and the design will hang.

For this design, we chose the data bus width to be 32-bits to conceptually line up with the 32-bit values written into the DDR by the Mi-V. For ease of implementation, you can also make the data bus wider in order to get more data per read. For example, if the data bus was 256-bits wide, you can read all inputs with two `axi_m_read_data` calls and write the output in a single `axi_m_write_data` call.

Lastly, once the main loop is finished, the module writes the aggregated error code into its target memory and then sets `run` to 0, signaling to the Mi-V that it can read the results back from the DDR.

```
// Send aggregate response code back to MiV
data.error = error;

// Signal accelerator finished
data.run = 0;
```


9.2.2.1 Simulation and Generated Hardware



In order to simulate the design in co-simulation, the operations of an AXI4 target must be modeled in software. On line 161 you will find the main function with the co-simulation testbench. Inside the testbench, there are 47 test cases taken from the *WideMultTopPf_Tb.sv* testbench.

```
int main() {  
  
    // input test vectors  
    const int NUM_INPUTS = 47;  
  
    ...
```

Starting on line 316, we prepare some constants and a SmartHLS initiator object to pass into the top-level interface. The `axi_if` object can be thought of as a FIFO with the AXI channels described in Section 8. We provide the FIFO depth as the argument. We must make sure that it is deep enough to hold all inputs before writing to it.

```
// Prepare for AXI initiator interface reads/writes  
const int NUM_READS_PER_INPUT = 16;  
const int NUM_WRITES_PER_INPUT = 8;  
const int WORD_SIZE = 4;  
  
AxiInterface<ap_uint<32>, ap_uint<32>, ap_uint<4>> axi_if(NUM_INPUTS *  
NUM_READS_PER_INPUT);
```

We will write all of the input words to be read by the initiator using the `prepare_axi_read` function, found on line 150. This function provides an easy interface to set necessary control signals and write the data to the initiator read data channel.

```
void prepare_axi_read(AxiInterface<ap_uint<32>, ap_uint<32>, ap_uint<4>> &axi_if,  
    ap_uint<32> input, int last = 0) {  
  
    RdDataSignals<ap_uint<32>> r_sig;  
    r_sig.data = input;  
    r_sig.resp = 0;  
    r_sig.last = last;  
    axi_if.r.write(r_sig);  
  
}
```

On line 323, we use the `prepare_axi_read` function to write all inputs to the initiator read data channel, making sure to set the last signal on the last read per transaction. We also pre-write the write response for the accelerator for each transaction so that the accelerator can read it after a write transaction to the interface.

```
for (int i = 0; i < NUM_INPUTS; i++) {  
    prepare_axi_read(axi_if, A[i](31, 0));  
    ...  
    prepare_axi_read(axi_if, E[i](127, 96), /*last=*/1);  
}
```

```

    WrRespSignals b_sig;
    axi_if.b.write(b_sig);
}

```

Next, we set the run field in the target memory to the number of inputs we have written and then set the base address to 0. We then run the accelerator to produce the results.

```

// AXI target inputs
data.run = NUM_INPUTS;
data.base_addr = 0;

// Run the top-level function that will be synthesize to hardware.
wide_mult_axi(axi_if);

```

Starting on line 352, we check that the addresses read from and written to are correct by reading from the read and write address channels and checking each address line up.

```

int mismatch_addr_cnt = 0;

ap_uint<32> addr = 0;

// Check that the read and write addresses were as expected.
for (int i = 0; i < NUM_INPUTS; ++i) {
    if (addr != axi_if.ar.read().addr)
        mismatch_addr_cnt++;
    addr += NUM_READS_PER_INPUT * WORD_SIZE;
    if (addr != axi_if.aw.read().addr)
        mismatch_addr_cnt++;
    addr += NUM_WRITES_PER_INPUT * WORD_SIZE;
}

```

Next, we check that the result data is correct by reading them from the write data channel and comparing them with the pre-generated results taken from the SystemVerilog testbench.

```

// Read all of write data.
int mismatch_cnt = 0;
for (int i = 0; i < NUM_INPUTS; ++i) {
    ap_fixpt<256, 192> result;
    result(31, 0) = axi_if.w.read().data;
    ...
    result(255, 224) = axi_if.w.read().data;
    std::cout << "A: " << A[i].raw_bits() << " B: " << B[i]
        << " C: " << C[i] << " D: " << D[i].raw_bits()
        << " E: " << E[i].raw_bits() << "\n";
    std::cout << "Expected: " << expected_results[i].raw_bits()
        << ", Actual: " << result.raw_bits() << "\n";
    if (result != expected_results[i]) {
        mismatch_cnt++;
        std::cout << "ERROR: mismatch\n";
    } else {
        std::cout << "Matched!\n";
    }
}
}

```

We report the final results and return any errors found in the testbench for co-simulation to check.

```
std::cout << "Incorrect addresses: " << mismatch_addr_cnt << "\n";
std::cout << "Mismatches: " << mismatch_cnt << "\n";
if (mismatch_addr_cnt || mismatch_cnt)
    std::cout << "FAIL\n";
else
    std::cout << "PASS\n";
return mismatch_addr_cnt || mismatch_cnt;
```



Next, run the co-simulation (🏃). You should see the following in the output:

```
Incorrect addresses: 0
Mismatches: 0
PASS
...
Number of calls:          1
Cycle latency:           3,403
SW/HW co-simulation: PASS
```



Now generate the hardware for this design (🔧). You will the same AXI Slave (target) interface generated as before, but now there is also an AXI Master (initiator) interface.

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

RTL Interface Generated by SmartHLS				
C++ Name	Interface Type	Signal Name	Signal Bit-width	Signal Direction
	Control	clk	1	input
		finish	1	output
		ready	1	output
		reset	1	input
		start	1	input
initiator	AXI Master	initiator_ar_addr	32	output
		initiator_ar_burst	2	output
		initiator_ar_len	8	output
		initiator_ar_ready	1	input
		initiator_ar_size	3	output
		initiator_ar_valid	1	output
		initiator_aw_addr	32	output
		initiator_aw_burst	2	output
		initiator_aw_len	8	output
		initiator_aw_ready	1	input
		initiator_aw_size	3	output
		initiator_aw_valid	1	output
		initiator_b_resp	2	input
		initiator_b_resp_ready	1	output
		initiator_b_resp_valid	1	input
		initiator_r_data	32	input
		initiator_r_last	1	input

		initiator_r_ready	1	output
		initiator_r_resp	2	input
		initiator_r_valid	1	input
		initiator_w_data	32	output
		initiator_w_last	1	output
		initiator_w_ready	1	input
		initiator_w_strb	4	output
		initiator_w_valid	1	output
data	AXI Slave	axi_s_ar_addr	32	input
		axi_s_ar_burst	2	input
		axi_s_ar_len	8	input
		axi_s_ar_ready	1	output
		axi_s_ar_size	3	input
		axi_s_ar_valid	1	input
		axi_s_aw_addr	32	input
		axi_s_aw_burst	2	input
		axi_s_aw_len	8	input
		axi_s_aw_ready	1	output
		axi_s_aw_size	3	input
		axi_s_aw_valid	1	input
		axi_s_b_resp	2	output
		axi_s_b_resp_ready	1	input
		axi_s_b_resp_valid	1	output
		axi_s_r_data	64	output
		axi_s_r_last	1	output
		axi_s_r_ready	1	input
		axi_s_r_resp	2	output
		axi_s_r_valid	1	output
		axi_s_w_data	64	input
		axi_s_w_last	1	input
		axi_s_w_ready	1	output
		axi_s_w_strb	8	input
		axi_s_w_valid	1	input

10 Programming and Running the Design on the PolarFire® Video 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 8. We will use the HDMI 1.4 TX (J2) to display output on a computer monitor and the USB-UART (J12) for bitstream programming, and communication with the Mi-V.

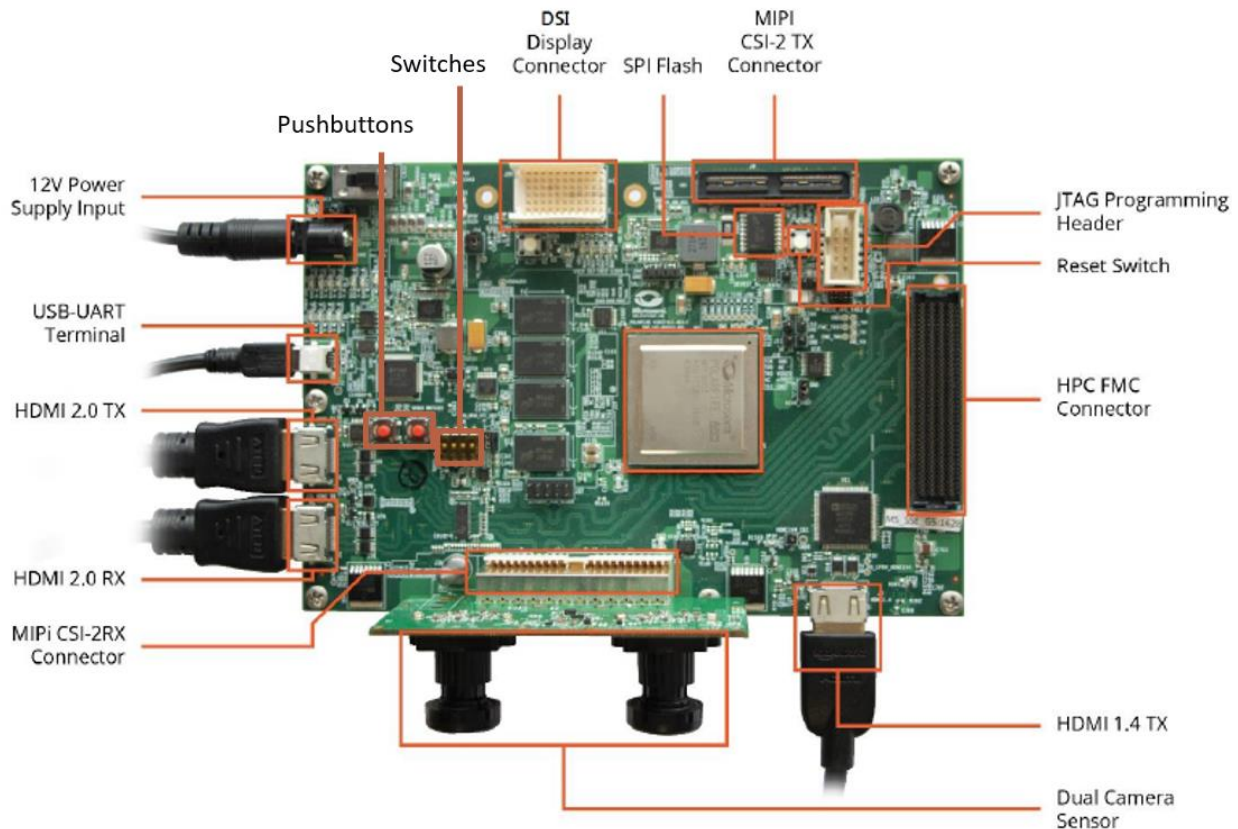
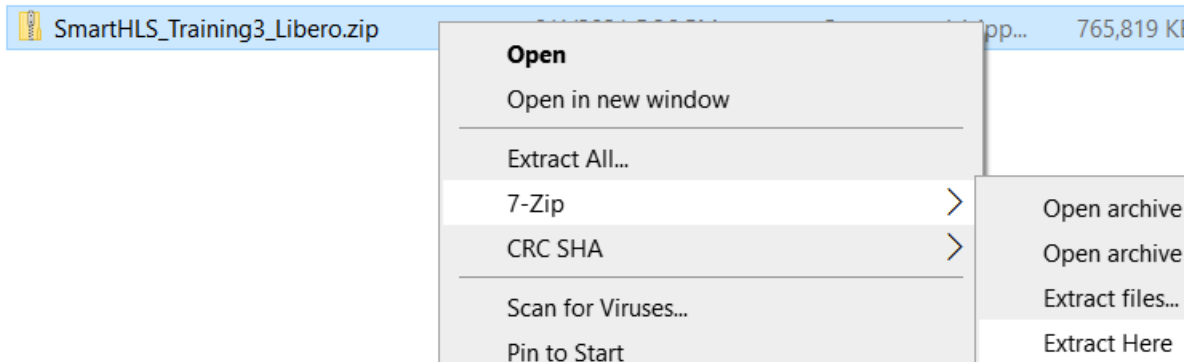


Figure 8: PolarFire® Video and Imaging Kit Peripherals

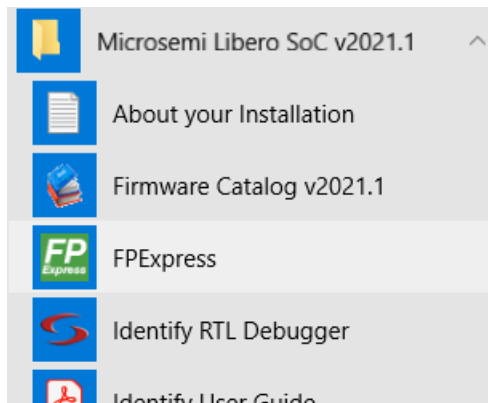


To program the design to the PolarFire board:

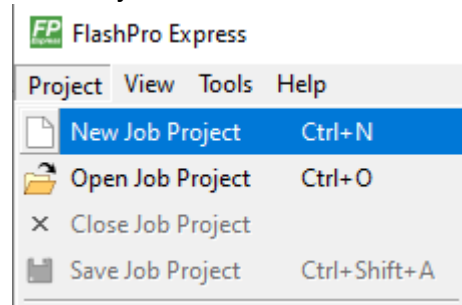
1. If you have not already, download a terminal emulator like PuTTY. We will use this to communicate with the Mi-V via UART.
2. If you have not already, download the file SmartHLS_Training3_Libero.zip. Extract the zip file contents. On Windows you will need to extract the project to a directory with a short name (such as C:\Downloads or C:\Workspace) and extract with 7-Zip to avoid issues with long filenames:



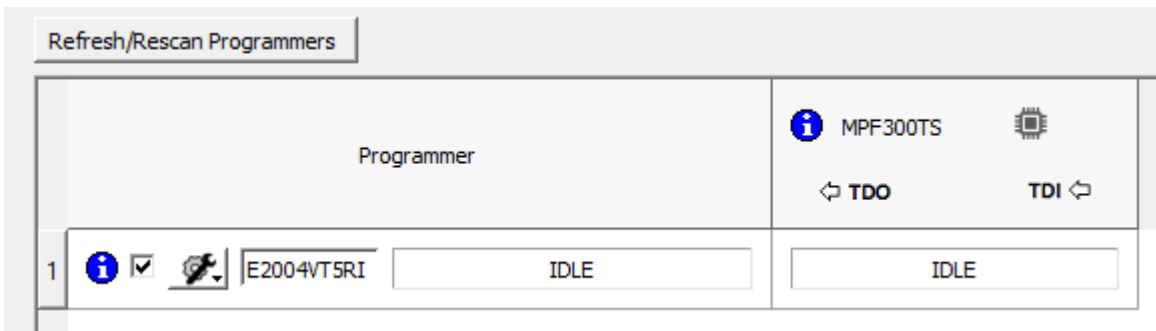
3. Connect the USB cable from J12 on the PolarFire® board to your PC.
4. Connect the HDMI cable from the PolarFire Video Kit (J2) to your external Monitor.
5. Refer to [DG0849](#) and the [Video Kit schematics](#) for jumper settings. We use the default jumper settings shipped with the board.
6. Connect the AC adapter to the board and power it on (SW4).
7. Open FlashPro Express, which you can find in the Start Menu, listed under “Microsemi Libero SoC v2021.1”:



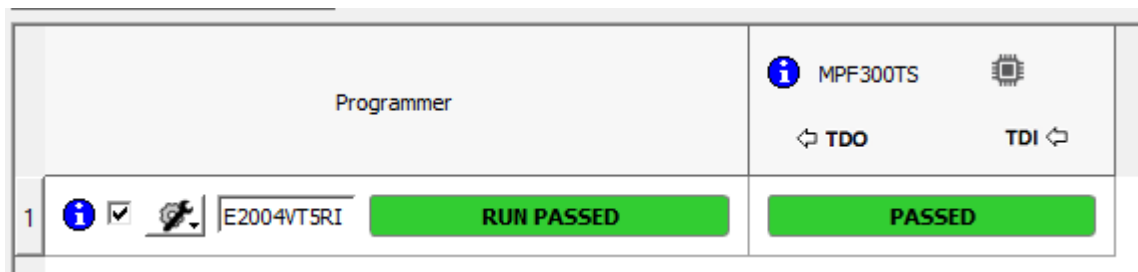
8. Select Project and New Job Project.



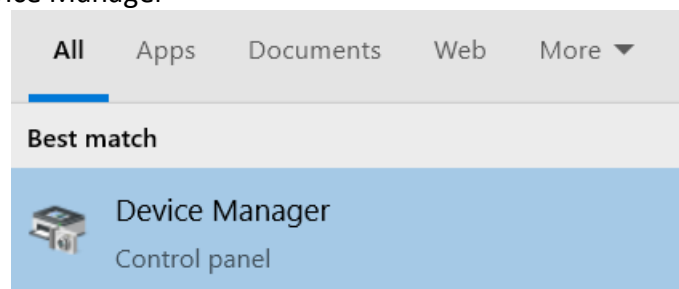
9. Now select the job file “SmartHLS_Training3_job/SmartHLS_Training3.job” in the folder you extracted in step 1.
10. Enter a project location. Click OK.
11. Now the Programmer window will open. If you do not see the Programmer for the MPF300TS PolarFire® FPGA, then click Refresh/Rescan Programmers.



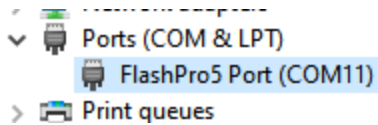
12. Now click the RUN button to program the FPGA.
13. After programming you should see the RUN PASSED. Now close FlashPro Express.



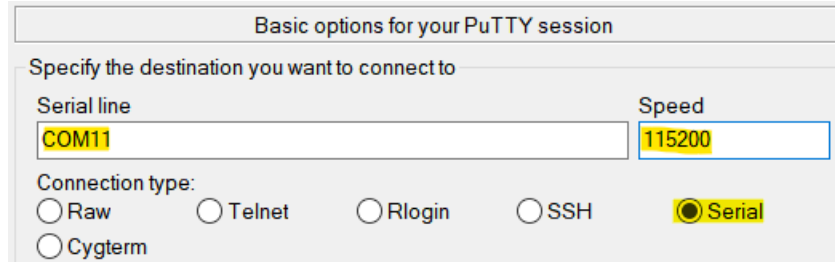
14. Now open the Windows Device Manager menu by opening the Start Menu and then searching for Device Manager



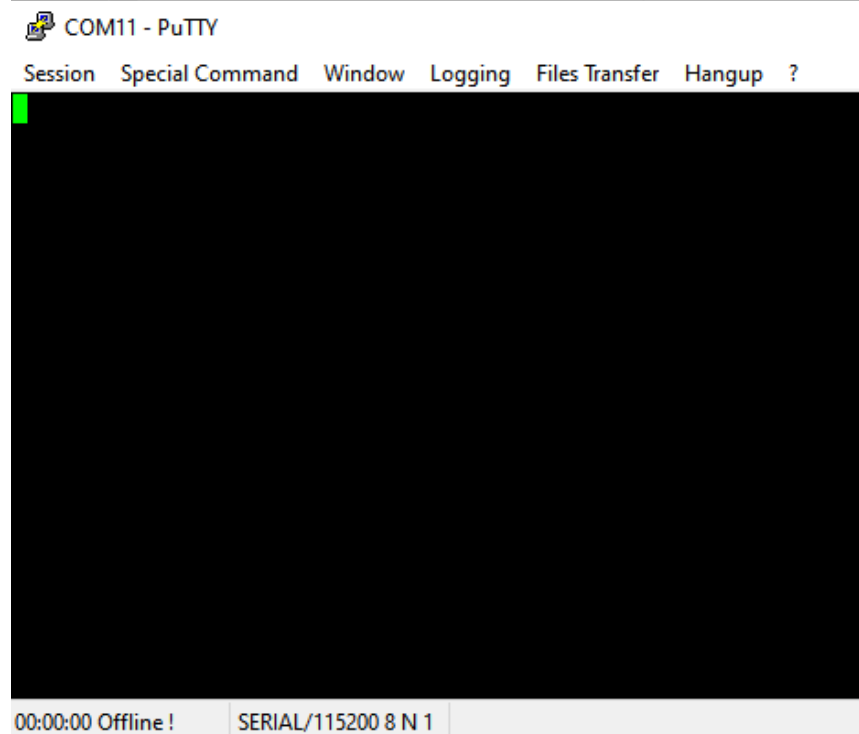
15. In the Device Manager, look for a device type called Ports (COM & LPT) and open it. You should find a COM port corresponding to the USB connection with the PolarFire Video Kit. Note this COM port number. If there is confusion about which COM port it is, unplug then re-plug it to see which one disappears and then appears.



16. Now open PuTTY, select Serial connection, enter a baud rate of 115200, and enter the same COM port you saw in Device Manager. Next click Open.



17. This will open an empty PuTTY console.



18. Now double check the HDMI cable from the PolarFire Video Kit (J2) to your external Monitor as the design will block until there is a screen detected.

19. Power cycle the board. You should see a rotating image on your HDMI output. This video feed is coming from the texture mapper SmartHLS core.



20. You should also see output in PuTTY with some setup messages and a line asking for “mode”. This is the wide multiply operation mode.

```
SmarrHLS Training 3 v1.0 2021-05-20 MCHP  
  
ADV7511 HDMI Transmitter Initialization  
Hotplug (HPD) state check(Blocking)  
HDMI Sink Detected, HPD = High  
ADV7511 Chip Revision (Address 0x00):0x12  
ADV7511 Input VIC #1 (VGA 640x480 4:3) detected  
Enter mode (1 for single, 2 for burst):
```

21. Choose to run in single mode (1). The Mi-V will prompt you to enter in each input to the wide multiply as 32-bit unsigned chunks. Enter the values in hexadecimal. If the Mi-V complains that you are sending an invalid character when pressing enter, make sure that your terminal is sending CR when enter is pressed rather than LF or CR LF.

```

Enter mode (1 for single, 2 for burst): 1
1
Enter A0: 2
2
Enter A1: 0
0
Enter A2: 0
0
Enter A3: 0
0
Enter B0: 2
2
Enter B1: 0
0
Enter C0: 2
2
Enter C1: 0
0
Enter D0: 2
2
Enter D1: 0
0
Enter D2: 0
0
Enter D3: 0
0
Enter E0: 2
2
Enter E1: 0
0
Enter E2: 0
0
Enter E3: 0
0
Sending the inputs (hex):
A: 0 0 0 2
B: 0 2
C: 0 2
D: 0 0 0 2
E: 0 0 0 2
Sending input to SmartHLS accelerator
Starting SmartHLS accelerator
Waiting for SmartHLS accelerator to finish
Reading result from SmartHLS accelerator:
Result (hex): 0 0 0 0 0 2 0 a
Enter mode (1 for single, 2 for burst): █

```

22. Next, choose to run in burst mode (2) The Mi-V will automatically send some testbench values to DDR and signal to the wide multiply core to operate on them. Afterwards the Mi-V will print out the results stored to DDR by the multiply core.

```

Inputs (hex) from DDR for run 2:
A: 0 0 0 f4240
B: 0 54c563
C: 0 dc5640
D: 0 0 0 64
E: 0 0 0 3b9aecb8
Result (hex) from DDR for run 2:
0 0 0 0 0 54db76 d29b88e9 d0eab340
Expected result (hex) for run 2:
0 0 0 0 0 54db76 d29b88e9 d0eab340

Inputs (hex) from DDR for run 3:
A: 0 0 2 4cb016ea
B: 2 4cb016ea
C: 61234567 80000000
D: 0 0 2 4cb016ea
E: 71234567 89abcdef ffffffff fffffffe
Result (hex) from DDR for run 3:
82509c5 dd467a31 5835a072 2fecbafe db24408e aff1f66d 35c559d 4cb016ea
Expected result (hex) for run 3:
82509c5 dd467a31 5835a072 2fecbafe db24408e aff1f66d 35c559d 4cb016ea

Inputs (hex) from DDR for run 4:
A: 0 0 2 4cb016ea
B: 2 4cb016ea
C: 71234567 80000000
D: 0 0 2 4cb016ea
E: 71234567 89abcdef ffffffff fffffffe
Result (hex) from DDR for run 4:
b0c7aa5 4087e648 192a179a afecbafe cdffd7e1 e4bcf7db a35c559d 4cb016ea
Expected result (hex) for run 4:
b0c7aa5 4087e648 192a179a afecbafe cdffd7e1 e4bcf7db a35c559d 4cb016ea

Enter mode (1 for single, 2 for burst): █

```

11 Reference A: Wide Multiply Implementation

In this section we will go through the reference SmartHLS implementation of the wide multiply block. The reference implementation achieves the performance requirement of 156.25 MHz and is pipelined with initiation interval equal to 1 and a latency of 43. It uses the same number of MACCs as the RTL implementation but uses more 4LUTs, DFFs, and uSRAMs. Part of this extra usage is due to the longer pipeline SmartHLS generated compared to the RTL design.



To see this implementation, open the *wide_mult* project in SmartHLS and open the *wide_mult.cpp* source file.

```
▼ wide_mult
  > Includes
  > wide_mult.cpp
  config.tcl
  custom_config.tcl
  makefile
  WideMultTopPf_Tb.sv
```

At the top of the file, we have included the *ap_fixpt* and *ap_int* SmartHLS libraries. These are the SmartHLS types that we can use to easily implement wide data types.

```
#include <hls/ap_fixpt.hpp>
#include <hls/ap_int.hpp>
...
```

On line 15, we define the *wide_mult* top-level function, which will implement our wide multiply block. From this function, we take the arguments A-E as arguments and return the result as a return value. The types of the inputs and output corresponds to the types described in Section 7.1. Note, the *ap_fixpt* type takes the width as the first template argument and the number of **integer** bits as the second argument.

```
ap_fixpt<256, 192> wide_mult(ap_fixpt<128, 64> A, ap_uint<64> B, ap_int<64> C,
                             ap_fixpt<128, 64> D, ap_fixpt<128, 64> E) {
    #pragma HLS function top
```

Inside the function body, on line 18, we use the function pipeline pragma to tell SmartHLS to generate a pipeline for the function body.

```
#pragma HLS function pipeline
```

On line 21 and 22, we declare the intermediate types used for the calculations. Since overflow bits must be handled, we declare the width of the intermediate types to be the sum of the widths of the two input types for multiplies and the same size as the wider input for additions. For the multiply between E and C (128x64) the result becomes 192-bits wide with 128 integer bits (64 fractional bits). For the multiply between the o intermediate result and C (192x64), the result becomes 256-bits wide with 192 integer bits (64 fractional bits).

```
ap_fixpt<192, 128> m, n, o;
```

```
ap_fixpt<256, 192> p, q;
```

Starting on line 24, we have the implementation of the wide multiply operations. This performs the operations described in Section 7.1 while making sure to assign each result to the appropriately sized intermediate result to handle overflow. We return the final addition value as the result of the operation.

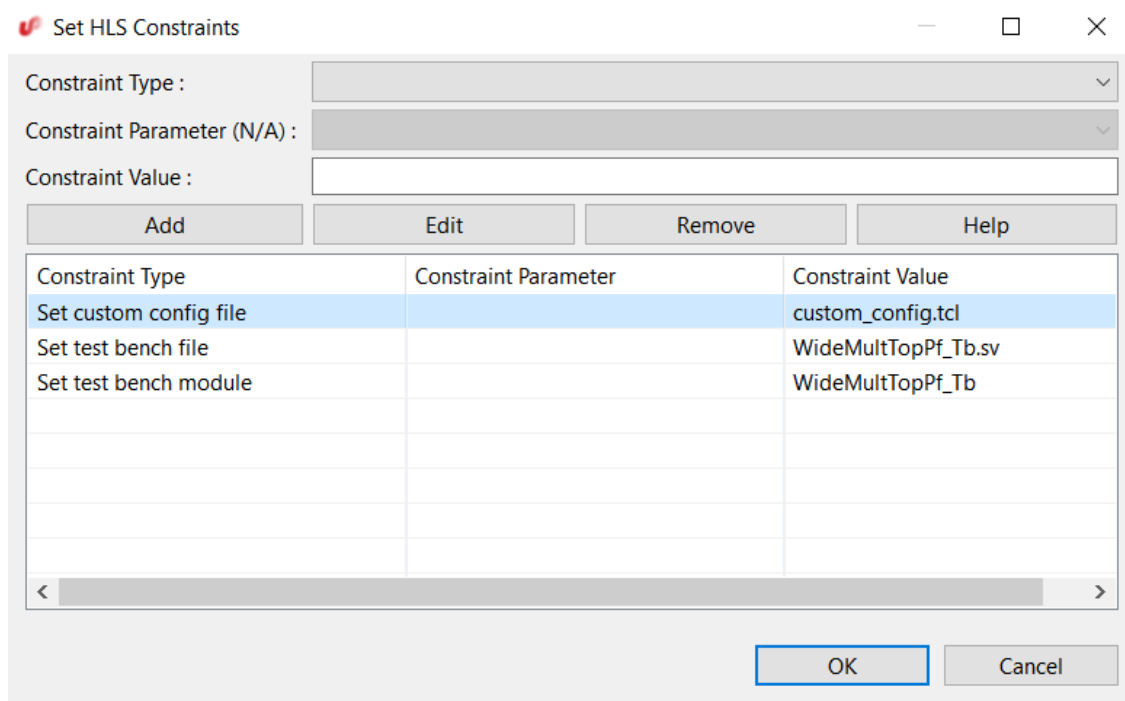
```
m = C * E;
n = m.ashr(1);
o = D + n;
p = C * o;
q = A + p;
return B + q;
```

As mentioned in Section 7.2.2, SmartHLS automatically handles sign extension, padding, and shifting when operating on arbitrary precision fixed-point and arbitrary precision integers of different sizes. SmartHLS also automatically removes the padding when generating the hardware operations to prevent generating operations that are unnecessarily wide.

Also mentioned in Section 7.2.2, the pipeline registers used to store inputs until they are needed in the original RTL design were implemented with uSRAM FIFOs. SmartHLS does not do this by default. We can perform a similar optimization in SmartHLS by setting the `USE_FIFO_FOR_PIPELINE_REG` parameter to 1 in a custom configuration file in. This will enable using uSRAMs for pipeline register generation.



Open the constraints menu (⚙️) and check that a custom configuration file was added as a constraint.





Open the *custom_config.tcl* configuration file. You will find the command to enable the setting:

```
set_parameter USE_FIFO_FOR_PIPELINE_REG 1
```

11.1 Simulation and Generated Hardware



Go back to *wide_mult.cpp*. On line 32, we have a software testbench that checks that the result from the wide multiply works for the cases in the SystemVerilog testbench. This can be used to verify the design in both software and co-simulation.

```
int main() {

    // input test vectors
    const int NUM_INPUTS = 47;

    ...
```

On line 188, the main testbench loop runs each wide multiply operation and checks to see if the results match results generated by the SystemVerilog testbench.

```
for (int i = 0; i < NUM_INPUTS; i++) {
    ap_fixpt<256, 192> result;
    result = wide_mult(A[i], B[i], C[i], D[i], E[i]);

    std::cout << "A: " << A[i].raw_bits() << " B: " << B[i]
                << " C: " << C[i] << " D: " << D[i].raw_bits()
                << " E: " << E[i].raw_bits() << "\n";
    std::cout << "Expected: " << expected_results[i].raw_bits()
                << ", Actual: " << result.raw_bits() << "\n";
    if (result != expected_results[i]) {
        mismatch_cnt++;
        std::cout << "ERROR: mismatch\n";
    } else {
        std::cout << "Matched!\n";
    }
}
```

We report the final results and return any errors found in the testbench for co-simulation to check.

```
std::cout << "Mismatches: " << mismatch_cnt << "\n";
if (mismatch_cnt)
    std::cout << "FAIL\n";
else
    std::cout << "PASS\n";
return mismatch_cnt;
```



Run software simulation by clicking the Run Software button (🏃). SmartHLS will automatically detect saved dependency changes and compile them before running the software. You should see that the software simulation passed:

```
...
Mismatches: 0
PASS
```



Now generate the hardware (🔌). In the *summary.hls.rpt* file, scroll down to Section 1: RTL Interface. Verify that the arguments A-E have been generated as scalar interfaces with the correct widths, and that the `return_val` port has been generated as a 256-bit output.

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

RTL Interface Generated by SmartHLS				
C++ Name	Interface Type	Signal Name	Signal Bit-width	Signal Direction
	Control	clk	1	input
		finish	1	output
		ready	1	output
		reset	1	input
		return_val	256	output
		start	1	input
A	Scalar Argument	A	128	input
B	Scalar Argument	B	64	input
C	Scalar Argument	C	64	input
D	Scalar Argument	D	128	input
E	Scalar Argument	E	128	input



Next, scroll down to Section 3: Pipeline Result then scroll to the right to check the initiation interval and pipeline latency of the design. Verify that the initiation interval for the circuit is 1 and that the pipeline latency is 43 cycles.

===== 3. Pipeline Result =====

Location in Source Code	Initiation Interval	Pipeline Length
line 15 of wide_mult.cpp	1	43

+-----+-----+-----+-----+

Since we have a custom testbench with the same test cases as the software testbench, we will skip running co-simulation.



Now open the *WideMultTopPf_Tb.sv* file in SmartHLS by right clicking and selecting Open With->Text Editor. This is the same testbench as the one present in the *wide_mult_template* project. This testbench can also be used to verify the correctness of the hardware generated by SmartHLS. Run the custom testbench (▶). You should see the following output:

```
...
# This testbench expects a pipelined wide multiply with initiation interval of
# 1 and pipeline latency of 43.
# If there are mismatches in this testbench but not in co-simulation, check
# your pipeline results.
# Simulation finished!           0 mismatches.
# PASS
...
```

11.2 Quality of Results



Now run synthesis (🔧). This should take 10-15 minutes. In the *summary.results.rpt* file, you should find something similar to the following:

===== 2. Timing Result =====

Clock Domain	Target Period	Target Fmax	Worst Slack	Period	Fmax
clk	10.000 ns	100.000 MHz	5.257 ns	4.743 ns	210.837 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 =====

Resource Type	Used	Total	Percentage
Fabric + Interface 4LUT*	1494 + 3216 = 4710	299544	1.57
Fabric + Interface DFF*	11135 + 3216 = 14351	299544	4.79
I/O Register	0	1536	0.00
User I/O	0	512	0.00
uSRAM	28	2772	1.01
LSRAM	0	952	0.00
Math	80	924	8.66


```
+-----+-----+-----+-----+
```

The results for this design might vary quite a bit. It is normal to see somewhat higher Fmax and somewhat higher DFF usage in your synthesis run.

The Fmax for this synthesis run was 210.837 MHz, which is above the required 156.25 MHz. The 4LUT and DFF usage are very similar to the original RTL implementation. We are also using 28 uSRAM blocks due to the use of the USE_FIFO_FOR_PIPELINE_REG parameter included in the *custom_config.tcl* file. The design uses the same number of MACCs as the RTL implementation thanks to SmartHLS generating the multiplies as 128x64 and 192x64.

Without making the uSRAM optimization as described in Section 11, we would be able to get a higher Fmax but significantly more DFF usage. The timing result and resource usage would look something like the following:

===== 2. Timing Result =====

Clock Domain	Target Period	Target Fmax	Worst Slack	Period	Fmax
clk	10.000 ns	100.000 MHz	6.091 ns	3.909 ns	255.820 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 =====

Resource Type	Used	Total	Percentage
Fabric + Interface 4LUT*	1412 + 2880 = 4292	299544	1.43
Fabric + Interface DFF*	20119 + 2880 = 22999	299544	7.68
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	80	924	8.66

The Fmax is significantly higher but the DFF usage is also significantly higher. This is because no uSRAMs are used for pipeline stages. Since the version using uSRAMs for pipelining meets the timing requirements and matches the resource usage of the RTL implementation better, this is the preferable solution for our case.

The remaining extra 4LUT and DFF usage might be due to the deeper pipeline generated by

SmartHLS to maximize Fmax. SmartHLS currently does not have a feature to adjust the latency of our split multiply blocks to tune resource usage vs performance so there is nothing we can do to fix this for now. Split multiply correctly uses MACC registers to pipeline the multiplies.

Table 2: Resource comparison between RTL and SmartHLS

Implementation	Fabric + Interface 4LUT	Fabric + Interface DFF	uSRAM	MACC
RTL	1958 + 3288 (5246)	10839 + 3288 (14127)	34	80
SmartHLS	1412 + 2880 (4292)	20119 + 2880 (22999)	0	80
SmartHLS with uSRAM	1494 + 3216 (4710)	11135 + 3216 (14351)	28	80

Table 2 shows the resource usage comparison between the RTL implementation and SmartHLS. SmartHLS with uSRAM is the results for the design found in the wide_mult project while SmartHLS is the version without the USE_FIFO_FOR_PIPELINE_REG parameter.

Table 3 shows the number of lines of code required to implement this design in both RTL and in C++ using SmartHLS. As you can see the SmartHLS implementation offers much fewer lines of code allowing for much better productivity.

Table 3: Productivity comparison between RTL and SmartHLS

Implementation	Lines of Code for Core	Lines of Code for Testbench without Test Cases
RTL	290	198
SmartHLS	16	18

Overall, the SmartHLS implementation has comparable resource usage to the original RTL implementation, while allowing the user to design at a higher level of abstraction.

12 Reference B: AXI4 Target Implementation

In this section we will go over the reference implementation of the AXI target interface. The Mi-V provides the A-E inputs used by the wide multiplier and sends control signals to run the accelerator. The accelerator takes the A-E inputs and writes the result back to the Mi-V through the read channels when requested by the Mi-V. This can easily be implemented in SmartHLS as an AXI target struct with the AXI target interface pragma.



Open the *wide_mult_axi_target* project and open the *axi4_target.h* source file.

```
▼ wide_mult_axi_target
  > Includes
  > axi4_target.h
  > wide_mult_axi.cpp
    config.tcl
    initiator_layout.txt
```

The TargetLayout struct defines the data that will be sent from and to the Mi-V. You will find that the structure of the target struct is the same as the struct found in *initiator_layout.txt*. All elements are 32-bit unsigned integer words to match the data types used in the Mi-V SoftConsole code. In the struct, we have the run control variable used by the Mi-V to start the accelerator. We have eight words to represent the 256-bit result from the wide multiply block. We also have four words to represent each of the 128-bit A, D, and E arguments, and two words to represent the 64-bit B, and C arguments. There is an extra “padding” word at the end of the struct as SmartHLS only supports AXI target structs with a size that is a multiple of 64-bits. This is due to the generated AXI data bus being fixed to 64-bits.

```
struct TargetLayout {
    uint32_t run;
    uint32_t result0;
    ...
    uint32_t result7;
    uint32_t A0;
    ...
    uint32_t E3;
    uint32_t padding;
};
```



Open *wide_mult_axi.cpp* and go to line 17, we declare a global variable of the above type that will serve as our AXI target channel. We use the interface pragma with the *axi_slave* type to tell SmartHLS to generate this global variable as an AXI4 target (slave) interface. The concurrent access flag specifies that the AXI initiator can access the target values while the wide multiply core is running. The struct does not need to be declared volatile as SmartHLS will handle this struct differently compared to a normal struct variable due to the pragma.

```
// global variable for AXI4-Lite target
```

```
#pragma HLS interface variable(data) type(axi_slave) concurrent_access(true)
TargetLayout data;
```

SmartHLS will implement this interface by using the address provided in the write and read address channels to write and read to internal RAMs representing the elements of the struct.

In the `wide_mult` function body, starting on line 30, we check if the run control signal has been set by the Mi-V and then read the inputs and assign them to the corresponding arbitrary precision variable slice. We perform the same calculation as before, then write each slice of the results to the AXI target interface struct. We then set the run control signal to 0 to signal to the Mi-V that the calculation has finished. The Mi-V will poll run until it becomes 0 and then send a read request to read the computation results from the SmartHLS target.

```
if (data.run) {

    A(31, 0) = data.A0;
    ...
    E(127, 96) = data.E3;

    m = C * E;
    n = m.ashr(1);
    o = D + n;
    p = C * o;
    q = A + p;
    r = B + q;

    data.result0 = r(31, 0);
    ...
    data.result7 = r(255, 224);

    data.run = 0;
}
```

12.1 Simulation and Generated Hardware



Go to `wide_mult.cpp` line 70. we have a software testbench that checks that the result from the wide multiply works for the cases in the SystemVerilog testbench. This can be used to verify the design in both software and co-simulation.

```
int main() {

    // input test vectors
    const int NUM_INPUTS = 47;

    ...
}
```

On line 225, the main testbench loop runs each wide multiply operation and checks to see if the results match results generated by the SystemVerilog testbench. This time the testbench writes the input data to and reads the result from the AXI4 TargetLayout struct.

```
unsigned mismatch_cnt = 0;
```

```

for (int i = 0; i < NUM_INPUTS; i++) {
    data.A0 = A[i](31, 0);
    ...
    data.E3 = E[i](127, 96);
    data.run = 1;

    wide_mult();

    ap_fixpt<256, 192> result;
    result(31, 0) = data.result0;
    ...
    result(255, 224) = data.result7;

    std::cout << "A: " << A[i].raw_bits() << " B: " << B[i]
                << " C: " << C[i] << " D: " << D[i].raw_bits()
                << " E: " << E[i].raw_bits() << "\n";
    std::cout << "Expected: " << expected_results[i].raw_bits()
                << ", Actual: " << result.raw_bits() << "\n";
    if (result != expected_results[i]) {
        mismatch_cnt++;
        std::cout << "ERROR: mismatch\n";
    } else {
        std::cout << "Matched!\n";
    }
}

```

We report the final results and return any errors found in the testbench for co-simulation to check.

```

std::cout << "Mismatches: " << mismatch_cnt << "\n";
if (mismatch_cnt)
    std::cout << "FAIL\n";
else
    std::cout << "PASS\n";
return mismatch_cnt;

```



Next, run the co-simulation (▶). You should see the following in the output:

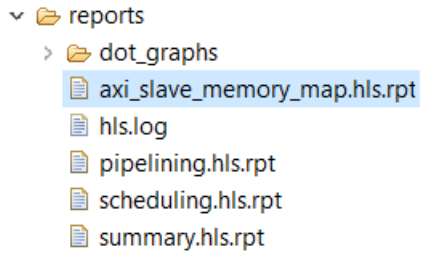
```

Mismatches: 0
PASS
...
Number of calls:      47
Cycle latency:       2,773
SW/HW co-simulation: PASS

```



Compile to hardware (⚙️). There is a new report that was generated that we have not seen before. This *axi_slave_memory_map.hls.rpt* file holds the memory mapped addresses for the generated AXI target struct. These addresses can be used by the Mi-V initiator to communicate with the generated AXI target.



From the Mi-V, we need to correctly setup the correct base address in the Mi-V IP interface configuration and any AXI Interconnect IPs used to connect the Mi-V to the SmartHLS target. We can write and read to those memory mapped addresses to communicate with the target. More information on the processor code and AXI Interconnect can be found in Appendix A.



Now open the *summary.hls.rpt* file and scroll down to Section 1: RTL Interface. Check that an interface of type AXI Slave (target) has been generated. This is the AXI4 target interface generated for our data global variable.

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

RTL Interface Generated by SmartHLS				
C++ Name	Interface Type	Signal Name	Signal Bit-width	Signal Direction
	Control	clk	1	input
		finish	1	output
		ready	1	output
		reset	1	input
		start	1	input
data	AXI Slave	axi_s_ar_addr	32	input
		axi_s_ar_len	8	input
		axi_s_ar_ready	1	output
		axi_s_ar_valid	1	input
		axi_s_aw_addr	32	input
		axi_s_aw_len	8	input
		axi_s_aw_ready	1	output
		axi_s_aw_valid	1	input
		axi_s_b_resp	8	output
		axi_s_b_resp_ready	1	input
		axi_s_b_resp_valid	1	output
		axi_s_r_data	64	output
		axi_s_r_last	8	output
		axi_s_r_ready	1	input
		axi_s_r_resp	8	output
		axi_s_r_valid	1	output
		axi_s_w_data	64	input
		axi_s_w_last	8	input
		axi_s_w_ready	1	output
		axi_s_w_strb	8	input
		axi_s_w_valid	1	input

13 Appendix A: Integrating with SmartDesign and SoftConsole

13.1 Replacing Existing Component in SmartDesign

Figure 9, shows the SmartDesign IP component that contains the SmartHLS-generated wide multiply block.

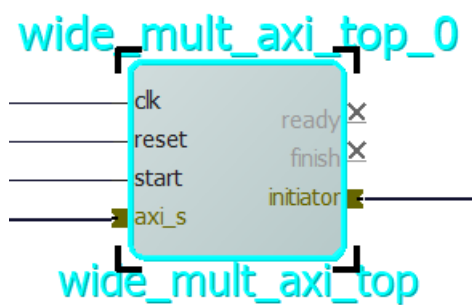


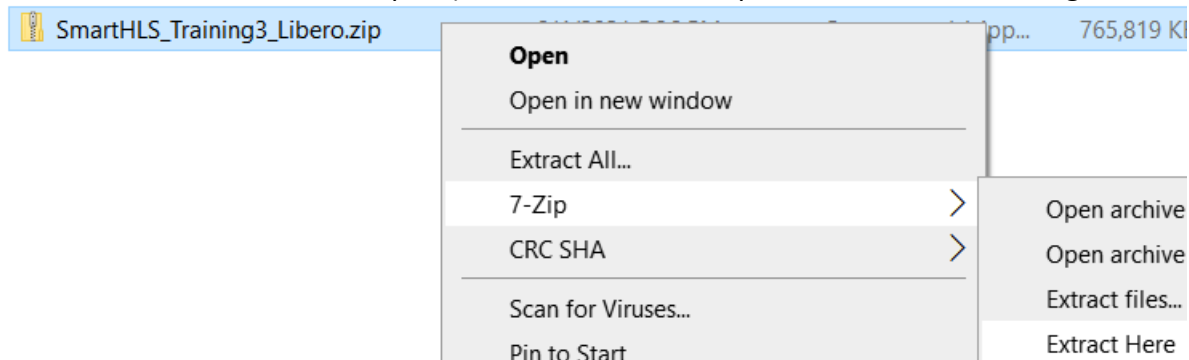
Figure 9: Block Diagram of the SmartHLS-generated wide multiply block



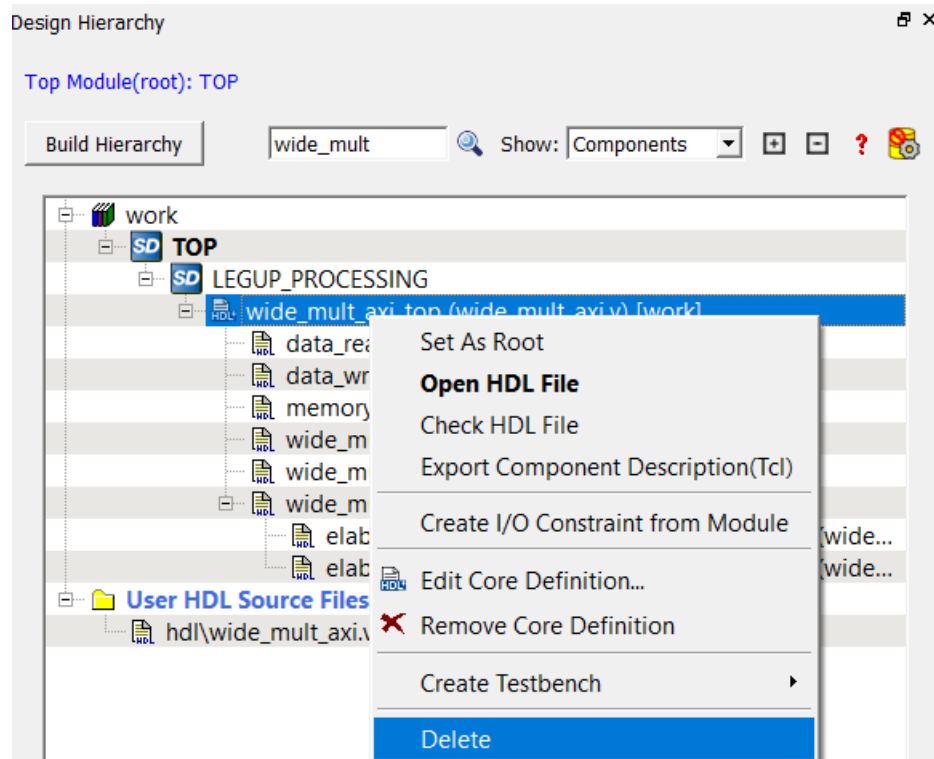
To integrate the PolarFire® Video Kit design in Libero SoC 2021.2:

1. Download the SmartHLS_Training3_Libero.zip file if you have not already.
2. Extract the SmartHLS_Training3_Libero.zip file contents containing the project and job file.

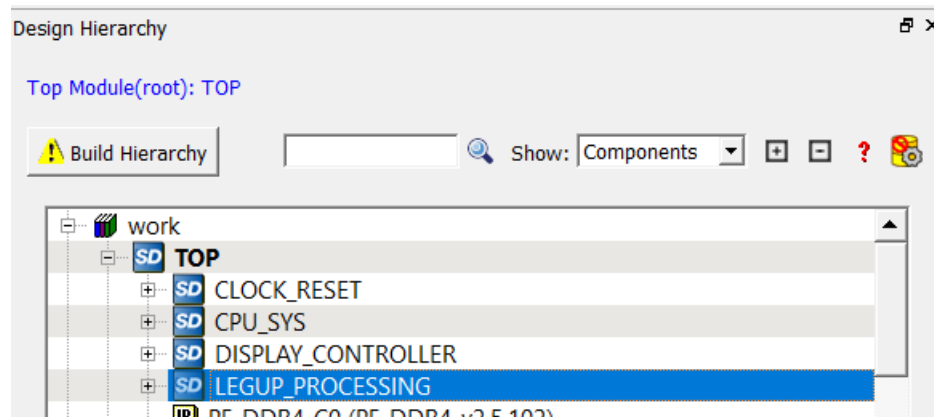
On Windows you need to extract the project to a directory with a short name (such as C:\Downloads or C:\Workspace) and extract with 7-Zip to avoid issues with long names:



3. Launch Libero SoC 2021.2 and open the SmartHLS_Training3 project by navigating to and clicking SmartHLS_Training3.prjx.
4. Navigate to the Design Hierarchy and search for “wide_mult”. Right click the wide_mult_axi_top design component and select Delete. We want to avoid any duplicate blocks when importing the new wide multiply design from SmartHLS.

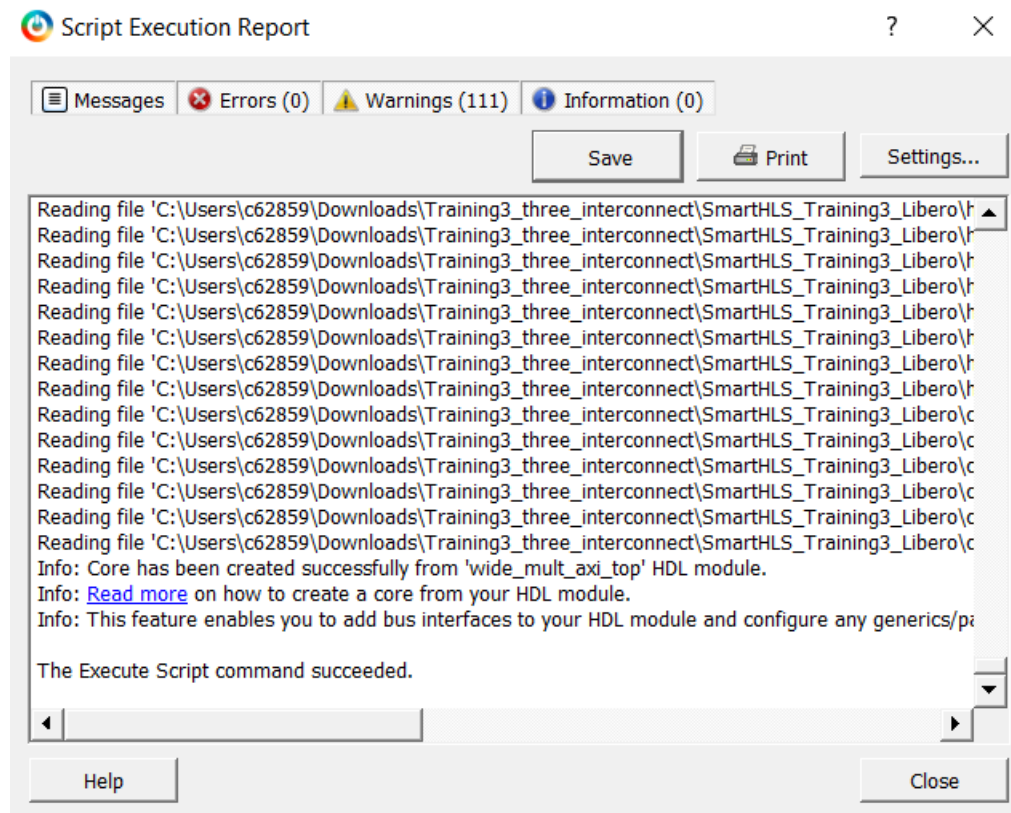


5. Now clear the search, then click the plus next to the TOP SmartDesign file and then double click LEGUP_PROCESSING to open it in the SmartDesign Canvas.

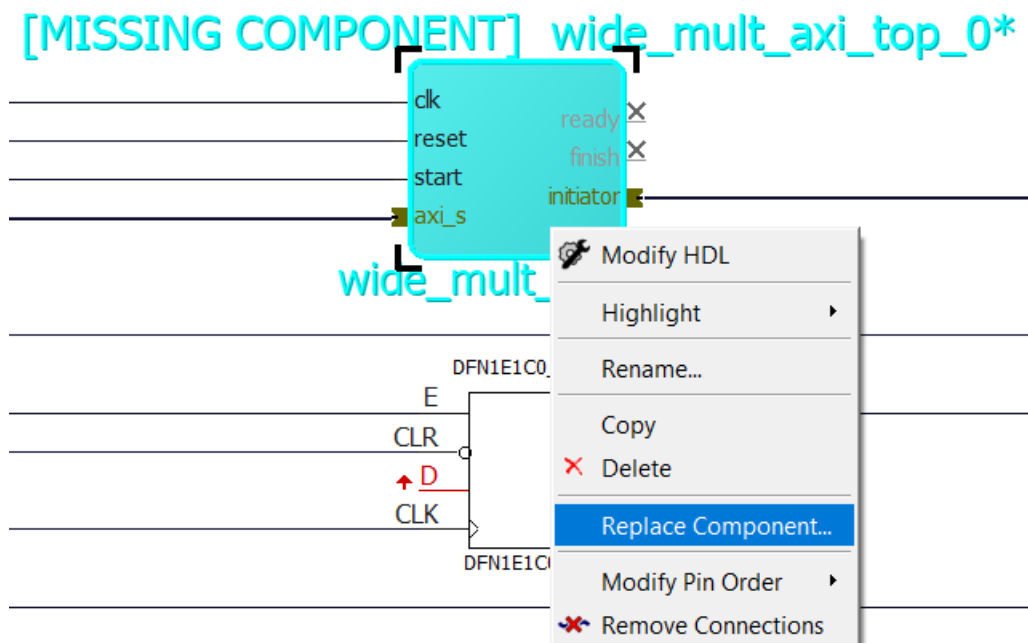


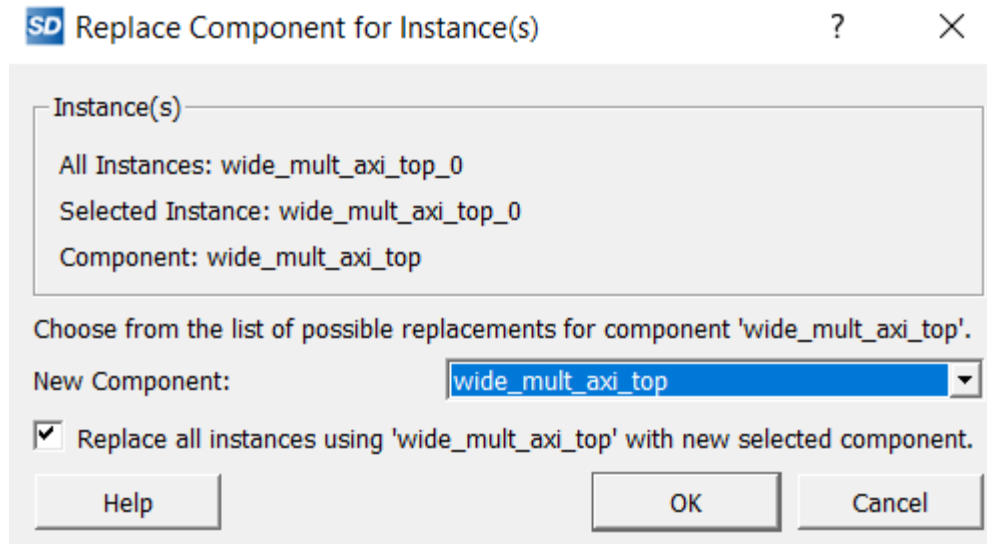
6. On the top toolbar, click Project->Execute Script... and run the create_hdl_plus.tcl file from the wide_mult_axi SmartHLS project directory. SmartDesign will open a report window when it finishes. Make sure the script executed successfully and then close the


report window.



- Next find the wide_mult_axi_top block in LEGUP_PROCESSING and right click and select Replace Component. Select the newly imported wide_mult_axi_top to replace the SmartDesign block without having to reconnect any wires.





8. Click the “Generate Component” () button in the SmartDesign toolbar for LEGUP_PROCESSING.
9. The wide_mult_axi_top SmartHLS block has now been integrated and the project is ready for synthesis, place, and route.

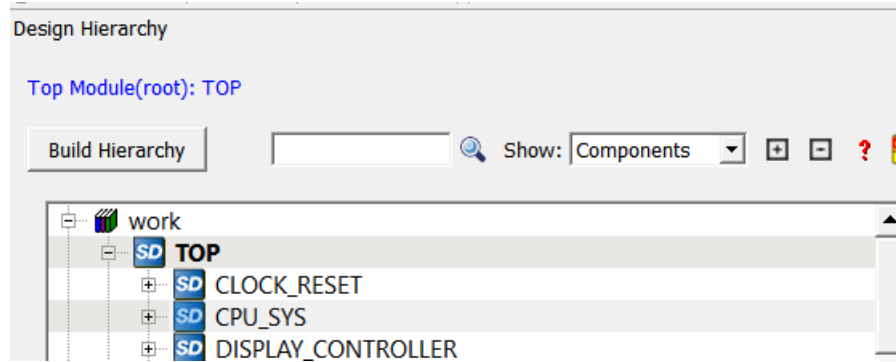
13.2 SmartDesign AXI4 Connection

In order to communicate with the SmartHLS module through AXI4, the Mi-V must be configured properly to have an AXI4 initiator interface generated for a certain address range. The Mi-V can then write to this address range in order to interact with the module through the AXI4 initiator interface.

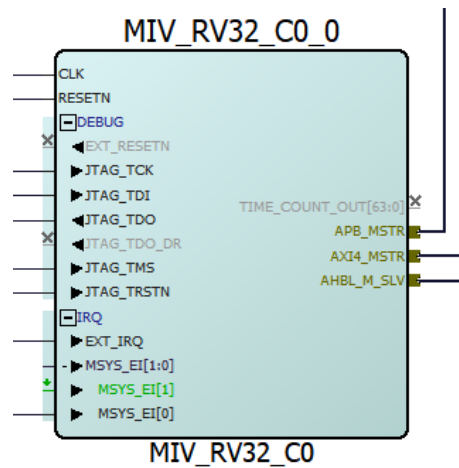
Note again, the AXI4 protocols define a master and a slave. In this training, will refer to them as the AXI4 initiator and the AXI4 target. However, the diagrams, tools, and online references used for this training will still refer to them as AXI4 master and AXI4 slave. Whenever you see these terms in SmartDesign or configurators, know that we will refer to master as initiator and slave as target.



To see how to configure the Mi-V to have an AXI4 initiator interface, open the Libero project. Open the Design Hierarchy tab and expand the TOP SmartDesign canvas, then double click the CPU_SYS canvas to open it. This canvas has the Mi-V core.



In the opened SmartDesign canvas, double click on the MIV_R32_CO_0 core to bring up the configuration menu.



In the Memory Map tab of the configurator, scroll down to the third entry where the AXI Master Address configuration is. This range defines the range of addresses in the Mi-V to map the AXI initiator interface to. Leave the address range as default.

Mi-V RV32 Configurator

Microsemi:MiV:MIV_RV32:3.0.100

Configuration
Memory Map

AHB Master Address

Start Address: Upper 16bits (Hex): <input type="text" value="0x8000"/>	Lower 16bits (Hex): <input type="text" value="0x0"/>
End Address: Upper 16bits (Hex): <input type="text" value="0x8fff"/>	Lower 16bits (Hex): <input type="text" value="0xffff"/>

APB Master Address

Start Address: Upper 16bits (Hex): <input type="text" value="0x7000"/>	Lower 16bits (Hex): <input type="text" value="0x0"/>
End Address: Upper 16bits (Hex): <input type="text" value="0x7fff"/>	Lower 16bits (Hex): <input type="text" value="0xffff"/>

AXI Master Address

Start Address: Upper 16bits (Hex): <input type="text" value="0x6000"/>	Lower 16bits (Hex): <input type="text" value="0x0"/>
End Address: Upper 16bits (Hex): <input type="text" value="0x6fff"/>	Lower 16bits (Hex): <input type="text" value="0xffff"/>



Now close the configurator.



If you right click on the AXI4_MSTR bus and select Show/Hide BIF Pins, you will find that the read and write data bus has a width of 32-bits. If you remember from before, the AXI4 target interface generated by SmartHLS has a data bus width of 64-bits. In order to connect the Mi-V initiator with the SmartHLS target interface, we need to bridge these different data bus widths.

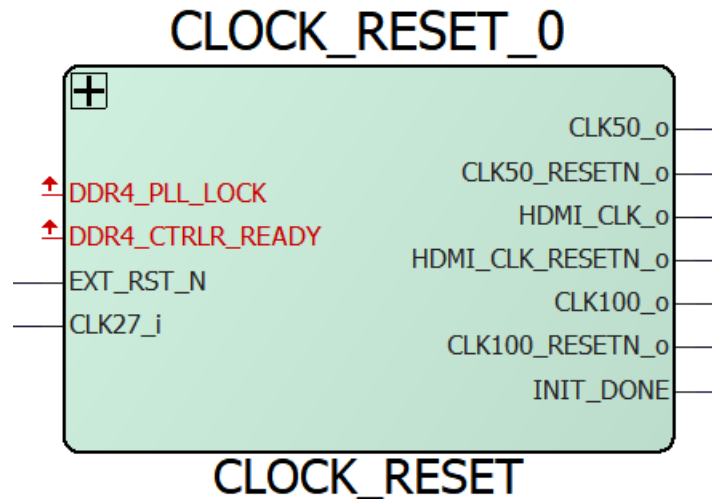
Pins to Expose - 'AXI4_MSTR' on 'MIV_RV32_C0_0'

<input type="checkbox"/>		AXI_MSTR_AWREADY
<input checked="" type="checkbox"/>		AXI_MSTR_WDATA[31:0]
<input checked="" type="checkbox"/>		AXI_MSTR_WSTRB[3:0]
<input checked="" type="checkbox"/>		AXI_MSTR_WLAST
<input checked="" type="checkbox"/>		AXI_MSTR_WVALID
<input type="checkbox"/>		AXI_MSTR_WREADY
<input type="checkbox"/>		AXI_MSTR_BID
<input type="checkbox"/>		AXI_MSTR_BRESP[1:0]
<input type="checkbox"/>		AXI_MSTR_BVALID
<input checked="" type="checkbox"/>		AXI_MSTR_BREADY

Help
OK
Cancel



Now close the CPU_SYS canvas and open the TOP canvas. Find the CLOCK_RESET_0 block.



If you follow the CLK outputs, you will find that the Mi-V runs on the 50MHz clock (CLK50_o) while the SmartHLS generated accelerators run on the 100MHz clock (CLK100_o). In order for the Mi-V to communicate with the SmartHLS module through AXI4, these two clock domains must also be bridged.

The last problem we need to consider when connecting the Mi-V initiator is how many targets it will need to control. In this design, the Mi-V initiator has three AXI4 targets, the texture mapper, the wide multiply, and the DDR controller. The same AXI4 initiator interface cannot be used to control all three targets at the same time. We need a way to differentiate which target we want to send commands to in any given write and read request.

Luckily, these three problems can easily be solved by using an AXI Interconnect. This core allows multiple initiators to map to multiple targets, with different address ranges mapping to different targets. The AXI Interconnect can also handle different clocks for each initiator and target connected to it.



Close the TOP canvas and open the LEGUP_PROCESSING canvas and find the COREAXI4INTERCONNECT_C0_0 block. Double click it to open the configurator.



In the Configuration tab of the configurator, you will be able to find settings to control the

number of initiators and targets connected to this AXI Interconnect. You can also modify some other settings such as address width, but for this design those do not need to be modified from default. You can see that this AXI Interconnect is configured to connect one initiator to three targets.

CoreAXI4Interconnect Configurator

Microsemi:DirectCore:COREAXI4INTERCONNECT:2.8.103

❗ Configuration
Master Configuration
Slave Configuration
Crossbar Configuration

☐ **Bus Configuration**
❗

Number of Masters:

Number of Slaves:

ID Width:

❗ Address Width:

User Width:

In the Master Configuration tab, we can choose the protocol type and data bus width of the initiator interface. The Mi-V initiator interface is a AXI4 interface with data width 32. You may also notice that the M0 Clock Domain Crossing checkbox is checked. This will generate a clock input that will be used to provide the clock used by the initiator. The clock used by the targets is provided through the main clock input on the core.

❗ Configuration
Master Configuration
Slave Configuration
Crossbar

☐ **Master0 Configuration**

M0 Type:

M0 Data Width:

M0 DWC Data FIFO Depth:

M0 Register Slice: ☒

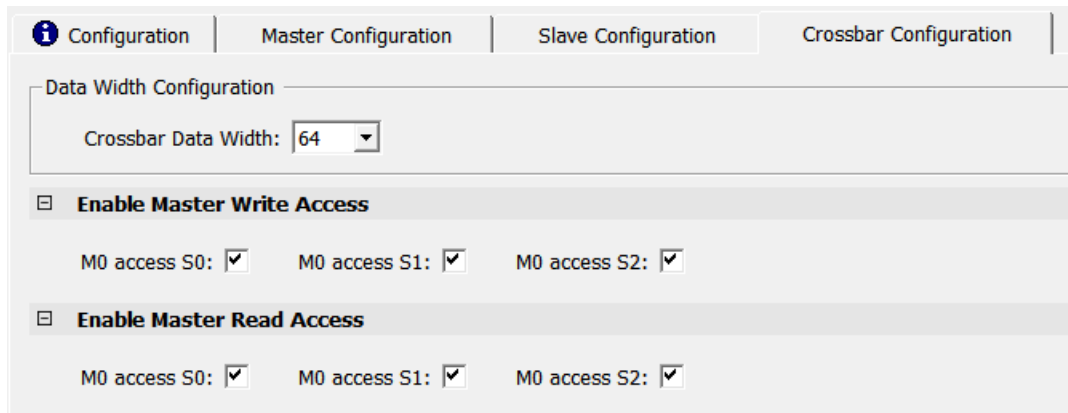
M0 Clock Domain Crossing: ☒

M0 Read Interleaving: ☐

In the Slave Configuration tab, we have three targets configured. Note, the first target interface is connected to the texture mapper SmartHLS core. The target type generated is AXI4-Lite to match the AXI4-Lite with burst enabled interface generated by SmartHLS. When connected to an AXI4-Lite interface, the burst option will not be used. Optionally, the interface type may be changed to AXI4 in the configuration to enable burst. The data width is set to 64-bits, which is the generated data width for the SmartHLS target interface. We also assign an address range of 0x60000000 to 0x60ffffff out of the original 0x60000000 to 0x6ffffff address range to map to the texture mapper module. Writing to and reading from this range from the Mi-V will cause the AXI Interconnect to send the requests only to the first target interface. The second target interface is connected to the wide multiply module and is configured the same way as the texture mapper. The wide multiply module is given the address range 0x61000000 to 0x61ffffff. The last target interface connects to a second AXI Interconnect which then connects to DDR and is given the remaining address range.

Configuration	Master Configuration	Slave Configuration	Crossbar Configuration
Slave0 Configuration			
S0 Type:	AXI4-Lite	S0 Data Width:	64
S0 DWC Data FIFO Depth:	16	S0 Register Slice:	<input checked="" type="checkbox"/>
S0 SLAVE Start Address (Upper 32 Bits):	0x0	S0 SLAVE Start Address (Lower 32 Bits):	0x60000000
S0 SLAVE End Address (Upper 32 Bits):	0x0	S0 SLAVE End Address (Lower 32 Bits):	0x60ffffff
S0 Clock Domain Crossing:	<input type="checkbox"/>	S0 Read Interleaving:	<input type="checkbox"/>
Slave1 Configuration			
S1 Type:	AXI4-Lite	S1 Data Width:	64
S1 DWC Data FIFO Depth:	16	S1 Register Slice:	<input checked="" type="checkbox"/>
S1 SLAVE Start Address (Upper 32 Bits):	0x0	S1 SLAVE Start Address (Lower 32 Bits):	0x61000000
S1 SLAVE End Address (Upper 32 Bits):	0x0	S1 SLAVE End Address (Lower 32 Bits):	0x61ffffff
S1 Clock Domain Crossing:	<input type="checkbox"/>	S1 Read Interleaving:	<input type="checkbox"/>
Slave2 Configuration			
S2 Type:	AXI4	S2 Data Width:	64
S2 DWC Data FIFO Depth:	16	S2 Register Slice:	<input checked="" type="checkbox"/>
S2 SLAVE Start Address (Upper 32 Bits):	0x0	S2 SLAVE Start Address (Lower 32 Bits):	0x62000000
S2 SLAVE End Address (Upper 32 Bits):	0x0	S2 SLAVE End Address (Lower 32 Bits):	0x62ffffff
S2 Clock Domain Crossing:	<input type="checkbox"/>	S2 Read Interleaving:	<input type="checkbox"/>

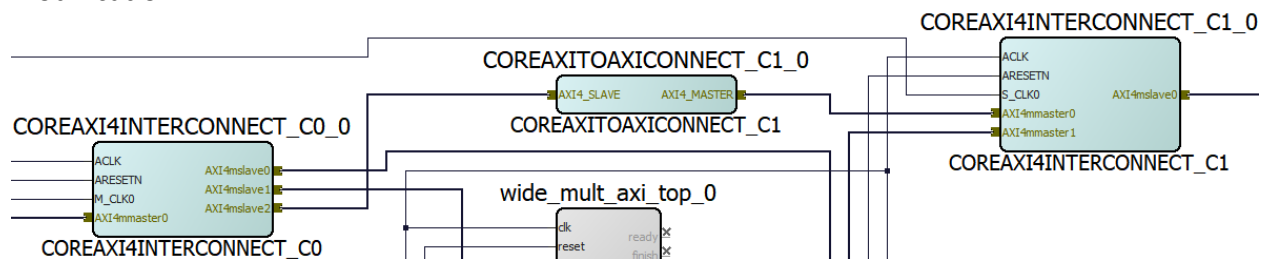
In the Crossbar Configuration, we can optionally change which initiators have read and write permissions to which target. Since we have one initiator that should be able to communicate with all targets, all checkboxes are checked.



The Slave2 output of COREAXI4INTERCONNECT_C0_0 is connected to the first initiator input of a second AXI Interconnect. The wide multiply initiator is connected to the second initiator input. This AXI Interconnect takes two initiator interfaces and arbitrates its access to a single target. Since the DDR is on a different clock domain than the two initiators, while the initiators are on the same clock domain, a single target clock input is used to provide the clock that is used by the target while the clock for the initiators is provided as the main clock input.



To connect an AXI target interface to an AXI initiator interface, as in the case where we want to connect the output from one interconnect to a second interconnect, the AXI4TOAXI4CONNECT core can be used. This converts an AXI target interface to an AXI initiator interface with no modification.



With the help of AXI Interconnects, the AXI initiators were able to be connected to their AXI targets at set address ranges.

13.3 Code Running on the Mi-V Soft Processor

We have talked about designing the interfaces to match with the code running on the Mi-V soft processor when designing the wide multiply AXI-target interface. Now we will see how to write

to and read from the target using the Mi-V processor. The following Mi-V code corresponds to the wide multiply implementation that has both single and burst mode operation.



In the *wide_mult_axi* SmartHLS project, open the *sc_main_c* file. This file is a copied version of the SoftConsole code running on the Mi-V. The name has been changed from *main.c* so that SmartHLS does not pick it up as an HLS source and try to compile it.

```
▼ wide_mult_axi
  > Includes
  > axi4_target.h
  > wide_mult_axi.cpp
  config.tcl
  initiator_layout.txt
  sc_main_c
```

Near the top of the source file, on line 59 and 96, we declare some addresses. The addresses match the base addresses of the ranges we have seen in the Mi-V configurator for the AXI4 initiator and the AXI Interconnect configurators in Section 13.2.

```
// SmartHLS Wide Multiplier AXI-Slave Base Address
#define SHLS_WIDE_MULT_BASE_ADDR      0x61000000UL

...

// DDR AXI-Slave Base Address
#define DDR_BASE_ADDR                 0x62000000UL
```

Starting on line 14 and 62, we have the structs used line up with the address offsets from the generated SmartHLS AXI target interface report. The address above will be cast to pointers to these structs and the struct pointers will be used to write to and read from the addresses. Note, the wide multiply struct matches the struct declared in the SmartHLS core, while the DDR struct matches the memory format expected by the wide multiply core.

```
// SmartHLS Wide Multiplier AXI Target Layout
struct wide_mult {
    ...
};

...

// SmartHLS Wide Multiplier DDR Layout
struct wide_mult_ddr {
    ...
};
```

In the main function on line 535, we run one of two functions, *run_single_multiply* and *run_burst_multiply*, depending on user input.

```
// Run Wide Multiplier loop
while (1) {
    uint32_t mode =
```

```

        get_user_input("Enter mode (1 for single, 2 for burst): ", 41);
    if (mode == 1) {
        run_single_multiply();
    } else {
        run_burst_multiply();
    }
}

```

If the user enters 1, the Mi-V will run the multiply core in single mode using the `run_single_multiply` function found on line 234. This function takes user input as the inputs A-E and sends them to the accelerator. To write to the accelerator, the Mi-V casts the base address we defined into the appropriate struct format and uses this format to write and read from the addresses that line up with addresses in the AXI target interface in the SmartHLS core. For burst mode, the processor will also write to and read from addresses corresponding to the DDR memory. Note, the pointers are cast as volatile to prevent the SoftConsole compiler from optimization away these reads and writes. The Mi-V then asserts the run signal and waits until the accelerator de-asserts it, signally the computing is done. The Mi-V then reads from the memory to issue read requests and get the results from the accelerator.

```

void run_single_multiply() {
    struct wide_mult wm;
    ...

    print("Sending input to SmartHLS accelerator\r\n");
    volatile struct wide_mult *wm_accel_addr =
        (volatile struct wide_mult *) SHLS_WIDE_MULT_BASE_ADDR;
    wm_accel_addr->A0 = wm.A0;
    ...
    wm_accel_addr->E3 = wm.E3;
    print("Starting SmartHLS accelerator\r\n");
    wm_accel_addr->run = 1;

    print("Waiting for SmartHLS accelerator to finish\r\n");
    while (wm_accel_addr->run);

    print("Reading result from SmartHLS accelerator:\r\n");
    wm.result0 = wm_accel_addr->result0;
    ...
    wm.result7 = wm_accel_addr->result7;
    ...
}

```

If the user enters 2, the Mi-V will run the multiply core in burst mode using the `run_burst_multiply` function. This function writes five testbench cases into DDR and allows the accelerator to process all five inputs before reading and checking the results from DDR. Communication with the accelerator works the same way as in single mode but now data is transferred between the accelerator and the Mi-V through the DDR.

```

void run_burst_multiply() {
    // 5 test inputs

```

```

#define NUM_BURST 5
...

print("Writing inputs to DDR\r\n");
volatile struct wide_mult_ddr *ddr_addr =
    (volatile struct wide_mult_ddr *) DDR_BASE_ADDR;

for (uint32_t i = 0; i < NUM_BURST; i++) {
    ddr_addr->A0 = A0[i];
    ...
    ddr_addr->E3 = E3[i];
    ddr_addr->result0 = i;
    ...
    ddr_addr->result7 = i;
    ddr_addr++;
}

```

Then we can start the SmartHLS accelerator by giving it the base address of the DDR to start operating on and writing the number of operations to processes to the run.

```

print("Starting SmartHLS accelerator\r\n");
volatile struct wide_mult *wm_accel_addr =
    (volatile struct wide_mult *) SHLS_WIDE_MULT_BASE_ADDR;
wm_accel_addr->base_addr = DDR_BASE_ADDR;
wm_accel_addr->run = NUM_BURST;

```

Then we poll on the run address until this becomes 0, indicating the SmartHLS accelerator is done.

```

print("Waiting for SmartHLS accelerator to finish\r\n");
while (wm_accel_addr->run);

```

Then we read the SmartHLS accelerator results from DDR memory and check against the expected results.

```

print("Reading results from DDR\r\n");
ddr_addr = (volatile struct wide_mult_ddr *) DDR_BASE_ADDR;
const int BUF_LEN = 200;
char buf[BUF_LEN];

for (uint32_t i = 0; i < NUM_BURST; i++) {
    snprintf(buf, BUF_LEN, "Inputs (hex) from DDR for run %d:\r\n", i);
    "A: %x %x %x %x\r\n",
    "B: %x %x\r\n",
    "C: %x %x\r\n",
    "D: %x %x %x %x\r\n",
    "E: %x %x %x %x\r\n", i,
    ddr_addr->A3, ddr_addr->A2, ddr_addr->A1, ddr_addr->A0,
    ddr_addr->B1, ddr_addr->B0,
    ddr_addr->C1, ddr_addr->C0,
    ddr_addr->D3, ddr_addr->D2, ddr_addr->D1, ddr_addr->D0,
    ddr_addr->E3, ddr_addr->E2, ddr_addr->E1, ddr_addr->E0);
    print(buf);

    snprintf(buf, BUF_LEN, "Result (hex) from DDR for run %d:\r\n", i);
}

```

```

        ...

        snprintf(buf, BUF_LEN, "Expected result (hex) for run %d:\r\n"
        ...

        if (ddr_addr->result0 != result0[i]) {
            snprintf(buf, BUF_LEN, "Mismatched result (hex) for result
%d run %d:\r\n"
                "Expected: %x, Got: %x\r\n", 7, i,
                result7[i], ddr_addr->result7);
            print(buf);
        }
        ...

        print("\r\n");

        ddr_addr++;
    }
}

```

14 Appendix B: Rotozoom and Texture Mapper

As shown in Figure 7, the PolarFire® design also contains another SmartHLS hardware block called the Texture Mapper. The Texture Mapper SmartHLS project is based on a [Rotozoom demo](#) design created by one of our FAEs. Rotozoom stands for rotation and zoom, and the design rotates and zooms in and out on a 512x512 greyscale image and outputs to the monitor over HDMI. After programming the design onto the PolarFire Video kit in Section 10, you will see an output on your monitor like in Figure 10.



Figure 10: Monitor output of Text Mapper SmartHLS hardware block.

The Rotozoom code is based on the algorithm description here:

[https://www.flipcode.com/archives/The Art of Demomaking-Issue 10 Roto-Zooming.shtml](https://www.flipcode.com/archives/The+Art+of+Demomaking-Issue+10+Roto-Zooming.shtml)

The Mi-V soft processor performs rotation and scaling calculations and passes coordinate and delta values to the Texture Mapper SmartHLS core as shown in Figure 11.

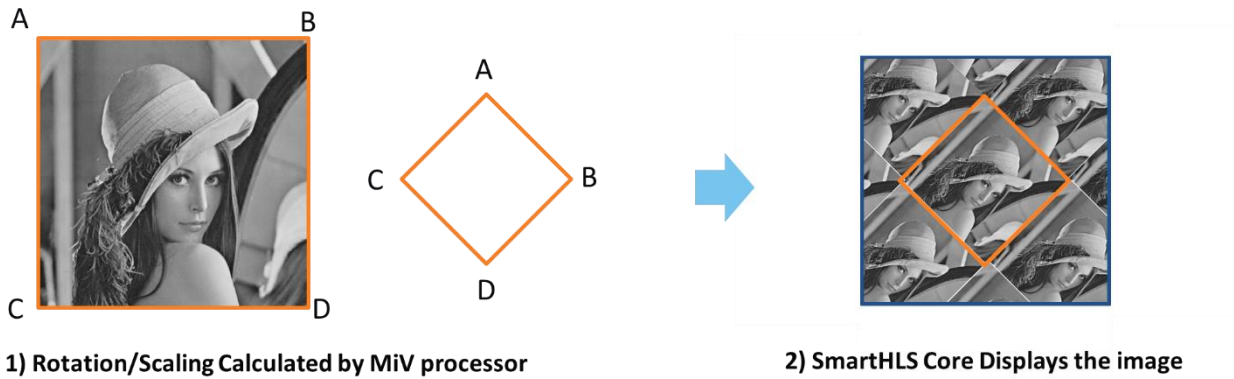


Figure 11: Breakdown of computation between the Mi-V processor and SmartHLS core.



In SmartHLS, open the *axi4_target.h* source file in the project called *texture_mapper*.

```

v texture_mapper
> Includes
> axi4_target.h
> texture_mapper.cpp
> config.tcl

```

The AXI target interface memory-map layout can be found here. These AXI target control status registers hold the inputs sent to the SmartHLS by the Mi-V soft processor. The AX, AY and BX, BY values are corners of the rotated/scaled texture image. The DX1DY, DY1DY, DX2DY, and DY2DY are delta values used during the interpolation in the SmartHLS core.

```

struct TargetLayout {
    //uint64 texture_frame[texture_size*texture_size/8];
    uint32 AX;
    // Offset 0x00040000 (+ 512*512 bytes)
    uint32 AY;
    // Offset 0x00040004
    uint32 BX;
    // Offset 0x00040008
    uint32 BY;
    // Offset 0x0004000C
    int32 DX1DY;
    // Offset 0x00040010
    int32 DY1DY;
    // Offset 0x00040014
    int32 DX2DY;
    // Offset 0x00040018
    int32 DY2DY;
    // Offset 0x0004001C
    // uint32 padding;
    // 8-byte (64-bit) data bus alignment
};

```

The SmartHLS core receives a new set of AXI4 target input values for every single line of the output image. The SmartHLS core will calculate the texture value for all the output pixels and output them to a FIFO interface to the monitor.



If you open *texture_mapper.cpp*, you can find the definition of the *gv* global variable on line 32. This AXI4 target struct will be accessible by the Mi-V processor over the AXI4 target interface.

```
#pragma HLS interface variable(gv) type(axi_slave) concurrent_access(true)
TargetLayout gv;
```

On line 35, we store the texture frame, which is a 512x512 grayscale Lena image, in a block RAM in the SmartHLS core. This memory is declared in C++ by the following line:

```
volatile uint64 texture_frame[texture_size * texture_size / 8];
```

The texture frame memory is uninitialized in the C++. Therefore, we must mark the array as volatile to prevent SmartHLS from optimizing away the computation. Then we must specify the RAM initialization in Libero by going to Configure Design Initialization Data and Memories -> Fabric RAMs. As shown in Figure 12, we specified the memory initialization hex file of the texture_frame RAM to the relative path: "mem_init\lena_8Bit_Greyscale_512x512.hex".

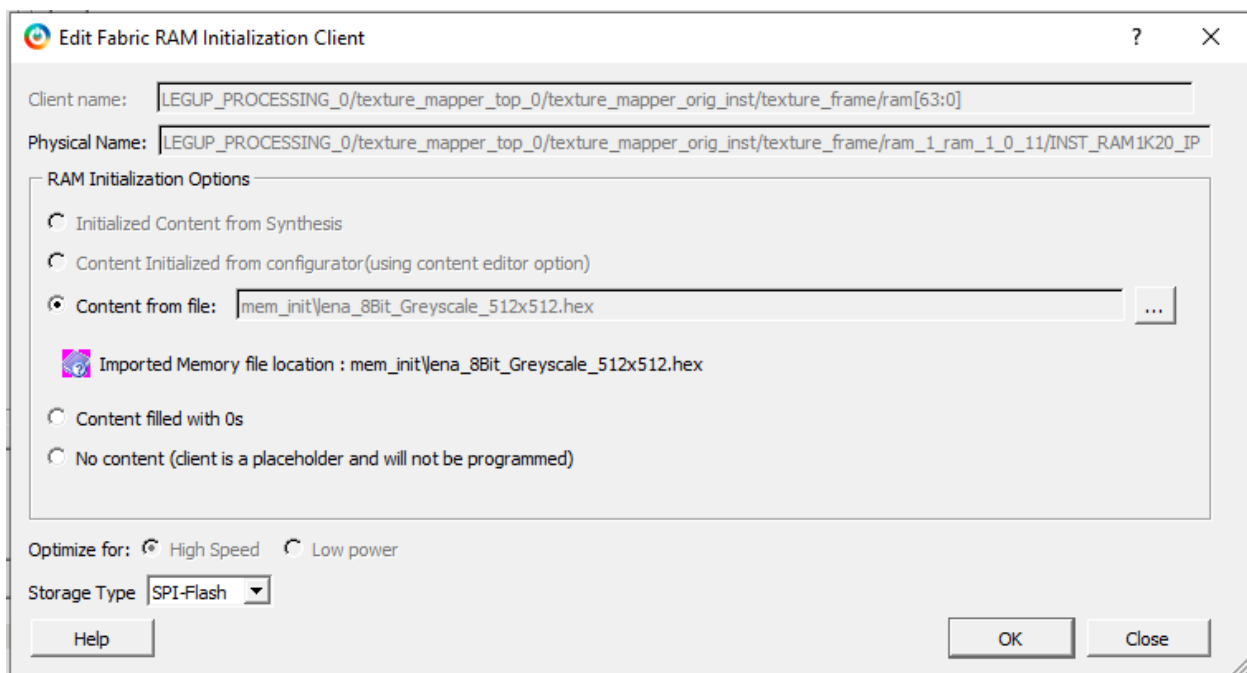


Figure 12: Manual memory initialization of a SmartHLS-generated RAM using Libero

The top-level function is *texture_mapper*, which can be found on line 98.

```
void texture_mapper(uint8 run_loop, uint8 &run_done,
                    FIFO<uint64> &output_fifo) {
    #pragma HLS function top
```

Starting on line 129, the top-level function reads the AXI target inputs from the Mi-V processor and contains a pipelined loop to calculate the output pixels (8 at a time) for the current output line.

```
#pragma HLS loop pipeline
for (n = 0; n < image_width / 8; n++) {
    // calculate texture coordinates of 8 pixels
    p0 = get_texture_pixel(hls_bit_select(cx, 31, 16),
                          hls_bit_select(cy, 31, 16));
    ...
    // concatenate 8 pixels and place result in output FIFO
    output_fifo.write(hls_bit_concat_8(p0, 8, p1, 8, p2, 8, p3, 8, p4,
                                      8, p5, 8, p6, 8, p7, 8));
}
```

At the end of the top-level function, we write a 1 to the run_done output to indicate that the current output line has been finished:

```
run_done = 0x01; // processing phase completed, one line processed
```

This run_done output clears a register external the texture mapper, which then triggers an interrupt handler on the Mi-V to run the processor side code.

14.1 Code Running on the Mi-V Soft Processor

We can perform a rotation of a x, y point around the origin with the following equation:

$$\begin{aligned}x' &= x * \cos(\text{angle}) - y * \sin(\text{angle}) \\ y' &= y * \cos(\text{angle}) + x * \sin(\text{angle})\end{aligned}$$

Inside the SoftConsole code running on the Mi-V soft processor, we perform the rotation inside the rotate_points function when an interrupt is triggered. You can find this code on line 594 in the source file *sc_main_c* found in the *wide_mult_axi* project.

```
void rotate_points(void)
{
    ...
    int A_x_rot = (-scaled_vect1*g_cos_table[g_angle] -
                  scaled_vect2*g_sin_table[g_angle])>>15;
    int A_y_rot = (-scaled_vect1*g_sin_table[g_angle] +
                  scaled_vect2*g_cos_table[g_angle])>>15;
```

Each time the rotate_points function is called the g_angle increments by 1 until wrapping around at 360 degrees. The scaling factors (scaled_vect2, scaled_vect1) perform the zooming in/out, and are incremented each time the function is called up to a maximum and then decremented back down.

The memory mapped addresses of the SmartHLS Texture Mapper AXI target interface are found on line 564 as a series of defines.

```
// SmartHLS Texture Mapper AXI Target Addresses
```



```

#define reg_AX          0x60040000UL
#define reg_AY          0x60040004UL
#define reg_BX          0x60040008UL
#define reg_BY          0x6004000CUL
#define reg_DX1DY       0x60040010UL
#define reg_DY1DY       0x60040014UL
#define reg_DX2DY       0x60040018UL
#define reg_DY2DY       0x6004001CUL

```

At the end of the `rotate_points` function on line 629, we write the updated inputs to the SmartHLS core using memory-mapped writes by casting each address into a volatile pointer of the correct type and then writing to it.

```

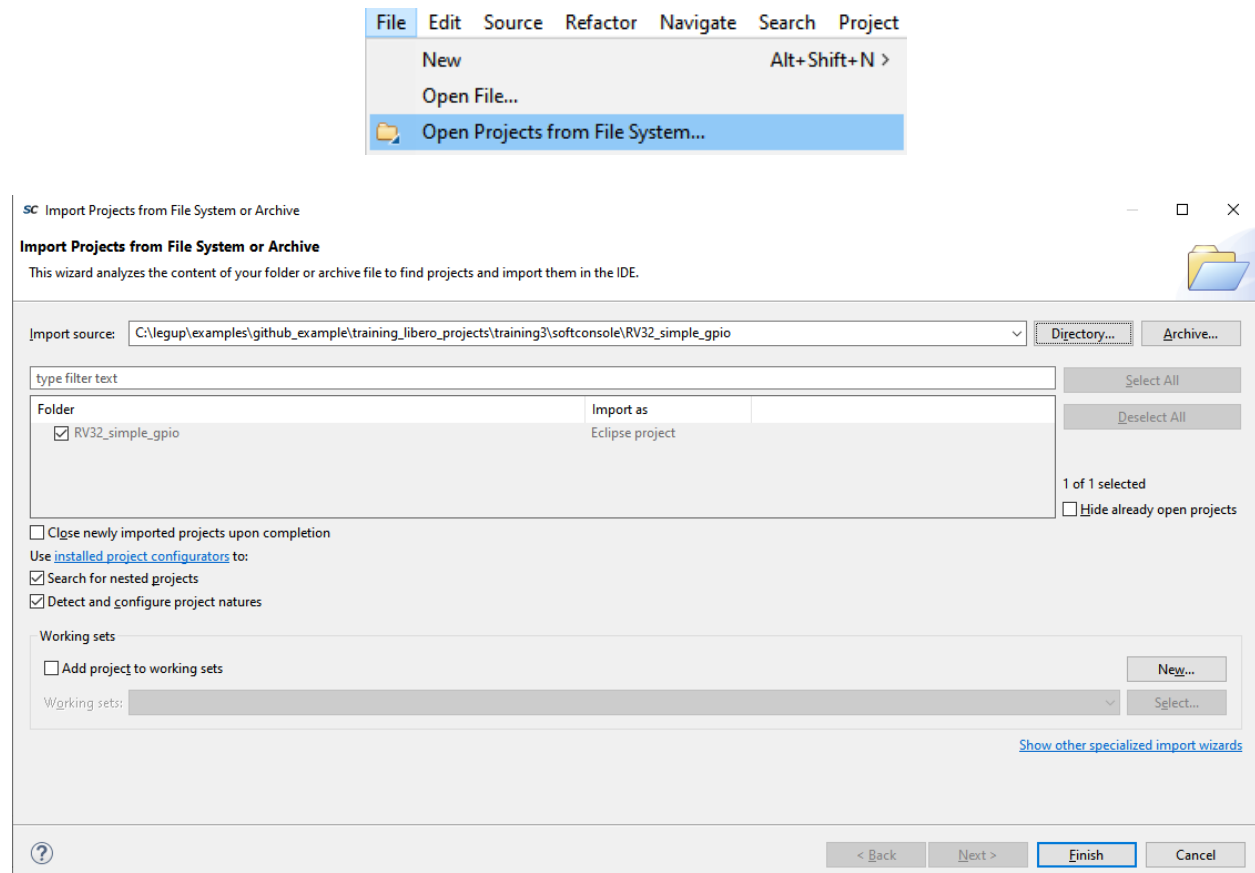
// update registers
*((uint32_t volatile *) reg_AX) = AX<<16;
*((uint32_t volatile *) reg_AY) = AY<<16;
*((uint32_t volatile *) reg_BX) = BX<<16;
*((uint32_t volatile *) reg_BY) = BY<<16;
*((int32_t volatile *) reg_DX1DY) = (((int32_t)CX-(int32_t)AX)<<16)/y_size;
*((int32_t volatile *) reg_DY1DY) = (((int32_t)CY-(int32_t)AY)<<16)/y_size;
*((int32_t volatile *) reg_DX2DY) = (((int32_t)DX-(int32_t)BX)<<16)/y_size;
*((int32_t volatile *) reg_DY2DY) = (((int32_t)DY-(int32_t)BY)<<16)/y_size;

```

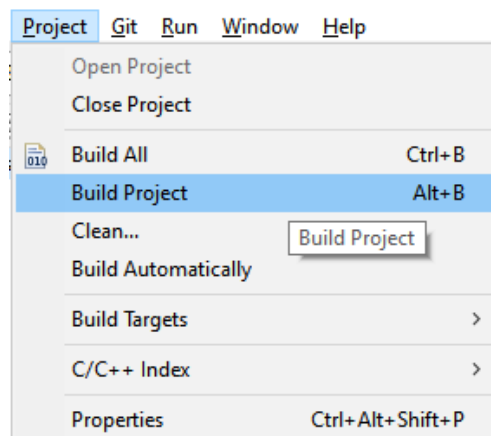
15 Appendix C: Rebuilding the SoftConsole Project

If you make changes to the software for the MiV, follow these steps to generate a new .hex file:

1. Open SoftConsole
2. Open the RV32_simple_gpio project



3. Build the project



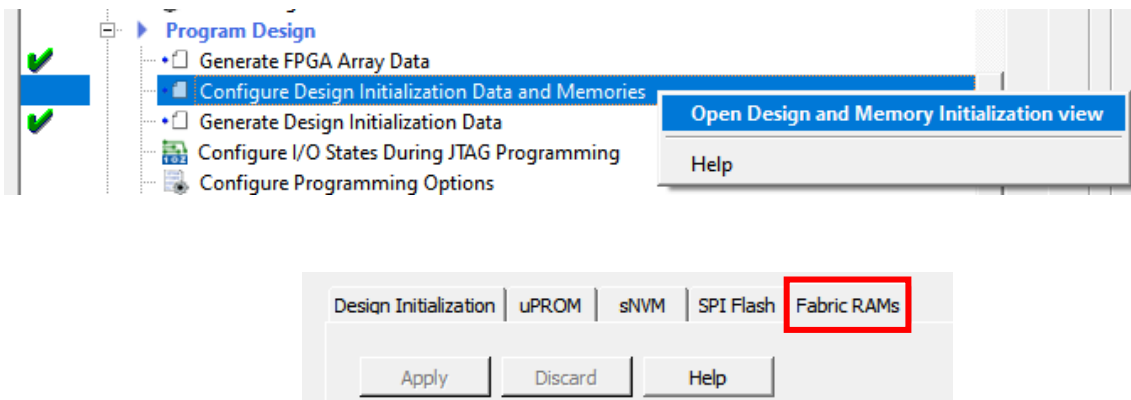
You should see this output:

```
Invoking: GNU RISC-V Cross Create Flash Image
riscv64-unknown-elf-objcopy -O ihex --change-section-lma *-0x80000000 "RV32_simple_gpio.elf" "RV32_simple_gpio.hex"
Finished building: RV32_simple_gpio.hex

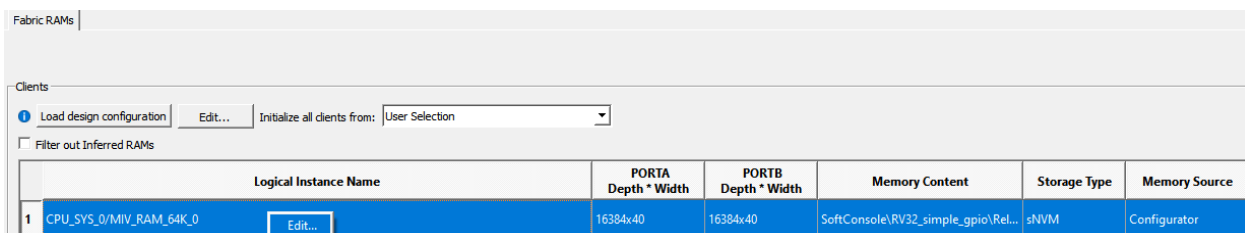
Invoking: GNU RISC-V Cross Print Size
riscv64-unknown-elf-size --format=berkeley "RV32_simple_gpio.elf"
   text    data    bss     dec     hex filename
 19360    3056    8640   31056   7950 RV32_simple_gpio.elf
Finished building: RV32_simple_gpio.siz

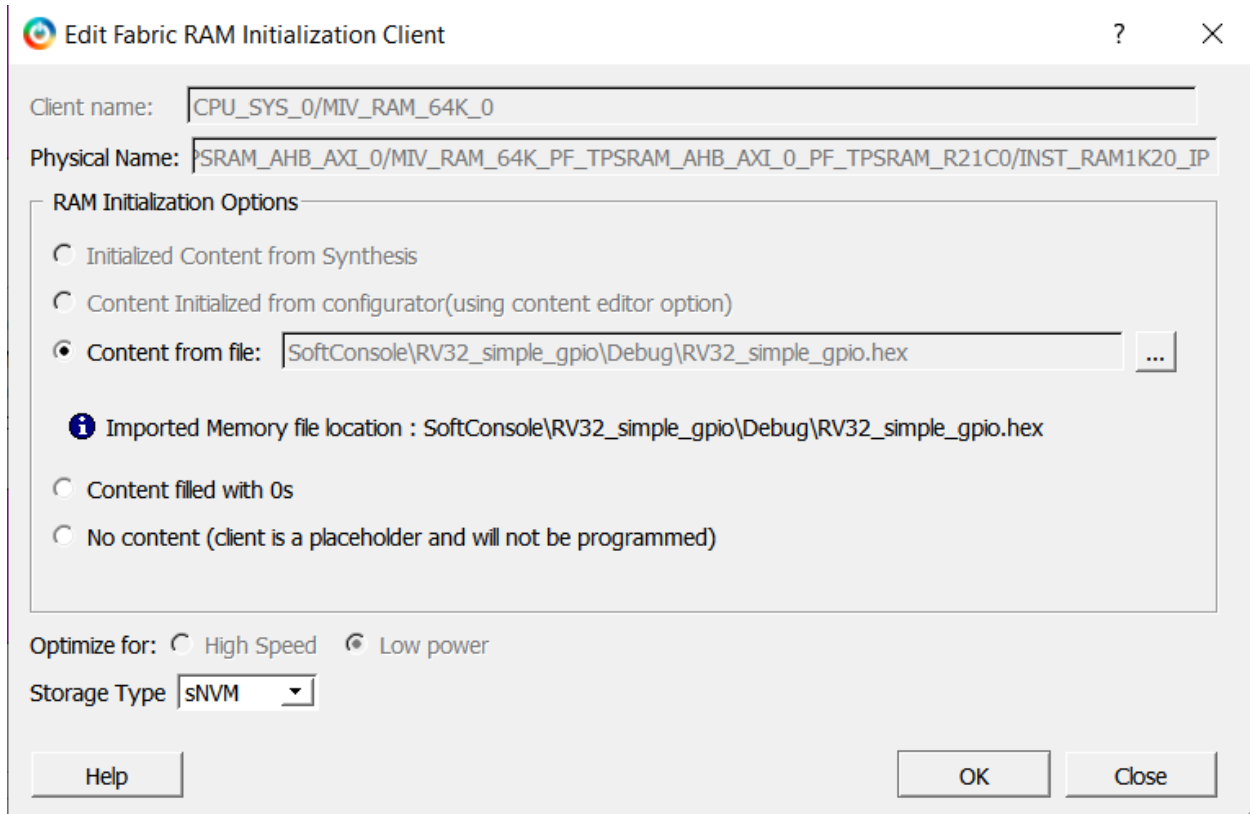
13:39:18 Build Finished. 0 errors, 0 warnings. (took 3s.589ms)
```

4. In Libero, open the Design and Memory Initialization window and select the “Fabric RAMs” tab



5. Right click CPU_SYS_0/MIV_RAM_64K_0 (it should be the first RAM listed) and import the new .hex file in Libero





6. Generate Design Initialization Data and run the remaining steps to program the bitstream

