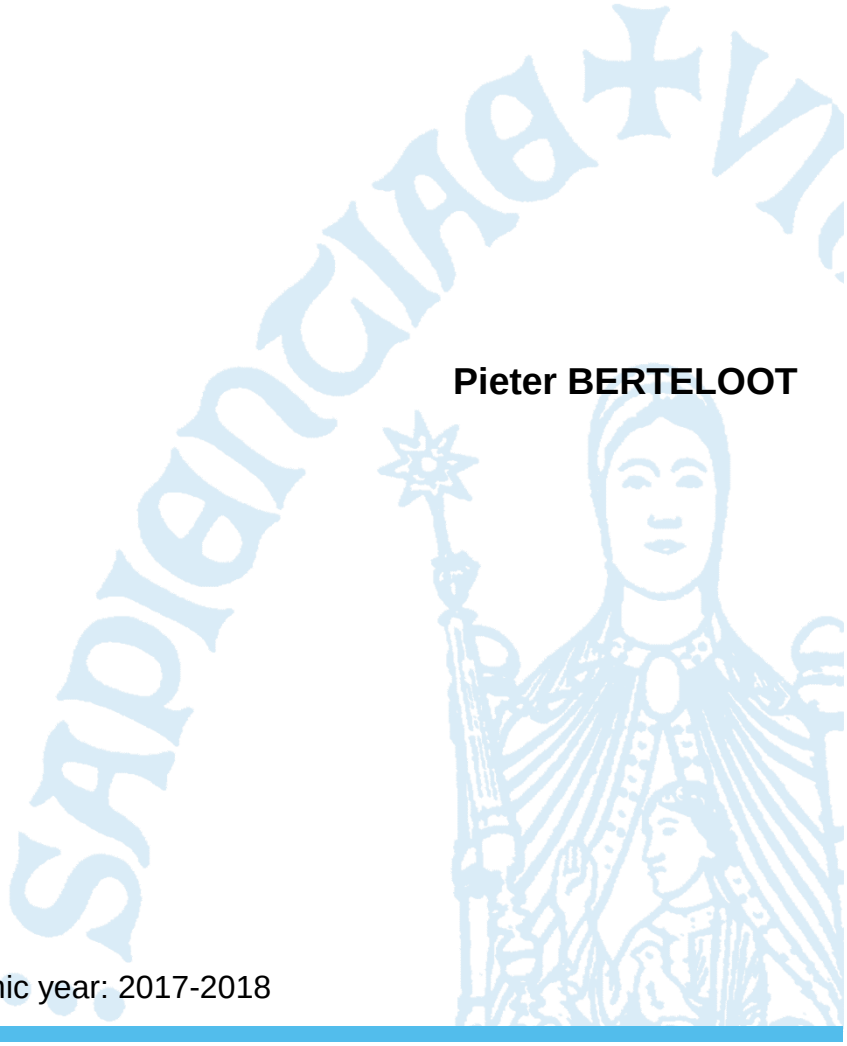
KU LEUVEN

FACULTEIT INDUSTRIËLE INGENIEURSWETENSCHAPPEN

CAMPUS BRUGGE

Hardware acceleration of video processing

Pieter BERTELOOT



Supervisor: Sammy Verslype

Team number: xohw18-279

Academic year: 2017-2018

Abstract

This project is a study on accelerating software processes using hardware resources. Hardware is used to perform functions faster and more time efficiently than is possible in software running on a processor. The focus in this project is placed on video processing.

A PYNQ board is used to perform the hardware acceleration. This board is equipped with a Xilinx Zynq® All Programmable Systems on Chips (APSoCs). This is a combination of a field-programmable gate array (FPGA) and a cortex A9 processor along with other peripherals. It can be used for parallel hardware execution, real-time signal processing, high frame-rate video processing, ….

Overlays are used to make this possible. These are programmable and configurable FPGA designs. PYNQ provides a Python interface that allows overlays to be controlled in the processing system. A custom video overlay is created that reads an incoming HDMI signal, processes it and sends it back to the HDMI output.

The processing of the signal is done in a custom Intellectual Property (IP) block, which is made using Vivado high level synthesis (HLS). This program is an automated design process that implements C code in hardware logic. HLS provides extensive libraries for data types, video processing, DSP and much more, which can be used to accelerate and optimize projects. Directives can be given for user-specified optimization such as pipelining, loops, ….

In this project, a wide variety of IP's are made. Starting with basic functions, to get to know HLS, to more complex functions such as edge detection. Edge detection is identifying points in a image where discontinuities occur. To perform this edge detection, the Sobel filter is used. This filter is based on convolving a frame with a specific kernel. This kernel defines in which direction the detection should be (X or Y). Edge detection is used for image segmentation (dividing an image into multiple parts) and data extraction for computer/machine vision.

To speed up the IP creation and optimization, a test bench is created that makes C simulation possible. This simulation loads a local image, creates an Axi stream with the image data and feeds this to the IP. This outputs an image which can be used to check a correct working functionality.

Two final IP's were made based on convolution with kernels. The first IP is a Sobel_IP. Here the user can use the Sobel filters to modify the video signal. The second IP is a Sharpening_IP which uses the Laplacian high pass filter to sharpen the video signal.

Finally, the comparison is made between a software and hardware implemented video filter. Here the conclusion can be made that it is possible to accelerate software processes using the PYNQ board. The video signal can be processed, keeping the frame rate of 60fps with only a low latency between the input and the output signal.

All created IP's, accompanied by input and output images, are described in this report and can be used as a beginner's guide for creating PYNQ overlays. Chapter 1 is an introduction of the project, chapter 2, 3 and 4 are a bundle of created IP with the intend to learn more about IP and overlay creation. In chapter 5 this information is used to create 2 final IP's. Chapter 6 compares the hardware acceleration with the software process. And the last chapter gives a conclusion about this project.

Table of Contents

List of Figures

1 INTRODUCTION

1.1 Objective

The objective of this project is to study the use of FPGA hardware resources to accelerate software processes with focus on video processing. All elements on how the project is realized are discussed in this report and can be used as beginner guide for new PYNQ [1] users. All HSL, bit, Tcl and Python files are available on github:

https://github.com/Pieter-Berteloot/PYNQ_Projects

This project makes use of the PYNQ-Z1 board. This board is the hardware platform for the PYNQ open-source framework. This includes ARM A9 CPUs where the following software runs:

- Linux
- Python
- Jupyter notebook
- Hardware libraries and API for the FPGA

These are used to create a user-friendly and customizable video processing system.

Hardware libraries are the programmable logic circuits and are called overlays. These are like software libraries. The programmer can select which one matches their application the best. The advantage of using these overlays is that once an overlay is build, it can be reused in other applications.

1.2 Setup

For the setup, a laptop is used to generate a 1080p HDMI video signal. This video signal is connected to the PYNQ board in HDMI IN. Here the signal is processed and send back out through the HDMI OUT connector. This connector is connected with a HD 1080p monitor.

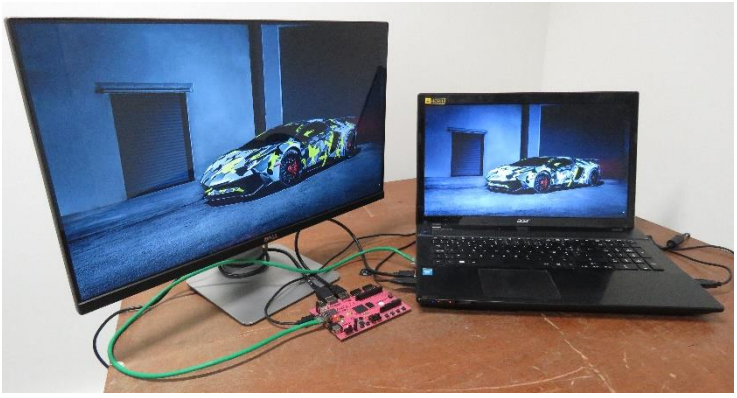


Figure 1-1: setup

2 OVERLAYS

Overlays are programmable and configurable FPGA designs. These are used to accelerate software applications. PYNQ provides a python interface that allows overlays to be controlled in the processing system.

An overlay includes:

- Bitstream file

File that contains the programming information for the FPGA.

- Tcl file

Determines the available IPs

- Python API

Handles the configuration and communication with the IPs

The default base overlay is loaded at boot time on the PYNQ board. This overlay can be replaced with other overlays while the system is running.

2.1 Base overlay

The base overlay allows PYNQ to use the hardware peripherals (video, audio, GPIOs, ...) that are on the board. It connects the IP blocks to the Zynq processing system. These peripherals can then be used from the Python environment. Let's now look what's inside the base overlay. To do this, rebuild the overlay following these steps:

- First clone/download the board files and overlays from the PYNQ github page: https://github.com/Xilinx/PYNQ.

- Open Vivado Design Suite (for this project Vivado 2016.2 is used) and run the following code in the TCL console:

```
cd <PYNQ repository>/boards/Pynq-Z1/base
vivado -mode batch -source build_base_ip.tcl
vivado -mode batch -source base.tcl
```

- Wait until both scripts have finished (this will take some time). When this is done the base overlay can be found in:

```
<PYNQ repository>/boards/Pynq-Z1/base/base
```

This base overlay will be used as a starting point for our project because it already defines all the configurations needed for the processing system interface and the peripherals. An important part in the block design of the overlay is the processing system AXI peripherals. This is a General-Purpose AXI-Lite interface (GP0) that controls and configures IP blocks in the design and runs on a 100MHz clock. [2] [3]

2.2 Rebuilding the base overlay

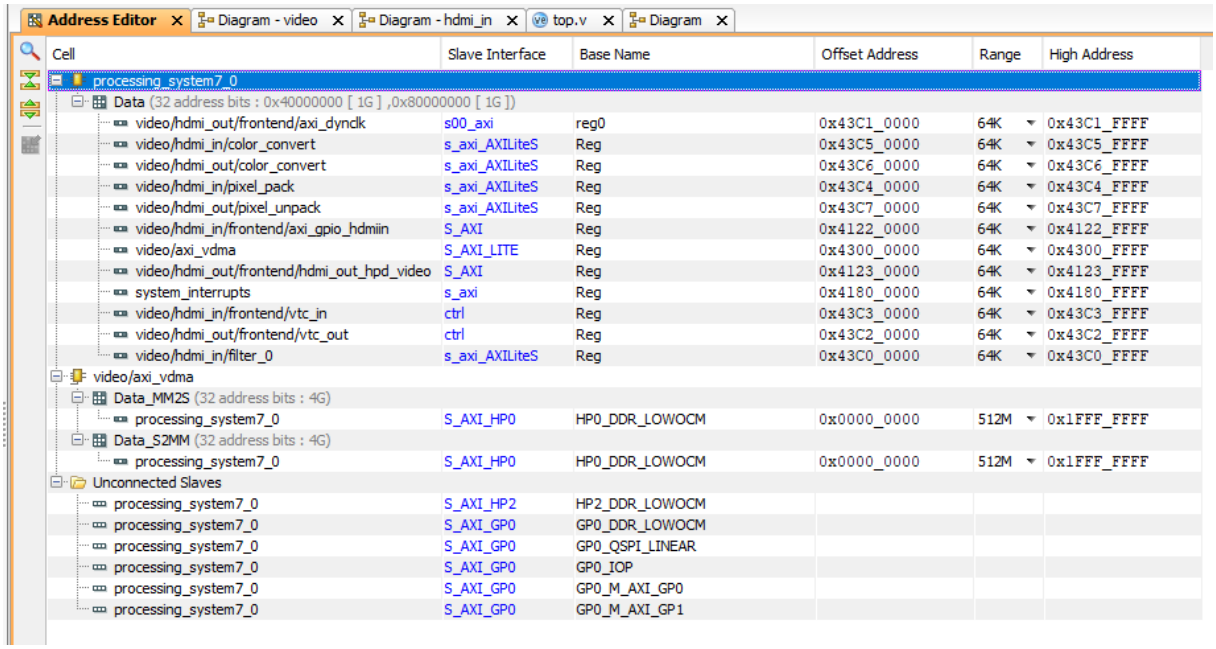
Every peripheral can be found in the base overlay. The routing of these blocks takes time and hardware so to reduce this, only the components that are needed for video streaming/processing are kept. Our edited base overlay (Figure 2-2) can be found on the next page.

The following blocks are needed for video streaming:

- AXI interface
- ZYNQ processing system
- System interrupts
- Reset processing system fclk0
- Reset processing system fclk1
- Video

The rest of the IP blocks have been removed from the block design. To prevent errors, the deleted input and output signals are removed from the top.v file that can be found in the project Manager.

The **Address Editor** (Figure 2-1) is also a very important subject in the IP Integrator. The offset Address and range of each IP is displayed here. This address will later be used for **Memory-mapped I/O** (MMIO). When a new IP which has AXI-Lite communication is added, the user has to map the IP to give it an address.



Cell	Slave Interface	Base Name	Offset Address	Range	High Address
processing_system7_0					
Data (32 address bits : 0x40000000 [1G] ,0x80000000 [1G])					
video/hdmi_out/frontend/axi_dynclk	s00_axi	reg0	0x43C1_0000	64K	0x43C1_FFFF
video/hdmi_in/color_convert	s_axi_AXILiteS	Reg	0x43C5_0000	64K	0x43C5_FFFF
video/hdmi_out/color_convert	s_axi_AXILiteS	Reg	0x43C6_0000	64K	0x43C6_FFFF
video/hdmi_in/pixel_pack	s_axi_AXILiteS	Reg	0x43C4_0000	64K	0x43C4_FFFF
video/hdmi_out/pixel_unpack	s_axi_AXILiteS	Reg	0x43C7_0000	64K	0x43C7_FFFF
video/hdmi_in/frontend/axi_gpio_hdmiin	S_AXI	Reg	0x4122_0000	64K	0x4122_FFFF
video/axi_vdma	S_AXI_LITE	Reg	0x4300_0000	64K	0x4300_FFFF
video/hdmi_out/frontend/hdmi_out_hpd_video	S_AXI	Reg	0x4123_0000	64K	0x4123_FFFF
system_interrupts	s_axi	Reg	0x4180_0000	64K	0x4180_FFFF
video/hdmi_in/frontend/vtc_in	ctrl	Reg	0x43C3_0000	64K	0x43C3_FFFF
video/hdmi_out/frontend/vtc_out	ctrl	Reg	0x43C2_0000	64K	0x43C2_FFFF
video/hdmi_in/filter_0	s_axi_AXILiteS	Reg	0x43C0_0000	64K	0x43C0_FFFF
video/axi_vdma					
Data_MM2S (32 address bits : 4G)					
processing_system7_0	S_AXI_HP0	HP0_DDR_LOWOCM	0x0000_0000	512M	0x1FFF_FFFF
Data_S2MM (32 address bits : 4G)					
processing_system7_0	S_AXI_HP0	HP0_DDR_LOWOCM	0x0000_0000	512M	0x1FFF_FFFF
Unconnected Slaves					
processing_system7_0	S_AXI_HP2	HP2_DDR_LOWOCM			
processing_system7_0	S_AXI_GP0	GP0_DDR_LOWOCM			
processing_system7_0	S_AXI_GP0	GP0_QSPI_LINEAR			
processing_system7_0	S_AXI_GP0	GP0_IOP			
processing_system7_0	S_AXI_GP0	GP0_M_AXI_GP0			
processing_system7_0	S_AXI_GP0	GP0_M_AXI_GP1			

Figure 2-1: Vivado Address Editor

Full vidado project: https://github.com/Pieter-Berteloot/PYNQ_Video_overlay

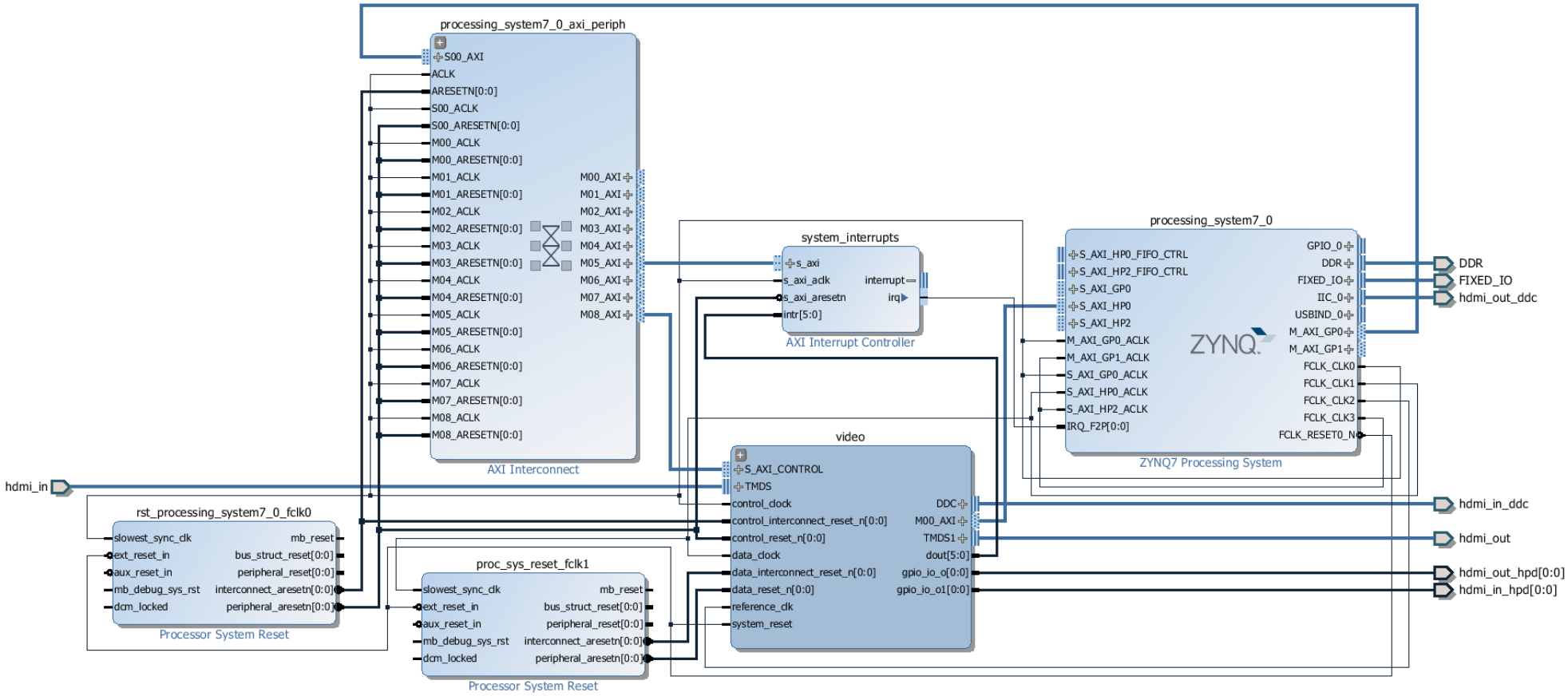


Figure 2-2: Video overlay

2.3 Creating our first IP

When it's known how to create, edit and communicate with the overlay, it's possible to create our own IP. Let's start with a simple adder. The objective is to make an IP that has 2 integer values as input and 1 integer value as output. The user provides these 2 integers and the IP calculates the sum.

For constructing this IP, Vivado High Level Synthesis (HLS) is used. This transforms complex algorithms into VHDL code. It accelerates the IP creation transforming C, C++ and System C code to VHDL code.

When creating a new project using the PYNQ-Z1 board, select the xc7z020clg400-1 board part.

Let's analyze the following code:

```
#include <ap_fixed.h>
#include <ap_int.h>

void add_function( int a, int b, int *c){

#pragma HLS INTERFACE s_axilite port=return bundle=control

#pragma HLS INTERFACE s_axilite port=a bundle=control
#pragma HLS INTERFACE s_axilite port=b bundle=control
#pragma HLS INTERFACE s_axilite port=c bundle=control

      *c = a + b;
}
```

First, start with importing C++ libraries so the Fixed-Point Data Types and Integer Data Types can be used.

The TOP function comes after this. This function is very important because the arguments of the top functions are the interfaces. These will become ports on the RTL design and directives can be specified to select the IO protocol ports. The Axi-Lite protocol is used for communication with the IP. Pragmas are used to tell the program that the Axi-Lite protocol needs to be used.

Consequently, the functionality of the IP is programmed. When this is done, C synthesis and RTL export can be performed.

Now import the IP in our overlay. To do this, import the IP in the IP catalog (project manager -> IP Catalog) and add the IPs repository. Once this is done, open the block design and add the IP as shown in Figure 2-3. Connect the AXI control input to an open AXI connection on the PS AXI peripheral.

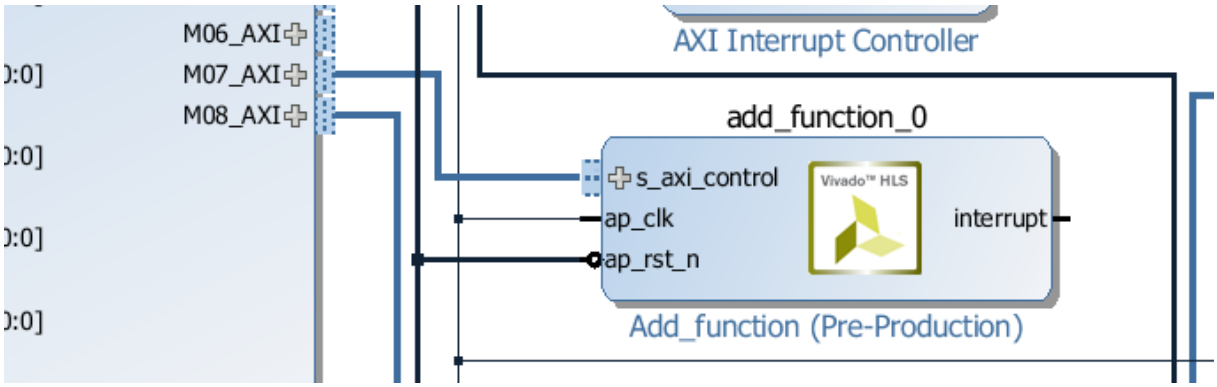


Figure 2-3: Position add IP

Assign an address to our IP in the **Address Editor**. Figure 2-4 shows how this is done.

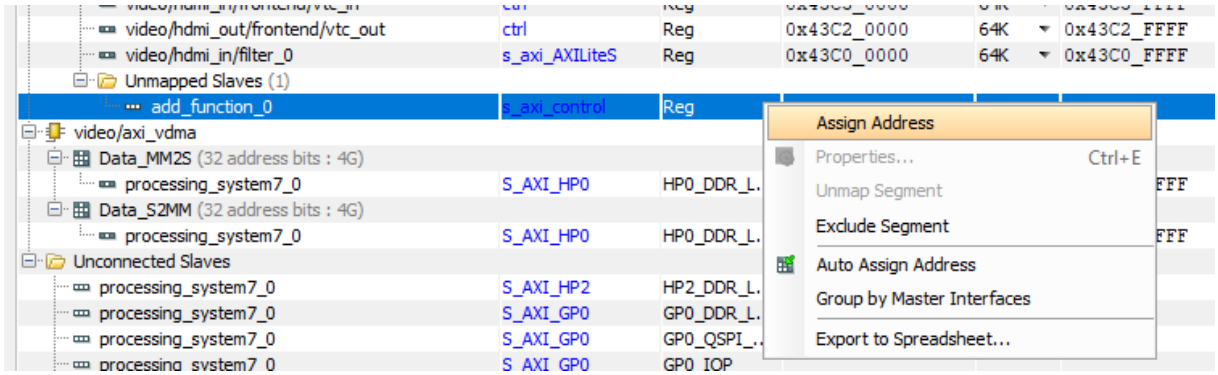


Figure 2-4: Assign IP address

In my case the **offset address** is 0x43C8_0000. Now let's check what's inside this register. In the HLS-project, open **add_function_control_s_axi.vhd** (solution1 -> syn -> vhdl). Scroll down till you see the Address Information. **Fout! Verwijzingsbron niet gevonden.** shows the signals that our important for us.

Table 2-1 Address information

Address	Name	Function
0x00	Control signals	Controls the ip, bit 0 makes the IP start, bit 1 will be high when it's done, …
0x10	Data signal of a	Stores integer a
0x18	Data signal of b	Stores integer b
0x20	Data signal of c	Stores integer c

Now it's possible to generate our BIT- and Tcl file. Generate the bitstream and run the following code in the Tcl console:

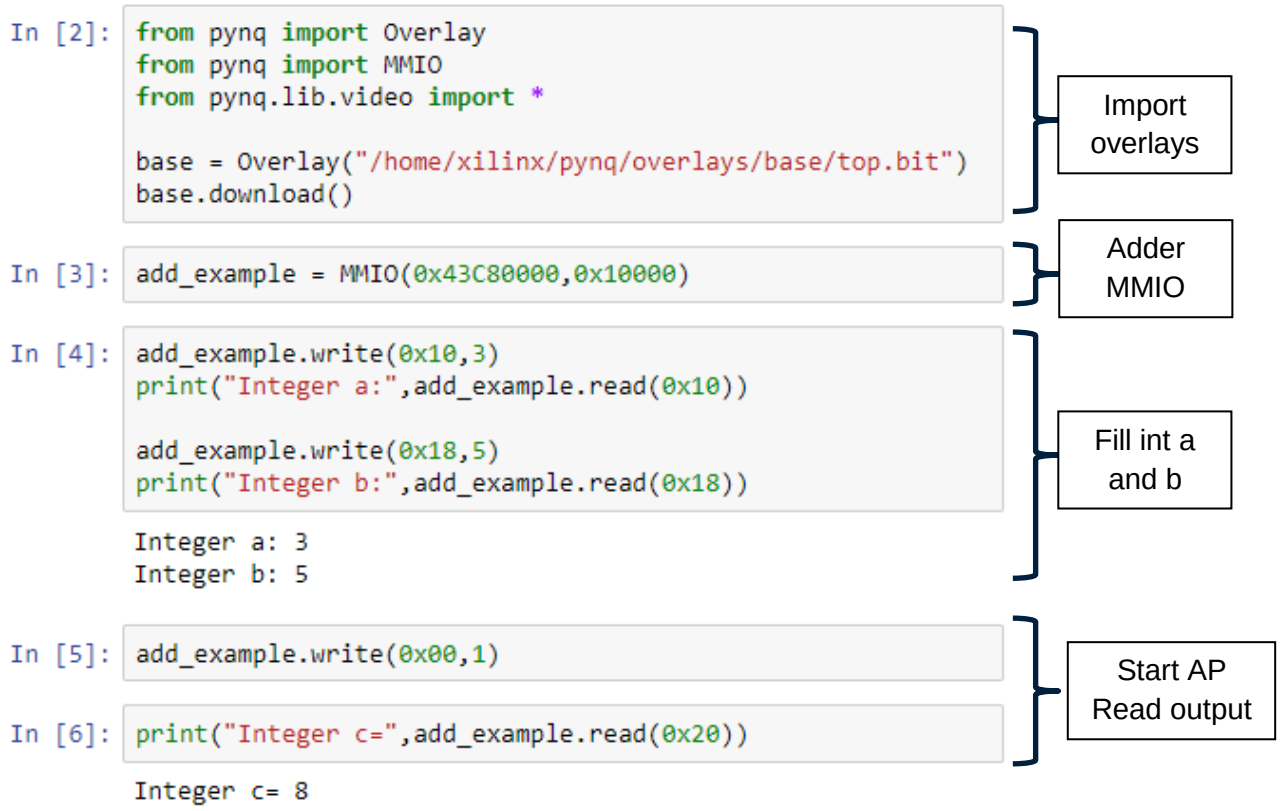
```
write_bd_tcl top.tcl
```

Note: generating the BIT-file can take some time.

Once this is done, copy the BIT-and Tcl file to the following location on a PYNQ board:

\\192.168.2.99\xilinx\pynq\overlays\base

And run the following code in a notebook:



After creating and using the overlay successful, it's possible to use this to create more complex systems. The next chapter will focus on creating the Sobel edge detection filter in a video stream. [4]

3 VIDEO PROCESSING

3.1 Video signal

Before the video can be processed, a closer look is taken on how the video signal is transmitted in the base overlay. Here the video processing can be split into different parts:

- HDMI in
  - Frontend
  - Color_convert
  - Pixel_pack
    - Dvi2RGB decoder
    - Video in to Axi-stream
- VDMA
- HDMI out
  - Pixel_unpack
  - Color_convert
  - Frontend

3.1.1 Frontend

The HDMI signal is transmitted in a transition minimized differential signal (TMDS). The DVI to RGB video decoder decodes this signal and transforms it to a RGB signal. This IP outputs a 24-bit RGB signal with V -and H sync signals. The video in to axi4-stream converts this signal to the **Xilinx video protocol** [5].

Function	Width	Direction	AXI4-Stream Signal Name	Video Specific Name
Video Data	Any number of bytes	Out	m_axis_video_tdata	DATA
Valid	1	Out	m_axis_video_tvalid	VALID
Ready	1	In	m_axis_video_tready	READY
Start Of Frame	1	Out	m_axis_video_tuser	SOF
End Of Line	1	Out	m_axis_video_tlast	EOL

Figure 3-1: Xilinx video protocol signals

The following signals are important for us:

- Video data

  Contains the video data which is 24 bit (8 bit for each color).

- Start of Frame

  Start of frame indicates that the first pixel of a new frame is transmitted.

- End of Line

  End of Line indicates that the last pixel of a line is transmitted.

More information about this protocol can be found on:

https://www.xilinx.com/support/documentation/ip_documentation/axi_videoip/v1_0/ug934_axi_videoIP.pdf

3.1.2 VDMA

The video direct memory access is designed to allow efficient high-bandwidth access between the Axi4-stream video interface and the AXI4 interface. This IP reads and writes frames to DDR memory. [6]

3.1.3 HDMI out

HDMI out is the same as HDMI in but now it transforms the Xilinx video protocol back to HDMI signals.

3.1.4 Video pipeline

For more information about the video pipeline and how to use it, study hdmi_video_pipeline example notebook.

3.2 Processing the signal

The video processing will take place in the HDMI-in package show in Figure 3-2

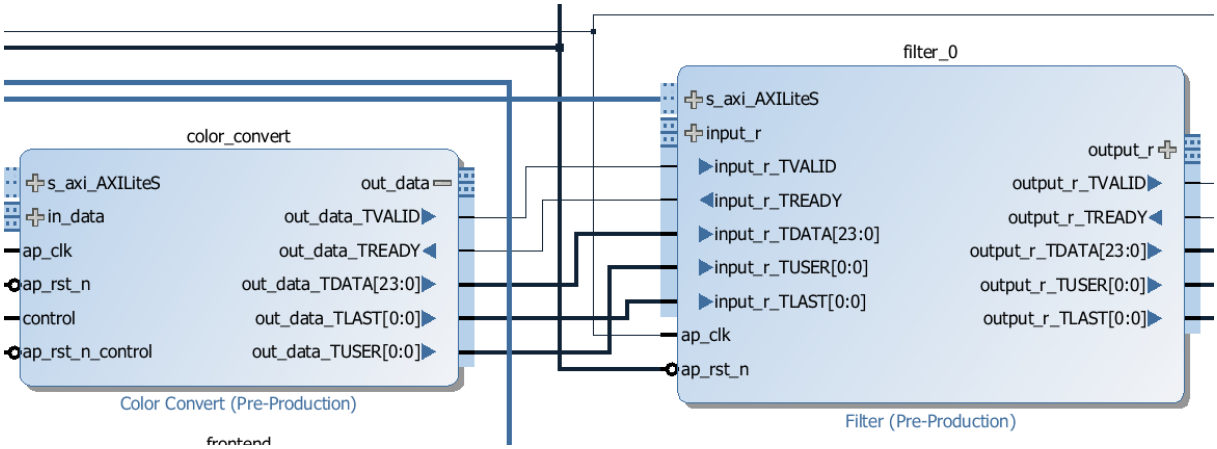


Figure 3-2: Connection video filter

3.2.1 Our first video processing: Screen splitter

Let's start with a simple video processing system. In this project an IP is created that:

- Splits the screen in 2 parts
  - First part: Full original image passes
  - Second part: Only the red component passes
- The horizontal split position can be defined in real time.

All the code can be found on github:

https://github.com/Pieter-Berteloot/PYNQ_Projects/tree/master/Video%20Processing/Split

Implementation:

Let's start making our IP in High Level Synthesis. First, define some types.

```
#include <ap_fixed.h>
#include <ap_int.h>

typedef ap_uint<8> pixel_type;
typedef ap_int<8> pixel_type_s;
typedef ap_ufixed<8,0, AP_RND, AP_SAT> comp_type;

int col = 0;
```

pixel_type: 8 bit unsigned integer

pixel_type_s: 8 bit signed integer

comp_type: 8-bit integer value

0 decimal places

Rounding to plus infinity

Saturation (no wrap-around)

Now define which input and output signals are used. Note that the signals are the same as defined in 3.1.1.

```
struct video_stream {
    struct {
        pixel_type p1;
        pixel_type p2;
        pixel_type p3;
    } data;
    ap_uint<1> user;
    ap_uint<1> last;
};
```

After this, create the TOP function. This project uses a video signal input and an integer. The output is also a video stream.

```
void split_ip(video_stream* in_data, video_stream* out_data, int a) {
```

The program doesn't know what kind off interfaces these input and output signals must use. Pragmas are used to define this. An Axi interface is used for the video stream and an Axi-Lite interface for the split position (integer).

```
#pragma HLS INTERFACE axis port=in_data
#pragma HLS INTERFACE axis port=out_data
#pragma HLS INTERFACE s_axilite port=a

#pragma HLS INTERFACE ap_ctrl_none port=return
```

The pixels are sequentially streamed. Every time a new pixel is received, the EOL and SOF signal are put directly to the output. The data line is stored in temp variables.

```
comp_type in1, in2, in3, out1, out2, out3;
out_data->user = in_data->user;
out_data->last = in_data->last;

in1.range() = in_data->data.p1;
in2.range() = in_data->data.p2;
in3.range() = in_data->data.p3;
```

When the data is read, it is checked in which column the pixel is located. The column counter is reset when the end of line signal is high.

```
if(col <= a){
    out1 = in1;
    out2 = in2;
    out3 = in3;
} else {
    out1 = in1;
    out2 = 0;
    out3 = 0;
}

if(in_data->last)
    col = 0;

col++;
```

After this, the out variable can be assigned to the output stream.

```
out_data->data.p1 = out1.range();
out_data->data.p2 = out2.range();
out_data->data.p3 = out3.range();
```

Synthesize and export and add the IP in the Vivado block design as following:

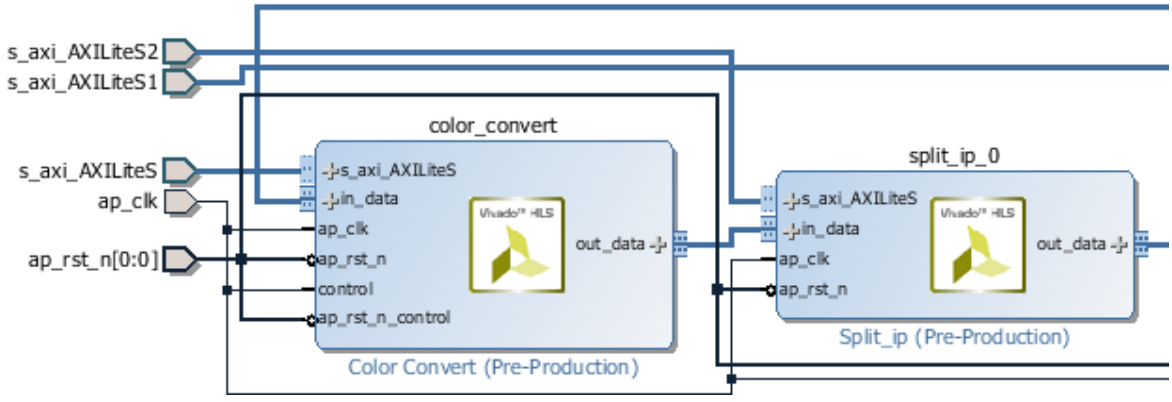
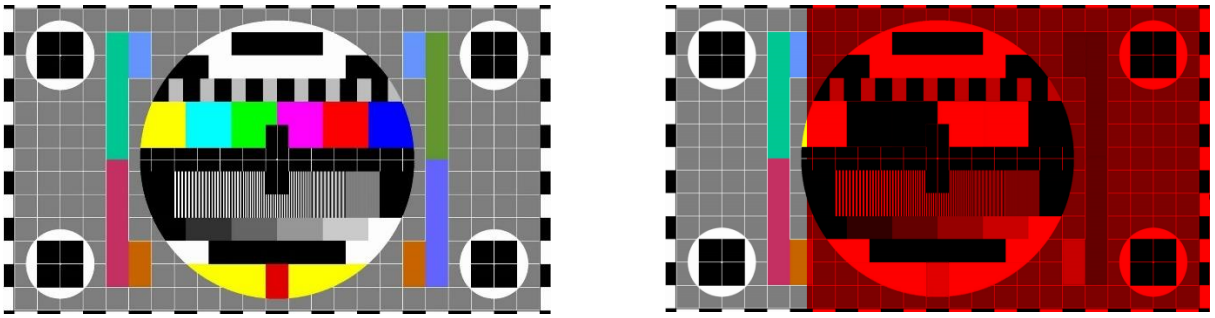


Figure 3-3: placement IP

Note: Don't forget to connect the Axi-Lite interface to the axi_interconnect IP and map the new IP in the address editor.

Generate the Tcl and BIT files and import the overlay in PYNQ, using the same method as in the adder IP. Write a value to 0x10 to define where the split should be. The gives the following result:



Input **Output**

Figure 3-4: Result Screensplitter

3.2.2 C simulation and test bench [7]

Building the overlays takes a lot of time. Let's change the code and make a test bench in HLS so it is possible to simulate instead of creating the overlay. The hls_video and hls_opencv library is used to make this simulation possible. The following changes are made:

- **Input and output type**

For input and output the AXI_STREAM is used. This makes use of HLS::stream. An hls::stream object can be used to store data samples in the same manner as an array. The data in an hls::stream can only be accessed sequentially. In the C code, the hls::stream behaves like a FIFO of infinite depth.

Multiple reads of the same data from an hls::stream are impossible. Once the data has been read from an hls::stream it no longer exists in the stream.

- **Use of hls::mat**

hls::mat represent an image in HLS Video Library [8]. It can be seen as a frame for the programmers. In hardware it is implemented the same way as a stream (with FIFO).

- **Make a seperate split function that can be called in the TOP function**

Implement the functionality of 3.2.1 in a function.

- **The use of AXIvideo2Mat and Mat2AXIvideo**

These functions will convert the video input to a Mat object and convert the Mat object to the output stream. The system handles all the EOL and SOF signals.

- **A test bench and H file is created**

This test bench loads an image, converts it to an Axi stream, sends it through our IP and converts the stream back to an image.

As always: all code can be found on github:

https://github.com/Pieter-Berteloot/PYNQ_Projects/tree/master/Video%20Processing/C%20simulation

Let's first look at the split function:

```
void split(
            RGB_IMAGE& img_in,
            RGB_IMAGE& img_out,
            int index) {

      RGB_PIX pin;
      RGB_PIX pout;

L_row: for(int row = 0; row < 1080; row++) {
#pragma HLS LOOP_TRIPCOUNT min=1 max=1080

      L_col: for(int col = 0; col < 1920; col++) {
#pragma HLS LOOP_TRIPCOUNT min=1 max=1920
#pragma HLS loop_flatten off
#pragma HLS PIPELINE II = 1

           img_in >> pin;

               if(col <= index){
                     pout.val[0] = pin.val[0];
                     pout.val[1] = pin.val[1];
                     pout.val[2] = pin.val[2];
               }
               else{
                     pout.val[0] = pin.val[0];
                     pout.val[1] = 0;
                     pout.val[2] = 0;
               }

           img_out << pout;
         }
    }
}
```

The functionality of this function is the same as in 3.2.1. The only difference is that it uses RGB_IMAGE's as input and output. These are hls::mat objects and can be seen as frames. In the function, there are 2 loops which iterates over all the pixels in the frame.

There are also 2 important pragmas: Loop_flatten and Pipeline. Loop flatting allows nested loops to be collapsed into a single loop with improved latency. Pipeline reduces the initiation interval (number of cock cycles between the start times of consecutive loops) for a function or loop by allowing the concurrent execution of operations.

More information about pragmas can be found here:

Pipeline [9]: https://www.xilinx.com/html_docs/xilinx2018_1/sdsoc_doc/oyc1517254361139.html

Loop flatten [10]: https://www.xilinx.com/html_docs/xilinx2018_1/sdsoc_doc/hid1517254361170.html

Once the split function is made, it can easily be called it in our top function:

```
// Convert AXI4 Stream data to hls::mat format
hls::AXIvideo2Mat(in_data, img_0);

//call the split function
split(img_0, img_1, a);

//Convert the mat to Axi video stream
hls::Mat2AXIvideo(img_1, out_data);
```

The header file is made so the project can be included into a test bench. The H file is self-explanatory and can be found on github. It is important to define the input and output image here.

```
#define INPUT_IMAGE           "test_1080p.bmp"
#define OUTPUT_IMAGE    "test_output_1080p.bmp"
```

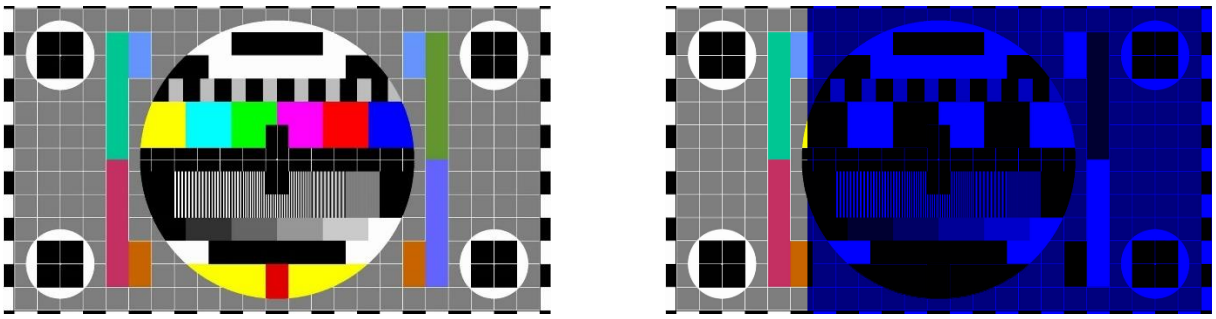
Also place the input image in the HLS project directory. The test bench code is:

```
#include "example_split.h"
#include "hls_opencv.h"


int main(int argc, char** argv){

        IplImage* src = cvLoadImage(INPUT_IMAGE);
        IplImage* dst = cvCreateImage(      cvGetSize(src),
                                            src->depth,
                                            src->nChannels);


        AXI_STREAM src_axi, dst_axi;
        IplImage2AXIvideo(src, src_axi);

        split_ip(src_axi, dst_axi, 500);

        AXIvideo2IplImage(dst_axi, dst);
        cvSaveImage(OUTPUT_IMAGE, dst);
}
```

The opencv functions are used to load, convert and save images. When this is all done, run the C simulation. Hense an image should appear in the following directory:

<hls project>\solution1\csim\build

The result:



Input **Output**

Figure 3-5: Result C simulation

Notice that the Output is now blue instead of red. This is because the color mode is by default BGR but RGB is used in the hardware examples.

Note: for C simulations, streams have an **infinite** depth while in hardware they have a depth of **1** by default. This can result in a difference in results when latency is important.

3.2.3 Screen splitter 2

The next step is performing operations on the pixel data. The objective is to transform the original screen splitter to an RGB and GRAY screen. To convert RGB to GRAY the following formula is used:

$$gray = 0.2989 * \mathrm{R} + 0.587 * \mathrm{G} + 0.114\ \mathrm{B}$$

All 3 colors are set to this gray value so it can be displayed on a monitor. First a type to represent the coefficients is needed. This is done with ap_fixed:

```
typedef ap_fixed<10,2, AP_RND, AP_SAT> coeff_type;
```

This defines a 10-bit variable with 2 bits representing the numbers above the decimal point and 8 bits representing the value below the decimal point. The variable is signed, rounding to plus infinity and uses saturation for overflows.

This is used for:

```
coeff_type const1 = 0.114;
coeff_type const2 = 0.587;
coeff_type const3 = 0.2989;
```

The result of the calculation also needs to be stored.

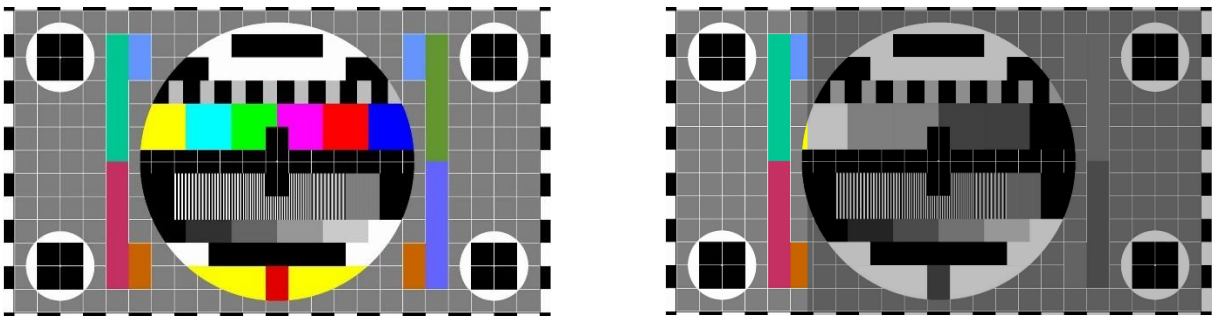
```
char gray;
```

Now the gray value can be calculated. Change the code from the original split to the following:

```
gray = const1 * pin.val[0] + const2 * pin.val[1] + const3 * pin.val[2];
pout.val[0] = gray;
pout.val[1] = gray;
pout.val[2] = gray;
```

Because the interfaces have changed, the inputs and outputs of the IP have to change too. Make sure that every input and output is properly connected.

The result:



Input **Output**

Figure 3-6: Result Screensplitter 2

4 VIDEO FILTERS

It's possible now to read an incoming signal, process it and write it back to the HDMI output. Now it's time to implement filters on the video signal. Let's start with a basic Sobel filter in the x or y direction. **hls::sobel** makes use of the **hls::filter2d** function.

4.1 Edge detection

Edge detection is used to identify and locate discontinuities in an image. These can be detected by checking the change in pixel intensity (high pass filter). The most common way to detect these is by convoluting the image with a kernel. When there is no drastic change this will return a low or zero value. The type of edge detected is in function of the used kernel. In this project the Sobel kernels are used:

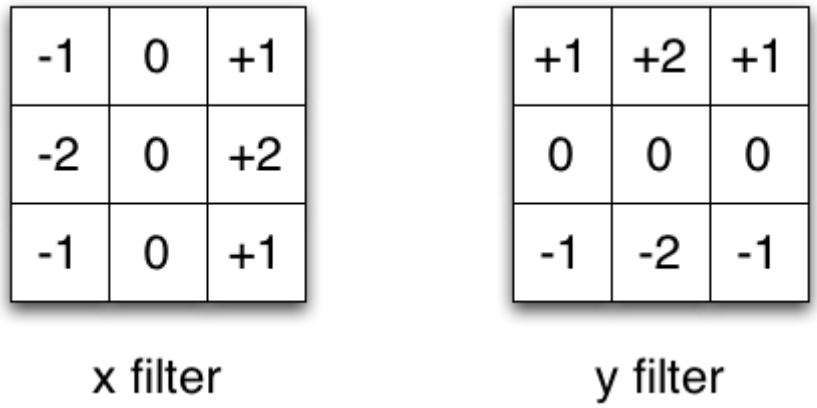


Figure 4-1: Sobel kernel X and Y

These kernels are used to detect edges vertically and horizontally. They can also be combined so it can detect vertical and horizontal edges. The magnitude is calculated with this formula:

$$XY = \sqrt{X^2 + Y^2}$$

Because this is difficult to implement in hardware, a simplified formula is used:

$$XY = |X| + |Y|$$

4.2 Sobel X or Y

First, the split function in 3.2.3 is modified to a RGB2Gray function.

```
void RGB2Gray(
            RGB_IMAGE& img_in,
            RGB_IMAGE& img_out
        ) {

      RGB_PIX pin;
      RGB_PIX pout;
      char gray;

L_row: for(int row = 0; row < 1080; row++) {
#pragma HLS LOOP_TRIPCOUNT min=1 max=1080

      L_col: for(int col = 0; col < 1920; col++) {
#pragma HLS LOOP_TRIPCOUNT min=1 max=1920
#pragma HLS loop_flatten off
#pragma HLS PIPELINE II = 1

           img_in >> pin;

               gray =   const1 * pin.val[0] +
                        const2 * pin.val[1] +
                        const3 * pin.val[2];
               pout.val[0] = gray;
               pout.val[1] = gray;
               pout.val[2] = gray;

           img_out << pout;
        }
    }
}
```

This function has an RGB image as input and converts it to a gray image. Another function is needed to calculate the Sobel filter in X or Y direction:

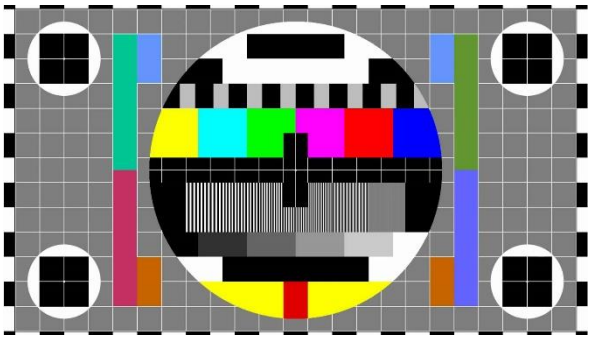
```
void sobel(
            RGB_IMAGE& img_in,
            RGB_IMAGE& img_out,
            char direction
        ) {

      if(direction==1)
            hls::Sobel<1,0,3>(img_in, img_out); //Sobel filter in X direction
      else if(direction == 0)
            hls::Sobel<0,1,3>(img_in, img_out); //Sobel filter in Y direction
      else
            hls::Sobel<0,1,3>(img_in, img_out);
}
```

The hls::sobel function has the following template:

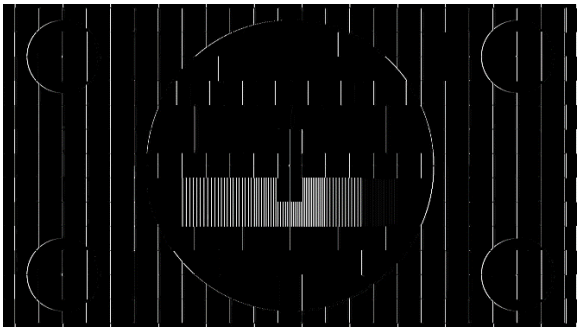
```
template<int XORDER, int YORDER, int SIZE, int ROWS, int COLS, int SRC_T,
int DROWS, int DCOLS, int DST_T>
```

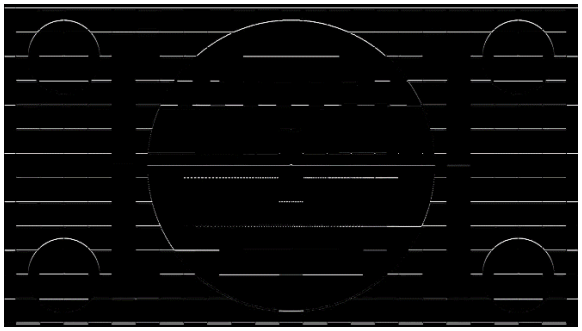
When XORDER=1 and YORDER=0 it computes the horizontal derivative and the other way around for the vertical derivative. SIZE is the kernel size.

After implementing this, the result should be:



Input





Output X **Output Y**

Figure 4-2: Result Sobel X or Y

Note: Our RGB image type is HLS_8UC3:

```
typedef hls::Mat<1080,1920, HLS_8UC3> RGB_IMAGE;
```

This is an 8 bit unsigned char with 3 channels (R, G, B). This means that **negative values cannot be represented** in our image which will lead to information is loss. To demonstrate this, take the following picture as input:

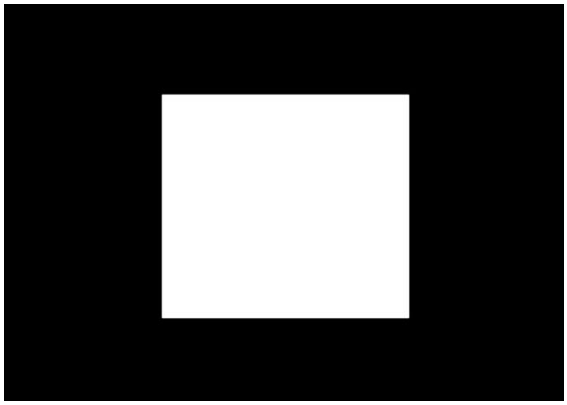


Figure 4-3: Input information loss

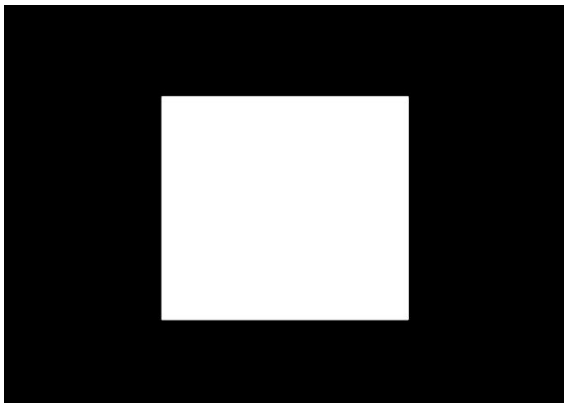
this input from left to right, there is a transition from perfect black <0,0,0> to perfect white <255,255,255> and a transition from perfect white to perfect black.

When this transition happens, the following calculation is made:

$$pixel\ left\ transition = \begin{bmatrix} 0 & 255 & 255 \\ 0 & 255 & 255 \\ 0 & 255 & 255 \end{bmatrix} * \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix} = 1 * 255 + 2 * 255 + 1 * 255 = 1020$$

$$pixel\ right\ transition = \begin{bmatrix} 255 & 255 & 0 \\ 255 & 255 & 0 \\ 255 & 255 & 0 \end{bmatrix} * \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix} = -1 * 255 - 2 * 255 - 1 * 255 = -1020$$

A value of 1020 will be transformed to a 255 value because an unsigned char ranged from 0 to 255. The -1020 value cannot be represented with an unsigned char thus no line will be drawn.



Input

Output

Figure 4-4: Result of information loss

4.3 Sobel X + Y

The following block diagram is used to calculate the sobel function in x and y direction:

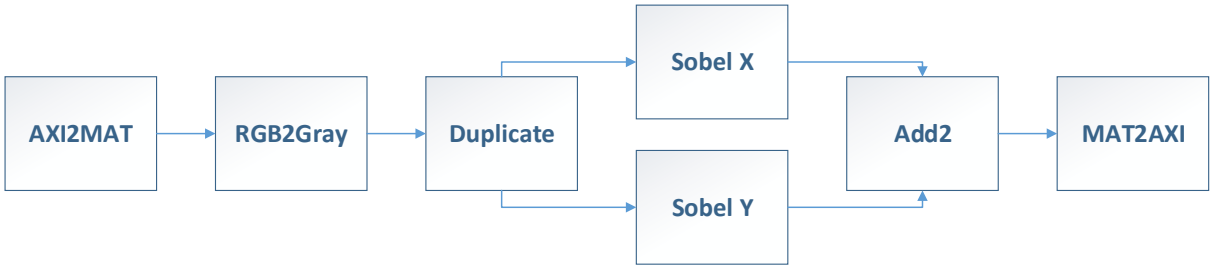


Figure 4-5: block diagram sobel X+Y

To display the horizontal and vertical edges, the sum must be taken from Sobel X and Sobel Y. The following formula should be used:

$$XY = |X| + |Y|$$

In our example, the following formula is used because there are no negative values in the unsigned char data type.

$$XY = X + Y$$

First an add function is made so this formula can be used on an image.

```
void add(RGB_IMAGE& img_in0, RGB_IMAGE& img_in1, RGB_IMAGE& img_out) {

      RGB_PIX pin0, pin1;
      RGB_PIX pout;

L_row: for(int row = 0; row < 1080; row++) {
#pragma HLS LOOP_TRIPCOUNT min=720 max=1080

L_col: for(int col = 0; col < 1920; col++) {
#pragma HLS LOOP_TRIPCOUNT min=1280 max=1920
#pragma HLS loop_flatten off
#pragma HLS PIPELINE II = 1

           img_in0 >> pin0;
           img_in1 >> pin1;

               pout = (pin0 + pin1);

           img_out << pout;
        }
    }
}
```

This function adds 2 images together. It can be used to add the Sobel X and Y results. These are obtained using the Sobel function in 4.2. As seen in the block diagram of/in Figure 4-5, the input image must be duplicated so the Sobel functions can be used on the same image.

```
void copy2(RGB_IMAGE& img_in, RGB_IMAGE& img_out0, RGB_IMAGE& img_out1) {

      RGB_PIX pin;
      RGB_PIX pout;

L_row: for(int row = 0; row < 1080; row++) {
#pragma HLS LOOP_TRIPCOUNT min=720 max=1080

L_col: for(int col = 0; col < 1920; col++) {
#pragma HLS LOOP_TRIPCOUNT min=1280 max=1920
#pragma HLS loop_flatten off
#pragma HLS PIPELINE II = 1

           img_in >> pin;

               pout = pin;

           img_out0 << pout;
           img_out1 << pout;
        }
    }
}
```

This function is similar to the add function. The only difference is that it uses 1 input and 2 outputs. Now these functions can be used in our TOP function.

```
// Convert AXI4 Stream data to hls::mat format
hls::AXIvideo2Mat(in_data, img_0);

//Convert to gray image
RGB2Gray(img_0, img_1);

//copy the input image
copy2(img_1, img_2, img_3);

//sobel functions
sobel(img_2, img_4, 0);
sobel(img_3, img_5, 1);

//add sobel x and y
add2(img_4, img_5, img_6);

//Convert the mat to Axi video stream
hls::Mat2AXIvideo(img_6, out_data);
```

Note: Don’t forget to define the new RGB images.

It’s always good to have a look in the Analysis tab. The schedule and dataflow can be seen here. This is useful to check your system before building it.

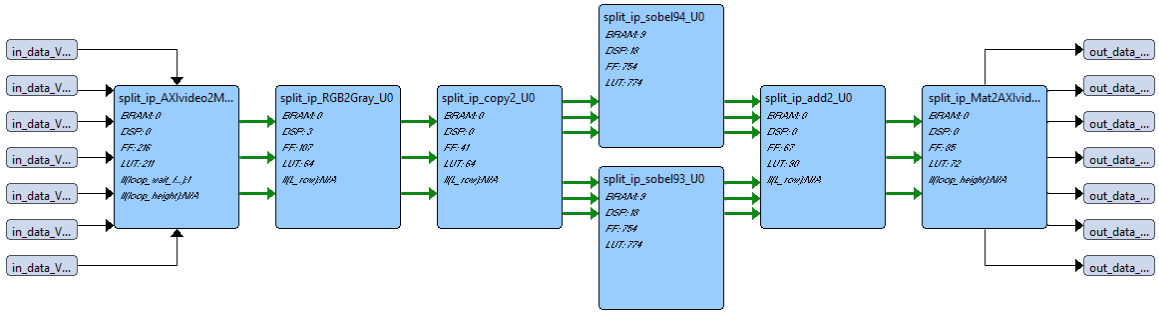
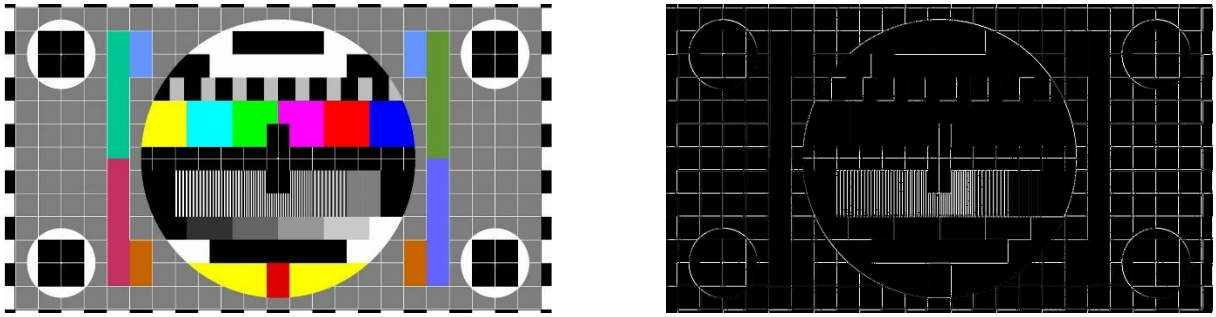


Figure 4-6: HLS dataflow

The result of this should be:



Input **Output**

Figure 4-7: Result Sobel X + Y

4.4 A system using these filters

Now it’s time to make a system with these filters. Let’s try to make a system that is the sum of

- The original frame
- Sobel X
- Sobel Y

The brightness of all the separate frames can be lowered and also the color mode can be chosen.

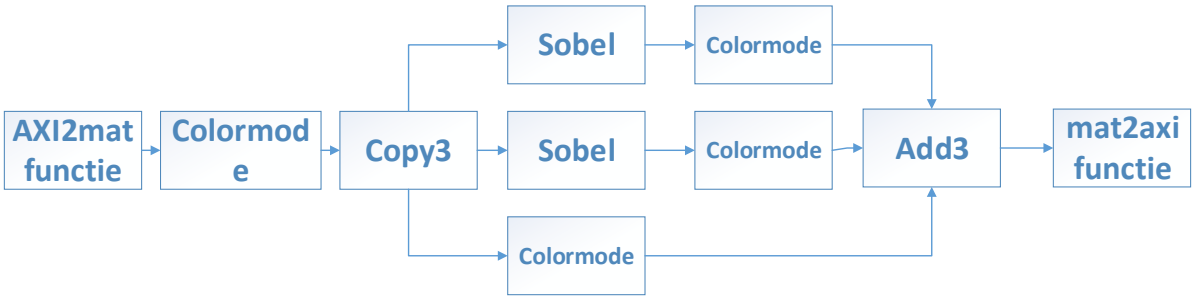


Figure 4-8: Block diagram system

Let's have a look at the colormode function. This function has 4 modes:

- RGB passthrough
- RGB2Gray
- Screen split in half: left RGB, right Gray
- Full black

```
void colorMode(RGB_IMAGE& img_in, RGB_IMAGE& img_out, char mode){
      RGB_PIX pin;
      RGB_PIX pout;
      char gray;
      L_row: for(int row = 0; row < 1080; row++) {
#pragma HLS LOOP_TRIPCOUNT min=1 max=1080
            L_col: for(int col = 0; col < 1920; col++) {
#pragma HLS LOOP_TRIPCOUNT min=1 max=1920
#pragma HLS pipeline rewind
                  img_in >> pin;
                        if(mode == 0){
                     pout.val[0] = pin.val[0];
                     pout.val[1] = pin.val[1];
                     pout.val[2] = pin.val[2];

                  } else if(mode == 1) {
                           gray =   const1 * pin.val[0] +
                                    const2 * pin.val[1] +
                                    const3 * pin.val[2];
                           pout.val[0] = gray;
                           pout.val[1] = gray;
                           pout.val[2] = gray;

                  } else if(mode == 2){
                        if(col <= 960){
                           pout.val[0] = pin.val[0];
                           pout.val[1] = pin.val[1];
                           pout.val[2] = pin.val[2];

                        } else {
                           gray =   const1 * pin.val[0] +
                                    const2 * pin.val[1] +
                                    const3 * pin.val[2];
                           pout.val[0] = gray;
                           pout.val[1] = gray;
                           pout.val[2] = gray;
                        }

                  }  else if(mode == 3){
                     pout.val[0] = 0;
                     pout.val[1] = 0;
                     pout.val[2] = 0;

                  } else {
                     pout.val[0] = pin.val[0];
                     pout.val[1] = pin.val[1];
                     pout.val[2] = pin.val[2];

                  }
                  img_out << pout;
            }
      }
}
```

RGB

GRAY

SPLIT

Black

The next step is to modify the Copy2 to a Copy 3 function. When this is done, the Sobel and Colormode functions can be reused. Notice that the block diagram is not symmetric. The output of the original image will arrive before the output of the filters do. This will cause the pixels to be out of sync and won't be able to be added in the add3 function.

To prevent this from happening, a buffer is placed between Colormode and the add3 function. First the minimum depth of this buffer needs to be known. The Sobel function, with a kernel size of 3, needs at least 3 lines of data to begin calculating. So, the latency in a Sobel function is 3 * 1920 = 5760.

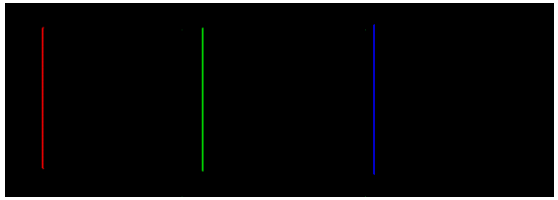
5760 pixels will be stored in this buffer so the 3 lines will be in sync again. hls::mat is basically the same as a stream (mat is implemented as a hls::stream). So, the depth of the image can be set where the buffer is needed.

```
#pragma HLS stream depth=19200  variable=img_1.data_stream
```

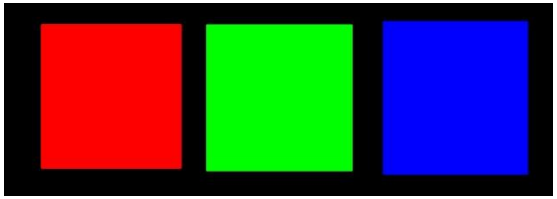
Now the only thing that needs to be done is making the add3 function. This function adds 3 images. The intensity of each image can also be changed in this function.

```
void add3(RGB_IMAGE& img_in0,int a, RGB_IMAGE& img_in1, int b, RGB_IMAGE&
img_in2, int c, RGB_IMAGE& img_out) {

      RGB_PIX pin0, pin1, pin2;
      RGB_PIX pout;

L_row: for(int row = 0; row < 1080; row++) {
#pragma HLS LOOP_TRIPCOUNT min=720 max=1080

L_col: for(int col = 0; col < 1920; col++) {
#pragma HLS LOOP_TRIPCOUNT min=1280 max=1920
#pragma HLS loop_flatten off
#pragma HLS PIPELINE II = 1

            img_in0 >> pin0;
            img_in1 >> pin1;
            img_in2 >> pin2;
            pout = (pin0/a + pin1/b + pin2/c);

            img_out << pout;
        }
    }
}
```

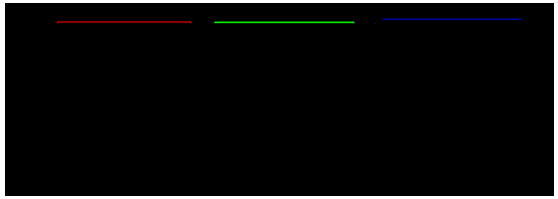
The result:



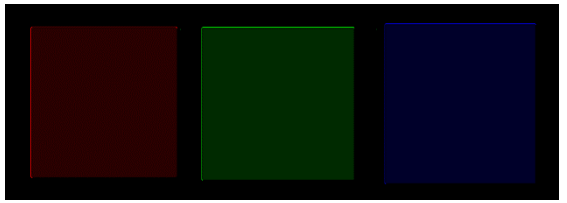
Sobel X



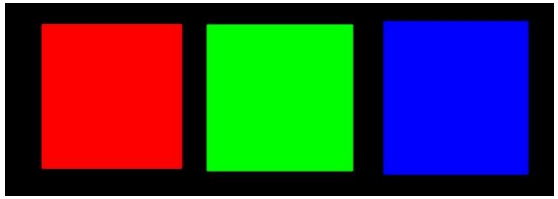
Input



Sobel Y



Output



Input

Figure 4-9: Result system using filters

4.5 Sobel negative values

Let's take a closer look on how to solve the problem discussed in 4.2. The negative values of the Sobel filter are needed for video processing. These negative values contain information that is not used in our previous projects.

To show these negative values, it's possible to multiply the X and Y kernel by -1. This will only output the "negative" values. Taking the sum of these two will double the needed resources.

A better solution is to change the datatype so negative values can be represented.

The objective in this project is to construct the following system:

Figure 4-10: image types

These functions must be made:

- convertToSigned
- convertToUnsigned

First, a new type of image is needed to store the 16-bit data.

```
typedef hls::Mat<1080,1920, HLS_16SC3> RGB16_IMAGE;
```

Now there must be a data type to fill this image up:

```
typedef hls::Scalar<3, short > RGB16_PIX;
```

The convert to signed function takes a RGB_IMAGE and converts it to a RGB16_IMAGE.

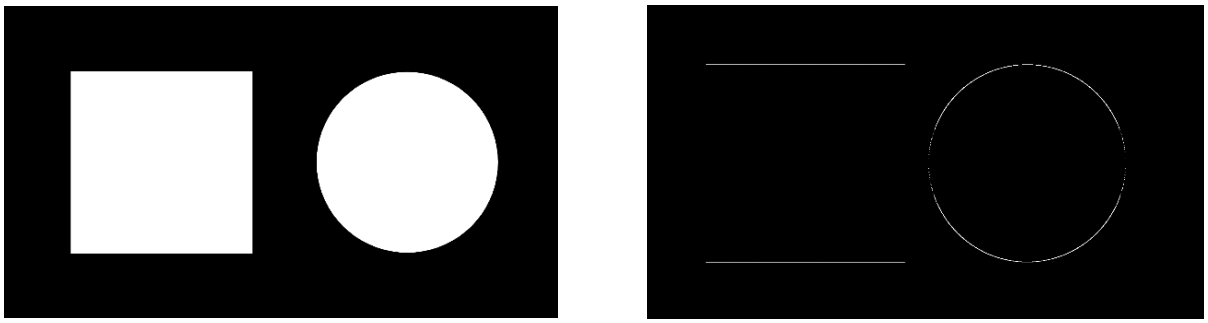
```
void convertToSigned(RGB_IMAGE& img_in0, RGB16_IMAGE& img_out){

      RGB_PIX   pin; //the input
      RGB16_PIX pout; // for the output

L_row: for(int row = 0; row < 1080; row++) {
#pragma HLS LOOP_TRIPCOUNT min=720 max=1080

      L_col: for(int col = 0; col < 1920; col++) {
#pragma HLS LOOP_TRIPCOUNT min=1280 max=1920
#pragma HLS loop_flatten off
#pragma HLS PIPELINE II = 1

            img_in0 >> pin;
            pout.val[0]=pin.val[0];
            pout.val[1]=pin.val[1];
            pout.val[2]=pin.val[2];
            img_out << pout;
        }
    }
}
```

The convert to unsigned function takes a RGB16_IMAGE and converts it back to a RGB_IMAGE. It's similar to the convert to unsigned function but now the absolute value must be taken to convert it to unsigned.

```
img_in0 >> pin;
if (pin.val[0] >= 255)
      pout.val[0]=255;
else if(pin.val[0]>=0)
      pout.val[0]=pin.val[0];
else
      pout.val[0] = 0;

if (pin.val[1] >= 255)
      pout.val[1]=255;
else if(pin.val[1]>=0)
      pout.val[1]=pin.val[1];
else
      pout.val[1] = 0;

if (pin.val[2] >= 255)
      pout.val[2]=255;
else if(pin.val[2]>=0)
      pout.val[2]=pin.val[2];
else
      pout.val[2] = 0;

img_out << pout;
```

Note: Don't forget to change the image type in the Sobel function to RGB16_IMAGE.

The result:



Input **Output**

Figure 4-11: Result Sobel with negative values

The data is kept as a short which is 16 bits. This has a range from −32,768 to 32,767. Most values will never be used and will take unnecessary resources. Let's change it to 10 signed bits because the HLS_10SC3 data type already exists in hls_video_types.h. This is a 3 channel 10-bit signed data type. This ranges from -512 to 511.

Image type:

```
typedef hls::Mat<1080,1920, HLS_10SC3> RGB10_IMAGE;
```

Pixel type:

```
typedef hls::Scalar<3, ap_int<10> > RGB10_PIX;
```

Summary

Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	-	-	-
Expression	-	-	-	-
FIFO	0	-	75	348
Instance	18	3	1390	1815
Memory	-	-	-	-
Multiplexer	-	-	-	-
Register	-	-	5	-
Total	18	3	1470	2163
Available	280	220	106400	53200
Utilization (%)	6	1	1	4

Figure 4-13: Resources 16 bit

Summary

Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	-	-	-
Expression	-	-	-	-
FIFO	0	-	75	312
Instance	18	3	1102	1473
Memory	-	-	-	-
Multiplexer	-	-	-	-
Register	-	-	5	-
Total	18	3	1182	1785
Available	280	220	106400	53200
Utilization (%)	6	1	1	3

Figure 4-12: Resources 10 bit

Some FF's and LUT's are saved in the 10-bit version. The BRAM's and DSP's stay the same because the 10-bit integer is stored in a 16-bit value.

The resources can be lowered even more by using only 1 channel for the gray images. Using 3 channels for gray images is useless because the values in the channels are all the same. This can easily be done by adding 2 more image and pixel types:

```
typedef hls::Mat<1080,1920, HLS_8UC1> GRAY_IMAGE;
typedef hls::Mat<1080,1920, HLS_10SC1> GRAY10_IMAGE;

typedef hls::Scalar<1, unsigned char> GRAY_PIX;
typedef hls::Scalar<1, ap_int<10> > GRAY10_PIX;
```

To easiest way to change from rgb to gray is using the hls::CvtColor converter. This function converts a RGB image to a 1 channel Gray image.

```
hls::CvtColor<HLS_RGB2GRAY>(img_0, img_gray1);
```

After some simple changes the following synthesis is created:

Summary

Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	-	-	-
Expression	-	-	-	-
FIFO	0	-	70	300
Instance	6	3	895	1065
Memory	-	-	-	-
Multiplexer	-	-	-	-
Register	-	-	6	-
Total	6	3	971	1365
Available	280	220	106400	53200
Utilization (%)	2	1	~0	2

Figure 4-14: Resources 1 channel, 10 bit

The amount of BRAM is divided by 3 because it uses 3 times less data. The FF's and LUT's have also decreased.

4.5.1 Visualizing the positive and negative values

A common way to visualize the positive and negative values is by using an offset. This is done with the following code:

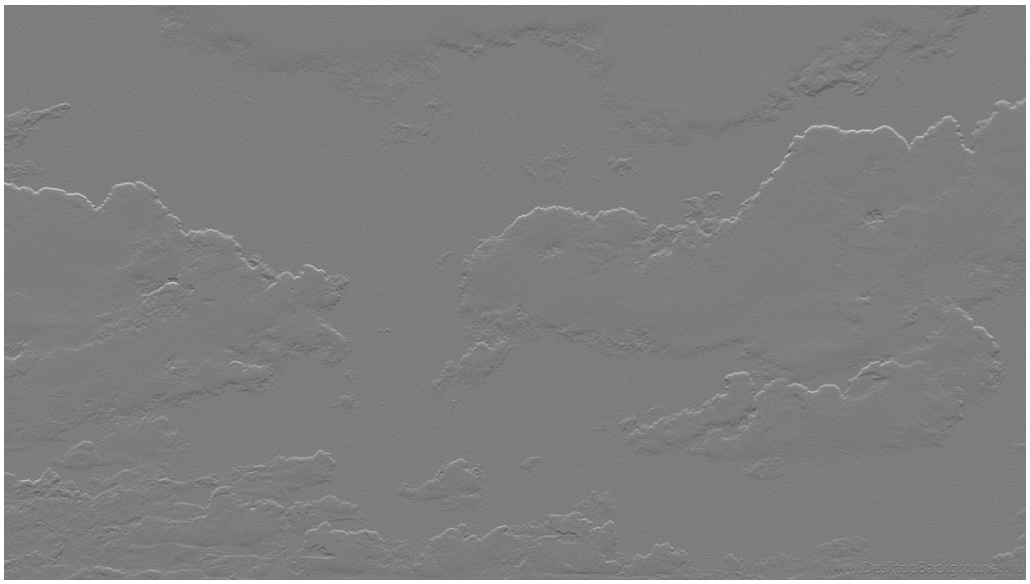
```
pout.val[0]=(pin.val[0] / 2) + 127;
```

The input is divided by 2 and a value of 127 is added. Black (value 0) will become a value of 127. Negative values will be darker and positive values will be brighter (negative values less than 127 and positive greater than 127).

The following result is a visualization of a Sobel Y filter:

Input

Output



5 DESIGN

Previous chapters explained how to create, add functionality and implement IP's it in hardware. This information is used to create two new IP's with the following functionality:

Sobel_IP: The sobel IP is a combination of the IP's in chapter 4.4 and 4.5. The user has the ability to perform following operations on the video signal:

- Passthrough
- Convert to grayscale
- Split the image: left RGB and right grayscale
- Split the image: left RGB and right black
- Split the image: left black and right gray

This signal is then send to 3 function blocks:

- Sobel X
- Sobel Y
- Passthrough

The user can select the color mode again in each block. After this, all signals are added in a single image. All data from the images are 10 bits signed, and thus negative values can be stored in the image matrix.

Some examples for what the IP can create:

1. Split screen: Original video left and Sobel X or Y right
2. Split screen: Original video left and Sobel X and Y right
3. Image sharpening on original video using Sobel filters
4. Image sharpening on grayscale video using Sobel filters
5. Split screen: Original video left and image sharpening right

Sharpening_IP: This IP performs image sharpening using the Laplacian high pass filter. First the video signal is duplicated. One signal is first blurred using a gaussian blur filter. The video signal is then reduced to a one channel 8-bit signal to reduce hardware resources. This signal is then converted to a 10-bit signed signal so negative values can be stored. These negative values are needed to perform the image sharpening. After this, the signal is scaled so the user can set the sharpening intensity. The last step is to add this and the original video signal together and convert this back to a 3 channel 8-bit video signal.

5.1 Hardware

5.1.1 Sobel_IP

The Sobel_IP is based on the code in chapter 4.4. Two more modes are added to the colorMode function: left RGB/right black and left black and right gray.

```
} else if(mode == 4){
      if(col <= 960){
          pout.val[0] = pin.val[0];
          pout.val[1] = pin.val[1];
                pout.val[2] = pin.val[2];

      } else {
          pout.val[0] = 0;
          pout.val[1] = 0;
          pout.val[2] = 0;
      }

} else if(mode == 5){
      if(col <= 960){
          pout.val[0] = 0;
          pout.val[1] = 0;
                pout.val[2] = 0;

      } else {
          gray =   const1 * pin.val[0] +
                   const2 * pin.val[1] +
                   const3 * pin.val[2];
          pout.val[0] = gray;
          pout.val[1] = gray;
          pout.val[2] = gray;
      }
```

These two modes make it possible to split the screen left in RGB and right is the Sobel output.

The top function is self-explanatory:

```
//Axi stream to Matrix
hls::AXIvideo2Mat(input, img_input_8U3);

//convert input image to 10 signed bit data type
convertToSigned(img_input_8U3, img_input_10S3);

//set colorMode of the videoInput
colorMode(img_input_10S3, img_colorMode_10S3, mode);

//copy video input 3 times
copy3(img_colorMode_10S3,     img_colorMode_copy1_10S3,
                              img_colorMode_copy2_10S3,
                              img_colorMode_copy3_10S3);

//calculate sobel X and Y
hls::Sobel<0,1,0,3>(img_colorMode_copy2_10S3, img_SobelX_10S3);
hls::Sobel<0,0,1,3>(img_colorMode_copy3_10S3, img_SobelY_10S3);

//set colorMode for all images
colorMode(  img_colorMode_copy1_10S3,
            img_colorMode2_copy1_10S3,
            modeOriginal);
colorMode(img_SobelX_10S3, img_SobelX_colorMode_10S3,mode1);
colorMode(img_SobelY_10S3, img_SobelY_colorMode_10S3,mode2);

//add all images together
add3( img_SobelY_colorMode_10S3,index2,
      img_SobelX_colorMode_10S3, index3,
      img_colorMode2_copy1_10S3, index,
      img_output_10S3);

//convert back to 8 unsinged bit datatype
convertToUnsigned(img_output_10S3, img_output_8S3);

// Convert the hls::mat format to AXI4 Stream format with SOF and EOL
hls::Mat2AXIvideo(img_output_8S3, output);
```

The functions from 4.5 are added to the project. This will allow us to sharpen the video signal. The Sobel filters detect the edges (positive and negative values) so all non-edges are unchanged and appear black in the Sobel output. Adding the Sobel output to the original image will change the contrast of the edges, which will give a sharpening illusion.

5.1.2 Sharpening_IP

A more common way to sharpen images is by using a Laplacian filter instead of the Sobel filter. The Sobel filter calculates the first derivatives separately for the X and Y axes. The Laplacian filter calculates the second derivatives using a single kernel which makes the Laplacian more hardware efficient.

The following Laplacian kernel is used in our IP:

$$\begin{bmatrix} -1 & -1 & -1 \\ -1 & 8 & -1 \\ -1 & -1 & -1 \end{bmatrix}$$

To reduce noise, the image is first send through a gaussian blur filter which smooths the image.

To implement this in hardware, first a gaussian blur function is made:

```
void gaussianBlur(RGB_IMAGE& img_in, RGB_IMAGE& img_out) {

      const ap_fixed<16,2,AP_RND> coef_gaussian[5][5]= {
        {0.000874,      0.006976,   0.01386,    0.006976,   0.000874},
        {0.006976,      0.0557,     0.110656,   0.0557,     0.006976},
        {0.01386,       0.110656,   0.219833,   0.110656,   0.01386},
        {0.006976,      0.0557,     0.110656,   0.0557,     0.006976},
        {0.000874,      0.006976,   0.01386,    0.006976,   0.000874},
      };


      hls::Window<5, 5, ap_fixed<16,2,AP_RND>  > kernel;
      for (int r=0; r<3; r++) for (int c=0; c<3; c++)
            kernel.val[r][c] = coef_gaussian[r][c];

      hls::Point_<INDEX_T> anchor;
      anchor.x=-1;
      anchor.y=-1;

      hls::Filter2D <hls::BORDER_REFLECT> (     img_in,
                                                img_out,
                                                kernel,
                                                anchor);
}
```

This function performs a Filter2D convolution with a 5x5 kernel. The kernel values were calculated with a sigma value of 0.8. The following site was used to perform this calculation:

http://dev.theomader.com/gaussian-kernel-calculator/

Next step is to create a function that calculates the Laplacian image. This is similar to the previous function, only now a 3x3 kernel is used.

```
void laplacian(GRAY10_IMAGE& img_in, GRAY10_IMAGE& img_out) {

	const COEF_T coef_v[3][3]= {
	  {-1,-1,-1},
	  {-1, 8,-1},
	  {-1,-1,-1}
	};

	hls::Window<3,3,COEF_T> Sv;
	for (int r=0; r<3; r++) for (int c=0; c<3; c++)
	      Sv.val[r][c] = coef_v[r][c];

	hls::Point_<INDEX_T> anchor;
	anchor.x=-1;
	anchor.y=-1;

	hls::Filter2D <hls::BORDER_REFLECT> (img_in, img_out, Sv, anchor);
}
```

The TOP function is self-explanatory:

```
	//axi stream to matrix
	hls::AXIvideo2Mat(in_data, input_image_8U3);

	//duplicate the input image
	hls::Duplicate( input_image_8U3,    input_image_8U3_copy1,
	                                    input_image_8U3_copy2 );

	//reduce noise with smoothing
	gaussianBlur(input_image_8U3_copy1, image_blur);

	//convert to 8 bit unsigned
	hls::CvtColor<HLS_RGB2GRAY>(image_blur, image_blur_8U1);

	//convert to 1 channel 10 bit signed datatype
	convertToSigned(image_blur_8U1, image_blur_10S1);

	//calculate laplacian
	laplacian(image_blur_10S1, image_laplacian_10S1);

	//scale the intensity of the laplacian
	hls::Scale( image_laplacian_10S1,
	            image_laplacian_reduced_10S1,
	            ((double)reduce)/100,0.0);

	//convert data types so they can be added
	hls::CvtColor<HLS_GRAY2RGB>(  image_laplacian_reduced_10S1,
	                              image_laplacian_10S3);
	convertToSigned(input_image_8U3_copy2, input_image_10S3);
	hls::AddWeighted( image_laplacian_10S3, 1,
	                  input_image_10S3, 1 ,
	                  0 , ouput_image_10S3);

	//convert back to 3 channel 8 bit unsigned datatype
	convertToUnsigned(ouput_image_10S3, output_image_8U3);

	hls::Mat2AXIvideo(output_image_8U3, out_data);
```

hls::Scale is used so the user can define the intensity of the Laplacian image in percentage.

5.2 Software

The **Sobel_IP** makes use of a wrapper. This takes care of all MMIO writes and contains following functions:

- setOriginalMode: set the color mode of the video input
- setPassthroughMode: set the color mode of the passthrough output
- setSobelXMode: set the color mode of the Sobel X output
- setSobelYMode: set the color mode of the Sobel Y output
- setPassthroughIntensity: set the intensity of the passthrough output
- setSobelXIntensity: set the intensity of the Sobel X output
- setSobelYIntensity: set the intensity of the Sobel Y output

The above functions all write a user defined value to a register. For example the setOriginalMode writes the value to 0x40 which is the address to set the video input color mode.

```
def setOriginalMode(self,mode):
    self.mmio.write(0x40,mode)
```

The init function is called when the user makes the Sobel_Split object. This function initializes the MMIO connection and sets all the modes and intensity to a normal passthrough.

```
def __init__(self):
    self.mmio = MMIO(0x43C00000,0x10000)

    self.setPassthroughIntensity(1)
    self.setSobelYIntensity(1)
    self.setSobelXIntensity(1)

    self.setOriginalMode(0)
    self.setPassthroughMode(0)
    self.setSobelXMode(3)
    self.setSobelYMode(3)
```

The wrapper also contains a imageSharpening and sobelSplit function, which sets the modes and intensity for image sharpening and a half passthrough half Sobel filtering using the above functions.

```
def sobelSplit(self):
    self.setPassthroughIntensity(1)
    self.setSobelYIntensity(1)
    self.setSobelXIntensity(1)

    self.setOriginalMode(0)
    self.setPassthroughMode(4)
    self.setSobelXMode(5)
    self.setSobelYMode(5)
```

To run the hardware overlay on the PYNQ, the following notebook is used:

In [1]:
```
from pynq import Overlay
from pynq.lib.video import *
from pynq.lib.Sobel_Split import *

base = Overlay("/home/xilinx/pynq/overlays/base/Sobel_IP.bit")
base.download()

Sobel_ip = Sobel_Split()
```

First Overlay is imported, this allows us to download our custom overlay. After this the video and Sobel_split libraries are imported. The video library controls all other IP's in the video pipeline. The Sobel_split library is our own wrapper.

When the imports are done, the Sobel_IP overlay is loaded and a Sobel_Split object is made, which calls the init function of the Sobel_Split library.

In [2]:
```
hdmi_in = base.video.hdmi_in
hdmi_out = base.video.hdmi_out
hdmi_in.configure()
hdmi_out.configure(hdmi_in.mode)

hdmi_in.start()
hdmi_out.start()
```

Out[2]:
```
<contextlib._GeneratorContextManager at 0x300588d0>
```

In [3]:
```
RGB = 0
GRAY = 1
RGB_GRAY = 2
BLACK = 3
RGB_BLACK = 4
BLACK_GRAY = 5
```

Set up the HDMI connections and set some constants which makes the of the Sobel_Split functions easier.

In [4]:
```
hdmi_in.tie(hdmi_out)
```

Tie the input to the output signal. Running this command should turn on the monitor.

After this, the user can control our Sobel_IP using the wrapper. For example:

```
In [5]: Sobel_ip.sobelSplit()
```

```
In [6]: Sobel_ip.imageSharpening()
```

Which produces the split Sobel and image sharpening respectively.

The **Sharpening_IP** doesn't make use of a wrapper but uses a slider to control the MMIO communication using the generator approach from:

http://ipywidgets.readthedocs.io/en/latest/examples/Widget%20Asynchronous.html

This makes the following code possible:

```
In [8]: sharpining_ip = MMIO(0x43C00000,0x10000)

        from ipywidgets import IntSlider, VBox, HTML
        IntensityLaplacian=IntSlider()

        IntensityLaplacian.max = 300
        IntensityLaplacian.step = 50

        @yield_for_change(IntensityLaplacian, 'value')
        def f():
            for i in range(50000):
                x = yield
                sharpining_ip.write(0x10,x)

        f()
```

```
In [9]: IntensityLaplacian
```

Every time the user changes the slider, the corresponding value is send to the register which sets the intensity of the Laplacian image.

5.3 Design Reuse

Both IP's can be implemented in any Xilinx video protocol pipeline using 3 channels with 8 bits data. The design can process all resolutions up to 1080 x 1920 pixels. Both IP's can also leave the video signal unchanged, either by setting the Sobel modes to black or the intensity to 0%.

All ip's in this report can be found on github and can be implemented in your own Vivado design:

https://github.com/Pieter-Berteloot/PYNQ_Projects

5.4 Results

The Sobel_IP produces the following example results (top left: original image, top right: SobelX, bottom left: RGB and SobelXY, bottom right: image sharpening):

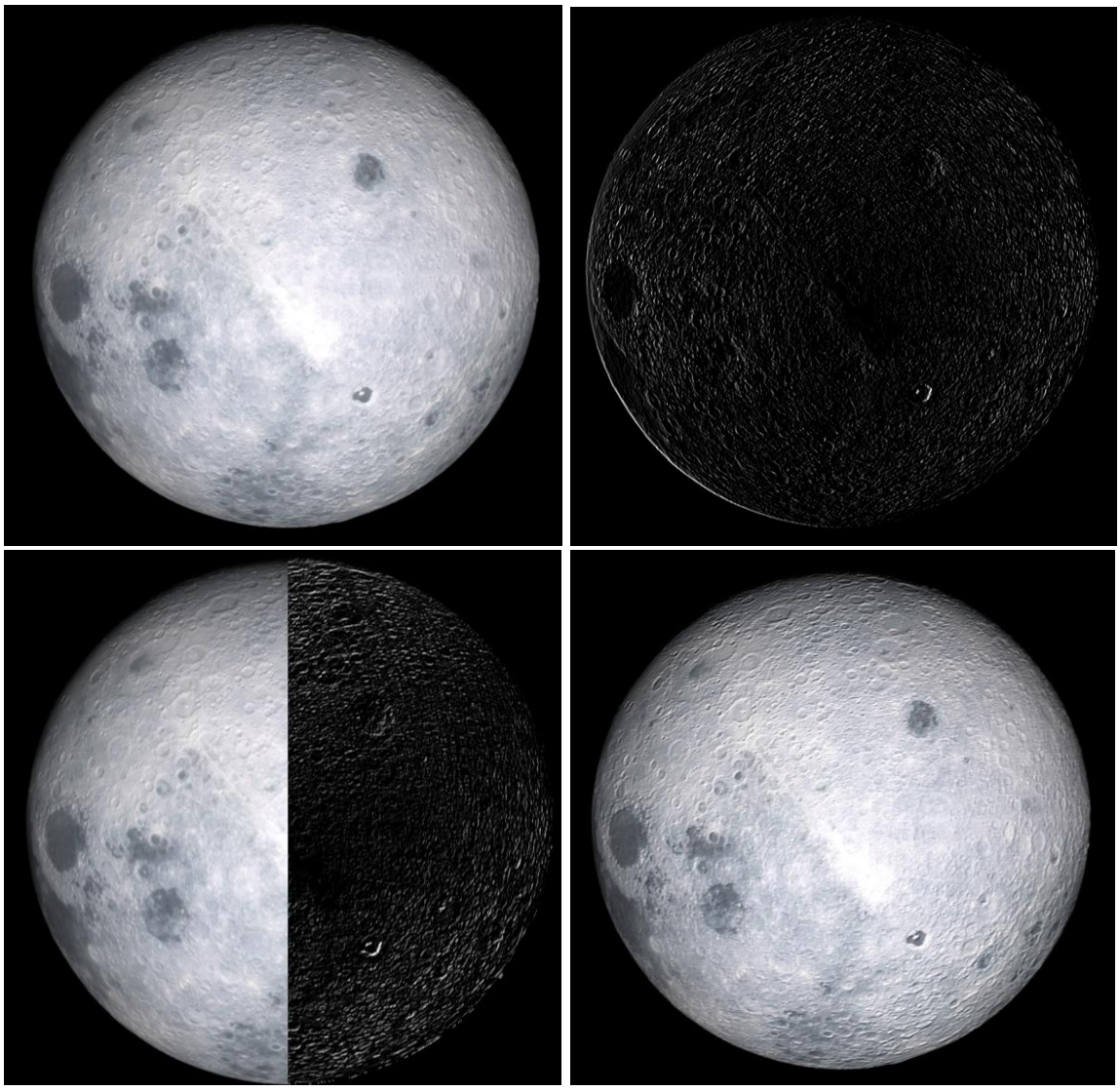


Figure 5-1: Sobel_IP results

The image sharpening produces the following results with left original image and right the sharpened image:

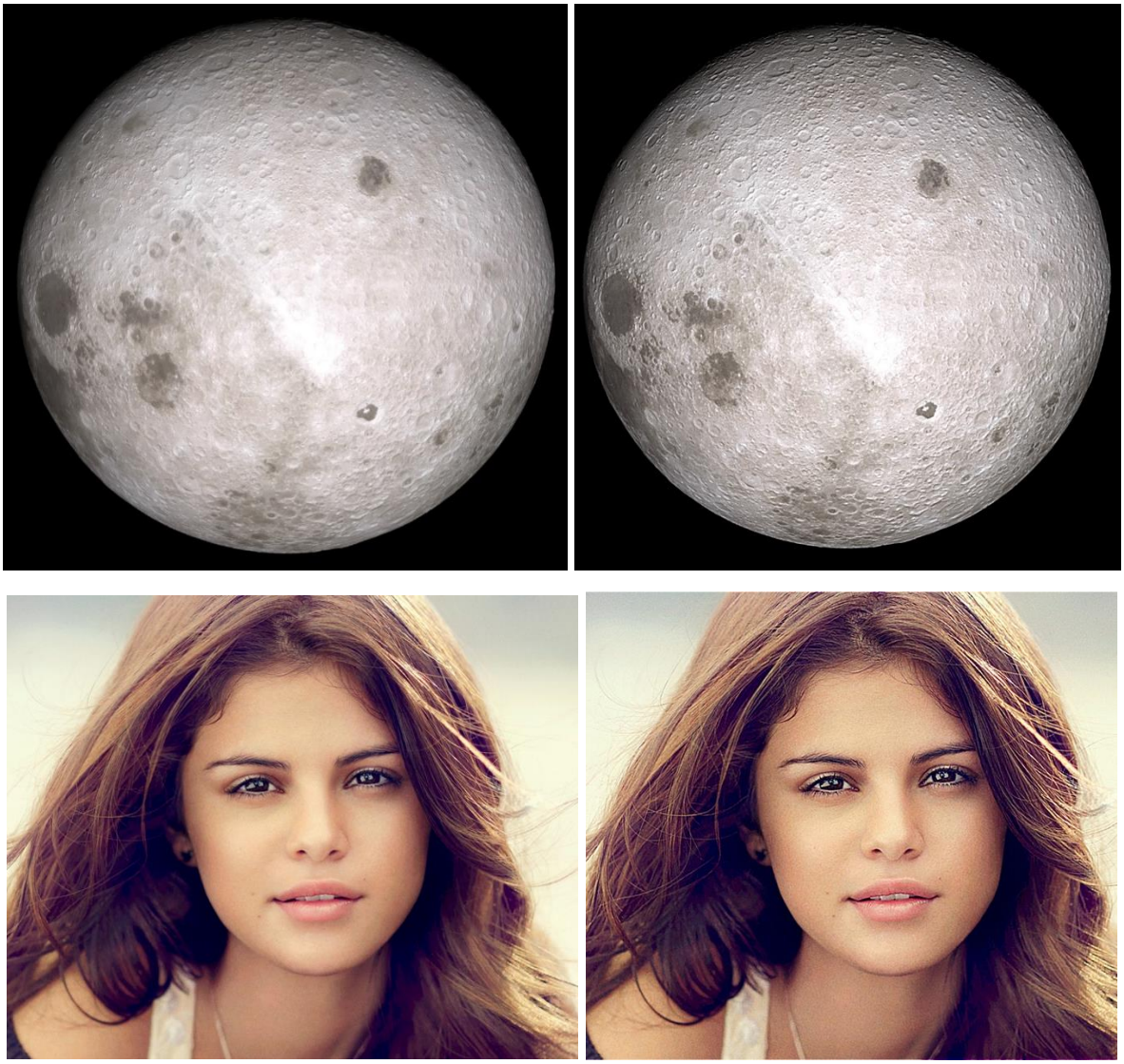


Figure 5-2: Result Image sharpening

Hardware resources

Sobel_IP:

Summary

Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	-	-	-
Expression	-	-	-	-
FIFO	120	-	1466	5111
Instance	36	48	5135	6751
Memory	-	-	-	-
Multiplexer	-	-	-	-
Register	-	-	11	-
Total	156	48	6612	11862
Available	280	220	106400	53200
Utilization (%)	55	21	6	22

Note: The BRAM usages can be lowered by lowering the depth of the pixel buffers: img_colorMode_copy1_10S3 and img_colorMode2_copy1_10S3.

Hardware resources Sharpening_IP:

Summary

Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	-	-	-
Expression	-	-	-	-
FIFO	63	-	1006	3346
Instance	11	35	8153	10973
Memory	-	-	-	-
Multiplexer	-	-	-	-
Register	-	-	13	-
Total	74	35	9172	14319
Available	280	220	106400	53200
Utilization (%)	26	15	8	26

The Sharpening_IP needs less hardware resources because it's made for a specific task. All calculations are done on a one channel data image (8 or 10 bits) which lowers the DSP and BRAM usage allot. Here the BRAM usage can also be lowered by lowering the depth of the pixel buffers.

Frames per second:

The Sobel_IP runs on a stable 60 frames per second but the Sharpening_IP runs on a 55-56 frames per second for a unknown reason. The reason for this is most likely a sync problem.

5.5 Comparison Software vs hardware

The video processing IP block is placed in the video stream. This block uses a 142 MHz clock, which is the same as the pixel clock outputted in the HDMI-in frontend. This means that the 60-fps can be kept using the hardware overlay.

To test what fps the software has, the following code is run in Python:

In [8]:
```
import cv2
import numpy as np

numframes = 10
grayscale = np.ndarray(shape=(hdmi_in.mode.height, hdmi_in.mode.width),
                       dtype=np.uint8)

result = np.ndarray(shape=(hdmi_in.mode.height, hdmi_in.mode.width),
                    dtype=np.uint8)

start = time.time()

for _ in range(numframes):
    inframe = hdmi_in.readframe()
    cv2.cvtColor(inframe,cv2.COLOR_BGR2GRAY,dst=grayscale)
    inframe.freebuffer()
    cv2.Sobel(grayscale, cv2.CV_8U,1,0, dst=result)
    outframe = hdmi_out.newframe()
    cv2.cvtColor(result, cv2.COLOR_GRAY2BGR,dst=outframe)
    hdmi_out.writeframe(outframe)

end = time.time()
print("Frames per second:  " + str(numframes / (end - start)))
```

```
Frames per second:  1.4096244493570476
```

The OpenCV library is used to process the images. In this example, a new image frame is read. The frame is converted to gray and sent to a Sobel X filter. This frame is then send to the output. This process results in a frame rate of 1.4096 frames per second which is remarkably lower than the 60 fps that is achieved in hardware.

Now let's recreate our Sobel X+Y function:

```
for _ in range(numframes):
    inframe = hdmi_in.readframe()
    cv2.cvtColor(inframe,cv2.COLOR_BGR2GRAY,dst=grayscale)
    inframe.freebuffer()
    cv2.Sobel(grayscale, cv2.CV_8U,1,0, dst=sobelX)
    cv2.Sobel(grayscale, cv2.CV_8U,0,1, dst=sobelY)
    result = cv2.addWeighted(sobelX,1,sobelY,1,0)
    outframe = hdmi_out.newframe()
    cv2.cvtColor(result, cv2.COLOR_GRAY2BGR,dst=outframe)
    hdmi_out.writeframe(outframe)
```

This process results in a frame rate of 1.0857 frames per second which means that when more steps are taken (more convolutions), the frame rate drops. The maximum number of steps in hardware is determined by the amount of hardware resources and a stable 60fps can be kept.

6 CONCLUSION

A wide variety of IP's and overlays are made. This project started with building a custom video overlay. Here an overlay is created that can only be used for video processing. After this a simple adder IP block was created to test if it's possible to create our own IP's and use them in our python environment. The Axi-Lite communication between the processing system and the programable logic is also tested here. Next chapter was about the video processing, how the signal is transmitted and some simple IP's reading and processing the signal.

When it is known how to read the data, perform calculations and write the data back, it is possible to use this knowledge and create video filters. In this project a high pass filter is made using the Sobel operators for X and Y direction. The functionality of these IP's is simulated in a C test bench and when this looked fine, it was made into an overlay. A few changes were made to the filters so the negative values are kept, which is important for the image sharpening in the Sobel and Sharpening IP.

Finally, the system is compared to a software video filter. In this system, the frame rate between 1 – 2 frames per second. Compared to our 60 frames per second with hardware logic. Because video processing requires processing large data in high speed, it makes this a very suitable task for a FPGA that is capable of parallel processing. Our project illustrated the power of an FPGA on processing large amount of data.

Televisions perform sharpening in a similar way like our Sharpening IP. The difference is that in our system using the PYNQ, the hardware can be reprogrammed with a software update.

Problems encountered during this project were mainly sync based (different latency in pipelines, no buffers,…). Most of the problems were beginner mistakes were quickly fixed using the HLS User manual [11] and the Xilinx forums [12]. Some limitations of HLS were discovered like switching streams in a dataflow region, which can be fixed in VHDL by using a Axi-stream switch. Overall, HLS gave a promising impression and will be used again in future projects.

The conclusion of this project is that it is possible to accelerate video processing using the PYNQ hardware and that it can be used in other applications that require large and fast data processing.

6.1 Future work

The goal of this project was to learn the PYNQ environment and use it to accelerate software processes in hardware logic. Basic filters are made here. These filters can be used in more complex systems, described in the book Digital Image Processing [13], such as:

- Canny edge detector [page 729]
- Image thresholding [page 742]
- Image Eroding and Dilating [page 638]
- And more if possible

7 REFERENCES

[1] „PYNQ,” [Online]. Available: http://www.pynq.io/.

[2] „PYNQ Base Overlay,” [Online]. Available: https://pynq.readthedocs.io/en/latest/pynq_overlays/base_overlay.html.

[3] A. Taylor, „Adam Taylor’s MicroZed Chronicles, Part 157,” [Online]. Available: https://forums.xilinx.com/t5/Xcell-Daily-Blog/Adam-Taylor-s-MicroZed-Chronicles-Part-157-Exploring-PYNQ-s-Base/ba-p/734085.

[4] Y. Tavares, „Creating a simple Overlay for PYNQ-Z1 board from Vivado HLx,” [Online]. Available: https://yangtavares.com/2017/07/31/

creating-a-simple-overlay-for-pynq-z1-board-from-vivado-hlx/.

[5] „AXI4-Stream Video IP,” [Online]. Available: https://www.xilinx.com/support/documentation/ip_documentation/axi_videoip/v1_0/

ug934_axi_videoIP.pdf.

[6] „AXI Video Direct,” [Online]. Available: https://www.xilinx.com/support/documentation/ip_documentation/

axi_vdma/v6_3/pg020_axi_vdma.pdf.

[7] „image_filter_test,” [Online]. Available: https://github.com/Xilinx/HLx_Examples/blob/master/Vision/video_edge/image_filter_test.cpp
.

[8] „HLS Video Library,” [Online]. Available: http://www.wiki.xilinx.com/HLS+Video+Library.

[9] „pragma HLS pipeline,” [Online]. Available: https://www.xilinx.com/html_docs/xilinx2018_1/sdsoc_doc/oyc1517254361139.html.

[10] „pragma HLS loop_flatten,” [Online]. Available: https://www.xilinx.com/html_docs/xilinx2018_1/sdsoc_doc/hid1517254361170.html.

[11] „Vivado Design Suite User Guide UG902,” [Online]. Available

: https://www.xilinx.com/support/documentation/sw_manuals/xilinx2014_1/

ug902-vivado-high-level-synthesis.pdf.

[12] „community Forums,” [Online]. Available: https://forums.xilinx.com/.

[13] ,. R. E. w. Rafael C. Gonzalez, Digital image processing, Pearson.

[14] „Gaussian Kernel Calculator,” [Online]. Available:

http://dev.theomader.com/gaussian-kernel-calculator/.

[15] „Asynchronous Widgets,” [Online]. Available: http://ipywidgets.readthedocs.io/en/latest/examples/Widget%20Asynchronous.html.

[16] Community Forums. [Online]. Available: https://forums.xilinx.com/