Norwegian University of Science and Technology

Demolicious
A General-Purpose Graphics Processing Computer

TDT4295 Computer Design Project

Authors:

Aleksander Gustaw Wasaznik

Aleksander Vognild Burkow

Kristian Fladstad Normann

Christoffer Tønnessen

Andreas Løve Selvik

Julian Ho-yin Lam

Eirik Jakobsen

Stian Jensen

Torkel Berli

November 25, 2014

[This page is intentionally left blank.]

Abstract

Demolicious is a general purpose SIMT inspired computer. It is specialized to handle parallel rendering of computer graphics. The performance of the system scales linearly with the amount of cores.

In addition, this system is the first of its kind in this project with an HDMI output. Computer generated graphics was chosen as a showcase application, and was successfully demonstrated on an HDMI enabled screen.

It features a multi-core architecture, and is designed to maximize processor core utilization by hiding memory latency.

The system has been implemented on a PCB, using a backup oriented design philosophy, and has passed testing and verification.

Acknowledgements

A huge thank you goes out to the following people for tremendous help during the project:

Gunnar Tufte for guiding us towards a finished, working product.

Yaman Umuroğlu for great project coordination and answers to questions at close to any hours of the day.

Odd Rune S. Lykkebø for fantastic email response time, anywhere in the world.

Omega Verksted for help with electronic circuits, as well as being able to provide endless amounts of headers at 4 a.m.

Elster's fridge for storing our food throughout the semester.

Geir Kulia for providing us with dinner when we were too swamped to leave the lab.

Contents

I Introduction 1

1 Computer Design Project 2
1.1 Assignment . . . 2
1.1.1 Original Assignment Text . . . 3
1.2 A Parallel Processing Accelerator . . . 4
1.3 Structure of the Report . . . 4

2 Modern Graphics Processing Units 5
2.1 What is a GPU . . . 5
2.2 The GPU Market . . . 5
2.3 An Enormous Task . . . 6
2.4 Taking Advantage of Parallelism . . . 6
2.5 At The Mercy of Memory . . . 7
2.6 Shortcomings of a GPU . . . 7

3 NVIDIA's Fermi Architecture and CUDA Programming Model 9
3.1 The Fermi Architecture . . . 9
3.2 CUDA Programming Model . . . 9
3.3 Streaming Multiprocessors . . . 11
3.4 Warps of Threads . . . 12
3.5 Helpful Inspiration . . . 12

4 Demolicious 13
4.1 A Graphical Demo Machine . . . 13
4.2 Demolicious Requirements . . . 14

II Solution 15

5 System Overview 16
5.1 Logical System Structure . . . 16
5.2 Programming Model . . . 17
5.2.1 Executing Kernels . . . 17
5.2.2 More Advanced Kernels . . . 19

6 CPU 21
6.1 Functionality . . . 21
6.2 Communication with the GPU . . . 21
6.3 Loading a Kernel . . . 22
6.4 Running a Kernel . . . 23

6.5 Loading parameters 24
6.5.1 Summary 25

7 GPU **26**
7.1 Responsibilities 26
7.2 Architecture Overview 27
7.3 Receiving a Kernel Call 27
7.4 Running a Kernel 28
7.4.1 Warps 28
7.4.2 Static Scheduling 28
7.5 Module Details 31
7.5.1 Processor Core 31
7.5.2 Thread Spawner 32
7.5.3 Register Directory 34
7.5.4 SRAM Arbiter 36
7.5.5 Load/Store Unit 36
7.5.6 HDMI 37
7.5.7 Summary 38

8 Physical Implementation **39**
8.1 Backup oriented design 40
8.2 Input 41
8.3 Output 41
8.3.1 HDMI implementation 41
8.4 Power 41
8.5 Bus 42
8.6 Clocks 42
8.7 Main Components 42

9 Additional Tools **44**
9.1 Assembler 44
9.2 Simulator 44

III Results & Discussion **45**

10 Testing **46**
10.1 GPU Component testing 46
10.2 VHDL system integration tests 47
10.2.1 Memory Stores and Kernel Parameterization 48
10.2.2 Predicated Instruction Execution 49
10.2.3 Loads from Primary Memory 50
10.3 Verification of Communication Channels 52
10.3.1 JTAG test 52
10.3.2 EBI Bus 53
10.3.3 HDMI Output 53
10.3.4 SRAM Communication 53
10.4 PCB tests 53
10.4.1 Solder, signal and power test 54
10.4.2 Oscillator and clock test 54

11 Results **55**
11.1 Scalability of the Demolicious system 56
11.2 Performance 56
11.3 Video output 58
11.4 Single vs double buffering 59
11.5 Demolicious Power efficiency 60

12 Discussion **63**
12.1 Energy Efficiency 63
12.1.1 Optimizing FPGA power usage 63
12.1.2 Physical Implementation 64
12.2 Programming Challenges 64
12.3 SRAM - Alternative Memory Layouts 65
12.4 Multiple clock domains in HDMI unit 65
12.5 Troubleshooting and Workarounds 65
12.6 Hardware components 68
12.6.1 Clocks 69
12.7 Budget 70

13 Conclusion **71**
13.1 Further Work 71
13.1.1 Architecture 71
13.1.2 Physical design 72
13.1.3 Compiler 72

List of Tables **74**

List of Figures **74**

IV Appendices **78**

A Expenses **79**
A.1 PCB manufacturing 80
A.2 Component purchases 81

B Instruction set architecture **82**
B.1 Registers 83
B.2 Predicated instructions 84
B.3 Instructions 85
B.3.1 R-type Instructions 85
B.3.2 I-type Instructions 86
B.3.3 Pseudo Instructions 86

C Commented tunnel kernel **87**

D PCB schematics **91**

Part I

Introduction

Chapter 1

Computer Design Project

This report presents the TDT4295 Computer Design Project at NTNU for the fall semester of 2014.

The course is held every fall, and consists of a single task in which a group of students make a working computer from scratch. This report's group was made out of 9 students from the Computer Science department. Gunnar Tufte and Yaman Umuroğlu served as advisors for the group throughout the semester and assisted in administrative tasks.

1.1 Assignment

The Computer Design Project's primary tasks included making a custom printed circuitboard (PCB) and implementing a custom processor architecture on an FPGA (Field-Programmable Gate Array). Together with a microcontroller (MCU) and a choice of I/O components, these were to form a complete and working system. The project is evaluated based on this report and an oral presentation of the work, as well as a prototype demonstration.

The specific assignment for this year was to create a processor inspired by GPU architectures. Core requirements included having multiple processor cores and a graphical display output.

1.1.1 Original Assignment Text

1.1.1.1 Construct a graphics processing unit (GPU) inspired processor

GPUs play a large role in graphical applications as well as high performance computing. They are typically constructed around the SIMD (single instruction multiple data) paradigm and include special hardware for accelerating graphics-related operation. The idea is to make a GPU-inspired processor architecture that exploits the possibility of parallel computation on a single chip. The GPU must be a multi-core system.

1.1.1.2 Additional requirements

Your processor will be implemented on an FPGA, and you are free to choose how to realize your computer architecture. Studying the architecture of general multi-core processors and parallel machines options can be a good starting point.

Energy efficiency should be a primary consideration in all phases of the project, from early design decisions to how software is written.

The task should also include a suitable application that can produce a graphical output on a display to demonstrate the processor.

The unit must utilize a Silicon Labs EFM32 series microcontroller (to act as an I/O processor) and a Xilinx FPGA (to implement your architecture on). The budget is 10 000 NOK, which must cover components and PCB production. The unit design must adhere to the limits set by the course staff at any given time.

1.2 A Parallel Processing Accelerator

The group decided to make the custom GPU inspired processor as an accelerator processor for parallelizable computations. This means that it would not be designed to run entire programs on its own, but would instead be tasked with particular parallelizable parts of a program. Its purpose would primarily be to run graphics related operations. However, it would be designed to be programmable with general arithmetic and logic operations, and not have specialized hardware units for performing graphics operations. The group was determined to also send graphics output directly from the accelerator to screen, as is common for a GPU.

The accelerator design needed be implemented on an FPGA. The FPGA was to be mounted on a PCB together with a microcontroller, memory, and HDMI port, and form a graphical computer system similar to how modern PCs are organized.

1.3 Structure of the Report

The accelerator processor will fill the role of a modern GPU in the system presented by this report. Therefore, it will be referred to in this report as the GPU.

Having introduced the problem, the report will continue by giving a short introduction to what a GPU is and their design. It will present concepts and challenges in graphical computations and GPU design. These topics help to appreciate the need for a GPU in modern computers, and to understand the trade-offs and optimizations involved in its design.

The system created in this project, and its purpose, will be introduced in light of modern GPU concepts. In the solution part, a detailed explanation of the system will be given by roughly following the path of an executing program, going from high to low levels of abstraction.

The physical product produced in this project, and the test and verification methods involved in making it work, will also be presented. The results chapter will go through which parts of the system worked and which did not, and will look at measurements of its performance. Lastly, the discussion chapter will comment on some of the difficulties with the completed system, and on topics such as possible further work.

Note that the reader of this report is expected to have a basic understanding of logic design, computer components, and programming.

Chapter 2

Modern Graphics Processing Units

2.1 What is a GPU

Modern GPUs are, in a way, an evolution of former Video Graphics Array (VGA) controllers [2]. A VGA controller of the early 1990s served as a memory controller and display generator that wrote framebuffer values to a display. As technology advanced, it received hardware to perform specific graphics related functions. This eventually evolved into a processor, with its own memory, that incorporated a full set of graphical functions.

A GPU's primary purpose has traditionally been to offload graphical calculations from the CPU and render images to a screen. Graphical functions are accessed through APIs like DirectX and OpenGL. Today, GPUs also have general computing capabilities and may serve as co-processors for the CPU in addition to handling their graphical duties. Non-graphics applications for a GPU include image processing, video encoding, and many scientific computing problems and other large, highly regular calculations.

2.2 The GPU Market

In general, at least one GPU is present in every PC these days, and the market for PCs has been relatively stable for many years [5]. These GPUs can take the form of discrete chips or be integrated with the CPU. Intel, AMD, and NVIDIA are the big actors in the PC GPU market [14]. Intel is largest overall but only makes GPUs integrated with their CPUs. AMD and NVIDIA share the discrete GPU market [13]. One of NVIDIA's GPU architectures will be used as an example later. A big market for the more powerful discrete GPUs is the computer gaming industry. This market has contributed a great deal to the rapid progression of graphics technologies. With the advent of general purpose GPUs, scientific computing has also gained an interest in powerful GPUs.

In the past 6 years, the booming market for mobile smart devices has introduced a new arena for GPUs. Mobile devices are currently outselling workstations, and every new mobile device sold today is shipped with a small-format GPU. These GPUs are generally integrated in the device's system-on-a-chip (SoC), and have slightly different design considerations than traditional workstation GPUs. As for everything else that runs on batteries, power consumption is a primary concern in this format. Traditionally, this is a concern that GPU design has not needed to prioritize. The mobile GPU format has opened up for other GPU design companies than the three mentioned above. The three dominant GPU design companies in the SoC market are Qualcomm, ARM, and Imagination Technologies [15].

2.3 An Enormous Task

Producing computer graphics is a highly processing intensive task. Consumers expect their computers to display videos and games in high resolutions and at high framerates. To color one pixel accurately in a 3D environment, the processor typically needs to calculate vectors in 3D space, interpolate texture data, adjust for light intensity, and more. All of these tasks require several demanding floating point operations.

A CPU is optimized for serial programs, and will compute a picture frame in such a fashion. The problem is that, even with the processing power of 4 GHz, it is not able to handle the amount of calculations well enough for modern graphical demands. Fortunately, pixels can often be computed independently of each other. A CPU fails to take advantage of this.

2.4 Taking Advantage of Parallelism

For a CPU, latency is of primary concern because the next instruction often depends on the result of the previous one. Therefore, the CPU needs to complete each instruction as quickly as possible. This makes the CPU very good at problems with a low level of parallelism, which after all characterizes most programs. Finishing an instruction as quickly as possible is less of a concern when problems are more parallel in nature. A GPU therefore optimizes for throughput instead of latency. It gains throughput by making its architecture highly parallel in all respects.

The fast single-threaded cores in CPUs are replaced with a large quantity of smaller, slower cores. A GPU gains throughput by spending resources on execution units instead of more cache, prediction logic, or dynamic reordering logic to make one execution unit very effective. It also exploits parallelism with its deep pipelines, executing many instructions concurrently within each core. These architectural decisions cause each individual thread to require more wait cycles to wait for memory access or pipeline hazards. Fortunately, a GPU is not concerned about each individual thread's tardiness, and instead fills these wait cycles by interleaving many threads at once in each core. Filling every cycle

and every pipeline stage with useful work is, of course, essential in optimizing throughput.

2.5 At The Mercy of Memory

A GPU can achieve a very high throughput because of its massive amount of parallelism. But with great computing power comes great memory demand. Keeping all the computational units in a GPU supplied with enough data is a difficult challenge. Using NVIDIA's top-end GPU in 2006, GeForce 8800, as an example, it can process 32 pixels per clock at 600 MHz [2]. A pixel will usually be 4 bytes in size. Calculating its value will typically require a read and write of color and depth, and a few texture element reads. So, on average, there may be a demand of 7 memory accesses of 4 bytes each per pixel. To sustain 32 pixel values per clock, this requires up to 896 bytes per clock, or 537 GB/s, in memory transfers.

Many techniques are used by the memory system to meet such demanding requirements. First of all, the memory system consists of many different types of memories that are optimized for different access patterns. Some data must be globally available to all threads, others are local to one or a handful of threads. Some data is read only, some data is accessed with a high spatial locality in 3D space, and so on. Tailoring the memory system to suit important and commonly used access patterns is key to high performance. Some examples of memory types that NVIDIA use in their GPU architectures are texture memory, constant memory, local memory, and shared memory.

The different types of memories are usually organized into several separate banks, often with their own memory controller, so that more requests can be handled at a time. Interleaving consecutive virtual addresses over different physical memory banks will assist in spreading requests evenly across the set of banks. All memories typically have multiple levels of cache associated with them as well. Other techniques to improve memory bandwidth include data compression to reduce the amount of bits to transfer, and bundling memory accesses into sets that the memory system handle well. Bundling adds latency to memory requests. But, as mentioned earlier, this is not an issue as the GPU has plenty of other work to do while a particular thread waits.

2.6 Shortcomings of a GPU

A GPU obviously does its parallel calculations effectively, but it comes at a cost. It gains throughput over a CPU in part by sacrificing low latency in individual instructions. As we have seen, this happens as a result of deep pipelines and delayed memory requests, among other things. Such architectural features reduce the performance of serial programs, especially ones with a lot of branching. Branching instructions on a deeply pipelined processor is heavily punished by control hazards.

Since most programs are not particularly parallelizable and commonly contain branching, a GPU is insufficient for running an average program by today's standards. Computer users today are accustomed to the performance provided by the cooperation of a CPU and a GPU, a so-called heterogeneous computer system. This is so essential that no computer, or even mobile unit, with a graphical display is produced today without the combination of both.

Chapter 3

NVIDIA's Fermi Architecture and CUDA Programming Model

To explore some concepts of modern GPU architectures in detail, this chapter will take a look at an example from NVIDIA. The programming model used by NVIDIA for its modern GPUs, called CUDA, will be be examined. Then, NVIDIA's first architecture to use this programming model, the Fermi architecture, will be used as the example for illustrating GPU design concepts. The system presented in this report is heavily inspired by NVIDIA, and the following concepts are relevant for both its programming model and its architecture.

3.1 The Fermi Architecture

In 2006 NVIDIA released their first GPUs with a so called *unified shader architecture*. Before this, GPUs had separate, dedicated hardware for all common graphics operations. Beginning with the Fermi architecture, the majority of operations are executed on the same hardware. The hardware is able to perform general operations. This significant difference compared to earlier GPU microarchitectures has enabled other calculations on a GPU than graphics processing. This ability been given the term General-Purpose GPU computing, or GPGPU.

3.2 CUDA Programming Model

To expose the computing powers of the graphics card for general applications, there was a need for a new application programming interface (API) that could be used to run code on the GPUs. It was possible to do non-graphical calculations on GPUs previously, but it required redefining any problem as a graphics

problem. This was highly impractical. With the Fermi architecture, NVIDIA released a framework for running code on their GPUs called CUDA.

CUDA is an extension to the programming language C, and it lets the programmer write functions for executing on the GPU. Such a function is called a *kernel*. An example will be used to illustrate how a kernel executes on the Fermi architecture.

Let's say you want to fill a frame buffer, an area of memory, with the color green. In a sequential programming model, you would typically write a loop that would fill the memory locations with the value for green one by one. See Listing 3.1.

```
1 int green = 0x00FF00;
2 for (int i = 0; i < nr_of_pixels; i++){
3   framebuffer[i] = green;
4 }
```

Listing 3.1: A sequential program filling the screen with green

For a CUDA kernel, on the other hand, same program would be written as if filling only a single pixel with green. The kernel would then be executed by one thread for every pixel on the screen. Each thread would receive a unique id number corresponding to the 'i' value from the serial example. Below, in Listing 3.2, is an example CUDA kernel that fills a single memory location, or pixel, with green.

```
1 __global__ void fill_screen(int* framebuffer){
2   /* Calculate global id of this thread */
3   int global_id = blockIdx.x * blockDim.x + threadIdx.x;
4   int green     = 0x00FF00;
5   framebuffer[global_id] = green;
6 }
```

Listing 3.2: A CUDA kernel filling a single pixel with green

The CPU will need to load a kernel into the GPU at run time before calling it. When calling a kernel, the CPU specifies how many threads to run. Managing memory, uploading kernels, and calling kernels on the GPU is done using CUDA specific syntax. Listing 3.3 shows example CUDA code for calling the "fill_screen" kernel.

```
1 int* frame_buffer_device = cudaMalloc(data_size); // allocate memory at
      GPU
2 fill_screen<<<1080,1920>>>(frame_buffer_device); // call kernel
```

Listing 3.3: Starting the CUDA kernel with one thread per pixel on a 1920x1080 screen

Running a separate thread for each small task, like in CUDA, has another more general term, SIMT. SIMT is short for Single Instruction Multiple Threads. It is a common programming paradigm for modern GPUs because it a very flexible way of writing parallel programs. Its programs are easily portable and scalable

because they ignore the underlying architecture of the machine.

3.3 Streaming Multiprocessors

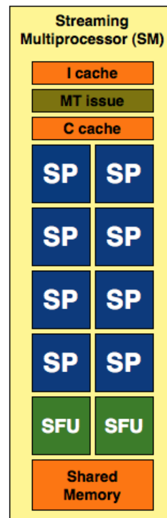


Figure 3.3.1: Streaming Multiprocessor

A CUDA function typically creates many, many threads. To handle all these, the Fermi architecture has a large number of cores organized into a hierarchy of groups that share different amounts of extra resources. At its heart lies 16 Streaming Multiprocessors (SMs).

A Streaming Multiprocessor is a collection of tightly coupled Streaming Processor cores (SP) along with some shared accessories. To enable a large number of cores, NVIDIA makes each of the SP cores simple. For example, common but difficult functions like sin and reciprocal are delegated to a few separate hardware units within the SM, called Special Function Units (SFU).

In Fermi, each SM contains 32 SP cores. Each core is capable of doing both integer and floating point operations. Their shared accessories include four Special Function units, 16 Load/Store units, and 64KB SRAM for shared memory and shared cache. A simplified diagram of a smaller SM is shown in Figure 3.3.1.

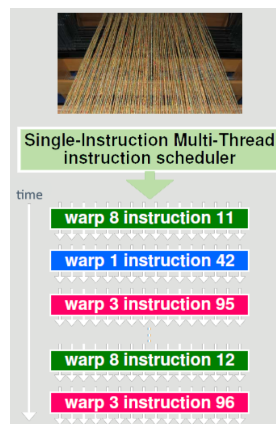


Figure 3.3.2: Warps of 32 threads execute

3.4 Warps of Threads

Threads in Fermi are grouped together in logical units of up to 32 threads. This corresponds to the 32 Streaming Processor cores in a Streaming Multiprocessor. The 32 threads are collectively called a *warp*. Threads in the same warp always execute the same instruction at the same time. This way, only a single instruction fetch operation per warp is required for the SM.

Many warps may be active in an SM at any time, and each thread in each warp has its own set of registers within the SM. This enables fine-grained interleaving of warps without needing expensive context switches with every warp switch. A simplified example execution order is shown in figure 3.3.2. Scheduling warps is difficult, but it is important for filling the pipeline with useful instructions, as discussed in chapter 2

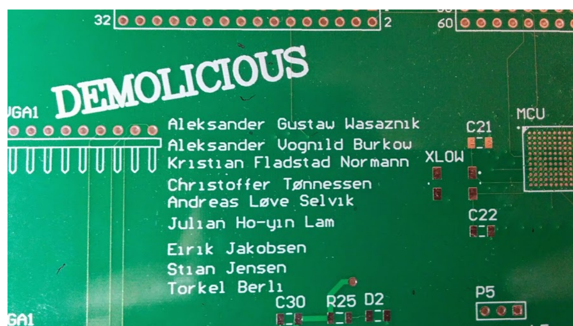
As noted, all instructions in the same warp runs the same instruction. After a conditional statements, however, different threads of the same warp will likely need to perform different instructions. This is the concept of thread divergence. NVIDIA's GPUs support dynamic thread divergence by moving threads between warps as they diverge.

3.5 Helpful Inspiration

The concepts employed in NVIDIA's Fermi architecture and CUDA programming model are, needless to say, effective for parallel calculations. They have been a source of inspiration for this project. Most these concepts, like kernels, warps, and easy context switches, will return later in the report. Although they return in simplified versions, they are helpful for exploiting parallelism.

Chapter 4

Demolicious



4.1 A Graphical Demo Machine

The system created in this project is named Demolicious. It is made for running graphical demo's, which is the inspiration for its name. A graphical demo is a visually pleasing programming feat made to demonstrate the capabilities of the computer as well as the programmer.

Demolicious is inspired by modern PCs' CPU-GPU coordination, both in its programming model and its architectural design. The CPU will handle programs by default, and offload parallelizable tasks to the GPU. It will run on a microcontroller. The GPU will handle parallelizable tasks, like the many graphical operations in a demo, and send graphical data to screen. Its architecture will be designed and implemented on an FPGA.

Modern GPUs have long development cycles and are very complex. Demolicious' GPU architecture is necessarily a greatly simplified version. Because of time constraints, many features that define modern GPUs had to be left out in its design. The GPU has no branching, and there are no caches. Modern GPUs have dynamic scheduling to better utilize the resources. The Demolicious GPU uses barrel processing as a static scheduling scheme, and to hide memory latency.

4.2 Demolicious Requirements

For Demolicious to support interesting demos, a set of basic requirements was listed for it. It certainly needed to display graphics, and a goal was to use HDMI for it. This decision was partially based on the excitement that it had not been done before in earlier Computer Design Projects, and partially on the fact that it is a modern technology. Driving video output from the custom GPU was important because a key point of the design was for it to mirror a modern GPU's purpose in a computer system.

Energy efficiency needed to be a primary concern throughout this project. In an experimental system like Demolicious, that concern will often be sidelined to making anything work in the first place. Energy efficiency is, after all, an optimization. Design philosophies like simplicity in GPU architecture and redundancy in PCB options collide somewhat with energy concerns, and both were essential in producing a working system in the short timespan of this project. One very significant energy optimization for Demolicious made the list of functional goals: setting the CPU to sleep mode when it is idle.

The performance goals were set to be challenging but still reasonable, based on a very rough estimate of potential memory bandwidth and FPGA clock frequency. Writing interesting demos for the completed machine would be a bonus. A related bonus would be to write helpful tools to assist in writing the demos. Table 4.2.1 below lists the initial functional goals, together with priorities, that were decided upon for Demolicious.

Demolicious Functional Goals	**Priority**
Demolicious should be general purpose	HIGH
Demolicious should display graphics on screen	HIGH
Demolicious should let its CPU sleep while the GPU executes	MEDIUM
Demolicious should use HDMI for its graphics output	MEDIUM
Demolicious should drive video output from its GPU	MEDIUM
Demolicious should display a frame of 512 by 256 pixels	MEDIUM
Demolicious should handle an output rate of about 30 frames per second	MEDIUM
Demolicious should have an example application in the form of a visual demo displayed on screen	LOW
Demolicious should have a toolchain to make life easier for programmers	LOW

Table 4.2.1: Goals set for the Demolicious system

Part II

Solution

Chapter 5

System Overview

5.1 Logical System Structure

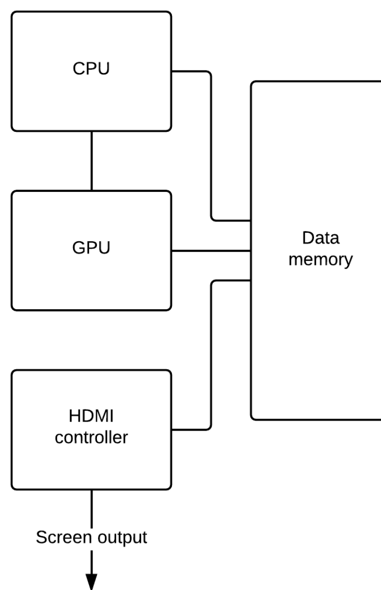


Figure 5.1.1: Logical overview of the system.

A conceptual overview of the Demolicious computer is depicted in figure 5.1.1. On a large scale, it is made up of a CPU and a GPU. In addition, an external memory chip is used for storing framebuffers for screen output. A dedicated HDMI module reads pixel data from the framebuffer and displays it on screen.

5.2 Programming Model

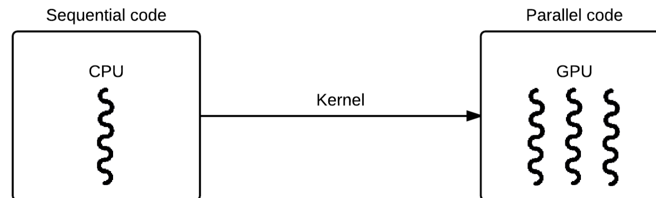


Figure 5.2.1: Relationship between CPU and GPU code.

The programming model for Demolicious is heavily inspired by CUDA.

So let's see how we can implement the same program filling a framebuffer with green, as we did in the CUDA introduction in section 3.2.

Below, in listing 5.1, the fillscreen kernel has been converted for the Demolicious programming paradigm. The first thing you'll notice is that it is not written in C, but assembly. The complete introduction to the Demolicious instruction set and assembly programming can be found in appendix B.

```
1  ldi $data, 0b0000011111100000
2  mv $address_lo, $id_lo
3  mv $address_hi, $id_hi
4  sw
5  thread_finished
```

Listing 5.1: A simple kernel that fills the screen with the color green

Let's walk through the kernel one line at a time.

The first line uses the `ldi` instruction, which stands for load immediate. It loads the value 0b0000011111100000, which corresponds to the color green in the Demolicious color space, into the special register **$data**.

The second and third line move the kernel's thread ID into the address registers.

Finally, the store instruction is executed, storing the value in the **$data** register to the address given by the two address registers. This means all pixels starting with address zero and up until the number of executed threads will get colored green.

The thread stops running after executing the `thread_finished` instruction.

5.2.1 Executing Kernels

Now, how do we execute this kernel on the GPU? The program running on the CPU, referred to as the *host program*, has to upload the assembled kernel to

the GPU, and then tell the GPU to run it. Kernels can be assembled using the provided assembler (section 9.1).

```
instruction_t fill_screen_kernel[] = {
    0x08050000, // ldc $data, 0
    0x00402004, // mv $address_lo, $id_lo
    0x00201804, // mv $address_hi, $id_hi
    0x10000000, // sw
    0x40000000 // thread_finished
};

kernel_t fill_screen = load_kernel(fill_screen_kernel);

run_kernel(fill_screen, 4096);
```

Listing 5.2: Loading and executing a kernel

In listing 5.2, we first see the assembled kernel stored in the `fill_screen_kernel` array. It is then uploaded to the GPU using the `load_kernel` function. A reference to the uploaded kernel is returned, which will be used when starting the kernel. In addition, the `run_kernel` function is also provided with the number of threads to spawn. Here, we spawn 4096 threads, enough to color 64*64 pixels green.

While being able to make the screen green by running a specialized kernel is nice, it would require many similar kernels to color the screen in different colors. To improve on this, kernels can take parameters as input, which lets them be reused with varying output. The CPU can set these parameters to different values each time a kernel is run. For instance, the CPU can set the desired color as a parameter, and the kernel will store that value to memory instead of a predefined immediate. Listing 5.3 shows an example kernel where the color is stored as a parameter.

```
ldc $data, 0
mv $address_lo, $id_lo
mv $address_hi, $id_hi
sw
thread_finished
```

Listing 5.3: A kernel loading the color value from a parameter

The only changed line in this kernel is the first one, where instead of using an immediate value, we load a value using the `ldc` (load constant) instruction. The value is a constant from the kernel's viewpoint, as it cannot be changed from the GPU. Instead, the value is set from the CPU using the `load_constant` function, as seen in listing 5.4.

```
load_constant(0, 0x001F);
run_kernel(fill_screen, 4096);
```

Listing 5.4: Now drawing a blue screen using parameters

5.2.2 More Advanced Kernels

The instruction set available to kernels is fairly limited. Most notably, the control flow in kernels is linear, meaning they cannot do any branches or jumps. See section for more detail.

Although the kernels don't support diverging control flow, conditional execution is accomplished through predicated instructions.

Each of the instructions in the instruction set (except for loads and stores) can be executed conditionally by prefixing them with '**?**'. Whether a conditional instruction is executed is controlled by a dedicated mask register. The programmer may use arithmetic and logic operations to manipulate this register (such as the `srl` instruction on line 3 in listing 5.5). The result of a predicated instruction will be discarded if the mask register is **1**.

```
1  ldc $10, 0 ; Load color one
2  ldc $11, 1 ; Load color two
3  srl $mask, $id_lo, 6 ; Shift to the right converts ID to y pos
4  mv $data, $10
5  ? mv $data, $11 ; Will only be executed every other row
6  mv $address_lo, $id_lo
7  mv $address_hi, $id_hi
8  sw
9  thread_finished
```

Listing 5.5: Conditional execution using predicated instructions

The kernel starts by loading two color parameters. It then stores a shifted thread ID into the mask register. Shifting a thread ID to the right is a trick to convert the ID to a y value, which works when the screen width is a power of two. The mask register is only 1 bit, and will just store the least significant bit written to it. This means that masking will be enabled for odd rows and disabled for even rows. Line 4 first writes a color value to the **$data** register. When masking is disabled this value will be overwritten on line 5. The kernel finishes by writing the data value to memory.

This section contains some simplifications, namely some `nop` instructions that are required under some specific circumstances have been omitted. The actual kernel code that is sent to the GPU for the kernel that fills the screen with green, can be seen in listing 5.6. The kernel from listing 5.5 however, is executed as is. The reason for these `nop` instructions will be explained in later chapters.

```
1  ldi $data, 0b0000011111100000
2  mv $address_lo, $id_lo
3  mv $address_hi, $id_hi
4  sw
5  nop
6  nop
7  nop
8  thread_finished
```

Listing 5.6: The green-screen kernel as it is actually

As illustrated by these examples, the kernels on Demolicious are more limited than CUDA kernels, but the programming models are very similar. With these examples in mind, it's time to take a look behind the scenes and see how the CPU and GPU actually run this code.

Chapter 6

CPU

Now that we have seen how to write a kernel that colors the screen with a beautiful green color, it is time to see what is actually happening on the Demolicious System during its execution.

Our journey starts in the main control unit of the Demolicious computer, namely the CPU, which is implemented on an Silicon Labs Giant Gecko Microcontroller.

In this chapter we will follow the kernel from load to execution and explain what happens behind the scenes on the CPU, which is the component on Demolicious that runs the C code seen in the previous chapter.

6.1 Functionality

The main tasks of the CPU are summarized below:

- Load kernels into instruction memory on the GPU.
- Load kernel parameters into the GPU's constant memory.
- Load data sets into external memory.
- Start the execution of a kernel.
- Read data back from external memory.

6.2 Communication with the GPU

The backbone of the interaction between the CPU and the GPU is the wide bus connecting them. Since the GPU is designed to increase the performance of the system by parallelizing operations, it's important that the bus does not introduce too much overhead. We therefore designed a parallel bus based on the EBI (External Bus Interface) specifications provided in the reference manual for the EFM32GG[8, p.175]. The bus has a 16-bit data line, and a 20-bit address line, as well as six control signals.

All the pins connected between the MCU and the FPGA follow the specifications from the EFM32GG's Reference Manual[8, p.175]. This allows us to utilize the built-in support for EBI which memory maps the whole bus on the microcontroller, and take advantage of utility functions in the *emlib* software library[10]. The mapping of addresses can be seen in figure 6.2.1.

The CPU can interface three different memories for different purposes:

1. Constant memory, for storing constants which may be read by kernels.
2. Instruction memory, for storing the kernels.
3. Data memory, for storing the framebuffer, in addition to other data for kernels.

0x80000000	0x80040000	0x80080000	0x80100000	0x84000000
Constant Memory	Instruction Memory	Kernel start area	X	External memory

Figure 6.2.1: Overview of memory mapped address spaces on the CPU.

Writing to all of these memories is done by writing to the memory addresses outlined in figure 6.2.1, in the same way as writing data locally on the microcontroller. Data can also be read back from the external GPU memory. Even the task of starting kernels is done the same way, to be able to utilize the same bus for all GPU communication.

6.3 Loading a Kernel

As seen in the Programming Model chapter, host programs can upload kernels to the GPU. They should preferably do so at the beginning of the program, to avoid delays during execution.

The CPU's Demolicious library has a built-in function for this called `load_kernel`, seen in listing 6.1. It takes an array of assembled instructions as parameter, allocates memory on the GPU and uploads the kernel, before it returns a reference to GPU memory pointing to the start of the kernel. This reference is used when starting kernels.

```
1 kernel_t fill_screen = load_kernel(fill_screen_kernel);
```

Listing 6.1: A load_kernel function call with the fillscreen kernel

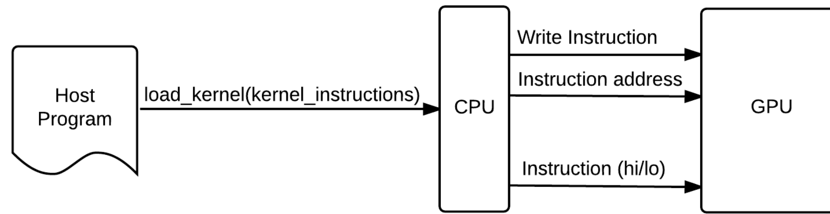


Figure 6.3.1

Seeing as instructions are 32-bit large, and the data bus is only 16 bits, every instruction load must be divided in two writes.

6.4 Running a Kernel

To run a kernel, the host program executes the `run_kernel` function with a reference to the kernel address, as well as the number of threads it wants to spawn. Listing 6.2 shows the code required for running a kernel.

```
1 run_kernel(fill_screen, 4096);
```

Listing 6.2: Running a kernel

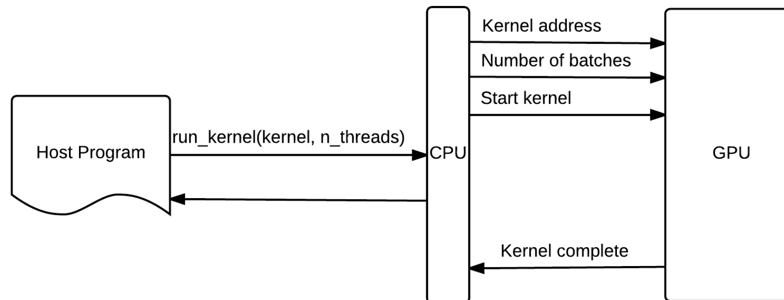


Figure 6.4.1: Starting a kernel on the GPU.

To tell the GPU to start executing a kernel, there are two pieces of information the GPU needs. Which kernel should be started, and how many threads should be executed. As can be seen in figure 6.2.1, there exists a dedicated address space for starting kernels. It is the same size as the instruction memory, meaning any valid instruction address will be valid also in the *start kernel* space. Writing to the instruction address, but in the *start kernel* space instead of the instruction memory will tell the GPU to start executing from that particular instruction.

When writing to this *start kernel* address, we use the data value for telling the GPU how many threads should be spawned. For example, say the `fill_kernel` reference points to instruction 100 in instruction memory. Then, executing the code in listing 6.2 will write the value 4096 to address 100 in the *start kernel* area.

Actually, writing the number of threads to spawn will limit the maximum number of threads to 2^{16}, as the data bus is 16 bits wide. This will not suffice for our target resolution of 512*256, when running one thread per pixel, as that requires 2^{17} number of threads. Instead, the number of *batches* of threads is written to the GPU. A batch represents the number of threads which are active in the GPU at the same time. The number of threads in a batch will typically be between 8 and 32 depending on the size of the GPU. What's important is that this work-around will help in increasing the number of threads that can be spawned from 2^{16} up to 2^{21}.

Only one kernel can execute at the time in the GPU. Because of this, the `run_kernel` call is designed to block until the kernel has completed execution. This is implemented by having a dedicated signal from the GPU to the CPU which will be asserted whenever the GPU is idle. The CPU can sleep when the GPU is executing and wait for an interrupt on this signal, saving energy while waiting for the GPU.

6.5 Loading parameters

The last important feature of the CPU library is to provide support for setting kernel parameters. As seen in listing 5.4, this is implemented in the `load_constant` function. The call takes an address and a value as parameters, as can be seen in listing 6.3

```
1 load_constant(0, 0x07E0);
```

Listing 6.3: Setting a kernel parameter

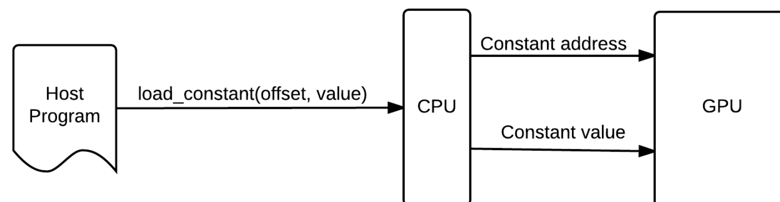


Figure 6.5.1: CPU-GPU interaction when loading a parameter.

The `load_constant` function is simply implemented by writing the constant value to the requested address, as seen in figure 6.5.1.

6.5.1 Summary

We have now seen how our kernel is bootstrapped and executed. The assembled kernel is uploaded to the GPU which is then given a command to run said kernel with one thread per pixel. Kernels can also read parameters which the host program can easily vary between kernel runs.

Chapter 7

GPU

We know how to write a kernel, and how to start it. Let's deep dive into the heart of Demolicious; the GPU! In this chapter we follow our kernel all the way from the GPU gets the commands from the CPU, through the execution of all the threads, into memory and finally to the screen over HDMI.

The Demolicious GPU is implemented on a Spartan-6 FPGA, a programmable hardware chip. The architecture has been designed, sketches drawn, and lastly implemented with VHDL; a hardware definition language.

7.1 Responsibilities

The GPU has the following responsibilities:

1. Receive instructions and constants from the CPU
2. Handle kernel invocations from the CPU
3. Write results to external SRAM
4. Assert the 'computation finished' signal to the CPU

7.2 Architecture Overview

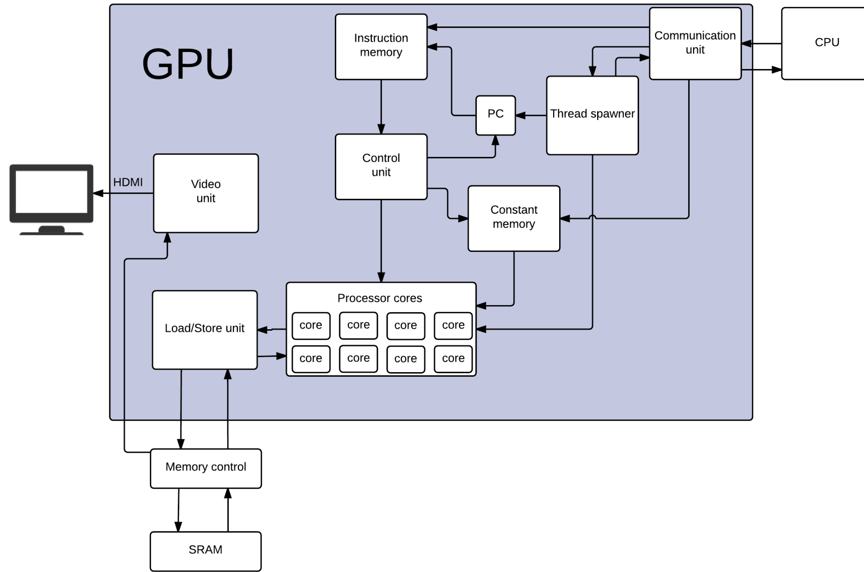


Figure 7.2.1: A high level overview of the GPU.

Figure 7.2.1 presents a high level overview over the GPU. The CPU issues commands to the communication unit in the GPU. Instructions are fetched from the instruction memory and decoded by the control unit, which has the responsibility of setting the control signals for the instructions. Control signals go to all cores of the GPU. The processor cores access memory through the load/store unit, and get constants from the constant memory. The video unit reads pixels from the data memory and outputs them to a screen over HDMI.

7.3 Receiving a Kernel Call

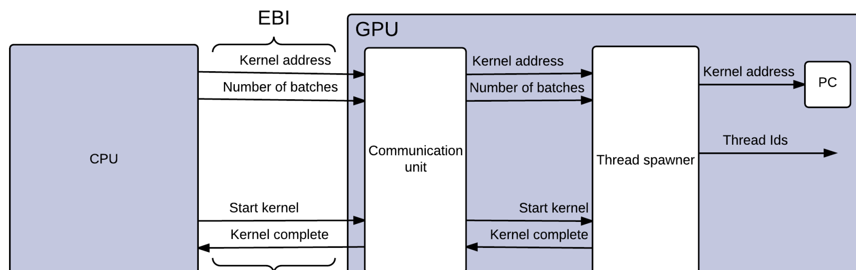


Figure 7.3.1: Launching a kernel from the GPU's viewpoint.

The communication unit is responsible for receiving kernel call requests from the CPU. When a kernel call is received, the kernel launch signal is asserted. A kernel call consists of the address of the kernel, and the number of threads to launch.

The kernel launch signals are forwarded to the thread spawner, which writes the kernel start address to the PC register, and starts distributing thread IDs to the processor cores. After holding the kernel launch signals high, the communication unit has completed its role in launching the kernel. When the kernel completes executing, the thread spawner asserts the kernel done signal, and the communication unit forwards the signal to the CPU, indicating that the kernel call has completed.

7.4 Running a Kernel

7.4.1 Warps

As in NVIDIA's Fermi architecture (see chapter 3), Demolicious organizes threads into *warps*. In Demolicious, the warp size is the same as the number of processor cores. For versions with fewer cores, the warp size changes accordingly. Just as the warps in Fermi, every thread in these warps always execute the same instruction at the same time, with one thread in each core. Note that while Fermi has multiple *Streaming Multiprocessors* which can each execute warps concurrently, Demolicious can only execute a single warp at a time. The Demolicious system is analogous to having a single Fermi *Streaming Multiprocessor*.

An important difference between Demolicious and Fermi, is that Demolicious kernels have no jumps, so its threads never diverge. This significantly reduces the complexity of scheduling these warps, and let's us get away with a single program counter.

7.4.2 Static Scheduling

As every thread in a warp always executes the same instruction, our system needs to handle 8 memory requests being issued during the same cycle. The memory system for Demolicious consists of two SRAM chips, and can return two words every cycle. This bandwidth is not sufficient to satisfy the set of 8 memory requests produced by a warp without stalling. Occasionally, these stalls can be hidden by the programmer if loads are issued ahead of their usage. However, when the program does not have enough useful instructions to do while waiting for loads, the GPU must stall. Stalling a throughput-oriented machine is obviously unfortunate.

But there are many warps waiting for execution at (almost) any time. If these warps could execute while other warps are waiting for memory, we can utilize the system better. However, changing between warps that execute requires a context switch. That is, the old thread put on hold has to store all its register values somewhere so that they can be available when it starts executing again.

Instruction A:	ldc $data, 0
Instruction B:	mv $address_lo, $id_lo
Instruction C:	mv $address_hi, $id_hi
Instruction D:	sw
Instruction E:	thread_finished

A^{0}	B^{4}	C^{8}	D^{12}	E^{16}	A^{20}	B^{24}	C^{28}	D^{32}	E^{35}
A^{1}	B^{5}	C^{9}	D^{13}	E^{17}	A^{21}	B^{25}	C^{29}	D^{32}	E^{36}
A^{2}	B^{6}	C^{10}	D^{14}	E^{18}	A^{22}	B^{26}	C^{30}	D^{33}	E^{37}
A^{3}	B^{7}	C^{11}	D^{15}	E^{19}	A^{23}	B^{27}	C^{31}	D^{34}	E^{38}

Figure 7.4.1: Execution timing of warps without jagged scheduling

In software multi-threading, a context switch is typically achieved by storing all registers to memory and loading in the registers for the new thread that is scheduled. This is an expensive operation, and would introduce more latency than we are trying to hide. Demolicious, however, has a set of active warps that all have their own registers. So a context switch can be carried out with virtually no overhead, at the expense of some additional hardware to store all these extra registers.

To keep the architecture as simple as possible, Demolicious employs a simple static scheduling algorithm. The active warps are simply rotated after every instruction. In figure 7.4.1 we revisit our green-screen kernel, and see how the warps are scheduled. The numbers in the top right corner is the GPU cycle, and we refer to the rows as barrel lines. During cycle 0, instruction A is executed on warp 0. The next cycle, instruction A is executed on warp 1. And when we get to cycle 4, warp 0 is scheduled again, this time executing instruction B. This continues until we reach clock cycle 16-19 where warps 0 through 3 executes instruction E, the *thread_finished* instruction, which does no computation, but allows the GPU to set up new warps. On cycle 20, warp 4 is set up and ready to execute the first instruction. This pattern continues until all threads have run to completion.

However, there is an issue with this simple scheduling algorithm. Let's see what happens when we get to instruction D, marked in red. In clock cycle 12[1] all threads in warp 0 execute a memory request, resulting in 8 simultaneous memory requests. As our memory system handles only two requests per clock, it is busy the next 4 cycles with these requests. When warp 0 gets to execute its next instruction, the memory operation is complete for all the threads in the warp. Had this been a load instead of a store, the loaded value could already be used its instruction, and we have avoided a stall. So far, so good. However, when warp 1 executes the store instruction in cycle 13, the memory is already busy with the request from warp 0. The memory requests from warp 1 will not be started before cycle 16, and will therefore not be complete for its next instruction. It is even worse for warp 2 and 3. As all warps execute the same code, this can introduce up to 8 *nop* instructions for every warp.

[1]The cycle numbers are written in the upper right corner of each square

Instruction A:	ldc $data, 0
Instruction B:	mv $address_lo, $id_lo
Instruction C:	mv $address_hi, $id_hi
Instruction D:	sw
Instruction E:	thread_finished

A^{0}	B^{4}	C^{8}	D^{12}	E^{16}	A^{20}	B^{24}	C^{28}	D^{32}	E^{35}	A^{39}	B^{45}	C^{49}
nop^{1}	A^{5}	B^{9}	C^{13}	D^{17}	E^{21}	A^{25}	B^{29}	C^{32}	D^{36}	E^{40}	A^{46}	B^{50}
nop^{2}	nop^{6}	A^{10}	B^{14}	C^{18}	D^{22}	E^{26}	A^{30}	B^{33}	C^{37}	D^{41}	E^{47}	A^{51}
nop^{3}	nop^{7}	nop^{11}	A^{15}	B^{19}	C^{23}	D^{27}	E^{31}	A^{34}	B^{38}	C^{42}	D^{48}	E^{52}

Figure 7.4.2: Execution timing of warps with jagged scheduling

Clearly, the introduction of this barrel processing technique did not solve our problems. Fortunately, this was a simplified version of the scheduling in Demolicious. The actual scheduling is shown in figure 7.4.2. By introducing an offset in the instructions executed, henceforth called jagged execution[2], we can avoid this issue.

With this scheduling, you can see that there are 4 cycles without a memory operation between every memory operation. This is true as long as memory operations have 4 non-memory operations between them in the kernel. It is up to the programmer or assembler to ensure that this holds.

This warp scheduling algorithm will guarantee that nops are not required after loads, as long as they are spaced far enough apart.

[2] Or "jaktstart" as we like to call it in Norwegian

7.5 Module Details

7.5.1 Processor Core

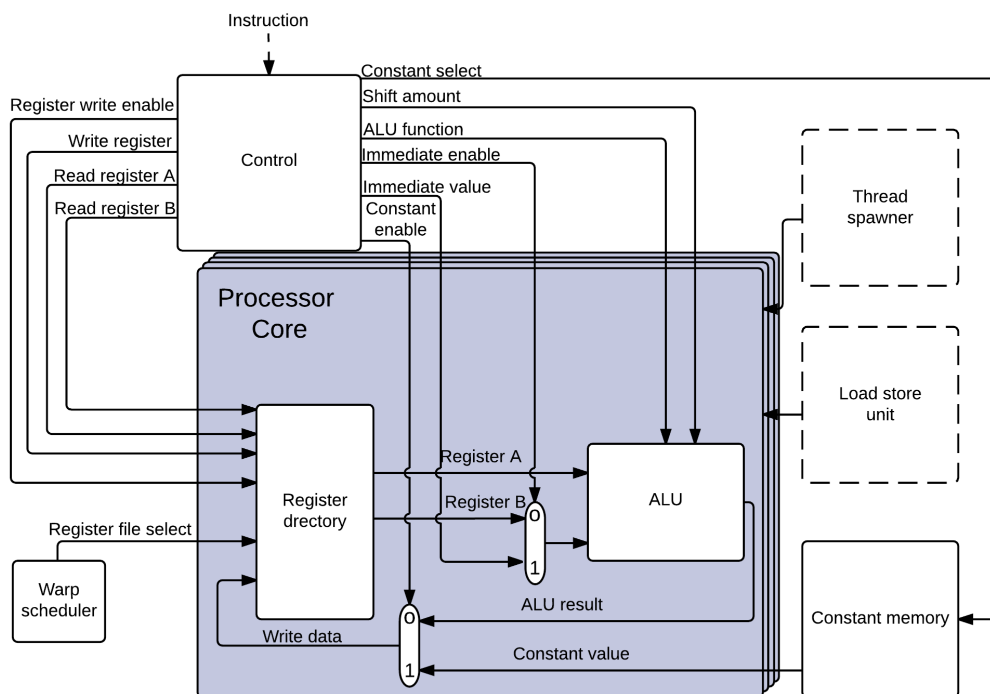


Figure 7.5.1: A single processor core.

Each processor core contains an ALU and a register directory. On each clock cycle the ALU receives the selected values from the register directory. On the same cycle, the value is written back to the register selected by the control unit.

If the current instruction is a *load constant* instruction, the control unit selects which constant to fetch from constant memory. The constant enable signal is asserted, causing the multiplexer to select the constant value as the write back value.

To perform immediate instructions, the control unit asserts the *immediate enable* signal. The immediate value is then selected by the multiplexer and used as an operand in the ALU.

The architecture contains multiple processor cores like the one in figure 7.5.1. Each of the cores receive the same control signals for the instructions.

7.5.2 Thread Spawner

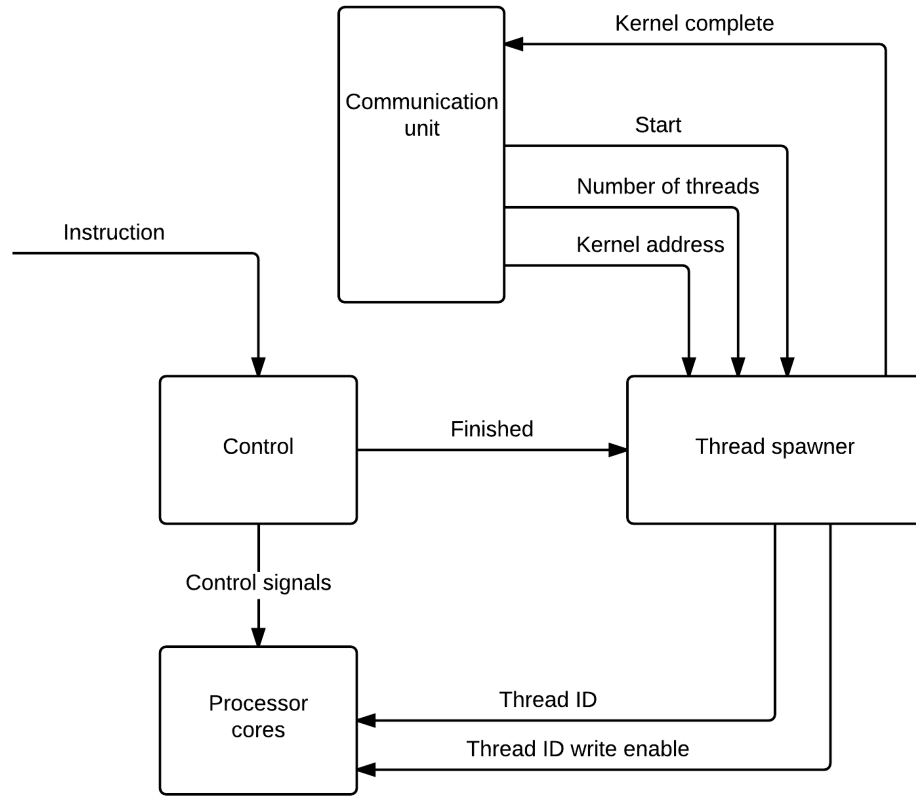


Figure 7.5.2: Thread spawner and neighboring components.

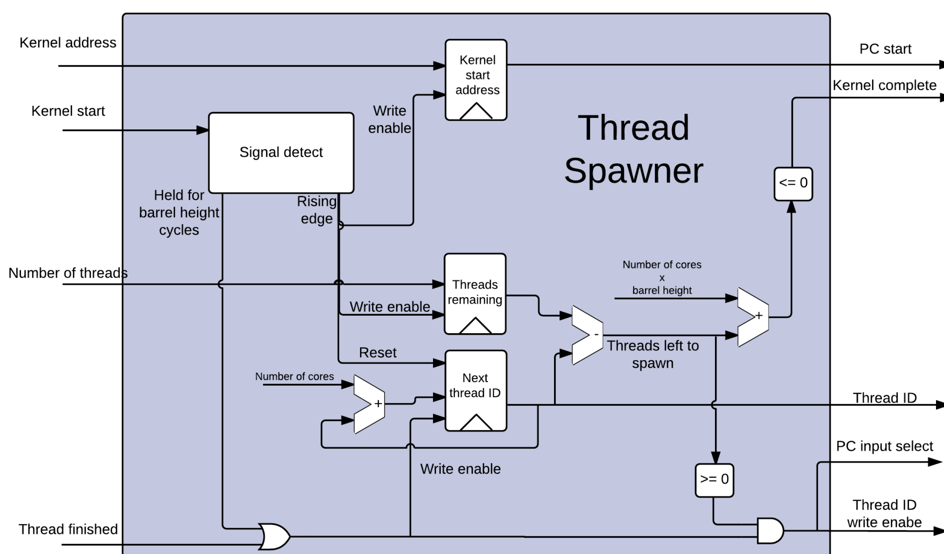


Figure 7.5.3: Thread spawner RTL implementation.

The thread spawner is responsible for overseeing kernel execution. It will spawn threads whenever necessary, handling thread setup and ensuring that the requested kernel is executed. When all threads have finished executing, it will assert the kernel done signal, notifying the host program that computation has finished.

When a kernel invocation request is received, the thread spawner stores the provided base address of the kernel, the number of threads to spawn, and sets the next thread ID register to zero. Threads are spawned one warp at a time into the currently active barrel line. Therefore, on kernel start the thread spawner will spawn threads until the warp scheduler has made an entire rotation, or a barrel roll as we call it.

When a thread finishes execution, the control unit will assert the *finished* signal to the thread spawner. If there are threads left to spawn, the currently active barrel line will be filled with a new warp of threads.

But what does the spawning of a warp of threads actually entail? All threads need a unique thread ID. If the value of the next thread ID register is 4, and we have 4 processor cores, core zero will write 4 to its thread ID register, core one 5, and so forth. The next thread ID register is then incremented by the warp size, increasing it from 4 to 8.

7.5.3 Register Directory

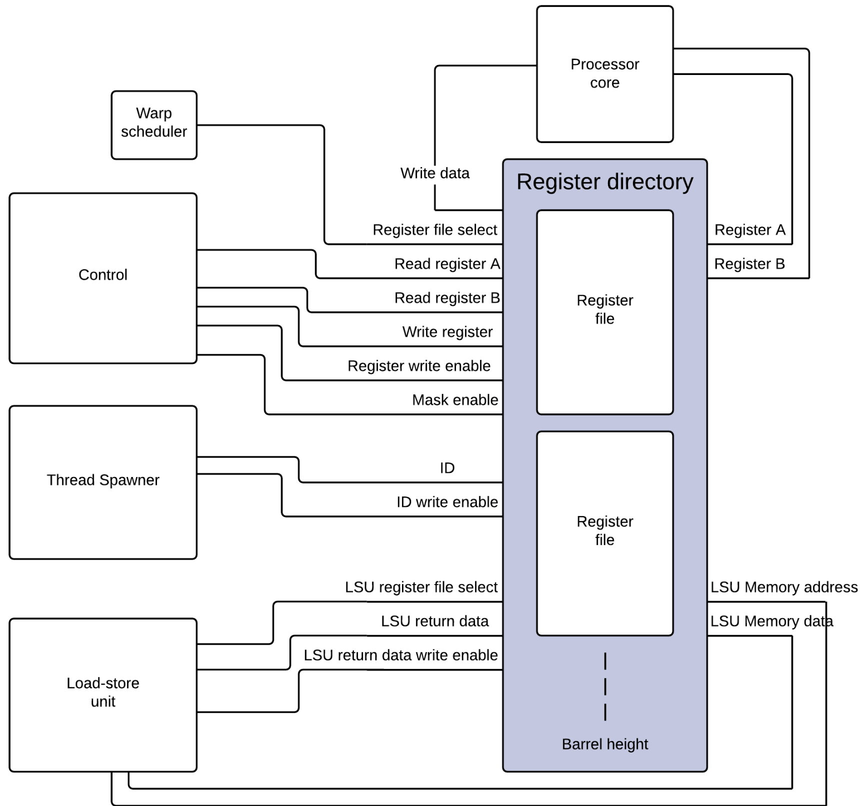


Figure 7.5.4: The register directory, and its neighboring components.

There is one register directory per processor core. Each register directory contains one register file per barrel line. The register files include seven dedicated registers, and nine general purpose registers, listed in table 7.5.1.

Register number	Description	RW
$0	Zero register	Read-only
$1	ID High	Read/Write
$2	ID Low	Read/Write
$3	Address High	Read/Write
$4	Address Low	Read/Write
$5	LSU data	Read/Write
$6	Masking register	Write
$7 - $15	General purpose registers	Read/Write

Table 7.5.1: The registers contained within the register files.

Individual register files are selected by using the *register file select* signal. Using this signal, the warp scheduler selects the active register file. Within the register directory, signals are routed to the active register file using the select signal. Consequently, from the processor core's point of view, there's just one register file.

In the architecture the dedicated registers have special functions. Ignoring registers $0 and $6, the dedicated registers may be used as general purpose registers.

The masking register is used to enable conditional execution, as introduced in section 5.2.2. When executing predicated instructions, the masking register is used to disable writes to the register file. This makes the instruction have no effect. The physical implementation is seen in figure 7.5.5. To reduce the complexity of the architecture, *store word* instructions cannot be predicated. Instead, the programmer has to manage the values written explicitly.

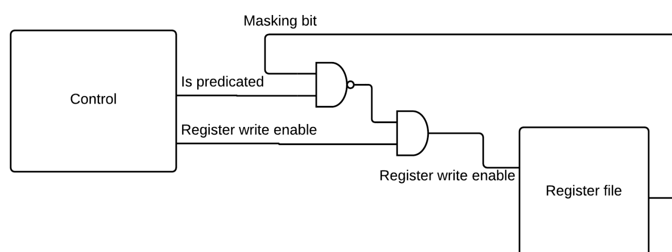


Figure 7.5.5: Using the mask bit to disable register writes.

Data memory addresses are 20 bit wide, and thread IDs are 19 bit wide. The word size is only 16 bits. To represent these values, they are split into two registers.

The load store unit (LSU) has a separate signal for selecting which register file

to write to. Using this signal, the LSU can write data loaded from memory into any register files on its own.

7.5.4 SRAM Arbiter

The SRAM arbiter connects the LSU, HDMI unit and CPU to the SRAM. As multiple units may request access to SRAM at the same time, and the Demolicious system is using an interlaced memory model, dedicated access to SRAM cannot be guaranteed. Therefore, a prioritization scheme is required.

The CPU has the highest read-write priority. As the host program by convention shouldn't access memory when kernels are executing, this is fine. The LSU comes in second, as it needs uninterrupted access to SRAM during execution. If the HDMI unit could prevent the LSU from accessing memory, the result of a kernel executions could differ based when the HDMI unit chooses to read the current framebuffer. Therefore the HDMI unit has the lowest priority, having to scavenge pixels whenever it can.

7.5.5 Load/Store Unit

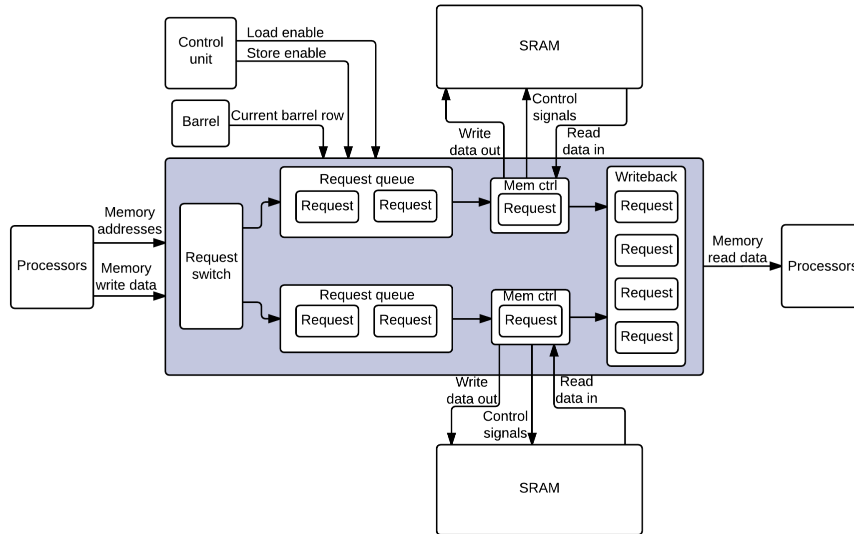


Figure 7.5.6: RTL of the load/store unit. Notice the dual queueing system to handle the interlaced SRAM.

A load/store unit, shown in figure 7.5.6, is responsible for handling memory requests from the core processors. The load/store unit of Demolicious needs to be capable of simultaneously servicing incoming requests from all the processor cores. Requests are asynchronously carried out in the background, and replies

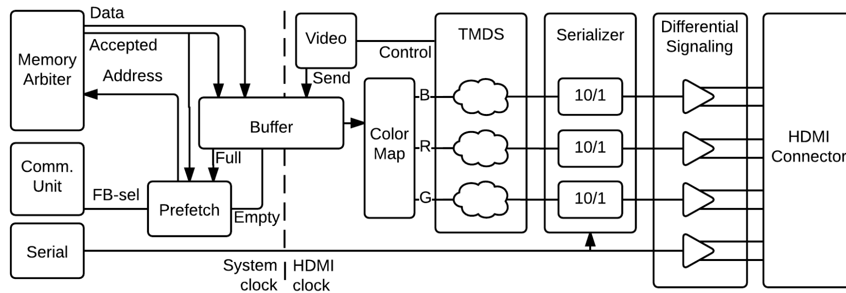


Figure 7.5.7: The logical overview of the video unit. A prefetching unit tries to fill the buffer when it is not full. The *Accepted* signal signifies a successful memory request. Across the clock domain boundary, the video timing unit generates control signals. When pixel data should be transferred, the *Send* signal moves pixels from the buffer to the transmission units. The 16-bit pixel data is converted to three 8-bit colors, TMDS encoded, serialized and output using differential signaling. Should the buffer go empty, the prefetch unit needs to immediately advance to the next pixel to stay synchronized with the timing unit.

are delivered directly into the appropriate registers of the threads that made the request.

Demolicious has two independent memory banks, each capable of reading or writing one word every cycle. To maximize the throughput to the memory, a word-striping scheme is used: The lowest bit of the memory address determines which bank holds that location.

Because there are two independent memory banks, the incoming requests are routed to two separate queues, one for each memory. Each queue then feeds requests to its associated SRAM chip. Read responses are then handed to the write-back unit, which delivers the data to the appropriate register file.

The queues are necessary because Demolicious has more processor cores than memory banks, so all the requests from a warp can not be completed in a single cycle, as discussed in section 7.4.1. The processor cores work in tandem, and issue requests simultaneously. It takes several cycles to complete the requests for a single warp. Since there is a limit to how many requests can be in the queue at any time, there is also a limit to how often requests can be issued.

7.5.6 HDMI

From a demo makers perspective, a GPU without a video output is commonly known as a space heater. Demolicious uses HDMI for its video output, making it easy to connect to any recent video display.

HDMI is a streaming protocol; the receiver reads data from the cable at a fixed rate. In Demolicious, the GPU has priority access to the memory. This

means that the video unit may not have access to the framebuffer (which lies in memory) when it's time to send a pixel. To alleviate this issue, as much of the framebuffer as possible is prefetched into a buffer whenever memory is available. Should the buffer underflow, black pixels will be sent instead. Otherwise, pixels will not be synchronized with the position they should appear at on the screen.

Control signals assert where in the data stream a new frame of video starts and ends. These allow the receiver to determine the resolution and refresh rate of the video.

The lowest resolution supported by HDMI is 640x480. As this is larger than our framebuffers (64 x 64 pixels), a letterbox is added around the picture. For debugging purposes, the letterbox consists of a low-contrast checker pattern.

To actually send the data over HDMI, control signals and pixel data are split into three channels. They are then encoded using a scheme known as TMDS. The purpose of TMDS is to minimize the effect of noise over the physical connection.

TMDS uses 10 bits to encode either an 8-bit color value when sending an image, or control values when not. Demolicious uses a 16-bit word size, so colors are represented with 5 bits for red, 6 for green and 5 for blue. These are resized to 8-bit values using a scheme that allows for both complete black and white colors. Each channel is then serialized before being output together with a clock using differential-signaling.

Finally, to avoid a visual artifact known as *screen tearing*, a technique known as *V-sync* with *double buffering* is used. These techniques ensure that only complete frames of video are output, increasing visual fidelity.

7.5.7 Summary

The journey of our kernel is complete. We have followed it all the way from the initial *load_kernel* call to the screen. It has traveled through the massively parallel GPU, which can handle a vast amount of threads, using a round-robin static scheduling technique called barrel processing.

Chapter 8

Physical Implementation

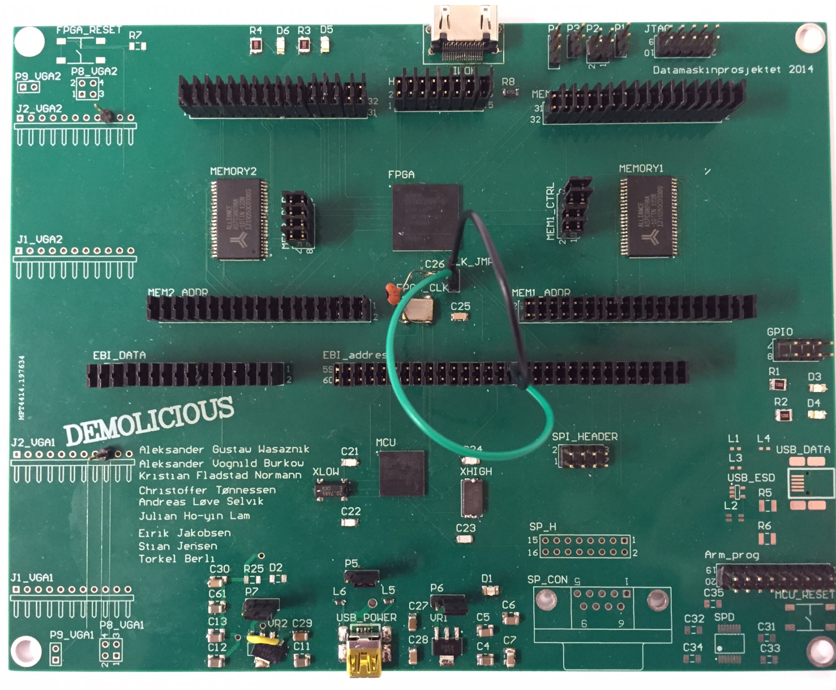


Figure 8.0.1: Completed PCB

The entire Demolicious system is implemented on a PCB (Printed Circuit Board). This chapter will give an overview of the hardware architecture, and detail design choices made during the realization of the design.

8.1 Backup oriented design

Verifying a PCB design is difficult. If the circuitry has been designed incorrectly, there may not be much that can be done, except for designing a new PCB. Because of this, the PCB has backup solutions for all major components, and exposes as much as possible of the important circuitry on headers. This enables us to probe signals for debugging purposes, and also do manual rewiring in case of faulty wiring.

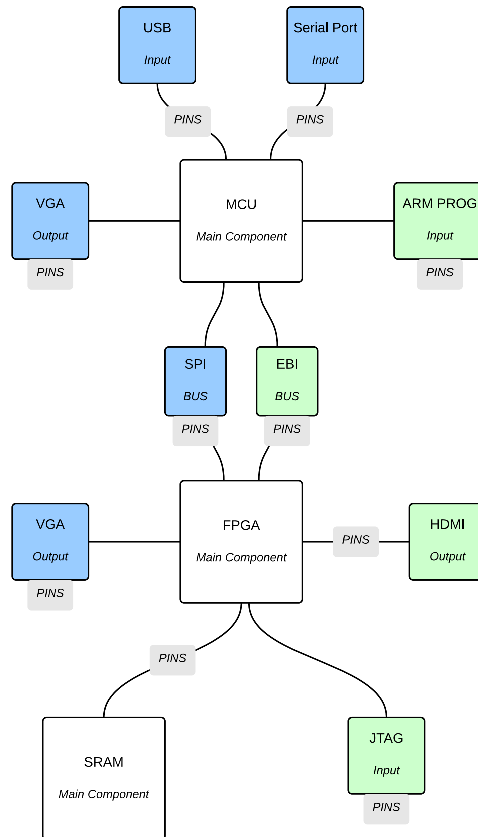


Figure 8.1.1: Conceptual overview of the PCB. Green boxes are main solutions. Blue boxes are backup plans. Gray boxes labeled "PINS" mean that these signals are exposed on headers.

These backup plans are in place to make sure the board will work, even if some parts are broken. That way, each component can be connected to other sources than those on the board alone. Because of this, the board is not optimized for the smallest size possible, but was rather made to optimize for highest possible chance of success.

8.2 Input

The main method of communication between the microcontroller and a host PC is by USB. The USB circuitry is designed with ESD protection in an effort to lessen the risk of frying the PCB. Should the USB fail, a serial port (RS-232) has been implemented as a backup solution. If this also fails, the wires from the serial port is put on headers, which can be used as GPIO pins. If all else fails, the program to be run on the machine can be included when programming the MCU through the programming interface.

8.3 Output

The main source of output from the PCB is an HDMI connector. This is a novel feature this year, as no previous group has tried to implement it before.

Because of this new challenge, a lot of backup schemes were put in place. The HDMI connector is put on headers, in case the connector fails. A VGA module is connected to the FPGA, in case the HDMI does not work. And if this fails, a separate VGA module is connected to the microcontroller.

8.3.1 HDMI implementation

The general HDMI specification consists of the Transition Minimized Differential Signaling (TMDS), and some additional wires for details regarding the transmitted signal [4]. However, we were able to produce a video feed on a screen from an FPGA using only the 8 TMDS wires by cutting up an HDMI cable and using only those wires. Upon seeing that this was feasible, we decided to make the HDMI hardware with a TMDS connection only, going in to the FPGA. The resulting hardware was then an HDMI type-A receptacle footprint, the HDMI receptacle and a header between it and the FPGA. The reason for adding the header was that this setup is equivalent to the aforementioned prototype version. If the HDMI output wouldn't work we could connect the terminated HDMI cable onto the header.

8.4 Power

The system is powered by a 5V mini USB. This connection powers two voltage regulator circuits. Respectively, the 1.2V wire that powers the FPGA (excluding the I/O banks) and the 3.3V power plane that powers up the rest of the machine. The 5 volt USB connection has a backup solution in the form of a header through which it passes immediately after entering the PCB. This is done so that the incoming electricity can easily be probed for voltage level and also so that in case the USB connection fails, an external power supply can be connected to the board.

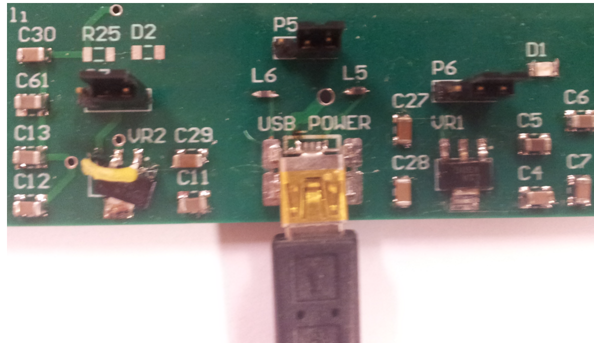


Figure 8.4.1: Picture of the physical power circuit, where P5 is the header on which an external power source can be connected

8.5 Bus

The bus between the MCU and FPGA is an EBI connection. The EFM32GG990 has dedicated pins for the EBI, which can be found in the datasheet [9] to the MCU. Headers are placed in between the MCU and FPGA in case of failure on either side of the connection, as well as an easy way to check transmitted signals during debugging.

8.6 Clocks

Both the MCU and the FPGA need an external clock. According to the design consideration AN0002 [7] the MCU needs two crystals, a low frequency crystal at 32.768kHz and a high frequency crystal at 48MHz. For the FPGA, an oscillator of 120MHz was chosen.

Headers are placed between the oscillator and the FPGA. The headers serve as a backup in case an external oscillator is needed and act as a debugging tool. The microcontroller clocks on the other hand have a backup solution in that the microcontroller has internal RC-oscillators for use in case the crystals malfunction.

8.7 Main Components

MCU (EFM32GG990F512-BGA112)

The EFM32 Giant Gecko Microcontroller from former Energy Micro, now Silicon Labs, is chosen as the microcontroller for this project. One aspect of the task at hand is energy efficiency and this microcontroller is particularly efficient in that manner. This is a proven microcontroller with a lot of development boards available to us, so it seemed like a safe choice.

FPGA (XC6SLX45-2CSG324I)

The FPGA of the Spartan-6 family from Xilinx was chosen as the FPGA. This particular FPGA has been used for different tasks on the university before, and the support systems are therefore available to us. A less powerful version of this one was available for testing on development boards in the lab.

SRAM (AS7C38098A)

The SRAM AS7C38098A was chosen as the memory storage for this project [6]. The reason for why this particular SRAM was chosen, was that to be able to output graphical demos, we need to be able to store framebuffers, i.e. pixel data which will be displayed on screen. We need memory large enough to hold two framebuffers and a latency small enough to be able to read and write data to the memory at an acceptable rate. This specific SRAM met the requirement with a storage space at 512K words by 16 bit and a 10ns access time.

Chapter 9

Additional Tools

9.1 Assembler

An assembler is implemented, to be able to run kernels written for Demolicious on the GPU. The assembler is written in Python. In addition to supporting assembly of all instructions supported by the GPU, it provides additional pseudo instructions, as well as aliases for special purpose registers.

The assembler is available on GitHub `https://github.com/dmpro2014/simulator/`.

9.2 Simulator

In addition to the assembler, a tool for verifying the correctness of kernels before they are run on the GPU has been developed. It is based on the same parsing backend as the assembler, and simulates the execution of kernels, instruction for instruction. When the set of threads have been simulated, a resulting framebuffer is rendered to screen. This tool has proven itself very useful when developing kernels, as it reduces the need of re-assembling and uploading kernels to the GPU to verify the correctness of the assembly code.

The simulator is part of the same project as the assembler described in section 9.1.

Part III

Results & Discussion

Chapter 10

Testing

10.1 GPU Component testing

The GPU consists of a number of fairly isolated components. It's valuable to test the components in isolation before they are connected. Both implementation errors, and design flaws, can be uncovered before the components are introduced to the system. The unit tests were implemented as VHDL test benches. Table 10.1.1 describes the unit tests that were run, and provides a summary of which features were tested.

COMPONENT	WHAT	STATUS
Barrel selector	Barrel row is incremented each clock cycle. Barrel row wraps around to 0. PC write enabled high on row 0.	Passed
Constant memory	Constants can be written. Constants can be read.	Passed
Inst decode/Control unit	Decodes instruction types correctly. Sets control signal values correctly.	Passed
Instruction memory	Instructions can be written. Instructions are read from the correct address.	Passed
Instruction memory	Instructions can be written. Instructions are read from the correct address.	Passed
Register file	Read/write to general purpose registers. Dedicated registers behave correctly. Does the masking bit work?	Passed
Register directory	Multiplexes input/output signals to the correct register file.	Passed
Processor core	Arithmetic operations. Can mask instructions.	Passed
ALU	Computes arithmetic operations. Can do left/right shifts. Performs *Set if less than* correctly.	Passed

Table 10.1.1: Unit tests for components in the Demolicious system.

10.2 VHDL system integration tests

Before deploying to an actual FPGA, it is important to ensure correct behavior in system-level testbenches. This testing is valuable, as if correct behavior can be verified in simulation, there are fewer potential errors when debugging the FPGA itself.

System tests for each of the major datapaths through the design have been created and run successfully. The result of each test is verified by comparing all

pixels in the framebuffer after the kernel has been run to precomputed values in ISim.

10.2.1 Memory Stores and Kernel Parameterization

To verify that stores to memory, as well as constant memory, actually works, the kernel presented in listing 5.3 is used as a system test. It has been re-listed in listing 10.1 for convenience.

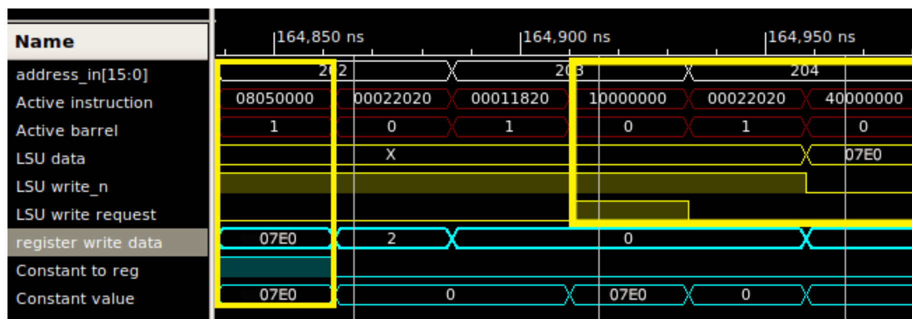
```
1 ldc $data, 0
2 mv $address_lo, $id_lo
3 mv $address_hi, $id_hi
4 sw
5 thread_finished
```

Listing 10.1: Kernel to test constant memory and parameterization

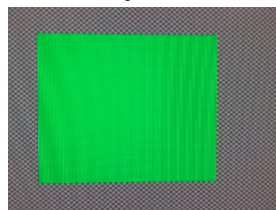
Expected behavior of test:

1. The color green should successfully be loaded from constant memory.
2. It should be stored to memory.
3. The screen should be filled with the color green.

Simulation Results



(a) Isim simulation showing constant load and store word



(b) LX16 run

Figure 10.2.1: Results from simulation and LX16 of fillscreen kernel

In the left yellow square of figure 10.2.1a, one can see the load constant instruction being executed (0x08050000) in barrel line 1. The Constant to reg signal is asserted, and the constant value 0x07E0 is passed into the register write data signal.

In the right yellow square, the store word instruction (0x100000000) executes in barrel line 0. The LSU accepts the write request same cycle (LSU write request goes high), and two cycles later the request packet reaches the LSU data line. The LSU asserts LSU write_n, (the signal is active low), and external RAM handles the store request.

The testbench passes, and values have now been successfully written to memory. It also runs on actual hardware, the result shown in figure 10.2.1b.

10.2.2 Predicated Instruction Execution

Predicated instructions are used to allow for some degree of conditional execution in the lack of proper branching and jumps. This requires that the architecture actually respects the mask bit when set. The predicated execution kernel presented earlier in listing 5.5 is used for this test. It has been re-listed in listing 10.2 for convenience.

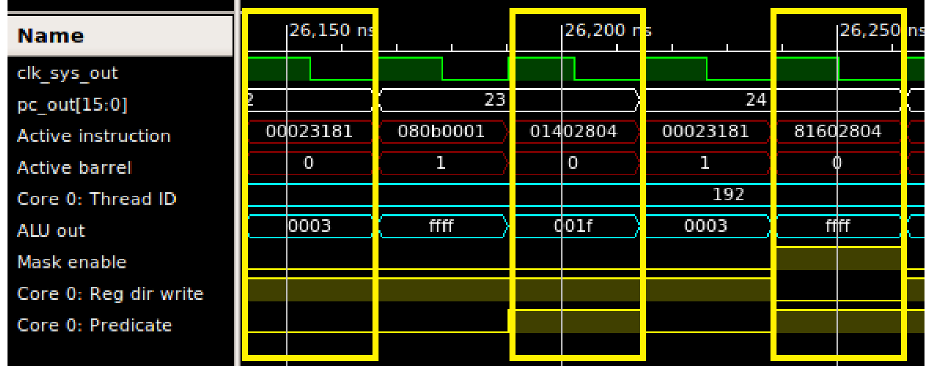
```
1 ldc $10, 0 ; Load color one
2 ldc $11, 1 ; Load color two
3 srl $mask, $id_lo, 6 ; Shift to the right converts ID to y pos
4 mv $data, $10
5 ? mv $data, $11 ; Will only be executed every other row
6 mv $address_lo, $id_lo
7 mv $address_hi, $id_hi
8 sw
9 thread_finished
```

Listing 10.2: Conditional execution using predicated instructions

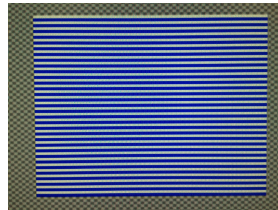
Expected behavior of test:

1. Each row should be colored according to the last bit of their y position.

Simulation Results



(a) Isim simulation showing successful masking.



(b) LX16 run

Figure 10.2.2: Results from simulation and LX16 of predicated execution

In the left yellow square of figure 10.2.2a, we can see the `srl` instruction being executed (0x00023181). As this thread has thread id 192, the result out is 3. The low bit is stored into the mask register, enabling masking for this thread.

In the middle yellow square, barrel 0 is once again active, and we can see that the predicate bit of core 0 has been asserted. As this instruction isn't masked, the predicate bit is ignored and the value of 001f is stored into the data register.

In the right yellow square, the conditional data move is executed (0x81602804). As the mask enable signal goes high, the register write enable signal is pulled low due to the predicate bit, resulting in the data not being written to registers.

The testbench passes, and predicated instructions are not executed when masking is enabled. As can be seen in figure 10.2.2b, the kernel runs on actual hardware, and every second line is colored differently.

10.2.3 Loads from Primary Memory

If one wishes to support multi-pass kernels, that is a kernel that uses the result of previous kernels to compute its own result, then results have to be stored and loaded from somewhere persistent. Stores to main memory have already been confirmed working, so load instructions are up next for testing. With functioning loads, it is fairly straightforward to implement things like Conway's Game of Life [1] for the Demolicious system.

The fillscreen kernel from earlier is modified to load its value from main memory, and store it back to the framebuffer. It is presented in figure 10.3.

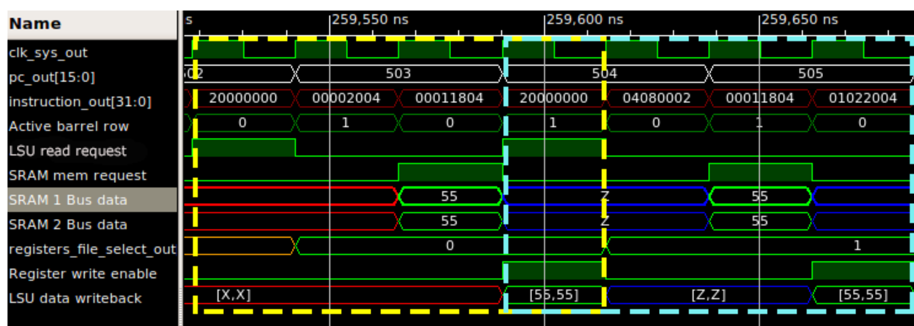
```
1  mv $address_hi, $0          ; Load some color from main memory
2  mv $address_lo, $0
3  lw
4  mv $address_hi, $id_hi
5  ldi $8, 2
6  add $address_lo, $8, $id_lo
7  sw                          ; Store data to ID + 2, to avoid overriding
       address 0
8  nop
9  thread_finished
```

Listing 10.3: Kernel to test loads from main memory

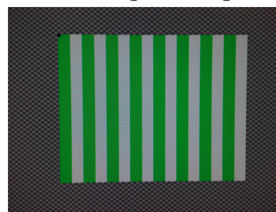
Expected behavior of test:

1. The loaded color should be stored to the entire framebuffer

Simulation Results



(a) Isim simulation showing working loads from memory



(b) LX16 run

Figure 10.2.3: Results from simulation and LX16 of load kernel

Figure 10.2.3a shows a barrel of height 2 performing successful loads.

In the first cycle of the yellow dotted square, barrel row 0 executes a load word instruction (0x20000000). LSU read request is asserted in the control, and routed to the LSU. Two cycles later the request has passed through the LSU unit, and SRAM responds with the value 55. In the last cycle of the yellow

square, the LSU asynchronously writes the result back to the registers storing the values on the LSU data write-back line to barrel row 0.

In the blue dotted square, the same procedure is executed again, this time for the second barrel row. Notice that register_file_select is now set to 1, writing the result back to the second barrel row.

The testbench passes. Multi-pass kernels can now successfully be implemented.

There is however a discrepancy between the simulation and the actual implementation, resulting in the LSU dropping some write-backs, as shown in figure 10.2.3b. Both the RAM used for simulation, as well as the SRAM located on the PCB is combinatoric. On the development kit however, SRAM had to be mapped onto block RAM due to a lack of space. This forces the RAM to be clock driven, not allowing for the same-cycle memory responses required, and therefore dropping write-backs.

10.3 Verification of Communication Channels

10.3.1 JTAG test

In these tests we attempted to connect to respectively, the FPGA and the MCU by way of the appropriate headers, thereby verifying that they had been properly soldered in place and were functional.

WHAT	HOW	STATUS
Verify FPGA JTAG's ability to flash FPGA	Flash code on FPGA that makes a LED blink	Passed
Verify ARM debug interface's ability to flash MCU	Flash code on MCU to make a LED blink	Passed

Table 10.3.1: JTAG tests

10.3.2 EBI Bus

WHAT	HOW	STATUS
EBI bus configuration on MCU	Configure MCU and its GPIO pins and write to memory mapped addresses for EBI. Inspect signals with a logic analyzer.	Passed
EBI receiver module on FPGA	Connect MCU and FPGA with EBI, and use MCU to write to and read from FPGA block RAM.	Passed

Table 10.3.2: EBI bus tests

10.3.3 HDMI Output

WHAT	HOW	STATUS
Writing to screen from FPGA using HDMI	Connect the wires of a DVI cable to GPIO pins on FPGA. Write color values using HDMI standard.	Passed

Table 10.3.3: HDMI tests

10.3.4 SRAM Communication

WHAT	HOW	STATUS
Write to and read from SRAM over EBI	Connect the working EBI protocol of MCU to an SRAM chip. Write words to SRAM and read them back.	Passed
FPGA to SRAM communication over EBI	Connect FPGA GPIO pins to SRAM. Use FPGA to write words to SRAM and read them back	Unfinished

Table 10.3.4: SRAM tests

10.4 PCB tests

After receiving the finished PCB and the components, a series of tests were performed to make sure all parts of assembling the finished computer were done correctly.

10.4.1 Solder, signal and power test

The purpose of these tests was to solder components and check that their connections held, as well as check that power correctly propagated through the board.

WHAT	HOW	STATUS
5 V power from USB connector	Solder components, measure voltage on headers on 5 V power line, verify that it is 5 V	Passed
3.3 V power from regulator and VDD headers	Solder components, measure voltage on 3.3 V power plane, verify that it is 3.3 V	Passed
1.2 V power from regulator and power indicator LED	Solder components, measure voltage from 1.2 V regulator and power indicator LED, verify that it is 1.2 V	Passed
MCU and FPGA soldering	Place MCU and FPGA on PCB and bake them in oven, make sure baking looks good and balls have melted into sockets	Passed
MCU and FPGA connected correctly	Solder helping headers, connect FPGA and MCU to power, verify correct soldering by flashing MCU and FGPA	Passed

Table 10.4.1: Solder plan and verification

10.4.2 Oscillator and clock test

These tests consisted of using an oscilloscope in order to probe the outputs of the FPGA's oscillator and the high frequency and low frequency crystals of the MCU.

WHAT	HOW	STATUS
FPGA oscillator wave frequency at 120 MHz	Use oscilloscope on out pin on the FPGA oscillator	Passed
MCU low frequency crystal clock at 32.768 kHz	Use oscilloscope to measure low frequency crystal	Incomprehensible results
MCU high frequency crystal clock at 48 MHz	Use oscilloscope to measure high frequency crystal	Incomprehensible results

Table 10.4.2: Frequency output tests

Chapter 11

Results

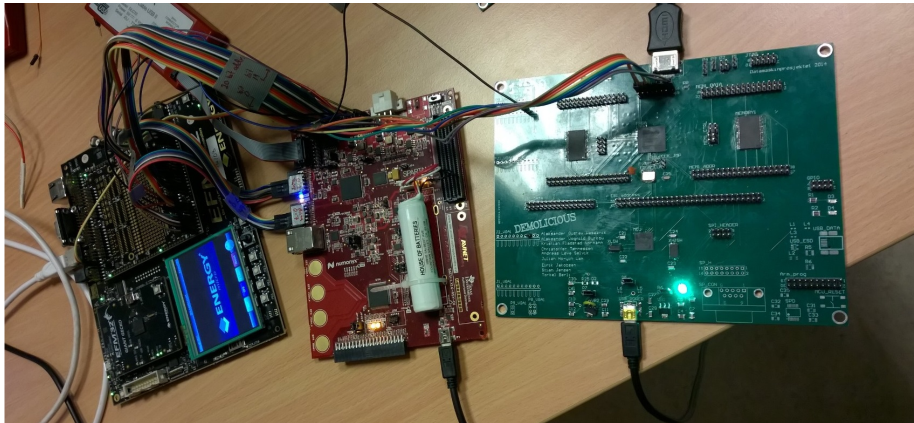


Figure 11.0.1: The working Demolicious devkit setup

In this section, the fruit of our labor, namely the Demolicious system is presented. The system has been sucessfully run on a concoction of FPGA and MCU devkits, using the HDMI port of the PCB for video output. This setup is showcased in figure 11.0.1.

A PCB version is almost fully operational. A version with both FPGA and MCU flashed with their respective images managed to output animations, with some undiagnosed showstopping software glitches. The venture had to be paused due to a lack of time towards the end of the project.

Kernel images presented in this section are taken from the same setup presented in figure 11.0.1.

11.1 Scalability of the Demolicious system

The architecture of the Demolicious system easily scales to a larger number of processors. As there is a linear relationship between the number of processor cores on chip and processor throughput, one can simply add more cores to increase performance.

Limiting factors to this scalability include:

1. The more cores, the lower the clock frequency, as signal propagation time and fanout increases
2. Space available on the chosen FPGA
3. Power consumption constraints, as more cores increase active power consumption

Cores	Crit. path	Max freq.	Dynamic+quiescent power	LX16	LX45
2	17.103ns	58.469MHz	0.272 W: 0.088 + 0.184	✓	✓
4	17.959ns	55.682MHz	0.292 W: 0.107 + 0.184	✓	✓
8	19.722ns	50.108MHz	0.353 W: 0.168 + 0.186	X	✓
16	X	X	X	X	X

Table 11.1.1: Hardware configurations compared. Harvested from post place & route static simulation.

The 4-core design fits with room to spare on the LX16, but the 8-core design does not fit. The LX45 shifts this up one notch, fitting the 8-core design, but being unable to place & route the 16-core design. For all processor designs, the critical path passes from the immediate field of the instruction through the ALU into the active register file. This is something that could be decreased drastically by pipelining the processor.

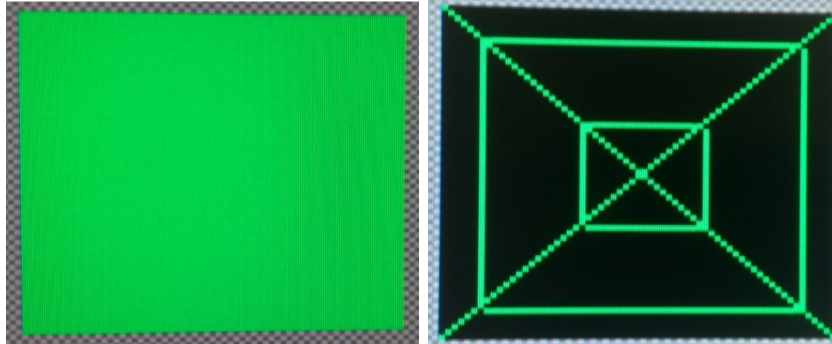
Figure 11.1.1 shows that there is a negligible drop in maximum frequency from two to eight cores. At 50.108MHz with 8 cores, the GPU has an instruction throughput of 400 MIPS. This compares favorably to the 117 MIPS of the two-core architecture.

It does however come with a 100mAh increase in power draw. Luckily this only constitutes a 30% total increase in power, considerably less than the fourfold improvement in GPU throughput.

11.2 Performance

In the Demolicious system, kernel calls can be issued by the CPU, taking only a few cycles. This means that the number of frames per second (FPS) that the

system can display is dominated by the run time of the kernels.



(a) Output from the green screen kernel.

(b) Output from the tunnel kernel.

Figure 11.2.1: Running two example kernels.

Figure 11.2.1 shows the output from the green screen kernel, and a more complex tunnel effect kernel (listing C). It's desirable that both these kernels can be run at about 30 FPS. Using the results presented in section 11.1, the expected frames per second for varying resolutions can be estimated.

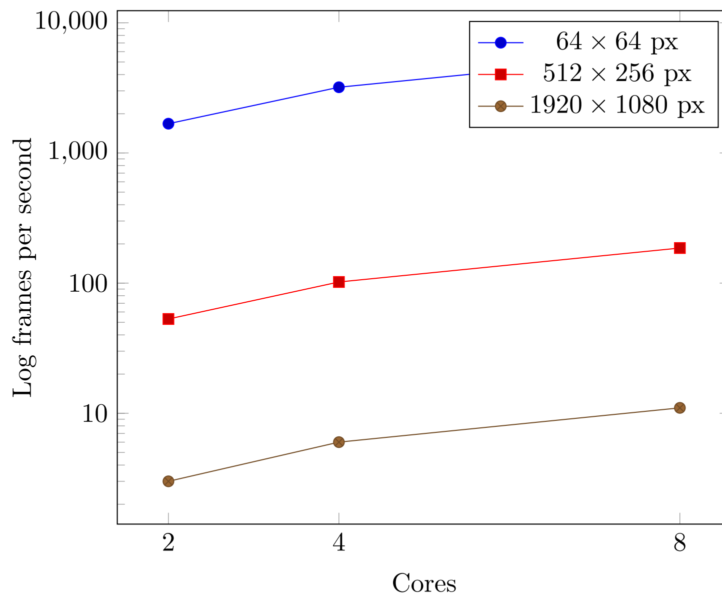


Figure 11.2.2: Running the green screen kernel.

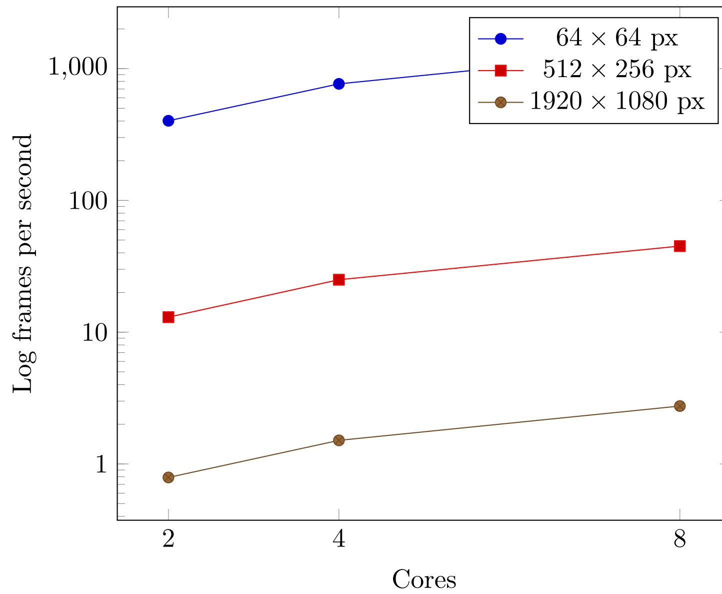


Figure 11.2.3: Running the tunnel kernel.

The figures 11.2.3, and 11.2.2 display the relationship between frame rate, resolution and number of cores. Both figures display that doubling the amount of core roughly doubles the frame rate.

For a configuration of cores, the time it takes to process one pixel is constant. This means that the time to execute one kernel scales linearly with the resolution. When the output resolution is increased, the amount of pixels to process increases quadratically. As a consequence the frame rate decreases quadratically when the resolution grows.

For the target resolution for the project, which is 512×256, the project goal of maintaining 30 fps is achieved.

11.3 Video output

The Demolicious system can output to a screen using HDMI. The minimum resolution permitted by the HDMI protocol is 640×480, but the size of the data memory limits the actual resolution to 512×256 pixels. The rest of the screen is padded with a checker pattern. Most of the time the output image is correct. However, the output image is distorted under certain conditions.

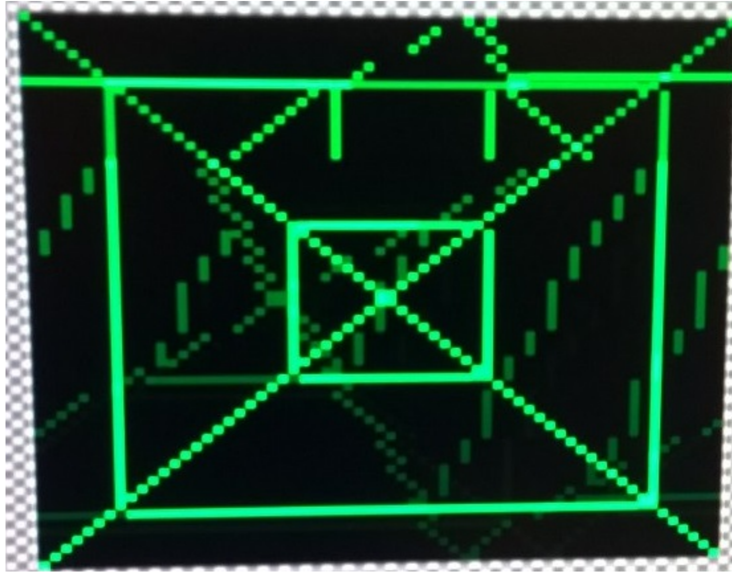


Figure 11.3.1: Flickering when running the tunnel kernel. This picture was exposed over two frames.

Some kernels exhibit intermittent flickering (Figure 11.3.1). The exact reason for why this occurs is unclear, but it can be observed that the flicker contains parts of the last frame. This may be caused by a failure in the synchronization mechanism in the video unit.

Since the GPU has priority on memory access, the video unit may get starved for data. When the video unit is starved the buffer containing pixels to output will underflow. For the duration of the starve, a line containing the previous pixel on the bus will be displayed on the screen.

11.4 Single vs double buffering

Screen tearing is a visual artifact where parts of two consecutive frames are displayed at the same time. This occurs because the video unit reads the frame buffer before the GPU has finished rendering it. In figure 11.4.1 the artifact can be observed, occurrences are marked with red circles.

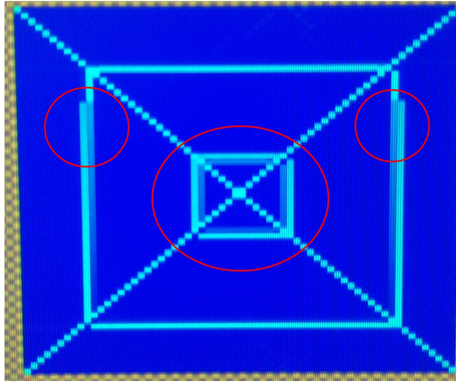


Figure 11.4.1: Single buffering.

Double buffering is a technique to remove this artifact. As the name implies two independent frame buffers are used. While one frame buffer is being read and displayed on screen, the next frame is rendered to an off-screen frame buffer. Once the frame has finished rendering the frame buffers are swapped, and the frame is displayed by the video unit. In figure 11.4.2 it can be observed that double buffering improves the quality of the image substantially. The image in the figure does have some artifacts, but the ones caused by single buffering are no longer present.

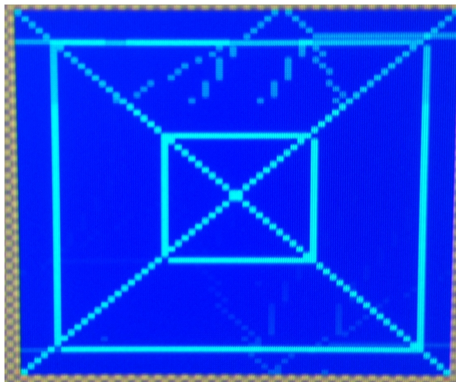


Figure 11.4.2: Double buffering.

11.5 Demolicious Power efficiency

A Demolicious setup of 8 cores 11.1.1 has a power draw of 0.353 W. By quickly and efficiently executing the kernel at hand, the GPU can reduce its total static power consumption. Reducing the dynamic consumption however, is more difficult. The current GPU architecture will execute nops when no kernels are active, keping dynamic power consumption sligthly lower than average, as no memory requests need to be served. Therefore, Demolicious has been designed with the goal 4.2.1 of allowing the host system to sleep as much as possible.

The Giant Gecko microcontroller used as the host device has four different energy modes, each lower energy mode more efficient, but more constrained. Deeper power levels may conserve more power, but require a longer time to wake up. EM0 is normal operation, while EM2 (Deep sleep mode) is the deepest power level with acceptable wake-up times for our usecase (somewhere around 2 ms) [11].

There are two different sleep patterns used by the MCU based on the executing program:

1. A visual animation or similar, rendering at a known target FPS. The MCU can stay at energy mode EM2 when kernels are executing, waking at fixed intervals to update and relaunch kernels.
2. Programs working on some large dataset, where it is beneficial to tightly pack kernel execution, reducing GPU idling. The host device can therefore enter EM2 after launching a kernel, using the kernel complete pin to trigger an interrupt, waking the host device from sleep.

The design is sound, and has been implemented successfully before by the authors, but there was not enough time to fully implement it for this project in the host code. ehe following numbers are therefore harvested from Silicon labs Simplicity studio energyAware Battery estimator.

Demolicous with 8 execution cores can run the tunnel kernel at 50 FPS @ 512x256, spending 12.5 ms per frame 11.2.3. The tunnel kernel is to be executed 30 times per second, with the MCU waking via a timer interrupt to spawn the new kernels. Kernel launch overhead is roughly 10µ + $2ms \ = 2ms$. When waking 30 times per second, this results in 60 ms awake per second. EM0 with EBI consumes 142.6 mA, EM2 190µa. This then averages to roughly $142.6mA * 0.060 + 0.19mA * 0.940 = 8.734$ mA average consumption.

For the large dataset kernel, assume it takes 100ms to complete execution. The MCU will have to wake 10 times per second, using 2 ms to wake and a negligible overhead to deploy new kernels. This results in $10 * 2ms = 20ms$ spent in EM0, and 980 ms spent in EM2. This averages to $142.6mA*0.020+0.19mA*0.980 = 3.038mA$ average consumption.

Without any use of sleep mode for the MCU, the 142.6 mA drawn make for a total power consumption of $142.6mA * 3.3V = 470.6mW$. However, for the tunnel kernel example, making use of sleep mode puts the power consumption at $8.734mA*3.3V = 28.8mW$. For the large dataset kernel, power consumption is at $3.038mA * 3.3V = 10.0mW$.

These results show that there is a clear gain to be had from introducing low power modes.

Power budget

The computer is powered by a EH-70p USB charger for the Nikon Coolpix S2700 camera [12, p. 196]. This charger outputs a 5 V voltage and delivers 550 mA. This gives the power input an upper bound of power consumption at $550mA * 5V = 2.75W$.

Based on checking the memory datasheet [6], page 4, the average current drawn from the SRAM is 100mA. At two components, the memory energy consumption becomes $2 * 100mA * 3.3V = 660mW$, which makes them the most energy demanding part of the computer.

With these major components and their estimated power consumption, one can see that the final system draws approximately $660mW + 353mW + 28mW = 1041mW$. The total power consumed is likely a little higher on account of signal propagation, but it is still significantly less than the $2.75W$ we accounted for in the power design.

Chapter 12

Discussion

Hindsight is 20/20. Looking back at the development of Demolicious, some design decisions have stood the test of time better than others. A project of this size with such a short timeframe requires quick decision making, sometimes sacrificing the optimal solution for a working solution. For example, an extreme focus on redundancy in the design of the PCB has resulted in almost all components being usable, sacrificing some power efficiency in the process. This section will present controversial design decisions, better solutions where available, and workarounds.

12.1 Energy Efficiency

12.1.1 Optimizing FPGA power usage

While it was relatively easy to save power on the CPU side of things, the FPGA posed a bigger challenge. The CPU easily enters deeper sleep modes, but no such thing exists by default for the Spartan-6 FPGA. One way to reduce FPGA power consumption is to clock gate components, in effect turning them off when unneeded. It turns out that Xilinx® supports dynamic clock gating for the Spartan-6 series used in this project [16]. This feature can reduce power consumption up to 30% for components opting to use chip enable signals.

The presented architecture has some usage of chip select signals, but does not shut down execution cores, instruction fetch or the LSU unit when idling. The addition of such a system-wide chip enable signal would reduce power usage during GPU idle time, a feature that the next iteration of Demolicious should include. This was not implemented for this project however, mainly due to time constraints.

Now for a quick calculation to see how much power can be saved in an absolute best-case scenario. Demolicious with 8 execution cores can run the tunnel kernel at 50 FPS @ 512x256 (as seen in figure 11.2.3), spending 12.5 ms per frame. When targeting 30 FPS, the GPU would spend 375 ms calculating frames, allowing the GPU to be in a low power state for almost 62.5% of the time. Using

a theoretical maximum of 30% dynamic power savings for the low power state, the dynamic power consumption of 0.168 W from figure 11.1.1, can be reduced to $0.70 * 0.625 * 0.168W + 0.375 * 0.168W = 0.1365W$. This represents a savings in dynamic power consumption of 18.7%. The total FPGA power draw would be reduced to $0.1365W + 0.1860W = 0.3225W$, a 8.6% reduction over the original 0.3530 W. This shows that FPGA power consumption still is dominated by the quiescent draw.

12.1.2 Physical Implementation

To make a PCB as energy efficient as possible, one relies upon good practices. The wires have to be as short as possible and the board as small as possible. This will optimize for lowest possible static power consumption. Short wires for signal wires will make dynamic power consumption as low as possible.

The backup oriented design of the PCB doesn't fit very well in with this. The amount of headers make wires unnecessarily long, as well as making the board itself quite large. Headers need to make a hole through the whole board, which makes it impossible to put wires on that place. Since the PCB has six signal layers, it means that each header removes the possibility of six wires going through where the header is. This makes the static power consumption higher than it could be.

By having the entire bus on headers, the wires between the CPU and the GPU is longer than they could have been. This makes the dynamic power consumption higher than if the wires are shorter. However all of this is a trade-off as there was only one chance for the PCB to work.

12.2 Programming Challenges

There are several limitations and issues a programmer must have in mind when developing programs for the Demolicious system. First of all, since no compiler for high-level languages is available, kernel code must be written in assembly. The limitations of the language and relatively low expressiveness it provides, makes kernels verbose and require substantial mental overhead.

The implementation of conditional execution, through predicated instructions only, can also be quite difficult to get accustomed to. Chaining multiple conditions together to form a correct logical structure, can require sketching and some upfront planning of the program flow even for simple kernels.

These are, however, problems which are quite easily fixed by allowing compilation from a language with a higher abstraction level. A tougher limitation is, although also one which can be handled by a compiler, the limitations on the frequency of memory instructions. Depending on the number of GPU cores, memory instructions can't occur more than each X cycles. Dealing with this, without the performance drop of executing nop instructions is important when rendering live graphics.

12.3 SRAM - Alternative Memory Layouts

The presented architecture uses interlaced SRAM banks, presenting one large unified memory space. An alternative approach is to dedicate one SRAM bank to the LSU, the other to the HDMI unit. This approach will work when using double buffering, as the LSU and HDMI unit can swap banks on each framebuffer flip. With dedicated banks there will be no need for arbitration, as there never will be more than one unit accessing each at bank the time. This would result in the HDMI unit never starving due to memory lockout from the LSU. The LSU would also be simpler, needing only a single queue to service memory requests.

The major advantage of interlacing the address space of the two memory banks is that memory throughput effectively doubles. Two requests can be processed in parallel from the unit currently granted access by the SRAM arbiter. When a warp of size n requests memory access, it will take $n/2$ cycles to dispatch all requests, as each SRAM bank services half of the requests. Therefore, the barrel height only has to be equal to half of the number of processor cores. With the dedicated memory banks, however, only one memory request can be serviced each cycle. This requires that the barrel height is increased to n.

As the register directories compose a significant part of the design footprint, the barrel height reduction is significant. Signal fanout is also reduced from and to the register banks, dampening the reduction in clock frequency when scaling the number of computation cores. More hardware resources can now be allocated to processor cores, power is saved due to less registers being used, and the clock frequency can stay higher. Drawbacks include the occasional flicker due to HDMI unit starvation.

12.4 Multiple clock domains in HDMI unit

The HDMI unit needs to output words with a frequency of 25 MHz. If it were to share clock with the rest of the system, the system clock would have to be 25 Mhz itself or a multiple thereof. To avoid imposing this limitation, the buffer in the HDMI unit crosses clock domains. Since the buffer receives data at native memory speed, it is filled faster than the HDMI unit can read from it. The data read out at 25 MHz also need to be serialized to a frequency of 250 MHz before being output from the FPGA. This is to the knowledge of the authors, the first project in the Computer Project course with a design incorporating multiple clocks.

12.5 Troubleshooting and Workarounds

During testing, it became apparent that there were two major issues that needed to be fixed with the PCB. Firstly, the pin on the FPGA used for the clock was not one of the dedicated clock pins. Secondly, the clock component had not been implemented according to spec.

FPGA clock pin problem

An error with the pin connection between the FPGA and the clock was discovered during testing. The FPGA demanded a special pin for the clock, but the PCB routed the clock to a regular GPIO pin. After checking the PCB layout, a pin on the EBI bus was shown to be able to work as clock for the FPGA.

Since the clock is connected by jumpers to the FPGA, the pin on the EBI bus could be wired to the clock jumper on the FPGA, solving the problem.

Wrong implementation of the clock

Due to a poor schematic in the clock component datasheet, the clock was wired incorrectly on the PCB, as illustrated in figure 12.5.1.

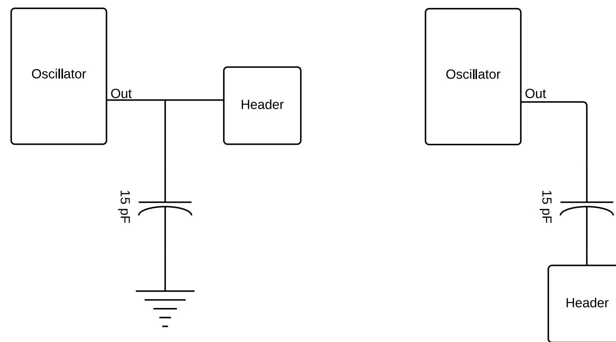


Figure 12.5.1: On the left: How the clock from the FPGA should be connected. On the right: how it was connected on the PCB.

Because of this, the clock did not work. This problem was solved by soldering together the pads of the SMD (surface mounted device) capacitor to allow the clock signal to flow freely to the header pin. A through-hole version of the removed capacitor was placed on one of the soldering pads and the other end grounded, see figure 12.5.2. This caused the circuit on the PCB to match up with the correct circuit from the data sheet.

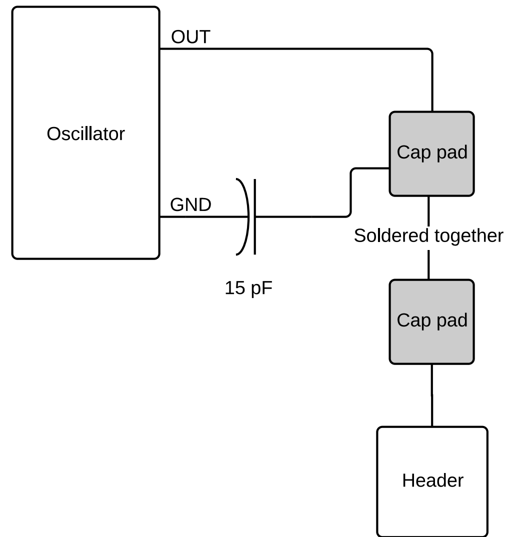


Figure 12.5.2: How the FPGA clock fix was implemented.

Mismatch LDO

During the testing phase, the LDO for the 1.2V did not output the correct voltage. The 1.2V LDO is the same model as the 3.3V LDO, but has a different footprint, see figure 12.5.3.

It was assumed that both components had the same, which resulted in mismatch between the footprint and the component. The solution was to place the LDO diagonally, and manually solder a wire to the correct solderpad on the PCB.

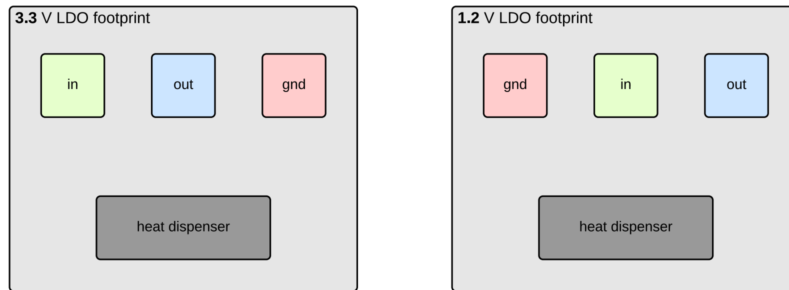


Figure 12.5.3: On the left: 3.3 V LDO footprint. On the right: 1.2 V LDO footprint.

12.6 Hardware components

In this section we will discuss some of the physical design, alternative solutions and the practical solutions that arose during physical implementation.

USB data input

In the initial design, a USB connector was introduced to receive data from a host PC. However, during the implementation phase it was discovered to be unnecessary. We discovered that the software applications we wanted to run on the computer, could be included when programming the microcontroller with the JTAG. This worked sufficiently for our needs. Since at the time we still had much else to do, implementing input by USB became a very low priority and in the end never happened. This also applied to the serial backup as well.

VGA-port

The PCB was designed with two VGA headers, one for the FPGA and one for the MCU. An alternative solution is having one VGA port with exposed headers. With exposed headers from the MCU and the FPGA as well, one could change which one the VGA port is connected to by jumpers like 12.6.1.

By removing one of the VGA ports, the size of the PCB could have been reduced, or used for other components.

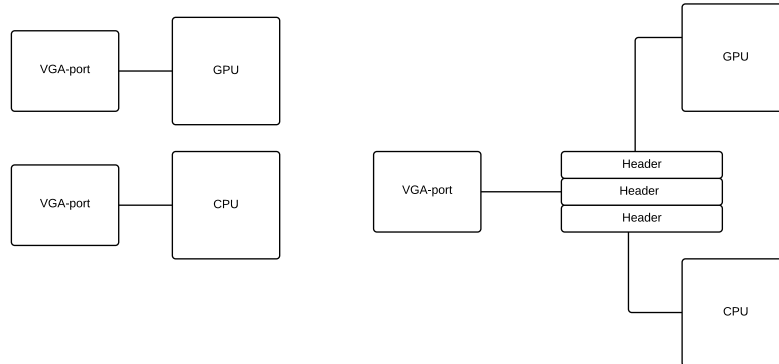


Figure 12.6.1: On the left: The current setup on our PCB. On the right: The alternative solution.

12.6.1 Clocks

Oscillator

As seen on figure 12.6.2, the measured period of the oscillator, with a ruler, is around $8.5ns$, which translates to around 118 MHz. The tolerance of the oscillator is up to 100 ppm, see datasheet [3]. Since using a ruler as a mesure tool will give unreliable results, one can assume the oscillator gives the correct output.

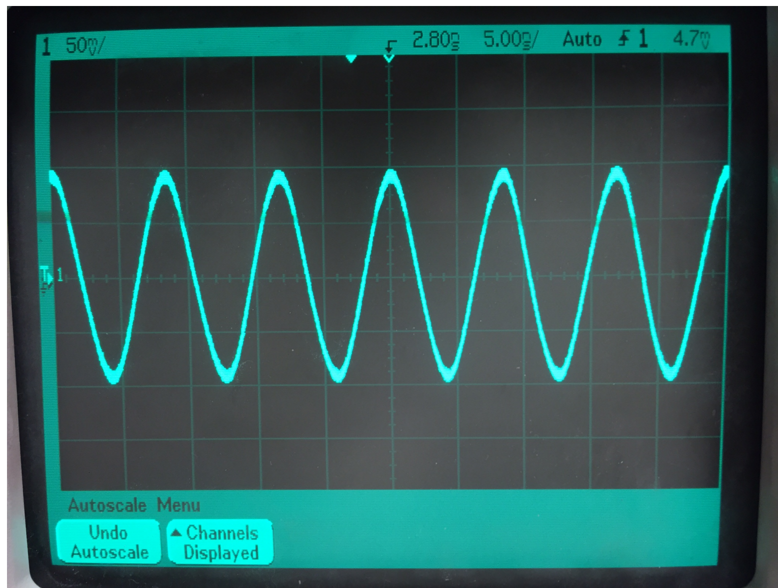


Figure 12.6.2: The oscillator on an oscilloscope.

MCU crystals

As can be seen from the frequency tests 10.4.2, the crystal output frequency tests did not pass their requirements. They were probed for output, but this resulted in no discernable signal on the oscilloscope. However, as the implementation continued, we found that the MCU worked as we desired. This led us to conclude that we had likely misunderstood how to measure the frequencies of the crystals.

12.7 Budget

The group was given a budget of 10 000 NOK for production of the PCB and component purchases. Ten boards were manufactured at the price of 10 103 NOK and components were ordered at the price of 7 571 NOK. (see appendices A.1.1 and A.2.1). This made for a total cost of 17 674 NOK, which results in a cost overrun of 76 %. There were mainly two reasons for this.

Firstly, the PCB design took so long that a fast production time was required. This, along with a very large board, drove the cost up greatly. Secondly, components for at least 5 PCBs were bought. If fewer components were bought, the price would be lower. However, this all is a part of the backup oriented design.

For this reason there are two main ways to spend less money. The total price could be driven down by having a less backup oriented design with fewer components. This would significantly lower the total cost. Also, if the time limit weren't that strict, and the board could use longer time in production, the price would be lower.

Chapter 13

Conclusion

We designed and implemented a graphics processing unit inspired system, capable of executing a large amount of threads fast enough to display graphics to a display over HDMI.

The final system meets all the requirements set in the beginning of the project. The goal of at least 30 FPS has been reached for several kernels, at the target screen resolution of 512x256 pixels when running with 8 processor cores. CPU power savings were simulated successfully, showing great promise.

A fully functional PCB was created with each individual component working. The implementation of the Demolicious system on the PCB was almost finalized, with only minor issues remaining.

There is still room for further improvement of the computer, which will be detailed in the next section.

This project has been extremely challenging and demanding. Because of this, the group as a whole have gained great insights into the inner workings of computers and GPUs.

13.1 Further Work

A beautiful thing with projects like this, is that they can be done significantly better when done the second time. This section will focus on possible further improvements of the system.

13.1.1 Architecture

As one of the consequences of the barrel processor, the processor lends itself very well to pipelining. The consecutive instructions in the pipeline will always be from different threads. As long as there are fewer pipeline stages than the height of the barrel, an instruction will complete its path through the pipeline before the next one from the same thread starts. This means that there are no

data hazards in the pipeline. Thus it is as simple as dividing the processor into stages and adding registers. Currently the critical path in Demolicious is going through the processor cores. Adding a pipeline to the processor would therefore increase the maximum frequency the processor can run at. As the SRAMs on the Demolicious can handle 100 MHz, there is room for improvement in the system clock frequency. This directly translates to higher throughput.

13.1.2 Physical design

The physical design of Demolicious worked as intended. A second version will allow for a greater energy efficiency and a smaller size. All headers and unnecessary backup solutions can be removed, letting a new design focus on a small PCB with short wires and small power planes. If a more powerful computer is to be made, a more powerful FPGA can be introduced along with a matching number of SRAMs, as a more powerful FPGA will have a greater memory need.

Furthermore, power and data can be unified in a single USB connection. The power USB and the FPGA are currently positioned far from one another, but in a new design they can be moved closer so that the 1.2V wire that powers the FPGA can be made as short as possible. Lastly, it would have been convenient to have had flash memory for the FPGA as this would allow for retaining the program after the power has been shut off. This would eliminate the need to flash the FPGA after each power off.

13.1.3 Compiler

A major hurdle to exploiting the full potential of the Demolicious system was the difficulty and lack of user-friendliness when writing programs in Demolicious assembly. To greatly simplify the programming, a C compiler with Demolicious assembly as target could be implemented.

Bibliography

[1] John Conway. *Conway's Game of Life*. http://en.wikipedia.org/wiki/Conway%27s_Game_of_Life.

[2] John L. Hennessy David A. Patterson. *Comuter Organization and Design*.

[3] Fox Electronics. *Fox XpressO Oscillator*. http://www.foxonline.com/pdfs/FXO_HC73.pdf.

[4] fpga4fun. *HDMI pinout*. http://www.fpga4fun.com/HDMI.html.

[5] IDC. *PC Market Beats Expectations with Mild Improvement in Business Outlook*. http://www.idc.com/getdoc.jsp?containerId=prUS24375913.

[6] Alliance Memory inc. *SRAM datasheet*. http://alliancememory.com/pdf/sram/fa/as7c38098a.pdf.

[7] Silicon Laboratories Inc. *EFM32GG Design consideration*. http://www.silabs.com/Support%20Documents/TechnicalDocs/AN0002.pdf.

[8] Silicon Laboratories Inc. *EFM32GG Reference Manual*.

[9] Silicon Laboratories Inc. *EFM32gg990 datasheet*. http://www.silabs.com/Support%20Documents/TechnicalDocs/EFM32GG990.pdf.

[10] Silicon Labs. *EFM32 emlib Peripheral Library: EBI*. http://devtools.silabs.com./dl/documentation/doxygen/EM_CMSIS_3.20.7_DOC/emlib_gecko/html/group__EBI.html.

[11] Silicon Labs. *EFM32GG Reference Manual*. http://www.keil.com/dd/docs/datashts/energymicro/efm32gg/d0053_efm32gg_reference_manual.pdf.

[12] Nikon. *Coolpix S2700 Reference Manual*. http://cdn-10.nikon-cdn.com/pdf/manuals/coolpix/S2700RM_%28En%2903.pdf.

[13] Jon Peddie Research. *Add-in board market up in Q3, Nvidia increases market share lead*. http://jonpeddie.com/publications/add-in-board-report/.

[14] Jon Peddie Research. *GPU shipments Q2 2014 - Charts and Images*. http://jonpeddie.com/news/comments/gpu-shipments-marketwatch-q2-2014-charts-and-images/.

[15] Jon Peddie Research. *Qualcomm Single Largest Proprietary GPU Supplier*. http://jonpeddie.com/press-releases/details/qualcomm-single-largest-proprietary-gpu-supplier-imagination-technologies-t/.

[16] Xilinx®. *Xilinx® ISE® Design Suite 12 what's new*. http://www.xilinx.com/support/documentation/white_papers/wp370_Intelligent_Clock_Gating.pdf.

List of Tables

4.2.1 Goals set for the Demolicious system 14

7.5.1 The registers contained within the register files. 35

10.1.1 Unit tests for components in the Demolicious system. 47
10.3.1 JTAG tests 52
10.3.2 EBI bus tests 53
10.3.3 HDMI tests 53
10.3.4 SRAM tests 53
10.4.1 Solder plan and verification 54
10.4.2 Frequency output tests 54

11.1.1 Hardware configurations compared. Harvested from post place & route static simulation. 56

A.2.1 Component order 81

B.1.1 Register overview 83
B.3.1 R-type instructions 85
B.3.2 Instruction format for R-type instructions. 85
B.3.3 I-type instructions. 86
B.3.4 Instruction format for I-type instructions. 86
B.3.5 Pseudo instructions 86

List of Figures

3.3.1 Streaming Multiprocessor 11
3.3.2 Warps of 32 threads execute 12

5.1.1 Logical overview of the system. 16

5.2.1 Relationship between CPU and GPU code. 17

6.2.1 Overview of memory mapped address spaces on the CPU. 22
6.3.1 23
6.4.1 Starting a kernel on the GPU. 23
6.5.1 CPU-GPU interaction when loading a parameter. 24

7.2.1 A high level overview of the GPU. 27
7.3.1 Launching a kernel from the GPU's viewpoint. 27
7.4.1 Execution timing of warps without jagged scheduling 29
7.4.2 Execution timing of warps with jagged scheduling 30
7.5.1 A single processor core. 31
7.5.2 Thread spawner and neighboring components. 32
7.5.3 Thread spawner RTL implementation. 33
7.5.4 The register directory, and its neighboring components. 34
7.5.5 Using the mask bit to disable register writes. 35
7.5.6 RTL of the load/store unit. Notice the dual queueing system to handle the interlaced SRAM. 36
7.5.7 The logical overview of the video unit. A prefetching unit tries to fill the buffer when it is not full. The *Accepted* signal signifies a successful memory request. Across the clock domain boundary, the video timing unit generates control signals. When pixel data should be transferred, the *Send* signal moves pixels from the buffer to the transmission units. The 16-bit pixel data is converted to three 8-bit colors, TMDS encoded, serialized and output using differential signaling. Should the buffer go empty, the prefetch unit needs to immediately advance to the next pixel to stay synchronized with the timing unit. 37

8.0.1 Completed PCB 39
8.1.1 Conceptual overview of the PCB. Green boxes are main solutions. Blue boxes are backup plans. Gray boxes labeled "PINS" mean that these signals are exposed on headers. 40
8.4.1 Picture of the physical power circuit, where P5 is the header on which an external power source can be connected 42

10.2.1 Results from simulation and LX16 of fillscreen kernel 48
10.2.2 Results from simulation and LX16 of predicated execution 50
10.2.3 Results from simulation and LX16 of load kernel 51

11.0.1 The working Demolicious devkit setup 55
11.2.1 Running two example kernels. 57
11.2.2 Running the green screen kernel. 57
11.2.3 Running the tunnel kernel. 58
11.3.1 Flickering when running the tunnel kernel. This picture was exposed over two frames. 59
11.4.1 Single buffering. 60
11.4.2 Double buffering. 60

12.5.1 On the left: How the clock from the FPGA should be connected. On the right: how it was connected on the PCB. 66

12.5.2 How the FPGA clock fix was implemented. 67
12.5.3 On the left: 3.3 V LDO footprint. On the right: 1.2 V LDO footprint. 68
12.6.1 On the left: The current setup on our PCB. On the right: The alternative solution. 69
12.6.2 The oscillator on an oscilloscope. 70

A.1.1 PCB order 80

Listings

3.1 A sequential program filling the screen with green 10
3.2 A CUDA kernel filling a single pixel with green 10
3.3 Starting the CUDA kernel with one thread per pixel on a 1920x1080 screen 10
5.1 A simple kernel that fills the screen with the color green 17
5.2 Loading and executing a kernel 18
5.3 A kernel loading the color value from a parameter 18
5.4 Now drawing a blue screen using parameters 18
5.5 Conditional execution using predicated instructions 19
5.6 The green-screen kernel as it is actually 19
6.1 A load_kernel function call with the fillscreen kernel 22
6.2 Running a kernel 23
6.3 Setting a kernel parameter 24
10.1 Kernel to test constant memory and parameterization 48
10.2 Conditional execution using predicated instructions 49
10.3 Kernel to test loads from main memory 51

Part IV

Appendices

Appendix A

Expenses

A.1 PCB manufacturing

Att: Yaman Umuroglu

10	Board	1010.300	10103.00
		All prices in NOK	10103.00

DMProsj2014-v3

Silkscreen and annular ring fixes

187.4 147.4
8 layer: 8001 Standard 1,6mm No
1,6 mm 1 oz (38 µm) outer, 1 oz (35 µm) inner
FR-4 -- IPC-4101C /21
Lead-free HASL 2
Yes
No
White None No
Green Green Yes
No No No
No No No
Yes
0.100 0.100 No
0.100 0.100 No
0.200 0.200 No
0.20 No
No

1034 4 None None

The customer is responsible for checking that this order confirmation is in accordance with the order. Any discrepencies must be reported without delay. See http://www.elprint.no/terms.html for complete terms and conditions of sale.

29.10.2014
Anne-Lise Dahle
Elprint Norge AS

Figure A.1.1: PCB order

A.2 Component purchases

#	Description	Amount	Unit Price (NOK)	Total (NOK)
1	HDMI Receptacle	5	8.20	41.00
2	Memory	10	104.00	1040.00
3	1k Res	100	0.19	19.00
4	Buttons	25	3.35	83.75
5	LEDs	30	5.30	159.00
6	Inductors	25	1.35	33.75
7	SPD	5	16.45	82.25
8	LED Resistor	20	1.05	21.00
9	USB Receptacle	10	4.25	42.50
10	1.2V res	25	1.33	33.25
11	Low Freq. Crystal	5	9.75	48.75
12	15 Ohm res	10	1.05	10.50
13	1.2V regulator	5	4.50	22.50
14	3.3V regulator	5	22.70	113.50
15	4.7 kOhm res	10	1.05	10.50
16	ESD protection	5	1.20	6.00
17	Headers	20	9.95	199.00
18	Jumpers	200	1.05	210.00
19	470nF caps	60	3.00	180.00
20	100uF caps	35	5.39	188.65
21	4.7uF caps	40	1.65	66.00
22	100nF caps	100	0.75	75.00
23	10uF caps	10	3.58	35.80
24	1uF caps	10	2.55	25.50
25	12pF caps	10	1.12	11.20
26	22pF caps	10	1.65	16.50
27	15pF caps	10	1.25	12.50
28	10nF caps	10	3.60	36.00
29	FPGA	10	402.32	4023.20
30	MCU	10	51.35	513.50
31	48 MHz Crystal	5	14.27	71.30
32	120 MHz Oscillator	5	17.10	85.50
33	Noise Suppressor	5	10.84	54.18
			Total cost (NOK)	7571.08

Table A.2.1: Component order

Appendix B

Instruction set architecture

This appendix will provide an overview of the instruction set supported by the GPU.

B.1 Registers

The following registers are available:

Number	**Name**	**Description**	**R/W**	**Size**
$0	zero	Always contains the value zero	Read-only	16
$1, $2	id_hi, id_lo	The current thread's ID	Read-only	16
$3, $4	address_hi, address_lo	Address used by load & store instructions	Read/Write	16
$5	data	Data loaded/stored by load & store instructions	Read/Write	16
$6	mask	Conditional instructions will be masked (section B.2) when this register is set to 1.	Read/Write	1
$7-$15	General-purpose		Read/Write	16

Table B.1.1: Register overview

Special registers may be referenced by their name in assembly code.

B.2 Predicated instructions

The only supported way of conditional execution is through predicated instructions (masking). All instructions except for `sw` have a predicated version which will only execute when masking is disabled.

In assembly, an instruction prepended with a question mark will be executed conditionally. The first bit of the instruction will be set to one for the conditional versions. These predicated instructions will still be executed, but they will never store their result in the destination register.

Masking is controlled by the dedicated masking register. Instructions can write to this register to turn masking on and off. The register is only one bit, and will therefore only keep the least significant bit of data written to it.

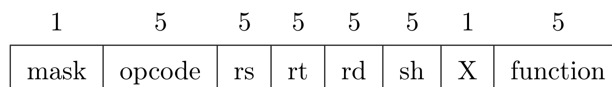
B.3 Instructions

B.3.1 R-type Instructions

All R-type instructions have opcode `00000`. Their syntax, meaning and underlying ALU function are described in table B.3.1.

Instruction	**Example**	**Meaning**	**ALU Function**
Add	add $1, $2, $3	$1 = $2 + $3	0x4
Subtract	sub $1, $2, $3	$1 = $2 - $3	0x5
Multiply	mul $1, $2, $3	$1 = $2 * $3	0x9
And	and $1, $2, $3	$1 = $2 & $3	0x6
Or	or $1, $2, $3	$1 = $2 | $3	0x7
Xor	xor $1, $2, $3	$1 = $2 ^ $3	0x8
Set on less than	slt $1, $2, $3	$1 = ($2 < $3) ? 1 : 0	0x3
Set on equal	seq $1, $2, $3	$1 = ($2 == $3) ? 1 : 0	0xA
Shift Left Logical	sll $1, $2, 10	$1 = $2 << 10	0x0
Shift Right Logical	srl $1, $2, 10	$1 = $2 >>> 10	0x1
Shift Right Arithmetic	sra $1, $2, 10	$1 = $2 >> 10	0x2

Table B.3.1: R-type instructions



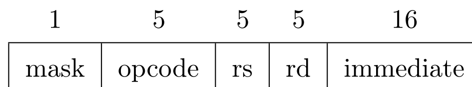
1	5	5	5	5	5	1	5
mask	opcode	rs	rt	rd	sh	X	function

Table B.3.2: Instruction format for R-type instructions.

B.3.2 I-type Instructions

Instruction	**Example**	**Meaning**	**Opcode**
Load constant	ldc $7, 1	$7 = constant_memory[1]	0x2
Add immediate	addi $7, $7, 2	$7 = $7 + 2	0x1
Load	lw	$data = memory[$address]	0x8
Store	sw	memory[$address] = $data	0x4
Thread finished	thread_finished	Stops executing the kernel	0x10
Nop	nop	Do nothing	0x0

Table B.3.3: I-type instructions.



1	5	5	5	16
mask	opcode	rs	rd	immediate

Table B.3.4: Instruction format for I-type instructions.

B.3.3 Pseudo Instructions

Some additional instructions are supported by the assembler in order to make programming for the GPU easier. An overview of these instructions is provided in table B.3.5.

Instruction	**Example**	**Translated instruction**
Move	mv $1, $2	add $1, $0, $2
Load immediate	ldi $1, 1	addi $1, $0, 1

Table B.3.5: Pseudo instructions

Appendix C

Commented tunnel kernel

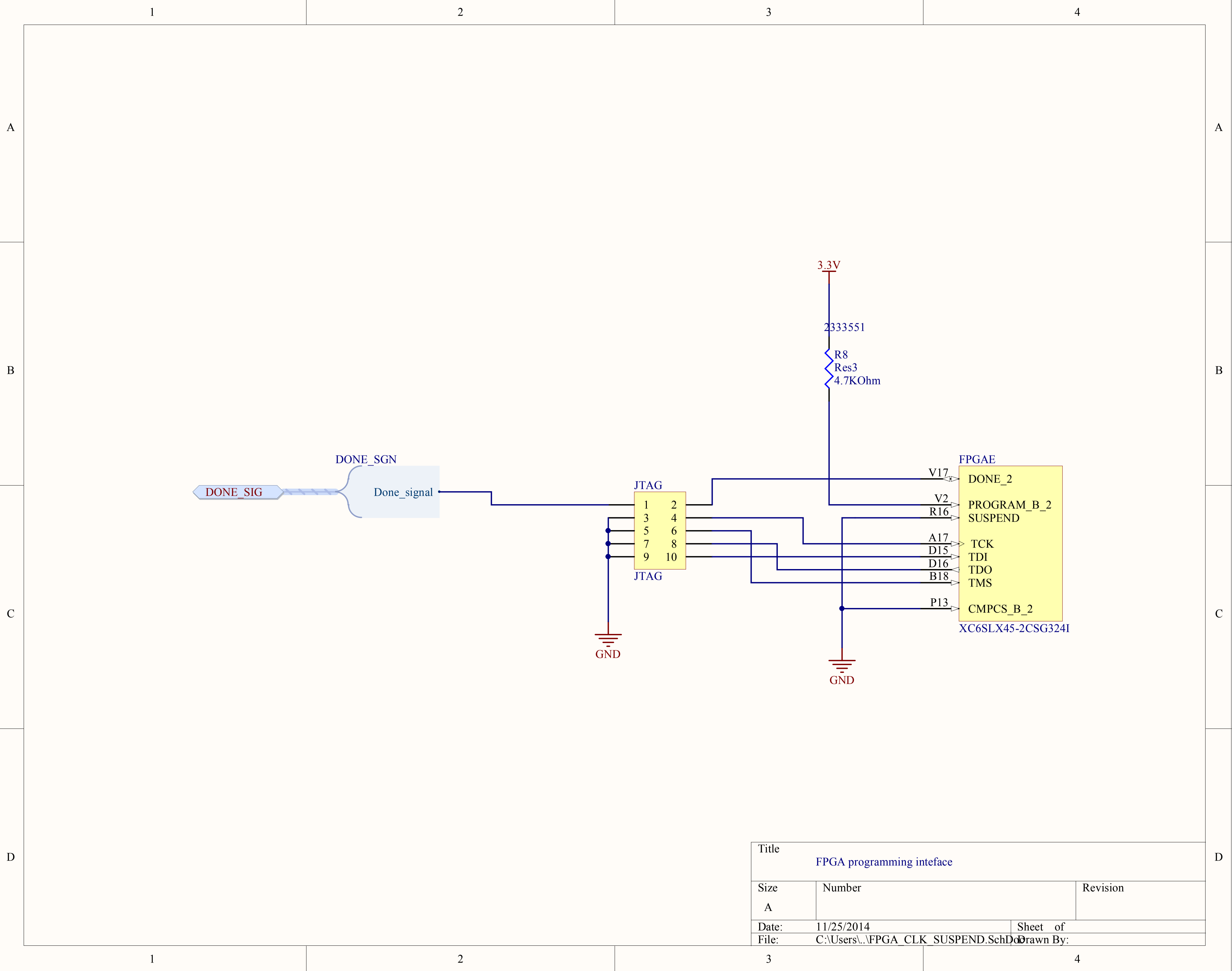
```
1  ; The idea is to draw a green tunnel.
2  ; This effect consists of 2 squares in different sizes and a X across
       the screen
3  ; To make this happen assume we are on a pixel that should be drawn
       green
4  ; We then make a series of checks to verify if we are on the correct
       pixel or not
5  ; If we ever see that we are on the wrong pixel, we will set our mask
       bit high
6  ; This will make it so that we draw black instead of green
7  ;
8  ; To make this happen we have to use a general purpose register as
       masking register
9  ; This is because if the masking value gets set high, we don't want to
       overwrite
10 ; it with low, so we need to remember it.
11
12 ; First square
13 ldi $15, 0b111111         ; 63 bitmask
14 ldi $14, 64               ; Set width & height of screen into $14
15 and $7, $15, $id_lo       ; Load x value into $7
16 srl $8, $id_lo, 6         ; Load y value into $8
17
18 ldi $data, 0b0000011111100000    ; Set default color to green
19
20 mv $12, $0                ; shadow mask - used as temporary mask bit to
21                           ; be able to read masking value
22
23 ldc $13, 10               ; Set offset from edges
24 sub $14, $14, $13         ; Set offset from opposite edges
25
26 ; Draw LEFT line of box
27 seq $10, $7, $13          ; Checks that we are on correct x value
28
29 slt $11, $13, $8          ; Checks that we are inside our box from the
       top
30 and $10, $10, $11
31
32 slt $11, $8, $14          ; Checks that we are inside our box from the
       bottom
33 and $10, $10, $11
34
35 or $12, $12, $10          ; Adds result into masking bit
```

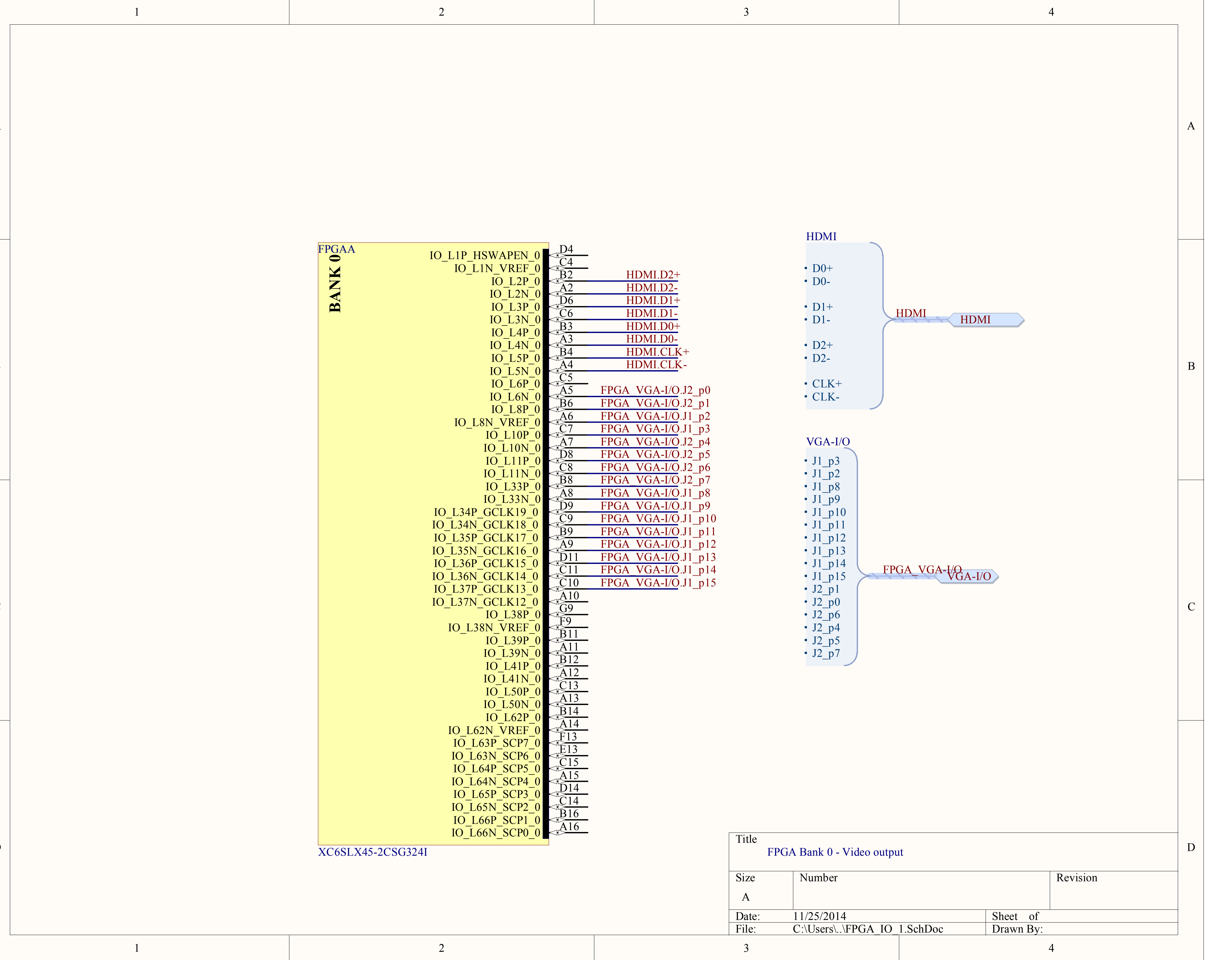
```
36  ; Draw RIGHT line of box
37  seq $10, $7, $14        ; Same system as above, but checking the other
38                          ;side of the box
39  slt $11, $13, $8
40  and $10, $10, $11
41
42  slt $11, $8, $14
43  and $10, $10, $11
44
45  or $12, $12, $10
46
47
48  ; Draw UPPER line of box
49  seq $10, $8, $13        ; Instead of checking the vertical lines, the
        horizontal lines
50                          ; are checked
51  slt $11, $13, $7
52  and $10, $10, $11
53
54  slt $11, $7, $14
55  and $10, $10, $11
56
57  or $12, $12, $10
58
59
60  ; Draw LOWER line of box
61  seq $10, $8, $14        ; Same as above, but lower line instead of
        upper
62
63  slt $11, $13, $7
64  and $10, $10, $11
65
66  slt $11, $7, $14
67  and $10, $10, $11
68
69  or $12, $12, $10
70
71
72  ; Next square
73  addi $14, $0, 64        ; Load screen width & height into $14
74  ldc $13, 11             ; Set new offset from edges
75  sub $14, $14, $13       ; Set offset from opposite edges
76
77  ; Draw LEFT line of box
78  seq $10, $7, $13        ; Same check for the square as previous square
79
80  slt $11, $13, $8
81  and $10, $10, $11
82
83  slt $11, $8, $14
84  and $10, $10, $11
85
86  or $12, $12, $10
87
88  ; Draw RIGHT line of box
89  seq $10, $7, $14
90
91  slt $11, $13, $8
92  and $10, $10, $11
93
94  slt $11, $8, $14
95  and $10, $10, $11
96
97  or $12, $12, $10
```

```
 98  ; Draw UPPER line of box
 99  seq $10, $8, $13
100
101  slt $11, $13, $7
102  and $10, $10, $11
103
104  slt $11, $7, $14
105  and $10, $10, $11
106
107  or $12, $12, $10
108
109  ; Draw LOWER line of box
110  seq $10, $8, $14
111
112  slt $11, $13, $7
113  and $10, $10, $11
114
115  slt $11, $7, $14
116  and $10, $10, $11
117
118  or $12, $12, $10
119
120
121  ; Half cross
122  seq $10, $7, $8     ; if x = y then we are on \
123
124  or $12, $12, $10
125
126  ; Other half cross
127  addi $13, $0, 0b111111      ; 63 bitmask
128  sub $14, $13, $8     ; negative y valye, check /
129
130  seq $10, $14, $7
131
132  or $12, $12, $10
133
134  mv $mask, $12        ; Store temporary mask value into actual mask
         register
135
136  ? ldi $data, 0x0000 ; Write black if not on either squares or on the
         cross
137
138  ldc $10, 5
139  add $address_lo, $10, $id_lo
140  sw                   ; save values
```

Appendix D

PCB schematics


3.3V
2333551
R8
Res3
4.7KOhm
DONE_SIG
DONE_SGN
Done_signal
JTAG
1 2
3 4
5 6
7 8
9 10
JTAG
GND
FPGAE
V17
DONE_2
V2
PROGRAM_B_2
R16
SUSPEND
A17
TCK
D15
TDI
D16
TDO
B18
TMS
P13
CMPCS_B_2
XC6SLX45-2CSG324I
GND
Title
FPGA programming inteface
Size
A
Number
Revision
Date: 11/25/2014
Sheet of
File: C:\Users\..\FPGA_CLK_SUSPEND.SchDoc
Drawn By:



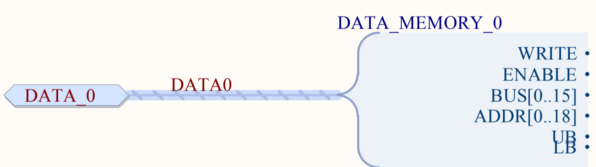
FPGAA
BANK 0
IO_L1P_HSWAPEN_0 D4
IO_L1N_VREF_0 C4
IO_L2P_0 B2 HDMI.D2+
IO_L2N_0 A2 HDMI.D2-
IO_L3P_0 D6 HDMI.D1+
IO_L3N_0 C6 HDMI.D1-
IO_L4P_0 B3 HDMI.D0+
IO_L4N_0 A3 HDMI.D0-
IO_L5P_0 B4 HDMI.CLK+
IO_L5N_0 A4 HDMI.CLK-
IO_L6P_0 C5
IO_L6N_0 A5 FPGA_VGA-I/O.J2_p0
IO_L8P_0 B6 FPGA_VGA-I/O.J2_p1
IO_L8N_VREF_0 A6 FPGA_VGA-I/O.J1_p2
IO_L10P_0 C7 FPGA_VGA-I/O.J1_p3
IO_L10N_0 A7 FPGA_VGA-I/O.J2_p4
IO_L11P_0 D8 FPGA_VGA-I/O.J2_p5
IO_L11N_0 C8 FPGA_VGA-I/O.J2_p6
IO_L33P_0 B8 FPGA_VGA-I/O.J2_p7
IO_L33N_0 A8 FPGA_VGA-I/O.J1_p8
IO_L34P_GCLK19_0 D9 FPGA_VGA-I/O.J1_p9
IO_L34N_GCLK18_0 C9 FPGA_VGA-I/O.J1_p10
IO_L35P_GCLK17_0 B9 FPGA_VGA-I/O.J1_p11
IO_L35N_GCLK16_0 A9 FPGA_VGA-I/O.J1_p12
IO_L36P_GCLK15_0 D11 FPGA_VGA-I/O.J1_p13
IO_L36N_GCLK14_0 C11 FPGA_VGA-I/O.J1_p14
IO_L37P_GCLK13_0 C10 FPGA_VGA-I/O.J1_p15
IO_L37N_GCLK12_0 A10
IO_L38P_0 G9
IO_L38N_VREF_0 F9
IO_L39P_0 B11
IO_L39N_0 A11
IO_L41P_0 B12
IO_L41N_0 A12
IO_L50P_0 C13
IO_L50N_0 A13
IO_L62P_0 B14
IO_L62N_VREF_0 A14
IO_L63P_SCP7_0 F13
IO_L63N_SCP6_0 E13
IO_L64P_SCP5_0 C15
IO_L64N_SCP4_0 A15
IO_L65P_SCP3_0 D14
IO_L65N_SCP2_0 C14
IO_L66P_SCP1_0 B16
IO_L66N_SCP0_0 A16
XC6SLX45-2CSG324I
HDMI
D0+
D0-
D1+
D1-
D2+
D2-
CLK+
CLK-
HDMI
HDMI
VGA-I/O
J1_p3
J1_p2
J1_p8
J1_p9
J1_p10
J1_p11
J1_p12
J1_p13
J1_p14
J1_p15
J2_p1
J2_p0
J2_p6
J2_p4
J2_p5
J2_p7
FPGA_VGA-I/O
VGA-I/O
Title
FPGA Bank 0 - Video output
Size
A
Number
Revision
Date: 11/25/2014
Sheet of
File: C:\Users\..\FPGA_IO_1.SchDoc
Drawn By:

FPGAB

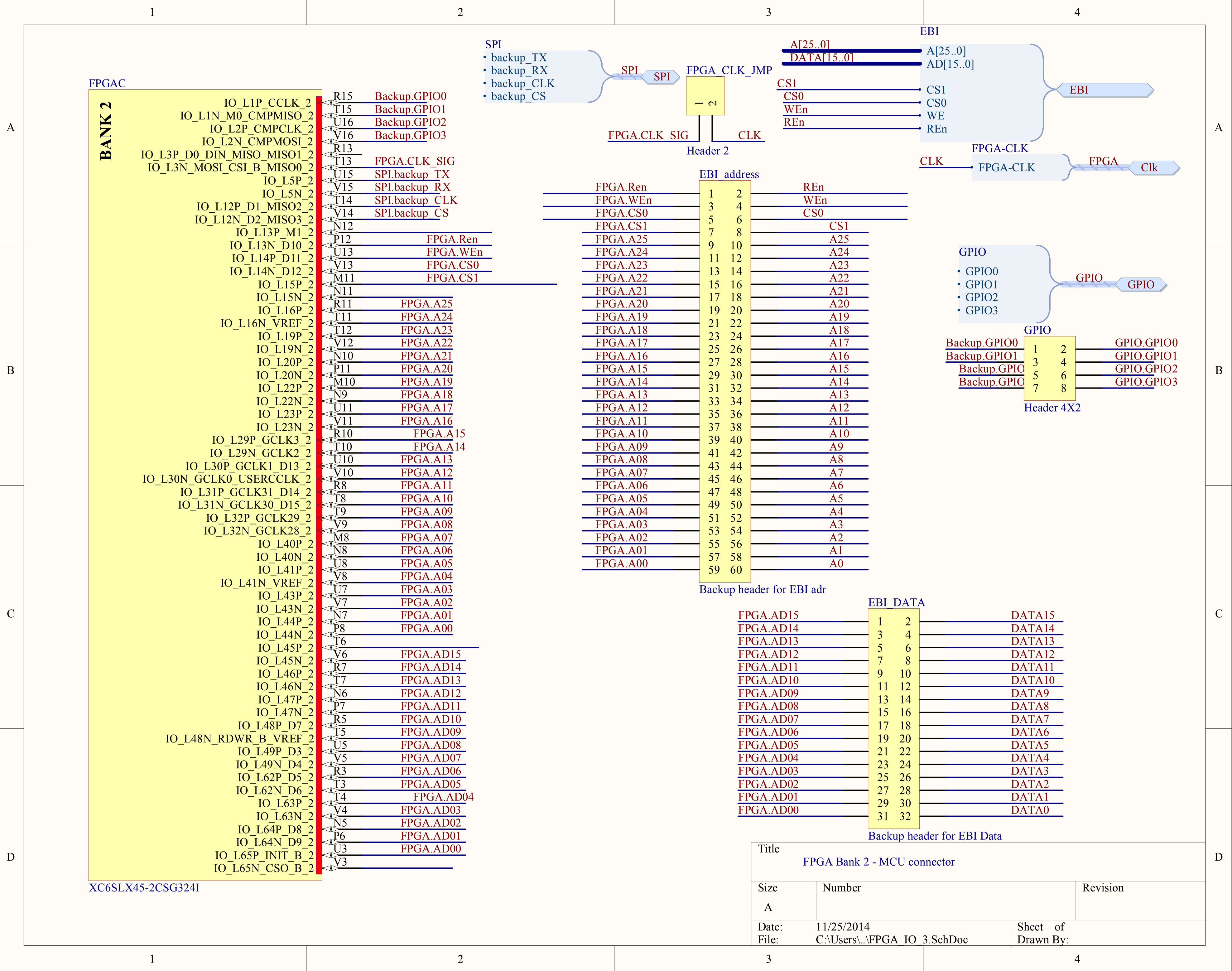
BANK 1

Pin name	Pin	Net
IO_L1P_A25_1	F15	DATA0.ADDR0
IO_L1N_A24_VREF_1	F16	DATA0.ADDR1
IO_L29P_A23_M1A13_1	C17	DATA0.ADDR2
IO_L29N_A22_M1A14_1	C18	DATA0.ADDR3
IO_L30P_A21_M1RESET_1	F14	DATA0.ADDR4
IO_L30N_A20_M1A11_1	G14	DATA0.ADDR5
IO_L31P_A19_M1CKE_1	D17	DATA0.ADDR6
IO_L31N_A18_M1A12_1	D18	DATA0.ADDR7
IO_L32P_A17_M1A8_1	H12	DATA0.ADDR8
IO_L32N_A16_M1A9_1	G13	DATA0.ADDR9
IO_L33P_A15_M1A10_1	E16	DATA0.ADDR10
IO_L33N_A14_M1A4_1	E18	DATA0.ADDR11
IO_L34P_A13_M1WE_1	K12	DATA0.ADDR12
IO_L34N_A12_M1BA2_1	K13	DATA0.ADDR13
IO_L35P_A11_M1A7_1	F17	DATA0.ADDR14
IO_L35N_A10_M1A2_1	F18	DATA0.ADDR15
IO_L36P_A9_M1BA0_1	H13	DATA0.ADDR16
IO_L36N_A8_M1BA1_1	H14	DATA0.ADDR17
IO_L37P_A7_M1A0_1	H15	DATA0.ADDR18
IO_L37N_A6_M1A1_1	H16	DATA0.BUS0
IO_L38P_A5_M1CLK_1	G16	DATA0.BUS1
IO_L38N_A4_M1CLKN_1	G18	DATA0.BUS2
IO_L39P_M1A3_1	J13	DATA0.BUS3
IO_L39N_M1ODT_1	K14	DATA0.BUS4
IO_L40P_GCLK11_M1A5_1	L12	DATA0.BUS5
IO_L40N_GCLK10_M1A6_1	L13	DATA0.BUS6
IO_L41P_GCLK9_IRDY1_M1RASN_1	K15	DATA0.BUS7
IO_L41N_GCLK8_M1CASN_1	K16	DATA0.BUS8
IO_L42P_GCLK7_M1UDM_1	L15	DATA0.BUS9
IO_L42N_GCLK6_TRDY1_M1LDM_1	L16	DATA0.BUS10
IO_L43P_GCLK5_M1DQ4_1	H17	DATA0.BUS11
IO_L43N_GCLK4_M1DQ5_1	H18	DATA0.BUS12
IO_L44P_A3_M1DQ6_1	J16	DATA0.BUS13
IO_L44N_A2_M1DQ7_1	J18	DATA0.BUS14
IO_L45P_A1_M1LDQS_1	K17	DATA0.BUS15
IO_L45N_A0_M1LDQSN_1	K18	DATA0.UB
IO_L46P_FCS_B_M1DQ2_1	L17	DATA0.LB
IO_L46N_FOE_B_M1DQ3_1	L18	DATA0.WRITE
IO_L47P_FWE_B_M1DQ0_1	M16	DATA0.ENABLE
IO_L47N_LDC_M1DQ1_1	M18	
IO_L48P_HDC_M1DQ8_1	N17	
IO_L48N_M1DQ9_1	N18	
IO_L49P_M1DQ10_1	P17	FPGA_LEDS.LED_1
IO_L49N_M1DQ11_1	P18	FPGA_LEDS.LED_2
IO_L50P_M1UDQS_1	N15	
IO_L50N_M1UDQSN_1	N16	
IO_L51P_M1DQ12_1	T17	
IO_L51N_M1DQ13_1	T18	
IO_L52P_M1DQ14_1	U17	
IO_L52N_M1DQ15_1	U18	
IO_L53P_1	M14	
IO_L53N_VREF_1	N14	
IO_L61P_1	L14	
IO_L61N_1	M13	
IO_L74P_AWAKE_1	P15	
IO_L74N_DOUT_BUSY_1	P16	

XC6SLX45-2CSG324I



Title	FPGA Bank 1 - Memory interface	
Size: A	Number	Revision: v.1
Date: 11/25/2014		Sheet of
File: C:\Users\..\FPGA_IO_2.SchDoc		Drawn By:



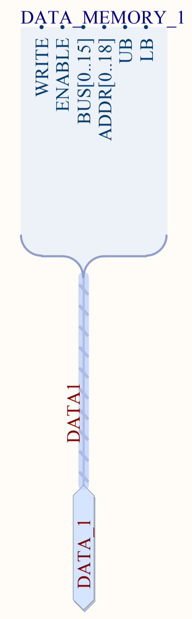
FPGAC
BANK 2
XC6SLX45-2CSG324I
IO_L1P_CCLK_2
IO_L1N_M0_CMPMISO_2
IO_L2P_CMPCLK_2
IO_L2N_CMPMOSI_2
IO_L3P_D0_DIN_MISO_MISO1_2
IO_L3N_MOSI_CSI_B_MISO0_2
IO_L5P_2
IO_L5N_2
IO_L12P_D1_MISO2_2
IO_L12N_D2_MISO3_2
IO_L13P_M1_2
IO_L13N_D10_2
IO_L14P_D11_2
IO_L14N_D12_2
IO_L15P_2
IO_L15N_2
IO_L16P_2
IO_L16N_VREF_2
IO_L19P_2
IO_L19N_2
IO_L20P_2
IO_L20N_2
IO_L22P_2
IO_L22N_2
IO_L23P_2
IO_L23N_2
IO_L29P_GCLK3_2
IO_L29N_GCLK2_2
IO_L30P_GCLK1_D13_2
IO_L30N_GCLK0_USERCCLK_2
IO_L31P_GCLK31_D14_2
IO_L31N_GCLK30_D15_2
IO_L32P_GCLK29_2
IO_L32N_GCLK28_2
IO_L40P_2
IO_L40N_2
IO_L41P_2
IO_L41N_VREF_2
IO_L43P_2
IO_L43N_2
IO_L44P_2
IO_L44N_2
IO_L45P_2
IO_L45N_2
IO_L46P_2
IO_L46N_2
IO_L47P_2
IO_L47N_2
IO_L48P_D7_2
IO_L48N_RDWR_B_VREF_2
IO_L49P_D3_2
IO_L49N_D4_2
IO_L62P_D5_2
IO_L62N_D6_2
IO_L63P_2
IO_L63N_2
IO_L64P_D8_2
IO_L64N_D9_2
IO_L65P_INIT_B_2
IO_L65N_CSO_B_2
R15 Backup.GPIO0
T15 Backup.GPIO1
U16 Backup.GPIO2
V16 Backup.GPIO3
R13
T13 FPGA.CLK_SIG
U15 SPI.backup_TX
V15 SPI.backup_RX
T14 SPI.backup_CLK
V14 SPI.backup_CS
N12
P12 FPGA.Ren
U13 FPGA.WEn
V13 FPGA.CS0
M11 FPGA.CS1
N11
R11 FPGA.A25
T11 FPGA.A24
T12 FPGA.A23
V12 FPGA.A22
N10 FPGA.A21
P11 FPGA.A20
M10 FPGA.A19
N9 FPGA.A18
U11 FPGA.A17
V11 FPGA.A16
R10 FPGA.A15
T10 FPGA.A14
U10 FPGA.A13
V10 FPGA.A12
R8 FPGA.A11
T8 FPGA.A10
T9 FPGA.A09
V9 FPGA.A08
M8 FPGA.A07
N8 FPGA.A06
U8 FPGA.A05
V8 FPGA.A04
U7 FPGA.A03
V7 FPGA.A02
N7 FPGA.A01
P8 FPGA.A00
T6
V6 FPGA.AD15
R7 FPGA.AD14
T7 FPGA.AD13
N6 FPGA.AD12
P7 FPGA.AD11
R5 FPGA.AD10
T5 FPGA.AD09
U5 FPGA.AD08
V5 FPGA.AD07
R3 FPGA.AD06
T3 FPGA.AD05
T4 FPGA.AD04
V4 FPGA.AD03
N5 FPGA.AD02
P6 FPGA.AD01
U3 FPGA.AD00
V3
SPI
backup_TX
backup_RX
backup_CLK
backup_CS
SPI
SPI
FPGA_CLK_JMP
1
2
FPGA.CLK_SIG
CLK
Header 2
A[25..0]
DATA[15..0]
CS1
CS0
WEn
REn
EBI
A[25..0]
AD[15..0]
CS1
CS0
WE
REn
EBI
CLK
FPGA-CLK
FPGA-CLK
FPGA
Clk
EBI_address
FPGA.Ren 1 2 REn
FPGA.WEn 3 4 WEn
FPGA.CS0 5 6 CS0
FPGA.CS1 7 8 CS1
FPGA.A25 9 10 A25
FPGA.A24 11 12 A24
FPGA.A23 13 14 A23
FPGA.A22 15 16 A22
FPGA.A21 17 18 A21
FPGA.A20 19 20 A20
FPGA.A19 21 22 A19
FPGA.A18 23 24 A18
FPGA.A17 25 26 A17
FPGA.A16 27 28 A16
FPGA.A15 29 30 A15
FPGA.A14 31 32 A14
FPGA.A13 33 34 A13
FPGA.A12 35 36 A12
FPGA.A11 37 38 A11
FPGA.A10 39 40 A10
FPGA.A09 41 42 A9
FPGA.A08 43 44 A8
FPGA.A07 45 46 A7
FPGA.A06 47 48 A6
FPGA.A05 49 50 A5
FPGA.A04 51 52 A4
FPGA.A03 53 54 A3
FPGA.A02 55 56 A2
FPGA.A01 57 58 A1
FPGA.A00 59 60 A0
Backup header for EBI adr
GPIO
GPIO0
GPIO1
GPIO2
GPIO3
GPIO
GPIO
Backup.GPIO0 1 2 GPIO.GPIO0
Backup.GPIO1 3 4 GPIO.GPIO1
Backup.GPIO 5 6 GPIO.GPIO2
Backup.GPIO 7 8 GPIO.GPIO3
Header 4X2
EBI_DATA
FPGA.AD15 1 2 DATA15
FPGA.AD14 3 4 DATA14
FPGA.AD13 5 6 DATA13
FPGA.AD12 7 8 DATA12
FPGA.AD11 9 10 DATA11
FPGA.AD10 11 12 DATA10
FPGA.AD09 13 14 DATA9
FPGA.AD08 15 16 DATA8
FPGA.AD07 17 18 DATA7
FPGA.AD06 19 20 DATA6
FPGA.AD05 21 22 DATA5
FPGA.AD04 23 24 DATA4
FPGA.AD03 25 26 DATA3
FPGA.AD02 27 28 DATA2
FPGA.AD01 29 30 DATA1
FPGA.AD00 31 32 DATA0
Backup header for EBI Data
Title
FPGA Bank 2 - MCU connector
Size
A
Number
Revision
Date: 11/25/2014
File: C:\Users\..\FPGA_IO_3.SchDoc
Sheet of
Drawn By:

FPGAD

BANK 3

Pin name	Pin	Net
IO_L1P_3	N4	DATA1.ADDR0
IO_L1N_VREF_3	N3	DATA1.ADDR1
IO_L2P_3	P4	DATA1.ADDR2
IO_L2N_3	P3	DATA1.ADDR3
IO_L31P_3	L6	DATA1.ADDR4
IO_L31N_VREF_3	M5	DATA1.ADDR5
IO_L32P_M3DQ14_3	U2	DATA1.ADDR6
IO_L32N_M3DQ15_3	U1	DATA1.ADDR7
IO_L33P_M3DQ12_3	T2	DATA1.ADDR8
IO_L33N_M3DQ13_3	T1	DATA1.ADDR9
IO_L34P_M3UDQS_3	P2	DATA1.ADDR10
IO_L34N_M3UDQSN_3	P1	DATA1.ADDR11
IO_L35P_M3DQ10_3	N2	DATA1.ADDR12
IO_L35N_M3DQ11_3	N1	DATA1.ADDR13
IO_L36P_M3DQ8_3	M3	DATA1.ADDR14
IO_L36N_M3DQ9_3	M1	DATA1.ADDR15
IO_L37P_M3DQ0_3	L2	DATA1.ADDR16
IO_L37N_M3DQ1_3	L1	DATA1.ADDR17
IO_L38P_M3DQ2_3	K2	DATA1.ADDR18
IO_L38N_M3DQ3_3	K1	DATA1.BUS0
IO_L39P_M3LDQS_3	L4	DATA1.BUS1
IO_L39N_M3LDQSN_3	L3	DATA1.BUS2
IO_L40P_M3DQ6_3	J3	DATA1.BUS3
IO_L40N_M3DQ7_3	J1	DATA1.BUS4
IO_L41P_GCLK27_M3DQ4_3	H2	DATA1.BUS5
IO_L41N_GCLK26_M3DQ5_3	H1	DATA1.BUS6
IO_L42P_GCLK25_TRDY2_M3UDM_3	K4	DATA1.BUS7
IO_L42N_GCLK24_M3LDM_3	K3	DATA1.BUS8
IO_L43P_GCLK23_M3RASN_3	L5	DATA1.BUS9
IO_L43N_GCLK22_IRDY2_M3CASN_3	K5	DATA1.BUS10
IO_L44P_GCLK21_M3A5_3	H4	DATA1.BUS11
IO_L44N_GCLK20_M3A6_3	H3	DATA1.BUS12
IO_L45P_M3A3_3	L7	DATA1.BUS13
IO_L45N_M3ODT_3	K6	DATA1.BUS14
IO_L46P_M3CLK_3	G3	DATA1.BUS15
IO_L46N_M3CLKN_3	G1	DATA1.UB
IO_L47P_M3A0_3	J7	DATA1.LB
IO_L47N_M3A1_3	J6	DATA1.WRITE
IO_L48P_M3BA0_3	F2	DATA1.ENABLE
IO_L48N_M3BA1_3	F1	
IO_L49P_M3A7_3	H6	
IO_L49N_M3A2_3	H5	
IO_L50P_M3WE_3	E3	
IO_L50N_M3BA2_3	E1	
IO_L51P_M3A10_3	F4	
IO_L51N_M3A4_3	F3	
IO_L52P_M3A8_3	D2	
IO_L52N_M3A9_3	D1	
IO_L53P_M3CKE_3	H7	
IO_L53N_M3A12_3	G6	
IO_L54P_M3RESET_3	E4	
IO_L54N_M3A11_3	D3	
IO_L55P_M3A13_3	F6	
IO_L55N_M3A14_3	F5	
IO_L83P_3	C2	
IO_L83N_VREF_3	C1	

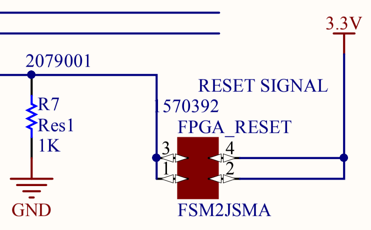
XC6SLX45-2CSG324I



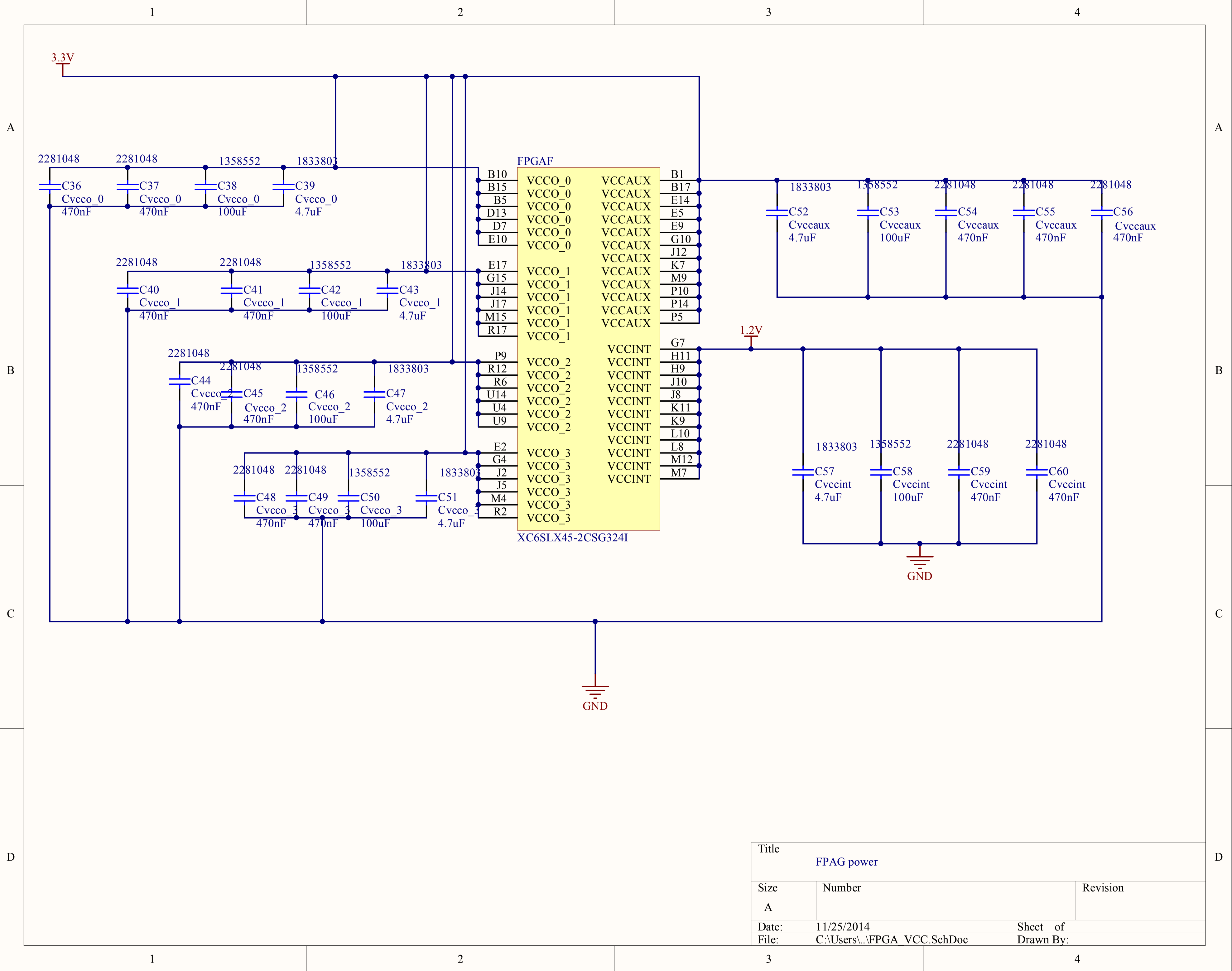
2079001

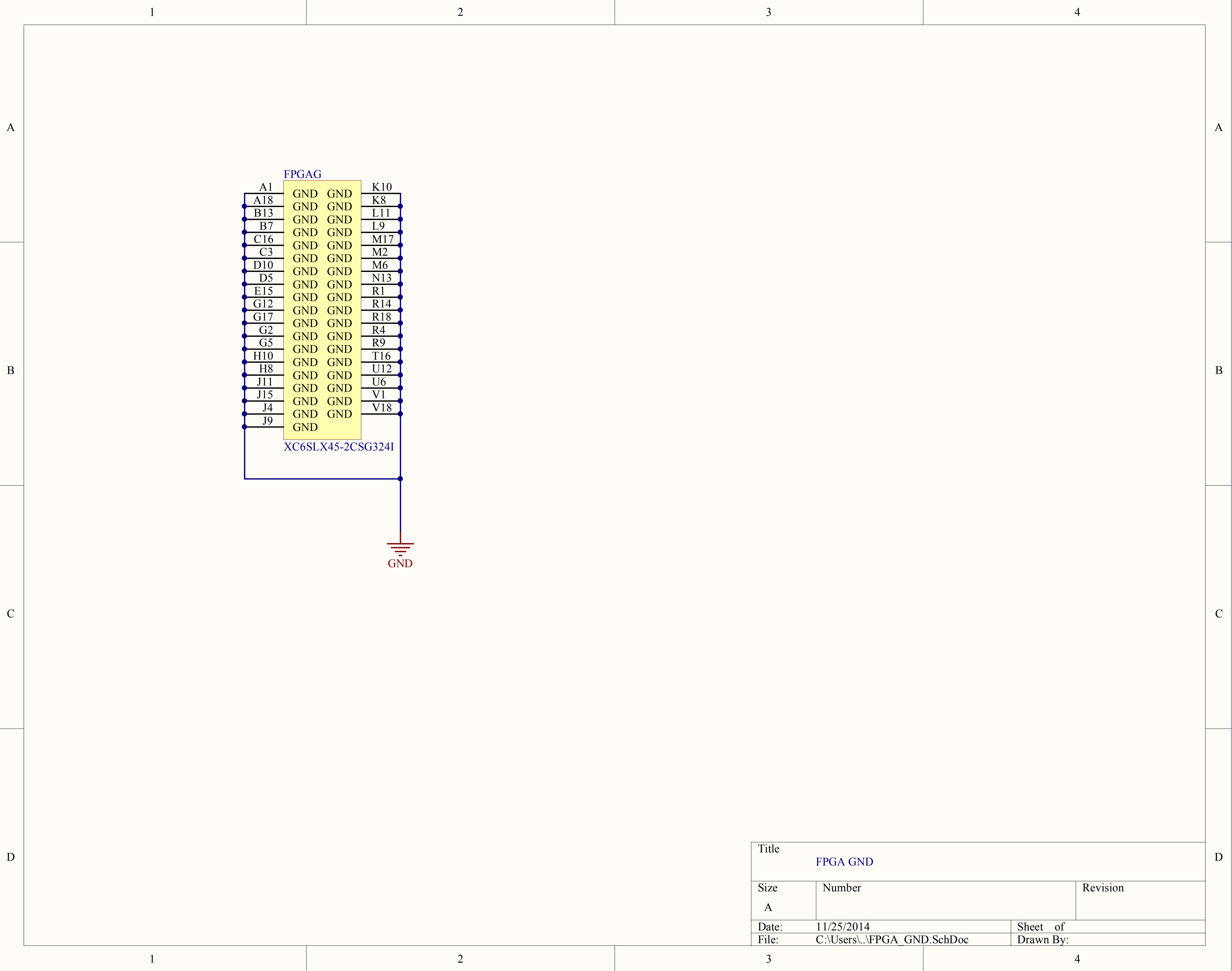
R7
Res1
1K

GND

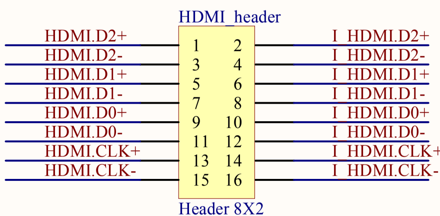


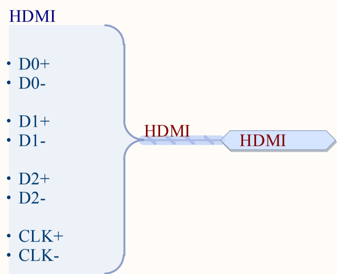
Title	FPGA Bank 3 - Memory interface		
Size	Number		Revision
A			
Date:	11/25/2014	Sheet of	
File:	C:\Users\..\FPGA_IO_4.SchDoc	Drawn By:	

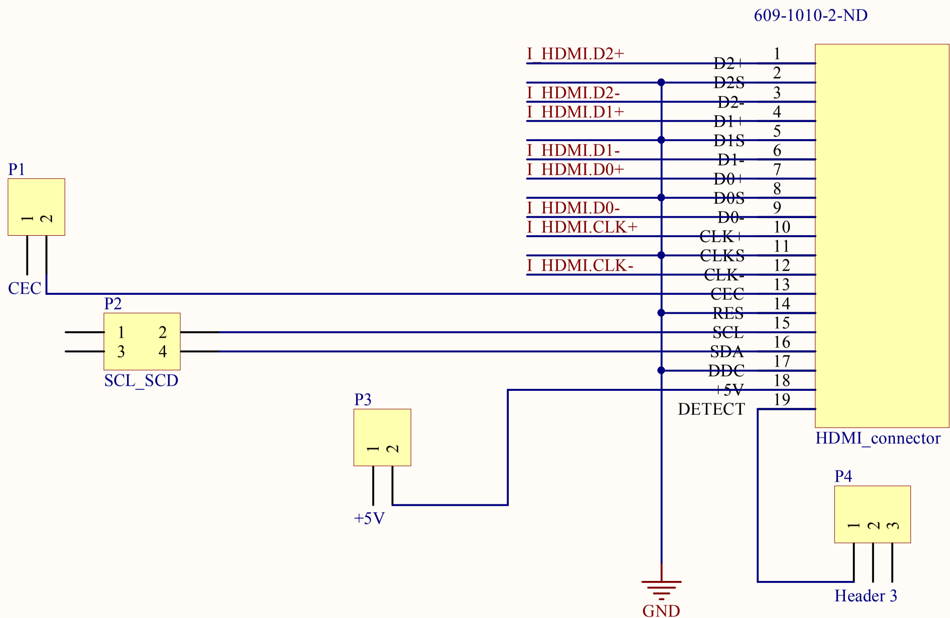

3.3V
1.2V
GND
FPGAF
XC6SLX45-2CSG324I
C36 Cvcco_0 470nF 2281048
C37 Cvcco_0 470nF 2281048
C38 Cvcco_0 100uF 1358552
C39 Cvcco_0 4.7uF 1833803
C40 Cvcco_1 470nF 2281048
C41 Cvcco_1 470nF 2281048
C42 Cvcco_1 100uF 1358552
C43 Cvcco_1 4.7uF 1833803
C44 Cvcco_2 470nF 2281048
C45 Cvcco_2 470nF 2281048
C46 Cvcco_2 100uF 1358552
C47 Cvcco_2 4.7uF 1833803
C48 Cvcco_3 470nF 2281048
C49 Cvcco_3 470nF 2281048
C50 Cvcco_3 100uF 1358552
C51 Cvcco_3 4.7uF 1833803
C52 Cvccaux 4.7uF 1833803
C53 Cvccaux 100uF 1358552
C54 Cvccaux 470nF 2281048
C55 Cvccaux 470nF 2281048
C56 Cvccaux 470nF 2281048
C57 Cvccint 4.7uF 1833803
C58 Cvccint 100uF 1358552
C59 Cvccint 470nF 2281048
C60 Cvccint 470nF 2281048
B10 B15 B5 D13 D7 E10 VCCO_0
E17 G15 J14 J17 M15 R17 VCCO_1
P9 R12 R6 U14 U4 U9 VCCO_2
E2 G4 J2 J5 M4 R2 VCCO_3
B1 B17 E14 E5 E9 G10 J12 K7 M9 P10 P14 P5 VCCAUX
G7 H11 H9 J10 J8 K11 K9 L10 L8 M12 M7 VCCINT
Title FPAG power
Size A
Number
Revision
Date: 11/25/2014
File: C:\Users\..\FPGA_VCC.SchDoc
Sheet of
Drawn By:



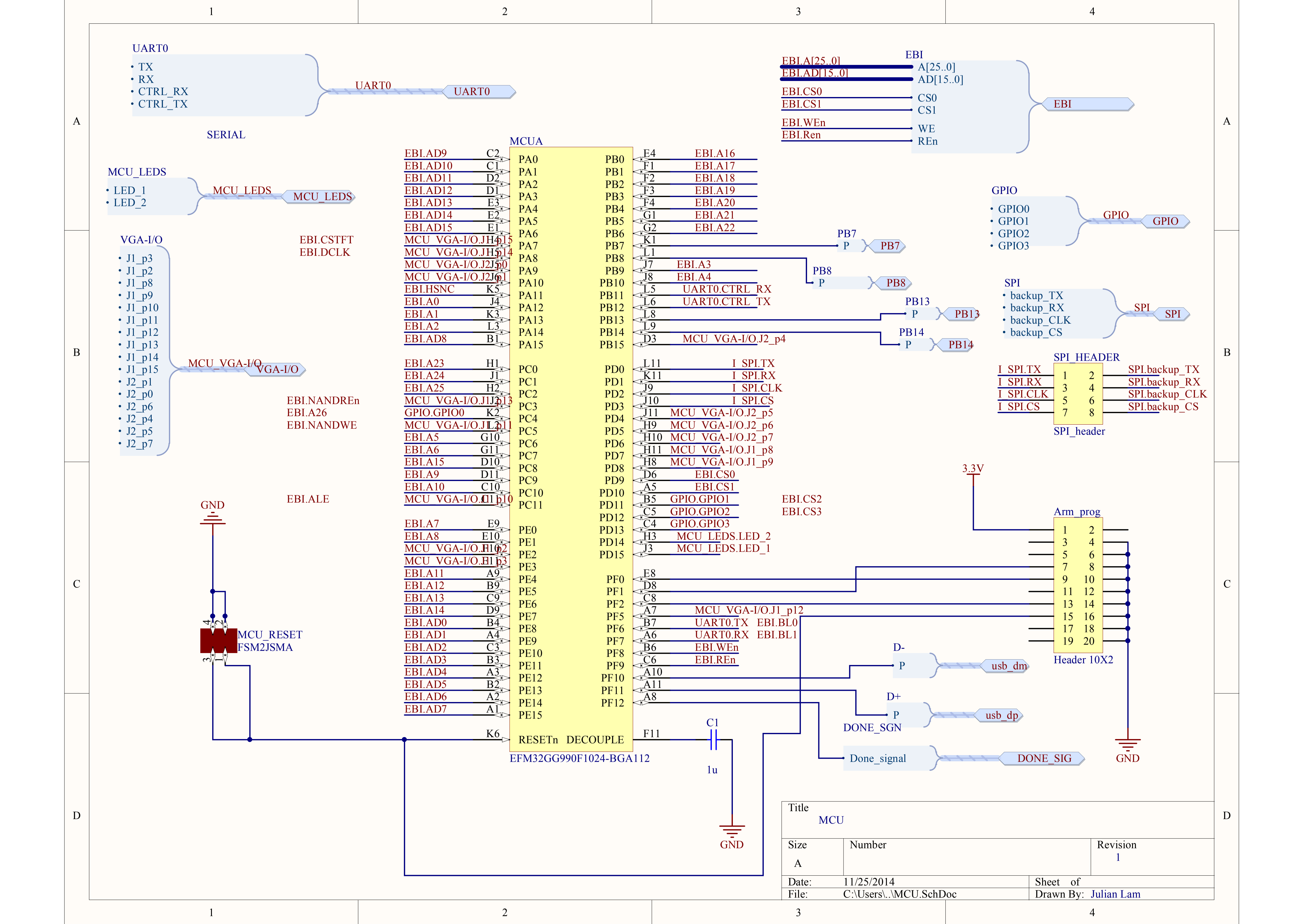
FPGAG
A1
A18
B13
B7
C16
C3
D10
D5
E15
G12
G17
G2
G5
H10
H8
J11
J15
J4
J9
K10
K8
L11
L9
M17
M2
M6
N13
R1
R14
R18
R4
R9
T16
U12
U6
V1
V18
GND
XC6SLX45-2CSG324I
GND
Title
FPGA GND
Size
A
Number
Revision
Date: 11/25/2014
Sheet of
File: C:\Users\..\FPGA_GND.SchDoc
Drawn By:



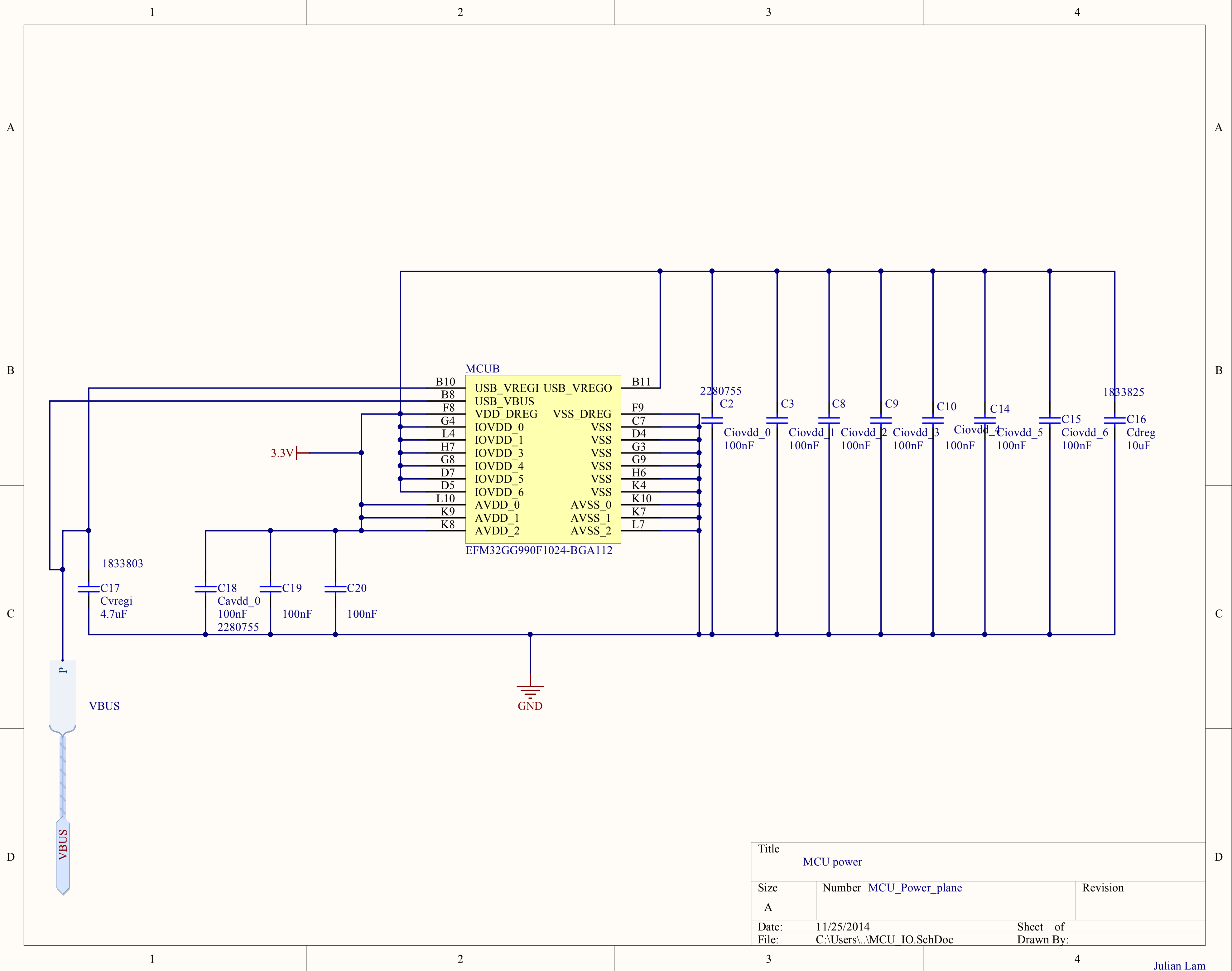




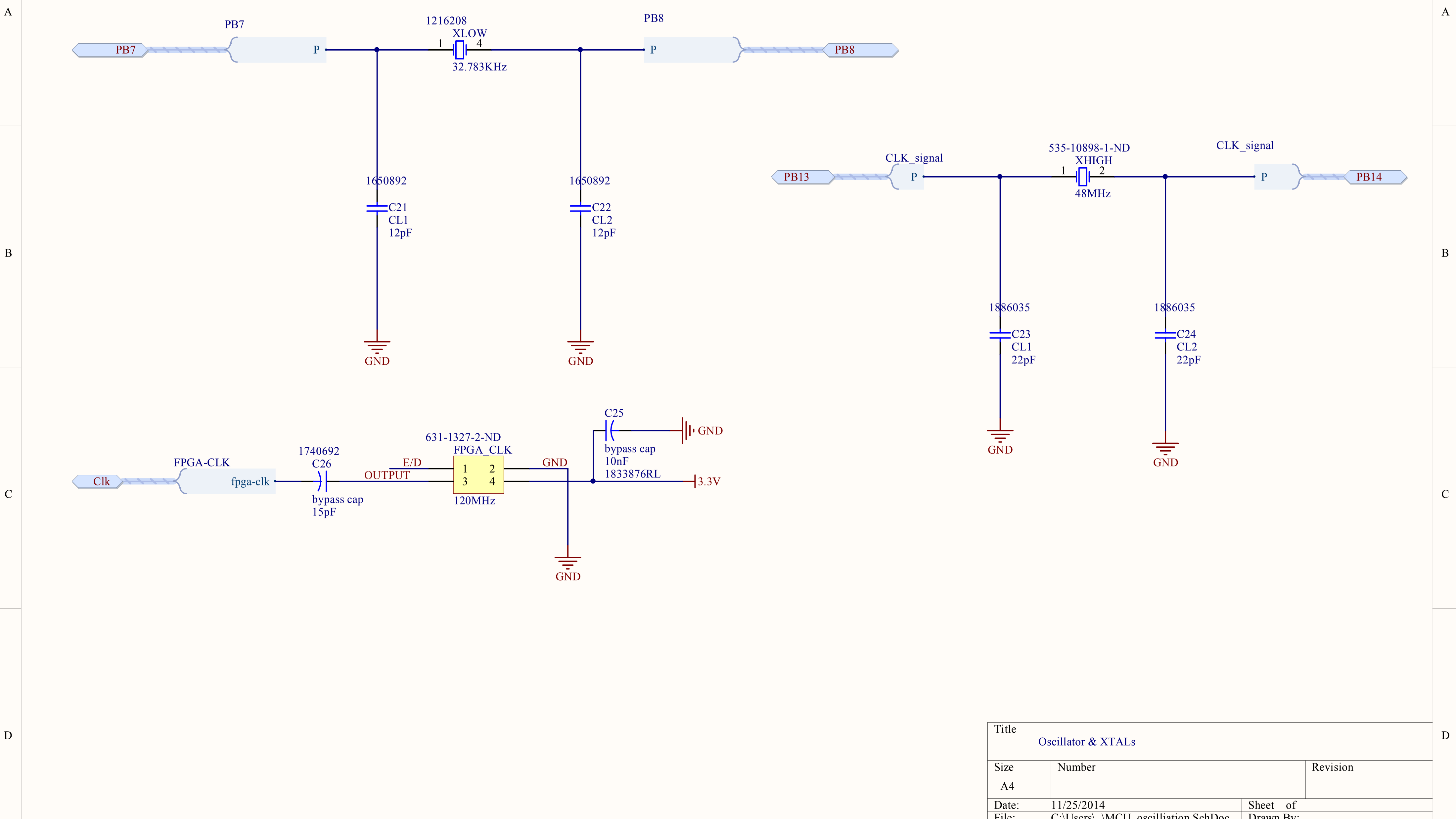
Title		
HDMI connector		
Size	Number	Revision
A4		
Date:	11/25/2014	Sheet of
File:	C:\Users\..\HDMI.SchDoc	Drawn By:

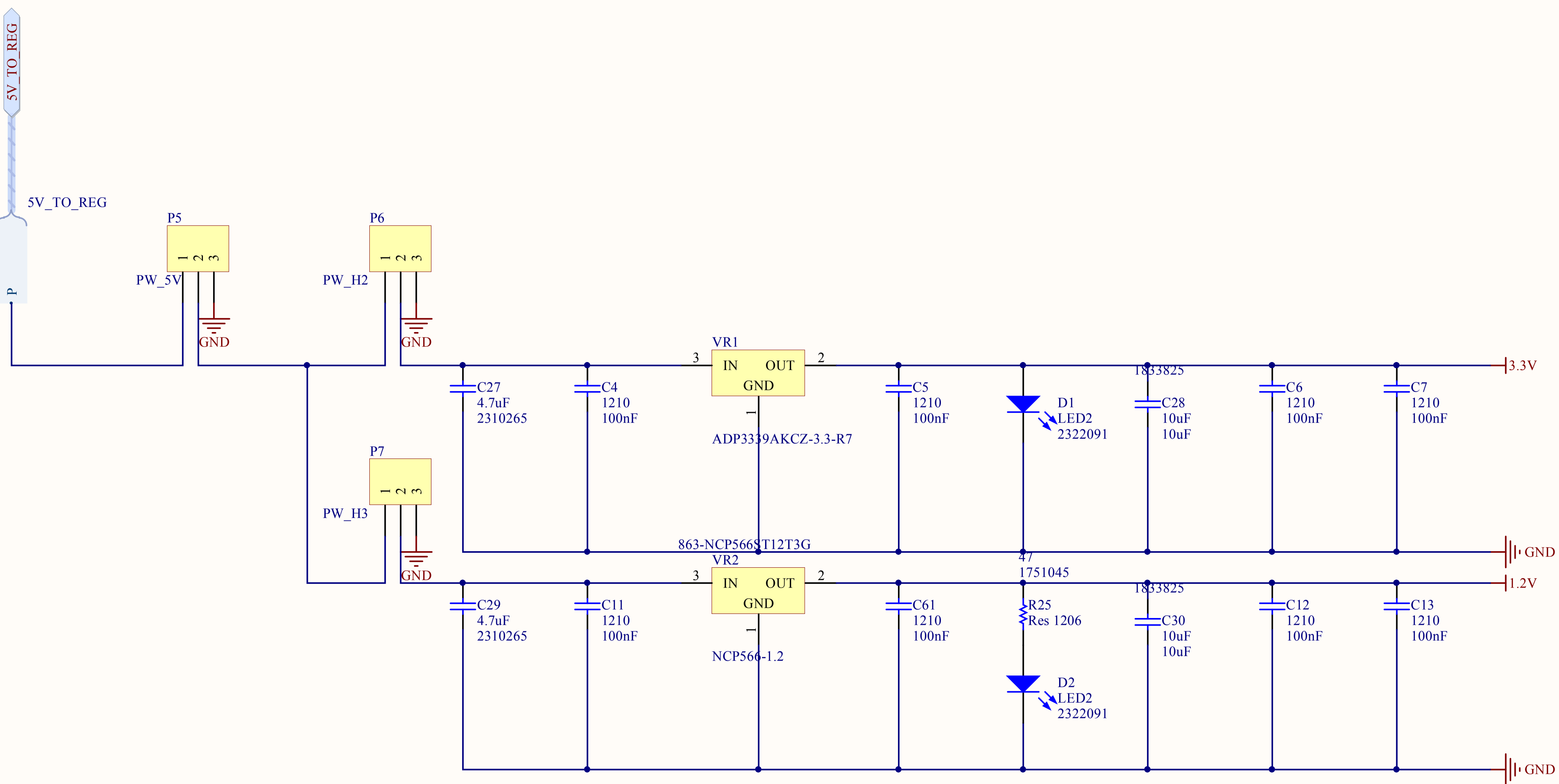


UART0
TX
RX
CTRL_RX
CTRL_TX
SERIAL
MCU_LEDS
LED_1
LED_2
VGA-I/O
J1_p3
J1_p2
J1_p8
J1_p9
J1_p10
J1_p11
J1_p12
J1_p13
J1_p14
J1_p15
J2_p1
J2_p0
J2_p6
J2_p4
J2_p5
J2_p7
MCU_VGA-I/O
EBI.CSTFT
EBI.DCLK
EBI.NANDREn
EBI.A26
EBI.NANDWE
EBI.ALE
GND
MCU_RESET
FSM2JSMA
MCUA
EFM32GG990F1024-BGA112
RESETn
DECOUPLE
C1
1u
EBI
A[25..0]
AD[15..0]
CS0
CS1
WE
REn
GPIO
GPIO0
GPIO1
GPIO2
GPIO3
SPI
backup_TX
backup_RX
backup_CLK
backup_CS
SPI_HEADER
SPI_header
3.3V
Arm_prog
Header 10X2
EBI.CS2
EBI.CS3
D-
D+
usb_dm
usb_dp
DONE_SGN
Done_signal
DONE_SIG
Title
MCU
Size
A
Number
Revision
1
Date:
11/25/2014
File:
C:\Users\..\MCU.SchDoc
Sheet of
Drawn By:
Julian Lam



MCUB
EFM32GG990F1024-BGA112
USB_VREGI
USB_VBUS
VDD_DREG
IOVDD_0
IOVDD_1
IOVDD_3
IOVDD_4
IOVDD_5
IOVDD_6
AVDD_0
AVDD_1
AVDD_2
USB_VREGO
VSS_DREG
VSS
AVSS_0
AVSS_1
AVSS_2
B10 B8 F8 G4 L4 H7 G8 D7 D5 L10 K9 K8
B11 F9 C7 D4 G3 G9 H6 K4 K10 K7 L7
2280755 C2 Ciovdd_0 100nF
C3 Ciovdd_1 100nF
C8 Ciovdd_2 100nF
C9 Ciovdd_3 100nF
C10 Ciovdd_4 100nF
C14 Ciovdd_5 100nF
C15 Ciovdd_6 100nF
1833825 C16 Cdreg 10uF
3.3V
1833803 C17 Cvregi 4.7uF
C18 Cavdd_0 100nF 2280755
C19 100nF
C20 100nF
GND
VBUS
Title
MCU power
Size
A
Number MCU_Power_plane
Revision
Date: 11/25/2014
File: C:\Users\..\MCU_IO.SchDoc
Sheet of
Drawn By:
Julian Lam


PB7
P
PB8
1216208
XLOW
32.783KHz
1650892
C21
CL1
12pF
C22
CL2
GND
PB13
CLK_signal
535-10898-1-ND
XHIGH
48MHz
PB14
1886035
C23
22pF
C24
Clk
FPGA-CLK
fpga-clk
1740692
C26
bypass cap
15pF
E/D
OUTPUT
631-1327-2-ND
FPGA_CLK
120MHz
C25
10nF
1833876RL
3.3V
Title
Oscillator & XTALs
Size
A4
Number
Revision
Date:
11/25/2014
Sheet of
File:
C:\Users\..\MCU_oscilliation.SchDoc
Drawn By:

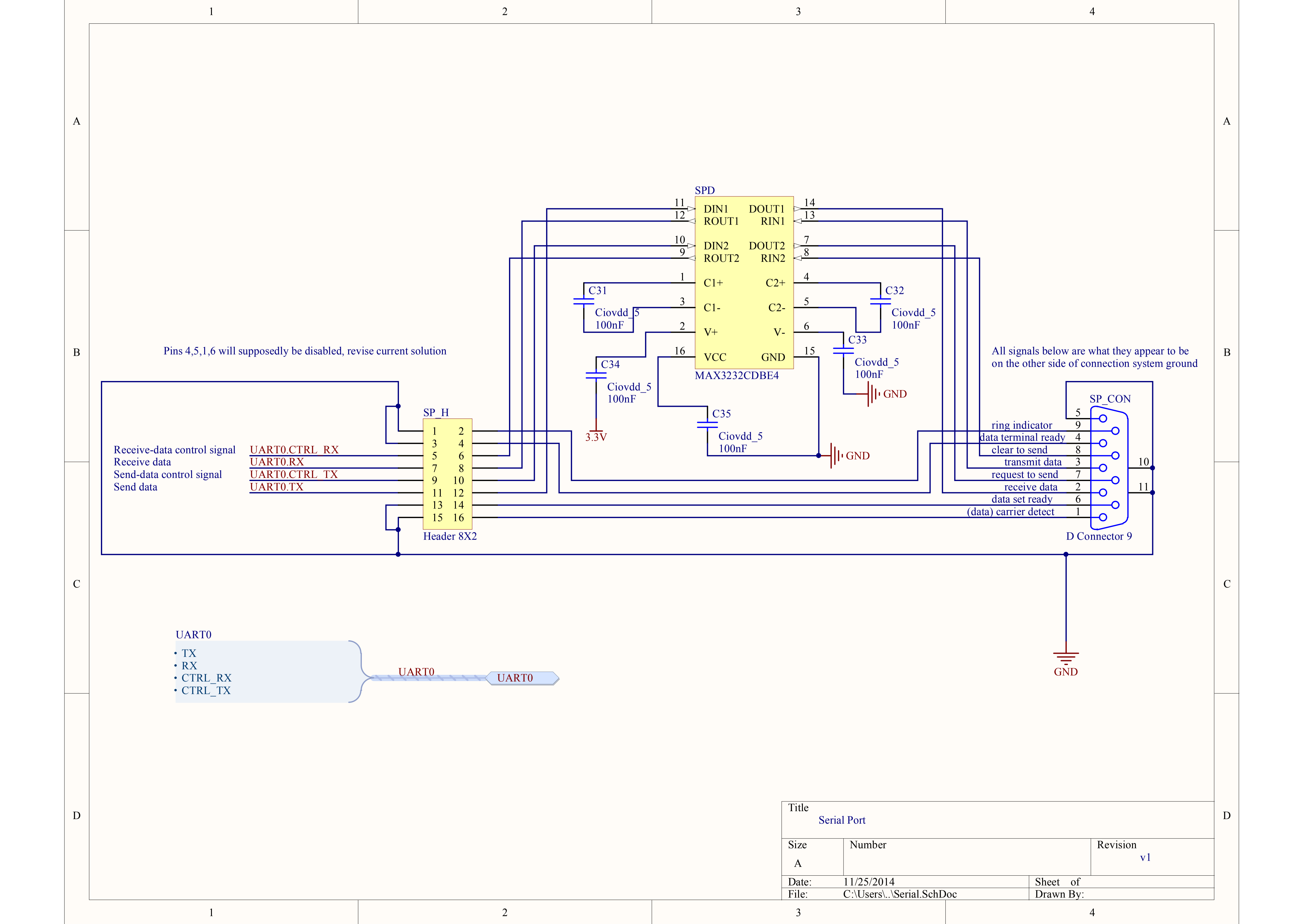


Title: Power supply

Size	Number	Revision
A4		

Date: 11/25/2014 — Sheet of

File: C:\Users\..\Power_supply.SchDoc — Drawn By:



Pins 4,5,1,6 will supposedly be disabled, revise current solution
All signals below are what they appear to be
on the other side of connection system ground
SPD
DIN1
ROUT1
DIN2
ROUT2
C1+
C1-
V+
VCC
DOUT1
RIN1
DOUT2
RIN2
C2+
C2-
V-
GND
MAX3232CDBE4
C31
C32
C33
C34
C35
Ciovdd_5
100nF
3.3V
GND
SP_H
Header 8X2
Receive-data control signal
Receive data
Send-data control signal
Send data
UART0.CTRL_RX
UART0.RX
UART0.CTRL_TX
UART0.TX
SP_CON
ring indicator
data terminal ready
clear to send
transmit data
request to send
receive data
data set ready
(data) carrier detect
D Connector 9
UART0
TX
RX
CTRL_RX
CTRL_TX
Title
Serial Port
Size
A
Number
Revision
v1
Date:
11/25/2014
Sheet of
File:
C:\Users\..\Serial.SchDoc
Drawn By:

MEMORY
- WRITE
- ENABLE
- BUS[0..15]
- ADDR[0..18]
- UB
- LB

MEMORY — MEMORY

MEM_BUS_JUMPER (Header 16X2)

Net	Pin	Pin	Net
I MEMORY.BUS0	1	2	MEMORY.BUS0
I MEMORY.BUS1	3	4	MEMORY.BUS1
I MEMORY.BUS2	5	6	MEMORY.BUS2
I MEMORY.BUS3	7	8	MEMORY.BUS3
I MEMORY.BUS4	9	10	MEMORY.BUS4
I MEMORY.BUS5	11	12	MEMORY.BUS5
I MEMORY.BUS6	13	14	MEMORY.BUS6
I MEMORY.BUS7	15	16	MEMORY.BUS7
I MEMORY.BUS8	17	18	MEMORY.BUS8
I MEMORY.BUS9	19	20	MEMORY.BUS9
I MEMORY.BUS10	21	22	MEMORY.BUS10
I MEMORY.BUS11	23	24	MEMORY.BUS11
I MEMORY.BUS12	25	26	MEMORY.BUS12
I MEMORY.BUS13	27	28	MEMORY.BUS13
I MEMORY.BUS14	29	30	MEMORY.BUS14
I MEMORY.BUS15	31	32	MEMORY.BUS15

MEM_ADDR_JUMPER (Header 19X2)

Net	Pin	Pin	Net
I MEMORY.ADDR0	1	2	MEMORY.ADDR0
I MEMORY.ADDR1	3	4	MEMORY.ADDR1
I MEMORY.ADDR2	5	6	MEMORY.ADDR2
I MEMORY.ADDR3	7	8	MEMORY.ADDR3
I MEMORY.ADDR4	9	10	MEMORY.ADDR4
I MEMORY.ADDR5	11	12	MEMORY.ADDR5
I MEMORY.ADDR6	13	14	MEMORY.ADDR6
I MEMORY.ADDR7	15	16	MEMORY.ADDR7
I MEMORY.ADDR8	17	18	MEMORY.ADDR8
I MEMORY.ADDR9	19	20	MEMORY.ADDR9
I MEMORY.ADDR10	21	22	MEMORY.ADDR10
I MEMORY.ADDR11	23	24	MEMORY.ADDR11
I MEMORY.ADDR12	25	26	MEMORY.ADDR12
I MEMORY.ADDR13	27	28	MEMORY.ADDR13
I MEMORY.ADDR14	29	30	MEMORY.ADDR14
I MEMORY.ADDR15	31	32	MEMORY.ADDR15
I MEMORY.ADDR16	33	34	MEMORY.ADDR16
I MEMORY.ADDR17	35	36	MEMORY.ADDR17
I MEMORY.ADDR18	37	38	MEMORY.ADDR18

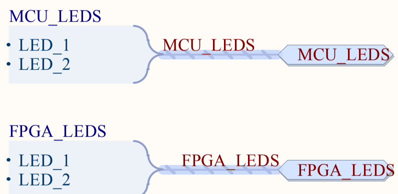
2103743 MEMORY (SRAM-AS7C38098A)

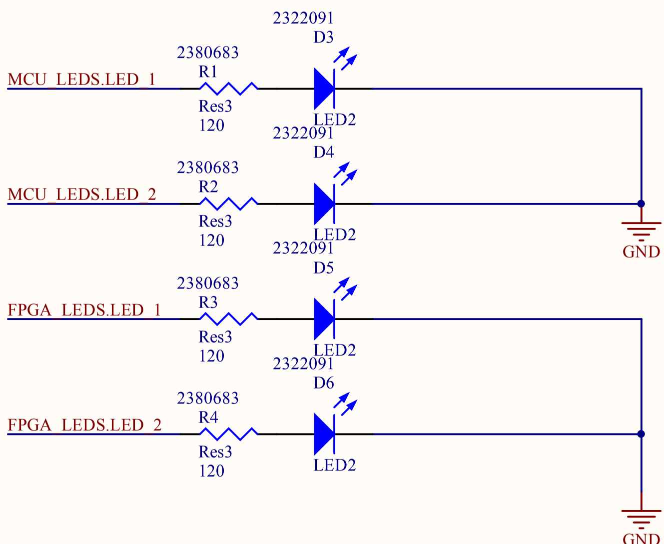
Net	Pin	Name	Name	Pin	Net
			UB#	40	I MEMORY.UB
			LB#	39	I MEMORY.LB
			OE#	41	GND
I MEMORY.BUS15	38	DQ15	WE#	17	I MEMORY.WRITE
I MEMORY.BUS14	37	DQ14	CE#	6	I MEMORY.ENABLE
I MEMORY.BUS13	36	DQ13			
I MEMORY.BUS12	35	DQ12			
I MEMORY.BUS11	32	DQ11	A18	18	I MEMORY.ADDR18
I MEMORY.BUS10	31	DQ10	A17	19	I MEMORY.ADDR17
I MEMORY.BUS9	30	DQ9	A16	20	I MEMORY.ADDR16
I MEMORY.BUS8	29	DQ8	A15	21	I MEMORY.ADDR15
			A14	22	I MEMORY.ADDR14
I MEMORY.BUS7	16	DQ7	A13	23	I MEMORY.ADDR13
I MEMORY.BUS6	15	DQ6	A12	24	I MEMORY.ADDR12
I MEMORY.BUS5	14	DQ5	A11	25	I MEMORY.ADDR11
I MEMORY.BUS4	13	DQ4	A10	26	I MEMORY.ADDR10
I MEMORY.BUS3	10	DQ3	A9	27	I MEMORY.ADDR9
I MEMORY.BUS2	9	DQ2	A8	28	I MEMORY.ADDR8
I MEMORY.BUS1	8	DQ1	A7	42	I MEMORY.ADDR7
I MEMORY.BUS0	7	DQ0	A6	43	I MEMORY.ADDR6
			A5	44	I MEMORY.ADDR5
			A4	1	I MEMORY.ADDR4
GND	12	Vss	A3	2	I MEMORY.ADDR3
GND	34	Vss	A2	3	I MEMORY.ADDR2
3.3V	33	Vcc	A1	4	I MEMORY.ADDR1
3.3V	11	Vcc	A0	5	I MEMORY.ADDR0

MEM_CONTROL_JUMPER (Header 4X2)

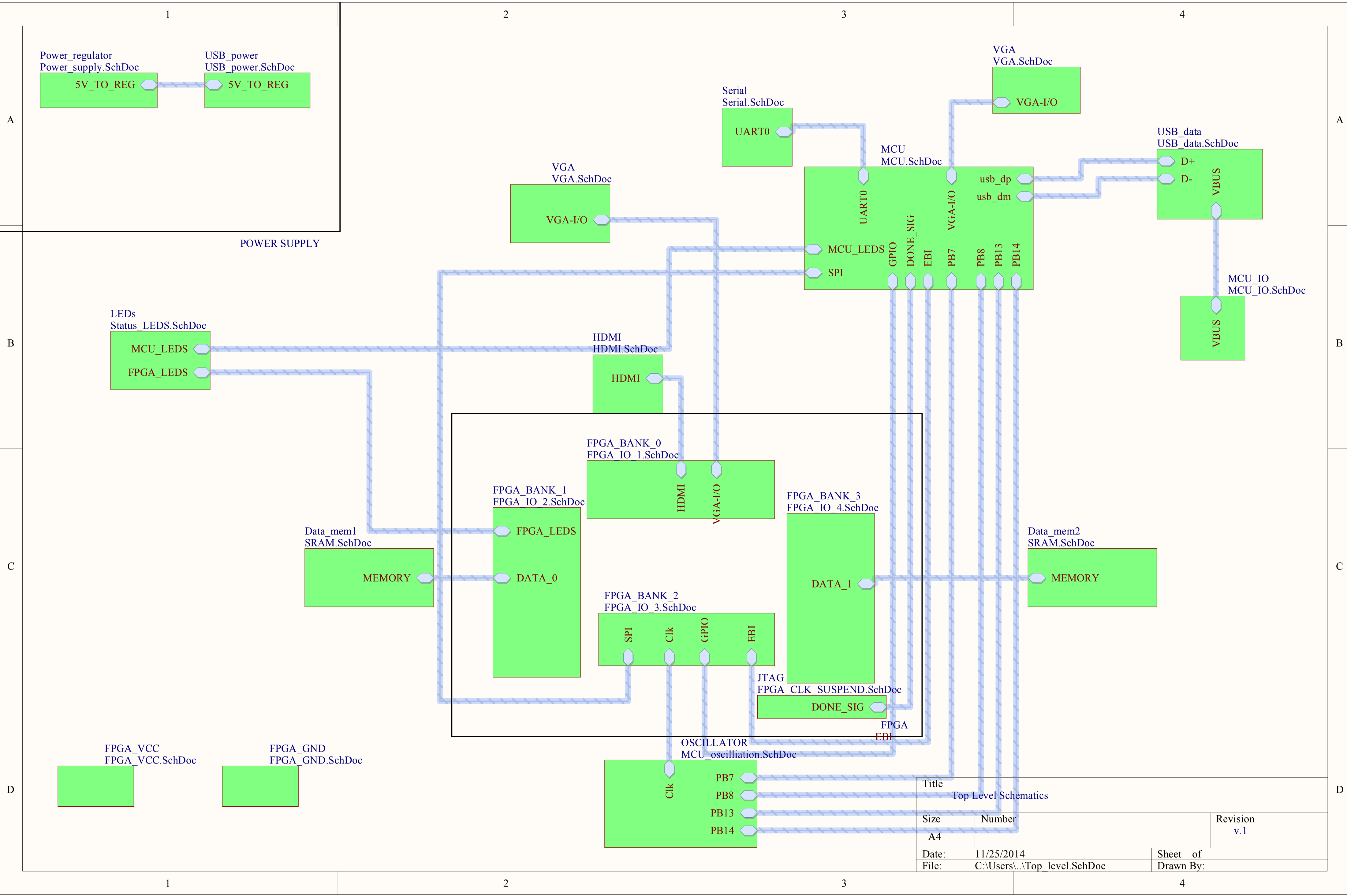
Net	Pin	Pin	Net
I MEMORY.UB	1	2	MEMORY.UB
I MEMORY.LB	3	4	MEMORY.LB
I MEMORY.WRITE	5	6	MEMORY.WRITE
I MEMORY.ENABLE	7	8	MEMORY.ENABLE

Title	Memory	
Size: A	Number	Revision: version 1
Date: 11/25/2014		Sheet of
File: C:\Users\..\SRAM.SchDoc		Drawn By:

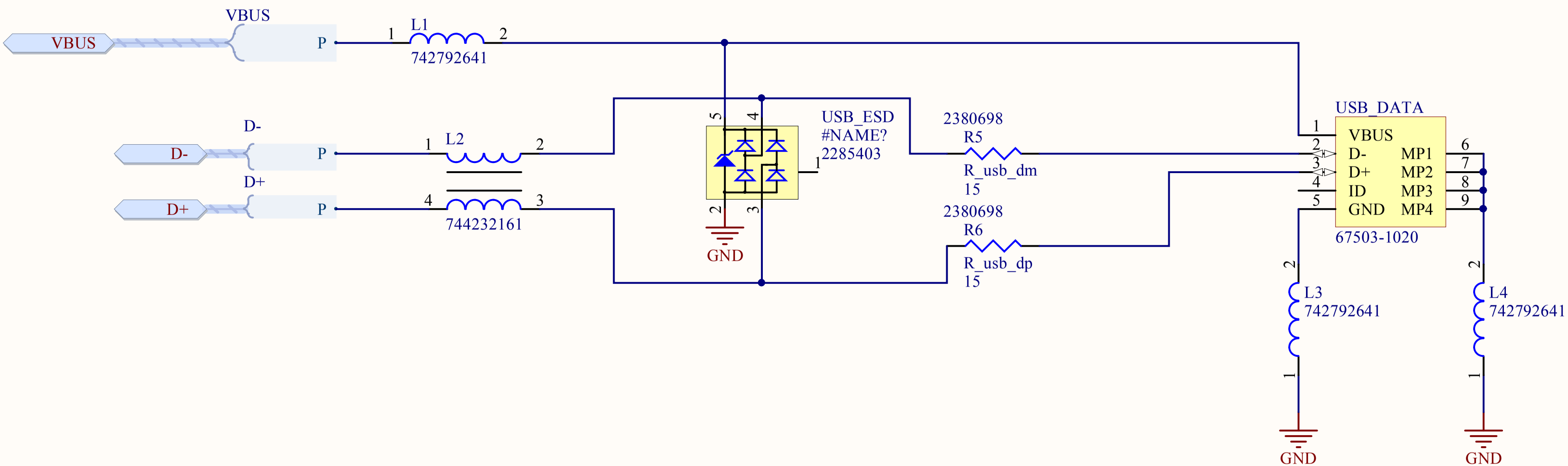




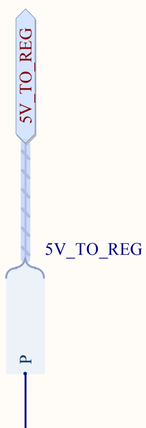
Title	LEDs	
Size: A4	Number	Revision
Date: 11/25/2014		Sheet of
File: C:\Users\..\Status_LEDS.SchDoc		Drawn By:

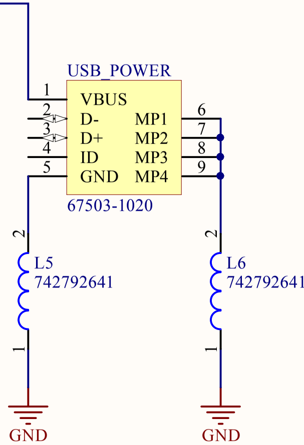


Power_regulator
Power_supply.SchDoc
5V_TO_REG
USB_power
USB_power.SchDoc
5V_TO_REG
POWER SUPPLY
Serial
Serial.SchDoc
UART0
VGA
VGA.SchDoc
VGA-I/O
MCU
MCU.SchDoc
UART0
VGA-I/O
usb_dp
usb_dm
MCU_LEDS
SPI
GPIO
DONE_SIG
EBI
PB7
PB8
PB13
PB14
USB_data
USB_data.SchDoc
D+
D-
VBUS
MCU_IO
MCU_IO.SchDoc
VBUS
VGA
VGA.SchDoc
VGA-I/O
LEDs
Status_LEDS.SchDoc
MCU_LEDS
FPGA_LEDS
HDMI
HDMI.SchDoc
HDMI
FPGA_BANK_0
FPGA_IO_1.SchDoc
HDMI
VGA-I/O
FPGA_BANK_1
FPGA_IO_2.SchDoc
FPGA_LEDS
DATA_0
FPGA_BANK_3
FPGA_IO_4.SchDoc
DATA_1
Data_mem1
SRAM.SchDoc
MEMORY
Data_mem2
SRAM.SchDoc
MEMORY
FPGA_BANK_2
FPGA_IO_3.SchDoc
SPI
Clk
GPIO
EBI
JTAG
FPGA_CLK_SUSPEND.SchDoc
DONE_SIG
FPGA
EBI
OSCILLATOR
MCU_oscilliation.SchDoc
Clk
PB7
PB8
PB13
PB14
FPGA_VCC
FPGA_VCC.SchDoc
FPGA_GND
FPGA_GND.SchDoc
Title
Top Level Schematics
Size
A4
Number
Revision
v.1
Date: 11/25/2014
File: C:\Users\..\Top_level.SchDoc
Sheet of
Drawn By:

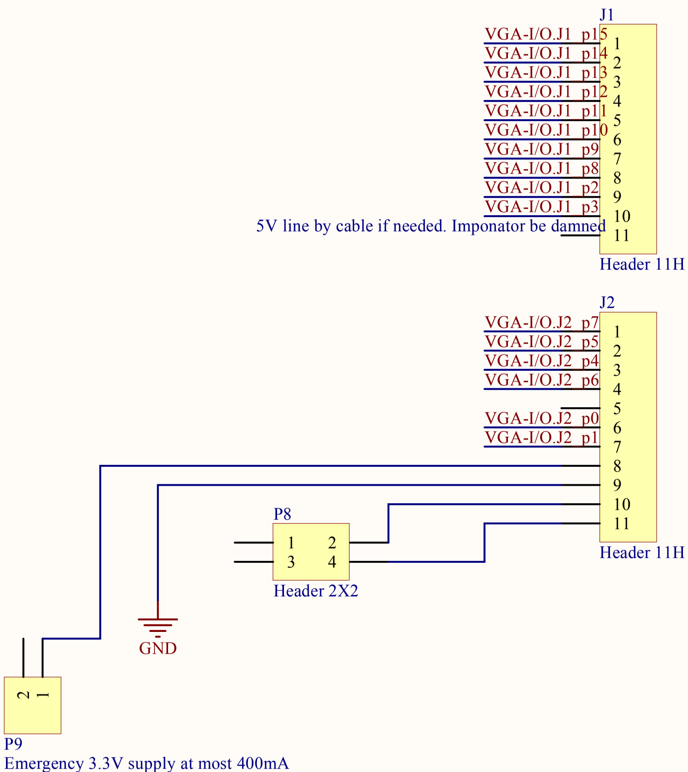


Title		
USB data		
Size	Number	Revision
A4		
Date:	11/25/2014	Sheet of
File:	C:\Users\..\USB_data.SchDoc	Drawn By:

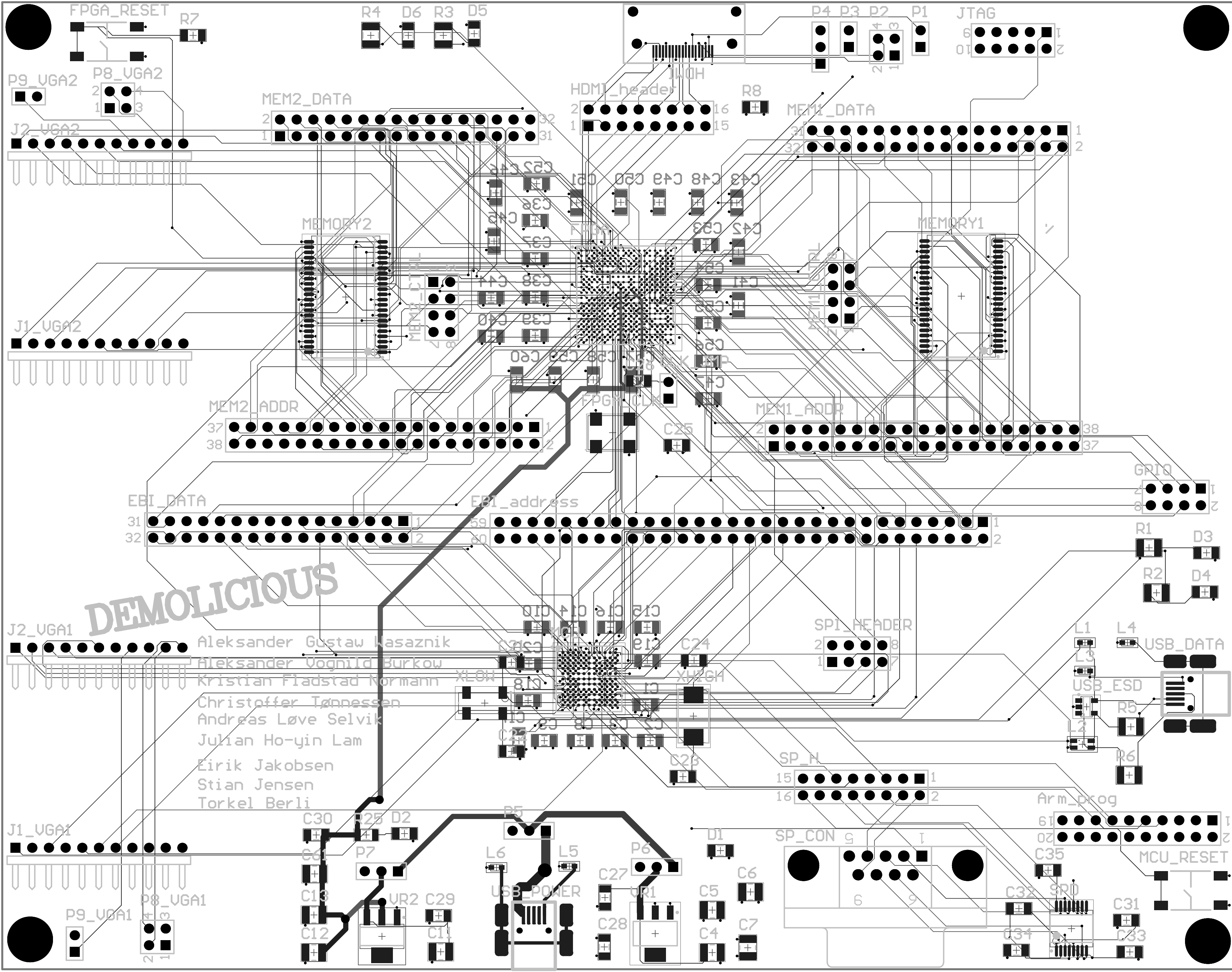




Title		
USB power		
Size: A4	Number	Revision: 1
Date: 11/25/2014		Sheet of
File: C:\Users\..\USB_power.SchDoc		Drawn By: Kristian F. Normann



Title	VGA header		
Size: A4	Number		Revision
Date:	11/25/2014	Sheet of	
File:	C:\Users\..\VGA.SchDoc	Drawn By:	



DEMOLICIOUS
Aleksander Gustaw Wasaznik
Aleksander Vognild Burkow
Kristian Fladstad Normann
Christoffer Tønnessen
Andreas Løve Selvik
Julian Ho-yin Lam
Eirik Jakobsen
Stian Jensen
Torkel Berli
FPGA_RESET
MEM2_DATA
HDMI_header
MEM1_DATA
JTAG
MEMORY2
MEMORY1
MEM2_ADDR
MEM1_ADDR
FPGA_CLK
EBI_DATA
EBI_address
GPIO
SPI_HEADER
USB_DATA
USB_ESD
SP_H
Arm_prog
SP_CON
MCU_RESET
USB_POWER
XLOW
XHIGH
J1_VGA2
J2_VGA2
J1_VGA1
J2_VGA1
P8_VGA2
P9_VGA2
P8_VGA1
P9_VGA1