DIGILENT®

A National Instruments Company

ZYBO Z7 & Pcam 5C HLS Video Workshop

VERSION 2.0

1 Theoretical background

Software is everywhere. The flexibility it offers to designers allows it to be used in a multitude of applications. Many consumer, industrial or military products are either running software or began as a software model or prototype executing on a generic circuit or processor. Decades of advances in software engineering resulted in ever higher abstractions, ever smarter tools, ever increasing numbers of automatic optimizations that improve code re-use, shorten design time and increase performance. Continuous performance increase quantified by the number of instructions executed per second has been driven at first by the increase in processing frequencies, then by parallelization of algorithms and simultaneous execution of tasks by multiple processing cores.

The ubiquitous nature of software lead to most engineering problems first being approached with software solutions. Depending on the application, a software-only approach might not meet the requirements, be those latency, throughput, power or others. An expensive option would to hand the algorithm over to a hardware engineer for a custom circuit implementation. The entry cost of application-specific integrated circuit (ASIC) design is still high despite advancements in fabrication technologies. Depending on the product forecasts, an ASIC design might not be economically viable.

FPGAs bridge the gap between generic processor circuits and ASICs, allowing the use of blank reprogrammable hardware logic elements to implement custom circuits. They offer a lower barrier of entry to the power savings and performance benefits of fabrication technologies without the cost of ASIC. An algorithm optimized for FPGA implementation benefits from the inherently parallel nature of custom circuits.

2 Hardware

The Digilent Zybo Z7-10 development board is well-suited for first prototyping an algorithm running in software and then off-loading sub-tasks for processing to custom circuits. It is based on a Xilinx Zynq 7010 SoC, a hybrid between a dual-core ARM A9 (processing system, PS) and Artix-7 based FPGA (programmable logic, PL). Low-latency, high-throughput coupling between PS and PL allows for solutions using both software implementation, where design-time is more important than performance, and hardware, where performance is critical. The board’s video input and output interfaces make prototyping easy.

The programming model for software usually makes use of programming languages that abstract from hardware particularities. While this offers increased portability and ways to apply automatic compiler optimizations, avoiding knowledge about the underlying hardware is not possible anymore, close to the performance limits.

FPGA design can make use of two different programming models. One being RTL description in VHDL/Verilog, the other being high level synthesis in C/C++. High level synthesis represents a somewhat similar programming model to software programming. However, for worthwhile improvements over software implementation of the same algorithm, one needs to have a good understanding of the underlying hardware architecture. Much more so than for software in general.

2.1 FPGA Architecture

Field programmable gate arrays (FPGA) are large arrays of configurable logic blocks (CLB), interconnect wires and input/output (I/O) pads. The CLB is made up of look-up tables (LUT) and flipflops (FF), in varying numbers depending on the exact FPGA architecture. This structure is generic enough to implement any algorithm. During programming, the LUTs are programmed to implement a certain logic function, and the FFs are programmed to pipeline the data flow synchronous to a clock signal. Interconnect is programmed to wire LUTs, FFs, input pads and output pads together, resulting in a custom hardware circuit implementing a certain algorithm.

Current FPGA architecture also include hard primitive blocks that specialize a certain function that would otherwise be too costly in terms of logic utilization or too slow in terms of throughput to implement in generic logic. For example, digital signal processing (DSP) blocks are available to implement a multiply-accumulate circuit with no generic logic utilization. These blocks are optimized enough to offer superior performance for the specific task. Another example is dual-port static RAM (BRAM), that offers higher capacities than RAM implemented in LUTs. These primitive blocks are by default automatically inferred for certain HDL constructs like the multiply operator (*) for DSP or array access for BlockRAM.

A LUT is a memory element that implements a truth-table. Depending on the exact architecture, each LUT has a specific number of inputs that address a location in the truth-table. The value stored at that address is the output of the function implemented. During programming, the truth-table is populated to implement the desired function. A LUT can also be thought of and used as a 2^N-memory, called distributed RAM. This is a fast memory type because it can be instantiated all over the FPGA fabric, local to the circuit that needs data from it.

An FF is a storage element that latches new data on its input when clock and clock enable conditions are true and permanently provides the stored data on its output.

A BRAM is a dual-port RAM that stores a larger set of data. It holds 18Kb or 36Kb and can be addressed independently over two ports for both read and write. In essence, two memory locations can be accessed simultaneously in the same clock cycle.

2.2 Parallelism and program execution

A processor core executes software instructions in a sequence. Higher-level programming languages translate language statements into assembly instructions that perform the function. Under this abstraction, the addition of two variables usually involves more than one instruction. Apart from the actual arithmetic operation that accesses internal registers, memory load and store instructions will also be needed. Performance improvements result in optimizing those memory accesses using caches. Each memory level trades access latency for storage capacity, so less and less data is available at memories of lower latencies. The job of the programmer and compiler is to ensure that for critical areas of an algorithm the spatial locality of data is high, and the data can be accessed with the lowest latency possible. It requires considerable effort and performance analysis tools to optimize code for execution time.

The FPGA is massively parallel by nature. Every LUT can execute a different function at the same time, so it is possible to have multiple arithmetic logic units (ALU) executing addition operations, for example, in parallel. On a processor, the ALU is shared and these operations would have to be executed sequentially. Memories can be instantiated close to where they are needed, resulting in high instantaneous memory bandwidth.

The role of high level synthesis tools is to extract the best possible circuit implementation from C/C++ code that is still functionally correct and meets requirements. It analyzes data dependencies determining which operations could and should execute in each clock cycle. Depending on the targeted clock frequency and FPGA, some operations might take more cycles to complete. This step is called **scheduling**. Next, the hardware resources that best implement the scheduled operation are determined. This is called **binding.** The last step in synthesis is the **control logic extraction** which creates a finite state machine that controls when the different operations are executed in the design. For multi-cycle operations **pipelining** is performed in the scheduling phase. Consider the following C statement:

```
x=a*b+c;
```

If the clock period is too small for the multiplication and addition operations to complete in one clock cycle, it will be scheduled for two cycles. For every set of inputs - a, b, and c - it takes two cycles to obtain the result. It follows that in cycle 2 the multiplier does not perform any operation; it only provides the result calculated in the previous cycle.

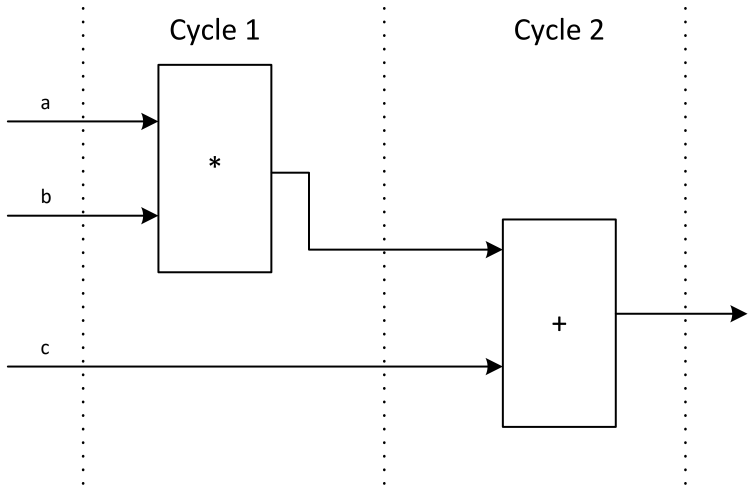


Figure 1 Hardware scheduling

This inefficiency becomes more apparent, when this statement is executed in a loop, i.e. the circuit processes more than one set of input data. If there was a storage element between cycles, the result from cycle 1 would be saved, and the multiplier would be free to perform a calculation for the next set of inputs. This concept is called pipelining and it is a major optimization opportunity increasing throughput tremendously.

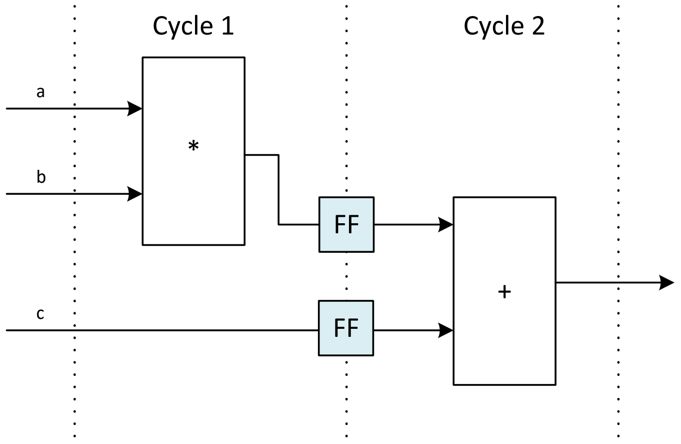


Figure 2 Hardware scheduling with pipe-lining

2.3 Performance metrics

The previous example provides a great opportunity to introduce some performance metrics.

Latency is the time it takes for the function to calculate all output values. The **latency** of the statement above is two, as it takes two cycles to compute the result.

The **initiation interval (II)** is the number of clock cycles it takes for the function to accept a new set of input values. In the first, non-pipelined, case the **initiation interval** is two, since it takes two cycles for the circuit to accept a new set of inputs. However, in the second pipelined case the II is just one, because the circuit can accept a new set of inputs and output a result in every cycle. The latency is still two, as the result for the first set of inputs will appear after two cycles.

The metrics above apply to an N-sized set of inputs too. If the circuit processes 10 sets of input data (N=10), the non-pipelined versions will have a total latency of 20 cycles ((N-1) * II + latency). The pipelined versions will only take 11 cycles ((N-1) * II + latency) to provide all the 10 results.

These performance metrics are calculated by the tools for both loops and functions and are considered the most important feedback mechanism for the designer to evaluate the synthesized hardware circuit.

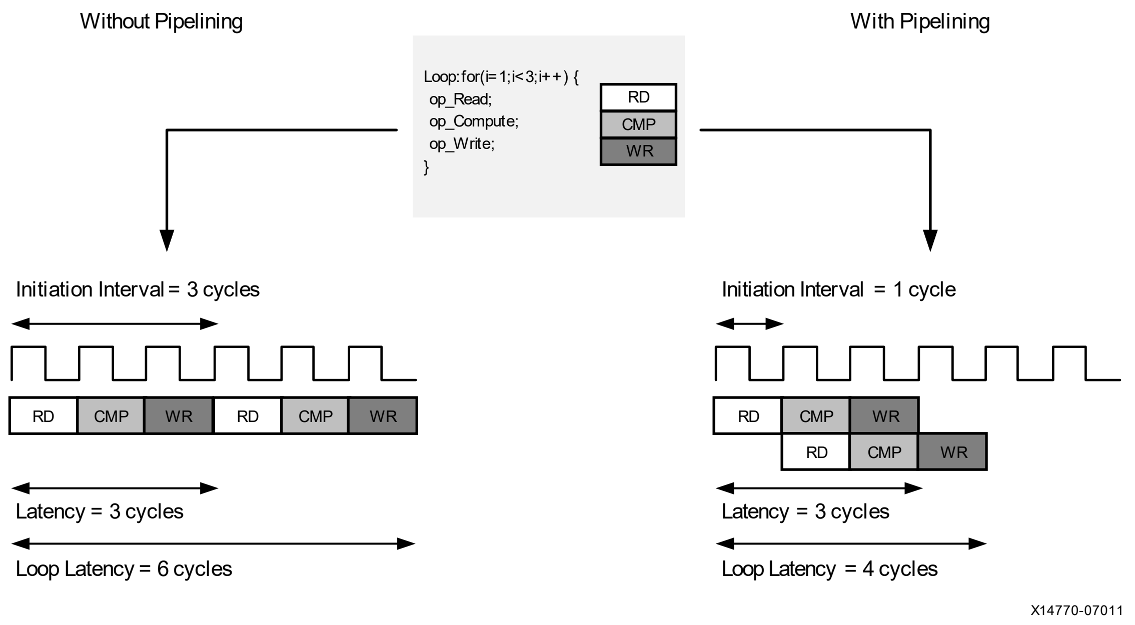
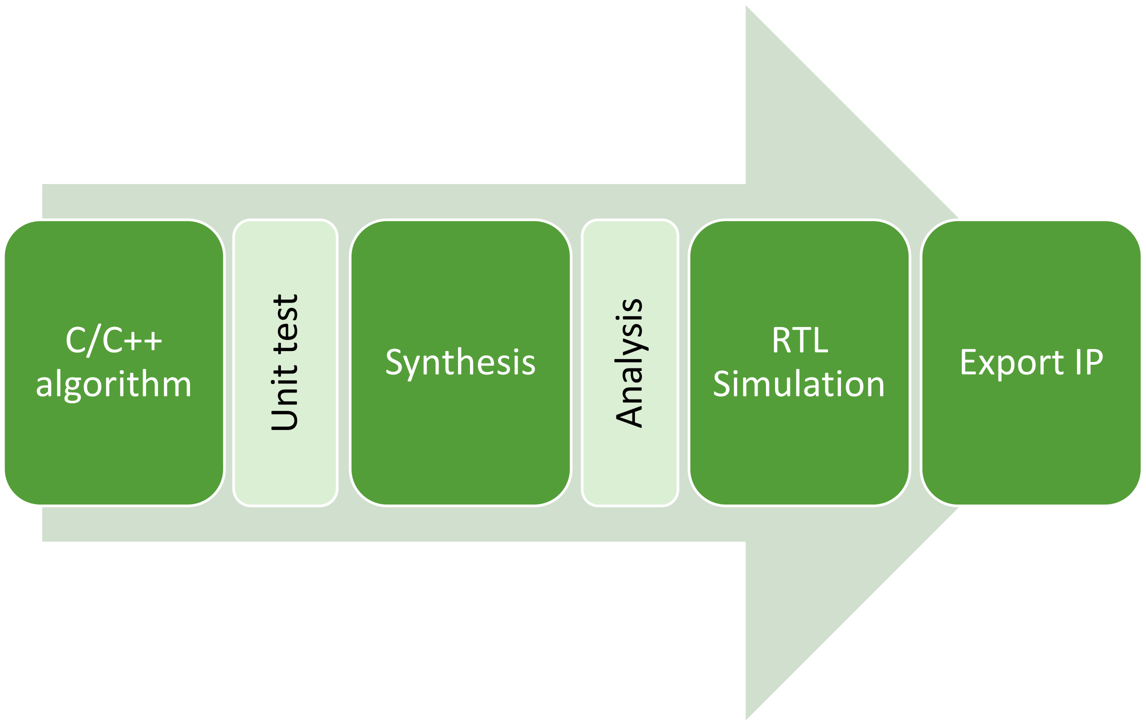


Figure 3 Loop pipelining (From Xilinx documentation)

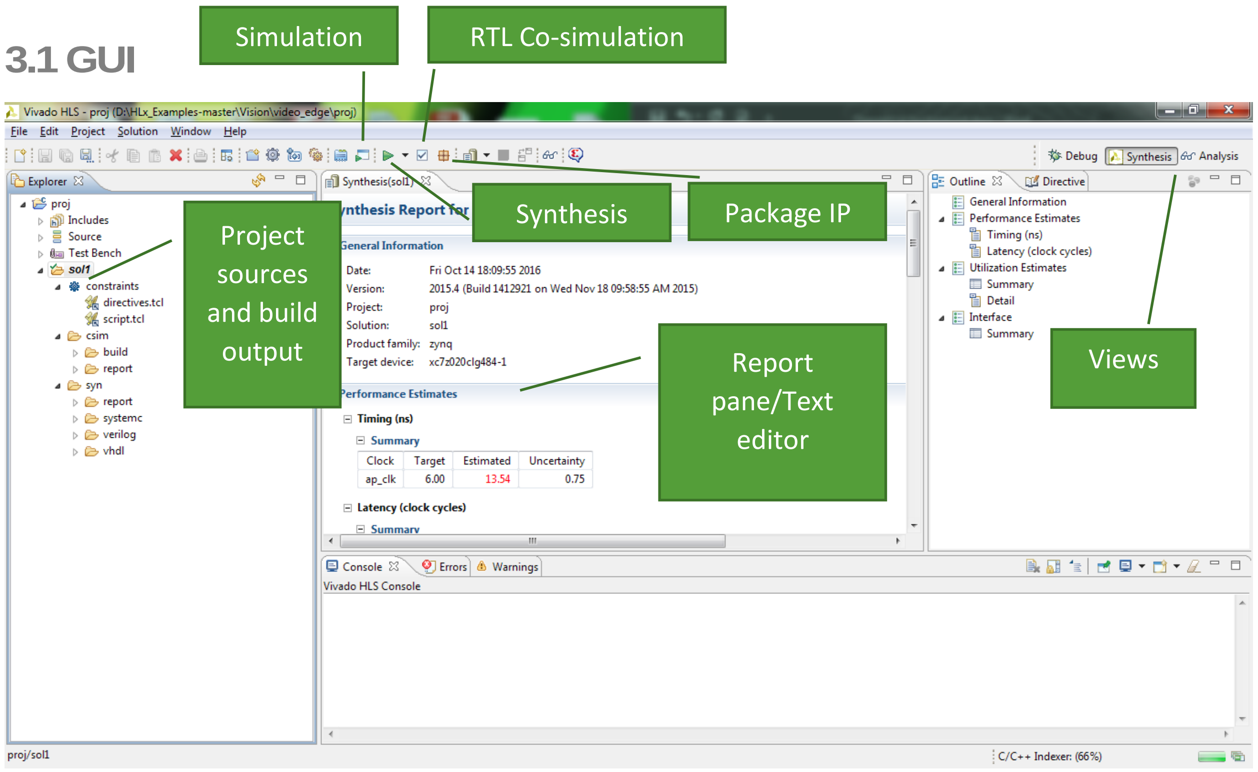
3 Vivado HLS

Xilinx's offering in high-level synthesis is part of the Vivado suite and is called Vivado HLS. The workflow is an iterative approach with simulations as verification steps inserted along the way to make sure the design meets the requirements and is functionally correct right from the initial stages. Vivado HLS can:

- Compile, execute and debug the C/C++ algorithm,
- Synthesize into RTL implementation,
- Provide analysis features,
- Generate and execute RTL simulation testbenches,
- Export the RTL implementation as an IP module.



3.1 GUI



ynthesis Report for

General Information

Date: Fri Oct 14 18:09:55 2016
Version: 2015.4 (Build 1412921 on Wed Nov 18 09:58:55 AM 2015)
Project: proj
Solution: sol1
Product family: zynq
Target device: xc7z020clg484-1

Performance Estimates

Timing (ns)

Summary

Clock	Target	Estimated	Uncertainty
ap_clk	6.00	13.54	0.75

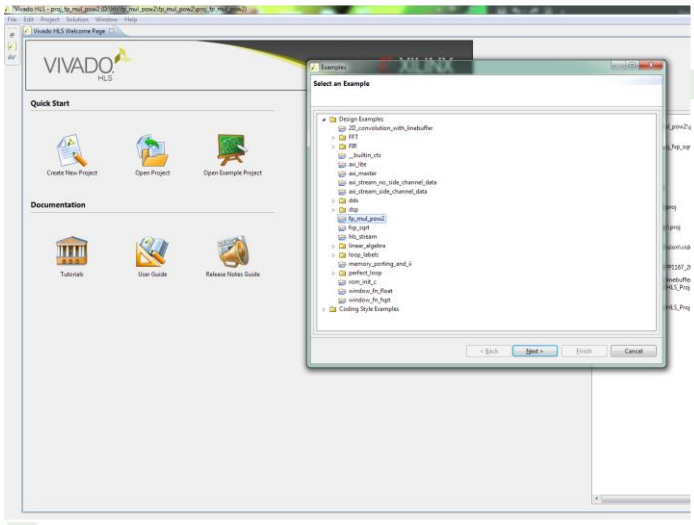
Latency (clock cycles)

Summary

The GUI layout is quite like other software IDEs. The project explorer lists the source, include and testbench files. Simulation and synthesis outputs are also visible here grouped into solutions. The workflow action buttons are in the toolbar ordered by their sequence in the workflow. In the upper right corner three layout views are available each fitting the current workflow step.

4 Task One – Getting familiar with the interface

Let us open an example project to get more familiar with the interface.



Select an Example

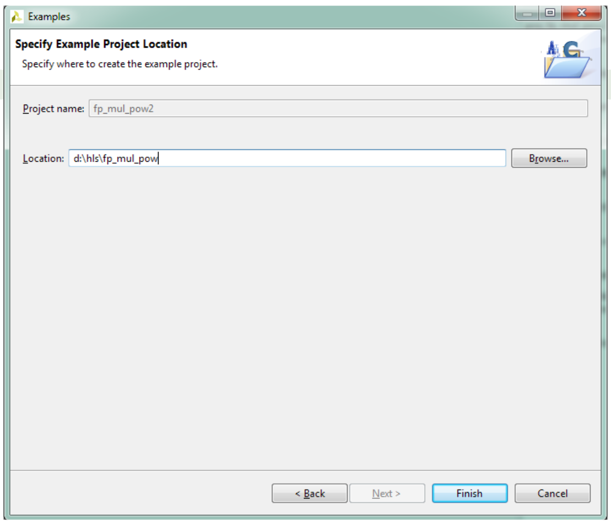
Open example project

Launch Vivado HLS 2018.1 from the Start Menu.

On Linux run vivado_hls from the shell.

Click the Open Example Project button on the Welcome Page.

Choose Design Examples/fp_mul_pow2 from the list of projects



Specify Example Project Location

Specify where to create the example project.

Project name: fp_mul_pow2

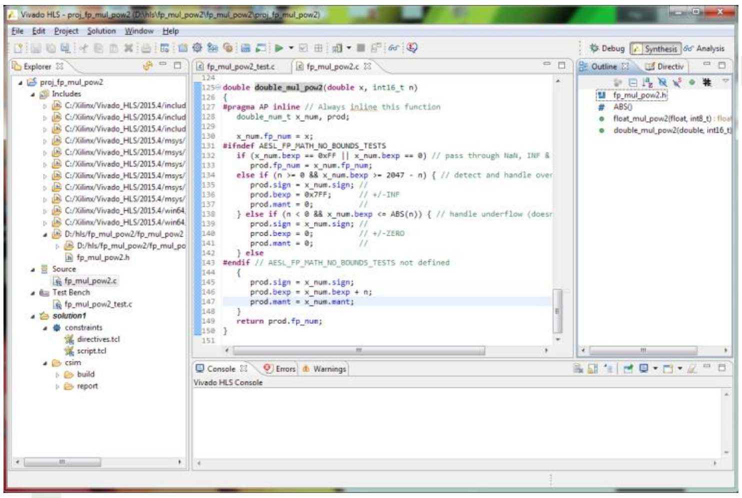
Location: d:\hls\fp_mul_pow

Save project

Browse to the location of your choice on your local storage drive

You may choose zyboz7_workshop/hls_project for location.

Click OK



```
double double_mul_pow2(double x, int16_t n)
{
#pragma AP inline // Always inline this function
   double_num_t x_num, prod;

   x_num.fp_num = x;
#ifndef AESL_FP_MATH_NO_BOUNDS_TESTS
   if (x_num.bexp == 0xFF || x_num.bexp == 0) // pass through NaN, INF &
      prod.fp_num = x_num.fp_num;
   else if (n >= 0 && x_num.bexp >= 2047 - n) { // detect and handle over
      prod.sign = x_num.sign; //
      prod.bexp = 0x7FF;      // +/-INF
      prod.mant = 0;          //
   } else if (n < 0 && x_num.bexp <= ABS(n)) { // handle underflow (doesr
      prod.sign = x_num.sign; //
      prod.bexp = 0;          // +/-ZERO
      prod.mant = 0;          //
   } else
#endif // AESL_FP_MATH_NO_BOUNDS_TESTS not defined
   {
      prod.sign = x_num.sign;
      prod.bexp = x_num.bexp + n;
      prod.mant = x_num.mant;
   }
   return prod.fp_num;
}
```

Analyze project structure

Open fp_mul_pow2.h below Includes.

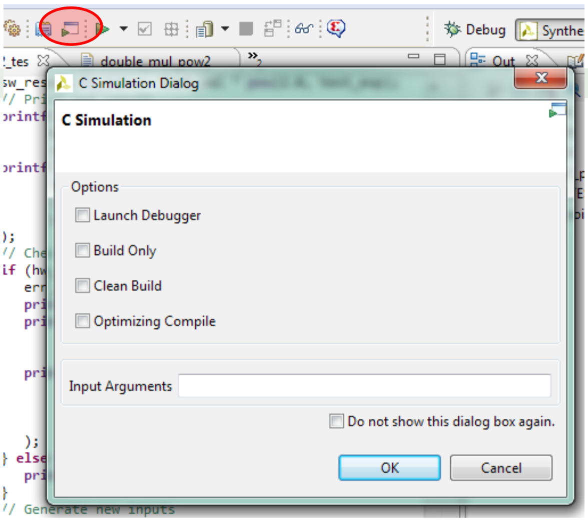
Open fp_mul_pow2.c below Source.

Open fp_mul_pow2_test.c below Test Bench.

Notice that we are in the Synthesis view (upper right corner).

A Vivado HLS project is much like any other C/C++ software project. There is a source file defining two functions, a header file declaring the functions and some data types. There is also a test bench source file, which is a regular application with a main entry point that runs test on the functions, validating them on functional correctness. Test benches are used for C simulation, which is the first validation step in the design process. The successfulness of C simulation is determined by the return value of the test bench. It is expected to return 0 for a success, and any non-zero value for failure.

Discuss the implementation of the double_mul_pow2 function and the test bench.



C Simulation Dialog

C Simulation

Options

- [ ] Launch Debugger
- [ ] Build Only
- [ ] Clean Build
- [ ] Optimizing Compile

Input Arguments

- [ ] Do not show this dialog box again.

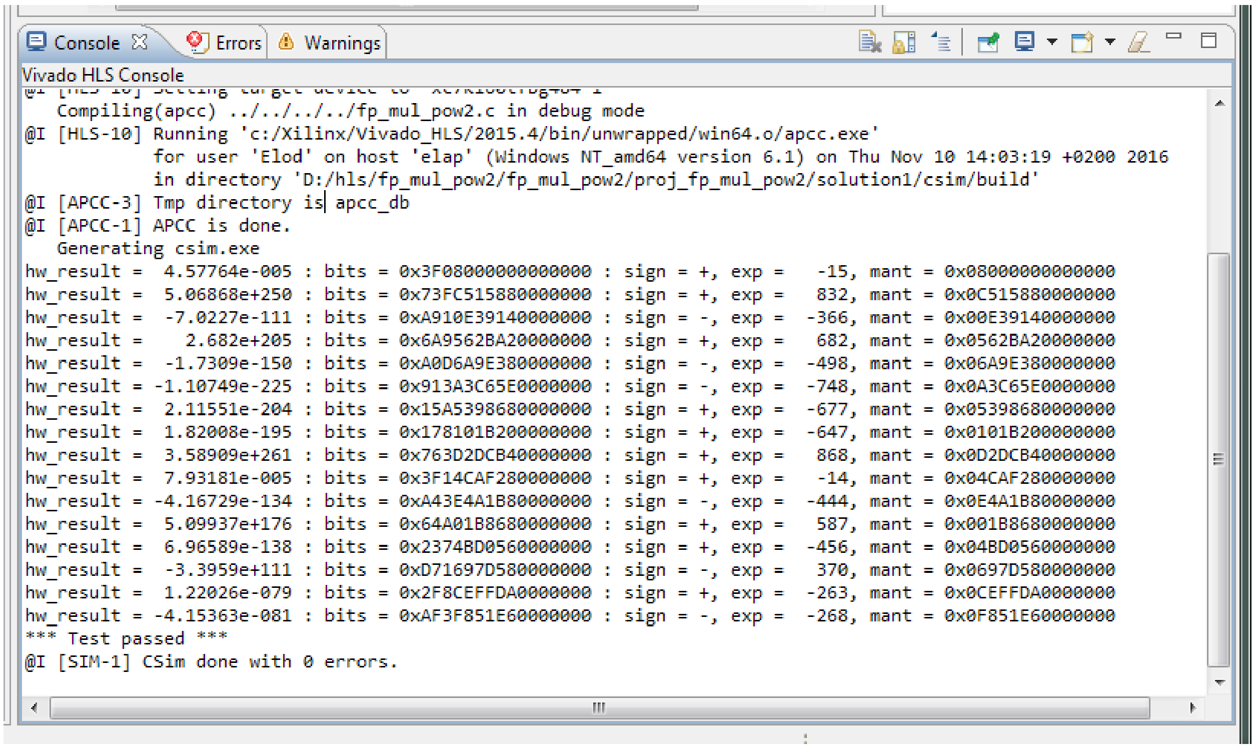
OK | Cancel

Run C simulation

Click the Run C Simulation button on the toolbar.

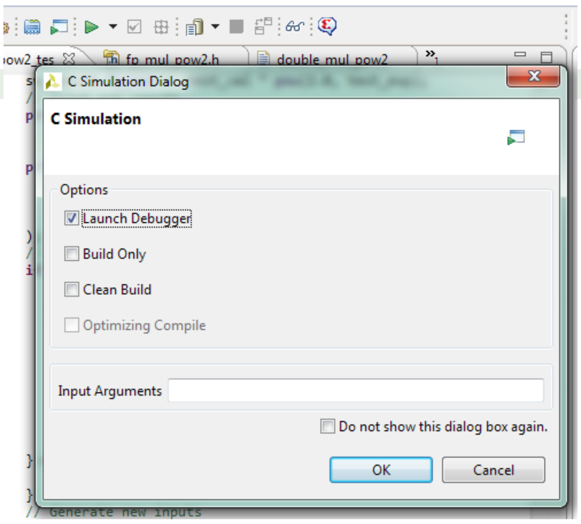
Leave simulation options at their default values and click OK

Discuss the results of the C simulation and the messages shown in the console.



```
   Compiling(apcc) ../../../../fp_mul_pow2.c in debug mode
@I [HLS-10] Running 'c:/Xilinx/Vivado_HLS/2015.4/bin/unwrapped/win64.o/apcc.exe'
            for user 'Elod' on host 'elap' (Windows NT_amd64 version 6.1) on Thu Nov 10 14:03:19 +0200 2016
            in directory 'D:/hls/fp_mul_pow2/fp_mul_pow2/proj_fp_mul_pow2/solution1/csim/build'
@I [APCC-3] Tmp directory is apcc_db
@I [APCC-1] APCC is done.
   Generating csim.exe
hw_result =  4.57764e-005 : bits = 0x3F08000000000000 : sign = +, exp =  -15, mant = 0x08000000000000
hw_result =  5.06868e+250 : bits = 0x73FC515880000000 : sign = +, exp =  832, mant = 0x0C515880000000
hw_result =  -7.0227e-111 : bits = 0xA910E39140000000 : sign = -, exp = -366, mant = 0x00E39140000000
hw_result =    2.682e+205 : bits = 0x6A9562BA20000000 : sign = +, exp =  682, mant = 0x0562BA20000000
hw_result =   -1.7309e-150 : bits = 0xA0D6A9E380000000 : sign = -, exp = -498, mant = 0x06A9E380000000
hw_result = -1.10749e-225 : bits = 0x913A3C65E0000000 : sign = -, exp = -748, mant = 0x0A3C65E0000000
hw_result =  2.11551e-204 : bits = 0x15A5398680000000 : sign = +, exp = -677, mant = 0x05398680000000
hw_result =  1.82008e-195 : bits = 0x178101B200000000 : sign = +, exp = -647, mant = 0x0101B200000000
hw_result =  3.58909e+261 : bits = 0x763D2DCB40000000 : sign = +, exp =  868, mant = 0x0D2DCB40000000
hw_result =  7.93181e-005 : bits = 0x3F14CAF280000000 : sign = +, exp =  -14, mant = 0x04CAF280000000
hw_result = -4.16729e-134 : bits = 0xA43E4A1B80000000 : sign = -, exp = -444, mant = 0x0E4A1B80000000
hw_result =  5.09937e+176 : bits = 0x64A01B8680000000 : sign = +, exp =  587, mant = 0x001B8680000000
hw_result =  6.96589e-138 : bits = 0x2374BD0560000000 : sign = +, exp = -456, mant = 0x04BD0560000000
hw_result =  -3.3959e+111 : bits = 0xD71697D580000000 : sign = -, exp =  370, mant = 0x0697D580000000
hw_result =  1.22026e-079 : bits = 0x2F8CEFFDA0000000 : sign = +, exp = -263, mant = 0x0CEFFDA0000000
hw_result = -4.15363e-081 : bits = 0xAF3F851E60000000 : sign = -, exp = -268, mant = 0x0F851E60000000
*** Test passed ***
@I [SIM-1] CSim done with 0 errors.
```

Figure 4 C simulation results in console

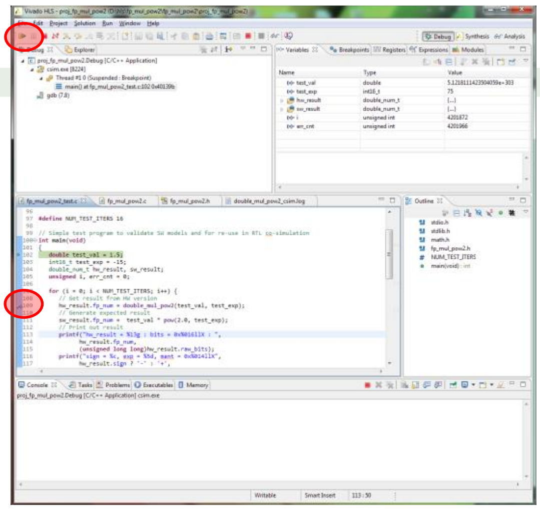


Run C simulation with debugger

Click the Run C Simulation button on the toolbar.

Check the Launch Debugger option and click OK

Notice how the Debug view gets activated, the test bench started and stop at the first instruction of the main function. The test bench can be run step-by-step, breakpoints set, variables and expressions evaluated just like any other software project.



```
proj_fp_mul_pow2.Debug [C/C++ Application]
  csim.exe [8224]
    Thread #1 0 (Suspended : Breakpoint)
      main() at fp_mul_pow2_test.c:102 0x40139b
  gdb (7.8)
```

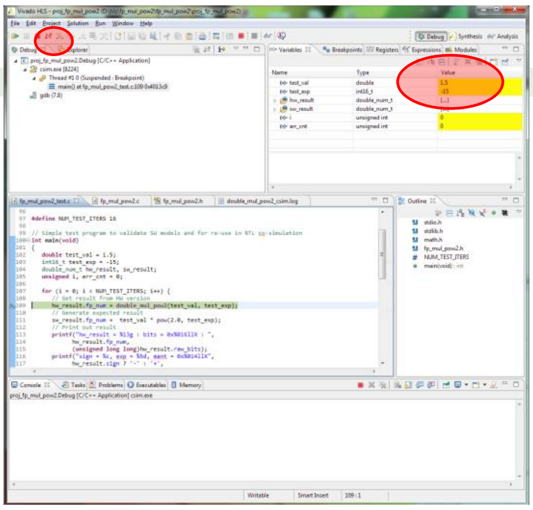
Name	Type	Value
test_val	double	5.1218111423504059e+303
test_exp	int16_t	75
hw_result	double_num_t	[...]
sw_result	double_num_t	[...]
i	unsigned int	4201872
err_cnt	unsigned int	4201966

```
96
97  #define NUM_TEST_ITERS 16
98
99  // Simple test program to validate SW models and for re-use in RTL co-simulation
100 int main(void)
101 {
102    double test_val = 1.5;
103    int16_t test_exp = -15;
104    double_num_t hw_result, sw_result;
105    unsigned i, err_cnt = 0;
106
107    for (i = 0; i < NUM_TEST_ITERS; i++) {
108       // Get result from HW version
109       hw_result.fp_num = double_mul_pow2(test_val, test_exp);
110       // Generate expected result
111       sw_result.fp_num =  test_val * pow(2.0, test_exp);
112       // Print out result
113       printf("hw_result = %13g : bits = 0x%016llX : ",
114             hw_result.fp_num,
115             (unsigned long long)hw_result.raw_bits);
116       printf("sign = %c, exp = %5d, mant = 0x%014llX",
117             hw_result.sign ? '-' : '+',
```

Debug test bench

Double click on the blue column in line 109 to place a breakpoint at the line that calls the double_mul_pow2 function.

Click the Resume button in the toolbar to run the test bench until the breakpoint is hit



```
proj_fp_mul_pow2.Debug [C/C++ Application]
  csim.exe [8224]
    Thread #1 0 (Suspended : Breakpoint)
      main() at fp_mul_pow2_test.c:109 0x4013c9
  gdb (7.8)
```

Name	Type	Value
test_val	double	1.5
test_exp	int16_t	-15
hw_result	double_num_t	[...]
sw_result	double_num_t	[...]
i	unsigned int	0
err_cnt	unsigned int	0

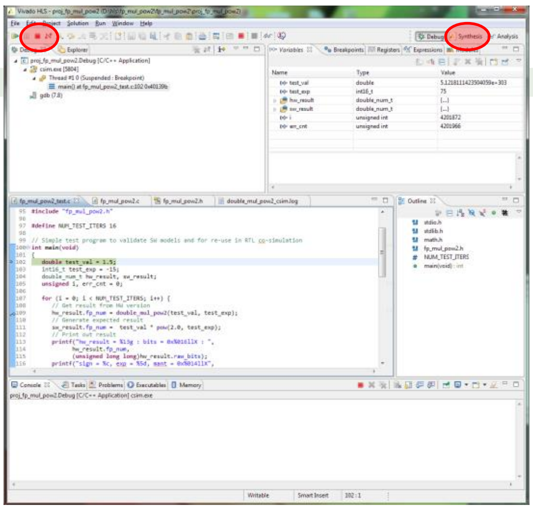
```
96
97  #define NUM_TEST_ITERS 16
98
99  // Simple test program to validate SW models and for re-use in RTL co-simulation
100 int main(void)
101 {
102    double test_val = 1.5;
103    int16_t test_exp = -15;
104    double_num_t hw_result, sw_result;
105    unsigned i, err_cnt = 0;
106
107    for (i = 0; i < NUM_TEST_ITERS; i++) {
108       // Get result from HW version
109       hw_result.fp_num = double_mul_pow2(test_val, test_exp);
110       // Generate expected result
111       sw_result.fp_num =  test_val * pow(2.0, test_exp);
112       // Print out result
113       printf("hw_result = %13g : bits = 0x%016llX : ",
114             hw_result.fp_num,
115             (unsigned long long)hw_result.raw_bits);
116       printf("sign = %c, exp = %5d, mant = 0x%014llX",
117             hw_result.sign ? '-' : '+',
```

Step into the double_mul_pow2 function

Notice how the variables test_val and test_exp changed before the breakpoint was hit.

Click the Step Into button in the toolbar or press F5 on your keyboard.

Keep pressing F6 to execute the function statement-by-statement.



Name	Type	Value
test_val	double	5.1218111423504059e+303
test_exp	int16_t	75
hw_result	double_num_t	{...}
sw_result	double_num_t	{...}
i	unsigned int	4201872
err_cnt	unsigned int	4201966

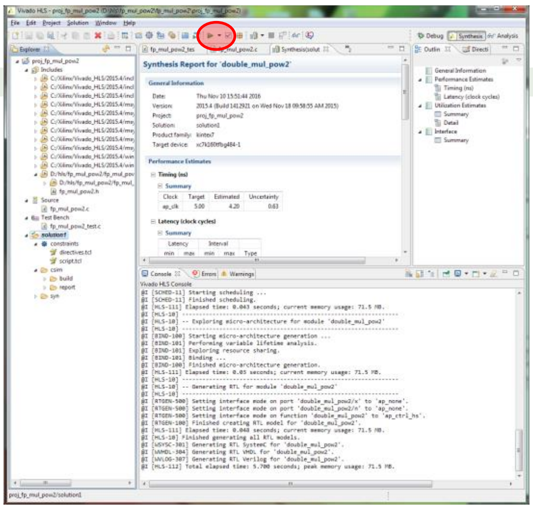
```
#include "fp_mul_pow2.h"

#define NUM_TEST_ITERS 16

// Simple test program to validate SW models and for re-use in RTL co-simulation
int main(void)
{
   double test_val = 1.5;
   int16_t test_exp = -15;
   double_num_t hw_result, sw_result;
   unsigned i, err_cnt = 0;

   for (i = 0; i < NUM_TEST_ITERS; i++) {
      // Get result from HW version
      hw_result.fp_num = double_mul_pow2(test_val, test_exp);
      // Generate expected result
      sw_result.fp_num =  test_val * pow(2.0, test_exp);
      // Print out result
      printf("hw_result = %13g : bits = 0x%016llX : ",
            hw_result.fp_num,
            (unsigned long long)hw_result.raw_bits);
      printf("sign = %c, exp = %5d, mant = 0x%013llX",
```

Exit the debugger

Stop the debugger

Go back to Synthesis view.

Notice how solution1 in the project view has a csim folder now. Synthesis directives and simulation/synthesis results are grouped into solutions. Having multiple solutions allows us to try different settings, devices, clock periods on the same set of source files and analyze the results for each.



Synthesis Report for 'double_mul_pow2'

General Information

Date:	Thu Nov 10 15:51:44 2016
Version:	2015.4 (Build 1412921 on Wed Nov 18 09:58:55 AM 2015)
Project:	proj_fp_mul_pow2
Solution:	solution1
Product family:	kintex7
Target device:	xc7k160tfbg484-1

Performance Estimates

Timing (ns)

Summary

Clock	Target	Estimated	Uncertainty
ap_clk	5.00	4.20	0.63

Latency (clock cycles)

Summary

Latency		Interval		
min	max	min	max	Type

Vivado HLS Console

```
@I [SCHED-11] Starting scheduling ...
@I [SCHED-11] Finished scheduling.
@I [HLS-111] Elapsed time: 0.043 seconds; current memory usage: 71.5 MB.
@I [HLS-10] ----------------------------------------------------------------
@I [HLS-10] -- Exploring micro-architecture for module 'double_mul_pow2'
@I [HLS-10] ----------------------------------------------------------------
@I [BIND-100] Starting micro-architecture generation ...
@I [BIND-101] Performing variable lifetime analysis.
@I [BIND-101] Exploring resource sharing.
@I [BIND-101] Binding ...
@I [BIND-100] Finished micro-architecture generation.
@I [HLS-111] Elapsed time: 0.05 seconds; current memory usage: 71.5 MB.
@I [HLS-10] ----------------------------------------------------------------
@I [HLS-10] -- Generating RTL for module 'double_mul_pow2'
@I [HLS-10] ----------------------------------------------------------------
@I [RTGEN-500] Setting interface mode on port 'double_mul_pow2/x' to 'ap_none'.
@I [RTGEN-500] Setting interface mode on port 'double_mul_pow2/n' to 'ap_none'.
@I [RTGEN-500] Setting interface mode on function 'double_mul_pow2' to 'ap_ctrl_hs'.
@I [RTGEN-100] Finished creating RTL model for 'double_mul_pow2'.
@I [HLS-111] Elapsed time: 0.048 seconds; current memory usage: 71.5 MB.
@I [HLS-10] Finished generating all RTL models.
@I [WSYSC-301] Generating RTL SystemC for 'double_mul_pow2'.
@I [WVHDL-304] Generating RTL VHDL for 'double_mul_pow2'.
@I [WVLOG-307] Generating RTL Verilog for 'double_mul_pow2'.
@I [HLS-112] Total elapsed time: 5.700 seconds; peak memory usage: 71.5 MB.
```

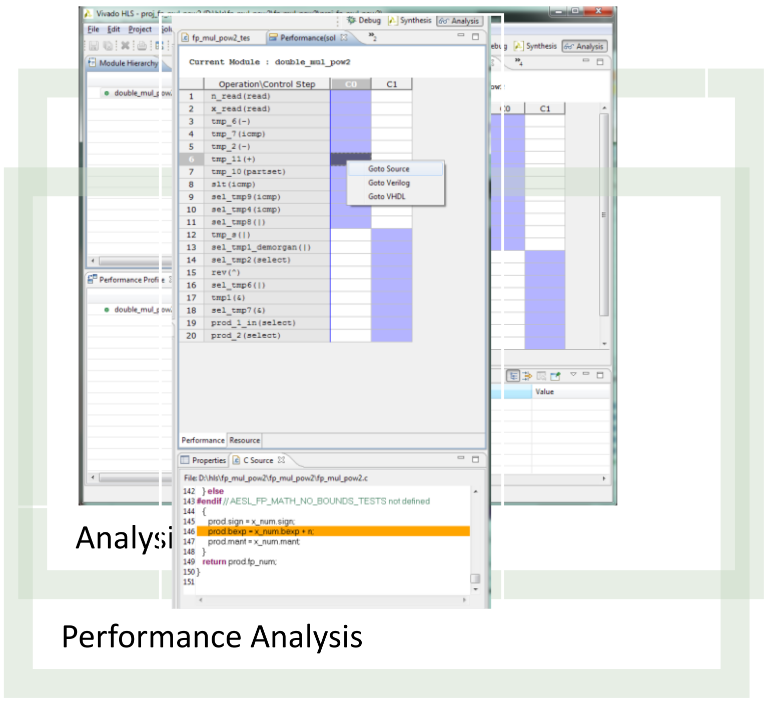
Synthesis

Synthesize the design by clicking the C Synthesis button in the toolbar.

Watch the messages in the console until synthesis completes.

Notice the new syn folder in solution1 and the Synthesis Report that opened automatically.

Discuss the report. What did HLS synthesize? What are the latency and interval values? What are the interfaces that got generated?



Current Module : double_mul_pow2

	Operation\Control Step	C0	C1
1	n_read(read)	▨	
2	x_read(read)	▨	
3	tmp_6(-)	▨	
4	tmp_7(icmp)	▨	
5	tmp_2(-)	▨	
6	tmp_11(+)	▨	
7	tmp_10(partset)	▨	
8	slt(icmp)	▨	
9	sel_tmp9(icmp)	▨	
10	sel_tmp4(icmp)	▨	
11	sel_tmp8(|)	▨	
12	tmp_s(|)		▨
13	sel_tmp1_demorgan(|)		▨
14	sel_tmp2(select)		▨
15	rev(^)		▨
16	sel_tmp6(|)		▨
17	tmp1(&)		▨
18	sel_tmp7(&)		▨
19	prod_1_in(select)		▨
20	prod_2(select)		▨

File: D:\hls\fp_mul_pow2\fp_mul_pow2\fp_mul_pow2.c

```
  } else
#endif // AESL_FP_MATH_NO_BOUNDS_TESTS not defined
  {
    prod.sign = x_num.sign;
    prod.bexp = x_num.bexp + n;
    prod.mant = x_num.mant;
  }
  return prod.fp_num;
}
```

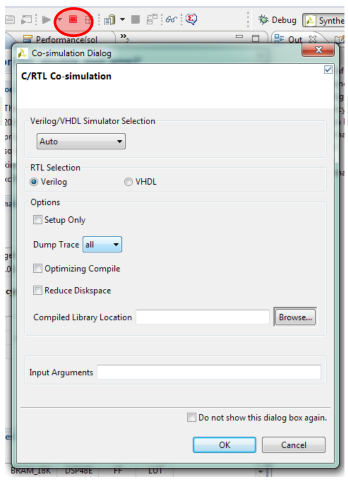
Open the Analysis view.

Right-click the purple cell in column C0 and row #6, operation tmp_10(+).

Choose Goto Source.

The Analysis view helps in understanding and evaluating the synthesized design. The synthesized modules and loops can be seen on the left, along with timing and logic utilization information. In this case double_mul_pow2 does not have any sub-blocks, it is a flat function. Selecting an item will bring up the Performance view on the right. This shows the control states of the logic (C0, C1) and each operation that is scheduled to execute in that state.

When the synthesized design satisfies all the project requirements, the next step is running an RTL simulation to verify that it is functionally correct. In Vivado HLS terminology this is called C/RTL Cosimulation. Vivado HLS is capable of automatically generating an RTL test bench by running the C test bench and using the inputs from there as stimuli and the outputs as expected values.



Co-simulation Dialog

C/RTL Co-simulation

Verilog/VHDL Simulator Selection: Auto

RTL Selection: [✓ Verilog] VHDL

Options

- ☐ Setup Only
- Dump Trace: all
- ☐ Optimizing Compile
- ☐ Reduce Diskspace
- Compiled Library Location: Browse...

Input Arguments

☐ Do not show this dialog box again.

C/RTL Cosimulation

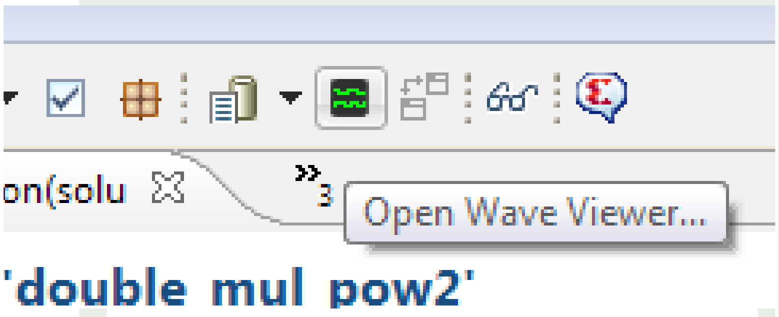
Click on the C/RTL Cosimulation button on the toolbar

Choose "all" for the Dump Trace option

Click OK.

Review the messages in Console

The Dump Trace option will export the RTL simulation waveforms that can be opened in Vivado Simulator, for example.



View simulation waveforms

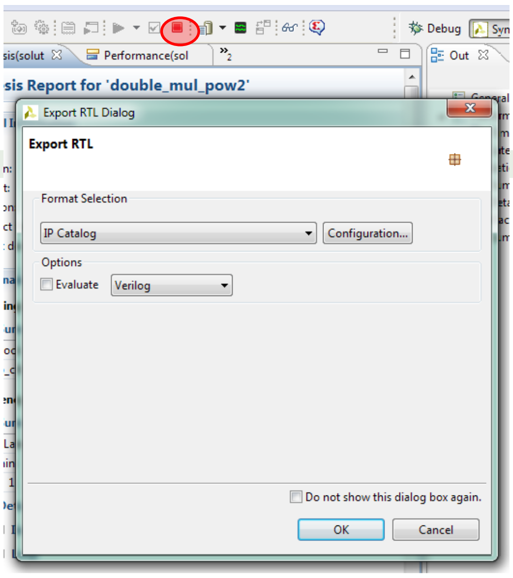
Click on the Open Wave Viewer button on the toolbar

Wait for Vivado to open

Open the Window menu and go to Waveform

Analyze the simulation waveforms. Look for input values, results. Measure latencies and initiation intervals.

Since the hardware is now validated, all that is left is to package it up into a reusable format.



Package IP

Click on Export RTL button on the toolbar

Leave options on their defaults

Click OK

Wait for export to complete

The exported IP files are generated in the active solution folder under impl. Locate the files and explore the sub-folders.

This concludes our first task – Getting familiar with the interface.

5 Task Two – Create a pass-through video pipeline

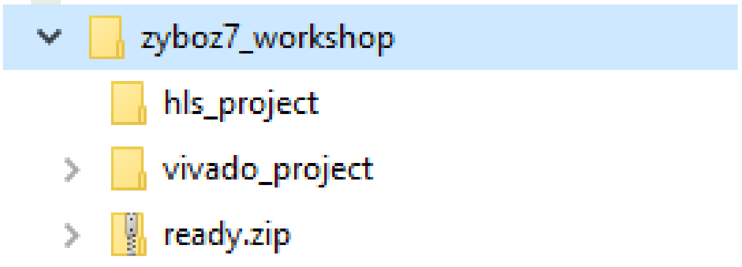
Vivado HLS creates re-usable IP blocks that need a top-level application to integrate into. In our case the application is a hardware project that:

- configures the camera for the right mode and resolution,
- decodes video data from a MIPI CSI-2 interface,
- buffers video data in memory, and
- encodes video data to a DVI interface.

Such an application provides a base infrastructure typical for video processing applications: it configures both input and output paths, uses a video frame buffer to decouple those paths and adds additional control logic.

Usually the focus is not on the exact video interfaces used or how video is buffered in memory. To allow us to focus on the video processing algorithm, we use ready-made IP blocks have the infrastructure quickly up and running. Even assembling and wiring these blocks can get complex, so it helps to start from a known good design.

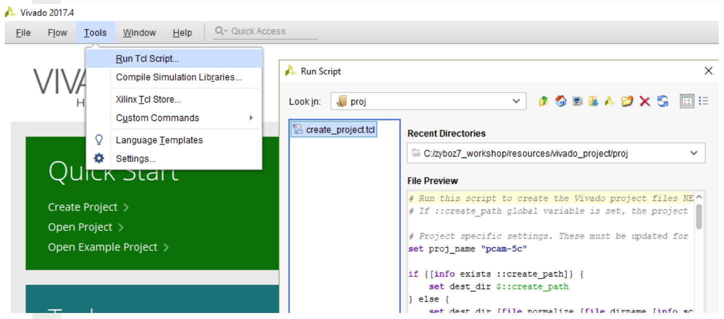
In this task we will be looking at a demonstrative project of the Pcam 5C passing video through the Zybo Z7 and outputting on a monitor. It will serve as the top-level application where the HLS IP created in the next task will be inserted. It is a Vivado block design project using Digilent and Xilinx IP and can be found among the workshop resources. The original and most up-to-date version is also available here: https://github.com/Digilent/Zybo-Z7-20-pcam-5c. This workshop uses a slightly modified version of the original: re-targeted for the Z7-10, IP blocks renamed and grouped for readability, default resolution changed to 720p and project files generated for Vivado 2018.1.



Create project tree

Copy the folder called "zyboz7_workshop" to the root of your local hard drive.

If you choose a location other than root, make sure the path has no spaces in it. Take note of the path as you will need it later.



C:/zyboz7_workshop/resources/vivado_project/proj

```
# Run this script to create the Vivado project files NE
# If ::create_path global variable is set, the project

# Project specific settings. These must be updated for
set proj_name "pcam-5c"

if {[info exists ::create_path]} {
    set dest_dir $::create_path
} else {
```

Open application

Launch Vivado 2018.1 (NOT Vivado HLS) from the Start Menu

Click Tools, Run Tcl Script.

Choose C:/zyboz7_workshop/vivado_project/proj/create_project.tcl

Click OK

Wait for the project to generate and open

Let us now look at what this project is made of.

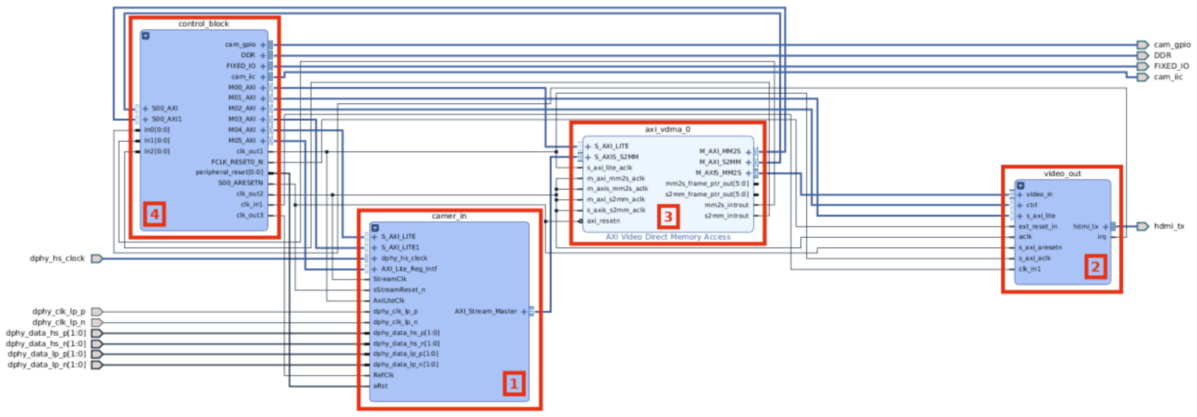


Figure 5 Top-level block diagram

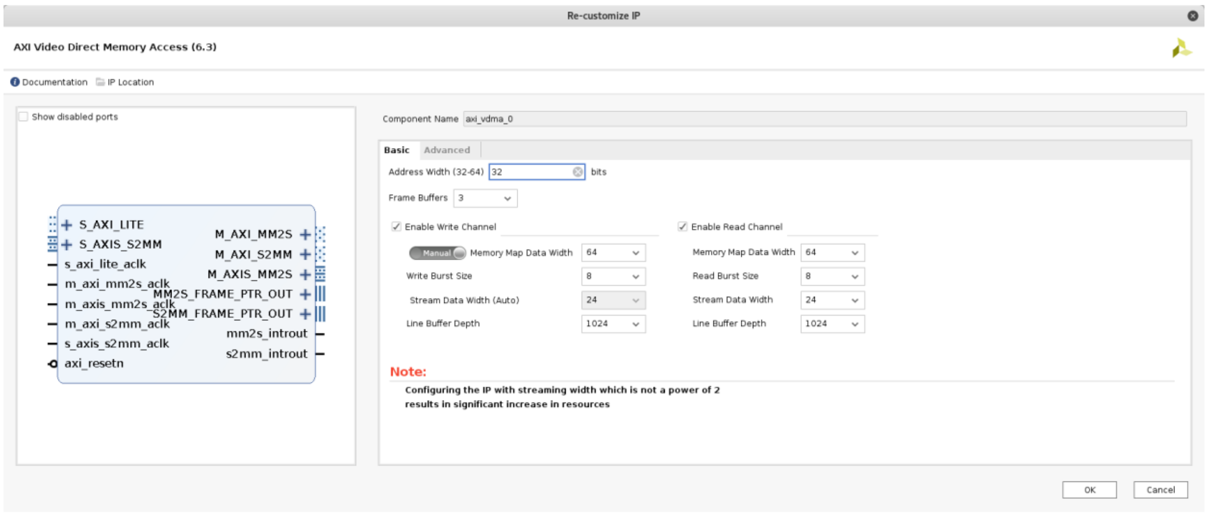
The light blue block (3) is a regular IP, while blue blocks (1,2, and 4) are hierarchy blocks, grouping IP block together.

Block no. 1, named camera_in, is the original data producer. It groups together IP blocks needed to decode image data coming from the camera and format it to suit our needs.

Block no. 2, named video_out, is the ultimate data consumer. It groups IP blocks doing DVI encoding so that the image data can be displayed on a monitor. We are going to look at these two hierarchy blocks later.

Block no. 3 is an actual IP, named axi_vdma. It is a Xilinx IP with the full name AXI Video Direct Memory Access. The VDMA sits in the middle of the video data flow and its central role makes it an interesting addition. It is needed to decouple two incompatible video interfaces, the image sensor's MIPI CSI-2 and the monitor's DVI. These two interfaces work with diverse clock frequencies, timing and perhaps different resolutions. In our case, the sensor is capable of outputting at 30 frames per second, but the monitor will need 60 frames per second for a standard resolution. The VDMA assumes the slave role, consuming data on its input side (S2MM) and writing it to DDR memory. It also assumes the master role, producing data on its output side (MM2S) by reading it from DDR memory beforehand. This allows both the sensor and the monitor to work at their own pace.

Double-clicking the IP block will open its customization settings. See what options are available.


Re-customize IP

AXI Video Direct Memory Access (6.3)

Documentation IP Location

Show disabled ports

Component Name: axi_vdma_0

Basic | Advanced

Address Width (32-64): 32 bits

Frame Buffers: 3

✓ Enable Write Channel
- Manual — Memory Map Data Width: 64
- Write Burst Size: 8
- Stream Data Width (Auto): 24
- Line Buffer Depth: 1024

✓ Enable Read Channel
- Memory Map Data Width: 64
- Read Burst Size: 8
- Stream Data Width: 24
- Line Buffer Depth: 1024

Note:
Configuring the IP with streaming width which is not a power of 2 results in significant increase in resources

OK Cancel

Figure 6 axi_vdma customization settings

Now close the axi_vdma customization window.

Let us turn our attention to the camera_in hierarchy block. Double-clicking it will open the hierarchy block in a separate tab.

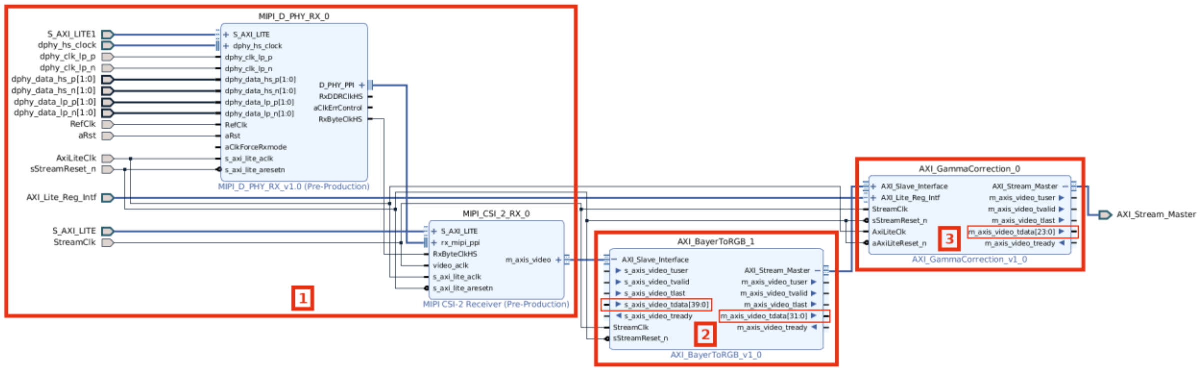


Figure 7 Camera_in hierarchy block

The two blocks marked 1 are the implementation of the MIPI interfaces. D-PHY is the physical layer protocol, that is transparent to the nature of data transmitted. CSI-2 is the Camera Serial Interface that is the packet layer protocol, specifying how pixel data is packetized.

Block no. 2 converts the Bayer-filtered pixel data into RGB data by interpolating missing color components from neighbours. After interpolation the 10-bit RAW data produced by the sensor is widened to 30 bits for RGB data.

Block no. 3 applies a gamma correction algorithm resulting in a 24-bit RGB-encoded pixel stream.

Discuss the interfaces and timing between these blocks.

Let us go back to the top level block diagram and double-click the video_out hierarchy block now.

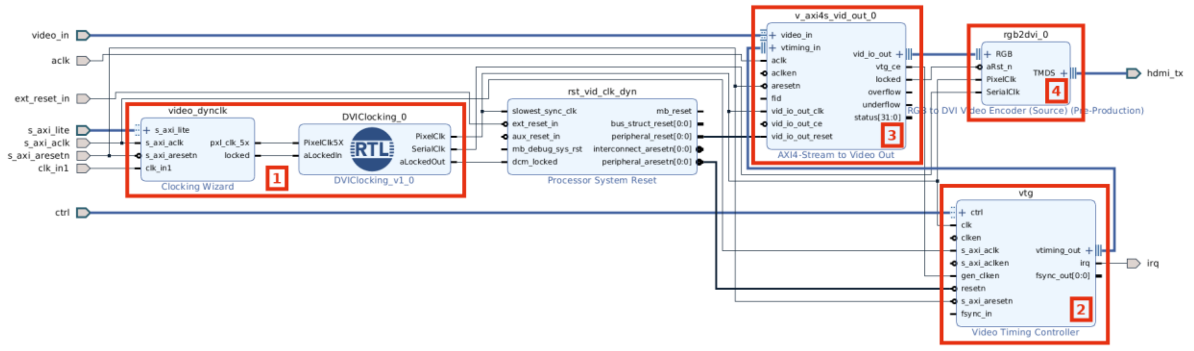


Figure 8 Expanded video_out hierarchy block

Blocks marked no. 1 are part of the clocking infrastructure. These dynamically reconfigurable blocks generate and buffer the pixel clock of the frequency correct for the display resolution. PixelClk and SerialClk are used by block no. 4 for DVI encoding.

Block no. 2 is a Video Timing Controller that can be dynamically configured to generate the horizontal and vertical synchronization signals for the display resolution.

Block no. 3 is where AXI-Stream formatted pixel data is synchronized with the display timing pulses generated by the Video Timing Controller.

The now pixel-clocked and synchronized video data is sent to Block no. 4, named rgb2dvi. This implements DVI encoding and is wired straight to the Zybo-Z7's HDMI connector. This is a Digilent IP and is available in our vivado-library repository.

Discuss the interfaces and timing between these blocks.

Let us now go back to the last hierarchy block in our top-level design, control_block. Open it by double-click.

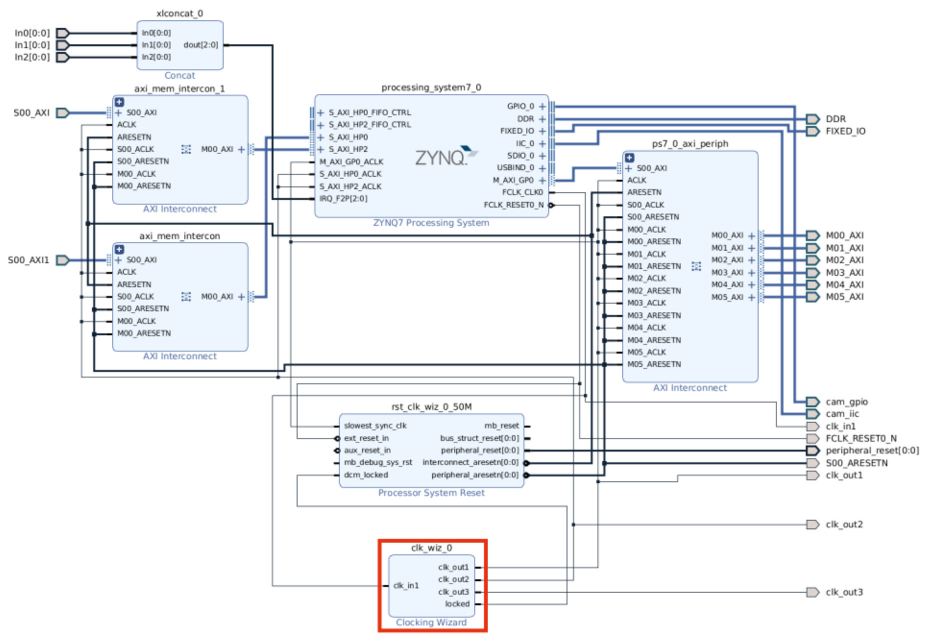
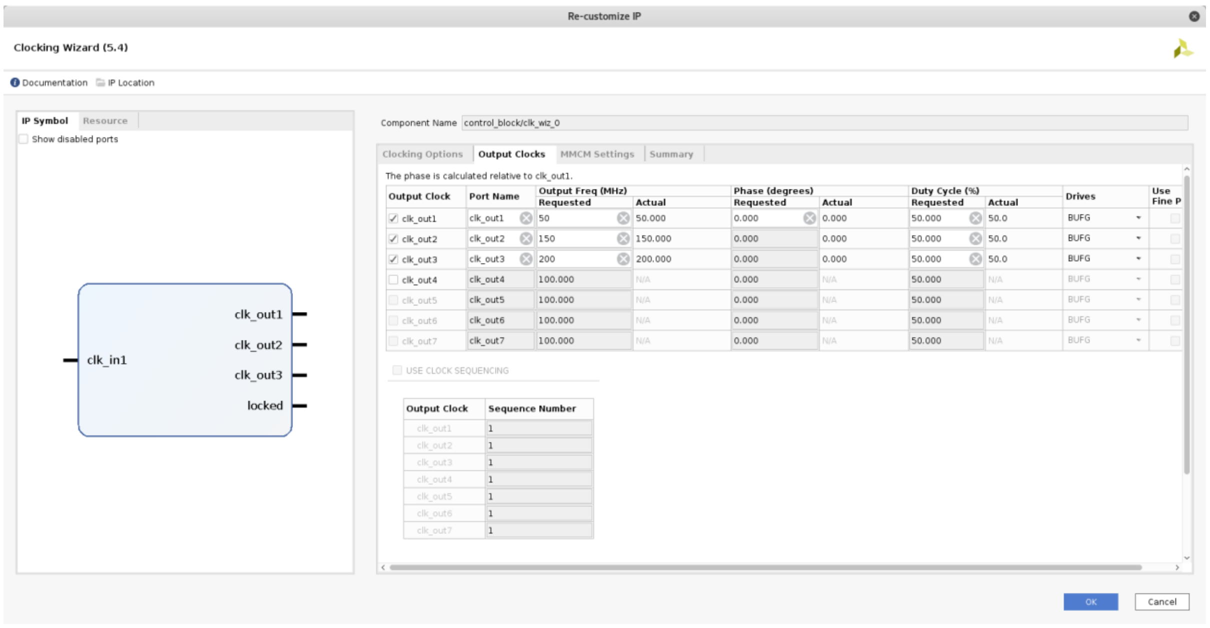


Figure 9 Expanded control_block hierarchy

Every non-trivial system will need control logic. In our application, the dual-core ARM in the Zynq chip will run our control software, and its associated blocks will give us access to main memory and control the sensor over I^{2}C.

Double-click the Clocking Wizard IP block to see the generated clock signals.


Re-customize IP

Clocking Wizard (5.4)

Documentation IP Location

IP Symbol Resource

Show disabled ports

Component Name control_block/clk_wiz_0

Clocking Options Output Clocks MMCM Settings Summary

The phase is calculated relative to clk_out1.

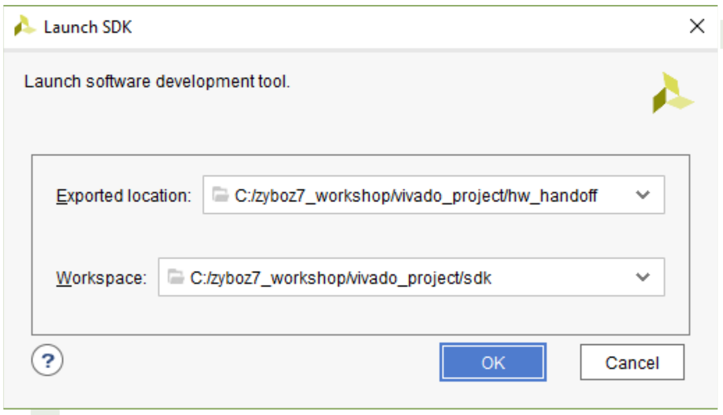
Output Clock	Port Name	Output Freq (MHz)		Phase (degrees)		Duty Cycle (%)		Drives	Use Fine P
		Requested	Actual	Requested	Actual	Requested	Actual		
✓ clk_out1	clk_out1	50	50.000	0.000	0.000	50.000	50.0	BUFG	
✓ clk_out2	clk_out2	150	150.000	0.000	0.000	50.000	50.0	BUFG	
✓ clk_out3	clk_out3	200	200.000	0.000	0.000	50.000	50.0	BUFG	
clk_out4	clk_out4	100.000	N/A	0.000	N/A	50.000	N/A	BUFG	
clk_out5	clk_out5	100.000	N/A	0.000	N/A	50.000	N/A	BUFG	
clk_out6	clk_out6	100.000	N/A	0.000	N/A	50.000	N/A	BUFG	
clk_out7	clk_out7	100.000	N/A	0.000	N/A	50.000	N/A	BUFG	

USE CLOCK SEQUENCING

Output Clock	Sequence Number
clk_out1	1
clk_out2	1
clk_out3	1
clk_out4	1
clk_out5	1
clk_out6	1
clk_out7	1

Figure 10 Clocking Wizard output frequencies

The hardware project is pre-built and exported, so let us turn our attention to the software running on the ARM.



Exported location: C:/zyboz7_workshop/vivado_project/hw_handoff

Workspace: C:/zyboz7_workshop/vivado_project/sdk

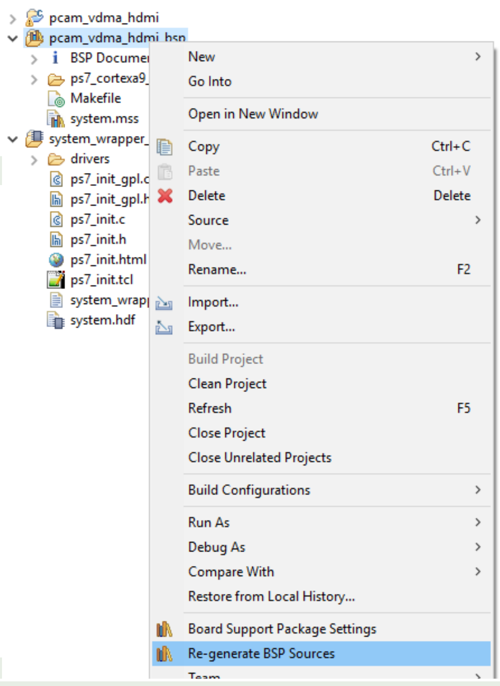
Open the SDK workspace

In Vivado go to File, Launch SDK.

Choose for Exported Location: zyboz7_workshop/vivado_project/hw_handoff.

Choose for Workspace: zyboz7_workshop/vivado_project/sdk.

Click Yes on the out-of-date warning dialog.

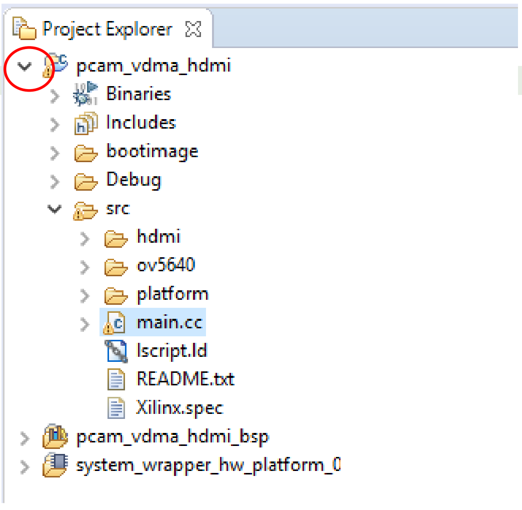


Right-click on pcam_vdma_hdmi_bsp and choose Re-generate BSP Sources.

Confirm with Yes.

Once the BSP re-generates, press Ctrl+B to re-build the whole workspace (or go to Project menu -> Build All...)

The application should be without errors.



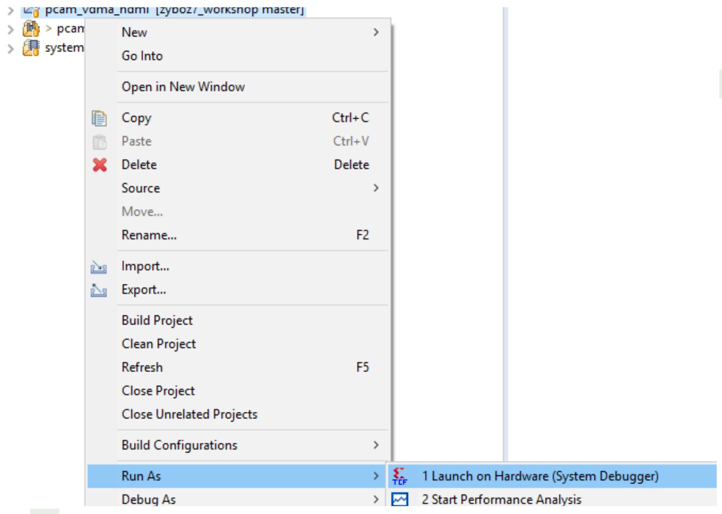
View project source files

Open the pcam_vdma_hdmi project structure by clicking on the arrow.

Also open the src subfolder.

Open main.cc by double-clicking it.

Now do the hardware set-up. Carefully connect the Pcam 5C to the Zybo Z7, paying attention to the correct cable side. Connect the Zybo Z7 to the PC with the included USB micro cable. Make sure the boot jumper JP5 is on JTAG (pins 3-4). Make sure the power jumper JP6 is on USB (pins middle-bottom). Connect the HDMI TX port of Zybo Z7 to the monitor with the included HDMI cable. Turn the power switch on.

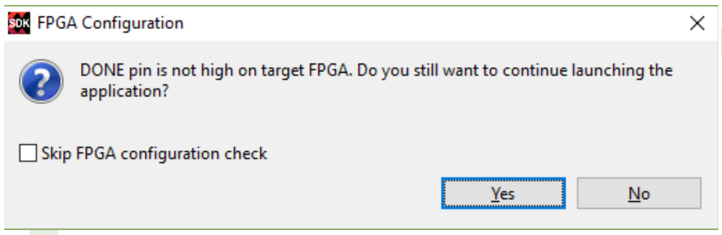


Launch application on hardware

Make sure the Zybo Z7 is connected to the PC via USB, it is turned on and the red PGOOD LED is lit.

Right click on the pcam_vdma_hdmi project.

Go to Run As and Launch on Hardware.


FPGA Configuration

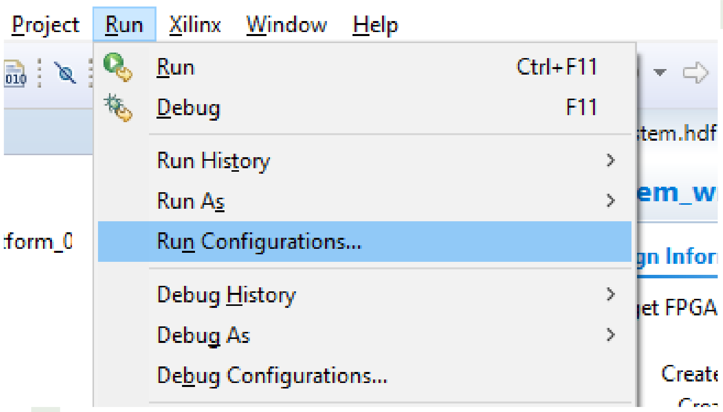
DONE pin is not high on target FPGA. Do you still want to continue launching the application?

☐ Skip FPGA configuration check

Launch warning

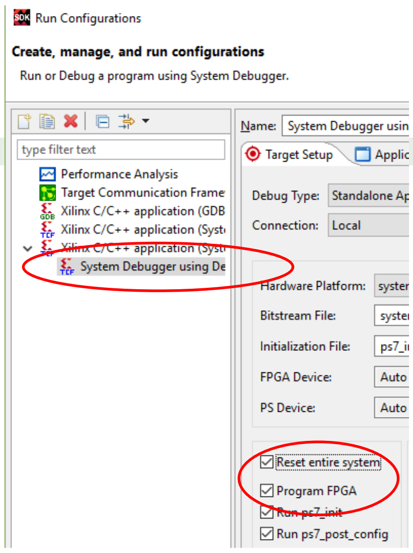
The previous step created a launch configuration that does not include programming the FPGA.

Click No to cancel launch, since the configuration needs to be edited first.



Edit Run Configuration

Go to the Run menu and choose Run Configurations...


Run Configurations

Create, manage, and run configurations

Run or Debug a program using System Debugger.

- Performance Analysis
- Target Communication Frame
- Xilinx C/C++ application (GDB
- Xilinx C/C++ application (Syst
- Xilinx C/C++ application (Syst
  - System Debugger using De

Name: System Debugger usin

Debug Type: Standalone Ap

Connection: Local

Hardware Platform: syster

Bitstream File: syster

Initialization File: ps7_i

FPGA Device: Auto

PS Device: Auto

☑ Reset entire system

☑ Program FPGA

☑ Run ps7_init

☑ Run ps7_post_config

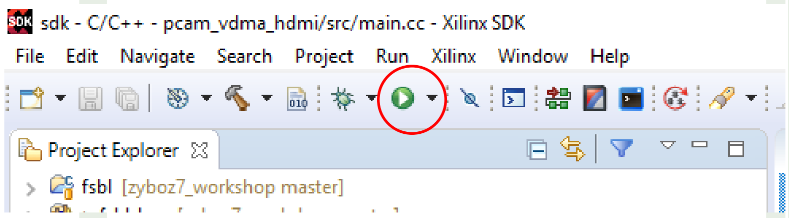
Open Run Configurations

Make sure the pcam_vdma_hdmi.elf configuration is selected in the left panel.

Now tick both "Reset entire system" and "Program FPGA"

Click Run to see the application in hardware.

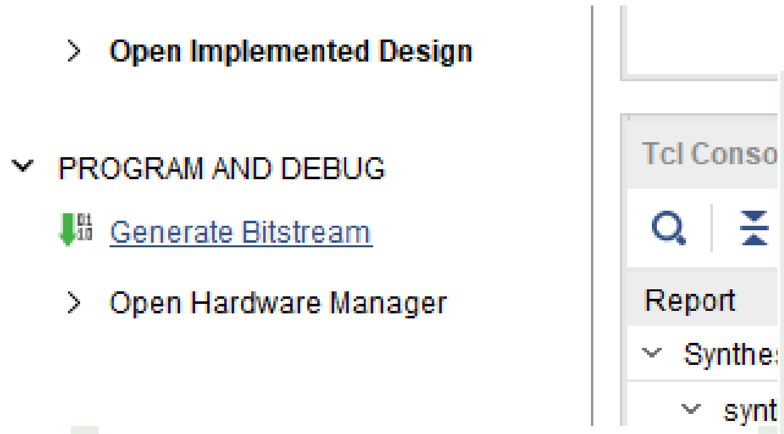
The Zybo Z7 is now forwarding video data from the Pcam 5C to the DVI output. Go ahead, move the camera around and analyze the image on the monitor.


sdk - C/C++ - pcam_vdma_hdmi/src/main.cc - Xilinx SDK

Subsequent runs

Now that the launch configuration is set, it is enough to press the green button on the toolbar to re-program the Zynq and launch the application

To shorten the hardware build time for the next task, re-build the Vivado project to cache synthesis results.



Subsequent runs

Switch back to Vivado.

Click on Generate Bitstream on the left toolbar.

Click Save to keep the re-generated block design.

Click Yes to start both synthesis and implementation.

You may now leave Vivado to re-build the hardware design and complete the next task in the meantime.

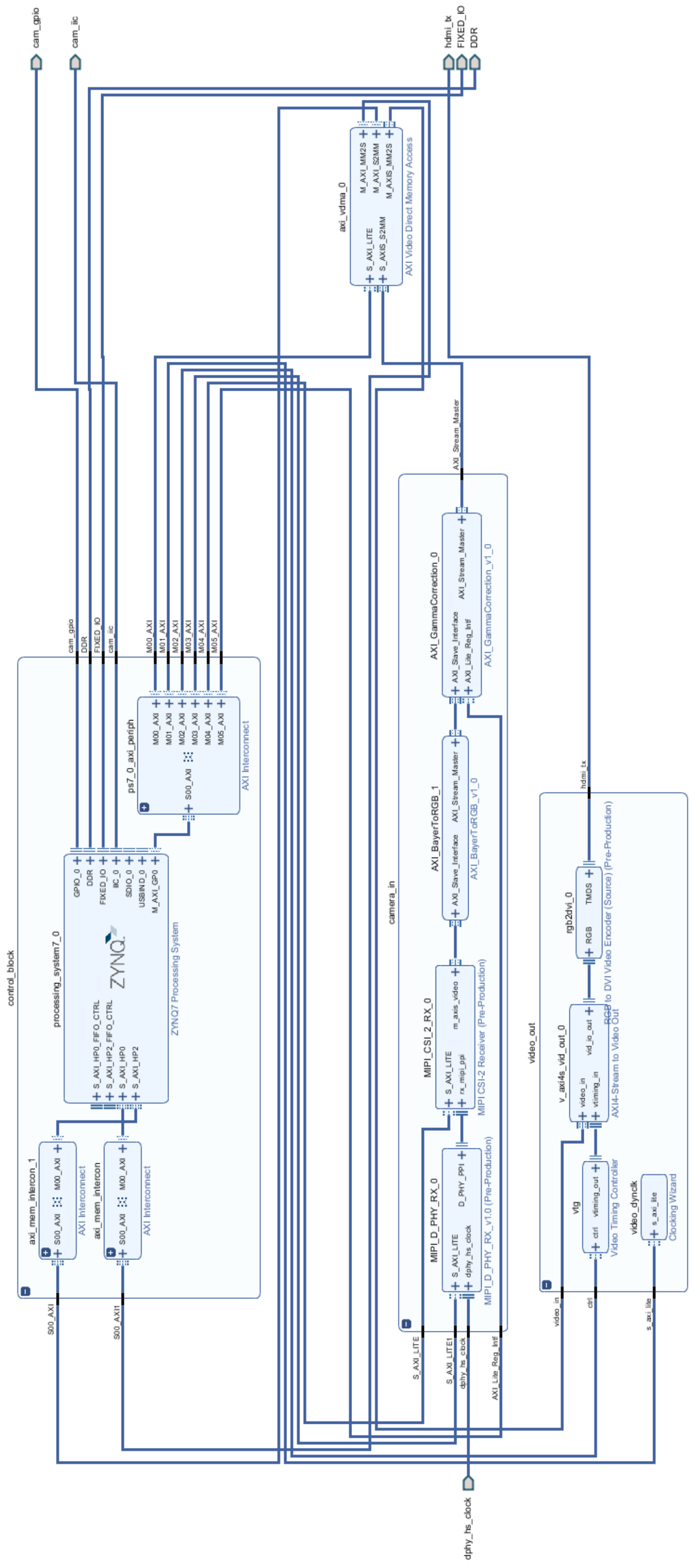


Figure 11 Application block diagram with only interfaces visible

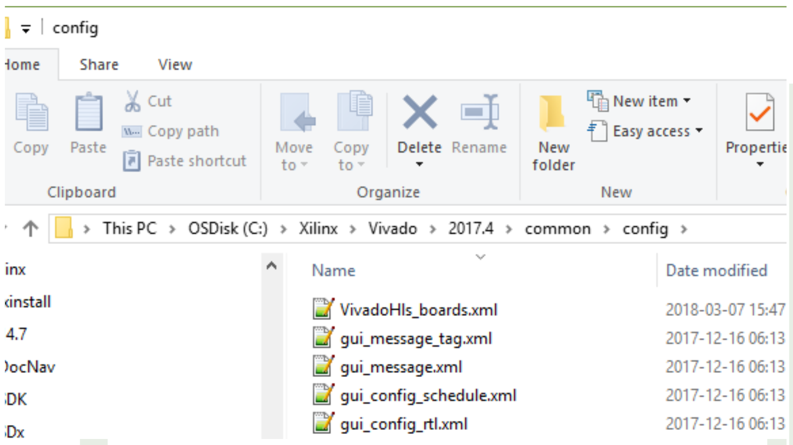
6 Task Three – Edge detection in HLS

The video pipeline analyzed in Task 2 provides a good basis for image processing functions defined in HLS. The bus between blocks "AXI Video Direct Memory Access" and "AXI4-Stream to Video Out" is a streaming interface sending data pixel-by-pixel in raster format. AXI-Stream is a simple interface that suits video data well and Xilinx adopted it for their own IP offering. Using AXI-Stream ensures the greatest interoperability between IPs.

Neither the sensor input, nor the display output is AXI-Stream however. Video interfaces use a continuous stream of pixels forming lines interleaved by blanking intervals. They lack a hand-shake mechanism that could stop the stream for a while when the downstream processing logic requires it. AXI-Stream transmits data more efficiently by packing pixel data and framing signals. Furthermore, thanks to hand-shake signals it allows for buffering and stopping the stream momentarily. All Xilinx Video Processing IP use AXI-Stream interfaces, if needed these can be easily inserted into the stream. The AXI Video DMA seen earlier needs to be able to stop accepting data from time-to-time due to memory access arbitration. It uses line buffers that can for short periods store pixels until the main memory is free. Due to the streaming nature of the data, different processing blocks can even be daisy-chained by attaching the output of one to the input of another. This is called video processing pipeline.

The interface of the pipeline is an essential design aspect of an HLS processing core. The input, output and control interfaces all need to be modeled in C/C++. Fortunately, the data type modeling AXI-Stream already exists in HLS template libraries.

So, our task is writing a processing block (function) that accepts an AXI-Stream RGB video input (argument), and outputs the similarly formatted processed video data (argument). The project requirements are 1280x720@60Hz resolution and a stable video feed.



Name	Date modified
VivadoHls_boards.xml	2018-03-07 15:47
gui_message_tag.xml	2017-12-16 06:13
gui_message.xml	2017-12-16 06:13
gui_config_schedule.xml	2017-12-16 06:13
gui_config_rtl.xml	2017-12-16 06:13

Add Zybo Z7 board definition to Vivado HLS

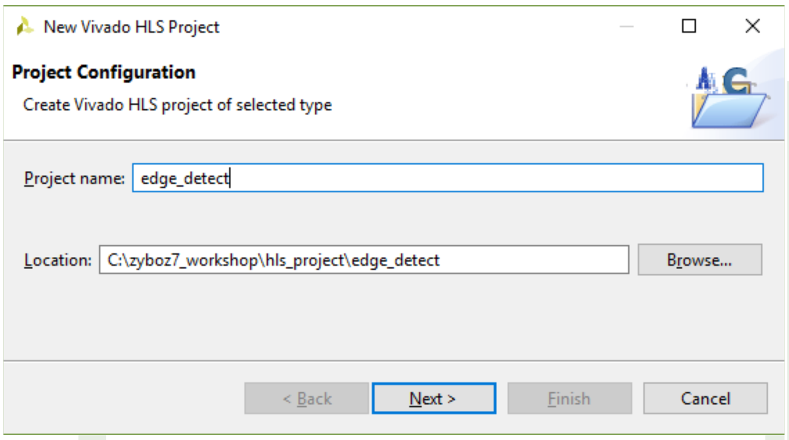
Browse to your Vivado installation folder.

For example, on Windows: C:\Xilinx\Vivado\2018.1\common\config

Or on Linux:

/opt/Xilinx/Vivado/2018.1/common/config

Overwrite VivadoHls_boards.xml with the one provided among the workshop materials


New Vivado HLS Project

Project Configuration

Create Vivado HLS project of selected type

Project name: edge_detect

Location: C:\zyboz7_workshop\hls_project\edge_detect

New Vivado HLS project

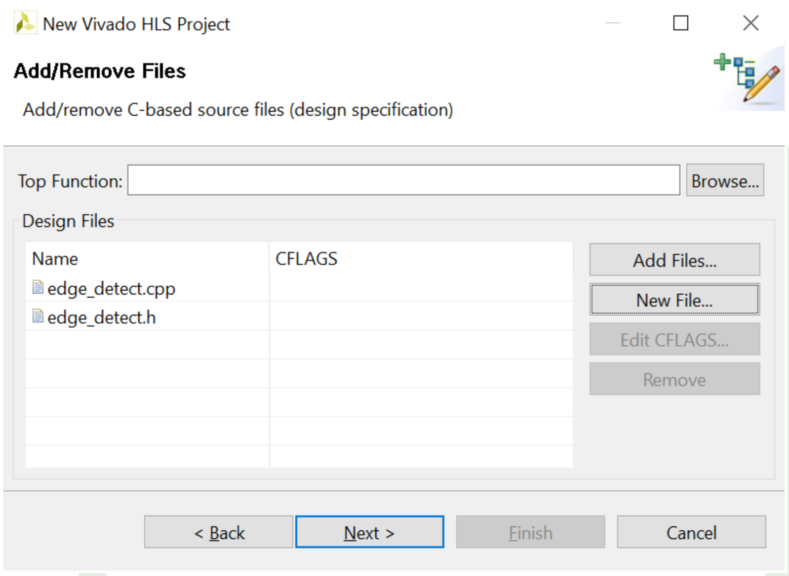
Launch Vivado HLS 2018.1 (NOT Vivado 2018.1) from the Start Menu

Click Create New Project

Name the project edge_detect

Place it under zyboz7_workshop\hls_project\edge_detect

Click Next


New Vivado HLS Project

Add/Remove Files

Add/remove C-based source files (design specification)

Top Function:

Design Files

Name	CFLAGS
edge_detect.cpp	
edge_detect.h	

Create new source and header files

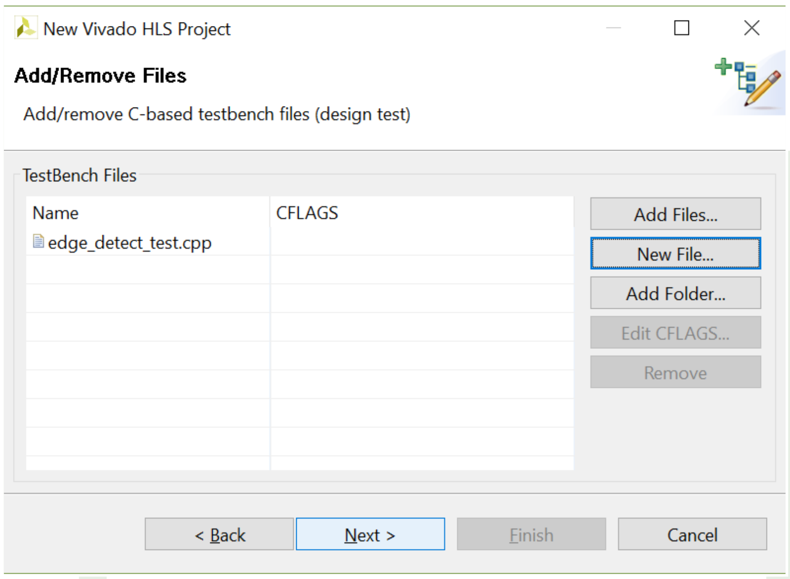
Click New File

Browse to zyboz7_workshop\hls_project\edge_detect.

Name it edge_detect.cpp

Repeat for edge_detect.h

Click Next


New Vivado HLS Project

Add/Remove Files

Add/remove C-based testbench files (design test)

TestBench Files

Name	CFLAGS
edge_detect_test.cpp	

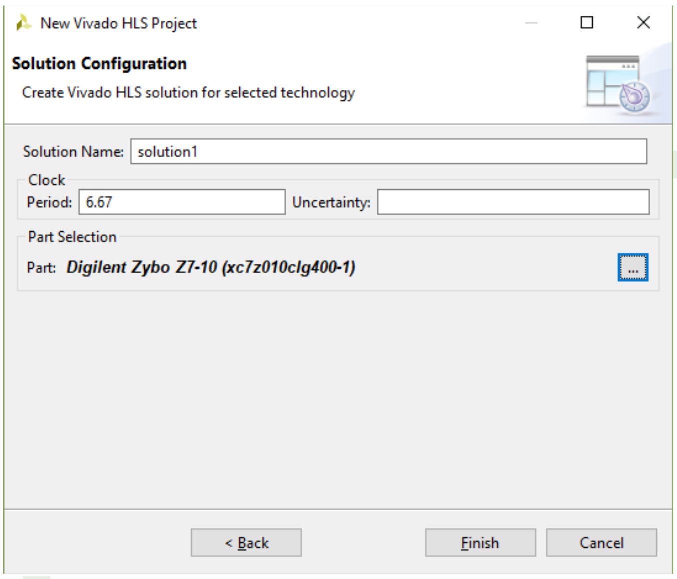
Create new source file for test bench

Click New File

Browse to zyboz7_workshop\hls_project\edge_detect

Name it edge_detect_test.cpp

Click Next


New Vivado HLS Project

Solution Configuration

Create Vivado HLS solution for selected technology

Solution Name: solution1

Clock

Period: 6.67 Uncertainty:

Part Selection

Part: *Digilent Zybo Z7-10 (xc7z010clg400-1)*

Create project constraints

Enter 6.67 for clock period

Click the browse button for part selection

Click Boards

Choose Digilent Zybo Z7-10 in the list of boards

Click Finish

Now that the project is created we can get on with writing actual C++ code. The following files will be written.

```
#include "hls_video.h"

typedef ap_axiu<24,1,1,1> interface_t;
typedef hls::stream<interface_t> stream_t;

void edge_detect(stream_t& stream_in, stream_t& stream_out);

#define MAX_WIDTH 1280
#define MAX_HEIGHT 720

typedef hls::Mat<MAX_HEIGHT, MAX_WIDTH, HLS_8UC3> rgb_img_t;

#define INPUT_IMAGE "fox.bmp"
#define OUTPUT_IMAGE "fox_output.bmp"
```

Figure 12 edge_detect.h

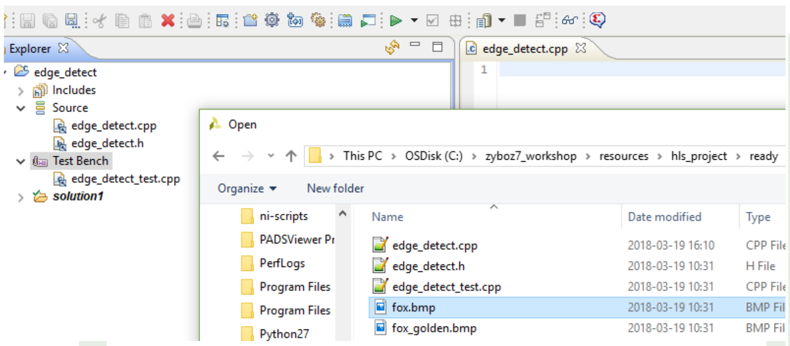
```
#include "edge_detect.h"

void edge_detect(stream_t& stream_in, stream_t& stream_out)
{
  int const rows = MAX_HEIGHT;
  int const cols = MAX_WIDTH;

  rgb_img_t img0(rows, cols);
#pragma HLS STREAM variable=img0 depth=1 dim=1
  rgb_img_t img1(rows, cols);
#pragma HLS STREAM variable=img1 depth=1 dim=1
  rgb_img_t img2(rows, cols);
#pragma HLS STREAM variable=img2 depth=1 dim=1
  rgb_img_t img3(rows, cols);
#pragma HLS STREAM variable=img3 depth=1 dim=1

  hls::AXIvideo2Mat(stream_in, img0);
  hls::CvtColor<HLS_RGB2GRAY>(img0, img1);
  hls::Sobel<1,0,3>(img1, img2);
  hls::CvtColor<HLS_GRAY2RGB>(img2, img3);
  hls::Mat2AXIvideo(img3, stream_out);
}
```

Figure 13 edge_detect.cpp

```
#include "edge_detect.h"
#include "hls_opencv.h"

int main()
{
  int const rows = MAX_HEIGHT;
  int const cols = MAX_WIDTH;

  cv::Mat src = cv::imread(INPUT_IMAGE);
  cv::Mat dst = src;

  stream_t stream_in, stream_out;
  cvMat2AXIvideo(src, stream_in);
  edge_detect(stream_in, stream_out);
  AXIvideo2cvMat(stream_out, dst);

  cv::imwrite(OUTPUT_IMAGE, dst);

  return 0;
}
```

Figure 14 edge_detect_test.cpp

As shown in Task 1, the HLS flow is going to be followed: the processing function written, a test bench written for it, synthesis, report analysis, C/RTL co-simulation and IP export. The process is iterated until all the requirements are met.



Explorer
- edge_detect
  - Includes
  - Source
    - edge_detect.cpp
    - edge_detect.h
  - Test Bench
    - edge_detect_test.cpp
  - solution1

edge_detect.cpp

Open

This PC › OSDisk (C:) › zyboz7_workshop › resources › hls_project › ready

Organize ▾ New folder

ni-scripts
PADSViewer Pr
PerfLogs
Program Files
Program Files
Python27

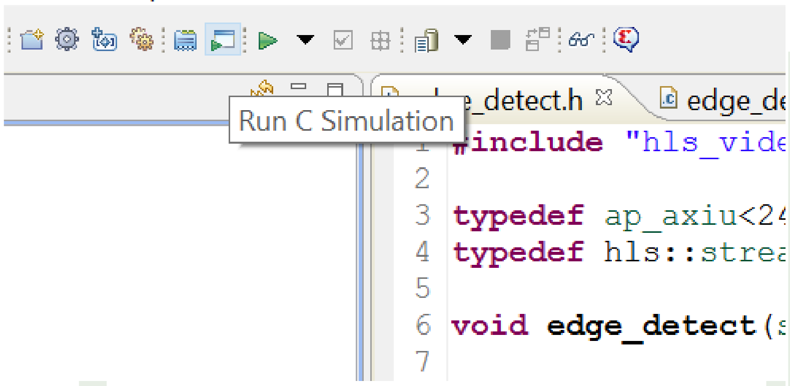
Name	Date modified	Type
edge_detect.cpp	2018-03-19 16:10	CPP File
edge_detect.h	2018-03-19 10:31	H File
edge_detect_test.cpp	2018-03-19 10:31	CPP File
fox.bmp	2018-03-19 10:31	BMP Fil
fox_golden.bmp	2018-03-19 10:31	BMP Fil

Add testbench images

Right-click on Test Bench

Choose Add Files

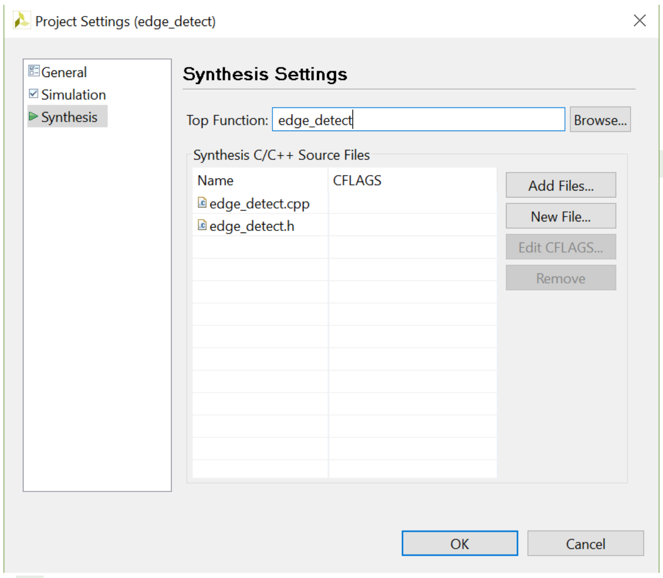
Select fox.bmp to be added from zyboz7_workshop\hls_project.



```
#include "hls_vide
typedef ap_axiu<24
typedef hls::strea
void edge_detect(s
```

Run C Simulation

Click on the Run C Simulation button in the toolbar.



Synthesis Settings

Top Function: edge_detect

Synthesis C/C++ Source Files

Name	CFLAGS
edge_detect.cpp	
edge_detect.h	

Setting top-level function to synthesize

Click on the Project Menu

Choose Project Settings

Choose Synthesis on the left

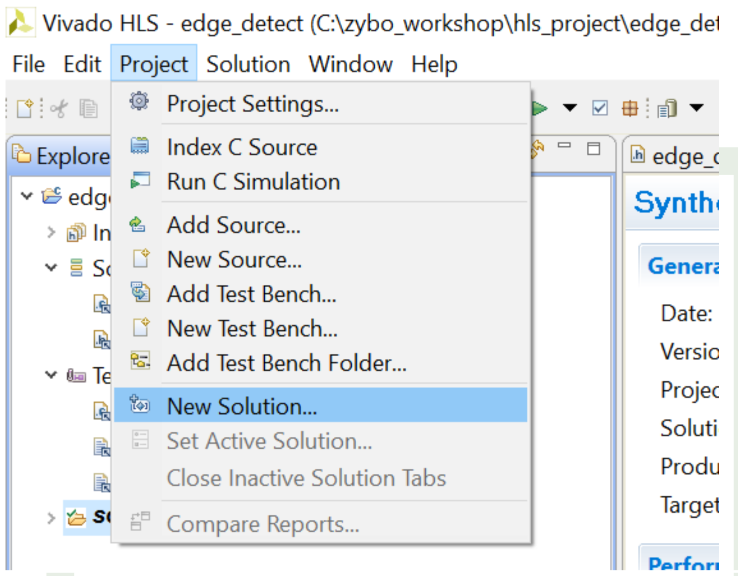
Click Browse next to Top Function

Choose edge_detect

Click OK

Click on the Run C Synthesis button in the toolbar to start hardware synthesis.

After hardware synthesis completes, review the report for clues on whether project requirements are met. If analysis determines that the synthesized code does not meet the requirements, HLS can be directed towards a better design. This is achieved using directives. These influence the choice HLS makes during synthesis both relating to generated logic and interfaces. In this task, we are going to set the DATAFLOW and INTERFACES directives. To be able to compare the results with and without the directives, a new solution can be created.

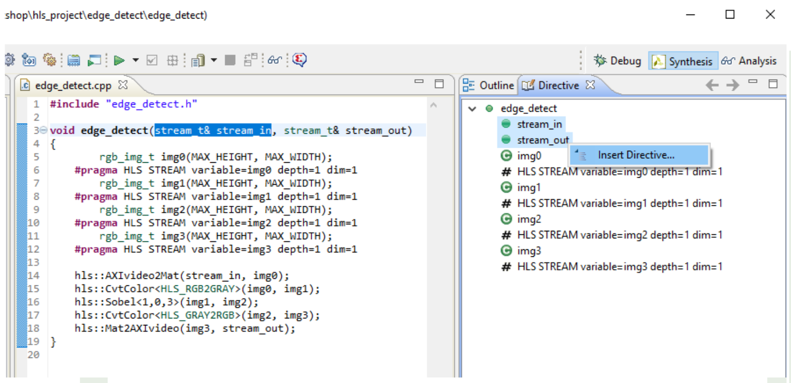


Create a new solution

Open the Project menu and choose New Solution.

Click on Finish to accept the defaults. Notice that settings from solution1 are going to be copied to the new solution.

Solution2 now becomes active.


shop\hls_project\edge_detect\edge_detect)
Debug
Synthesis
Analysis
edge_detect.cpp
Outline
Directive
edge_detect
stream_in
stream_out
Insert Directive...
img0
HLS STREAM variable=img0 depth=1 dim=1
img1
HLS STREAM variable=img1 depth=1 dim=1
img2
HLS STREAM variable=img2 depth=1 dim=1
img3
HLS STREAM variable=img3 depth=1 dim=1

```
#include "edge_detect.h"

void edge_detect(stream_t& stream_in, stream_t& stream_out)
{
        rgb_img_t img0(MAX_HEIGHT, MAX_WIDTH);
    #pragma HLS STREAM variable=img0 depth=1 dim=1
        rgb_img_t img1(MAX_HEIGHT, MAX_WIDTH);
    #pragma HLS STREAM variable=img1 depth=1 dim=1
        rgb_img_t img2(MAX_HEIGHT, MAX_WIDTH);
    #pragma HLS STREAM variable=img2 depth=1 dim=1
        rgb_img_t img3(MAX_HEIGHT, MAX_WIDTH);
    #pragma HLS STREAM variable=img3 depth=1 dim=1

    hls::AXIvideo2Mat(stream_in, img0);
    hls::CvtColor<HLS_RGB2GRAY>(img0, img1);
    hls::Sobel<1,0,3>(img1, img2);
    hls::CvtColor<HLS_GRAY2RGB>(img2, img3);
    hls::Mat2AXIvideo(img3, stream_out);
}
```

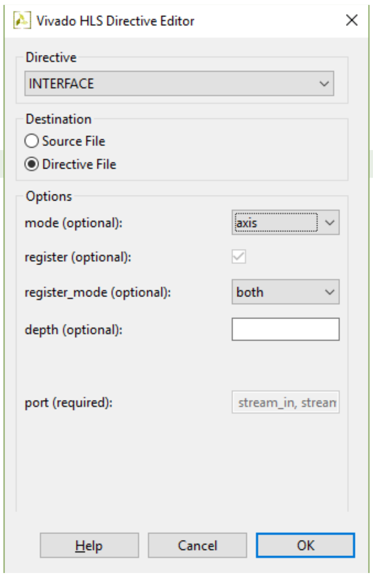
Setting directives

Open edge_detect.cpp, which has the function that needs directives applied

On the right side panel, click on the Directives tab

Select stream_in **and** stream_out interfaces

Right-click on the selection and choose Insert Directive


Vivado HLS Directive Editor

Directive: INTERFACE

Destination: Source File [✓ Directive File]

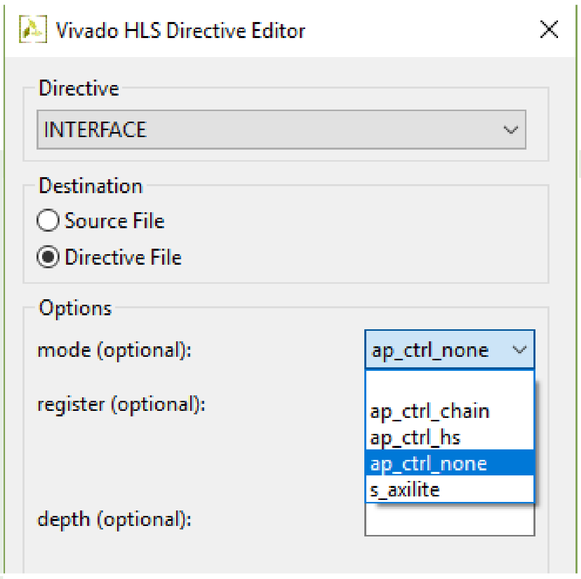
Options
mode (optional): axis
register (optional): ✓
register_mode (optional): both
depth (optional):
port (required): stream_in, stream

Choosing directive options

In the dialog that opens choose the INTERFACE directive

For mode option choose axis to instruct synthesis to generate an AXI-Stream interface for stream_in and stream_out.

Click OK.


Vivado HLS Directive Editor

Directive: INTERFACE

Destination: Source File [✓ Directive File]

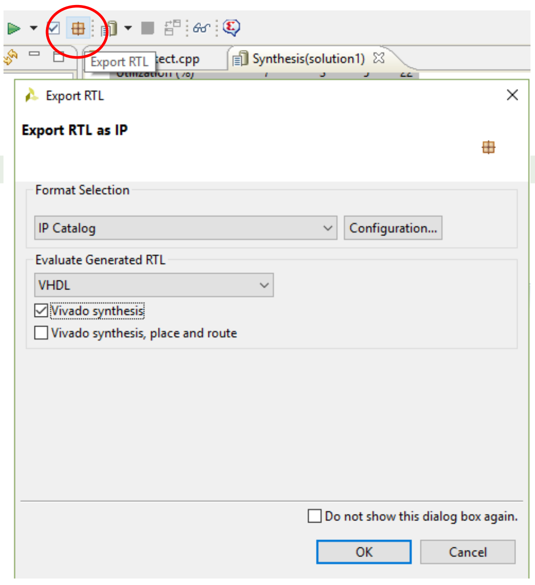
Options
mode (optional): ap_ctrl_none
register (optional):
depth (optional):

Choosing directive options

Similarly, **select function edge_detect** and activate the INTERFACE directive with ap_ctrl_none option for mode.

Lastly, **select function edge_detect** and activate the DATAFLOW directive on it.

Run hardware synthesis one more time and compare the results to that of solution1. Once the design meets the requirements, it can be packaged and exported as an IP. Just choose the Export RTL action in the top toolbar.


Export RTL

Export RTL as IP

Format Selection: IP Catalog

Evaluate Generated RTL: VHDL

☑ Vivado synthesis

☐ Vivado synthesis, place and route

☐ Do not show this dialog box again.

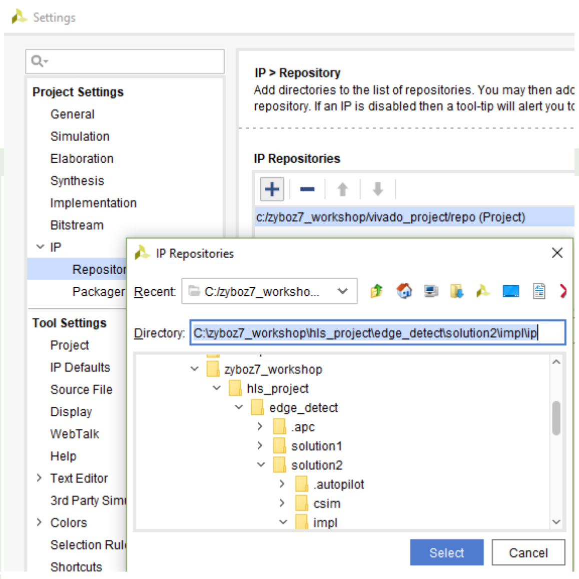
Package and export IP

Click on the Export RTL button in the top toolbar.

Keep the defaults by clicking on OK.

Notice the impl subdirectory in solution2 that will be created.

The next step is importing the video processing IP in the Vivado project and inserting it into the video pipeline. By now Vivado should have finished building the project. You may close the dialog announcing just that.


Settings

IP > Repository

Add directories to the list of repositories. You may then add repository. If an IP is disabled then a tool-tip will alert you to

IP Repositories

c:/zyboz7_workshop/vivado_project/repo (Project)

IP Repositories

Recent: C:/zyboz7_worksho...

Directory: C:\zyboz7_workshop\hls_project\edge_detect\solution2\impl\ip

Adding HLS IP to the video pipeline project

Switch back to the zyboz7_pcam project in Vivado 2018.1 from task two.

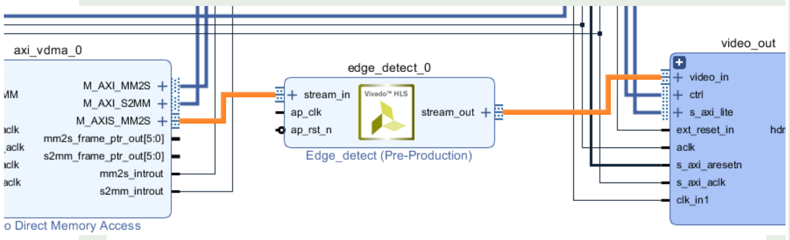
Click Project Settings on the left toolbar

Select IP and Repository

Click the plus sign

Browse to the HLS project path zyboz7_workshop\hls_project\edge_detect\solution2\impl\ip

Click Select and OK



Wiring IP into the pipeline

Right-click on an empty space in the diagram and choose Add IP

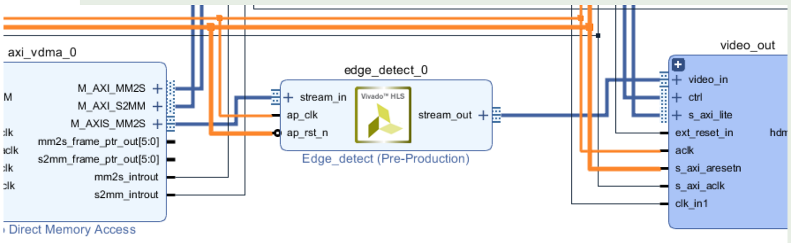
Search for and double-click on Edge_detect, which is our HLS IP

Click on the wire between axi_vdma_0/M_AXIS_MM2S and video_out/video_in

Press the Delete key

Wire M_AXIS_MM2S to stream_in of edge_detect

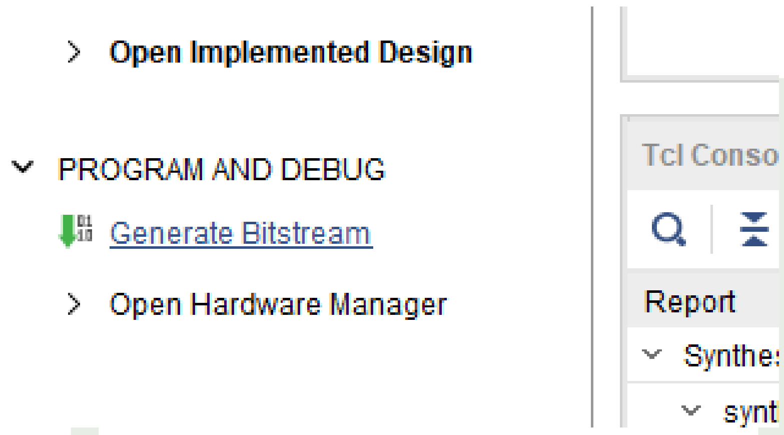
Wire stream_out of edge_detect to video_in



Wiring control bus

Wire ap_clk of edge_detect to clk_out2 of control_block. This is the same clock as for the rest of the AXI-Stream IP in the pipeline.

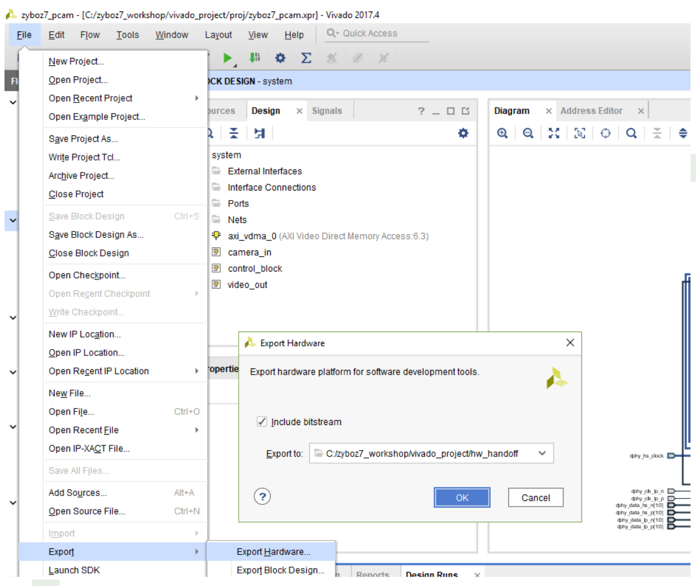
Wire ap_rst_n to S00_ARESETN[0:0] of control_block. This is the same reset as for the rest of the AXI-Stream IP in the pipeline.



Download bitstream

Click on Generate Bitstream to implement the hardware project including the HLS IP.

Wait for Vivado to re-build the hardware project with your HLS IP inserted into the video pipeline.


zyboz7_pcam - [C:/zyboz7_workshop/vivado_project/proj/zyboz7_pcam.xpr] - Vivado 2017.4

Export Hardware

Export hardware platform for software development tools.

[✓] Include bitstream

Export to: C:/zyboz7_workshop/vivado_project/hw_handoff

Export hardware

Go to File, Export, Export Hardware.

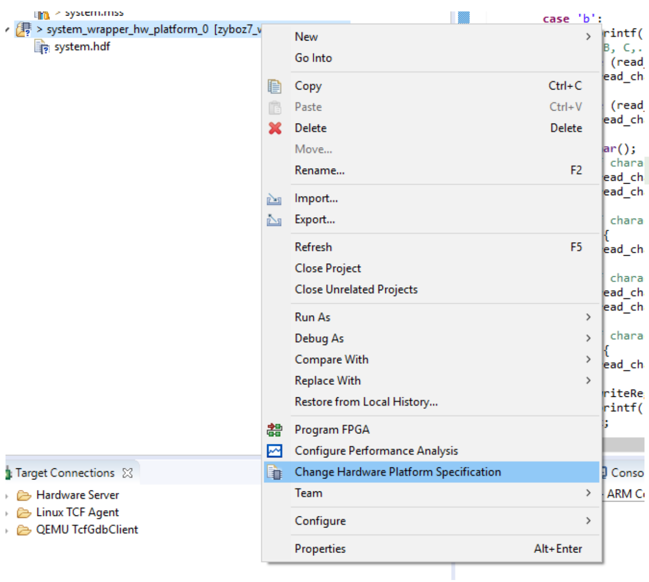
Tick Include bistream.

Click on Export to, Choose Location.

Export to zyboz7_workshop/ vivado_project/ hw_handoff.

Click OK

Confirm overwrite of existing hardware platform.



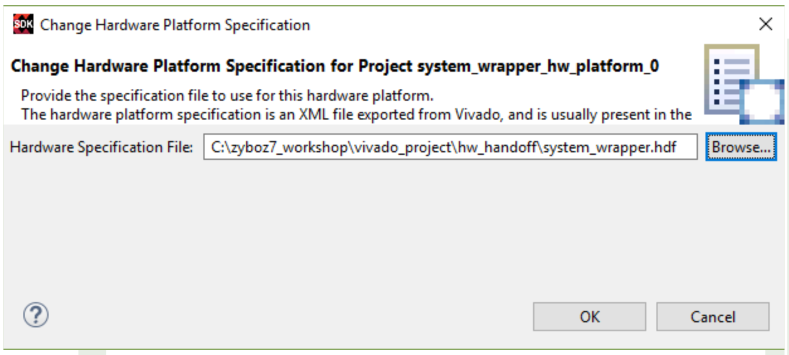
Importing new hardware platform

Switch now to the SDK workspace opened earlier.

Right-click on system_wrapper_hw_platform_0.

Click on Change Hardware Platform Specification.

Choose Yes to confirm the change.



Change Hardware Platform Specification

Change Hardware Platform Specification for Project system_wrapper_hw_platform_0

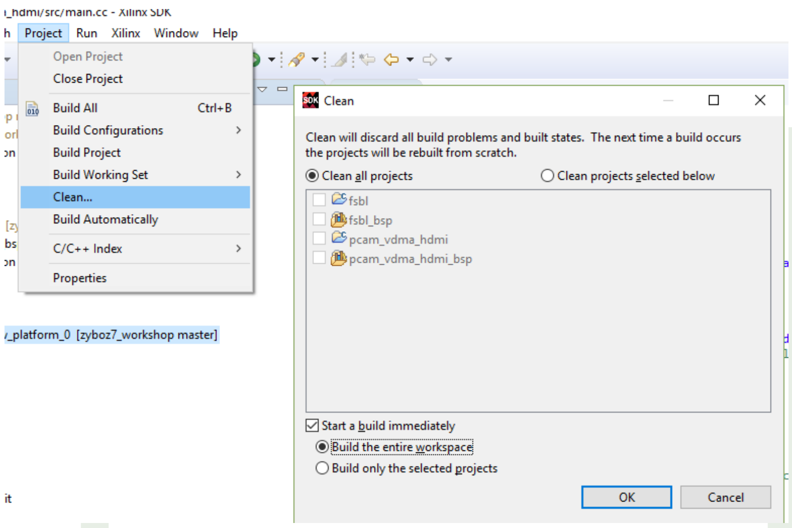
Provide the specification file to use for this hardware platform.
The hardware platform specification is an XML file exported from Vivado, and is usually present in the

Hardware Specification File: C:\zyboz7_workshop\vivado_project\hw_handoff\system_wrapper.hdf

Importing new hardware platform

Browse to the path zyboz7_workshop\vivado_project\hw_handoff\system_wrapper.hdf.

Click OK.



Clean will discard all build problems and built states. The next time a build occurs the projects will be rebuilt from scratch.

[✓ Clean all projects] Clean projects selected below

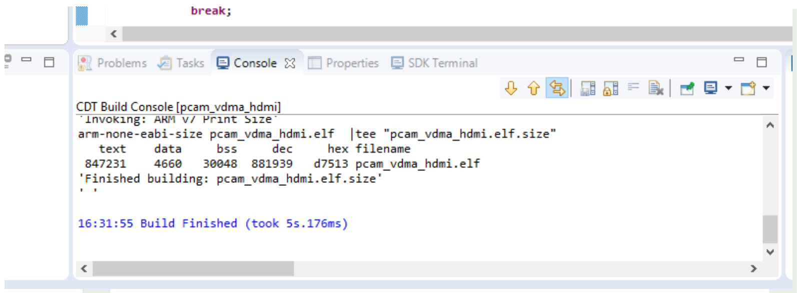
[✓ Start a build immediately]

[✓ Build the entire workspace] Build only the selected projects

Clean and re-build workspace.

Go to the Project menu and choose Clean.

Choose Clean all projects and Start building the entire workspace.



CDT Build Console [pcam_vdma_hdmi]

```
'Invoking: ARM v7 Print Size'
arm-none-eabi-size pcam_vdma_hdmi.elf  |tee "pcam_vdma_hdmi.elf.size"
   text    data     bss     dec     hex filename
 847231    4660   30048  881939   d7513 pcam_vdma_hdmi.elf
'Finished building: pcam_vdma_hdmi.elf.size'
' '

16:31:55 Build Finished (took 5s.176ms)
```

Successful build

Once the build finishes, confirm in the console that there are no errors.

Launch the application by clicking on the green play button on the toolbar.

Zybo Z7 should now forward incoming video data to the DVI monitor after applying the edge detection algorithm on it.

This concludes our workshop. Thank you for attending!