ECE 5727

Lab 4: HLS: Video Processing

Lab Point Value: 10
Post-Lab Point Value: 40, +5 bonus

Objectives

In this lab you will:

1. Create a real-time video processing pipeline that uses a Sobel filter to highlight edges in the video
2. Work through the process of creating an algorithm using an HLS design flow, and adding the resulting RTL to an FPGA design

Lab Overview

High-level Synthesis (HLS) is the usage of normal programming languages (C, C++, SystemC) to describe a desired algorithm from which an RTL design can be extracted. This allows the designer greater abstraction and ease of implementation of complex algorithms, such as those used in image processing, than HDLs (VHDL, Verilog, SystemVerilog) can provide.

Xilix and Digilent partnered to create an example of using HLS to implement video processing on their Zynq chip. For this lab, you will be working through this example to gain a high-level understanding of how HLS is used, and how it differs from normal C programming.

Instructions

Workbook.pdf contains instructions with images of each step necessary. Note however, those instructions were created with a slightly older version of Vivado, so when in doubt, refer to my instructions which reflect doing all this on Vivado 2018.3. I have also attempted to include a little more information in my version of the instructions.

Parts 1-3 of *Workbook.pdf* are an overview of FPGAs and HLS. You can read this at your leisure. Part 4 uses the example HLS project that I went through in Lecture. You can skip this, as the remainder of the Lab will take you through everything this covers in more detail, except for running a C/RTL Co-Simulation.

The lab begins in Part 5 of the Workbook. Download and extract the lab from the .zip file on Canvas, or clone the repo. Assume this location is the base to all future referenced paths.

Pass-Through Video Pipeline

Set Up Vivado Project

The first thing we need to do is set up the Vivado FPGA project. To do this, open **Vivado 2018.3**, then click *Tools > Run TCL Script...*

Select *.../vivado_project/proj/create_project.tcl* and click OK. It will take a couple minutes to run.

Once finished, you will be left viewing the block diagram for the video pipeline. Currently it is setup as a pass-through, reading video frames from the camera and then outputting them unaltered over the HDMI TX connection on the board. Refer to the Workbook for descriptions of each of the blocks' functionality. Some are Xilinx IP Cores, others are Digilent IP Cores, and others are just RTL files from Digilent.

Set up SDK Project

With the FPGA project setup, now the SDK project needs to be created. Close the Board Design (click save if it asks), then click *File > Launch SDK*. Use the following settings:

- Exported Location: *.../vivado_project/hw_handoff*
- Workspace: *.../vivado_project/sdk*

The *Exported Location* is where the SDK looks for the information exported by Vivado about the chip being used. The *Workspace* is where the SDK stores all the projects.

Click OK, then Yes when asked about the export being out-of-date.

At this point, the Workbook expects the SDK to have opened and discovered the projects. If pcam_vdma_hdmi and pcam_vdma_hdmi_bsp are not appearing in the Project Explorer:

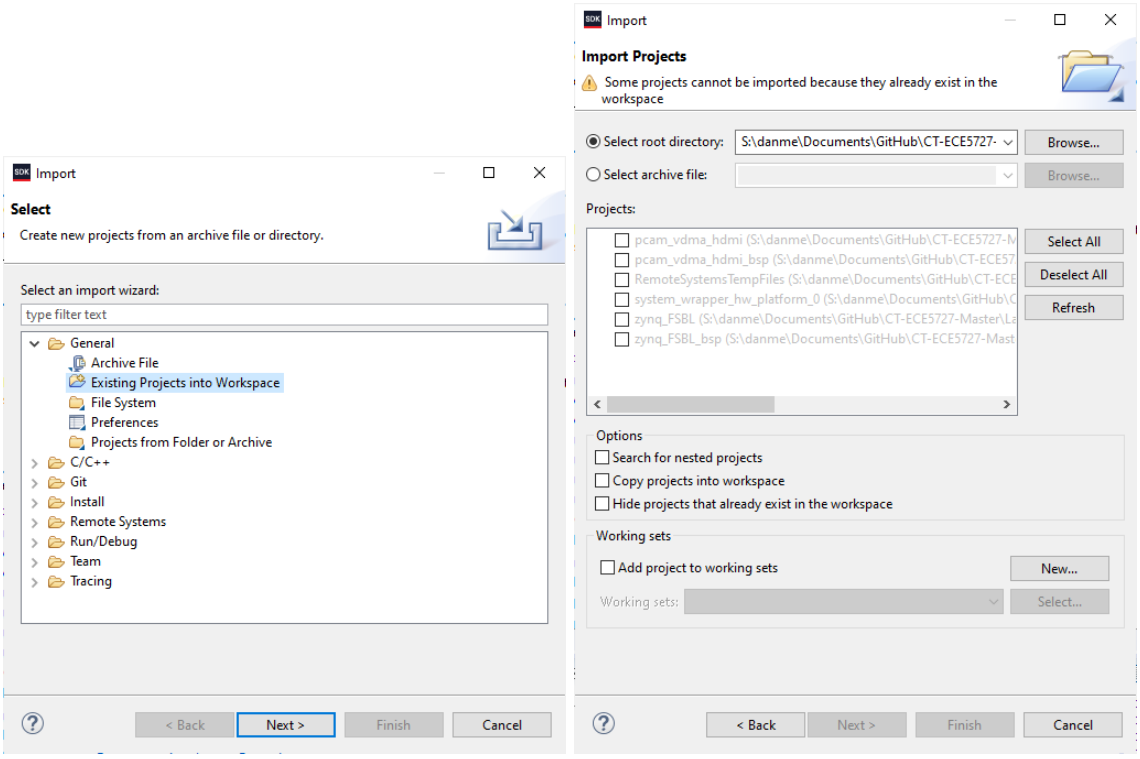
1. Right click in the Project Explorer, then select *Import...*. In the pop-up, select *General > Existing Projects into Workspace*. Click **Next**.
2. Browse to the workspace directory (*.../vivado_project/sdk*) to select as root directory.
3. Make sure the pcam project and pcam bsp are selected under *Projects*, then click **Finish**.

If the project does not automatically generate the board support files and compile, click *Project > Build All* or *Ctrl+B*.

Run Video Pipeline

It is now time to set up the Zybo-Z7 and run the pipeline on it.

- If you'd like to power the board via the power cord, instead of the USB, plug in the wall adapter and power cord, and move the jumper to select WALL instead of USB.



Import Projects into SDK

- Plug the camera in. To do so, pop up the white clip, then slide in the end of the ribbon with the blue-side facing IN. This means the camera is also facing IN. Then push the retaining clip back down to secure the connection. Do not force anything. Remove the lens cover from the camera (yes, there is a cover and yes, I missed it and got very confused when I first went through this and only saw a black screen).
- Connect the USB cord between your computer and the board.
- You will need an HDMI cord and monitor/tv. Plug in the HDMI cable. If you cannot connect the HDMI cable to the board while the USB cable is plugged into your computer (because its not near the monitor/tv), follow the instructions posted to Canvas for how to store the program files to the board for use on power-up. You can then unplug the USB cable, and use the wall power near your tv/monitor to power the board. If you have problems, please reach out (over Piazza so the answer is available to everyone).

Power on the Zybo, then right click on the pcam_vdma_hdmi project and select *Run As* > *Launch on Hardware*. Your choice of System Debugger or GDB. Click **No** when it prompts about the FPGA not being programmed. We are going to configure the run to automatically program the FPGA everytime we run, so that we don't have to remember to do it ourselves.

Go to *Run* > *Run Configurations...*. In the left-hand panel, select the configuration you tried to run, then tick both *Reset entire system* and *Program FPGA*. Click **Run**.

You should now see video streaming from the camera to your display! ***Submit a picture or short (1-5 sec) video of it working*** (*5 points*)

With the configuration set, subsequent runs can be done just by clicking the green Run button (if you haven't stored the program to the Zybo flash).

Edge Detection

Set Up HLS

With the video pipeline working, it is now time to create our edge detection algorithm in HLS and insert it. *The Workbook says that to speed things up later on, you can go back to Vivado and* ***Generate Bitstream*** *for the FPGA. While possible, this is very resource intensive and will make doing the HLS stuff very slow while it is running. I only recommend doing this with a desktop or non-VM laptop. With that running in the background...*

We first need to add board files for our HLS project, just as like we needed to add board files when we first set up Vivado. From the Lab files, find *VivadoHLS_boards.xml*. Navigate to where Vivado was installed (e.g. C:/Xilinx/Vivado on Windows, /tools/Xilinx/Vivado on Linux), and copy the file into *./2018.3/common/config* to overwrite the existing file there.

Launch **Vivado *HLS* 2018.3**, not **Vivado 2018.3**.

Create a New Project

1. Name the project edge_detect
2. Select *.../hls_project/* as the Location
3. Click **Next**
4. Add *edge_detect.cpp* and *edge_detect.h* from *.../hls_project/*. **Next**
5. Add *edge_detect_test.cpp*. **Next**
6. Our pipeline operates at 150 MHz, so enter 6.67 for the clock period.
7. For Part Selection, Browse and select Boards, then the Zybo-Z7
8. **Finish**

The HLS project is now created, and we have added the source and testbench files we need for it. Take a look through the three files added and try to understand what each is doing.

To run the testbench, we need to a test image. Right click on Test Bench in the Project Explorer, then choose *Add Files*. Select *fox.bmp*. Click *Run C Simulation* from the toolbar. Leave all the options unchecked. Take a look at the output image created: (*.../hls_project/edge_detect/solution1/csim/build/fox_output.bmp*). Compare it to the input image. How'd it do? For reference, *fox_golden* is what the outputs should look like. If you'd like to do different images, you can add them as well, and then just change the value of INPUT_IMAGE in the header file.

Generate RTL Algorithm

Solution 1
To generate the RTL, we need to let the program know what our top-level function is. Select *Project Menu* > *Project Settings* > *Synthesis*. For Top Function, browse to select edge_detect. Click **OK**, then click **Run C Synthesis** from the toolbar. It will run for a couple minutes.

In the Project Explorer tab, under **solution1**, there are now two new folders: *impl* and *syn*. For now, the implementation folder just holds the generated RTL files. The Synthesis folder holds the same files, but it calso contains a report folder, detailing the synthesis results for each submodule. The report for *edge_detect* should have opened automatically.

Notice under the timing summary that we appear to have not hit our timing requirement (8.3ns > 6.67ns). This means that based upon the generate RTL, the compiler believes there is a part of the design that requires a clock with period >= 8.3ns, and could not find a way to optimize the RTL to meet the 6.67ns requirement. Notice also that there are 3 generated instances (Detail>Instance) in addition to our top-level: *Filter2D, AXIvideo2Mat* and *CvtColor*.

Take a look at the **Interface** section at the bottom of the report. Here we see that the compiler has decided to use two interface standards: *ap_ctrl_hs* and *ap_fifo*. The first is a handshake protocol to control flow. The second indicates that a fifo will be used to send the data in/out. However, we are intending to use the AXI-Stream protocol to get our video frame data in/out of the module. We need to inform the compiler of this, through the use of directives.

Solution 2 - Add Directives
Create a new solution, so that the existing solution results are available for comparison. Select *Project* > *New Solution...*. Leave everything as the default, which will copy all current settings from solution1 to the new solution2. Click **Finish**. Doing so sets solution2 to be the active solution. We can always switch back to solution1 by right-clicking it in the Explorer tab, and selecting to set it as active.

With solution2 as the active solution, any directives we add/change will apply to this solution instead of solution1. Begin by opening *edge_detect.cpp*. In the right-hand panel, select the **Directives** tab. Ctrl+left-click on *stream_in* and *stream_out* to select both at once. Right-click the selection, then choose *Insert Directive*. Choose INTERFACE as the directive, then choose *axis* as the optional mode. Click **OK**.

We have now told the compiler that we want the two pointers that are parameters to our function to use the AXI-Stream interface. However, that only handles the mis-assignment of the ap_fifo interface type. We also need to take care of the compiler making the module itself have a control interface. We will not be passing any control signals in/out of the module beyond the stream data itself. As such, select the function itself, *edge_detect* from the Directives tab to apply a directive. Select INTERFACE, then for the option mode, set it as *ap_ctrl_none* and click ok. This tells the compiler to generate no control interface for the module.

If you go back to the synthesis report for solution1, you will see that our design has a latency and iteration interval (II) of 4.6 million. This means that each frame in the design will be delayed by 4.6 million clock cycles, and that we can only accept a new frame every 4.6 million clock cycles. With a clock frequency of 150MHz, this implies we can only process a little over 30 frames per second, which is decent but not great. However, we can tell the compiler to pipeline the data, allowing for new data enter and begin processing while earlier data is still being processed further on. E.g. we can begin doing the RGB to Grayscale conversion on a new frame once we have finished the previous frame and begun passing it through the Sobel filter. We do this through the DATAFLOW directive. Select the edge_detect function again, and apply this directive.

It is time to see if our directives have improved the solution to achieve our desired results. Click on *Run C Synthesis* and wait for the RTL to be generated. Once the compilation finishes, the report for solution2 synthesis will open. Unfortunately, it appears the timing requirement has not improved - it is still not being met. However, we can see that our interfaces are now correct: stream_in and stream_out are both assigned to the axis protocol, and the module edge_detect is only getting clock and reset signals for the ap_ctrl_none protocol. Additionally, there are instances that make much better sense now: Sobel, AXIvideo2Mat & Mat2AXIvideo, and two CvtColors (for RGB2GRAY and GRAY2RGB).

Export RTL to Vivado

It doesn't seem like there is much we can do about the timing, and that is only an estimate anyway. Lets see what happens if we actually try to use this design. *The Workbook seems to indicate that the timing estimate should be saying we are meeting spec - with 2018.3 it seems that will not occur, but don't worry!* Click on *Export RTL* in the toolbar. Make sure Verilog is the evaluated RTL and Vivado Synthesis is checked (not synthesis, place and route). Click **OK**.

Again, this will take a couple minutes to run. By checking 'synthesis' as part of the export options, we are having the compiler actually take the RTL design through synthesis completely, rather than just analyze and elaborate the RTL. The results of this are stored in the 'impl' directory, and when complete, the Implementaion report will automatically open. The report shows an updated timing estimate, and indicates that timing will be met.

Switch back to our Vivado FPGA project, and open the block design. Vivado HLS exported our algorithm as an IP Core, and so it is necessary to tell Vivado where to look for it. Select *Tools > Settings... > IP > Repository*, then click the plus sign. Browse to *.../hls_project/edge_detect/solution2/impl/ip* and then click **OK**.

Go to the block diagram and right click to *Add IP*. Search for, then add **Edge_detect**. This will add our HLS algorithm into the diagram. The last thing we need to do is insert it in the pipeline between the VDMA and the Video Out blocks.

Select and delete the wire connecting axi_vdma_0.M_AXIS_MM2S and video_out.video_in Now connect the edge detect algorithm block.

- stream_in: axi_vdma_0.M_AXIS_MM2S

- stream_out: video_out.video_in
- ap_clk: axi_vdma_0.m_axis_mm2s_aclk
- ap_rst_n: axi_vdma_0.axi_resetn

Save and then **Generate Bitstream**. If you did not run this step earlier, it will take roughly 15-30 minutes while it synthesises all the components. If it was run earlier, it should only take a few minutes, as the synthesis results for the other blocks will have already been cached.

Run Edge Detection

Once the bitstream is generated go to *File > Export > Export Hardware*. Be sure to include the bitstream, and that you are exporting *.../vivado_project/hs_handoff/*. Click **OK** and confirm overwriting existing hardware.

Switch back to the SDK workspace. Right click on the system_wrapper project, and select *CHange Hardware Platform Specification*. Click **Yes**. Make sure the Hardware Specification File is set to *.../vivado_project/hw_handoff/system_wrapper.hdf*. Click **OK**.

Go to *Project > Clean...* and select to clean all projects. Click **OK**.

Once the projects finish building, you can run the new program on your Zybo! You should see video streaming from the camera, with edges being highlighted, like with the fox.bmp image (although less clean probably). ***Submit a picture or short (1-5 sec) video of it working*** (*5 points*)

Post Lab

Please submit answers to the following questions

1. (*10 points*) In this design, the video pipeline is being handled by the FPGA. Take a look at the software code. What purpose does the software serve in this design? Do you think this is a good/necessary division of labor?
2. (*20 points*) Take a look at the analysis for the HLS Solution 2.
   (a) (*5 points*) What is the latency for the algorithm? With a 150MHz clock (6.67ns period), how much does the algorithm delay the image?
   (b) (*5 points*) What is the iteration interval (II) of the algorithm? This is the number of clock cycles that must elapse between inputs. With a 150 MHz clock, what is the maximum frame rate (fps) we can handle? Is this an improvement over Solution 1?
   (c) (*5 points*) Which part of the algorithm appears to be the limiting factor for latency and II?

(d) (*5 points*) Both solutions seemed to indicate there was a timing error within one of the Cvt_Color modules post-RTL generation. But, when we exported the design, the timing error disappeared. Why do you think this is? Did anything else change between the initial analysis and post-export results?.

3. (*10 points*) Look at the FPGA post-implementation results

   (a) (*5 points*) Did our design meet timing? What is the WNS (positive or negative) and what is the path?

   (b) (*5 points*) What is the utilization of the FPGA's resources? How much of this is from the edge detect algorithm.

4. *Bonus Question:* (*5 points*) Take a look at the relationship between the II for solution 1 and solution 2. If you add up the individual module's IIs in solution 2, they roughly equal the overall II for solution 1. By adding pipelining, we can process a different frame within each module, reducing the II to the greatest II of any individual module, rather than have it pass all the way through all of them as in solution 1.

   The latency for solution 2 is also decreased, which I do not have an explanation for. I expected it to still take roughly the same amount of time as solution1. Instead, the overall latency in solution2 seems to be only a few clock cycles more than the max latency of the instances. Does this make sense or not to you? Why or why not?