NTNU

TDT4295 - Computer Design Project

Camvolution

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

November 16, 2015

Abstract

Contents

1 Introduction

This report present the results from the TDT4295 Computer Design Project course at The Norwegian University of Science and Technology, where groups of students are to design and create their own computers.

1.1 Project Description

1.1.1 Assignment Text

The original assignment text for the project follows.

> **A Convolution Engine-like processor.**
>
> Image processing (and GPUs) are typically constructed around the SIMD (single instruction multiple data) paradigm [1] where the same operations are performed on different data simultaneously. This project will look at a sub set of such processors inspired by the Convolution Engine [2]. Figure 1 illustrates a 2D convolution operation, which takes in an input matrix (e.g. the source image) and a convolution kernel, then produces an output image. This can be thought of as a series of map reduce operations: the kernel matrix is "placed" on top of the source matrix at each position, a map operator (e.g multiplication) is used to combine the source and kernel, then a reduce operator (e.g addition) is used to generate a single output value from the mapped values.
>
> The task is to design and implement a processor that is optimized to perform convolution operation on data with two dimensional structure (e.g. images).
>
> **Additional requirements**
>
> Your processor will be implemented on an FPGA, and you are free to choose how to realize your computer architecture. Studying the architecture of general multi core processors [6], and parallel machines options [4, 5, 6, 7] can be a good starting point.

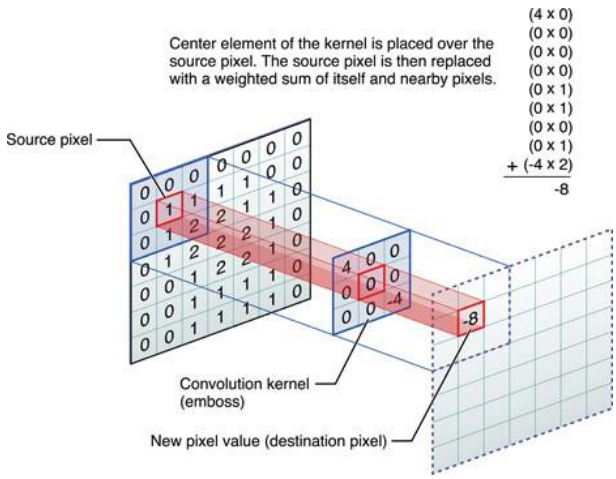


Figure 1: Example of 2D convolution for emboss, reproduced from [8].

The task should also include a suitable application that can process/produce graphical data. The output is to be displayed in order to demonstrate the processor.

The unit must utilize a Silicon Labs EFM32 series microcontroller (to act as an I/O processor) and a Xilinx FPGA (to implement your architecture on). The budget is 10.000 NOK per group, which must cover components and PCB production. The unit design must adhere to the limits set by the course staff at any given time. Deadlines are given in a separate time schedule.

And a final tip; Keep it simple, as simple as possible, but not simpler.

References

1. M Flynn; Some Computer Organizations and Their Effectiveness; IEEE Transactions on Computers. Volume:C 21 , Issue: 9. 1972
2. Qadeer, et al.; Convolution Engine: Balancing Efficiency & Flexibility in Specialized Computing. Proceding of ISCA 2013, ACM 2013. http://csl.stanford.edu/ christos/publications/2013.convolution.isca.pdf (Slides from presentation: http://csl.stanford.edu/ christos/publications/2013.convolution.isca.slides.pdf)
3. Open VG: https://www.khronos.org/openvg/
4. Bell et al.; TILE64 Processor: A 64 Core SoC with Mesh Interconnect; ISSCC; 20M Flynn; Some Computer Organizations and Their Effectiveness; IEEE Transactions on Computers. Volume:C 21, Issue: 9. 1972
5. Kongetira et al.; Niagara: A 32 way Multithreaded SPARC Processor; IEEE MICRO; 2005
6. Wikipedia; Goodyear MPP; http://en.wikipedia.org/wiki/Goodyear_MPP
7. Borkar and Chien; The Future of Microprocessors; Communications of the ACM; 2011.
8. http://www.westworld.be/convolution kernels/
9. http://www.ke5fx.com/crt/8753aft.jpg

1.2 Definitions

Processor
The processor implementation on the FPGA.

Board
The printable circuit board (PCB).

Computer
The board with soldered components.

System
The computer with peripherals, power supply, etc.

1.3 Convolution

1.3.1 Calculus

Mathematically, convolution is an operation that maps two input functions to an output function:

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

Its discrete counterpart is:

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

A monochrome image can be thought of as a 2D matrix with a width W and height H. 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 a special matrix called the *kernel*, or 0 if x' or y' is out of bounds for the matrix.

The convolution of the image and the kernel is then:

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

Depending on the kernel matrix, various effects like blurring or sharpening may be applied to the image with convolution. For a colored RGB image convolution is performed once per color channel.

1.3.2 Matrix Operation

As described above, an image and a kernel may be convolved with pure calculus by defining functions for accessing matrices. By imposing restrictions on the dimensions of

the kernel matrix we can conceptualize the convolution process in a way that lends itself well to processing images: We let the kernel matrix be a square matrix with side length D, hereby referred to as the kernel dimensions. We then change the definition of the kernel function $K(x', y')$ to evaluate to 0 whenever x' or y' is outside of the kernel matrix bounds.

Imposing this restriction on the kernel function means each pixel in the image matrix is a function of its neighbourhood rather than the entire image. To visualize this, for each pixel in the image matrix, overlay the kernel with its center on the currently evaluated pixel. Each neighbour pixel, including the center which is its own neighbour, is multiplied with its corresponding kernel value relative to the center pixel. For each multiplication the result is summed, and the final sum represents the output pixel. Each pixel has as many neighbours as there are elements in the kernel matrix. Conversively each pixel is part of as many neighbourhoods as there are elements in the kernel matrix, and in each neighbourhood it is associated with a unique element of the kernel.

TODO: nice illustration of convolution

1.3.3 Generalize convolution

We can abstract the convolution process by using mapping function $f : RxR \to R$ in place of multiplication, and $g : RxR \to R$ in place of addition. A convolution can then be described as (f, g), from here on referred to as the map function and the reduce function, where the standard convolution can be expressed whit $(*, +)$, while $(min, +)$ describes finding the shortest path. The only restriction we impose on f and g is that they form a semiring. Abstract algebra is outside of the scope of our report, but informally we impose that the TODO

Can we use domains outside of R? Describe associativity, for instance show minus not nescessarily working?

1.4 Implementing convolution

On modern computers the memory bandwidth will usually be the bottleneck of any operation involving large amounts of data, such as an image. TODO reference (or is it considered common knowledge?) The essence of implementing efficient convolution is therefore removing redundant data movement. Whenever we move a pixel into working memory we ideally want to perform a map operation for every neighbourhood it is part of. If a pixel is ejected from memory it will have to be retrieved again until all convolutions it is part of has been calculated. Ideally we would have enough memory to store an entire image and work on all parts simultaneously. While this can certainly be done, if we want speed we have to sacrifice memory size. The challenge is therefore to use as little memory as possible balanced with reducing redundant loads.

1.5 Digital Video

Through the times video has been encoded in a variety of formats, each with its advantages and drawbacks. In our project, we need to find a balance between quality, size and performance that maximizes our throughput without sacrificing too much image quality.

A common encoding for images is to encode the pixels in a 2-dimensional array in a row-major order, or in other words as an array of scanlines.

1.5.1 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 an image with visible pixel, thus of poor quality. Too many pixels gives a huge performance penalty as higher resolutions require more storage space and therefore also more throughput.

1.5.2 Colors

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

Separate Component

One way of achieving this goal is to separate the color into three components and store each individually. A common choice of components is red, green and blue, giving rise to the RGB color 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 color of a pixel resides in one place.

As data stored by computers is discrete and of limited size, we have a limited number of values to use when describing the amount of each component present in a given color. 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 color.

Indexed

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

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 1.1: Bit rates given different video parameters

1.5.3 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, we need to know the frame rate in addition to the image resolution and color representation to be able to determine the bit rate of the video.

Table 1.1 displays the bit rate required at some common resolutions, depths/index sizes and frame rates.

1.6 High-Definition Multimedia Interface (HDMI)

We chose HDMI to show the output result from the FPGA. The HDMI cable can carry audio, Ethernet and other information in addition to the video stream itself, but only the video stream was utilized for this project. Our implementation of HDMI was based on the Xilinx application note 495[2] and its reference design files. In our implementation it is possible to take an HDMI input, do convolution on the input and then transmit the result as HDMI output. As our PCB consists of two HDMI connectors, it is possible to do this. Another way is getting the image data from the MCU or directly from the camera through the FPC connector, do convolution and then send the result as HDMI output.

1.6.1 Transition-Minimized Differential Signaling

Transition-Minimized Differential Signaling(TMDS) is a standard used for transmitting video data over HDMI. HDMI uses it at the physical layer, and our implementation use the native TMDS I/O interface that is featured by the Spartan-6 FPGAs. Four channels of serial data establish the HDMI video transmission. It has three channels for the RBG color information and one for the pixel data rate clock. Each pixel has a 24-bit color depth, with each color being 8 bits each. These are separately convertered into 10-bit symbols before being serialized and transmitted onto the TMDS data channels. It is this 10:1 serialization 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 synchronization 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. Notice that, for the same voltage, this doubles the distance between the high and low states compared to using only a single cable and ground. The lack of a need for a common ground also eliminates problems related to ground offset

TMDS transfers 10 bits at a time using an encoding that minimizes the number of transitions required between each set of bits.

TMDS transmitter

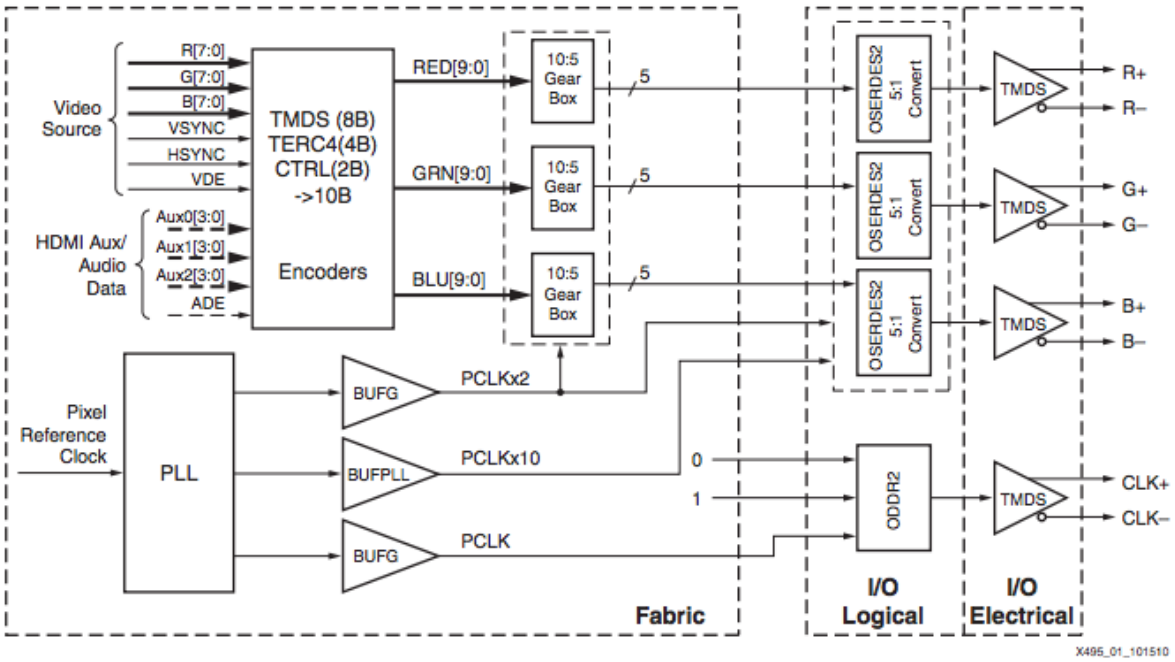


Figure 1.1: Overview of the TMDS transmitter design

The TMDS transmitter design can be seen in figure 1.1. Each color, VSYNC, HSYNC and VDE is sent to the TMDS encoders. In the verilog code this module is called dvi_encoder. The output data is then serialized, using the serdes_n_to_1 module. These serial bitstreams are then sent to the OBUFDS cores to produce the wanted output differential signals. The clock is produced using the PLL_BASE module. The wanted pixel clock is then sent to the ODDR2 module, to get the wanted differential signal output.

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 and can be seen in figure 1.2. 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:

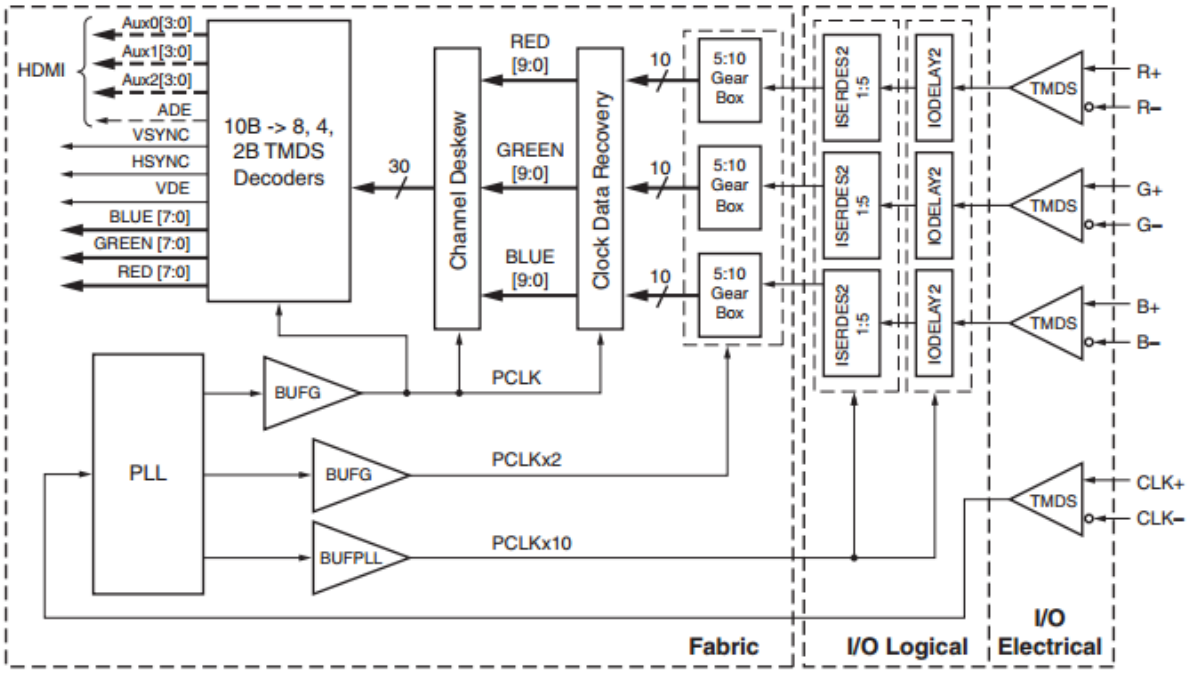


Figure 1.2: Overview of the TMDS receiver design

- 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

1.7 Camvolution

The given task can be solved in multiple ways. This section presents our interpretation and the requirements for our system.

1.7.1 Area of Focus

As the task places no restrictions on what we should focus on as long as the computer performs convolution, we considered several possibilities, 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 Maximize the throughput of the system.

In the end, the group decided to focus on performance as this would give us a more engaging demonstration, possibly with convolution performed on a live video stream.

1.7.2 Requirements

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

Functional Requirements

The functional requirements set for this project is listed in table 1.2 and a more thorough descriptions follows.

Name	Description	Priority
FR1	The processor should do convolution on a 2 dimensional image with at least a 3x3 convolution kernel	High
FR2	The processor should have a video port to show graphical output	High
FR3	It should be able to use a camera as input	Medium
FR4	The machine should boot and operate without an external computer	Medium

Table 1.2: Functional Requirements

FR1 The processor should be able to do convolution. As the assignment states, this should be done on a 2 dimensional structure for which we have chosen an image. Also, convolution is meaningless with a convolution kernel smaller than 3x3, thus this is also included in the requirements.

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

FR3 As we decided to focus on performance, one of our goals is to be able to do convolution on a live video stream. This requires an input stream which we have decided should come from a video camera connected to our computer.

FR4 To make the demonstration a bit smoother and ensure our computer can accomplish our goals independently, we also require that the machine should be able to boot and operate without an external computer.

Non-functional Requirements

In addition to our functional requirements, the task at hand also enforces requirements not related to the functionality of the computer produced. These are listed in table 1.3.

Name	Description
NFR1	The processor should be implemented on a Xilinx FPGA
NFR2	The processor should use Silicon Labs EFM32 microcontroller as I/O processor
NFR3	The cost of developing the system should not exceed a budget of 10 000 NOK

Table 1.3: Non-functional Requirements

1.8 Raspberry Pi Camera

After some consideration it was decided that the *Raspberry Pi Camera Board*[1] would be used as the camera module for the system. The *Raspberry Pi Foundation* has designed this peripheral for use with the *Raspberry Pi* computer.

There were several advantages leading to this choice, including

- Availability from retailers
- Associated open source software
- Wdiely used in other projects, meaning there is a lot of experience to draw from

The camera module can take still shots as well as record continuous video. The camera sensor sits 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. Parameters that may be controlled include video encoding, resolution in two dimensions and framerate.

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

2 | Description & Methodology

2.1 System Overview

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. The convolution is to be done on the FPGA using a custom processor implementation named *Daisy*, described in section 2.4. To make the FPGA as simple as possible, the camera is connected to and controlled by the MCU, and the video is passed to the convolution processor in a suitable format over an EBI bus.

Furthermore, 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 2.1 displays the overall system architecture.

2.1.1 Data Flow

Notice how data flows from left to right in figure 2.1. The data we are going to process is first created by the camera and sent to the MCU. The MCU may have to do some conversion should the camera provide the image in a format unsuitable for the processor, but will mainly be forwarding the data directly to the FPGA. However, interfacing with the camera is assumed to be simpler to do from the MCU than from the FPGA as libraries most likely already exists.

On the FPGA, the data is received by an EBI controller which takes care of the protocol used on the bus and forwards data to the processor. When data arrives at the processor,

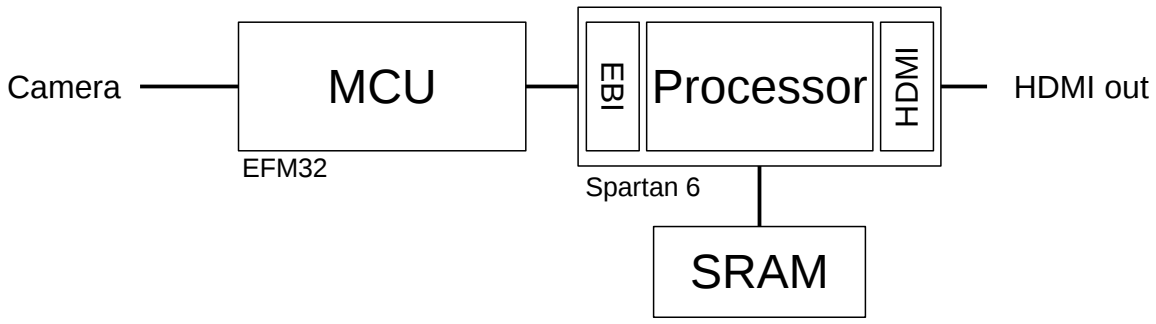


Figure 2.1: System Architecture

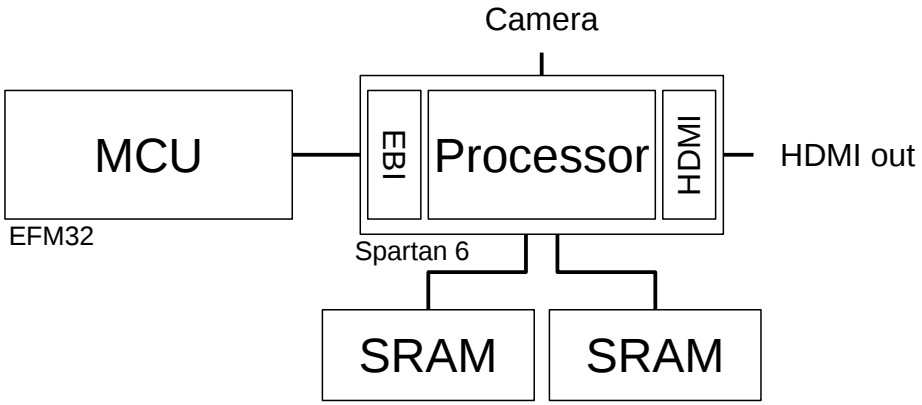


Figure 2.2: Alternative System Architecture

the convolution starts and the output is written to SRAM.

The SRAM is temporary storage to avoid synchronization issues between the FPGA and the HDMI controller. Synchronization at this point would require the rest of the system to always be able to provide data at the exact rate provided by HDMI and was considered to be too risky.

2.1.2 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.

One of the measures taken was to add an extra HDMI port connected to the FPGA in case we failed to transfer video from the MCU to the FPGA or the throughput proved to be insufficient. This allows us to connect the camera directly to the FPGA and circumvent both the EBI bus and the MCU, as shown in figure 2.2

Also shown in the figure 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, we can ensure they never access the same memory at the same time.

2.2 Hardware

This chapter targets the 1st revision of Camvolution board layout. The board is intended for use of camera input and to display a convoluted output to one of the HDMI connectors. It has several redundant headers, in case the 1st revision should have any faults. The

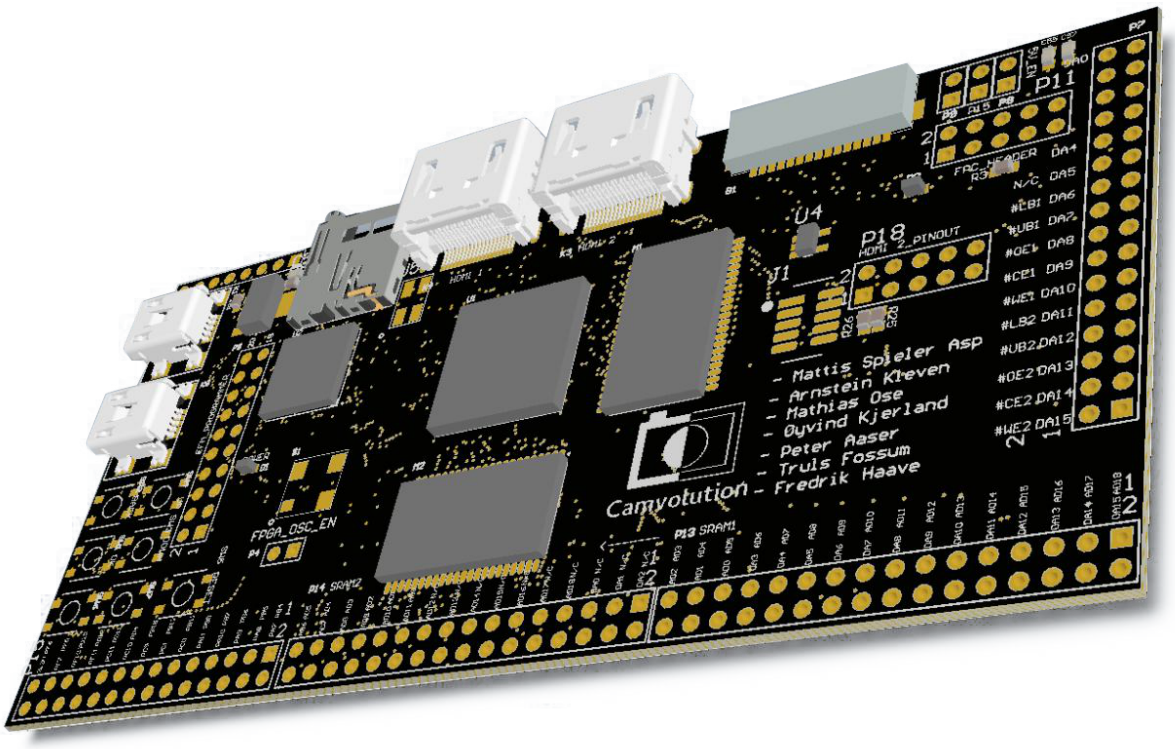


Figure 2.3: Camvolution board layout

hardware that relates to the Xilinx Spartan 6 FPGA, and Giant Gecko 990 EFM is not covered here.

2.2.1 Features

- Xilinx Spartan 6 FPGA
  - Target control
- Giant Gecko 990 EFM
  - Board Controller
- Connectors
  - HDMI I/O
  - FPC Camera
  - MicroSD
- 120MHz Oscillator for FPGA
- 48MHz Crystal for EFM
- Digital I/O
  - 7 Mechanical Buttons
  - 7 Expansion Headers

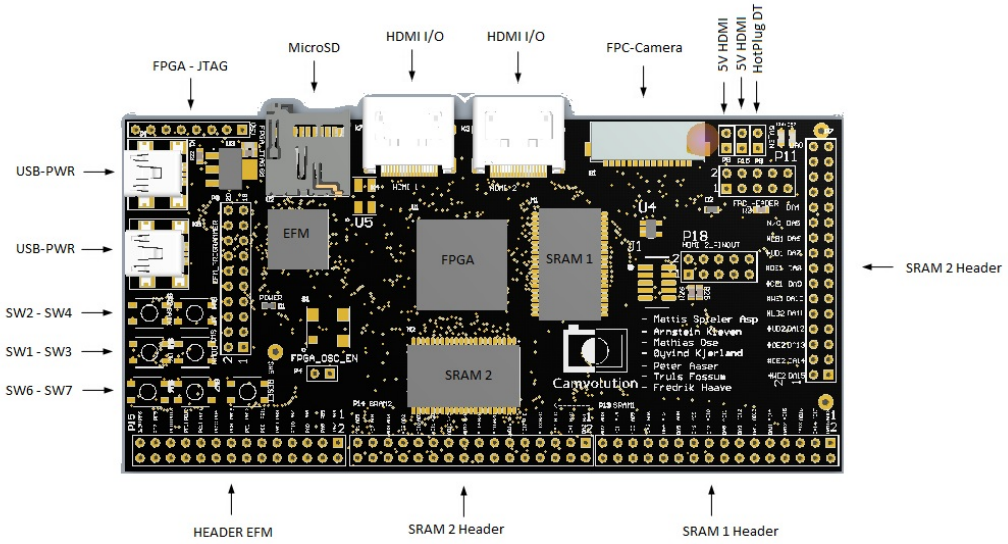


Figure 2.4: Overview of Camvolution board

  - 2 LED's

- External memory
  - 2 AS7C38098A SRAM

2.2.2 Power Supply

The board is powered by a MiniUSB connector, with two possible options for powering it. Either connect it to the pc, or use an 5V USB power supply.

The 5V is regulated with a 3.3V LDO regulator, which gives power to most of the board. This 3.3V is also connected to a 1.2V LDO regulator, that powers a minimum required power pins on the Xilinx Spartan 6 FPGA.

2.2.3 Measuring the current consumption

There is no onboard solutions for measuring the current consumption, this must be done externally if required.

2.2.4 Clock Generation

FPGA Oscillator

The oscillator is connected to an enable signal on the EFM32 pin PF12. To enable the 120MHz oscillator PF12 must be set to output and driven high. The oscillator enable pin is also connected to an external input and can be set high by connecting this pin to VCC. The easiest way of enabling the oscillator is to add a jumper on the FPGA_OSC_EN

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 2.1: HDMI Connector

header.

EFM External Crystal

The EFM has internal and one external clock. The external clock source is 48MHz and can be set by using the guide from the EFM, on how to enable external clock source.

2.2.5 Connectors

HDMI

There are two HDMI connectors on the board, both of them HDMI1 and HDMI 2 must also be connected to 5V, and can be set with a jumper at header 8, and header 15 respectively at the upper right corner. The HDMI 2 is also connected to output pins at P18, this can be used for debugging.

FPC Raspberry Pi camera connector

There is 1 Raspberry Pi camera connector on the board. It is also connected to header P11, and can be used for debugging.

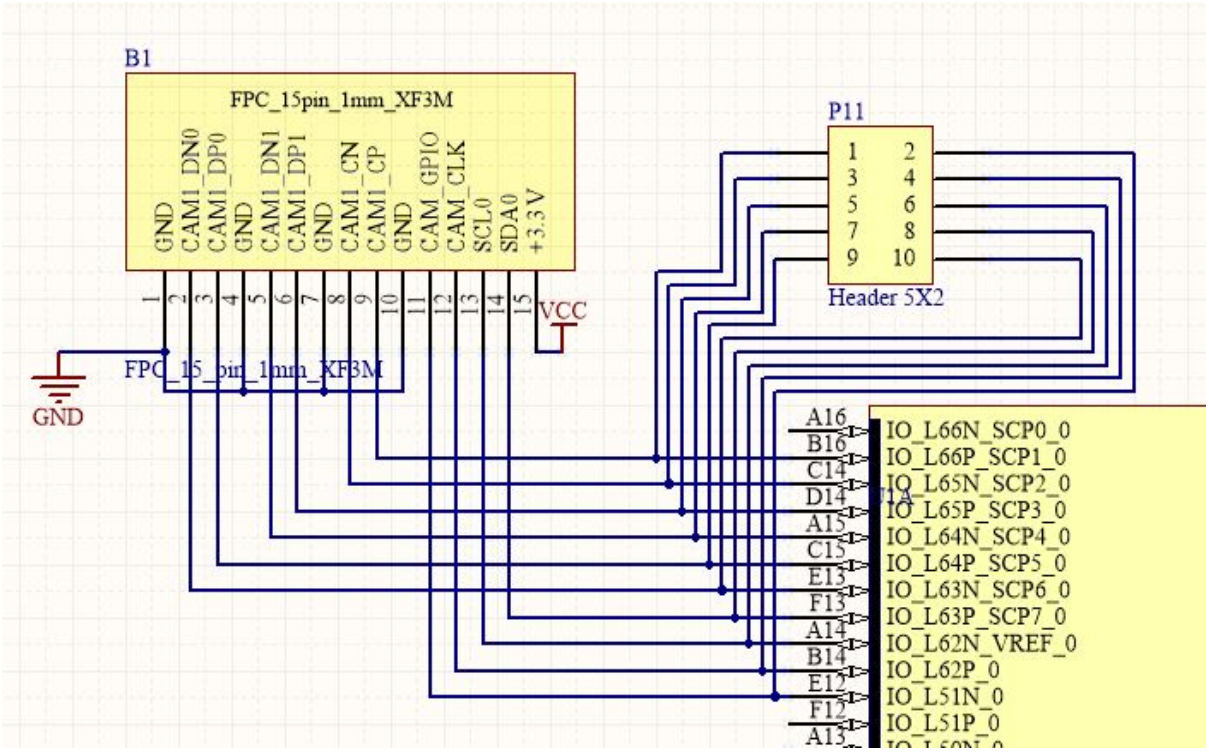


Figure 2.5: Raspberry Pi connector

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 2.2: I/O Buttons

2.2.6 Digital I/O

Buttons

The board contains 7 buttons, and are all connected to pull-up resistors. When the button is pushed down the input pin will be grounded.

The FPGA has 2 buttons for control. These are connected to bank 1 on the P15, and P16 pin. The MCU has 4 buttons for control. These are connected to PD8, PD7, PD6 and PD5. Se table 2.2 for info. The last buttons is connected to reset on EFM.

Header output

The EFM has spare pins that all are available on the P15 header pins. Pin 28 on P15 header is 3.3V VCC. The rest is pinouts from the EFM. Se table 2.3 for location of pins.

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 2.3: Header Output

2.2.7 SRAM

The FPGA is connected to two SRAM, recognised as SRAM1 and SRAM2. These are connected to pins on the FPGA bank 1 and bank 2. They are also in case of failure, or for any reason connected to output pins on the development board. We will not go through all the pins here. This can easily be seen from the figure in the schematics. The schematics will be available in the appendix.

2.2.8 JTAG and Programmer

Both the FPGA and the EFM is connected with pinouts to a programmer. The EFM is connected to a 20 pin header that can be used with the generic programming pin from a EFM development kit. The FPGA is connected to a 8 pin JTAG header, this can be

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 2.4: Programming Output for EFM

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

Table 2.5: Programming FPGA using EFM SPI

routed to a debugger used for FPGA programming. See table 2.4 for EFM programmer pin, see figure 2.6 for FPGA programmer pin.

Program_B is connected to PB11, Done is connected to PB12.

2.2.9 Programming FPGA using EFM SPI

The FPGA can also be programmed from the EFM using SPI from the EFM.

INIT_B shares the same pin as the SRAM2 data line, but when programming the FPGA the SRAM will not be active. Se table 2.5

2.2.10 LED

There are 2 LEDs on the board. One is connected to power on the board and will light up if power is connected. The last LED is programmable from the FPGA on pin N11. Set this pin high and the LED wil light up.

2.2.11 EBI-BUS

The FPGA and the EFM is connected with a high speed parallel bus. 20 addressable pins and 16 data lines. Se figure 2.7 and 2.8 for EFM pins.

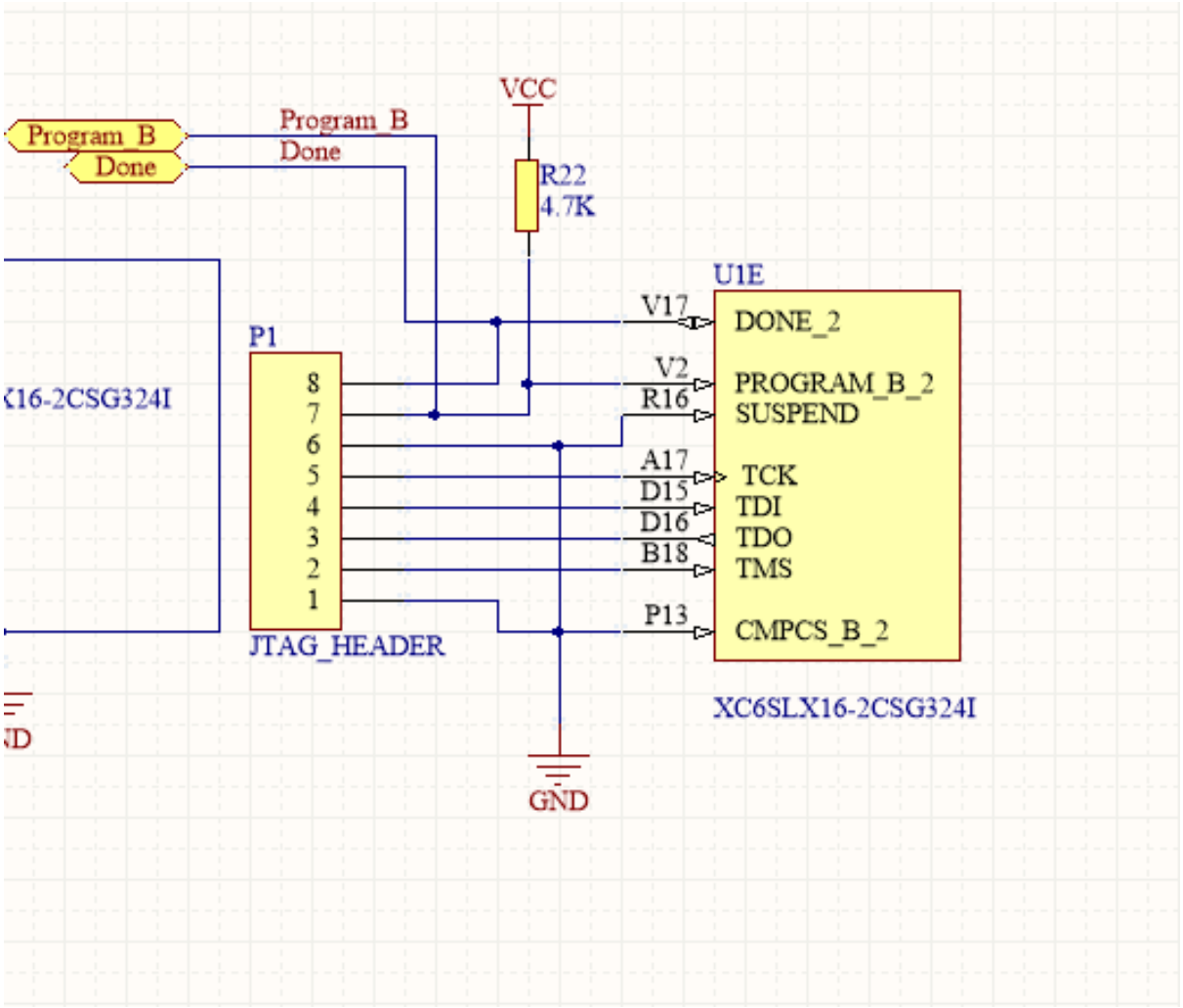


Figure 2.6: Programming output for FPGA

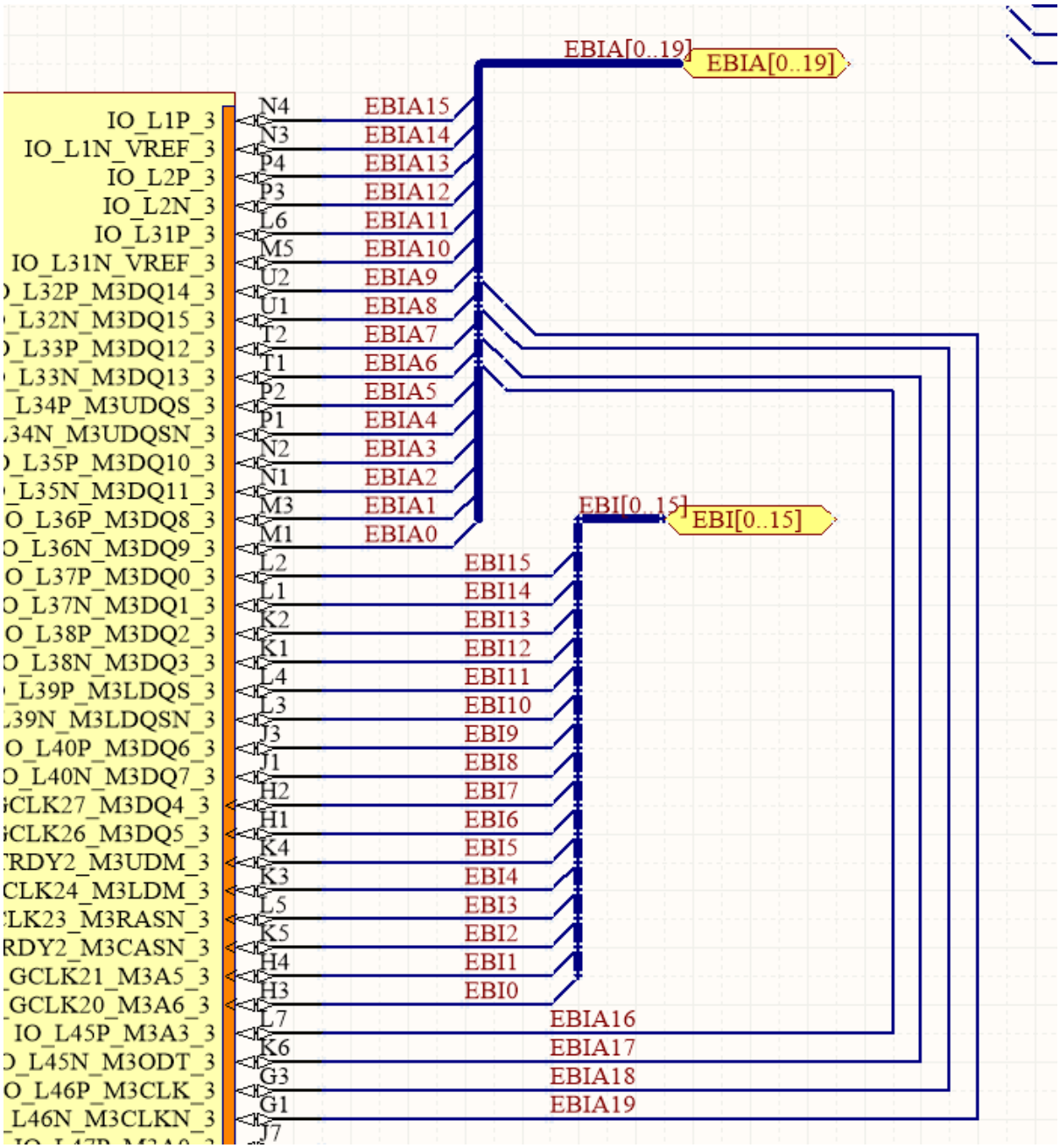


Figure 2.7: EBI-Bus input

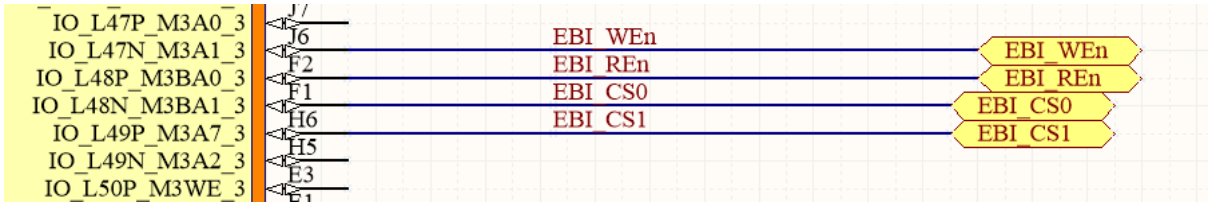


Figure 2.8: EBI-Bus control signals

There has been a change in the layout, so the EBI_WEn pin is connnected to U10 instead of J6.

2.3 MCU

The MCU is used as an I/O-processor, reading from a MicroSD card, configuring the FPGA, and feeding kernel and image data to the FPGA over an EBI bus interface. An EFM32GG is used as the MCU for the system, providing 128kB RAM, 1024kB internal flash memory and a range of modules including EBI, USART, GPIO and DMA. The board has headers connected to unused pins on the MCU, making it possible for the MCU to communicate with other devices. Figure 2.9 shows how the components are connected.

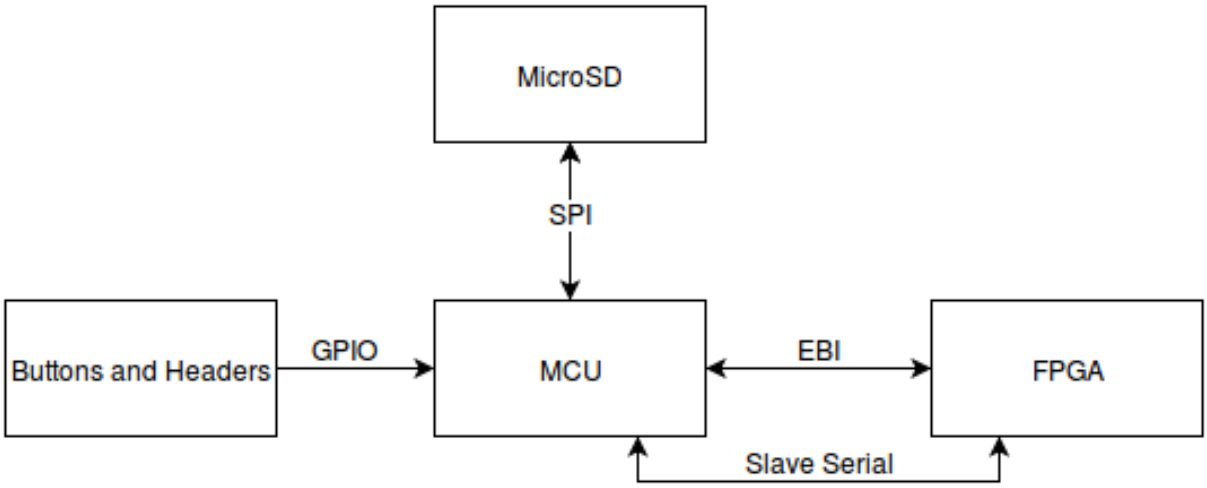


Figure 2.9: Overview of the components connected to the MCU

2.3.1 Storage

Due to the limited size of the MCU non-volatile memory not exceeding 1024kB, an additional larger source of data is necessary. A MicroSD card is used for this storage, as 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[1]. 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. 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 2.6.

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 2.6: MicroSD card folder structure

2.3.2 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 binfiles from the storage. Pins between the MCU and the FPGA are connected according to Slave Serial as described in Spartan 6 Configuration User Guide[3] page 28. This configuration scheme relies on sending a binfile as a serial data stream using a clk 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. Configuration status pins set by the FPGA informs the MCU if it is ready to be programmed, if an error occurred while programming or if the FPGA has successfully programmed.

The FPGA can also be reprogrammed later. For example when using kernels of different sizes, a configuration optimized 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.

2.3.3 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, 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 us to a maximum size of 15x15 kernels. The next byte is the divisor, which is used for averaging filters. The rest of the bytes in the files are the elements of the kernel.

2.3.4 External Bus Interface

The FPGA and the MCU is connected through an EBI bus. This interface is supported natively by the MCU 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. To accomplish this, we utilize the large address space available. An overview can be seen in figure 2.10.

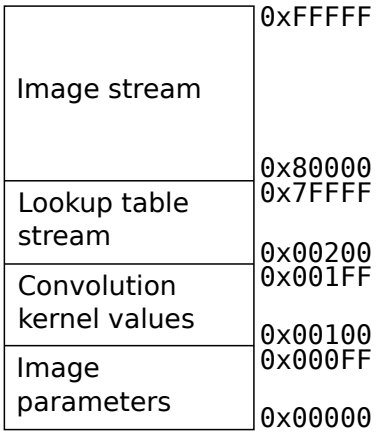


Figure 2.10: EBI Address Space

Processor Configuration

Configuration values for the processor have their own memory addresses so that configuring it is only a matter of saving the correct values to the correct addresses.

Note that the amount of space allocated to each group of options in figure 2.10 is more than needed. This simplifies the process of adding more configuration options if required at a later point.

Streaming Data

Some data is of unknown or unlimited length. This data is streamed to daisy by reusing the same addresses several times. By assigning a large block of the memory space to each data stream, we can utilize the DMA without having to worry about unintendedly overwriting configuration values.

The reuse of addresses also provides a natural sync point for the video stream; each write to the first address of the address space reserved for streaming video is assumed to be a *frame sync*, which signals the start of a new frame. This makes the system less fragile as the processor can catch up with the data stream should it ever end up being out of sync.

2.3.5 User Interface

Even though the MCU is able to configurate values on the processor, it does not do so until it receives input telling it to perform the configuration. This input comes from a simple user interface based sending a graphical interface over the image stream and let the user control it with buttons. The operation of the system can be controlled by the buttons only, making it easier to use and removing the need for being controlled by an external computer.

The system provides four buttons for the user interface:

- UP

- DOWN
- BACK
- OK

Startup and Reset

When the MCU is first started up or reset, it reconfigures the FPGA with a default binfile. After configuration of the FPGA, the MCU configures the processor with default values for the convolution kernel, map operation and reduce operation. The processors 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.

2.4 Processor

2.4.1 Convolution engine

In this section we will provide a top down view of our system, introducing how components interface with each others and what their purpose is before we describe their implementation.

2.4.2 FPGA

The very heart of our computer is our custom made architecture implemented on an FPGA. In this section we will first describe the overall architecture of the convolution engine, and then we will drill down and examine the core modules. Modularity and extendibility is a core principle in our design, however in this section we will describe the processor with the minimum viable kernel dimension, 3 x 3.

Overall design

The overall design of our convolution processor, named "Daisy", shown in fig TODO consists of four main modules with the following responsibilities as follows:

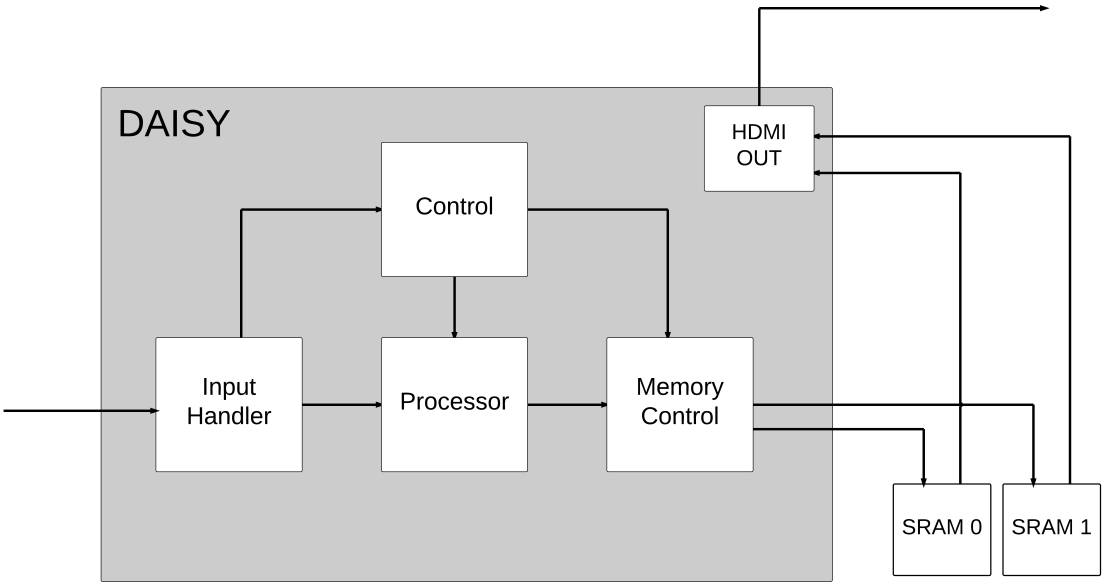


Figure 2.11: The data path for our processor, named daisy after the way it daisy chains many control signals

Input handler

The interface between the data stream and our processor is responsible for gathering data from a wide variety of input streams of different bandwidths and speeds and

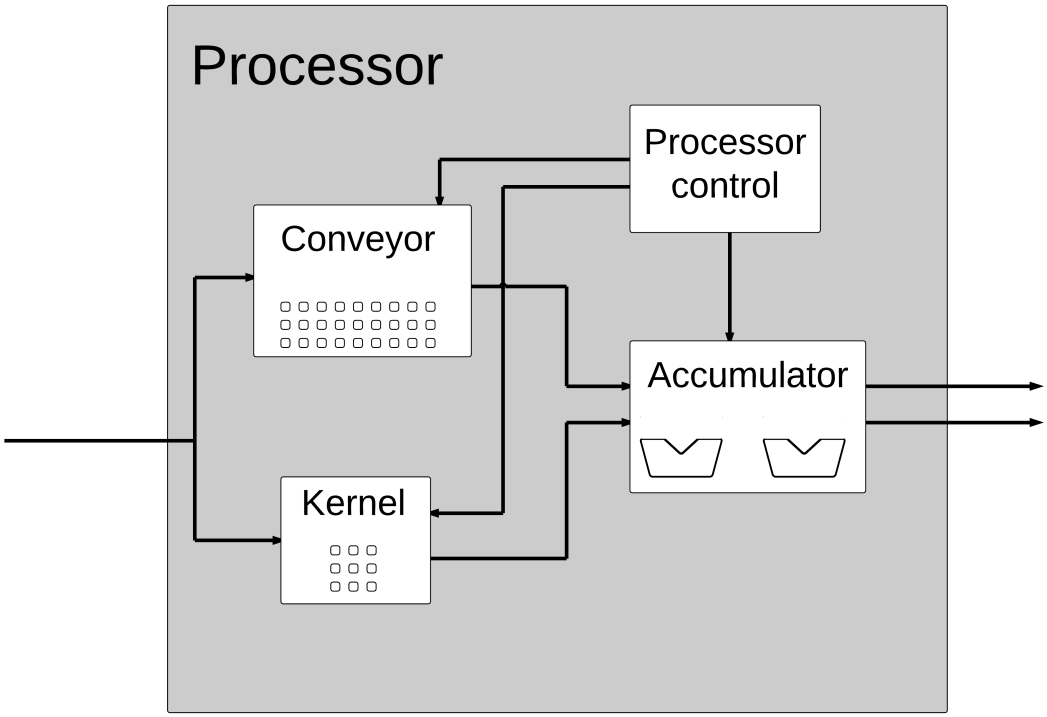


Figure 2.12: TODO: put this where it belongs

present this data to the processor as a continuous data stream with a fixed width. In order to do this the input handler must be able to translate from any input width to the desired output width. The processed input is then stored in a double buffer, allowing the input handler to both recieve data and feed the processor at the same

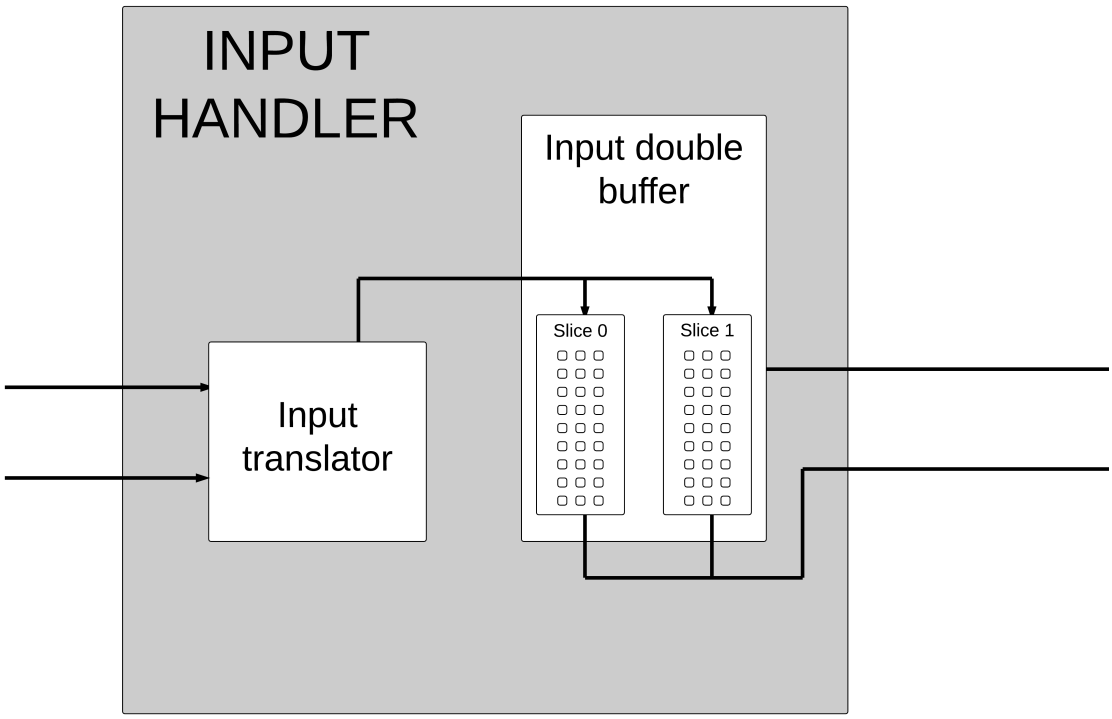


Figure 2.13: TODO: put this where it belongs

time. As soon as a buffer is filled the input handler indicates that data is available to the processor and outputs data continously until the buffer is empty.

Processor

The processor recieves a pixel stream from the input handler and outputs convoluted

data. It consists of a pixel buffer called the conveyor, a buffer for kernel data, an accumulator which performs the map and reduce operations, and a control module. In addition to providing convoluted data it also indicates that the data is valid. This is nescessary because not all pixels entering the conveyor has access to their full neighbourhood, so the processor cannot provide valid output every cycle.

Control
: The control module keeps track of the system state by monitoring the input data stream, and issues control signals according to the state. In its initial state the control module is in programming mode. In this state the control module makes sure the initial instructions are dispatched to the correct modules before it enters data mode. When in data mode the controller keeps track of the data stream from the input handler, and uses this along with the valid signal from the processor to indicate to the output handler whether the output is valid or not.

Memory Control
: Processed data is fed to one of the two SRAM chips serving as frame buffer. The output handler keeps track of which SRAM chip should be written to, and at which address.

TODO: This is where the overview should go. The next sections needs to be fleshed out A LOT! The content below lacks context because this part is not yet done.

To illustrate how the components work we describe how the first frame of a videostream is processed.

1. In its initial state the control module is in programming mode. It intercepts data from the input handler and fills the kernel buffer and decodes which map and reduction operation should be used

2. After the instruction collected the control module enters data mode, and the input module starts filling its double buffer.

3. When the input module has filled its first buffer it indicates that it is now outputting valid data which the processor will process.

4. The control module keeps track of how much data has been fed to the processor. When the processor indicates it has valid output the control module signals the memory controller to be ready to write output to SRAM 0.

5. When the input buffer is exhausted the control waits until all valid data has been flushed from the processor before it resets the processor and waits for the next buffer to fill.

6. This cycle continues until the memory controller notices it has finished filling SRAM 0 with an entire frame. The memory controller switches to SRAM 1 and prepares for next frame.

2.4.3 Input handler

TODO describe how the input handlers work.

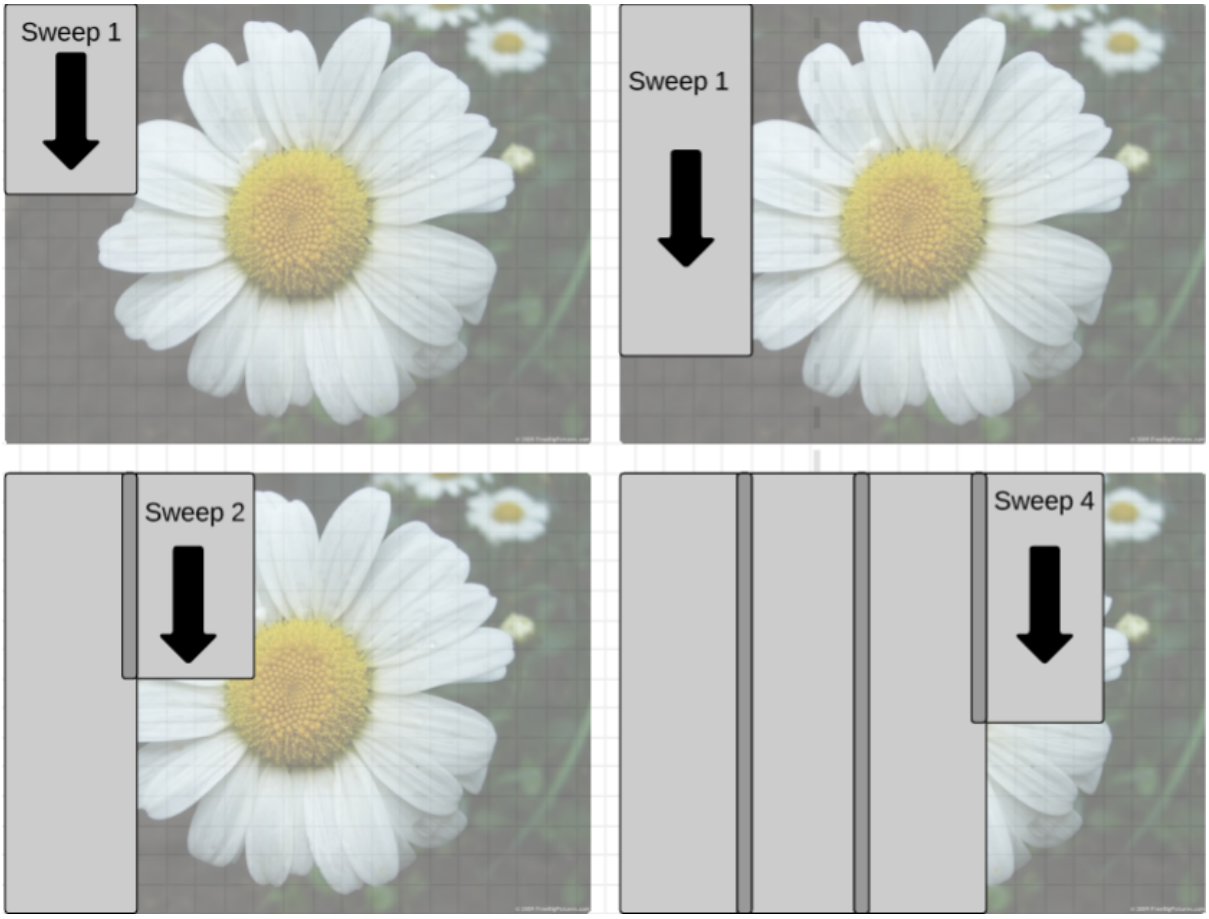


Figure 2.14: The sweep pattern used to collect data for convolution. The deep grey region indicates overlap, pixels which will be collected twice

2.4.4 Processor

The processor recieves a data stream from the input handler and performs convolution on it, outputting a stream of convoluted image data. The four main components are:

Conveyor
: The conveyor holds data in several rows, each one representing a column slice. The conveyor maintains these rows, moving data from row to row like a conveyor belt, and outputting data from each row to the accumulators.

the kernel buffer
: The kernel buffer maintains the kernel data and must provide the correct kernel data for each map operation performed in the accumulators.

the accumulator
: The accumulator consists of several buffers, each buffer storing the progress of convoluting a pixel. Input to the accumulator is data from each row of the conveyor, along with kernel data from the kernel buffer, and it is up to the accumulator to make sure the correct kernel data and pixel data is mapped, and that the result is reduced with the content of the correct accumulator.

the control unit

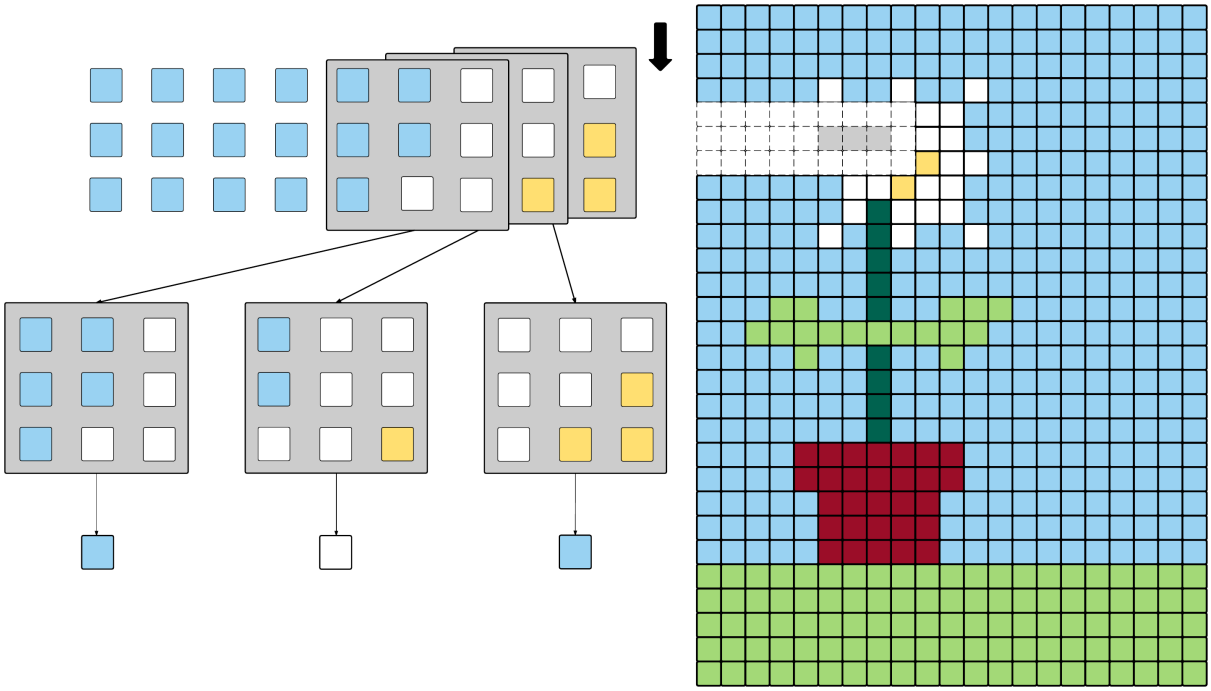


Figure 2.15: The image three greyed out pixels in the source image represent the three pixels which the regions in grey boxes help calculate. The window is three pixels deep and nine pixels wide

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.

Fig:DaisyView shows an overview of the four components in the heart of daisy.

The accumulator

Although the accumulator unit is the last unit in the datapath of daisy we will cover it first since it helps motivate the designs further up in the chain. The accumulator actually consists of seven accumulators, hereby referred to as pixel accumulators. Each pixel accumulator corresponds to a pixel in the middle row. The leftmost and the rightmost pixel in the middle row has no accumulator associated with it because it lacks three pixels necessary for a full convolution, necessetating the slight overlap in our feed pattern. In fig:DaisyView the input from the conveyor is shown as three wires, one from each row. Each accumulator is responsible to read from the correct row at the correct time, such that it accumulates only the subpixels of the source pixel.

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

We will first focus on the leftmost accumulator starting at time T0 doing the follow-

ing reads. It first reads three values from Row 3 which corresponds to its southwestern subpixel at T_0, its southern subpixel at T_1, and its southeastern subpixel at T_2, shown as the greyed out cells in row 3. At time T_3 it starts reading from Row 2, reading its western, middle and eastern subpixels at respectively T_3, T_4, and T_5, shown as the greyed out cells in row 2. Finally it reads its northwestern, northern and northwestern subpixel at time T_6 to T_8 from Row 1. What about the second leftmost accumulator then? By following the exact same read pattern, but waiting one cycle allows it to read all its values in the same manner as the leftmost accumulator! The second accumulator reads from Row 3 at time T_1, switches to Row 2 at T_4 and again to Row 1 from T_7 to T_9. In fact, every accumulator starts one cycle later than its left neighbour, meaning every accumulator has accumulated its necessary pixels at different times.

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

The conveyor belt

As established last section, the job of the conveyor is to feed data from each of its rows to the accumulator. In addition to serving data to the accumulator the conveyor must also feed new data to the belt and propogate that data downward which gives rise to its name. In order to both propagate data downward and feed correct data to the accumulator as efficently as possible the conveyor will utilize the data from a read operation to both feed the accumulator and to transfer data laterally. When a register is read, it is therefore the job of the register directly beneath it to read, conveying data downwards. By studying the tables in the previous section we see that reads are done in a wave pattern, and consequentially so is the data conveying, which has a very powerful implication: Since every accumulator and register will perform the same operation as its left neighbour offset by one cycle we can let each element be responsible for controlling the element to its right. Rather than using a central control module we can instead simply daisy chain the control signals and interface only with the leftmost components, which also explains the name of the convolution core, Daisy. An analogy to the read signals is a key. At time T_0 the control element gives the leftmost register the read key. The register does as it is told and reads its new value from the register above, and hands the key over to its neighbour. At T_1 the neighbour recieves the key, and reads from the register above it and sends the key further on.

TODO: Figure out a succint way to explain this part

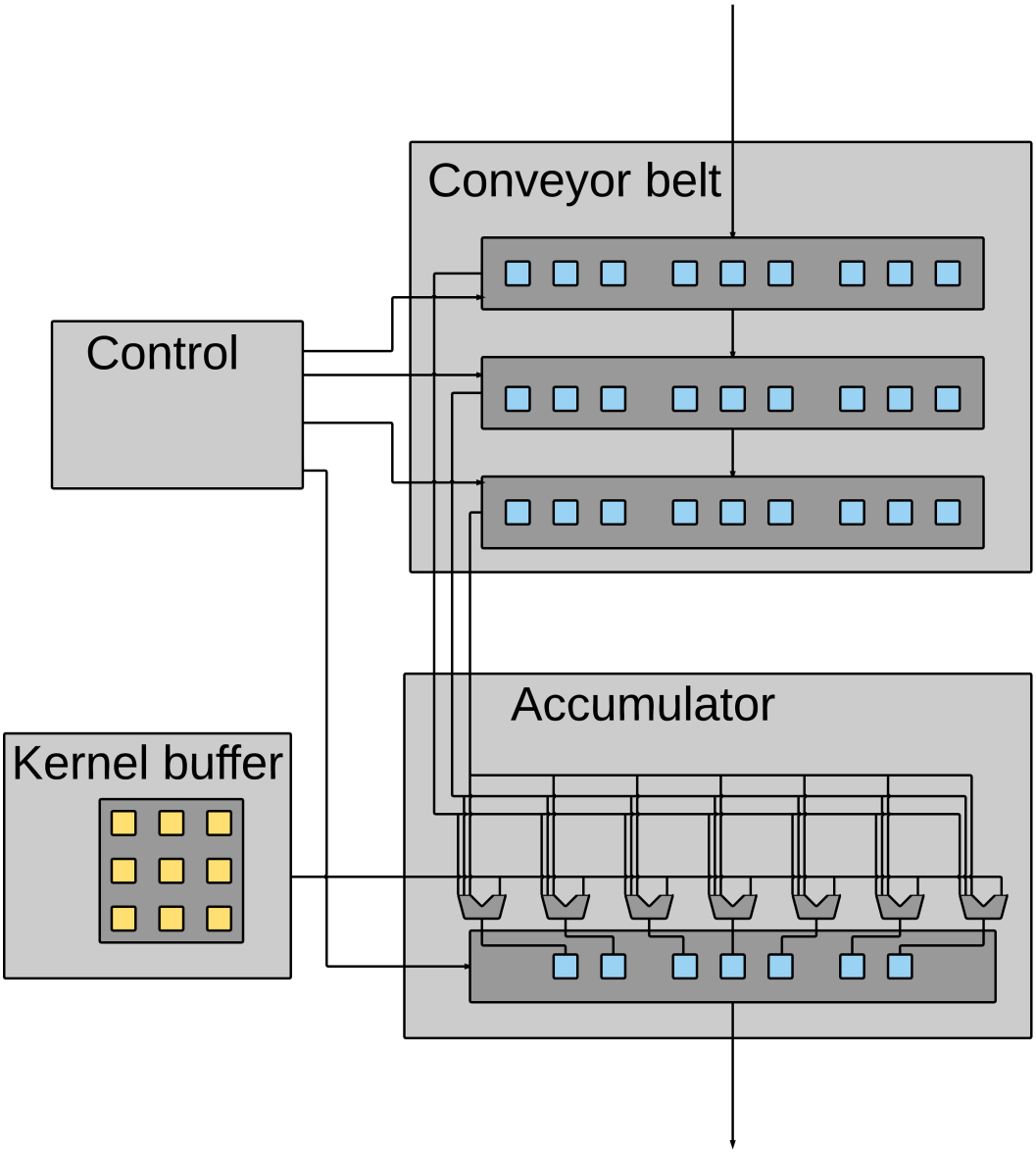
The controller

We have established how control flows throughout the system, but we still need a module to give out keys to the leftmost accumulators and registers. The controller is responsible for sending out the keys in the correct order and at the correct time. Timing is everything, if we misalign read signals and write signals we may end up in a situation where a register

fails to read when the register above it writes. For example, if the controller sends each read signal one cycle too late each register will read when the register to the right side of the one directly above it, causing the data to be sheared as it moves laterally! Other than issuing read and write keys to the registers, the controller also issues a flush key for the accumulators which tells accumulators to drive DATA_OUT with its contents, and to reset.

The kernel buffer

In our schematic we present the kernel buffer as a separate submodule. However, having already introduced how instructions are daisy chained it makes sense to do the same with kernels since seven different kernel values are in use at all times. Thus the kernel values are kept in a shift register queue living as close to the accumulators as possible, needing only two registers to hold the currently unused kernel values. The responsibilities of the kernel unit is thus to collect the first data values into the kernel buffer chain and to buffer the two kernel values which are not currently in use.


Conveyor belt
Control
Accumulator
Kernel buffer

2.5 Video

2.5.1 Throughput

For a video feed of resolution $W * H$, B bits per pixel and F frames per second the required bandwidth is given by the product of the terms

$$H * W * B * F$$

This number may grow very quickly when increasing some of the terms. It was clear from the beginning that we would no be able to output full HD video, but it was not not clear what exactly the limits would be, nor where the bottlenecks would be located.

The Raspberry Pi camera can output resolutions and framerates based on commands from the master device. It may output encoded video, but this would have to be decoded before convolution, so it was decided to go with 24-bit raw RGB as camera output.

2.5.2 Options for Video Path

There were several options for how to move the video stream to the processor.

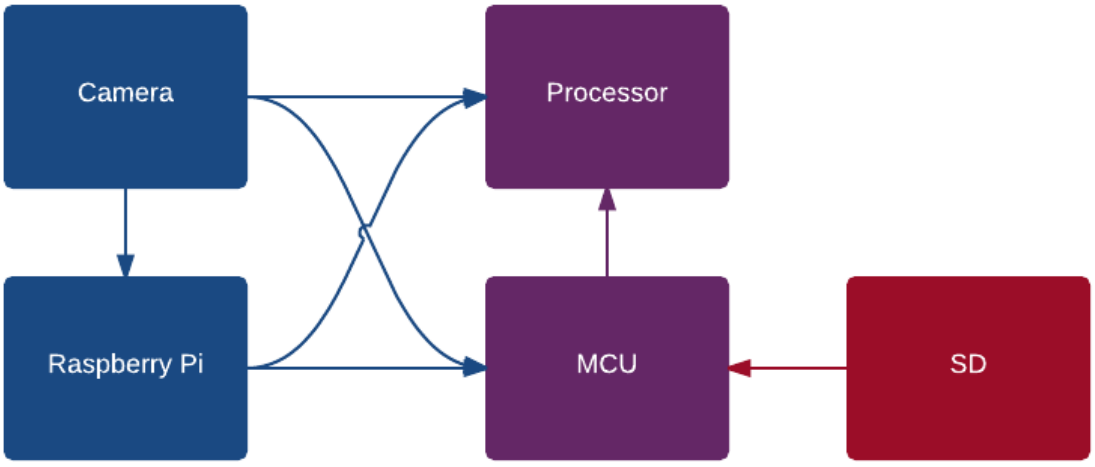


Figure 2.17: Paths an input video stream could take to reach the processor.

Direct Path from Camera to Processor

The path with the fewest bottleneck would be to send data directly from the camera to the processor. This would also mean that the FPGA would have to act as the camera master, sending control signals over I2C.

Camera to MCU to Processor

The signals from the camera could also be routed to the MCU. This would mean that the MCU would be the camera master, leaving the FPGA free to work on other problems. Another advantage of this is that the processor would not need to mode-switch between SD and camera input, since the MCU would handle this.

Using the Raspberry Pi as Camera Controller

Instead of reverse-engineering the camera control protocol, the system could include a whole Raspberry Pi with a camera as the camera module. Programming the Pi to start recording is very easy thanks to the provided software. However the video stream would then have to be outputted from the Pi to the Camvolution computer somehow, which would introduce a considerable bottleneck.

GPIO

Software was written for the Pi which started recording then used the Pi GPIO pins as a clocked parallell bus to output color values. Getting this program to work was not very difficult, however it was very slow.

HDMI

The Pi has an HDMI output. Since the Camvolution board has two HDMI ports, it would be possible to output video from the Pi to Camvolution over HDMI. This would require a HDMI decoding module to be implemented on Camvolution.

3 | Results & Discussion

3.1 Testing

3.1.1 FPGA

3.1.2 Hardware

After soldering the board, different components were tested to ensure proper operation of components and set of components. This is advantageous because potential bugs can be found before all components are combined into a complex system that is hard to debug.

SRAM Connection

The resistance between SRAM pins and the corresponding header pins was measured to ensure all wires were connected properly. It was discovered that two of the data pins for the second memory chip was exchanged, but we 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.

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, verified by inspecting variables in the debug mode of Simplicity Studio.

SD Card

Reads and writes 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 degrees 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 us to control the on-board LED using one of the buttons. The button controlled the led after configuration, thus we concluded the configuration was successful.

HDMI output

3.2 Results

3.2.1 Scalability

What convolution kernel matrix sizes can we handle, what image resolutions and depths...

3.2.2 Performance

Performance was the main focus of our project.

TODO: Add more info on achieved throughput given image format, convolution kernel matrix size...

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 number which were verified on the FPGA. A setup time of 1 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 $16\,\text{bits} \cdot 48\,\text{MHz}/2 = 384\,\text{Mbps}$.

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.

3.3 Discussion

3.3.1 Camera Input

The backup solution presented in 2.1.2 where input from the camera is provided through HDMI directly to the FPGA might have been a better option from the start. Our decision to send the data through the MCU on its way to the FPGA was based on the wrong assumptions; It was rather hard to find a camera module that interfaced easily with the MCU.

3.3.2 Image Compression

TODO: Verify this; not final

The bottleneck in our system is not the image processing itself, but how to ensure enough data is available at all times. There are several ways we could have mitigated this problem.

First of all we could have increased the dimensions, that is, increase the width of our buses. Second, we could have tried to increase the clock rate. This is not easy to do on the EFM, but the processor implementation can always be pipelined more until we reach 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. The indexed colors 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.

3.4 Conclusion

References

[1] Silicon Labs. Fat on sd card an0030 - application note. https://www.silabs.com/Support%20Documents/TechnicalDocs/AN0030.pdf, 2013.

[2] 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.

[3] Xilinx. Spartan-6 fpga configuration user guide ug380. http://www.xilinx.com/support/documentation/user_guides/ug380.pdf, 2015.