ZYBO Video Workshop

Paris, FRANCE

23.03.2017

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 number 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 of the engineering problems to be approached with software solutions at first. Depending on the application a software-only approach might not meet the requirements, be those latency, throughput, power or other. An expensive option would be handing 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, and ASIC design might not be viable economically.

Bridging the gap between generic processor circuits and ASICs are FPGAs, allowing the use of blank reprogrammable hardware logic elements to implements a custom circuit. It offers a lower barrier of entry to power savings and performance benefits of fabrication technologies without the cost of ASIC. Also, an algorithm optimized for FPGA implementation benefits from the inherently parallel nature of custom circuits.

2 Hardware

The Digilent Zybo development board is well-suited for prototyping an algorithm running in software at first 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 software implementation, where design-time is more important than performance, and hardware, where performance is critical.

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 is RTL description in VHDL/Verilog, the other is high level synthesis in C/C++. High level synthesis represents a somewhat similar programming model to software programming. However, for a worthwhile improvement 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 array (FPGA) is a large array of configurable logic blocks (CLB), interconnect wires and input/output (I/O) pads. The CLB is made up of look-up tables (LUT) and flip-flops (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 FFs to pipeline the data flow synchronous to a clock signal. Interconnect is also 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.

LUT is a memory element that implements a truth-table. Depending on the exact architecture, each LUT has a 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. It can also be thought of and used as a 2^N-memories, called distributed RAM. It is a fast memory type because it can be instantiated all over the FPGA fabric, local to the circuit that needs data from it.

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.

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 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 can be accessed with the lowest latency possible.
It requires considerable effort and performance analysis tools to optimize code for execution time.

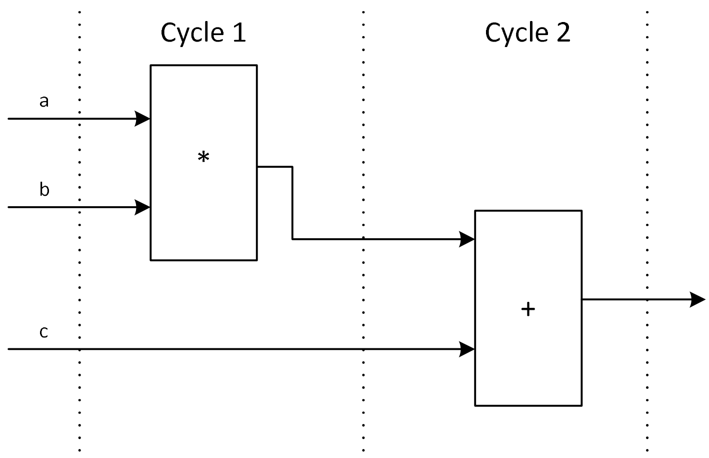
The FPGA is massively parallel by nature. Every LUT can execute a different function at the same time, so it is possible to have multiple arithmetic logic units (ALU) executing addition operations, for example is parallel. On a processor, the ALU is shared and these would have to be executed sequentially. 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 a C/C++ code that is functionally correct and meets the 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 are determined that implement the scheduled operation best. This is called **binding.** The last step in the synthesis is the **control logic extraction** which creates a finite state machine that controls when the different operations should execute in the design.

For multi-cycle operations **pipelining** is performed in the scheduling phase. Imagine the following C statement:

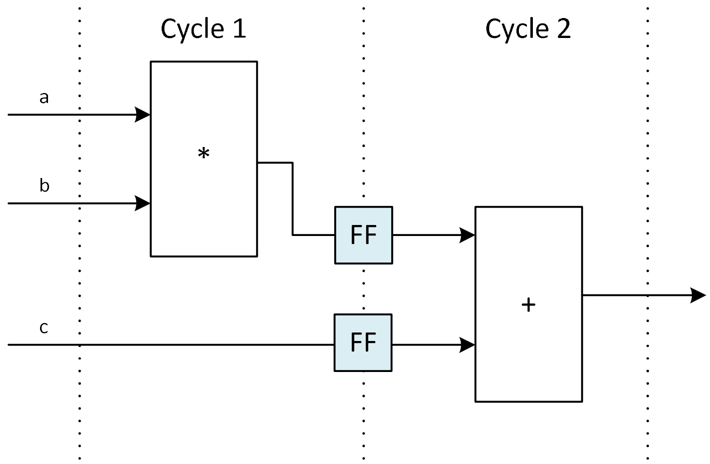
```
x=a*b+c;
```

If the clock period is too small for the multiplication and addition 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.



This inefficiency becomes more apparent, when this statement is executed in a loop, ie. 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 the throughput tremendously.



2.3 Performance metrics

The previous example is a great opportunity to introduce some performance metrics definitions. The **latency** of the statement above is two, as it takes two cycles to output the result. In the first non-pipelined case the **initiation interval (II)** is also 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 is able to accept a new set of inputs in every cycle, and will 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. If the circuit processes 10 sets of input data, the non-pipelined versions will have a total latency of 20 cycles ((#-1) * II + latency). The pipelined versions will only take 11 cycles ((#-1) * II + latency) to provide all the 10 results.

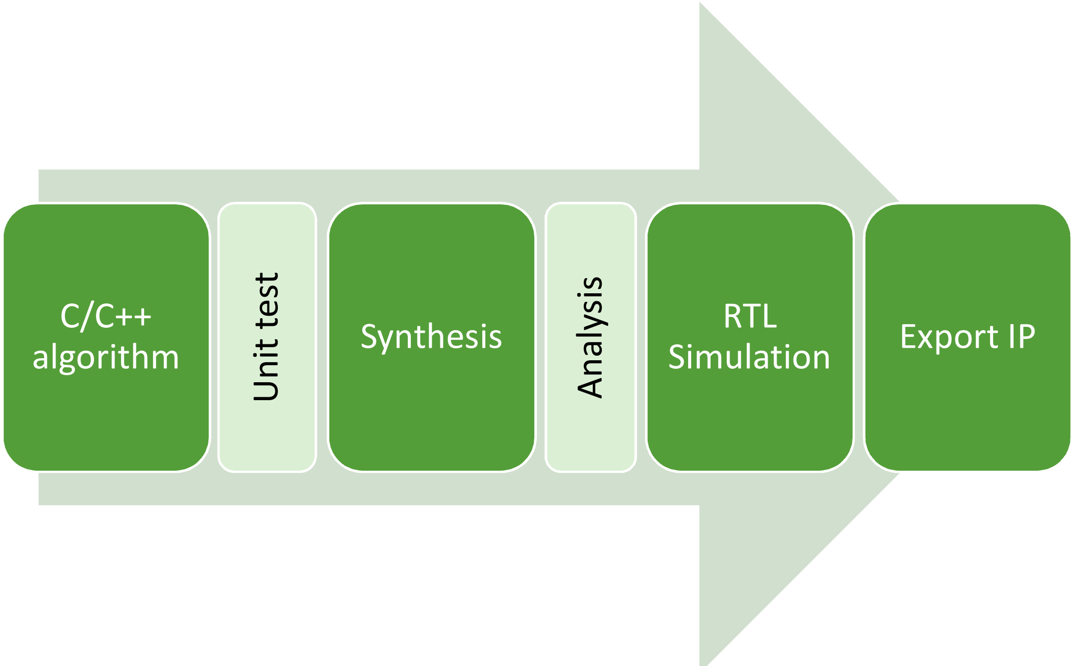
These performance metrics are calculated by the tools for both loops and functions, and are considered the most important feedback mechanism for the designer to evaluate the synthesized hardware circuit.

3 Vivado HLS

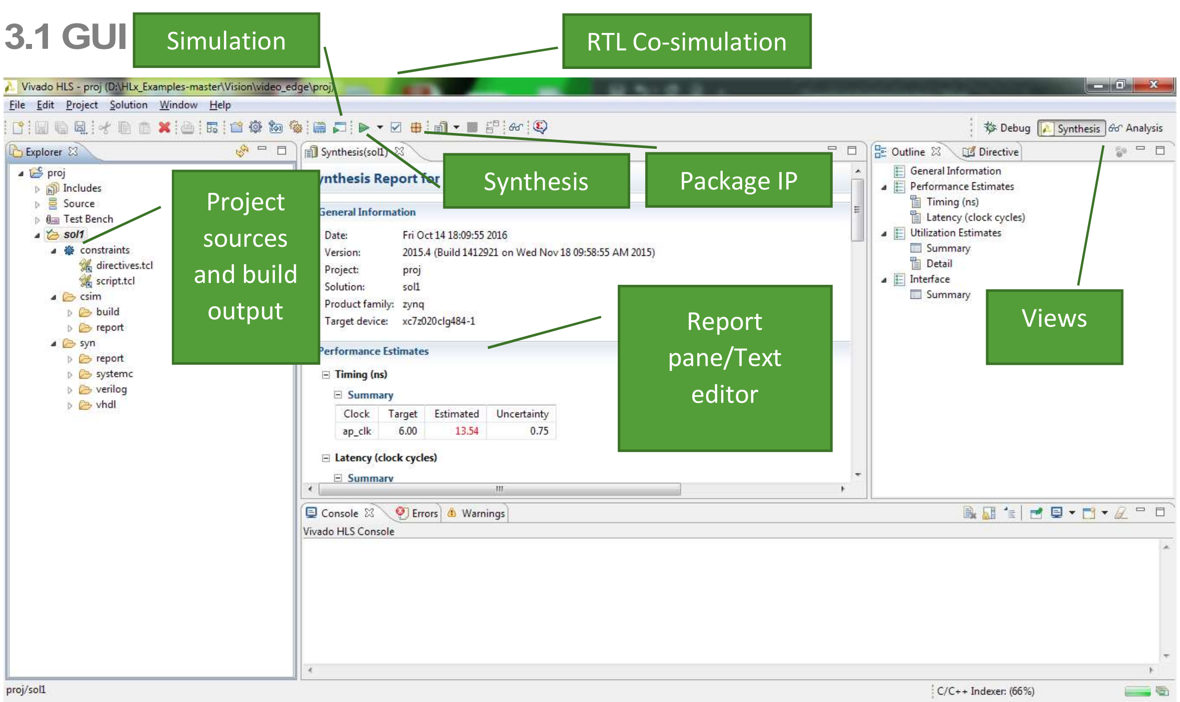
Xilinx's offering in high-level synthesis is part of the Vivado suite and is called Vivado HLS. The workflow is an iterative approach with simulations as verification steps inserted along the way to make sure the design meets the requirements and is functionally correct right from the initial stages.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



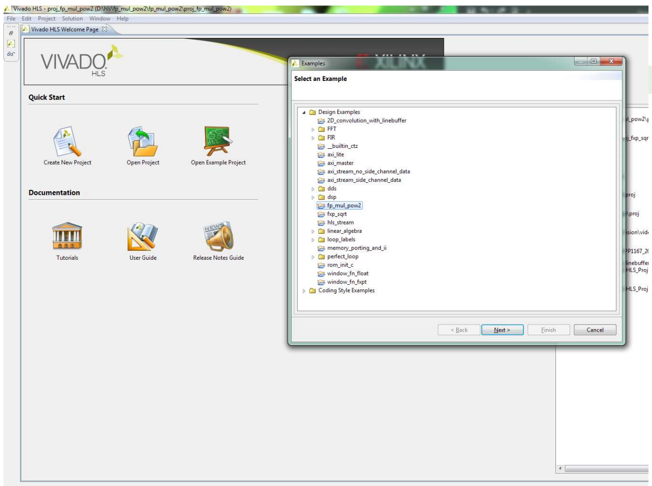
3.1 GUI



The GUI layout is quite similar to 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.



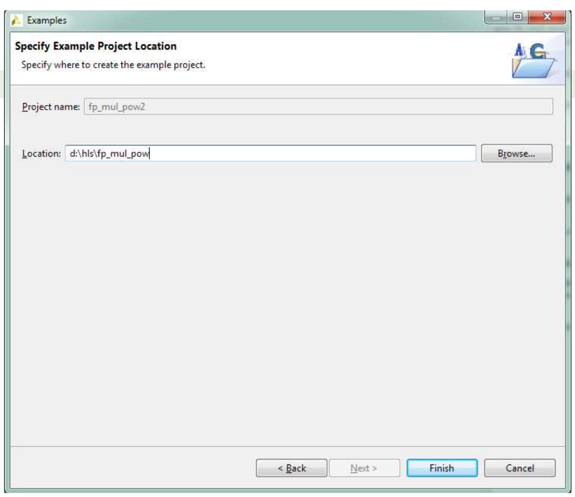
Open example project

Launch Vivado HLS 2015.4 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

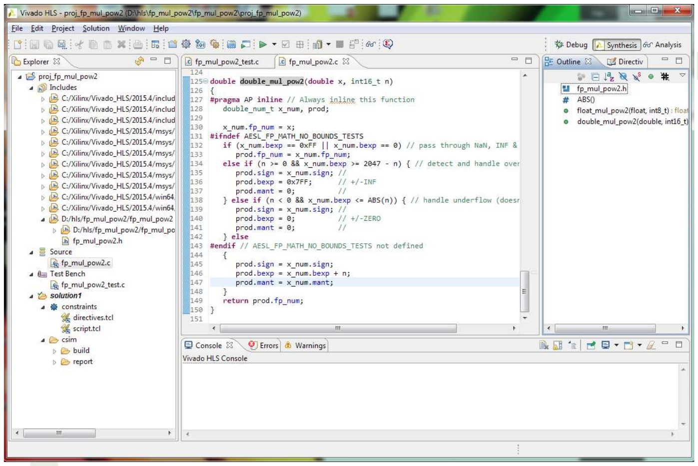


Save project

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

You may choose zybo_workshop/hls_project for location.

Click OK



Analyze project structure

Open fp_mul_pow2.h below Includes.

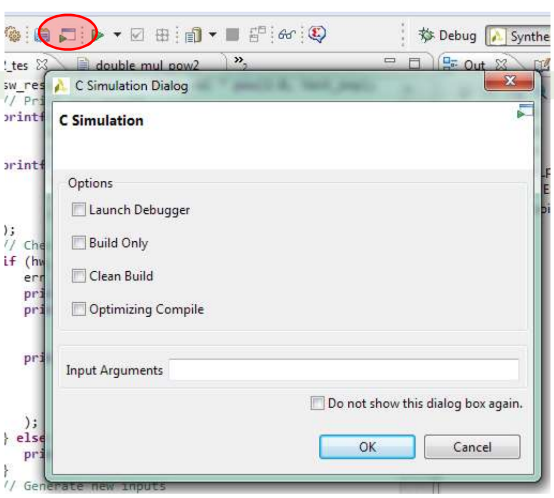
Open fp_mul_pow2.c below Source.

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.

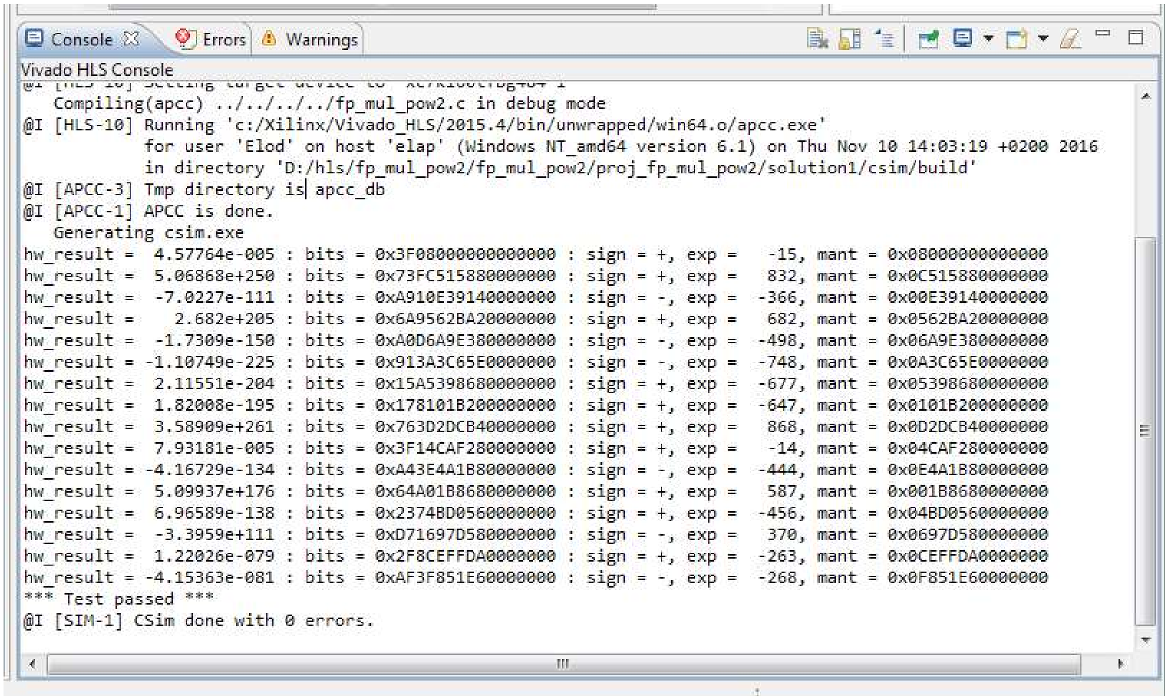


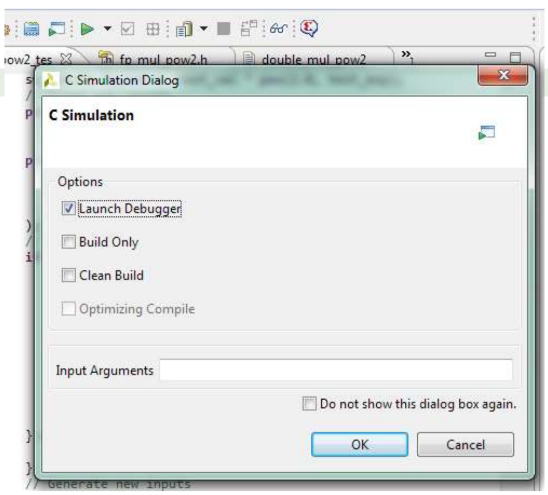
Run C simulation

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.



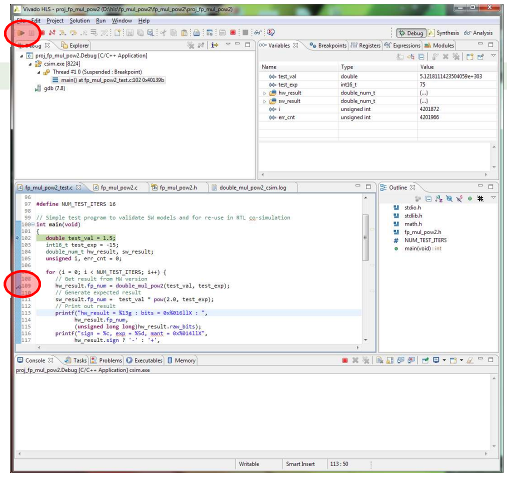


Run C simulation with debugger

Click the Run C Simulation button on the toolbar.

Check the Launch Debugger option and click OK

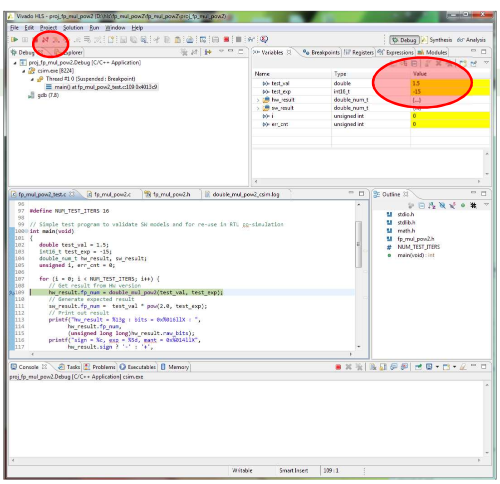
Notice how the Debug view gets activated, the test bench started and stop at the first instruction of the main function. The test bench can be run step-by-step, breakpoints set, variables and expressions evaluated just like any other software project.



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

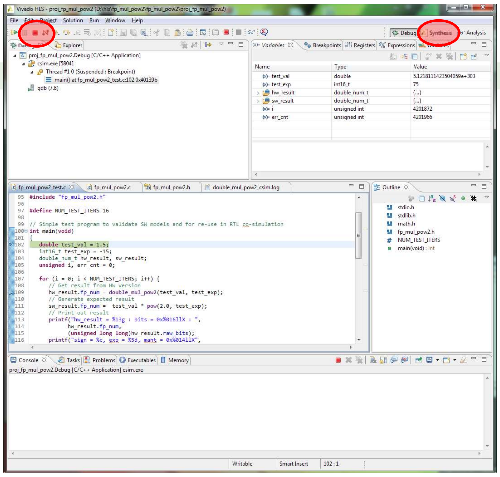


Step into the double_mul_pow2 function

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.

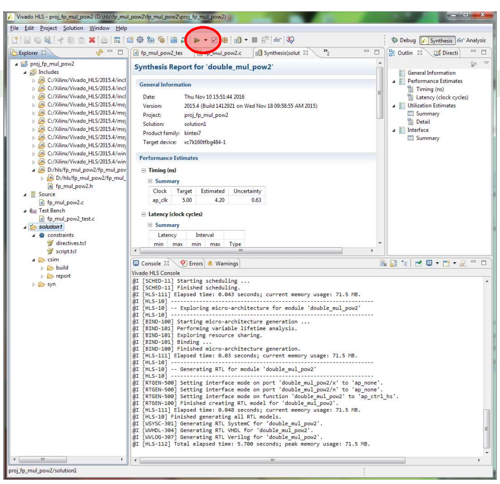


Exit the debugger

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.



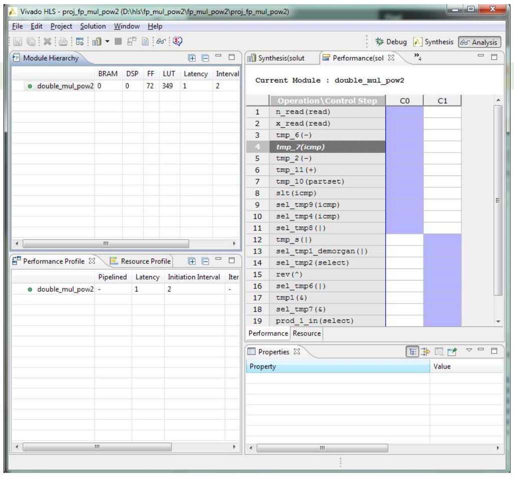
Synthesis

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.

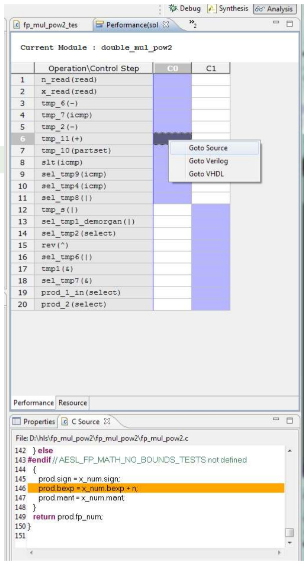
Discuss the report. What did HLS synthesize? What are the latency and interval values? What are the interfaces that got generated?



Analysis

Open the Analysis view.

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.

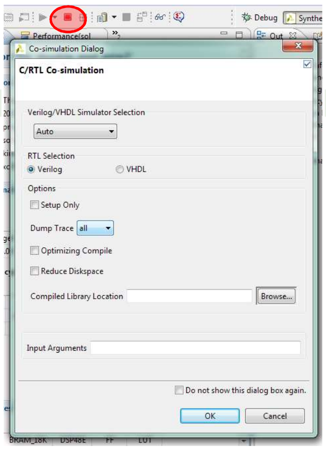


Performance Analysis

Right-click the purple cell in column C0 and row #6, operation tmp_11(+).

Choose Goto Source.

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.



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

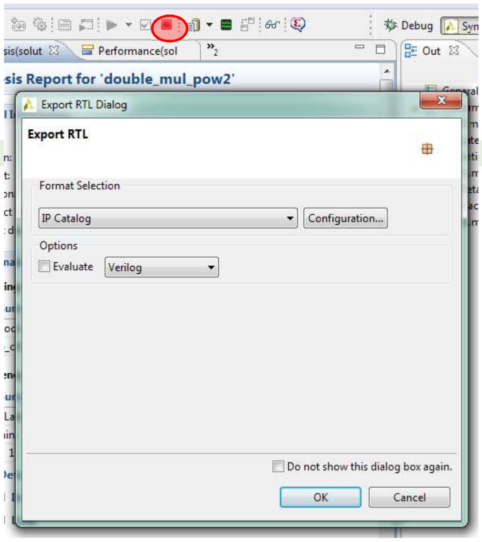
Click on the Open Wave Viewer button on the toolbar

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

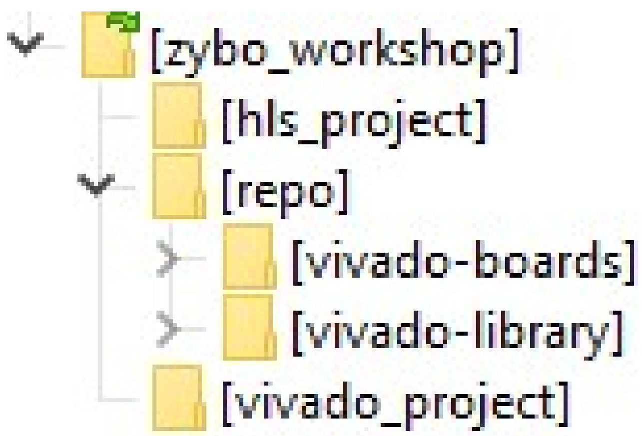
Wait for export to complete

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

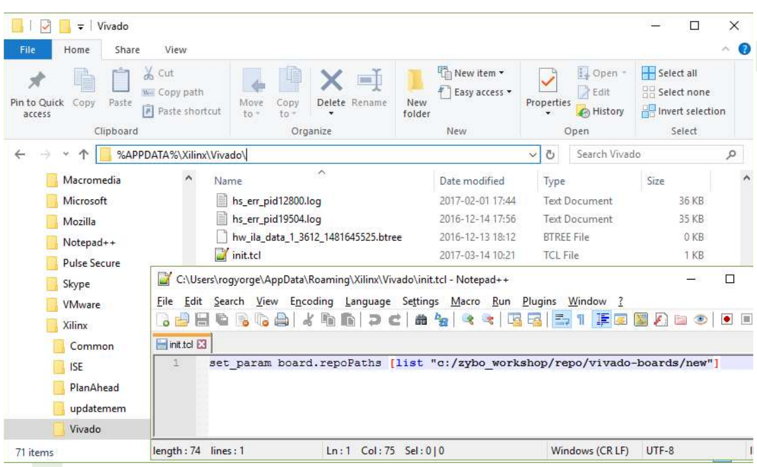
In this step we are going to create an FPGA project that decodes DVI input and forwards it to the VGA output. This pipeline will serve as the base design that will accept the IP exported from HLS. We are going to create it in Vivado block design re-using IP available from Digilent and Xilinx. The Digilent IPs are available online at https://github.com/Digilent/vivado-library/archive/master.zip or among the workshop materials.



Create project tree

Copy the folder called "zybo_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.



Add Digilent board definition files to Vivado

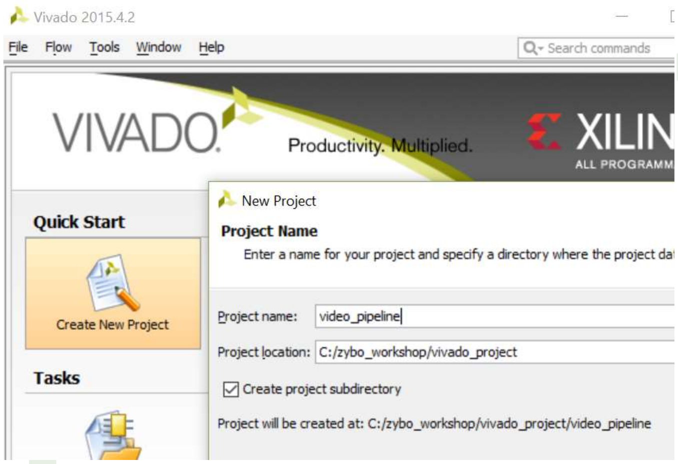
On Windows browse to: %APPDATA%\Xilinx\Vivado\

On Linux cd to: $HOME/.Xilinx/Vivado/

Copy the provided "init.tcl" there.

If you copied "zybo_workshop" to a location other than c:\, edit this file. Make sure the path is absolute and use forward slash "/" as path separator even on Windows.

Save the file and close the editor.



Create video pipeline project

Launch Vivado 2015.4 (NOT Vivado HLS) from the Start Menu

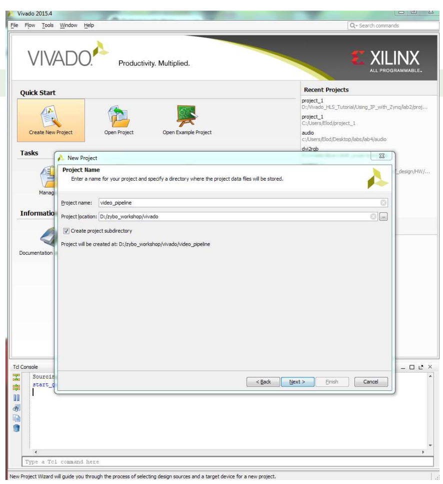
Click Create New Project

Click Next

Name the project "video_pipeline"

Choose zybo_workshop/vivado for Project Location

Click Next twice



Create video pipeline project

Launch Vivado 2015.4 from the Start Menu

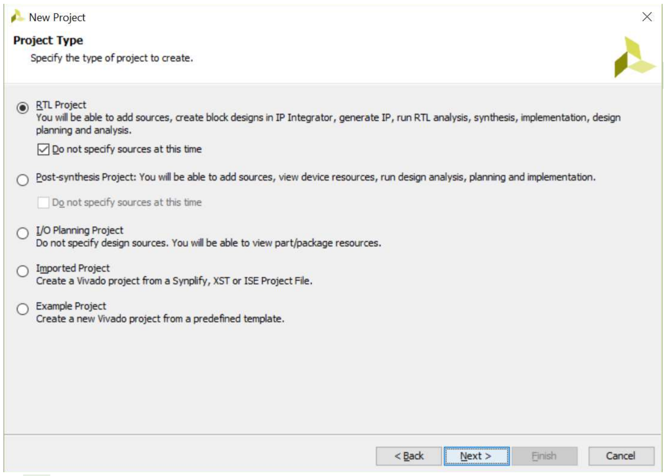
Click Create New Project

Click Next

Name the project "video_pipeline"

Choose zybo_workshop/vivado for Project Location

Click Next

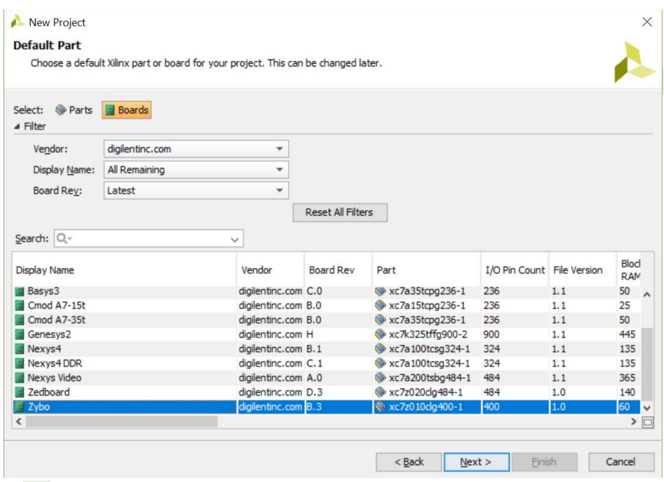


Choose project type

Choose "RTL Project" for project type.

Make sure "Do not specify sources at this time" is ticked.

Click Next.



Choose the Digilent Zybo as target

Select Boards

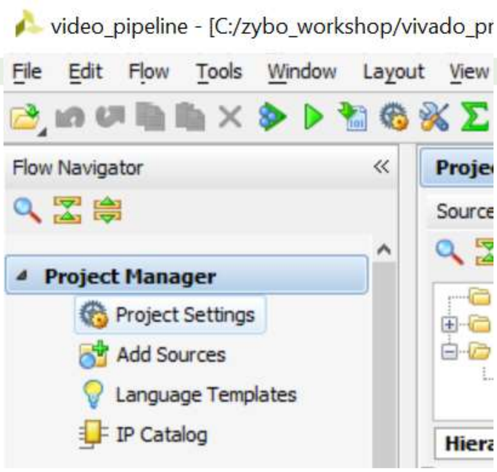
Choose digilentinc.com for Vendor

Choose Zybo from the list below.

Click Next

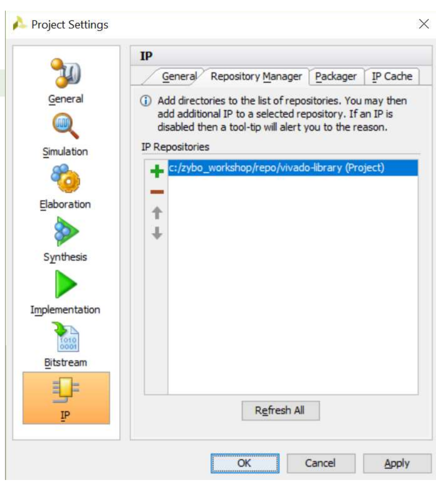
Click Finish

If “Zybo” is not showing among the known boards, go back a few steps and make sure init.tcl is installed at the correct location and it has a valid path in it. Restart Vivado and make sure the Tcl Console is showing that init.tcl has been successfully sourced.



Open Project Settings

Open Project Settings from the Flow Navigator on the left



Add Digilent IP definitions to Vivado

Select the IP category

Switch to the Repository Manager tab

Click the green plus button

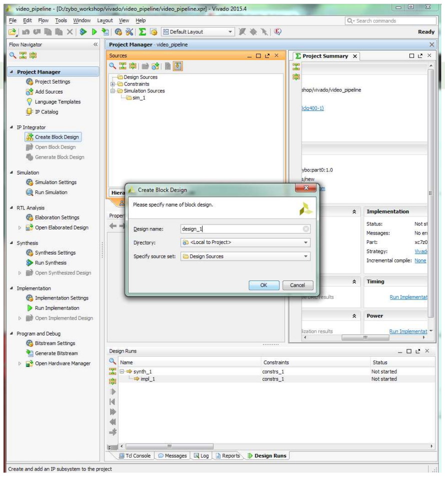
Browse to the zybo_workshop/repo/vivado-library folder

Click Select

Vivado will parse the folder and should find IP definitions there

Click OK to close Project Settings.

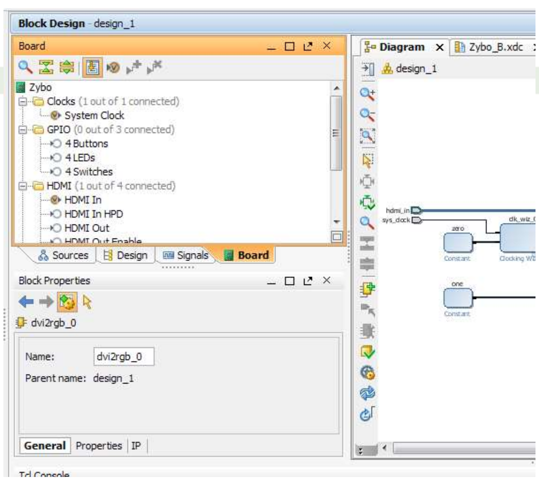
In this project we are going to use the block design flow to create the FPGA design. This helps us re-use any available IP so that we can focus on the processing IP created in HLS. The following IPs are going to be used from Digilent: DVI-to-RGB (DVI Sink), RGB-to-VGA.
And from Xilinx: Video In to AXI4-Stream, AXI4-Stream to Video Out.



Create block design in project

Click Create Block Design on the left toolbar

Leave the defaults and click OK



Use Board interfaces

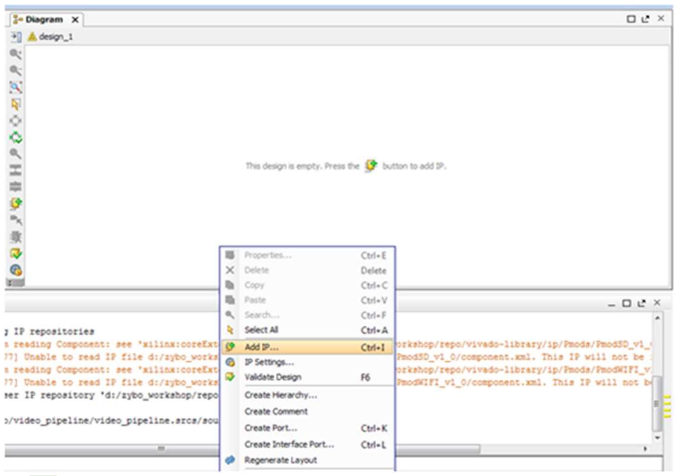
Click on the Board tab to see the interfaces that are available for the Zybo in board design flow.

Double-click on System Clock

Accept the default of instantiating a new Clocking Wizard

Double-click on HDMI In

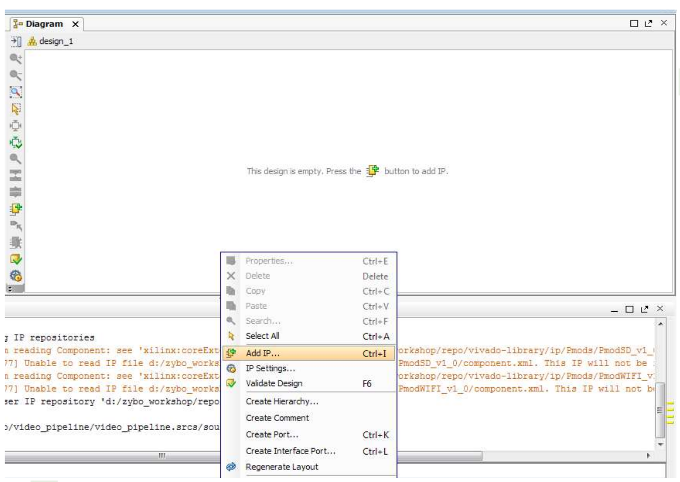
Accept the default of instantiating a new DVI to RGB Converter IP



Add IPs to the block design

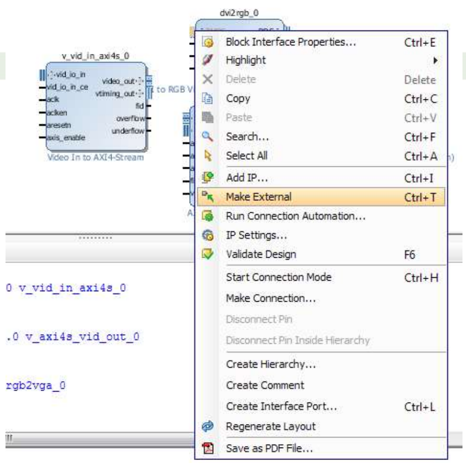
Right-click on an empty space in the diagram and choose Add IP

Search for VGA and double-click "RGB to VGA output"



Add IPs to the block design

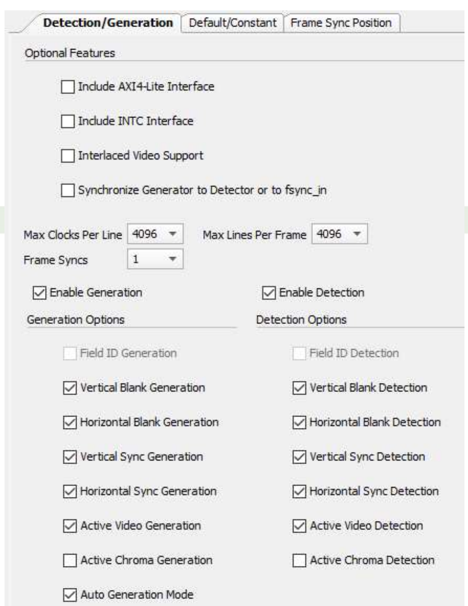
Repeat for "Video In to AXI4-Stream", "AXI4-Stream to Video Out,, "Video Timing Controller", and two instances of "Constant"



Make external interfaces

Right-click the dout interface on the xlconstant_1 block and choose "Make External"

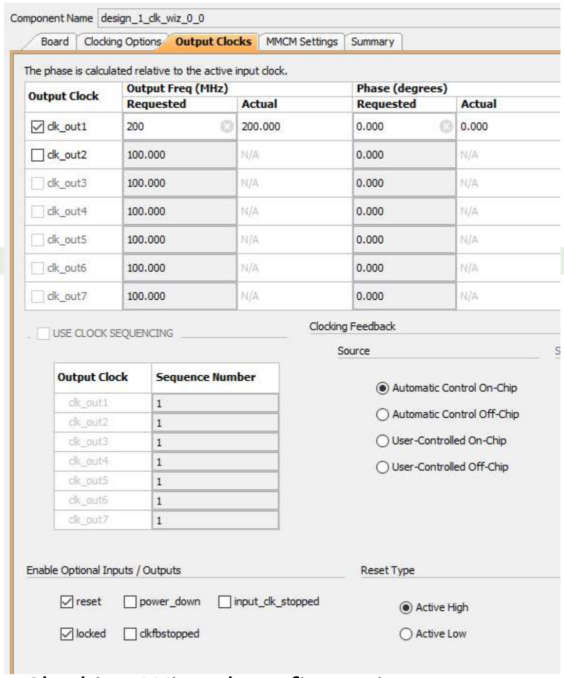
Repeat for rgb2vga_0/vga_pRed, rgb2vga_0/vga_pGreen, rgb2vga_0/vga_pBlue, rgb2vga_0/vga_pHSync, rgb2vga_0/vga_pVSync.



Video Timing Controller configuration

Double-click the v_tc_0 block

Configure it like shown on the left.



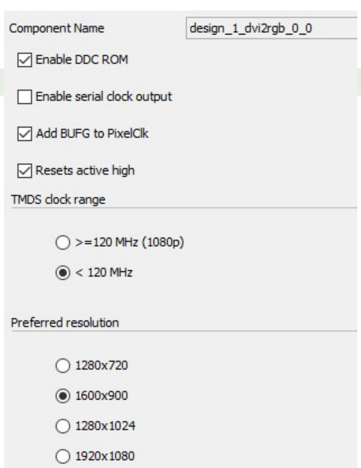
Clocking Wizard configuration

Double-click the clk_wiz_0 block

Open the Output Clocks tab.

Configure it like shown on the left

The IP should generate a 200MHz clock from the 125MHz on-board clock

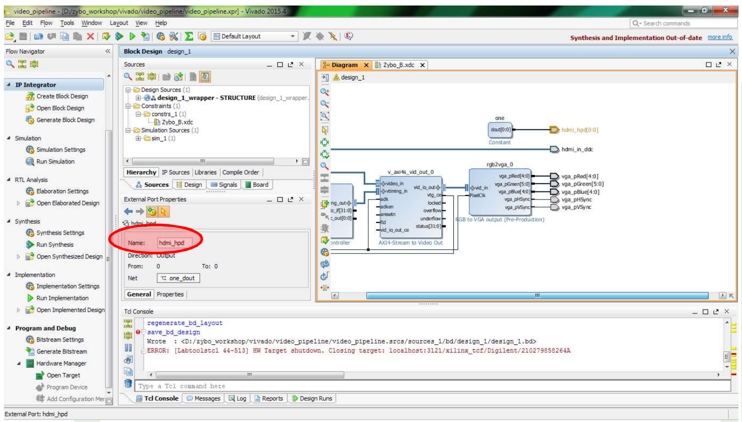


DVI to RGB Video Decoder configuration

Double-click the dvi2rgb_0 block

Configure it like shown on the left.

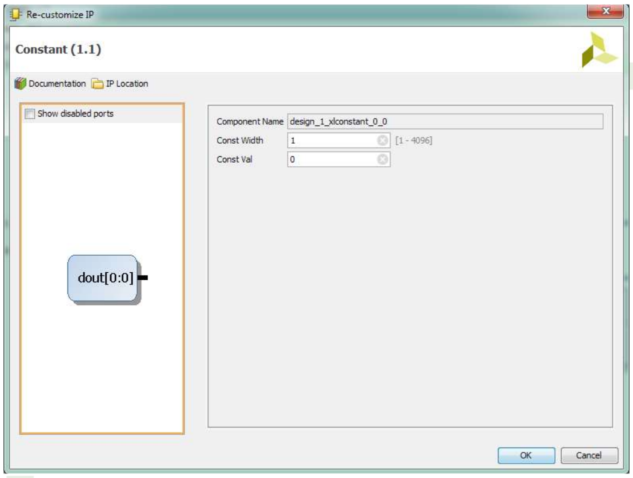
DVI Sinks are required to bring the hot plug detect (HPD) pin high to signal their presence to DVI Sources. We will name this port hdmi_hpd and tie to a constant high value.



Click on the external port created for xlconstant_1, named dout[0:0] by default.

On the left, under External Port Properties modify the name field to "hdmi_hpd"

Rename ports/blocks



Constant configuration

In the same manner, select the xlconstant_0 block and rename it to "zero".

Double-click the xlconstant_0 block

Configure it so the Const Val is "0".

Repeat for xlconstant_1 block, but rename it to "one" and configure it for Const Val 1.

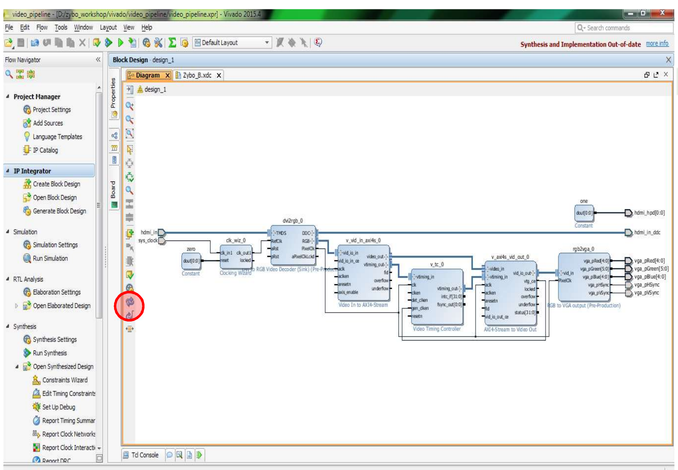
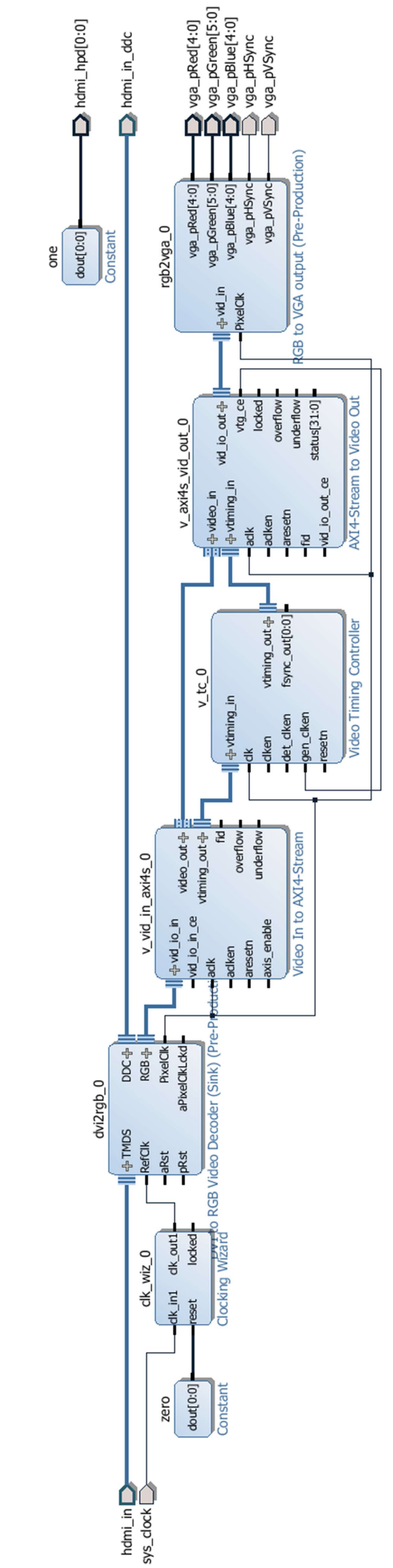


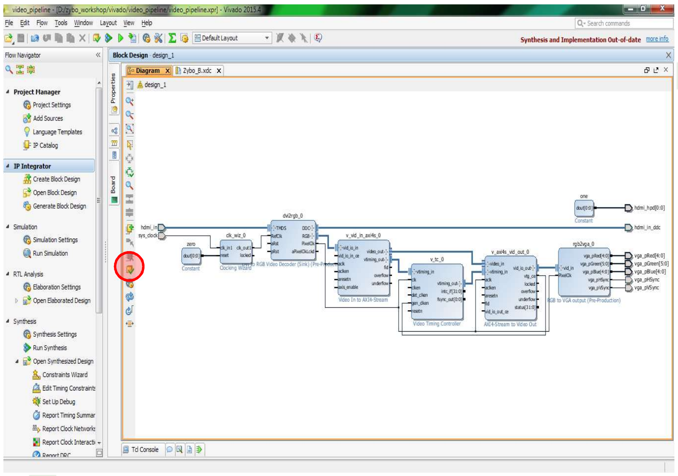
Diagram wiring

Wire the blocks like shown on the next page.

Click-and-hold on one interface and drag it to another to establish a connection.

The Regenerate Layout button on the toolbar to the left of the diagram will re-arrange the blocks into a more readable layout.


hdmi_hpd[0:0]
hdmi_in_ddc
vga_pRed[4:0]
vga_pGreen[5:0]
vga_pBlue[4:0]
vga_pHSync
vga_pVSync
one
dout[0:0]
Constant
rgb2vga_0
vid_in
PixelClk
vga_pRed[4:0]
vga_pGreen[5:0]
vga_pBlue[4:0]
vga_pHSync
vga_pVSync
RGB to VGA output (Pre-Production)
v_axi4s_vid_out_0
video_in
vtiming_in
aclk
aclken
aresetn
fid
vid_io_out_ce
vid_io_out
vtg_ce
locked
overflow
underflow
status[31:0]
AXI4-Stream to Video Out
v_tc_0
vtiming_in
clk
clken
det_clken
gen_clken
resetn
vtiming_out
fsync_out[0:0]
Video Timing Controller
v_vid_in_axi4s_0
vid_io_in
vid_io_in_ce
aclk
aclken
aresetn
axis_enable
video_out
vtiming_out
fid
overflow
underflow
Video In to AXI4-Stream
dvi2rgb_0
TMDS
RefClk
aRst
pRst
DDC
RGB
PixelClk
aPixelClkLckd
DVI to RGB Video Decoder (Sink) (Pre-Production)
clk_wiz_0
clk_in1
reset
clk_out1
locked
Clocking Wizard
zero
dout[0:0]
Constant
hdmi_in
sys_clock

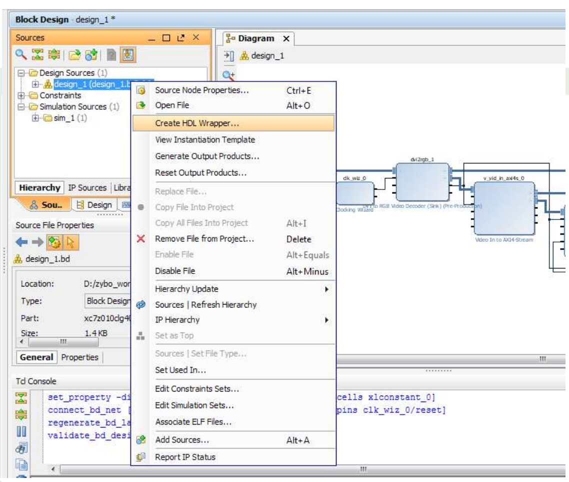


Validating the block design

Validate the design by clicking on the corresponding button on the toolbar on the left

There should be no errors reported

If there are, revisit the wiring between blocks

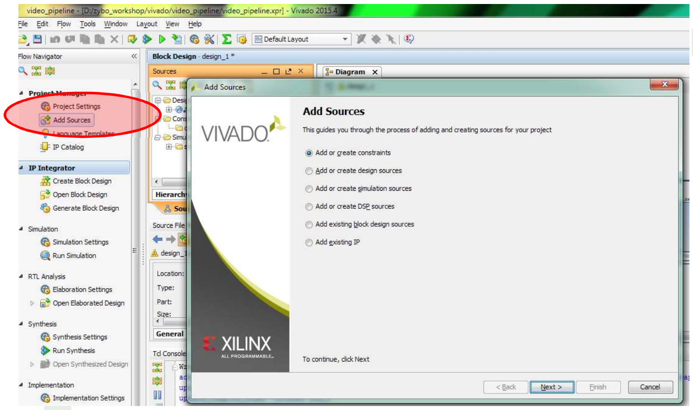


Generating HDL Wrapper

Right-click on the block design source file in the project hierarchy and choose Create HDL Wrapper.

Let Vivado manage the HDL wrapper by clicking OK.

All that is left is adding the constraint file which tells the synthesis tool about physical constraints for the design like which FPGA pin to use for each interface and timing constraints like the maximum frequency for the DVI pixel clock.



Importing constraints

Click the Add Sources button on the left toolbar

Choose Add or create constraints

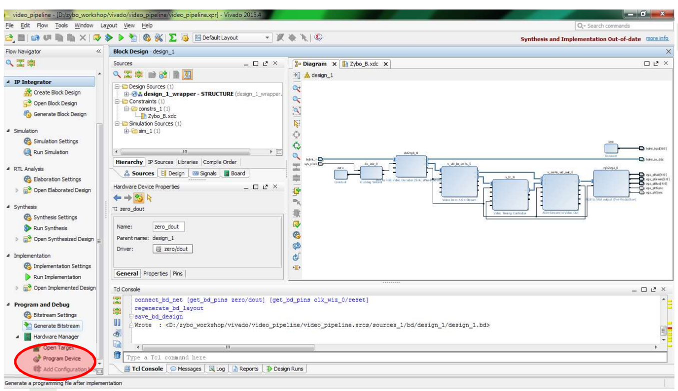
Click Next

Click Add files

Browse to the provided Zybo_B.xdc

Make sure the "Copy constraints files" option is ticked.

Click Finish

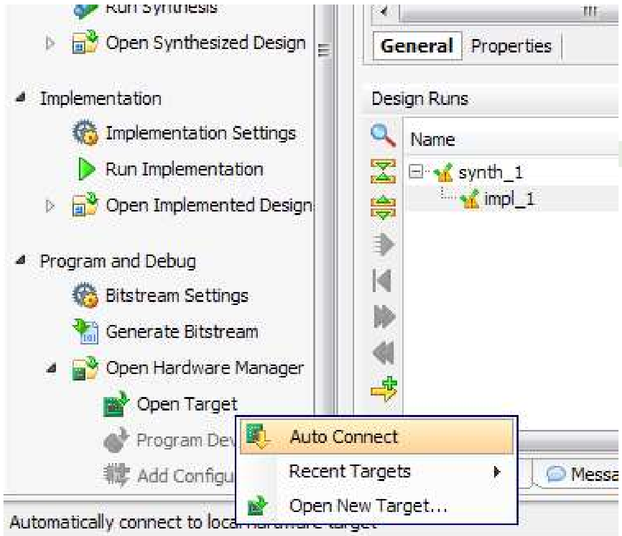


Generating Bitstream

Click Generate Bitstream in the Flow Navigator on the left.

If asked, save the design and confirm that synthesis and implementation should be run.

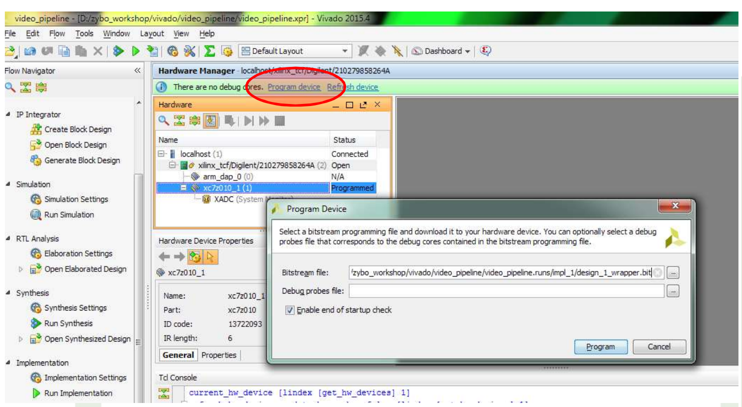
When bitstream generation is completed, choose "Open Hardware Manager", which is also accessible in Flow Navigator.



Program hardware with bitstream

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

Choose Open Target and Auto Connect from the Flow Navigator on the left



Program hardware with bitstream

Click on Program device on the top

Click Program to download the bitstream file shown there to the Zybo

The green DONE LED on the Zybo should light up

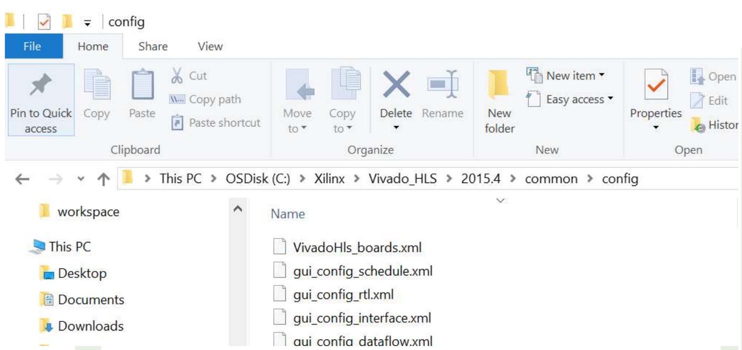
The board is now ready to forward video input on its DVI port to VGA. Connect the Zybo to an HDMI source like a laptop and to a VGA monitor. The laptop should recognize it as a display and you should be able to extend your desktop to it. The extended desktop should be forwarded by the Zybo to the VGA monitor.

6 Task Three – Edge detection in HLS

The video pipeline created in Task 2 provides a good basis for image processing functions defined in HLS. The bus between blocks “Video In to AXI4-Stream” and “AXI4-Stream to Video Out” is a streaming interface sending data pixel-by-pixel in raster format. While it may seem unnecessary to convert the RGB video data to AXI-Stream then back, this step ensures the greatest interoperability between IPs. The RGB video stream is a continuous stream of pixels forming lines interleaved by blanking intervals. It lacks 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. 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.



Add Zybo board definition to Vivado HLS

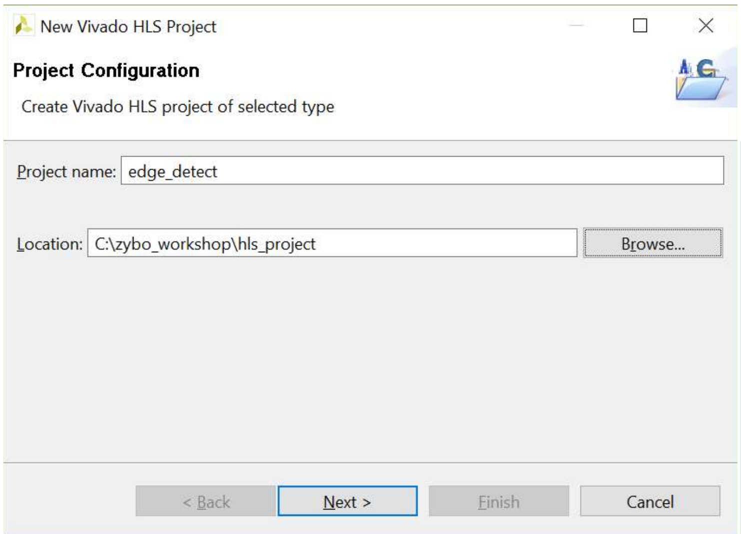
Browse to your Vivado_HLS installation folder.

For example, on Windows: C:\Xilinx\Vivado_HLS\2015.4\common\config

Or on Linux:

/opt/Xilinx/Vivado_HLS/2015.4/common/config

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



New Vivado HLS project

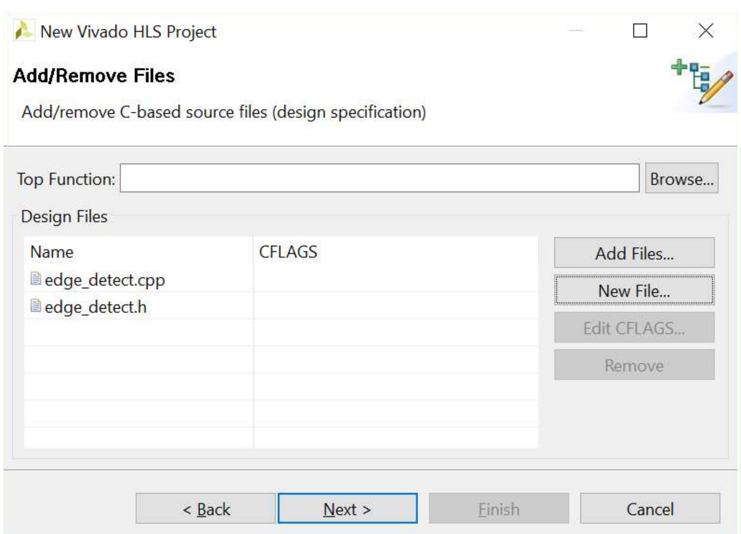
Launch Vivado HLS 2015.4 (NOT Vivado 2015.4) from the Start Menu

Click Create New Project

Name the project edge_detect

Place it under zybo_workshop\hls_project

Click Next



Create new source and header files

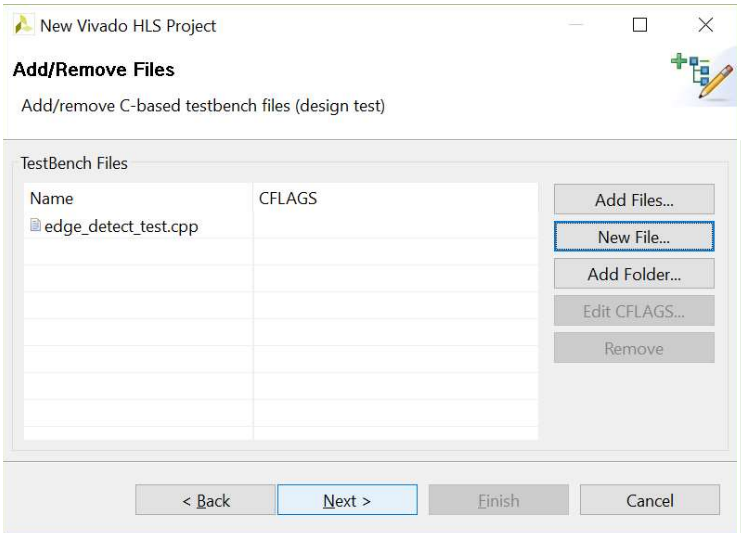
Click New File

Browse to zybo_workshop\hls

Name it edge_detect.cpp

Repeat for edge_detect.h

Click Next



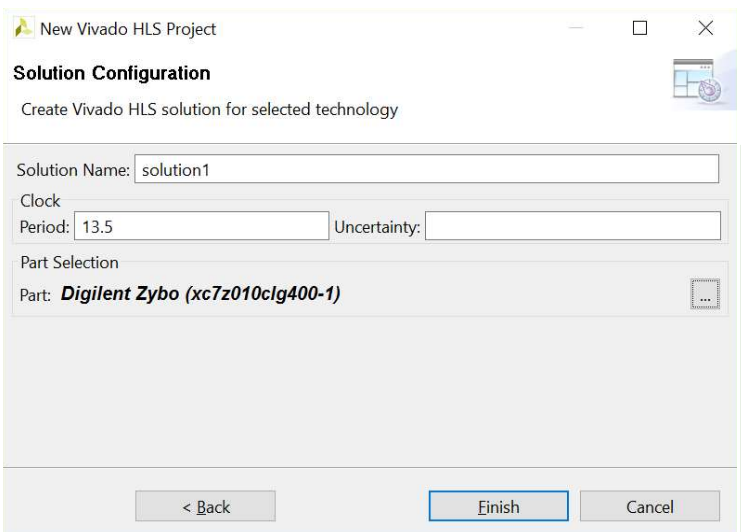
Create new source file for test bench

Click New File

Browse to zybo_workshop\hls

Name it edge_detect_test.cpp

Click Next



Create project constraints

Enter 13.5 for clock period

Click the browse button for part selection

Click Boards

Choose Digilent Zybo in the list of boards

Click Finish

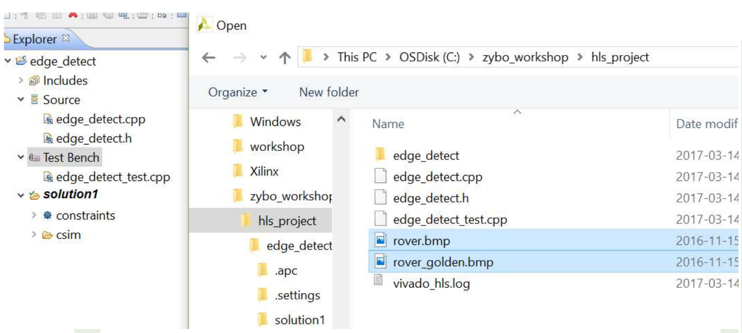
Now that the project is created we can get on with writing actual C++ code. The following files will be written.

```
#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_HEIGHT 720
#define MAX_WIDTH 1280

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

#define INPUT_IMAGE "rover.bmp"
#define OUTPUT_IMAGE "rover_output.bmp"
```

```
#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);
  rgb_img_t img1(rows, cols);
  rgb_img_t img2(rows, cols);
  rgb_img_t img3(rows, cols);
  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);
}
```

```
#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;
}
```

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.

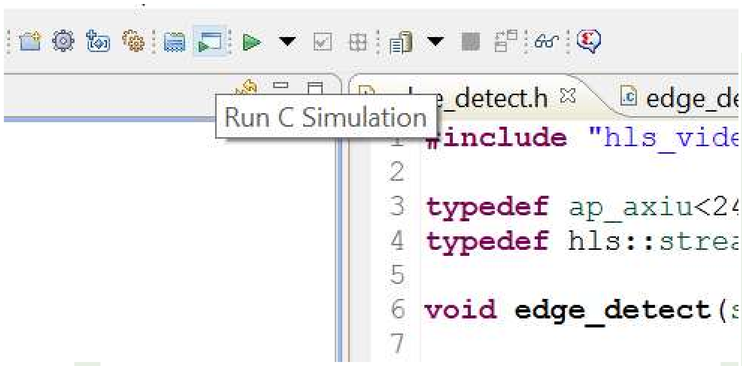


Add testbench images

Right-click on Test Bench

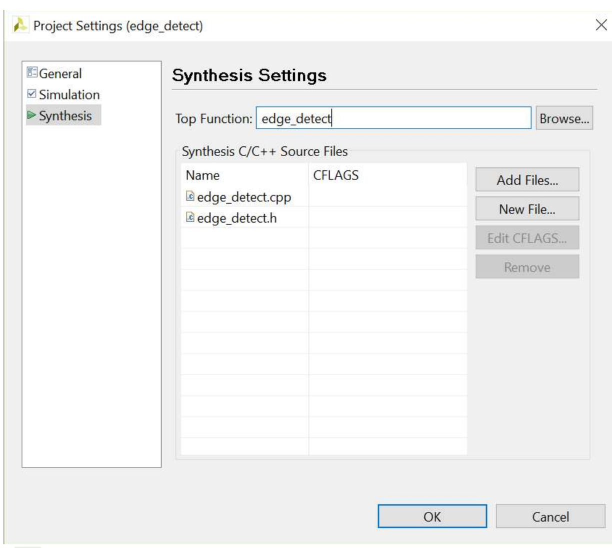
Choose Add Files

Select both rover.bmp and rover_golden.bmp to be added from zybo_workshop/hls_project



Run C Simulation

Click on the Run C Simulation button in the toolbar.



Setting top-level function to synthesize

Click on the Project Menu

Choose Project Settings

Choose Synthesis on the left

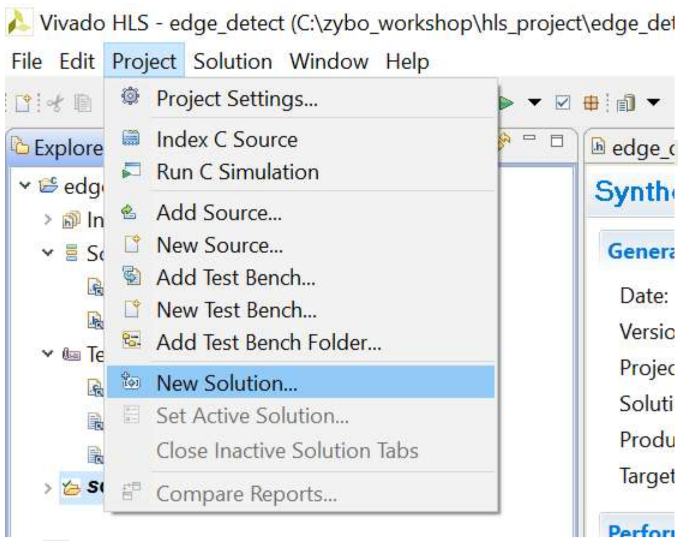
Click Browse next to Top Function

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.

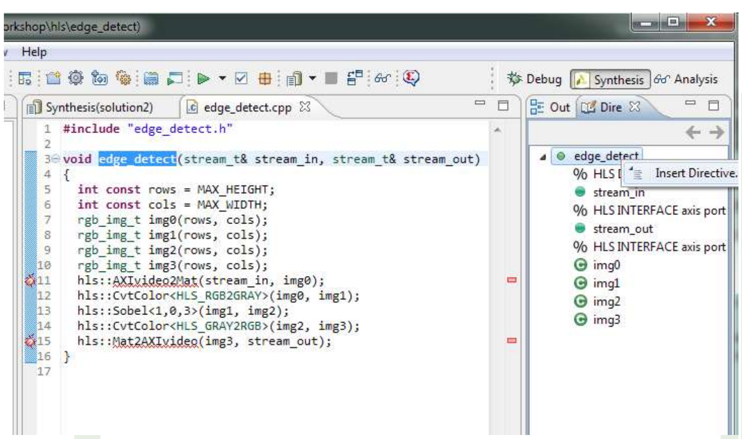


Create a new solution

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.



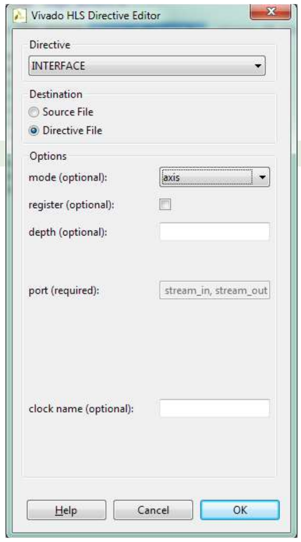
Setting directives

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



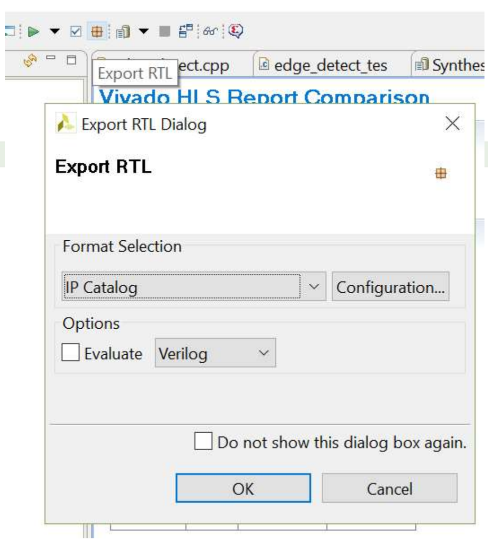
Choosing directive options

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.

Similary, 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.



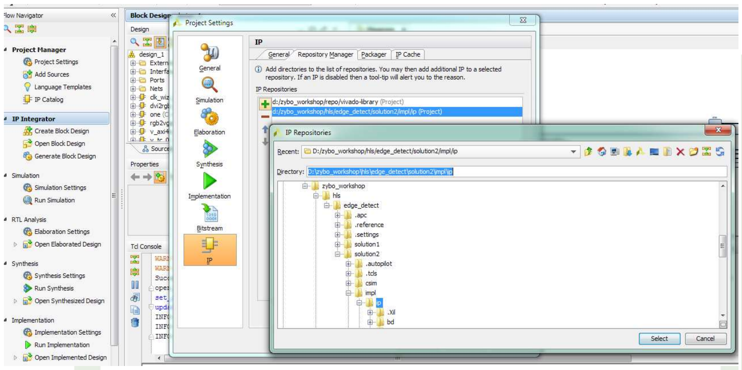
Package and export IP

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.



Adding HLS IP to the video pipeline project

Switch back to the video_pipeline project in Vivado 2015.4 we created in task two.

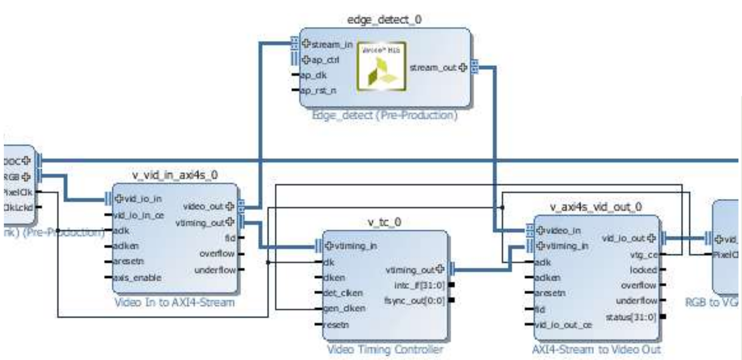
Click Project Settings on the left toolbar

Select IP and Repository Manager

Click the green plus sign

Browse to the HLS project path zybo_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

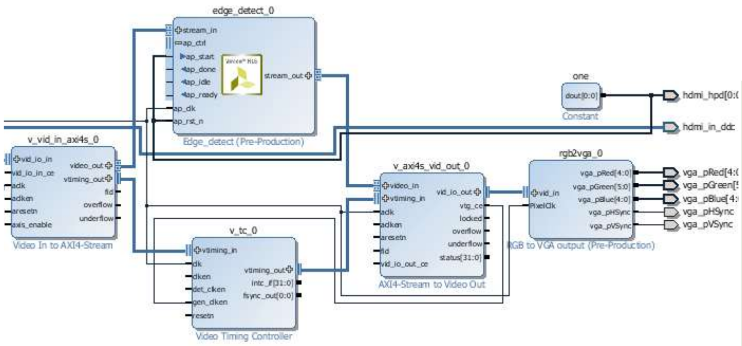
Search for and double-click on Edge_detect, which is our HLS IP

Click on the wire between v_vid_in_axis4s/video_out and v_axi4s_vid_out/video_in

Press the Delete key

Wire video_out to stream_in of edge_detect

Wire stream_out to video_in



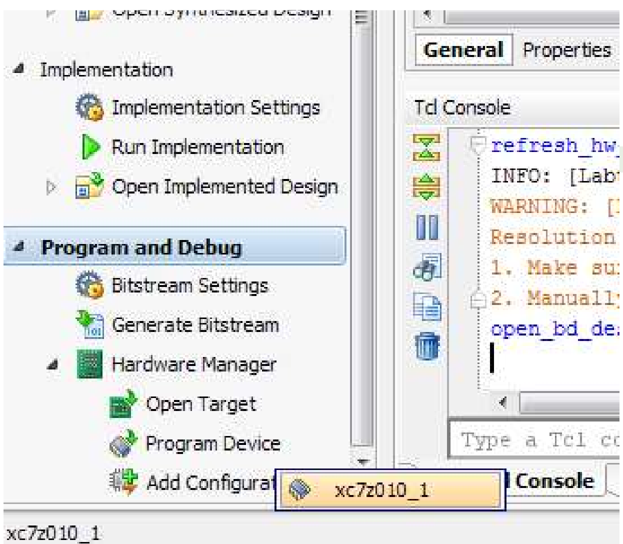
Wiring control bus

Wire ap_clk of edge_detect to the pixel clock of the pipeline (PixelClk of dvi2rgb)

Click the plus icon next to ap_ctrl

Wire ap_start to the "one" Constant block

Wire ap_rst_n to the same "one" Constant block



Download bitstream

Generate bitstream

Click Program Device under Hardware Manager and choose xc7z010.

Click Program to download to programmable logic on the Zynq.

Zybo should now forward incoming video data to the VGA after applying the edge detection algorithm on it. Display any image, movie or just the Windows desktop on the secondary display to see edge detection in action.

This concludes our workshop. Thank you for attending!