NTNU

TDT4295 - Computer Design Project

Camvolution

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

October 27, 2015

Abstract

Contents

1 Introduction

1.1 Project Description

1.2 Camvolution

2 | Description & Methodology

2.1 Overview

2.2 Hardware

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
  - 2 LED's
- External memory
  - 2 AS7C38098A SRAM

General information

This Document targets the 1 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 reduntant headers, in case the 1 revision should have any faults. The hardware that relates to the Xilinx Spartan 6 FPGA, and Giant Gecko 990 EFM is not covered here.

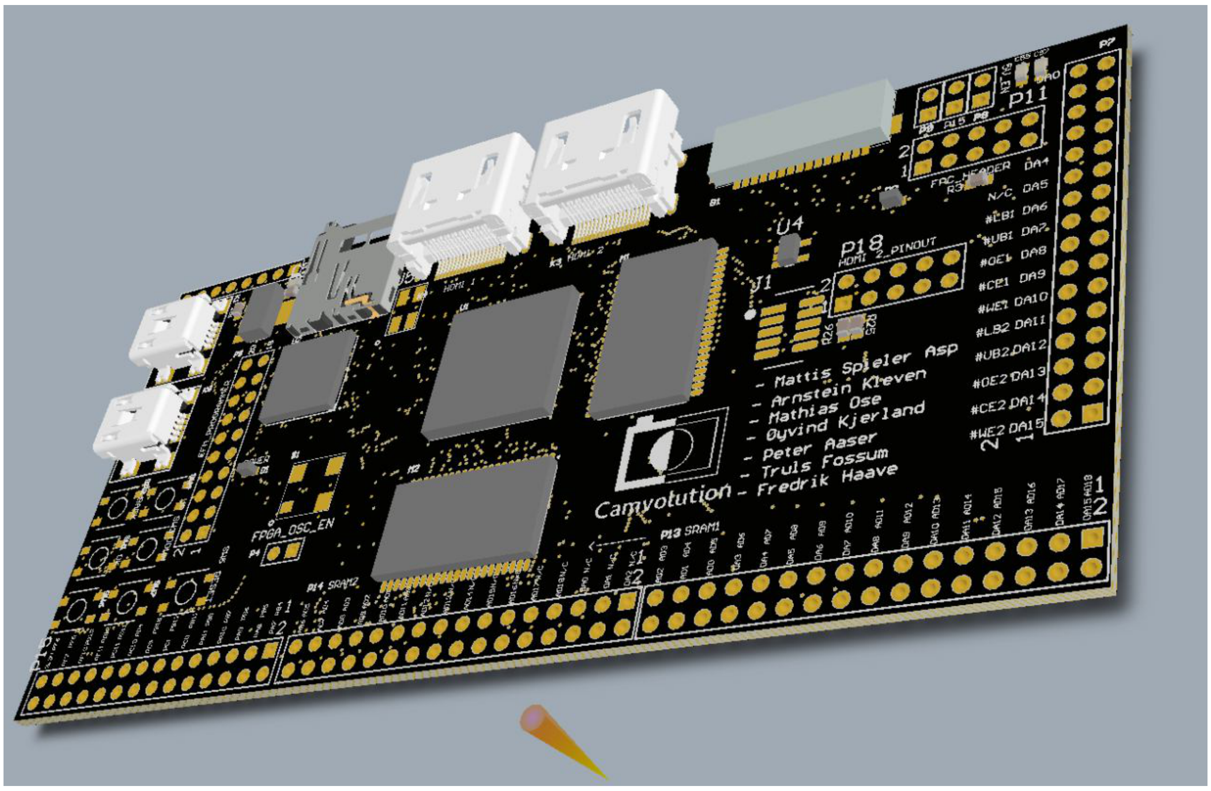


Figure 2.1: Camvolution board layout

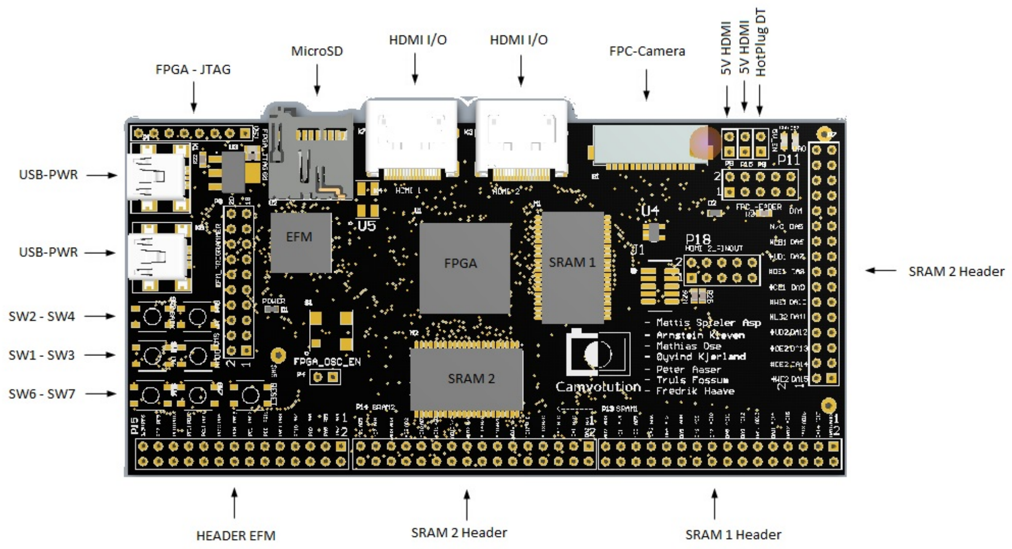


Figure 2.2: Overview of Camvolution board

Table 2.1: I/O Buttons

Button	Connection	Pullup/pulldown	Name
SW6	FPGA P15	Pullup	
SW7	FPGA P16	Pullup	
SW1	EFM PD8	Pullup	BTN OK
SW2	EFM PD7	Pullup	BTN Back
SW3	EFM PD6	Pullup	BTN Down
SW4	EFM PD5	Pullup	BTN Up
SW5	EFM K6	Pullup	Reset

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 powerpins 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 connectet 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 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 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 controll. These are connected to bank 1 on the P15, and P16 pin. The MCU has 4 buttons for controll. These are connected to PD8, PD7, PD6 and PD5. Se table [buttons] for info. The last buttons is connected to reset on EFM.

Table 2.2: Header Output

PIN NUM	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		

Header output

The EFM has spare pins that all are availiable on the P15 header pins. Pin 28 on P15 header is 3.3V vcc. The rest is pinouts from the EFM. Se table [header_out] for location of pins.

2.2.6 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 availiable in the appendix.

Table 2.3: Programming Output for EFM

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 FPGA using EFM SPI

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

2.2.7 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 routed to a debugger used for FPGA programming. Se table 2 for EFM programmer pin, se figure 1 for FPGA programmer pin.

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

2.2.8 Programming FPGA using EFM SPI

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

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

2.2.9 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.

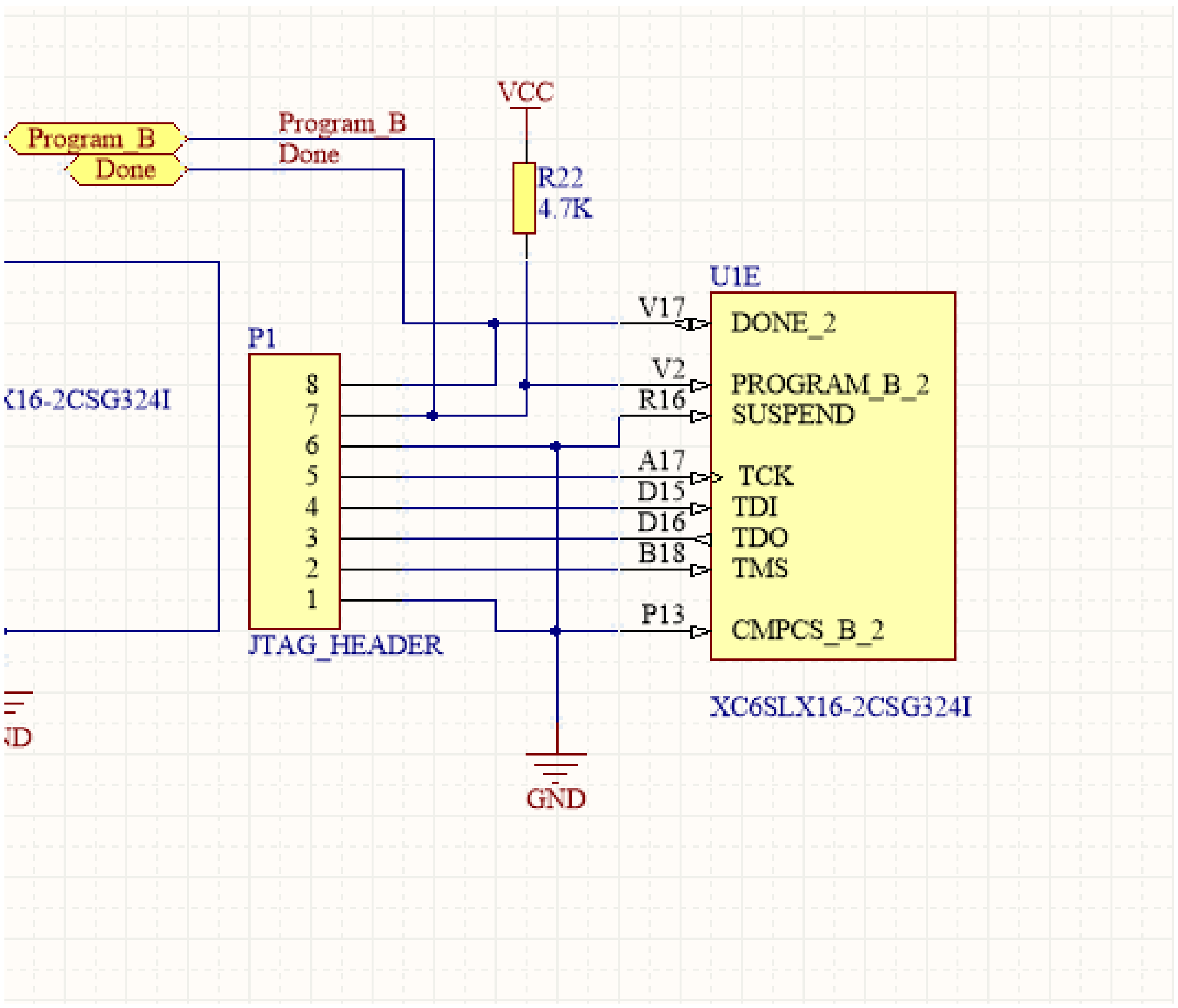


Figure 2.3: Programming output for FPGA

2.2.10 EBI-BUS

THE FPGA and the EFM is connected with a high speed parallel bus. 20 adressable pins and 16 datalines. Se figure 2 and 3 for EFM pins.

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

2.3 MCU

2.4 Processor

2.4.1 Convolution

Some words on convolution. Background-ish

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 our 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 as if only being able to do 3x3 convolutions.

Overall design

Convolution is a very regular task where data flows only forward. This means that our processor can be simplified, removing the need for a central control module. Instead each module simply operates under the assumption that all data inputs are correctly formatted and ordered and does its operations accordingly. In the figure we see this design principle reflected, no data ever flows backwards, only forwards. Breaking up the individual steps we get the typical flow of data.

1. Data from the bus is read by the bus controller. This controller serves as a clock domain crossover and is responsible for delivering data to the input handler in a clock domain crossing FIFO queue.

2. The input handler recieves data from the queue and does magic

3. Pixel data is held in a special buffer with rows representing the current sweep window.

4. The accumulator recieves kernel data from the kernel buffer and performs its mapping function on the kernel value and the pixel from the conveyor belt. When an accumulator has accumulated all its pixels it flushes its value, resetting the accumulator register and writing the old data to the memory control unit.

5. Accumulated pixels are reassembled in the mem ctrl unit and written to one of the two SRAM banks, allowing us to double buffer.

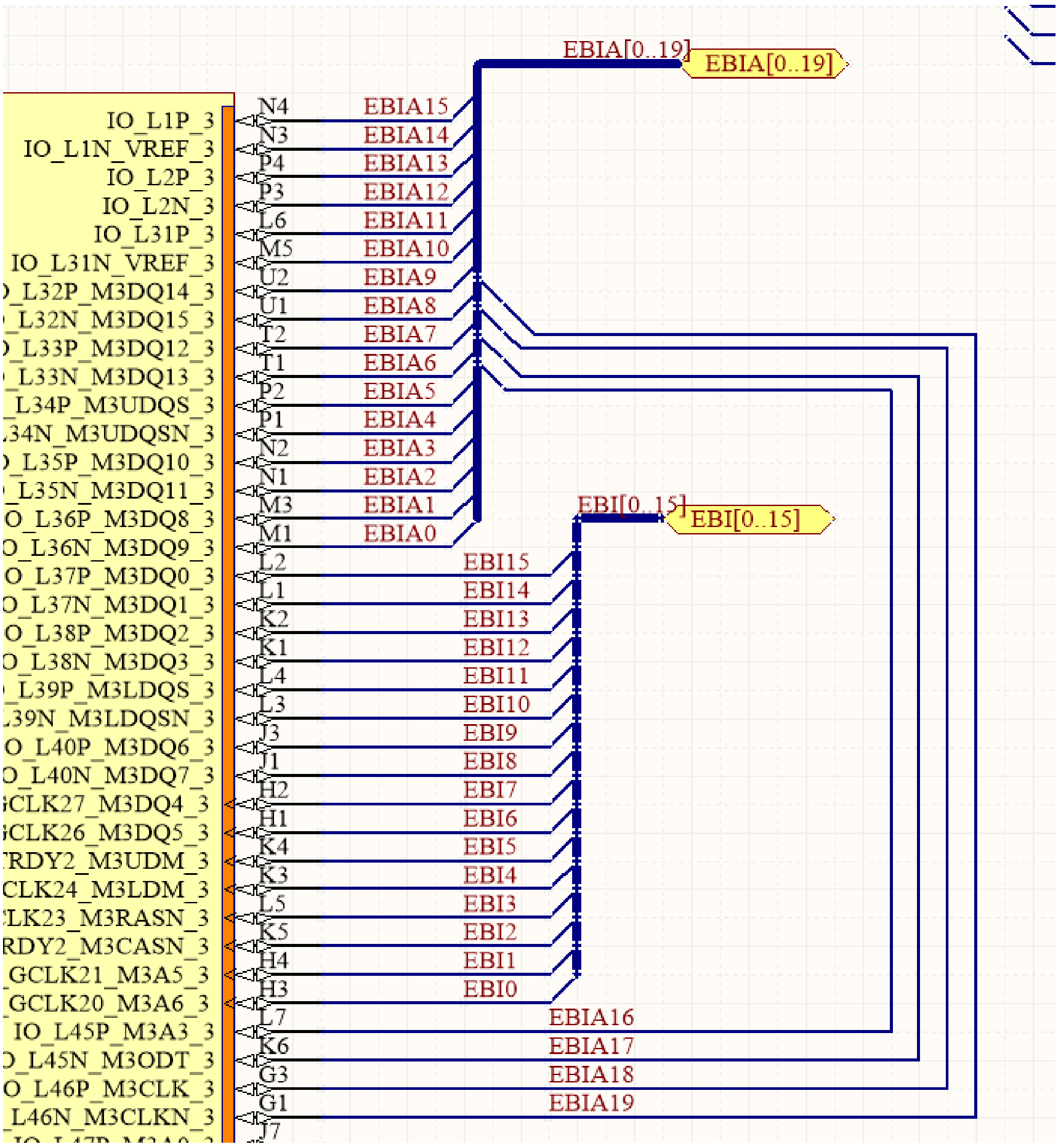


Figure 2.4: EBI-Bus input

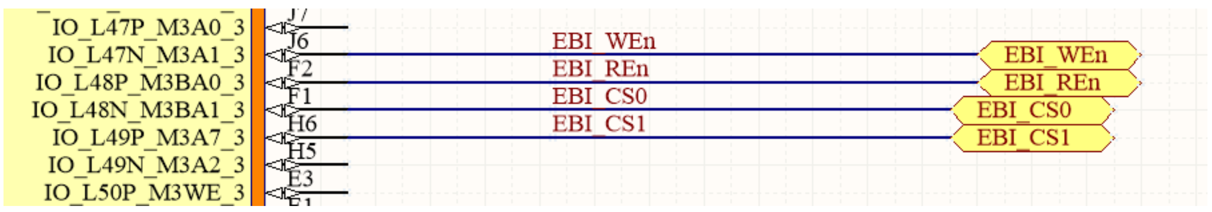


Figure 2.5: EBI-Bus controll signals

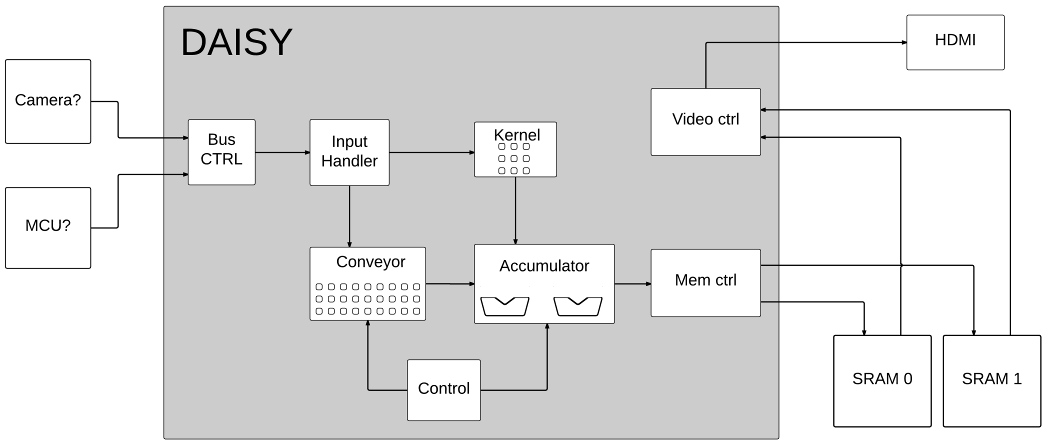


Figure 2.6: The data path for our processor, named daisy after the way it daisy chains control

6. After being buffered in SRAM pixel data is read by the video ctrl module which outputs video to an HDMI cable, ensuring crisp image quality served in a modern fashion.

2.4.3 Data in

Fig:Sweeps shows the pattern in which data is being fed into the processor. A good analogy is how a squeegee (nal) is used to clean a window, by doing either horizontal or vertical sweeps. In fig:SweepFrontier we show the area of the image we are currently working on. Our workarea, or sweep window, is nine pixels wide and three deep, and for each row the convolutions that have the middle row as center pixels are calculated.

2.4.4 Convolver

This section encompasses the four modules forming the heart of the convolution engine as depicted in fig:DaisyView. In the order they are covered, they are:

the conveyor belt
: A buffer responsible for maintaining the three rows of the sweep window and to feed the accumulator with three reads each cycle, one read from each row every cycle. The three rows must be read in such a way that each accumulator may collect the nine subpixels it needs per convolution by reading each conveyor output at the correct time.

the kernel buffer
: The kernel buffer in the figure is an abstraction, in the implementation the kernel buffer resides much closer to the accumulators, but we separate it in our overview to show how each ALU in the accumulator reads kernel values.

the accumulator
: The accumulator is being fed data over three wires, and by choosing which wires to read from ensures that each accumulator works on the correct subpixels.

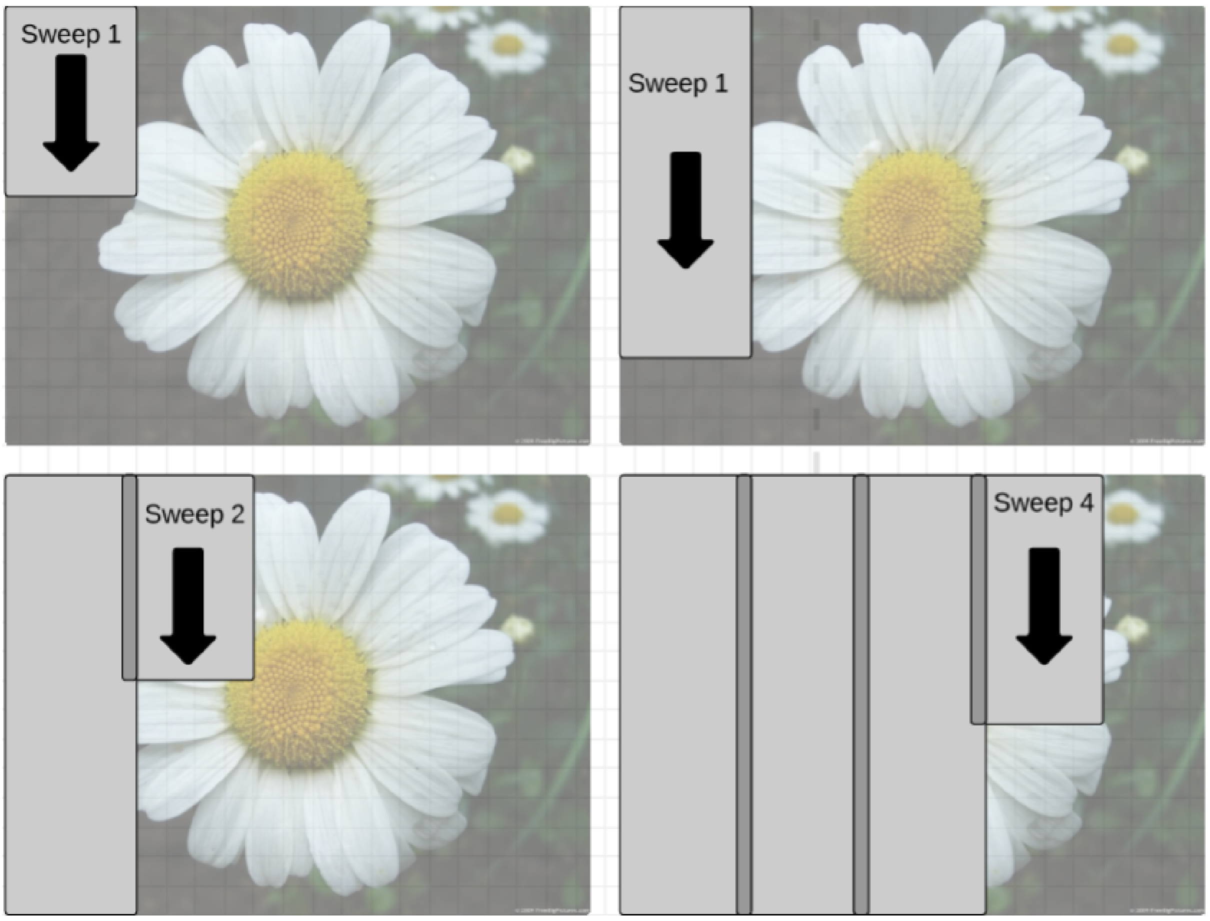


Figure 2.7: The sweep pattern used to collect data for convolution. The deep grey region indicates overlap, pixels which will be collected twice

the 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.

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

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.

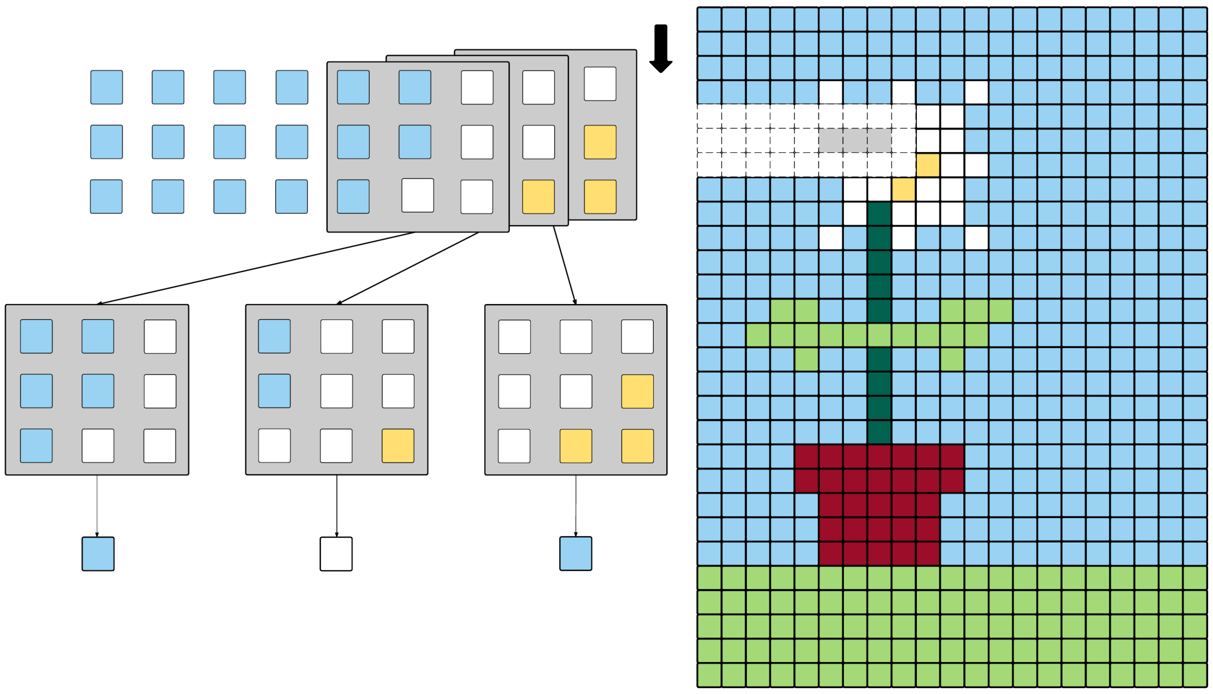


Figure 2.8: The image three greyed out pixels in the source image represent the three pixels which the regions in grey boxes help calculate. The window is three pixels deep and nine pixels wide

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 following 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.

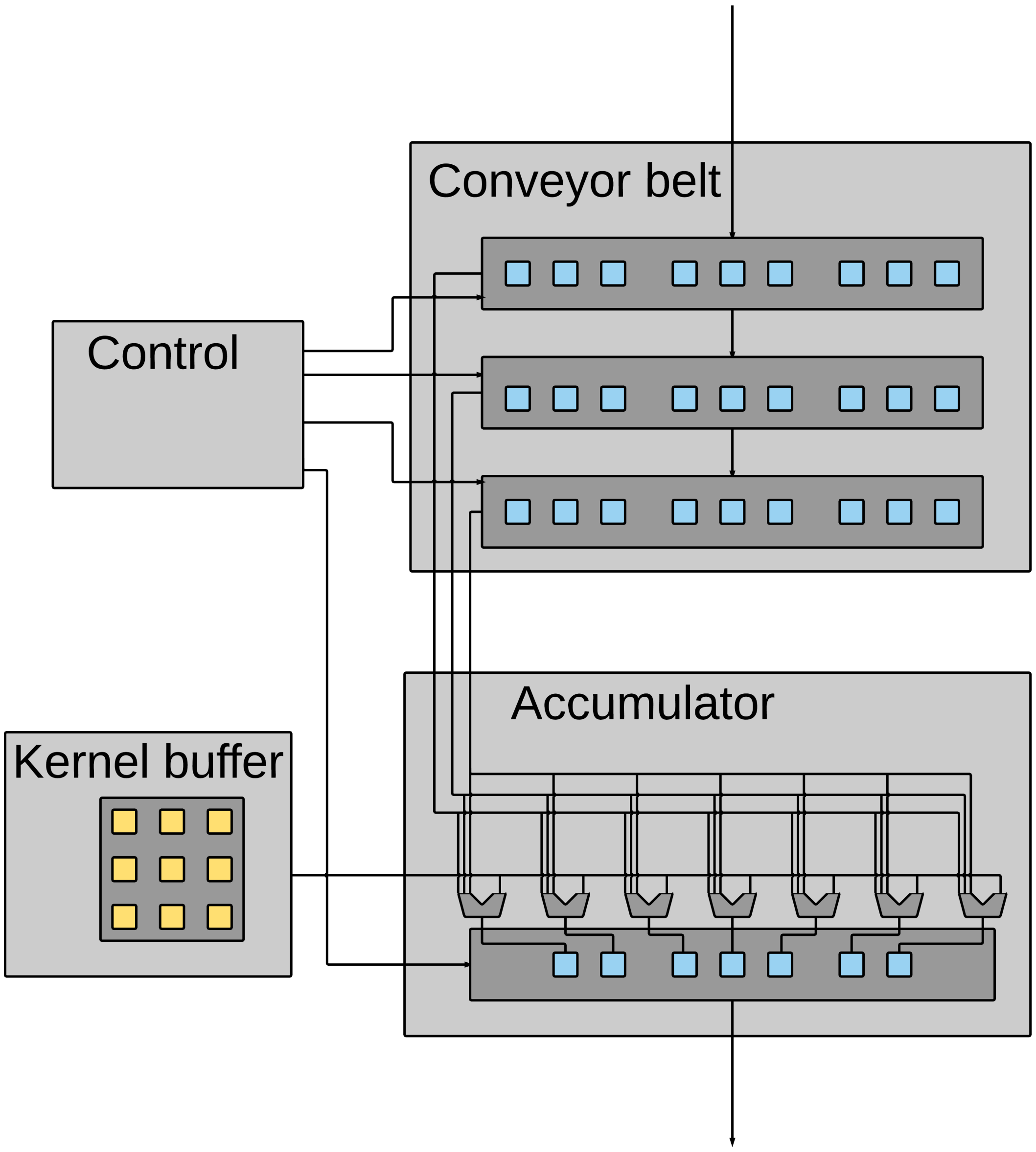
TODO: Figure out a succint way to explain this part

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

3 | Results & Discussion

3.1 Testing

3.2 Results

3.3 Discussion

3.4 Conclusion

References

[1] Edgar Xavier Ample. Foo. Technical report, Foxford University, 3000.

[2] Mathias Ose. ma.thiaso.se. `http://ma.thiaso.se/`, 2014.