256 SHADES OF GRAY

Christer Emil Haga Bru
Vegard Edvardsen
Sondre Andreas Engebråten
Hans Kristian Flaatten
Martin Gammelsæter
Jean Niklas L'orange
Erik Lothe
Andreas Steensnæs Morland
Mads Buvik Sandvei
Einar Johan Trøan Sømåen
Lichao Tang
Håkon Opsvik Wikene
Trygve Aaberge

TDT4295 Computer Design, Project Work
Department of Computer and Information Science
Computer Architecture and Design
Norwegian University of Science and Technology

[November 19, 2012 at 13:56 – classicthesis 0.1.0-alpha03]

ABSTRACT

As heat generation and power consumption starts to limit speed and effectiveness of single core architectures, parallelism is becoming more and more important within all performance demanding computing. Everything from cell phones to desktop computers now carries multiple processing cores to perform their functions. Parallelism is of course not a new concept; for many decades, exploiting the parallel nature of certain data to effectivize computation has been a known tactic. Especially within the field of image processing, parallelism has proved itself to be an extremely effective tool.

While Multiple Instruction, Multiple Data (MIMD) systems are dominant within general computing, we wanted to explore alternatives which may work better for certain applications. The result of this project is the computer "256 Shades of Gray", which is an array-based Single Instruction, Multiple Data (SIMD) architecture optimized for massively parallelizable tasks such as image processing.

256 Shades of Gray consists of a custom PCB design, a microprocessor operating as a System Control Unit and an FPGA which implements our SIMD architecture. It takes input through an SD card reader (with RS-232 and USB as backup), and shows output through a custom VGA controller.

We have been inspired by the Goodyear MPP architecture, which clearly shows in our design. Our main focus was performance, and because of that simplicity in design has been strived for in order to fit as many parallel cores on our FPGA as possible. As such, the SIMD nodes have been stripped of many complex instructions, such as integer multiplication and division, as well as all floating point capabilities. The SIMD array is controlled by a control core that handles I/O and memory access through a DMA unit.

Our design has gone through testing, and has met all functional and non-functional requirements that were set. The architecture has shown itself effective and well suited for common image processing tasks.

ACKNOWLEDGMENTS

Parallel machines are hard to program, and we should make them even harder – to keep the riff-raff off them.

— Gary Montry

We would like to express our appreciation to the following people for advice and assistance throughout the project:

MAGNUS JAHRE for providing valuable feedback and project coordination.

STEFANO NICHELE for administrative work.

GUNNAR TUFTE for great input and pointers on PCB design.

LARS-IVAR HESSELBERG SIMONSEN for general advice and letting us know everything was better last year.

MARIUS GRANNÆS for assistance, advice, and help to speed up SD card reading.

THE FORTITUDO FLORIS PROJECT for healthy competition.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF LISTINGS

ABBREVIATIONS

ASF Atmel Software Foundation

EBI External Bus Interface

FPGA Field-Programmable Gate Array

GPIO General Purpose Input/Output

JTAG Joint Test Action Group

LENA Lightweight, Efficient Node Array

MIMD Multiple Instruction, Multiple Data

PCB Printed Circuit Board

PDCA Peripheral DMA Controller

SCU System Control Unit

SDHC Secure Digital High-Capacity

SDRAM Synchronous Dynamic RAM

SD Secure Digital

SIMD Single Instruction, Multiple Data

SMC Static Memory Controller

SPI Serial Peripheral Interface Bus

USART Universal Synchronous/Asynchronous Receiver Transmitter

VGA Video Graphics Array

VHDL VHSIC Hardware Description Language

VHSIC Very-High-Speed Integrated Circuits

1

INTRODUCTION

The computing scientist's main challenge is
not to get confused by the complexities of his own making.

— E. W. Dijkstra

1.1 ASSIGNMENT

The task given was to create an *array-based* parallel image processor with focus on performance. An array processor is a grid of processing elements, where each of the processing elements are only able to communicate with its north, south, east and west neighbors. All the processing elements will perform the same instruction at all times, which means that the more processing elements one has, the more data is processed simultaneously, leading to better performance.

Fast image processing is essential in robots[10, 14] and in artificial intelligence[7] in order to work in the real world. Autonomous cars and robots need to process images from image sensors fast enough to react and e.g. prevent accidents[1]. Image processing in general is also highly applicable in the medical field[9, 13] and in the petroleum industry[5].

1.1.1 *Original Assignment Text*

The original assignment was given as follows:

> *The performance increase available from harvesting Instruction Level Parallelism (ILP) from the serial instruction stream is limited because we have reached the maximum power consumption that can be handled without expensive cooling solutions [12]. Consequently, there is a significant interest in single-chip parallel processor solutions (e.g. [3, 8]).*
>
> *The processor cores in commercial multi-core chips are conventional designs and therefore reasonably complex. In this work, your task is to design an array-based parallel image processor.*

An array processor is organized as a matrix of processing elements where each element communicates with its neighbors in the north, south, east and west directions.

Your image processor will be implemented on an FPGA, and you are free to choose how to realize your array-based computer architecture. The system should be shown to work with a suitable application. Studying the architecture of the Goodyear MPP [2, 15] might be a possible starting point. Due to a large number of students this year, we will divide the work into two independent projects: a) Performance and b) Energy efficiency. The goal of group a) is to achieve maximum performance while group b) should try to balance performance and energy. The reports from both groups should include an evaluation of prototype performance and energy consumption.

Additional requirements

The unit must utilize an Atmel AVR micro controller and a Xilinx FPGA[1]. The budget is 10.000 NOK, which must cover components and PCB[2] 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.

1.2 FOCUS ON PERFORMANCE

Because the assignment task of image processing was combined with a focus on performance, we decided early on that we wanted our system to be able to process *video*.

After estimating what kind of demands various video qualities would place on our system, we decided on aiming for 320×240 pixel resolution, 8 bit grayscale video at 10 frames per second. This would require a data throughput rate of about 768 kB per second throughout our system. Given an FPGA design clocked at for example 25 MHz, we would have about 30 cycles available to process each pixel. This seemed like reasonable yet challenging constraints.

1.3 REQUIREMENTS SPECIFICATION

A few functional and most non-functional requirements were given to us by our instructor, Magnus Jahre. The rest were decided by the

1 Field-Programmable Gate Array
2 Printed Circuit Board

NAME	DESCRIPTION	PRIORITY
FR1	The image processor should be fast enough to output unmodified 8-bit images of at least 320×240 resolution at 10 frames per second.	High
FR2	The image processor should be generally programmable.	High
FR3	The machine should have a video port for image output.	High
FR4	It should be possible to operate the machine with on-board buttons.	Medium
FR5	The machine should have on-board LEDs for visual feedback.	Medium
FR6	There should be developer tools (assembler) available for the machine.	Low
FR7	The machine should have example programs demonstrating its capabilities, such as a video player and a median filter.	Low

Table 1.1: The functional requirements

group as goals we thought were realizable, and some to help us see how far away we are from our goals. Table 1.1, which shows the functional requirements, includes a relative priority between the different requirements. This priority tells us what we have focused on, as well as what is important in terms of success of the computer. Clearly, focusing on performance, as specified in FR1, is more important than having developer tools for the machine (FR6).

FR1 ensures a focus on performance. FR2 ensures that we do not end up with a system which is not generally programmable. This is important, as a computer specialized for image processing has less usability than a generally programmable computer. FR3 and FR7 makes it easier to show that the computer is capable of processing images, whereas FR4 through FR6 makes it easier to use, debug and create programs for the computer.

Table 1.2 shows the non-functional requirements. These were all determined by the assignment. They were all treated as absolute requirements, and our design has been centered around those. NFR1 and NFR2 gave us little choice in what hardware to use, and naturally this hardware have been used. NFR3 limits what components we can use to the cheaper ones. NFR4 Constraints our design choices when designing the image processor. We could not make any type of core organization, we had to make this specific type. NFR5 leads to some design choices for the entire system. Our goal was to get the best possible performance, and we did not have to care about concerns

NAME	DESCRIPTION
NFR1	The machine must use a Xilinx Spartan 3 XC3S500E PQG208 FPGA
NFR2	The machine must use one AVR32 UC3A microcontroller
NFR3	The budget of $\sim$ 10 000 NOK must cover all PCB and component costs
NFR4	The image processor should consist of multiple cores arranged in a matrix
NFR5	The machine should be optimized for performance

Table 1.2: The non-functional requirements

such as energy consumption. However, we may have sacrificed some simplicity in our design for the sake of performance.

1.4 STRUCTURE OF THE REPORT

This report may be viewed as divided into three parts: chapters 1, 2-5 and 6-9. The current chapter has given you an introduction to the assignment and our goals.

Chapters 2 to 5 explains in deeper detail *how* the machine works. Chapter 2 gives an overview of the whole machine and how the different parts are connected together. Chapter 3 elaborates on the Lightweight, Efficient Node Array (LENA) Architecture, which is the FPGA design of the machine. SIMD nodes, control core as well as the FPGA Video Graphics Array (VGA) module is explained in detail, along with the interconnection between these. Chapter 4 describes the software and structures for the System Control Unit (SCU), such as the file format and the program and data menu. Chapter 5 explains the work with designing the PCB, as well as the soldering and other workarounds we had to do.

Chapters 6 to 9 is the conclusion part of the report. Here we test the machine, discuss our choices and present our conclusion, as well as possible further work. Chapter 6 contains tests to check whether we have managed to reach our requirements or not, along with their results. Chapter 7 describes the result, performance and energy consumption of the machine with different cores and different programs. Chapter 8 discusses different choices we did in our project, and the result of these. In chapter 9, our conclusion is given along with further work.

2

SYSTEM OVERVIEW

On the other side of the screen, it all looks so easy.

— Kevin Flynn (TRON, 1982)

2.1 SYSTEM ARCHITECTURE

A general concern when designing large systems is the accidental complexity[6, p. 8-9] one may create by poor design choices early on. Many solutions designed to reduce accidental complexity are based around software systems, and sacrifices performance in both the time domain and in the space domain[11]. While these solutions may be applicable within hardware systems where these kinds of performance degradations are not a problem, it is unacceptable in systems where one or multiple of the system requirements are a performance increase in one or both of these domains. As one of our main requirements is focus on performance, we have to accept a certain level of inherent complexity.

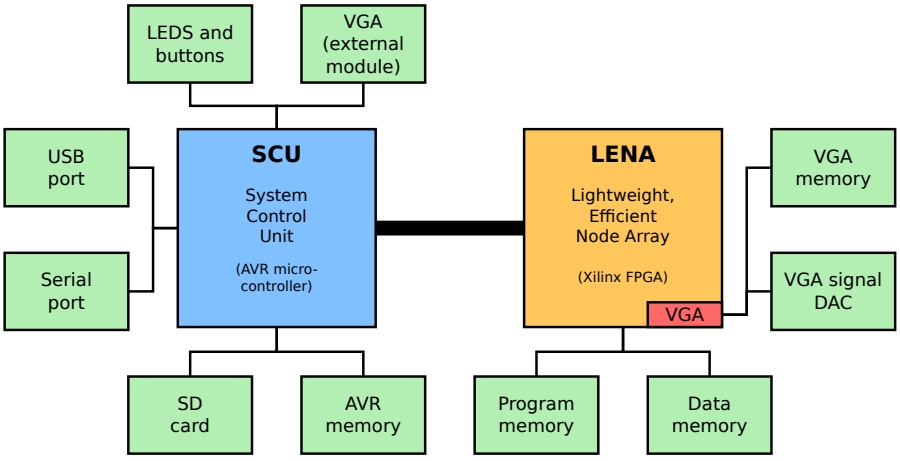

Figure 2.1: System Architecture

To remedy the complexity which inevitably follows from our requirements, we focused on making it possible to isolate errors and make it possible to test the individual components as early as possible: The

LENA architecture was tested through VHSIC Hardware Description Language (VHDL) test benches, the VGA component was tested with prototypes on a breadboard, and the SCU was tested with an EVK1100 development board as well as with the buttons and LEDs on the PCB once it arrived. By doing this, we could assume that errors occurring when connecting different components are mostly due to errors in the protocol implementation(s).

Figure 2.1 shows our resulting architecture. We also included a VGA connector connected to the AVR, in case the LENA architecture should fail to implement its VGA module.

To be able to process data, we needed an Input/Output (I/O) device which could serve the machine with data it should process. To ensure that we would have at least one functional I/O component, we included both a serial port, a Universal Serial Bus (USB) port and an Secure Digital (SD) card reader on the PCB. In addition, to be able to keep enough data in memory and allow full overlap of different memory transactions to make processing fast, the LENA architecture has three separate memory components. One for VGA, one for instructions and one for data.

CHECK: *Sounds okay now?*

2.2 COMPONENT FUNCTIONALITY

This section describes the different components and their functionality. A list of ordered parts can be found in B.2.

2.2.1 *System Control Unit*

The System Control Unit (SCU) is used to control the LENA architecture and as a user interface. The SCU sends data and instructions from the SD card to LENA, which stores it in it's data and instruction memory, respectively. The SCU also starts and stops LENA's program.

Selecting programs and data is done by the user interface on the SCU using buttons as input and LENA's VGA as output.

2.2.2 *LENA*

NF_4 constrained our high level choices on the image processor architecture: There was a requirement that we had to have multiple cores arranged in a matrix. As the matrix would perform image processing,

it was natural for us to choose a SIMD architecture. Many image processing algorithms do the exact same operation on every pixel, and having a SIMD architecture reduces both complexity and size needed per core on the FPGA significantly.

Other design choices that followed was the introduction of a control core, a Direct Memory Access (DMA) and a VGA controller. The control core is responsible for sending data to the SIMD nodes and the VGA controller, whereas the DMA is responsible for writing data from the SIMD nodes back into memory. The VGA controller is responsible for handling the VGA memory and sending the correct signals to the VGA port. In addition, as the SIMD nodes usually depend heavily on their neighbor's data, we decided to have "dummy nodes" outside the real SIMD nodes. Their only function is to transmit data to the edge nodes. As such, we can still utilize the edge nodes for computation when neighbor data is needed.

2.3 AN IMAGE'S ODYSSEY THROUGH THE SYSTEM

To summarize the previous sections, let us take a quick example of how the complete system will process a video, an image or something like the Game of Life. We will not go into deep details, and as such, some figurative language is used to simplify complex parts of the system.

At first, the machine is powered up, and all the components wake up from their sleep. LENA will reload its FPGA setup from flash, and the SCU will do the same. When these are ready to run, the SCU creates a menu and sends it to LENA, which will in turn show the menu trough the VGA port it has control over. Whenever a user pushes any of the buttons, the menu is updated and a new menu image is sent over to LENA.

When we have chosen the program to run and the data it should run on, the SCU sets the LENA state to a state where it is able to receive instructions. It then loads the instructions for the given program to LENA. The SCU then sets the state to a state where it is able to load data, before it does exactly that. Finally, the SCU allows LENA to run. The SCU will feed LENA with more data after some time, depending on what data stream we have chosen and what program we have decided to run.

LENA now has all it needs to perform its instructions. The control core starts by activating the DMA. The DMA starts sending data from memory to the SIMD array. This is done by putting the different values on multiple "conveyor belts" which lies below the SIMD node. When

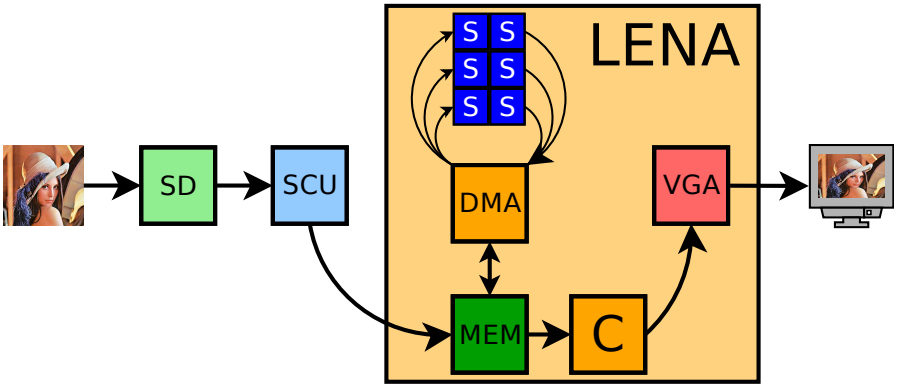

Figure 2.2: The Journey of an Image

every SIMD node has data to work with, every node "picks up" the data and places it in a special register.

The control core then sends the instruction the SIMD nodes should use next. In parallel, the DMA starts sending in new data into the SIMD nodes, to minimize latency. When the SIMD nodes have finished processing this data, they swap the finished result with the new data on the "conveyor belt". The DMA puts on new data, and the finished results are moved on and out of the "conveyor belt". The DMA takes the finished result and puts it back into the data memory.

The control core then takes the results from memory and sends them to the VGA module, which finally sends the image to the device connected to the VGA port. The device, a screen, for instance, then displays the processed image.

The odyssey has ended for our image, but there is still work to be done. If we are playing a video, more images will be processed in the way described above. If we are doing something entirely different — for instance, playing Game of Life — we may just keep working on the result from last iteration until the end of time. If we stop before that, the SCU will stop LENA by turning its state to a stop state. The SCU will then set up the original menu, and our machine is ready to let new images go on another odyssey with another program.

3

THE IMAGE PROCESSOR ARCHITECTURE

The way the processor industry is going, is to add more and more cores,
but nobody knows how to program those things.
I mean, two, yeah; four, not really; eight, forget it.

– Steve Jobs

3.1 INTRODUCTION

We have named our image processor LENA (Lightweight, Efficient Node Array). Its main components are a SIMD array consisting of 6 cores, a control module to manage the data, and a VGA controller. LENA is responsible for receiving data from the SCU, processing it and sending the output to the VGA port.

LENA is implemented on a Xilinx Spartan-3E XC3S500E PQG208 FPGA. The FPGA uses a clock rate of 50 MHz for the VGA memory controller and 25 MHz for the VGA signal generator and the processor cores.

The first section in this chapter gives an overview of the image processor architecture. The following sections describe the SIMD array, the control module and the VGA controller in further detail.

3.2 ARCHITECTURE

An overview of the architecture of LENA as well as its memory and communication lines is shown in Figure 3.1. The data flows mostly through it in one direction. Data is received from the SCU and stored to the data memory. The data is then read in parts from the data memory and sent through the SIMD array for processing. The result from the SIMD array is stored to the data memory again. At last, the data is copied from the data memory to the VGA memory and shown on screen.

TODO: *Say why the DMA doesn't send data to both data and VGA ram, yet have to use control core instead. (Supposedly because the output image is more stable this way?)*

The processing part of LENA is the SIMD array. It is organized in a two-dimensional matrix, as specified by the assignment. Each node

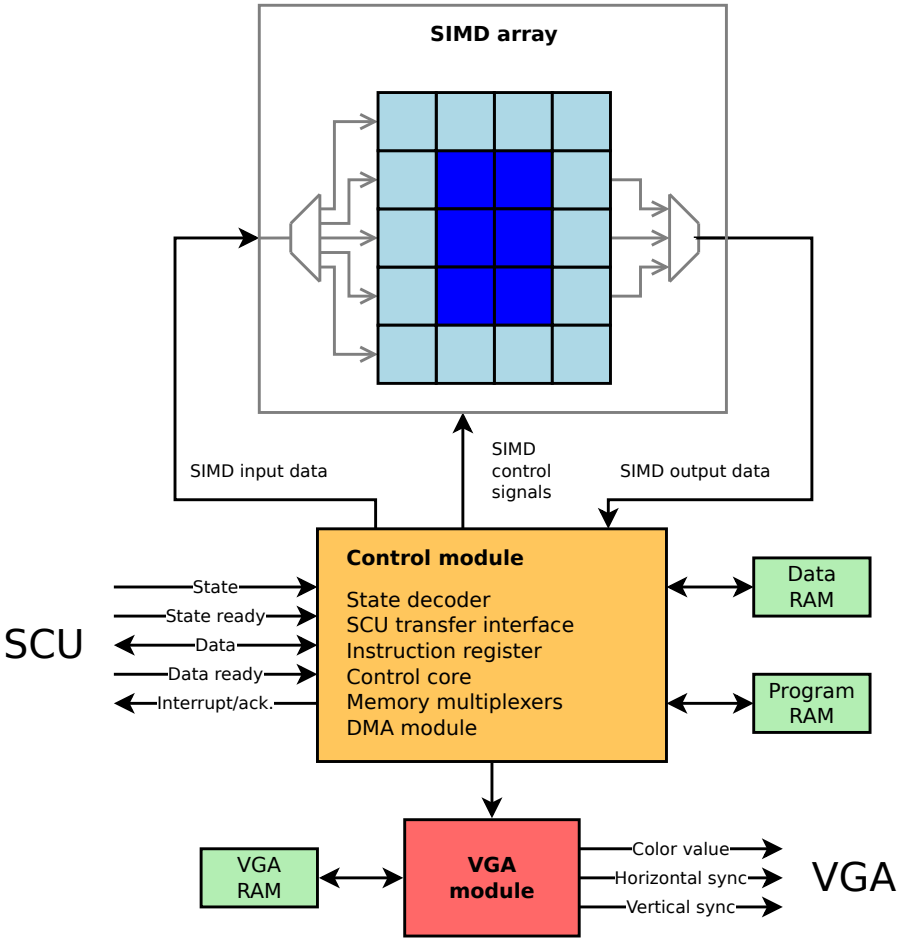

Figure 3.1: An overview of the LENA image processor architecture.

processes one word at a time and can communicate with its four direct neighbors. In order for the nodes to receive valid data from the edges where they have no neighbors, special *edge nodes*, that only hold data values, are inserted along the edges.

These features make the architecture especially well suited for applying 3x3 image filters, such as noise reduction or edge detection filters. See Appendix **??** for an example implementation of an edge detection filter.

TODO: *JN: Fix reference AFTER Minted-merge*

To first load the data into the SIMD array, it is sent to the first column of the nodes and copied to the right, passing through all the nodes until the last column. This is done by using a special register for sending data, called the S register. Data is continuously received by the first column, so the whole array is filled. At that time, data processing can start. The data is swapped onto another register of the node, and new data can be loaded through the S register. When both data processing and transmission of new data is done, the registers are swapped again. At this point, we have results in the S registers and can send this out to

If this could be shortened somehow and fixed in the simd.tex-file, that would be great. However, it has low impact and requires MUCH work

the right while loading new data from the left. The results are copied out of the array from the last column.

Data communication between SCU, the SIMD nodes and the VGA controller is handled by the control module which communicates with the data- and program memory, and keeps track of the data location on these. The control module sends instructions to all of the SIMD nodes and is responsible for splitting the data and loading it into the SIMD array in correct order by setting the S registers of the first column. It also retrieves the results from the last column of the array and stores it to the data memory.

The FPGA receives data from the SCU on 24 input lines. This is enough to transmit one instruction per transmission. All of the instructions are copied into the instruction memory before any other data is sent. The image data is sent one word (8 bits) at a time. We considered using all 24 lines for data transmission, but as we tested the transmission rate, we realized that it was fast enough by only sending one word at a time. This was easier to implement and was therefore the chosen implementation.

The SIMD array is created dynamically in VHDL, hence the number of cores is easily changeable. The final number of cores in our implementation is only limited by the available space on the FPGA, and we have tried to fit as many nodes as possible. On the Spartan-3E XC3S500E FPGA we had enough space for 6 nodes, in addition to the 14 corresponding edge nodes, with a total slice utilization of 93 %.

3.3 SIMD NODES

SIMD nodes are the main processing elements of the image processor and are responsible for carrying out image processing instructions in parallel. They share the same instruction set and executes them in parallel, hence Single Instruction, Multiple Data (SIMD).

Each SIMD node is fully equipped with a register bank, an Arithmetic Logic Unit (ALU) and data exchange with adjacent nodes. Everything inside the node is operated through the SIMD node instruction set explained in **??**.

The schematic of a single SIMD node is shown in Figure 3.2, and details the entire data-path for the node. External inputs to the node are drawn on the left side. Data-buses are marked as blue lines while control signals are red dashed lines. In the following section we will refer to this image when explaining the different components.

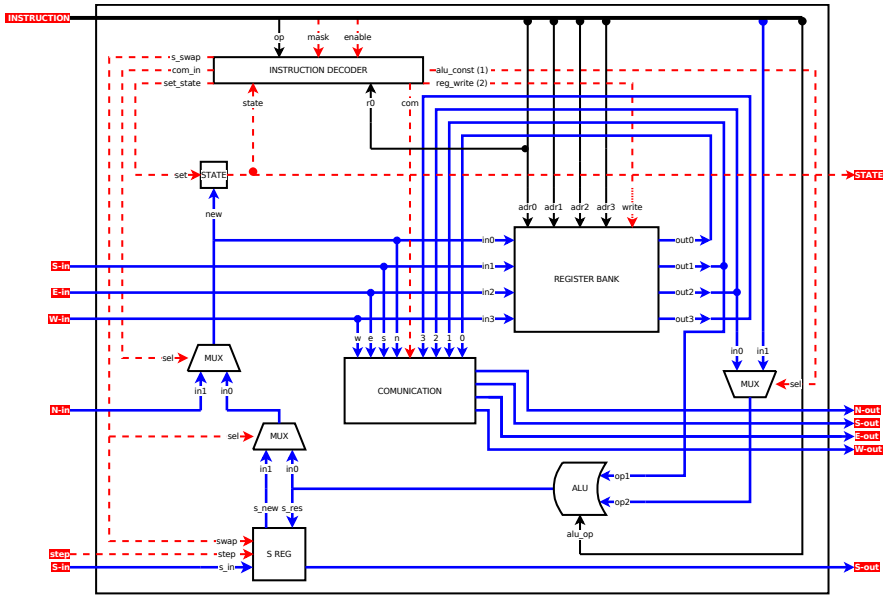

Figure 3.2: LENA SIMD architecture

3.3.1 *Components*

In this section we briefly describe the different components, their purpose, and how they together form the data-path for the node. All components has been thoroughly thought out and design to suit the SIMD nodes.

3.3.1.1 *Instruction Decoder*

Whenever a new instruction is received by the SIMD node it is immediately decoded by the instruction decoder. Based on the enable-bit, mask-bit and OP-code of the instruction correct control signals are set up for the remaining components throughout the node.

For a list of all SIMD instructions, see Appendix A.1.

3.3.1.2 *Communication*

The communication component controls outbound data to adjacent SIMD nodes in the grid array; north, south, east and west. The communication component is fairly simple and can either send register data or forward inbound data according to the counterclockwise forwarding scheme described in 3.3.2.

3.3.1.3 *Register Bank*

Each SIMD node is equipped with $2^4 = 16$ 8-bit general purpose registers, 14 of which are available for general storage during execution. The remaining 2 are reserved for the special purpose registers `$zero` and `$state`.

R0	R1	R2	R3	R4	R5	R6	R7
`$zero`	`$r1`	`$r2`	`$r3`	`$r4`	`$r5`	`$r6`	`$r7`
R8	R9	R10	R11	R12	R13	R14	R15
`$r8`	`$r9`	`$r10`	`$r11`	`$r12`	`$r13`	`$r14`	`$state`

Table 3.1: Registers in the SIMD nodes

The register bank has been specially designed to suit the four-way data transfer by allowing 4 registers to be read, or written, at during one clock-cycle.

3.3.1.4 *ALU*

Each SIMD node has an 8-bit simple ALU supporting 8 arithmetic instructions: addition (ADD), subtraction (SUB), set less than (SLT), AND, OR, equality check (EQ), shift logical left (SLL) and shift logical right (SLR).

The main reason for only implementing the most basic arithmetic instruction was to save space in the instruction set since addition, subtract, equality and less then are supported by special hardware inside the FPGA. All SIMD node instructions carries a 3-bit ALU function at the end.

Upon researching image processing algorithms well suited for parallel processing, we found that most of then could be implemented using only these 8 ALU operations. Image processing algorithms such as median filter, various blur filters, edge detection, color manipulation such as inversion and thresholding are fully feasible using. More high level arithmetic, such as multiply and divide by constants, can be implemented through a series of additions, subtractions and shifts.

CHECK: *Proof-read this section!*

CHECK: *Add some reffernces?*

CHECK: *Talk about general programability?*

3.3.1.5 *Source Data Register*

The source data register, or just **S-REG!** (**S-REG!**) for short, is a special purpose register within the SIMD node. It is a communication unit,

separated form the rest of the node so it can transfer new and processed image data through the array while the SIMD node is otherwise busy executing instructions.

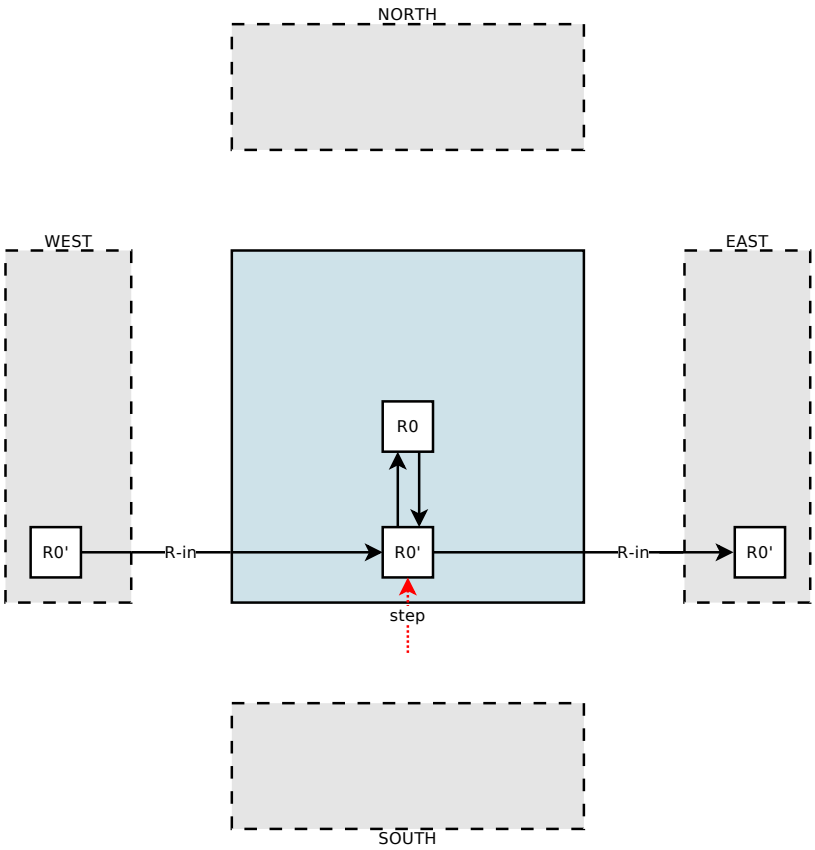

Figure 3.3: Source data flow through the LENA SIMD array.

The **S-REG!** is partly controlled by the SIMD node instruction set where it can swap data in and out from the **S-REG!** and partly by a special `step` signal sent from the DMA.

3.3.1.6 *State Register*

In order to handle branches, through masked instructions, each SIMD must have an internal state register. Even though the register is 8-bit wide only the least significant bit is the current state for the node.

The state register can be written and read as any other register in the register bank and the state can hence be shifted left or right in order to achieve nested branches.

3.3.2 *Communication*

Data exchange between SIMD nodes are multiplexed in all directions. SIMD nodes are capable of sending source data in all directions during one clock cycle and storing them within the next as shown in Figure 3.4.

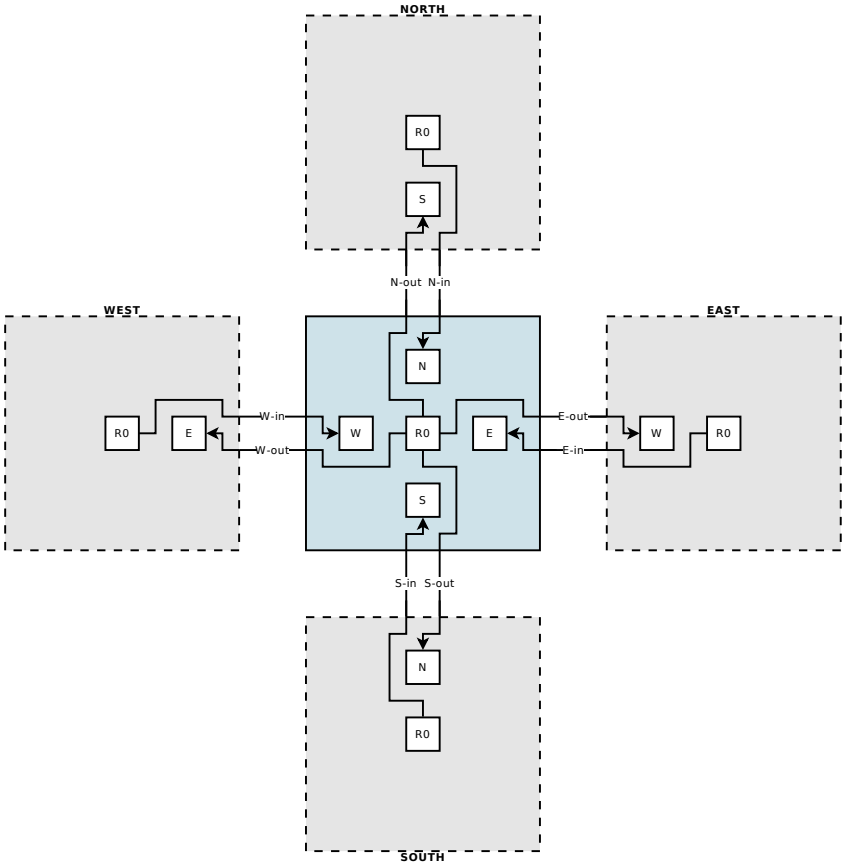

Figure 3.4: Sending and recieving data from and to adjacent nodes in the LENA SIMD array.

3.3.2.1 *Forwarding*

The counterclockwise forwarding, illustrated by Figure 3.5, is an optimized way we devised for distributing a 3×3 array of source data to all SIMD nodes in the SIMD array using only 3 clock cycles; send, forward and sore, and store.

TODO: *We invented this one by ourselves! Should we brag some more?*

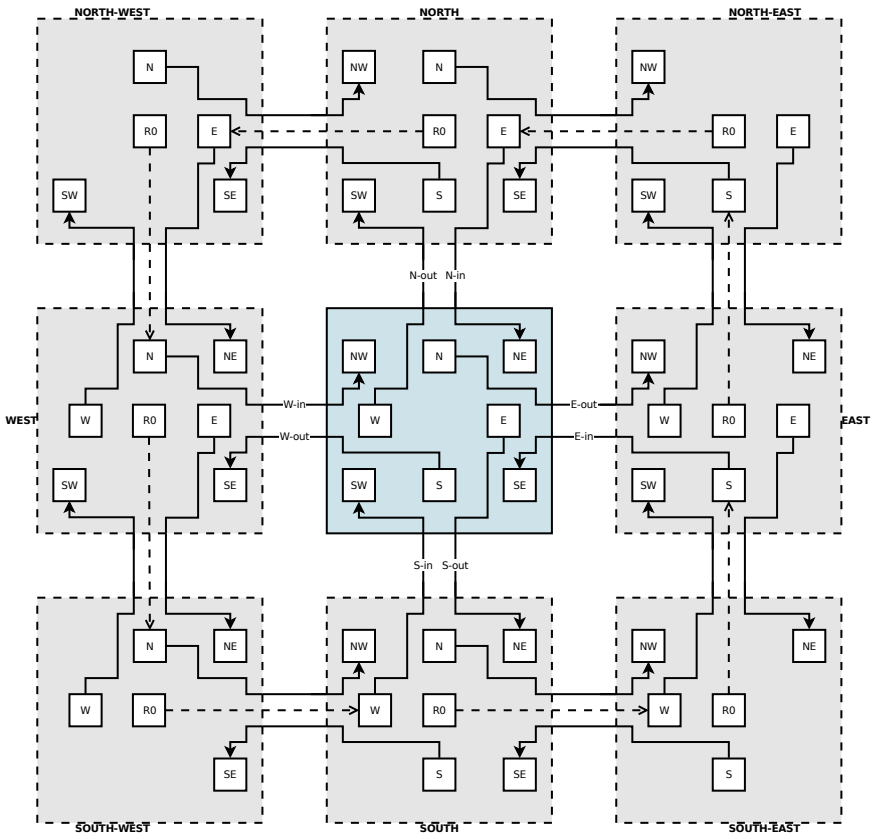

Figure 3.5: Counterclockwise forwarding for nonadjacent nodes in the LENA SIMD array.

3.3.3 *Instruction Set*

The SIMD node instruction set, detailed in Appendix A.1, controls all aspects of the SIMD node. It has been designed with regularity in mind in order to keep the number of special cases in the hardware to a minimum with as few formats as possible.

Much of the design behind the instruction set is borrowed from **MIPS!** (**MIPS!**) which also favors regularity in order to simplify the hardware, thus enhancing the performance.

3.3.3.1 *Branching*

Since all nodes run the same instructions, both parts of a branch must be executed. Nodes are setting the `state` to 1 in order to indicate that they are executing within that part of the branch. An example of single branching is shown in Listing 3.1.

```
    node eq STATE R1 R2       # set state
      node mask # ... # Instructions for when the
      node mask # ... # condition is satisfied
    node eq STATE STATE ZERO # negate state
      node mask # ... # Instructions for when the
      node mask # ... # condition is not satisfied
    node move STATE ZERO      # Set state to zero
    # node addi STATE ZERO 1 # Or one, if you want

    # code after branching would follow here
```

Listing 3.1: Single level branching in SIMD nodes

Since the state register is 8 bits, it is possible to have up to 8 nested branches by shifting the current state left and adding the new state to the end. Listing 3.2 contains code which performs such a multilevel branch.

```
    # Initial state.
    node sll R3 STATE       # Save current register
                            # by shifting left
    node eq R4 R1 R2        # Calculate if branch is taken
                            # by the node
    node add STATE R4 R3    # Set the new state
      node mask # ... # Instruction for when the
      node mask # ... # condition is satisfied
    node andi R3 STATE 254 # Save old state
                            # (254 = 1111 1110)
    node andi R4 STATE 1    # Save new state
                            # (001 = 0000 0001)
    node eq R4 R4 ZERO      # Negate current state
    node add STATE R4 R3    # Set new state
      node mask # ... # Instruction for when the
      node mask # ... # condition is not satisfied
    node srl STATE STATE    # Revert to old state

    # code after branching would follow here
```

Listing 3.2: Multilevel branching in SIMD nodes

The control module is responsible for coordinating the activity of the other parts of the image processor and for data transfer into and within the image processor. A visual overview of the control module is given in Figure 3.6.

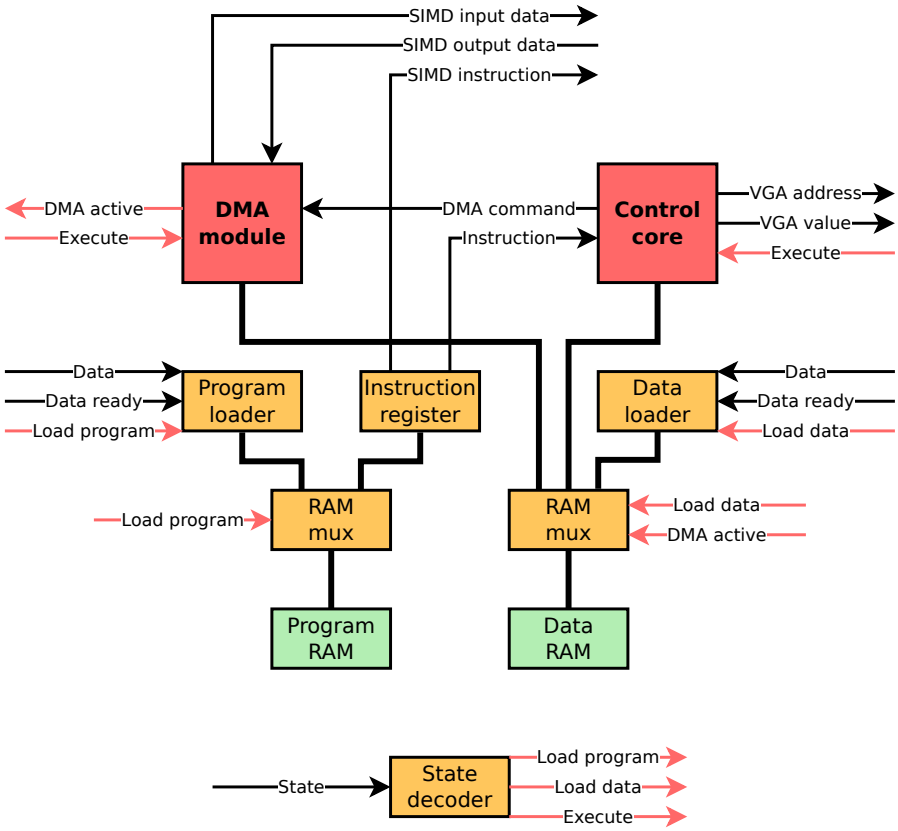

Figure 3.6: An overview of the control module of the image processor.

The control module decodes the state signal set by the SCU, and enables, disables and resets components accordingly. It is responsible for performing data transfers between the SCU and the program and data memories, and for transferring data internally between the SIMD node array, data Random Access Memory (RAM) and the VGA screen buffer.

TODO: *Isn't this the DMA's work?*

The control module contains the Central Processing Unit (CPU) that is used for the non-SIMD instructions embedded in a program. This CPU, henceforth referred to as the *control core*, is mainly concerned with loop control, initiating data transfers between the SIMD node array and the data RAM, and copying image data to the VGA controller.

3.4.1 *States*

The operation of the image processor is fully controlled by the SCU. The SCU sets a state value that is used by the control module to select which components of the image processor are active. This is indicated by the red signals in Figure 3.6.

CHECK: *Må skrives ut i farge om vi skal ha dette med*

The states recognized by the image processor are listed in Table 3.2.

STATE	DESCRIPTION
000	Idle
001	Run program
010	Load data from SCU
100	Load program from SCU

Table 3.2: Image processor states.

3.4.2 *Control Core*

A schematic overview of the control core is given in Figure 3.7. The control core consists of a register bank of 21 bit registers, an ALU and a connection to the data RAM. The word width of 21 bits was chosen to match the address width of the data RAM. This allows the control core to do pointer arithmetic referring to arbitrary words in the data RAM.

To send pixels to the VGA controller, two of the registers are dedicated to contain an address and a pixel value, respectively. The contents of these registers are constantly sent to the VGA controller. By incrementing and loading RAM values into these registers, using the control core's regular instruction set, the control core program can upload images to the VGA screen buffer.

A third register is wired to the DMA controller, to enable 21 bit wide DMA parameters to be set programmatically.

The control core runs instructions from the same instruction stream as the SIMD nodes, and thus the control core and the SIMD array share a common program counter. The leftmost bit in the instructions specifies whether the instruction is a control core instruction or a SIMD instruction.

The control core instruction set is specified in detail in Appendix A.2.

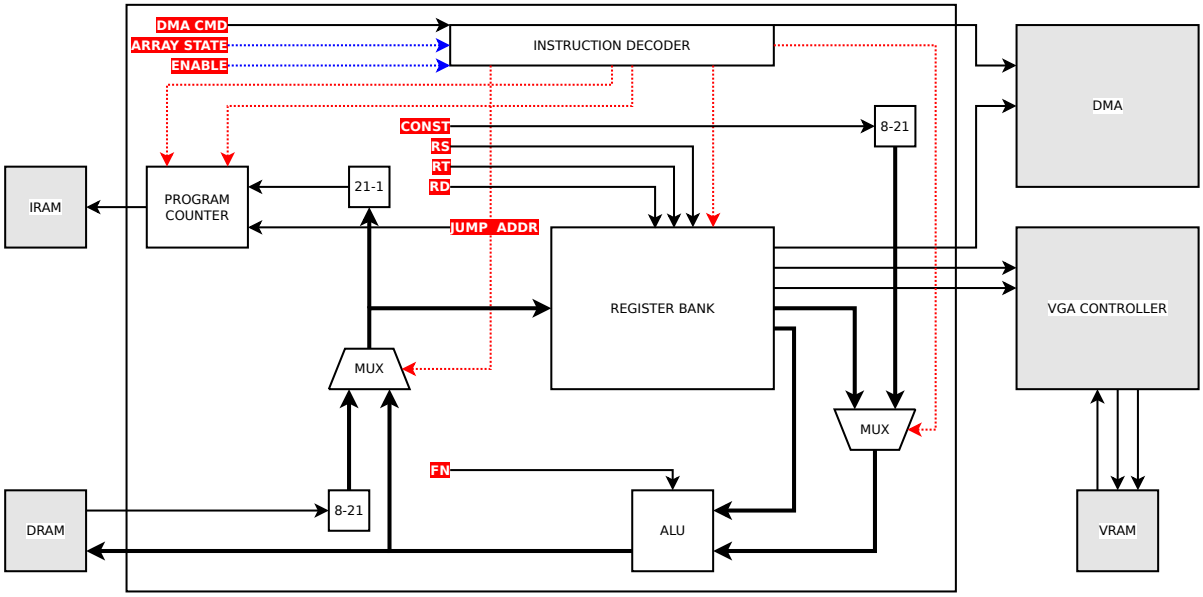

Figure 3.7: Image processor control core.

3.4.3 *DMA Module*

Because of the common instruction stream between the SIMD array and the control core, the SIMD array will be inactive whenever the control core is busy. An important task that normally would be the control core's responsibility is to load the SIMD array with new data from RAM and to store the processed data back to RAM. To maximize the utilization of the data RAM I/O capacity and the processing power of the SIMD array, a *Direct Memory Access module* was devised, allowing us to overlap computation and communication.

The DMA module performs loading of new SIMD data planes and storing of processed SIMD data planes *in parallel* with the normal execution of program instructions. The S registers in the SIMD nodes are reserved for the DMA module, and the module can send the contents of the S registers one step to the right by setting a *step S* signal.

With these accommodations of the SIMD array, the DMA module can load new data values to the S registers at the left edge of the SIMD array, send the step signal to have these value propagate into the array and read the old, processed data values from the S registers at the right edge of the array. The SIMD program executing simultaneously will operate on a different set of registers and will not be obstructed by the DMA data transfer.

During DMA transfer, the data RAM is reserved for the DMA module and is not available for the control core. To initiate a DMA transfer, the control core sets *base addresses* for the reading of new data and for the writing of old data, in addition to address increments to be used for moving along the columns and rows of the current slice of the data. These parameters allow flexibility in how the data slices are laid out in memory.

3.5 VGA CONTROLLER

We learned early on that last year's group had problems with their off-the-shelf VGA controller being a major bottleneck, and were only able to squeeze out less than one frame per second. Using a similar approach would most likely leave us with a machine unable to hit one our main goals (FR1, Table 1.1) of a minimum of ten frames per second.

After having scoured the Internet for higher performing VGA controllers in our price range without any luck, and after consulting with Jahre, we decided to implement our own VGA controller on the FPGA.

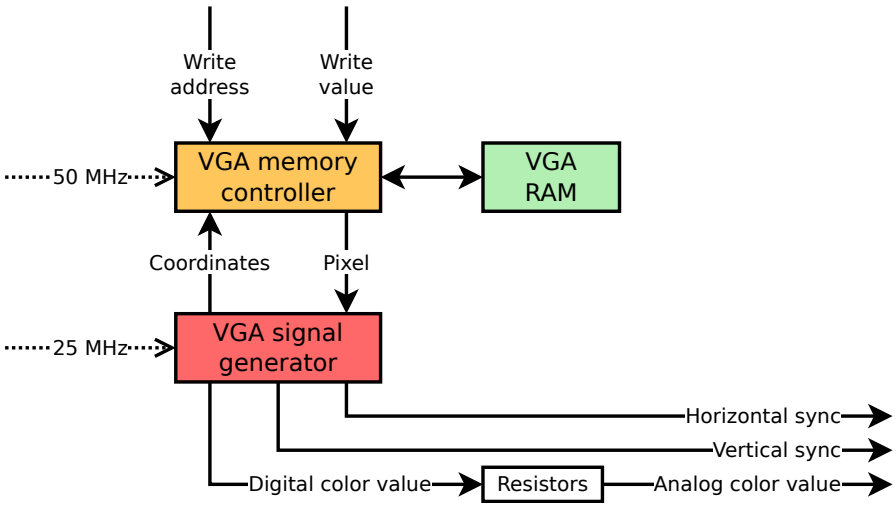

Figure 3.8: The parts of the VGA controller.

3.5.1 *Design*

We recognized the importance of allowing the control core to dump pixels to the VGA controller at its own pace without having to meet certain timing criteria, so a physical memory and hence a memory controller was needed. Since the FPGA has a 50MHz clock available and the pixel clock should be running on approximately 25MHz, a

fairly simple solution for dividing the memory between the control core and the actual signal generator was found: The signal generator must be able to read from memory at most every other cycle, so the remaining cycles are all free for the control core to use. To simplify the design as much as possible, the memory controller simply alternates every cycle between writing what is asserted on the signals from the control core, and reading a pixel for the signal generator.

The signal generator itself is pretty straightforward. It calculates when to pulse the V-sync and H-sync signals for a given resolution, and outputs a (8-bit greyscale) pixel fetched from the memory controller at the appropriate time.

3.5.2 *Circuitry*

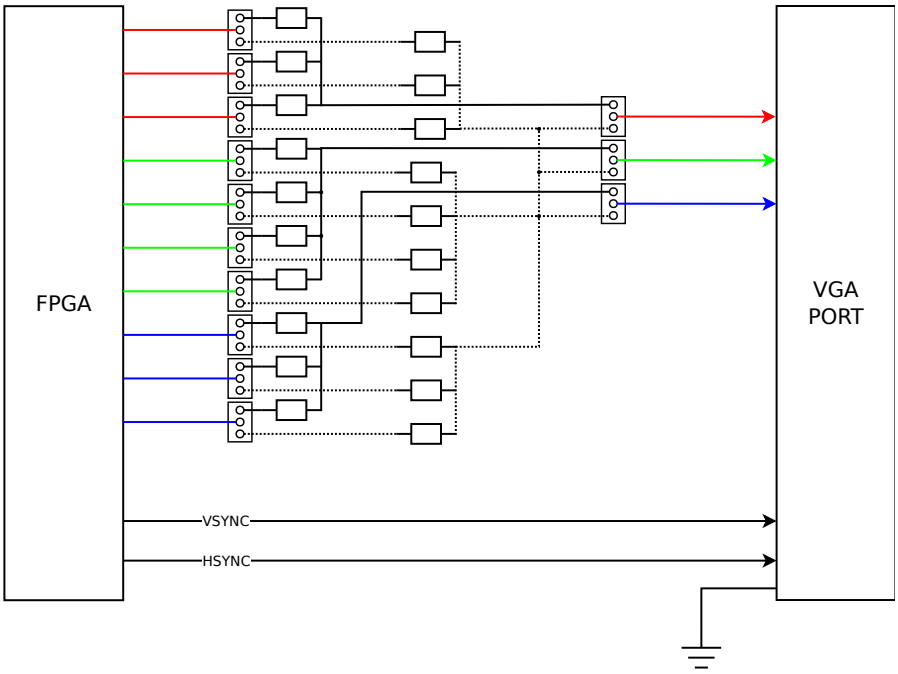

Figure 3.9: The resistor network of the DAC.

When designing the circuit, we had to choose between a black box Digital-to-Analog Converter (DAC) and making our own. Preliminary research on the VGA-protocol showed that for our purposes and our fairly low requirements on quality, making the DAC with simple resistors in parallel would be sufficient. Each resistor doubles in resistance for each step from the most significant bit to the least significant. We thought this would be easier to prototype and debug. The resulting circuit is shown in Figure 3.9. Since color was a "nice-to-have" in case we had time to spare and not at all a priority, we made the decision

to greatly simplify the design by making the switch from greyscale to color a manual operation of moving a set of jumpers. This way we reduced both pin usage on the FPGA and the complexity of the circuit.

4

SYSTEM CONTROL UNIT

Any sufficiently advanced bug is indistinguishable from a feature.

— Rich Kulawiec

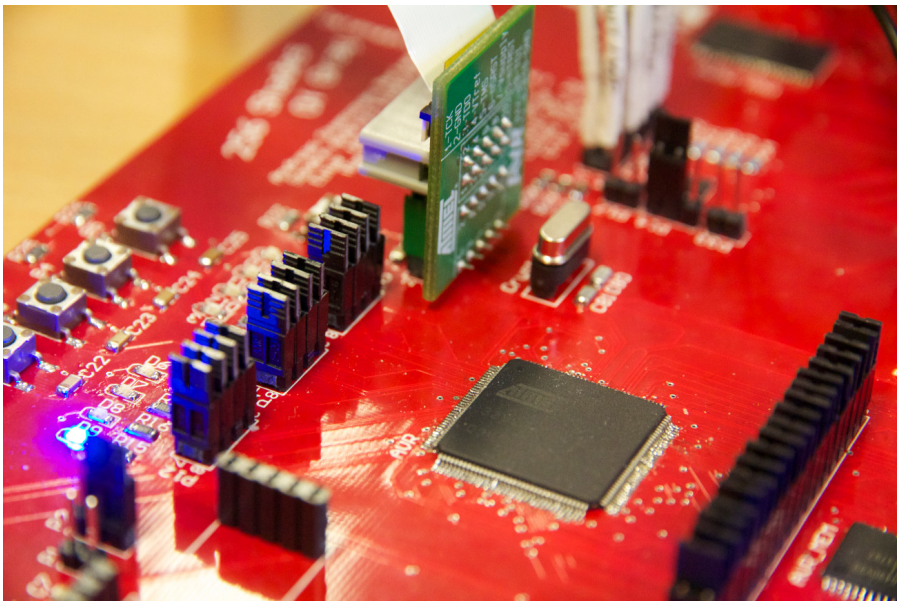

Figure 4.1: The System Control Unit

The System Control Unit (SCU) is implemented on a AT32UC3A0512 AVR microcontroller. This was both an advantage and a disadvantage to us. The same AVR had been used in several previous projects which allowed us to learn from their successes and failures to some degree. However, this specific AVR also gave us some issues, especially in regard to performance of the SD card.

As the name implies, the SCU is the controlling unit in our system. The SCU is in charge of setting up both LENA's instruction and data memory. The SCU can start LENA by putting it in a run state.

The SCU also implements the user interface for the project. With the help of the VGA implementation on the LENA, it is able to display a menu to the user which allows for selection of a program and data source from the SD card connected to the SCU.

This section documents the hardware considerations we had to make in relation to the the microcontroller used in the SCU, the AVR AT32UC3A0, hereby referred to as AVR. Also included are specifics in regard to what choices were made in terms of AVR specific hardware.

Trying to design and plan some of the hardware associated with the AVR, it was plain that while some components could be connected to arbitrary GPIO pins, others would require specific pins. To mention some of these:

TODO: *only mention some, not all?*

1. Serial port
2. SD card
3. USB connectivity
4. External Bus Interface (EBI) AVR memory

We also wanted to have some visual input and debugging tools for the project (buttons and LEDs). These were also connected to the AVR. We did however not have to take that into account mapping pins as they could be fitted on any General Purpose Input/Output (GPIO) pin. This allowed us greater flexibility in terms of what had to go where and made it possible to fit the other interfaces on the AVR better.

Same goes for the VGA controller we had mapped up to the AVR. Seeing as this didn't require any specific ports on the AVR, we fit this in where there were leftover pins when the rest was mapped out.

For the actual pinout of the AVR, see Appendix B.4.

AVR CRYSTAL

Previous projects have chosen to use multiple crystals on their AVR. Our AVR has support for up to 3 crystals, where 2 are high frequency crystals that can be used as external clocks for the AVR. The last crystal is a low frequency crystal that can replace an internal clock for power saving functions and real time measurement for greater accuracy, since the internal clock speed may vary by temperature.

As we were to optimize for performance we tried to use the most of the given resources. This included saving pins wherever we could, and as such we have gone with only one crystal. This crystal is connected to port 0 on the AVR and has a socket. We ordered a number of crystals in order to make sure that we could adapt to most any requirements that may or may not have been clear to us at the time of the design.

SD CARD

We also needed a storage medium. Since previous projects have had good results with using SD cards, this was what we chose as well. However, working on the design we encountered a few problems with this, mostly in regard to performance. This problem is elaborated in Section 7.1.

SERIAL PORT

The serial feature was implemented as a way of debugging. This is connected to Universal Synchronous/Asynchronous Receiver Transmitter (USART) 0 on the AVR and was not a crucial part of the design, but rather a way for us to communicate and get quantitative data from the SCU. We decided that having serial access would be handy as there is only so much information you can get out of 8 blinking LEDs. The serial had some implementation issues in regard to our mapping of the lines to the port. Since we had the ability to debug from the VGA controller, we did not prioritize fixing the serial until later. In the end, we never had to use it, more than to test whether it worked or not.

USB CONNECTIVITY

Our AVR supports USB, but it was not implemented in the SCU. As both the serial and the SD card reader was supposedly easier to implement[4], we decided to defer the implementation until other I/O components were implemented. This feature was therefore given a low priority and was not implemented in the final design due to time constraints.

EXTERNAL BUS INTERFACE - AVR MEMORY

Early in the design process, we came to doubt whether the SD card would perform as well as we needed it to. In order to try to safeguard us from total failure in a case where this would come true (which it did) we gave the AVR a separate memory in order to be able to have a buffer for data coming from the SD card or from the FPGA. The AVR itself does not have enough memory for even a single frame of video, which is why – after some discussion – we deemed it a worthwhile use of ports at the expense of some I/O lines to the FPGA.

μVGA-II

Questioning whether or not we would be able to implement a VGA controller on the LENA, we thought it was wise to have a backup in case this would fail. The μVGA-II is a prebuilt type

that has been used in previous projects with some success. In the backup solution, LENA would pipe data back to the AVR for display on screen, should the LENA VGA controller fail.

4.2 SCU MODULES

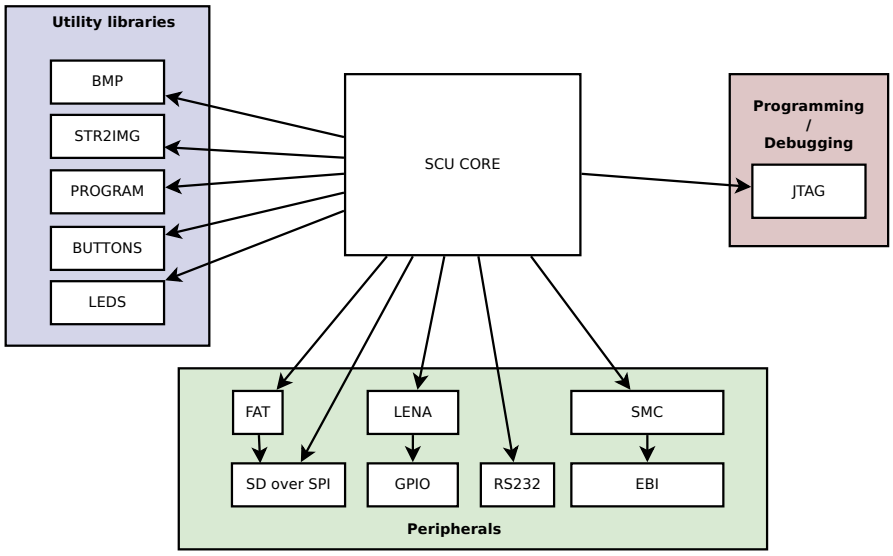

Figure 4.2: The different modules of the SCU

The AVR is responsible for controlling the entire system, and has several important tasks and functionalities related to this. These have been split into several modules that behave largely independently of each other.

SD OVER SPI A simple to use source of data/data storage, with easy to use framework code available. It is rather slow, however, as the AT32UC3A0 only has support for SPI. Section 4.5.1 covers this in detail.

FAT FAT is a file system implementation with easy to use framework code available. However, using it also incurs a performance penalty and is not used in performance critical sections. This is detailed in Section 7.1.

LENA/GPIO Interface for communicating with LENA. It is built on top of GPIO and provides a natural interface without low level details of GPIO.

RS232 A protocol for communication with a PC over the serial port. Very simple to use and is very useful for debugging. We did,

however, not use it past the EVK1100 test board as we had an issue with it (See Section 4.5.2), nor did we really need it.

STR2IMG Str2Img is a small text-to-image library that we use to render text on the monitor. Primary use is debugging and creating the menu the users are presented with when the SCU resets.

PROGRAM This is responsible for presenting the user with an interface to select the program to run. It is built on top of other utility libraries such as Str2Img and our LENA abstraction layer.

LEDS Functions for turning the LED on and off. Useful for blinking lights to show information about internal states of the SCU and debugging.

JTAG JTAG is an interface used to connect the SCU to a computer. This allows us to flash the AVR, debug and profile our programs using familiar-looking tools: avr32program, avr32-gdb and avr32-gprof, for flashing, debugging and profiling, respectively.

4.3 COMMUNICATION WITH LENA

This section documents how the SCU communicates and controls the LENA. The communication is usually performed by the SCU assigning a state to the LENA, followed by some data transfer. The communication utilizes three buses: the LENA out bus, the LENA in bus, and the LENA state bus. A full pin reference of these buses can be found in Appendix A.3.

4.3.1 *LENA states*

The SCU can set LENA to 5 different states, controlling LENA's behavior. The states are set with a 3-line bus and a fourth line which when toggled, indicates that the LENA should set its state to the state currently on the bus. The three state lines represent a binary number referring to one of the states. The different states are listed in Table 3.2.

4.3.2 *Data transfer from SCU to LENA*

When the SCU transfers data, it first sets the LENA state to `LOAD_DATA` or `LOAD_INSTRUCTION` to load data or instructions, respectively. The

NAME	DESCRIPTION	COMMENT
STOP	LENA stops	default state
RUN	LENA runs program	
LOAD_DATA	LENA receives data	
STORE_DATA	SCU receives data	not used
LOAD_INSTRUCTION	LENA receives instructions	

Table 4.1: LENA states

procedure for the SCU is then to put one word on the bus, toggle the data ready line and repeat. The LENA state STOP is used to signal the end of transfer. When the data are simply data, the first 8 lines of the LENA input bus is used to transfer data one byte at a time. When the data are instructions for the LENA, data are transferred one instruction (24 bit) at a time, and the first 24 lines of the bus are used.

4.3.3 *Data transfer from LENA to SCU*

The LENA has the ability to transfer data back to the SCU. This is a fallback routine, implemented in case of a bottleneck or failure in the VGA controller on the LENA. In this case, the LENA can transfer the processed data back to the SCU for storage or to be sent through an alternative VGA controller. In this transfer the data bus are 8 pins connected to the LENA's I/O pins.

This transfer is rather unusual because it is, unlike all other operations, not initiated by the SCU. The LENA will initiate the transfer by putting one byte on the bus, and then interrupt the SCU. After the SCU has acknowledged, another byte can be transferred. The first four bytes of the transfer will represent, as a 32 bit unsigned integer, the number of bytes of actual data to be transferred.

4.4 USER INTERFACE

The User Interface (UI) is a simple program running on the SCU letting the user control the 256 Shades of Gray. The UI has three states which the user goes through: Select program, select data and run program. The UI is implemented by converting text into bitmap files and sending these files to LENA for output to VGA. The user can use the buttons to scroll through a menu of, and choose, a program and a data file, respectively. When a data file is chosen, the SCU will run the selected program with the selected data on LENA, and then reset after the program has terminated.

4.4.1 *File system utilization*

To ease the handling of data on the SD card, we chose to use the FAT file system. The file system contains files such as scripts, LENA programs and data pointers. Because of the performance limitations of the Atmel Software Foundation (ASF) FAT driver in conjunction with multiple block reads[1], the data is actually placed outside of the file system. The transfer speed of data is crucial to our system's performance, and this increased the frame rate substantially. We were still able to use the file system to create an abstraction for this. The file system contains data files containing ”pointers” to the memory location of the actual data. The data pointer files on the file system is a text file containing two numbers: The location of the first block of data on the SD card, and the number of frames to read from that location.

4.5 ISSUES

While we did not have any serious problems with the SCU, we did encounter a few. This section is intended to shed some light on the issued we faced and how we were able to either fix or get around them.

4.5.1 *SPI*

Reading from the SD card over Serial Peripheral Interface Bus (SPI) turned out to be the single biggest bottleneck of our system. This was in part due to the serial nature of SPI, but also the poorly optimized SPI driver from the Atmel Software Framework.

The limitations of SPI is not something we could change. Our microcontroller does not have hardware support for using the 4-bit SD protocol, which left us with SPI as our only choice. We ended up running the SD card at a frequency of 39 MHz, which gives us a theoretical upper speed bound of 4.88 MB/s. In practice, we managed to reach around 1.1 MB/s.

Some of the changes we made to make this happen, was:

- Turn on compiler optimizations
- add SDHC support and increase clock rates
- use multiple block reads and bypass the file system

This is explained in more detail in Section 7.1.

1 See Section 7.1 for details

4.5.2 *Serial*

Communicating with the AVR over the serial port worked perfectly fine on the EVK1100. However, on 256 Shades of Gray it did not work at once. The only obvious error with the serial device was that the TXD/CTS (T for Transmit) pins were wired into RXD/RTS (R for Receive), and vice versa. This is covered in more detail later on, in the PCB workaround chapter. However, as the Str2img was completed, debugging could be done by writing text to screen. As debugging was the main purpose of the serial communication, we did not prioritize fixing the serial. In the end, we only made it work to convince ourselves that our hypothesis was correct.

4.5.3 *Dead pin between SCU and LENA*

One of the unidirectional pins between SCU and LENA turned out to be dead. This was simple enough to work around by replacing it with one of the bidirectional pins.

4.5.4 *Bug in RTC Driver*

When initializing the Real Time Clock (RTC) driver, we encountered a known bug[2]. This meant that we could not use the internal 32kHz crystal oscillator for measuring time, but had to use the 115kHz RC oscillator instead. This meant slightly higher inaccuracies, but for our purposes, it was not a a serious issue.

2 Atmel Bug Report

5 PCB

Electricity is really just organized lightning.

— George Carlin

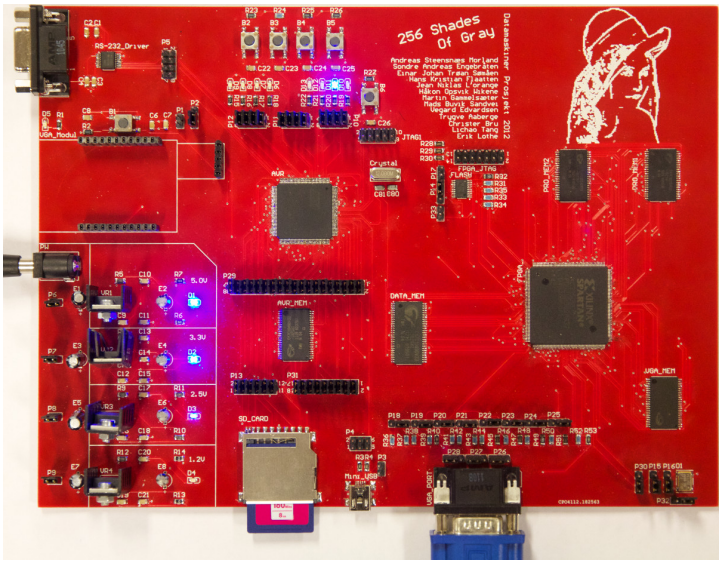

Figure 5.1: The PCB With All The Components.

The entire system is connected together on the PCB. Designing and soldering a correct and working PCB is thus important to be able to construct the system as a whole.

This chapter will start by explaining how some of our overall design choices affected the PCB design. We elaborate our power layout before giving an overview of where we got our footprints from. Finally, we explain the process of making the PCB, as well as some of the issues and solutions we met during the process.

5.1 DESIGN CHOICES

5.1.1 *Memory*

The overall design for our machine specifies various different, separate memories. One for data, one for instructions and one for the VGA. The

data, VGA and AVR memories use 8-bit chips to match their word size. And the instruction memory uses 2x16-bit chips with address-lines connected together and 8 ignored I/O-pins, to match its 24-bit word size.

5.1.2 *VGA*

Our initial plans were to create our own VGA controller in the FPGA, but since having video-output was mission-critical for our project, we really wanted to have a fallback-solution in case our own VGA controller solution ended up not working. To allow for this fallback, we added the necessary connectors for the external VGA module that Festina Lente used last year, even though that module was known to be too slow to be able to display images at our required speeds, it would atleast give us some way of outputting anything at all if our own solution failed.

5.1.3 *Communication*

The intention was to use the SD card reader as our main source of data/instructions, however, as our system would not fulfill our functional requirements unless we were able to get data into it at the speeds specified in requirement FR1, we opted to also have a fallback solution here, thus we also added USB and RS232 as fallback solutions for getting data into/communicating with the computer.

5.2 POWER SUPPLY

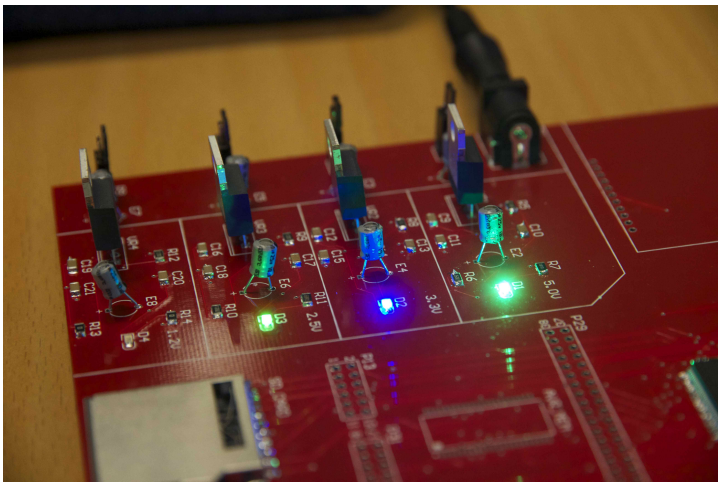

Figure 5.2: The PCB with the powersupply soldered.

After talking to Tufte and Jahre, we decided to reuse the power supply from Festina Lente without any changes. Tufte said the power supply had evolved year by year, and that it would therefore be a better solution to reuse it rather than trying to create one from scratch. By doing so, we reduced the possibility of introducing new issues.

5.3 POWER PLANE

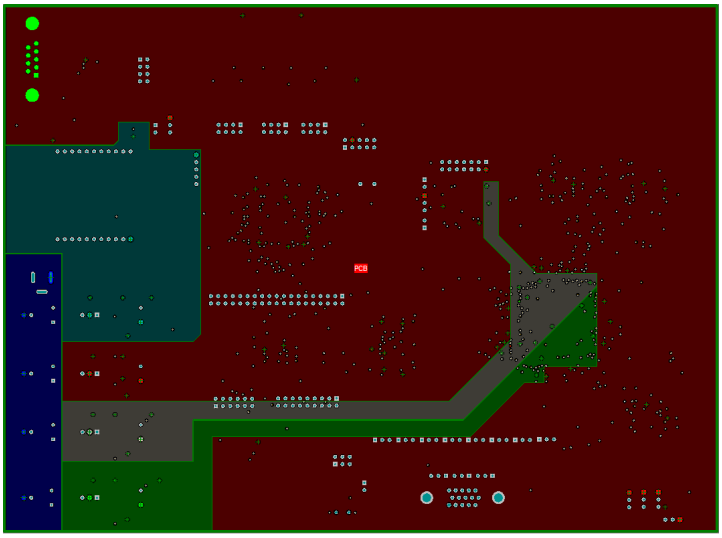

Figure 5.3: The Power Planes

Since our power supply was exactly the same as Festina Lente's we also ended up with similar power planes, as seen in Figure 5.3; 12V (dark blue), 5V (teal), 3.3V (red), 2.5V (gray) and 1.2V (green). The 5V was only used for the external VGA module. 2.5V and 1.2V were split across the FPGA.

The rest of the board got 3.3V. The entire power plane was put in internal layer 1, with a ground layer in internal layer 2.

5.4 FOOTPRINTS

Some of the components we chose did not have footprints readily available, which meant we either had to look for them on the internet, or create them.

This meant either staring at datasheets and carefully placing pins relative to each other, or running the IPC wizard.

5.4.1 *We made the following footprints*

COMPONENT	PURPOSE
SD card reader	For fast data access
VGA plug	For our own VGA implementation
CY7C1069DV33 54TSOP	Data-memory (2M × 8bit)
CY7C1021DV33 44TSOP	Program-memory (64k × 16bit)
CY7C1049DV33 44TSOP	VGA memory (512 × 8bit)

Table 5.1: The Customized Footprints

SD CARD
: With the pin-designations selected by googling the pinouts for SD cards in general, and comparing a few of those hits to make sure that the results agreed with each other. Finally we selected a single hit[1] to base the pinout on. The schematic[2] for the reader itself was particularly hard to read, to the point that we had to ask Tufte for help to understand it. Even after that, there was no way to be sure precisely where the grounding pads were relative to the rest of the SD card reader. Thus we did some guesswork, and added a decent amount of slack to the ground pads by adding extra tin to these pads.

MEMORY (TSOP54/TSOP44)
: The TSOP54/TSOP44 footprints were created using the IPC Footprint Wizard, as their datasheets fitted nicely with the Small Outline Package setting in that Wizard.

5.4.2 *Footprints from other sources:*

As Festina Lente used some components that we also ended up using, we decided, after talking to Tufte, to use their Footprints[3] (as they were known to work) for these components:

The oscillator had one issue last year, the footprint for it was mirrored , noting this from Festina Lente's report, we read through the datasheet, and remapped the pin-numbering (taking care to understand why it was mirrored last year) on the foot-print before using this footprint.

After reading the Festina Lente report[4], we decided to follow their choice of footprint for capacitors and resistors: 1206 SMT. As those

1 http://pinouts.ru/Memory/sdcard_pinout.shtml
2 http://katalog.we-online.de/em/datasheet/693063010911.pdf
3 http://org.ntnu.no/datamaskinerprosjekt2011/altium_libraries/dmprolibrary/

COMPONENT	SOURCE
Button (FSM2JSMA)	Festina Lente
Crystal	Festina Lente
LUMBERG - 2486 01 - MINI USB TYPE B	Festina Lente
Oscillator (CSX750FJC50.000M-UT)	Festina Lente
Power Connector	Festina Lente
External VGA module	Festina Lente
AVR	AVR-freaks
FPGA	Altium Design Library

Table 5.2: Footprints and sources

were physically bigger than the 0804 SMTs they mentioned as an alternative, they would be easier to solder by hand.

5.5 PROCESS

This section describes the work and design challenges faced related to the PCB.

5.5.1 *Schematics*

The workflow of creating the schematic consisted of reading data sheets, and looking at the reports from earlier computer design projects, and then applying the knowledge we found from those to properly place the necessary components in our schematic.

We decided to design the entire PCB in one schematic, as the Festina Lente report mentioned that Festina Lente-report does mention that "The decision to make the central components appear in multiple schematic sheets made Altium issue a lot of warnings and errors during the design rule check"[4, p. 49]. Something we avoided from day one, as we never attempted to use multiple schematics-documents.

A downside of this approach was the fact that this partially serialized our work on the schematic, since we could not make concurrent changes to our single document. When not working on the schematic, the other people in the PCB group did whatever could be done without touching the schematic. (Making footprints, verifying design, looking up parts and data sheets) Since the schematic was the biggest amount of work, this produced something of a bottleneck for the PCB work.

However, having overall control of the entire thing in one document did help to smooth out some issues we met while working. For instance, we were having quite a few issues with net labels. This was quite easy to solve when everything was in one document with no ports, as getting complete overview was doable, without having to cross-reference schematics. We also avoided the need to use any buses, although we did end up adding some for the sake of readability.

The overall layout of the Schematic B.4 is logically grouped "geographically", to allow for easy reading of the schematic.

5.5.1.1 *Buses/Wirelabels*

We initially worked from the assumption that all pins should be connected to a bus, and then that bus should be connected to the pins in the other end. This gave us some issues with duplicate naming. After digging through quite a bit of Altium documentation, we became assured that simply wire-labeling the pins would create the necessary connections. This is because any pin/wire with the same name as any other pin/wire will be connected by definition in Altium.

This arguably made for a less readable schematic, as the bus-connected solution was quite a lot easier to follow when tracing. However, as our schematic still is logically grouped, it is not very hard to find the correct connections even without the buses drawn in. We are quite sure that given a logical overview of which components are directly connected to each other, finding the various pins that perform this connection should be trivial even without the buses/wires drawn in.

5.5.2 *Routing*

Our plan was to start with a big PCB document to be able to easily place all the components. We placed the components and started routing. A lot of the routing was done by the auto route feature of Altium. At first, we did not set any Design Rules. We found that the default rules did not comply with capabilities listed on Elprint's web page.[4]. After fixing this, we shrank the board size. The main reason was to reduce production cost, but Tufte also told us that having the board too big could cause short circuits between the power planes.

We then remapped some pins in the schematic to increase physical proximity, and to untangle the amount of crossing wires. In the end, we did 2 days of manual routing to reduce the number of vias. Figure

4 http://www.elprint.no/products/pcb/capabilities

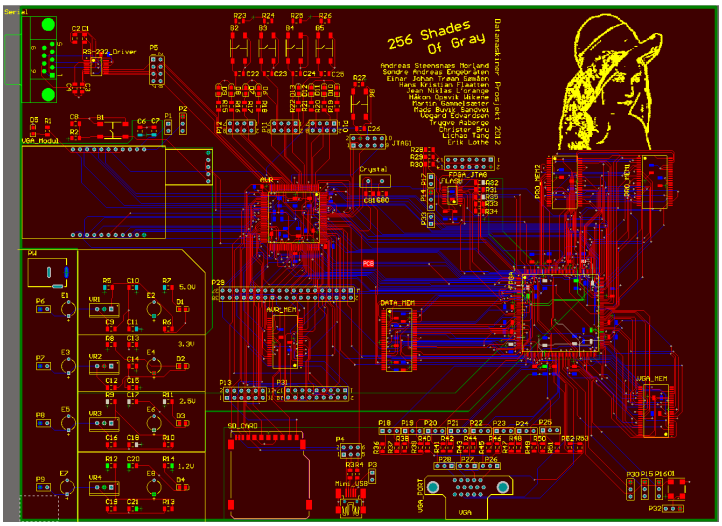

Figure 5.4: Routed PCB

5.5 shows an example where we coupled together several pins to the same via going to the ground layer. Even though we wanted to reduce the number of vias, we went with the default routing in Altium. The amount of manual work could perhaps have been reduced by selecting the "Via Mixer" strategy instead.

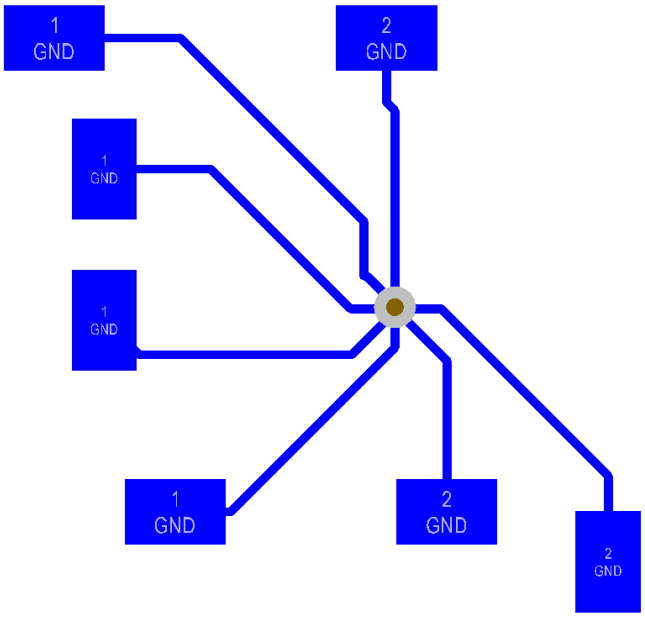

Figure 5.5: Removing vias

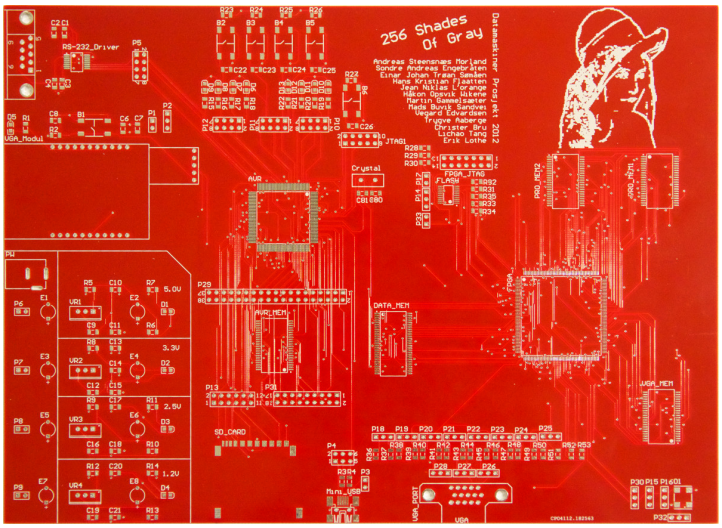

Figure 5.6: The PCB Without Components

5.5.3 *Soldering*

When the PCB arrived from production, the soldering proved to be a bit more difficult than originally anticipated. First, we tested whether there were any short circuits in the board itself. We used the multimeter to test that there was no current passing from one layer to another. Then, we started to solder the power supply, one power plane at time, testing the board for short circuit after each iteration. The Table 5.3 shows the observed voltage from each plane.

TEST	RESULT	PASSED
Power supply 12.0V	Measured 12.045	OK
Power supply 5.0V	Measured 4.995	OK
Power supply 3.3V	Measured 3.286	OK
Power supply 2.5V	Measured 2.510	OK
Power supply 1.2V	Measured 1.240	OK

Table 5.3: Results of power supply

After about a days work we managed to completely destroy a pin on the AVR, therefore we had to start soldering a new board. On this new board we started soldering the AVR and FPGA, as they were the hardest components to solder on the first board. After soldering each side on the AVR and FPGA, we tested the board for short circuits. As no short circuits were found, we moved on to soldering the power supply, as well as the Joint Test Action Groups (JTAGs), FLASH and LEDs. In order to check that the PCB was working, the AVR and FPGA groups tested the PCB without the capacitors. After both groups had

tested that they could connect to the AVR and FPGA respectively, we soldered the rest of the PCB.

5.6 problems and workarounds

5.6.1 *Serial port*

We found many pin mappings for the serial port on the web, which turned out to be different from Festina Lente's serial system. After some discussion, we chose the one on the web, and figured that if anything did not work, we could just use our header to change the mapping. Yet as we had not thought of the fact that the footprint for male and female serial ports have mirrored numbering, we ended up with a male pin mapping and a female footprint. Obviously, this did not work very well. Yet as the TXD pin is the only one we actually need, we used a female plug, and dragged a wire over the header pins so that the TXD data from SCU were mapped to the original RXD. This workaround made it possible to send data from the system and out to a computer, yet as we had no need for this option, our final card does not have the serial system soldered on.

5.6.2 *Routing*

We spent the entire final week before production working on the routing. In comparison, the energy group used only 2 days. There are quite a few possible reasons why we ended up using so much more time than them for this:

One of the reasons were simply that we were the first group to start routing. We thus had no one to warn us about possible mistakes. An example of such a mistake was that we did not set the correct Design Rules before auto-routing, until a few days into the week. This naturally did not give us the routing that we wanted, and gave us a few headaches.

Other reasons might be the fact that we did not experiment enough with different routing strategies, and the fact the other group had to route only 1 memory chip, where we had 5. This naturally made the routing much more complex.

5.6.3 *Soldering*

Soldering was the biggest problem that we faced, as none of the members of the PCB group had any experience with it beforehand. We managed to destroy our first board, as we used too much tin on one of the AVR pins, and ruined them by using the tin removal unit wrongly.

However, the second attempt was more successful. But there still remained some minor problems. Problems we found were lack of tin on the oscillator, as well as a few of the FPGA pins. They were, as opposed to our problem with the first board, easily fixed, once discovered.

5.6.4 *The third board*

After using the second board for development for about a week, several problems started appearing.

First, we discovered that one of the connections between the AVR and FPGA was broken. We managed to work around this by replacing it with another previously unused connection.

Second, we discovered that one of the voltage regulators was a bit loose. Touching it could make the whole machine stop working.

If these had been the only problems, we could have avoided them by making sure not to touch the voltage regulator. We did however also discover a problem with the VGA memory. It would randomly stop working or misbehave, visible as noisy images on the screen.

We therefore decided that we had to make another board. To be able to do this we had to order some spare components. After the components arrived, we soldered the new board.

The testing of this board was done by running code on it. The programs that ran on the second board, also worked on the third board. The problem with the VGA memory was gone, and the voltage regulator seemed to be stable.

6

SYSTEM TESTING

If you don't care about quality, you can meet any other requirement.

— Gerald M. Weinberg

Any system with functional requirements needs to verify whether the functional requirements are satisfied or not. This is done through system tests, which are essential to get a confirmation or a rejection of whether we satisfy the requirements. Usually we will test the functionality of a system as a whole, but the nature of our requirements means that some of the tests will only test specific parts of the system.

This chapter will start by presenting the design of the system tests. It will then go into how we will test the different functional requirements. We summarize by presenting our results with additional comments.

6.1 TEST DESIGN

The text below describes what we are going to do to test the system, and includes what we expect as a result for each test. The functional requirements are repeated here to ease reading. Some of the tests may contain dependencies[1]. If so, they are specified.

FUNCTIONAL REQUIREMENT 1

> *The image processor should be fast enough to output unmodified 8-bit images of at least* 320×240 *resolution at 10 frames per second.*
>
> TEST: Create a program which shows a video through a VGA port with the resolution and frame rate specified.
>
> PASS CRITERIA: The video shows correctly on the device connected to the VGA port at the specified number of frames per second.
>
> DEPENDENCIES: FR3

1 Even though the functional requirements are independent of each other, the tests may be simplified by adding dependencies to other functional requirements.

Functional Requirement 2 and Functional Requirement 7

The image processor should be generally programmable.

The machine should have example programs demonstrating its capabilities, such as a video player and a median filter.

Test: Create a 3×3 edge detection filter in the instruction set of the machine, load it up on the machine along with one or several image files or videos, and make it run. It should view the video or image with a edge detection filter applied.

Pass criteria: The edge detection filter is visibly applied on the image or video chosen.

Dependencies: FR3

Functional Requirement 3

The machine should have a video port for image output.

Test: Create a program which utilizes one of the VGA ports on the board and writes an image to it.

Pass criteria: The image is shown on the device which is connected to the VGA.

Functional Requirement 4

It should be possible to operate the machine with on-board buttons.

Test: Create a program for the SCU which turns on LEDs depending on which button is pressed.

Pass criteria: The LEDs should be lit up when the corresponding button is pressed.

Dependencies: FR5

Functional Requirement 5

The machine should have on-board LEDs for visual feedback.

Test: Create a program for the SCU which turns the LEDs on in a pattern easily recognizable. All LEDs must be utilized at some point in the pattern.

Pass criteria: The LEDs lights up in the pattern expected.

Functional Requirement 6

There should be developer tools (assembler) available for the machine.

Test: Create one or several assembly programs which together utilizes all different aspects of the assembler. Then verify that the programs made correspond to the correct instructions through analysis of the output.

Pass criteria: The output generated should correspond to the instruction set specification, and should jump to labels correctly.

6.2 test procedures

Some of these tests are strictly dependent on other tests. In theory, all the different tests could be done in any order to test the correctness of the system. However, some of the functional requirements makes it easy to test other functional requirements. Therefore, we have decided to let some of these tests have dependencies. In addition, their ordering is a natural way to approach and test such a system, mostly because some of the test have a slight overlap. For instance, it is easier to have example programs if the assembler works correctly.

Test 1 - Functional Requirement 5

The machine should have on-board LEDs for visual feedback.

1. Program the SCU with a program which lights up the LEDs in every binary representation of the numbers 0 to 255.
2. Wait a specified amount of time between LED updates.
3. Verify that the observed pattern is the correct representation of binary numbers.

Test 2 - Functional Requirement 4

It should be possible to operate the machine with on-board buttons.

1. Program the SCU with a program which toggles a LED when a button is pushed.
2. Verify that the buttons toggle the LEDs when pushed.

Test 3 - Functional Requirement 6

There should be developer tools (assembler) available for the machine.

1. Write a correct program utilizing every available instruction and combinations thereof, and run it through the assembler.

2. Use a program, such as `hexdump`, to verify that the binary output corresponds to the specification, and that jump targets are the correct addresses.

Test 4 - Functional Requirement 3

The machine should have a video port for image output.

1. Load the LENA with a program which copies an image out to the VGA.

2. Read an image from the SD card.

3. Load the image from the SCU onto the LENA.

4. Start the LENA.

5. Verify that the image shows correctly on the screen.

Test 5 - Functional Requirement 2 and 7

The image processor should be generally programmable.

The machine should have example programs demonstrating its capabilities, such as a video player and a median filter.

1. Load the LENA with a program which performs edge detection and copies the result out to the VGA

2. Read an image from the SD card.

3. Load the image from the SCU onto the LENA.

4. Start the LENA.

5. Verify that the edge detection filter has been applied.

Test 6 - Functional Requirement 1

The image processor should be fast enough to output unmodified 8-bit images of at least 320×240 *resolution at 10 frames per second.*

1. Load the LENA with a program which copies images out to the VGA as they are loaded.

2. Read an image from the SD card.

3. Load the image onto the LENA.
4. Start the LENA.
5. Repeat stage 2 to 4 until the video has finished.
6. Measure the frame rate.

6.3 TEST RESULTS

Table 6.1 shows the results of every system test, along with eventual comments.

TEST Nº	REQUIREMENT	COMMENT	PASSED?
1	FR5	Leds turn on and off.	Yes
2	FR4	Buttons behave correctly.	Yes
3	FR6	Output corresponds to spec.	Yes
4	FR3	Image(s) are successfully drawn to screen.	Yes
5	FR 2 & FR7	Filter has been successfully applied.	Yes
6	FR1	Passed with a margin of 2fps!	Yes

Table 6.1: Test results.

7 RESULTS

If it's not on fire, it's a software problem.

— Unknown

When designing a system aimed for performance, one has to test the system and compare it to other solutions to see whether we have gained any mentionable speedup.

As the assignment was to create an array-based image processor, it is interesting to see how our system performs at typical image processing tasks, and whether utilizing the SIMD array yields any performance advantages over doing everything in the control core.

In this chapter, we will present the results of our benchmarks of two critical parts of the system; the SD card read performance of the SCU and the image processing performance of LENA. We will also show screen shots from the system running several different image processing programs.

7.1 SD CARD PERFORMANCE

As mentioned in Section 4.5.1, reading data from an SD card over SPI turned out to be the biggest bottleneck for throughput in our system. In this section, we will document the optimizations we made to improve the SD card's read performance.

Table 7.1 shows the gradually improving performance as we optimized the code. This is also shown in Figure 7.1. Each of the optimizations are explained in greater detail below.

REMOVED STATUS CHECKING + FUNCTION INLINING

The spi_send function busy-waited on the TXEMPTY flag for each byte it transmitted. In our case of alternating sends and reads, we do not have to check whether the SPI send register is empty every time. Inlining spi_read and spi_write also helped, although -O1, -O2 or -O3 would have done this for us.

LESS SPI STATUS REGISTER POLLING

In spi_read, two conditions are busy-waited on before returning: whether the send register is empty, and whether the receive buffer is full yet. We only have to check the latter of these conditions.

INCREASED CLOCK RATE

Our CPU and main bus is driven by an external 12MHz crystal oscillator and the SPI clock frequency is calculated by dividing this by an integer number. We kept this divisor to be either 1 or 2 (so SPI ran at either the same or half the speed of the CPU). Increasing to clock rate from 12MHz to 78 MHz boosted the speed from 157kB/s to 456 kB/s.

COMPILER OPTIMIZATIONS

Turning on compiler optimizations improved increase performance by another 14%. There was no noticeable difference in performance from -O1, -O2 or -O3. However, compiler optimizations had a more profound effect on the overall system performance. We did not benchmark this extensively, but it was probably caused by optimizations the compiler made to the LENA communication code.

BYPASS FAT AND USE MULTIPLE BLOCK READS

We patched the framework code to support multiple block reads. Apparently, this was not something the FAT driver was able to take full advantage of out of the box. It was easier to simply bypass the file system than to try to fix it, so that is what we did.

By reading the blocks directly from the SD card, we could read all 150 blocks of a picture in one function call. Limiting the file system to the first half of the SD card and add metadata files containing the block offsets into the second half of the card was easy. This more than doubled our performance, from 515 kB/s to 1172 kB/s.

TODO: *This must be rewritten*

After implementing these optimizations, the SCU was capable of pushing out around 12 FPS to the LENA, around 920 kB/s. As we managed to reach our goal of 10 FPS, we did not optimize it any further. However, using the Peripheral DMA Controller (PDCA), we could have effectively eliminated the 25% time spent sending to the LENA while the SD card waited. Thus, giving as a frame rate of around 16 FPS.

MILESTONE	DESCRIPTION	PERFORMANCE
1	Baseline before optimization	74.87 kB/s
2	Less status checking in `spi_send` plus inlining	137.00 kB/s
3	Less SPI status register polling	156.59 kB/s
4	Clock rate of 20MHz, 10MHz SPI	214.40 kB/s
5	Clock rate of 20MHz, 20MHz SPI	243.00 kB/s
6	Clock rate of 40MHz, 20MHz SPI	327.80 kB/s
7	Clock rate of 48MHz, 24MHz SPI	369.50 kB/s
8	Clock rate of 60MHz, 30MHz SPI	423.40 kB/s
9	Clock rate of 72MHz, 36MHz SPI	432.70 kB/s
10	Clock rate of 78MHz, 39MHz SPI	455.80 kB/s
11	Turn on `-O1`	521.70 kB/s
12	Turn on `-O2`	523.40 kB/s
13	Turn on `-O3`	515.30 kB/s
14	Bypass FAT and use multiple block reads	1171.90 kB/s

Table 7.1: The read performance over SPI at various milestones.

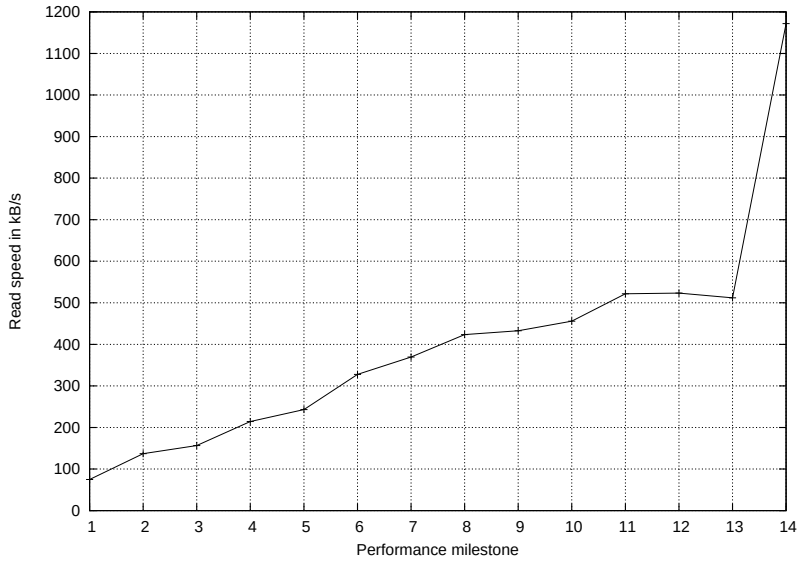

Figure 7.1: The results from Table 7.1 plotted. Milestones are along the x-axis and read performance along the y-axis.

As described in Section 1.2, a major goal of our project was to acheive frame rates of at least 10 frames per second for unprocessed video. This goal attends to our group's assignment of focusing on performance.

Here we present results from running different image processing programs on the system. For each program a frame rate is listed. This is the maximum frame rate at which the program could be run without generating artifacts on the screen. We also list power consumption for each of the programs. These measurements were made using an ammeter across the headers connecting the power planes to 12 V.

7.2.1 *Test programs*

Here are the numbered test programs used to generate the results:

PROGRAM 1
: Show unprocessed video on the screen using only the control core. The control core's task is to continually copy the incoming image data into the screen buffer.

PROGRAM 2
: Show video with inverted colors, using only the control core. The control core operates as in program 1, but it subtracts the pixel values from 255 before copying them to the screen buffer.

PROGRAM 3
: Show unprocessed video using the SIMD nodes. All of the pixels in a video frame are sent through the SIMD array before being copied to the screen by the control core.

PROGRAM 4
: Show inverted video using the SIMD nodes. The SIMD cores operate as in program 3, but they invert the pixel colors before passing the pixels back out to data RAM.

PROGRAM 5
: Show video processed by a 3x3 edge detection filter on the SIMD nodes.

7.2.2 *Results*

The test results are listed in Table 7.2 and plotted in Figure 7.2.

Program	Frame rate	Current through power planes			Power
		3.3 V	2.5 V	1.2 V	
Idle	-	347 mA	44 mA	77 mA	5.6 W
1	21 fps	375 mA	37 mA	69 mA	5.8 W
2	18 fps	375 mA	37 mA	71 mA	5.8 W
3	13 fps	370 mA	37 mA	65 mA	5.7 W
4	13 fps	375 mA	37 mA	67 mA	5.7 W
5	13 fps	363 mA	37 mA	75 mA	5.7 W

Table 7.2: LENA performance and power results.

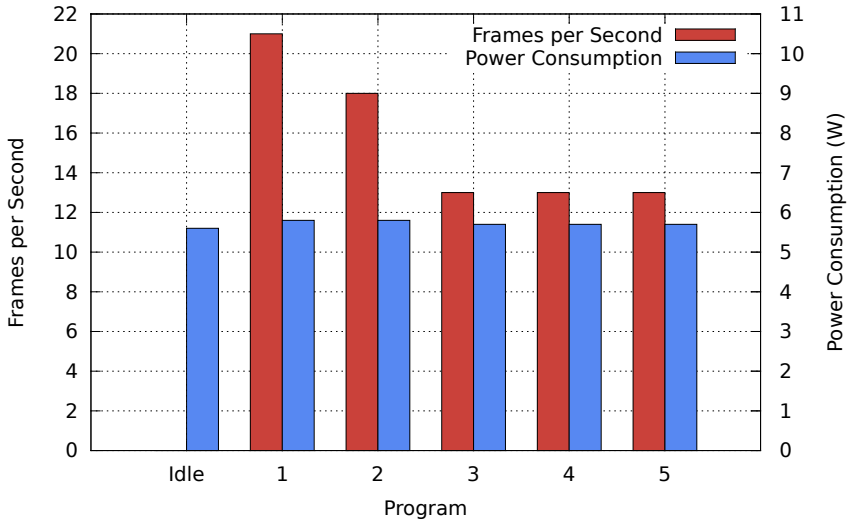

Figure 7.2: The LENA results from Table 7.2.

The system performance behaved as expected. The fastest program was program 1, which simply performs a copy iteration from data RAM to video RAM. This is a step the other programs have to perform as well. Program 2 is slightly slower than program 1, which probably is due to the extra subtraction in the inner loop of the program.

Programs 3, 4 and 5 are slower than the previous two programs. The copying of data into and out of the SIMD array entails a certain overhead that cannot be avoided. However, the increase in complexity in the programs from 3 to 5, does not result in a decrease in their performance. This is because all of the three programs are limited by the DMA speed, not the computational speed. Data transfer is the bottleneck in all of the three programs.

The power consumption stays relatively stable throughout the tests. This was to be expected, as no power management features have been implemented in the system.

In this section we show screen shots from the system running several different image processing algorithms. The image used for the examples is shown in Figure 7.3. This figure shows a screen shot when the system runs a simple *show image/video* program.

Figure 7.4 shows the image after it has had its colors negated by a program. Figure 7.5 shows the image after being run through an embossing algorithm. In Figure 7.6, the image has been run through an edge detection filter.

The images in these examples have all been processed by the cores in the SIMD array.

Figure 7.3: Unprocessed image.

Figure 7.4: Negated image.

Figure 7.5: Embossed image.

Figure 7.6: Edge detected image.

8 DISCUSSION

Debugging is twice as hard as writing the code in the first place. Therefore, if you write the code as cleverly as possible, you are, by definition, not smart enough to debug it.

— Brian Kernighan

8.1 INSTRUCTION SIZE

We ended up with a 24-bit instruction set for both SIMD nodes and the control core. As we ended up with a 16-bit program counter, the maximal instruction count would be 64k and the memory chip size is chosen on that basis. While it may sound little, it is still more than enough. To see why, note that the control core can start the image processing algorithm from the beginning when the SIMD nodes have finished processing the current data. Also, we usually define algorithms for a single frame, without any concern for the one coming after it. The program is simply restarted for every new frame, reducing the amount of code needed and the complexity of the programs.

8.2 MEMORY ARCHITECTURE

In the early phases of the project, multiple solutions as to how the data memory should work were suggested.

The top left example of Figure 8.1 shows a shared memory architecture with separate data and instruction memories. The top right example shows one single memory connected to both the SCU and LENA by a common bus. In the bottom left example, the memory is only connected to the LENA, and the SCU communicates with it through LENA. The bottom right example shows a switched memory architecture. There are two memories that can both be connected to the SCU and to the LENA, but not at the same time.

We decided to let the SCU send all the data to the LENA, and then let it store them to its own local memory, like in the bottom left example. The reason we chose this over shared memory was all the potential

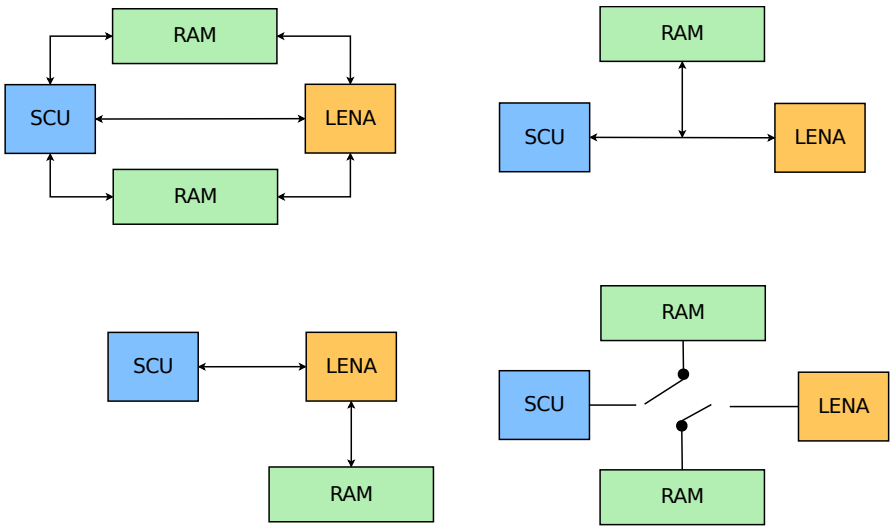

Figure 8.1: Considered data memory architectures

issues one may get with the shared memory solution. A shared memory would be prone to errors from synchronization problems such as knowing when, and to which addresses, a read/write operation should read/write. We figured that we would rather send all data from the SCU to the LENA, and let it store them into its own memory as it saw fit.

To allow overlap between loading instructions and data in the LENA architecture, we chose the Harvard Architecture and used a separate memory for instructions. The program memory is separated into two chips. The reason for this is mainly that we found no good 24 bit word memory (24 bit to match instruction size). One solution would be to just buy a 32-bit memory. However, a single 32 bit memory was more expensive than two 16 bit memories. As both gave us the needed memory, we chose the cheaper alternative.

In addition, we figured a shared VGA and data memory would obstruct the flow of data from the SCU. Since data for the VGA would be both read and written frequently, the VGA needed its own memory, rather than sharing memory with the rest of the LENA architecture. As we were supposed to optimize for performance, we had to take possible bottlenecks into consideration. We therefore invested in an extra memory chip. For the VGA we chose an 8-bit word memory that was big enough to hold at least a full screen-frame, at 8 bits per pixel (since each pixel is an 8-bit greyscale pixel).

Finally, to reduce the possibility of having too slow data-access from the SCU, (it had only 64kB on-chip memory) an extra memory was added to work as a buffer for the SCU as well. This design choice was made *after* ordering, which meant that we had to choose from

the chips we had already ordered to fit this purpose. Since this was intended to carry data to all other parts of the system, and as the rest of the system is working with data in 8-bit bytes, we ended up using one of the extra chips ordered as VGA memory for this purpose.

A result of this is that we now had 4 separate memories to allow overlap of memory accesses. Since the requirements for the data/instruction memories differed in both size and word-width we wound up with not only separate, but also different chips for this purpose.

8.3 REDUNDANCIES

We had multiple redundancies to avoid potential errors which would have rendered our whole system unusable. This was recommended by Jahre, and previous groups had done so as well. In order to send data into the system, our two main options would be to use the SD card or the mini USB plug. We had serial and JTAG for debugging purposes. For the graphical output we had the two VGA options. Luckily, both SD card and and our own VGA worked, and we therefore never needed to use any of our redundant solutions.

8.4 HARDWARE

As we decided to use many of the same solutions as previous years, we naturally decided to use many of the same components as they were proven to work. Following the same logic, some of our solutions are more or less a copy of previous years, such as the power supply. This is because we were heavily advised to use working solutions if we found any. We always tried to double check and find out how they worked before creating a new part in our system. This proved to work well, and we had no problems in the parts of our system that we reused from last year. Our only design problem in the hardware was when we managed to use a footprint for the serial, which had the opposite gender of our pin mapping.

9 CONCLUSION

Hofstadter's Law:
It always takes longer than you expect,
even when you take into account Hofstadter's Law.

— Douglas Hofstadter

9.1 CONCLUSION

We managed to design and implement an image processor capable of handling simple video in real time. The final system meets all the functional and non-functional requirements laid down at the beginning of the project. We fulfilled the given assignment, as well as our own goals for the project.

Our goal of a frame rate of at least 10 fps was of particular importance. This goal was intended to keep us focused on the performance part of the assignment. The final system plays video at 13 frames per second, even when performing tasks such as edge detection.

We did not expect that the SD card would be a bottleneck in our system. However, when this turned out to be the case, it was fortunate that we were able to mitigate this by putting effort into optimizing the code.

Although our system achieves our goals, there are certainly areas left for further exploration. Some examples of this are detailed in the next section.

This project has been one of the most challenging and rewarding experiences during our endeavours into computer science. Given our limited previous experience in the area of hardware design, we are thrilled to have accomplished as much as we have.

9.2 FUTURE WORK

The limited number of cores we were able to fit on the FPGA is a major limitation of our system. Future work could be to work on reducing

the complexity of the SIMD nodes to fit more onto the FPGA, and another option may be to choose a larger FPGA.

Once the core design has been improved to fit more cores, or a much larger FPGA is used, we might consider adding some instructions we chose to strip out. Multiplication and division are two very useful instruction we did not include that, if included, would simplify general coding on the LENA architecture.

A second major limitation was the reading speed from the SD card. The chosen microcontroller, the AT32UC3A0512, has limited possibilities of input from mass storage devices. A possibly faster solution is Ethernet input, but this would require another computer to feed information to the system so the group as not looked into this possibilitiy.

Another possibility for improving input speed could be to switch to an AT32UC3A3xxx microcontroller, as these have support for High Speed USB (USB 2.0), which should be able to read much faster than our SD card reader.

Appendices

A

LENA

A.1 SIMD INSTRUCTION SET

SIMD nodes operates on a 24 bit instruction set divided in 2 main formats. Arithmetic instructions (R, I and S) and message passing instructions (M-send, M-store and M-forward).

Split this up into instruction set and assembler section?

A.1.1 *R Format (OP = 000)*

Arithmetic register functions instructions

CTRL	MASK	OP	RS	RT	RD	N/A	FN
1 bit	1 bit	3 bit	4 bit	4 bit	4 bit	4 bit	3 bit

Table A.1: Arithmetic register function instructions

- `ctrl` must be set to 0 in order to be executed on the SIMD node.
- `mask` is set to 1 to selectively enable the node when executing conditional branches.
- `op` the instruction opcode.
- `rs` write data register address.
- `rt` read data 1 register address.
- `rd` read data 2 register address.
- `n/a` not assigned for R instructions.
- `fn` the artimetic operation to perform.

NAME	FN	ASSEMBLY CODE	BINARY REPRESENTATION
ADD	000	`add $rs, $rt, $rd`	`0 m 000 ssss tttt dddd ---- 000`
SUB	001	`sub $rs, $rt, $rd`	`0 m 000 ssss tttt dddd ---- 001`
SLT	010	`slt $rs, $rt, $rd`	`0 m 000 ssss tttt dddd ---- 010`
AND	011	`and $rs, $rt, $rd`	`0 m 000 ssss tttt dddd ---- 011`
OR	100	`or $rs, $rt, $rd`	`0 m 000 ssss tttt dddd ---- 100`
EQ	101	`eq $rs, $rt, $rd`	`0 m 000 ssss tttt dddd ---- 101`
SLL	110	`sll $rs, $rt, $rd`	`0 m 000 ssss tttt ---- ---- 110`
SRL	111	`srl $rs, $rt, $rd`	`0 m 000 ssss tttt ---- ---- 111`

Table A.2: List of R instructions

CTRL	MASK	OP	RS	RT	CONST	FN
1 bit	1 bit	3 bit	4 bit	4 bit	8 bit	3 bit

Table A.3: Immediate functions using constants

A.1.2 *I Format (OP = 001)*

Immediate functions using constants.

- `ctrl` must be set to 0 in order to be executed on the SIMD node.
- `mask` is set to 1 for selectively enabling the node when executing conditional branches.
- `op` the instruction opcode.
- `rs` write data register address.
- `rt` read data 1 register address.
- `const` constant value — immediate.
- `fn` the artimetic operation to perform.

A.1.3 *S Format (OP = 010)*

Swap source data register with processed data and store new source data in register.

- `ctrl` must be set to 0 in order to be executed on the SIMD node.
- `mask` set to 1 for selectively enabling the node when executing conditional branches.

NAME	FN	ASSEMBLY CODE	BINARY REPRESENTATION
ADD	000	addi $rs, $rt, $rd	0 m 001 ssss tttt cccccccc 000
SUB	001	subi $rs, $rt, $rd	0 m 001 ssss tttt cccccccc 001
SLT	010	slti $rs, $rt, $rd	0 m 001 ssss tttt cccccccc 010
AND	011	andi $rs, $rt, $rd	0 m 001 ssss tttt cccccccc 011
OR	100	ori $rs, $rt, $rd	0 m 001 ssss tttt cccccccc 100
EQ	101	eqi $rs, $rt, $rd	0 m 001 ssss tttt cccccccc 101
SLL	110	slli $rs, $rt, $rd	0 m 001 ssss tttt -------- 110
SRL	111	srli $rs, $rt, $rd	0 m 001 ssss tttt -------- 111

Table A.4: List of I instructions

CTRL	MASK	OP	RS	RT	RD	N/A	FN
1 bit	1 bit	3 bit	4 bit	4 bit	4 bit	4 bit	3 bit

Table A.5: S format instructions

- `op` the instruction opcode.
- `rs` new source data register address.
- `rt` old source data register address.
- `rd` must be 000000.
- `n/a` not assigned for S instructions.
- `fn` must be 000.

NAME	FN	ASSEMBLY CODE	BINARY REPRESENTATION
SWAP	000	swap $rs, $rt	0 m 001 ssss tttt 0000 ---- 111

Table A.6: List of S instructions

A.1.4 *M Format (OP = 100, 101, 110)*

Send, receive and forward data from and to neighbor nodes in all directions (north, south, east and west).

- `ctrl` must be set to 0 in order to be executed on the SIMD node.
- `mask` set to 1 for selectively enabling the node when executing conditional branches.
- `op` the instruction opcode.

CTRL	MASK	OP	N	S	E	W	N/A
1 bit	1 bit	3 bit	4 bit	4 bit	4 bit	4 bit	3 bit

Table A.7: M format instructions

- `n` north data write or read register address.
- `s` south data write or read register address.
- `e` east data write or read register address.
- `w` west data write or read register address.
- `n/a` no ALU operations applicable.

NAME	OP	ASSEMBLY CODE
SEND	100	`send $r1, $r2, $r3, $r4`
STORE	101	`store $r1, $r2, $r3, $r4`
STORE & FORWARD	110	`frwrd $r1, $r2, $r3, $r4`
NAME	**BINARY REPRESENTATION**	
SEND	`0 m 100 nnnn ssss eeee wwww ---`	
STORE	`0 m 101 nnnn ssss eeee wwww ---`	
STORE & FORWARD	`0 m 110 nnnn ssss eeee wwww ---`	

Table A.8: List of M instructions

In Table A.8 we see the different .... STORE & FORWARD forwards according to the 4-way data exchange patterns (N $\rightarrow$ E, E $\rightarrow$ S, S $\rightarrow$ W, W $\rightarrow$ N).

TODO: *Something's amiss here*

M-instructions must be issued with mask bit set to 0 in order to get data from all the nodes. Otherwise, one will only get proper data from the ones executing within a conditional branch.

CHECK: *Should this be in an appendix, or?*

A.2 CONTROL CORE INSTRUCTION SET

TODO: *Fill in.*

A.3 LENA SCU BUS

The LENA has two types of pins connected to the SCU. There are 9 I/O pins and 29 input pins. The I/O pins are mostly used for the LENA to acknowledge transfers, but the 8 remaining pins can be used for the fallback solution of having LENA transfer data back to the SCU. The 29 input pins are used to set LENA's state, acknowledge

LENA and as a 24 bit wide one-way data bus used to transfer data and instructions to the LENA. Table A.9, A.10 and A.11 shows the pin mapping of the buses.

LINE	USAGE
LENA_IO_00..07	Data bus (LSB..MSB)
LENA_IO_CTRL	Interrupt / Ack from LENA.

Table A.9: LENA out bus

LINE	USAGE
LENA_IN_00..23	Data bus (LSB..MSB)
LENA_IN_24	Data bus ready

Table A.10: LENA in bus

LINE	USAGE
LENA_IN_25	Set LENA state
LENA_IN_26	State bus (least significant bit)
LENA_IN_27	State bus
LENA_IN_28	State bus (most significant bit)

Table A.11: LENA state bus

```
    ctrl nop

    # R1: Size of a screen buffer (320*240)
    ctrl addi R1 ZERO 150
    ctrl sll R1 R1
    ctrl sll R1 R1
    ctrl sll R1 R1
    ctrl sll R1 R1
    ctrl sll R1 R1
    ctrl sll R1 R1
    ctrl sll R1 R1
    ctrl sll R1 R1
    ctrl sll R1 R1

    # Set DMA read to active
    ctrl addi DMA ZERO 1
    ctrl dma set_read_active
    ctrl nop

    # Set DMA write to active
    ctrl addi DMA ZERO 1
    ctrl dma set_write_active
    ctrl nop

    # Set DMA read and write increments to 320
    ctrl addi DMA ZERO 160
    ctrl sll DMA DMA
    ctrl dma set_write_vertical_incr
    ctrl dma set_read_vertical_incr
    ctrl nop

    ctrl addi R4 ZERO 160           # R4 = 320
    ctrl sll R4 R4
    ctrl sll R4 R4
    ctrl sll R4 R4
    ctrl addi DMA R4 1              # DMA = R4 + 1 = 1280 + 1 = 1281
    ctrl dma set_write_horizontal_incr
    ctrl dma set_read_horizontal_incr
    ctrl nop

    ctrl addi R7 ZERO 0             # R7: Row start address accumulator
    ctrl addi R8 ZERO 80            # R8: Row counter

dma_loop:
    ctrl addi R6 ZERO 160           # R6: Column counter
    ctrl addi DMA R7 0
    node addi R14 ZERO 255

dma_row_loop:
    ctrl dma set_read_base_addr     # write base addr = 19328
```

```
51      ctrl subi DMA DMA 2
52      ctrl add DMA DMA R1
53      ctrl dma set_write_base_addr    # write base addr = 19328
54      ctrl sub DMA DMA R1
55      ctrl addi DMA DMA 4
56      ctrl nop
57
58      node swap R1 R1
59      ctrl dma start
60
61      node send R1 R1 R1 R1
62      node store R2 R3 R4 R5 # N S E W
63      #node fwrd R2 R3 R4 R5 # N S E W
64      #node store R6 R7 R8 R9
65
66      node add R10 ZERO R1
67
68      node srl R2 R2
69      node srl R2 R2
70      node srl R2 R2
71
72      node srl R3 R3
73      node srl R3 R3
74      node srl R3 R3
75
76      node srl R4 R4
77      node srl R4 R4
78      node srl R4 R4
79
80      node srl R5 R5
81      node srl R5 R5
82      node srl R5 R5
83
84      node addi R1 ZERO 127
85
86      node sub R1 R1 R2 # North
87      node add R1 R1 R3 # South
88      node add R1 R1 R4 # East
89      node sub R1 R1 R5 # West
90
91      node subi R1 R1 255
92
93      ctrl nop
94      ctrl nop
95
96      ctrl nop
97      ctrl nop
98      ctrl nop
99      ctrl nop
100     ctrl nop
101
102     ctrl nop
```

```
103     ctrl nop
104     ctrl nop
105     ctrl nop
106     ctrl nop
107
108     ctrl nop
109     ctrl nop
110     ctrl nop
111     ctrl nop
112     ctrl nop
113
114     ctrl nop
115     ctrl nop
116     ctrl nop
117     ctrl nop
118     ctrl nop
119
120     ctrl nop
121     ctrl nop
122     ctrl nop
123     ctrl nop
124     ctrl nop
125
126     ctrl subi R6 R6 1
127     ctrl beq dma_row_loop
128     ctrl slt ZERO ZERO R6
129     ctrl nop
130
131     ctrl addi R7 R7 160
132     ctrl addi R7 R7 160
133     ctrl addi R7 R7 160
134     ctrl addi R7 R7 160
135     ctrl addi R7 R7 160
136     ctrl addi R7 R7 160
137
138     ctrl subi R8 R8 1
139     ctrl beq dma_loop
140     ctrl slt ZERO ZERO R8
141     ctrl nop
142
143     ctrl addi R3 R1 0               # R3 = 320*240
144     ctrl addi VADDR ZERO 0          # VADDR = 0
145
146 pixel_loop:
147
148     ctrl lw VDATA R3
149     ctrl lw VDATA R3
150     ctrl nop
151     ctrl addi VADDR VADDR 1
152     ctrl addi R3 R3 1
153
154     ctrl beq infinite_loop
```

```
155     ctrl slt ZERO R1 VADDR
156     ctrl nop
157
158     ctrl jump pixel_loop
159
160 infinite_loop:
161
162     ctrl nop
163     ctrl jump infinite_loop
```

Listing A.1: Emboss code in SIMD.

B

PCB

B.1 OVERVIEW OF PIN USE

B.1.1 *FPGA*

TO COMPONENT	PIN TYPE	GENERIC PINS	INPUT PINS
Flash	Total	6	1
Oscillator	Total	0	1
AVR	Data	8	24
	Interrupt	1	0
	State	0	3
	Data ready	0	1
	State changed	0	1
	Total	9	29
Program memory	Data	24	0
	Address	16	0
	WE	1	0
	Total	41	0
Data memory	Data	8	0
	Address	21	0
	WE	1	0
	Total	30	0
VGA memory	Data	8	0
	Address	19	0
	WE	1	0
	Total	28	0
VGA	Hsync	1	0
	Vsync	1	0
	Value	10	0
	Total	12	0
USED		126	31
FREE		0	1

B.1.2 *AVR*

TO COMPONENT	PIN TYPE	PINS
Crystal	Total	2
FPGA	Data	32
	Interrupt	1
	State	3
	Data ready	1
	State changed	1
	Total	38
Memory	Data	8
	Address	19
	WE	1
	Total	28
VGA	Data	16
	TX	1
	RX	1
	Total	18
SD-card	Data	4
	$\overline{SS}$	1
	SCLK	1
	Total	6
RS232	RXD	1
	TXD	1
	RTS	1
	CTS	1
	Total	4
Buttons and LEDs	Buttons	4
	LEDs	8
	Total	12
USED		108
FREE		1

B.2 ORDERED PARTS

We ordered the following components. They were ordered from Farnell. All prices are in NOK.

NAME	PRODUCT ID	COUNT	PRICE	TOTAL
SD card reader	2081360	2	38.78	77.56
Serial port	1653975	2	8.63	17.26
Serial driver	1287435	2	24.25	48.50
USB port (Mini B)	1243250	2	17.89	35.78
VGA port	1056018	2	91.61	183.22
Data memory (2M × 8bit)	2115432	2	326.02	652.04
Program memory (64k × 16bit)	2115418	2	19.85	39.7
VGA memory (512 × 8bit)	1688739	2	57.43	111.88
Switch	3801287	25	1.84	46.00
LED	1716765	20	1.91	38.20
Voltage regulator adjustable	1685484	3	38.34	115.02
Voltage regulator 3.3V	1469037	1	31.96	31.96
Oscillator	1611846	1	26.09	26.09
Socket for Crystals	4695434	10	8.99	89.90
Crystal 12.0 MHz	1640871	2	10.21	20.42
Crystal 30.0 MHz	1842246	2	4.04	8.08

Table B.1: Components ordered in the first order

NAME	PRODUCT ID	COUNT	PRICE	TOTAL
Crystal 50.0 MHz	1843360	2	6.09	12.18
Crystal 40.0 MHz	1611785	2	3.45	6.90
Crystal 16.0 MHz	1640875	2	14.59	29.18
Crystal 20.0 MHz	1640878	2	14.59	29.18
Socket	224960	1	48.59	48.59
Flash	1605850	1	62.95	62.95
Capacitor 100μf	1889290	1	6.66	99.90
Resistor 105R	2142134	12	0.42	5.04
Resistor 39R	9337377RL	150	0.16	23.70
Resistor 360R	9336400	50	0.22	11.00
Resistor 191R	2139374	50	0.09	4.55
Resistor 100R	2074543	100	0.14	13.70
Resistor 0R	9240273RL	150	0.07	9.75
Micro SD 2GB	1795130	1	113.52	113.52
Header	9728856	2	22.32	44.64
Capacitor 47nf	1740718	60	1.97	118.20

Table B.2: Components ordered in the first order continued

NAME	PRODUCT ID	COUNT	PRICE	TOTAL
Program memory (64k $\times$ 16bit)	2115418	1	21.27	21.27
VGA memory (512 $\times$ 8bit)	1688739	2	55.94	111.88
Oscillator	1611846	1	26.09	26.09
Voltage regulator adjustable	1685484	4	38.34	153.36
Voltage regulator 3.3V	1469037	2	31.96	63.92
Flash	1605850	1	62.95	62.95

Table B.3: Spare parts ordered for the third board

B.3 MACAOS

This part is intended to provide a few insights about the ins and outs of using Macaos for creating production-files, as we feel it might be useful to note down what we learned for those that come after us.

1. Visit the Macaos homepage and apply for a license key[1], remember to leave some time for a reply on this, as these requests seem to be handled manually.

2. In Altium Designer, use File-Fabrication Outputs-Gerber Files. See Macaos documentation for decent settings, take specific care regarding the Digit format, and zero-suppression settings.

3. In Altium Designer, use File-Fabrication Outputs-NC Drill Files. As above, see the Macaos documentation for proper settings, they are likely the same as in the Gerber Files.

4. The necessary files should now exist in your Project folder under "Project Outputs for Projectname".

5. Install and open Macaos. You will be asked for a license key, which you should have gotten along with the download link.

6. Click "Import".

7. At this point it might be useful to have file endings enabled in Explorer, so open an Explorer window, and go to Tools, Folder-Options, and remove the checkmark for "Hide extensions for known file types".

8. Click "Open Files".

9. Select the relevant files for import, cross referencing against the Altium documentation might give a decent idea about which files go where. Importing all the files might be a good idea, as you might have an easier time selecting the correct files when watching the preview of each file.

10. Bind the files to their respective layers. Most of the layers should be easily spottable, like GTO for Notation Top, GTS for Top Solder Mask, GP1 and GP2 for internal layers 1 and 2. Take care to make sure these are not negative. If so then select Negative Layer 1/2 instead of Inner Layer 1/2. The Copper layers will likely be GTL and GBL. make sure these are correct too. Additional files include the Keepout (GKO), which can be used as "Board".

11. Verify that the drill files are properly selected, by clicking on "Drill Tools" in Macaos. You will probably have to link a text file ending in -RoundHoles.txt to "Drill". At this point it might also be a good idea to verify that the amount of holes match.

12. Verify that the list of layers in Macaos is complete.

1 http://www.macaos.com/products/me/get

13. Click “Contour” in Macaos, a Board contour will be auto generated for you.

14. Click “Board specs”, select a stackup. You would typically use 4001, which is listed as standard.

15. If wanted, you can select a different color for your board/silk layer here as well.

16. Click “Board stats”, check the hole amount, and set the “Minimum feature sizes”. These values depend a bit on what settings you ended up with in Altium, but usually 0.125 and 0.15 mm seem to be decent values.

17. Click “Place Product/Batch number”, and place that on your board. Alternatively skip this, and let Macaos pick a place for it.

18. Click “Publish to product browser”, which allows you to get a price estimate on your board.

19. If you did all this with your own license, you might also want to click “File, Save to local disk”, to generate a `.mei`-file that you can deliver to whoever will be doing the actual ordering.

Useful links:

- `http://www.macaos.com/products/me/get` — Applying for a Macaos-license
- `http://www.macaos.com/support/gerber/other` — Generating Gerber-files
- `http://wiki.altium.com/display/ADOH/Gerber+Output+Options` — Altium-information about generated Gerber-files.

The following figures will show schematics of our PCB.

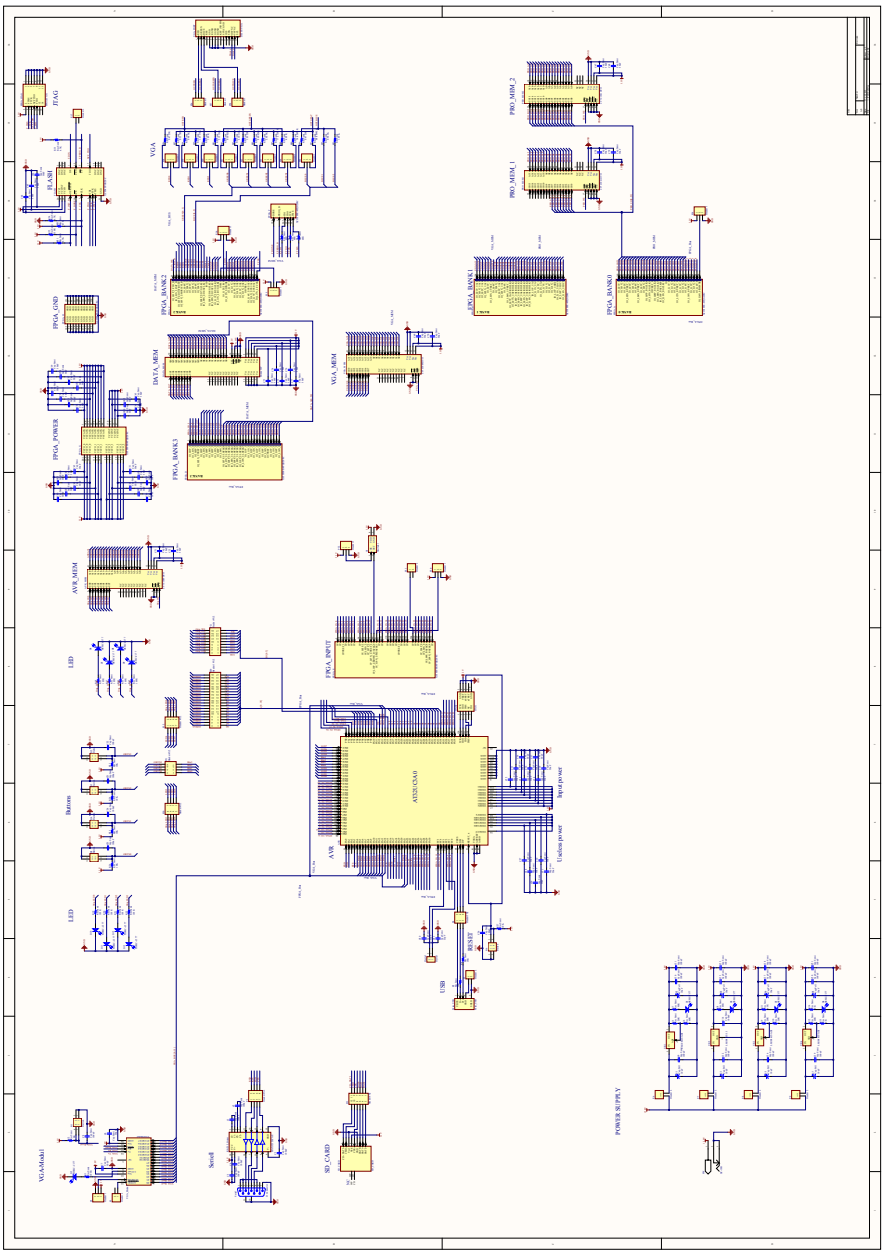

Figure B.1: PCB Schematic Overview

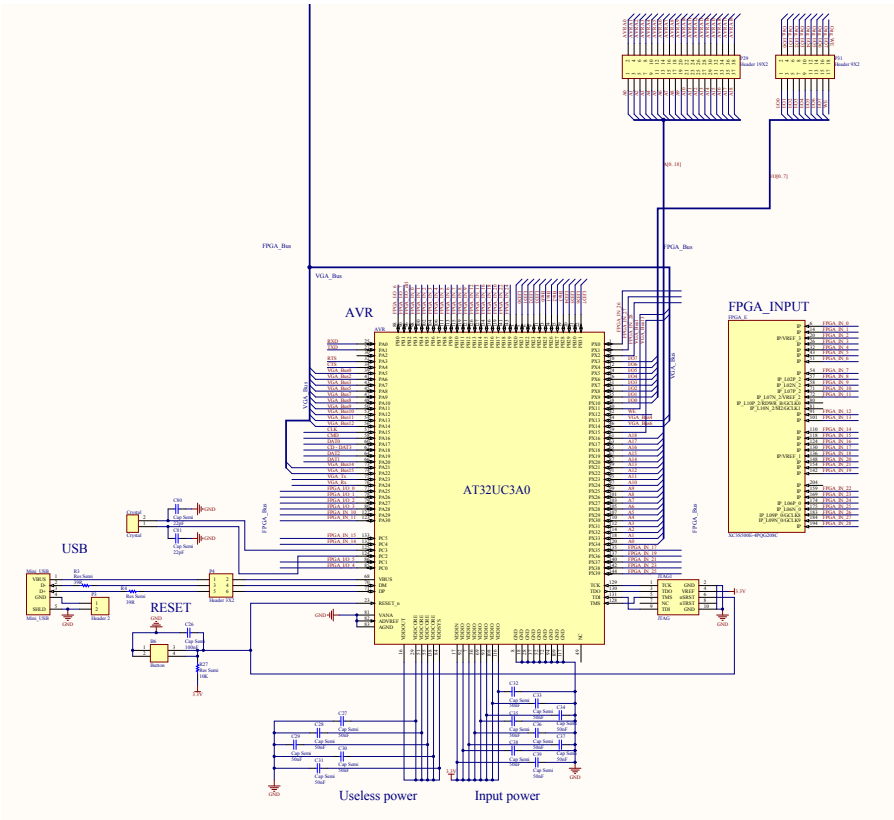

Figure B.2: AVR Schematic and FPGA Input

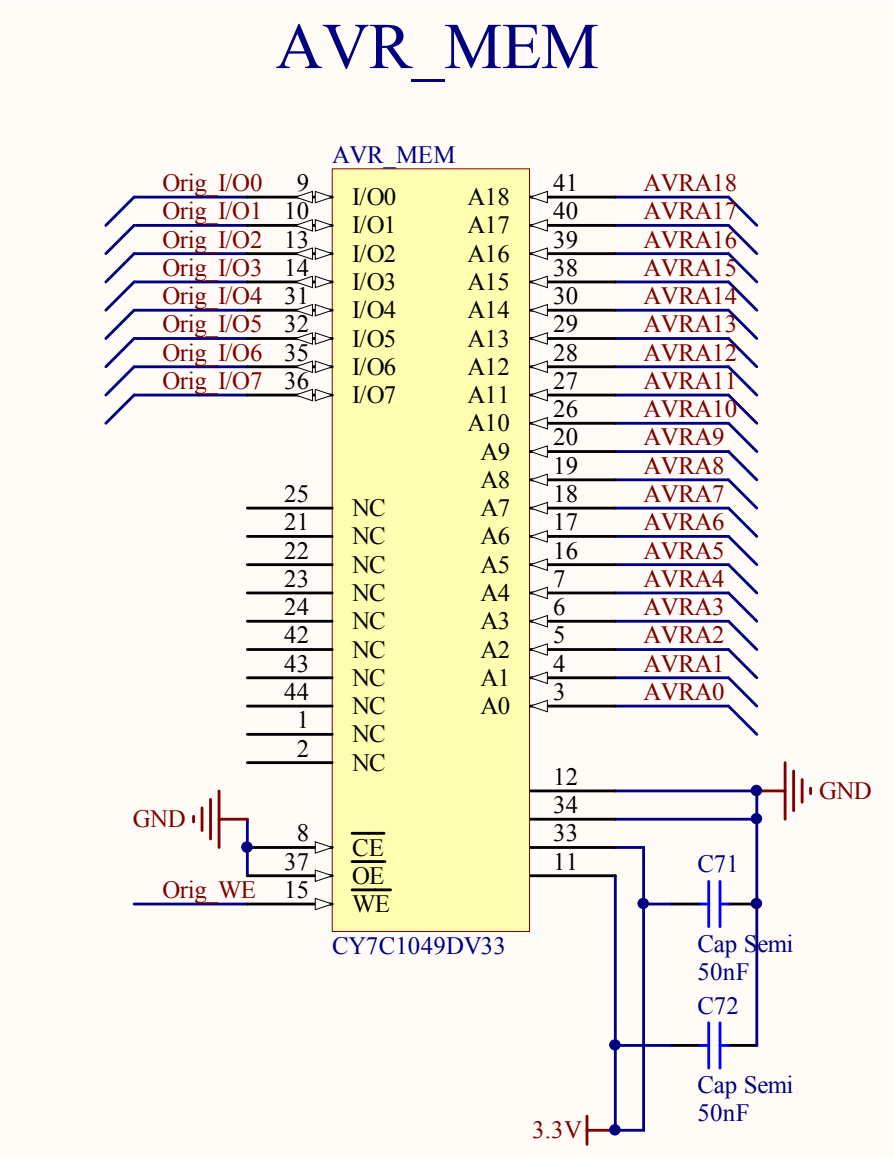

Figure B.3: AVR Memory

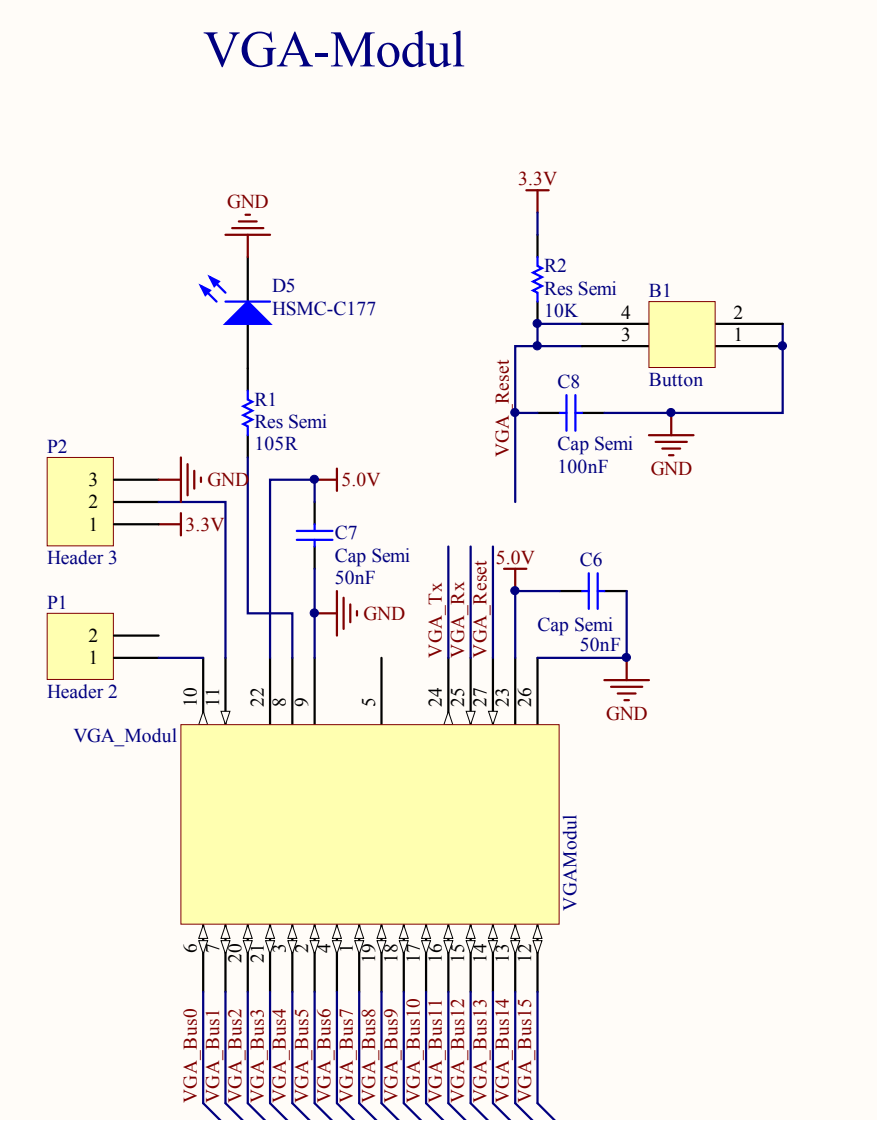

Figure B.4: VGA Module

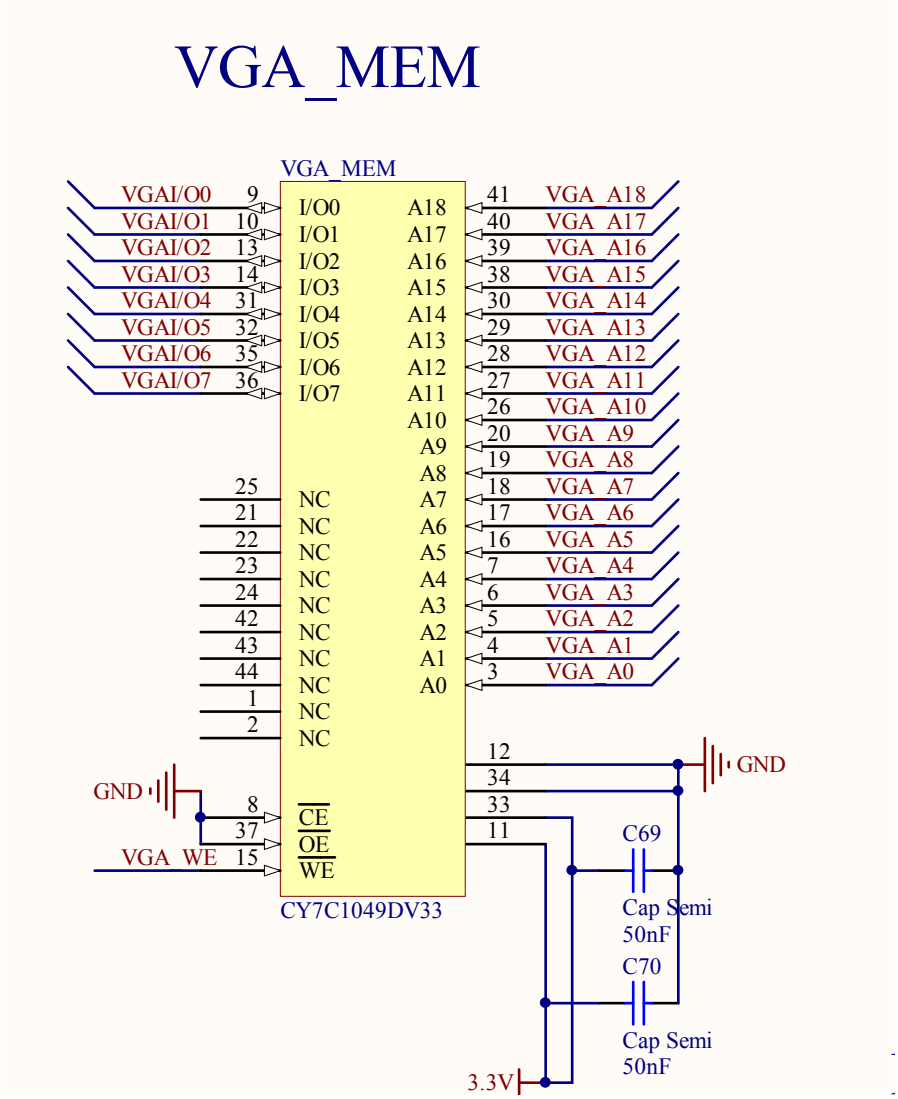

Figure B.5: VGA Memory

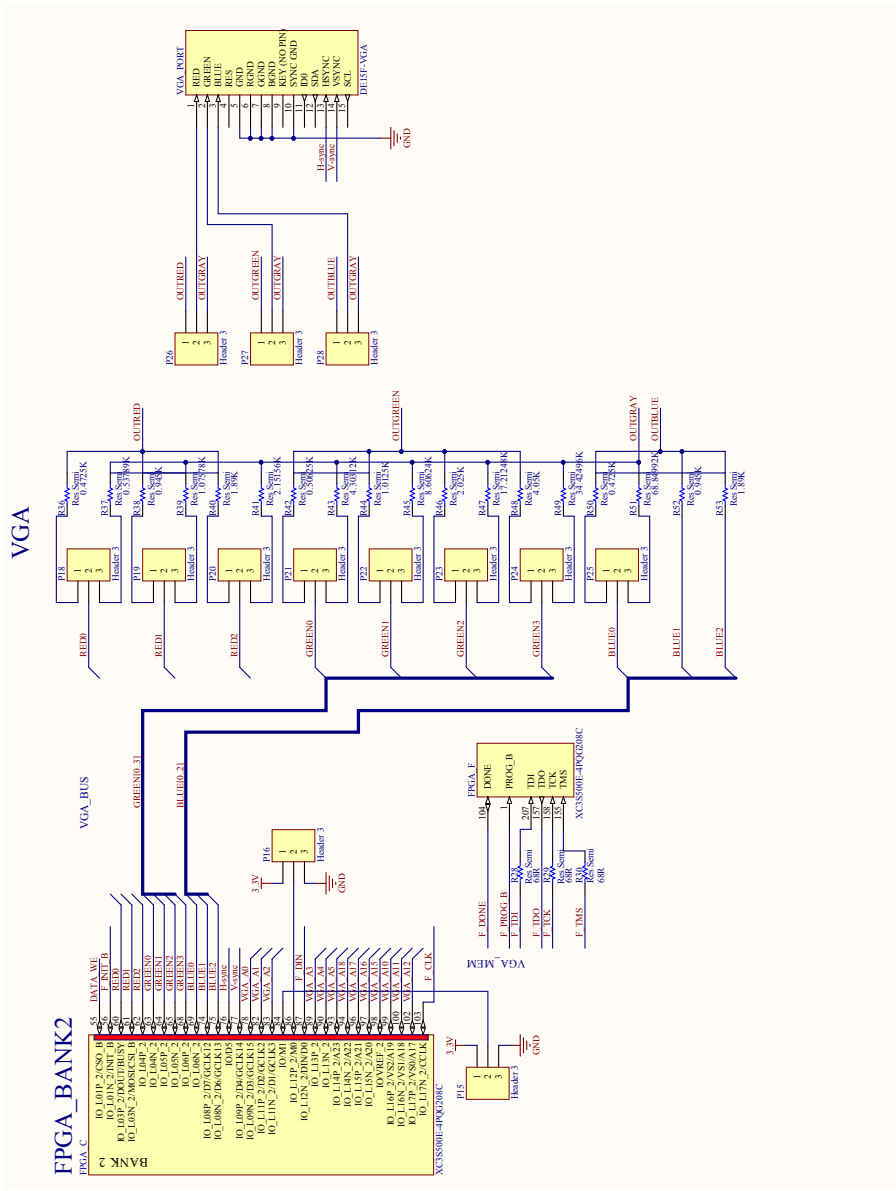

Figure B.6: VGA Controller

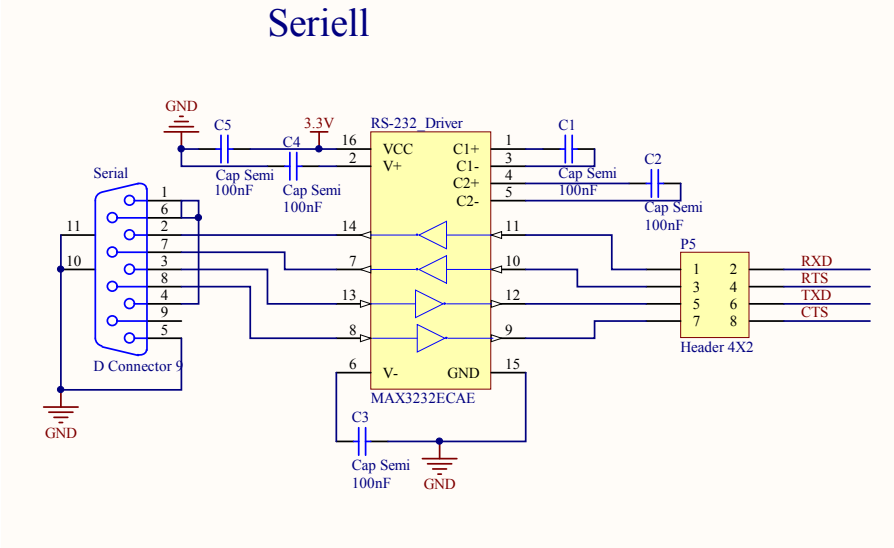

Figure B.7: Serial

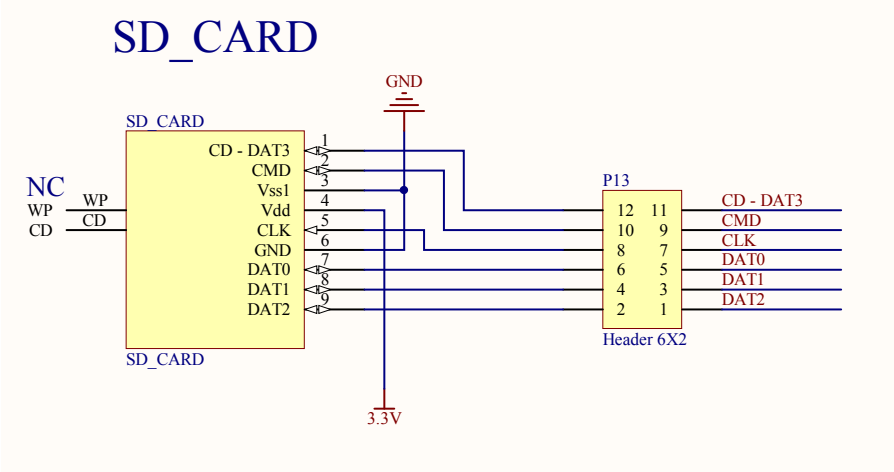

Figure B.8: SD Card Reader

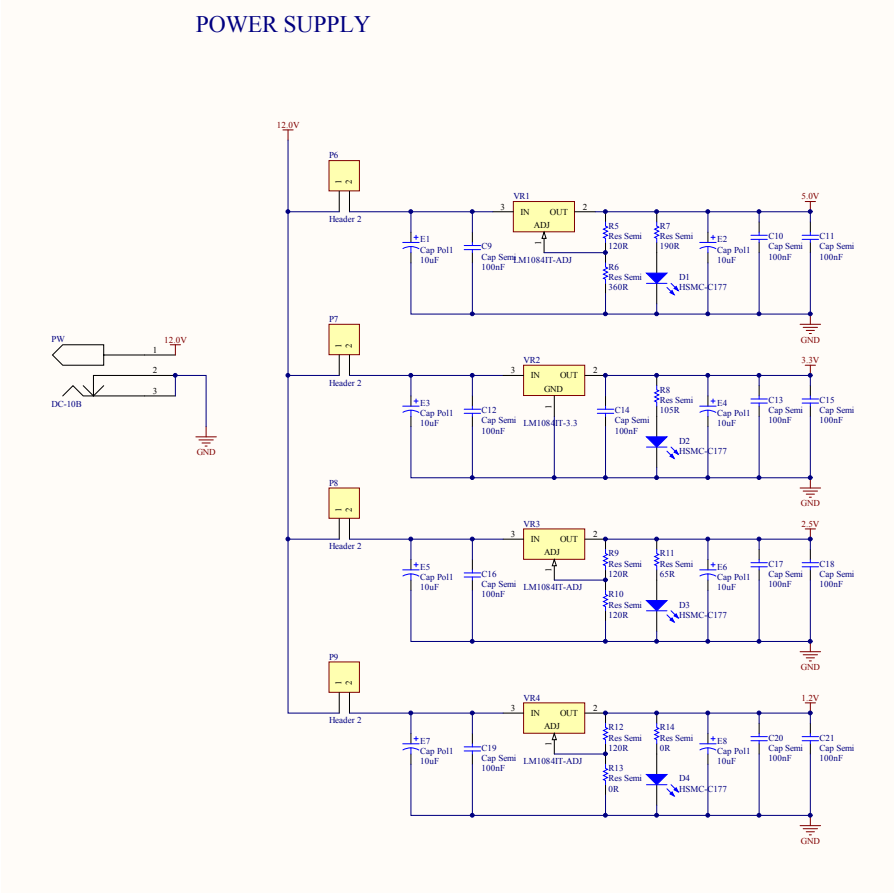

Figure B.9: Power Supply

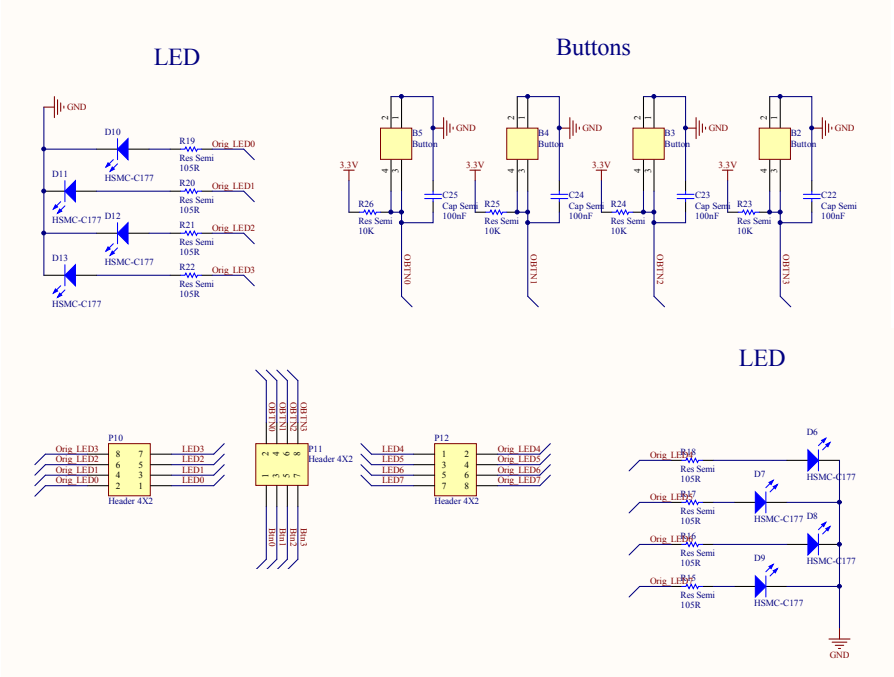

Figure B.10: LEDs and Buttons

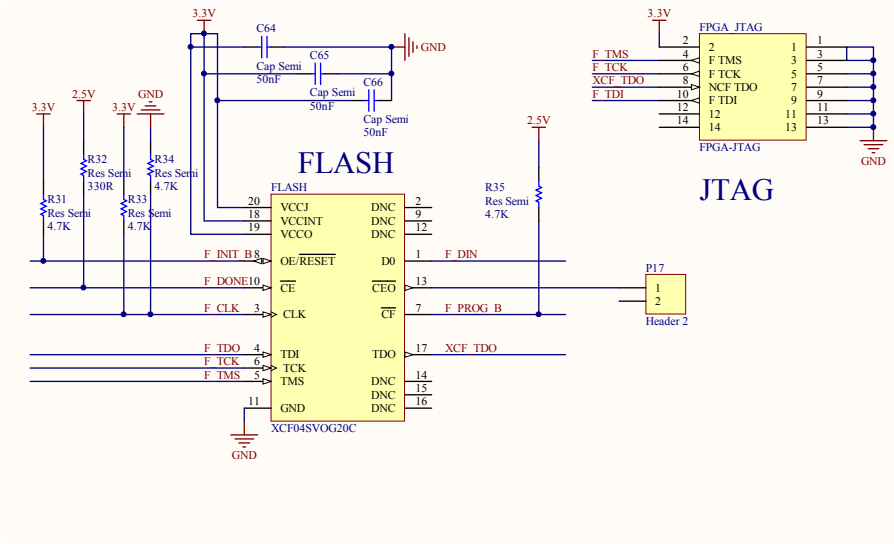

Figure B.11: FPGA JTAG Flash

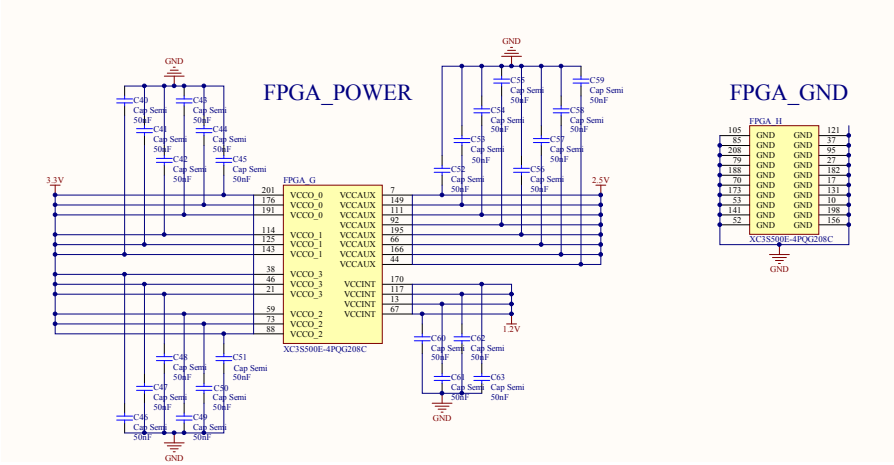

Figure B.12: FPGA Power Supply

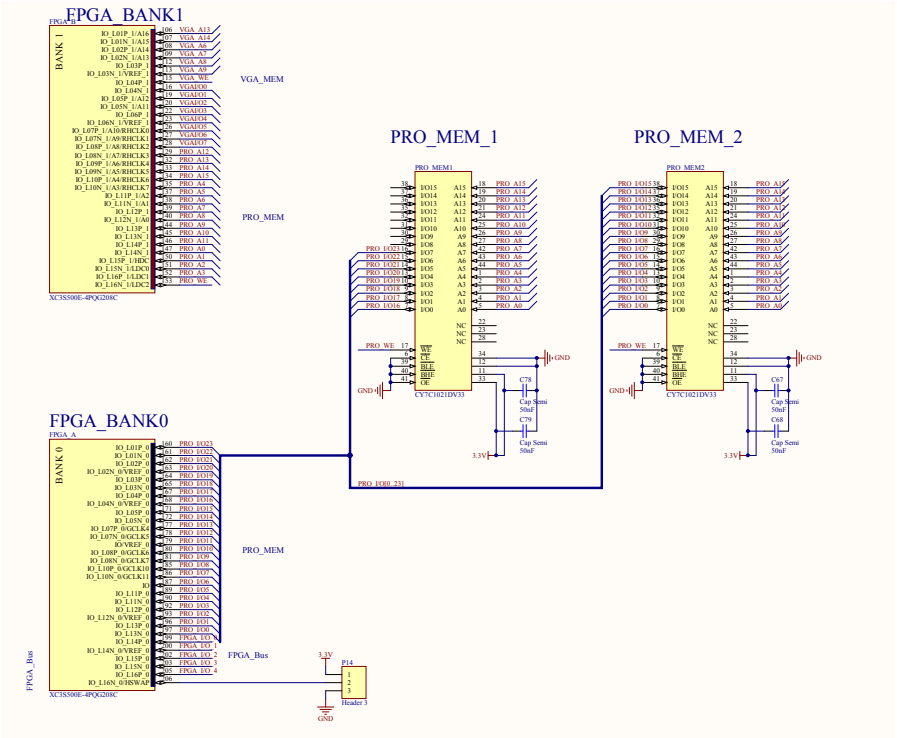

Figure B.13: FPGA Bank 0 and Bank 1

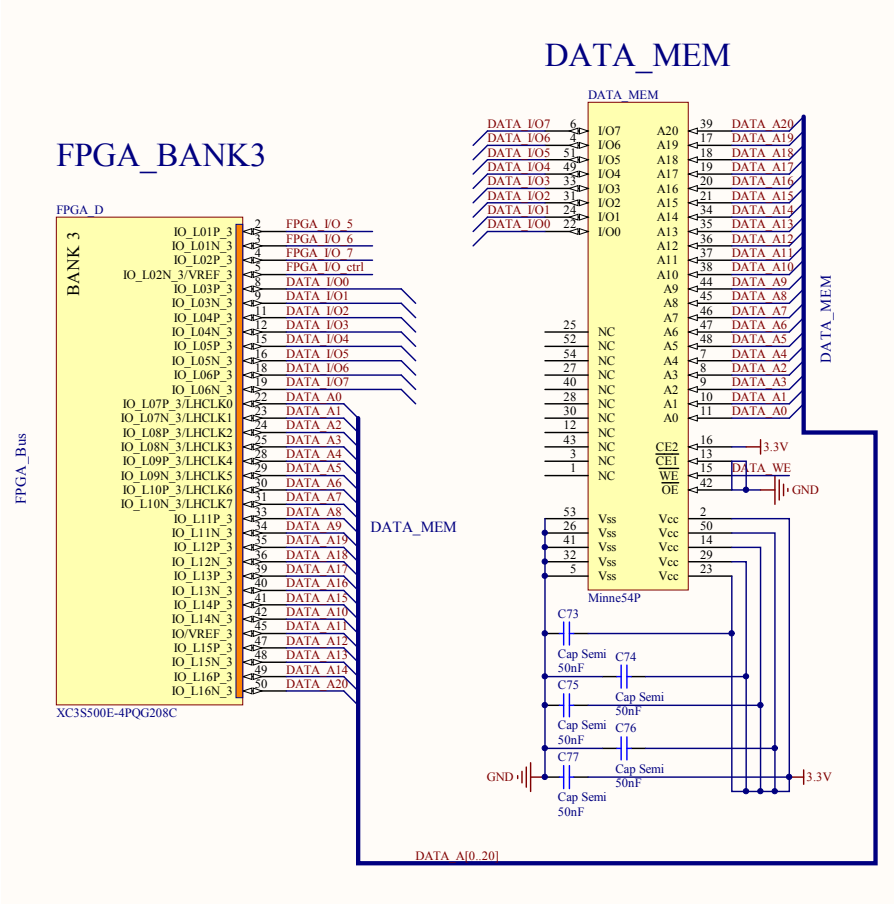

Figure B.14: FPGA Bank 3

[1] Romuald Aufrère, Jay Gowdy, Christoph Mertz, Chuck Thorpe, Chieh-Chih Wang, and Teruko Yata. Perception for collision avoidance and autonomous driving. *Mechatronics*, 13(10):1149 – 1161, 2003.

[2] K.E. Batcher. Design of a massively parallel processor. *Computers, IEEE Transactions on*, 100(9):836–840, 1980.

[3] S. Bell, B. Edwards, J. Amann, R. Conlin, K. Joyce, V. Leung, J. MacKay, M. Reif, L. Bao, J. Brown, et al. Tile64-processor: A 64-core soc with mesh interconnect. In *Solid-State Circuits Conference, 2008. ISSCC 2008. Digest of Technical Papers. IEEE International*, pages 88–598. IEEE, 2008.

[4] S. Berg, A. Behne Eie, M. Guise, J. Jansen, O.P. Skarre Lund, S. Natvig, and L. Hesselberg Simonsen. Festina lente, a parallel image processor. Technical report, Norwegian University of Science and Technology, 11 2011.

[5] R.J. Ferrari, H. Zhang, and C.R. Kube. Real-time detection of steam in video images. *Pattern Recognition*, 40(3):1148 – 1159, 2007.

[6] J. Holt. *UML for systems engineering watching the wheels*. Institution of Electrical Engineers, London, 2004.

[7] Kai. Hwang and Doug. DeGroot. *Parallel processing for supercomputers and artificial intelligence*. McGraw-Hill, New York, 1989.

[8] P. Kongetira, K. Aingaran, and K. Olukotun. Niagara: A 32-way multithreaded sparc processor. *Micro, IEEE*, 25(2):21–29, 2005.

[9] Huynh Van Luong and Jong Myon Kim. A massively parallel approach to affine transformation in medical image registration. In *High Performance Computing and Communications, 2009. HPCC '09. 11th IEEE International Conference on*, pages 117 –123, june 2009.

[10] III Miller, W.T. Real-time application of neural networks for sensor-based control of robots with vision. *Systems, Man and Cybernetics, IEEE Transactions on*, 19(4):825 –831, jul/aug 1989.

[11] B. Moseley and P. Marks. Out of the tar pit. 2006.

[12] K. Olukotun and L. Hammond. The future of microprocessors. *Queue*, 3(7):26–29, 2005.

[13] S.R. Sternberg. Biomedical image processing. *Computer*, 16(1):22 – 34, jan 1983.

[14] S. Thrun, M. Montemerlo, H. Dahlkamp, D. Stavens, A. Aron, J. Diebel, P. Fong, J. Gale, M. Halpenny, G. Hoffmann, et al. Stanley: The robot that won the DARPA grand challenge. *The 2005 DARPA Grand Challenge*, pages 1–43, 2007.

[15] Wikipedia. Goodyear MPP — Wikipedia, the free encyclopedia, 2012. [Online; accessed 30-October-2012].