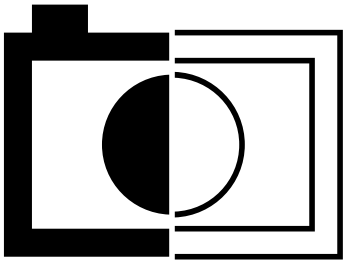
NTNU

TDT4295 - Computer Design Project



Camvolution

Peter Aaser
Mattis Spieler Asp
Truls Fossum
Fredrik Berdon Haave
Øyvind Kjerland
Arnstein Kleven
Mathias Ose

November 28, 2015

Abstract

This report presents the results of the autumn 2015 TDT4295 course at NTNU for the *Camvolution* group.

Camvolution is a computer designed and built from scratch for the purpose of performing 2D convolution on a video stream. It receives an external video input stream through an HDMI port, performs convolution on the frames of the stream, then outputs the manipulated stream to another HDMI port.

Acknowledgements

We wish to thank the following for their assistance:

Gunnar Tufte for guidance, encouragement and endless enthusiasm.

Yaman Umuroğlu for giving us valuable help and assistance, and also for introducing us to Chisel.

Odd Rune Lykkebø for advice and help with soldering.

Erlend Hestnes for advice on PCB layout.

Omega Verksted for supplying components and junk food at any time of day and night.

Hackerspace NTNU for letting us borrow equipment.

V.E.C.T.O.R. -3000 for cameradery, assistance and feedback.

Contents

List of Figures

List of Tables

1 Introduction

This report present the results from the *TDT4295 Computer Design Project* course at *The Norwegian University of Science and Technology* for the *Camvolution* group. In this course groups of students design and create their own computers from scratch.

The task for the Camvolution group is to create a computer optimised for performing convolution on two-dimensional data. The processor should be able to take images and convolution kernels as input, perform the convolution, and output the convolved images.

The processor architecture should be implemented on a Xilinx FPGA, and a Silicon Labs EFM32 series microcontroller should act as an I/O processor. The output from the processor should be displayed for demonstration purposes. This project has a budget of 10.000 NOK.

1.1 Convolution

(a) Original image (b) After sharpening

Figure 1.1: Sharpening performed with convolution on *Lena* test image

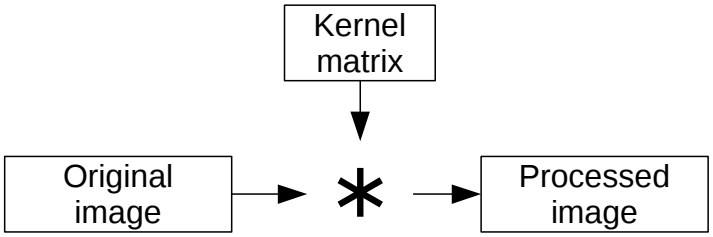


Figure 1.2: Basic convolution in image processing

Convolution is a mathematical operation that can be applied to images or other signals in one or multiple dimensions. This is commonly used by image processing software to perform blurring, edge detection, sharpening and more.

An example of the result produced by a sharpening filter implemented with convolution can be seen in Figure 1.1. The convolution itself requires two inputs, the original image and a *convolution kernel matrix*. In the case of image processing, the kernel defines what kind of filter is applied. The inputs and outputs of convolution when used for image processing is illustrated in Figure 1.2.

1.2 Camvolution

The given task [1] is very open-ended and can be solved in multiple ways. This section presents the groups' interpretation and the requirements for the system.

1.2.1 Area of Focus

As the task places no restrictions on what should be the area of focus on as long as the computer performs convolution, several possibilities were considered, some inspired by the previous project reports:

Energy Efficiency Do convolution in an energy efficient manner.

Generality Make it possible to use the system for a wide variety of tasks.

Performance Maximise the throughput of the system.

After some deliberation, the group decided to focus on performance. We decided to perform convolution on a video stream and output this manipulated video. This video stream should come from a video camera attached to the computer.

1.2.2 Requirements

A set of goals or requirements is useful to guide the efforts in the direction of the assignment, and to ensure a demonstration can be held at the end of the project.

Functional Requirements

The functional requirements set for this project is listed in Table 1.1 and more thorough descriptions follow.

Name	Description	Priority
FR1	The processor should do convolution on a two-dimensional image with at least a 3x3 convolution kernel	High
FR2	The system should be able to display output on an external screen	High
FR3	The system should be able to use a camera as input	Medium
FR4	The system should boot and operate without an external computer	Medium

Table 1.1: Functional Requirements

FR1 The processor should be able to do convolution. As the assignment text states, this should be done on a two-dimensional structure, for which an image was chosen. Since image convolution is meaningless with a convolution kernel smaller than 3x3, this is also included in the requirements.

FR2 It follows directly from the assignment text that the system must be able to display output from an application that produces graphical data. To accomplish this, the computer is required to have a display port that can output graphics.

FR3 As it was decided to focus on performance, one of the goals set was to be able to do convolution on a live video stream. This requires an input stream which it was decided should come from a video camera connected to the computer.

FR4 To make the demonstration a bit smoother and ensure the computer can accomplish the goals independently, it was also also required that the system should be able to boot and operate without an external computer.

Non-Functional Requirements

In addition to the functional requirements, the task at hand also enforces requirements not related to the functionality of the computer produced. These are listed in Table 1.2.

Name	Description
NFR1	The processor should be implemented on a Xilinx FPGA
NFR2	The computer should use a Silicon Labs EFM32 microcontroller as an I/O processor
NFR3	The cost of developing the system should not exceed a budget of 10.000 NOK

Table 1.2: Non-Functional Requirements

2 Background

2.1 Convolution

This section introduces convolution from a more mathematical perspective and then gives some less traditional applications of a more generalised version.

2.1.1 Definition

Conceptually, the convolution operation gives the overlap between two functions as a function of the offset between the functions.

Mathematically, convolution is defined by an integral [14]:

$$(f * g)(t) = \int_{-\infty}^{\infty} f(\tau)g(t - \tau)d\tau$$

The discrete counterpart to this is [2]:

$$(f * g)(n) = \sum_{i=-\infty}^{\infty} f(i)g(n - i)$$

This discrete variant is visualised in Figure 2.1 where only three of the values in the input functions have been highlighted. Notice how the corresponding values from the input functions are multiplied before the products are summed.

Convolution can also be extended to multiple dimensions by summing across the additional dimensions. For the discrete case, a two-dimensional convolution is expressed as [2]:

$$(f * g)(n, m) = \sum_{i=-\infty}^{\infty} \sum_{j=-\infty}^{\infty} f(i, j)g(n - i, m - j)$$

2.1.2 Application in Image Processing

As mentioned in the introduction, convolution can be used in image processing to apply various filters to an image.

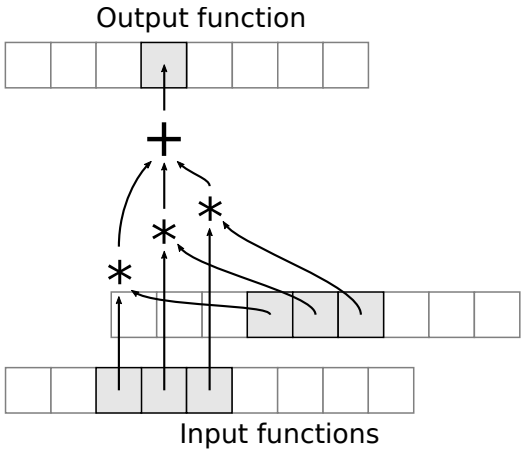


Figure 2.1: Convolution performed on discrete functions. Each box represent a discrete value.

A monochrome image can be thought of as a 2D matrix with a width W and height H. The operation to be performed on the image matrix can be expressed as another matrix called the *kernel*. Let $I(x, y)$ be a function for accessing a specific value in the image matrix. Let $K(x, y)$ be a function for accessing a specific value in the kernel, or 0 if x or y is out of bounds for the kernel matrix.

The convolution of the image and the kernel is then given by

$$C(x, y) = \sum_{h=0}^{H} \sum_{w=0}^{W} I(w, h)K(w - x, h - y)$$

where $C(x, y)$ is the output image. Because $K(w - x, h - y)$ is 0 for most w, h, the resulting values are determined by only a small *neighbourhood* part of the input matrix around x, y.

For a coloured image, convolution can be performed once per colour channel. The convolved channels can then be put together again to produce a convolved coloured image.

Depending on the kernel matrix, various effects like blurring or sharpening may be applied to the image with convolution. A small selection of possible kernel matrices and the results when applied to the image in Figure 1.1a can be found in Table 2.1.

As the matrices in Table 2.1 are good examples of, the kernel matrices used for image processing are usually small compared to the images, and often of a constant size. This means each output pixel is a function of the same pixel in the original image and its neighbourhood.

2.1.3 Generalised Convolution

The convolution described above can be separated into two stages: first multiplication, then a summation. The multiplication stage simply maps a set of input values to a set of output values by using the kernel matrix values as factors. These products are then reduced into a single value by summation.

By recognising that the process consists of a map and a reduce operation, it becomes

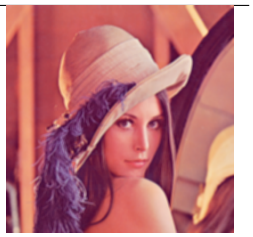
Description	Matrix	Result
Blur	$\begin{bmatrix} 1/9 & 1/9 & 1/9 \\ 1/9 & 1/9 & 1/9 \\ 1/9 & 1/9 & 1/9 \end{bmatrix}$	
Edge-detect	$\begin{bmatrix} -1 & -1 & -1 \\ -1 & 8 & -1 \\ -1 & -1 & -1 \end{bmatrix}$	
Emboss	$\begin{bmatrix} -2 & -1 & 0 \\ -1 & 1 & 1 \\ 0 & 1 & 2 \end{bmatrix}$	

Table 2.1: Some kernels used in image processing

possible to substitute the operations with other map and reduce operations. A generalised convolution operation is therefore fully defined by two input matrices and two operations.

In the realm of image processing, this paves the way for new filters, such as the median, minimum and maximum filters. These filters are commonly used to remove noise in images and consists of a convolution kernel matrix of ones (the multiplicative identity for scalars), the multiplication operation for the map stage and the minimum, maximum or mean operations respectively for the reduce operations. The resulting images created by applying the effect to the *Lena* test image can be seen in Table 2.2.

Generalised convolution can also be used in other areas, such as to calculate the next iteration of cellular automatons. This requires a more advanced reduce operation.

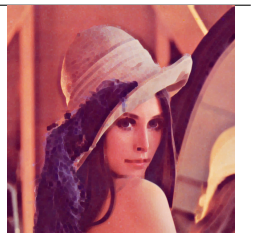
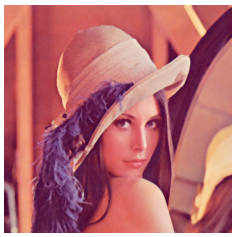
Reduce operation	Result
Minimum	
Maximum	
Median	

Table 2.2: Some image processing filters using a generalised form of convolution

2.2 Digital Video

Video can be encoded in a variety of formats, each with its advantages and drawbacks. In this project, a balance between quality, size and performance must be found, such that the required throughput can be minimised without sacrificing too much image quality.

A common encoding for images is to encode the pixels in a two-dimensional array in a row-major order, in other words as an array of scanlines.

2.2.1 Resolution

The number of pixels in each scanline and the number of scanlines, more commonly referred to as the width and height of the image, determines the size of each pixel on the screen.

Too few pixels results in a "pixelated" image where the pixels are distinct. Too many pixels gives a huge performance penalty as higher resolutions require more storage space and throughput.

2.2.2 Colours

Each element stored must provide the colour of the pixel it represents.

Separate Components

One way of achieving this goal is to separate the colour into three components and store each individually. A common choice of components is red, green and blue, giving rise to the RGB colour model.

Furthermore, it is possible to store each component as a separate pixel or to interlace the different components in the same image so that all the information required to determine the colour of a pixel resides in one place.

As data stored by computers is discrete and of limited size, the number of values available when describing the amount of each component present in a given colour is limited. The number of values that can be represented is referred to as the *depth* of the image, commonly given in the number of bits used to represent a single value.

Indexed

Another way to store the colour information is to select a set of colours and assign each colour a number. This is especially useful when reducing the size of an image by reducing the number of available colours. With an index colour representation, the most commonly used colours can be selected and assigned numbers, thus reducing the error introduced by forcing the colours into a smaller space.

2.2.3 Bit Rates

An important measure in determining the efficiency of a given video format is the bit rate, the rate at which bits needs to be provided in order to display the video. For uncompressed video, knowledge of the frame rate in addition to the image resolution and colour representation is necessary to be able to determine the bit rate of the video.

Table 2.3 displays the bit rate required at some common resolutions, depths/index sizes and frame rates.

Resolution	Depth/index size	Bit rate @ 15Hz	Bit rate @ 30Hz	Bit rate @ 60Hz
320x240	8 bit	9,2 Mbps	18,4 Mbps	36,9 Mbps
320x240	16 bit	18,4 Mbps	36,9 Mbps	73,7 Mbps
320x240	24 bit	27,6 Mbps	55,3 Mbps	110,6 Mbps
640x480	8 bit	36,9 Mbps	73,7 Mbps	147,5 Mbps
640x480	16 bit	73,7 Mbps	147,5 Mbps	294,9 Mbps
640x480	24 bit	110,6 Mbps	221,2 Mbps	442,4 Mbps

Table 2.3: Bit rates given different video parameters

2.3 High-Definition Multimedia Interface (HDMI)

HDMI is a standard for transferring digital video between devices and is commonly used to connect video sources to displays. The cable can carry audio, Ethernet and other information in addition to the video stream itself, but only the video stream was utilised for this project.

2.3.1 Transition-Minimised Differential Signaling

Transition-Minimised Differential Signaling (TMDS) is a standard used for transmitting video data over HDMI. HDMI uses it at the physical layer. Four channels of serial data establish the HDMI video transmission. It has three channels for the RGB colour information and one for the pixel data rate clock. Each pixel has a 24-bit colour depth, with each colour being 8 bits each. These are separately convertered into 10-bit symbols before being serialised and transmitted onto the TMDS data channels. It is this 10:1 serialisation ratio that makes the bit rate 10x faster than the actual pixel rate. During the video transmission the pixel symbol is periodically interlaced with four distinct control tokens representing blanking intervals. These control tokens provide accurate video line scan (HSYNC) and frame update (VSYNC) information. Control tokens are also used to identify word boundaries for synchronisation purposes.

The technology is based on twisted pairs of cables transferring data using a differential encoding. This encoding uses two cables for each bit and inverts the voltage difference between the cables in a pair whenever the bit represented by the pair should change. Electromagnetic interference tends to affect both conductors identically, and the receiving circuit will only detect the difference beetwen the wires (see Figure 2.2). The advantage of this this technique is that it resists electromagnetic noise better than single-ended conductors [4].

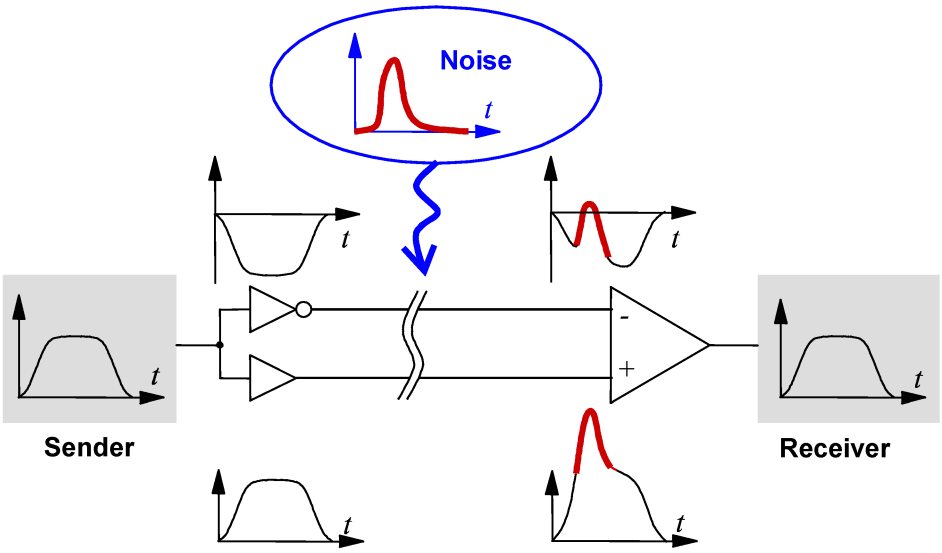


Figure 2.2: Resistance of electromagnetic noise by using differential signaling [9].

TMDS Receiver

The TMDS receiver needs to do clock and data recovery (CDR) to convert the serial data stream back into the 10-bit words. This is done by using the incoming pixel clock to recover the bit sampling clock and then applying the bit clock to recover the serial data stream. Then skews among the three data channels are removed by using a channel deskew circuit. Then the 10-bit word is decoded into one of three possible formats:

- 8-bit video pixel data through the DVI or HDMI decoder.
- 4-bit auxiliary data, i.e. information and audio frames through the HDMI decoder only.
- 2-bit control data, e.g. the HSYNC and VSYNC through the DVI or HDMI decoder.

TMDS Transmitter

Each colour, VSYNC, HSYNC and VDE is sent through a TMDS encoder, which turns each colour into a 10-bit word respectively. This 10-bit word is then made into a serial data stream before converting it to TMDS signals to be transmitted on the HDMI cable.

2.4 Field-Programmable Gate Array (FPGA)

Speeding up computations can be done in several ways, but a common approach for operations performed often is to implement the operation in hardware. A good example is multiplication in modern processors. This operation is not strictly necessary as the same behaviour can be simulated by repeatedly summing values in a loop, but it is still commonly available in hardware because this makes the implementation more efficient. The drawback of hardware acceleration is that the hardware resources are used to improve a special case rather than a general case.

An FPGA is an integrated circuit which is designed to be configured after it has been manufactured, as opposed to an ASIC (Application-Specific Integrated Circuit). It consists of reconfigurable logic blocks and configurable connections between them, allowing the behaviour of the FPGA to be specified at run time. Usually this is done by describing the behaviour using a hardware description language, which is then used to create a configuration file for the FPGA.

FPGAs are a good compromise between software and ASICs; the FPGA implementation runs considerably faster than software for many operations, and as the configuration can be changed at any time, the resources can be used for many different purposes without doing changes to the actual hardware. For example, the very same FPGA can be used to decode video in one moment, and to perform convolution in the blink of an eye in the next.

This approach has in fact already been implemented by Intel on some of their Xeon processors [5], and is expected to launch in early 2016 [11].

2.4.1 Chisel

Another drawback of hardware implementation is the time required for development [7]. Even though the hardware can be reconfigured, the configurations still need to be coded using techniques similar to the ones used for developing ASICs. One attempt to reduce the development time is the hardware description language Chisel, which aims to be a flexible alternative to VHDL and Verilog [6], and thereby reduce development time. Chisel creates a graph representation of the design, which can then be transpiled to Verilog for synthesis.

3 | Description & Methodology

3.1 System Overview

Definitions

Processor
The processor implementation on the FPGA.

Board
The printed circuit board (PCB).

Computer
The board with soldered components.

System
The computer with peripherals, power supply, etc.

MCU
The I/O controller.

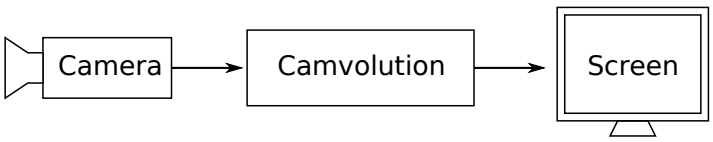


Figure 3.1: Overview of the data flow

As stated in the introduction, the system is to perform convolution on a live video stream. An overview of the data flow can be seen in Figure 3.1. The video stream flows into the system from an external source, preferably a camera as depicted in the figure. The system then performs the convolution and passes the image on to an external display, in this case a screen. As stated in the requirements, the system should handle streaming video from a camera, perform convolution and be able to provide the result as an output stream.

3.1.1 Implementation

The convolution is to be done on the FPGA using a custom processor implementation described in Section 3.6. First of all, this satisfies non-functional requirement NFR1. Sec-

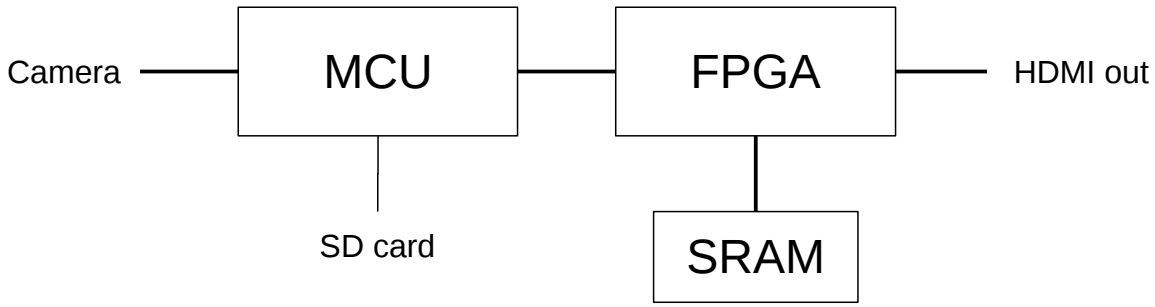


Figure 3.2: Video Data Flow

ondly, this allows hardware accelerated convolution, which will hopefully make it possible to achieve the speeds required for real time convolution. Thirdly, this demonstrates how an FPGA can be used as reconfigurable hardware to accelerate application specific tasks.

This configuration of the FPGA is going to be done by the MCU, which will fetch the configuration from an SD card so that the configuration can be easily exchanged for a different one to harvest the advantages of reconfigurable hardware.

In addition to configuring the FPGA, the MCU should also handle the input stream arriving from a camera or another external video source. As convolution is done on the FPGA, the MCU should forward the video stream to the FPGA and possibly perform some preprocessing should the video format from the camera be unsuitable for the processor on the FPGA. This may include indexing the colours or reducing the colour depth to a different colour representation as described in Section 2.2, which may be necessary to speed up the processing and reduce throughput.

Connected to the FPGA is also some extra memory as the internal memory on the FPGA is limited in size. This memory in supposed to work as a buffer to make the system more tolerable to variable data bit rates and to allow the output to read old data while waiting for the processed input to arrive.

As HDMI output is not available on the MCU and the bus between the MCU and FPGA is already assumed to be a bottleneck, the HDMI output is placed on the FPGA.

Figure 3.2 displays the overall data flow between the components in the system. Notice that most of the data flows from the left to the right. Video arrives through the camera and flows through the system all the way to the HDMI output; FPGA configurations flow from the SD card through the MCU to the FPGA on configuration.

3.1.2 Backup Solutions

During the project, a risk assessment was done to find and reduce the risk of failure. This lead to some backup solutions being implemented in the design.

One such solution was to exaggerate the number of address lines from the MCU to the FPGA to support addressing the memory connected to the FPGA directly (through the FPGA). This will be useful should the communication directly from the MCU to the processor prove hard to establish. This is a critical point because the data needs to cross clock domains correctly.

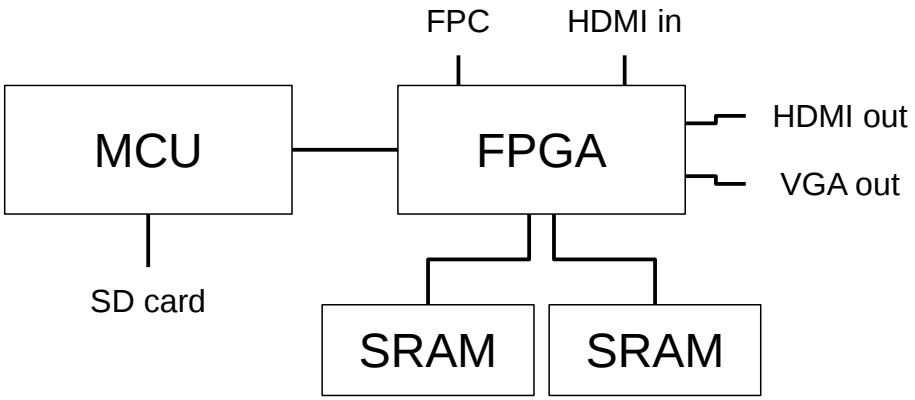


Figure 3.3: Alternative System Architecture

One of the measures taken was to add an extra HDMI port connected to the FPGA in case the attempt to transfer video from the MCU to the FPGA failed or the throughput proved to be insufficient. This allows the camera to be connected directly to the FPGA and thereby circumvent both the MCU and the bus between the MCU and FPGA, as shown in Figure 3.3 In addition, a connector for a Raspberry Pi camera (FPC) exists for connecting a camera to the FPGA.

Should all camera input strategies fail, video or images can be loaded from the SD card or generated by the MCU.

Also shown in Figure 3.3 is an extra SRAM chip. The extra chip doubles the available throughput between the memory and the FPGA and reduces the chance of conflicts between the processor and the HDMI controller which will access the memory at the same time. Using double buffering, they will never access the same memory at the same time.

Header pins for VGA output were also added to the FPGA in case HDMI output ended up being harder than first assumed.

3.2 Development Tools

Throughout the development process various tools were used. This section lists the most important ones.

Chisel
Used to write the processor implementation. See Section 2.4.1 for background.

Xilinx ISE, ISim, Impact
Used to create various FPGA modules with VHDL and Verilog and make them work together with the processor implementation.

Simplicity Studio
Used to develop for and debug the MCU.

Giant Gecko Development Kit EFM32GG-DK3750
Used to flash and debug the MCU and to test code during development.

Saleae Logic Analyzer
Used to inspect digital communication over various busses during development.

Altium Designer
Used to design the PCB layout.

3.3 PCB Design

This section describes the PCB design and explains the decisions related to this. See Table A.1 for a list of components used on the board.

3.3.1 Board Features

The board has been designed to put important components close to each other, to reduce the distance and time to transfer data between components. Header pins are present for most components for debugging and in case something should be wrongly wired inside the board. Figure 3.4 shows an overview of the board with various components labelled.

The main features of the board are listed below.

- Xilinx Spartan-6 FPGA
- Silicon Labs EFM32GG MCU
- Connectors
  - HDMI I/O
  - FPC Camera
  - MicroSD
- 120MHz Oscillator for FPGA
- 48MHz Crystal for MCU
- Digital I/O
  - 7 Mechanical Buttons
  - 7 Expansion Headers
  - 2 LEDs
- External memory
  - 2 AS7C38098A SRAM

3.3.2 Power Supply

The board is powered by a 5V MiniUSB connector. The input voltage is regulated with a 3.3V low-dropout (LDO) regulator, which gives power to most of the board. This 3.3V is also connected to a 1.2V LDO regulator which powers the minimum required power pins on the FPGA. There is a secondary power connection for backup in case the first should fail somehow.

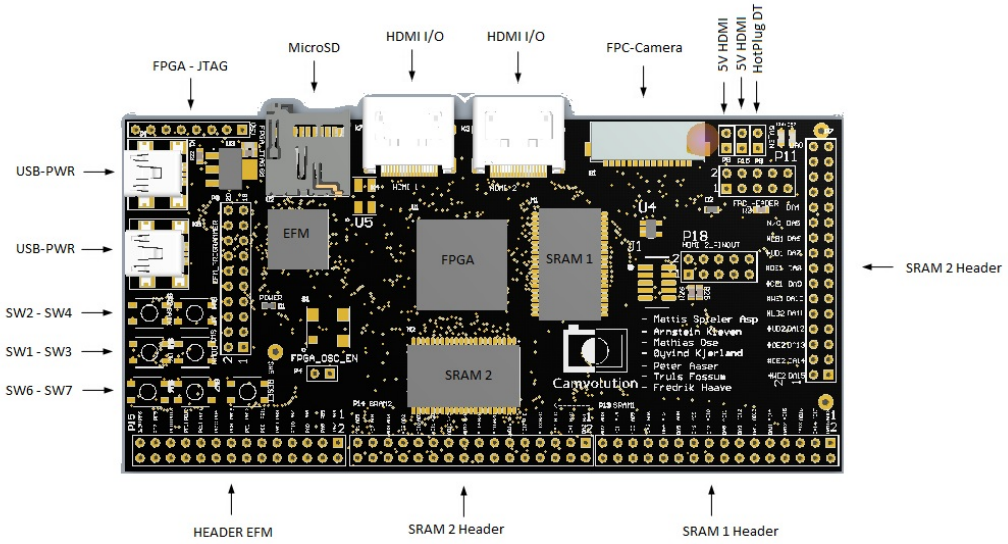


Figure 3.4: Overview of Camvolution board

3.3.3 Clock Generation

The FPGA oscillator is connected to an enable signal on the MCU. To enable the 120MHz oscillator, the signal must be set to output and driven high. This enables the MCU to start and stop the FPGA and reduce the power consumption. The oscillator enable pin is also connected to an external input and can be set high by connecting this pin to VCC. This can be done by adding a jumper to on the FPGA_OSC_EN header.

In addition to the oscillator, the MCU has one internal and one external clock.

3.3.4 Connectors

HDMI

There are two HDMI connectors on the board. Both of them can be connected to 5V with jumpers. This is not required for the HDMI to work, but is added in case power should be required by other devices. The port labelled HDMI 2 is connected to header pins. See Table B.1 for pin-outs.

FPC Camera Connector

There is one flexible printed circuit (FPC) connector on the board intended for the Raspberry Pi Camera. It is also connected to header pins. See Figure B.1 for pin configuration.

3.3.5 Digital I/O

Buttons

The board contains seven buttons. When a button is pushed down, the input pin will be grounded.

The two buttons are for FPGA control, and four buttons for MCU control. The last button is connected to reset on the MCU. See Table B.2 for info.

Header Output

The MCU has spare I/O pins that all are available on header pins. These can be used as GPIO for the system if neccessary. See Table B.3 for location of pins.

3.3.6 SRAM

The FPGA is connected to two SRAM chips, one on bank 1 and another on bank 2. The chips support a data width of 16 bits and have an access time of 10 ns [3], which gives a maximum throughput of $16\,\text{bit} * (10\,\text{ns})^{-1} = 1,6\,\text{Gbps}$ to each chip. This is more than sufficient for this projects' application as can be seen in Table 2.3.

To make debugging easier, all wires connecting the FPGA and SRAM chips have been placed on headers.

3.3.7 JTAG

Both the FPGA and the MCU is connected to a JTAG header. The MCU is connected to a 20 pin JTAG header that can be used with the generic programming header from the MCU development kit. The FPGA is connected to a 8 pin JTAG header. This can be routed to a debugger used for FPGA programming. See Table B.4 for MCU JTAG header and Figure B.4 for FPGA JTAG header.

3.3.8 Programming FPGA Using MCU SPI

The FPGA can also be programmed using serial peripheral interface (SPI) from the MCU. INIT_B shares the same pin as the SRAM2 data line, but when programming the FPGA the SRAM will not be active. See Table B.5.

3.3.9 LED

There are two LEDs on the board. One is connected to power on the board and will light up if power is connected. The second is programmable from the FPGA.

3.3.10 External Bus Interface (EBI)

The FPGA and the MCU is connected with a high speed EBI bus with 20 addressable pins and 16 data lines. They are not connected to header pins. If any wire design error had occured, there would not be any purpose of having them on headers, as it could be manually changed in the FPGA. See Appendix B, Figure B.2 and B.3 for EBI pinouts.

3.4 MCU

An EFM32GG is used as the MCU for the system. It provides 128kB RAM, 1024kB internal flash memory and a range of modules including EBI, USART, GPIO and DMA. The MCU is used as an I/O-processor, reading from a MicroSD card, configuring the FPGA, and feeding configuration and image data to the processor over an EBI bus interface. Figure 3.5 shows how the components are connected.

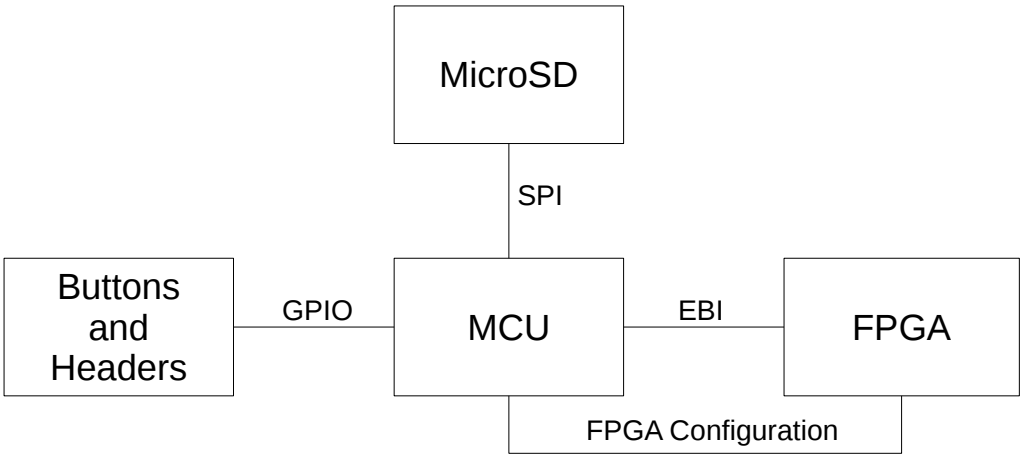


Figure 3.5: Overview of the components connected to the MCU

3.4.1 Configuration

In this section the term configuration is used in two different contexts.

- Configuration of the FPGA means configuring the logic on the chip by resetting it and programming it with a binary configuration file.
- Configuration of the processor means configuring the processor that is implemented on the FPGA with kernels, map operators and reduce operators.

3.4.2 Storage

The MCU has a limited non-volatile memory of 1024kB. Because of this, an additional larger source of data is necessary for storing files such as images and configuration files for the FPGA. A MicroSD card is used for this storage. It gives the MCU an external storage of up to 32GB and lets the system operate independently without being connected to an external computer.

The code for interfacing the MicroSD card is based on the *FAT on SD Card Application Note* [8]. The MCU communicates with the MicroSD card using SPI transfers with a

USART module. FAT32 is used as a file system to make reading content from the MicroSD card more manageable, with the trade-off being increased computational overhead for the MCU [10]. This also allows users to add and replace files without having to reprogram the MCU.

In the event of the MCU being unable to access the MicroSD card, it is possible to include kernel files and a few small images on the internal flash memory. Functionality for flashing the FPGA from the MCU will be lost because the size of the configuration files exceeds that of the internal flash memory.

The folder structure on the MicroSD card is shown in Table 3.1.

Folder Name	Contents
binfile	Contains binary configuration files that can be used to program the FPGA to alter its behaviour.
image	Contains images that can be fed to the FPGA. Serves as a backup for the camera input. Supports 24-bit BMP files and 24-bit raw image data.
kernel	Contains different kernels used for convolution.

Table 3.1: MicroSD card folder structure

3.4.3 FPGA Configuration

When the system is powered up the FPGA will not contain a configuration. To be able to use the FPGA properly, the MCU has to program it with one of the binary files from the storage. Pins between the MCU and the FPGA are connected according to Slave Serial as described in *Spartan-6 Configuration User Guide* [16, Page 28]. This configuration scheme relies on sending a binary file as a serial data stream using a clock pin and a single data pin. The MCU is able to reset the configuration of the FPGA at any time by setting the program pin on the FPGA low. The configuration status pins are set by the FPGA. This informs the MCU if it is ready to be programmed, if an error occurred while programming or if the FPGA has been successfully programmed. Figure 3.6 illustrates the FPGA configuration procedure.

The FPGA can also be reprogrammed later. For example when using kernels of different sizes, a configuration optimised for a specific size can be configured on the FPGA.

If the MCU is unable to read a configuration file and thus unable to flash the FPGA, it is still possible to use a JTAG header to flash manually.

3.4.4 Kernels

Being able to change the convolution kernels while the system is running is considered an important functionality. To support this, the MCU is able to read convolution kernels from MicrodSD or the internal flash memory. In order to make the process of changing kernels for the system easy, the kernel files has a simple file structure. The kernel files consists of 8-bit values. The first byte tells the size of the kernel. For a 3x3 kernel the size will be 9. This limits the system to a maximum size of 15x15 kernels. The rest of the bytes in the files are the elements of the kernel.

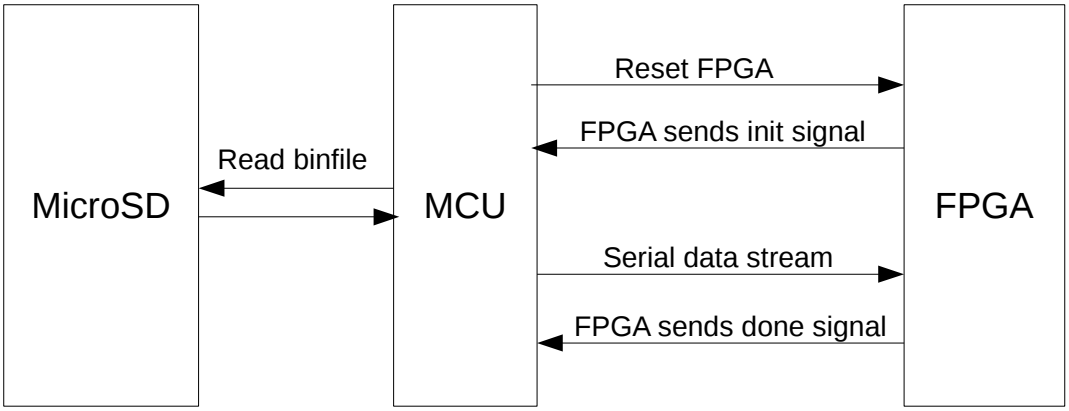


Figure 3.6: FPGA configuration procedure

3.4.5 EBI

The FPGA and the MCU is connected through an EBI bus. This interface is supported natively by the EFM and can therefore be handled by its built-in direct memory access (DMA) unit, which hopefully makes streaming video from the MCU faster, more reliable and easier to implement.

In addition to streaming video, the MCU should also control the operation of the processor.

EBI Banks

The EBI module on the MCU supports reading and writing to multiple memory banks, where each bank has its own memory space on the MCU and its own chip select signals. For the system, one bank is used to access SRAM, and another bank is used to communicate with the processor.

Even though the width of the EBI bus is 20-bit, only 3 addresses are used when communicating with the processor. Table 3.2 explains the functionality of the different adresses.

Address	Functionality
0	Data stream. Used for sending images and configuration data to the processor.
1	EBI mode override. Writing 1 to this address will give EBI access to SRAM.
2	Reset. Writing anything to this address will reset the processor.
>2	Data written to addresses larger than 2 will be ignored by the processor.

Table 3.2: Address functionality for processor bank

Processor Configuration

To configure the processor with new kernel, map and reduce values, the MCU needs to be able to reset the processor. After the processor has been reset, the MCU writes a sequence of configuration values to address 0. When the processor is fully configured, the MCU can either be set to send an image stream or just wait for input. The procedure for configuring the processor is illustrated in Figure 3.7.

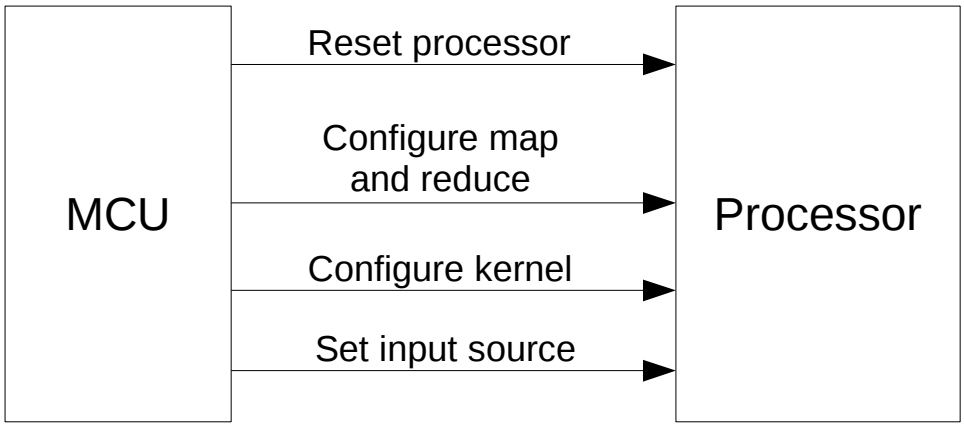


Figure 3.7: Processor configuration procedure

SRAM Access

Instead of sending everything as a data stream to the processor, the MCU can access the image memory in SRAM. This makes it possible for the MCU to only update a part of the screen, instead of updating the whole screen every time it needs to be refreshed.

For the MCU to access SRAM it needs to enable the EBI mode override by writing 1 to address 1 on the processor bank. By doing this, the MCU will receive direct access to the SRAM through the FPGA. The EBI bank for accessing SRAM can then be used to read and write from memory.

Writing 0 to address 1 on the processor bank will disable the EBI mode override, and resume normal operation of the processor.

Figure 3.8 shows the MCU writing and reading from SRAM.

3.4.6 User Interface

The system has a simple graphical user interface which is displayed on the screen. The interface has menus where it is possible to change different configuration values for the processor, such as kernels, map and reduce operators and FPGA configuration. Controlling the user interface is done with buttons, making it easier to use and removing the

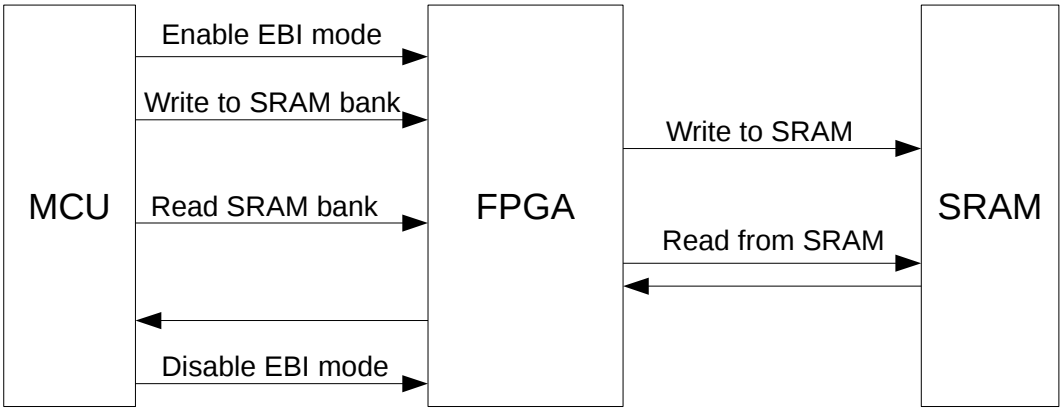


Figure 3.8: SRAM access over EBI

need for being controlled by an external computer.

The system provides four buttons for the user interface:

- UP
- DOWN
- BACK
- OK

Start-up and Reset

When the MCU is first started up or reset, it reconfigures the FPGA with a default binary file. After configuration of the FPGA, the MCU configures the processor with default values for the convolution kernel, map operation and reduce operation. The processor's image input is then configured to the default input, which can be the camera input or HDMI input.

Main Menu

When pressing the BACK or the OK button, the MCU configures the processor image input to be the EBI bus and the map and reduce operations on the processor to not alter the image. An image containing the main menu of the grapical interface is then loaded from the MicroSD card and sent to the processor over the image stream. The main menu displays the following options:

- Configure kernel
- Configure FPGA

- Configure map operation
- Configure reduce operation
- Select image input source

The UP and DOWN buttons are used to select an option. The OK button bring up a new menu screen listing possible options for the chosen selection. When selecting configure kernel or configure FPGA, a list of the availiable files are presented. Choosing any of the other main menu options will bring up a list of predefined choices. Selecting any of these choices will make the MCU reconfigure the processor with the new values, and go back to showing the camera image stream.

Figure C.1 shows a concept image of the Main Menu.

3.5 FPGA

The FPGA must handle an incoming video stream and the configuration data arriving from the MCU before the actual convolution is performed and the final video stream outputted. In addition, since some of the different components run at different clock speeds, queues are needed to synchronise data on the boundaries between clock domains.

In Figure 3.9, the top level modules on the FPGA are depicted.

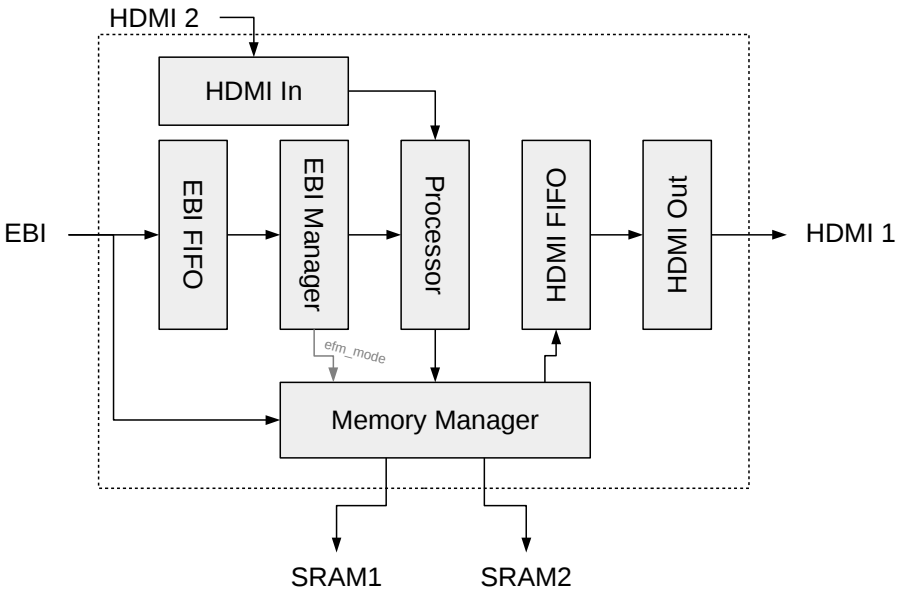


Figure 3.9: FPGA Overview.

3.5.1 EBI

As EBI is asynchronous, the incoming data for the FPGA must be synchronised to the processor clock domain before being used by clocked logic in the FPGA. This is done by sending the data through a first-in first-out (FIFO) queue immediately after arrival.

After being synchronised, the EBI manager checks the addresses to determine what action to perform. In some cases, the data is resized from 16 to 24 bits and forwarded to the processor. In other cases, the data sets a control bit in the EBI manager which turns on EFM mode for the FPGA.

In EFM mode, EBI is routed to one of the memory chips to allow the MCU to read and write data and the HDMI output module is locked to the very same chip. This allows the MCU to paint user interfaces on the screen so that the system can be controlled through buttons. Configuration may include loading a new bit file onto the FPGA or setting up a different kernel for the processor.

3.5.2 Memory

The memory manager handles reading and writing to memory, and swapping the chips used as a double buffer when a frame is finished. This includes calculating what addresses

to place the arriving data at and what data to read.

In addition, the chip enable (or write enable) signal for the memory chips have to be pulsed in order to save data, which requires write cycle management. This consists of a set-up phase and a write phase, giving a duty cycle of 50 % for the chip enable signal.

As stated in Section 3.3.6, the memory chips are 16 bits wide, thus data from the processor must be resized from 24 bits to 16 bits before being written to memory. Similarly, data read from the memory must be resized back to 24 bits before being sent to HDMI.

3.5.3 HDMI

Before the data from memory arrives is used by the HDMI module, it passes through a FIFO queue which has two functions:

- Synchronise the data to the clock domain of the HDMI.
- Act as a buffer in case the memory is too slow.

This way, it may be possible to hide the speed difference between the memory and the HDMI because there is a pause between each frame that the memory can use to catch up with the HDMI data consumption. An additional challenge is that the HDMI clocks out a 24-bit pixel on each cycle while the memory only reads 16 bits each cycle.

3.6 Processor

In this section describes the processor which lies at the heart of the architecture of Camvolution. The processor will first be described as seen from an outside perspective before each component inside is explored.

3.6.1 The Processor as a Component

A design goal for the system was to reconfigure the FPGA depending on the task, which necessitates synthesising several versions of the architecture so that the correct architecture may be chosen at run time. In addition to using several architectures, each version is also programmable, allowing the kernel values, map operators and reduce operators to be programmed at runtime. To facilitate generating different versions of the architecture, the design was parameterised, allowing the generation of different architectures by simply changing a parameter once. The parameters available are:

Kernel Dimensions
: The most fundamental parameter in the design, the kernel dimension dictates how many pixels are needed to calculate a single output pixel. This is very fundamental parameter, and setting the kernel dimension will effect other important sizes in the system.

Image Data Input Width
: The width which input image data is presented.

Pixel Width
: The width of each pixel. When image input width and pixel width does not match, for instance with 16 bit input width and 24 bit pixel width, the processor must buffer and translate the input stream.

Control Data Input Width
: As with input data width, the processor must know the width of the incoming control data.

Image Data Output Width
: The data width that the processor should output. As with input, translating widths is necessary.

By parameterising the processor it can be viewed as any other module with the inputs and outputs shown in Table 3.3.

To the outside system the processor is now a module that once programmed operates on a data stream of an image and outputting a processed stream. By using valid signals the outside system can provide data at any pace, and throttle the output by requesting data when the processor indicates that data is ready.

3.6.2 Architecture of the Processor

Figure 3.10 shows the top level schematic of the processor with the following components:

	Signal	Width	
Input	Image data in	Image input width	Data stream from camera on row format
	Image data valid	1	image data input is valid
	Control data in	Control data width	Data stream from MCU
	Control data valid	1	MCU data is valid
	Request image data	1	Current output data is received
	Reset	1	Processor should reset
Output	Image data out	Image input width	The processed image data on row format
	Image data valid	1	Current output data is valid

Table 3.3: The interface of the processor, specified by its parameters

Input and Output Buffers

The input buffer is a double buffer resonsible for buffering rows of image data and feed it to the processor. The buffer holds several rows of image data, and when it is full it feeds data to the processor as a series of column slices while the other buffer is filled.

The output buffer retrieves data from the processor as a series of column slices and rearranges the output back to row format. Figure 3.13 shows how the input buffer feeds the processor with data from a buffered set of rows. When data is fed on a column major format within its set of rows we call the data stream a *sweep*, inspired by the way a window is washed by sweeping it with a squeegee. The column slices in the sweep are called *sweep slices*.

Control

The control unit is responsible for keeping track of the input and output buffers, waking the processor once an input buffer is ready and making sure the output buffer is empty before a new feeding cycle is started. The control unit is also responsible for programming the processor after a reset.

Core

The heart of the convolution design, where the actual convolution happens

The processor does the following tasks:

- Provide a clean interface to the outside system, accepting an image stream and providing an output stream on row major format, adding tolerance for input streams which may come at uneven intervals.
- Once the input buffer is ready, wake up the core and provide data as an uninterrupted column major image stream, essentially working as a wrapper for the core.

3.6.3 Core

This section provides a description of the heart of the architecture of Camvolution.

From the outside, the core is a module working on sweep slices as shown in Figure 3.13. A good analogy for how the image is processed is how a window cleaner washes

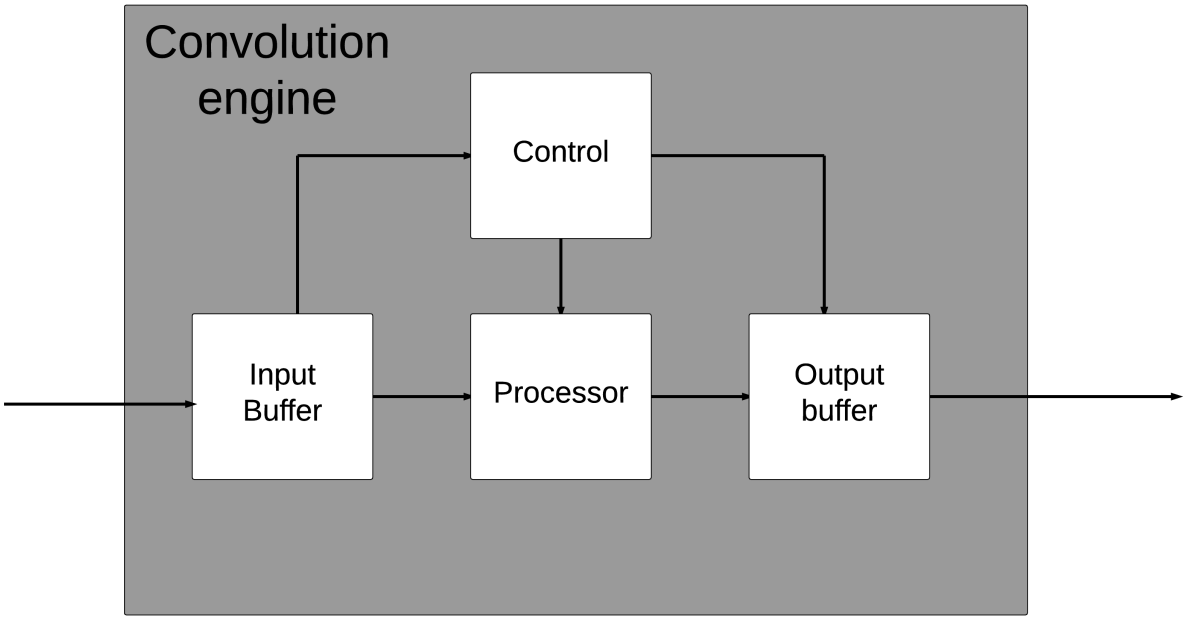


Figure 3.10: The top level schematic of the processor

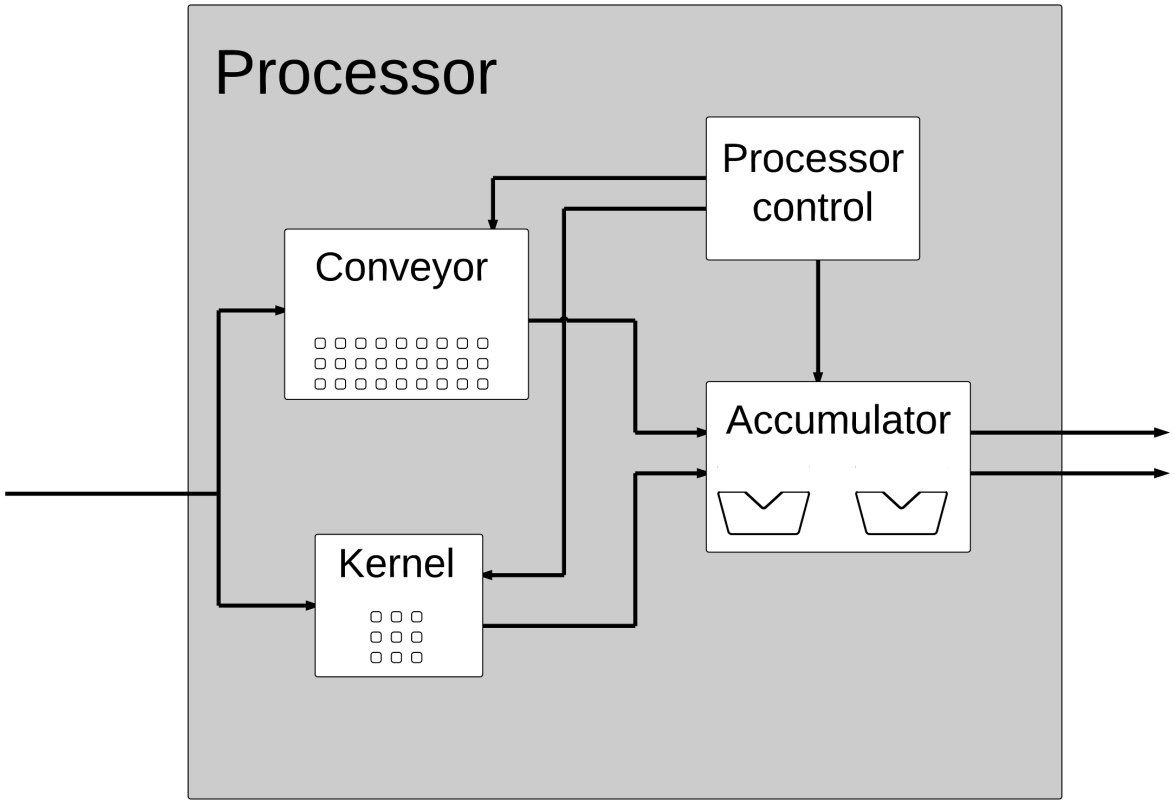


Figure 3.11: The four main components of the core

a window. If we let the dirty window be the original picture, and the clean image be the processed image, the process of convolution is akin to sweeping a squeegee over the window horizontally. In this analogy, a very small part of the window is under the squeegee at any time. Similarily, the core works on a small part of the image at any time,

only large enough to contain enough pixels to calculate a single slice of output pixels, called the *sweep frontier*, as shown in Figure 3.14 and Figure 3.15. To illustrate how data is fed each row in the sweep can be viewed as a stack. Each stack is filled with a row from the input image, and once all rows are full they take turns popping one of their elements such that the core recieves sweep slices. Figure 3.11 shows an overview of the core which contains four modules:

Conveyor
: The conveyor is a special register file consisting of several rows of pixel registers, which holds the pixels currently in the sweep frontier. Data is delivered to the accumulator in a pattern controlled by the control unit, it is however never explicitly requested.

Kernel Buffer
: The kernel buffer maintains the kernel data and must provide the correct kernel data for each map operation performed in the accumulators. This module is also responsible for programming operators.

Accumulator
: To convolve a pixel, a mapping unit which maps the neighbourhood of a pixel is employed, paired with a reducing unit working exclusively with the mapping unit it is paired with. The mapping unit reads the pixels of a pixel neighbourhood in sequence, and each mapped pixel is processed by the reduction unit which performs a reduce operation with the new pixel and the already accumulated pixel data. Together these two units forms a single accumulation unit, and by employing a row of these units together several pixels may be computed in parallel.

Control Unit
: In order for the accumulators and pixel conveyor to be in sync a control unit sends control signals periodically, which are then propogated throughout the system.

Figure 3.11 shows the insides of the core. The conveyor belt shows three rows, each row keeping a column slice from the sweep frontier shown in Figures 3.14 and 3.15.

Conveyor

The conveyor maintains the sweep frontier, and feeds pixel data to the accumulators. It consists of a grid of pixel registers, each row holding a sweep slice. Since each pixel in a slice corresponds to a different row, every pixel register in a row has a one to one correspondence to one of the rows of the sweep currently feeding the core.

To reiterate on this, a row in the conveyor corresponds to one of the slices in the input buffer, not a row in the input image.

Data management on the conveyor is done by sending keys which are are issued from the control unit, and passed from register to register within a row. At any time, there is on each row two keys held by different registers: a read key and a write key. When holding a write key the register is write enabled and should read data from the row above it, while a read key tells the register that its content should be driven its content to an interconnect which the accumulator and the row below it can read from.

Input to the conveyor is available to the entire top row, so in order to ensure that only the corresponding pixel register reads the data, the control module has to issue a key to the row at the right time. To illustrate, Table 3.4 and Table 3.5 show when each row has access to a read key for a 3x3 kernel.

Time (cycle)	T_0	T_1	T_2	T_2	T_4	T_5	T_6	T_7	T_8	T_9	T_{10}	T_{11}	T_{12}	T_{13}	T_{14}
Row 1	4	5	6	7	8	9	1	2	3	4	5	6	7	8	9
Row 2	7	8	9	1	2	3	4	5	6	7	8	9	1	2	3
Row 3	1	2	3	4	5	6	7	8	9	1	2	3	4	5	6

Table 3.4: A sequence diagram showing which registers in each row is being read at each timestep. Greyed out cells shows the pixels in the neighbourhood of the output pixel corresponding to output row 1.

Time (cycle)	T_0	T_1	T_2	T_2	T_4	T_5	T_6	T_7	T_8	T_9	T_{10}	T_{11}	T_{12}	T_{13}	T_{14}
Row 1	4	5	6	7	8	9	1	2	3	4	5	6	7	8	9
Row 2	7	8	9	1	2	3	4	5	6	7	8	9	1	2	3
Row 3	1	2	3	4	5	6	7	8	9	1	2	3	4	5	6

Table 3.5: A sequence diagram showing which registers in each row is being read at each timestep. Greyed out cells shows the pixels in the neighbourhood of the output pixel corresponding to output row 1.

The second leftmost output pixel follows the same pattern, only shifted left by one cycle.

A register is given a read key not only to make its content available to the accumulators, but also so that the row below may have its content written to it in order to propagate data downwards. Each register should read the register directly above, so the write keys must be issued such that a register only has write enabled when the register above it is currently driving the interconnect.

For the conceptual design, this is easy, however in the actual implementation data is moved using register balanced multiplexer trees rather than simply driving a bus. This means a write key is not really given to the register, but to a multiplexer deciding which register should currently be read.

Additionally a secondary write key is issued since the register balanced multiplexer tree has two stages, but this will not be elaborated further upon in order to not clutter the report with implementation details.

Accumulator

As each pixel register in the conveyor corresponds to a row in the sweep, each accumulation unit corresponds to an output pixel. For a 3x3 kernel each sweep consists of nine rows, which provides seven output rows since the top and bottom row of each sweep misses some of its neighbours. Each accumulation unit corresponds to one output row, requiring seven accumulation units in total.

Conversely, for a 5x5 convolution kernel would use 25 input rows, and lose two rows in each end from missing neighbours, resulting in 21 output rows, and 21 accumulation units.

Table 3.4 shows that each accumulation unit can map all the necessary pixels by correctly timing which row to read from. In the case of the first accumulator, it reads from row 3 at time $T0, T1$ and $T2$, row 2 at time $T3, T4$ and $T5$, and row 1 at time $T6, T7$ and $T8$, as shown in grey. At $T8$ all necessary pixel data has been read, and the accumulated result is output and a new cycle begins.

The second table in the conveyor section shows the same pattern for the accumulator corresponding to output row 2, only delayed by one cycle. This delay means that every accumulator can access the data it needs by simply timing when it reads output from the rows. Additionally, since each accumulator is delayed one cycle relative to its neighbour, the access patter for kernel values is similarily delayed.

This kernel access pattern means every accumulator needs the kernel used by its predecessor last cycle, meaning that rather than keeping kernels in a central repository they can keep in a shift register with each element very close to an accumulator.

Rather than keeping track of their progress, each accumulator passes a special flush key akin to how registers use read and write keys. Upon recieving a flush key an accumulation unit resets its content and its finished data is written on the output wire.

Control Unit

The control unit is responsible for issuing keys to the rows at periodic intervals such that reads, writes, flushing and selecting accumulator output is correctly synchronised. In addition to synchronising the accumulators and conveyor, the control unit puts the core to sleep when no data is available, and is responsible for sending signals indicating that the accumulators should read instruction data. The issuing of keys is done by a simple state machine, for which the logic can be calculated at synthesis time, since it is dependant on kernel size alone.

Kernel Buffer

The kernel buffer is responsible for collecting kernel data at programming time, and to supply the correct kernel at the correct time to the mapping unit. Figure 3.11 shows the kernel buffer holding all the kernels in a separate buffer, but as described in the Accumulator Section this buffer is actually a chained set of registers where each register is close to one of the accumulation units.

When in programming mode, the kernel buffer gets instructions from the pixel stream rather than a separate channel. First, instructions are sent, which are then sent as kernels to the mappers instead of using a separate instruction channel. When piggybacking instructions on the kernel chain, it is nescessary to wait for the instructions to fully propagate, but this is an acceptable delay since programming is done only once per run.

In addition to providing instructions, the kernel buffer is also responsible for realigning kernels when the core goes to sleep, such that when a new feed cycle begins the system is in the state that the control state machine expects.

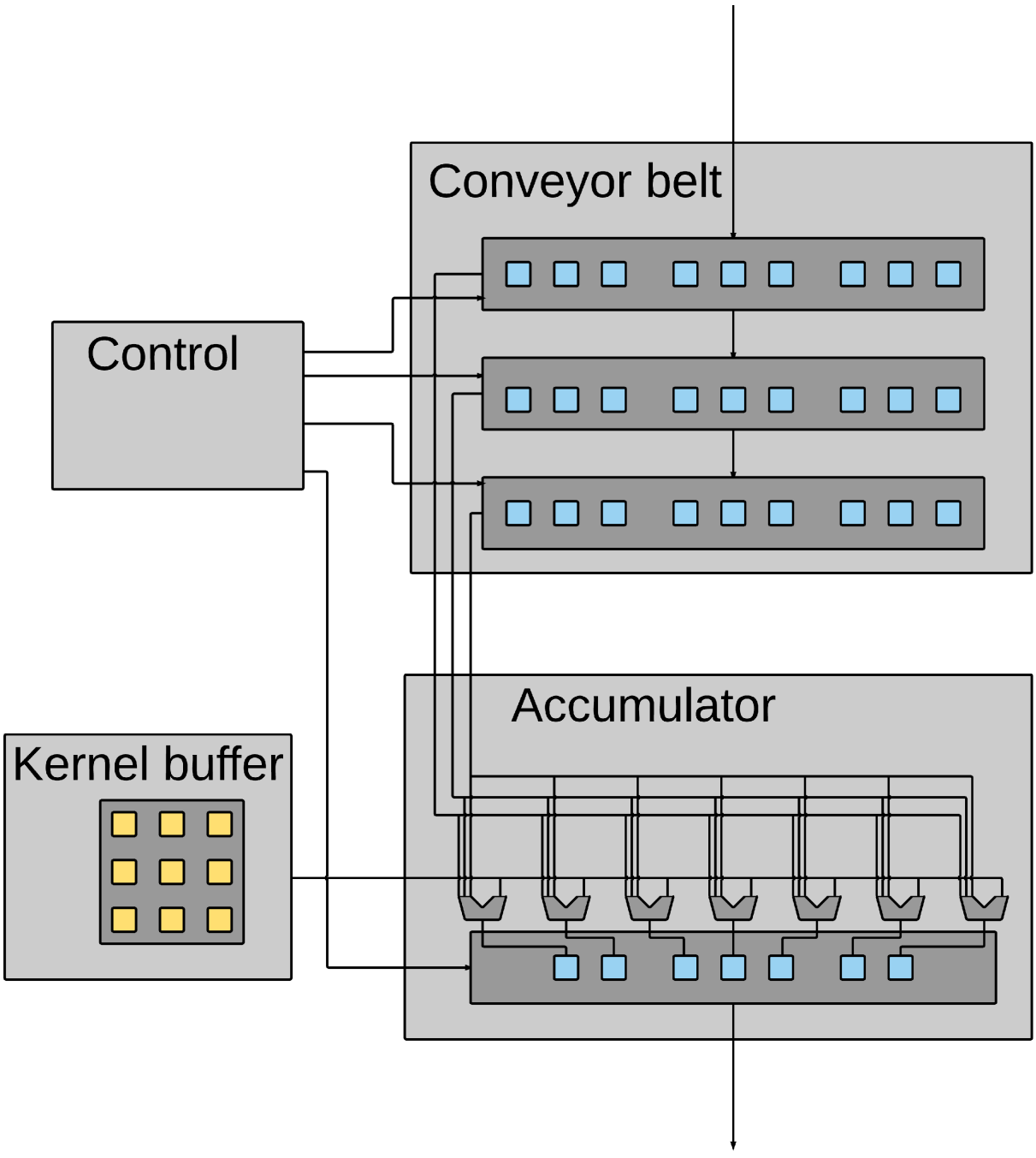


Figure 3.12: The four core units in the convolution process

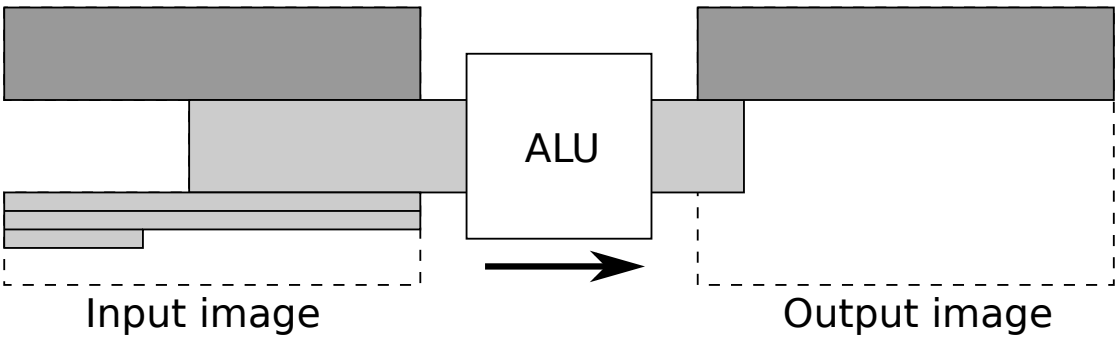


Figure 3.13: A full sweep has been buffered, and is being fed to the processor. Meanwhile, a new sweep is being buffered.

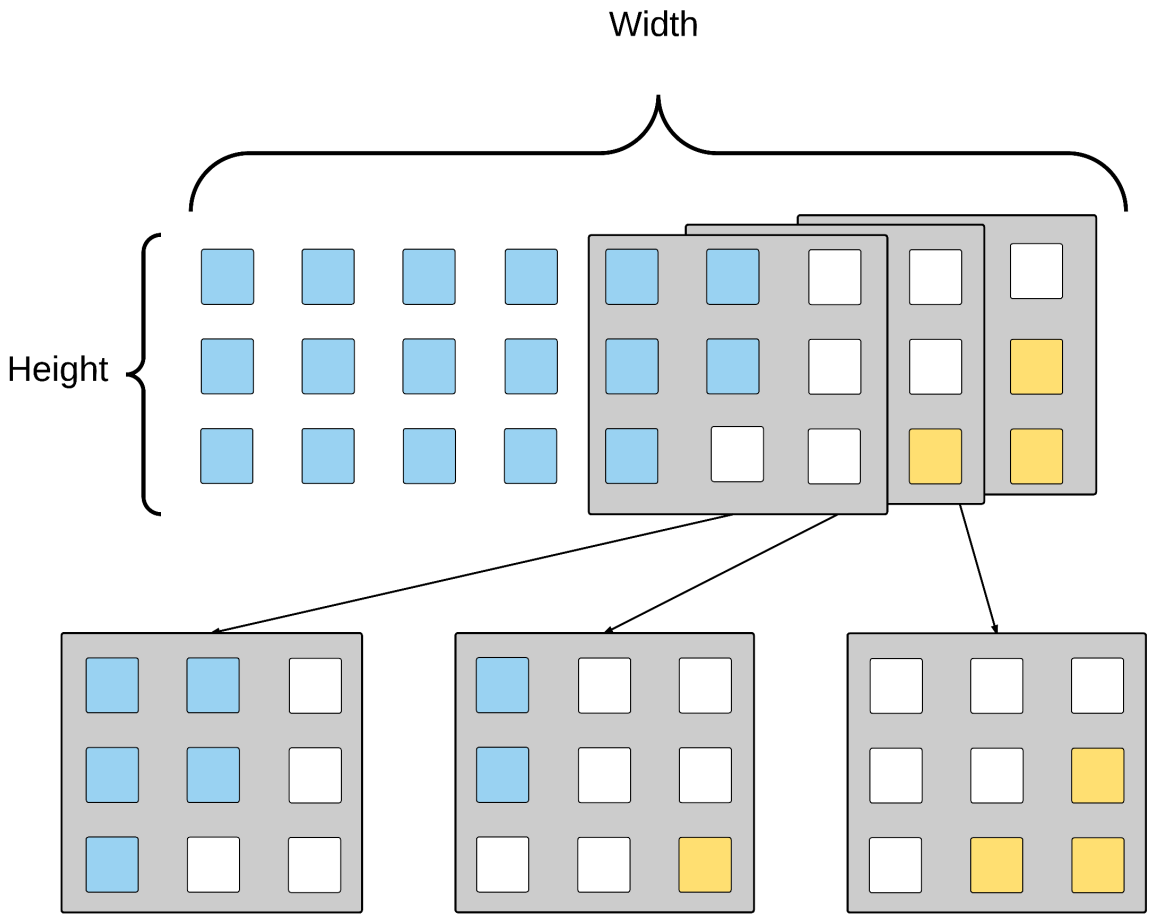


Figure 3.14: The frontier. The three rightmost pixels are shown with their full neighbourhoods, which overlap with their neighbours.

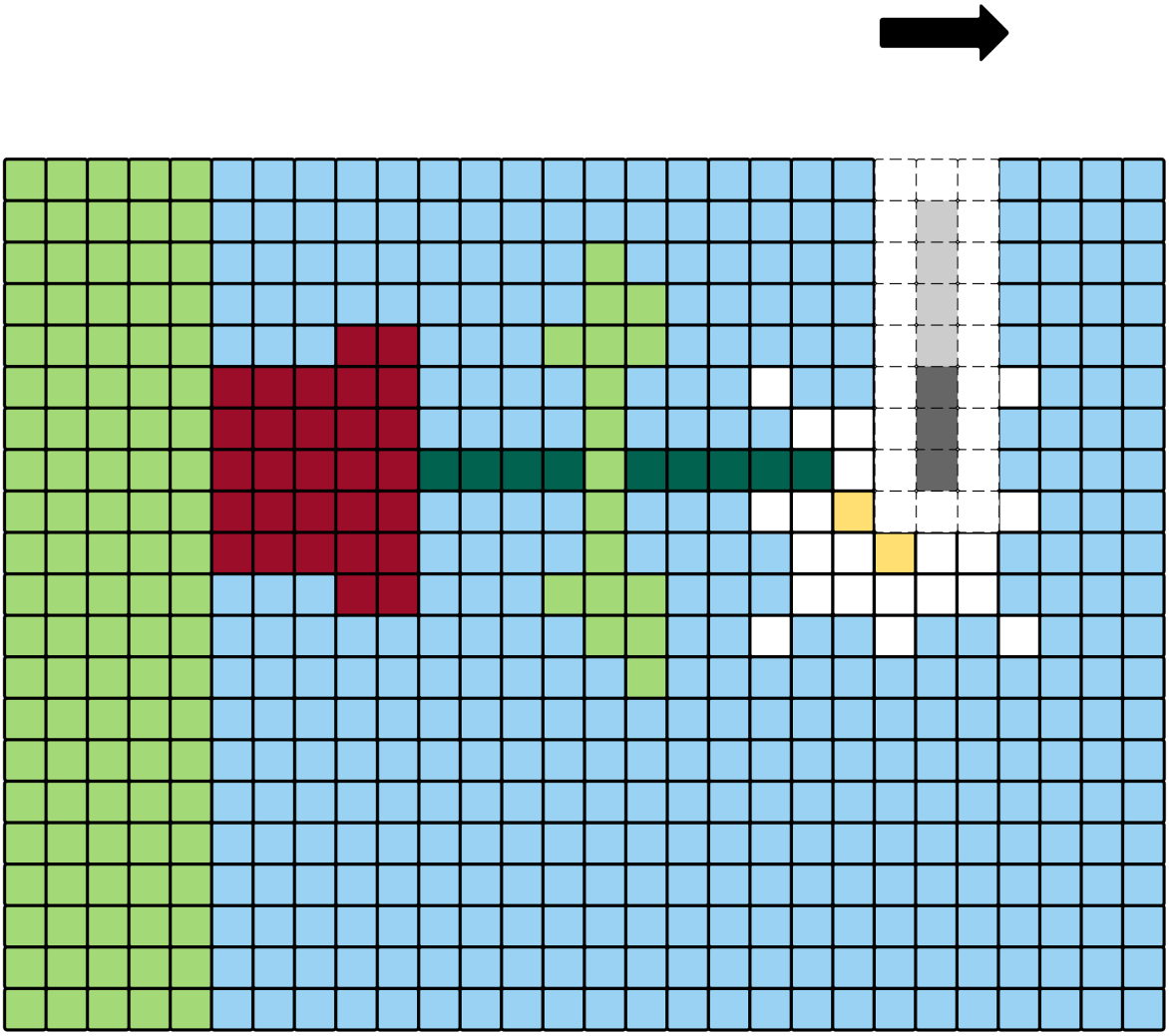


Figure 3.15: The image from which the frontier in Figure 3.14 is taken

3.7 HDMI

The implementation of HDMI on the FPGA was based on the Xilinx application note 495 [15] and its reference design files.

An HDMI input was used as the source for the picture to be convolved, and an HDMI output to show the result. HDMI uses TMDS at the physical layer, and the implementation uses the native TMDS I/O interface that is featured by the Spartan-6 FPGAs.

3.7.1 Input

The TMDS clock signals are sent to a differential buffer, a Spartan-6 primitive called BUFIO2, and then into a phase-locked loop (PLL) to get the required clocks. The required clocks for making HDMI signals into 10-bit words are:

- Pixel clock, which is the same rate as the TMDS clock.
- 2x pixel clock, used when making 5-bit words into 10-bit words.
- 10x pixel clock, used as I/O clock.

After the clock is retrieved from the TMDS signals, it sends the TMDS signals for each of the colours into their own instance of the decode sub-module, together with the PLL generated pixel clock that is based on the incoming TMDS signal clock. An overview of the HDMI receiver can be seen in Figure 3.16.

After the colour channel signals are decoded, the pixel data is written to a FIFO queue which is to be read by the convolution processor.

3.7.2 Output

The processed pixel is read from a FIFO queue. It is a 24-bit word, with each colour being 8 bits. A pixel clock for the transmitting end of the HDMI interface is generated by a PLL, and each colour and the pixel clock is serialised with a frequency of 10 times the pixel clock before being sent out as differential signals. An overview of the HDMI transmitter can be seen in Figure 3.17.

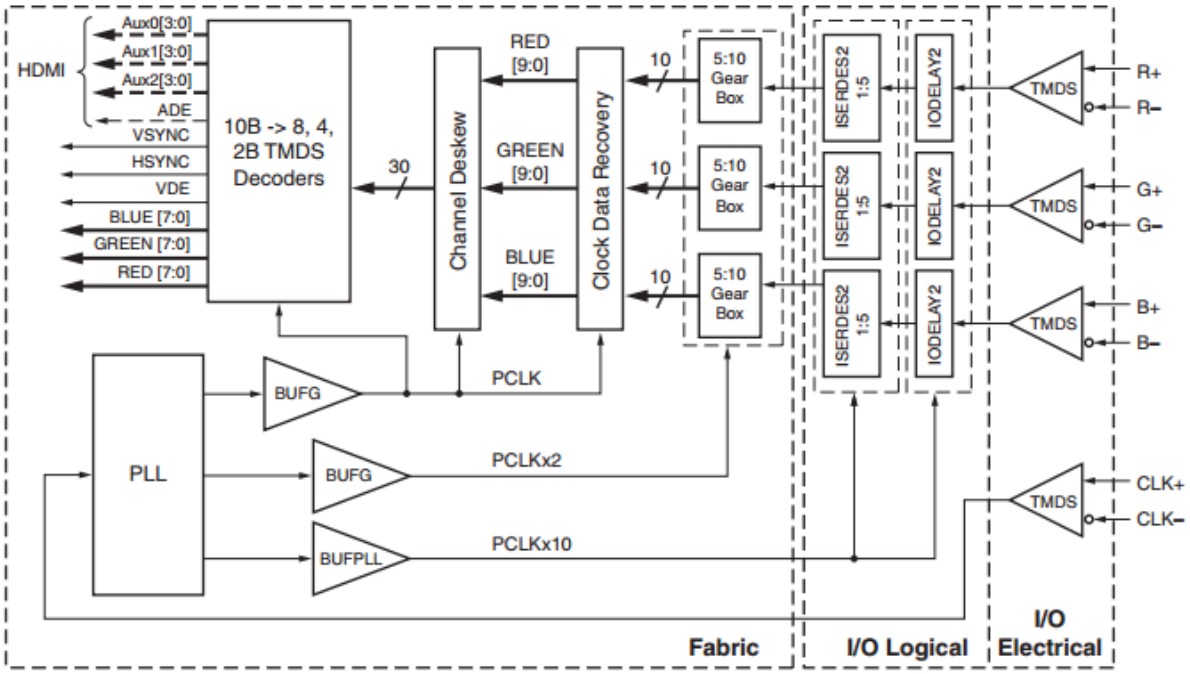


Figure 3.16: Overview of the TMDS receiver design. Taken from the Xilinx application note 495 [15].

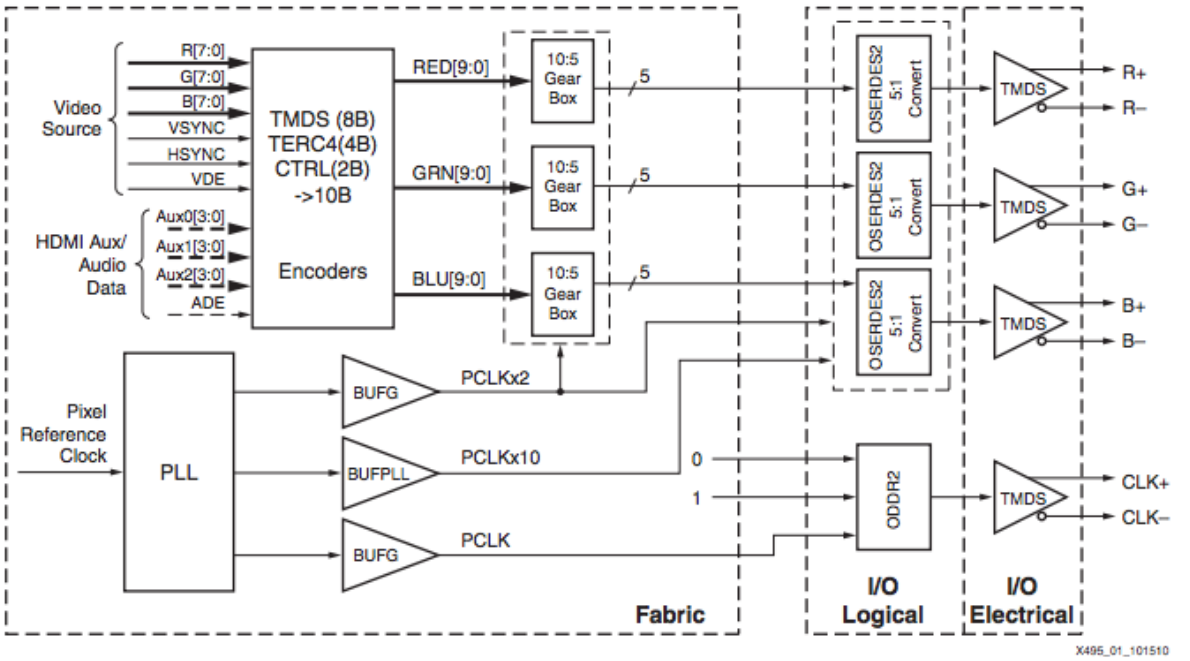


Figure 3.17: Overview of the TMDS transmitter design. Taken from the Xilinx application note 495 [15].

3.8 Video Input

This section describes the effort that was put into getting camera input for the system (functional requirement FR3). Throughout the project period the approach to this was changed multiple times as more information about the hardware was discovered.

3.8.1 Raspberry Pi Camera

After some consideration it was decided that the *Raspberry Pi Camera Board*[1] (PiCamera) would be used as the camera module for the system. The circumstances leading to this choice are discussed in Section 4.3.1.

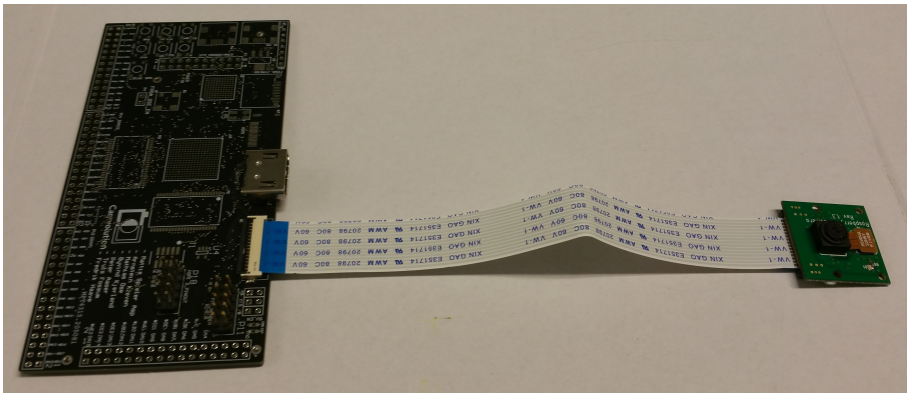


Figure 3.18: Raspberry Pi Camera connected to Camvolution board

The *Raspberry Pi Foundation* has designed this peripheral for use with the *Raspberry Pi* computer. The PiCamera can take still shots as well as record continuous video. The module consists of a camera sensor on a board with a controller unit which connects to the Pi (or another master device) via a 16-pin ribbon cable. Communication over this cable is defined by the proprietary *MIPI Camera Serial Interface* (CSI) specification.

The master device controls the camera module by sending instructions over an I2C bus on the ribbon cable, and the camera module responds with picture data over two clocked differential busses [13]. Parameters that may be controlled include video encoding, resolution in two dimensions and framerate.

3.8.2 SD Card Video

One possible source of video is a file stored on the SD card connected to the board. The MCU is able to open files and pass the data from it to the processor over EBI. For simplicity of development the MCU should not need to handle encoded video. Instead, the video files should contain raw pixel data so that it can easily be streamed to the processor.

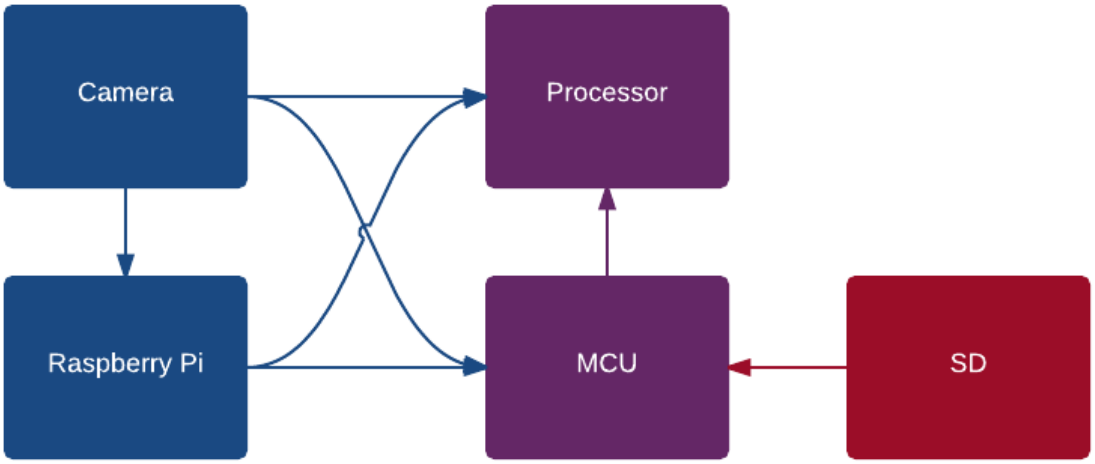


Figure 3.19: Paths an input video stream could take to reach the processor

3.8.3 Options for Video Path

Because the group had no experience with video input, there was a lot of uncertainty of how to transfer video from the camera to the processor. It was decided to make both video camera and data from SD card possible sources, with the possibility to switch between them by using the buttons on the board. Figure 3.19 shows an overview of the possible paths a video stream could take from the camera, and some options that were explored are briefly outlined below:

PiCamera Connected to the Board, Controlled by the MCU
Wiring the camera connector to the MCU, having software on the MCU control the camera and receive the video stream before forwarding it to the FPGA.

PiCamera Connected to the Board, Controlled by the FPGA
Wiring the camera connector directly to the FPGA, having an FPGA sub-module control the camera and receive the video stream.

PiCamera Connected to a Raspberry Pi, Transfer Over With Bus
Having the Raspberry Pi control the camera using the provided libraries. Transfer video stream to Camvolution using a parallel bus controlled by GPIO software.

PiCamera Connected to a Raspberry Pi, Transfer With SPI
Having the Raspberry Pi control the camera using the provided libraries. Transfer the video stream to Camvolution using built-in SPI hardware.

SD Card Video
Instead of using the camera, have a recorded video stored on the SD card which the MCU can forward to the FPGA.

[1] `https://www.raspberrypi.org/products/camera-module/`

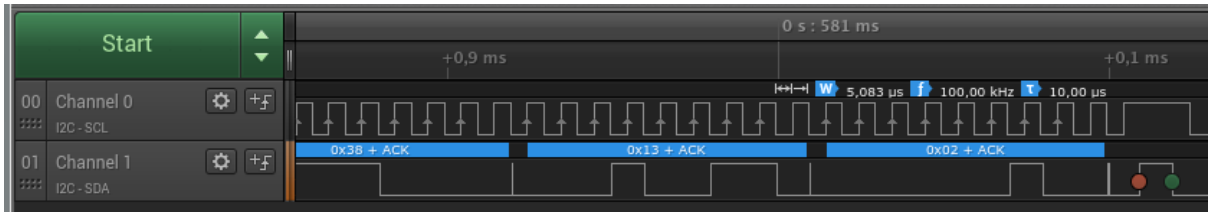


Figure 3.20: I2C communication between Raspbery Pi and PiCamera

HDMI Input

Instead of using the camera, use the second HDMI port on the board for input from any HDMI source.

3.8.4 Reverse-Engineering the Camera Control Protocol

The Raspberry Pi controls the PiCamera by sending signals over a two-wire serial bus (I2C). In order to make use of the PiCamera, it is necessary to reimplement the control system. Since there is little documentation on-line, this must be reverse-engineered. An attempt was made to use a logic analyser to "listen in" on the communication between the Raspberry Pi and PiCamera. A screenshot of this can be seen in Figure 3.20.

3.8.5 Using the Raspberry Pi To Control the Camera

Another possibility is to drop using the PiCamera connected to the board, and instead use the Raspberry Pi with its provided software to control the camera. The video data can then be transferred from the Pi to Camvolution using one of the possible outputs of the Pi, including parallel GPIO, SPI or HDMI.

4 | Results & Discussion

4.1 Testing

In this section the tests performed to verify the functionality of the different components that makes up the system, and also some of the connections between them, is presented. Some tests were performed on the physical hardware, while others were pure software simulations.

4.1.1 FPGA

The SRAM and EBI managers were tested using VHDL test benches before being implemented on the FPGA.

For the processor core, Chisel test benches were used, as testing is quite well supported by the language itself. The processor core was tested using VHDL test benches near the end of the procjet, but the tests failed with invalid values being reported by the simulation tool.

EBI Throughput

During the project, the throughput over EBI between a EFM MCU and a Spartan-6 FPGA was tested by sending a sequence of increasing numbers which were verified on the FPGA. A set-up time of one cycle was required in addition to the mandatory cycle always introduced by the MCU EBI controller for data to be transferred correctly.

Since the MCU is running with a clock speed of 48MHz, this resulted in a maximum throughput of $16\,\text{bits} \cdot 48\,\text{MHz}/2 = 384\,\text{Mbps}$. From Table 2.3, it can be concluded that this is sufficient to accommodate all options listed except the most data intensive.

During the testing, a default FIFO queue implementation from Xilinx was used to cross the clock domain. As EBI is asynchronous and therefore does not provide a clock, the last value sent was not received on the FPGA. As a workaround, a final *don't care* value was sent over the bus to push the final data value through the FIFO at the FPGA.

SRAM Throughput

During SRAM testing, several clock rates were tried to determine the speed achievable to SRAM. After some binary search through the frequency spectrum, 49 MHz was found to be the highest working frequency. This gives a throughput of $49\,\text{MHz} \cdot 16\,\text{bit} = 784\,\text{Mbps}$

for reading. During writing however, this number must be halved as a write requires two cycles (one for address set-up and one for writing), giving a throughput for writing of 392 Mbps.

4.1.2 Hardware

After soldering the board, each component was singled out and tested to ensure proper functionality.

SRAM Connection

The resistance between each SRAM pin and the corresponding header pin was measured to ensure all wires were connected properly. It was discovered that two of the address pins for the second memory chip had been exchanged, but it was concluded this must be due to a swapping of the labels on the board, not the actual wires. Nevertheless, this would not make a difference even if the wires were swapped as the data would be swapped twice, first when writing, then when reading, resulting in the correct data being read.

The chips were confirmed to be working by configuring the FPGA with an image that performed the following steps:

1. Iterate through all memory addresses and write the lower 16 bits of the address XOR some random values (in our case, 0xDEAD and 0xBEEF for SRAM1 and SRAM2 respectively).
2. Start over at the lowest address, read the data and compare with the expected value.
3. Light the LED with a light characteristic dependent on the result:
    - Fixed on success.
    - Occulting on SRAM1 failure.
    - Flashing on SRAM2 failure (if SRAM1 was successfully read).

After fixing some malfunctioning solder points not detected during initial measuring, the test passed.

Flashing the MCU

The ARM Debug header on the computer was connected to the corresponding header on an EFM32GG development kit. The MCU was successfully flashed, and verified by inspecting variables in the debug mode of Simplicity Studio.

SD Card

Reading and writing to the MicroSD card through the MCU was tested by reading a small file from the card into a buffer and inspecting it with the debug mode of Simplicity Studio. It turned out the wires for the connector had been flipped, but the test passed after rotating the SD card slot 180° to make the connections align.

Configuring the FPGA

The FPGA was configured both using JTAG and through the MCU with a bit file. The file contained a configuration for the FPGA that would allow the on-board LED to be controlled using one of the buttons. The button controlled the LED after configuration, thus it was concluded that the configuration was successful.

HDMI Output

HDMI output was confirmed to be working by configuring the FPGA with bit files outputting an image generated on the fly. Several images were tried, some filled with a single colour, and some with a repeating gradient pattern.

The image showed up on a connected display, thus it was concluded that the HDMI output was working.

4.1.3 HDMI Input

When trying to configure the FPGA for receiving HDMI through the second port it was discovered that the pins used for the incoming clock signal was connected to a non-clock input pin. Ironically, all other pins used for the given HDMI connector were clock pins, and a regular HDMI cable was opened and one of the colour wire pairs exchanged with the clock pair.

HDMI input was confirmed to be working by forwarding the HDMI input to the HDMI output.

4.2 Results

Figure 4.1: An image of a flower is sent from a laptop using HDMI. The image is convolved by the Camvolution board and sent to a screen.

In this section, the results of the groups work is presented. All the individual components worked when tested independently and a demonstration bit file assembled with a slimmed down version of the convolution processor successfully outputted processed data, bypassing both EBI and SRAM completely. This is presented in more detail below.

4.2.1 Convolution in Software with Chisel

Before the Camvolution system was implemented in hardware, the convolution engine was tested and found to correctly perform convolution in Chisel software tests.

(a) Original image (b) After convolution

Figure 4.2: Convolution performed in software simulation with Chisel

Figure 4.3: The delayed data stream shows up as a shifted image. The laptop displays the original input signal and the screen behind displays the output. A 3x3 matrix of ones is used as kernel.

4.2.2 HDMI Timing

When sending HDMI data directly from input to output without delaying the signal, the control signal from the input HDMI signals can be used as output control signals. This results in perfect synchronisation with the top left corner of the input being displayed in the top left corner on the screen.

When processing the data, the data is delayed and arrives too late compared to the control signals. This results in a image shifted slightly to the right as can be seen in Figure 4.3.

4.2.3 Slimmed Down Processor

The processor implemented in the final design does not do true convolution as the Chisel design gave no output on the display when used to configure the FPGA. Due to time constraints, the bug was never found, but a simpler version of the processor, without the input and output handlers, was found to be working.

As the input and output handlers take care of reordering the input data before performing the convolution, the data is fed through the processor in an incorrect order. In practice this results in the convolution being applied to 9 consecutive pixels instead of a 3x3 square. This can also be seen in Figure 4.3 as a yellow regions between the red and blue squares in the output image. Notice that this does not happen in the vertical direction as the image is sent through the processor in a row-major order.

Visible in Figure 4.4 are vertical lines not present in the input. The lines are caused by the processor being designed to run some rows twice since a sweep of nine input rows only produces seven output rows. As the input handler is not included, the rows are never repeated and thus these pixels are missing. In addition, the missing pixels appears

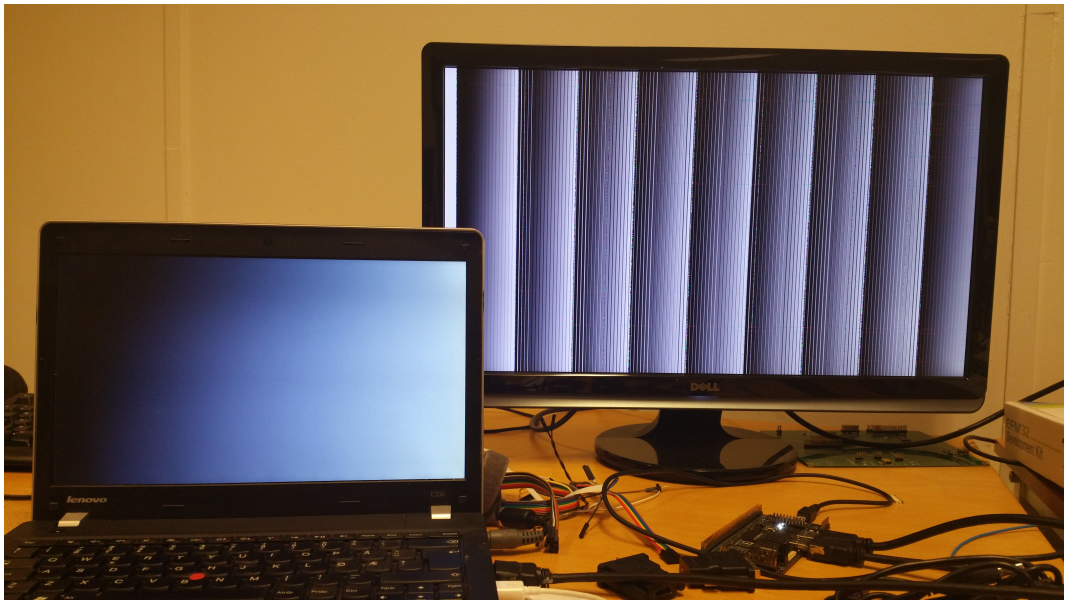


Figure 4.4: The laptop displays the input image, the display behind displays the output. The original gradient is turned into several shorter periods due to overflow when accumulating the final values. A 3x3 matrix of ones is used as kernel.

as vertical lines as the processor core expects columns and not rows.

Also missing from the slimmed down version of the processor is the possibility of setting convolution kernel and operators on run-time. Instead, several versions of the bit file was stored to the SD card for testing. Buttons on the MCU were set up to load the next bit file when pushed.

4.2.4 Overflow

Another artefact in the design is especially visible when gradients are displayed, as demonstrated in Figure 4.4. As the data width is the same through the entire processor, the values generated when adding several pixels together exceeds the maximum value that can be represented by the data format used.

These values wrap around and is offset by the maximum value, causing an otherwise wide spectrum to be split into several spectrum with the same dynamic range as the original.

4.2.5 HDMI Channel Path

During the first test of HDMI output from HDMI input, the HDMI input pins for R and B were swapped in the FPGA constraints to produce a picture with correct colour values. This led to the HDMI channel path for R and B through the FPGA to be remapped twice as shown in Figure 4.5.

The image will have the correct colours at the output, but since VSYNC and HSYNC signals are only sent over channel B, information used to synchronise the image is lost.

This caused our attempts to generate the correct output signals to fail, but by swapping the corresponding output channels instead of the input channel, a picture with correct

colours, as well as VSYNC and HSYNC signals, appeared on the display.

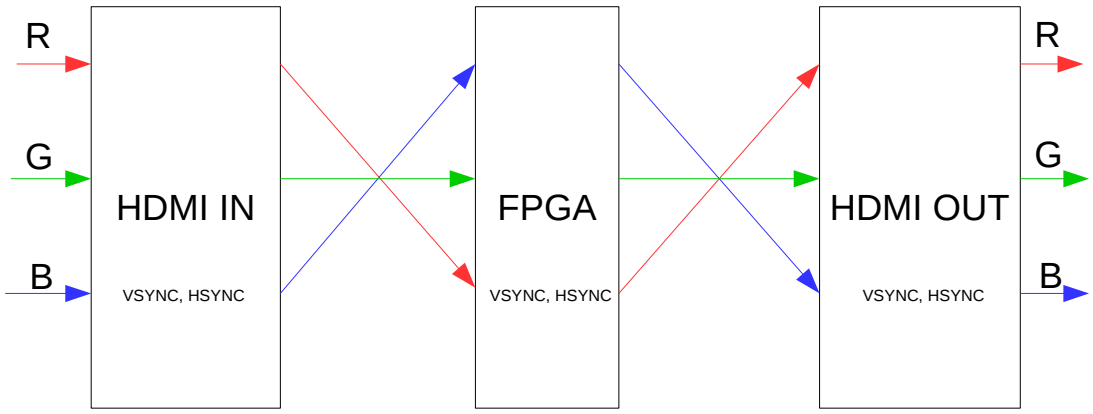


Figure 4.5: HDMI channels being swapped in the FPGA constraints, VSYNC and HSYNC is lost

4.2.6 Scalability

As stated in Section 3.6, the processor has several parameters, some of which are hard coded into the final design. These parameters can be adjusted, but many are limited by the available hardware.

In the final implementation the kernel values, map operands and reduce operands were hard coded for each configuration file. This does not scale well with multiple configurations, as a small change in any of these parameters would need its own configuration file. The time it takes for the MCU to program the FPGA with a new configuration file is about 3 seconds.

In the final implementation HDMI frame synchronisation signals are being passed from input to output without going through the processor. This means that the resolution of the image output is dependent on the input.

The size of the kernel matrices are hard coded both in the final design and in the final implementation. To use larger kernel matrix sizes, it is possible to generate new configuration files capable of processing these.

According to the reports produced by Xilinx ISE during building of the FPGA bit file, the design for 3x3 convolution utilised around 16% of the total number of available slices.

Since each component in the core of the processor only relies on its neighbours the processor can tackle large kernels without sacrificing clock speed. The limiting factor is fast memory since data has to be buffered in order to be delivered as sweep slices.

The modularity of the processor allows it to be duplicated, letting as many cores as feasible work in parallel since convolution has no data dependency issues. On our available hardware block ram and input bandwidth was the limiting factor, however on different

hardware we believe instantiating many cores with different parameters should be a simple task.

4.2.7 Video Throughput

The highest correctly working resolution was found to be 1366x768. This worked in real time with no perceivable delay. 1600x900 was also tried and found to be working, but resulted in an unexpected image with more artefacts. Finally, 1920x1200 was tried, but no sensible output was detected by the display.

4.3 Discussion

4.3.1 Camera Input

In the early stages of the project, we looked at various camera modules on-line and tried to assess how they would work in our system. Some modules were very simple, and some had many features. The PiCamera was the most feature-complete module we looked at. When we looked into the camera control protocol for the PiCamera, we found *some* information about it, like the open source C and Python camera control libraries for the Raspberry Pi and the pin-outs for the ribbon connector. From what found, it was assumed that it would be possible to implement camera control with the MCU or FPGA.

When the time came to make a choice for the camera module and order it, the PiCamera was the only option that was available from the retailers we were using, so the choice was easy.

Had we looked for more information about the PiCamera on-line before this, we would have perhaps found the Raspberry Pi engineer recommending not using the PiCamera for an FPGA project[1] or the unclaimed $1000 bounty for an open-source driver for the PiCamera[2]. This would have indicated that we should have looked for another camera module.

MCU as Camera Controller

The first option explored was to have the MCU be the camera controller and receive the video stream. This idea was abandoned eventually as the EFM32GG does not have hardware to receive the differential bus data, so the FPGA would be more appropriate for this role.

FPGA as Camera Controller

After aquiring a snapshot of the I2C communication between the Pi and the PiCamera with a logic analyser, a VHDL module was written that could output the same sequence, and this was tested with the Spartan-6 development board and the PiCamera. While the FPGA was able to output the I2C sequence, the camera would not send acknowledgements for any of the transmissions. Efforts into figuring out why the PiCamera wouldn't respond were fruitless, so the idea was eventually abandoned.

Using the Raspberry Pi Since reverse-engineering the camera control protocol was difficult, we tried to use a Raspberry Pi to control the camera, then transfer the video stream to the Camvolution board. The provided software on the Pi makes it very easy to have full control of the camera, but transferring video out of the Pi at high rates is more challenging.

[1] `https://www.raspberrypi.org/forums/viewtopic.php?t=119977&p=812191`
[2] `https://parallella.org/forums/viewtopic.php?f=10&t=2514`

Figure 4.6: At 7 FPS recorded, the SPI transmission is unable to keep up and sends the seventh frame too late.

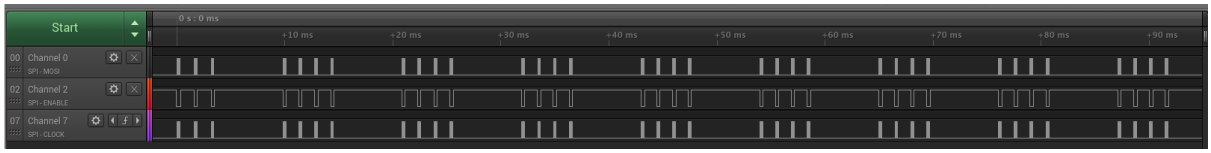


Figure 4.7: Inspecting a single frame transmission closer reveals a lot of idle time.

GPIO Pins The Raspberry Pi has GPIO pins that are controllable in software. A script was written which could output pixel data to the GPIO pins as a parallel bus with one clock pin and eight data pins. This was very simple to implement, however it was also very slow. With a Raspberry Pi 2 the best clock frequency achieved on this bus was approximately $600kHz$. This would only allow sending very small frames of video at low frame rates.

The Raspberry Pi also has hardware support for SPI over some of the GPIO pins. Directing the video stream to this instead was more promising, since the frequency of the clock in this mode would be able to go up to $32MHz$[3], which would be a massive improvement even if the transfer was serial instead of 8-bit parallel. Unfortunately what was observed was that while the transfer of chunks of data was fast, there were long delays between chunks. At 240x240 resolution, 6 frames per second was the best transfer rate that was able to keep up with the recording. Figures 4.6 and 4.7 shows the logic analyser output for SPI transfers.

HDMI Output Seeing that all efforts into using the camera were either failing or at best offering a tiny stream of data, we decided to concentrate on the HDMI input of the Camvolution system. The Raspberry Pi has HDMI out, so it would definitely be possible to use the PiCamera with this set-up.

4.3.2 Image Compression

In the final implementation of the system the processor receives its input from HDMI exclusively. If the processor were to receive video from an SD card as described in Section 3.8, the bottleneck of the system would not be the image processing itself, but ensuring that the processor has sufficient data to work on.

[3] `http://elinux.org/index.php?title=RPi_SPI#Speed_2`

There are several different ways that can reduce this bottleneck. First of all the dimensions could have been increased, that is, the width of the buses increased. On the EBI bus the address pins and the chip select signals could be used for data, which yields a 38-bit wide data bus, but at the same time loses addressing and DMA functionality. Second, the clock rate could have been increased. This is not easy to do on the MCU as it already uses its maximum specified clock frequency. The processor implementation was assumed not to be a limitation, as it always can be pipelined more until it reaches the limit of what the FPGA can manage.

A third, less obvious option, is to compress the data. We could have employed both lossy and lossless encodings to reduce the size of the data and thus increase the effective throughput. A simple encoding could be to map 8-bit values sent from the MCU to 24-bit colours on the FPGA, which reduces the amount of data sent to a third of the original data. This can be viewed as a form of lossy compression.

More advanced methods can be applied, but these would obviously require more work. In addition, there is no guarantee the MCU can do more advanced compression at the rate required. A faster solution might have been to store encoded video files on the SD card, read them on the MCU with DMA, send the data to the FPGA and let it handle the decoding of the video.

4.3.3 EBI and SRAM Throughput

As stated in Section 4.1.1, a throughput of 384 Mbps was achieved over EBI. This must be considered to be an absolute maximum as there is no guarantee the MCU is capable of delivering this amount of data anyway, but since the final design did not utilise EBI for video transfers, the throughput over EBI is not a limiting factor.

When it comes to SRAM, the throughput measured in Section 4.1.1 is limited by the achievable write speed. Even though the chip has an access time of 10 ns, writing requires a write pulse width of minimum 8 ns, and since the computer uses a duty cycle of 50 %, the real lower bound on a write is 16 ns. Thus the specifications for the SRAM chips guarantees a write speed of $(2 \cdot 8\,\text{ns})^{-1} \cdot 16\,\text{bits}/2 = 500\,\text{Mbps}$ using the implemented write cycle.

The speed being lower may be due to latencies in the FPGA as the signals have to propagate from the registers, through logic and some multiplexers before reaching the bus and finally the SRAM chips themselves. In addition, more experimenting with the write and read cycles could probably have improved the clock rate slightly giving a slightly higher throughput.

4.3.4 Design Faults

In this first revision of the PCB, a few design faults were made, and some small hacks had to be made to compensate.

Voltage Regulator

The first flaw encountered was the voltage regulator that was connected to ground, but through a 100Ω resistor and a capacitor. This forced the voltage output up to 4.15V and may have bricked the FPGA that has a voltage maximum at 3.95V. The reason for this flaw was a failure to read the full specifications in the data sheet.

SD Card Connector

After mounting the SD Card connector we found that all the pin-outs on the SD card were inverted. The easiest solution was to the turn the SD Card connector 180° and mount part of it outside the board.

MCU Clock

There was a fault in the design of the 48MHz external clock pads for the MCU. This was due to failure of wiring the pads to the right pins on the external crystal. To fix this a crystal with a smaller surface and only two pins was bought. This flaw could have been avoided had the PCB been printed on an A4 sheet, and the component sizes verified using real components. Unfortunately, the components were not available at the time of ordering the PCB.

Decoupling Capacitors

We used an 0603 footprint for all capacitors and resistors. We then had to change to a bigger 0805 or 1208 footprint when mounting a capacitor larger then 10μF. A capacitor at 10μF does not exist in an 0603 footprint. In the heat of the last few days, the PCB team forgot about this, and therefore the board does not have 100μF decoupling capacitors on the PCB.

G_CLOCK Problems on Input Because of later problems with the PiCamera we had to look for other input solutions. The HDMI input clock needs to be connected to G_CLOCK pins on the FPGA. Unfortunately, only TDMS 0, TMDS 1 and TMDS 2 had been connected to G_CLOCK. Because this wiring is impossible to change afterwards, an HDMI cable was opened and the cable's cable's internal pins were switched to make it work. Only HDMI input requires the clock to be mapped to a G_CLOCK pin, thus this was not discovered during the testing that had been done before the PCB had been ordered.

Data Communication Between Devices HDMI also has an I2C interface that it uses to communicate its display data [12]. This includes supported devices, monitor parameters, etc. This is not required for HDMI video transfer to actually work, but on most computers the HDMI will not output anything unless you send in display data or force output. To make this job easier, routing could have been added to the I2C pins on the HDMI connector.

4.3.5 FPGA Resource Utilisation

The low resource utilisation on the FPGA suggests the architecture can be used for larger kernels or can be optimised for higher speed at the cost of more hardware resources. This is in addition to the higher throughput that can be achieved by clocking the processor faster, which means higher quality video can be processed in real time.

In addition to increasing the video quality, the size of the kernel can also be increased. As the area of the kernel increases by the square of the kernel size, the hardware requirements are also expected to increase quadratically, if not faster.

4.4 Further Work

Unfortunately, this project has a limited timeframe. Had we had more time, we would have prioritised working on the following:

4.4.1 Camera

In order to achieve camera input as originally intended, we would go back to some of the other camera modules we looked at, and find a more suitable module. It would probably to be a good idea to try a module with less features that is simpler to control.

4.4.2 PCB

We would have fixed the errors on the PCB related to HDMI signals and the SD connector.

We would also add more LEDs, to make it easier to debug. In addition it would be nice to support a USB interface to the MCU for development and debugging.

4.4.3 MCU

The EBI described in Section 3.4, was not implemented at the time of the deadline. When this is implemented, it will no longer be necessary to reset the FPGA configuration every time the processor should be configured.

Because of the HDMI working properly as an input source, and further work on implementing a camera module, streaming video from the SD card would have a low priority.

If the MCU's only task is to configure the FPGA and the processor, the amount of communication between them is minimal and a serial interface could have been used instead of the EBI bus. This would have siginficantly reduced the amount of connections between the MCU and the FPGA on the PCB.

This is not a large problem and it would be preferrable to keep the wide EBI bus when implementing the graphical user interface functionality. This way the MCU is able to bypass the processor and write the graphical user inteface directly to the video memory.

4.4.4 FPGA

Most importantly the Convolution feature of the system should be made to work as intended. As mentioned, 2D convolution worked in Chisel test benches, but not when implemented on the FPGA. We were not able to get to the bottom of why this was by the deadline, but given more time this should be possible.

4.5 Conclusion

The final processor performs a simplified form of convolution on a live video stream. This does not quite satisfy the first functional requirement, but works as a proof of concept and shows that real convolution is possible given more time. The fact that the processor was shown to be working in software simulation makes this even more likely.

The computer accepts input over HDMI from any device, including a camera, and can deliver output to a display over HDMI. In addition, it loads the FPGA configuration from an SD card and can load other configurations by the push of a button.

The hardware utilisation numbers means even a small FPGA can make a huge difference when accelerating convolution, which suggests an FPGA can make a big difference when doing application specific computations.

References

[1] TDT4295 Computer Design Project - Assignment Text. ItsLearning, 2015.

[2] Song Ho Ahn. Convolution. http://www.songho.ca/dsp/convolution/convolution.html, 2014.

[3] Alliance Memory. 512K X 16 BIT HIGH SPEED CMOS SRAM, 2 2012. http://www.alliancememory.com/pdf/sram/fa/as7c38098a.pdf.

[4] Douglas Brooks. Differential Signals - Rules to Live By. http://www.ieee.li/pdf/essay/differential_signals.pdf, 2001.

[5] Diane Bryant. Disrupting the Data Center to Create the Digital Services Economy. https://communities.intel.com/community/itpeernetwork/datastack/blog/2014/06/18/disrupting-the-data-center-to-create-the-digital-services-economy, 2014.

[6] Design Automation Conference. Chisel: Constructing Hardware in a Scala Embedded Language, 2012.

[7] Scott Hauck and André DeHon. Systems on Silicon : Reconfigurable Computing : The Theory and Practice of FPGA-Based Computation, chapter 21.1. Morgan Kaufmann, 2008.

[8] Silicon Labs. FAT on SD Card AN0030 - Application Note. https://www.silabs.com/Support%20Documents/TechnicalDocs/AN0030.pdf, 2013.

[9] Linear77. DiffSignaling.png. https://commons.wikimedia.org/wiki/File:DiffSignaling.png, 2012.

[10] Microsoft. Overview of FAT, HPFS, and NTFS File Systems. https://support.microsoft.com/en-us/kb/100108, 2007.

[11] James Niccolai. Intel's First Processor With Performance-Boosting FPGA to Ship Early Next Year. http://www.pcworld.com/article/3006601/components-processors/intels-first-server-chip-with-performance-boosting-fpga-to-ship-early.html, 2015.

[12] Various Contributors to Wikipedia. Display Data Channel. https://en.wikipedia.org/wiki/Display_Data_Channel, 2015.

[13] Peter Vis. Raspberry Pi CSI Interface Connector Pinout. http://www.petervis.com/Raspberry_PI/Raspberry_Pi_CSI/raspberry-pi-csi-interface-connector-pinout.html.

[14] Eric W. Weisstein. Convolution. http://mathworld.wolfram.com/Convolution.html, 2015.

[15] Xilinx. Implementing a TMDS Video Interface in the Spartan-6 FPGA. http://www.xilinx.com/support/documentation/application_notes/xapp495_S6TMDS_Video_Interface.pdf, 2010.

[16] Xilinx. Spartan-6 FPGA Configuration User Guide UG380. http://www.xilinx.com/support/documentation/user_guides/ug380.pdf, 2015.

A | List of Components

Retailer	Amount	Price NOK/unit	Part Number	Name
Digikey	3	8,66	ADP170AUJZ	Voltage regulator
Farnell	3	121,35	1210018	SARONIX CRYSTAL OSCILLATOR
Farnell	20	1,5	2251493	LEDS
Farnell	10	19,5	2356206	Header pins
Farnell	100	0.0118	2367994	FPC-HEADER
Farnell	20	5.1	1593446	Header pins
Farnell	4	105,75	2103743	AS7C38098A
Farnell	2	601,95	1876230	FPGA
Farnell	2	85,52	2314200	EFM
Farnell	20	2,45	1217764	Button
Farnell	6	6,15	9778195	LM1117MP-3.3
Farnell	10	14,40	1654540	HDMI
Farnell	6	22,0	1842014	EFM crystal
Farnell	2	157,86	2302279	RPi camera
Farnell	30	1	1653253	10kΩ resistor
Farnell	30	0	1469803	330Ω resistor
Farnell	30	0	2309108	1kΩ resistor
Farnell	30	0	2112878	50Ω resistor
Farnell	30	0	1652884	4.7kΩ resistor
Farnell	30	0.126	2333587	100Ω resistor
Farnell	20	2,09	2211173	4.7μF capacitor
Farnell	20	1.55	2346908	0.47μF capacitor
Farnell	40	0	9406140	0.1μF capacitor
Farnell	20	2,35	2309028	10μF capacitor
Farnell	20	1.49	2211179	1μF capacitor
Farnell	20	13,82	2494476	100μF capacitor

Table A.1: List of components

B | Pin-outs

HDMI Pin Name	Header Pin Nr	FPGA HDMI 1	FPGA HDMI 2	External
TMDS Data2+	3	B2	B9	
TMDS Data2-	5	A2	A9	
TMDS Data1+	7	D6	D11	
TMDS Data1-	9	C6	C11	
TMDS Data0+	10	B3	C10	
TMDS Data0-	8	A3	A10	
TMDS Clock+	6	B4	G9	
TMDS Clock-	4	A4	F9	
CEC	N/C	N/C		
HEC Data (Opt)	GND	N/C		
SCL	N/C	N/C		
SDA	N/C	N/C		
DDC	GND	N/C		
5+ Power	P15 Header	N/C		
Hot Plug Detect	N/C	N/C		
HP Detect (HDMI1)	P9 Header	N/C		
GND	1			GND
VCC	2			VCC

Table B.1: HDMI Connector

Button	Connection	Pull-up/pull-down	Name
SW6	FPGA P15	Pull-up	
SW7	FPGA P16	Pull-up	
SW1	EFM PD8	Pull-up	BTN OK
SW2	EFM PD7	Pull-up	BTN Back
SW3	EFM PD6	Pull-up	BTN Down
SW4	EFM PD5	Pull-up	BTN Up
SW5	EFM K6	Pull-up	Reset

Table B.2: I/O buttons

Pin number	EFM pin	Usage	Shared with
1	PB4	I/O	
2	PA7	I/O	
3	PB5	I/O	
4	PA8	I/O	
5	PB6	I/O	
6	PA9	I/O	
7	PB7	I/O	
8	PA10	I/O	
9	PB8	I/O	INIT_B
10	PA11	I/O	
11	PB11	I/O	Program_B
12	PC0	I/O	
13	PB12	I/O	Done
14	PC1	I/O	
15	PB15	I/O	
16	PC9	I/O	
17	PD4	I/O	
18	PC10	I/O	
19	PD11	I/O	
20	PC11	I/O	
21	PD12	I/O	
22	PF11	I/O	
23	PD13	I/O	
24	PF10	I/O	
25	PF5	I/O	
26	PF7	I/O	
27	PF6	I/O	
28	3.3V		

Table B.3: Header outputs

Pin	USAGE
1	VCC
3	N/C
5	N/C
7	CS / PF0
9	CLK / PF1
11	N/C
13	PF2
15	RESET
17	N/C
19	N/C
4,6,8,10,12,14,16,18,20	GND
2	N/C

Table B.4: Programming Output for MCU

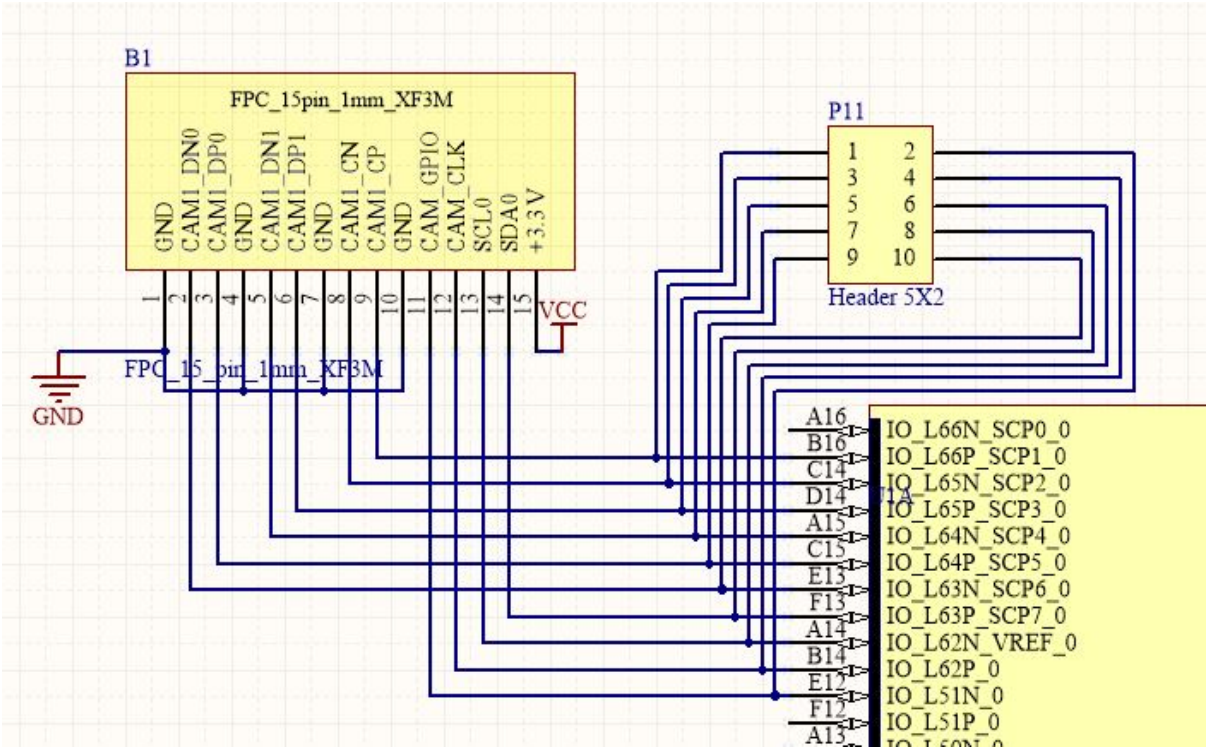


Figure B.1: Raspberry Pi connector

MCU pin	FPGA	Usage
PB8	U3	INIT_B
PD3	U15	CS
PD2	R15	CLK
PD1	V16	RX
PD0	R13	TX

Table B.5: Programming FPGA using MCU SPI

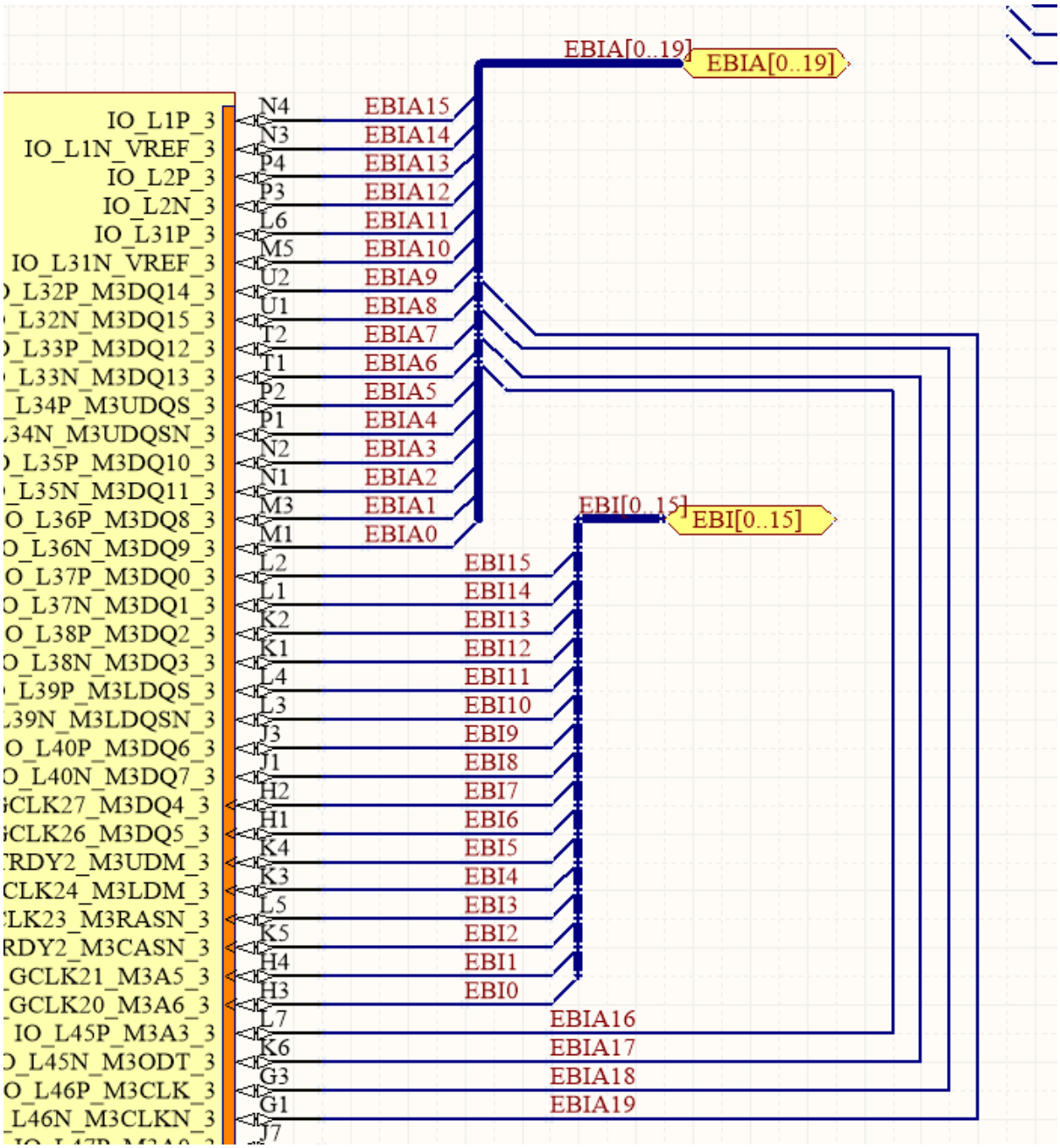


Figure B.2: EBI bus input

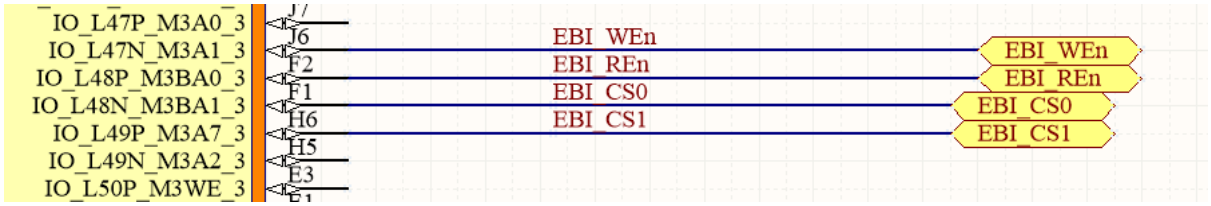


Figure B.3: EBI bus control signals

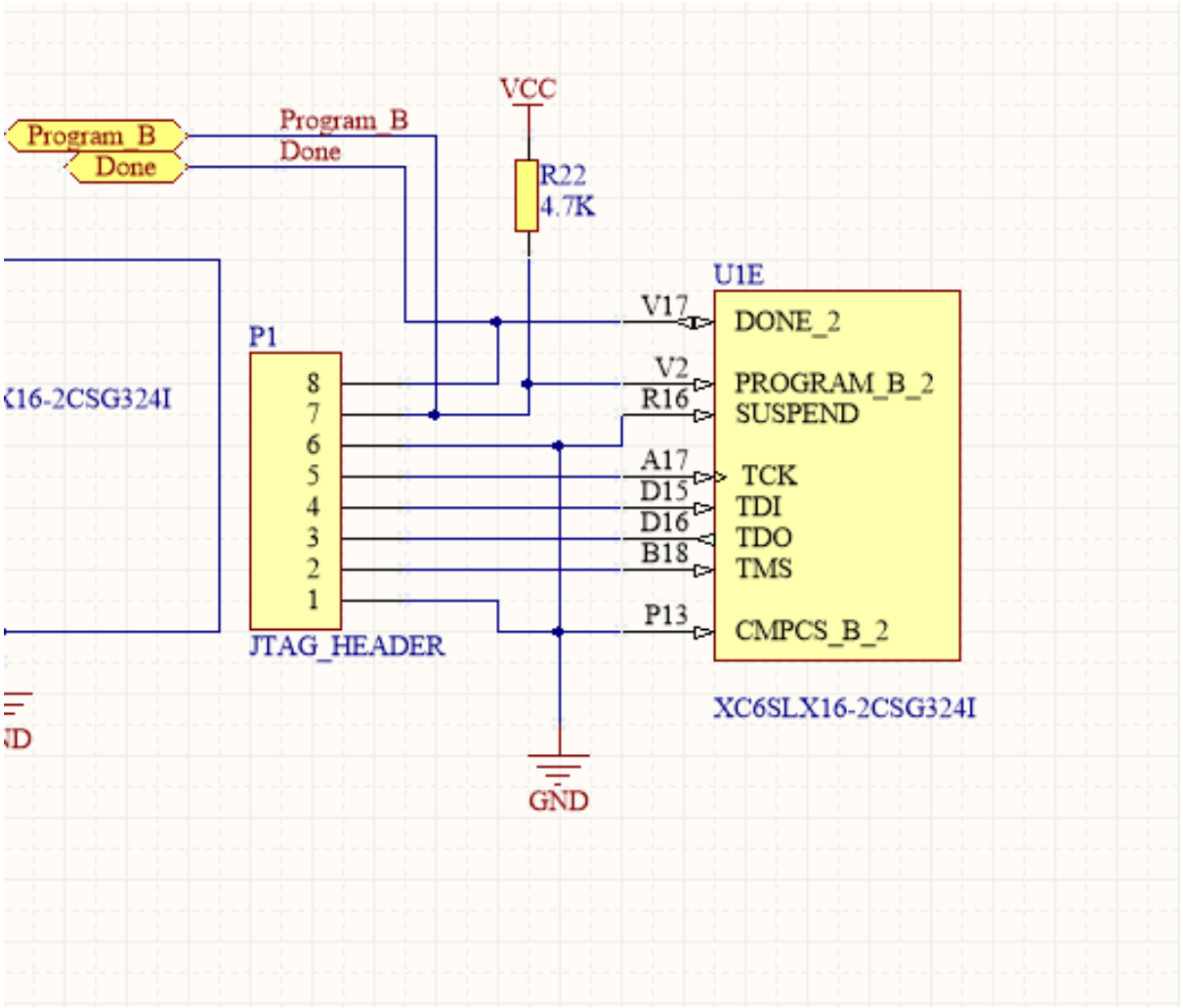


Figure B.4: Programming output for FPGA

C | Graphical User Interface Concept

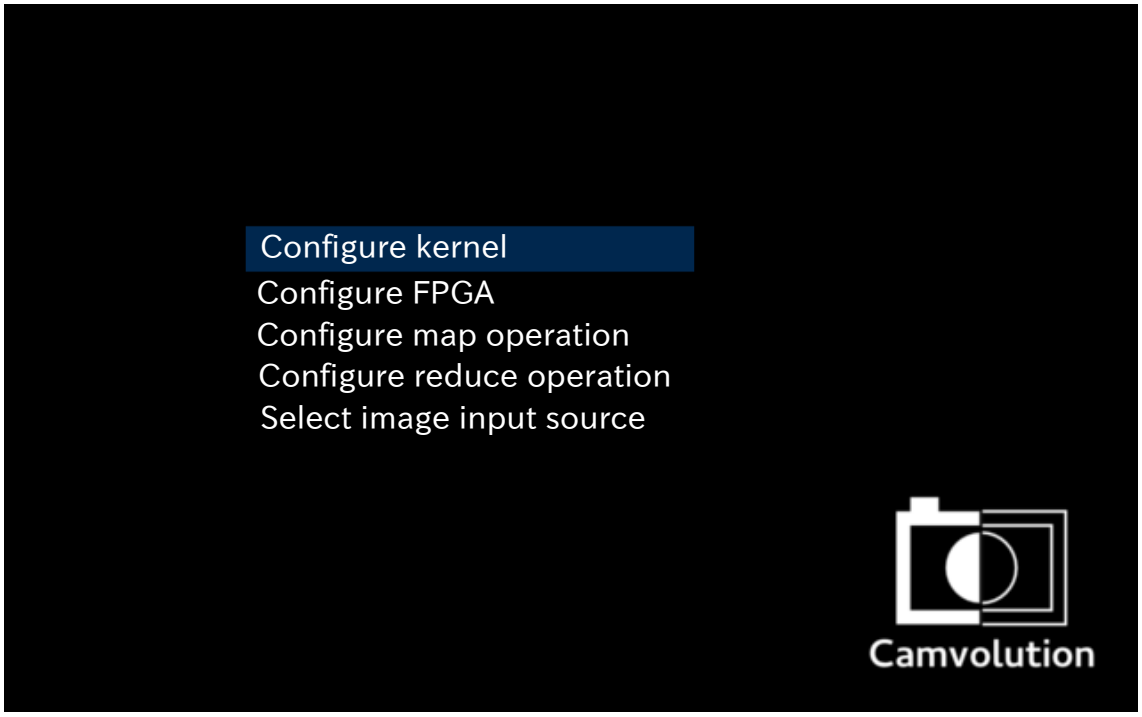
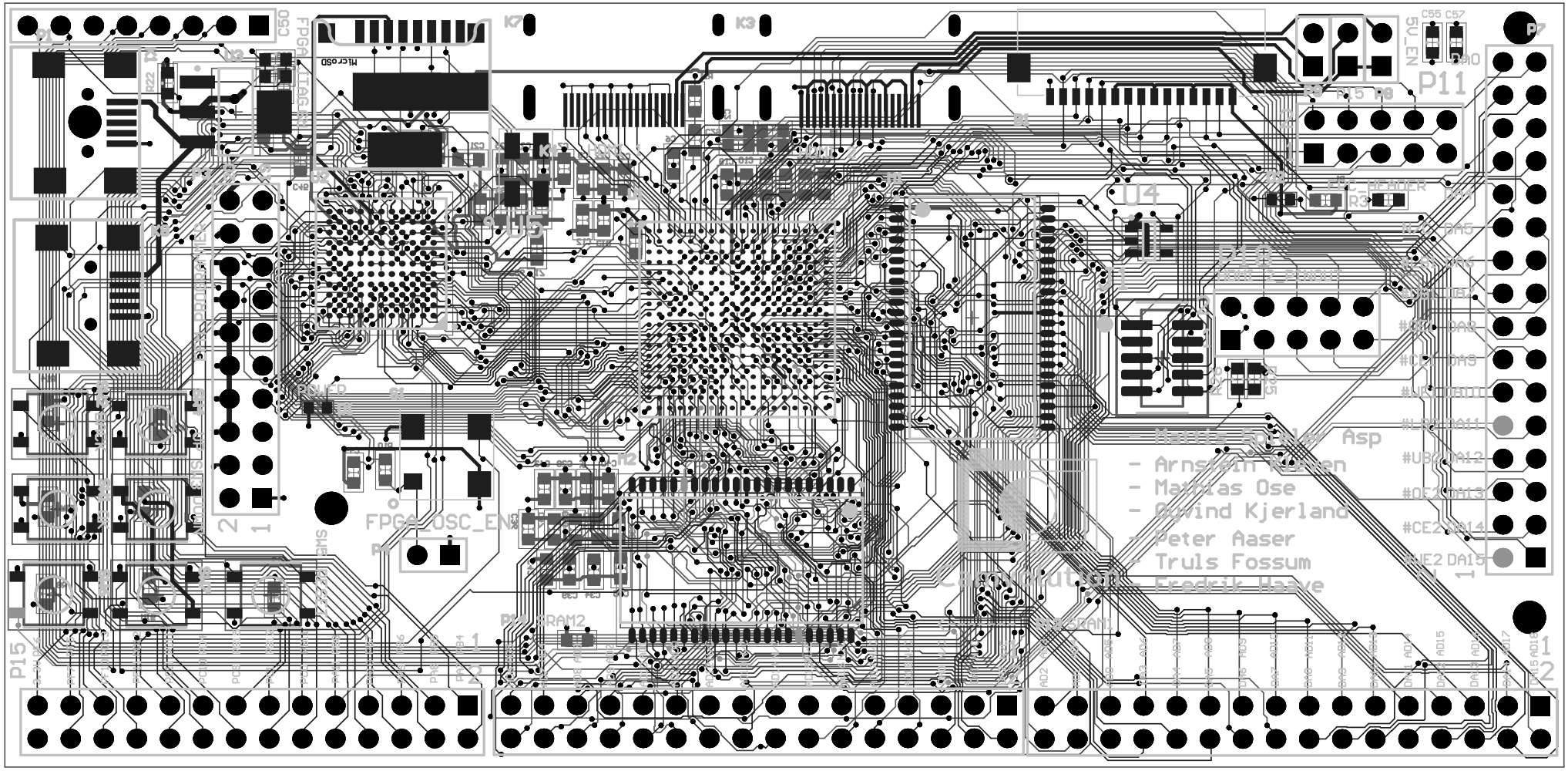


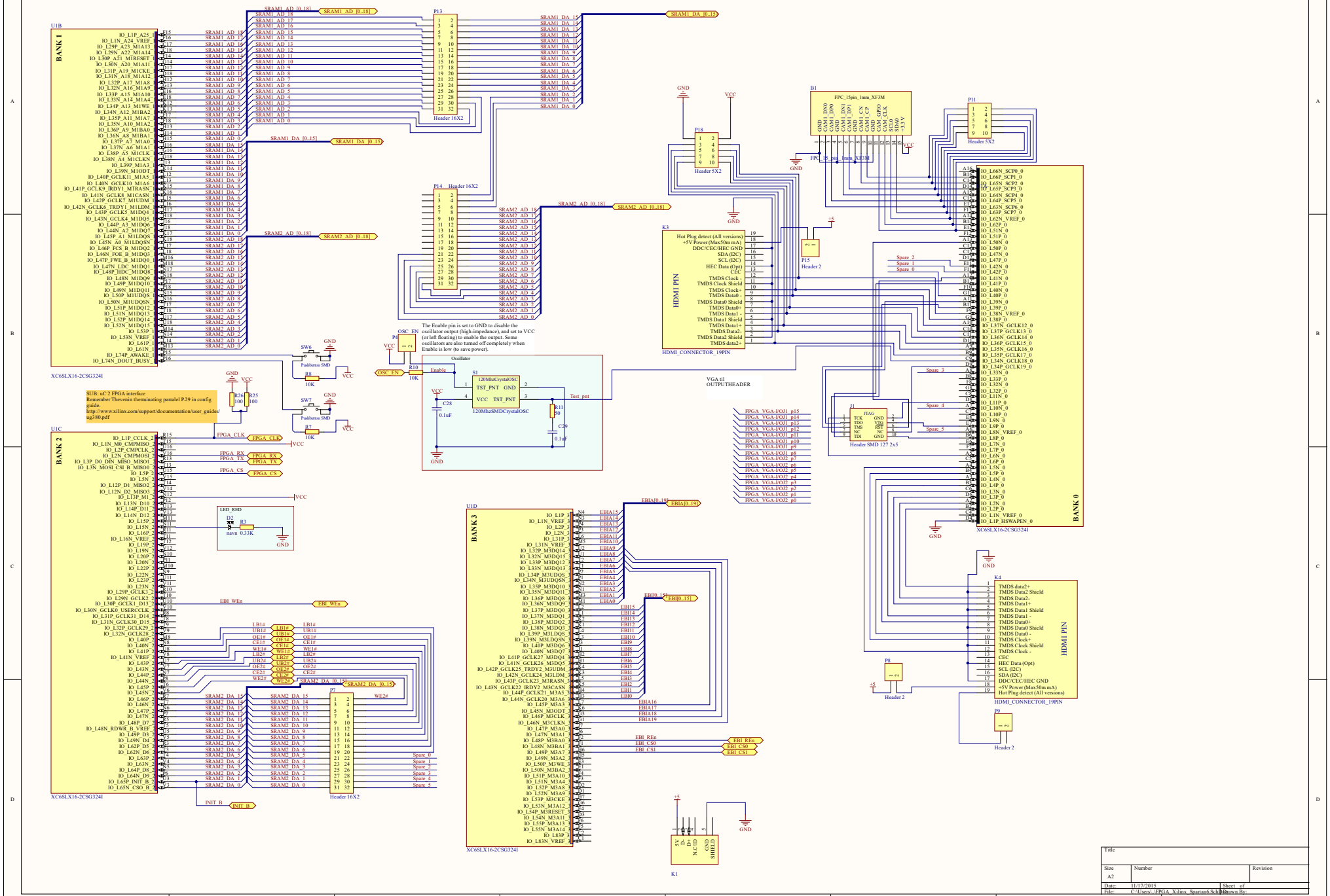
Figure C.1: Concept for a menu based graphical user interface

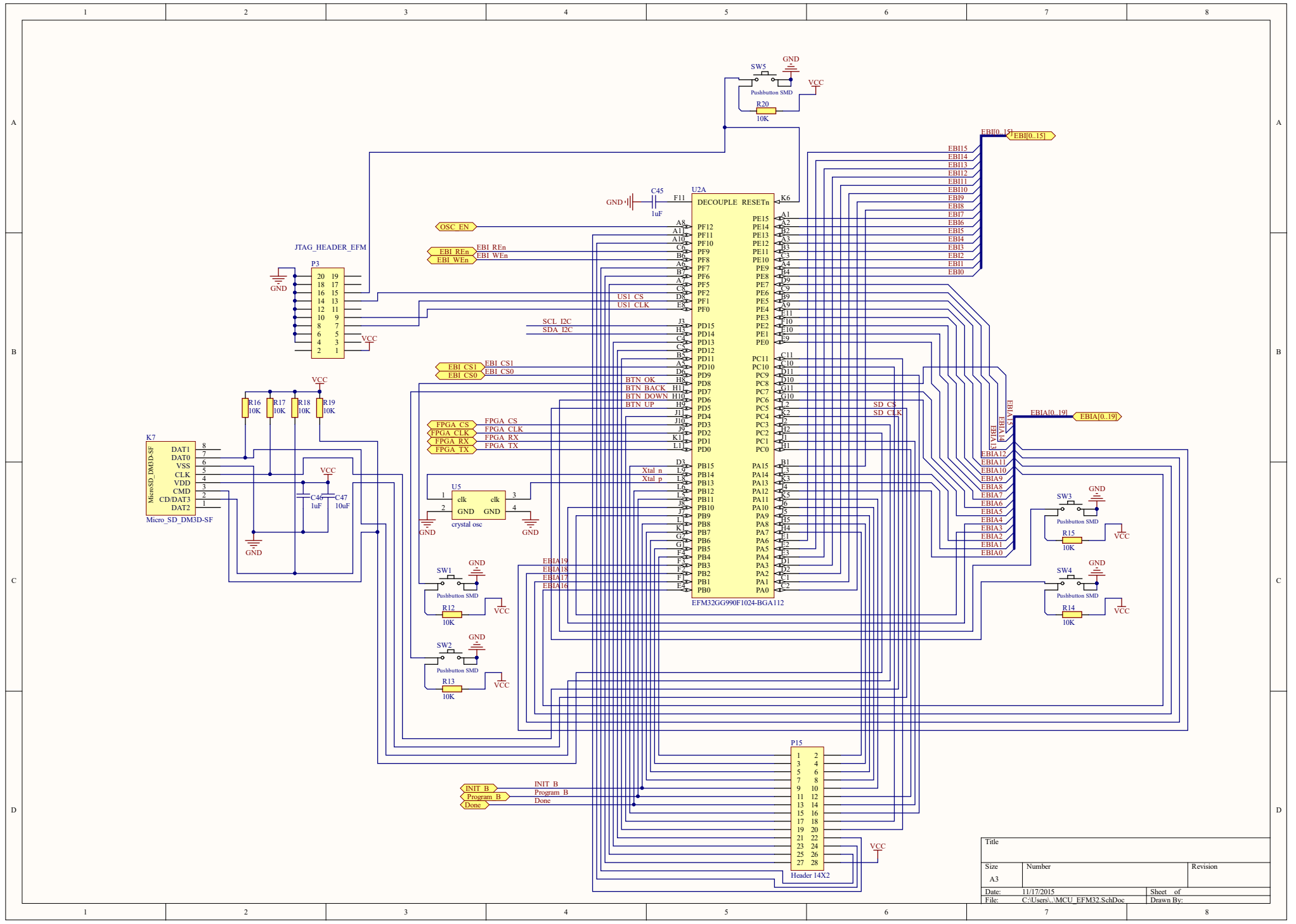
D | PCB Schematics

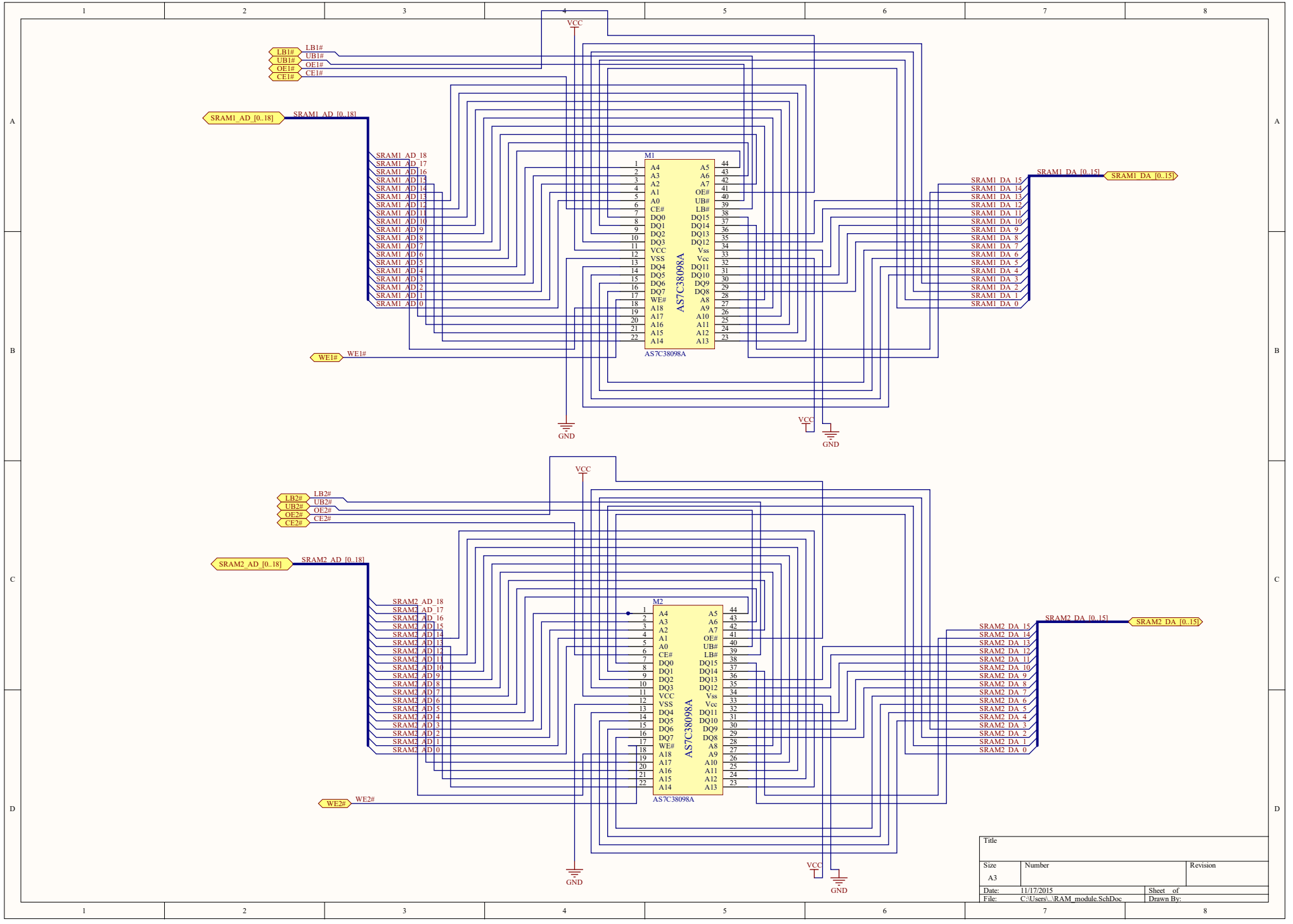
Schematic attachements follow:

1. Finished PCB
2. Xilinx Spartan-6 FPGA
3. Silicon Labs EFM32 MCU
4. RAM module


Asp
- Arnstein
- Matthias Ose
- Øyvind Kjerland
- Peter Aaser
- Truls Fossum
- Fredrik Haave


U1B
BANK 1
XC6SLX16-2CSG324I
SRAM1_AD_[0..18]
SRAM1_DA_[0..15]
P13
Header 16X2
SRAM2_AD_[0..18]
SRAM2_DA_[0..15]
P14 Header 16X2
OSC_EN
P4
Oscillator
Enable
R10
10K
S1
120MhzCrystalOSC
C28
0.1uF
C29
0.1uF
R11
50
Test_pnt
The Enable pin is set to GND to disable the oscillator output (high-impedance), and set to VCC (or left floating) to enable the output. Some oscillators are also turned off completely when Enable is low (to save power).
SW6
SW7
Pushbutton SMD
R8
10K
R7
10K
R26
100
R25
100
SUB: uC 2 FPGA interface
Remember Thevenin theminating parralel P.29 in config guide.
http://www.xilinx.com/support/documentation/user_guides/ug380.pdf
U1C
BANK 2
FPGA_CLK
FPGA_RX
FPGA_TX
FPGA_CS
LED_RED
D2
R3
navn 0.33K
EBI_WEn
LB1#
UB1#
OE1#
CE1#
WE1#
LB2#
UB2#
OE2#
CE2#
WE2#
SRAM2_DA_[0..15]
P7
Header 16X2
Spare_0
Spare_1
Spare_2
Spare_3
Spare_4
Spare_5
INIT_B
U1D
BANK 3
XC6SLX16-2CSG324I
EBIA[0..19]
EBI[0..15]
EBI_REn
EBI_CS0
EBI_CS1
B1
FPC_15pin_1mm_XF3M
FPC_15_pin_1mm_XF3M
P18
Header 5X2
P11
Header 5X2
K3
HDMI1 PIN
HDMI_CONNECTOR_19PIN
P15
Header 2
Spare_2
Spare_1
Spare_0
Spare_3
Spare_4
Spare_5
J1
JTAG
Header SMD 127 2x5
FPGA_VGA-I/OJ1_p15
FPGA_VGA-I/OJ1_p14
FPGA_VGA-I/OJ1_p13
FPGA_VGA-I/OJ1_p12
FPGA_VGA-I/OJ1_p11
FPGA_VGA-I/OJ1_p10
FPGA_VGA-I/OJ1_p9
FPGA_VGA-I/OJ1_p8
FPGA_VGA-I/OJ2_p7
FPGA_VGA-I/OJ2_p6
FPGA_VGA-I/OJ2_p5
FPGA_VGA-I/OJ2_p4
FPGA_VGA-I/OJ2_p3
FPGA_VGA-I/OJ2_p2
FPGA_VGA-I/OJ2_p1
FPGA_VGA-I/OJ2_p0
VGA til
OUTPUT/HEADER
BANK 0
XC6SLX16-2CSG324I
P8
Header 2
P9
Header 2
K4
HDMI1 PIN
HDMI_CONNECTOR_19PIN
K1
5V
D-
D+
N.C/ID
GND
SHIELD
Title
Size
A2
Number
Revision
Date:
11/17/2015
Sheet of
File:
C:\Users\...\FPGA_Xilinx_Spartan6.SchDoc
Drawn By:


SW5
Pushbutton SMD
R20
10K
GND
VCC
EBI[0..15]
EBI15
EBI14
EBI13
EBI12
EBI11
EBI10
EBI9
EBI8
EBI7
EBI6
EBI5
EBI4
EBI3
EBI2
EBI1
EBI0
U2A
DECOUPLE
RESETn
C45
1uF
F11
K6
OSC_EN
EBI_REn
EBI_WEn
JTAG_HEADER_EFM
P3
US1_CS
US1_CLK
SCL_I2C
SDA_I2C
EBI_CS1
EBI_CS0
BTN_OK
BTN_BACK
BTN_DOWN
BTN_UP
FPGA_CS
FPGA_CLK
FPGA_RX
FPGA_TX
SD_CS
SD_CLK
EBIA[0..19]
EBIA19
EBIA18
EBIA17
EBIA16
EBIA12
EBIA11
EBIA10
EBIA9
EBIA8
EBIA7
EBIA6
EBIA5
EBIA4
EBIA3
EBIA2
EBIA1
EBIA0
Xtal_n
Xtal_p
U5
crystal osc
K7
Micro_SD_DM3D-SF
DAT1
DAT0
VSS
CLK
VDD
CMD
CD/DAT3
DAT2
R16 10K
R17 10K
R18 10K
R19 10K
C46 1uF
C47 10uF
SW1
SW2
SW3
SW4
R12 10K
R13 10K
R15 10K
R14 10K
P15
Header 14X2
INIT_B
Program_B
Done
EFM32GG990F1024-BGA112
Title
Size
A3
Number
Revision
Date: 11/17/2015
File: C:\Users\..\MCU_EFM32.SchDoc
Sheet of
Drawn By:


VCC
LB1#
UB1#
OE1#
CE1#
SRAM1_AD_[0..18]
SRAM1_AD_18
SRAM1_AD_17
SRAM1_AD_16
SRAM1_AD_15
SRAM1_AD_14
SRAM1_AD_13
SRAM1_AD_12
SRAM1_AD_11
SRAM1_AD_10
SRAM1_AD_9
SRAM1_AD_8
SRAM1_AD_7
SRAM1_AD_6
SRAM1_AD_5
SRAM1_AD_4
SRAM1_AD_3
SRAM1_AD_2
SRAM1_AD_1
SRAM1_AD_0
M1
AS7C38098A
WE1#
GND
SRAM1_DA_[0..15]
SRAM1_DA_15
SRAM1_DA_14
SRAM1_DA_13
SRAM1_DA_12
SRAM1_DA_11
SRAM1_DA_10
SRAM1_DA_9
SRAM1_DA_8
SRAM1_DA_7
SRAM1_DA_6
SRAM1_DA_5
SRAM1_DA_4
SRAM1_DA_3
SRAM1_DA_2
SRAM1_DA_1
SRAM1_DA_0
LB2#
UB2#
OE2#
CE2#
SRAM2_AD_[0..18]
SRAM2_AD_18
SRAM2_AD_17
SRAM2_AD_16
SRAM2_AD_15
SRAM2_AD_14
SRAM2_AD_13
SRAM2_AD_12
SRAM2_AD_11
SRAM2_AD_10
SRAM2_AD_9
SRAM2_AD_8
SRAM2_AD_7
SRAM2_AD_6
SRAM2_AD_5
SRAM2_AD_4
SRAM2_AD_3
SRAM2_AD_2
SRAM2_AD_1
SRAM2_AD_0
M2
AS7C38098A
WE2#
GND
SRAM2_DA_[0..15]
SRAM2_DA_15
SRAM2_DA_14
SRAM2_DA_13
SRAM2_DA_12
SRAM2_DA_11
SRAM2_DA_10
SRAM2_DA_9
SRAM2_DA_8
SRAM2_DA_7
SRAM2_DA_6
SRAM2_DA_5
SRAM2_DA_4
SRAM2_DA_3
SRAM2_DA_2
SRAM2_DA_1
SRAM2_DA_0
Title
Size
A3
Number
Revision
Date: 11/17/2015
Sheet of
File: C:\Users\..\RAM_module.SchDoc
Drawn By: