High Performance Embedded Systems Design Coursework Report

Felix Bowyer (u2064363)

1 Introduction

1.1 Zynq Platform

Zynq[1] is a Xilinx SoC (System-on-Chip) platform which combines a traditional FPGA programmable logic chip with an ARM Cortex-A9 microprocessor. This allows us to run both compiled software on a microprocessor and custom logic on a single chip, and have them interact. Notably, it allows us to create hardware accelerators which seamlessly interact with software appications, providing a rich platform for high performance, real time embedded applications, such as video or signal processing.

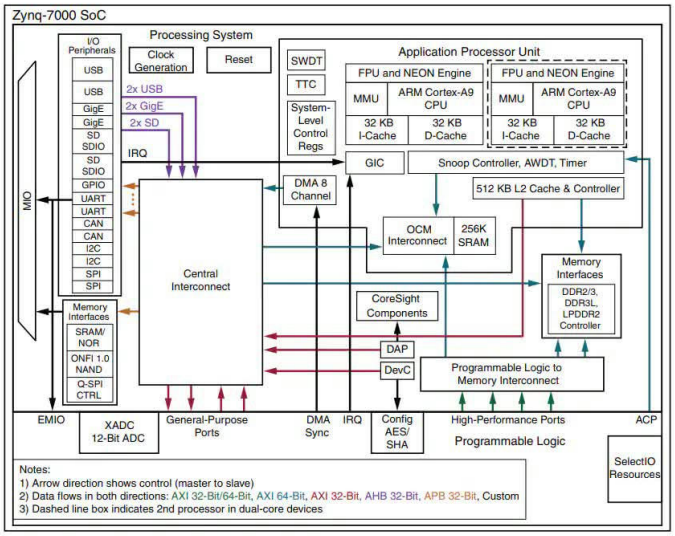

Figure 1: The structure of the Zync-7000 SoC.

The onboard programmable logic consists of Xilinx CLBs, DSPs, block memories, and other compoinents. These are freely configurable, and similar to what is found on any other FPGA chip. Xilinx's Vivado design suite is used to program these devices, and contains tools to integrate custom hardware with software running on the ARM microprocessor. This can give us the flexibility of a software system, whilst getting the benefits of the speed of an FPGA chip when needed.

1.2 HLS Design

High Level Synthesis[2] is a method of deisgning custom logic. Instead of specifying functionality directly in HDL, we use high level languages (usually C or C++), which are converted directly into RTL code which can be implemented on an FPGA chip. This means we can design complex tasks (and verify their functionality in software), whilst running them in hardware and gaining a performance benefit. The scale of this benefit will be evaluated as part of this report for some applications.

1.3 The Task

2 Labs

2.1 Lab 1

This lab is an introduction to development on the Zynq platform, as we run our first code on the ARM processor interacting with hardware.

We first specify our hardware in Vivado, instantiating IP for the Zynq processing system[3] in a block design. Once this is synthesised and implemented for our board (as per the usual FPGA deisgn flow), we export our hardware platform (including bitstream) for use with Vitis.

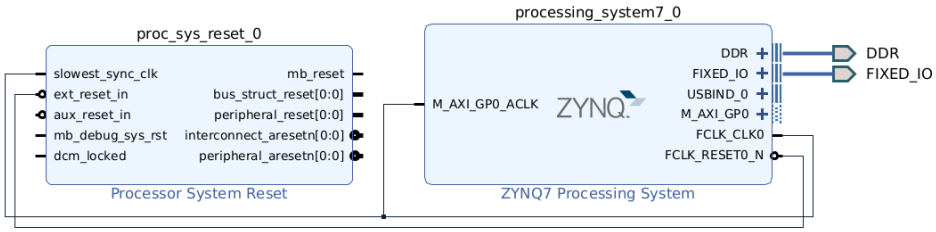

Figure 2: Block design created in Lab 1.

Vitis IDE (which succeeds Xilinx SDK) is the tool we use to create, build and run software applications on hardware we deisgn in Vivado. To do this, we create a new project, importing our exported hardware and using the provided hello world template. This contains an example C program to print "Hello World". We then use Vitis to program our FPGA with the bitstream we generated, and compile and launch our hello world program on the hardware (which includes IP for the ARM microprocessor). Over a serial console, we see the output "Hello World".

2.2 Lab 2

Lab 2 introduces HLS through Vitis HLS, in which we create custom hardware defined in high level languages. We first open Vitis HLS, and create a project. We use the provided C++ code as our sources. This code performs matrix multiplication, and uses 3 loops: the first iterates over the rows of the matrices, the second over the columns, and the third loop multiplies and accumulates the numbers.

Using Xilinx HLS pragmas[4], we can tell the compiler to apply different optimisations ot different parts of the code. Two that are useful are `#pragma HLS PIPELINE`, which pipelines the loop, and `#pragma HLS unroll`, which unrolls the loop. Seven different solutions were trialled.

1. No optimisations made.
2. Unroll the outer loop.
3. Unroll the outer 2 loops, pipeline the inner loop.
4. Pipeline the outer loop.
5. Pipeline the outer loop, unroll the inner 2 loops.
6. Unroll the outer 2 loops.
7. Pipeline the inner loop only.

By changing the optimisations used, we see the following variations in performance and resource usage. Notably, solutions 4 and 5 perform equally (likely due to automatic compiler optimisations), and are the best performing solutions. Solutions 3 and 7 are the worst performing - since the inner loop multiplies and accumulates, parts of which may be done in parallel, it makes sense that enforcing a pipeline here would hurt performance.

Solution	Latency	#DSP	#FF	#LUT
#1	180	33	2825	1924
#2	109	126	9598	4544
#3	386	3	1848	3131
#4	85	75	6491	2935
#5	85	75	6491	2935
#6	109	126	9604	4544
#7	284	3	624	1082

Table 1: Comparison of optimisation solutions for 5x5 matrices.

In general, we see a tradeoff between latency and resource utilisation, in that solutions with lower latency use more resources. It is also interesting how this changes as we change the size of the matrices we are multiplying. Below we see how latency and resource utilisation change with matrix size for solutions 1 and 5.

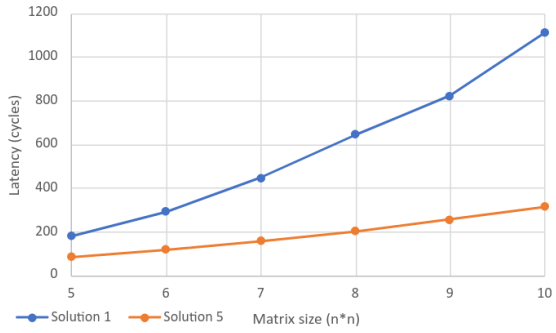

Figure 3: Latency.

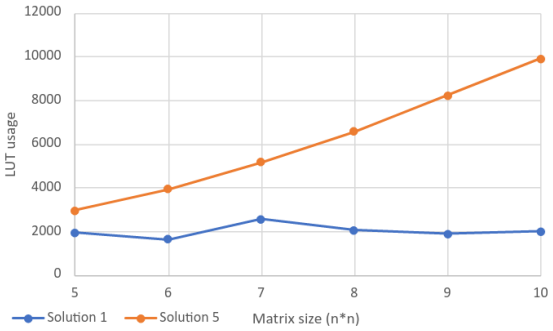

Figure 4: Look-Up Table usage.

Without optimisations, latency is much higher, and increases at a faster rate. The same is true of LUT usage, but for our optimised solution. This demonstrates a clear tradeoff between the two. Note that LUT usage appears to vary little around matrix size 8 - this is likely due to 8 being the native size for certain hardware components, and therefore certain optimisations are made relating to that. This is a hardware-specific optimmisation, so likely won't translate to outside this particular application and setup.

2.3 Lab 3

Note: Labs 1 and 2 were made with Vivado 2022.2, and lab 3 was created with Vivado 2019.1. Due to conflicts, the hardware designed in lab 2 is not the exact same as the hardware used in lab 3, which instead uses unrolling optimisations on the two inner loops. Therefore, statistics quoted are not suitable for cross-comparison between labs.

Lab 3 benchmarks the HLS designed hardware against a software implementation of matrix multiplication. The exported hardware is first loaded as IP in Vivado, and combined with the Zync processing system IP, such that the matrix multiplication IP can interrupt the processor. This block design is then re-exported for use in Vivado SDK (predecessor to Vitis IDE).

In SDK, we adapt the hello world template to use instantiate and drive our hardware and software implementations of matrix multiplication. We then introduce timing for both, and get the following results for multiplying two 5x5 matrices:

1. In software, it takes 7442 clock cycles (11.2us).
2. In hardware, it takes 5640 clock cycles (8.5us), a speed up of 1.3x.

We can also compare performance as we make our hardware implementation sequentially multiply many matrices. Whilst it takes 5640 cycles to process just one multiplication, we see that as we stack multiple multiplication in a row, we get a lower number of cycles per multiplication. This is because the hardware is pipelined, meaning that once we have filled the pipeline of multiplications to process, we get optimal throughput. The number of pipeline stages is relatively low, so over a larger scale we see a linear relationship.

Number of multiplications	Clock cycles
1	5640
3	9452
5	13518
10	24158
25	55576

,

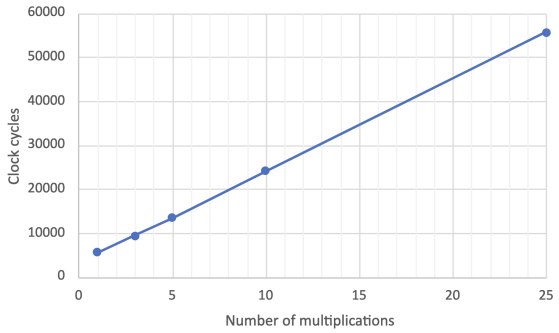

Figure 5: Number of multiplications and number of clock cycles to process.

3 Filter Design

For the project, we designed 3 filters.

1. Colour channel switch filter, implementing *single pixel operations*.
2. Facial anonymisation filter, using *pixel location*.
3. X and Y axis sobel filter, implementing *edge detection* with *spatial filtering*.

We also implemented some extra features, using Xilinx Petalinux.

1. Writing filtered images directly to the Linux framebuffer.
2. Streaming camera output to the internet with ffmpeg.

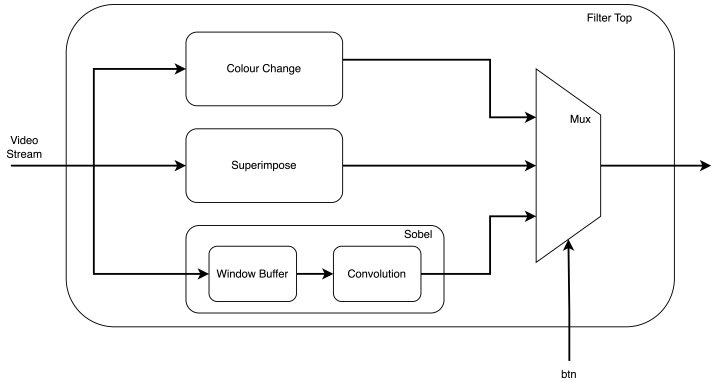

Figure 6: Top-down structure of the main filters.

3.1 Modules

3.1.1 Colour channel switch

This filter uses on-board switch input to control which colour channels are passed through, and which are removed from an image.

Figure 7: Demonstration of the colour channel switch filter.

Before being passed to the filter, each pixel is represented as a 24-bit value (8 bits per colour channel). In the filter, this is separated into each channel, and may be set to 0 if the corresponding switch is not active. The modified channels are then concatenated back together.

```
wire [7:0] red   = sw[0] ? i_vid_data[23:16]   : 0;
wire [7:0] blue  = sw[1] ? i_vid_data[15:8]    : 0;
wire [7:0] green = sw[2] ? i_vid_data[7:0]     : 0;
[...]
o_vid_data <= {red, blue, green};
```

This filter is a simple adaptation of the provided `colour_change` filter. With more granular control, this could easily be adapted to eg. fade in and out certain colour channels.

3.1.2 Facial anonymisation

This filter superimposes a *creative* comical image of a disguise over the camera output. To do this, it used *pixel location* to decide when to draw particular pixels as a point operation. It uses a *windowing function* to define a "draw area" on screen, and decide when to begin drawing the superimposed image.

The image to superimpose is stored in a block RAM, whose contents are loaded by the directive `$readmemh("glasses.mem", img)`. The file `glasses.mem` contains a list of 24-bit pixel values, and is interpreted correctly by specifying an `IMG_WIDTH` and `IMG_HEIGHT` in code.

The variables `x` and `y` are used to keep track of the location of the current pixel being considered, and are reset to 0 suitably on hsync and vsync stages. By keeping track of `x` and `y`, we can determine the logic for when to draw as follows:

```
wire write_img = (x >= IMG_POS_X) && (x < IMG_POS_X + IMG_WIDTH) && (y >= IMG_POS_Y)
&& (y < IMG_POS_Y + IMG_HEIGHT)
```

We index into the block RAM with `x` and `y` to get the pixel to next superimpose, and then draw it to our output with as follows:

```
o_vid_data <= (write_img && (img_px != 24'hffffff)) ? img_px: i_vid_data
```

Note that if the pixel to superimpose is `ffffff`, we do not draw it. This value is used as a transparent pixel, so that drawn images do not have to be perfectly rectangular.

Figure 8: A picture of me, fully anonymised by the filter.

This could be easily extended to interact with the pixels being drawn on, eg. drawing with lower opacity. It could also be expanded to include a transparency value as a 4th "colour" channel, allowing for variable opacity over an image.

3.1.3 Sobel filter

The sobel filter[5] is made of three components:

1. `window_buffer.v`, a FIFO-style buffer which stores 3 rows of image data, and outputs 9 pixels representing a square part of the stored image.
2. `sobel_convolve.v`, which accepts 9 pixels of data and outputs the result of applying an x and y 3-by-3 sobel kernel to the pixels.
3. `sobel.v`, which instansiates the two previous and feeds data from one to another.

As mentioned, the window buffer stores only 3 lines of data. This is the minimum required for the kernel filter - once the filter has moved across the image, it moves down a line, and the top row can be rewritten over to contain new data. As such, the buffer must also handle determining the correct order of output from the stored pixel lines.

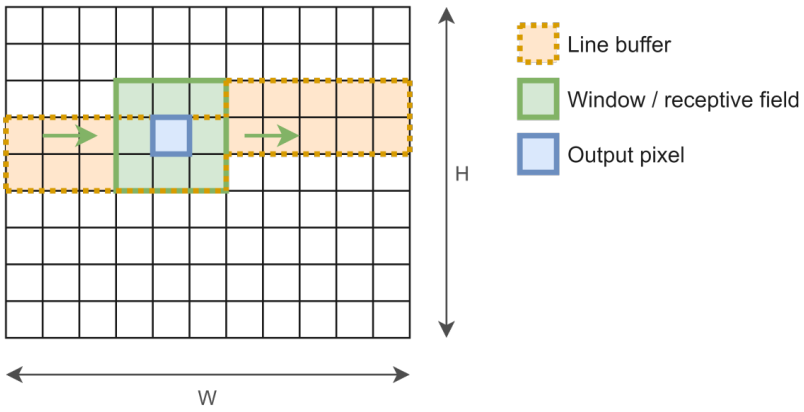

Figure 9: Management of stored data in the row buffer.

Three block RAMs are created to store each line. In practice, due to timing constraints, they are implemented in LUTs. `line_write_ptr` is used to select which index within a line incoming pixels should be written to, and `active_write_line` tracks which line buffer should be written to. Using these, we can track where to write incoming data, as well as which stored pixels should be output.

We maintain values `row_{0,1,2}_sel` to track which indexes on each line should be output. These indexes are the same for each line, since we wish to output a square section of the image. As each new pixel comes in, the pixels are shifted down the buffers - this ensures data is stored only as long as possible, and also helps to simplify the logic needed to select the pixels to output.

`sobel_convolve.v` implements the sobel filter kernels. This part is pipelined to improve throughput. Below are the sobel filter kernels used. Note that the central column and row respectively are 0 - we can use this to optimise our logic. Additionally, we only ever need to multiply by +- 1 or 2; multiples of 2 can be achieved with bit-shifts, which can be much quicker than other multiplications.

$$\mathbf{G}_x = \begin{bmatrix} +1 & 0 & -1 \\ +2 & 0 & -2 \\ +1 & 0 & -1 \end{bmatrix} * \mathbf{A} \quad \text{and} \quad \mathbf{G}_y = \begin{bmatrix} +1 & +2 & +1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix} * \mathbf{A}$$

Figure 10: Sobel X and Y filter kernels.

The first pipeline stage handles the summation of each row/column required (being the outer 2 for x and y). The middle term is shifted by 1 bit. After this pipeline, we have 4 numbers, each of which represents the weighted summation one row or column, in `gy0`, `gy2`, `gx0`, and `gx2`.

The second pipeline results in 2 values, one representing the final value of the y filter, and one for the x filter, in `gy` and `gx`. this is achieved by calculating `gy0-gy2` and `gx0-gx2` respectively - because column/row 2 is always negative, and 0 is positive, we can factor in the signs at this stage.

In our final pipeline stage, these two are summed together to yield our final bidirectional sobel filter output. This value is drawn to `pixel`.

`sobel.v` instansiates these two modules using input data from the camera and connects them together. Despite the name, it is very modular, and allows this filter to be easily extended to use any 3x3 kernel by switching out the `sobel_convolve` submodule.

3.2 Extra Features

We also use Petalinux[6] to *control* and *interact with filters through software*. Petalinux is a family of tools for building a distribution of Linux, created by Xilinx. It is based on the Yocto Linux operating system, and allows us to deploy custom hardware as part of an embedded Linux system.

The default Petalinux structure transforms the Pcam device into a standard Linux video device, using a combination of userspace drivers and Video For Linux 2 (v4l2) drivers. We are able to apply our filters to the video stream before it reaches Petalinux, enabling us to use our filtered camera stream just as a standard Linux video stream. This opens up some fun use-cases.

3.2.1 Writing to framebuffer

Linux gives ultimate flexibility, and lets us easily manipulate aspects of our system through well defined software interface devices. One way to output a video stream in Linux is to write directly to the framebuffer device. This is achieved by first configuring our camera device, and then using ffmpeg[7]:

```
v4l2-ctl --stream-mmap --set-fmt-video=width=$width,height=$height,pixelformat='YUYV'
--stream-to=- | \
ffmpeg -f rawvideo  -s 1280x720 -pix_fmt yuyv422 -framerate 15 -i pipe:0 \  # pipe to
ffmpeg, set input format
-vf "transpose=2,transpose=2,colorchannelmixer=1:0:0:0:0:0:1:0:0:1" \  # add filters
to rotate and correct colour format
-f fbdev -pix_fmt bgra /dev/fb0 # output to framebuffer
```

This first captures images from our (filtered) camera input using `v4l2-ctl`, and pipes it as raw video to ffmpeg. This then performs come colour correction, rotates the image (since writing to a framebuffer with the data we get uses the wrong endianness), converts the codec to `bgra` (which is understood by the framebuffer device), and then writes to the device `/dev/fb0`.

This uses our hardware-implemented filters for effects, and only runs the necessary elements in software. Due to the large amound of rescaling and codec swapping required, the framerate is noticably lower than when output directly to HDMI.

3.2.2 Streaming

The framebuffer is not the only device we can write to with ffmpeg. We can also stream to the internet, using a site such as YouTube.

```
v4l2-ctl --stream-mmap --set-fmt-video=width=$width,height=$height,pixelformat='YUYV'
--stream-to=- | \
ffmpeg -f rawvideo  -s 1280x720 -pix_fmt yuyv422 -framerate 15 -i pipe:0 \  # pipe to
ffmpeg, set input format
-re -f lavfi -i anullsrc -vf \  # restream option, add null audio source to make youtube
happy
-vf "transpose=2,transpose=2" \  # flip video, input is upside down for some reason
-f flv rtmp://a.rtmp.youtube.com/live2/$STREAM_KEY # output in flv format to RTMP url
with configured stream key
```

This command is similar to the last, but instead transcodes to the format desired by YouTube, flv. It also adds null audio to the video stream. This satisfies the requirements for streaming to YouTube. Surprisingly, the microprocessor is fast enough to keep up with the framerate requirements for streaming to be continuous.

3.3 Testing

To test our filters, we used `cocotb`[8], a framework for evaluating VHDL and SystemVerilog.

To use it, we first implement the desired functionality in Python. This is faster and much more flexible to implement than in HDL. We then use `cocotb` alongside a simulator such as `verilator` to simulate our HDL implementation, and compare values output by it to our desired values.

3.3.1 Superimposition

To verify the superimpose anonymisation feature, we used `PIL`, the Python Image Library, to convert an input image to binary data. We then pass it through our simulated hardware unit, capture the output, and save it as an image. This image can then be manually inspected to ensure it acts as intended.

```
data = image_to_data("img.jpeg")
...
# stream data through module
dut.i_vid_data.value = int(px)
await Timer(1, units="ns")
out_data[i, j] = dut.o_vid_data.value
...
print("filtering complete, saving image to out.jpeg")
data_to_image(out_data, "out.jpeg")
```

3.3.2 Sobel

The approach of direct parallel implementation was taken to verify sobel filtering elements function correctly. To test the sliding window aspects, a 5x5 buffer to use in verification are generated in Python.

```
image_padded = np.r_[
    np.zeros((1, 7)), np.c_[np.zeros(5), image, np.zeros(5)], np.zeros((1, 7))
```

```
]
windows = np.lib.stride_tricks.sliding_window_view(
    image_padded, window_shape=(3, 3)
).reshape(25, 3, 3)
```

Then, the buffer is past through the simulated hardware unit, and compared to the desired result as implemented in Python.

```
for buffer in generate_buffers(test_image):
  dut.buffer.value = buffer
  await Timer(4, units="ns")  # 3 clock cycles of delay
  out_px.append(dut.pixel.value)

output_image = np.fromiter((x.integer for x in out_px), dtype=np.uint8).reshape(
    5, 5
  )
```

This method of testing works very well to ensure exact correct outputs, and is applicable to a variety of scenarios. Tests were also created for alternative implementations and filters that didn't make it to the final product.

4 Conclusion

I have learned a lot from this project. Firstly from the labs - being introduced to new techniques to create complex systems in hardware is and has been extremely useful. Learning to tie together software and hardware elements is a stepping stone that has me convinced of the increasing practicality of hardware development. Evaluating the differences between different hardware implementations has also been insightful.

I have especially enjoyed the open-endedness and freedom of doing this project. There are many filters that didn't make it to the final product - from integrating an ML system to draw over a person to using audio to influence image filters. In the end, the ideas that made it through have been due to both practicality and personal interest. Especially using Petalinux in the system has been very interesting, as I have been interested in Linux development for a long time and have been very keen to get the opportunity to integrate custom hardware into an established system.

Some parts of the project have had a sharper learning curve, particularly when straying away from lab material. In the real world, using tools such as Petalinux is rarely straightforward, particularly for such a niche use-case. However, coming over challenges has been fulfilling and rewarding. I would like to see more encouragement to explore other systems and tools to be integrated into the system, particularly where they may not be straightforward to use. I would also like to see elements of formalised testing be encouraged more strongly in the labs.

Overall, it has been a fun experience and I look forward to putting my new skills to use!

5 Appendix

colour_sel.v

```
module colour_sel
(
    input  wire                 clk,
    input  wire                 n_rst,

    /*
     * Pixel inputs
     */
```

```
    input  wire [23:0] i_vid_data,
    input  wire                 i_vid_hsync,
    input  wire                 i_vid_vsync,
    input  wire                 i_vid_VDE,

    /*
     * Pixel output
     */
    output reg [23:0] o_vid_data,
    output reg                  o_vid_hsync,
    output reg                  o_vid_vsync,
    output reg                  o_vid_VDE,

    /*
     * Control
     */
    input wire [3:0]            sw
);

wire [7:0] red   = sw[0] ? i_vid_data[23:16]   : 0;
wire [7:0] blue  = sw[1] ? i_vid_data[15:8]  : 0;
wire [7:0] green = sw[2] ? i_vid_data[7:0] : 0;


always @ (posedge clk) begin
    if(!n_rst) begin
        o_vid_hsync <= 0;
        o_vid_vsync <= 0;
        o_vid_VDE <= 0;
        o_vid_data <= 0;
    end
    else begin
        o_vid_hsync <= i_vid_hsync;
        o_vid_vsync <= i_vid_vsync;
        o_vid_VDE <= i_vid_VDE;
        o_vid_data <= {red, blue, green};
    end
end

endmodule
```

filter_top.v

```
module filter_top(
    input  wire                  clk,
    input  wire                  n_rst,

    //button inputs for filter selection
    input [3:0] btn,

    //switches for single pixel colour selection filter
    input [3:0] sw,

    /*
     * Pixel inputs
     */
    input  wire [23:0] i_vid_data,
```

```
    input  wire                  i_vid_hsync,
    input  wire                  i_vid_vsync,
    input  wire                  i_vid_VDE,

    /*
     * Pixel output
     */
    output reg [23:0] o_vid_data,
    output reg                   o_vid_hsync,
    output reg                   o_vid_vsync,
    output reg                   o_vid_VDE

);

    //define various filters here
    localparam WIDTH = 1920;
    localparam HEIGHT = 1080;

    //sobel filter
    wire [23:0] sobel_data;
    wire        sobel_hsync;
    wire        sobel_vsync;
    wire        sobel_VDE;

    sobel #(.WIDTH(WIDTH))  f0 (
        .clk(clk),
        .n_rst(n_rst),
        .i_vid_data(i_vid_data),
        .i_vid_hsync(i_vid_hsync),
        .i_vid_vsync(i_vid_vsync),
        .i_vid_VDE(i_vid_VDE),
        .o_vid_data(sobel_data),
        .o_vid_hsync(sobel_hsync),
        .o_vid_vsync(sobel_vsync),
        .o_vid_VDE(sobel_VDE)
    );
    //superimpose filter
    wire [23:0] superimpose_data;
    wire        superimpose_hsync;
    wire        superimpose_vsync;
    wire        superimpose_VDE;

    superimpose #(.VID_WIDTH(WIDTH), .VID_HEIGHT(HEIGHT)) f1 (
        .clk(clk),
        .n_rst(n_rst),
        .i_vid_data(i_vid_data),
        .i_vid_hsync(i_vid_hsync),
        .i_vid_vsync(i_vid_vsync),
        .i_vid_VDE(i_vid_VDE),
        .o_vid_data(superimpose_data),
        .o_vid_hsync(superimpose_hsync),
        .o_vid_vsync(superimpose_vsync),
        .o_vid_VDE(superimpose_VDE)
    );

    //colour selection filter
```

```
wire [23:0] colour_sel_data;
wire        colour_sel_hsync;
wire        colour_sel_vsync;
wire        colour_sel_VDE;

colour_sel f2 (
    .clk(clk),
    .n_rst(n_rst),
    .i_vid_data(i_vid_data),
    .i_vid_hsync(i_vid_hsync),
    .i_vid_vsync(i_vid_vsync),
    .i_vid_VDE(i_vid_VDE),
    .o_vid_data(colour_sel_data),
    .o_vid_hsync(colour_sel_hsync),
    .o_vid_vsync(colour_sel_vsync),
    .o_vid_VDE(colour_sel_VDE),
    .sw(sw)
);


reg [1:0] select_btn;
always @(posedge clk) begin
    if (!n_rst) select_btn <= 2'b00;
    else begin
        case(btn)
            4'b0001: select_btn <= 2'd0;
            4'b0010: select_btn <= 2'd3;
            4'b0100: select_btn <= 2'd2;
            4'b1000: select_btn <= 2'd1;
            default: select_btn <= select_btn;
        endcase
    end
end

reg [23:0] data_sel;
reg hsync_sel;
reg vsync_sel;
reg VDE_sel;

//data passed into all filters
//mux the outputs based on button selection
always@(posedge clk) begin
    case (select_btn)
        2'd1: begin
            data_sel <= sobel_data;
            hsync_sel <= sobel_hsync;
            vsync_sel <= sobel_vsync;
            VDE_sel <= sobel_VDE;
        end
        2'd2: begin
            data_sel <= superimpose_data;
            hsync_sel <= superimpose_hsync;
            vsync_sel <= superimpose_vsync;
            VDE_sel <= superimpose_VDE;
        end
        2'd3: begin
```

```
                data_sel <= colour_sel_data;
                hsync_sel <= colour_sel_hsync;
                vsync_sel <= colour_sel_vsync;
                VDE_sel <= colour_sel_VDE;
            end
            default: begin
                data_sel <= i_vid_data;
                hsync_sel <= i_vid_hsync;
                vsync_sel <= i_vid_vsync;
                VDE_sel <= i_vid_VDE;
            end
        endcase
    end

    //assign outputs to mux outputs
    always @ (posedge clk) begin
        if(!n_rst) begin
            o_vid_hsync <= 0;
            o_vid_vsync <= 0;
            o_vid_VDE <= 0;
            o_vid_data <= 0;
        end
        else begin
            o_vid_hsync <= hsync_sel;
            o_vid_vsync <= vsync_sel;
            o_vid_VDE <= VDE_sel;
            o_vid_data <= data_sel;
        end
end

endmodule
```

sobel.v

```
module sobel # (parameter WIDTH=1280) (
    input  wire clk,
    input  wire n_rst,

    input  wire [23:0] i_vid_data,
    input  wire i_vid_hsync,
    input  wire i_vid_vsync,
    input  wire i_vid_VDE,

    output wire [23:0] o_vid_data,
    output wire o_vid_hsync,
    output wire o_vid_vsync,
    output wire o_vid_VDE
);

    wire [7:0] o_fifo_0;
    wire [7:0] o_fifo_1;
    wire [7:0] o_fifo_2;
    wire [7:0] o_fifo_3;
    wire [7:0] o_fifo_4;
    wire [7:0] o_fifo_5;
    wire [7:0] o_fifo_6;
    wire [7:0] o_fifo_7;
```

```
    wire [7:0] o_fifo_8;

    wire [71:0] i_sobel_bfr;
    assign i_sobel_bfr = {  o_fifo_0, o_fifo_1, o_fifo_2,
                            o_fifo_3, o_fifo_4, o_fifo_5,
                            o_fifo_6, o_fifo_7, o_fifo_8  };
    wire [7:0] o_sobel_pxl;

   wire [7:0] input_data = (i_vid_data[7:0] + i_vid_data[15:8] + i_vid_data[23:16])/3;
    window_buffer #(.WIDTH(WIDTH)) wb (
        .clk(clk),
        .n_rst(n_rst),
        .px(input_data),
        .i_vid_hsync(i_vid_hsync),
        .i_vid_vsync(i_vid_vsync),
        .i_vid_VDE(i_vid_VDE),
        .o_vid_data_0(o_fifo_0),
        .o_vid_data_1(o_fifo_1),
        .o_vid_data_2(o_fifo_2),
        .o_vid_data_3(o_fifo_3),
        .o_vid_data_4(o_fifo_4),
        .o_vid_data_5(o_fifo_5),
        .o_vid_data_6(o_fifo_6),
        .o_vid_data_7(o_fifo_7),
        .o_vid_data_8(o_fifo_8)
    );

    sobel_convolve conv (
        .clk(clk),
        .n_rst(n_rst),
        .buffer(i_sobel_bfr),
        .pixel(o_sobel_pxl)
    );

    assign o_vid_data = {o_sobel_pxl, o_sobel_pxl, o_sobel_pxl};
    assign o_vid_hsync = i_vid_hsync;
    assign o_vid_vsync = i_vid_vsync;
    assign o_vid_VDE = i_vid_VDE;

endmodule
```

sobel_convolve.v

```
module sobel_convolve #(parameter PX_SIZE=8) (
    input clk, n_rst,
    input wire [(PX_SIZE*9)-1:0] buffer,
    output wire [PX_SIZE-1:0] pixel
);

    wire [PX_SIZE-1:0] p00, p01, p02, p10, p11, p12, p20, p21, p22;
    assign {p00, p01, p02, p10, p11, p12, p20, p21, p22} = buffer;

    //first pipeline stage, do one row/column
    //add 2 extra bits to capture possible overflows
    reg [PX_SIZE+1:0] gx0, gx2, gy0, gy2;
    /* verilator lint_off WIDTH */
    always @(posedge clk) begin
```

```
        if (!n_rst) begin
            gx1 <= 0;
            gx2 <= 0;
            gy1 <= 0;
            gy2 <= 0;
        end else begin
            gy0 <= p00 + (p01 << 1) + p02; // top row of y matrix
            gy2 <= p20 + (p21 << 1) + p22; // bottom row of y matrix
            gx0 <= p00 + (p10 << 1) + p20; // first column of x matrix
            gx2 <= p02 + (p12 << 1) + p22; // last column of x matrix
        end
    end

    //2nd pipeline stage, do x and y sum
    //don't need extra bit, no underflow because always doing bigger - smaller
    reg [PX_SIZE+1:0] gx, gy;
    always @(posedge clk) begin
        if(!n_rst) begin
            gx <= 0;
            gy <= 0;
        end else begin
            // keep it positive, we take the magnitude anyway
            gx <= gx0 < gx2 ? gx2 - gx0 : gx0 - gx2;
            gy <= gy0 < gy2 ? gy2 - gy0 : gy0 - gy2;
        end
    end


    //3rd stage, do final sum and write to output
    reg [PX_SIZE+2:0] g;
    always @(posedge clk) begin
        if(!n_rst) begin
            g <= 0;
        end else begin
            //pixel <= (sqrt(gx * gx, gy * gy));
            //alternatively, if you don't want to normalise it
            //we have kept positive ints so don't need abs
            g <= gx + gy;
        end
    end

    //combinational always, mux the output based on if overflow or not
    //if overflow, saturate at max value, else use just lower bits
    assign pixel = g[PX_SIZE+2] ? {PX_SIZE{1'b1}}: g[PX_SIZE+1:2];

endmodule
```

superimpose.v

```
module superimpose #(parameter VID_WIDTH=1280 , parameter VID_HEIGHT=720)
(
    input  wire                  clk,
    input  wire                  n_rst,

    /*
     * Pixel inputs
     */
```

```
    input  wire [23:0] i_vid_data,
    input  wire                 i_vid_hsync,
    input  wire                 i_vid_vsync,
    input  wire                 i_vid_VDE,

    /*
     * Pixel output
     */
    output reg [23:0] o_vid_data,
    output reg                  o_vid_hsync,
    output reg                  o_vid_vsync,
    output reg                  o_vid_VDE
);

localparam IMG_WIDTH = 200;
localparam IMG_HEIGHT = 150;
localparam IMG_POS_X = 200;
localparam IMG_POS_Y = 150;

(*ram_style = "block" *) reg [23:0] img [0:IMG_WIDTH*IMG_HEIGHT];

initial begin
    $readmemh("glasses.mem", img);
end

reg [$clog2(VID_WIDTH)-1:0] x;
reg [$clog2(VID_HEIGHT)-1:0] y;

always @ (posedge clk) begin
    if(!n_rst || i_vid_vsync) begin
        x <= 0;
        y <= 0;
    end else if (x == VID_WIDTH) begin
        x <= 0;
        y <= y + 1;
    end else if (i_vid_VDE) begin
        x <= x + 1;
    end
end

wire write_img = (x >= IMG_POS_X) && (x < IMG_POS_X + IMG_WIDTH) && (y >= IMG_POS_Y)
&& (y < IMG_POS_Y + IMG_HEIGHT);

wire [$clog2(VID_WIDTH)-1:0] img_x = x - IMG_POS_X;
wire [$clog2(VID_WIDTH)-1:0] img_y = y - IMG_POS_Y;
wire [23:0] img_px = img[img_x + img_y * IMG_WIDTH];

always @ (posedge clk) begin
    if(!n_rst) begin
        o_vid_hsync <= 0;
        o_vid_vsync <= 0;
        o_vid_VDE <= 0;
        o_vid_data <= 0;
    end
    else begin
        o_vid_hsync <= i_vid_hsync;
```

```
        o_vid_vsync <= i_vid_vsync;
        o_vid_VDE <= i_vid_VDE;
        //write img px if we are in the image area and the pixel is not white (white
is transparent)
        o_vid_data <= (write_img && (img_px != 24'hffffff)) ? img_px: i_vid_data;
    end
end

endmodule
```

window_buffer.v

```
module window_buffer #(parameter WIDTH = 1280)
(
    input wire clk,
    input wire n_rst,

    // Pixel input
    input  wire [7:0] px,
    input  wire                 i_vid_hsync,
    input  wire                 i_vid_vsync,
    input  wire i_vid_VDE,

    // Pixels output
    /* 0 1 2
       3 4 5
       6 7 8 */
    output wire [7:0] o_vid_data_0,
    output wire [7:0] o_vid_data_1,
    output wire [7:0] o_vid_data_2,
    output wire [7:0] o_vid_data_3,
    output wire [7:0] o_vid_data_4,
    output wire [7:0] o_vid_data_5,
    output wire [7:0] o_vid_data_6,
    output wire [7:0] o_vid_data_7,
    output wire [7:0] o_vid_data_8
);

(*ram_style = "block" *) reg [7:0] line_0 [WIDTH-1:0];
(*ram_style = "block" *) reg [7:0] line_1 [WIDTH-1:0];
(*ram_style = "block" *) reg [7:0] line_2 [WIDTH-1:0];


reg [1:0] active_write_line;
reg [$clog2(WIDTH)-1:0] line_write_ptr;

//read into line buffers
always @ (posedge clk) begin
    if(i_vid_vsync || !n_rst) begin
        //on new frame, go back to line 0, px 0
        active_write_line <= 0;
        line_write_ptr <= 0;
    end else if(line_write_ptr == WIDTH) begin
        //when we get to end of line, reset
        line_write_ptr <= 0;
        //increment active line, on line 2 go back to 0
        active_write_line <= active_write_line == 2'd2 ? 0 : active_write_line + 2'd1;
```

```
    end else if(i_vid_VDE) begin
        //only increment if enabled
        line_write_ptr <= line_write_ptr + 1;
    end
    case (active_write_line)
            2'd0: line_0[line_write_ptr] <= px;
            2'd1: line_1[line_write_ptr] <= px;
            2'd2: line_2[line_write_ptr] <= px;
            default: line_0[line_write_ptr] <= px; //theoretically unreachable
    endcase
end

//window buffer
//three rows of 3 pixels
reg [7:0] row_0 [2:0];
reg [7:0] row_1 [2:0];
reg [7:0] row_2 [2:0];

//which line does each row read from
//0 indicates reading all zeroes, else 1 2 or 3
reg [1:0] row_0_sel;
reg [1:0] row_1_sel;
reg [1:0] row_2_sel;

always@(posedge clk) begin
    if(i_vid_vsync | !n_rst) begin
        row_0_sel <= 2'd3;
        row_1_sel <= 2'd2;
        row_2_sel <= 2'd1;
    end
    else if(i_vid_hsync) begin
        //pointers all increase (move up a row) and wrap to 1 on a 3
        row_0_sel <= row_0_sel == 2'd1 ? 2'd3 : row_0_sel - 2'd1;
        row_1_sel <= row_1_sel == 2'd1 ? 2'd3 : row_1_sel - 2'd1;
        row_2_sel <= row_2_sel == 2'd1 ? 2'd3 : row_2_sel - 2'd1;
    end
end

//choose from the 4 possible inputs (zeroes, or 1 of the 3 lines)
wire  [7:0]  row_0_in  =  row_0_sel  ==  2'd0  ?  0  :  (row_0_sel  ==  2'd1  ?
line_0[line_write_ptr-1]  :  (row_0_sel  ==  2'd2  ?  line_1[line_write_ptr-1]  :
line_2[line_write_ptr-1]));
wire  [7:0]  row_1_in  =  row_1_sel  ==  2'd0  ?  0  :  (row_1_sel  ==  2'd1  ?
line_0[line_write_ptr-1]  :  (row_1_sel  ==  2'd2  ?  line_1[line_write_ptr-1]  :
line_2[line_write_ptr-1]));
wire  [7:0]  row_2_in  =  row_2_sel  ==  2'd0  ?  0  :  (row_2_sel  ==  2'd1  ?
line_0[line_write_ptr-1]  :  (row_2_sel  ==  2'd2  ?  line_1[line_write_ptr-1]  :
line_2[line_write_ptr-1]));

always @(posedge clk) begin
    //shift each of the pixels along the rows
    row_0[2] <= row_0_in;
    row_0[1] <= row_0[2];
    row_0[0] <= row_0[1];

    row_1[2] <= row_1_in;
```

```
    row_1[1] <= row_1[2];
    row_1[0] <= row_1[1];

    row_2[2] <= row_2_in;
    row_2[1] <= row_2[2];
    row_2[0] <= row_2[1];
end

    assign  o_vid_data_0 = row_0[0];
    assign  o_vid_data_1 = row_0[1];
    assign  o_vid_data_2 = row_0[2];
    assign  o_vid_data_3 = row_1[0];
    assign  o_vid_data_4 = row_1[1];
    assign  o_vid_data_5 = row_1[2];
    assign  o_vid_data_6 = row_2[0];
    assign  o_vid_data_7 = row_2[1];
    assign  o_vid_data_8 = row_2[2];

endmodule
```

Bibliography

[1] L. H. Crockett, R. A. Elliot, M. A. Enderwitz, and R. W. Stewart, *The Zynq Book: Embedded Processing With the ARM Cortex-A9 on the Xilinx Zynq-7000 All Programmable Soc*, Strathclyde Academic Media, 2014.

[2] P. Coussy, D. D. Gajski, M. Meredith, and A. Takach, "An introduction to high-level synthesis," *IEEE Des. & Test Comput.*, vol. 26, no. 4, pp. 8–17, 2009, doi: 10.1109/MDT.2009.69.

[3] Xilinx, "Zynq-7000 processing system ip." (Online; https://www.xilinx.com/products/intellectual-property/processing_system7.html)

[4] Xilinx, "Hls pragmas." (Online; https://docs.xilinx.com/r/en-US/ug1399-vitis-hls/HLS-Pragmas)

[5] B. P. Lathi, and R. A. Green, *Essentials of Digital Signal Processing*, Cambridge University Press, 2014.

[6] Xilinx, "Petalinux tools." (Online; https://www.xilinx.com/products/design-tools/embedded-software/petalinux-sdk.html)

[7] FFmpeg, "Ffmpeg documentation." (Online; https://ffmpeg.org/ffmpeg.html)

[8] cocotb, "Cocotb documentation." (Online; https://docs.cocotb.org/en/stable/)