

SmartHLS™ Training Session 2: Multi-threaded Digit Recognition on the PolarFire® Video Kit

Training
November 24, 2021
Revision 3.0



Table of Contents

Table of Contents	2
Revision History	3
Revision 1.0	3
Revision 2.0	3
Revision 3.0	3
Overview.....	4
SmartHLS™ High-Level Synthesis Overview.....	5
Prerequisites	6
Setting up SmartHLS	7
Parallelism in SmartHLS	11
Instruction-level Parallelism	11
Loop-level Parallelism	12
Thread-level Parallelism	13
Dataflow Parallelism	14
Thread-level Parallelism.....	15
Implementing Threads and Parallel Modules in SmartHLS	15
Producer Consumer Example.....	15
Synchronization	21
Verification: Co-simulation of Multi-threaded SmartHLS Code	24
Verification: Custom Testbench.....	25
Threading vs. Function Pipelining	29
SmartHLS Digit Recognition Application on PolarFire® Video Kit.....	30
Design Overview.....	30
Digit Recognition CNN Implementation	31
Neural Network Basics	32
Digit Recognition CNN Architecture	35
Requirements for Digit Recognition CNN	36
SmartHLS C++ Design for the Digit Recognition CNN	39
Simulation of Digit Recognition CNN.....	51
Checking Quality of Results (QoR): Fmax and Area	53
Comparison Between RTL and HLS	55
Appendix A: Instruction-level Parallelism.....	56

Appendix B: Integrating into SmartDesign	69
Appendix C: Programming and Running the Design on the PolarFire® Video Kit.....	74

Revision History

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

Revision 1.0

First publication of the document.

Revision 2.0

Updated document for SmartHLS™ 2021.2 release.

Revision 3.0

Updated document for outdated figures.

Overview

Time Required: 4 hours

Goals of this Training:

- Explain how different types of parallelism are expressed in SmartHLS™.
- Show a complex SmartHLS design that uses multi-threading: digit recognition.

Training Topics:

- HLS optimizations:
 - Overview of HLS types of parallelism
 - Instruction level, pipeline level, function level parallelism
 - Instruction level parallelism
 - Automatic dependency analysis and scheduling
 - Loop level parallelism
 - Loop pipelining
 - Thread-level parallelism
 - Function pipelining
 - SmartHLS threading
 - Simple example
 - Synchronization
 - Digit Recognition
 - Dataflow parallelism
 - Function pipelining
 - SmartHLS threading
- Verification/Testing:
 - Defining a custom testbench written in Verilog
 - Applying settings for custom testbench in IDE
 - Running custom testbenches
- Hardware interface specification and integration of a generated block into SmartDesign
 - Connect memory interface into SmartDesign
 - Replacing an existing design in SmartDesign
 - Having a custom RTL wrapper around a SmartHLS-generated module

SmartHLS™ High-Level Synthesis Overview

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

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

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.

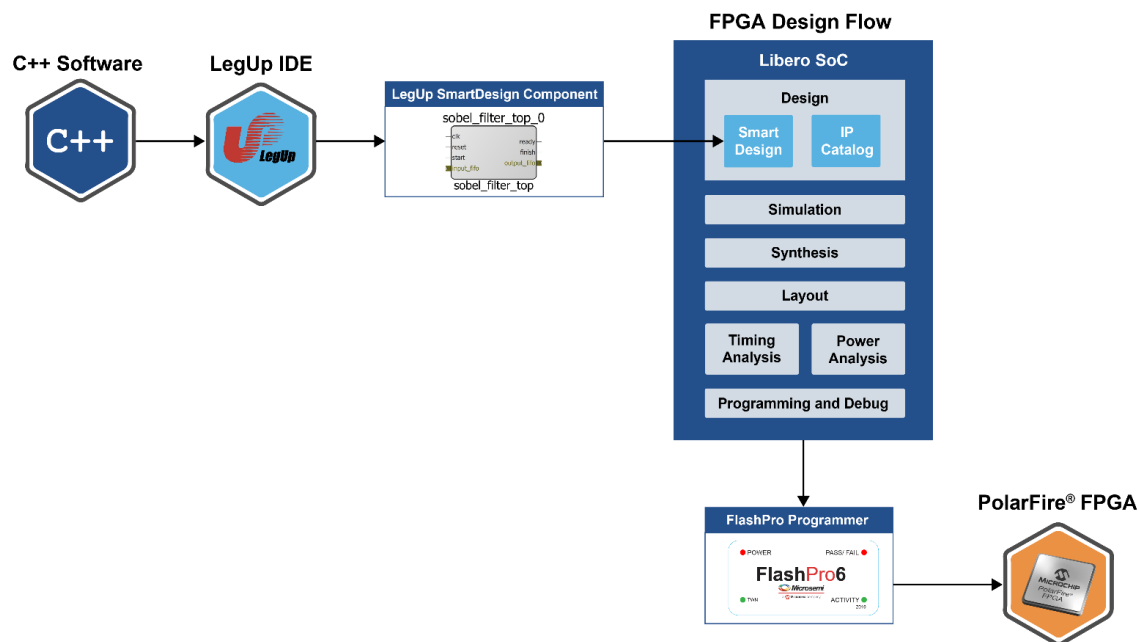


Figure 1: SmartHLS High-level Synthesis Design Flow Targeting a PolarFire FPGA

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](#)

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 Training2 folder for this training.
- Download the file [SmartHLS Training2 Libero.zip](#) (654MB)
MD5SUM: 26ef837916847388b376efc0349b58b9

The following hardware is required:

- PolarFire FPGA Video and Imaging Kit ([MPF300-VIDEO-KIT](#)).
- Monitor with an HDMI input.
- Smartphone to open the digit images used with the Digit Recognition design.

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.

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.

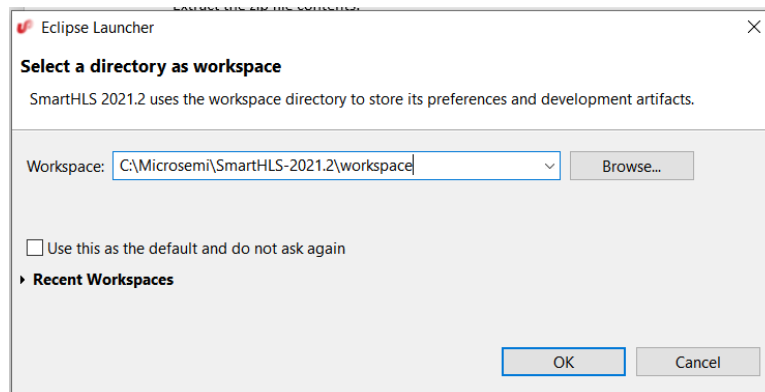


To set up SmartHLS 2021.2:

1. Download the zip file from github if you have not already (see Prerequisites). Extract the contents of the zip file. We will use the Training2 folder of the extracted content for this training.

Warning: Make sure to extract to a directory with a short path to avoid long path issues.

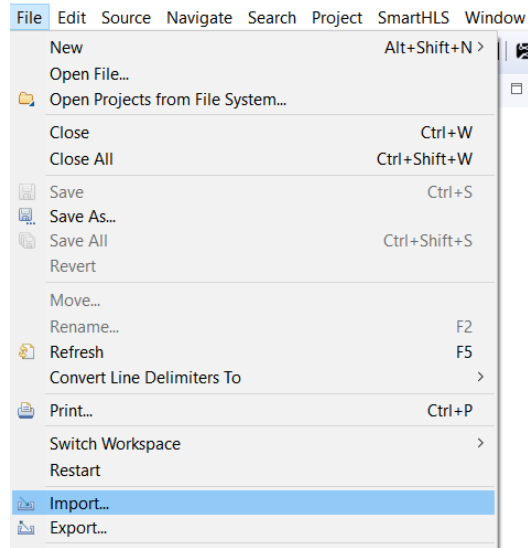
2. Open SmartHLS 2021.2 and choose a workspace.



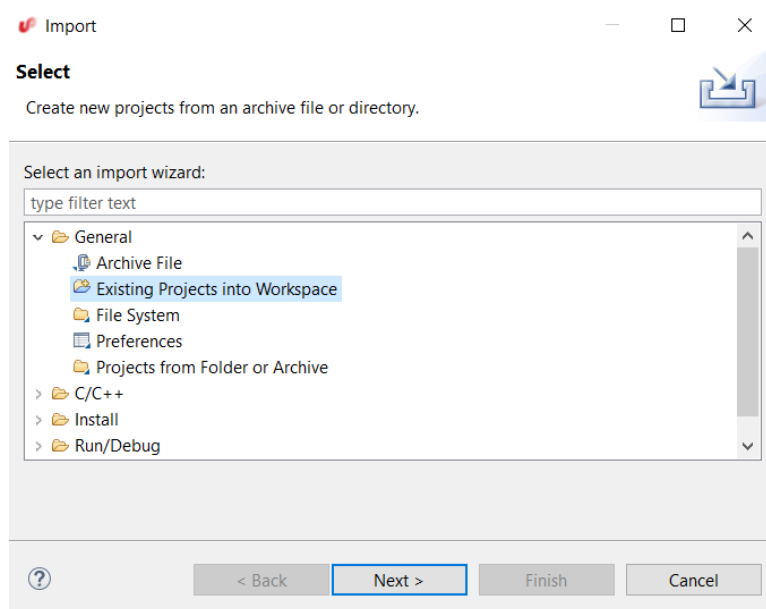
You may want to select a new folder so you can have a blank workspace for this training.

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

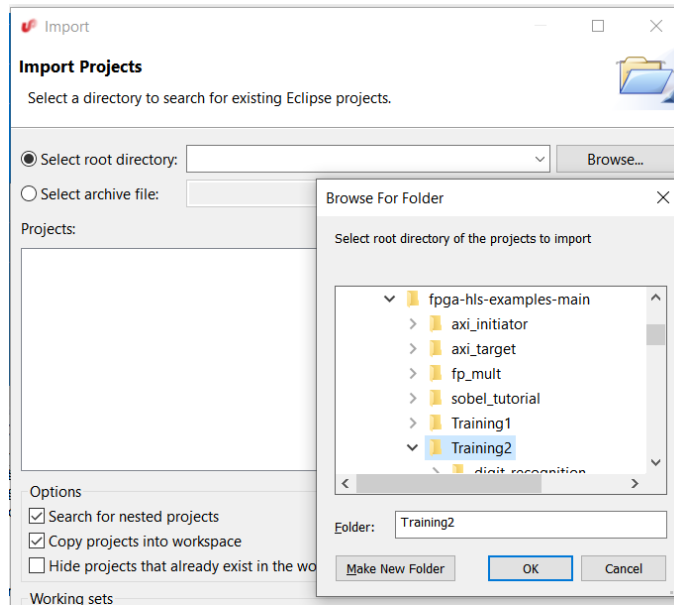
3. Select File -> Import...



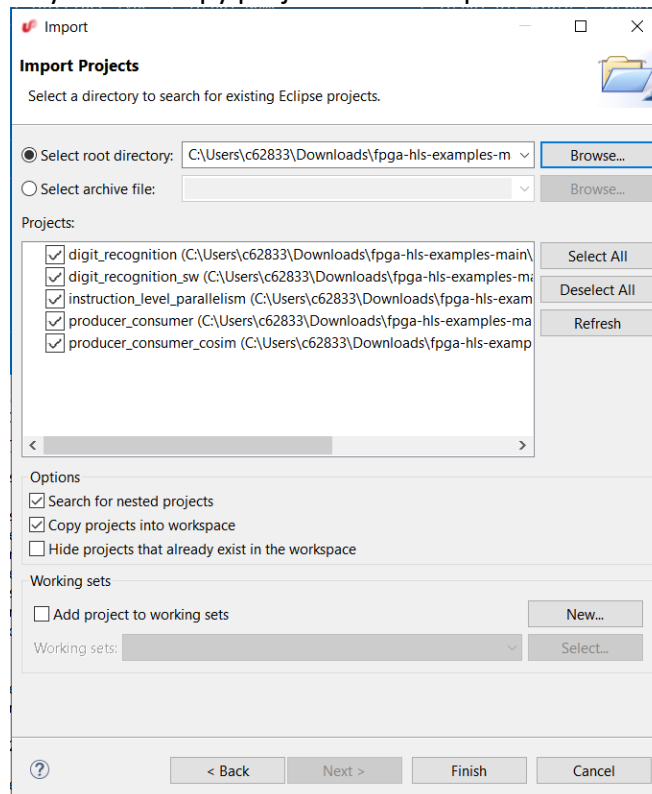
4. In the Import window, select General->Existing Projects into Workspace and then click Next.



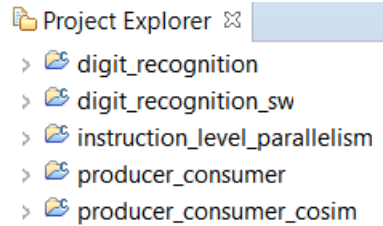
5. In the next step, select "Select root directory" and then click Browse... In the popup window browse to the Training2 directory and click OK.



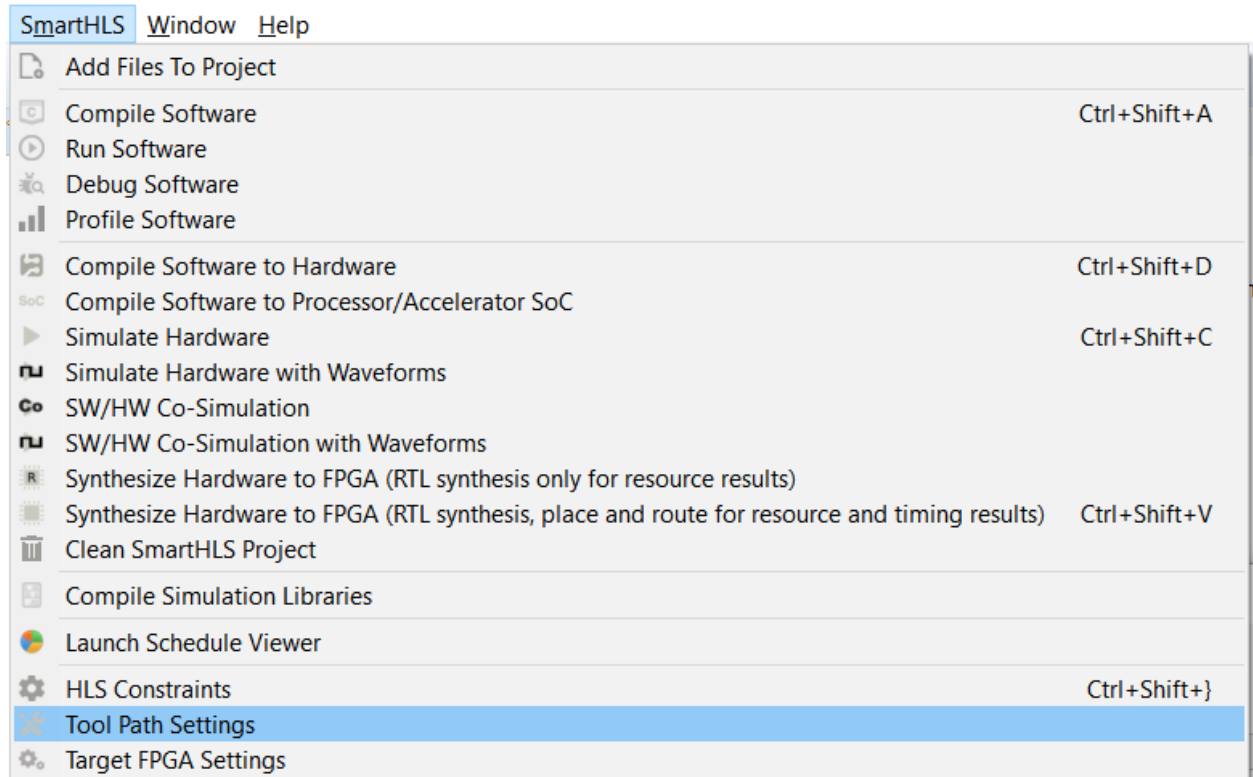
6. Now in the Projects box you should see that all SmartHLS projects have been selected. Optionally you may check “Copy projects to workspace”. Click Finish to import.



7. After importing you should see 5 projects in the Project Explorer on the left.



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

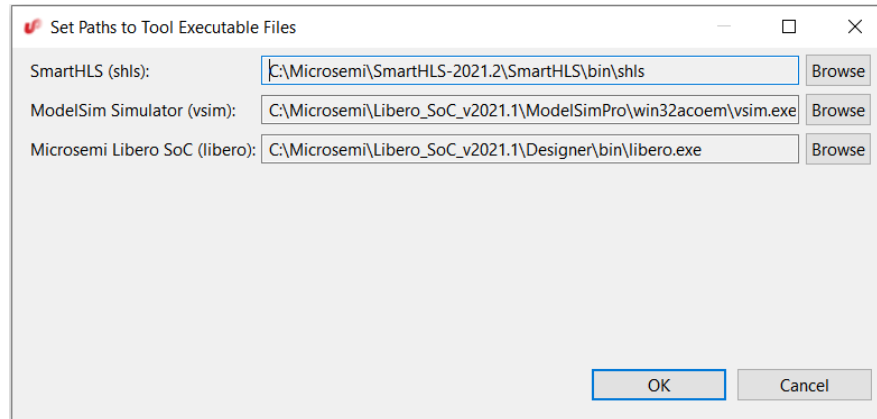


9. Make sure the tool paths to vsim.exe and libero.exe are properly set to:

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.

Parallelism in SmartHLS

When designing a hardware block using SmartHLS, parallelism is the main way of achieving performance gain. As mentioned in the [SmartHLS user guide](#), there are four main kinds of parallelism in SmartHLS: instruction-level, loop-level, thread-level, and dataflow parallelism. These concepts have some overlap between them, but they all focus on running as many tasks in parallel as possible, reducing total runtime of the generated hardware circuit. We will look at each of them in the sections below.

Goals of this section:

1. Understand different levels of parallelism found in SmartHLS.
2. Understand when to use each kind of parallelism.

Instruction-level Parallelism

Instruction-level parallelism is a common optimization in computer architecture and software. This kind of optimization attempts to reduce a series of computations into dependent and independent steps, then schedule as many independent steps to run as early as possible.

For example, in the code snippet in Figure 2, we have a sequence of add operations. Since none of the computations depend on the result of a previous computation, there are no dependencies between the three operations and all the add operations can run in parallel without affecting the functionality of the program.

```

e = b + c;
f = c + d;
g = b + d;

```

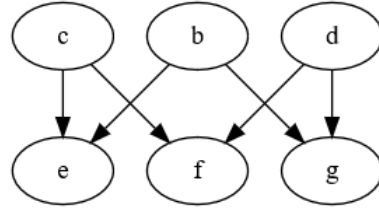


Figure 2: Source code and data dependency graph for 3 parallel additions

Instruction-level parallelism is especially relevant in software as C/C++ source code is inherently serial. As SmartHLS uses C/C++ as the input language to generate hardware, instruction-level parallelism is also used by SmartHLS to create performant hardware. SmartHLS will automatically apply this optimization to the software description, analyzing dependencies and scheduling as many independent instructions within the same **basic block** to start as early as possible.

A basic block is a group of instructions that always run together with a single entry point at the beginning and a single exit point at the end. Instruction-level parallelism only applies to instructions within the same basic block. The total amount of parallelism extracted with instruction-level parallelism is limited because in most programs the average basic block contains less than 10 instructions.

More details and examples of this kind of parallelism can be found in Appendix A.

Loop-level Parallelism

Loop-level parallelism occurs when iterations of a loop are run in parallel. SmartHLS leverages loop-level parallelism when pipelining a loop to allow multiple loop iterations to execute in parallel. Pipelining is a common hardware optimization to increase the throughput of a circuit that repeatedly processes data in a series of steps and achieve higher hardware utilization. We have already seen this kind of parallelism in the Sobel Filter tutorial as well as in the previous training on the Canny Edge Detection Filter.

When the *loop pipeline* or *function pipeline* SmartHLS pragmas are applied in the source code, SmartHLS will automatically generate a hardware pipeline for the loop or function body to take advantage of loop-level parallelism. The total amount of parallelism extracted with loop-level parallelism can be significant since compute-intensive programs spend most of their runtime executing loops. A pipeline schedule is shown in Figure 3.

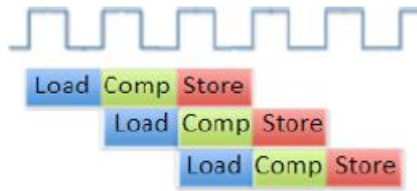


Figure 3: Pipeline schedule for a pipeline with three stages

Thread-level Parallelism

Thread-level parallelism, also called task-level or function-level parallelism, occurs when multiple software functions are run in parallel. By default, SmartHLS maps functions that execute sequentially in software into hardware modules that also run sequentially in hardware. We can use thread-level parallelism in SmartHLS to explicitly specify in software when we want multiple hardware modules to run in parallel.

In SmartHLS, we use a threading software API to specify functions that run in parallel. SmartHLS will map each threaded software function into a hardware module that runs in parallel. Figure 4 illustrates two software threads: Thread 1 and Thread2, which SmartHLS will map into two hardware modules that will run in parallel: Module 1 and Module 2.

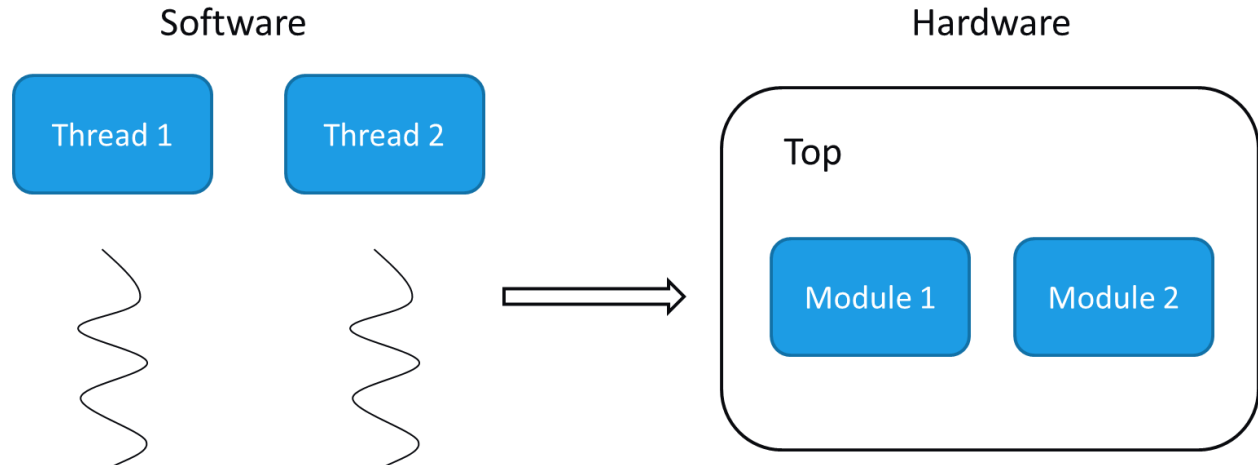


Figure 4: Parallel software threads map to parallel running hardware modules

Each thread in a software program executes in parallel while sharing the same memory space with the other threads. The parallel software threads do not need to do related work and the threads can work independently. You can use thread level parallelism any time you want to generate hardware blocks within the same design that can run in parallel.

Threading will be covered in depth in section 0.

Dataflow Parallelism

Dataflow parallelism is when data is streamed into successive tasks that run in parallel. We saw an example of this type of parallelism in Training 1 in the Canny Edge Detection design. The Canny Edge filter can be broken up into four tasks corresponding to the Gaussian blur, Sobel filter, non-maximum suppression, and hysteresis filters. Each of these tasks run in succession to contribute to processing a piece of data, and all tasks can run in parallel across different data. This is like a pipeline where the building blocks are functions instead of instructions. Figure 5 shows the Canny design, where multiple filter functions are connected with FIFOs and video pixels are continuously streamed in. Figure 6 shows that dataflow parallelism can be thought of as a pipeline of tasks. Dataflow parallelism is usually combined with pipelining inside the parallel functions to achieve better performance in each individual task.

Dataflow parallelism can be used whenever we have a task that can be broken down into a sequence of tasks that can all run in parallel on a data stream. We can implement dataflow parallelism with either function pipelining as we saw in Training 1, or with threading which we will see in section 0. The differences between the two will be discussed in section 0.

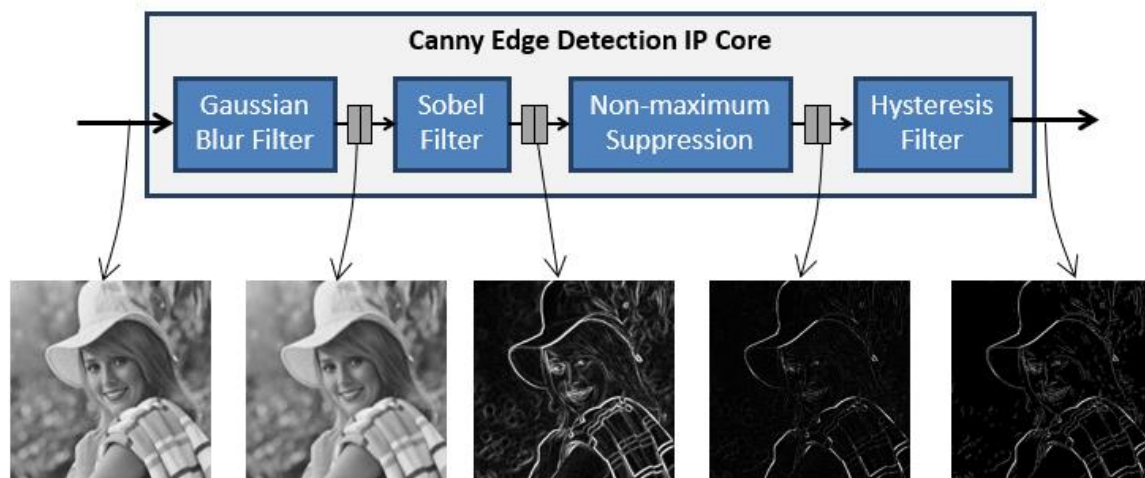


Figure 5: Dataflow Parallelism found in the Canny Edge example design.



Figure 6: Pipeline schedule where each block is a task instead of an instruction. GBF: Gaussian Blur Filter, SF: Sobel Filter, NMS: Non-maximum Suppression, HF: Hysteresis Filter.

Thread-level Parallelism

Threading is one of the ways SmartHLS supports defining modules that run in parallel within a single SmartHLS project. SmartHLS will not run any generated modules in parallel unless explicitly told to by using the *function pipeline* SmartHLS pragma or by running functions in threads.

When functions are run in threads, SmartHLS will automatically generate the control logic to run the resulting modules in parallel. Code that runs serially in software before and after creating the threads will also run serially before and after the module starts in hardware.

Goals of this section:

1. Learn how to use the SmartHLS threading library and thread API.
2. Understand synchronization and synchronization primitives.
3. Understand which threaded designs can run SmartHLS co-simulation and which designs will require a custom RTL testbench.
4. Learn how to write and run a custom RTL testbench.
5. Understand when to use threads and when to use function pipelining.

Implementing Threads and Parallel Modules in SmartHLS

Creating parallel modules using threads is easy in SmartHLS 2021.2. SmartHLS comes with a threading library with a simple API for creating threads. Detailed information on this API can be found in the [SmartHLS User Guide](#). SmartHLS previously supported POSIX threads (pthreads) but the pthreads API was deprecated in SmartHLS 2021.2 and the SmartHLS thread API is now the recommended way to create threads. Here we will present the basics on how to use threads.

Producer Consumer Example

A classic example of parallel blocks in both hardware and software is of a producer creating data and a consumer using the data. We have already seen this used in the Canny edge detection example, which was implemented as function pipelined modules with FIFOs between them. Function pipelining is sufficient for creating streaming, FIFO connected producers and consumers. However, if we wanted to use a memory or add any other interface on the consumer, we would need to use threading. This is because non-FIFO interfaces are not supported by function pipelining. A description of when to use function pipelining vs threading can be found in Section 0.

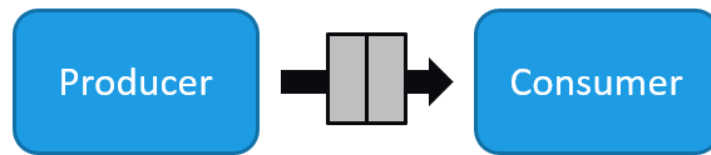
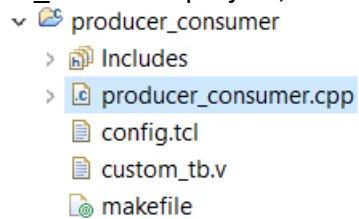


Figure 7: Producer and consumer with a communication channel between them



To see an implementation of this pattern using threads, go to SmartHLS and open the `producer_consumer` project, then open the `producer_consumer.cpp` source file.



This file implements a producer consumer pattern with the restriction that a memory is used as the communication channel between the producer and consumer. The shared memory buffer between them must be empty before the producer can start producing and the buffer must be full before the consumer can begin consuming.



After importing projects into SmartHLS you may see red underlines on function calls with the message that the function could not be resolved as shown in Figure 8.

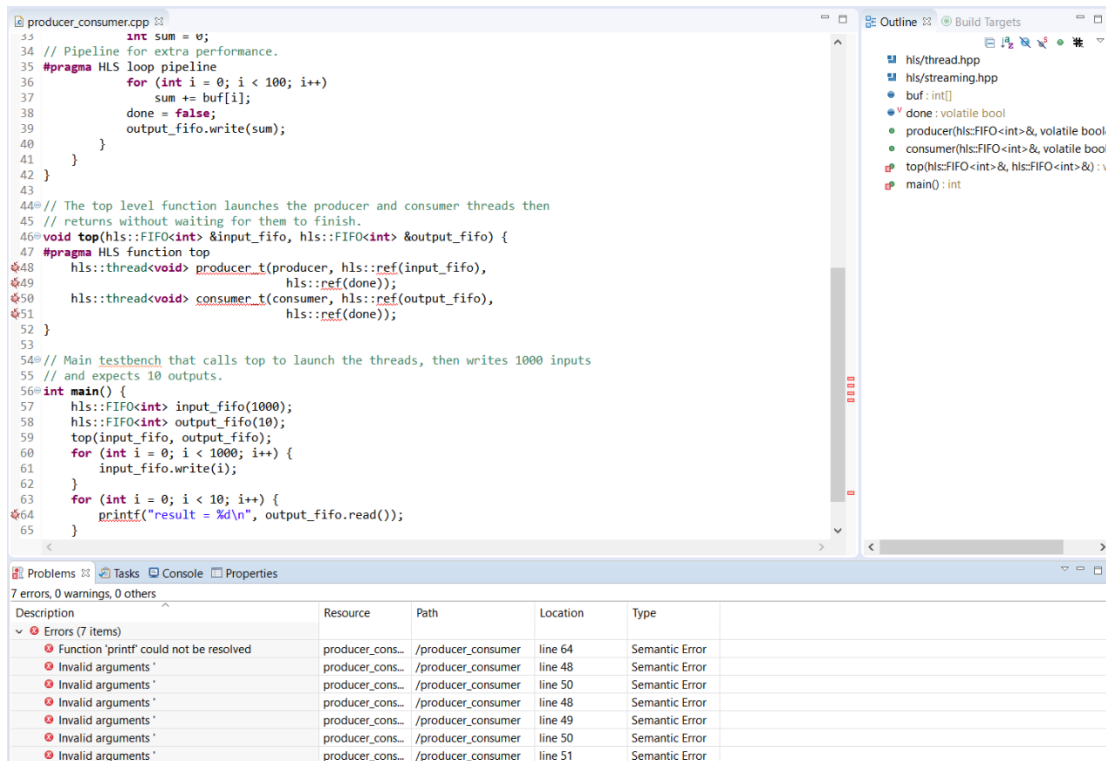
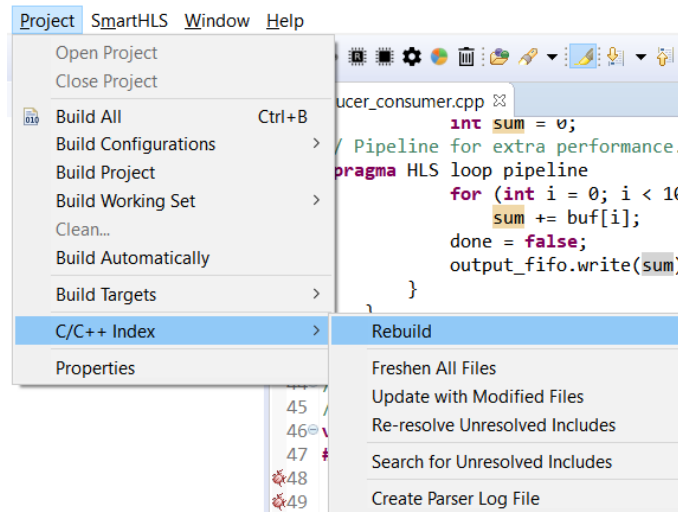


Figure 8: Eclipse indexing error causing function calls to be underlined in red

To fix these red underlines, you can go to the Project drop down menu and select C/C++ Index-> Rebuild. This will fix any Eclipse indexing issues which results in library functions being underlined in red.



At the top of the file producer_consumer.cpp file, we have included to hls/thread.hpp header file to be able to use the threading API.

- 1 // Import hls/thread.hpp to access the SmartHLS thread library and API.
- 2 #include "hls/thread.hpp"

```
3 #include "hls/streaming.hpp"
```

On line 8, we have two global variables shared between the two threaded functions. The variable `buf` will be the buffer between the producer and consumer. The variable `done` will be used for synchronization between the functions. We will talk about synchronization in more detail later. The synchronization variable must be declared as **volatile**. The volatile keyword tells the compiler that reads and writes to this variable must not be optimized. Usually, this keyword is used in firmware to force a memory location to be read in case something in the hardware has changed its value. In our case, we use it because the compiler does not consider accesses from other running threads and can incorrectly optimize the code.

```
9 // The contention free pragma tells SmartHLS not to generate an arbiter.
10 #pragma HLS memory impl variable(buf) contention_free(true)
11 int buf[100] = {0};
12 volatile bool done = false;
```

By default, SmartHLS will generate a round-robin arbiter for each variable (memory block) shared by a function running in a thread and any other software, threaded or otherwise. An arbiter is a hardware block that enforces the order of accesses to a shared resource. Round-robin arbiters ensure that all requestors get granted an equal use of the shared resource. Arbiters ensure that conflicting reads and writes do not happen, for example 3 reads to a dual-ported RAM in the same cycle.

If you are sure that there will never be conflicting accesses to a memory variable in any given cycle, you can use the contention free pragma to tell SmartHLS not to generate an arbiter for that memory. This simplifies the generated circuit slightly. In this implementation, we declare `buf` as contention free as the consumer cannot access the buffer until the producer is done writing and the producer cannot access the buffer until the consumer is done reading, therefore no conflicting accesses can occur. The memory `done` does not need an arbiter either, since one thread will be polling `done` until the other thread writes to it, there will only be a maximum of two reads, or one write, and one read per cycle. However, we do not declare the `done` variable as contention free to be able to show what the generated arbiter looks like in RTL. Note, the contention free pragma must precede the variable declaration unlike pragmas such as the function top pragma. For more information on where pragmas need to be defined, check the [SmartHLS Pragmas Manual](#).



To check for generated arbiters, compile the design to hardware (🔧) and open the generated Verilog file “producer_consumer.v”. On line 210, you will find an instantiation for a round robin arbiter for the variable `done`. If you search for “round_robin_arbiter” and cycle through the results, you will find that no such arbiter has been generated for `buf` which we declared as contention free.

```
210 round_robin_arbiter round_robin_arbiter_inst_arbiter_done_a (
    ...
216 );
```

Back in `producer_consumer.cpp` starting on line 14, we have the two functions to be threaded: `producer` and `consumer`. The `producer` function has an input FIFO and the `done` control variable as arguments. The `producer` will check that the `consumer` has finished consuming by checking `done`, then fill the buffer with more data and set `done`. The `consumer` function has an output FIFO and the `done` control variable as arguments. The `consumer` will check that the `producer` has finished filling the buffer by checking `done` and then read from the buffer and accumulate. The output will be written to an output FIFO and `done` is reset. For functions that are meant to run continuously in hardware, we use a while loop that always has a true condition to force the thread to run forever. We also use loop pipelining on the inner loops to increase performance.

```

14 // The producer function continuously reads from input fifo and writes to
    buf
15 // then waits for consumer to finish consuming.
16 void producer(hls::FIFO<int> &input_fifo, volatile bool &done) {
17     while (1) {
18         if (!done) {
19             // Pipeline for extra performance.
20             #pragma HLS loop pipeline
21             for (int i = 0; i < 100; i++)
22                 buf[i] = input_fifo.read();
23             done = true;
24         }
25     }
26 }
27
28 // The consumer function continuously waits for the producer to finish
    producing
29 // then reads from buf, sums and writes the result to output fifo.
30 void consumer(hls::FIFO<int> &output_fifo, volatile bool &done) {
31     while (1) {
32         if (done) {
33             int sum = 0;
34             // Pipeline for extra performance.
35             #pragma HLS loop pipeline
36             for (int i = 0; i < 100; i++)
37                 sum += buf[i];
38             done = false;
39             output_fifo.write(sum);
40         }
41     }
42 }

```

At this point we have declared the two functions but have not yet run them in threads. To run them in threads, we must create thread objects using the SmartHLS threading API. On line 46, we have the top-level function that passes the arguments to the threaded functions and then returns. The API for running a function as a thread is to provide the function as the first argument to a thread type object. Here we declare two void return type thread objects called `producer_t` and `consumer_t` that runs the `producer` and `consumer` functions. The void template parameter signifies that the threaded functions return void and matches the function

signatures of the two function. The first arguments following the function to be run are the arguments for that function. Here we have the FIFOs and the `done` control variable passed in by reference. To pass arguments by reference to a thread, we must use the `hls::ref` wrapper function around the argument.

```

44 // The top level function launches the producer and consumer threads then
45 // returns without waiting for them to finish.
46 void top(hls::FIFO<int> &input_fifo, hls::FIFO<int> &output_fifo) {
47 #pragma HLS function top
48     hls::thread<void> producer_t(producer, hls::ref(input_fifo),
49                                   hls::ref(done));
50     hls::thread<void> consumer_t(consumer, hls::ref(output_fifo),
51                                   hls::ref(done));
52 }

```

You may wonder why the shared variables `buf` and `done` variables cannot be local to `top` and be passed as arguments to the threads. This is because the top-level function runs the threads but does not wait for them to finish before returning. Any variables local to the top-level function will be unusable after the top-level function returns, resulting in the threads accessing invalid memory. This forces us to declare `buf` and `done` as global variables to ensure correct behavior.

Next, on line 56, we have a very simple software testbench to check the functionality in software. SmartHLS currently does not support co-simulation for designs that use threads that run forever so we cannot use this software testbench to run co-simulation. However, we can replicate the test in a custom testbench and use the software inputs and outputs as a reference. We first call `top` while the FIFOs are empty to launch the producer and consumer threads. The producer thread will block on the empty input FIFO and the consumer thread will continuously poll `done` until we give input via the input FIFO.

```

54 // Main testbench that calls top to launch the threads, then writes 1000
55 // and expects 10 outputs.
56 int main() {
57     hls::FIFO<int> input_fifo(1000);
58     hls::FIFO<int> output_fifo(10);
59     top(input_fifo, output_fifo);
60     for (int i = 0; i < 1000; i++) {
61         input_fifo.write(i);
62     }
63     for (int i = 0; i < 10; i++) {
64         printf("result = %d\n", output_fifo.read());
65     }
66     return 0;
67 }

```



Now click the compile software button (🔧) and then run the software (🏃) to verify its functionality. You should see the following output in the console:

```
result = 4950
```

```
result = 14950
result = 24950
result = 34950
result = 44950
result = 54950
result = 64950
result = 74950
result = 84950
result = 94950
```



Now close the producer_consumer project and opened files.

Synchronization

Before we move on to testing the producer_consumer design in hardware, we will discuss synchronization in more detail. Synchronization enforces certain order of execution between threads running in parallel when performing shared work. In the producer consumer example, we used synchronization to ensure that the producer thread wrote to the buffer before the consumer thread could read from the buffer. This kind of agreement was implemented with the *done* variable to facilitate handshaking between the two processes.

In general, there are other tools in software that can be used to synchronize processes that are useful in different scenarios. Synchronization primitives are mechanisms to allow programmers to implement synchronization between processes. The two kinds of **synchronization primitives** are available in SmartHLS are **mutexes** and **barriers**. Figure 9 gives a high-level block diagram for the interaction between parallel modules and synchronization primitives. Each synchronization primitive needs an arbiter to ensure that only one module can access the shared resource (mutex or barrier) at a time.

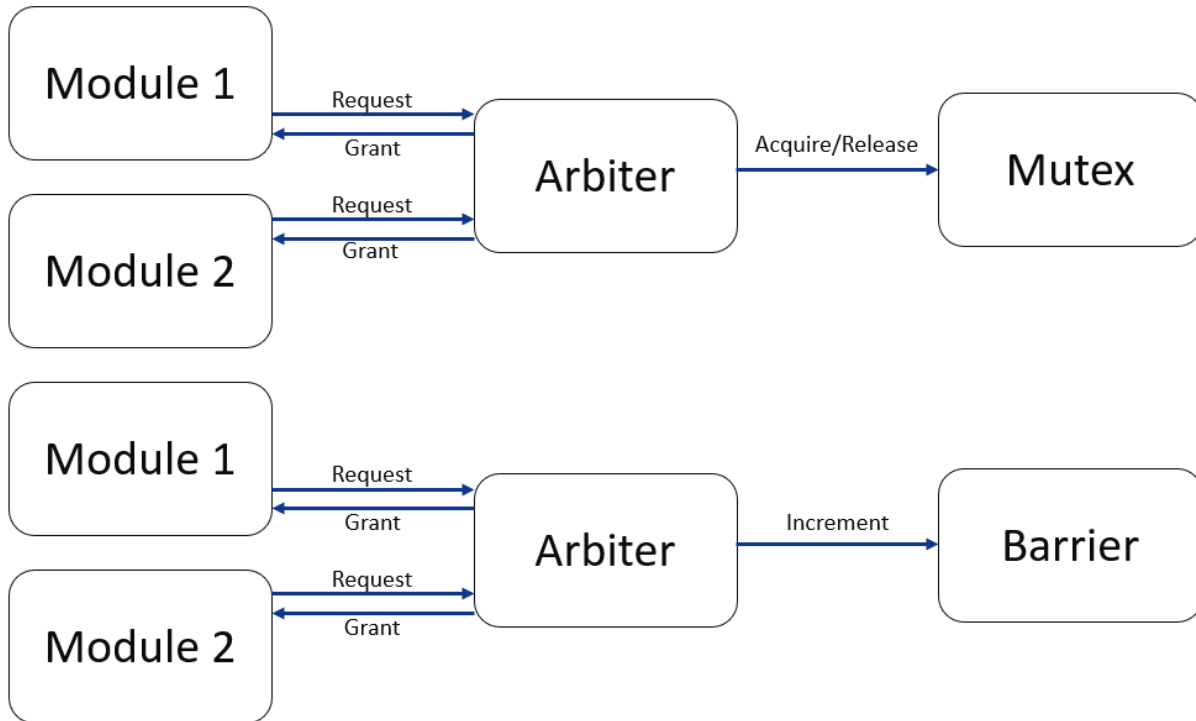


Figure 9: Block diagrams for generated hardware with two parallel modules accessing a mutex and a barrier

As mentioned previously, we used a `done` variable to synchronize the producer and consumer processes to prevent them from accessing a shared buffer out of turn. This kind of synchronization problem is called “mutual exclusion” of operations. Mutual exclusion enforces that some operations need to be done on a resource **atomically** for the resource to remain in a valid state. An atomic operation is one that happens completely or not at all, without any interruption in between. This kind of problem is commonly found in threaded functions and requires synchronization to correctly order operations. To make sure that a series of operations are atomic in both software and hardware, we can use a mutex (short for mutual exclusion). When a thread acquires a mutex, another thread cannot acquire the same mutex and must wait until the holder releases it. If we try to acquire a mutex before all operations on shared resources that are supposed to be atomic and release them afterwards, then we can guarantee that no other thread will operate on them until the operations are finished. When mutexes are used in software, the same functionality is replicated by SmartHLS in hardware.

A generic example on how to use the mutex can be found in the [User Guide](#), where a global mutex is locked and unlocked to ensure that the function body runs atomically.

```

hls::mutex m;

void f() {
    m.lock();
    ...
}

```

```

    m.unlock();
}

```

In our simple producer consumer example, we used a done signal to achieve a similar kind of mutual exclusion to prevent both the producer and consumer from accessing the shared resource at the same time. In more complicated scenarios where there are multiple readers and writers, and where you might not care about enforcing a specific access order, it is easiest to use a lock to achieve mutual exclusion in both software and hardware.

Another type of common synchronization problem is to enforce shared progress between multiple threads that are operating on a related task. For example, if we split an array in half and have two threads operate on the two halves in parallel, we would want the threads to finish at the same time before moving on to the next array input. In the case that one half is slightly larger than the other half, we would need some way to make sure the thread with the smaller half does not move ahead before the other thread finishes. In this case, we can use a barrier to synchronize them. A call to a barrier will block threads until all threads reach the barrier and would prevent one thread from moving on before the other threads are ready. A barrier can handle any number of threads but must be given the number during creation.

A generic example on how to use the barrier can be found in the [User Guide](#), where a global barrier is initiated in main for the two threads that will use it. Inside the threaded function, a call to wait is made to block the thread until the two threads reach the barrier.

```

hls::barrier bar;

void f1() {
    ....
    bar.wait();
}

void f2() {
    ....
    bar.wait();
}

int main() {
    bar.init(2);
    auto t1 = hls::thread<void>(f1);
    auto t2 = hls::thread<void>(f2);
    ....
}

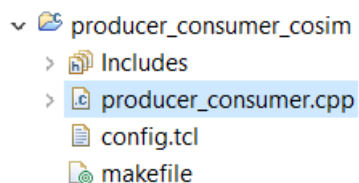
```

Verification: Co-simulation of Multi-threaded SmartHLS Code

As mentioned before the `producer_consumer` project cannot be simulated with co-simulation. This is because the `producer_consumer` project has threads that run forever and do not finish before the top-level function returns. SmartHLS co-simulation supports a single call to the top-level function which then calls functions in threads. All threads run from inside the top-level function should finish running before the end of the function. If your code does not meet this requirement then you need to write a custom RTL testbench as described in the next section. In this section, we will look at a modified version that has threads that finish before the top-level function returns and run co-sim for this project.



Open the `producer_consumer_cosim` project and open the `producer_consumer.cpp` source file.



In this version of producer consumer, we have modified the threads to run only 10 iterations then return.

```
14 // The producer function reads from input fifo and writes to buf then waits
    for
15 // consumer to finish consuming 10 times.
16 void producer(hls::FIFO<int> &input_fifo, volatile bool &done) {
17     for (int i = 0; i < 10; i++) {
18         while (done)
19             ;
20 // Pipeline for extra performance.
21 #pragma HLS loop pipeline
22         for (int i = 0; i < 100; i++)
23             buf[i] = input_fifo.read();
24         done = true;
25     }
26 }
27
28 // The consumer function waits for the producer to finish producing then
    reads
29 // from buf, sums and writes the result to output fifo 10 times.
30 void consumer(hls::FIFO<int> &output_fifo, volatile bool &done) {
31     for (int i = 0; i < 10; i++) {
32         while (!done)
33             ;
34         int sum = 0;
35 // Pipeline for extra performance.
36 #pragma HLS loop pipeline
37         for (int i = 0; i < 100; i++)
38             sum += buf[i];
```



```

39         done = false;
40         output_fifo.write(sum);
41     }
42 }

```

The top-level function waits for both threads to finish by using a call to join before returning.

```

44 // The top level function launches the producer and consumer threads then
    waits
45 // for them to finish before returning.
46 void top(hls::FIFO<int> &input_fifo, hls::FIFO<int> &output_fifo) {
47 #pragma HLS function top
48     hls::thread<void> producer_t(producer, hls::ref(input_fifo),
49                                     hls::ref(done));
50     hls::thread<void> consumer_t(consumer, hls::ref(output_fifo),
51                                     hls::ref(done));
52     producer_t.join();
53     consumer_t.join();
54 }

```



This type of thread use is supported by co-simulation. Run compile to hardware (🔧) then run co-sim (🎮). You should see the following in the console output. If this project's thread use was not supported by co-sim, SmartHLS would error out without running co-sim.

```

Number of calls:      1
Cycle latency:       2,091
SW/HW co-simulation: PASS
...
20:03:56 Build Finished (took 2m:57s.810ms)

```



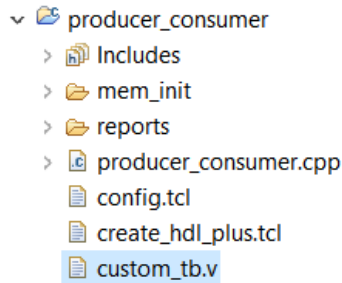
Now close the producer_consumer_cosim project and opened files.

Verification: Custom Testbench

Since our original producer consumer code does not meet the requirements for co-simulation, we must write a custom testbench to verify the generated hardware. For the producer-consumer design we have a Verilog testbench that continuously sends FIFO data into the producer and checks for the correct outputs from the consumer.



Re-open the producer_consumer project and open the custom_tb.v source file.



This is a simple testbench that feeds streaming input into the DUT and reads streaming output. The output is then compared to an expected golden output.

```
31 top_top DUT (  
32     .clk(clk),  
33     .reset(reset),  
34     .start(1'b1),  
35     .ready(),  
36     .finish(),  
37     // Return value.  
38     .output_fifo_ready( ready_TO_DUT_DS ),  
39     .output_fifo_valid( valid_FROM_DUT_DS ),  
40     .output_fifo( data_FROM_DUT_DS ),  
41     // Argument.  
42     .input_fifo_ready( in_ready_FROM_DUT_US ),  
43     .input_fifo_valid( in_valid_TO_DUT_US ),  
44     .input_fifo( in_data_TO_DUT_US )  
45 );
```

On line 50, we generate the valid and ready signals to be sent to the DUT. On line 65 we generate the data inputs and expected outputs from the DUT. For this testbench, we use the same inputs and have the same expected outputs as the ones from the software testbench.

```
50 // Input valid signals (upstream) and ready (downstream).  
51 always @ (posedge clk) begin  
52     if (reset) begin  
53         in_valid_TO_DUT_US <= 0;  
54         ready_TO_DUT_DS <= 0;  
55     end else begin  
56         if (in_data_TO_DUT_US == NUM_EXPECTED_INPUT + 1)  
57             in_valid_TO_DUT_US <= 0;  
58         else  
59             in_valid_TO_DUT_US <= 1;  
60  
61         ready_TO_DUT_DS <= 1;  
62     end  
63 end  
64  
65 // Input data, and expected output.  
66 always @ (posedge clk) begin  
67     if (reset) begin  
68         in_data_TO_DUT_US <= 0;  
69         expected_output <= 4950;  
70     end else begin  
71         if (in_valid_TO_DUT_US & in_ready_FROM_DUT_US)
```

```

72         in_data_TO_DUT_US <= in_data_TO_DUT_US + 1;
73     if (valid_FROM_DUT_DS & ready_TO_DUT_DS) begin
74         expected_output <= expected_output + 10000;
75     end
76 end
77 end

```

On line 79 we create a monitor to automatically check that the DUT outputs the expected number of results, with the expected values. The monitor finishes the test when the expected number of results are received, or when we have waited too long for the output which may indicate the design is stuck somewhere in simulation. On line 95, the monitor block will print "PASS" if the expected number of outputs with the expected values are read.

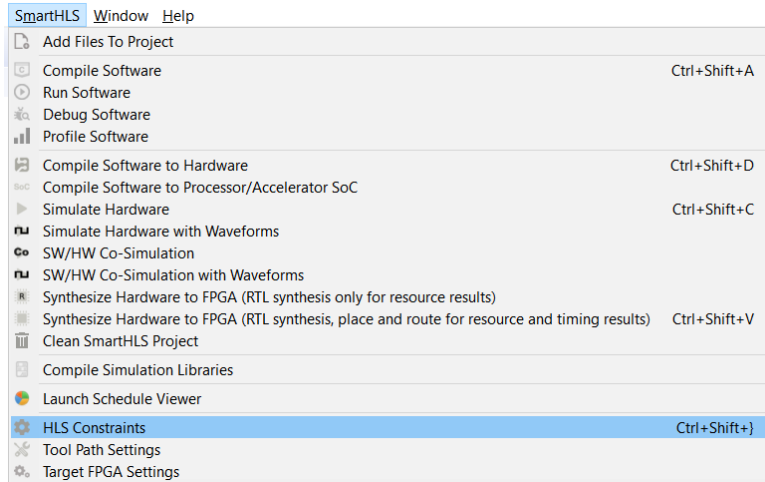
```

79 // Monitor to check number and correctness of outputs.
80 always @ (posedge clk) begin
81     clk_cntr <= reset ? 0 : clk_cntr + 1;
82     if (valid_FROM_DUT_DS & ready_TO_DUT_DS) begin
83         output_cntr <= output_cntr + 1;
84         if (expected_output == data_FROM_DUT_DS) begin
85             match_cntr <= match_cntr + 1;
86             // $display("At cycle %d: output matches expected value, %d ==
%d",
87                 //      clk_cntr, expected_output, data_FROM_DUT_DS);
88         end else begin
89             $display("At cycle %d: output != expected value, %d != %d",
90                 clk_cntr, data_FROM_DUT_DS, expected_output);
91         end
92     end
93     if (output_cntr == NUM_EXPECTED_OUTPUT) begin
94         if (match_cntr == output_cntr)
95             $display("At cycle %d: PASS", clk_cntr);
96         else
97             $display("FAIL: %d matches out of %d", match_cntr,
output_cntr);
98         $finish;
99     end
100 end

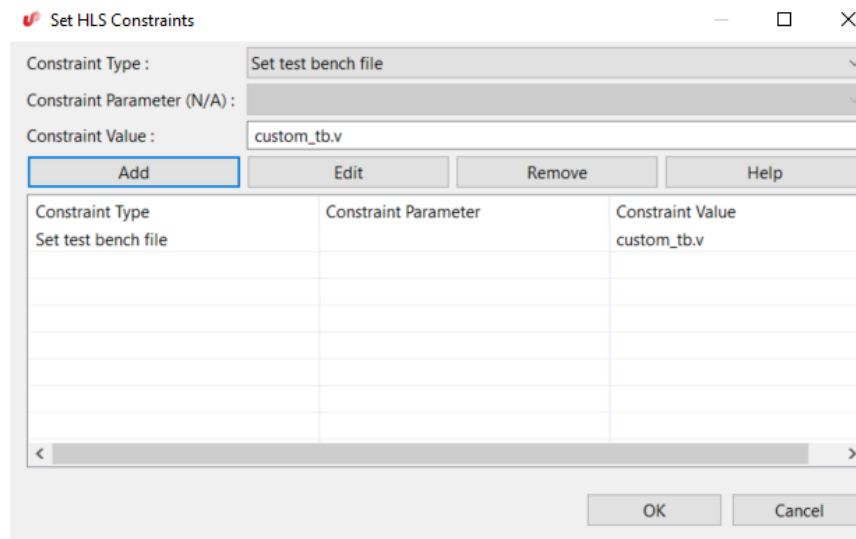
```



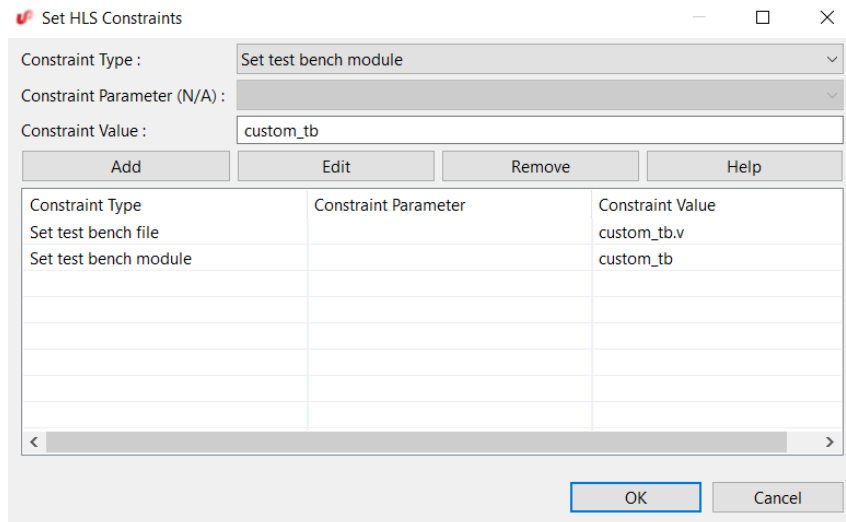
Next, we want to make sure our custom testbench is recognized by SmartHLS by adding it to our constraints. Navigate to the SmartHLS menu and select HLS Constraints.



Once the constraints editor is brought up, select “Set test bench file” as the constraint type and add `custom_tb.v` as the value and then click Add. This tells SmartHLS to use our `custom_tb.v` as the custom testbench file.



Next, select “Set test bench module” as the constraint type and add `custom_tb` as the value and click Add. This tells SmartHLS that the custom testbench module found in `custom_tb.v` is called `custom_tb`. Then click OK to close.



Next run this custom testbench by clicking (▶). Check that you see “PASS” in the simulation output which indicates that the design passed the test.

```
# run 1000000000000000ns
# At cycle      2083: PASS
# ** Note: $finish    : ../custom_tb.v(98)
#   Time: 20855 ns  Iteration: 1  Instance: /custom_tb
# End time: 00:34:29 on Jul 29,2021, Elapsed time: 0:00:16
# Errors: 0, Warnings: 0
```



Now close the producer_consumer project and opened files.

Threading vs. Function Pipelining

Threading and function pipelining can both be used to implement parallel modules, but function pipelining is typically used to implement pipelined dataflow designs. Even when implementing dataflow designs, pipelined functions have restrictions on what types of arguments can be passed into the parallel functions while threaded functions do not. This makes function pipelining good for implementing pipelines where only FIFO interfaces are used between the parallel blocks, while threading is good for implementing anything else. A few examples of good thread use cases are dataflow designs that do not use FIFOs to connect blocks and parallel blocks that perform unrelated tasks.

SmartHLS Digit Recognition Application on PolarFire® Video Kit

In this section, we will describe how the SmartHLS multi-threading basics we just looked at can be used to implement a digit recognition application running on the PolarFire Video Kit. We chose the digit recognition application for this training because video allows you to see and visualize the HLS design running on the FPGA, and because this application was suited to being implemented with SmartHLS multi-threading. Note, SmartHLS is not limited to video processing applications. SmartHLS can be used to implement other application areas such as security, motor control, and others. SmartHLS is also not limited to targeting the PolarFire Video Kit, SmartHLS generates an IP core that can be integrated into SmartDesign targeting other PolarFire boards.

There are three ways to implement neural networks on a Microchip FPGA: using the VectorBlox IP core, writing RTL by hand, or writing C++ with SmartHLS. There are pros and cons to each method. [VectorBlox](#) offers a flexible architecture and can easily handle different types of neural networks with only hex file changes and without running place & route. Neural networks with fixed architectures can be implemented in RTL or with SmartHLS in scenarios where FPGA area is limited. For this digit recognition application, we implemented a fixed neural network architecture using SmartHLS.

Goals of this section:

1. Understand how to use SmartHLS multi-threading to implement a complex parallel hardware design.
2. Learn how to port an existing RTL design into HLS using the same hardware architecture and interface requirements.
3. Learn ways of optimizing an HLS software implementation of a complicated design to meet design requirements (limited DSP blocks).
4. Show benefits of HLS design compared to an equivalent RTL design.

Design Overview

The PolarFire Video Kit Digit Recognition design builds off the PolarFire Video Kit demo. This demo has two camera inputs that get combined in a picture-in-picture format. We replaced the image processing portion of the design with our digit recognition design. The digit recognition design was an RTL to HLS port for the Hello FPGA digit recognition project.

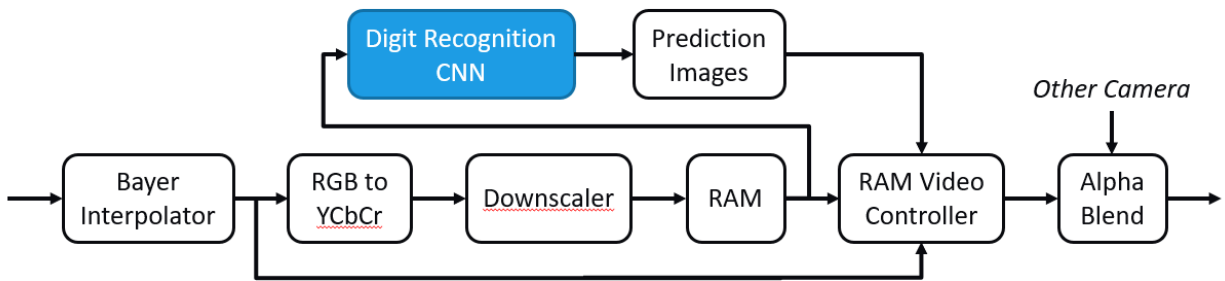


Figure 10: Digit Recognition System Diagram

Figure 10 shows the high-level hardware blocks used for the digit recognition application on the PolarFire® board. The *Digit Recognition CNN* block is highlighted in blue and is implemented using SmartHLS. We used SmartHLS to design this hardware block in C++ and generate a SmartDesign IP component. The rest of the blocks are implemented in RTL.

On the left of Figure 10, the camera input is converted from Bayer format into a 24-bit RGB video stream by the *Bayer Interpolator* block. The *RGB to YCbCr* block converts the 24-bit RGB input stream into 8-bit greyscale. The *Downscaler* block downscales the 448x448 picture-in-picture video input to a 28x28 output video stream. The *RAM* block stores the downsampled video stream as a 28x28 video frame. The *Digit Recognition CNN* block predicts the digit (0 to 9) from the 28x28 video frame stored in the *RAM* block. The *Prediction Images* block stores 80x80 images for each of the digits 0 to 9 and the corresponding digit image can be read depending on the CNN prediction. The *RAM Video Controller* block stitches the original PIP video stream together with the predicted digit image and downsampled 28x28 *RAM* image at fixed coordinates and then sends the resulting video stream to the *Alpha Blend* block. The *Alpha Blend* block then stitches the *RAM Video Controller* video output with the 1920x1080 camera video stream. The 1920x1080 video stream output of *Alpha Blend* goes to the Monitor display.

Digit Recognition CNN Implementation

In this section, we will go over the design process and implementation of the SmartHLS generated Digit Recognition CNN block. We will follow an approach that is suitable for a port of an existing hardware block to SmartHLS, as opposed to SmartHLS Training 1 where we used an approach suitable for porting a software implementation to SmartHLS. Many steps are similar.

We will show that by porting the digit recognition RTL implementation to HLS we can make the design easier to read, understand, and modify since C++ is at a higher abstraction level than RTL. Future design iteration, maintenance, and collaboration are easier with HLS designs since more engineers understand C++ than understand RTL.

Neural Network Basics

First, we will go over some neural network concepts and terminology to better understand the implementation details of the digit recognition CNN.

A neural network is a type of computation which takes a set of inputs and then generates a set of outputs that are predicted based on previous training data. During training, we use many input and expected output examples to train the neural network to give accurate predictions. After training, the neural network architecture and weights are fixed and we can now perform inference, where we provide new inputs and get predicted outputs from the neural network. The quality and size of the training data determines the accuracy of a neural network.

Depending on the application, neural networks can have different architectures. A convolutional neural network (CNN) is a neural network architecture that has convolution layers and is suited for image processing. Inside a CNN there are three kinds of layers that are commonly used: convolution layers, pooling layers, and fully connected layers.

Each neural network layer works with tensors of various sizes as inputs and outputs. A tensor is a generalization of scalars, vectors, and matrices, which are degree 0, degree 1, and degree 2 tensors, respectively. As shown in Figure 11, the size of a degree 2 tensor (matrix) is defined by the number of rows and columns. A degree 3 tensor is defined by the number of rows, columns, and the depth.

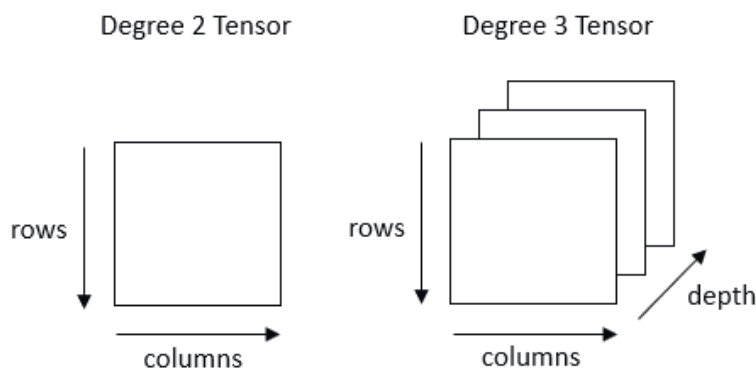


Figure 11: Examples of degree 2 and degree 3 tensors

Convolution layers are good for extracting geometric features from an input tensor. Convolution layers work in the same way as an image processing filter (such as the Sobel filter) where a square filter (called a **kernel**) is slid across an input image. The **size** of filter is equal to the side length of the square filter, and the size of the step when sliding the filter is called the **stride**. The values of the input tensor under the kernel (called the **window**) and the values of the kernel are multiplied and summed at each step, which is also called a convolution. Figure 12 shows an example of a convolution layer processing an input tensor with a depth of 1.

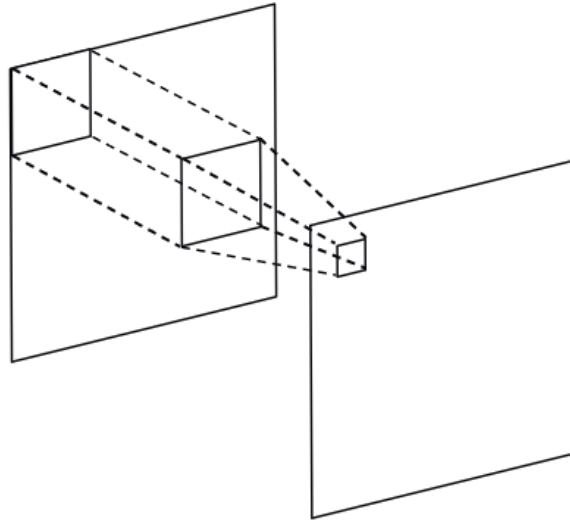


Figure 12: Structure of a convolution layer using a kernel to process the first window of an input tensor with a depth of 1

The main difference between a convolution layer and an image processing filter is that in a convolution layer, the input may be a tensor that can be treated as several stacked image channels which are equivalent to the submatrices of the input tensor. The number of channels or submatrices is defined by the depth parameter of the tensor. There might also be multiple kernels applied to each matrix in the input tensor. This results in a tensor input and a tensor output for each convolution layer. Convolution layers also usually have an activation function that introduces non-linearity to the function approximated by the neural net. A common activation used in convolution layers is called the rectified linear unit (ReLU), which is a threshold function that sets all negative values to 0 as shown by Figure 13.

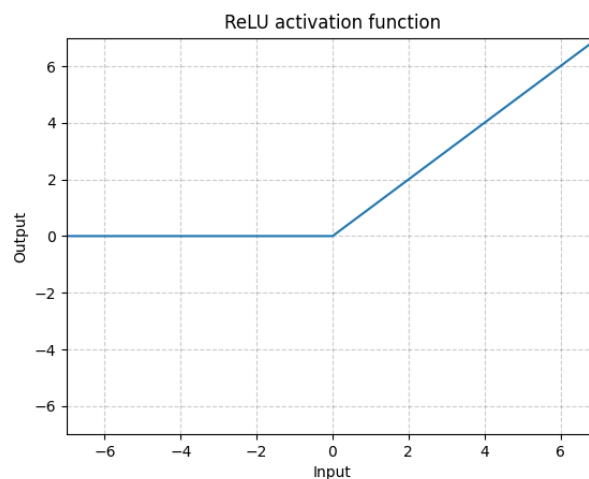


Figure 13: ReLu activation function

Pooling layers are a way to down sample the tensor outputs of convolution layers. Pooling layers work by applying a pooling function as the kernel to each window of each submatrix of an input tensor. The size of the pooling function is determined by the side length of the window and the step size when sliding the window is called the stride. For example, the maxpool in Figure 14 has kernel size 2 and stride 2. A commonly used pooling function is the maxpool, which outputs the maximum value within a window of values. The stride is commonly set to be the same as the size, ensuring that no two windows overlap. Pooling layers down samples the input and condenses the information for following layers.

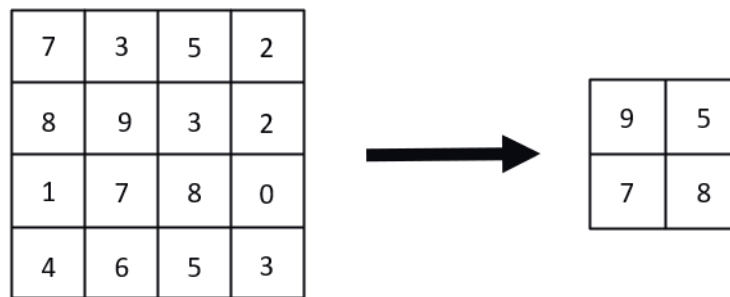


Figure 14: Maxpool layer with kernel size 2 and stride 2

Fully connected layers are a type of neural network layer where every input is connected with every output, with a set of weights and biases associated with them, as shown in Figure 15. This type of layer is good for creating associations with input patterns and output patterns. Each output is the result of all inputs multiplied with their associated weights and added with a bias.

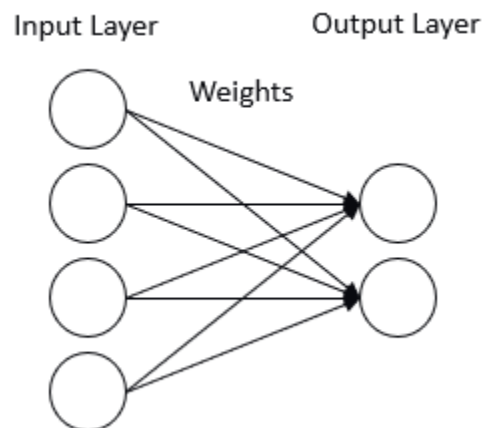


Figure 15: Structure of a fully connected layer with four inputs and two outputs

For two out of the three layers (convolution and fully connected) the main operation performed is multiply accumulate (MACC). We will need to keep this in mind when implementing the CNN in hardware as to manage MACC usage. The maxpool layer does not use any MACCs.

Digit Recognition CNN Architecture

The Digit Recognition CNN hardware block was based on the demo for the SmartFusion® 2 Hello FPGA kit ([M2S-HELLO-FPGA-KIT](#)). The CNN core is described in Section 4.4 of the [UG0891: Hello FPGA Libero Design User Guide](#).

Figure 16 shows the CNN architecture, which contains 4 convolution layers (orange), 1 max pool layer (blue), 1 fully connected layer (green), and a final maximum layer (white). The CNN was trained from the standard [MNIST](#) handwritten digit database which contains the digits in 28x28 resolution images. The CNN can detect a single digit in a 28x28 input image when the aspect ratio of the digit is approximately equal to the aspect ratio of the trained image data set. The CNN can only recognize 10 classes for digits from 0 to 9, there is no class to recognize when a digit was not detected.

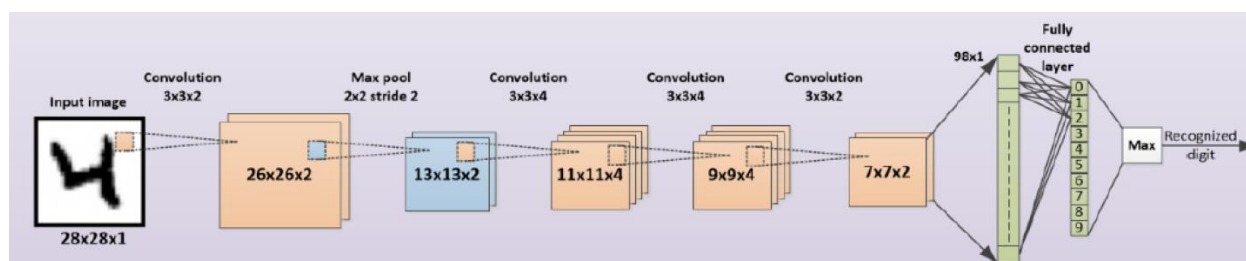


Figure 16: Digit Recognition CNN architecture

The CNN classifier input is a 28x28 image in 8-bit grayscale (1 channel) and the output is a 4-bit result indicating the prediction of recognized digit (from 0-9).

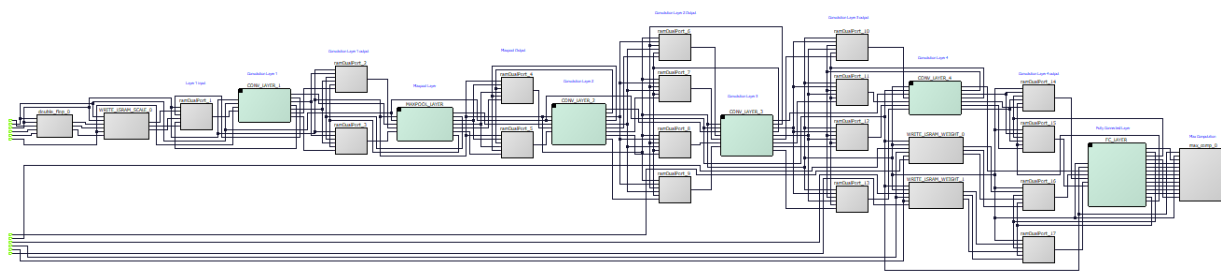


Figure 17: SmartDesign canvas with the original RTL implementation of the CNN

Our goal is to recreate a drop-in replacement for the CNN shown in Figure 17 using SmartHLS, while also having the same interfaces and architecture. Since we already have a set of requirements from the RTL implementation, we can use the same requirements to constrain our HLS implementation. We have then ported this CNN created in SmartHLS to the PolarFire Video Kit for demo purposes.

Requirements for Digit Recognition CNN

As shown in Figure 17, the original Hello FPGA design used a RAM as an input to the CNN model with RAMs to hold intermediate values between each layer. The output of the CNN is a 4-bit registered output representing the predicted digit which connects to the rest of the system. We will also use the same hardware interface in our HLS implementation.

The sizes of the inputs, parameters for the weights, and structure of the 6 layers of the CNN are given in Table 1.

Table 1: Digit Recognition CNN Model Parameters

Layers	Input Size: Width x Height x Channels	Filter Size	#. Kernels	Stride	MACCs
Convolution 1	28 x 28 x 1	3	2	1	2
Maxpool	26 x 26 x 2	2	n/a	2	0
Convolution 2	13 x 13 x 2	3	4	1	8
Convolution 3	11 x 11 x 4	3	4	1	4
Convolution 4	9 x 9 x 4	3	2	1	4
Fully Connected	7 x 7 x 2	98 x 10		n/a	2

All weight parameters, inputs, and intermediate values in the original design were represented as 16-bit integers. For the accumulation of the multiply products between inputs and weights, an integer wider than 16 bits is used to represent the accumulated value to prevent overflow. The same weight parameters and structure will be used in our design. Due to the limited number of MACC blocks on the target FPGA device, each layer's compute unit is allocated with a small number of MACCs. The number of MACCs (multipliers) allocated for each layer is presented in Table 1. We will implement the layers to restrict the number of MACCs to be the same to match the original RTL usage.

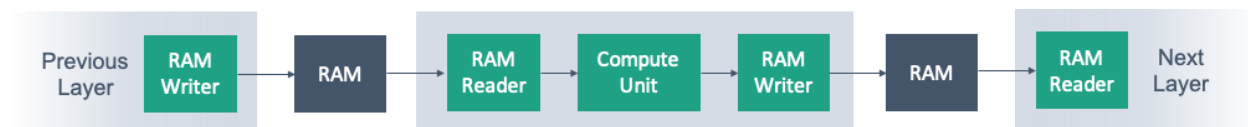


Figure 18: Structure of each layer in hardware

We also notice that all layers in the RTL reference design exhibit a similar hardware structure as illustrated in Figure 18. Each layer is connected to an input RAM and an output RAM and has three sub-modules for 1) reading inputs from an input RAM, 2) performing the convolution, maxpool, or fully-connected (FC) computation, and 3) writing the computed results to an output RAM which will be read by the next layer as input. All layers are connected in sequence and run in always-running, dataflow parallel manner. We can replicate this behavior using SmartHLS by creating a dataflow pipeline using threads.

In the CNN classifier task pipeline, multiple layers can be active at the same time while processing different input frames. A layer can start computing on a new frame as soon as two conditions are met: 1) the inputs of the new frame are available in the input RAM, and 2) the subsequent layer has completed its computation for the previous frame such that the output RAM (i.e., the next layer's input RAM) can be overwritten. Figure 19 below illustrates the task pipeline execution.

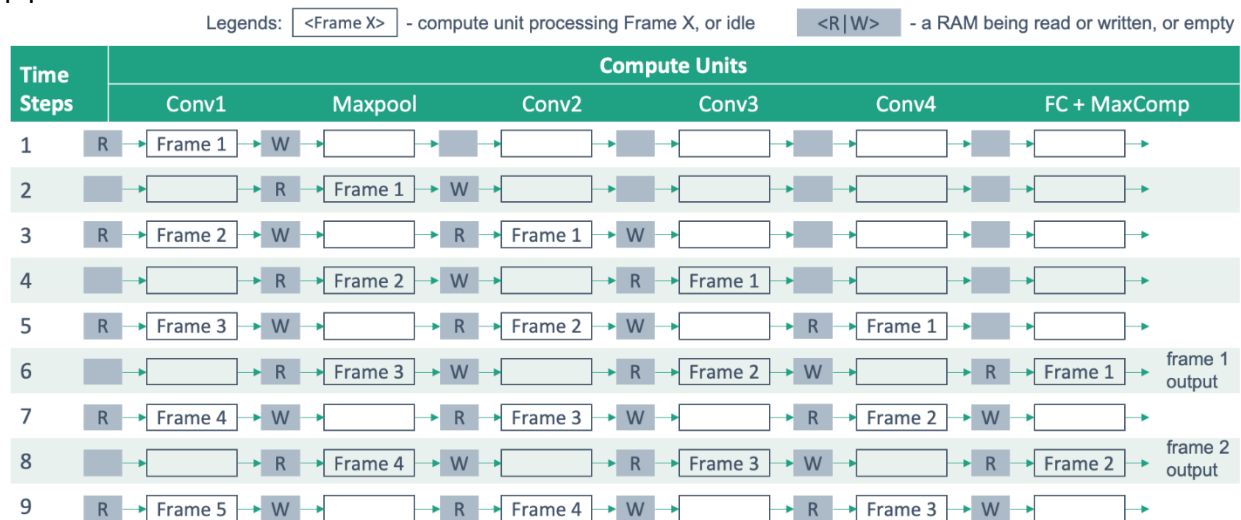


Figure 19: Illustration of Task Pipeline Execution.

- At time step 1, the first layer - Conv1 is processing Frame 1, reading from its input RAM and writing to its output RAM.
- At time step 2, when Conv1 finishes Frame 1, the second layer - Maxpool becomes active; at this time, Conv1 should stay idle because its output RAM is being read by Maxpool and is not yet ready to be overwritten.
- At time step 3, Conv2 picks up the output from Maxpool and starts the processing for Frame 1; meanwhile Conv1 can start processing a new Frame 2 since its output RAM is now “free” to be updated after Maxpool finishes.

The same pattern continues for the rest of the time steps. At time step 6, the task pipeline enters the steady state where three active layers can process three different frames simultaneously.

Although this is not handled in the original RTL, we also need to consider that the interfaces between the pipeline stages are RAMs, and that reading and writing to the same RAM must be controlled such that a later stage will not read invalid data while the previous layer is writing to the RAM. This can be easily achieved by the same “done” handshaking strategy we used in the simple project seen in section 0. This prevents the layers from reading buffers that are partially filled with new data, as new data does not come in as a continuous stream. In practice, this has very little effect on the prediction as the frame rate is fairly high and the video input should remain fairly stable, but we would still like to design correct hardware where possible.

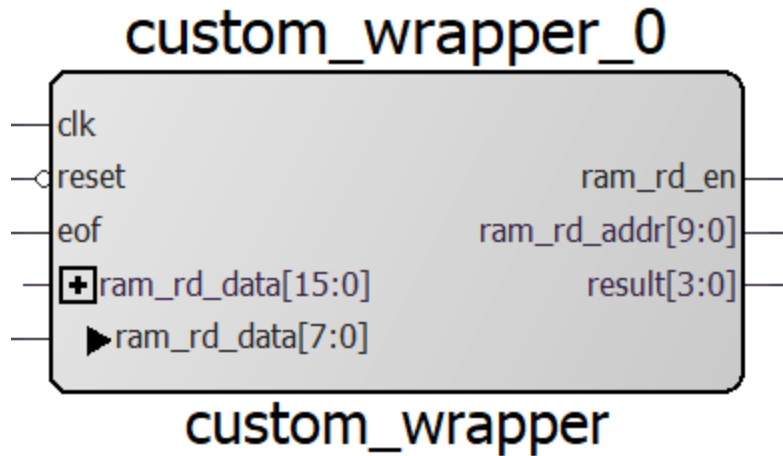


Figure 20: Target interface for digit recognition block

Table 2: Target Digit Recognition Interface

Signal Name	Direction	Width	Description
clk	Input	1-bit	System Clock
reset	Input	1-bit	Active low synchronous reset
eof	Input	1-bit	End of frame/frame valid
ram_rd_data	Input	16-bits	8-bit greyscale RAM data, top 8 bits unused and tied to 0
ram_rd_en	Output	1-bits	RAM read enable
ram_rd_addr	Output	10-bits	RAM read address
result	Output	4-bit	Registered prediction output

With the above requirements set, we can begin to think about how best to implement this design in SmartHLS. We will target the hardware top-level interface presented in Figure 20 and Table 2. We can look at each restriction in turn and decide on the best method to fulfil each requirement.

The first step in porting the design to a SmartHLS design is to create a functionally correct implementation of the digit recognition design in software while keeping in mind the hardware requirements. Once we have a working software implementation, we can then adjust it to meet the more difficult performance and resource requirements.

Digit Recognition Software Implementation



To see the software implementation, open the `digit_recognition_sw` project. We first start by defining some useful types we will use through-out the design. Open the `defines.h` source file.

```

v digit_recognition_sw
  > Includes
  > defines.h
  > digit_recognition.cpp
  > digit_recognition.h
  > test_images.h
  > test_main.cpp
  > weights.h
  config.tcl
  makefile
    
```

In the `defines.h` file, we have definitions for `DType` and `Tensor` types on lines 7 and 11:

```

6 // Data type used throughout the CNN.
7 using DType = hls::ap_int<16>;
8
9 // Tensor type representing the inputs and outputs of layers.
10 template <unsigned SIZE, unsigned DEPTH>
11 using Tensor = DType [SIZE][SIZE][DEPTH];
    
```

`DType` represents the 16-bit integers we will be using as data values throughout the design. The `Tensor` type represents the tensor inputs and outputs to each layer. We can use the template parameters of the `Tensor` type to easily specify the dimensions of the different 3rd-order tensors used in each CNN layer as shown in Figure 21. For this CNN, the size of the first two dimensions is always the same as each other, while the depth can be different.

Note, we use SmartHLS's `ap_int` arbitrary precision data type to represent the exact bitwidth we would want to use in the initial software implementation instead of using standard `int` data types to better approximate the functionality of the hardware during software simulation.

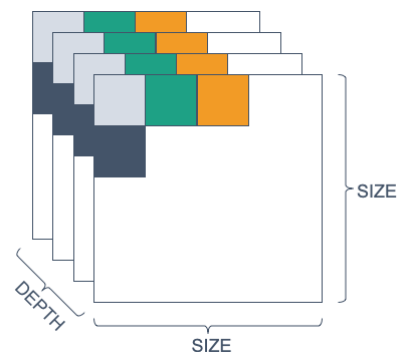


Figure 21: 3-Dimensional Tensor Representing Input and Intermediate Results of the CNN



Now open the `digit_recognition.h` source file.

This file holds the software implementation for each of the three types of layers we presented in section 0. Scroll to line 54 to see the definition of the convolution layer `Conv`. In the basic software implementation, this layer takes an input tensor, output tensor, weights, and biases as arguments. These arrays conveniently turn into the RAMs and RAM interfaces that we want the resulting RTL to have, fulfilling some of the interface requirements. On line 52, we have template arguments which are used to parameterize the size of the input and output tensors, the weights, and the biases.

```

52 template <unsigned INPUT_SIZE, unsigned INPUT_DEPTH, unsigned OUTPUT_SIZE,
53           unsigned OUTPUT_DEPTH, unsigned FILTER_SIZE>
54 void
55 Conv(Tensor<INPUT_SIZE, INPUT_DEPTH> input,
56      Tensor<OUTPUT_SIZE, OUTPUT_DEPTH> output,
57      const short weights[OUTPUT_DEPTH][INPUT_DEPTH][FILTER_SIZE *
58                                     FILTER_SIZE],
59      const short bias[OUTPUT_DEPTH]) {

```

We used template arguments for the `Conv` function so that this parameterized convolution layer can be used to implement all four convolution layers used in the CNN without changing the implementation. All C++ function template arguments must be known at compile time, allowing SmartHLS to generate efficient hardware for each parameterized function. C++ template arguments are like RTL module parameters but more powerful.

The body of the convolution layer performs the window sliding and multiply accumulate filtering steps for each of the channels of the input tensor. Likewise, the maxpool layer on line 95 and the fully connected layer on line 124 are also parameterized and perform their expected computations in software.



Next open the `weights.h` source files.

This file holds all the kernels, weights and biases for the convolution and fully connected layers. We made sure the sizes and values of each of these are the same as the ones in the RTL implementation presented in Table 1.



Next open `digit_recognition.cpp`.

On line 21 in this file, we have the `ClassifierPipeline` top-level function which runs all the CNN layers in sequence and provides the sizes, kernels, weights, and biases for layers. This function will also implement the interface for the top-level CNN hardware module.

```
20 // Top level function which calls all layers in sequence.
21 void ClassifierPipeline() {
22 #pragma function top
```

The `digit_recognition.cpp` source file also holds the implementation for the `MaxComp` function on line 51, which chooses the highest confidence output from the FC layer and reports the prediction result.

```
49 // MaxComp takes the output of the fully-connected layer and finds the
digit
50 // with the maximum confidence and reports it.
51 void MaxComp(DType input[10], FIFO<ap_uint<4>> &output_digit) {
```



Now open `test_main.cpp` to see the software testbench.

The main testbench function reads 10 digits from 0 – 9 taken from the Hello FPGA design GUI and calls the `ClassifierPipeline` to make sure that the CNN algorithm performs as expected.

```
7 // Run 10 iterations, each with a 28x28 input image of digit from 0 to 9.
8 for (unsigned n = 0; n < 10; n++) {
9     for (unsigned i = 0; i < 28; i++)
10         for (unsigned j = 0; j < 28; j++)
11             classifier_input[i][j][0] = test_images[n][i][j][0];
12 // Run the classifier, which will write the prediction to
13 // classifier_output FIFO.
14 ClassifierPipeline();
15 }
```



Click the Compile Software (🔧) and Run Software (▶) buttons to test the functionality of the software implementation. You should see the following output in the console stating that all the input digits from 0-9 were predicted correctly.

```
Highest confidence: 3267, digit-0
Highest confidence: 2686, digit-1
Highest confidence: 3284, digit-2
Highest confidence: 2690, digit-3
Highest confidence: 1935, digit-4
Highest confidence: 1370, digit-5
Highest confidence: 2913, digit-6
Highest confidence: 3555, digit-7
Highest confidence: 1879, digit-8
Highest confidence: 2093, digit-9
Received prediction: 0
Received prediction: 1
Received prediction: 2
Received prediction: 3
Received prediction: 4
Received prediction: 5
Received prediction: 6
Received prediction: 7
Received prediction: 8
Received prediction: 9
```

We have already met some of our hardware requirements in the software implementation of digit recognition, namely, we are using input interfaces, structure, and weight parameters as specified in our requirements. Next, we will modify the software version of the digit recognition CNN to meet our remaining hardware requirements.



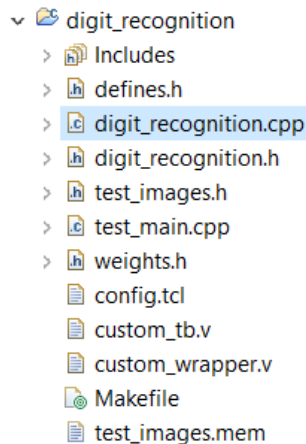
Close the digit_recognition_sw project and the opened files.

Adjusting the Software Implementation

In the software implementation, we are able to meet some of the hardware requirements we discussed before. The hardware requirements still missing are the output interface, running the modules in parallel, restricting the number of MACCs each layer uses, and synchronization. In the current implementation, the number of MACCs used in each layer is 1 as there is no loop unrolling. We need to increase the number of multipliers (DSPs) used to match the RTL implementation.



To see how we can meet these requirements, open the digit recognition project.



This project is the final implementation that we have turned into hardware and incorporated into the Hello FPGA and PolarFire® Video Kit designs.



Open the digit_recognition.h source file.

In this version, we have modified the CNN layers to use the number of MACCs allocated to them in the original RTL design (2, 8, 4, 4, and 2) shown previously in Table 1. In Figure 22, we provide the implementation from lines 69-105 of digit_recognition.h, simplified for readability. We assigned the labels from 1 to 8 to each of the loops. In Figure 23, we show the input tensor values and convolution filters involved in the computation of the set of colored output tensor values (see Loop 3 arrow).

Loop 1 and Loop 2 the code traverses along the row and column dimensions of the output tensor. Loop 3 traverses along the depth dimension of the output tensor, each iteration computes a PARALLEL_KERNELS number of outputs. The `accumulated_value` array will hold the partial dot-products. Loop 4 traverses along the row and column dimensions of the input tensor and convolution filter kernels. Then Loop 5 walks through each of the PARALLEL_KERNELS number of selected convolution filters and Loop 6 traverses along the depth dimension of the input tensor. Loop 7 and Loop 8 add up the partial sums together with biases to produce PARALLEL_KERNEL number of outputs.

```

const static unsigned PARALLEL_KERNELS = NUM_MACC / INPUT_DEPTH;
for (unsigned r = 0; r < OUTPUT_SIZE; r++) { // Loop 1.
  for (unsigned c = 0; c < OUTPUT_SIZE; c++) { // Loop 2.
    for (unsigned od = 0; od < OUTPUT_DEPTH; od += PARALLEL_KERNELS) { // Loop 3.
      ap_int<36> accumulated_value[PARALLEL_KERNELS][INPUT_DEPTH] = {0};

#pragma HLS loop pipeline
      for (unsigned m = 0, n = 0, i = 0; i < FILTER_SIZE * FILTER_SIZE; i++) { // Loop 4.
        for (unsigned k = 0; k < PARALLEL_KERNELS; k++) { // Loop 5.
          for (unsigned id = 0; id < INPUT_DEPTH; id++) { // Loop 6.
            accumulated_value[k][id] += weights[od + k][id][i] * input[r + m][c + n][id];
          }
        }
        n = (n == FILTER_SIZE - 1) ? 0 : n + 1;
        m = (n > 0) ? m : m + 1;
      }

      for (unsigned k = 0; k < PARALLEL_KERNELS; k++) { // Loop 7.
        DType sum = bias[od + k];
        for (unsigned id = 0; id < INPUT_DEPTH; id++) // Loop 8.
          sum += accumulated_value[k][id](29, 14);
        output[r][c][od + k] = ReLU(sum);
      }
    }
  }
}

```

Figure 22: Convolution implementation simplified for readability. See diagram in Figure 23.

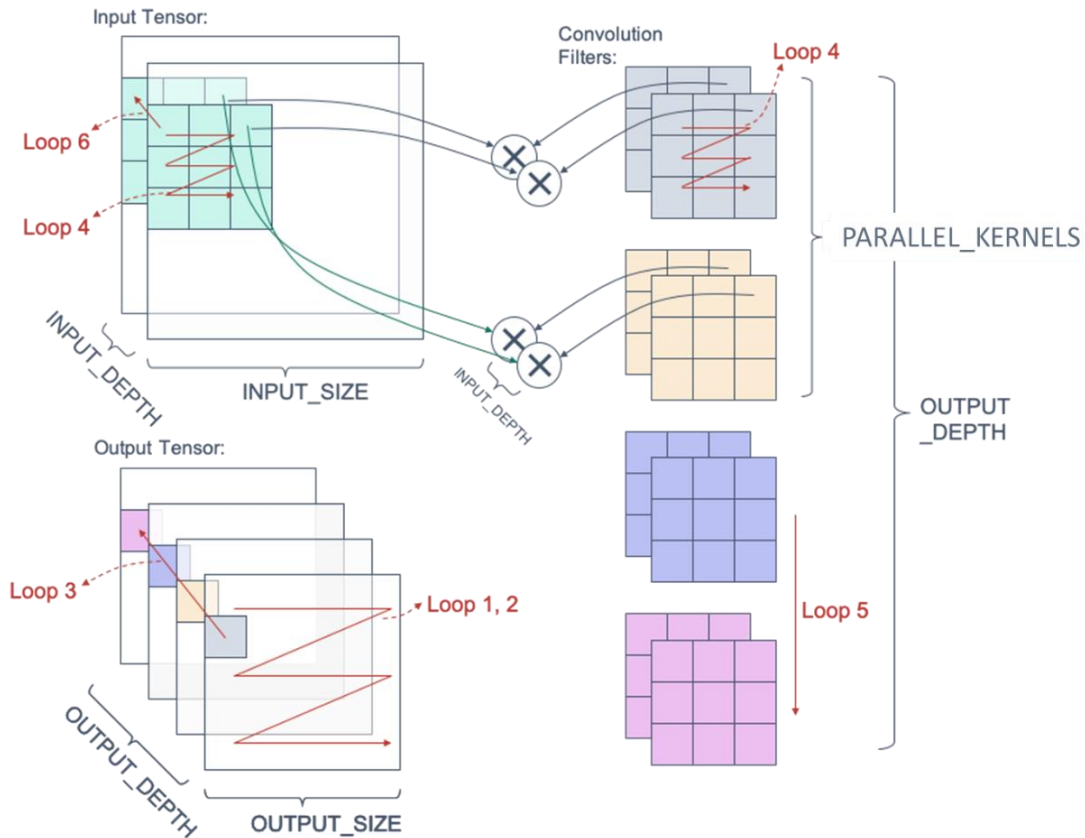


Figure 23: Convolution operation with parallel filters (loops labeled in Figure 22).

Now that we understand what the software is doing, we can focus on how to optimize the number of multipliers used in the hardware. Back in `digit_recognition.h` on line 87, we have applied the `SmarchLS loop pipeline` pragma to the inner loop that does the multiply accumulate operations. Loop pipelining will get extra performance out of the loop that does most of the processing. Loop pipelining also unrolls all loops and functions nested inside the loop body which creates hardware to run each unrolled iteration in parallel. We have also modified some loop bounds and added a `k` loop that runs `PARALLEL_KERNELS` times. By adding this extra loop, we can force `SmarchLS` to duplicate the inner-most loop body `PARALLEL_KERNELS * INPUT_DEPTH` times, effectively running `PARALLEL_KERNELS * INPUT_DEPTH` iterations in parallel. This in turn determines how many MACCs we will use, since each inner-most iteration uses a single MACC.

```

83 // Pipelining improves performance and unrolls inner loops, causing
84 // PARALLEL_KERNELS * INPUT_DEPTH iterations to run in parallel.
85 // Parallel iterations use one MACC each, resulting in NUM_MACC =
86 // PARALLEL_KERNELS * INPUT_DEPTH MACCs to be used.
87 #pragma HLS loop pipeline
88 // Perform the convolution between input and weights in a
89 // window.
90 for (unsigned m = 0, n = 0, i = 0;
91      i < FILTER_SIZE * FILTER_SIZE; i++) {
92     for (unsigned k = 0; k < PARALLEL_KERNELS; k++) {
93         for (unsigned id = 0; id < INPUT_DEPTH; id++) {
94             accumulated_value[k][id] +=
95                 weights[od + k][id][i] *
96                 input[r + m][c + n][id];
97         }
98     }
99     n = (n == FILTER_SIZE - 1) ? 0 : n + 1;
100     m = (n > 0) ? m : m + 1;
101 }

```

We have also adjusted the outer loop on line 74 to increment by `PARALLEL_KERNELS` instead of by one to account for this extra `k` loop.

```

74     for (unsigned od = 0; od < OUTPUT_DEPTH;
75          od += PARALLEL_KERNELS) {

```

The `PARALLEL_KERNELS` parameter is determined by the calculation on line 62, which divides the number of MACCs we want the current layer to use by the number of depth dimensions of the input tensor. The number of MACCs we want to use is provided as a template parameter `NUM_MACC` in the function signature on line 53. This code assumes that `NUM_MACC` is always divisible by the depth dimension of the input.

```

50 // Conv implements a generic convolution layer.
51 // Note that NUM_MACC has to be a multiple of INPUT_DEPTH,
52 template <unsigned INPUT_SIZE, unsigned INPUT_DEPTH, unsigned OUTPUT_SIZE,
53           unsigned OUTPUT_DEPTH, unsigned FILTER_SIZE, unsigned NUM_MACC>

60 // Based on the number of MACCs we want, calculate how many
61 // loop iterations we want to run in parallel.
62 const static unsigned PARALLEL_KERNELS = NUM_MACC / INPUT_DEPTH;

```

One hardware requirement was that all the layers should always stay running in parallel. We have made each of the layer functions into a threaded function that runs forever. To do this, we wrap the function body in a while loop with a true condition to make sure that this function can run in parallel with the other layers when implemented in hardware continuously. An example of this is on line 64 for the Conv layer.

```
64     while (1) {
```

We also must meet the synchronization requirements for the layers running in parallel as shown in Figure 19. We have added input valid and output valid signals to synchronize the RAMs feeding in and out of the layers. This prevents one layer from writing output to a RAM while the next layer reads from it, potentially resulting in the next layer reading invalid data. The input and output signals are declared as volatile references to be updated correctly when another thread changes the value and when the current thread wants to change the value. These handshaking signals are added between every layer. An example of this is on line 54 and 55 for the Conv layer.

```
54 void Conv(Tensor<INPUT_SIZE, INPUT_DEPTH> input, volatile bool &input_valid,
55           Tensor<OUTPUT_SIZE, OUTPUT_DEPTH> output, volatile bool &output_valid,
```

Lastly, we add input and output valid handling logic inside of each layer to finish implementing the synchronization. This is the same kind of handshaking we saw in Section 0. An example of this synchronization is shown on lines 66-99 and 114-115 for the Conv layer.

```
65 // Input valid handshaking
66 bool should_run = input_valid;
67 should_run &= !output_valid;
68 if (!should_run)
69     continue;

113 // More handshaking
114 input_valid = false;
115 output_valid = true;
```

The start conditions are captured by the `should_run` variable. The `should_run` variable is true when both the input tensor is valid (`input_valid` is true) and the output tensor is ready to be overwritten (`output_valid` is false), at which point the start conditions are met. When the computation for the current frame has finished, the `input_valid` flag is set to false to inform the previous layer that the input tensor RAM is now “free” to be overwritten, and the `output_valid` flag is set to true to inform the next layer that the output tensor RAM holds valid data for a new frame.



Next open the `digit_recognition.cpp` source file.

Once again, this file contains the `ClassifierPipeline` function. Scroll to line 47 to see this function.

```
44 // Top level function which calls all layers in sequence.
45 // Each layer runs in a thread, meaning all layers will run in parallel in both
46 // software and hardware.
47 void ClassifierPipeline() {
48 #pragma HLS function top
```

You will notice that just like before, this C++ function has no function arguments. SmartHLS will infer the RTL interface from the global variables used inside the top-level C++ function. This will automatically generate the RAM interface based on the variables used in the top-level function.

In the body of the `ClassifierPipeline` function you will find the function calls have been changed to thread API calls. On line 50, we have the declaration of the first thread `t0`. This is the exact same `Conv` layer function call as on line 24 of the software version, we just replaced the call with a thread declaration that we have seen in Section 0.

```
50 hls::thread<void> t0(
51     Conv</* INPUT_SIZE */ 28, /* INPUT_DEPTH */ 1, /* OUTPUT_SIZE */ 26,
52         /* OUTPUT_DEPTH */ 2, /* FILTER_SIZE */ 3, /* NUM_MACC */ 2>,
53     classifier_input, hls::ref(classifier_input_valid), conv1_output,
54     hls::ref(conv1_output_valid), conv1_weights, conv1_bias);
```

Likewise, the other function calls have also been replaced with thread declarations. Using threads will make these function calls run in parallel in software, as well as in hardware.

On line 94, the `MaxComp` function has also been modified to have the input valid signal. Note that the output to the `MaxComp` layer is a FIFO and not a 4-bit register as we need for the hardware interface. We will fix this later.

```
92 // MaxComp runs continuously, taking the output of the fully-connected
93 // layer and
94 // finds the digit with the maximum confidence and reports it.
94 void MaxComp(DType input[10], volatile bool &input_valid,
95             FIFO<ap_uint<4>> &output_digit) {
```

At the top of the `digit_recognition.cpp` file, you will see all the input and output tensors declared. These represent the RAMs that will be instantiated in hardware. Note that all of them have a contention free pragma applied to them. This is because we have already made sure that a single RAM will always have at most one accessor in a single cycle due to the synchronization handshaking between layers. Because of this, we do not need arbiters to be generated and we can remove them with the contention free pragma. Notice on line 34 that the contention free pragma is also applied to the input and output valid signals as well because there are only two users for each and the users' writes will never overlap, similar to the producer consumer case we saw in Section 0.


```
10 // Tensors outputs of each layer in the CNN.
11 #pragma HLS memory impl variable(conv1_output) contention_free(true)
12 Tensor<26, 2> conv1_output;

32 // Valid signals between CNN layers to make sure downstream layers are always
33 // accessing valid data
34 #pragma HLS memory impl variable(conv1_output_valid) contention_free(true)
```

After making these modifications, we have fulfilled most of the requirements we described in the above sections. We have created a CNN that has a memory interface as input. We have used the required CNN structure and the kernels, weights, and biases provided. We have also implemented data flow parallelism with threads to achieve the same performance of the CNN as the original RTL design. We have modified the number of MACCs used in each layer to match that of the original RTL.

However, there are a few requirements left unmet. These are some incompatibilities between the interfaces generated and the RTL design due to the added handshaking signals and the output FIFO interface.



You should now click Compile Software to Hardware (). When the SmartHLS compile is finished the summary.hls.rpt report file will open, showing the following RTL 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
classifier_input	Memory	classifier_input_address_a	10	output
		classifier_input_address_b	10	output
		classifier_input_clken	1	output
		classifier_input_read_data_a	16	input
		classifier_input_read_data_b	16	input
		classifier_input_read_en_a	1	output
		classifier_input_read_en_b	1	output
classifier_input_valid	Scalar Memory	classifier_input_valid_read_data	8	input
		classifier_input_valid_write_data	8	output
		classifier_input_valid_write_en	1	output
classifier_output	Output AXI Stream	classifier_output_ready	1	input
		classifier_output_valid	1	output
		classifier_output	4	output

These RTL interfaces were generated automatically by SmartHLS based on the global variables used by the top-level function ClassifierPipeline, corresponding to the classifier_input tensor, the classifier_input_valid handshaking signal, and the classifier_output FIFO in the source code.

```
26 volatile bool classifier_input_valid = false;
```

```
29 Tensor<28, 1> classifier_input;
```

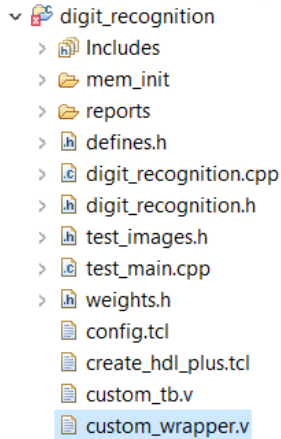
```
30 FIFO<ap_uint<4>> classifier_output;
```

We would like to match the expected RTL interface from Table 2. The classifier_input memory interface is compatible with the RTL interface: *ram_rd_data[15:0]*, *ram_rd_en*, *ram_rd_addr[9:0]*. The expected RTL interface has an *eof* input indicating the end of frame. Notice in the interface report that the classifier_input_valid is a scalar memory and is expected to stay high until the first layer sets it to 0 in our implementation. We can easily implement this in RTL by detecting the positive edge of the *eof* input to indicate that the input RAM is valid. The expected RTL interface has a registered output: *result[3:0]*. The classifier_output interface is a FIFO output instead of a registered output. But we can implement this using a register in RTL using the FIFO valid signal as the register enable.

The modifications needed to match the expected RTL interface were not implemented in SmartHLS as it is simpler to handle these in RTL. Generally, we recommend having standard interfaces (AXI interface) in your SmartHLS design. In other cases, we recommend creating a wrapper in RTL to handle a customized RTL interface such as in this case.



Open the custom_wrapper.v source file.



In this custom RTL wrapper file, we have bridged the SmartHLS-generated RTL interface to the expected RTL interface from Table 2 needed to interface with the rest of the design. On line 28, we have also added logic to register the output prediction whenever the output FIFO valid is high. To interface with the `input_valid` signal, we have added logic to detect the positive edge of the input end of frame (EOF) signal to create a one cycle start signal to initiate the CNN. On line 36, a new prediction is initiated (`input_valid` set to 1) when the end of frame start signal is asserted and the CNN has already finished generating the previous prediction (`input_valid` is 0).

```
3 module custom_wrapper (
4     input clk,
5     input reset,
6
7     input eof,
8     input [15:0] ram_rd_data,
9     output      ram_rd_en,
10    output [9:0] ram_rd_addr,
11
12    output reg [3:0] result
13 );

25 // Implements registered prediction output
26 always @ (posedge clk)
27     if (reset) result <= 0;
28     else if (classifier_output_valid) result <= classifier_output_data;
29
30 // Implements input valid memory using external EOF signal
31 always @ (posedge clk)
32     if (reset)
33         input_valid <= 0;
34     else if (classifier_input_valid_write_enable)
35         input_valid <= classifier_input_valid_write_data[0];
36     else if (eof & ~input_valid)
37         input_valid <= 1;
```

Simulation of Digit Recognition CNN

Now that we have met all requirements that we set out to meet, we can simulate in software to verify the functionality of the CNN is still correct. In `test_main.cpp`, we have a testbench that is the same as in the software version, but instead we call `ClassifierPipeline` at the beginning of the function to launch the threads before providing input via writing the input buffer and then setting the input valid signal to true. We wait for `classifier_input_valid` to be 0 before giving new input to make sure that the `ClassifierPipeline` has fully read all values from the input array, otherwise we might overwrite values that have yet to be read.

```
7 // Call ClassifierPipeline to start the layer threads.
8 ClassifierPipeline();
9
10 // Run 10 iterations, each with a 28x28 input image of digit from 0 to 9.
11 for (unsigned n = 0; n < 10; n++) {
12     // Wait for the CNN to finish reading input before writing new input.
13     while (classifier_input_valid)
14         ;
15     for (unsigned i = 0; i < 28; i++)
16         for (unsigned j = 0; j < 28; j++)
17             classifier_input[i][j][0] = test_images[n][i][j][0];
18     // Signal for the CNN that the input is ready for use.
19     classifier_input_valid = true;
20 }
```



Run Compile Software (🔧) and Run Software (▶) to run the software simulation. You should see something like the following console output where the CNN predicts the digits 0-9.

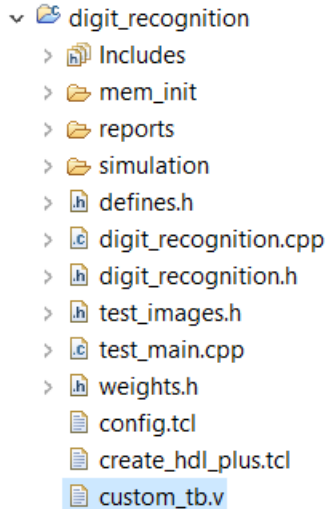
```
Highest confidence: 3267, digit-0
Highest confidence: 2686, digit-1
Highest confidence: 3284, digit-2
Highest confidence: 2690, digit-3
Highest confidence: 1935, digit-4
Highest confidence: 1370, digit-5
Received prediction: 0
Received prediction: 1
Received prediction: 2
Received prediction: 3
Received prediction: 4
Received prediction: 5
Highest confidence: 2913, digit-6
Received prediction: 6
Highest confidence: 3555, digit-7
Received prediction: 7
Highest confidence: 1879, digit-8
Received prediction: 8
Highest confidence: 2093, digit-9
Received prediction: 9
```

Next, we want to create a custom testbench to test this the generated hardware in simulation,

as co-simulation does not support threaded designs that run forever.



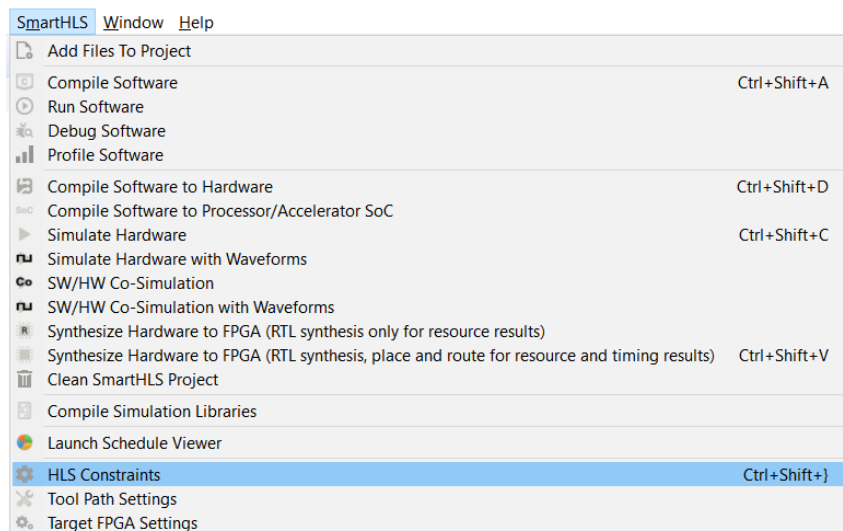
Open the custom_tb.v source file to see the custom testbench.



As we have already seen a testbench example before, we will not go into detail about this one. The structure is similar to the testbench in section 0 except for using a RAM interface to provide input. This testbench reads the same digits as the software testbench in hex format and then sends them as input to the hardware CNN.

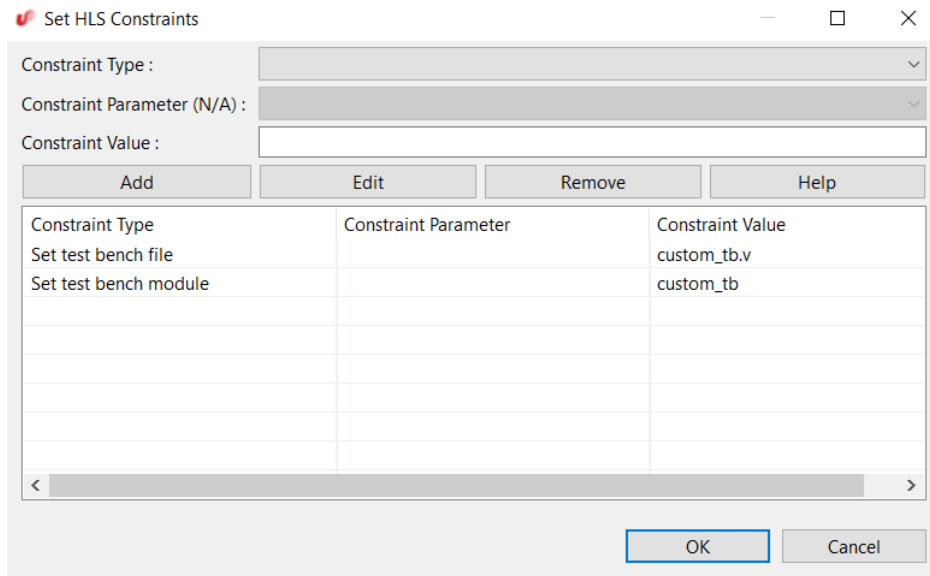


The custom testbench should already be set up in the constraints in this project. You can verify it by opening up the Constraints menu. Navigate to the SmartHLS menu and select HLS Constraints.





Once the constraints editor is brought up, check the “Set test bench file” and “Set test bench module” constraints exist.



Now we can test our design with our custom testbench. Click the Run Simulation button (►) to run our custom testbench. You should see similar output to the software test stating that the digits 0-9 were recognized.

```
# Highest confidence: 3267, digit- 0
# Prediction: 0 (cycle gap since last output = 25556)
# Highest confidence: 2686, digit- 1
# Prediction: 1 (cycle gap since last output = 13436)
# Highest confidence: 3284, digit- 2
# Prediction: 2 (cycle gap since last output = 13436)
# Highest confidence: 2690, digit- 3
# Prediction: 3 (cycle gap since last output = 13436)
# Highest confidence: 1935, digit- 4
# Prediction: 4 (cycle gap since last output = 13436)
# Highest confidence: 1370, digit- 5
# Prediction: 5 (cycle gap since last output = 13436)
# Highest confidence: 2913, digit- 6
# Prediction: 6 (cycle gap since last output = 13436)
# Highest confidence: 3555, digit- 7
# Prediction: 7 (cycle gap since last output = 13436)
# Highest confidence: 1879, digit- 8
# Prediction: 8 (cycle gap since last output = 13436)
# Highest confidence: 2093, digit- 9
# Prediction: 9 (cycle gap since last output = 13436)
```

Checking Quality of Results (QoR): Fmax and Area

Synthesize hardware to FPGA (🔌), this will take about 10 minutes. When finished, you should

see the following result in the summary.results.rpt report file:

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

Clock Domain	Target Period	Target Fmax	Worst Slack	Period	Fmax
clk	10.000 ns	100.000 MHz	5.259 ns	4.741 ns	210.926 MHz

The 210.926 MHz Fmax for the CNN meets both the 100 MHz frequency requirement of the Hello FPGA design presented in Figure 24, as well as the 148.500 MHz frequency requirement for the PolarFire® Video Kit design presented in Figure 25.

Clock Domain	Required Period (ns)	Required Frequency (MHz)
FlashFreeze_SB_0/CCC_0/GL0	10.000	100.000

Figure 24: Hello FPGA timing requirements

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

Figure 25: PolarFire® Video Kit timing requirements

In the resource usage table, the number of Math blocks is 25 instead of the 20 blocks we targeted in the design. This is due to a multiply-by-constant edge case that was not reduced to shifts and adds by SmartHLS and was also not reduced by Synplify during synthesis. This will be fixed in a future release of SmartHLS. We can also manually modify this in the generated Verilog to reduce the multiply and get Math usage to 20, but we will not do it for this training session. All other resource usage is below 3% of the PolarFire FPGA and is acceptable.

===== 3. Resource Usage =====

Resource Type	Used	Total	Percentage
Fabric + Interface 4LUT*	7067 + 1740 = 8807	299544	2.94
Fabric + Interface DFF*	5398 + 1740 = 7138	299544	2.38
I/O Register	0	1536	0.00
User I/O	0	512	0.00
uSRAM	46	2772	1.66
LSRAM	8	952	0.84
Math	25	924	2.71

We have now created a drop-in replacement for the original RTL digit recognition CNN that fits all our hardware requirements.

Comparison Between RTL and HLS

After porting the original RTL implementation to SmartHLS, we would like to make a comparison between the two using some productivity metrics.

Table 3: Productivity of SmartHLS compared to RTL

Productivity Metrics	SmartHLS	RTL Reference
Design Time	1 week	1 month
Simulation Runtime	0.3 second	55 seconds
Lines of Code	334	1984
Complexity in System Integration Design	8 instances & tens of connections	93 instances & hundreds of connections

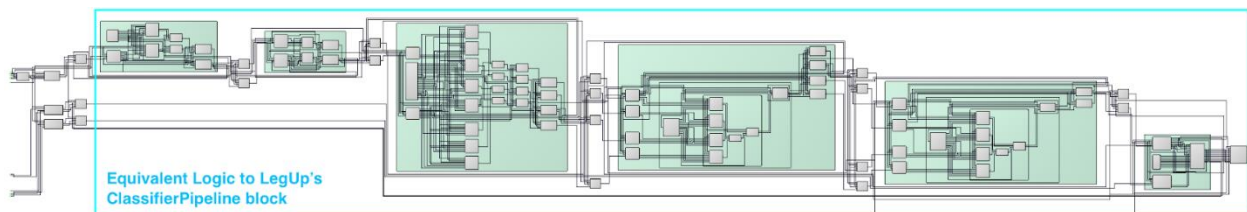


Figure 26: Number of blocks in original SmartDesign CNN

As you can see from the metrics in Table 3 and the complexity of the original SmartDesign CNN in Figure 26, SmartHLS allows a hardware designer to create the same hardware originally implemented in RTL much faster and simpler. The resulting code is simpler to read, understand, modify, and maintain.

Appendix A: Instruction-level Parallelism

Instruction-level parallelism is automatically achieved by SmartHLS by analyzing dependencies within the software and scheduling independent instructions as early as possible. SmartHLS applies instruction-level parallelism optimizations at the basic block level in the LLVM intermediate representation (IR). LLVM is the open source compiler framework that SmartHLS is built on. A basic block is a group of instructions that always run together with a single entry point at the beginning and a single exit point at the end. A basic block in LLVM IR always has a label at the beginning and a branching instruction at the end (`br`, `ret`, etc.). An example of LLVM IR is shown below, where the `body.0` basic block performs an addition (`add`) and subtraction (`sub`) and then branches unconditionally (`br`) to another basic block labeled `body.1`. Control flow occurs between basic blocks.

```
body.0:
  %0 = add i32 %a, %b
  %result = sub i32 %0, 5
  br label %body.1
```

We can look at how instructions are scheduled to have instruction-level parallelism in each basic block by compiling a project to hardware and looking at the schedules for each basic block in the Schedule Viewer. There are five examples we will look at to understand when instructions can run in parallel and when they cannot.



Open the `instruction_level_parallelism` project in SmartHLS and open the `instruction_level_parallelism.cpp` source file.

```
▼ instruction_level_parallelism
  > Includes
  > instruction_level_parallelism.cpp
  config.tcl
  makefile
```



Now click the Compile Software to Hardware button (🔧) to build the design and generate the schedule.

The first example we will look at is the `no_dependency` example on line 8 of `instruction_level_parallelism.cpp` with the code shown in Figure 27.

```
8 void no_dependency() {
9   #pragma HLS function noline
10   e = b + c;
11   f = c + d;
12   g = b + d;
13 }
```

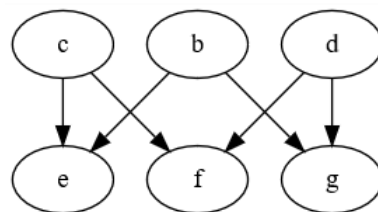
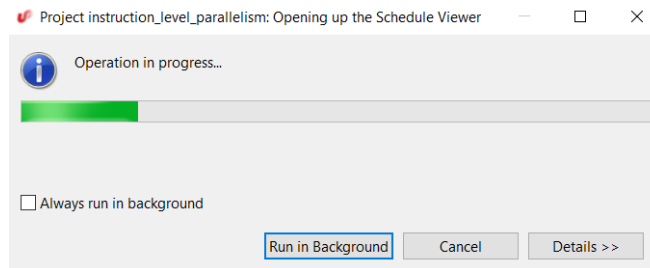


Figure 27: Source code and data dependency graph for `no_dependency` function.

In this example, values are loaded from `b`, `c`, and `d` and additions happen before storing to `e`, `f`, and `g`. None of the adds use results from the previous adds and thus all three adds can happen in parallel. The `noinline` pragma is used to prevent SmartHLS from automatically inlining this small function and making it harder for us to understand the schedule. Inlining is when the instructions in the called function get copied into the caller, to remove the overhead of the function call and to allow more opportunities to optimize.



We can see the instruction-level parallelism if we launch the Schedule Viewer (🌈).
Tip: After the Schedule Viewer opens, click on the SmartHLS IDE and click “Run in Background” on the pop-up menu to close the menu. Now we can easily see both the source code and the Schedule Viewer.



In the Explorer tab on the left, click on `BB_entry` under `no_dependency` to bring up the schedule for the `BB_entry` basic block in the `no_dependency` function. We can see three groups of instructions that load two values, add them, and then store the result. This corresponds to each of the three adds in the source code. SmartHLS schedules each of these add operations to start as early as possible: at clock cycle 1. The first clock cycle occurs when both top-level interface ports `ready` and `start` are high on the SmartHLS generated module.

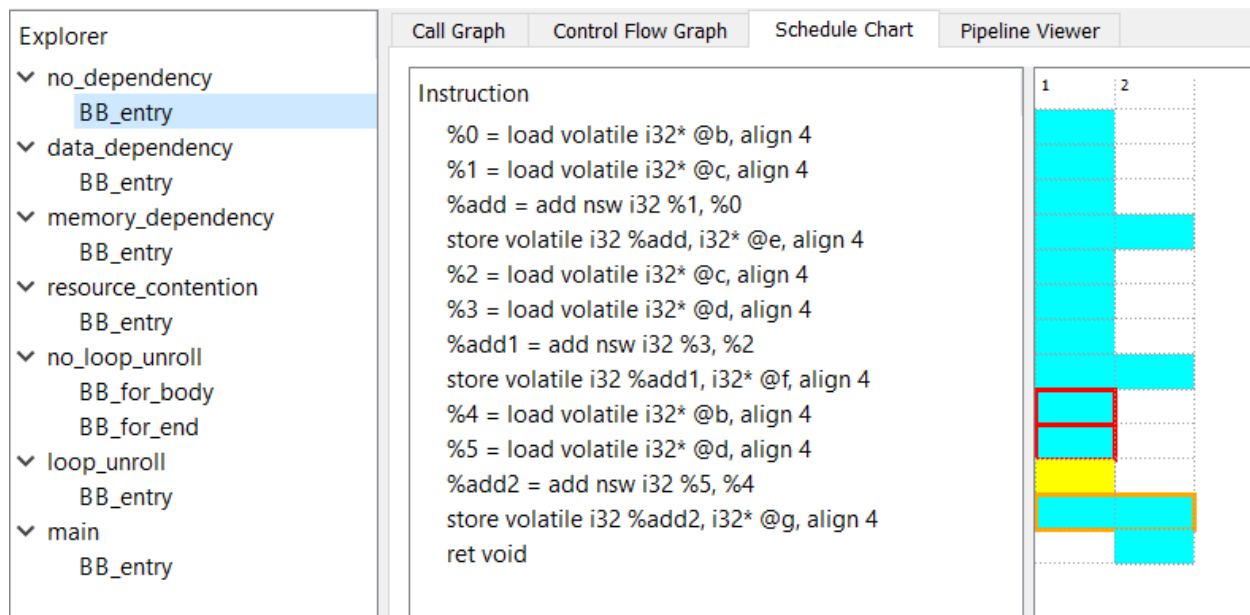


Figure 28: Schedule viewer for the `no_dependency` function.



If you highlight a store instruction in the Schedule chart as shown in Figure 28, SmartHLS will highlight the instructions the selected instruction depends on in red and highlight the instructions that depend on the selected instruction in orange. Here the store instruction highlighted in yellow depends on the result of the add instruction as we expect.

We have declared all the variables used in this function as **volatile**. The volatile C/C++ keyword specifies that the variable can be updated by something other than the program itself, making sure that any operation with these variables do not get optimized away by the compiler as every operation matters. An example of where the compiler handles this incorrectly is seen in section 0, where we had to declare a synchronization signal between two threaded functions as volatile. Using volatile is required for toy examples to make sure each operation we perform with these variables will be generated in hardware and viewable in the Schedule Viewer.

```
4 volatile int a[5] = {0};
5 volatile int b = 0, c = 0, d = 0;
6 volatile int e, f, g;
```

Next, we will look at some cases where there are dependencies in the code and SmartHLS cannot schedule all instructions in the first cycle.



Look back at the SmartHLS IDE and look at the `data_dependency` function on line 15 of `instruction_level_parallelism.cpp` as shown in Figure 29.

```

15 void data_dependency() {
16 #pragma HLS function noline
17     e = b + c;
18     f = e + d;
19     g = e + f;
20 }

```

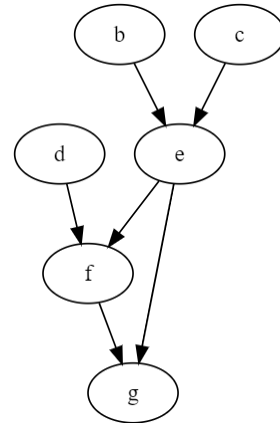


Figure 29: Source code and data dependency graph for the data_dependency function

In the data_dependency function, the result of the first add which is stored in e is used in the second and third adds. The result of the second add is also used in the third add. These are examples of data dependencies as later adds use the data result of previous adds. Because we must wait for the result e to be produced before we can compute f, and then the result f must be produced before we can compute g, not all instructions can be scheduled immediately. They must wait for their dependent instructions to finish executing before they can start, or they would produce the wrong result.

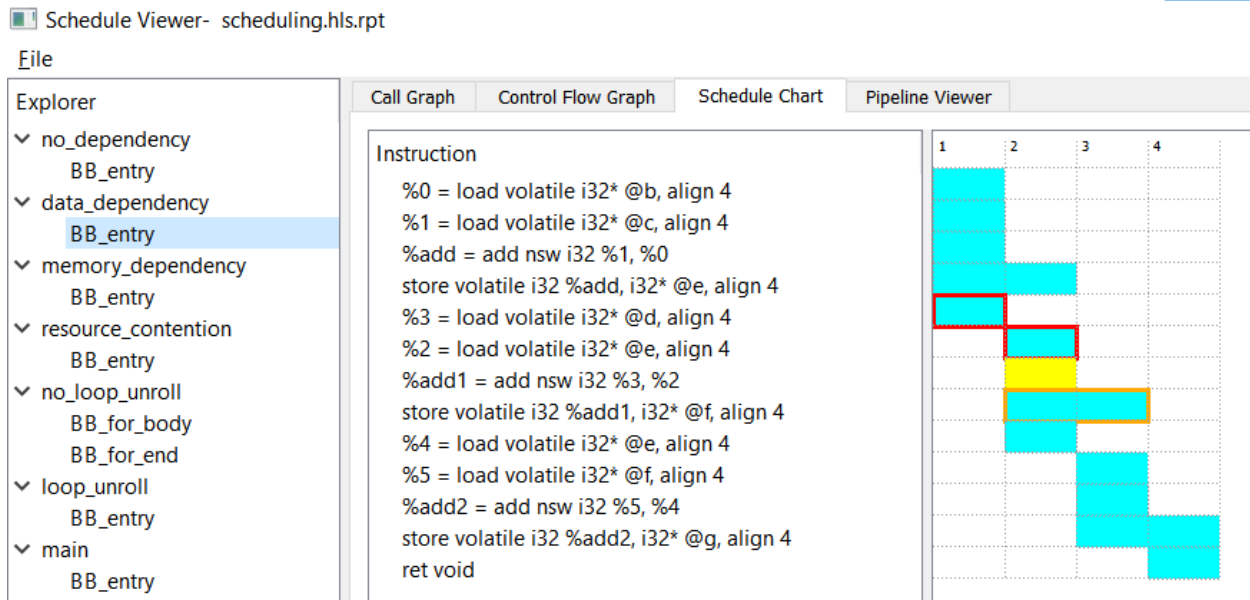


Figure 30: Schedule Viewer for data_dependency function.



To see the effects of this, open the Schedule Viewer and click BB_entry under the data_dependency function as shown in Figure 30. We can see the loads from e for

instruction %2 and %4 was scheduled for the second cycle when the store to ϵ finishes. This delays the execution of the second add by one cycle. Similarly, the loads from ϵ for instruction %5 were scheduled for the third cycle when the store to ϵ finishes. This delays the execution of the third add by another cycle.

Note, the SmartHLS scheduler uses the latency of the operations when scheduling instructions. If the latencies are not properly set up for a project or if a design generated for a different latency is used on a non-compatible FPGA, there will be functional errors in hardware. Make sure to select the correct FPGA when setting up the SmartHLS project or set it under the Set Target FPGA option in the SmartHLS drop-down menu. Also make sure to set custom latencies under HLS Constraints in the SmartHLS drop-down menu when necessary, as shown in Figure 31.

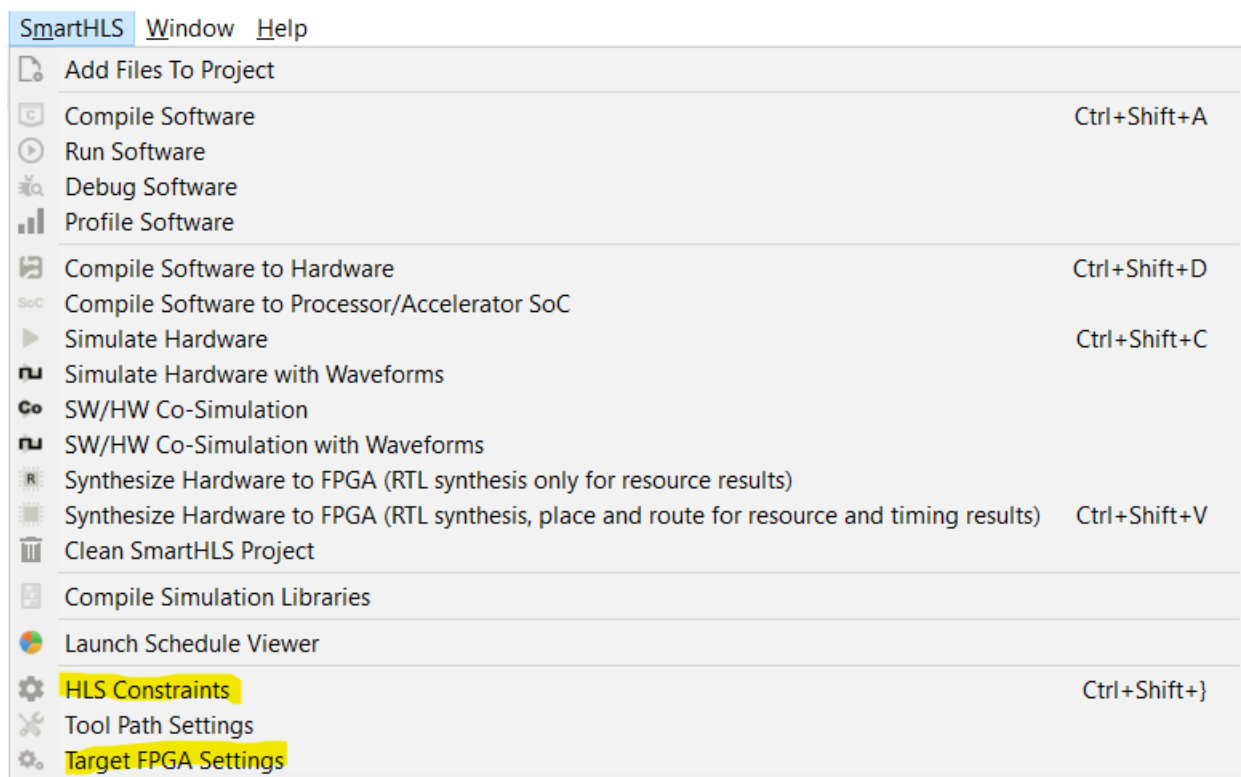


Figure 31: HLS Constraints and Target FPGA Settings options in the SmartHLS IDE.

Next, we will look at special kind of dependency that can happen for memories.



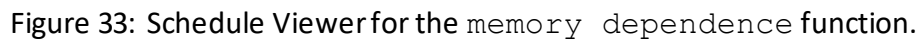
Look back at the SmartHLS IDE and look at the `memory_dependency` function on line 22 of `instruction_level_parallelism.cpp` as shown in Figure 32.

```

graph TD
    b((b)) --> a((a))
    a --> f((f))
    a --> g((g))

```

In Figure 32, a value `b` is stored into array `a` at a variable index `i`. Because the index `i` is volatile, we cannot be sure of the value used to index the array even though `i` is initialized to 0. As we cannot be sure which element is being stored to at compile time, we cannot be sure if there will be a data dependence on the value `b` when we load elements one and two in the loads that follow. SmartHLS will conservatively assume there is a dependence in this case and schedule all memory accesses after this store to wait until the store is finished, even if it turns out there is no dependence at run time. Even if the index was not volatile, SmartHLS currently cannot analyze the range of a variable index into an array and will still conservatively assume that any index in the array can be accessed, resulting in the same kind of dependency.





To see the effects of this, open the Schedule Viewer and click `BB_entry` under the `memory_dependence` function as shown in Figure 33. We can see the two loads from `a`, `%1` and `%2` were scheduled to run on cycle 3 when the store to `a` in cycle 2 finishes. The Schedule Viewer currently does not show memory dependencies, so the store instruction is not highlighted as an instruction the highlighted load depends on.

Next, we will see an example of when operations that use a limited resource cannot be scheduled in parallel due to a lack of resources.



Look back at the SmartHLS IDE and look at the `resource_contention` function on line 30 of `instruction_level_parallelism.cpp`.

```
30 void resource_contention() {  
31     #pragma HLS function noline  
32     e = a[0];  
33     f = a[1];  
34     g = a[2];  
35 }
```

In this example, three values are loaded from the array `a` and stored into `e`, `f`, and `g`. Arrays are mapped to RAMs in hardware with up to two ports. In this case, three loads are performed but the RAM only has two ports which can be used per cycle. The first two loads can be scheduled immediately but the third store must wait for the next cycle when a port becomes free to use.

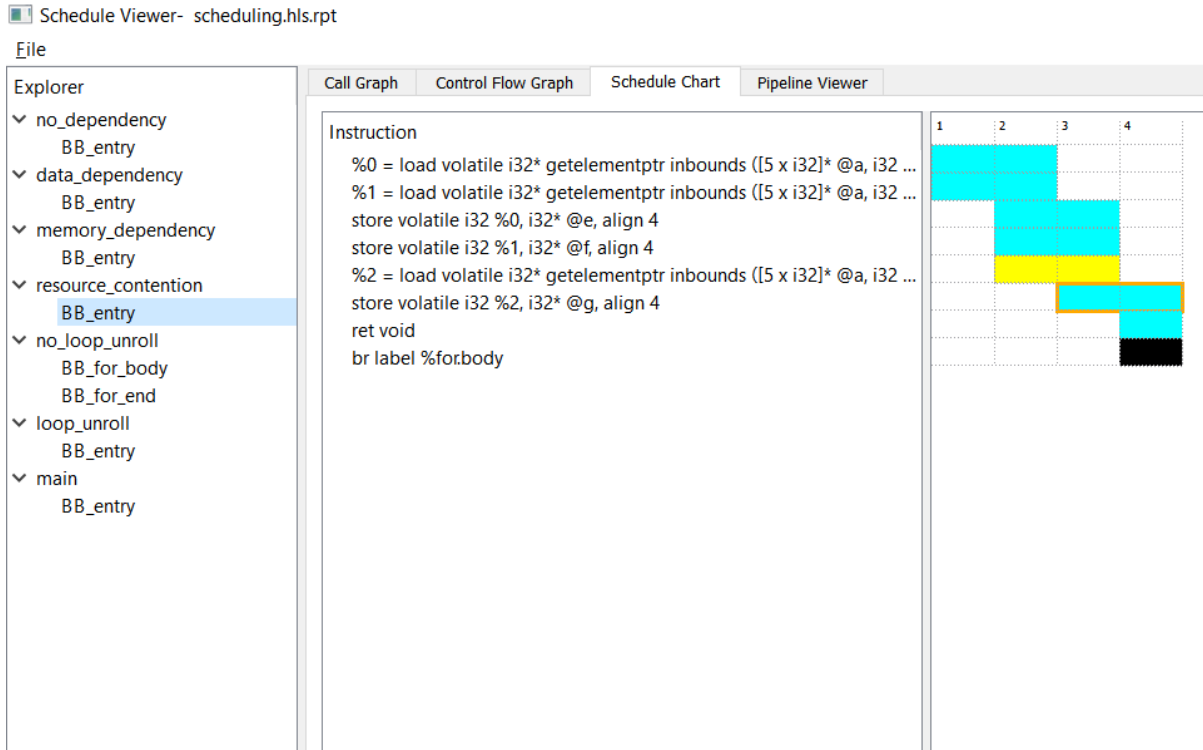


Figure 34: Schedule Viewer for `resource_contention` function.



To see the effects of this, open the Schedule Viewer and click `BB_entry` under the `resource_contention` function in Figure 34.

We can see the first two loads to `a` were scheduled for the first cycle, however the third load was scheduled for the second cycle due to resource contention even though it does not have any dependencies to previous instructions. This type of resource contention can happen with any kind of limited resource, which SmartHLS will automatically consider when generating the schedule for a design.

Next, we will see an example of how loops prevent operations from being scheduled in parallel.

```

37 void no_loop_unroll() {
38 #pragma HLS function noline
39     int h = 0;
40 #pragma HLS loop unroll factor(1)
41     for (int i = 0; i < 5; i++) {
42         h += a[i];
43     }
44     e = h;
45 }
46
47 void loop_unroll() {
48 #pragma HLS function noline

```

```

49     int h = 0;
50 #pragma HLS loop unroll
51     for (int i = 0; i < 5; i++) {
52         h += a[i];
53     }
54     e = h;
55 }

```

In this example, we have two functions with the same loop in the function body. The functions sum the elements of an array and store into `e`. The difference between these two functions is that `no_loop_unroll` has no unrolling on the loop and `loop_unroll` unrolls the loop completely. This affects the resulting hardware by removing the control signals needed to facilitate the loop and combining multiple loop bodies into the same basic block, allowing more instructions to be scheduled in parallel. The trade-off here is an unrolled loop does not reuse hardware resources and can potentially use a lot of resources. However, the unrolled loop would finish earlier depending on how inherently parallel the loop body is.



To see the effects of this, open the Schedule Viewer and first click on the `no_loop_unroll` function shown in Figure 35.

This will bring up the Control Flow Graph which will show us the flow of control from the basic blocks in the `no_loop_unroll` function. We can see that there is a loop body `BB_for_body` which has an arrow coming from and pointing to itself. This means that this basic block may run again when it finishes executing. The other arrow pointing to `BB_for_end` means that the `BB_for_end` block may also run after `BB_for_body` depending on the jump conditions. What these two basic blocks represent is the loop body which runs 5 times and the store to `e` after the loop finishes.

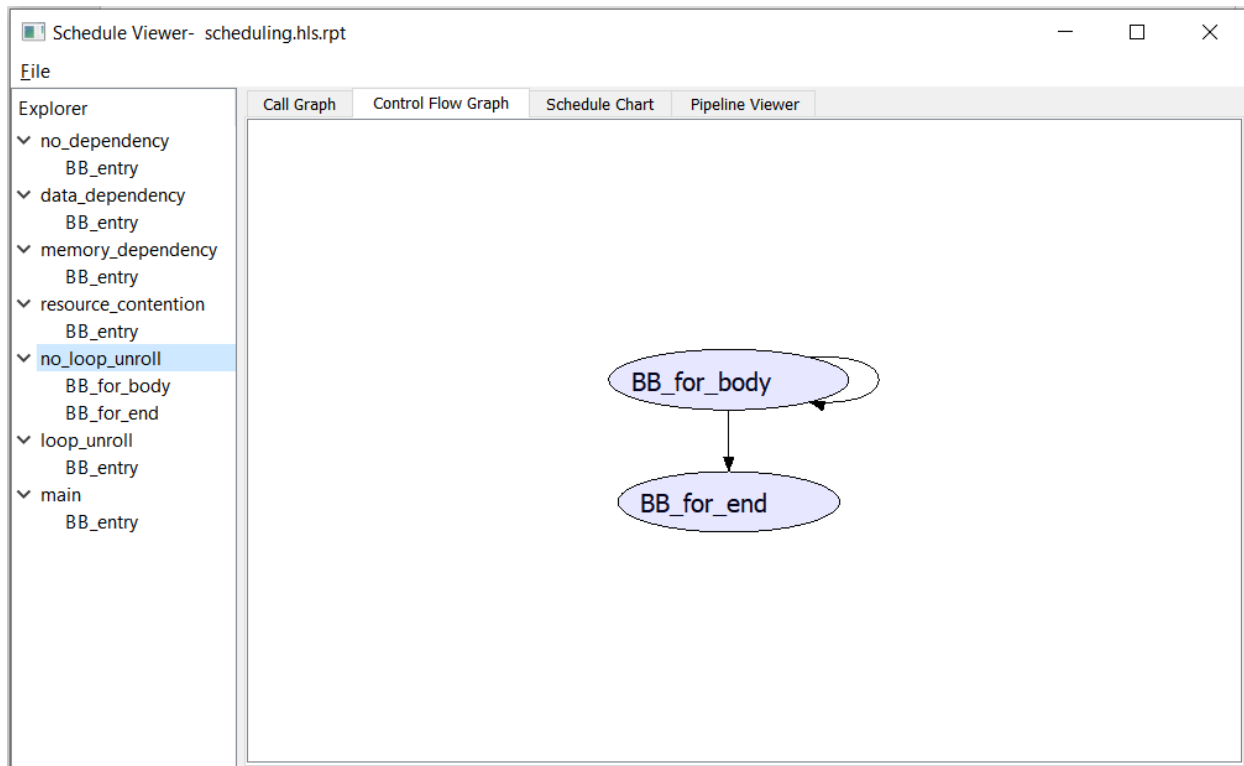


Figure 35: Schedule Viewer for `no_loop_unroll` function.



Next click on `BB_for_body` under `no_loop_unroll` to bring up the Schedule Chart for the loop body shown in Figure 36.

We can see that this loop body takes two cycles to run. Note there is no pipelining for this loop so there are no overlapping iterations. This means the entire loop of 5 iterations takes 10 cycles total to run.



Next click on the `BB_for_end` basic block under `no_loop_unroll` shown in Figure 37. This basic block runs once after the loop finishes and takes 2 cycles to finish. This means the `no_loop_unroll` function takes 12 cycles to finish in total.

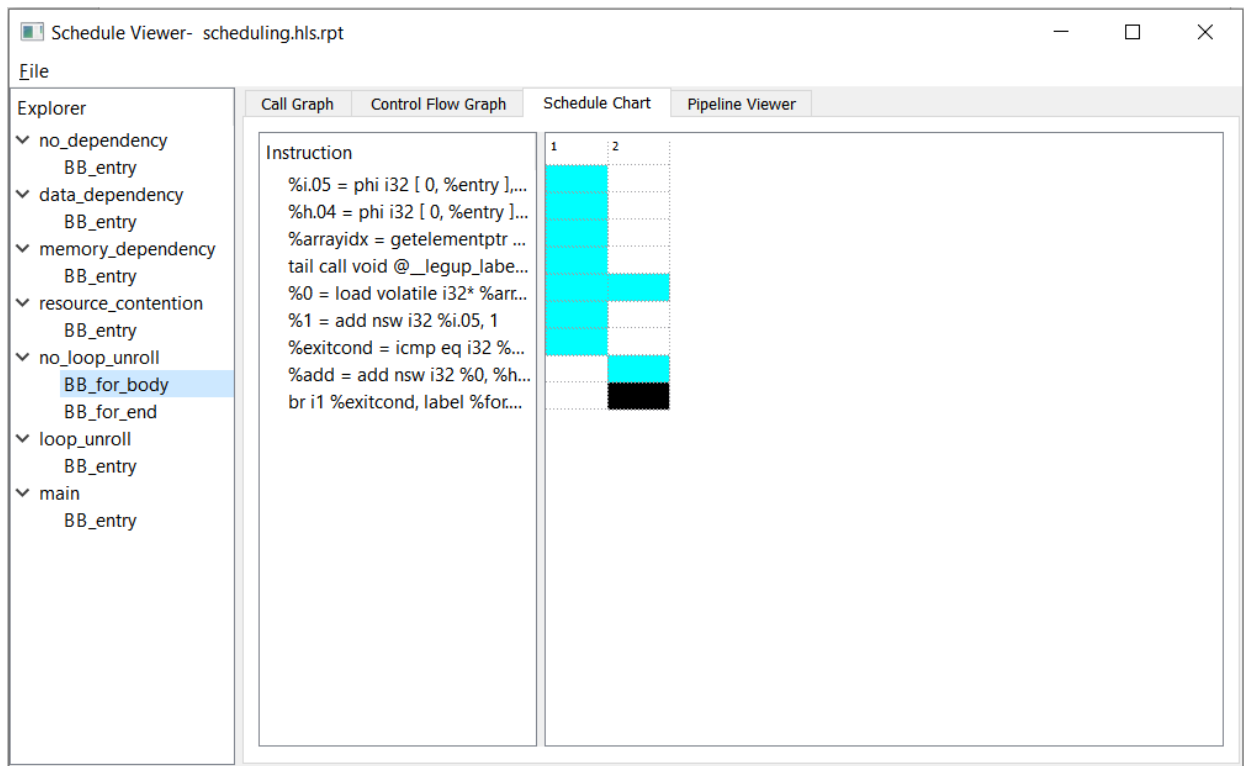


Figure 36: Schedule Viewer for BB_for_body.

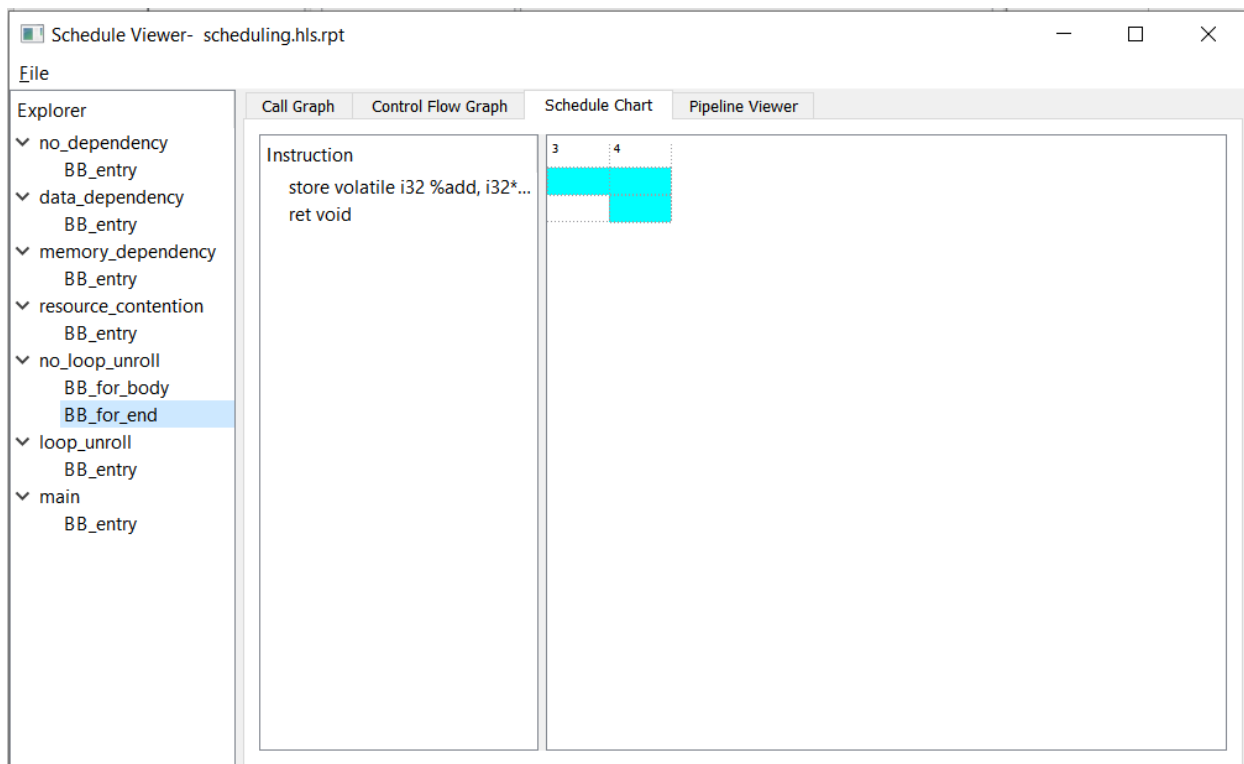


Figure 37: Schedule Viewer for BB_for_end.



Next, click on the `BB_entry` block under `loop_unroll` to bring up the schedule for `loop_unroll` shown in Figure 38.

With unrolling, this function takes 5 cycles to compute the same sum and store it to `e`. Unrolling this loop fully reduced the latency from 12 to 5.

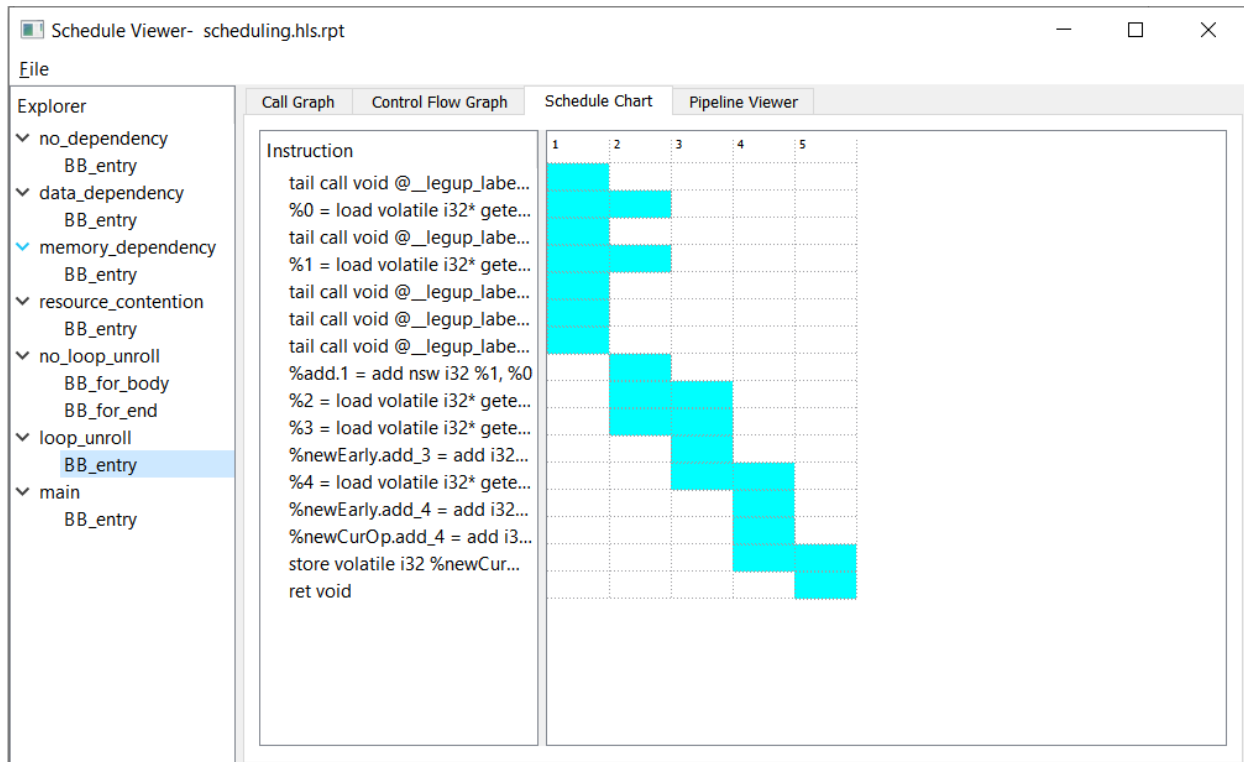


Figure 38: Schedule Viewer for `loop_unroll` function.

Although unrolling small loops can improve performance greatly without adding much area, the number of resources used will be replicated with each iteration of the loop that gets unrolled. This can make it unreasonable to unroll large loops, for example unrolling the nested loops that processes every pixel of the 1920x1080 frame in the Canny design from Training 1.

SmartHLS offers some alternatives that can be used to increase performance without causing a large increase in resource usage. The first alternative is using loop pipelining to allow iterations to run in parallel, which we have already seen. This would reduce the runtime of the loop from 10 to 6 cycles (number of iterations + pipeline latency - 1) and the runtime of the entire function from 12 to 8 cycles. The second option is to unroll the loop by a factor rather than unrolling the loop completely. This will expose some parallelism in the iterations that run together, while only replicating resources a fixed number of times. This can be done with the factor parameter of the unroll SmartHLS pragma.



Now close the Schedule Viewer and all the opened files for the `instruction_level_parallelism` project.

Appendix B: Integrating into SmartDesign

Figure 39, shows the SmartDesign IP component that contains the SmartHLS-generated CNN digit recognition block.

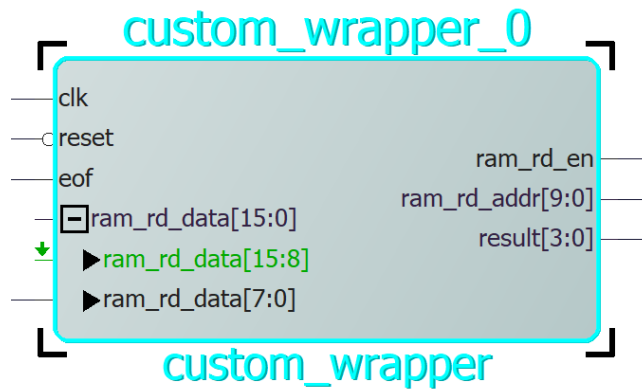
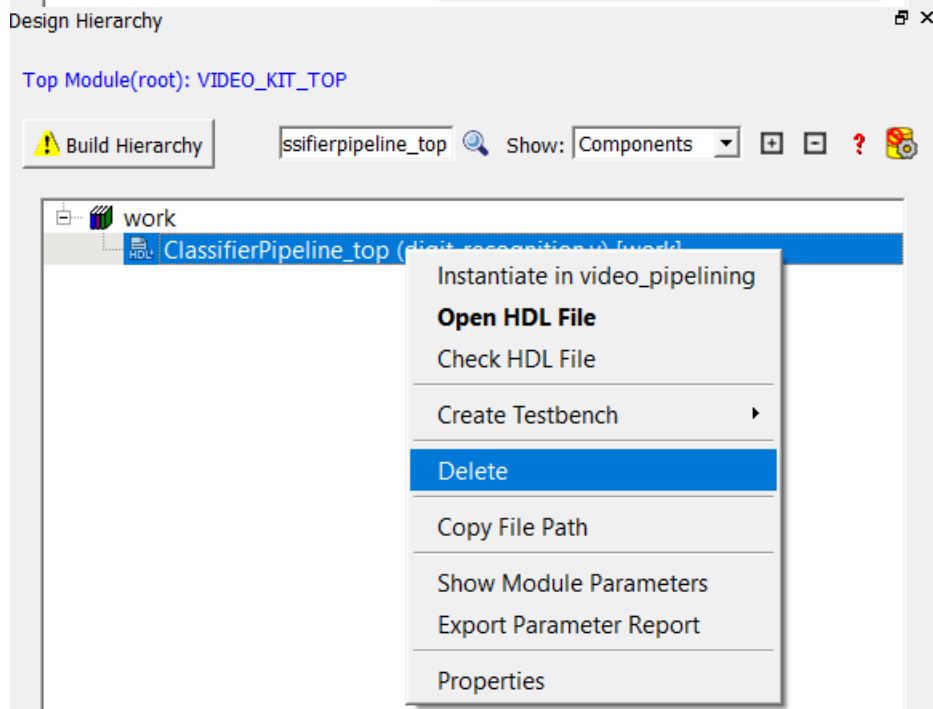
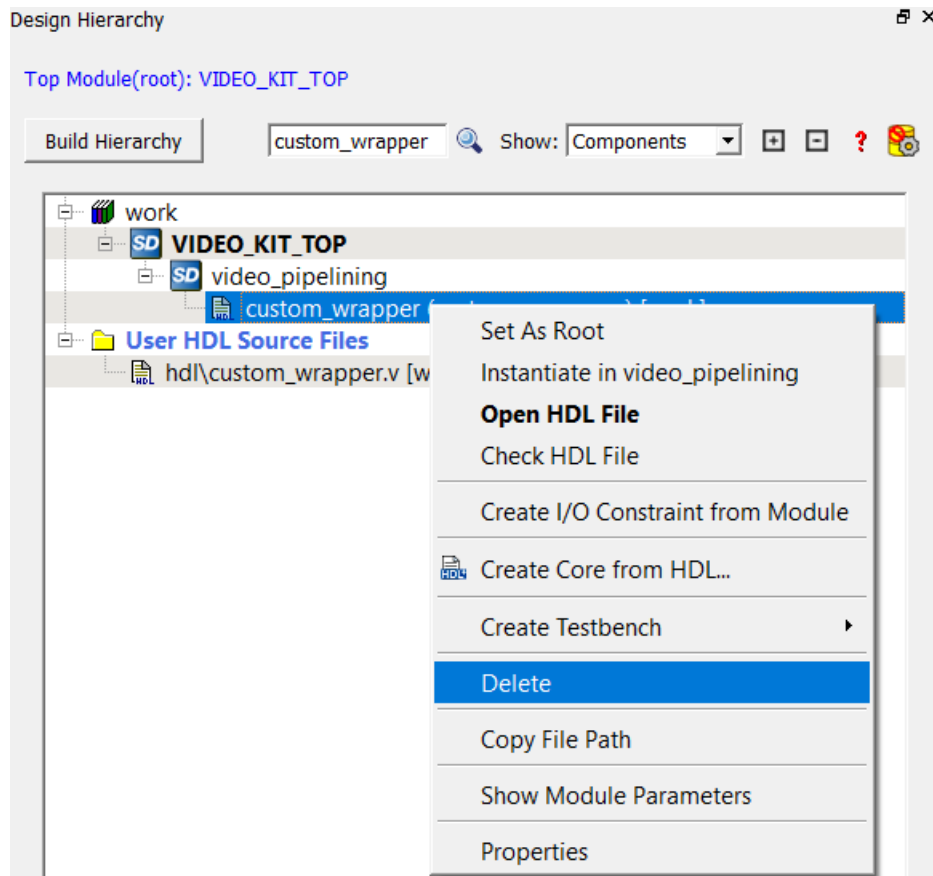


Figure 39: Block Diagram of Custom Wrapper for SmartHLS-generated CNN block

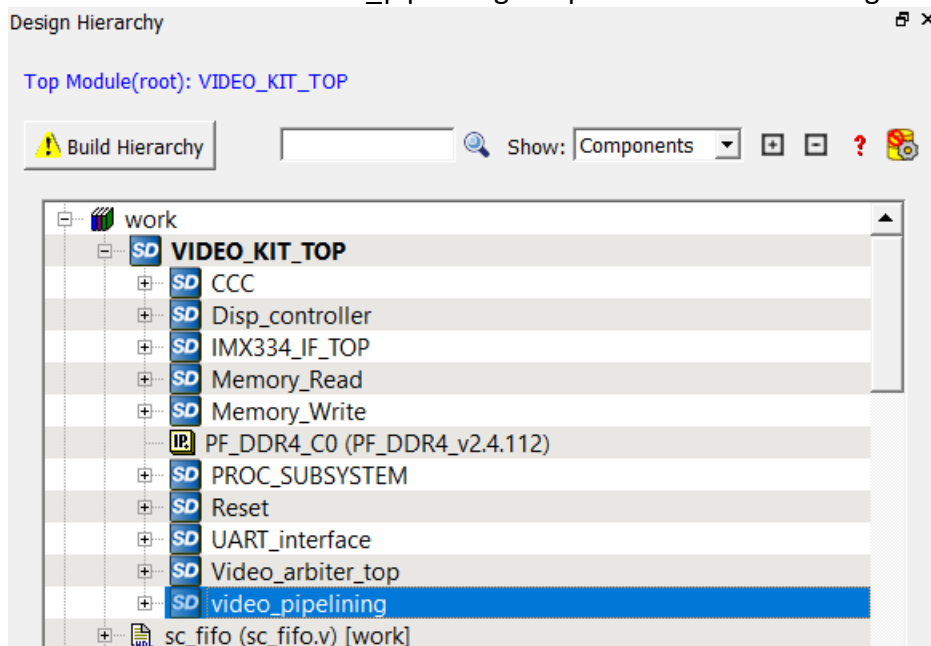


To integrate the PolarFire® Video Kit design in Libero SoC 2021.2:

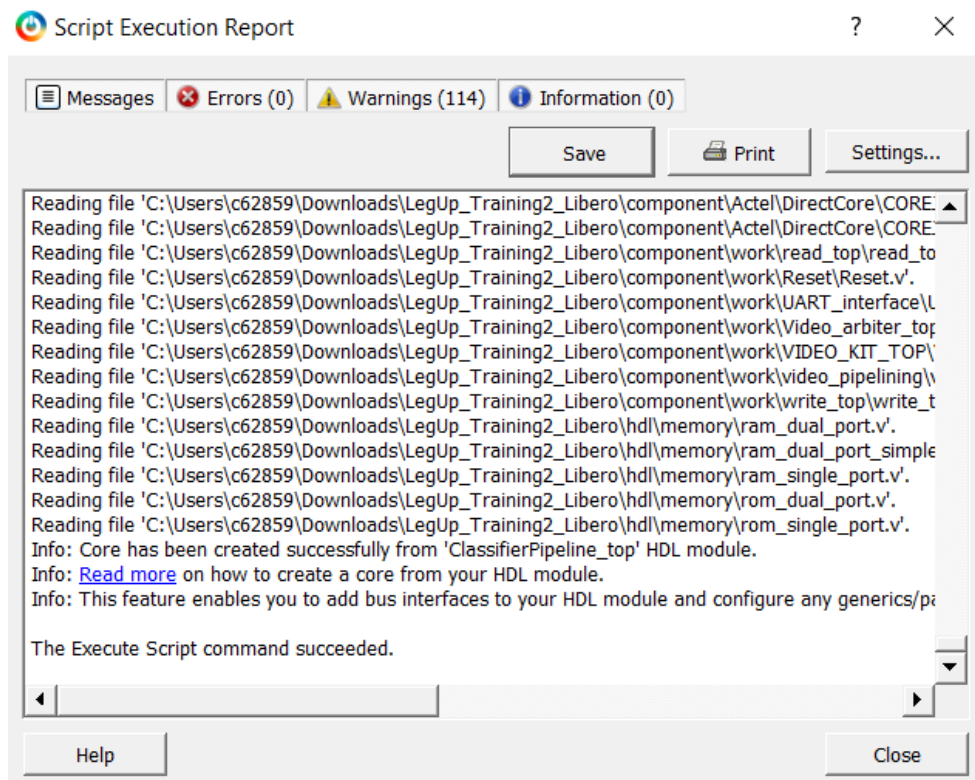
1. Download the SmartHLS_Training2_Libero.zip file if you have not already.
2. Extract the SmartHLS_Training2_Libero.zip file contents containing the project and job file.
On Windows you need to extract the project to a directory with a short name (such as C:\Downloads or C:\Workspace) and extract with 7-Zip to avoid issues with long names:
3. Launch Libero SoC 2021.2 and open the Digit Recognition project by navigating to and clicking LegUp_Training2.prjx.
Note: The Libero project was created from when SmartHLS still had the name “LegUp”, so you might see a lot of places with the word LegUp in it. This will be addressed in a future version of the training.
4. Navigate to the Design Hierarchy and search for “custom_wrapper”. Right click the custom_wrapper design component and select Delete. We want to avoid any duplicate blocks when importing the new custom_wrapper from SmartHLS. Next do the same for “ClassifierPipeline_top”.



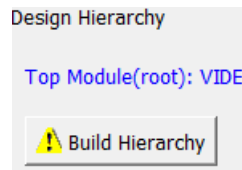
- Now clear the search, then click the plus next to the VIDEO_KIT_TOP SmartDesign file and then double click video_pipelining to open it in the SmartDesign Canvas.



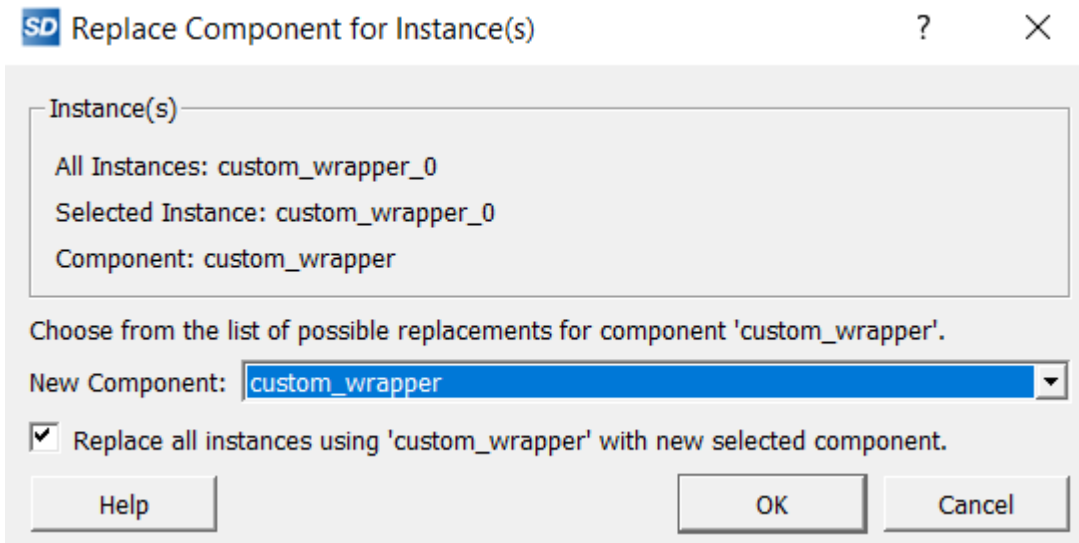
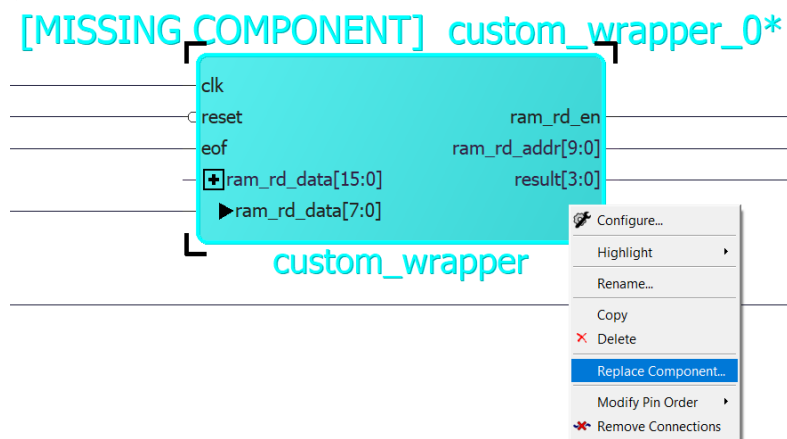
- On the top toolbar, click Project->Execute Script... and run the create_hdl_plus.tcl file from the digit_recognition SmartHLS project directory. SmartDesign will open a report window when it finishes. Make sure the script executed successfully and close the report window.




7. Manually copy the custom_wrapper.v into the Libero project hdl directory.
8. Next click the Build Hierarchy button in Libero to update the newly added files.



9. Next find the custom_wrapper block in video_pipelineing and right click and select Replace Component. Select the newly imported custom_wrapper to replace the SmartDesign block without having to reconnect any wires.



10. Click the "Generate Component" () button in the SmartDesign toolbar for video_pipelineing.

11. The custom_wrapper and ClassifierPipeline_top blocks have now been integrated and the project is ready for synthesis, place, and route. We skip this step for now since this will take 1-2 hours.

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

Appendix C: 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 40. We will use the Dual Camera Sensor inputs, the HDMI 1.4 TX (J2) to display output on a computer monitor and the USB-UART (J12) for bitstream programming.

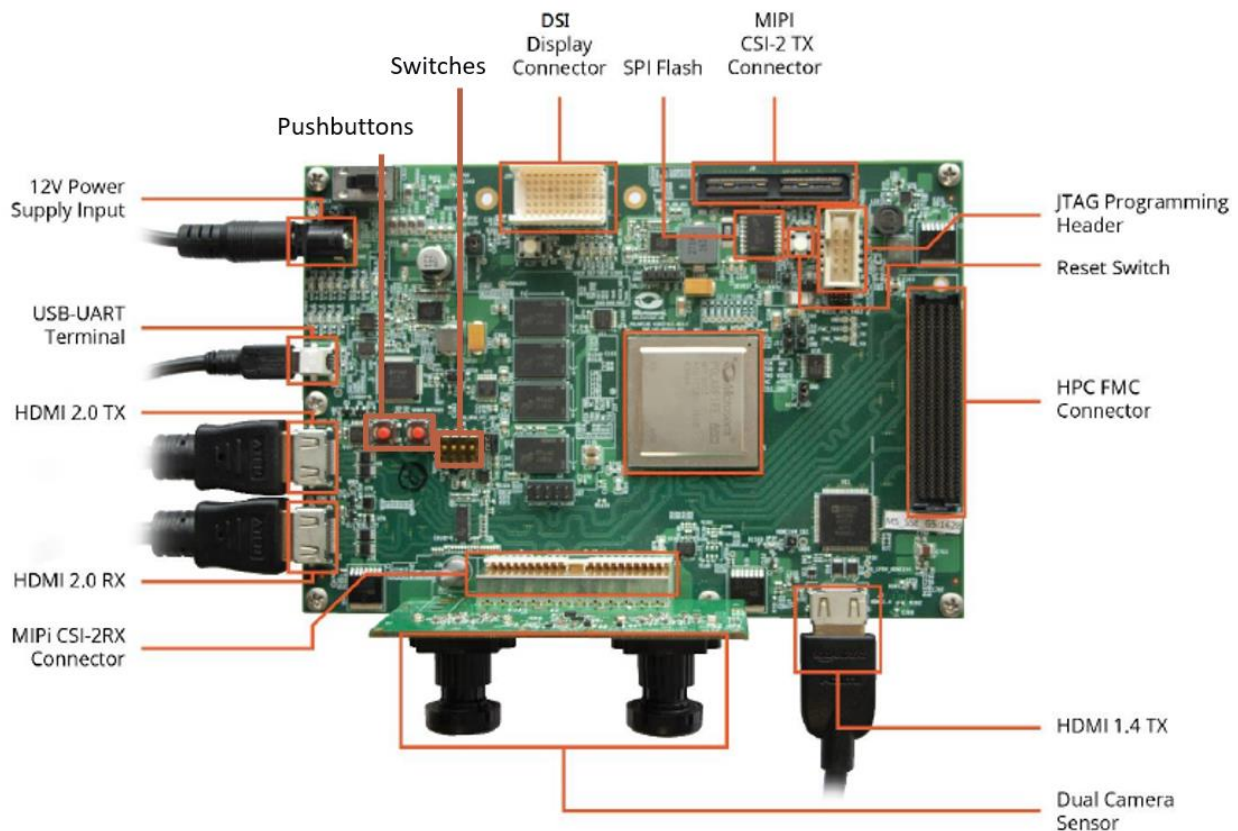
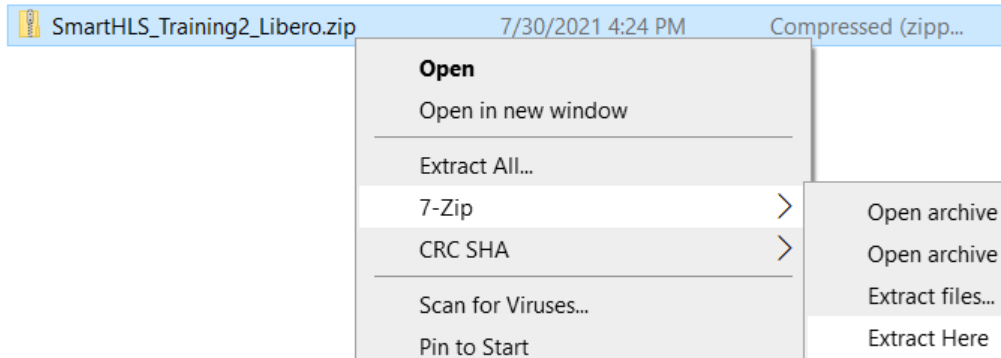


Figure 40: PolarFire® Video and Imaging Kit Peripherals

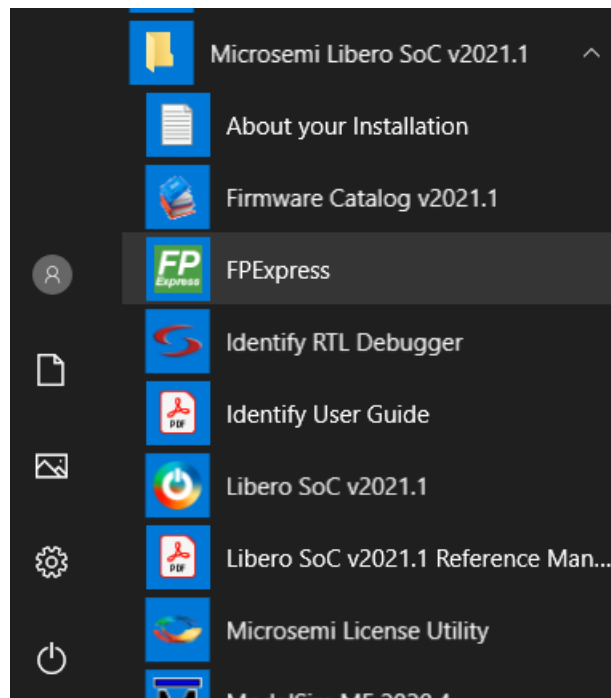


To program the design to the PolarFire board:

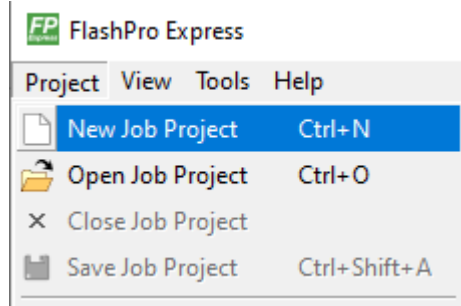
1. If you have not already, download the file `SmartHLS_Training2_Libero.zip`. Extract the zip file contents.
On Windows you will need to extract the project to a directory with a short name (such as `C:\Downloads` or `C:\Workspace`) and extract with 7-Zip to avoid issues with long filenames:



2. Connect the USB cable from J12 on the PolarFire® board to your PC.
3. Connect the camera board at J5 and remove the lens caps.
4. Connect the HDMI cable from the PolarFire Video Kit (J2) to your external Monitor.
5. Refer to [DG0849](#) for jumper settings. We use the default jumper settings shipped with the board.
6. Make sure all the DIP switches (SW6) are in the ON position.
7. Connect the AC adapter to the board and power it on (SW4).
8. Open up FlashPro Express, which you can find in the Start Menu, listed under “Microsemi Libero SoC v2021.1”:



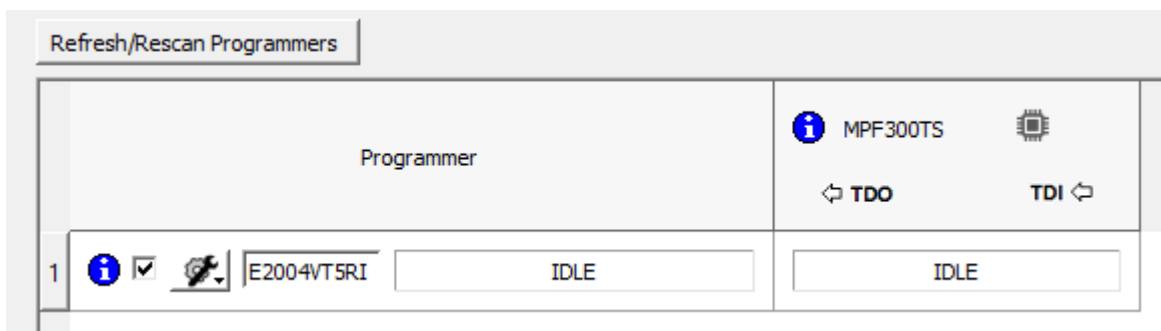
9. Select Project and New Job Project.



10. Now select the job file “LegUp_Training2_job/LegUp_Training2.job” in the folder you extracted in step 1.

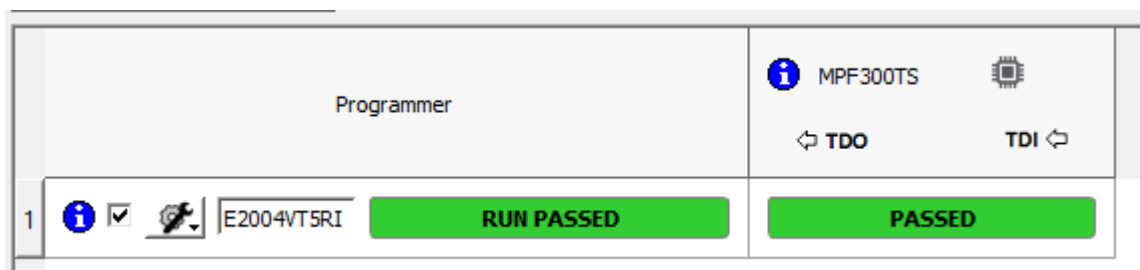
11. Enter a project location. Click OK.

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



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

14. After programming you should see the RUN PASSED. Now power cycle the board, then **make sure to press the user reset switch**, and close FlashPro Express.



15. Now you should see two video streams on your monitor, one smaller picture in picture on the top left and then the background. If the video streams look blurry, try focusing the camera by rotating the camera lens. You should also see a downscaled greyscale

version of the picture in picture video stream on the bottom left of the picture in picture represented in green, and a digit displayed under the picture in picture representing the CNN prediction for the current image. An example of this is shown in Figure 42: Expected output on Monitor for “5” digit.

16. Using your smartphone, open this document and zoom in on the example handwritten digits shown in Figure 41.

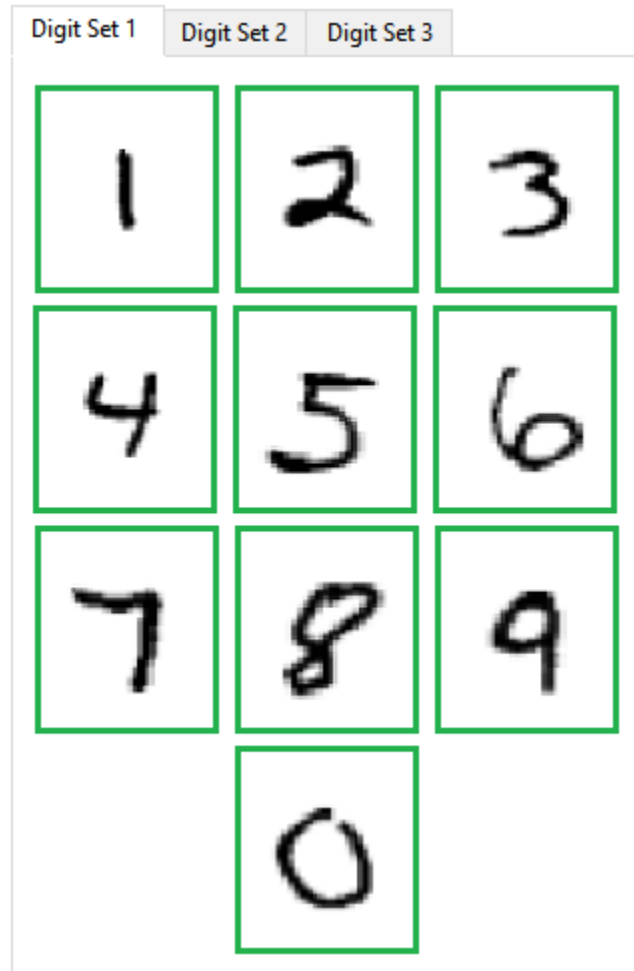


Figure 41: Example Handwritten Digits

17. Now hold your smartphone screen up the PolarFire® Video Kit camera and move your phone until the picture in picture shows only one digit. For example, if you hold the “5” digit from Figure 41 up to the camera, you should see the output on the top left of the monitor shown in Figure 42. If your prediction is stuck at one value, make sure you pressed the user reset switch to reset the circuit.

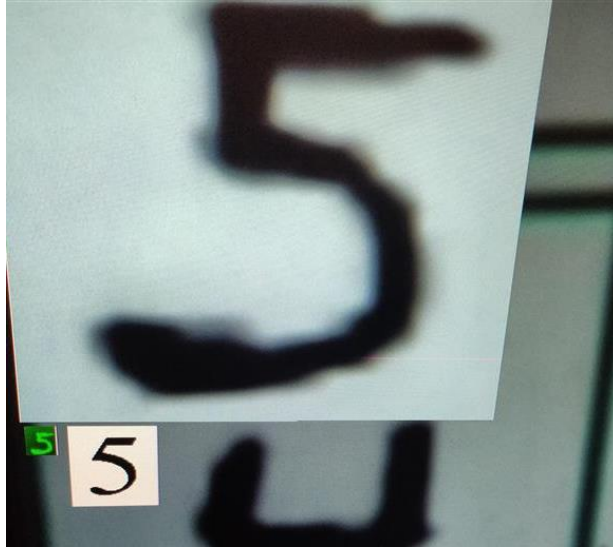


Figure 42: Expected output on Monitor for “5” digit

18. Below the picture in picture, as shown in Figure 43, you will see a small 28x28 green image which shows the downscaled greyscale image seen by the digit recognition CNN hardware block. On the right is the digit predicted by the hardware, in this case the prediction was 5. Keep in mind the prediction can sometimes be incorrect depending on the position, orientation, and readability of the input handwritten digit.



Figure 43: Downscaled digit recognition input and predicted digit output

19. You may notice the prediction may not be very accurate. This is due to the CNN structure being a smaller version of the standard MNIST digit recognition CNN. In general, you should restructure and retrain the CNN to get better effectiveness if the CNN is not performing as well as you need it to.