KU LEUVEN

FACULTEIT INDUSTRIËLE
INGENIEURSWETENSCHAPPEN

CAMPUS BRUGGE

Hardware acceleration of video processing

Pieter BERTELOOT

Supervisor: Sammy Verslype

Academic year: 2017-2018

Abstract

Inhoud

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 are discussed in this report how the final project is realized and can be used as beginner guide for new PYNQ users. All HSL, bit, tcl, python files are 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 runes:

- Linux
- Python
- Jupyther 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 overlays are like software libraries, the programmer can select the overlay that 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. Overlays are discussed in detail in the next chapter.

2 OVERLAYS

Overlays are programmable and configurable FPGA designs. These overlays 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. To gain a better understanding about overlays, we will take a closer look at the base overlay.

2.1 Base overlay

The base overlay allows the PYNQ to use the peripherals (video, audo, 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 take a look what’s inside this base overlay. To do this, we must 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 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.

2.2 Rebuilding the base overlay

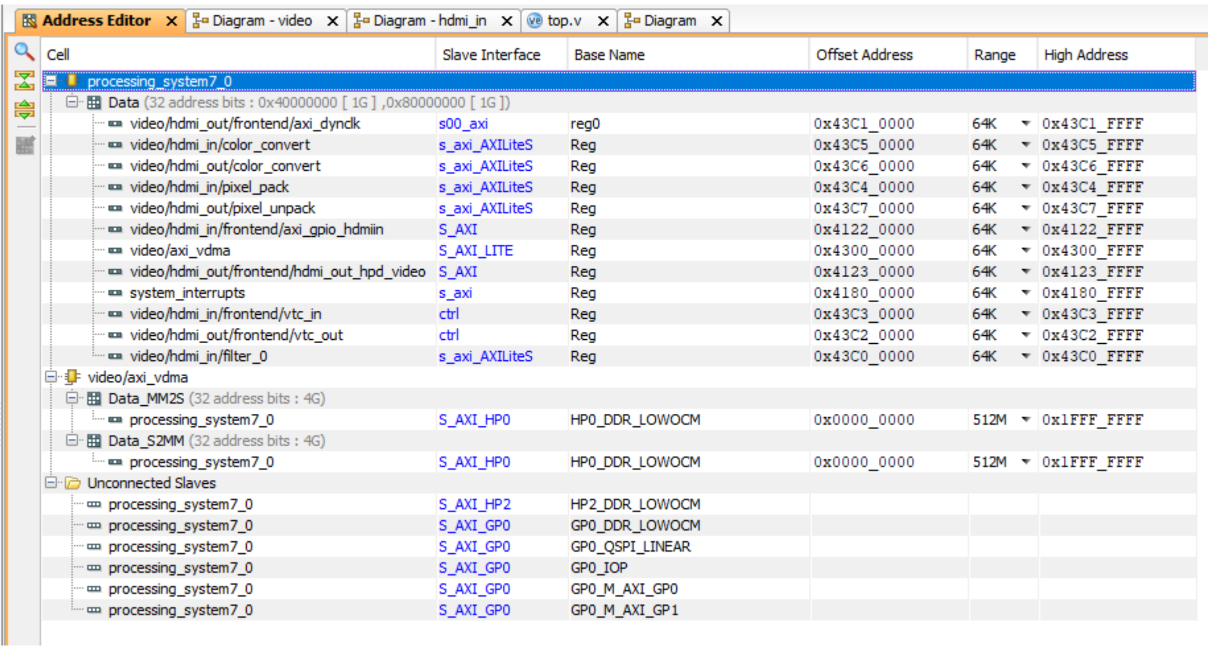
Every peripheral can be found in the base overlay. The routing of these blocks takes time and hardware so to reduce this, we will only keep the components that are needed for video streaming/processing. Our edited base overlay can be found on the next page.

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 block 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** is also a very important subject in the IP Integrator. Here we can see the the Offset Address of each IP. This address will later be used for **Memory-mapped I/O** (MMIO). When a new IP, that has AXI-Lite communication, 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

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

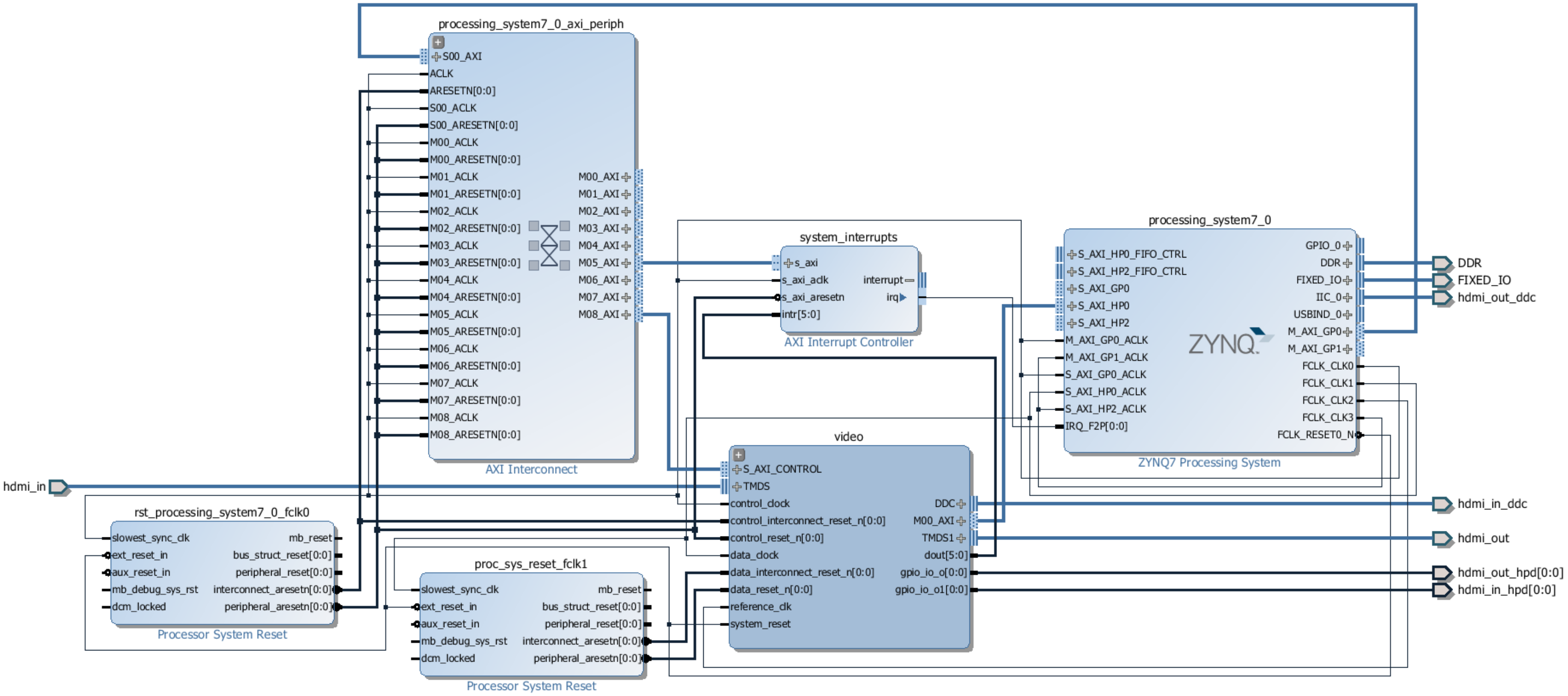


Figure 2-2

2.3 Creating our first IP

Now that we know how to create, edit and communicate with the overlay, we can 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. HLS is used to transform 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, we start with importing C++ libraries so we can use the Fixed-Point Data Types and Integer Data Types.

Then we have our TOP function. 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 on these to specify the IO protocol ports. We use the axilite protocol for communication. The pragmas define this protocol.

And at last, the functionality of the IP is programmed. When this is done, C synthesis and RLT export can be performed.

Now we can 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 periph.

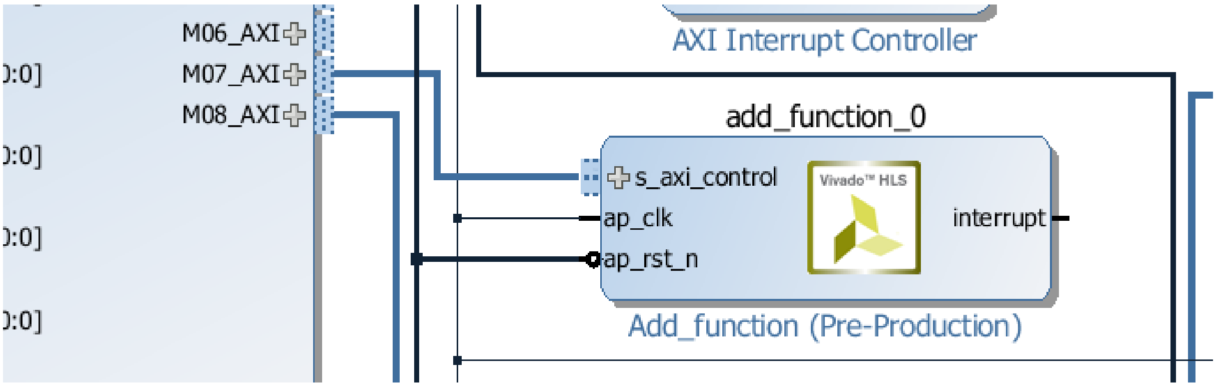
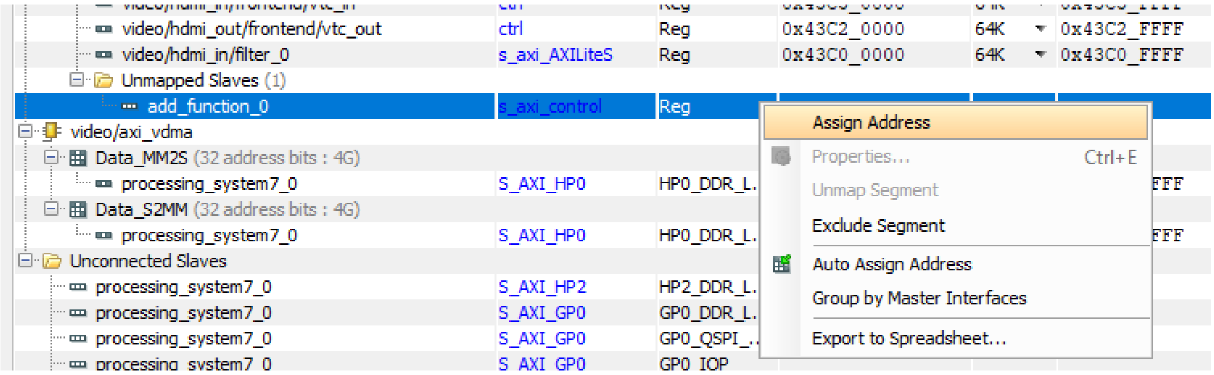


Figure 2-3

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



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
Unmapped Slaves (1)					
add_function_0	s_axi_control	Reg			
video/axi_vdma					
Data_MM2S (32 address bits : 4G)					
processing_system7_0	S_AXI_HP0	HP0_DDR_L...			FFF
Data_S2MM (32 address bits : 4G)					
processing_system7_0	S_AXI_HP0	HP0_DDR_L...			FFF
Unconnected Slaves					
processing_system7_0	S_AXI_HP2	HP2_DDR_L...			
processing_system7_0	S_AXI_GP0	GP0_DDR_L...			
processing_system7_0	S_AXI_GP0	GP0_QSPI_...			
processing_system7_0	S_AXI_GP0	GP0_IOP			

Figure 2-4

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. Tabel 2-1 shows the signals that our important for us.

Tabel 2-1

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 we are ready to generate our BIT- and TCL file. To do this generate the bitstream and run the following code in the Tcl console:

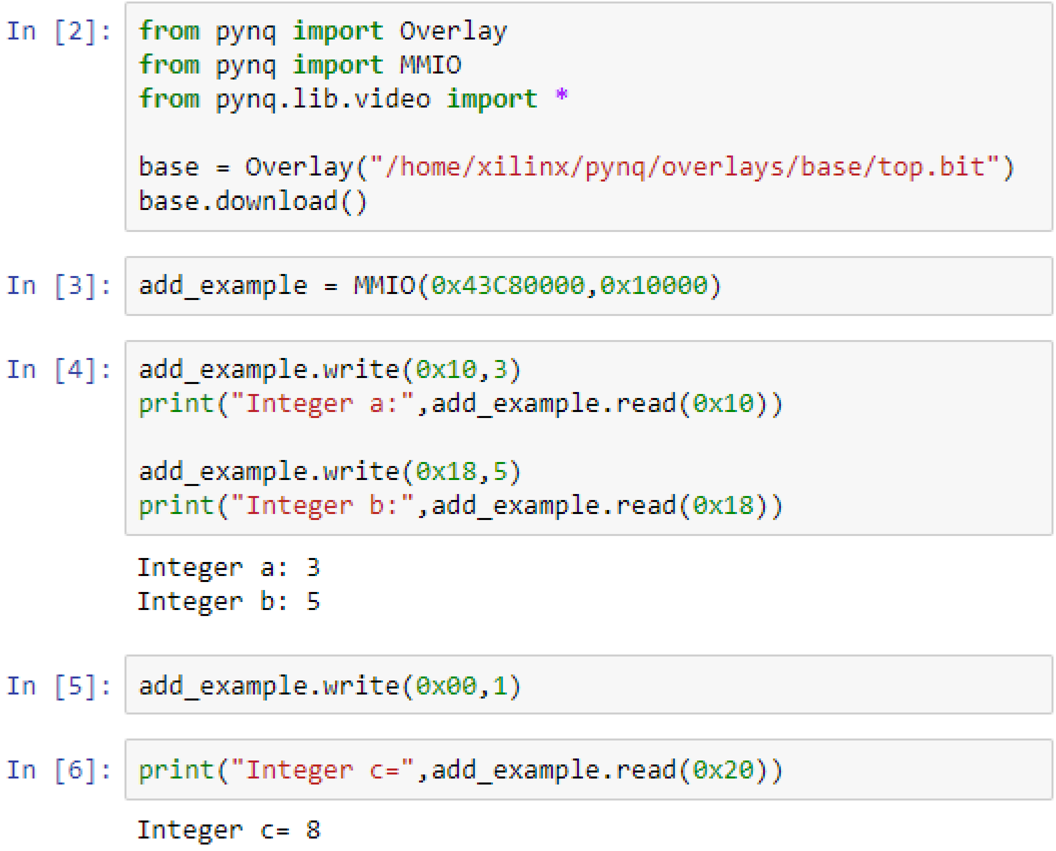
```
write_bd_tcl top.tcl
```

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

Once this is done, copy the BIT-and TCL file to the following location:

\\192.168.2.99\xilinx\pynq\overlays\base

And run the code seen in Figure 2-5 in a notebook.



```
In [2]: from pynq import Overlay
        from pynq import MMIO
        from pynq.lib.video import *

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

In [3]: add_example = MMIO(0x43C80000,0x10000)

In [4]: add_example.write(0x10,3)
        print("Integer a:",add_example.read(0x10))

        add_example.write(0x18,5)
        print("Integer b:",add_example.read(0x18))

        Integer a: 3
        Integer b: 5

In [5]: add_example.write(0x00,1)

In [6]: print("Integer c=",add_example.read(0x20))

        Integer c= 8
```

Figure 2-5

Now that we have successfully created and use the overlay, we can use this to create more complex systems. In the next chapter will the focus be on creating the Sobel edge detection filter in a video stream.

3 VIDEO PROCESSING

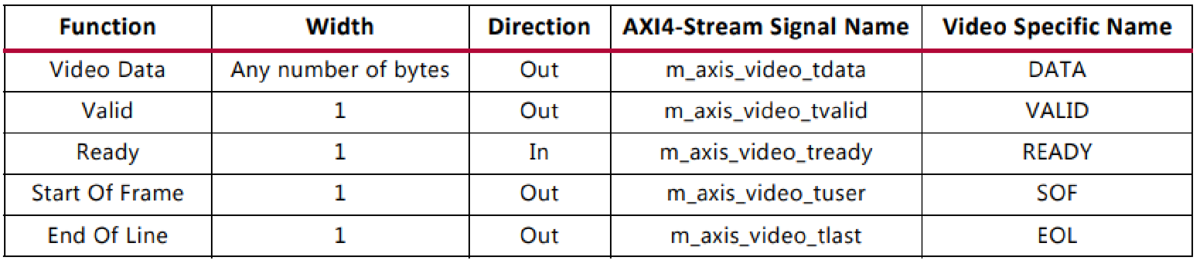
3.1 Video signal

Before the video can be processed, we will take a closer look on the video signal is transmitted in the base overlay. Here we can divide the video processing in different parts:

- HDMI in
    - Frontend
    - Color_convert
    - Pixel_pack
        - Dvi2RGB decoder
        - Video in to axi4-stream
- VDMA
- HDMI out
    - Pixel_unpack
    - Color_convert
    - Frontend

3.1.1 Frontend

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



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

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 for efficient high-bandwidth access between the AXI4-+stream video interface and the AXI4 interface. This IP reads and writes frames to the memory.

3.1.3 HDMI out

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

3.1.4 Video pipeline

For more information about the video pipeline and how to use it in the PYNQ notebooks can be found in the hdmi_video_pipeline notebook.

3.2 Processing the signal

The video processing will take place in the HDMI-in package show in Figure 3-1

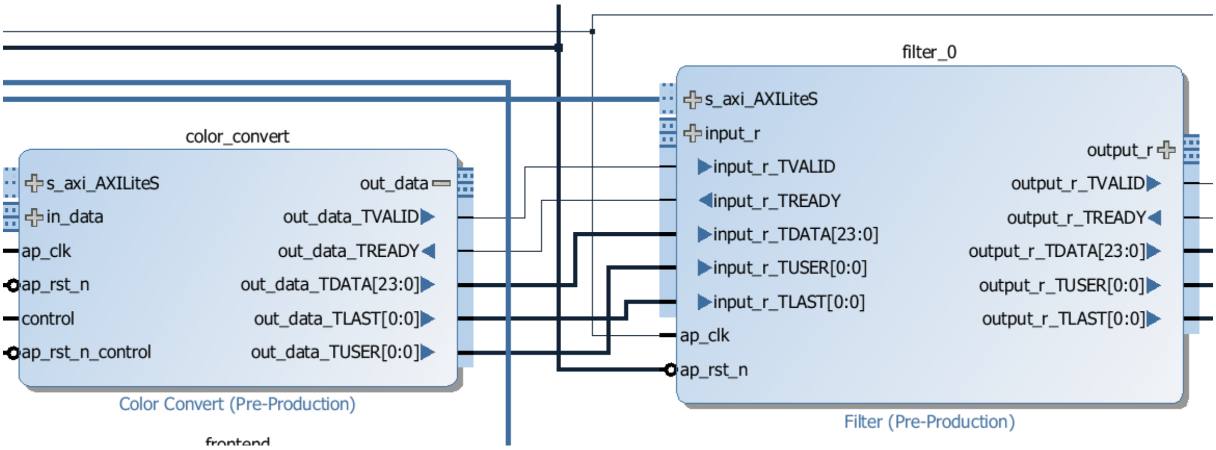


Figure 3-1

3.2.1 Our first video processing: Screen splitter

Let's start with a simple video processing system. In this project we will create an IP that:

- Splits the screen in 2 parts
  - First part: Full original image passes
  - Second part: Only the red component passes
- The split can be defined in real time on what column it will be

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, we 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

Now we must define what 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, we can create the TOP function. For this project we have 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 interface these input and output signals have. Pragmas are used to define this. An axis interface is used for the video stream and an axilite interface is used for the 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 streamed in sequentially. 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;
```

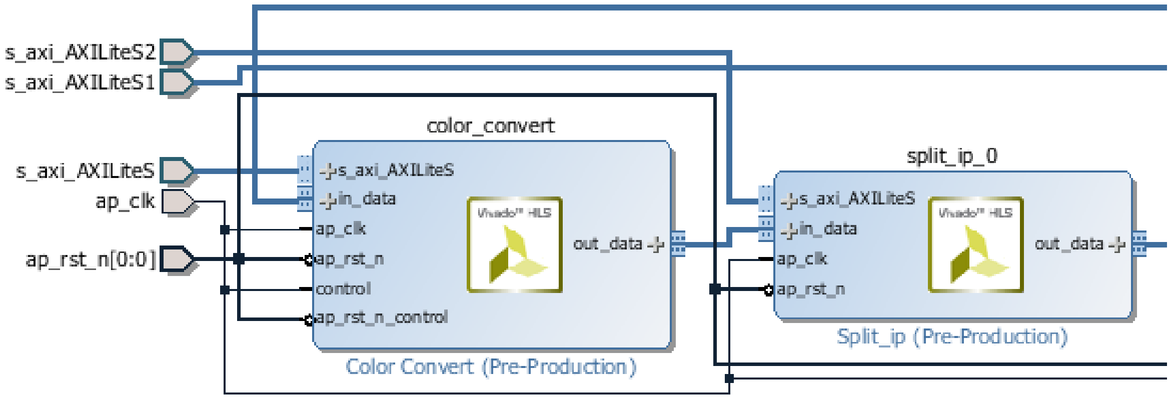
When the data is read, it is checked in which column this pixel is located. The column counter is reset when the end of line signal is high.

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

if(in_data->last)
    col = 0;

coll++;
```

After this we can assign the out variable to the output stream.

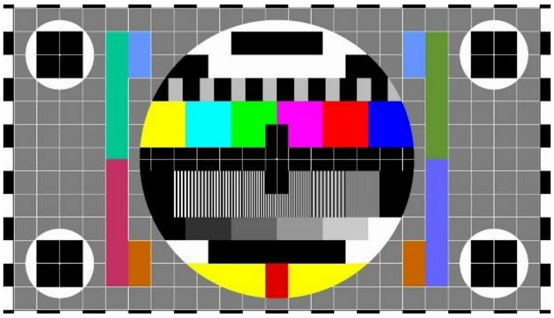
```
out_data->data.p1 = out1.range();
out_data->data.p2 = out2.range();
out_data->data.p3 = out3.range();
```

After synthesizing and exporting, add the IP in the vivado block design like this:

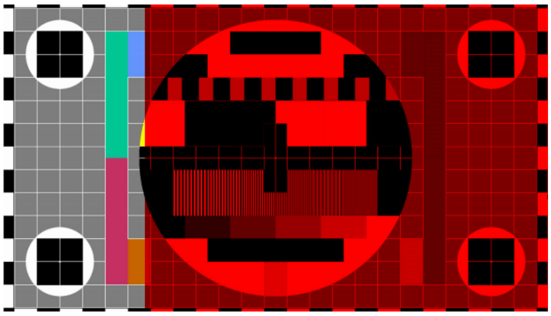


Note: Don’t forget to connect the axilite interface to the axi_interconnect IP and map the new IP in the address editor.

Generate the tcl and bit files and import the overlay in PYNQ the same way as in the adder IP. Write a value to 0x10 to define where the split will be. The result should be:



Input



Output

3.2.2 C simulation and test bench

Building the overlays take allot 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. This helps remove this coding practice.

- **Use of hls::mat**

hls::mat represent an image in HLS Video Library. 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 sperate 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 also 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**

In the test bench we will load an image, convert it to an axi stream, send it through our IP and convert 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 exactly the same as in 3.2.1. The only difference is that we now use RGB_IMAGE’s as input and output. These are hls::mat objects and can be seen as frames. In the function we have 2 loops where we iterate over all the pixels in the frame.

Then we have 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 for a function or loop by allowing the concurrent execution of operations.

More information about pragmas can be found here:

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

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

Once the split function is made, we can easily call 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 H file is made so we can include our project 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"
```

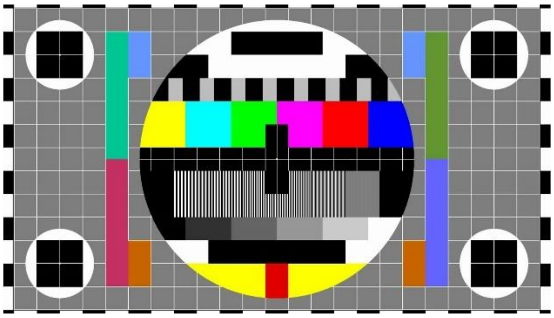
Also place the input image in the HLS project directory. The test bench code is:

```
#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);
}
```

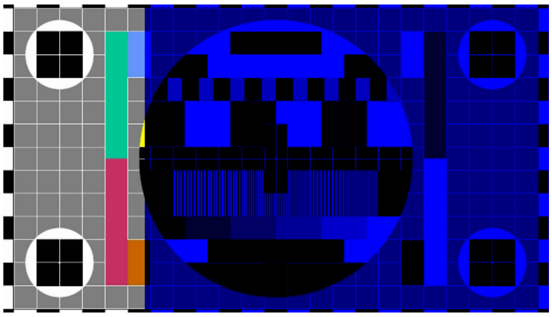
The opencv functions are used to load, convert and save images. If this is all done, run the C simulation and an image should appear in the following directory:

```
<hls project>\solution1\csim\build
```

The result:



Input



Output

Notice that the Output is now blue instead of red. This is because the color mode is by default BGR but we use RGB in our 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 edit the original screen splitter to a RGB and GRAY screen. To transform 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 is used for:

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

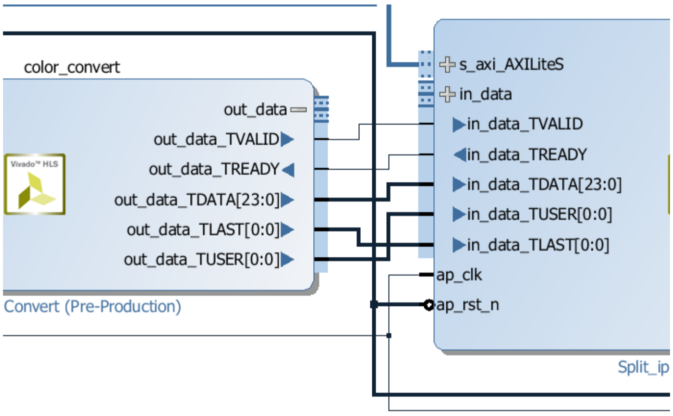
The result of the calculation also needs to be stored.

```
    char gray;
```

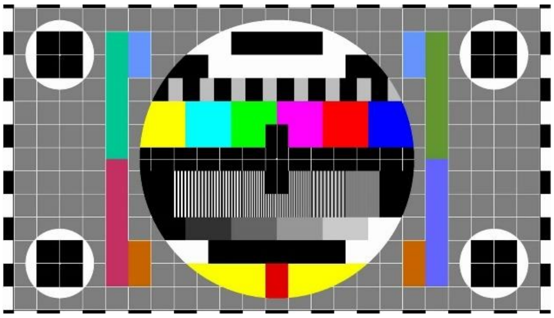
Now the gray value can be calculated. Change the code from the original split to the following:

```
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;
```

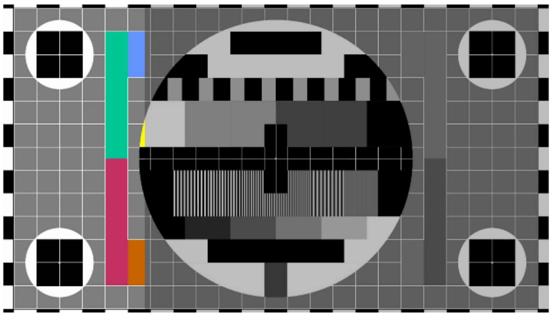
Because the interfaces have change, the inputs and outputs of the IP have change too.



The result:



Input



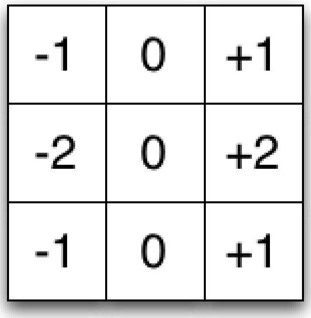
Output

4 VIDEO FILTERS

It’s possible now to read an incoming signal, process it, write it back to the output and transfer it back to hdmi. 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.

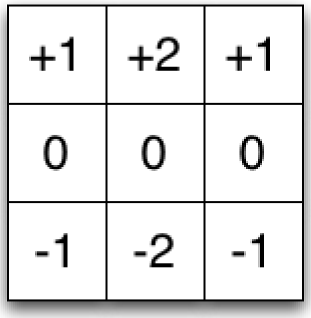
4.1 Edge detection

Edge detection is used to identify and locate discontinuities in an image. The discontinuities can be detected by checking the change in pixel intensity (high pass filter). The most common way to detect these is to convolute 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:



-1	0	+1
-2	0	+2
-1	0	+1

x filter



+1	+2	+1
0	0	0
-1	-2	-1

y filter

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 RBG image as input and converts it to a gray image. A other function is needed to calculate the Sobel filter in X or Y direction:

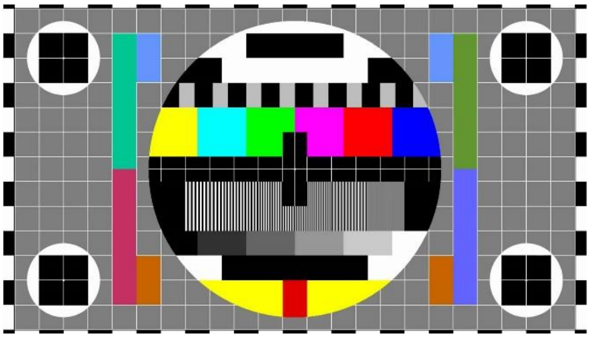
```
void sobel(
            RGB_IMAGE& img_in,
            RGB_IMAGE& img_out,
            char direction
        ) {

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

The hls::sobel function has the following template:

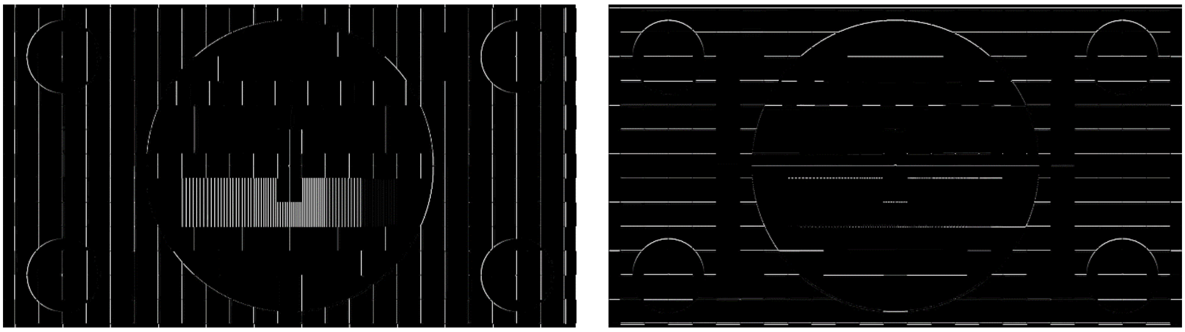
```
template<int XORDER, int YORDER, int SIZE, int ROWS, int COLS, int SRC_T,
int DROWS, int DCOLS, int DST_T>
```

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

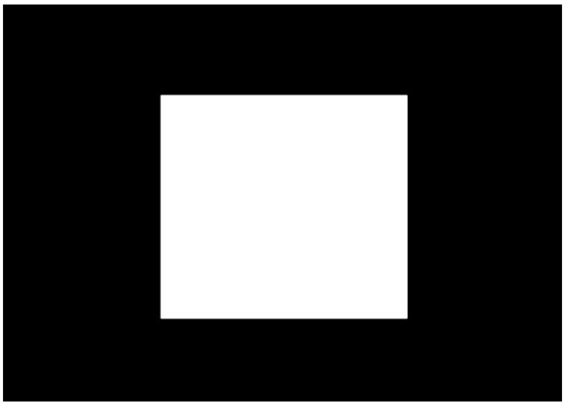


Output X **Output Y**

Note: Our RGB image type is HLS_8UC3:

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

This is a 8 bit unsigned char with 3 channels (R, G, B). This means that negative values cannot be represented in our image. This means that we lose information. To demonstrate this, take the following picture as input:



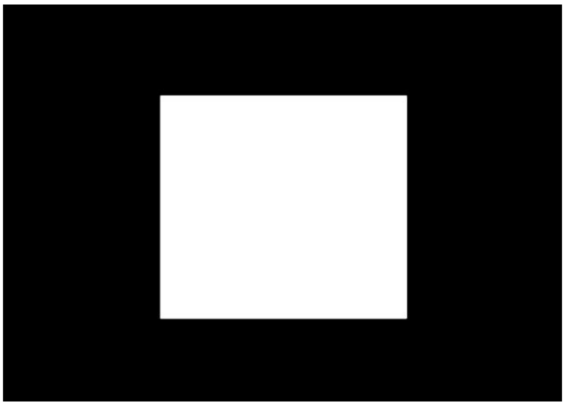
this input from left to right, we have a transition from perfect black <0,0,0> to perfect white <255,255,255> and a transition from perfect white to perfect black.

When this transition happens, the following calculation is made:

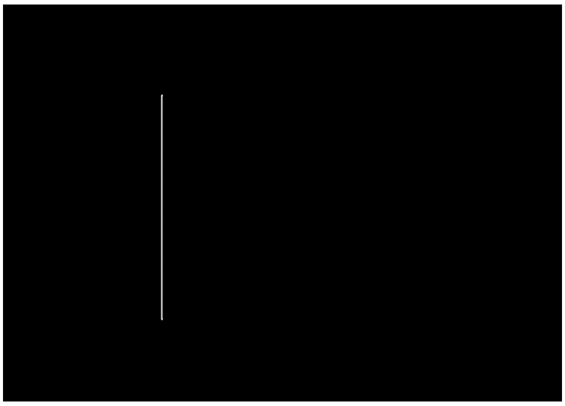
$$pixel\ left\ transition = \begin{bmatrix} 0 & 255 & 255 \\ 0 & 255 & 255 \\ 0 & 255 & 255 \end{bmatrix} * \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix} = 1 * 255 + 2 * 255 + 1 * 255 = 1020$$

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

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

4.3 Sobel X + Y

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

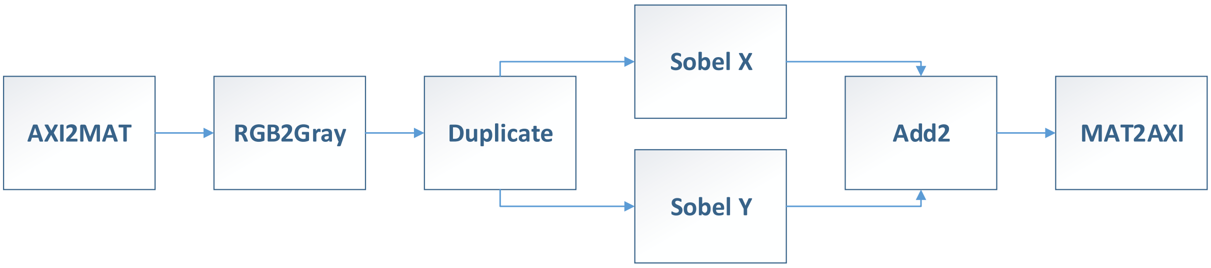


Figure 4-1: block diagram sobel X+Y

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 with unsigned char.

$$XY = X + Y$$

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. This function is used to add the Sobel X and Y results. These can be obtained using the Sobel function in 4.2. As seen in the block diagram Figure 4-1, the input image must be duplicated so the Sobel functions can be used on the same image.

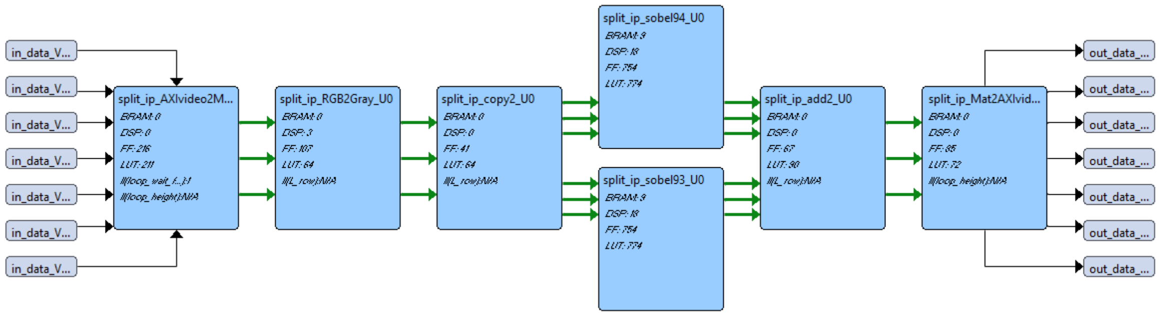
```
void copy2(RGB_IMAGE& img_in, RGB_IMAGE& img_out0, RGB_IMAGE& img_out1) {

      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 to 2 outputs. Now these functions can be used in our TOP function.

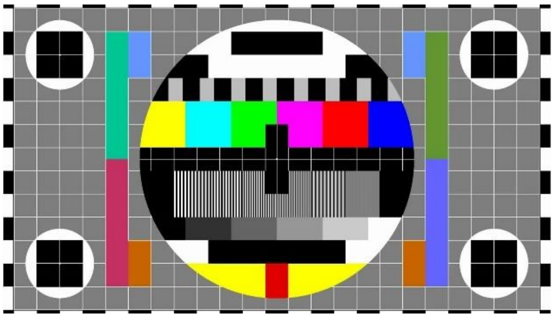
```
// Convert AXI4 Stream data to hls::mat format
hls::AXIvideo2Mat(in_data, img_0);

//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 add some more RGB images.

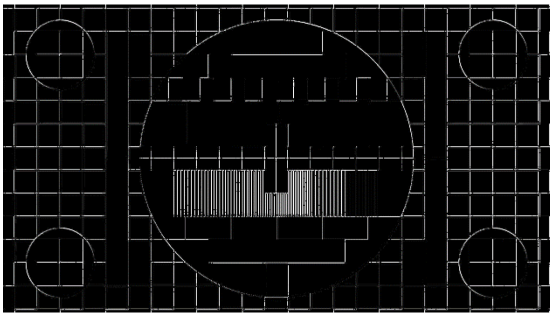
It’s always good to have a look in the Analysis tab. The schedule and dataflow can be seen. This can be used to check your system before building it.



The result of this should be:



Input



Output

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.

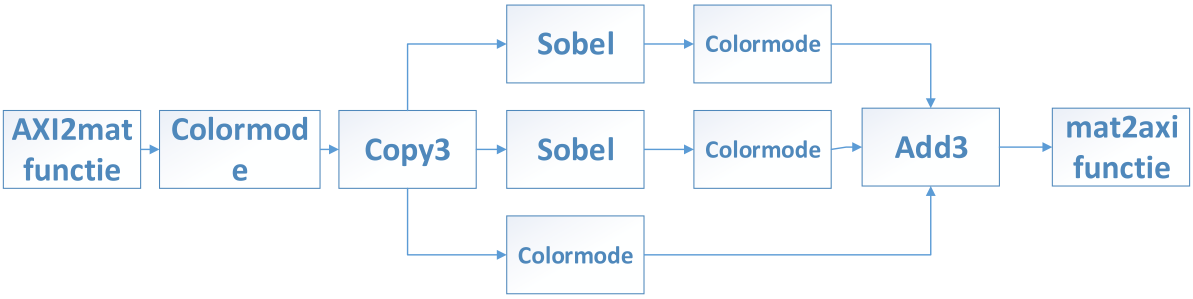


Figure 4-2: Block diagram system

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];                      RGB
                     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];                 GRAY
                           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];
                                                                         SPLIT
                       } 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;
                                                                       Black
                 } else {
                     pout.val[0] = pin.val[0];
                     pout.val[1] = pin.val[1];
                     pout.val[2] = pin.val[2];

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

The next step is to modify the Copy2 to a Copy 3 function. When this is done, we can reuse the Sobel and Colormode functions. 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 synq and won't be able to be added in de 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.

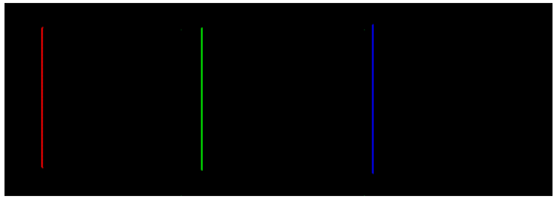
There will be 5760 pixels stored in this buffer so the 3 lines will be in synq again. hls::mat is basically the same as a stream (mat is implemented as a hls::stream). So, the depth of the image can be set where we want to add the buffer.

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

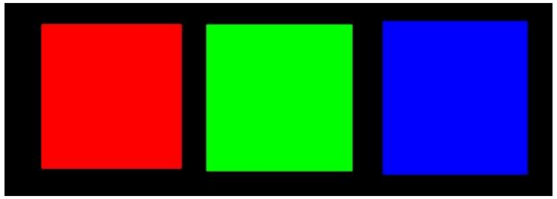
Now the only thing that needs to be done is making the add3 function. This function adds 3 images. The intensity of each image can also be changed in this function.

```
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;
        }
    }
}
```

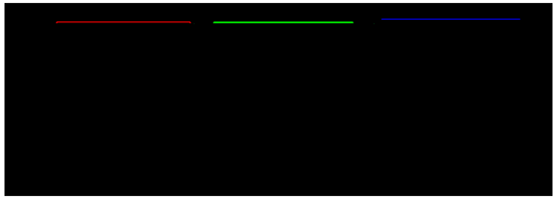
The result:



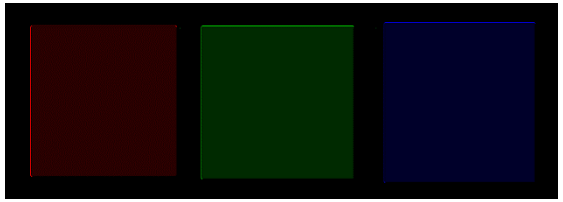
Sobel X



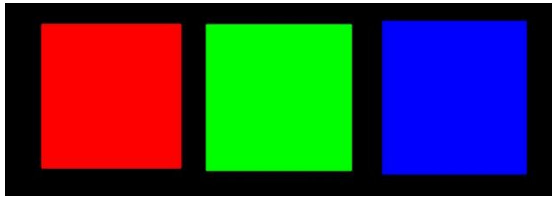
Input



Sobel Y



Output



Input

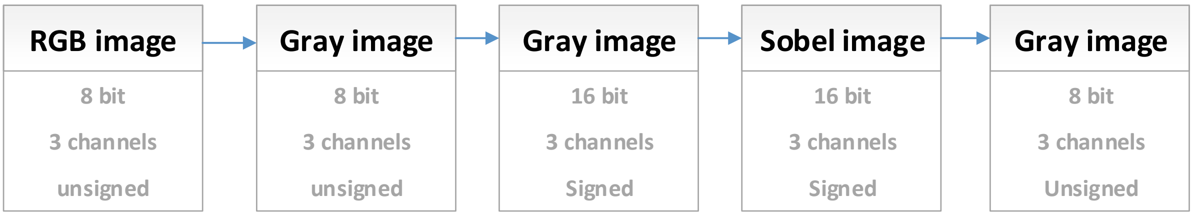
4.5 Sobel negative values

Let’s take a closer look 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’s not used in previous projects.

To represent these negative values, it’s possible to multiply the X and Y kernel by -1. This will only output the “negative” values and 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:



The following functions must be made:

- convertToSigned
- convertToUnsigned

First, a new type of image must be made to store the 16-bit data.

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

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

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

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

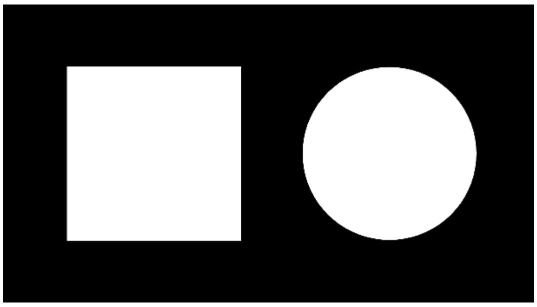
```
void convertToSigned(RGB_IMAGE& img_in0, RGB16_IMAGE& img_out){

      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 to a RGB_IMAGE. This function is like the convert to unsigned function but now the absolute value must be taken to convert it to unsigned.

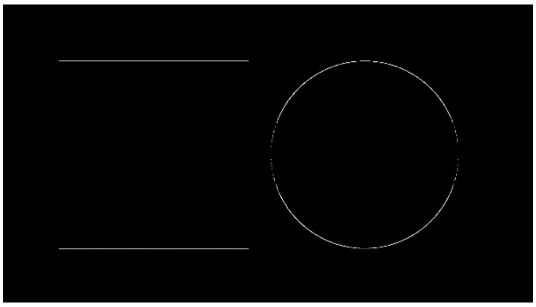
```
img_in0 >> pin;
if (pin.val[0] >= 0)
      pout.val[0]=pin.val[0];
else
      pout.val[0]=-(pin.val[0]+1);

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

if (pin.val[2] >= 0)
      pout.val[2]=pin.val[2];
else
      pout.val[2]=-(pin.val[2]+1);

img_out << pout;
```

Note: Don't forget to change the image type in the sobel function to RGM16_IMAGE.

The result:



Input



Output

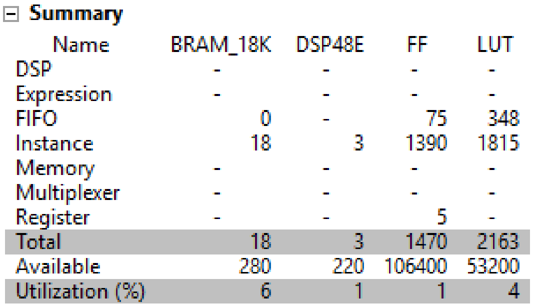
The data is kept as a short which is 16 bits. This has a range from −32,768 to 32,767. Most values will never be used and will take unnecessary resources. Let's change it to 10 signed bits because the HLS_10SC3 data type already exists in hls_video_types.h. This is a 3 channel 10-bit signed data type.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

Image type:

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

Pixel type:

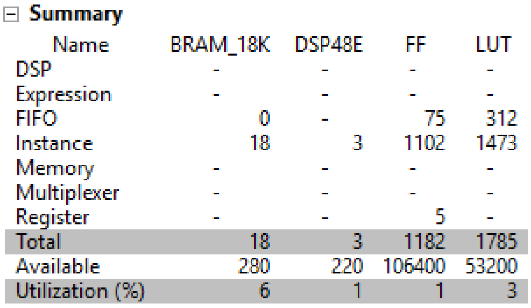
```
typedef hls::Scalar<3, ap_int<10> > RGB10_PIX;
```



⊟ **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 4-3: 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 4-4: 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.

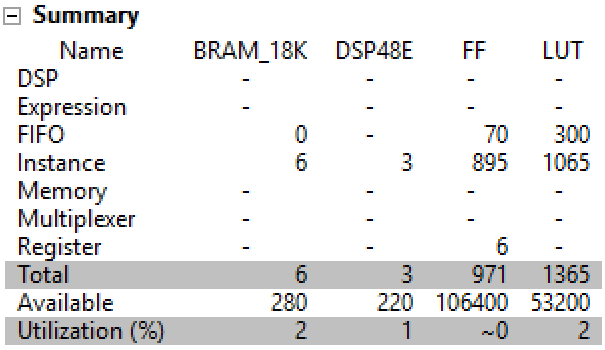
The resources can be lowered even more by using only 1 channel for the gray images. Using 3 channels for gray images is useless because the values in the channels are all the same. This can easily be done by adding 2 more image and pixel types:

```
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

The amound of BRAM is divided by 3 because we use 3 times less data. The FF’s and LUT’s have also decreased.

5 COMPARATION 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 we will keep our 60-fps using the hardware overlay.

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

```python
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.Laplacian(grayscale, cv2.CV_8U, 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)))
```

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 Laplacian filter (which uses a different 3x3 kernel). This frame is then send to the output. This process results in a frame rate of 1.3769 frames per second which is remarkably lower than the 60 fps that is achieved in hardware.