TDT4295 Computer Design Project

November 20, 2013

BARRICELLI

A high performance, parallel genetic algorithm computer

Eirik Flogard
Fedor Fadeev
Péter Gombos
Emil Taylor Bye
Rune Holmgren
Torbjørn Langland
Per Thomas Lundal
Bjørn Åge Tungesvik
Sigve Sebastian Farstad
Odd Magnus Trondrud

NTNU – Trondheim
Norwegian University of
Science and Technology

Abstract

Barricelli, the computer designed in this project, is a general purpose MIMD computer which is equipped with specialized hardware for high performance computation of hard problems using genetic algorithms. A fully functional system was designed and implemented "from scratch", however no physical realization of the design survived. This complicated testing.

Acknowledgements

A big thank you is extended to the following for invaluable help and guidance during this project:

Gunnar Tufte for fantastic insight, help and project coordination.

Yaman Umuroglu for help and advice.

Stephano Nichele for administrative work.

Odd Rune S. Lykkebø for assistance when all seemed lost.

The Bitless Team for company in the lab during late hours of the night.

Ole Jørgen Seeland and IDI for lending us their 3D-printers.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF ALGORITHMS

LIST OF LISTINGS

Part I

Introduction and Background

CHAPTER

1 INTRODUCTION

This report presents a solution to "Task: Construct a Multiple Program, Multiple Data" of the autumn 2013 course TDT4295 Computer Design Project at the Norwegian University of Science and Technology (NTNU).

The course is a single-task, whole-semester course in which a group of students collaborate to design and implement a computer in hardware. The computer presented in this report was made by a group of 10 students.

This report details the design and implementation of Barricelli, the solution computer designed for this project. The design process is also documented in this report.

1.1 Assignment

This section details the assignment given for the project.

1.1.1 Assignment Text

TDT4295 Computer Design Project

Assignment Text

2013

Task: Construct a Multiple Program, Multiple Data [sic]

The performance increase available from harvesting Instruction Level Parallelism (ILP) from the serial instruction stream is limited because we have reached the maximum power consumption that can be handled without expensive cooling solutions. Consequently, there is a significant interest in single-chip parallel processor solutions. The processor cores in commercial multi-core chips are conventional designs and therefore reasonably complex. In this work, your task is to design a multiprocessor,Multiple Instruction, Multiple Data streams (MIMD). A processor classified MIMD include a multiple-instruction stream organizations. Such processors can executing different instructions, i.e. minimum 2 independent programs, on different (independent) data.

The task also include that a suitable application is chosen to demonstrate the processor. Your processor will be implemented on an FPGA, and you are free to choose how to realize your MIMD computer architecture. The system should be shown to work with a suitable application. Studying the architecture of the Cell processor, or in general multi-core processors, can be a good starting point. And a final tip; Keep it simple, as simple as possible, but not simpler.

Due to a large number of students this year, we will divide the work into two independent projects: a) Performance and b) Energy efficiency. The goal of group a) is to achieve maximum performance while group b) should try to balance performance and energy. The reports from both groups should include an evaluation of prototype performance and energy consumption.

Additional requirements

The unit must utilize an Energy Micro microcontroller and a Xilinx FPGA. 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. Deadlines are given in a separate time schedule.

Evaluation

The project is evaluated based on the project report and an oral presentation of the work as well as a prototype demonstration. One grade will be given to the group as a whole, unless there are significant variations in the amount of effort put into the project. [16]

1.1.2 Interpretation

There are many different ways to solve the task described in the assignment text in Subsection 1.1.1 on the preceding page. To give direction to the project, an application for the solution computer was chosen at an early stage. The application chosen for the solution computer presented in this report is a generic solver of optimization and approximation problems using genetic algorithms. The main goal for the project is to be able to implement the generic solver by exploiting the principles of the MIMD architecture scheme.

As mentioned in the assignment text, there are two groups working on the computer design project, making two different computers. The solution presented in this report is the solution of the group that was to achieve maximum performance.

1.2 Requirements

This section describes the requirements for this assignment, and explains the rationale behind them. The functional requirements for the project are listed in table 1.1. The non-functional requirements for the project are listed in table 1.2 on page 6.

1.2.1 Functional Requirements

Requirement	Description	Priority
FR1	The system should be a MIMD computer.	High
FR2	The system should be a general computer.	High
FR3	The system should be able to solve hard problems using genetic algorithms.	High
FR4	The system should be able to receive instructions and data from an external entity.	High
FR5	The system should be able to send data to an external entity.	High
FR6	The product should have maximized performance rather than energy-efficiency.	Medium
FR7	The instruction set should contain custom made instructions for faster execution of genetic algorithms.	Medium

Table 1.1: Functional Requirements

FR1

FR1 in table 1.1 states that the system should be a MIMD computer. This requirement is set as the main requirement in the assignment text, and is therefore a high priority requirement.

FR2

FR2 in table 1.1 states that the system should be a general computer. That means that the computer should be Turing Complete, which implies that it can solve any computable problem. This was set as a goal because it enables the computation of fitness values of genetic individuals for arbitrary problems. Because this requirement is a necessity for the chosen genetic algorithm solver application, is has a high priority.

FR3

FR3 in table 1.1 on the previous page states that the system should be able to solve hard problems using genetic algorithms. (For a definition of a hard problem, see 2.2 on page 11.) This is the application that was chosen for the computer, and is its *raison d'être*. Therefore, this requirement also has a high priority.

FR4

FR4 in table 1.1 on the previous page states that the system should be able to receive instructions and data from an external entity. Without this requirement fulfilled, it is impossible to program, configure or run the program with other instructions or data than what comes 'pre-loaded', so to speak. That would make the system pretty useless. For this reason, this requirement also has a high priority.

FR5

FR5 in table 1.1 on the preceding page states that the system should be able to send data to an external entity. This is needed for the computer to return its computed results to a user. Again, without this requirement, the computer would be quite useless. Of course, that means that this requirement also must have a high priority.

FR6

FR6 in table 1.1 on the previous page states that the product should have maximized performance rather than energy-efficiency. This requirement is set from the assignment text. Although this is an important requirement, not meeting it does not imply that the solution computer is completely useless. A less-than-optimally performant computer will still be able to solve hard problems, even if it will do it slower. Because of this, even though this requirement is specifically mentioned in the assignment text, this requirement is assigned a medium priority.

FR7

FR7 in table 1.1 on the preceding page states that the instruction set should contain custom made instructions for faster execution of genetic algorithms. This requirement is set because it helps setting the focus on high performance. Another reason for this requirement is that it makes high performance genetic algorithm programming easier for developers using the computer. This greatly increases the usability of the computer.

1.2.2 Non-functional Requirements

Requirement	Description	Priority
NFR1	The system should be built so that the soldering process does not take longer then 8 hours when done by hand.	High
NFR2	The system should use a Xilinx FPGA.	High
NFR3	The system should use an "Energy Micro" microcontroller.	High
NFR4	The system should not cost more than 10000 NOK.	High
NFR5	A report detailing the product and process.	High
NFR6	A working demo program running a genetic algorithm.	Medium
NFR7	The system should have a 3D-printed case.	Low

Table 1.2: Non-functional Requirements

NFR1

NFR1 in table 1.2 states that the system should be built so that soldering does not take longer than 8 hours when done by hand. This would also include time consumed by baking ball grid array (BGA) components. This requirement is included because hand-soldering and simple BGA baking are the fabrication techniques readily available to the group without incurring prohibitively large costs, and is therefore a high priority requirement.

NFR2

NFR2 in table 1.2 states that the system should use a Xilinx FPGA. The assignment text requires that a Xilinx FPGA should be used [16], and the Spartan-6 was chosen. This is a high priority requirement.

NFR3

NFR3 in table 1.2 states that the system should use an "Energy Micro" microcontroller [16]. The microcontroller is needed in order to communicate properly with the FPGA and is a high priority requirement.

NFR4

NFR4 in table 1.2 states that the system should not cost more than 10000 NOK. This is a requirement specified in the assignment text [16], and is therefore a high priority requirement.

NFR5

NFR5 in table 1.2 on the previous page states that there should be a report detailing the product and process. This requirement is perhaps the most important requirement, as all the other work would have been for nothing if it were not documented for others to see. In addition, the assignment text states that the project will be evaluated partially based on the report [16]. This makes this requirement a high priority requirement.

NFR6

NFR6 in table 1.2 on the preceding page states that there should be a working demo program running a genetic algorithm on the system computer. Although this requirement is also specified in the assignment text, it is set for demonstration purposes. If a working demo program is not produced, it does not imply that the solution computer is defective. Because of this, this requirement has a medium priority.

NFR7

NFR7 in table 1.2 on the previous page states that the system should have a 3D-printed case. This requirement has a low priority, as, while it looks good, provides protective cover for the computer, and increases usability, it is not crucial for making the computer work.

1.3 Deliverables

Deliverable	Description
D1	A functioning version of the computer, with the processor running on an FPGA.
D2	A working assembler for the instruction set.
D3	A working demo program running a genetic algorithm.
D3	A report detailing the product and process.

Table 1.3: Deliverables

A deliverable is a tangible or intangible object produced as a result of a project that is intended to be delivered to an other party. In this case, the deliverables are what this group is handing in as a solution to the assignment stated in section 1.1.1 on page 2. An overview of the deliverables for this project can be found in table 1.3. Here follows a short description of each deliverable.

D1

D1 in table 1.3 on the preceding page is the physical manifestation of the Barricelli computer that was designed for this project.

D2

D2 in table 1.3 on the previous page is Galapagos Assembler, a modular Galapagos assembler written in python.

D3

D3 in table 1.3 on the preceding page is a collection of programs written in Galapagos assembly for the Barricelli computer, demonstrating its usefulness as a genetics algorithm solver, and also as a general computer.

D4

D4 in table 1.3 on the previous page is this report.

1.4 About the Name

Barricelli is the name of the solution computer presented in this report. The computer is named after Nils Aall Barricelli, who was a Norwegian-Italian 20th century mathematician. Barricelli the mathematician was a pioneer in artificial life research, and he performed early computer-assisted experiments in the symbiogenesis and evolution fields. It is in recognition of Barricelli the mathematician's groundbreaking seminal work that Barricelli the computer takes its name. [15]

1.5 Structure of this Report

This report is structured as follows:

Chapter 1 introduces the assignment and its interpretation. It also details the goals and requirements set for the project.

Chapter 2 aims to give the reader some general insight into the science and principles behind the project. Additionally, it defines the related terms used throughout this report.

Chapter 3 contains a bird's-eye overview of the overall system architecture and its major components.

Chapter 4 gives an in depth explanation of the processor architecture, its specialized features and most important components.

Chapter 5 details the design choices taken for the different hardware components and the layout of the PCB.

Chapter 6 describes the communication means between the processor on the FPGA and the different input and output devices through the microcontroller.

Chapter 7 covers the remaining major components of the system that does not fit within the above categories.

Chapter 8 contains the different test cases used to verify the correctness of the implementation for the different components.

Chapter 9 presents the results from simulations, performance tests and energy measurements of the processor and its components.

Chapter 10 discusses the results of the tests and measurements.

Chapter 11 presents the work process throughout the whole development, implementation and testing stages.

Chapter 12 states the conclusion for the project, as well as possible further work and improvements.

Finally, there is a glossary for uncommon terms followed by appendices.

CHAPTER

2

BACKGROUND

This chapter aims to put this report in context by providing a brief overview of some of the central concepts of the Barricelli computer.

2.1 MIMD Computing

Barricelli, the solution computer presented in this report, is a MIMD computer. MIMD, or Multiple Instruction, Multiple Data, is a class of computer architectures involving multiple autonomous computing units executing different instructions on different data. Thus, a MIMD computer system consists of several fully independent processing units or cores, interconnected in some way. Each core/processor is able to work fully independently and asynchronously with their counterparts. Because of this MIMD lends itself quite well to high performance computing through parallelism.

Communication between the different, possibly heterogeneous independent computing units is either done by message passing, or sharing memory between processors. Computers that solve communication using message passing might not have any shared memory between the processing units at all. Each processing unit instead holds its own private memory. This is called a distributed memory model. Most MIMD computers implement some hybrid memory solution using both shared memory and distributed memory.

A simplified block-level overview of a MIMD computer can be seen in figure 2.1 on the following page. The computer in the figure uses a shared memory model. In the figure, the blocks labeled PU are individual processing units.

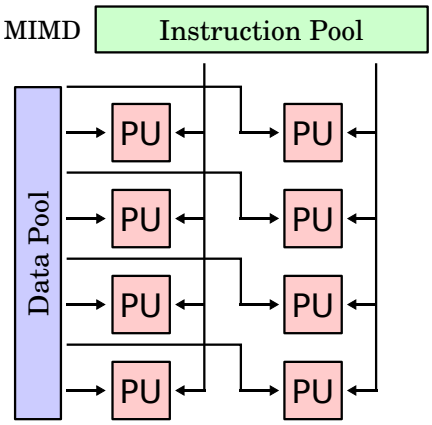


Figure 2.1: Block-level overview of a MIMD computer with a shared memory model. Reproduced from `http://en.wikipedia.org/wiki/File:MIMD.svg`.

MIMD architecture forms the basis in the great majority all today's multi-core super scalar processors. This is in contrast to the early uniprocessors, which were based on SISD, or Single Instruction stream, Single Data stream.

Examples of well-known MIMD computing platforms today include Intel's Larrabee platform, and computer clusters connected for instance over the internet.

2.2 Genetic Algorithms

Searching is a well known problem in computer science, because it can be used to solve many other problems. It is possible to search through a collection of solutions to a problem, rather than finding the correct one by traditional means. This collection is known as the Search space. It is generally easier to verify that a given solution to a hard problem is correct than finding the solution. A heuristic search tries to make the search more intelligent than looking at each parameter and comparing it to the other.

A genetic algorithm is a specific type of heuristic search algorithm that is very useful for finding approximate solutions for hard optimization and search problems. A hard problem in this context is a problem for which there exists no algorithm for feasibly computing a solution within a reasonable amount of time. In practice, this typically means NP-hard problems. Classical examples of hard problems for which genetic algorithms can find suitable approximations include the Satisfiability Problem, the Traveling Salesman Problem and the Binary Knapsack Problem, to name a few. Efficiently finding good approximations to this class of problems is extremely important for many heavy industry processes such as planning and scheduling of processes, finding optimal placement and routing of components in microchip manufacturing, and many other business-critical tasks.

The distinguishing feature of a genetic algorithm compared to other heuristic search algorithms is how it performs its search: it mimics the natural process of evolution to find fit approximations within the search space. A genetic algorithm represents possible solutions to a given problem as individuals of a population. The individuals' fitness is calculated by a problem-specific fitness function, and fit individuals are combined together to create new individuals, mimicking the process of natural selection. The idea is that after simulating a number of virtual generations of individuals, the fittest individual will have survived, and provides a good approximation to the solution.

2.2.1 Concepts and Definitions

To fully understand the inner workings of the Barricelli, it is useful to become acquainted with the basic concepts of genetic algorithms. This subsection of the document defines some genetic algorithm concepts that are used elsewhere in the report.

Experiment An attempt at solving a problem using a genetic algorithm.

Individual One possible solution (good or bad) to a given problem.

Population The group of individuals in a given experiment.

Fitness An evaluation of how close an individual is to the theoretically perfect individual. It is calculated by an objective rating function.

Selection Probabilistic selection of rated individuals allowed to reproduce. The probability of picking a certain individual is typically proportional to its fitness rating.

Crossover The combination of individuals (parents) in some way to form a new individual (child). Also called reproduction.

Mutation Mutation of an individual. It is either random or based on problem-specific rules.

2.2.2 Pseudo-code for a General Genetic Algorithm

Algorithm 1 on the next page shows example pseudo code for a general genetic algorithm.

Data: A population of random individuals
Result: A quite fit individual
begin
  $k \longleftarrow 0$
  $P_k \longleftarrow$ a population of n randomly-generated individuals
  Compute $fitness(i)$ for each $i \in P_k$
  while $max(fitness(i), i \in P_k) < threshold\ AND\ k < maxIterations$
  do
    Selection:
    Select $(1 - x_i) \times n$ members of P_k and insert them into P_{k+1}
    Crossover:
    Select $x_i \times n$ members of P_k, pair them up, produce offspring,
    insert offspring into P_{k+1}
    Mutation:
    Mutate $\mu \times n$ members of P_{k+1}
    Compute $fitness(i)$ for each $i \in P_{k+1}$
    $k \longleftarrow k + 1$
  end
  return *the fittest individual in* P_k
end

Algorithm 1: Generic genetic algorithm

2.2.3 Generational Genetic Algorithms

Traditionally, genetic algorithms are implemented with discrete generations [17] This means that after a loop in the genetic algorithm, a completely new generation has been created by crossover and mutation. The original population is replaced with the new one, and the algorithm continues only with the new population.

Generational genetic algorithms can be seen as performing these steps:

1. Start with an initial population
2. Calculate the fitness of each individual in the generation
3. Selection
4. Crossover
5. Mutation
6. Loop back to step 2 with a newly created population.

The initial population is normally made from randomly generated individuals. The diversity ensures that it is possible for the generation to evolve towards a solution without getting stuck in an early local maxima. In the next stage the fitness of each individual calculated, which is very specific for the problem at hand. If this fitness values exceeds some specified threshold, the solution is deemed good enough and the algorithm stops. Note that it will not always find a perfect solution, but one that is sufficient for the problem at hand. If however none of the individuals end up with a fitness value above the threshold, the algorithm continues.

The solution is found during several phases though a selection, crossover and a mutation phase. These phases are used to evolve the population through several generations until an appropriate solution is found.

The selection step aims to choose individuals for reproduction based on their fitness score, meaning that a higher score gives a higher propability of being chosen. However, this does not necessary mean that only the fittest individuals are allowed to reproduce. An important concept of the genetic algorithm is to achieve diversity while exploring the different solutions. Diversity is important to prevent the algorithm from reaching what is known as a local maximum, a result that has a comparative high fitness score for the immediate region in the Search space.

A sort of evolutionary dead-end, a stage where the solution found is far from optimal but discovering a more optimal one would require several generations of devolving (selecting individuals with non-max fitness from the current population's descendents).

In the crossover phase individuals from the selection are crossed to produce new individuals. Because of the initial diversity in the initial population, this will cause the diversity to be large in the beginning. However they will eventually converge. The goal of the crossover phase is to converge against a solution by continuously creating a better generation of individuals. There are several methods for achieving the crossover of individuals, however, the different methods will not be discussed in great detail here. The different algorithms are highly dependent on the task at hand. However, the different crossover methods implemented in the employed in the Galapagos architecture will be discussed in greater detail in section 4.3.6 on page 45.

As a precaution, the individuals are passed through yet another phase: mutation. The goal of the mutation is to ensure diversity by randomly mutating the individuals by some probability. The mutation process is usually a simple one that just modifies some of the bits in the individual. As with the crossover method, algorithms also exist for performing the mutation.

Lastly, it is important to mention that there exists a lot of different classes of genetic algorithms. They usually employ many different techniques for performing selection, crossover and mutation. The point of this project is not covering all of them. In fact, only a subset of problems will be supported on Galapagos architecture.

2.2.4 Steady State Genetic Algorithms

Steady state genetic algorithms do away with the concept of discrete generations. Rather they continuously select a few individuals, process and return them (or their offspring) to the population. Older members of the population eventually disappear, based on some selection scheme, preventing the population from growing infinitely large. This erases the generational border, and has been shown to converge faster towards an optimal solution. More specifically it allows the algorithm a possible way to work on both rated and unrated individuals at

the same time.[14] This approach is more suited for MIMD architectures, such as the architecture demonstrated in this project.

As steady state algorithms are faster and fitting for MIMD, it was chosen as the main algorithmic goal for the project.

Part II

Design and Implementation

CHAPTER

3

SYSTEM OVERVIEW

3.1 Application

The Barricelli computer, which is pictured in Figure is a computer designed for solving problems using genetic algorithms. An example of this can for instance be the Knapsack problem that is described in the results section in chapter 9. It is highly optimized for problems for which it is possible to express individuals in 64 bits or less. The Barricelli can be controlled through one of its external communication channels, such as USB, but can also be commandeered through the built-in buttons and LEDs.

3.2 System Architecture

This section aims to provide a bird's-eye view of the system architecture of the Barricelli computer.

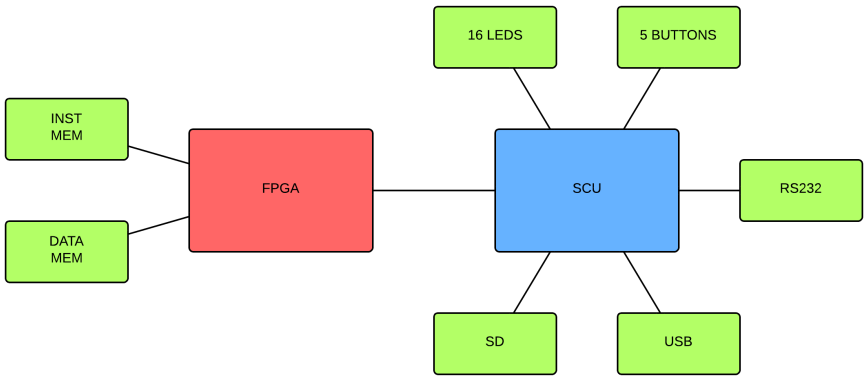


Figure 3.1: System Overview

The requirements in section 1.2 on page 4 state that the Barricelli should use an FPGA and a microcontroller as system components. The Barricelli's specialized CPU design is implemented and compiled onto an FPGA, and a microcontroller is used to facilitate Input/Output, control and communications between the FPGA and the outside world. A graphical overview illustrating the system architecture can be found in figure 3.1.

3.3 Components

This section contains a list of the components in figure 3.1, documenting their roles in the system architecture.

3.3.1 FPGA

The FPGA is a Xilinx Spartan6 LX45 Field-Programmable Gate Array, which is a microchip that can be programmed to behave like any arbitrary intergrated circuit. The FPGA is programmed to contain an implementation of the custom Galapagos Processor Architecture designed for the Barricelli computer. It is connected to the SCU by a 41-bit wide bus, though which the processor can be programmed by users. The FPGA is connected to its own external SRAM memory.

3.3.2 SCU

The SCU, or System Control Unit, is an Energy Micro EFM32GG390F1024-BGA112 microcontroller which administrates communication between the FPGA and the outside world. It is the SCU's job to react to user input, to program the custom processor implemented on the FPGA and to perform other administrative duties. Having a separate microcontroller to perform these tasks minimizes

the complexity of the design implemented on the FPGA, which means that more resources can be used to create a performant and clean custom processor design.

While only one input and one output are strictly needed to meet the functional requirements, multiple different types of input have been added to the design in the name of safety through redundancy. The different inputs and outputs are listed in the following subsections.

USB

Universal Serial Bus is a bus interface used to facilitate the communication between Barricelli and a host computer. USB support has become very widespread, to the point where it is difficult to find a computer without several USB ports, and was a natural choice for a main communication link. Another important reason for choosing USB was because of the large number of software libraries and examples available.

TIA-232

TIA-232, more commonly known as RS-232, is a set of standards defining communication over what is commonly known as serial ports. The serial link is an old standard and a comparatively slow way to transfer data, so it is mainly included as a fallback in case the USB link should fail.

SD

Secure Digital (SD) is a memory-card format chosen as a secondary backup as communications channel between the developers and the microcontroller. Should both the USB and serial ports fail, an SD card could be loaded with the desired instruction and data memory and be read by the microcontroller as if it was a bus.

LEDs

The simplest way to get output from the microcontroller is to use LEDs, short for Light-Emitting Diode. There are a total of 16 LEDs which can be used to display status from the IO controller, or simply show that the IO controller is working on something.

Buttons

Buttons is the fastest and simplest way of having some form of user input on the board. The buttons allow the user to interact with the program running on the SCU without requiring some external IO.

3.3.3 Memory

The Galapagos CPU is connected to two separate external memories, the instruction memory and the data memory. This separation means that the Galapagos architecture is a Harvard machine. This design choice was made because it increases the performance of the machine, since instruction memory and data memory can be accessed independently, and in parallel.

Instruction Memory

The Instruction Memory, labeled as "INST MEM" in figure 3.1 on page 18, is the memory that holds the program instructions for the processing units in the Barricelli running on the FPGA. The memory is read-only for the processors in the FPGA, but can be written to by the SCU. The instruction memory is shared by all the processing units in the FPGA.

Data Memory

The Data Memory, labeled as "DATA MEM" in figure 3.1 on page 18, is the memory that holds processing data for the processors in the FPGA. The memory can be read by both the processing units in the FPGA, and the SCU. The data memory is shared by all the processing units in the FPGA.

CHAPTER

4

PROCESSOR DESIGN

The Barricelli computer features a custom processor design. The processor is a MIMD processor designed for high performance genetic algorithms computing. This chapter describes the processor architecture and documents the design decisions made.

4.1 Initial requirement

The assignment requires the development of a MIMD processor architecture. Therefore, the main requirement is to be able to run multiple instruction streams working on multiple data streams.

The designed computer should be able to solve a limited set of genetic problems. The aim of the design is to achieve high performance of the evaluation of genetic problems by designing a highly specialized MIMD architecture. The resulting architecture aims to tackle the given problems with high performance, and high utilization of resources.

4.1.1 Parallelism

The Barricelli is a MIMD computer, which means that it can execute multiple different instruction streams on multiple different data streams simultaneously, in parallel. The cores in the archtecture have each their own program counters and may load different data independently of each other. The cores share the same data and instruction memory, however, which makes the Barricelli a shared memory model MIMD computer.

4.2 Instruction Set Architecture

The Galapagos Instruction Set Architecture is the instruction set architecture designed for the Barricelli computer for this project. The architecture is loosely based on the well-known and tested MIPS architecture, but borrows inspiration from many other different sources as well. Especially inspirational for the design of the Galapagos ISA have been the MIPS core design principles, which can be found in table 4.1.

Design principle 1	Simplicity favours regularity. [13, p. 79].
Design principle 2	Smaller is faster. [13, p. 81]
Design principle 3	Make the common case fast. [13, p. 86]

Table 4.1: MIPS Design Principles

The Galapagos ISA was designed and fully specified quite early in the project, which made it an important resource for the rest of the component design process.

While the ISA is thoroughly documented in appendix A on page 119, the rest of this section will present a short overview for the reader's convenience.

The Galapagos instruction set architecture is a RISC architecture. The instructions are kept simple, and only perform very specific and small tasks. That is, the instructions are low-level instructions executed directly in hardware without the need for additional decoding in form of microinstructions or the like.

4.2.1 Instruction Formats

As MIPS, Galapagos, in true RISC fashion, has relatively few instruction formats. These instruction formats are constructed to be regular, which implies that the different information types contained in an instruction are located in the same positions whenever possible. This makes the instruction decoding process in the processor much simpler. This is done in accordance with design principle 1 of table 4.1.

The three instruction formats used in Galapagos are the RRR, RRI and RI formats. They are named after the types of data they contain. RRR contains three register addresses. RRI contains two register addresses and one immediate. RI contains one register address and a larger immediate.

Every instruction has a set of conditional flags that may be set. Through these flags, a programmer can decide whether or not an instruction will be executed. This allows for branchless conditional execution of single instructions. These conditional signals allow for many clever applications - a nop instruction can be implemented as any instruction with the condition set to "never execute". Indeed, even conditional branching is implemented as a branch-less conditional! For a more detailed documentation of the Galapagos instruction set architecture, the reader is encouraged to read appendix A on page 119.

4.2.2 Genetics Instructions

One of the requirements in section 1.2 on page 4 was that the ISA should support genetic-specific instructions to facilitate performant genetic algorithms programming. Present in the Galapagos ISA are the genetic instructions `ldg`, `stg` and `setg`. They are the instructions for loading and storing individuals to the genetics accelerator, and configuring the genetics accelerator, respectively. With these instructions available to the programmer, using the genetics accelerator is easy and painless. The reader may refer to the ISA documentation in appendix A on page 119 for in-depth documentation about how the genetics instructions are used.

4.3 Processor Architecture

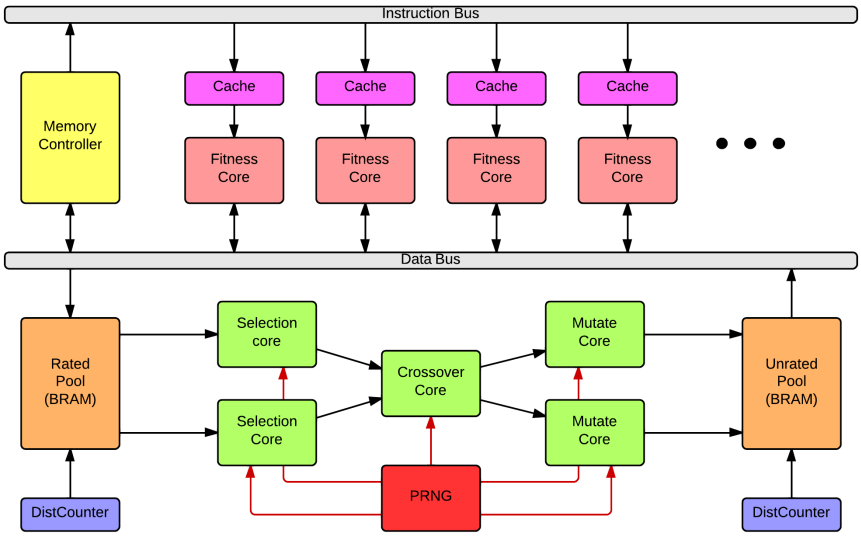


Figure 4.1: Processor Architecture

Figure 4.1 shows the final processor architecture. Most of the figures seen in this picture as for instance "fitness core", are abstractions of more complex logic at lower levels (mostly MSI and LSI components). The processor architecture designed for the Barricelli computer is a very clean design, and the key to its high performance lies in its simplicity. The architecture contains a number of general cores, which in this context are named fitness cores. The fitness cores are general purpose cores in the sense that they are programmable and Turing complete, but for genetic algorithm applications the cores are intended to calculate fitness scores of individuals.

The number of fitness cores is configurable. The reference implementation of the Barricelli computer is configured to have 7 fitness cores.

Common genetic algorithms operations are performed by a separate hardware accelerator pipeline. This accelerator consists of several operation-specific special cores for selection, crossover and mutation. The fitness cores and the genetic pipeline are all connected to a single data bus. To avoid any memory synchronization issues the data bus is controlled by a central arbitration unit.

The processor architecture is illustrated in figure 4.1 on the preceding page.

4.3.1 Instruction Memory

The Barricelli is a Harvard machine. Because of this the memory is split into instruction and data memory. This is done to achieve better memory throughput, because both memories can be accessed simultaneously. The instruction memory is organized in a two layer memory hierarchy, with slower external memory (SRAM) and faster, internal on-chip caches (BRAM). This separation combines the high instruction throughput of fast on-board memory with the comfortably spacious data storage capabilities of a larger, slower chip external chip.

Each fitness core has its own private instruction memory cache which buffers instructions to decrease the number of slow memory accesses needed during run-time. Access to an instruction cache is handled by a fitness core's dedicated cache controller, which is responsible for locating and transferring instructions from the instruction memory. In case of a cache miss, the data-requesting core is halted until the instruction is transferred from memory. A pseudo-algorithm describing the cache fetch operation can be found in algorithm 2. This scheme is created to resolve the conflicts that arise from using shared memory.

Data: a = an instruction address
Ci = an array of instructions
Ca = an array of the corresponding addresses
M = the instruction memory, indexable by instruction addresses
Result: The instruction at address a
begin
  if $a = Ca[A \bmod 512]$ **then**
    return $Ci[A \bmod 512]$
  end
  else
    $Caa \bmod 512] \longleftarrow a$
    $Ci[a \bmod 512] \longleftarrow M[a]$
    return $Ci[A \bmod 512]$
  end
end

Algorithm 2: Fetching an instruction from the cache

4.3.2 Data Memory

The Galapagos architecture is a MIMD architecture with shared memory. In the Barricelli computer, a central memory controller is responsible for synchronizing memory access on the shared data bus. Each component that wants to access memory must go through the memory controller, and follow the proper memory access request protocol. The controller is constructed in a way that only allows one fitness core to be able to carry out a memory request at a single time. In case of multiple memory requests, the controller performs a selection deciding in which order the requesting cores is granted the bus. The precise technique of selection can be seen in algorithm 3. This may introduce a potential bottleneck for memory-bound problems. For this reason, each fitness core has a generous 31 general purpose registers, which should reduce the data memory load quite a bit.

```
Data: Requests = requests signals from the fitness cores
      Request = 2-bits specifying the operation
begin
    Requests ⟵ requests from the fitness cores
    while True do
        for request in Requests do
            if request = asserted then
                performMemoryOperation()
            end
        end
    end
end
```

Algorithm 3: Round-robin selection

The selection algorithm is based on round-robin scheduling. The request signals of *fitness cores* are checked in turn to check if one of the cores has requested the memory bus. The type of request is determined by combination of two request signals sent by each *fitness core*. The signals refer to either a *NOP*, *READ*, or *WRITE* operation. In case of *NOP*, the algorithm moves on to check the next state request lines. It continues doing this in this fashion until a *READ* or *WRITE* request is encountered.

When a *READ* or *WRITE* operation is encountered, the *data controller* starts to carry out the request from the fitness core. Performing a memory operation takes at least four cycles, as the processor word size is 64 bits, while the memory bus to the external memory chips are only 16 bits wide. Because of this, data needs to be shuffled across the bus 16-bits at a time, which accounts for the four cycle minimum for data operations.

For the external memory to be operated correctly by the memory controller, the proper control signals need to be set at the correct times. The signals required differs depending on the type of operation, *READ* or *WRITE*. The timing diagrams can be seen in figure 4.2 and 4.3, respectively.

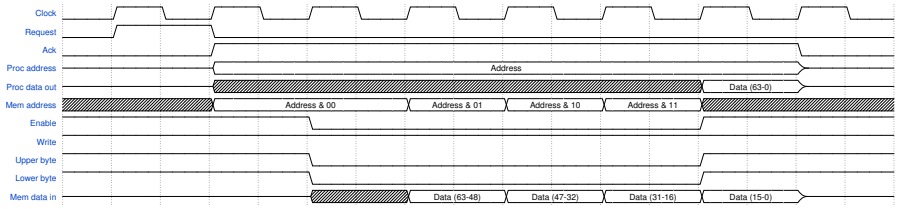


Figure 4.2: Data memory read cycle

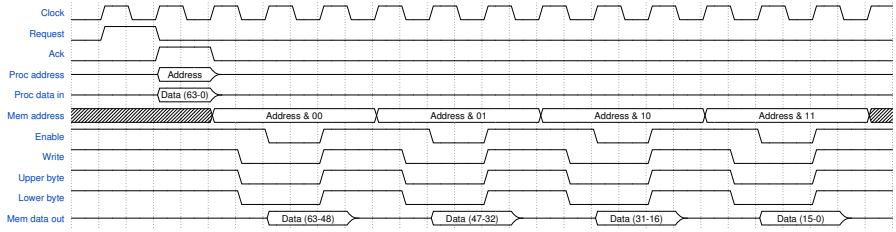


Figure 4.3: Data memory write cycle

As is immediately apparent in figures 4.2 and 4.3, the number of cycles required for load and store operations are are 5 and 13 cycles, respectively. A state machine is implemented in the *data memory controller* to handle interfacing with the external memory chips. This state machine is responsible for controlling that the different signals are set according to the diagrams. For more detailed view of the *Data memory controller*, the reader is advised to study the state machine diagram in figure 4.4 and the accompanying data path in figure 4.5.

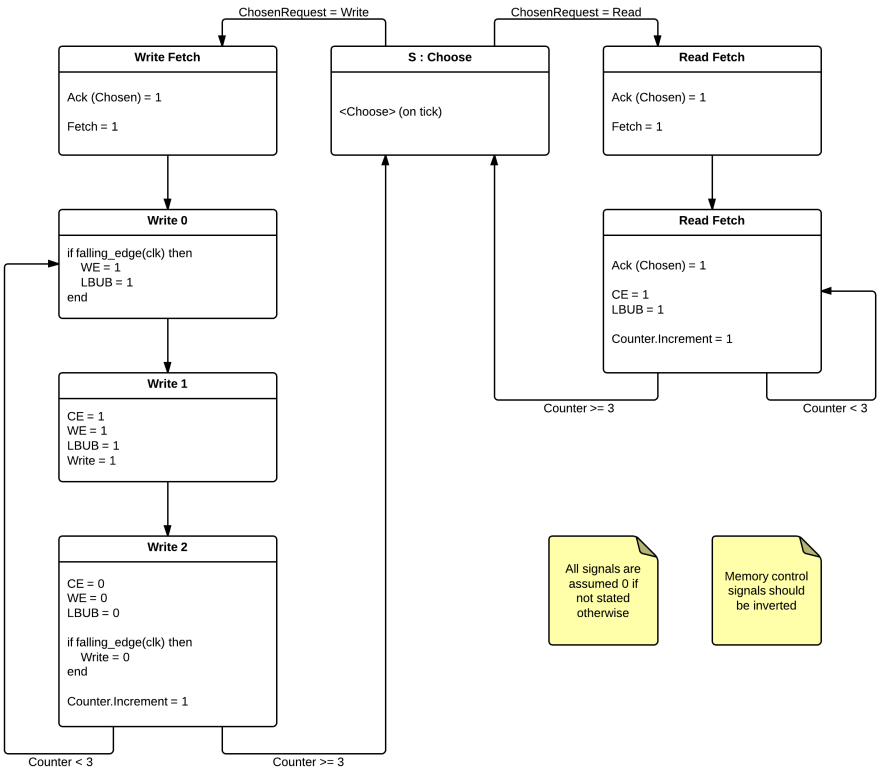


Figure 4.4: Data memory controller state machine

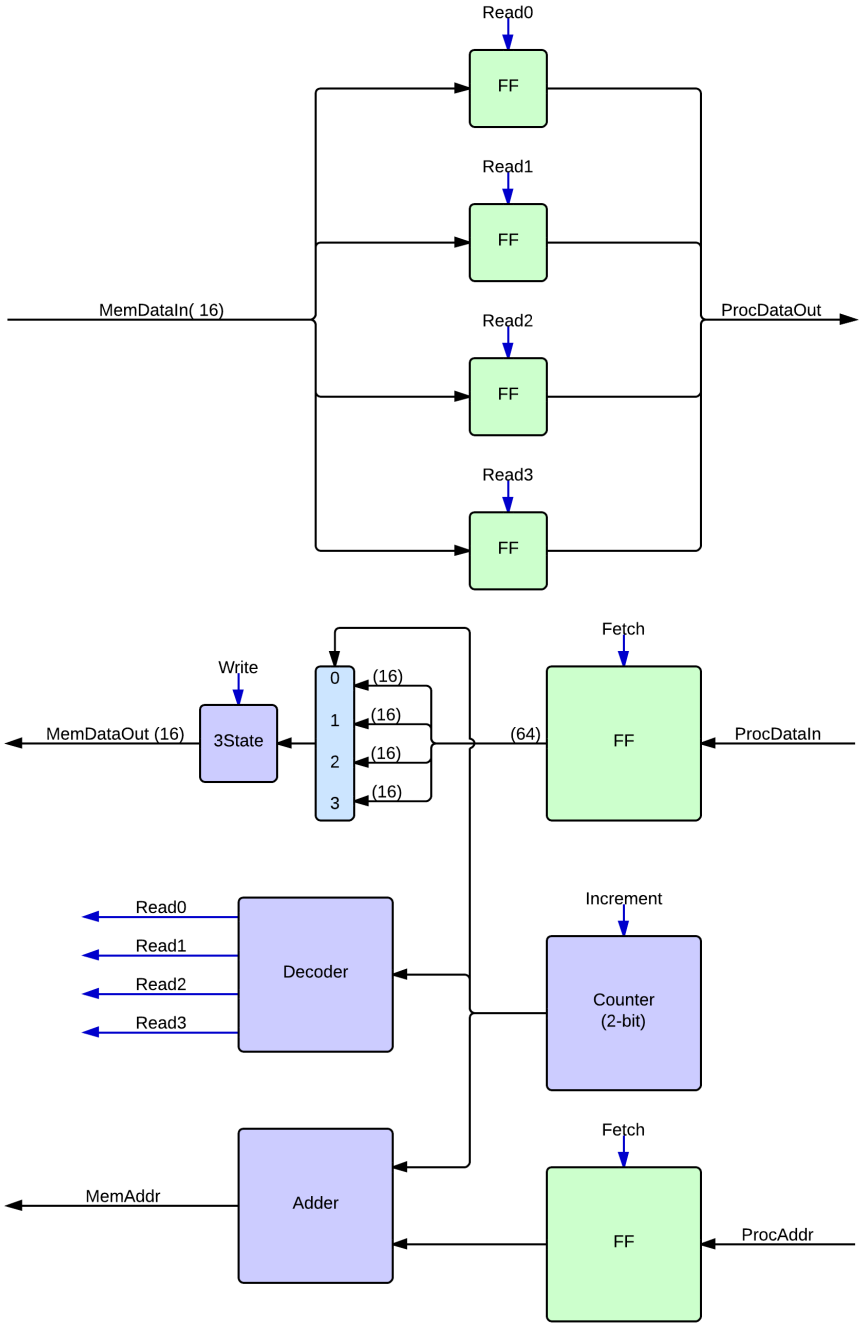


Figure 4.5: Data memory controller signals mapping

4.3.3 Rated and Unrated Pools

In the beginning and the end of the genetics accelerator pipeline lie the rated and the unrated pools, respectively. The rated and unrated pools are caches of genetics individuals that waiting to either 1) get selected and be sent through the pipeline, 2) die, or 3) get picked up by a fitness core for fitness ranking. The rated pool contains individuals stored with a fitness score, and are the individuals that have just been inserted into the accelerator by a fitness core. The unrated pool contains individuals that have no fitness score calculated. They are the new individuals produced by the accelerator pipeline, and are waiting to be picked up and rated by a fitness core.

The rated and unrated pools are implemented in BRAM on the FPGA for as fast access times as possible. This is essential to achieve a high memory throughput when executing the algorithms.

It is important to note that the two pools are designed as separate BRAM devices. This is done to achieve even better memory throughput. The increased throughput is achieved because the different computational units can work on the rated and unrated pools simultaneously. For instance while one fitness core is storing a ranked individual, another fitness core may be fetching a new individual for ranking.

Access to the Rated pool and the unrated controller is managed by control units referred to as the rated controller and the unrated controller. As with the data controller for data memory access, these controllers are responsible for granting access to the rated and unrated pools. As shown in figure 4.12 on page 46, the genetics accelerator has its own data buses connected to the fitness cores separate from the data bus that is connected to the regular shared data memory, to increase performance by reducing bus conflicts.

The controllers are based on the same principles as the data controller described in section 4.3.2 on page 25. When in need of performing genetic operations, the fitness cores need to request the data bus by setting some request signals. The combination of these signals refer to the operation the fitness core requests from the genetic controller.

The controllers continuously perform simple round-robin request handling schemes in order to grant bus access to the next requesting fitness cores, and to the genetic pipeline. The logic for the rated controller is implemented as a state machine, while the unrated controller's simple structure allows it not have one at all.

As the timing diagrams in figure 4.6 on the following page and 4.7 on the next page show, both controllers are highly optimized for speed. The single occasion where the bus is unused is the cycle in which the Genetic controller tells the rated controller that it no longer requires access.

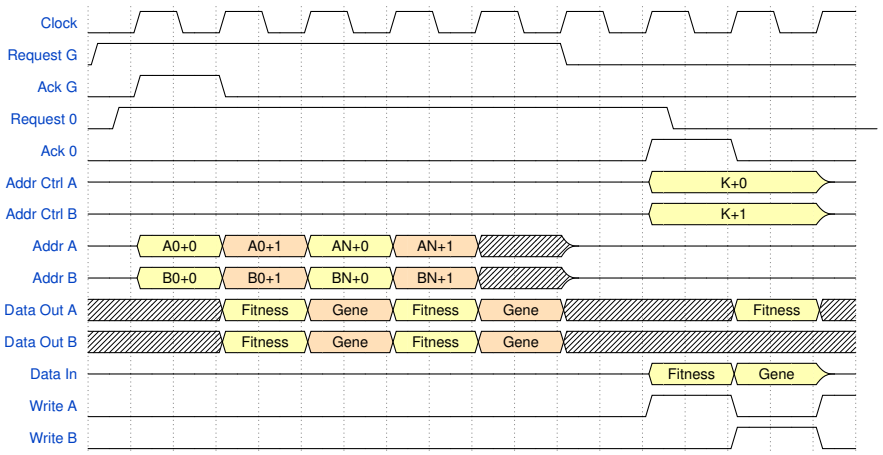


Figure 4.6: Fitness values and individuals being read by the selection cores followed by a fitness core writing a new fitness and individual.

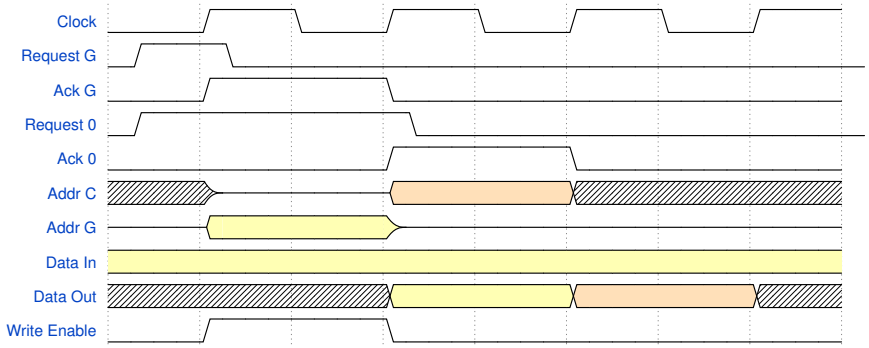


Figure 4.7: A two-individual selection core access

4.3.4 PRNG Module

A key component in any genetic algorithm is a decent source of (pseudo-)random numbers. Genetic algorithms require diversity in the individuals to prevent reaching a local maximum, as discussed in section 2.2.3. To achieve this, the architecture need to support a way to produce random numbers for the genetic pipeline.

The Barricelli computer has a hardware pseudo-random number generator module built into its genetics accelerator.

When designing a pseudo-random number generator, there is always a trade-off between generating "good" random numbers, and generating them quickly. As the group have high performance as a design goal as described in the introduction, it was desirable to design a pseudo-random number generator that is as fast as possible while still being usable for genetic algorithms.

The pseudo-random number generator uses a linear shift feedback-based taps algorithm to efficiently generate random numbers.

The design is inspired by the implementation made by Raymond Sung, Andrew Sung, Patrick Chan, and Jason Mah [1].

This algorithm is shown in algorithm 4.

Data:
$LFSR$ = a 32-bit linear shift feedback register containing the previous random number
$taps$ = "01000110000000000000000100000000"
Result: A new random number
begin
  $feedback \longleftarrow LFSR[31]$
  for *i from 31 to 0, non-inclusive* **do**
    if *taps[i - 1] = 1* **then**
      $LFSR[i] \longleftarrow LFSR[i-1] \oplus feedback$
    end
    else
      $LFSR[i] \longleftarrow LFSR[i-1]$
    end
  end
  $LFSR[0] = feedback$
  return $LFSR$
end

Algorithm 4: Pseudo-random number generation algorithm

4.3.5 Fitness Core

The fitness core in the processor architecture is the general processing core. For genetic algorithms, the fitness core is tasked with calculating fitness scores for individuals, hence the name.

The design of the fitness core is highly influenced by the classic pipelined MIPS core design [13, p. 362]. The core is designed as a five stage pipeline. The contents of the different stages in the pipeline, however, differs from the original MIPS architecture, as the CPU architecture has to accommodate for the ISA design which combines ideas from multiple existing architectures. An example of an ISA feature which differs from MIPS is the embedded branch-less conditionals[2]. An overview of the data path can be seen in figure 4.8 on the next page.

[1] http://www.ece.ualberta.ca/ elliott/ee552/studentAppNotes/1999f/Drivers_Ed/lfsr.html
[2] a feature which is shamelessly ripped whole-sale from ARM

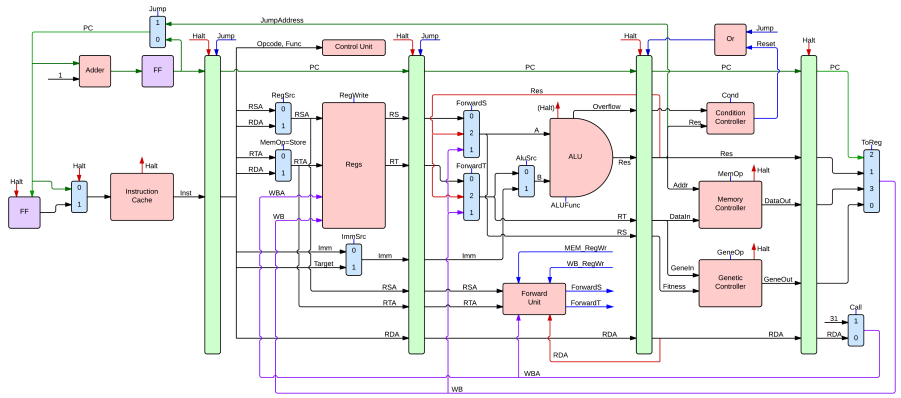


Figure 4.8: Architecture block diagram

The core is separated into several distinguishable stages referred to as the *fetch stage*, *decode stage*, *execution stage*, *memory stage* and *write-back stage*. These stages overlap in time and perform different tasks regarding the execution of instructions. The fetch stage is responsible for fetching a new instruction to the pipeline. The decode stage is responsible for decoding the instruction and set up the different control signals. The execution stage is involved in the actual computation of the instruction. The memory stage is responsible for performing memory related task on behalf of the instruction. The write-back stage is used to write data back into registers.

Data Path

The data path is simple. All instructions have the same length. This makes it easier to fetch the instruction in the first stage and decode it in the second. Also, the ISA only supports three different instructions classes with the register mappings located on the same positions. This allows the register file to be read on the same time as the control unit determines the correct control signals for that particular instruction. This allows for a shorter pipeline. If the instruction format was asymmetric, the instruction would have to split the register file and decode stage in two stages. A bigger pipeline would imply higher risk for pipeline hazards, and a more complicated hazard detection and correction schemes.

The Barricelli computer is a load/store machine. There is no preconception of a program stack at the hardware level. This simplifies the execution and memory interaction. Note that memory operands only appear in load and store instructions. This implies that the execution stages is responsible for calculating the memory address and that the memory access can happen in the following stage. Allowing the ALU to operate on operands in memory would expand the address stage, memory stage and the execution stage [13, p. 335]. The resulting architecture would involve a deeper pipeline.

The MIPS inspired pipeline allows the design to be simple, but still powerful. It will increase performance by effectively increasing the instruction throughput. Note that instruction throughput is an important metric because real applications execute millions of instructions [13, p. 335]

The group decided that the overall design should be as simple as possible. Hazard and branching schemes are kept simple. Hazards are resolved with the *forwarding unit*, which forwards data if dependencies are detected. Branching are resolved with use of *conditional* codes in each instructions. The reader may refer to their respective sections for more details.

The fitness core features “branch not taken” branch prediction, which means that it assumes that conditional jumps will not be executed. This gives a nice performance increase, because the pipeline does not have to be flushed when the branch is correctly predicted. More advanced features like instruction rescheduling have not been implemented.

Control Unit

The control unit is the "puppet master" of the processor, so to speak. Residing in the instruction decode stage, it is responsible for configuring all the different components for the current CPU operation so that the desired computation will emerge from the flow of data. The control unit achieves this by setting the relevant control signals of the relevant components to the values associated with the current instruction. Note that different instruction classes requires different use of the data path. Thus, combinations the control signals must accommodate the different instructions formats. The different instructions classes and their corresponding control signals can be seen in table 4.2.

The control unit sets up the components based on the *FUNCTION CODE* and the *OPERATION CODE* of the instruction. The *FUNCTION CODE* is the 4 least significant bits of the instruction, and is responsible for determining the operation the ALU should perform.

The following paragraphs exhaustively documents and explains the technical details of each of the control unit's control signals.

ALU_FUNC The *ALU_FUNC* is the least four significant bits of the instruction, and is responsible for determining the type of operation the *ALU* should perform. This are set according to table 4.4

ALU_SOURCE The *ALU_SOURCE* signal is used to specify the second *ALU operand*. When this signal is asserted the second ALU operand is the immediate field of the instruction. This implies that the instruction class of the instruction is either RRI or RI. On the other hand, if this signal is de-asserted, the operand comes from read register to of the instruction file. Note that this value may or may not have been forwarded.

REG_WRITE This signal specifies whether the write back value residing in the *write-back* stage should be written to the register file. When this signal is asserted, the value from the *write-back* stage is written to the specified register address. The register address is specified in the instruction, and always resides in the RT address field of the instruction.

IMM_SOURCE The Galapagos ISA has two different instruction classes (RI, RRI) that uses different lengths of the immediate field. In order to differentiate between these immediate fields, the *IMM_SOURCE* signal is responsible for selecting the correct immediate field based on the instruction class of the current instruction. The selection is done with a multiplexer. In the case of an RRI instruction the 10-bits immediate field is selected, and in the case of an RI instruction, the 19-bits field is chosen.

[3]Gives input 1 as output.
[4]Gives input 2 as output.

Control signals	RRR	JUMP	SW	STG
ALU_FUNC	FUNC	ADD	ADD	NA
ALU_SOURCE	0	1	1	0
IMM_SOURCE	0	1	1	0
REG_SOURCE	0	1	0	0
STORE_SOURCE	0	0	1	0
CALL	0	0	0	0
JUMP	0	1	0	0
GENE_OP	NA	NA	NA	10
MEM_OP	NA	NA	10	NA
TO_REG	01	NA	NA	NA
REG_WRITE	1	0	0	0
	RRI	LW	STI	SETG
ALU_FUNC	FUNC	ADD	FUNC	FUNC
ALU_SOURCE	1	1	1	0
IMM_SOURCE	0	0	1	0
REG_SOURCE	0	0	0	0
STORE_SOURCE	0	0	0	0
CALL	0	0	0	0
JUMP	0	0	0	0
GENE_OP	NA	NA	NA	11
MEM_OP	NA	01	NA	NA
TO_REG	01	11	NA	NA
REG_WRITE	1	1	0	0
	CALL	LDI	LDG	
ALU_FUNC	ADD	ADD	NA	
ALU_SOURCE	1	1	0	
IMM_SOURCE	1	1	0	
REG_SOURCE	1	0	0	
STORE_SOURCE	0	0	0	
CALL	1	0	0	
JUMP	1	0	0	
GENE_OP	NA	NA	01	
MEM_OP	NA	NA	NA	
TO_REG	NA	01	00	
REG_WRITE	0	1	1	

Table 4.2: Control unit output

Opcode	Instruction class
1000	RRR
1100	RRI
0000	LW
0001	SW
0100	LDI
0010	JMP
0011	CALL
0101	STI
1001	LDG
1010	SETG
1011	STG

Table 4.3: Overview of opcodes

FUNC code	operation
0000	ADD
0001	SUB
0010	MUL
0011	SRA
0100	OR
0101	AND
0110	XOR
0111	SLL
1000	SRL
1001	OP1 [3]
1010	OP2 [4]
1111	NOP

Table 4.4: Overview of function codes

REG_SOURCE For *JMP* and *CALL* instructions, the immediate address field of the instruction is added with the address content of Rd. The Rd address specified in the instruction must be read from the register file. Normally, the register file would use the *Rs* and *Rt* part of the instructions as inputs to the register file. In the special case of *RI* instructions the register file must read the Rd instead of Rs. This is accomplished by asserting the *REG_SOURCE* signal, which causes a multiplexer to choose the *Rd* portion of the instruction instead of the *Rs* as input to the register file. This will output the content of the correct data from the register file.

STORE_SOURCE For the *ST* instruction, the register address of the data to be stored is located in the *Rd* field of the instruction. Normally, the register file would use the *Rs* and *Rt* part of the instruction as inputs to the register file. In the special case of a *ST* instruction the register file must read the *Rd* register instead of *Rt*. This is accomplished by asserting the *MEM_WRITE* signal, which causes a multiplexor to chose the *Rd* portion of the instruction instead of the *Rt* as input to the register file.

This will output the content of the correct data from the register file. This ensures that the *Rt* signal seen in 4.8 contains the content of the *Rd* register. This is actually the data that is written to memory.

JUMP The JMP instruction uses the address in the immediate field of the instruction to load a new value into the *program counter* register. The jump signal selects between the jump address and the incremented program counter when deciding a new value for the program counter at each cycle. In case of jump, this signal is asserted and the jump address is chosen. When de-asserted the program counter is set to the incremented program counter, making the program execute normally. This signal is also asserted when performing call instructions.

CALL The Galapagos ISA supports non-recursive procedure calls at a hardware level. The call instruction works similar to the jump instruction described above. The difference is that the incremented value of the program counter before the jump is stored to register r31, which is conventionally used as a link register. This value can be used later when returning from the procedure call. To return from a procedure call, one needs simply to jump back to the address stored in r31.

The *CALL* signal is responsible for making sure that the incremented program counter is stored at register 31. When asserted the signal make sure that the write register address is changed to 31. If the signal is de-asserted the write register will stay unmodified, and the Rt register in the instruction is responsible for specifying the register address.

GENE_OP Loading and storing of fitness values and chromosomes is the responsibility of the *genetic controller*. This controller is able to perform three types of actions: *load*, *store* and *settings*, dependent of the instruction class executed. When performing a load, a chromosome is loaded from the unrated pool. The store operation is used to store a chromosome and its corresponding fitness value to the rated pool. The settings are used to apply settings to the genetic pipeline. In order to divide these cases, the Galapagos architecture relies on the *gene operation* vector. This bit-vector is set depending on the instruction class as seen in table 4.5.

Code	Meaning	Instruction class
00	NOP	OTHER
01	LOAD_GENE	LDG
10	STORE	STG
11	SETTINGS	SETG

Table 4.5: Gene operation codes

Depending on the bit-vector-signal, the *genetic controller* is able to perform the appropriate action.

MEM_OP As with the genetic controller, the memory controller must be able to distinguish between operations. In case of memory controller, these operations are loading and storing to and from the external data memory. These signals are set according to the instruction class currently being executed. An overview can be found in table 4.6

Code	Meaning	Instruction class
00	NOP	OTHER
01	LOAD_DATA	LW
10	STORE_DATA	SW

Table 4.6: Memory operation codes

TO_REG In the *write-back* stage, there is need to distinguish between several outputs from the *memory stage*. The *TO_REG* signal is responsible for selecting which value that should be written to the register file. The selection is aided by a 4-to-1 multiplexer. The different inputs are: *Gene*, *Data*, and *PC+1*, *Res*, as seen in figure. However, keep in mind that the *REG_WRITE* signals must be asserted for the data to be written.

The *TO_REG* bit-vector are set according to table 4.7

Code	Output	Instruction class
00	GENE	LDG
01	RES	OTHER [5]
10	PC+1	CALL
11	DATA	LW

Table 4.7: 4-to-1 multiplexor output

[5]OTHERS refers to instructions that store the ALU result to a register

The ALU

The Arithmetic Logic Unit, or ALU, is the heart of the fitness core, colloquially put. It is responsible for executing scheduled arithmetic and logical operations on data. The ALU is perhaps the single most important component of the fitness core.

The ALU in the fitness core is capable of performing a large array of different mathematical and logical operations on 64-bit words, including addition, subtraction, comparisons, shifts and multiplications, in addition to a slew of bitwise logical operations such as and, or, xor and not. Division is not supported.

On the FPGA, large parts of the ALU is implemented with DSP slices, which means that dedicated pre-fabricated ALU-specific circuitry is used in place of regular generic FPGA LUTs for increased performance and space utilization.

Multiplication Multiplication is handled as a special case in the architecture. The multiplication is designed to use two cycles in the *execution stage*, rather than the normal one cycle per operation. This is because the multiplication is the slowest of the ALU operations, and the maximum clock frequency would be severely limited, were the multiplication path allowed to become the longest path. This would result in a degradation of the overall performance of the processor, which is not desirable. To overcome this limitation, a state machine is designed in the execution stage to halt the pipeline when performing multiplication. This effectively makes the multiplication a multi-cycle operation. The result of this is that the multiplication operation is able to finish without the rest of the system needing to slow down to accommodate for it, meaning that the maximum clock speed is not limited by the multiplication circuitry.

Shifters A shifter is a subcomponent within the ALU designed for efficient generic bit shifting. A shifter can shift an input by M bits, either left or right unsigned, or right signed. M is set generically when the component is created, while signal inputs *Left* and *Arith* determines type of shift. If *Left* is set, there is unsigned shift to left. If *Left* is not set, but *Arith* is, then it is a signed shift to right. Otherwise, the shift is to the right unsigned. If *Enable* is not set, then there will be no shift.

ShifterVariables A ShifterVariable consists of a larger set of shifters. It takes two inputs: One main input I with size N (default 64) to be shifted, and a *count* input of size M that determines how many bits to be shifted. M should be log_2 of N. The number of shifters in the ShifterVariable is equal to M (default 1), and each shifter i may shift the main input by 2^i bits, where i is a number in the range 0 to M-1. Total shift is $\sum_{i=0}^{M-1} count_i * 2^i$ where $count_i$ is bit number i in the *count* input. For example, if it is 000110, shift of input I is $0*2^5+0*2^4+0*2^3+1*2^2+1*2^1+0*2^0 = 2^2+2^1 = 6$. This can be seen in figure 4.9 on the following page

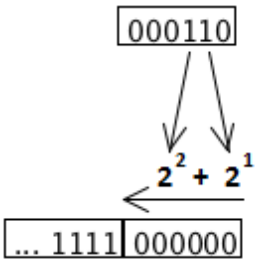


Figure 4.9: Example use of ShifterVariable

The ShifterVariable has also inputs for *Left*, *Arith* and *Enable* for each shifter. ShifterVariables are not only used in the ALU, but also in the Crossover Core and Mutation Cores in the Genetics Pipeline.

Forwarding Unit

Since the execution of instructions overlap in the pipeline, there is need for some mechanism to handle the data dependencies that arise between the instructions. These dependencies are known as data hazards. They occur when a planned execution of an instruction cannot happen in that cycle because some data is not yet available. This problem can be solved in two ways, either by stalling or forwarding. Stalling, the simplest solution, is done by avoiding the hazard by stalling until the data becomes available. This is accomplished by inserting *NOPs* in the pipeline. Although, this method works, it is unacceptably slow for a high performance architecture. Since this processor aims to achieve high performance, stalling is far from optimal. A far better approach is to rely on register forwarding. In this approach the aim is to resolve the dependencies by simply forwarding internal resources from one pipeline stage to another, if needed.

The forwarding logic is implemented by comparing register dependencies of the currently executing instruction with the other instructions currently flowing through the pipeline. If the forwarding unit detects a data hazard, it will forward the appropriate data to the execution stage so it can be used at once. An overview of the control signals and their meaning can be seen in table. 4.8.

A second, more unconventional forwarding unit is also implemented in the decode stage. It forwards data from the write back stage to the decode stage to solve data hazards caused by a single cycle write delay in the register file that the regular forwarding unit cannot detect.

Unfortunately, forwarding does not solve all possible data hazards in the processor. In the case of a memory load instruction being immediately proceeded by an instruction dependant on the data fetched from memory, a data hazard occurs. Since it does not make sense to forward data from memory, this class of data hazards is resulted by stalling. The Galapagos architecture does not solve this problem in hardware, but rather in software. The assembler is responsible for inserting *NOP* instructions in case of such hazards.

Control signal	Meaning
forwardA=00	The first ALU operand is from the register file
forwardA=10	The first ALU operand is from the prior ALU result
forwardA=01	The first ALU operand is either data from memory or an earlier ALU result
forwardB=00	The second ALU operand is from the register file
forwardB=10	The second ALU operand is from the prior ALU result
forwardB=01	The second ALU operand is either data from memory or an earlier ALU result

Table 4.8: Forwarding control signals

Conditional Unit

Like ARM, the Galapagos architecture embeds conditional codes in every single instruction in order to determine if it should be executed. In Barricelli's processor, every instruction is executed, but the conditional codes may disable the effect of the instruction. The *condition unit* is responsible for checking the condition code against the status flags set by the previous instruction, and using this information to effect the conditional.

Fitness Memory Controller

In order to communicate with the main data controller, each fitness cores contains a small version of the data controller to synchronize the communication to the main controller. This small controller, referred to as the *fitness data memory controller*, is responsible for mediating memory requests between the data memory controller and the fitness memory controller.

To use the data memory bus, the the *fitness memory controller* may send a request signal to the main *Data controller*. The *data controller* either handles this request immediately, or the request is handled when the bus is ready, in case some other core is currently using the bus. Either way, the requesting fitness core will halt the pipeline waiting for access to the bus. When the *data controller* is ready, it will send out an acknowledgment granting the bus. Upon receiving the bus, the *fitness memory controller* is able to use it freely.

In case of a *READ* operation, the address is put on the address bus. This address corresponds to a memory position on the external memory. When receiving this address, the *data memory controller*, starts fetching the memory word. During this time the fitness memory controller needs to wait. It stays in a wait loop unit the *data memory controller* is finished reading. Note that the bus to external memory is only 16-bits wide. This implies that four read instructions are required to read a 64-bits word. When the *memory data controller* is finished reading data from memory, the data is put on the data bus, and the can be read by the fitness core. Then the fitness core is disconnected from both the data and memory bus and it continuous its execution through the pipeline.

When performing a *WRITE* operation this is done in the same manner as with the *READ* operation. The difference lay in the fact that during a memory write, the data to be written is put on the outgoing data bus. The *fitness core* is in wait state at this moment. When the data is finished writing to memory, the *memory data controller* responds with an acknowledgement. The *fitness core* continuous with its computation. A state diagram showing this can be seen in figure 4.10.

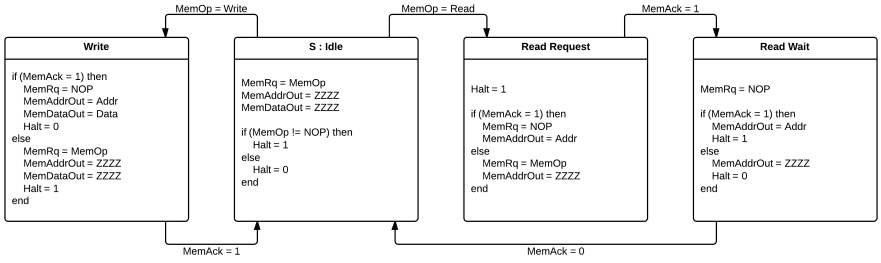


Figure 4.10: Fitness memory controller state machine

Fitness Genetic Controller

Analogously to the fitness core, the genetics pipeline also has a separate mediating memory controller. This controller is referred to as the *fitness genetic controller*. It is more complex than its sibling in the fitness core, since it interfaces with the genetics data controller, which arbitrates two independent memory stores. This controller is very similar the *fitness data controller*. The difference lays in the fact that two values are transferred in case of a *WRITE* operation. Note that both the fitness values needs to store both the fitness value and gene. The protocol is, however, still very similar the *fitness memory controller*. The state machine can be seen in figure 4.11.

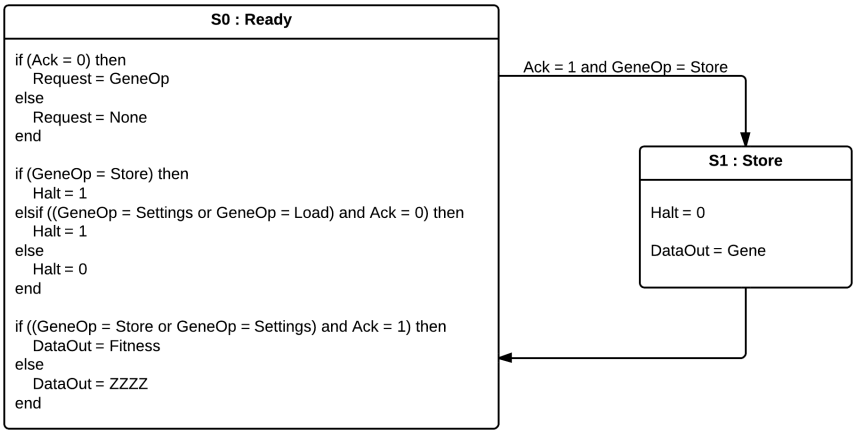


Figure 4.11: Fitness genetic controller state machine

Halting the Pipeline

In normal computer systems, the time to access the memory is variable. It depends on the different states of the machine. This imposes a problem when several cores competes about access to the memory. It is difficult to determine precisely the times that is actually used to receive the required data. In order to fix this issue, and to prevent the processor to compute when there is no data, each interaction with memory is able to halt the processor if need be. The different situations arise when accessing the instruction memory, data memory, and the genetic data pools. Their corresponding controllers are able to set a halt signal before performing the operation. When the one of the *halt* signals is asserted, the program counter and the pipeline registers will be halted, ensuring that the *fitness core* does not change state prematurely.

4.3.6 The Genetic Pipeline

The Galapagos architecture includes a highly specialized pipeline for performing genetic operations. The pipeline is based on the observation that selection, crossover, and mutation works similar for a specific subset of problems. These can therefore be implemented as hardware accelerators constructed for performing one specific task. Constructing such accelerators should give better performance. Designing specialized hardware is usually simpler and thereby more effective than constructing general purpose components. This pipeline will effectively relieve the general cores, the fitness cores, from performing the evolution of individuals. The idea is that these will make the fitness cores able to only focus on the computation of fitness ranking, which is considered computational intensive. In the mean time the *genetic pipeline* can produce new data for ranking. These operations could have been performed by the processor, however, the processor is badly suited for these kind of operations. Note that the instructions in the pipeline actually uses 5 cycles in order to completely propagate through the pipeline. It is far better to only use one cycle in order to complete the one specific operation.

The genetic pipeline is constructed with three types of specialized cores for performing selection, crossover, and mutation. These are operations that occurs frequently in genetic algorithms. These are connected to two internal memory banks on the *FPGA*, namely the unrated and rated pool.

In the selection core, on the other hand, the core needs to be able to access the data bus very frequently in order maximize the throughput. Since fitness calculation can be assumed to be computationally bound, the data bus is more or less available for the selection core. This allows the genetic pipeline to compute the chromosomes required for the next step before they are required by the fitness cores.

A detailed architecture drawing of the genetic pipeline can be seen in figure 4.12

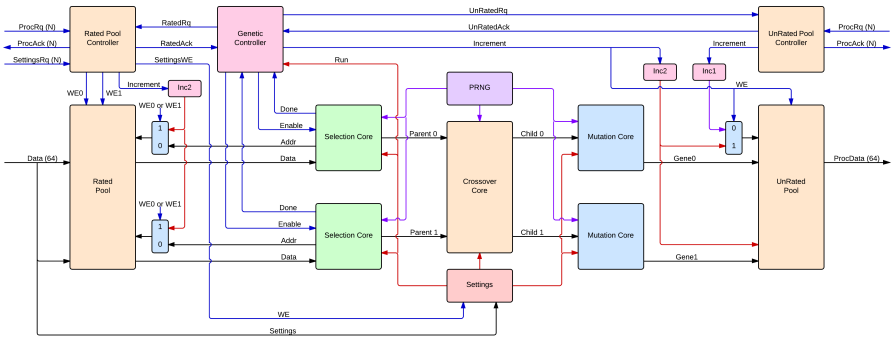


Figure 4.12: Genetic pipeline architecture

Selection Core

The Selection Core is the first of the cores in the genetics accelerator. It is responsible for selecting appropriate individuals from the rated pool population in the genetics pipeline and pass them on to the other cores so that they may operate on them.

The selection core is designed based on a tournament selection algorithm. Tournament selection is a selection scheme that aims to quickly find an individual with a high score from an unsorted list in a way that does not guarantee that the selected individual is the one with the highest score. These goals are healthy goals for a selection algorithm to have when used in a genetic algorithm.

The tournament selection-based selection algorithm is precisely described in algorithm 5 on the next page. In layman's terms, it is designed to select a single individual from a random position in the rated pool. The current best and the random selected is compared to each other with use of a comparator. The best chromosome is stored and used in the next tournament round. After some number of tournaments the current best is transferred to the crossover core. The selection core is responsible for letting the rest of the genetic pipeline know when it can fetch the next chromosome.

```
Data: P = A pool of rated individuals, r = number of rounds in
      tournament (configurable, 0 ≤ r ≤ 31)
Result: A selected individual
begin
    k ⟵ 0
    bestIndividual ⟵ random individual from P
    while k < r do
        individual ⟵ random individual from P
        if fitness(individual) > fitness(bestIndividual) then
            bestIndividual ⟵ individual
        end
        k ⟵ k + 1
    end
    return bestIndividual
end
```

Algorithm 5: The tournament selection used in the selection core.

The selection core is designed with efficiency in mind. The overall time spent in the genetic pipeline must be smaller than the time spent ranking the chromosomes. Note that the fitness cores are connected to the same memory bus as the genetic pipeline. This could potentially lead to a memory bottleneck resulting in starvation. The selection core tries to overcome this fact by reducing the memory access to a minimum. Note that the selection core has reserved the memory bus during the ongoing tournament. This implies that port used by the selection core is unavailable to others during this time. It is designed to not use the memory more than it absolutely have to. For instance, if the current fitness value is greater than the fitness value just fetched, the selection core will not bother fetching the accompanying chromosome. This ensures that the memory resources are not wasted, and is accomplished with an *state machine*.

Since the Crossover Core will need two inputs when performing crossover, two selection cores are implemented on the genetics pipeline.

Crossover Core

The crossover Core is the next part in the genetics accelerator after the selection cores. Two inputs are forwarded from the two selections cores as "parents", and two outputs are the "children" of the inputs, containing bits from both parents. All the bits from both the parents are forwarded in the children, but in some parts the bit-patterns are switched on the children, based a selected crossover function and on a random input from the PRNG. Henceforth this is called crossover.

There are three distinct crossover functions that are implemented: Split, double-split and XOR.

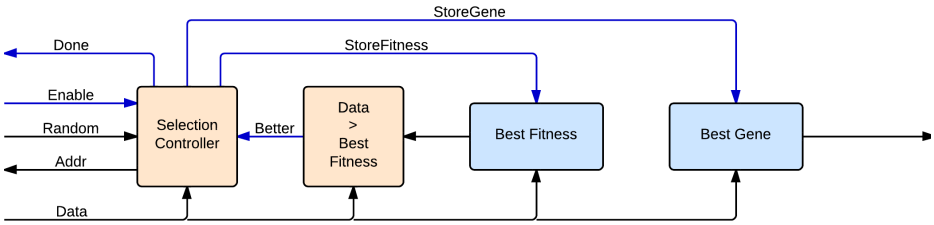


Figure 4.13: Architecture block diagram

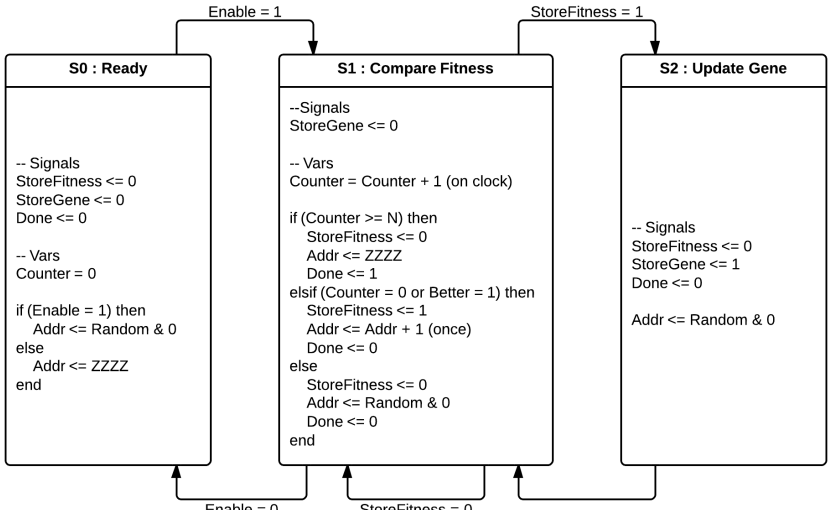


Figure 4.14: Selection core state machine

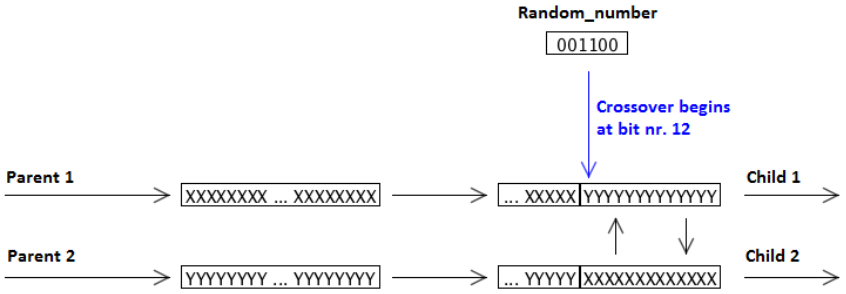


Figure 4.15: Crossover split function

Split Function The first function, crossover split, performs crossover from a selected bit number in the children and until the edge (which is bit number 0). This can be seen in figure 4.15. The values in the parents are represented with X's and Y's, and a single X or Y can have the value 0 or 1, independent of each other. The bit number for starting crossover is based on the value of a 6-bit input random_number, which is provided by the PRNG. This value ranges from 0 to 63. Figure 4.15 uses the value 001100 as example, and the selected bit number is 12. The function will perform crossover on bits 12-0 in the children.

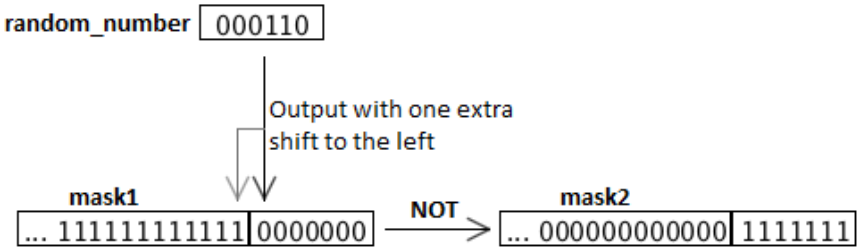


Figure 4.16: Masking for split function

The split function was originally implemented by using behavioural description, but this caused the synthesizer to create many latches. The same problem occurred also for the double-split function. In order to avoid this, the function now uses a ShifterVariable, that uses a 32-bit number of 1's as main input, and the random_number for shifting input. This yields an output that consists of 1's until the selected bit number, and 0's on the rest. Figure 4.16 shows an example where bit number 6 is selected for where to begin crossover. Bits 6-0 in the output consist of 0's, and the rest 1's. This output is called *mask1*. Notice that *mask1* has had an extra shift to the left, in order to perform the crossover from the correct selected bit. The ShifterVariable performs shifting with an input of 1's, and in this example bit number 6 would be the last bit with value 1, but it needs to be the first bit with value 0. Therefore, an extra shift is required. *Mask2* is set as a negation of *mask1*, so the bits in *mask2* would be 1's where they would

be 0's in *mask1*, and vice versa. Only bits 6-0 would be 1's in *mask2* in the figure. The output on the children are set by combining the parents and the masks:

child1 ← (parent1 and mask1) or (parent2 and mask2);
child2 ← (parent1 and mask2) or (parent2 and mask1);

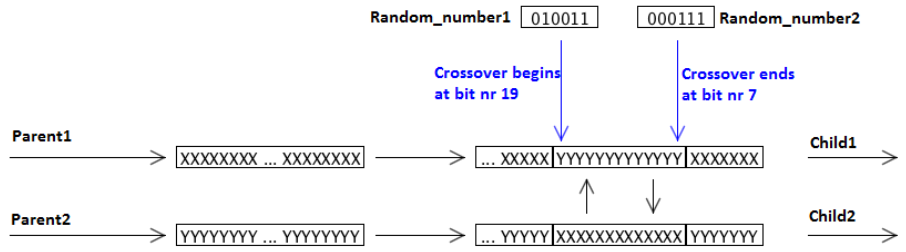


Figure 4.17: Crossover double-split function

Double-split Function The second function, crossover double-split, is similar to the split function, but in addition to having a starting bit for crossover, it also has an ending bit where the crossover ends, instead of reaching the edge at bit number 0. PRNG provides with 2 6-bit inputs, *random_number1* and *random_number2*, which values select the starting bit and the ending bit for the crossover. These values range from 0 to 63, and if both are the same, then crossover will only be performed on one bit.

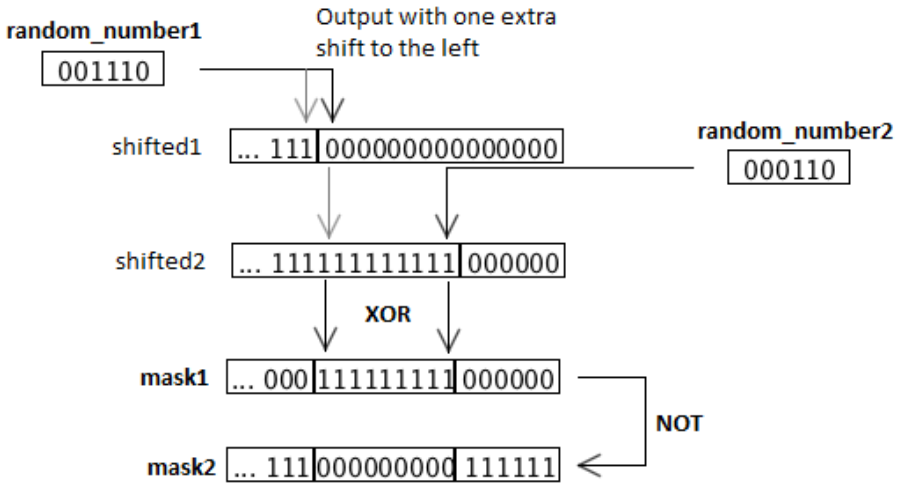


Figure 4.18: Masking for double-split function

The double-split function uses two ShifterVariables, which take an input each from the random_numbers. The ShifterVariables are used in the same way as in the split function, but the outputs will differ from each other as to where

the transition from 1's to 0's are set. One will have more 1's than the other (at least one if both the random numbers are the same). The masks are set by using XOR-function of these outputs, so that *mask1* will have 0's in both the MS and the LS portions of bits, but with a area of 1's between, and *mask2* is it's negation. Figure 4.18 provides such an example, where bits number 14 and 6 are selected. The bits 14-6 in *mask1* would be 1's and the rest 0's by using XOR-function with the outputs from the ShifterVariables, and vice versa in *mask2* by setting it as the negation of *mask1*. The final outputs are set the same way as in the split function.

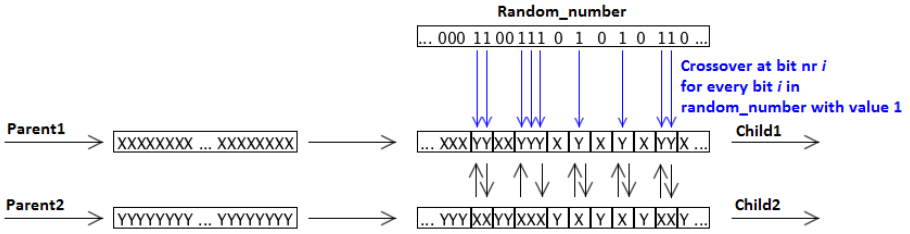


Figure 4.19: Crossover XOR function

XOR Function The third function, crossover XOR, performs crossover bit by bit, based on the 64-bit input random_number. For each bit number i in random_number that has the value 1, the function will perform crossover on the children at the same bit number i. This function is called XOR because of use of XOR-gates in earlier version of the function, and the principle is still the same: For each bit number i in the child, the value will the bit number i from one and only one parent. And which parent it is depends on the value of bit number i in random_number.

Crossover Core Toplevel The crossover core is implemented on the genetics accelerator as a toplevel containing 3 subcores, one for each function, as well as a fourth path with no crossover. In addition to the two parent inputs and 64-bit input random_number, the toplevel has a control_number input used for determining which crossover function is to be used: Split, doublesplit, xor, "party mode" or no crossover at all. Party mode is choosing crossover function at random, based on the 2 LS bits in the random_number. In this way, whenever inputs are sent through the crossover_toplevel, different functions may be used at different times. These are the control values:

- 000 - Split
- 001 - Double-split
- 010 - XOR
- 011 - No crossover
- 1XX - Party mode, in which case these are the random control values:

- 00 - Split
- 01 - Doublesplit
- 10 - XOR
- 11 - No crossover

Mutation Core

The mutation core is the final part in the genetics accelerator. The mutation core takes in a forwarded child from the crossover core as input and may perform mutation on a few selected bits before passing on the result.

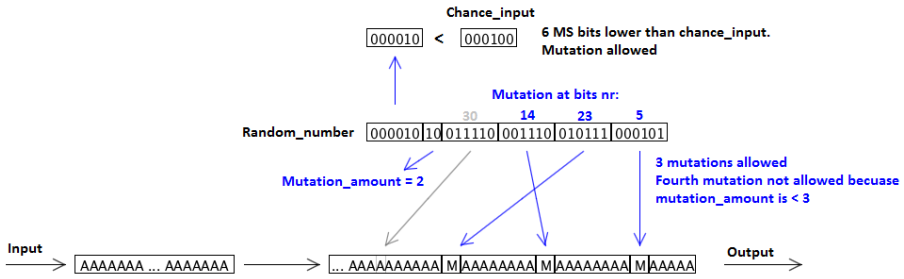


Figure 4.20: Mutation core function concept

In addition to the the 64-bit child, the mutation core also takes in a random_number and a chance_input as inputs. The random_number is used for performing mutation, like allowing mutations, setting how many mutations will occur, and where, while the chance_input is used to configure the chance of it to happen, or how often mutations happen on an avarage level. Default size P for chance_input is 6, and P + 26 for the random_number is 32. As it can be seen in the abstract example in figure 4.20, all bits that are not mutated are represented by an A, and mutated bits are represented with M. The values in each A or M can be 0 or 1, independent of each other. The value M at bit number i is the opposite of the original value A at same bit number i in the input. The P first bits in the random_number is compared to the chance_input, and mutation happens only if the value of these bits are less than the chance input. The example in figure 4.20 uses default size of P, so the sizes of the chance_input and the random_number are 6 and 32. For each different value in chance_input, the user may increase or decrease the chance of mutation by about 1,6%, or $(1/2^6)$. If the value is 001100, then there is 18.75% or $(12/2^6)$ chance for mutation. If the chance input is set to 000000, no mutation will ever happen, and the user may in this way fully disable the mutation core. In the final product, P is set to 8, so the sizes are 8 and 34, and the chance can be increased or decreased by about 0,4%, or $(1/2^8)$ for each value. The next two bits in the random number (bits 25-24) are used to determine how many mutations will happen. There are 4 different values, therefore there can be 1-4 mutations. The next 24 bits are used to determine which bits are to be mutated. 6 bits are used for finding

each bit number. This is similar to what is done in the split and double-split functions in the crossover core. These values are numbered, representing their bit field:

- Nr. 1: 5-0
- Nr. 2: 11-6
- Nr. 3: 17-12
- Nr. 4: 23-18

These are numbered after the amount of allowed mutation. Nr. 1 will always happen when a mutation occurs, while nr. 4 happens only when the amount_number allows for 4 mutations.

Note that if more than one of these numbers point to the same bit to be mutated, the output M will still be the inverted from the original input. For instance, if both numbers 1 and 2 (bits 11-6 and 5-0) have the value 000110, and therefore point at bit number 6, the same mutation will still happen as if only one of these numbers were 000110. If the input bit was 1, the mutated output will be 0, and vice versa. In the example provided in figure 4.20, the 6 first bits of the random_number are less than the chance_input, therefore a mutation happens. Bits 23-0 have the values 30, 14, 23 and 5. Because the value of bits 25-24 is 10 (mutation_amount has value 2), there will be 3 mutations, and the fourth does not occur (though the figure shows where it would have occurred if allowed).

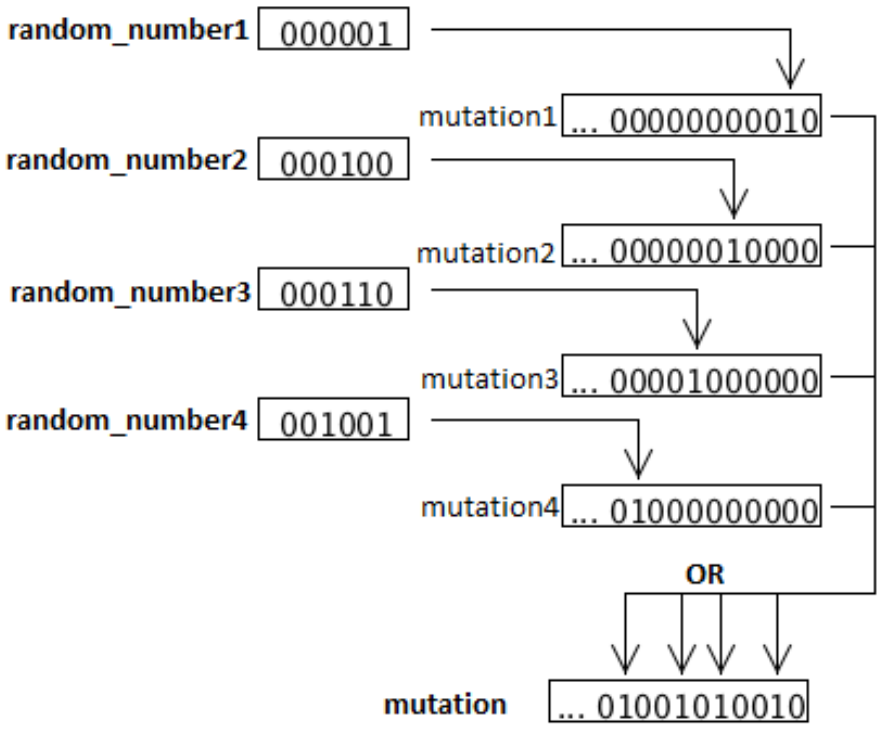


Figure 4.21: Setting mutation

Originally, the mutation core was implemented by behavioural description, but since this caused the synthesizer to generate latches, ShifterVariables have been implemented instead. The mutation core is implemented by use of four ShifterVariables, one for each possible mutation, and set so that only one bit is 1 for the output. A final mutation is set by combining the outputs from the ShifterVariables by using OR-function, and the mutation_amount determines how many of these outputs are combined. Figure 4.21 shows an example where bits 1, 4, 6 and 9 are set for mutation. In this case the final output is set by combining the input and mutation with the XOR-function, so that for each bit i, the bit is set to 1 if and only if bit i is set in either the input or the mutation, but not both. This can be seen in figure 4.22.

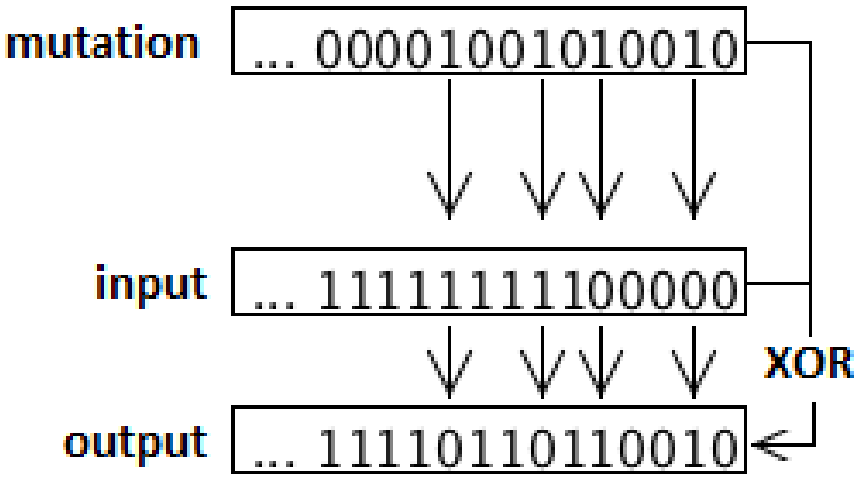


Figure 4.22: Performing mutation

CHAPTER

5

PCB

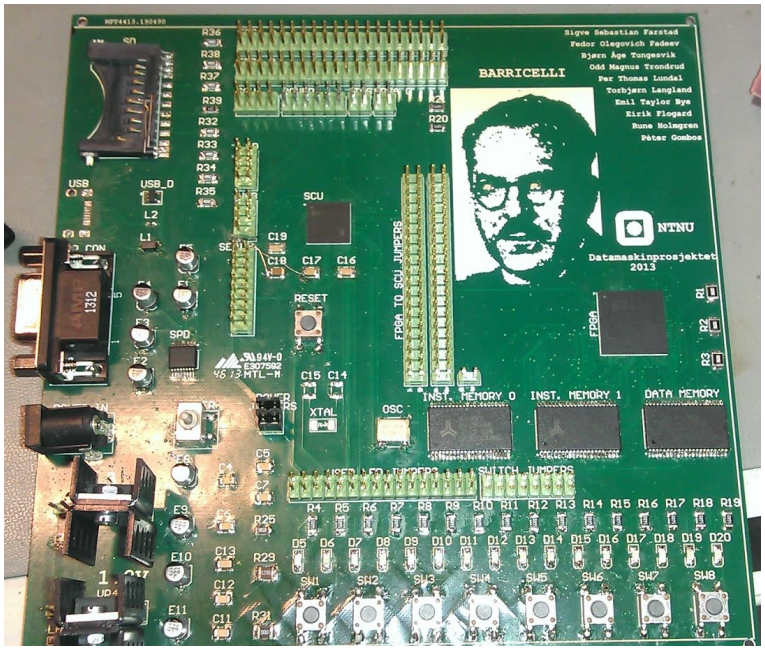


Figure 5.1: The final design of the PCB.

A lot of effort has been invested to make the board as small as possible under the design process in order to make the board fit a small cabinet that was made by the group for the project. While the various cores of the genetics pipeline are

the central components of the Barricelli system, the system itself would be little more than a simulation on some developer board without the printed circuit board (PCB) connecting all the components. Designing the PCB and soldering components onto it are therefore important aspects of the development process of the Barricelli system.

In this chapter the PCB is presented. The design and production processes are detailed. Explanations are provided for why certain components were chosen. And encountered problems and their workarounds are presented.

5.1 Design choices

This section highlights and explains the choices made relating to the hardware of the final PCB.

5.1.1 Field Programmable Gate Array (FPGA)

Per the project's second non-functional requirement 1.2 on page 6, the system's FPGA was required to be one produced by Xilinx. The Spartan-6 family was chosen because development boards with the Spartan-6 LX16 FPGA was available at the lab. Higher-numbered Spartan-6 FPGAs have more resources but fewer I/O ports and a higher price tag than lower-numbered ones.

A Xilinx Spartan-6 LX45 (specifically, the XC6SLX45-2CSG324I) was chosen because of it having a sufficient number of resources and enough I/O ports while sporting a reasonable price tag.

5.1.2 Microcontroller / System Control Unit (SCU)

The EFM32 Giant Gecko 32-bit Microcontroller, was chosen for the project. This microcontroller fullfills the non-functional requirement of using a microcontroller made by Silicon Labs (Energy Micro 1.2 on page 6). In addition there were development boards for the Giant Gecko 32-bit microcontroller available at the lab.

The EFM32GG390F1024-BGA112 was chosen because it was deemed powerful enough to satisfy the performance requirements and had the highest number of available general purpose I/O (GPIO) pins amongst the microcontrollers with the same kind of package.

5.1.3 Communication

The two major components on the system are the SCU and the FPGA. They each fill an important role, and work on essential tasks. The system needs them both to work together, and to accomplish this a communication channel is needed. On Barricelli this was a 41 bit bus. This bus has 16 bits of data, 19 bits of addressing, a small 2 bit bus to control the state of the processor and 3

control signals. The stated bus determine what the FPGA should do with the data, while the control signals tells the targeted unit what to do with the data.

5.1.4 Input/Output devices

This section presents the Input/Output (I/O) devices that were selected and discusses alternatives that were not. An I/O device or channel that can be used to communicate between a computer system and the outside world (or another computer system).

Secure Digital (SD)

The Secure Digital (SD) memory card format was chosen because there were SD cards available in the lab and most of the team's members' laptops had SD card slots. The microSD format was considered, but guidelines on how to use and implement it were scarce compared to information about the larger SD format.

In the SD interface, there are several protocols used for communication. The "Serial Peripheral Interface Bus" transfer mode ("SPI bus mode") was chosen for the project as it allows the microcontroller to communicate with the SD card as if it were a bus.

Universal Serial Bus (USB)

The USB interface was chosen because the chosen SCU has a built-in USB driver, which reduced the amount of work required to implement the standard considerably. USB connectors are prevalent on computers, and every team member's laptop had at least one USB connector. A micro-USB interface was chosen because it was the smallest USB compliant interface available, which meant the associated hardware would take up less physical space on the PCB than its larger siblings'.

The design also includes circuits to prevent undesirable effects like electrostatic discharge, preventing the signals from picking up unwanted background noise, and crosstalk (disturbance of the signal from signals in other circuits). Resistors were also added to prevent short circuiting.

There are several configurations applicable to the USB interface. The USB protocol specifies that there should be one host (or "master") and at least one or more "slaves". The master is responsible for managing the connection to its slaves, and should also be able to provide a 5V current to its slaves if needed.

In Barricelli's case, the microcontroller functions as a self-powered slave. The device that is connected through the USB interface in order to communicate with the microcontroller functions as the master.

The circuit design was copied from the microcontroller's developer's application notes [4, Figure 2.2].

Ethernet

Ethernet support was not added to the system in favor of USB the chosen microcontroller did not have built-in support for Ethernet.

Serial Port/RS-232

RS-232 (colloquially known as a "serial port") is a communication standard which is implemented through a serial port interface. Even though serial communication was not required due to presence of USB interface, the decision was made to implement it. The serial port serves as a backup for the USB as the chosen microcontroller supports communication over RS-232.

5.1.5 Memory

The FPGA group wanted memory that would provide fast accesses in combination with a large input/output capacity in order maximize speed and efficiency. SRAM (Static random-access memory) chips were selected because of its fast access times. Reasonably priced 8 Mbit (512K x 16 bit) modules with 10ns write and read cycle times were found to be suitable and selected.

5.1.6 Crystal

A 32.768KHz crystal was selected to drive the microcontroller. However as the project progressed and the microcontroller's application notes were read more closely it was discovered that it did not need an external crystal to drive it as it had an internal clock. In addition the 32.768KHz crystal was connected to the wrong ports.

5.2 Power supply

As our requirements for the power supply were quite similar to the requirements of earlier projects from the subject. The power supply from the Festiva Lente system was reused in our system. This power supply have been used for many years, with small changes improving the behavior and performance of the power supply. To avoid introducing new problems, reusing this power supply was a safe choice. The Barricelli system does however not require any 2.5 volt or 5 volt power. As a result of this, these parts of the power supply have been removed in our system, and only 12 volt, 3.3 volt and 1.2 volt power is available in our system.

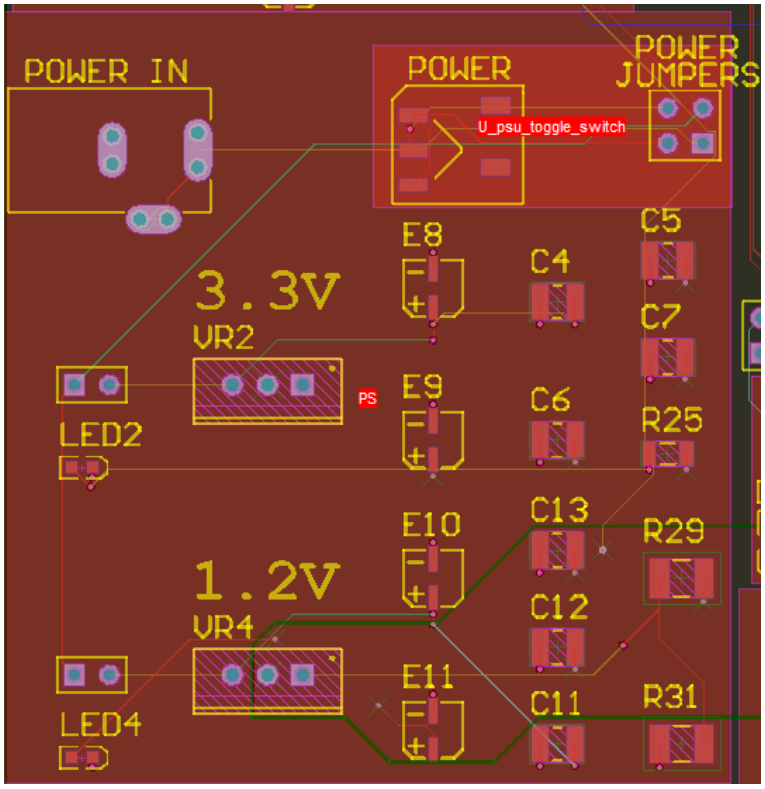


Figure 5.2: Final powersupply design.

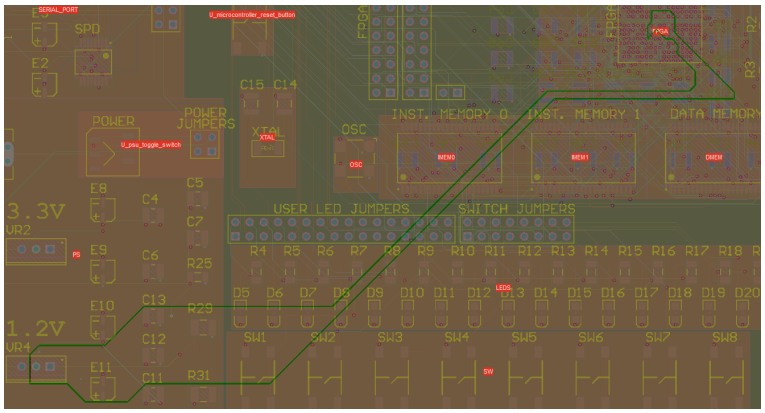


Figure 5.3: Final design of the power plane. The power plane for 1.2V is the long polygon that goes from the "powersupply" part of the board to the FPGA core. The design is also focusing on having as low path as possible to all the components that are using the power grid.

5.3 Power plane

For simpler routing, reduced noise and to reduce voltage drop we have used power nets in this project. As shown in above the PCB have a dedicated layer for power. There is a wide track with 1.2 volts for the FPGA and the rest of the layer is one large 3.3 volt power net for all the other components. In addition to these nets there is a dedicated layer for ground. The reason why we selected the design is done in this way is to provide as short routing path as possible for the sources using the power planes. Keeping a short distance as possible on all signals is important in order to ensure as low loss of effect (measured in Watts) as possible.

5.4 Footprints

This section details how various footprints for components used in the project were obtained.

5.4.1 Obtaining footprints

Once a type of component has been decided upon for a project a specific instance of said component must be decided upon. In order for said component to fit onto the PCB and properly function, its footprint, must be placed somewhere on the PCB. The footprint is a kind of blueprint containing a component's outline and pads.

Obtaining a footprint for a component typically involves creating one manually or using a wizard based on the information contained in the component's data sheet. Some manufacturers make footprints for their components available on

their websites, however they might not be available in a format that is understandable by whatever PCB design suite that is employed in the current project. Altium Designer (version 13.3) feature a browsable database of footprints for various components, all of which can be used immediately in any Altium project.

If a component's footprint is not readily available it has to be created manually. The most important aspect of this process is to obtain the component's technical data sheet and examine it for a description of the component's package and dimensions. This is sometimes labeled as an outline drawing, suggested land pattern (suggested pad size) or package outline. It is important to notice what system the supplied measurements are in, as mixing for example imperial and metric units in a project could lead to unforeseen incompatibilities. Once one gets a hang of Altium PCB editor it takes surprisingly little time to create a footprint.

For components with standardized packages, Altium has an IPC compliant footprint wizard that generates footprints for a component given its package type and some package specific measurements available in the component's data sheet.

The footprint for the polarized capacitor components were designed to match the component's outline, resulting in a footprint barely being large enough to contain the actual component 5.4 on the following page.

This made soldering nontrivial. See 5.5 on page 64.

To avoid this complication the pads on footprint should be designed to be larger than the pins so that the pads protrude beyond the actual component when it is placed on the PCB, resulting in simpler soldering.

Component	**Footprint source**
Headers	Available in Altium's Miscellaneous Connectors
FPGA	Available in Altium Vault
Serial port connector	Available in Altium's Miscellaneous Connectors
Serial port driver	Available in Altium Content Vault
Microcontroller	Available in Altium Content Vault
Power connector	Available in Altium's Miscellaneous Connectors
Memory chip	Created with Altium
LEDs	Available in Altium Vault
Crystal	Created with Altium
Oscillator	Created with Altium
micro USB connector	Available at manufacturer's website
SD card receptacle	Created with Altium
Switches	Created with Altium
Transient voltage suppressor	Available in Altium Vault
Capacitors	Available in Altium Vault
Capacitors (Electrolyte)	Created with Altium
Resistors	Available in Altium Vault

Table 5.1: Footprints used in the project and how or where they were obtained.

The overview over components that we used and the way footprint for each of

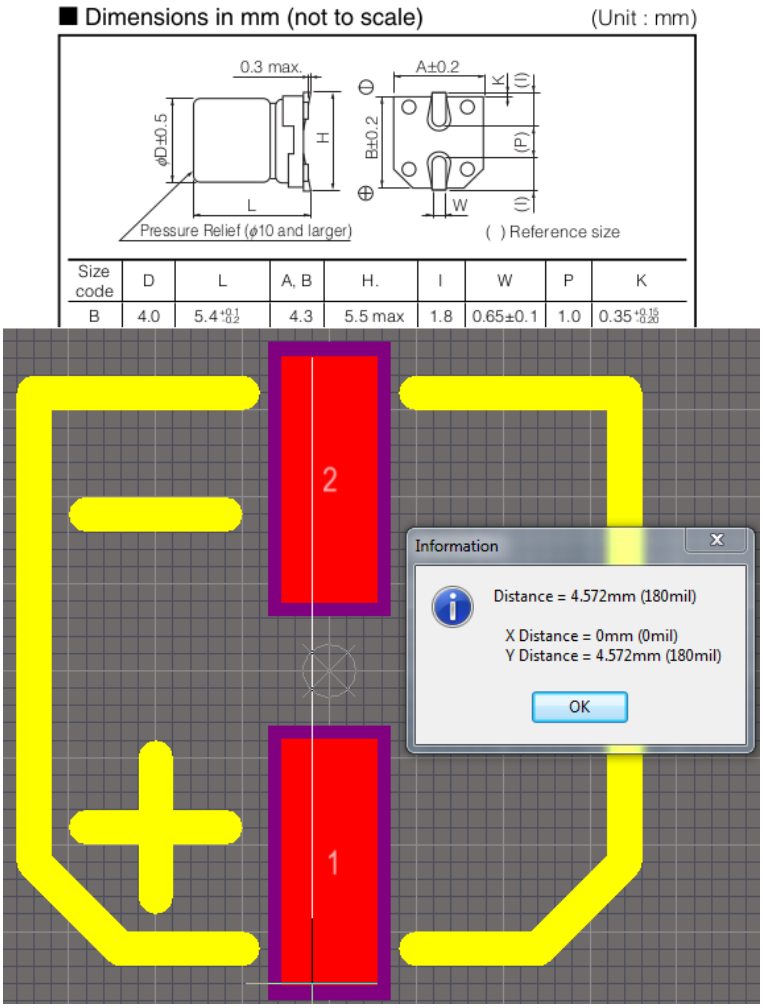


Figure 5.4: Polarized capacitor footprint dimensions

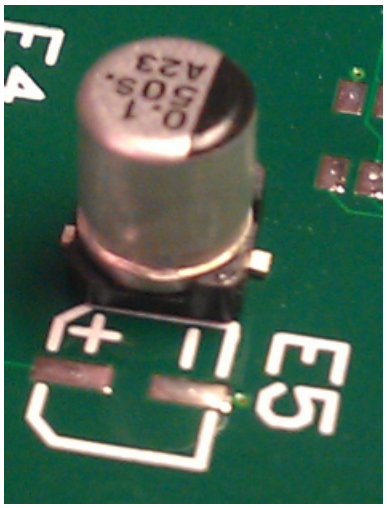


Figure 5.5: Pads on footprint are smaller than pins of component

them was acquired is presented in a table. See 5.1 on page 62

5.5 Budget

The project's budget was 10000 NOK [16]. The Computer Design project's budget was 23000 NOK per group in 2010 and 2011, and was reduced to 10000 NOK per group in 2012[12] while the price of a ham and cheese sandwich from SiT Storkiosk at Gløshaugen has increased from 29 NOK in 2010 to 43 NOK in 2013. Markets are weird like that.

An non-negotionable criteria for all components selected (excepting the FPGA and microcontroller) was that Farnell had enough of it in stock in their UK storage. Generic surface mounted devices (resistors and capacitors) were required to fit the 1206 or 1210 package so that the same footprint could be used for all of them.

Components were required to function at 3.3V or 1.2V. If two seemingly equal components passed the aformentioned requirements, the cheapest one was selected.

The minimum amount of required components to produce one functioning Barricelli system cost 1240 NOK. However, some components could not be ordered in increments of one. Taking this into account, the price of one functioning Barricelli system was 1472 NOK.

The PCB cost 7795 NOK to produce. In total, the price of producing a single Barricelli system is 9267 NOK – 733 below budget.

Approximately 300 NOK of the leftover money was used to construct the 3D-printed case for Barricelli.

5.6 Design Process

5.6.1 PCB design and routing

Here we will talk about how the design was transferred to the PCB, what problems were discovered in this process and how they were solved.

In the planning phase, the group estimated that the design of the PCB and the routing process using the auto-router, would take about 3-4 days. However in reality it took much longer than estimated. The reason for this was that several problems were encountered during the design phase and routing of the PCB. After the first auto-routing run, it was discovered that the auto-router had violated several design constraints for the board. It was then tried to reroute the board several times with different options in an attempt to fix the problem. Since the auto routing process took about six hours on the lab computers, the group decided to use more powerful private computers to perform the routing. This reduced the time it takes for auto-routing from about 6 hours to 1.

After some attempts with the auto-router, the group decided to manually route the last signals in order to fix the constraint violations. This was a time consuming process, but during this, several serious design flaws were uncovered which would have cost more time due to the need to produce new boards. Among the errors that were discovered, was that the footprint for the microcontroller was wrong: the diameter of the ball pads on the footprint were larger than its data sheet recommended they be. Several of the capacitors were also unconnected, or just connected in a wrong way. This means that even though the manual routing of the PCB took longer, the discovery of the design flaws probably saved us some time in total.

5.6.2 Soldering

Here we will talk about the soldering process and how we worked with that. This will not cover major problems that needed a workaround (They are covered in their own chapter), but rather the challenges we experienced in the soldering process.

Due to numerous delays in the design of the schematics and the routing of the PCB, the group had to complete the soldering process as fast as possible. In order to do this effectively, the group coordinated the work in shifts so that people were working on soldering the PCB both day and night time.

There were also some problems encountered during the soldering process. The most significant problem was that it was discovered that we received voltage regulators instead of the microcontrollers that were needed. This caused some delay to the soldering process.

Also the ordering of the needed components were also done in many turns instead of one large, single order. This happened because the components list were not always updated when the schematics changed. The result of this were that too few components were ordered. First after all the components

that was needed were ordered, it was discovered that there was functionality in Altium that could be used to generate component lists. Using Altium instead of manually updating the component list could probably have saved some time.

5.7 Problems and workaround

Here we will talk about various problems we discovered during the soldering process, and how we found ways to workaround them. We expect that there will be some things that may be possible to work around in code on the microcontroller, and other parts that require hardware fixes. There might also be problems that cause parts of the board to not function.

5.7.1 Power connector footprint

The footprint of the power connector had three pairs of holes instead of a milled groove. This caused the connector to not fit in the footprint. This was however solved by cutting away the parts of the connector that did not fit on the PCB using pliers. The result worked fine, and it's hard to spot that the power connector is modified if you do not have a correctly mounted connector as a reference.

5.7.2 FPGA to SCU bus routing

Because of an error made during the routing of the board, the header pin for FPGA_ENABLE is not connected to any FPGA pins. This error can be corrected by using one of our spare FPGA lines available on headers. A wire was pulled from FPGA_HEADER78 to the header from the SCU, and this header cable allowed us to run the rest of the bus as planed.

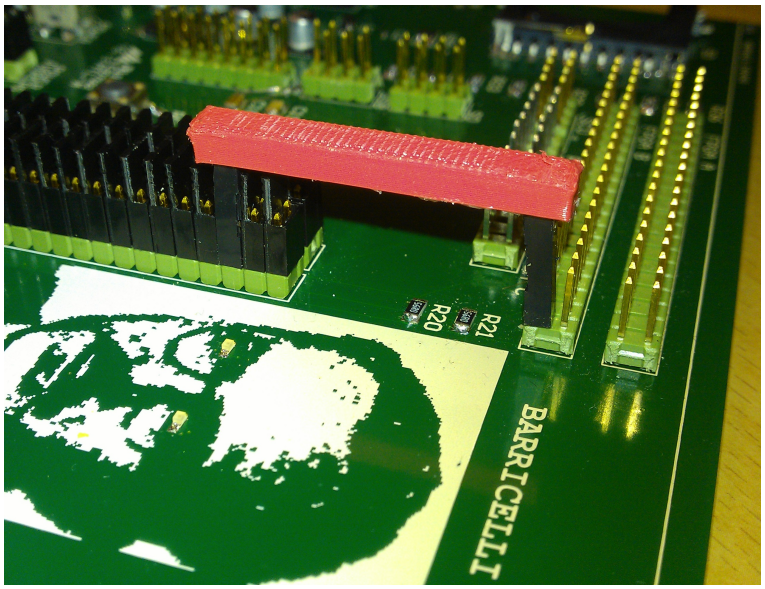


Figure 5.6: For aesthetic appeal we 3D printed a bridge for the wire to run in

5.7.3 USB port

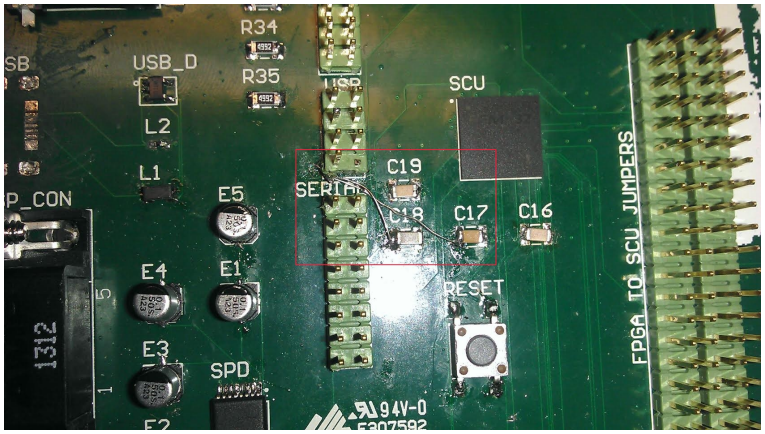


Figure 5.7: The figure shows the "hack" that were made on the PCB in an attempt to fix the USB. Sadly the board were accidentally shortcircuted and died before this the hack could be fully verified to be working

When the PCB came back from production, it was discovered that the USB was not connected to the microcontroller according to the recommended specifications from the manufacturer of the microcontroller. The problem was that the signals USB_VBUS and USB_VREGI were not connected to the VBUS-pin on the USB receptacle. This was fixed by soldering copper wires on the capacitors designated as C18 and C17 to USB-HEADER2 (called VBUS_ENABLE in the

schematics). According to tests performed on the PCB before and after this work-around, the problem was fixed successfully.

5.7.4 Oscillator

Measuring the pins between the oscillator and FPGA shows that the oscillator does output a signal, however the FPGA does not seem to receive it. No workaround or fix was found.

CHAPTER

6

INPUT/OUTPUT

6.1 Input and Output

The PCB contains a microcontroller used to manage all input and output between the FPGA and the IO devices shown in Figure 3.1 on page 18. The microcontroller listens on all IO channels for input, and acts on the input, either forwarding the request to another device or performing memory operations on the FPGA's memory.

6.1.1 Initial requirements

The assignment required a microcontroller to handle IO for the FPGA. To minimize the amount of things that could go wrong, much of the initial work was focused on finding a few reliable and relatively simple data connections.

Specifically, the microcontroller was required to be able to put some program and data on the FPGA's memory, and then later output values from the data memory through the proper communication lines. The I/O devices together with the microcontroller and it's software should be able to provide a reliable and stable I/O connection between the outside world and the FPGA.

6.1.2 Communication channels

SD Card

The SD card reader is primarily used as a storage for programs that are to be uploaded on the FPGA. However, it might also be used to store memory snapshots in order to look how the genetic algorithm converges to a solution over time.

The Energy Micro Application Note on Fat and SD cards, and its example code, describes an implementation of the FatFS library on the Giant Gecko microcontroller.[1]

FatFS [9]

FatFS is a generic FAT file system for microcontrollers, with a generic interface for the FAT operations, and a hardware specific interface for disk I/O. Because of this structure, the system is easily portable. To add read and write a FAT system on some disk drive, FatFS needs the following functions:

disk_initialize	Initialize disk drive
disk_status	Get disk status
disk_read	Read sectors on disk
disk_write	Write sectors on disk
disk_ioctl	Control device dependent features
get_fattime	Get current time for FAT

Table 6.1: Overview of disk I/O functions

USB

The USB is the main communication line with a host computer, allowing the host computer to start running programs on the FPGA and receive snapshots of the memory periodically. The microcontroller has a built in USB controller [8] and energy micro has supplied an application note [5] with code for utilizing the included USB controller in order to act as a USB device.

Serial

The serial port is meant as a backup solution in case USB doesn't work, with the exact same opportunities, but with an older, simpler interface. The microcontroller used in the project has a built in UART Receiver/Transmitter[8] which is easily activated with code from AN0045 [3].

LEDs and buttons

The most primitive form of IO we have are the on-board LEDs and buttons. They allow a quick and easy way to verify that a program is running, and

possibly letting the user change execution modes or the program on the FPGA with the buttons. All code interfacing with the LEDs and buttons are simple code either setting or reading the value of GPIO pins. The LEDs are driven by General Purpose IO pins on the SCU, requiring a minimal amount of code in order to get a working output, which is especially handy in the early stages of implementation.

FPGA

There are 41 wires between the FPGA and the SCU in order to facilitate communication (see Table 6.2 for a complete list of all the connections). The FPGA has no way of signalling that it wants to output something, so the SCU is responsible for periodically halting the CPU on the FPGA and reading from it's memory.

J-link

In order to program and debug the programs on the SCU, we utilize the built-in pins for debugging using J-Link™as described in AN0043 [2]. It can also be used as a form of last resort emergency output as it makes it possible to display text that is printed by the program running on the SCU.

6.2 FPGA Control

The only way of communication with the FPGA is with direct memory access to the FPGA's data and instruction memory. All the data is transferred directly over the SCU's GPIO pins, without any form for memory mapping or built-in bus interfaces. This is mostly due to the fact that we did not do enough research early in the design process and recognized that we could use something like External Bus Interface to access the memory.

Access to the FPGA's memory is controlled by the signals seen in Table 6.2. It should be noted that there are two states to access the FPGA's instruction memory, the upper and lower half. This is because the instruction memory stores 32 bits per address while the SRAM chips only stores 16 bits per address (see Section 4.3.1 for more details on the instruction memory).

The SRAM data sheet [6] specifies that the data signal has to be stable for at least 10ns in order to complete a write. This means that it is not necessary to worry about timing when accessing the SRAM since changing the signal more than every 10ns requires a clock speed of 100MHz since we can at most change the output of a single pin every cycle.

Signal	Bus width	
FPGA enable	1	Enables the FPGA on high, disables it on low
FPGA State	2	00: Processor enable 01: Instruction memory upper half access 10: Instruction memory lower half access 11: Data memory access
Chip enable	1	The chip enable signal in to the selected memory block.
Write enable	1	The write enable signal in to the selected memory block.
Address	19	The address bus to the selected memory block.
Data	16	The data bus to the selected memory block.
LBUB	1	The LB and the UB signal to the selected memory block.

Table 6.2: Lines between the SCU and FPGA

6.3 IO Program

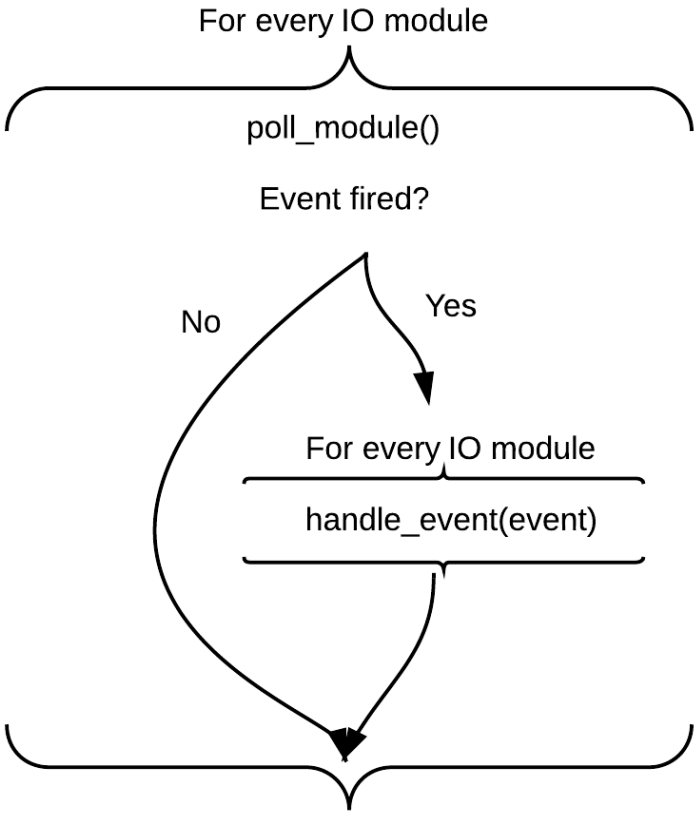


Figure 6.1: The body of the IO program's main loop

The IO program was designed to be as simple as possible in order to decrease the amount of things that could go wrong. The main idea is that every IO device is required to define two functions in order to be used: a function to poll for input and a function that is called whenever a device reports input.

In order to enable sending messages between different IO units, the poll functions return a pointer which may point to any object in memory, which allows other modules to read the data given that they know what type of data the pointer points to.

6.4 Design decision

This section contains a few key design decisions that were made during the process which have had an important impact on the project. This section mainly contains design decisions for the software on the SCU, for hardware design decisions see Section 5.1.

6.4.1 Operating system

Early in the process, a discussion arose about how it could be beneficial to run an operating system on the microcontroller such that familiar programs could be run directly on it. A scenario pitched was to have network access, and then be able to talk to the machine remotely using programs such as *SSH* or *telnet*. However, the Linux distribution available for the Energy Micro microcontroller was found lacking in the features we wanted, and the microcontroller lacks network support. It was therefore decided that running an OS was unnecessary as there were few rewards and little to gain from it.

6.4.2 FPGA Communication

During the initial design phase, the link between the FPGA and SCU was designed to be as simple as possible. The final version was the the 41 wires mapping all the signals needed for directly accessing the SRAM chips.

In retrospect, using the built-in EBI on the microcontroller would have made the job of accessing the FPGA's memory easier, letting the microcontroller map the contents of the different chips automatically to different memory regions. The main reason for not designing with EBI in mind was lack of knowledge of what it could offer in terms of ease of development. Another reason for not looking more into it was the fact that since we had a large enough bus, the code for communicating with the SRAM chips was quite simple, with the microcontroller not being fast enough that timing should be an issue.

6.5 Issues

6.5.1 Crystal

In the design phase it was decided to go with just a single high frequency crystal oscillator. Unfortunately the crystal that was selected had a clock frequency in the kHz range, instead of the MHz range, which was what was required to drive the high frequency crystal port. Luckily the microcontroller has a built-in RC oscillator, so the crystal oscillator was not essential to get code running on the microcontroller. A higher I/O throughput could have been achieved with the increased frequency the crystal oscillator would have given, but adding a high frequency crystal would have required a new PCB card.

6.5.2 I/O units failing

As written in Section 5.7.3, the USB never got working on one of the PCB boards. While falling back to UART, the code running on the microcontroller seemed to be pushing data out to the UART circuitry, but there did not seem to be any signal going out through the cable. Before managing to find a solution, yet another PCB board failed due to an accidental short circuting of the board and it was decided to rely on the debug link to send data to the computer as the deadline approached.

The SD card was never finished, as the FatFS example code for Micro SD never ran with the SD cards on the PCB. When further testing was to be done, the PCB failed, and as with the UART, it was decided that the debug link would suffice.

6.5.3 FPGA Memory access issues

At first, communicating with the memory on the FPGA seemed fine. However, it soon became obvious that we were not able to reliably read or write to the memory. Writing to and then subsequently reading the entire memory showed that we were either not able to successfully write to and/or reading the entire memory.

The first thing that was tried to fix this was to increase all the delays on reads and writes, to allow signals to stabilise, but this showed little improvement. Checking the signals sent with a logic analyzer showed that the memory should have enough time to update before the write signal was disabled.

The write routine was also updated to write the memory and then immediately reading the same adress to verify that the correct data had been written, redoing it if it hadn't. This did not fix out problems, as reading the memory later gave different results.

In order to verify that reading the memory did not work, the FPGA was flashed with a program already in it's memory. Reading the entire memory showed that while we managed to correctly read many of the addresses, some still gave wrong data.

CHAPTER

7

ADDITIONAL COMPONENTS

7.1 Galapagos Assembler

The Galapagos Assembler is an assembler for Galapagos Assembly that was written for this project. It assembles Galapagos assembly to be run on the programmable fitness cores of the Barricelli computer. The assembler is written in Python. It is designed with a modular, object-oriented software architecture, which makes it easily extensible and modifiable. Indeed, during the short time it has been published on the Internet, it has already been forked and adapted for use for other instruction set architectures and assembly languages. The assembler supports the entire Galapagos instruction set.

With a an assembler available, opportunities for performance optimizations through instruction re-ordering are made available. The idea is that instructions within a simple code block, i.e. a branch-less block of consecutive instructions with no labels, instructions may be carefully re-ordered to minimize data hazards. The assembler does not perform these instruction re-orderings. This is because the forwarding unit in the processor architecture already resolves many of the same issues that the assembler would work around using instruction re-ordering.

The only non-control related hazards that the forwarding unit doesn't already resolve are use-after-load conflicts. A use-after-load conflict is a conflict where the processor plans the execute a data load from memory, and then use the result from that load in the immediately proceeding execution. When this happens, the result from the memory load is not yet ready when the execution is planned to execute. This hazard is resolved off-line by the assembler. It can detect use-after-load hazards during assembly, and will insert a nop between the load

and use instructions, forcing the processor to wait until the data is available.

Galapagos Assembler is available in PyPi, the leading python package index. This means that it is easily installable for end users using `pip`, the python package manager. Installation is as simple as running `pip install galapagos-assembler` in a terminal where pip is available.

The source code of the Galapagos Assembler can be found in appendix D on page 190.

7.2 Case

Figure 7.1: Case

To give this project a nice presentable finish and professional appearance, the group decided to make a case for the computer. This was also a good opportu-

nity to develop our skills with 3D modeling. Because of this we chose to create the case using a 3D printer.

7.2.1 Design

It was early decided that the case should have all of the features of the real board. This would ensure that the user would be able to operate the computer without ever feeling the need to take the board out of the case. This is why the case have all of the user controlled LEDs and buttons from the PCB on the case itself, and the buttons and LEDs are available from the board from headers. The 16 LEDs are placed on the front panel, while the buttons are placed on a small keyboard that slide out on a drawer in front of the case. The keyboard drawer is loaded by a rubber band that keep it shut, while the mechanism shown here will keep it open when the user have pulled it out.

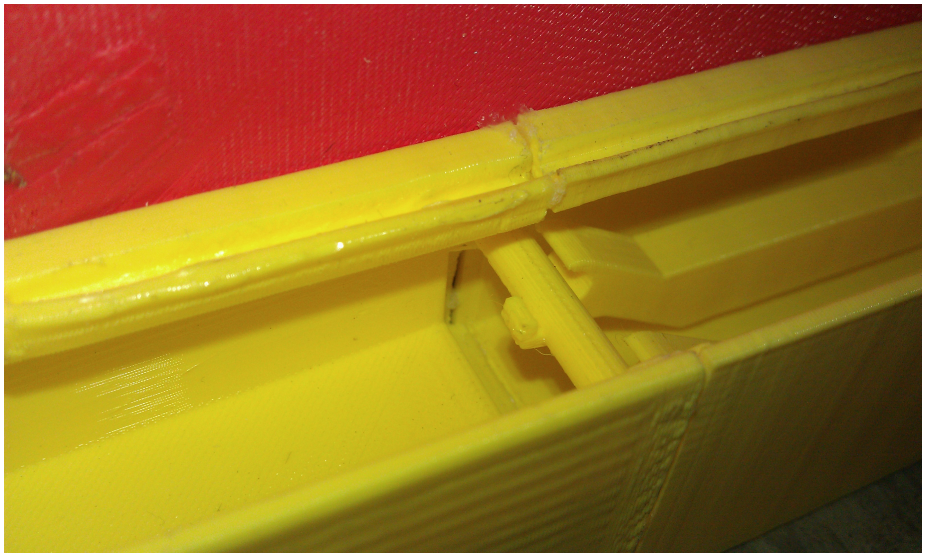


Figure 7.2: Keyboard mechanism

To access the boards IO ports the case is open in the back. Here the user will be able to pug in any of the IO devices, and also view the status of the two power net LEDs. This is also where the user may slide the board in and out of a tight track that hold the board.

The sides are decorated by two side panels. On the left side the projects name, Barricelli, is written, along with the names of all the group members. On the right side there is a picture of Nils Aall Barricelli. Both these side panels were printed laying flat to get a smoother print and higher resolution. Because this project have a focus on performance, we made these side panels red. This is because red is well known as the fastest color [22].

The board dissipate heat from the power supply, and this heat must get out of the case. To make the board run cool enough in the case the top of the case is

perforated with 42 ventilation holes.

To decorate the case, an NTNU logo was put on the front panel between the user LEDs and the reset button and power LED.

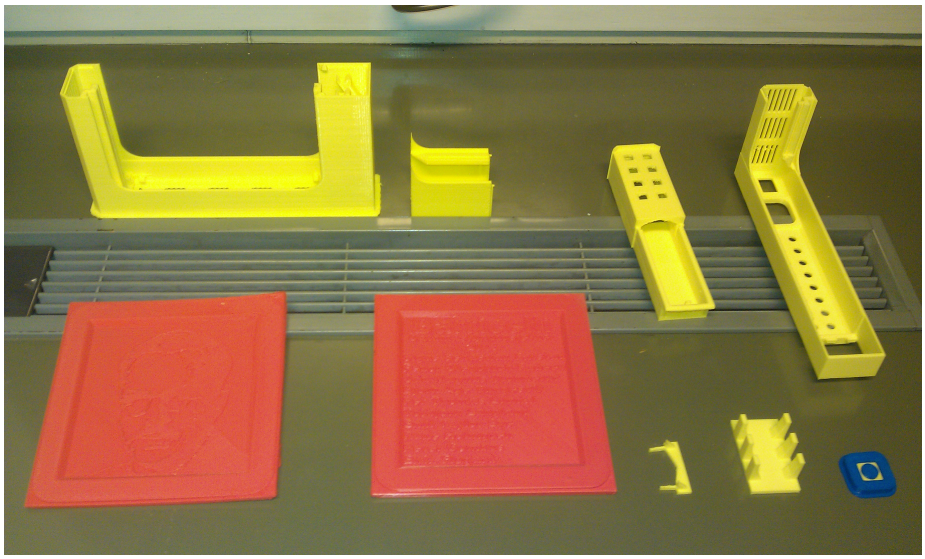


Figure 7.3: Case before assembly. (The two parts of the NTNU logo are already assembled)

The case is made up of 10 3D printed parts, 16 user controlled LEDs, a power LED, a reset button, a circuit board with 8 user readable buttons and a whole lot of wiring and resistors. All of this make a compact case of $20x21.5x4cm$ full of useful features.

7.2.2 Tools

- MakerBot Replicator 2 Desktop 3D Printer
- MakerWare
- Google SketchUp
- Fine grade sand paper (P240)
- A variety of pliers, knifes and pincers

7.2.3 Problems and workarounds

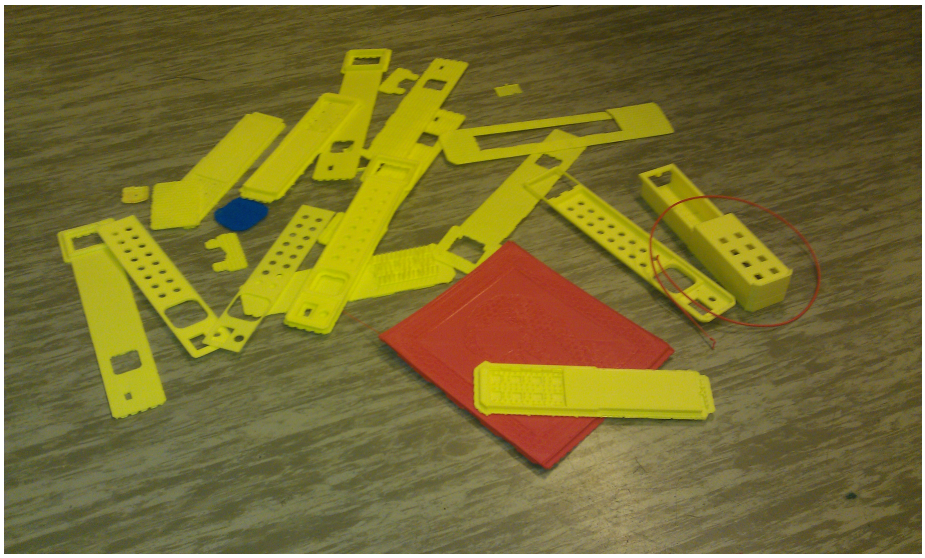


Figure 7.4: Some of the failed prints

3D printing is a new technology, and compact desktop 3D printers like the MakerBot 2 is even newer. As with all other new technology there are problems. 3D printers run into more problems than even normal printers do. Every time you print a job there is a risk that something will go horribly wrong. The print may have an error in the compilation, or maybe the PLA plastic filament get tangled or just stop feeding. In addition, IDI's 3D printer is in horrible shape. It have got a broken nozzle cooling fan, a partially broken printer head fan and a partially broken PLA filament feeding mechanism replaced with a new 3D printed part.

Because of all these problems, each of the big jobs had to be ran many times before the parts were good enough to be used.. The parts were not all as expected, but with limited time, and jobs that took up towards 20 hours to run they had to do. The parts that had imperfections were fixed through various hacks.

Side panel with Barricelli

One of the corners had not stuck to the surface during the printing operation. This may have been caused by the broken nozzle fan. The result was that one of the corners of the panels was warped by about 5 mm where it was supposed to be flat. It was however possible to get it almost flat using a heat gun and a hammer. After that a knife was used to cut away the parts that was supposed to keep the panel in place. Instead of these parts to keep it in the case, a lot of glue was used.

Keyboard tray

Because of imperfections in the printing process, the keyboard was not able to slide as intended. This could have been solved by designing it a bit smaller, but the group had almost no experience with 3D printing, and did not know this would be a problem. It was however solved by sanding it down with fine grade sand paper.

Keyboard tray stopper

The keyboard tray stopper was a bit too wide and tall to actually fit in the case. This part was small, and could quickly be reprinted. The problem could also be solved even quicker by cutting of a few millimeters with pliers, so we did that. The part will not be visible anyway, as it's role is purely functional.

Front of the case

The front of the case was the second larges part of the case. Because of the complexity of the job, there was a high risk of failure during the job. After 6 or 7 failed prints the printer had a print that ran about 50% of the job before failing. The piece of the part was good enough to be used, and we decided keep that piece, and print the rest as a separate job. This meant that we had to have an extra glued seem in the completed case, but time was running out so we did it anyway. The second part of the piece printed fine, and apart from the visible seem it worked out OK.

Back of the case

The back of the case was the largest and most complex part of the case. To make things worse, it had to be printed in one piece to actually work. This meant that the printer had to run almost 20 hours non-stop without failing After many failed prints the printer was abandoned, and we asked a company at NTNU School of Entrepeneurship to borrow their identical, but functioning printer. Their printer had some minor problems, but the job completed fine at first try.

Part III

Results and Discussion

CHAPTER

8

TESTS

When designing a computer with a custom architecture from scratch, it is important to continually test and evaluate the correctness of the solution at all possible stages, to ensure that final product is a success. This section documents and explains the rationale behind the different types of tests that have been performed.

8.1 Testing the Processor

Barricelli's processor has been tested at four different levels: VHDL-based unit test simulations of the different subcomponents, VHDL-based integration test simulations of each processing unit, VHDL-based system test simulations of the entire system interfacing against a mock SCU and mock memory, and finally physical integration tests of the processor programmed onto the FPGA of the Barricelli.

8.1.1 VHDL-based Subcomponent Unit Test Simulations

Unit testing VHDL entities is extremely important in a large and complex design like the Barricelli. For this project, almost every component, perhaps except the most trivial entities, is tested in an automated or semi-automated VHDL test bench. A tool was developed to ease the automation of VHDL test running and validation, modeled after the leading test runners in the software industry, such as JUnit[10] and Karma[11]. This tool enabled tests to be written using easy-to-use self-evaluating tests that compare signals at specific times against expected values.

The goal of these unit tests is to ensure that the building block components work as expected when reacting to specified input.

Screenshots of simulations of these tests can be found in Appendix F.

Fitness Core Components

What to test:	Test if all the ALU functions work as expected.
How to test:	Perform an automatic unit test where all the ALU functions are tested at least five times under different circumstances. This is written in a test bench and verified with Isim.
Pass criteria:	All the performed calculation should be as expected.
Results:	Successful. All calculations is able to calculate the correct result in every case.

Table 8.1: ALU

Genetic Pipeline Components

What to test:	Test if the selection core behaves as expected.
How to test:	Connect the selection core to a BRAM block filled with fitness values and induviduals, and a PRNG. Then set the counter to N and the enable signal high. Finally, wait for the done signal to become high.
Pass criteria:	The selection core shuld read N random fitness values from the BRAM. When it reads the first value or a value greater than the currently stored value or the first value it should store the individual at the next address. After it has compared N values, it should set done high and output the fittest individual.
Results:	Successful. The selection core compares exactly N fitness values and only stores the individual if the fitness value is greater than that of the previously best. Finally it sets done high and outputs the fittest.

Table 8.2: The selection core

Selection Core

Crossover core .

What to test:	Check if crossover split function performs crossover correctly, from correct bit
How to test:	Changes in any input should cause change in the outputs. Therefore parent inputs and random_number will be changed during test.
Pass criteria:	The output for child1 should have output from parent1 and child2 from parent2 before crossover point, and child1 should have output from parent2 and child2 from parent1 after crossover point. The starting point, which is the first bit in the crossover, should always be the bit number equal to the value of random_number.
Results:	Successful. Changes in parents cause expected changes in children, and starting point for crossover is always equal to the value of random_number

Table 8.3: Crossover Core Split function

What to test:	Check if Crossover Double-Split Function performs crossover correctly, from correct starting bit to correct ending bit
How to test:	Changes in any input should cause change in the outputs. Therefore parent inputs and random_numbers will be changed during test.
Pass criteria:	The output for child1 should have output from parent1 and child2 from parent2 before crossover starting point and after ending point, and child1 should have output from parent2 and child2 from parent1 between the crossover starting point and ending point. The random_number with the highest value should always be the starting point, and the one with the lowest value should always be the ending point. These points, which are the first and the last bit in the crossover, should always be the bit numbers equal to the value of the random_numbers. If both have same value, then only one bit location will have a crossover
Results:	Successful. Changes in parents cause expected changes in children, and starting point for crossover is always equal to the value of the highest random_number, and ending point for crossover is always equal to the value of the lowest random_number

Table 8.4: Crossover Core Double-Split function

What to test:	Check if Crossover XOR Function performs crossover correctly, from correct starting bit to correct ending bit
How to test:	Changes in any input should cause change in the outputs. Therefore parent inputs and random_number will be changed during test.
Pass criteria:	The output for child1 should have output from parent1 and child2 from parent2 for each bit i, where in the random_number the value is 0, and child1 should have output from parent2 and child2 from parent1 for each bit i, where in the random_number the value is 1.
Results:	Successful. Changes in parents cause expected changes in children, and for each bit i in the random_number, there are crossover at same bit i from the parents to the children.

Table 8.5: Crossover Core XOR function

What to test:	Check if Crossover Toplevel selects correct crossover function based on control_input, and random_number when in "Party Mode"
How to test:	Every input of control_input will be tested. Changes in random_number will be done with focus on the 2 LS bits when control_input is "1XX", and in party mode.
Pass criteria:	When control_input is set to 000, or 1XX and random_input-bits to 00, crossover should be split with the value of the 6 LS bits from random_number used for starting point. When control_input is set to 001, or 1XX and random_input-bits to 01, crossover should be double-split, with the value of the 12 LS bits from random_number used for starting and ending point. When control_input is set to 010, or 1XX and random_input-bits to 10, crossover should be xor, with crossover on every bit numbers that are 1 in random_number. When control_input is set to 011, or 1XX and random_input-bits to 11 there should be no crossover at all, and output children should be equal to each their input parent.
Results:	Successful. Each value in control_input was tested, and set the expected function. When set to 1XX, every value on the 2 LS bits in the random_number was tested, and set the expected function.

Table 8.6: Crossover Core Toplevel

Mutation core .

What to test:	Check if Mutation Core selects mutates when allowed, mutates the correct amount of bits, and the correct bit numbers, all based on chance_input and random_number.
How to test:	Changes on input will change output. Therefore input will have changes. Changes in random_number and chance_input will be done with focus to test allowing or denying mutation. Changes in random_number will also be done to test amount of allowed mutations, and to test selecting the locations of the mutations
Pass criteria:	When the P first bits in random_number is equal to or higher than chance_input (size P), there should be no mutation at all. When mutation is allowed, the next two bits should allow these amount of mutations: 1-4 depending on values 00-11. Bits 23-0 select four bit locations for mutations, and the output should have opposite value on these locations compared to the input. If more than one bit location pointer has the same value, the same bit location should still have the mutation on the output.
Results:	Successful. Mutation is allowed only when the P first bits are lower than the chance_input, the correct amount of mutations were set and each four bit locations were selected correctly as expected by bits 23-0

Table 8.7: Mutation Core

8.1.2 VHDL-based Processing Unit Integration Test Simulations

Each processing unit, which each consists of several interconnected subcomponents, has been simulated for integration testing. The goal of these tests are to verify that the different subcomponents interface correctly with each other, and that the behaviour of the supercomponent is as expected.

8.1.3 VHDL-based System test Simulations

The toplevel simulation test bench of the Barricelli computer, which simulates the entire FPGA as a black box interfacing against the external components, supports pre-loading entire programs into a mocked instruction memory component. The Galapagos Assembler supports outputting assembled programs compiled to one of these mock memory components, meaning that testing new programs in a simulated environment is an easy and fun process.

A formal description of the system tests performed at this level can be found in tables 8.18, 8.9, 8.10, 8.11, 8.12, 8.14, 8.13, 8.15, and 8.16.

What to test:	Observe that RRI and RRR instructions propagate correctly through the pipeline, and produce the correct result.
How to test:	The program in listing F.2.1, consisting of both RRR and RRI instructions, are loaded into memory with the test framework. The execution of the instructions are observed with ISim.
Pass criteria:	Register r1 should contain $0xBA1212ICECC1$ and register r2 should contain $0xBA1212ICECC1$
Results:	The contents of register r1 and r2 are according to the pass criteria.

Table 8.8: RRR and RRI instructions

What to test:	Check if the branch address is calculated correctly, and an conditional jump is performed to this address.
How to test:	The program in listing F.2.1, is loaded into a test bench. This simple program consists of a simple loop performing some arithmetic operations that store values to registers. The execution of the program is simulated with ISim to verify the result
Pass criteria:	The branch is taken. The instructions located in the $fetchstage$, $decodestage$, and $executestage$ are flushed. The result in register r1 equals 1. The r1 register is not incremented.
Results:	Register r1 contains the value 1. The instructions in $fetchstage$, $decodestage$ and $executestage$ does not perform any changes to the register file. Thus the value in register r1 continuous to contain the value 1.

Table 8.9: Branch taken

What to test:	Check if the conditional jump is disregarded when performing conditional that always evaluate to false.
How to test:	The program in listing F.2.1, is loaded into a test bench. The simple program consists of conditionals that evaluate to false. The execution of the program is simulated with ISim, and the results are verified.
Pass criteria:	The conditional jump is not taken. The values of r1 should be incremented until reaching the value of 4.
Results:	Register r1 contains 4 after end of simulation.

Table 8.10: Branch not taken

What to test:	Check if conditional instruction are executed when they always are evaluated to true.
How to test:	The program in listing F.2.1, is loaded into test bench. The simple program consists an *ADDI* and an conditional *ADDI* instruction that always evaluate to true. The execution of the program is simulated with ISim, and the result is verified. More specifically, the content of register r1 are verified.
Pass criteria:	The first *ADDI* results in the r1 to be incremented with 1. The second instruction, the conditional, increments the value of r1 to 2.
Results:	The contents of register r1 is 2. This proves that the conditional instruction was executed.

Table 8.11: Conditional instruction executed

What to test:	Check if conditional instructions are executed when they always evaluate to false.
How to test:	The program in listing F.2.1, is loaded into a test bench. The simple program consists of a set of simple conditional instructions that always evaluate to false. The execution of the program is observed in ISim,and the results are verified. The content of register r1 is observed.
Pass criteria:	The second instruction, the conditional *ADDI*, is not executed. The content of register r1 is 1.
Results:	The content of register r1 is 1. The conditional *ADDI* instruction is not executed. .

Table 8.12: Conditional instruction not executed

What to test:	Observe that STORE instructions stores data to the fake data memory.
How to test:	The program in listing F.2.1, consisting of mainly of STORE instructions. These are loaded into a test bench, and simulated with ISim. The memory dump is read after the simulation.
Pass criteria:	The memory dump shows that value 1 was stored to address zero.
Results:	The value of address zero is 1.

Table 8.13: Store data

What to test:	Observe that LOAD instructions is able to read from memory, and load the memory content into the specified registers.
How to test:	The program in listing F.2.1, consisting of a *LOAD* and *STORE* instruction. These are loaded into a test bench, and simulated with ISim. The content of register r1 is verified.
Pass criteria:	The stored values is loaded from memory and stored in register r1. The values in the register corresponds to the data written and loaded from memory, which is 1.
Results:	Register r1 is loaded with the value 1.

Table 8.14: Load data

What to test:	Observe that STORE GENE instructions are able to store gene and fitness values to the rated pool.
How to test:	The program in listing F.2.1, consisting mainly of STORE GENE instructions. These are loaded into a test bench, and simulated with ISim. The content of the rated pool is verified. This is verified by performing a memory dump of the rated pool.
Pass criteria:	The memory dump contains an individual corresponding to the value 1 and with the corresponding fitness value of 2. These values are easily spotted when the initial values of the pool are randomly generated.
Results:	The store of the fitness and gene is confirmed by the memory dump.

Table 8.15: Store gene

What to test:	Observe that a gene is fetched from the unrated code, and stored in the specified register
How to test:	The program in listing F.2.1, consisting of LOAD GENE instructions. These are loaded into a test bench and simulated with ISim. The content of the location of the distributed counters are checked against the data loaded to the fitness cores.
Pass criteria:	The data fetched from the rated pool is the same gene transmitted to the fitness core.
Results:	Success

Table 8.16: Load gene

What to test:	Test a specific genetic problem using the Galapagos architecture. The problem in question aims to find a specific color, *magicpink*, by genetic evolution.
How to test:	The program in listing E is loaded into a test bench. The programs consists of both genetic and fitness related instructions. Program is executed and verified with ISim. The registers containing the best chromosome and fitness values are studied during the run.
Pass criteria:	Execution shall show an improvement of the fitness scores and the chromosomes as the program simulates. E.g that it converges against a solution.
Results:	The problem converges and the color is found.

Table 8.17: Find color: A genetic solution

What to test:	Test a spesfic genetic problem using the barricelli computer. The problem in question aims to find a solution to the knapsack problem. The problems involves finding the best combination of items to put into a knapsack with a weight constraint. The test start with a set of items with a given score and weight. The program can be found in listing E.
How to test:	The program in listing E is loaded into a test bench. The program consists of both genetic and fitness related instructions. The program is executed and verified in isim. The registers containing the best solutions are studied during the run.
Pass criteria:	It is observed that the best solution converges against a better solution regularly. E.g that it continuously improve for the better.
Results:	It is observable that it improve after a number of microseconds. It is, however, difficult to determine if this solution is good since the simulation is limited to just a few microseconds. Note that the simulations create a lot of simulation related data for a small amount of simulation time.

Table 8.18: The knapsack problem : A genetic solution

8.1.4 Timing simulation

When designing an processor architecture on hardware it is important to take timing into considerations. Electric circuitry has some small delay for electric signals to propagate through the circuits. When performing normal behaviour simulation these delays are not detectable. Behaviour simulation considers circuits without delay; everything happens instant. This is fine when checking if the code behaves as intended. However, the real world is not perfect. There is need to check how this logic actually behaves on circuits with the accompanying delay.

This is accomplished by performing what is referred to as timing simulation. During the compilation phase of the logic it is possible to generate timing data to the simulations. With this timing data the simulation is able to simulate the logic with real delay. When observing the timing simulations it is possible to actually see how the different signals propagate. By observing the simulations with these delay it is possible to uncover errors that would have been undetected during the behaviour simulation.

In the Galapagos the logic was constructed very carefully to avoid such timing issues. This involved removing latches and being careful to synchronize the components with the clock. And not least, only use one clock to not complicate things. Because of this a very few errors were uncovered during this simulation.

Actually, only one error was discovered. As it turned out there existed some delay between loading the instructions and execute them. The cycles was delayed with one cycle. This was resolved by inserting a NOP.

8.2 Testing the PCB

During and after the components were soldered on the PCB board, the board were tested to ensure that the power grid were working as it was supposed to. For the first test, it was checked that all the various LEDs on the board was working in order to verify that the board actually was powered right, and that there was no short circuits on the power grid itself.

Some of the earliest test were also to check that the FPGA actually was working properly, and it was done by making a simple FPGA echo program to test the various pins on the FPGA. The pins on the FPGA were tested by connecting a led to the various FPGA-headers. If the FPGA worked correctly, the led will activate, indicating the the pins actually are operating right. When this test was conducted on the first board that were soldered, it came out that the FPGA was not "baked on" right, and that we had to start solder a new board.

8.3 Testing IO

8.3.1 IO device tests

Test Name	Buttons & LEDs
Steps	1. Upload a program reading the state of all buttons and turning off the LEDs corresponding to the buttons pressed down while leaving the rest of the LEDs on 2. Try pressing the different buttons
Expected Result	All LEDs light up initially and turn off when the corresponding button is pressed.
Actual Result	All LEDs light up initially and turn off when the corresponding button is pressed.

Test Name	Debug connection test
Steps	1. Connect the debug pins to the appropriate pins on the energy micro development kit 2. Turn the debug to OUT 3. Connect to the development kit using energyAware commander
Expected Result	EFM32GG990F1024 listed as microcontroller
Actual Result	EFM32GG990F1024 listed as microcontroller

Test Name	SD Card test
Steps	1. Edit the code from AN0030 [1] to use correct pins 2. Compile the code, upload and run it, with SD Card connected 3. Confirm data on SD Card
Expected Result	File with string "EFM32 ...the world's most energy friendly microcontrollers !" is added to the SD Card.
Actual Result	The SD card was not found, and the text not present on the SD Card

Test Name	USB test
Steps	1. Compile the code from AN0065 [5] 2. Upload and run it 3. Run the supplied host PC program while connected to the PCB through USB
Expected Result	The host programs runs successfully
Actual Result	The host program fails to connect through USB

Test Name	Serial test
Steps	1. Compile the code from AN0045 [3] 2. Upload and run it 3. Connect the host PC to the PCB and run terminal emulator of choice
Expected Result	"Energy Micro RS-232 - Please press a key" appears in the terminal
Actual Result	No output in terminal

8.3.2 FPGA bus

Test Name	SRAM test
Steps	1. Write a value to a range of addresses 2. Read the same address and compare with the value written
Expected Result	The values are identical
Actual Result	Most of the values are identical, with some addresses reporting the wrong value

Test Name	Running a program
Steps	1. Upload a program to fill the memory with fibonacci numbers 2. Let the CPU run for a while to ensure that somehting has been written to memory. 3. Read memory and check whether the fibonacci numbers are stored, the first on adress 0, the next on the next address and so on.
Expected Result	A sequence of fibonacci numbers in the memory.
Actual Result	Seemingly random data read from the memory

8.4 Additional Tests

8.4.1 The Pseudo-Random Number Generator

The pseudo-random number generator designed for the Barricelli has been tested extensively with a pseudo-random number generator test suite called DieHarder[7]. DieHarder is a test suite which measures the "goodness" of a pseudo-random number generator based a number of criteria.

The algorithm was implemented in python and tested against the DieHarder integration suite.

The shift-based algorithm used in the pseudo-random number generator scores quite poorly in the DieHarder tests when every single bit of the output is used. However, by only using every 7th number, the algorithm ranks quite well. A condensed DieHarder test result overview can be found in Table 8.19 on the next page. The descriptions in the table are modified from the descriptions in the output of the DieHarder test suite.

Test Name	Pass?
DieHard "Birthdays Test"	FAILED
Diehard Overlapping 5-Permutations Test	FAILED
Diehard 32x32 Binary Rank Test	FAILED
Diehard 6x8 Binary Rank Test	FAILED
Diehard Bitstream Test.	FAILED
Diehard Overlapping Pairs Sparse Occupance (OPSO)	FAILED
Diehard Overlapping Quadruples Sparce Occupancy (OQSO) Test	FAILED
Diehard DNA Test	FAILED
Diehard Count the 1s (stream) (modified) Test	FAILED
Diehard Count the 1s Test (byte) (modified)	FAILED
Diehard Parking Lot Test (modified)	FAILED
Diehard Minimum Distance (2d Circle) Test	FAILED
Diehard 3d Sphere (Minimum Distance) Test	FAILED
Diehard Squeeze Test	FAILED
Diehard Sums Test	WEAK
Diehard Runs Test	FAILED
Diehard Craps Test	FAILED
Marsaglia and Tsang GCD Test	FAILED
STS Monobit Test	WEAK
STS Runs Test	PASSED
STS Serial Test	WEAK
RGB Bit Distribution Test	FAILED/WEAK
the generalized minimum distance test	FAILED
RGB Permutations Test	PASSED
RGB Lagged Sums Test	PASSED
The Kolmogorov-Smirnov Test Test	WEAK
DAB Byte Distribution Test	PASSED
DCT (Frequency Analysis) Test	FAILED
DAB Fill Tree Test	FAILED
DAB Fill Tree 2 Test	FAILED
DAB Monobit 2 Test	FAILED

Table 8.19: DieHarder test results of the PRNG

Finally, some genetics algorithms convergence tests were run, also simulated in python, using the different pseudo-random algorithm candidates as a random number source in the experiments. Based on the results from these experiments, it is safe to conclude that, while Barricelli's pseudo-random number generator algorithm may not be best-in-class for producing convincing randomness, it is definitely good enough for problem solving using genetic algorithms, and most certainly quicker than other more "proper" algorithms.

CHAPTER

9

RESULTS

This section section describes the results of different measurements, calculations and tests that were run on the Barricelli computer. It also documents the different demonstration programs that showcase and illustrate Barricelli's purpose through practical use, as well as research done in the design of Barricelli.

9.1 Research

9.1.1 Steady State Genetic Algorithm

A big question in the design of the architecture, was how to implement the genetic algorithm. As 1.2.1 on page 5 states, the instruction set should include instructions to speed up genetic operations. To fulfill this requirment, the algorithm would have to be directly integrated with the computer.

In the research, some papers were found that discussed Steady State Genetic Algorithms.[19] As these were described as being advantageous on a MIMD architecture, there was interest in further research. However, few citations were found on how steady state algorithms performancewise related to traditional genetic algorithms. This section documents an original research on Steady State, where both a generational and a steady state solver was implemented in Python, and used to solve a problem.

Problem

Listing 9.1: Genetic problem

```
import random


class MaximizeBitstringGeneticProblem(object):

    def __init__(self, bitstring_length=500):
        self.bitstring_length = bitstring_length

    def select(self, population):
        if(len(population) < 10):
            return max(zip(map(self.fitness, population), population))[1]
        return max(
            self.select(population[:len(population)/2]),
            self.select(population[len(population)/2:])
        )

    def crossover(self, individual_a, individual_b):
        return individual_a[:len(individual_a)/2] + \
            individual_b[len(individual_b)/2:]

    def mutate(self, individual):
        return [bit ^ 1 if random.random() > 0.95 else bit
                for bit in individual]

    def createIndividual(self):
        return [int(random.random() * 2) for i in range(self.bitstring_length)]

    def fitness(self, population):
        return float(sum(population))/len(population)
```

The problem, implemented in listing 9.1, is to maximize a bit string, that is to make let have every bit be 1. Implemented are functions for selection, crossover, mutation, creation and calculating fitness. I. e. what is needed for a genetic search.

Solvers

Listing 9.2: Generational solver

```
class GenerationalGeneticAlgorithmSolver(object):

    @staticmethod
    def solve(problem, fitness_goal=0.95, population_size=100):

        # population size has to be a multiple of 4 for implementation reasons
        population_size = population_size - population_size % 4

        # generate initial population
        population = sorted(
            [problem.createIndividual() for i in range(population_size)],
            key=lambda individual: 1 - problem.fitness(individual))

        generation = 0

        best_individual = population[0]

        while problem.fitness(best_individual) < fitness_goal:

            print "generation", generation
            print "fitness:", problem.fitness(best_individual)
            print ""

            # possibly store new best individual
            if problem.fitness(population[0]) > problem.fitness(
                    best_individual):
                best_individual = population[0]

            # select
            selected = [problem.select(population)
                        for i in range(population_size/2)]

            # crossover
            new_individuals = [problem.crossover(
                                selected[i*2], selected[i*2+1])
                               for i in range(len(selected)/2)]

            # mutate
            mutated_new_individuals = [problem.mutate(individual)
                                       for individual in new_individuals]

            # replace least fit individuals with new individuals
            population.sort(
                key=lambda individual: 1 - problem.fitness(individual))
            population = population[:population_size*3/4] + \
                mutated_new_individuals
```

```
47
48              population.sort(
49                  key=lambda individual: 1 - problem.fitness(individual))
50
51              generation += 1
52
53          return {
54              "solution": best_individual,
55              "fitness": problem.fitness(best_individual),
56              "work": generation,
57          }
```

Listing 9.3: Steady state solver

```
1   class ContinuousGeneticAlgorithmSolver(object):
2
3       @staticmethod
4       def solve(problem, fitness_goal=0.95, population_size=100):
5
6           # generate initial population
7           population = [problem.createIndividual()
8                         for i in range(population_size)]
9
10          work = 0
11
12          best_individual = max(reversed(sorted(zip(map(
13              problem.fitness, population), population))))[1]
14
15          place_counter = 0
16
17          while problem.fitness(best_individual) < fitness_goal:
18
19              print "work", work
20              print "fitness:", problem.fitness(best_individual)
21              print ""
22
23              # select
24              selected_a = problem.select(population)
25              selected_b = problem.select(population)
26
27              # crossover
28              new_individual = problem.crossover(selected_a, selected_b)
29
30              # mutate
31              mutated_new_individual = problem.mutate(new_individual)
32
33              # replace random individual with new individual
34              population[place_counter] = mutated_new_individual
35              place_counter = (place_counter + 1) % len(population)
36
37              if problem.fitness(mutated_new_individual) > problem.fitness(
38                      best_individual):
39                  best_individual = mutated_new_individual
40
41              work += 1
42
43          return {
44              "solution": best_individual,
45              "fitness": problem.fitness(best_individual),
46              "work": work,
47          }
```

In listing 9.2, there is implemented a generational version of a genetic algorithm, while in listing 9.3 there is a steady state version. The main difference of these two solvers is how the new individuals are added to the population. In the generational, the new individuals are inserted for the worst individuals in the old population, maintaining the population size by keeping the best of the old population if there is not enough new individuals. The steady state version just replaces a random selected individual of the current population with the new one.

Result

Generational	Steady State
210	10899
243	27714
238	2336
134	4210
223	2048
143	4365
285	1526
244	8733
141	21515
180	3119

Table 9.1: Results of ten runs of the genetic programs

After 10 runs, the generational implementation had average number of generations equal 204.1, while the steady state has a comparative result of 86.5. The result of the steady state seems much higher than the generational, as it doesn't count generation in the same way. To get comparable results, the work of the steady state algorithm should be divided by the population size, in this case 100.

As can be seen from table 9.1, the number of generations varies a lot more in the steady state, but as the average was significantly lower than generational, the conclusion was that steady state was better, and chosen for the project.

9.2 Measurements

This section presents the measurements found during the project. These results are discussed in Chapter 10 on page 106.

9.2.1 Performance

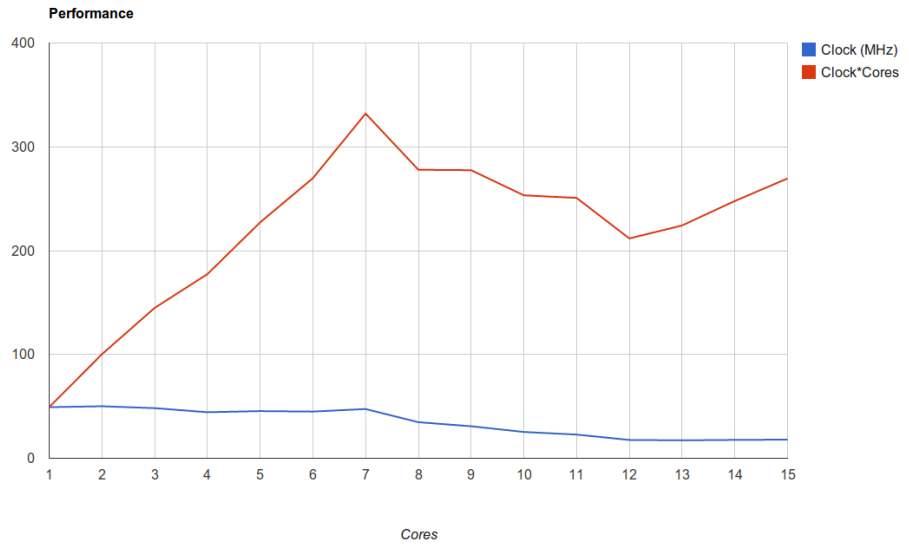


Figure 9.1: Total performance of Barricelli's fitness cores, as a function of number of cores

The total performance of Barricelli's fitness cores, measured as maximum theoretical clock speed times number of clock cores, is illustrated in Figure 9.1. This shows that having multiple cores in a MIMD architecture indeed increases performance, but it also shows that the performance only increases up to a certain limit.

9.3 Demonstration Programs

This section documents the demonstration programs written for the Barricelli computer to demonstrate its functionality. The programs are typically written in Galapagos assembly for programs running on the custom processor, and C for programs running on the SCU. The source code for these demonstration programs can be found in appendix E on page 198.

9.3.1 Genetic Algorithm: Color Search

The color search program is a very simple program demonstrating a basic usage of the genetics accelerator. The program tries to find a specific color in the search-space of all 24-bit colors.

Individual representation

An individual represents a specific 24-bit color in RGB format. The individual is coded to a 64-bit data word like in figure 9.2.

Figure 9.2: The binary coding of an individual for the color search problem

Fitness Function

The fitness for an individual is calculated using equation 9.1. The fitness a any given individual falls in the range $[0, 768]$.

$$\begin{aligned} fitness &= 768 \\ &- |red_{individual} - red_{target}| \\ &- |green_{individual} - green_{target}| \\ &- |blue_{individual} - blue_{target}| \end{aligned} \tag{9.1}$$

Results

Figure 9.3 on the following page shows the evolution of the approximation suggested as an answer by the genetic algorithm. In this problem instance the target color was magic pink, i.e. the color with color code $rgb(255, 0, 255)$. The program run illustrated in figure 9.3 on the next page ran on 7 seven cores, and the measurements are from regularly polling a single core for its current best solution. Figure 9.3 on the following page clearly illustrates a typical trait of genetic algorithm approximations: they are quite good at finding decent approximations, but iterating to improve accuracy of the result is a game of diminishing returns. The algorithm quickly finds a decent approximation of the target color, but finding the exact value down to the last bit still takes time.

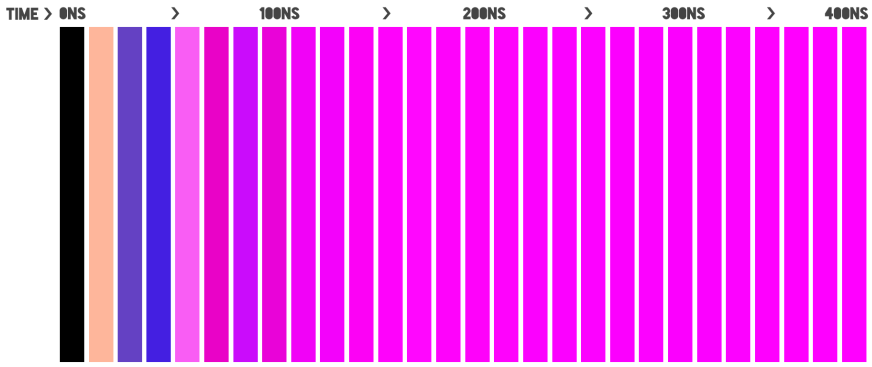


Figure 9.3: Color search progression (7 cores, 1 core sampled)

9.3.2 Genetic Algorithm: Binary Knapsack Problem

The knapsack problem is an optimization problem that is considered NP-hard. The problem to solve is given a set of items with a weight and a value and a knapsack that can hold a specific weight, what combination of items that can fit in the sack has the highest value. If there can be at most one of each item in the knapsack, we have what is known as the binary knapsack problem.

Individual representation

An individual represents a combination of items. The individual is coded to a 64-bit data word like in Figure 9.4.

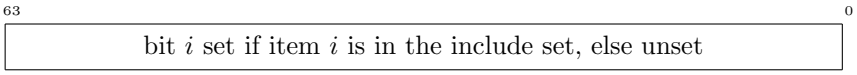


Figure 9.4: The binary coding of an individual for the binary knapsack problem

Fitness Function

The fitness for an individual is calculated using Algorithm 6. The fitness falls between 0 for invalid sacks (that contain too much weight) to a theoretical

maximum of the sum of the value of all the items.

```
Data: A bit array with one bit representing whether each item is
      included or not
Result: How fit the individual is
begin
    weight ⟵ 0
    value ⟵ 0
    for item in ItemSet do
        if item in genome then
            weight ⟵ weight + item.weight
            value ⟵ value + item.value
            if weight > maxWeight then
                value ⟵ 0
                break
            end
        end
    end

    return value
end
```

Algorithm 6: The fitness function for the binary knapsack problem

Results

This problem has not been run on the actual hardware, only simulated up to a few milliseconds. The simulation seemed fine, and we got increasing fitness values, but the problem space is huge, so we probably need to run the algorithm for quite some time before coverging to a ideal/near-ideal solution.

9.3.3 Blinkenlights

Blinkenlights is a program that demonstrates the use of the input and output of the Barricelli driven by the SCU. When a button is pressed, the correnspondingLEDs to the button lit up. This program is also described as a test for I/O, in 8.3.1 on page 94, named Buttons & LEDs.

CHAPTER

10

DISCUSSION

10.1 Performance

The Barricelli computer is designed to be a device for high performance parallel computing. Throughout the design process, choices have been made that further this goal.

The computer architecture is capable of executing multiple independent instruction streams working on multiple different independent data streams, which means that parallelism can be exploited to a large degree to achieve a high computational throughput. A heterogenic collection of cores, some general, and some working as specialized accelerators, combines the allround-ness and usability of general computing with the at times extreme performance boosts given by specialized workers.

The general cores have been designed to exploit instruction-level parallelism, iwth features such as pipelining, hazard detection and correction using forwarding, and branch prediction.

The intelligent off-line assembler also helps off-load some of the hazard detection work from the processor, which lets the processor spend more of its valuable time computing.

Since the computer uses a shared memory bus though a single memory controller, it is an obvious scalability limitor. For the instruction memory, this is somewhat mitigated by the inclusion of instruction caches, but there is still room for improvement. To improve the memory performance, memory could be organized in a more hierarchical fashion, with multiple cache levels.

10.1.1 Performance Measurements and Benchmarking

With the specified prototype of the Barricelli computer presented in this report, looking at the results from the performance measurements in Chapter 8 on page 83, the optimal number of parallel fitness cores is 7. Up to 7 processors, the performance scales beautifully, with calculated total performance scaling linearly with the number of cores. After 7 cores, the relative performance of each core drops. It is not easy to see exactly why this happens, but it is quite probably related to resource usage of the FPGA.

As the number of fitness cores increases, the more pressure it puts on the genetics accelerator. As the number of fitness cores increases, the number of accelerators should also increase.

10.1.2 Average Instructions per Cycle

In optimal condititons, i.e. the pipeline is filled, and no register spilling occurs, the processor can execute n instructions per cycle, where n is the number of processors. In the designed protoype of Barricelli, this means that the maximum instructions per second is $7cores * 50Mhz = 350000000$, or 350 Mega-instructions per second.

10.2 Theory

In this section theory related to MIMD architectures and how to further improve their performances are discussed. This also covers the discussions inside the group about the design of the architecture in the planning phase of the project.

10.2.1 SPMD and Concurrency

One of the first discussions that came up in the group after the assignment were given was if the processor cores should be able to synchronize themselves. By doing this, the processor cores would be able to execute code that is not completely independent in parallel in an elegant way. However this also raises some issues as for instance data hazards.

10.2.2 Using CISC or RISC ISAs

CISC and RISC instruction set architectures are two very different ways of thinking when it comes to creating instruction sets. While in the last years, RISC ISAs have been the most dominant instruction set architecture, we can also see that CISC architectures are on their way back into the markets. Some of the reasons for this is that increasing parallelism is gaining lesser performance increases.

Micro Operations: a Bridge Between Complex and Reduced Instruction Sets

The use of micro operations is based on the principle that you want to convert a complex instruction into a set of smaller micro operations. This may simplify the design of for instance a super scalar processor because dependencies between the converted micro instructions would already be known.

10.2.3 Memory Management Policies

The Galapagos architecture operates with several types of shared memory: *instruction memory*, *data memory*, *instruction caches*, *rated pool* and *unrated pool*. These types of memories can further be divided into two groups: memory and genetic related. These two groups are connected to separate data and address buses. The access to these buses are handled by a request-acknowledgement protocol. The responsible of this protocol is to control the access to the shared memories and their respective buses. The protocol is based in the ideas of round-robin scheduling. The request lines are continuously polled by the controllers in a round-robin fashion. The reason for choosing this algorithm is because it is considered to be fair. Since each request line is checked in turn, the algorithm is considered starvation free. A requesting core will eventually get its request handled by the controllers.

Since all the cores access the same memories this solution will, however, turn out to be quite slow. Note that every core is in fact competing for the access to memory. For memory intensive problems this bottleneck will be quite visible. Every time cores wishes to perform simultaneously some memory access only one of them will be granted access. For instance, consider a system with five cores. If these cores have a relatively frequent memory access pattern it pretty self explanatory that this will cause the the different cores being idle most of the time. This will imply that the memory scheme in the galapagos architecture does not scale very well. This implies that the more cores that are present the less performance would be achieved.

A possible solution for this problem would have been keeping separate data caches for each core. Then the cores could have been using the data located in the caches instead of accessing the memory so frequently. When first accessing the memory, the core could have loaded several data elements instead of just one word for each request. This would surely been an improvement for the memory system employed in the baracelli currently. This would, however, been very difficult to achieve, and is not in the scoop of this assignment. Private data caches would require implementing cache coherence algorithms for keeping the caches consistent. This is considered very difficult.

Cache Coherency

MIMD architecture use a shared memory models. This imposes a problem when using caches, and memory in general. When more core updates on the same values on the same memory positions; memory collisions occur. These problems

can be fixed by enforcing that only one core is able to access the memory at any given time. A far more difficult problem is the problem of cache coherence. Cache coherency issues occurs when several cores have private caches containing the same data, and some core changes the data. Then the data in the caches is not consistent among the cores. In order for the data to be consistent, in this example, is for each core having the same data. Note that same data in this context mean data from the same memory location.

The Galapagos architecture does not support private data caches. This design choice relieves the processor designer of implementing advanced cache coherency algorithms in hardware. Instead of private data caches the Galapagos architectures employ shared pools for rated and un-rated chromosomes. These are connected to a bus and the connected through the *genetic controller*. The controller is configured to only allow one core perform its operation on one of the pools at any given time. This implies that read and write operations are atomic. As a direct consequence cache coherency issues are not possible in the architecture.

CHAPTER

11

WORK PROCESS

11.1 Development model

For this project, it was agreed that an adaption of the Scrum development model should be used. The group wanted to use an agile development method in order to be able to make quick changes to the requirement specification during the project. The model is adapted however because many of the aspects in scrum, like working in sprints does not work so well for this project due to the unpredictable nature of the project. For the same reasons there was no daily meetings, but rather once in a week where everyone is informed about the current progress of the project.

11.2 Group Organization

From early on, the group divided itself into 3 work groups, focusing on three main areas: FPGA, PCB and IO. This was done largely based on the fact that it has been done similarly in the years before[12], and it seemed like a reasonable thing to do. One advantage in doing this is that each work group can become more specialized in their respective fields compared to if everyone were to work equally on everything. This allows for a more advanced execution in the project, which is a good thing. The group member work group allocation can be found in table 11.1 on the following page.

Other than this group allocation, no other hierarchical elements were introduced. The group functioned as a direct democracy in all other issues, with no appointed leader. This approach had both it's benefits and downsides. The

greatest benefit is that everyone got to take part in the important decisions made in the project. The downside was that decisions also take much longer time to take due to having to discuss matters over a meeting.

FPGA
Sigve Sebastian Farstad
Torbjørn Langland
Per Thomas Lundal
Bjørn Åge Tungesvik

PCB
Fedor Fadeev
Eirik Flogard
Rune Holmgren
Odd Magnus Trondrud

IO
Emil Taylor Bye
Péter Gombos

Table 11.1: Group Allocation

The group held weekly sync-up meetings in addition to a weekly status meeting with the course staff. Unfortunately, meeting attendance was occasionally some lower than what it should have been. Fortunately, meeting minutes were always kept, so information from missed meetings did not go lost.

11.3 Organizational tools

11.3.1 GitHub

GitHub was used in order for everyone to be able to work at different parts of the project at the same time. It also provides an excellent version control that would allow a user to work on experimental "branches". When using branches, the users does not need to worry about taking backups before trying out something new. These branches can also be merged into the main project later.

11.3.2 Trello

Trello is a tool that the group used as "scrum table" to keep everyone updated in real time about the current progress in the project. The tool resembles a scrum board which keeps track of what everyone is doing at a specific time.

11.4 Tools

Below here is a list of tools that were used directly to develop the system.

11.4.1 Software

ISE Project Navigator 12.4 (nt64) M.81d, expired licence
Main IDE for writing VHDL.

ISim 12.4 (nt64) M.81d, expired license
Main simulation environment for simulating VHDL.

ModelSim SE 6.6d
Secondary simulation environment for simulating VHDL.

Xilinx Platform Studio 12.4 (nt64) Build EDK_MS4.81d+1, expired licence
Used for preparing compiled VHDL for the FPGA board.

IAR Embedded Workbench for ARM 6.60.2.5507
Used for programming and debugging on the microcontroller.

Avnet Programming Utility
Used for configuring the FPGA.

energyAware Commander 2.82
Programming and troubleshooting of the microcontroller.

energyAware Designer 1.10
Used for configuring the GPIO pins and generating projects for the microcontroller.

Saleae Logic 1.1.9
Used to view the sampled waveforms from the logic analyzer

Tera Term Pro Web Version 3.1.3
Terminal emulator used for serial communication.

Text editors
Sublime Text 2, Vim 7.3, Notepad (©Copyright Microsoft Corporation).

GNU command-line tools
Grep, sed, find, etc.

git 1.8.1.2
Version control system.

GitHub
Remote code repository hosting, issue tracking, wiki for logging.

MakerWare 2.3.1.18
Compile 3D models.

Google SketchUp 8.0.16846
Create 3D models.

Google Spreadsheets
Keep lists organized.

texlive
Typesetting this report.

python 2.7.4
Writing the Galapagos assembler.

Adobe Creative Cloud InDesign CC
Designing the front page of this report.

Lucidchart
Creating charts.

11.4.2 Hardware

Xilinx Spartan-6 XC6SLX45 CSG324 FPGA board.

Energy Micro EFM32GG990F1024 Microcontroller.

Energy Micro EFM32GG-DK3750 Development kit used for testing code on the microcontroller before the PCB arrived.

Energy Micro EFM32GG-STK3700 Prototyping and development when the microcontroller on the PCB could not be used.

Saleae Logic Logic analyzer

Development PC, Windows 7 For development.

Mini USB cable For connecting the FPGA board to the development computer.

Makerbot Replicator 2 3D printing the case.

Fluke Multimeter 77 Checking currents on the PCB.

Altec Lansing ACS340 Play sweet tunes in the lab while working.

CHAPTER

12

CONCLUSION AND FURTHER WORK

We designed and tested a system capable of solving hard problems using genetic algorithms. That is, the system's PCB design looks sound on paper and the processor has been successfully tested with simulations. However we were unable to deliver a functioning physically integrated version of the system because the ones we produced were eventually bricked for various reasons. Even so, we fulfilled nearly all of our functional and non-functional requirements because as it turns out none of these require a working, physically integrated[1] system.

Our goal of constructing a general MIMD computer capable of solving hard problems using genetic algorithms was met.

It is capable of receiving instructions and data from external entities through the debug pins. The USB, SD and serial ports were not successfully tested due to the problems with the PCB.

We implemented performance increasing techniques that would not overly complicate the design.

The Galapagos instruction set includes instructions to load and store genes from and to a genetics pool, which have eased the task of working with genetic algorithms with the instruction set.

Soldering all the required components – including the Xilinx FPGA and "Energy Micro" (now Silicon Labs) microcontroller – onto the system took approximately eight hours. Although only one completely soldered board was produced: the

[1] with all the components soldered on

time required to solder a board would likely have been reduced with each further board produced.

The cost of producing one complete system was below budget. The PCB's production cost ate up 78% of the budget.

The system's 3D-printed case is pretty neat and a complete Barricelli system can fit inside of it. It has all the visual features of the barricelli board, including access to the I/O ports, I/O buttons and LEDs, reset and power indicator. The eight switches and 20 LEDs can be connected to the Barricelli system so that the Barricelli system can be interacted with even if it inside the case. The case also has a ninth switch that can be connected to the reset button. However it does not have a button that can be used as a power toggle switch.

We do not have a working demo program running a genetic algorithm. This was the one requirement we did not manage to fulfill.

And of course there's a report: the very same that you are reading right now.

The project has been challenging. It feels somewhat bizarre that we got so far. They say experience is its own reward, if that is true we've been rewarded plenty.

12.1 Further Work

To create a functioning system, care should be taken to not short circuit the board. Also the oscilator on the board did not commmunicate with the FPGA, and this should be looked into.

The PCB design does contain some elements that could (and would) be left out of a second revision. For example, there should be a complete debug port for the DK3750, so that a complete bus could be inserted, and not add random wires.

GLOSSARY

Barricelli is the name of the genetic algorithm-solving MIMD computer designed as a solution for the project documented in this report. Barricelli is also the name of a famous Norwegian-Italian mathematician, after whom the computer is named. 2, 10, 12, 17, 18, 20, 22, 24, 25, 105

BRAM Block RAM, dedicated circuitry within a FPGA used as RAM. 24, 29

data controller Memory controller responsible for the data memory access. 29

DSP slice Digital signal processor slices, dedicated circuitry in FGPAs that contain specialised components to perform certain operation as fast as possible. 39

EBI External Bus Interface. 74

FPGA Field Programmable Gate Array, an integrated circuit that can be programmed to perform a variety of operations, for instance acting as a processor. iv, 18–20, 39

Galapagos is the name of the instruction set architecture designed for the Barricelli computer. 8, 18, 20, 22, 23, 25, 34, 37, 42, 76, 102

Galapagos Assembler The assembler for Galapagos Assembly written in Python.. vi, 76, 77, 87

Genetic controller The controller that controls the genetic pipeline. 29

Harvard machine A computer architecture with physically separate instruction and data memory pools. 24

individual A single, feasible solution to a hard problem in genetic algorithms.. 23

ISA Instruction Set Architecture, a set of opcodes and the native commands implemented by a particular processor. 31

LUT LookUp Table, one of the base component of FPGAs, basically large truth tables implemented in hardware. 39

MIMD Multiple Instruction, Multiple Data. 10, 25

MIPS Microprocessor without Interlocked Pipeline Stages, a RISC instruction set architecture. 22, 31

nop No OPeration, an assembly instruction that does not perform anything, used when one has to delay execution for a specific amount of clock cycles.. 22, 76

rated controller Memory controller responsible for the rated pool. 29

Rated pool A memory pool of individuals that have been evaluated and are ready for the selection stage of a genetic algorithm. 29

RISC Reduced instruction set computing, a CPU design strategy focusing on simple instructions. 22

SCU is the EFM32 microcontroller which is used as a System control unit. v, 18, 20, 102, 105

Search space the collection of parameters to be searched over. 11, 14

SRAM Static Random Access Memory, very fast RAM that is more expensive and less dense than dynamic RAM. 18, 24

unrated controller A memory pool of "newborn" individuals that are ready to be evaluated. 29

VHDL is the programming language in which the processor in implemented. VHDL is short for VHSIC Hardware Description Language. vi, 83, 87

VHSIC is short for Very High-Speed Integrated Circuit. 117

Part IV

Appendices

APPENDIX

A

GALAPAGOS INSTRUCTION SET ARCHITECTURE DOCUMENTATION

Galapagos Instruction Set Architecture

November 20, 2013

Contents

1 Introduction

This document documents Baricelli's Galapagos Instruction Set Architecture. Barricelli is a general purpose computer which is equipped with specialized hardware for high performance computation of hard problems using genetic algorithms. It was designed by a team of 10 students at the Norwegian University of Science and Technology over the course of one semester.

Barricelli has multiple independent cores; this document documents a single core.

The first part of this document describes the general architecture, targeted toward assembly programmers. The second part is a reference manual explaining all the assembly instructions in detail.

2 Registers

General purpose registers

The architecture has 31 general purpose 64-bit registers, r1 - r31.

Note that for the purpose of procedure calls, the CALL and RET instructions treat r31 as the link register.

Special purpose registers

The architecture contains the following 3 special purpose registers.

r0 - Zero Register

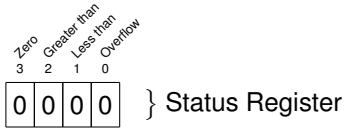
The zero register always holds the value 0. It can not be changed by any means, writing values to it simply has no effect.

PC - Program Counter

The program counter stores the address of the current instruction during execution time. During normal execution, PC increments by 1 after each instruction is complete. PC may be changed by the programmer using instructions like JMP. PC is a 19-bit register. It can not be directly accessed by any instruction.

ST - Status Register

The status register is a 4-bit register that holds flags that describe the outcome of the previous instruction. This register is read for every instruction to determine if conditionals are fulfilled. It can not be directly accessed by any instruction.

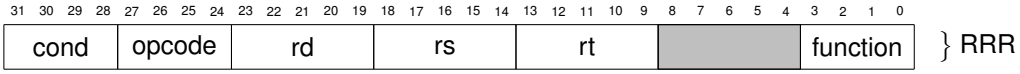


3 Instruction formats

There are three instruction formats: RRR, RRI and RI.

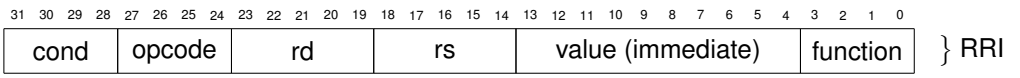
RRR - Register Register Register

RRR, or Register Register Register, has 3 register selectors and 1 function selector. Register selectors specify one of 32 available general purpose registers. The function selector selects a specific function for an opcode, when there are multiple functions available.



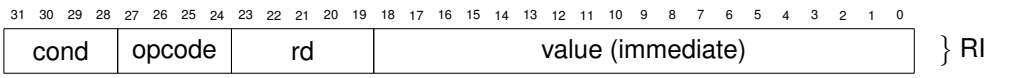
RRI - Register Register Immediate

RRI, or Register Register Immediate, has 2 register selectors and 1 10-bit immediate. Register selectors specify one of 32 available general purpose registers.



RI - Register Immediate

RI, or Register Immediate, has 1 register selector and 1 19-bit immediate. The register selector specifies one of 32 available general purpose registers.



Conditions

All instructions have the possibility to embed a condition for the execution of the instruction. The first 4 bits of each instruction specify the possible conditions, which are:

- 0000 - Never
- 0001 - Equal (Zero)
- 0010 - Not Equal (Not Zero)

- 0011 - Greater Than or Equal (Positive or Zero)
- 0100 - Greater Than (Positive)
- 0101 - Less Than or Equal (Negative or Zero)
- 0110 - Less Than (Negative)
- 0111 - Overflow
- 1000 - Not Overflow
- 1111 - Always

The conditions use the current value of the status flags at execution time to determine the outcome of the operation. If the condition is met, the instruction is executed. If not, the instruction is not executed. This allows for branchless conditionals.

Assembly

Galapgos assembly is written with one instruction per line. Instructions are case-insensitive. Multi-line comments start with the token '/*' and end with the token '*/'. Labels are all bare strings matching the following regular expression: `/[^:0-9\n\t\v][^:\n\t\v]*/`, followed by the token ':'. Labels must stand on their own line, but can be indented with whitespace. Numerical constants may be decimal on the form `1234567890`, hexadecimal on the form `0x1234567890abcdefABCDEF`, or binary on the form `0b10`.

In the assembly, all instructions can be prefaced with a condition. Conditions are prepended to the current instruction, using the token 'if', followed by a representation of the condition, and finally the token ':'.

An example might look like this:

```
if equal:  add r1, r2, r3
```

The available conditions are:

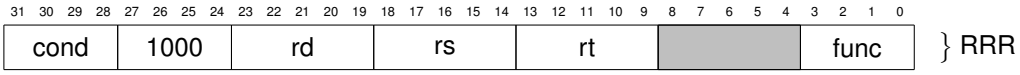
- `equal`
- `not equal`
- `greater than`
- `greater than or equal`
- `less than`
- `less than or equal`

- `zero`
- `not zero`
- `positive`
- `positive or zero`
- `negative`
- `negative or zero`
- `overflow`
- `not overflow`
- `never`
- `always`

4 CPU Instruction Description Explanation

The rest of this document documents individual instructions in a reference manual format. In this section, some of the figures and conventions used are explained.

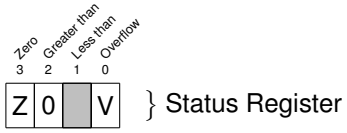
The instruction word box shows the bit layout of the instruction, and which instruction format it is using. Here is an example of what it might look like:



Textual descriptions denote the contents of a bit section. Binary strings denote the actual binary contents that need to be present in that instruction. Gray boxes denote unused or ”don’t care” sections.

Affected Status Flags

Many instructions may modify one or more flags in the status register. Each instruction has a status register diagram which displays how the different flags are affected. Here is an example of what the diagram looks like:



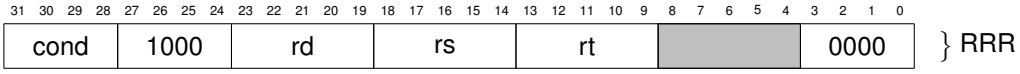
The different symbols in the diagram have the following meanings:

- **0**: the flag is unset
- **1**: the flag is set
- **Z**: the flag is set if the destination register is zero, else unset
- **P**: the flag is set if the destination register is positive, else unset
- **N**: the flag is set if the destination register is negative, else unset
- **V**: the flag is set if there is an overflow, else unset
- **grayed out**: the flag is unchanged

5 Arithmetic/Logic Instructions

This section describes all arithimetical and logical instructions. All operations are made available in both RRR and RRI formats. Note that the only difference between the operations is the ALU function code. Also note that all operations set all status flags.

ADD - Add



Format

ADD rd, rs, rt

Purpose

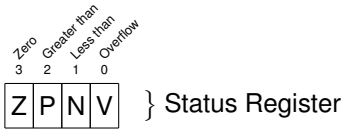
To add 64-bit signed integers. If overflow occurs, the overflow status bit is set.

Description

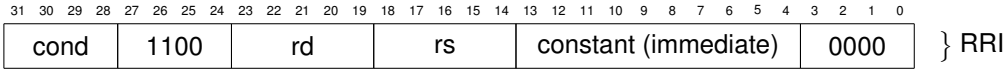
$$rd \leftarrow rs + rt$$

The 64-bit signed word value in register rs is added to the 64-bit signed value in register rt to produce a 64-bit signed result. The 64-bit signed result is put in register rd. If a 64-bit 2's complement arithmetic overflow occurs, the overflow status bit is set.

Affected Status Flags



ADDI - Add Immediate



Format

ADDI rd, rs, constant

Purpose

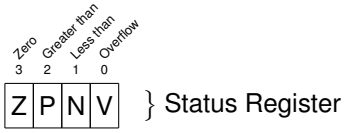
To add a constant to a 64-bit integer.

Description

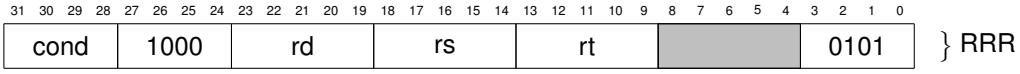
$$rd \leftarrow rs + immediate$$

The 10-bit signed immediate is added to the 64-bit value in register rs to produce a 64-bit result. The result is put in rd. If the addition results in a 64-bit 2's complement arithmetic overflow, the overflow register is set.

Affected Status Flags



AND - Logical AND



Format

AND rd, rs, rt

Purpose

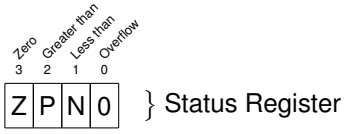
To logically AND two 64-bit words.

Description

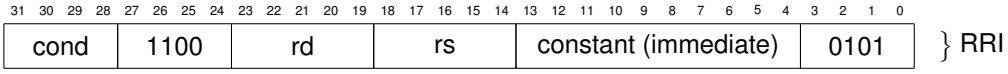
$$rd \leftarrow rs \wedge rt$$

The contents of register rs are combined with the contents of register rt in a bitwise logical AND operation. The result is placed into register rd.

Affected Status Flags



ANDI - Logical AND Immediate



Format

ANDI rd, rs, constant

Purpose

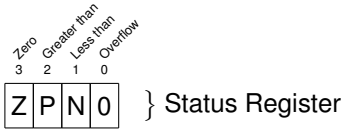
To do a bitwise logical AND with a constant.

Description

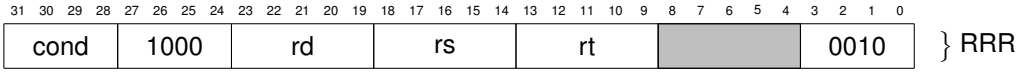
$$rd \leftarrow rs \wedge immediate$$

The 10-bit immediate is zero-extended to the left and combined with the contents of register rs in a bitwise logical AND operation. The result is place in to register rd.

Affected Status Flags



MUL - Multiply



Format

MUL rd, rs, rt

Purpose

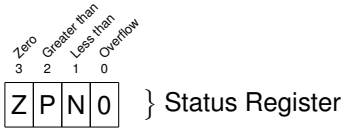
To multiply 32-bit signed integers. If overflow occurs, the overflow status bit is set.

Description

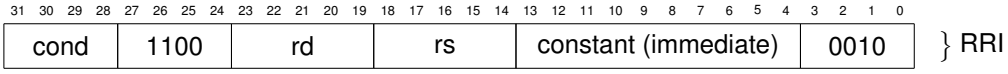
$$rd \leftarrow rs \times rt$$

The least 32 bits of the 64-bit signed word value in register rs is multiplied with the least 32 bits of the 64-bit signed value in register rt to produce a 64-bit signed result. The 64-bit signed result is put in register rd. This computation can not cause an overflow, but it is important to pay attention to the clipping of input registers.

Affected Status Flags



MULI - Multiply Immediate



Format

MULI rd, rs, constant

Purpose

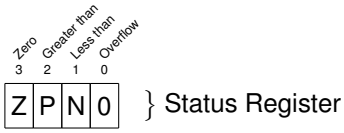
To multiply a 32-bit signed integer with a constant. If overflow occurs, the overflow status bit is set.

Description

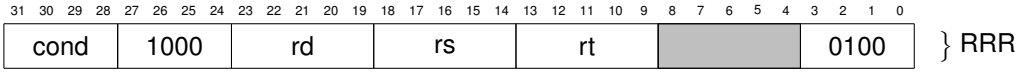
$$rd \leftarrow rs \times constant$$

The least 32 bits of the 64-bit signed value in register rs is multiplied by the 10-bit signed immediate $constant$ to produce a 64-bit signed result. The result is stored in register rd. This computation can not cause an overflow, but it is important to pay attention to the clipping of input register.

Affected Status Flags



OR - Logical OR



Format

OR rd, rs, rt

Purpose

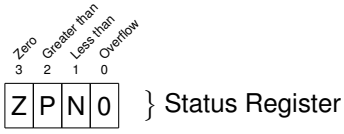
To logically OR two 64-bit words.

Description

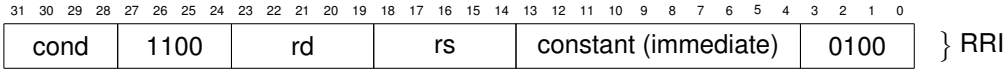
$$rd \leftarrow rs \vee rt$$

The contents of register rs are combined with the contents of register rt in a bitwise logical OR operation. The result is placed into register rd.

Affected Status Flags



ORI - Logical OR Immediate



Format

ORI rd, rs, constant

Purpose

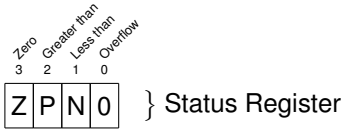
To do a bitwise logical OR with a constant.

Description

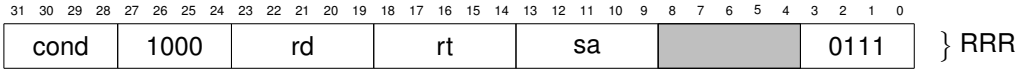
$$rd \leftarrow rs \vee immediate$$

The 10-bit immediate is zero-extended to the left and combined with the contents of register rs in a bitwise logical OR operation. The result is place in to register rd.

Affected Status Flags



SLL - Shift Left Logical



Format

SLL rd, rt, sa

Purpose

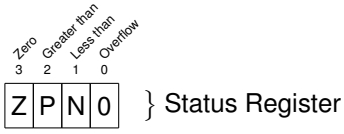
To logical left shift a 64-bit word.

Description

$$rd \leftarrow rt <<_{logical} sa$$

The 64-bit word in register rt is logically shifted left by the value in register sa. The result is stored in register rd.

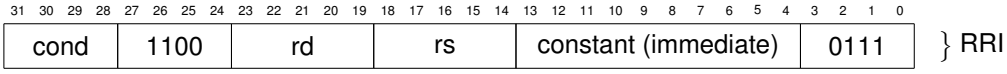
Affected Status Flags



Programming Notes

The maximum shift distance is 63 bits.

SLLI - Shift Left Logical Immediate



Format

SLLI rd, rt, constant

Purpose

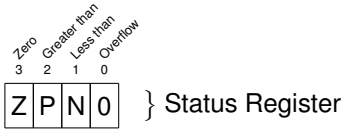
To logical left shift a 64-bit word by a constant.

Description

$$rd \leftarrow rt <<_{logical} constant$$

The 64-bit word in register rt is logically shifted left by the immediate value $constant$. The result is stored in register rd.

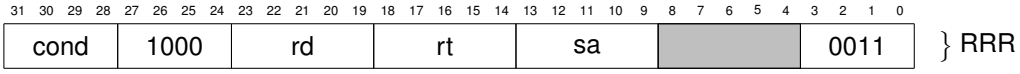
Affected Status Flags



Programming Notes

The maximum shift distance is 63 bits.

SRA - Shift Right Arithmetical



Format

SRA rd, rt, sa

Purpose

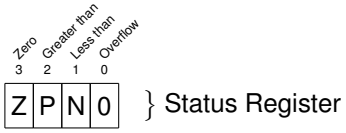
To arithmetic right shift a 64-bit value.

Description

$$rd \leftarrow rt >>_{arithmetic} sa$$

The 64-bit signed contents of register rt is shifted right, duplicating the sign bit into the emptied bits. The result is placed in register rd. The bit shift count is specified as the 64-bit unsigned value in register sa.

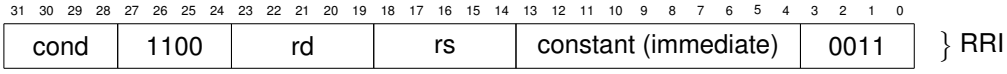
Affected Status Flags



Programming Notes

The maximum shift distance is 63 bits.

SRAI - Shift Right Arithmetical Immediate



Format

SRAI rd, rs, constant

Purpose

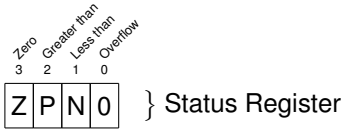
To arithmetic right shift a 64-bit value by a constant.

Description

$$rd \leftarrow rs >>_{arithmetic} constant$$

The 64-bit signed contents of register rs is shifted right, duplicating the sign bit into the emptied bits. The result is placed in register rd. The bit shift count is specified as the 10-bit unsigned immediate value $constant$.

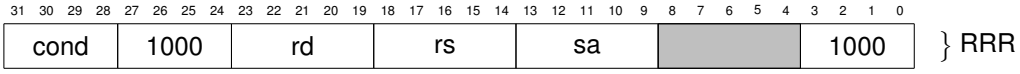
Affected Status Flags



Programming Notes

The maximum shift distance is 63 bits.

SRL - Shift Right Logical



Format

SRL rd, rs, sa

Purpose

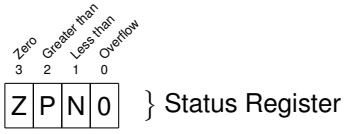
To logical right shift a 64-bit word.

Description

$$rd \leftarrow rs >>_{logical} sa$$

The 64-bit word in register rs is logically shifted right by the value in register sa. The result is stored in register rd.

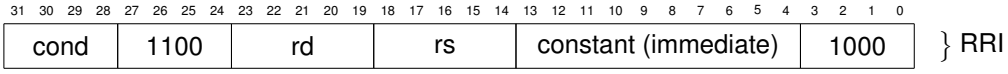
Affected Status Flags



Programming Notes

The maximum shift distance is 63 bits.

SRLI - Shift Right Logical Immediate



Format

SRLI rd, rs, constant

Purpose

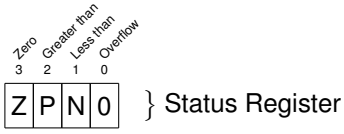
To logical right shift a 64-bit word by a constant.

Description

$$rd \leftarrow rs >>_{logical} constant$$

The 64-bit word in register rs is logically shifted right by the immediate value $constant$. The result is stored in register rd.

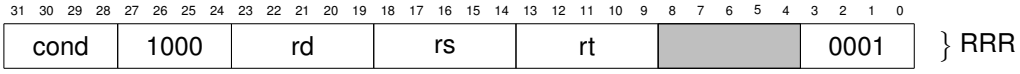
Affected Status Flags



Programming Notes

The maximum shift distance is 63 bits.

SUB - Subtract



Format

SUB rd, rs, rt

Purpose

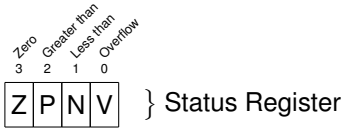
To subtract 64-bit signed integers. If overflow occurs, the overflow status bit is set.

Description

$$rd \leftarrow rs - rt$$

The 64-bit signed word value in register rt is subtracted from the 64-bit signed value in register rs to produce a 64-bit signed result. The 64-bit signed result is put in register rd. If a 64-bit 2's complement arithmetic overflow occurs, the overflow status bit is set.

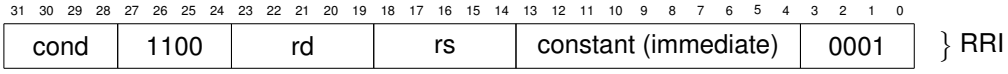
Affected Status Flags



Programming Notes

SUB is used as the compare instruction.

SUBI - Subtract Immediate



Format

SUB rd, rs, constant

Purpose

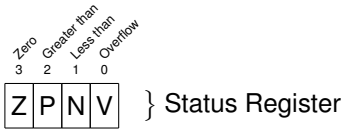
To subtract a 10-bit signed constant from a 64-bit signed integer. If overflow occurs, the overflow status bit is set.

Description

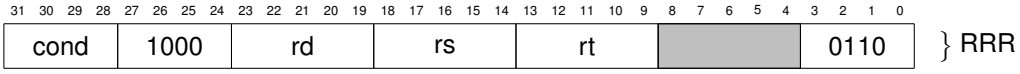
$$rd \leftarrow rs - constant$$

The 10-bit signed word immediate value $constant$ is subtracted from the 64-bit signed value in register rs to produce a 64-bit signed result. The 64-bit signed result is put in register rd. If a 64-bit 2's complement arithmetic overflow occurs, the overflow status bit is set.

Affected Status Flags



XOR - Logical XOR



Format

XOR rd, rs, rt

Purpose

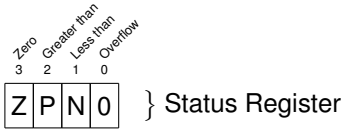
To logically XOR two 64-bit words.

Description

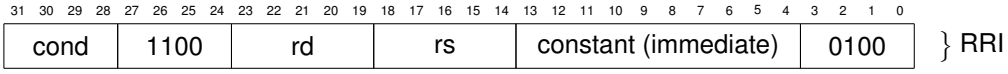
$$rd \leftarrow rs \oplus rt$$

The contents of register rs are combined with the contents of register rt in a bitwise logical XOR operation. The result is placed into register rd.

Affected Status Flags



XORI - Logical XOR Immediate



Format

XORI rd, rs, constant

Purpose

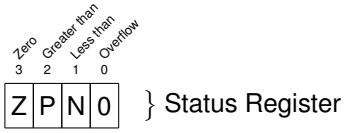
To do a bitwise logical XOR with a constant.

Description

$$rd \leftarrow rs \oplus constant$$

The 10-bit immediate is zero-extended to the left and combined with the contents of register rs in a bitwise logical XOR operation. The result is placed into register rd.

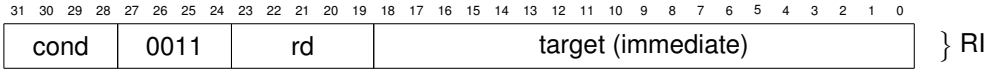
Affected Status Flags



6 Control/Memory Instructions

This section describes all instructions that manipulate program flow or handles memory access.

CALL - Call Procedure



Format

CALL rd, target

CALL label

Purpose

To call a procedure.

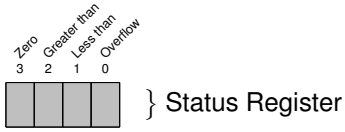
Description

$$r31 \leftarrow PC + 1$$

$$PC \leftarrow rd + target$$

Stores the return address in the link register, before jumping to the effective target address. The 64-bit signed offset value in register rd is added to the target, and the least 19-bits of the result is put in the unsigned program counter register.

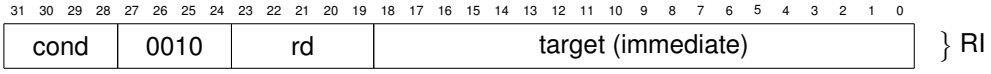
Affected Status Flags



Programming Notes

The programmer may use the alternative format `CALL label` to call a procedure by label.

JMP - Jump



Format

JMP rd, target

JMP label

Purpose

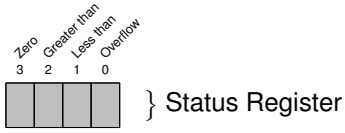
To change the PC to a new location.

Description

$$PC \leftarrow rd + target$$

Jumps to the effective target address. The 64-bit signed offset value in register rd is added to the target, and the least 19-bits of the result is put in the unsigned program counter register.

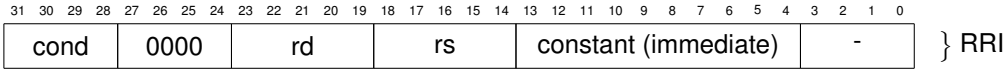
Affected Status Flags



Programming Notes

The programmer may use the alternative format `JMP label` to jump to a label.

LD - Load



Format

LD rd, rs, constant

Purpose

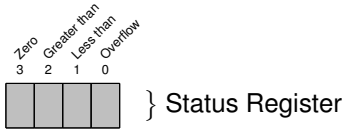
To load a 64-bit value from memory at a given address.

Description

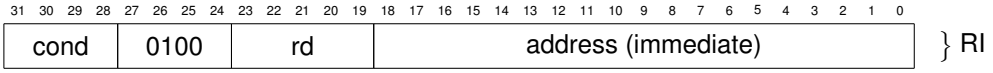
$$rd \leftarrow memory[rs + immediate]$$

The contents of the memory location specified by the 19 least significant bits of $rs + immediate$ is fetched and placed into register rd. The address is unsigned.

Affected Status Flags



LDI - Load immediate



Format

LDI rd, address

Purpose

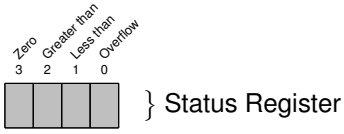
To load a 64-bit value from memory at a given address.

Description

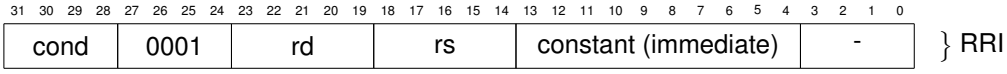
$$rd \leftarrow memory[address]$$

The contents of the memory location specified by the 19 bit $address$ is fetched and placed into register rd. The address is unsigned.

Affected Status Flags



ST - Store



Format

ST rd, rs, constant

Purpose

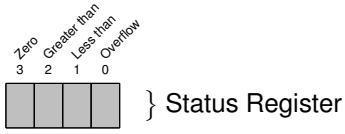
To store a 64-bit value in memory at a given address.

Description

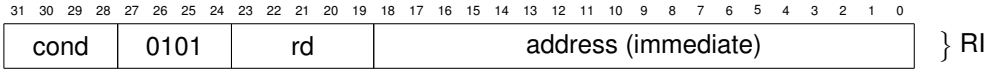
$$memory[rs + immediate] \leftarrow rd$$

The contents of register rd is stored to the memory location specified by the 19 least significant bits of $rs + immediate$ The address is unsigned.

Affected Status Flags



STI - Store Immediate



Format

ST rd, address

Purpose

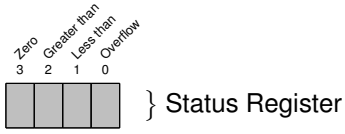
To store a 64-bit value in memory at a given address.

Description

$$memory[address] \leftarrow rd$$

The contents of register rd is stored to the memory location specified by the 19 bit $address$ The address is unsigned.

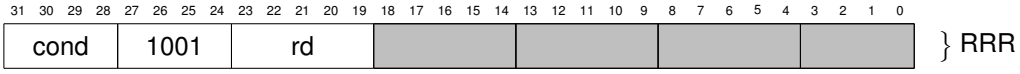
Affected Status Flags



7 Genetic Instructions

This section describes all instructions specialized for access to and manipulation of the genetics unit.

LDG - Load gene from pool



0000

Format

LDG rd

Purpose

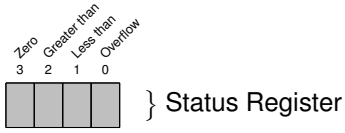
To load a 64-bit genetic algorithm individual from the unrated pool.

Description

$$rd \leftarrow individual from unrated pool$$

A 64-bit genetic algorithm individual is fetched from the dedicated pool of unrated genetic algorithm individuals. The individual is stored in register rd. If there are no individuals available, the processor will stall until there are.

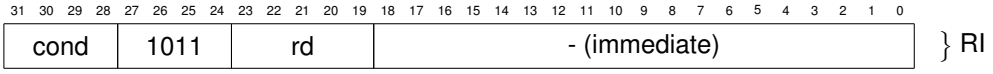
Affected Status Flags



Programming Notes

This instruction is used to interface with the genetics accelerator.

SETG - Set Genetics Pipeline Options



Format

SETG rd

Purpose

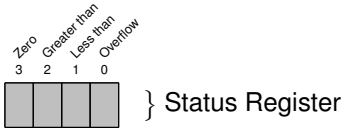
To set options for the genetics pipeline.

Description

The value of rd is passed to the genetics pipeline to set one or more settings.

TODO: list of things to tweak, format for options

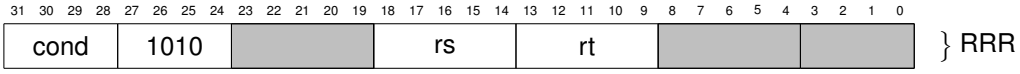
Affected Status Flags



Programming Notes

This instruction is used to interface with the genetics accelerator.

STG - Store Gene to Pool



Format

STG rs, rt

Purpose

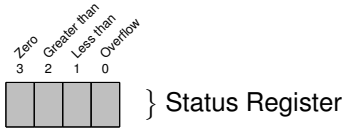
To put a 64-bit genetic algorithm individual with its 64-bit unsigned fitness score in the rated pool.

Description

$$slotinratedpool \leftarrow (rs, rt)$$

The 64-bit genetic algorithm individual in rs is put in the dedicated pool of rated genetic algorithm individuals, together with its 64-bit unsigned fitness score in rt.

Affected Status Flags



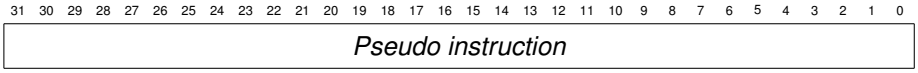
Programming Notes

This instruction is used to interface with the genetics accelerator.

8 Pseudo Instructions

This section describes common instructions that are not implemented in hardware but are given to programmers to ease coding. All the instructions are converted to other instructions during the preprocessing step of the assembler.

CMP - Compare



Format

CMP rs, rt

Purpose

To compare two 64-bit signed numbers.

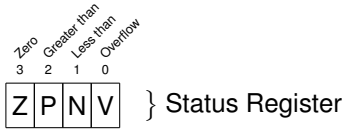
Description

$$statusflags[equal] \leftarrow rs == rt$$
$$statusflags[greaterthan] \leftarrow rs > rt$$
$$statusflags[lessthan] \leftarrow rs < rt$$

Compare the contents of register rs and register rt as signed 64-bit integers. Status flags are set based on the outcome of the comparison.

- If rs and rt are equal, the Equal status flag is set.
- If rs is greater than rt, the Greater status flag is set.
- If rs is smaller than rt, the Less status flag is set.

Affected Status Flags

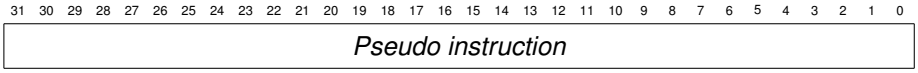


Programming Notes

This instruction is translated into:

SUB, r0, rs, rt

MV - Move



Format

MV rd, rs

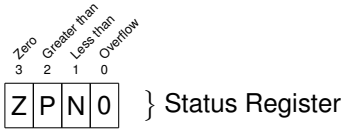
Purpose

To copy a value from one register to another.

Description

The contents of register rs are placed in rd.

Affected Status Flags

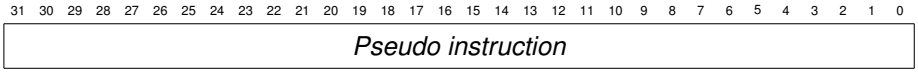


Programming Notes

This instruction is translated into:

ORI rd, rs, 0

NEG - Arithmetical Negation



Format

NEG rd, rs

Purpose

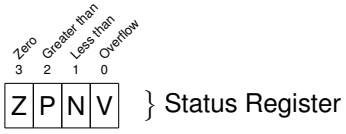
To twos-complement negate a 64-bit word.

Description

$$rd \leftarrow -rs$$

The 64-bit value in register rs is twos-complement negated. The negated result is stored in register rd.

Affected Status Flags

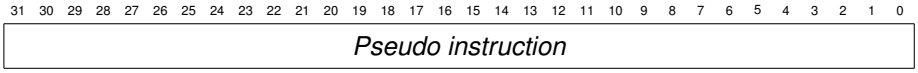


Programming Notes

This instruction is translated into:

SUB rd, r0, rs

NOP - No operation



Format

NOP

Purpose

To do nothing for one clock cycle.

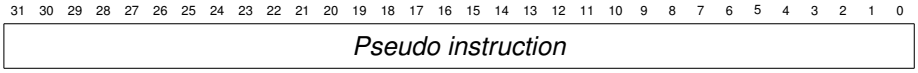
Description

No work is done for one clock cycle.

Programming Notes

The assembler will switch NOPs with an instruction where cond = never.

NOT - Logical NOT



Format

NOT rd, rs

Purpose

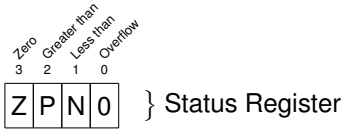
To bitwise logically invert a 64-bit word.

Description

$$rd \leftarrow \neg rs$$

The 64-bit value in register rs is bitwise logically inverted. The inverted result is stored in register rd.

Affected Status Flags



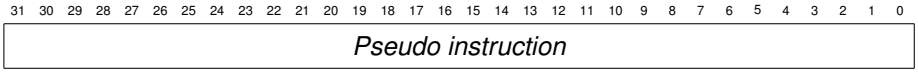
Programming Notes

This instruction is translated into:

XORI rd, rs, -1

TODO: XORI zero-extends, perhaps change all to sign-extend for simplicity?

RET - Return From Procedure



Format

RET

Purpose

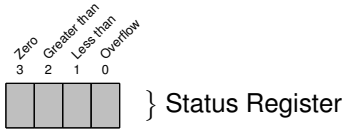
To return from a procedure call.

Description

$$PC \leftarrow r31$$

The least 19 bits of the link register is stored in the program counter.

Affected Status Flags



Programming Notes

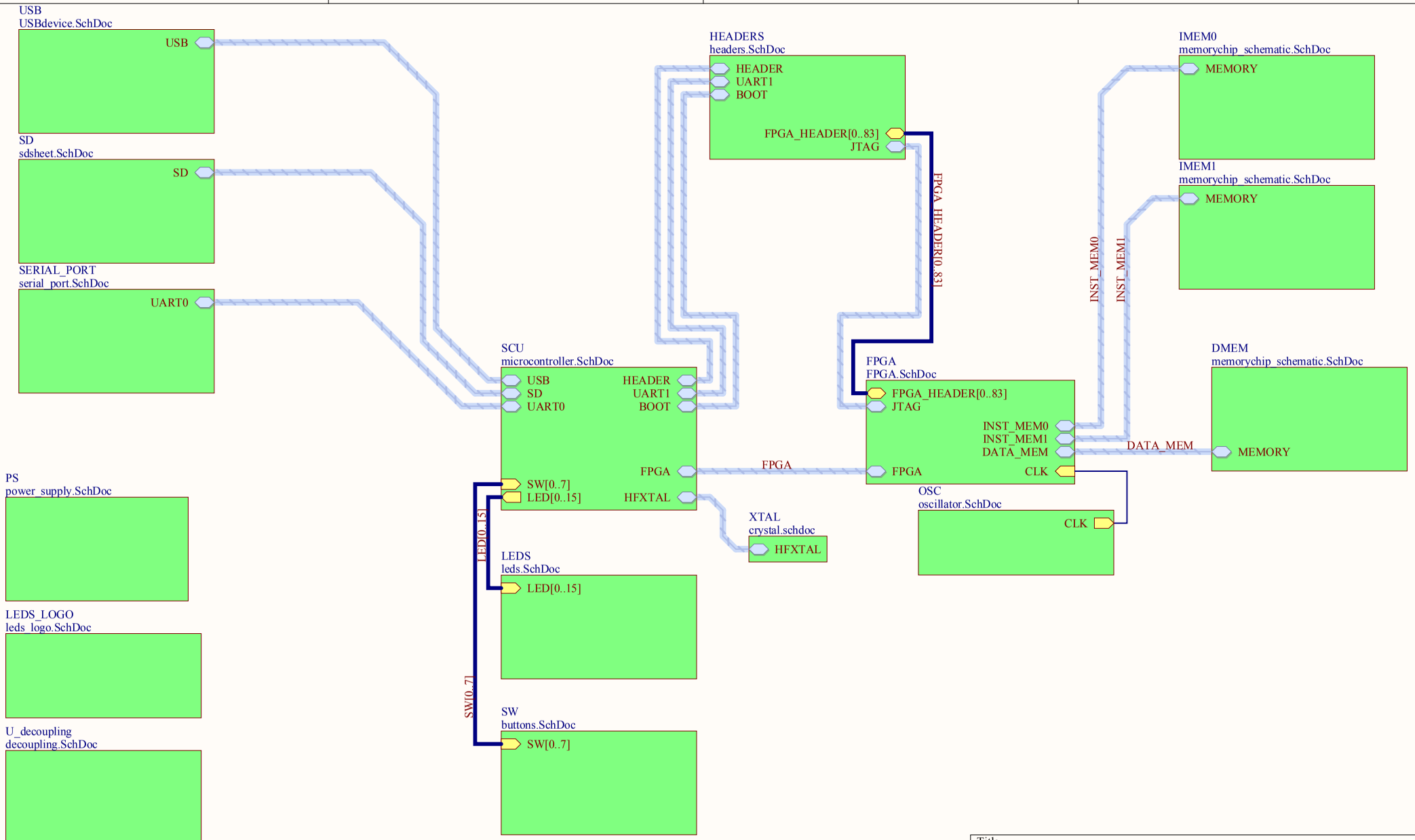
This instruction is translated into:

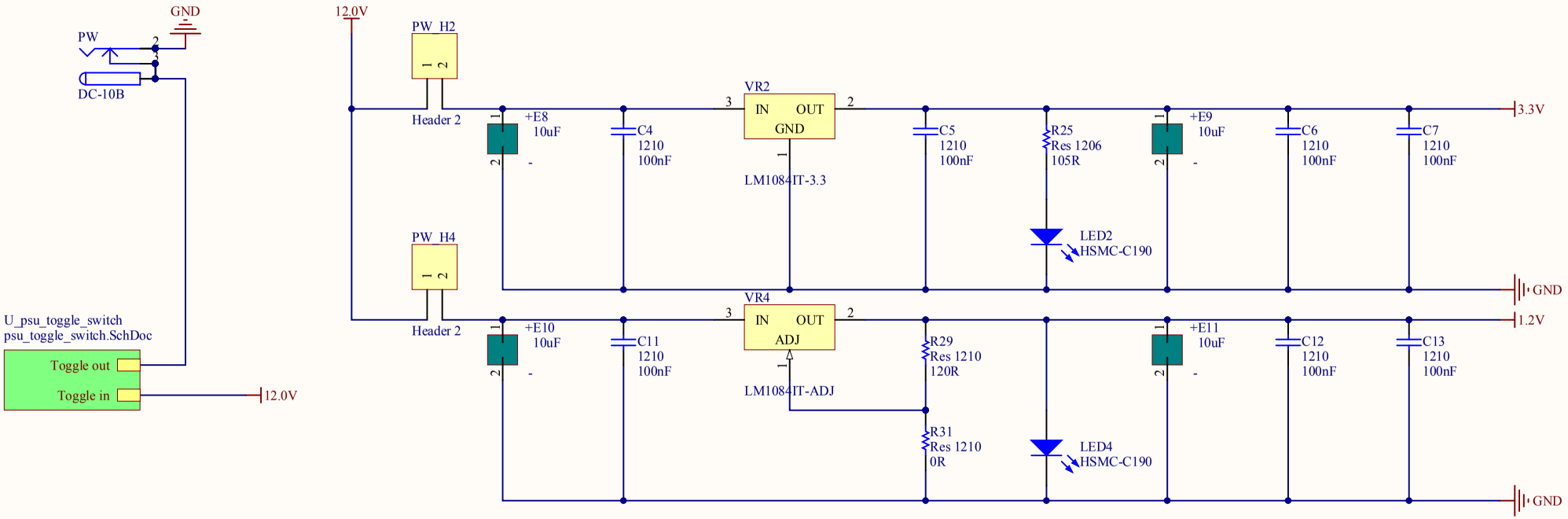
JMP r31, 0

APPENDIX

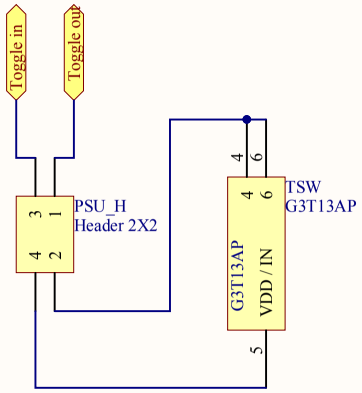
B

PCB SCHEMATICS

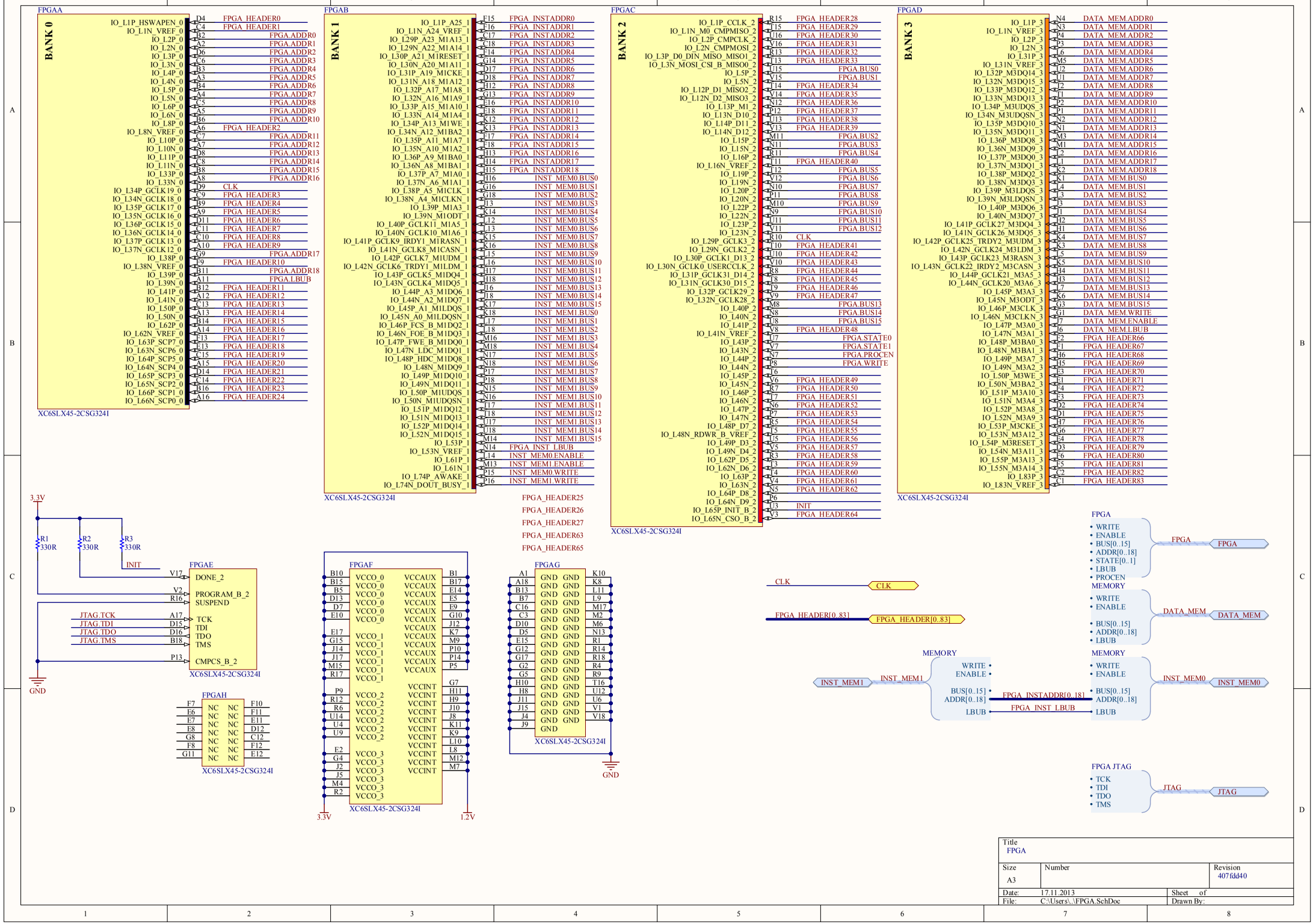

USB
USBdevice.SchDoc
USB
SD
sdsheet.SchDoc
SD
SERIAL_PORT
serial_port.SchDoc
UART0
PS
power_supply.SchDoc
LEDS_LOGO
leds_logo.SchDoc
U_decoupling
decoupling.SchDoc
SCU
microcontroller.SchDoc
USB
SD
UART0
HEADER
UART1
BOOT
FPGA
SW[0..7]
LED[0..15]
HFXTAL
HEADERS
headers.SchDoc
HEADER
UART1
BOOT
FPGA_HEADER[0..83]
JTAG
FPGA_HEADER[0..83]
FPGA
FPGA.SchDoc
FPGA_HEADER[0..83]
JTAG
INST_MEM0
INST_MEM1
DATA_MEM
FPGA
CLK
FPGA
XTAL
crystal.schdoc
HFXTAL
OSC
oscillator.SchDoc
CLK
LEDS
leds.SchDoc
LED[0..15]
LED[0..15]
SW
buttons.SchDoc
SW[0..7]
SW[0..7]
IMEM0
memorychip_schematic.SchDoc
MEMORY
IMEM1
memorychip_schematic.SchDoc
MEMORY
INST_MEM0
INST_MEM1
DMEM
memorychip_schematic.SchDoc
DATA_MEM
MEMORY
Title
Top level
Size
A4
Number
Revision
407fdd40
Date:
17.11.2013
Sheet of
File:
C:\Users\...\top_level.SchDoc
Drawn By:

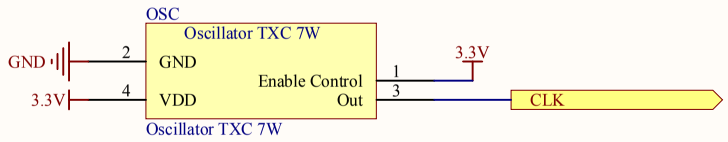


Title: Power supply		
Size: A4	Number:	Revision: 407fdd40
Date: 17.11.2013		Sheet of
File: C:\Users\...\power_supply.SchDoc		Drawn By:

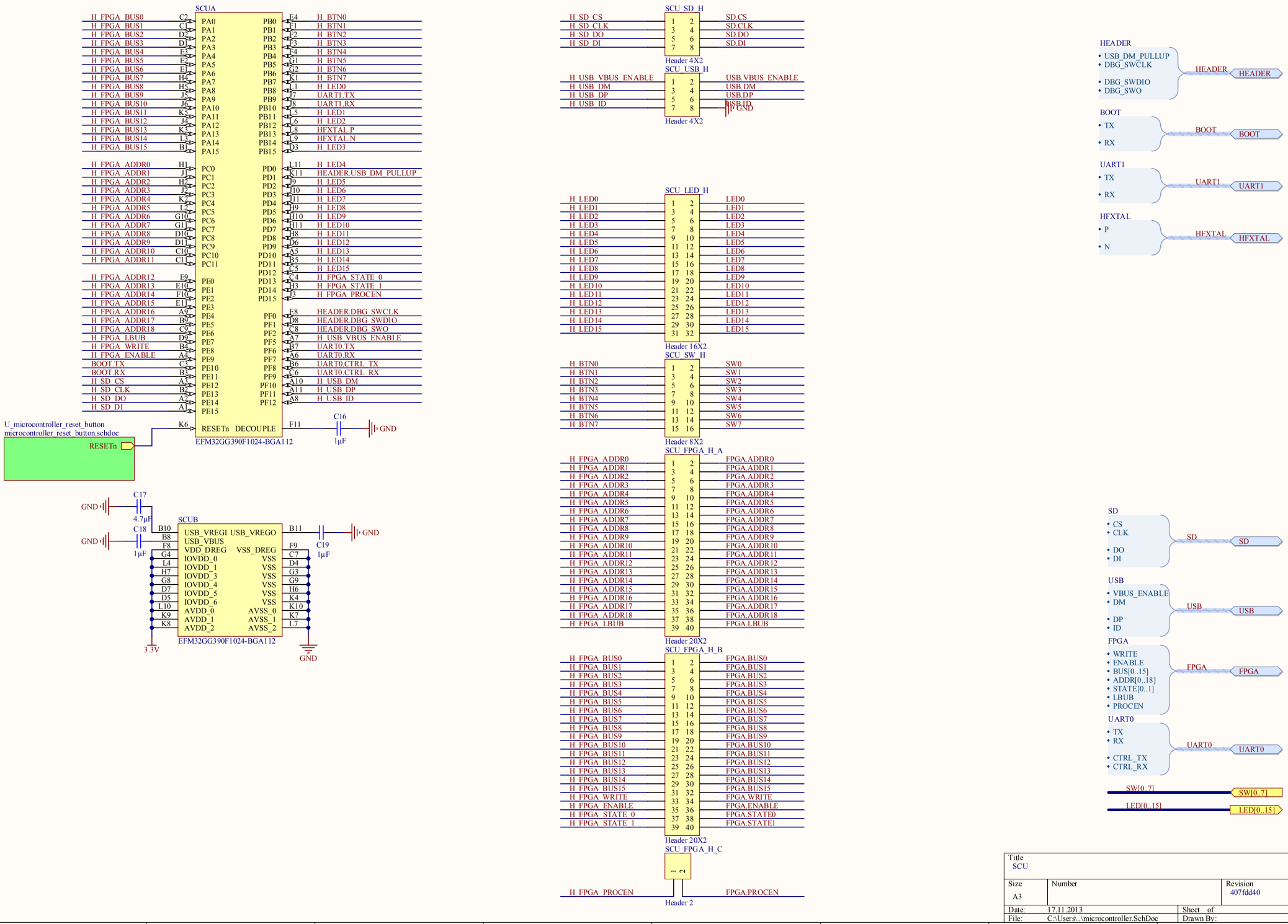


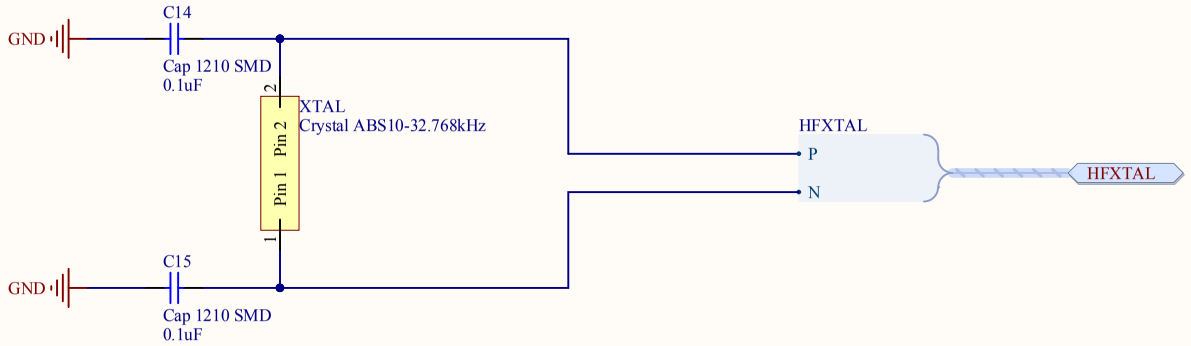
Title: PSU toggle switch		
Size: A4	Number:	Revision: 407fdd40
Date: 17.11.2013		Sheet of
File: C:\Users\...\psu_toggle_switch.SchDoc		Drawn By:


FPGAA
BANK 0
IO_L1P_HSWAPEN_0 D4 FPGA_HEADER0
IO_L1N_VREF_0 C4 FPGA_HEADER1
IO_L2P_0 B2 FPGA.ADDR0
IO_L2N_0 A2 FPGA.ADDR1
IO_L3P_0 D6 FPGA.ADDR2
IO_L3N_0 C6 FPGA.ADDR3
IO_L4P_0 B3 FPGA.ADDR4
IO_L4N_0 A3 FPGA.ADDR5
IO_L5P_0 B4 FPGA.ADDR6
IO_L5N_0 A4 FPGA.ADDR7
IO_L6P_0 C5 FPGA.ADDR8
IO_L6N_0 A5 FPGA.ADDR9
IO_L8P_0 B6 FPGA.ADDR10
IO_L8N_VREF_0 A6 FPGA_HEADER2
IO_L10P_0 C7 FPGA.ADDR11
IO_L10N_0 A7 FPGA.ADDR12
IO_L11P_0 D8 FPGA.ADDR13
IO_L11N_0 C8 FPGA.ADDR14
IO_L33P_0 B8 FPGA.ADDR15
IO_L33N_0 A8 FPGA.ADDR16
IO_L34P_GCLK19_0 D9 CLK
IO_L34N_GCLK18_0 C9 FPGA_HEADER3
IO_L35P_GCLK17_0 B9 FPGA_HEADER4
IO_L35N_GCLK16_0 A9 FPGA_HEADER5
IO_L36P_GCLK15_0 D11 FPGA_HEADER6
IO_L36N_GCLK14_0 C11 FPGA_HEADER7
IO_L37P_GCLK13_0 C10 FPGA_HEADER8
IO_L37N_GCLK12_0 A10 FPGA_HEADER9
IO_L38P_0 G9 FPGA.ADDR17
IO_L38N_VREF_0 F9 FPGA_HEADER10
IO_L39P_0 B11 FPGA.ADDR18
IO_L39N_0 A11 FPGA.LBUB
IO_L41P_0 B12 FPGA_HEADER11
IO_L41N_0 A12 FPGA_HEADER12
IO_L50P_0 C13 FPGA_HEADER13
IO_L50N_0 A13 FPGA_HEADER14
IO_L62P_0 B14 FPGA_HEADER15
IO_L62N_VREF_0 A14 FPGA_HEADER16
IO_L63P_SCP7_0 F13 FPGA_HEADER17
IO_L63N_SCP6_0 E13 FPGA_HEADER18
IO_L64P_SCP5_0 C15 FPGA_HEADER19
IO_L64N_SCP4_0 A15 FPGA_HEADER20
IO_L65P_SCP3_0 D14 FPGA_HEADER21
IO_L65N_SCP2_0 C14 FPGA_HEADER22
IO_L66P_SCP1_0 B16 FPGA_HEADER23
IO_L66N_SCP0_0 A16 FPGA_HEADER24
XC6SLX45-2CSG324I
FPGAB
BANK 1
IO_L1P_A25_1 F15 FPGA_INSTADDR0
IO_L1N_A24_VREF_1 F16 FPGA_INSTADDR1
IO_L29P_A23_M1A13_1 G17 FPGA_INSTADDR2
IO_L29N_A22_M1A14_1 F18 FPGA_INSTADDR3
IO_L30P_A21_M1RESET_1 E14 FPGA_INSTADDR4
IO_L30N_A20_M1A11_1 G14 FPGA_INSTADDR5
IO_L31P_A19_M1CKE_1 D17 FPGA_INSTADDR6
IO_L31N_A18_M1A12_1 D18 FPGA_INSTADDR7
IO_L32P_A17_M1A8_1 H12 FPGA_INSTADDR8
IO_L32N_A16_M1A9_1 G13 FPGA_INSTADDR9
IO_L33P_A15_M1A10_1 F16 FPGA_INSTADDR10
IO_L33N_A14_M1A4_1 F18 FPGA_INSTADDR11
IO_L34P_A13_M1WE_1 E12 FPGA_INSTADDR12
IO_L34N_A12_M1BA2_1 E13 FPGA_INSTADDR13
IO_L35P_A11_M1A7_1 F17 FPGA_INSTADDR14
IO_L35N_A10_M1A2_1 F18 FPGA_INSTADDR15
IO_L36P_A9_M1BA0_1 H13 FPGA_INSTADDR16
IO_L36N_A8_M1BA1_1 H14 FPGA_INSTADDR17
IO_L37P_A7_M1A0_1 H15 FPGA_INSTADDR18
IO_L37N_A6_M1A1_1 H16 INST_MEM0.BUS0
IO_L38P_A5_M1CLK_1 G16 INST_MEM0.BUS1
IO_L38N_A4_M1CLKN_1 G18 INST_MEM0.BUS2
IO_L39P_M1A3_1 J13 INST_MEM0.BUS3
IO_L39N_M1ODT_1 K14 INST_MEM0.BUS4
IO_L40P_GCLK11_M1A5_1 J12 INST_MEM0.BUS5
IO_L40N_GCLK10_M1A6_1 J13 INST_MEM0.BUS6
IO_L41P_GCLK9_IRDY1_M1RASN_1 K15 INST_MEM0.BUS7
IO_L41N_GCLK8_M1CASN_1 K16 INST_MEM0.BUS8
IO_L42P_GCLK7_M1UDM_1 L15 INST_MEM0.BUS9
IO_L42N_GCLK6_TRDY1_M1LDM_1 L16 INST_MEM0.BUS10
IO_L43P_GCLK5_M1DQ4_1 H17 INST_MEM0.BUS11
IO_L43N_GCLK4_M1DQ5_1 H18 INST_MEM0.BUS12
IO_L44P_A3_M1DQ6_1 J16 INST_MEM0.BUS13
IO_L44N_A2_M1DQ7_1 J18 INST_MEM0.BUS14
IO_L45P_A1_M1LDQS_1 K17 INST_MEM0.BUS15
IO_L45N_A0_M1LDQSN_1 K18 INST_MEM1.BUS0
IO_L46P_FCS_B_M1DQ2_1 L17 INST_MEM1.BUS1
IO_L46N_FOE_B_M1DQ3_1 L18 INST_MEM1.BUS2
IO_L47P_FWE_B_M1DQ0_1 M16 INST_MEM1.BUS3
IO_L47N_LDC_M1DQ1_1 M18 INST_MEM1.BUS4
IO_L48P_HDC_M1DQ8_1 N17 INST_MEM1.BUS5
IO_L48N_M1DQ9_1 N18 INST_MEM1.BUS6
IO_L49P_M1DQ10_1 P17 INST_MEM1.BUS7
IO_L49N_M1DQ11_1 P18 INST_MEM1.BUS8
IO_L50P_M1UDQS_1 N15 INST_MEM1.BUS9
IO_L50N_M1UDQSN_1 N16 INST_MEM1.BUS10
IO_L51P_M1DQ12_1 L17 INST_MEM1.BUS11
IO_L51N_M1DQ13_1 L18 INST_MEM1.BUS12
IO_L52P_M1DQ14_1 J17 INST_MEM1.BUS13
IO_L52N_M1DQ15_1 J18 INST_MEM1.BUS14
IO_L53P_1 M14 INST_MEM1.BUS15
IO_L53N_VREF_1 N14 FPGA_INST_LBUB
IO_L61P_1 L14 INST_MEM0.ENABLE
IO_L61N_1 M13 INST_MEM1.ENABLE
IO_L74P_AWAKE_1 P15 INST_MEM0.WRITE
IO_L74N_DOUT_BUSY_1 P16 INST_MEM1.WRITE
XC6SLX45-2CSG324I
FPGAC
BANK 2
IO_L1P_CCLK_2 R15 FPGA_HEADER28
IO_L1N_M0_CMPMISO_2 T15 FPGA_HEADER29
IO_L2P_CMPCLK_2 T16 FPGA_HEADER30
IO_L2N_CMPMOSI_2 V16 FPGA_HEADER31
IO_L3P_D0_DIN_MISO_MISO1_2 R13 FPGA_HEADER32
IO_L3N_MOSI_CSI_B_MISO0_2 T13 FPGA_HEADER33
IO_L5P_2 U15 FPGA.BUS0
IO_L5N_2 V15 FPGA.BUS1
IO_L12P_D1_MISO2_2 T14 FPGA_HEADER34
IO_L12N_D2_MISO3_2 V14 FPGA_HEADER35
IO_L13P_M1_2 N12 FPGA_HEADER36
IO_L13N_D10_2 P12 FPGA_HEADER37
IO_L14P_D11_2 U13 FPGA_HEADER38
IO_L14N_D12_2 V13 FPGA_HEADER39
IO_L15P_2 M11 FPGA.BUS2
IO_L15N_2 N11 FPGA.BUS3
IO_L16P_2 R11 FPGA.BUS4
IO_L16N_VREF_2 T11 FPGA_HEADER40
IO_L19P_2 T12 FPGA.BUS5
IO_L19N_2 V12 FPGA.BUS6
IO_L20P_2 N10 FPGA.BUS7
IO_L20N_2 P11 FPGA.BUS8
IO_L22P_2 M10 FPGA.BUS9
IO_L22N_2 N9 FPGA.BUS10
IO_L23P_2 U11 FPGA.BUS11
IO_L23N_2 V11 FPGA.BUS12
IO_L29P_GCLK3_2 R10 CLK
IO_L29N_GCLK2_2 T10 FPGA_HEADER41
IO_L30P_GCLK1_D13_2 U10 FPGA_HEADER42
IO_L30N_GCLK0_USERCCLK_2 V10 FPGA_HEADER43
IO_L31P_GCLK31_D14_2 R8 FPGA_HEADER44
IO_L31N_GCLK30_D15_2 T8 FPGA_HEADER45
IO_L32P_GCLK29_2 T9 FPGA_HEADER46
IO_L32N_GCLK28_2 V9 FPGA_HEADER47
IO_L40P_2 M8 FPGA.BUS13
IO_L40N_2 N8 FPGA.BUS14
IO_L41P_2 T8 FPGA.BUS15
IO_L41N_VREF_2 V8 FPGA_HEADER48
IO_L43P_2 T7 FPGA.STATE0
IO_L43N_2 V7 FPGA.STATE1
IO_L44P_2 N7 FPGA.PROCEN
IO_L44N_2 P8 FPGA.WRITE
IO_L45P_2 T6
IO_L45N_2 V6 FPGA_HEADER49
IO_L46P_2 R7 FPGA_HEADER50
IO_L46N_2 T7 FPGA_HEADER51
IO_L47P_2 N6 FPGA_HEADER52
IO_L47N_2 P7 FPGA_HEADER53
IO_L48P_D7_2 R5 FPGA_HEADER54
IO_L48N_RDWR_B_VREF_2 T5 FPGA_HEADER55
IO_L49P_D3_2 U5 FPGA_HEADER56
IO_L49N_D4_2 V5 FPGA_HEADER57
IO_L62P_D5_2 R3 FPGA_HEADER58
IO_L62N_D6_2 T3 FPGA_HEADER59
IO_L63P_2 U4 FPGA_HEADER60
IO_L63N_2 V4 FPGA_HEADER61
IO_L64P_D8_2 N5 FPGA_HEADER62
IO_L64N_D9_2 P6
IO_L65P_INIT_B_2 T3 INIT
IO_L65N_CSO_B_2 V3 FPGA_HEADER64
XC6SLX45-2CSG324I
FPGAD
BANK 3
IO_L1P_3 N4 DATA_MEM.ADDR0
IO_L1N_VREF_3 N3 DATA_MEM.ADDR1
IO_L2P_3 P4 DATA_MEM.ADDR2
IO_L2N_3 P3 DATA_MEM.ADDR3
IO_L31P_3 L6 DATA_MEM.ADDR4
IO_L31N_VREF_3 M5 DATA_MEM.ADDR5
IO_L32P_M3DQ14_3 J2 DATA_MEM.ADDR6
IO_L32N_M3DQ15_3 J1 DATA_MEM.ADDR7
IO_L33P_M3DQ12_3 L2 DATA_MEM.ADDR8
IO_L33N_M3DQ13_3 L1 DATA_MEM.ADDR9
IO_L34P_M3UDQS_3 P2 DATA_MEM.ADDR10
IO_L34N_M3UDQSN_3 P1 DATA_MEM.ADDR11
IO_L35P_M3DQ10_3 N2 DATA_MEM.ADDR12
IO_L35N_M3DQ11_3 N1 DATA_MEM.ADDR13
IO_L36P_M3DQ8_3 M3 DATA_MEM.ADDR14
IO_L36N_M3DQ9_3 M1 DATA_MEM.ADDR15
IO_L37P_M3DQ0_3 L2 DATA_MEM.ADDR16
IO_L37N_M3DQ1_3 L1 DATA_MEM.ADDR17
IO_L38P_M3DQ2_3 K2 DATA_MEM.ADDR18
IO_L38N_M3DQ3_3 K1 DATA_MEM.BUS0
IO_L39P_M3LDQS_3 J4 DATA_MEM.BUS1
IO_L39N_M3LDQSN_3 J3 DATA_MEM.BUS2
IO_L40P_M3DQ6_3 J3 DATA_MEM.BUS3
IO_L40N_M3DQ7_3 J1 DATA_MEM.BUS4
IO_L41P_GCLK27_M3DQ4_3 H2 DATA_MEM.BUS5
IO_L41N_GCLK26_M3DQ5_3 H1 DATA_MEM.BUS6
IO_L42P_GCLK25_TRDY2_M3UDM_3 K4 DATA_MEM.BUS7
IO_L42N_GCLK24_M3LDM_3 K3 DATA_MEM.BUS8
IO_L43P_GCLK23_M3RASN_3 L5 DATA_MEM.BUS9
IO_L43N_GCLK22_IRDY2_M3CASN_3 K5 DATA_MEM.BUS10
IO_L44P_GCLK21_M3A5_3 J4 DATA_MEM.BUS11
IO_L44N_GCLK20_M3A6_3 J3 DATA_MEM.BUS12
IO_L45P_M3A3_3 L7 DATA_MEM.BUS13
IO_L45N_M3ODT_3 K6 DATA_MEM.BUS14
IO_L46P_M3CLK_3 G3 DATA_MEM.BUS15
IO_L46N_M3CLKN_3 G1 DATA_MEM.WRITE
IO_L47P_M3A0_3 J7 DATA_MEM.ENABLE
IO_L47N_M3A1_3 J6 DATA_MEM.LBUB
IO_L48P_M3BA0_3 F2 FPGA_HEADER66
IO_L48N_M3BA1_3 F1 FPGA_HEADER67
IO_L49P_M3A7_3 J6 FPGA_HEADER68
IO_L49N_M3A2_3 J5 FPGA_HEADER69
IO_L50P_M3WE_3 F3 FPGA_HEADER70
IO_L50N_M3BA2_3 F1 FPGA_HEADER71
IO_L51P_M3A10_3 F4 FPGA_HEADER72
IO_L51N_M3A4_3 F3 FPGA_HEADER73
IO_L52P_M3A8_3 D2 FPGA_HEADER74
IO_L52N_M3A9_3 D1 FPGA_HEADER75
IO_L53P_M3CKE_3 J7 FPGA_HEADER76
IO_L53N_M3A12_3 G6 FPGA_HEADER77
IO_L54P_M3RESET_3 F4 FPGA_HEADER78
IO_L54N_M3A11_3 D3 FPGA_HEADER79
IO_L55P_M3A13_3 E6 FPGA_HEADER80
IO_L55N_M3A14_3 E5 FPGA_HEADER81
IO_L83P_3 D2 FPGA_HEADER82
IO_L83N_VREF_3 C1 FPGA_HEADER83
XC6SLX45-2CSG324I
FPGA_HEADER25
FPGA_HEADER26
FPGA_HEADER27
FPGA_HEADER63
FPGA_HEADER65
3.3V
R1 330R
R2 330R
R3 330R
INIT
FPGAE
V17 DONE_2
V2 PROGRAM_B_2
R16 SUSPEND
JTAG.TCK A17 TCK
JTAG.TDI D15 TDI
JTAG.TDO D16 TDO
JTAG.TMS B18 TMS
P13 CMPCS_B_2
XC6SLX45-2CSG324I
GND
FPGAH
E7 E6 E7 E8 G8 F8 G11
NC NC F10 F11 E11 D12 C12 F12 E12
XC6SLX45-2CSG324I
FPGAF
B10 B15 B5 D13 D7 E10 E17 G15 J14 J17 M15 R17 P9 R12 R6 U14 U4 U9 E2 G4 J2 J5 M4 R2
VCCO_0 VCCO_0 VCCO_0 VCCO_0 VCCO_0 VCCO_0 VCCO_1 VCCO_1 VCCO_1 VCCO_1 VCCO_1 VCCO_1 VCCO_2 VCCO_2 VCCO_2 VCCO_2 VCCO_2 VCCO_2 VCCO_3 VCCO_3 VCCO_3 VCCO_3 VCCO_3 VCCO_3
VCCAUX B1 B17 E14 E5 E9 G10 J12 K7 M9 P10 P14 P5
VCCINT G7 H11 H9 J10 J8 K11 K9 L10 L8 M12 M7
XC6SLX45-2CSG324I
3.3V
1.2V
FPGAG
A1 A18 B13 B7 C16 C3 D10 D5 E15 G12 G17 G2 G5 H10 H8 J11 J15 J4 J9
GND GND K10 K8 L11 L9 M17 M2 M6 N13 R1 R14 R18 R4 R9 T16 U12 U6 V1 V18 GND
XC6SLX45-2CSG324I
GND
CLK
CLK
FPGA_HEADER[0..83]
FPGA_HEADER[0..83]
MEMORY
INST_MEM1
INST_MEM1
WRITE
ENABLE
BUS[0..15]
ADDR[0..18]
LBUB
FPGA_INSTADDR[0..18]
FPGA_INST_LBUB
FPGA
WRITE
ENABLE
BUS[0..15]
ADDR[0..18]
STATE[0..1]
LBUB
PROCEN
FPGA
FPGA
MEMORY
WRITE
ENABLE
BUS[0..15]
ADDR[0..18]
LBUB
DATA_MEM
DATA_MEM
MEMORY
WRITE
ENABLE
BUS[0..15]
ADDR[0..18]
LBUB
INST_MEM0
INST_MEM0
FPGA JTAG
TCK
TDI
TDO
TMS
JTAG
JTAG
Title
FPGA
Size
A3
Number
Revision
407fdd40
Date: 17.11.2013
Sheet of
File: C:\Users\..\FPGA.SchDoc
Drawn By:

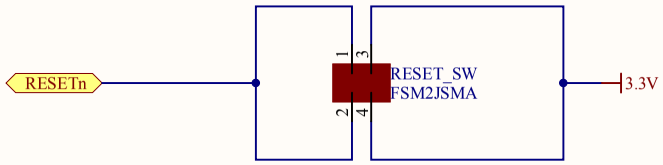


Title: Oscillator		
Size: A4	Number:	Revision: 407fdd40
Date: 17.11.2013		Sheet of
File: C:\Users\..\oscillator.SchDoc		Drawn By:

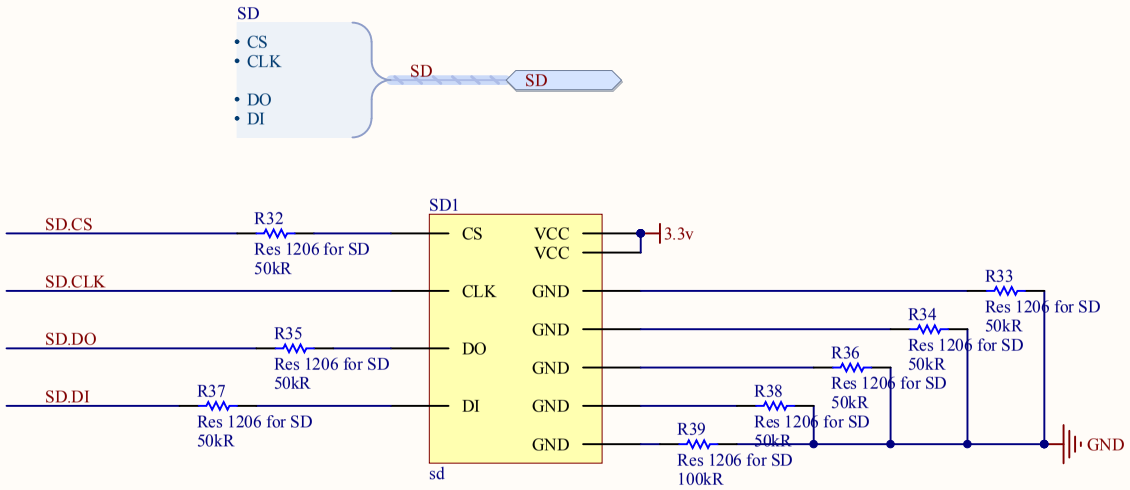

SCUA
PA0 PA1 PA2 PA3 PA4 PA5 PA6 PA7 PA8 PA9 PA10 PA11 PA12 PA13 PA14 PA15
PB0 PB1 PB2 PB3 PB4 PB5 PB6 PB7 PB8 PB9 PB10 PB11 PB12 PB13 PB14 PB15
PC0 PC1 PC2 PC3 PC4 PC5 PC6 PC7 PC8 PC9 PC10 PC11
PD0 PD1 PD2 PD3 PD4 PD5 PD6 PD7 PD8 PD9 PD10 PD11 PD12 PD13 PD14 PD15
PE0 PE1 PE2 PE3 PE4 PE5 PE6 PE7 PE8 PE9 PE10 PE11 PE12 PE13 PE14 PE15
PF0 PF1 PF2 PF5 PF6 PF7 PF8 PF9 PF10 PF11 PF12
RESETn DECOUPLE
EFM32GG390F1024-BGA112
H_FPGA_BUS0 H_FPGA_BUS1 H_FPGA_BUS2 H_FPGA_BUS3 H_FPGA_BUS4 H_FPGA_BUS5 H_FPGA_BUS6 H_FPGA_BUS7 H_FPGA_BUS8 H_FPGA_BUS9 H_FPGA_BUS10 H_FPGA_BUS11 H_FPGA_BUS12 H_FPGA_BUS13 H_FPGA_BUS14 H_FPGA_BUS15
H_FPGA_ADDR0 H_FPGA_ADDR1 H_FPGA_ADDR2 H_FPGA_ADDR3 H_FPGA_ADDR4 H_FPGA_ADDR5 H_FPGA_ADDR6 H_FPGA_ADDR7 H_FPGA_ADDR8 H_FPGA_ADDR9 H_FPGA_ADDR10 H_FPGA_ADDR11 H_FPGA_ADDR12 H_FPGA_ADDR13 H_FPGA_ADDR14 H_FPGA_ADDR15 H_FPGA_ADDR16 H_FPGA_ADDR17 H_FPGA_ADDR18 H_FPGA_LBUB H_FPGA_WRITE H_FPGA_ENABLE BOOT.TX BOOT.RX H_SD_CS H_SD_CLK H_SD_DO H_SD_DI
H_BTN0 H_BTN1 H_BTN2 H_BTN3 H_BTN4 H_BTN5 H_BTN6 H_BTN7 H_LED0 UART1.TX UART1.RX H_LED1 H_LED2 HFXTAL.P HFXTAL.N H_LED3
H_LED4 HEADER.USB_DM_PULLUP H_LED5 H_LED6 H_LED7 H_LED8 H_LED9 H_LED10 H_LED11 H_LED12 H_LED13 H_LED14 H_LED15 H_FPGA_STATE_0 H_FPGA_STATE_1 H_FPGA_PROCEN HEADER.DBG_SWCLK HEADER.DBG_SWDIO HEADER.DBG_SWO H_USB_VBUS_ENABLE UART0.TX UART0.RX H_USB_DM H_USB_DP H_USB_ID
U_microcontroller_reset_button
microcontroller_reset_button.schdoc
RESETn
C16
1µF
GND
SCUB
USB_VREGI USB_VREGO
USB_VBUS
VDD_DREG VSS_DREG
IOVDD_0 IOVDD_1 IOVDD_3 IOVDD_4 IOVDD_5 IOVDD_6
AVDD_0 AVDD_1 AVDD_2
VSS
AVSS_0 AVSS_1 AVSS_2
EFM32GG390F1024-BGA112
C17
4.7µF
C18
1µF
C19
1µF
3.3V
SCU_SD_H
SD.CS SD.CLK SD.DO SD.DI
Header 4X2
SCU_USB_H
USB.VBUS_ENABLE USB.DM USB.DP USB.ID
Header 4X2
SCU_LED_H
LED0 LED1 LED2 LED3 LED4 LED5 LED6 LED7 LED8 LED9 LED10 LED11 LED12 LED13 LED14 LED15
Header 16X2
SCU_SW_H
SW0 SW1 SW2 SW3 SW4 SW5 SW6 SW7
Header 8X2
SCU_FPGA_H_A
FPGA.ADDR0 FPGA.ADDR1 FPGA.ADDR2 FPGA.ADDR3 FPGA.ADDR4 FPGA.ADDR5 FPGA.ADDR6 FPGA.ADDR7 FPGA.ADDR8 FPGA.ADDR9 FPGA.ADDR10 FPGA.ADDR11 FPGA.ADDR12 FPGA.ADDR13 FPGA.ADDR14 FPGA.ADDR15 FPGA.ADDR16 FPGA.ADDR17 FPGA.ADDR18 FPGA.LBUB
Header 20X2
SCU_FPGA_H_B
FPGA.BUS0 FPGA.BUS1 FPGA.BUS2 FPGA.BUS3 FPGA.BUS4 FPGA.BUS5 FPGA.BUS6 FPGA.BUS7 FPGA.BUS8 FPGA.BUS9 FPGA.BUS10 FPGA.BUS11 FPGA.BUS12 FPGA.BUS13 FPGA.BUS14 FPGA.BUS15 FPGA.WRITE FPGA.ENABLE FPGA.STATE0 FPGA.STATE1
Header 20X2
SCU_FPGA_H_C
H_FPGA_PROCEN FPGA.PROCEN
Header 2
HEADER
USB_DM_PULLUP DBG_SWCLK DBG_SWDIO DBG_SWO
BOOT
TX RX
UART1
TX RX
HFXTAL
P N
SD
CS CLK DO DI
USB
VBUS_ENABLE DM DP ID
FPGA
WRITE ENABLE BUS[0..15] ADDR[0..18] STATE[0..1] LBUB PROCEN
UART0
TX RX CTRL_TX CTRL_RX
SW[0..7]
LED[0..15]
Title
SCU
Size
A3
Number
Revision
407fdd40
Date: 17.11.2013
File: C:\Users\...\microcontroller.SchDoc
Sheet of
Drawn By:



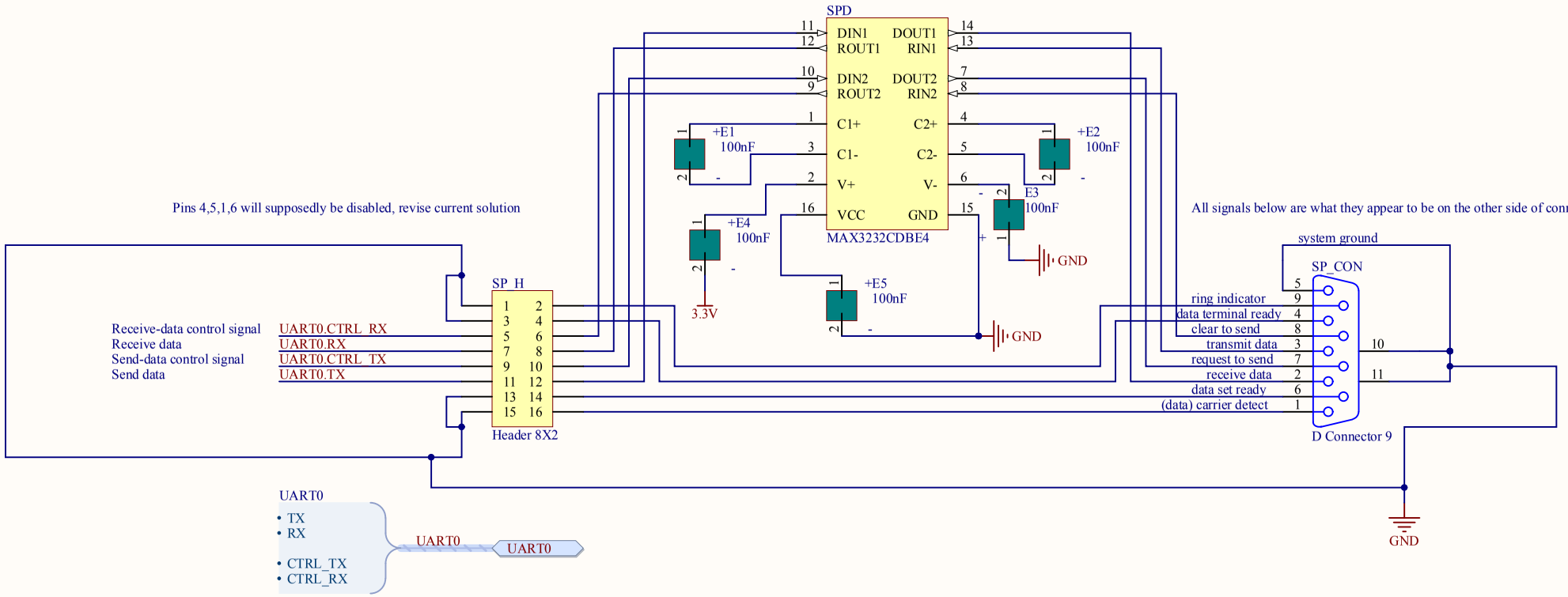
Title: Crystal		
Size: A4	Number:	Revision: 407fdd40
Date: 17.11.2013		Sheet of
File: C:\Users\...\crystal.schdoc		Drawn By:



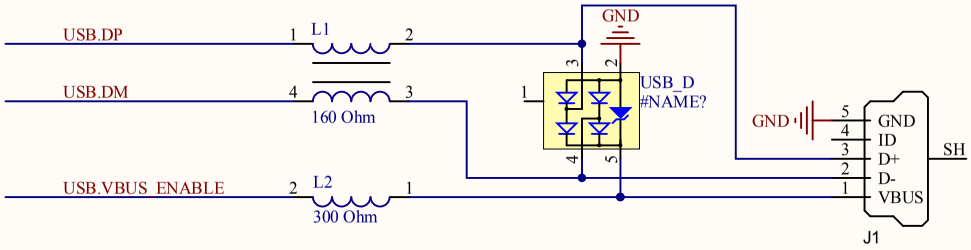
Title: Reset switch		
Size: A4	Number:	Revision: 407fdd40
Date: 17.11.2013	Sheet of	
File: C:\Users\...\microcontroller_reset_button.schdoc	Drawn By:	

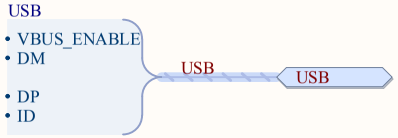


Title: SD		
Size: A4	Number:	Revision: 407fdd40
Date: 17.11.2013		Sheet of
File: C:\Users\..\sdsheet.SchDoc		Drawn By:

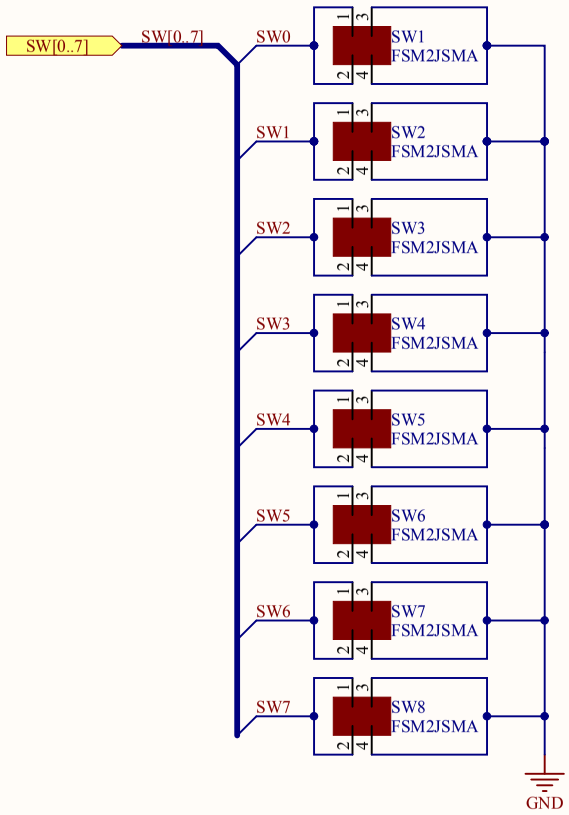


Title: Serial port		
Size: A4	Number:	Revision: 407fdd40
Date: 17.11.2013		Sheet of
File: C:\Users\...\serial_port.SchDoc		Drawn By:

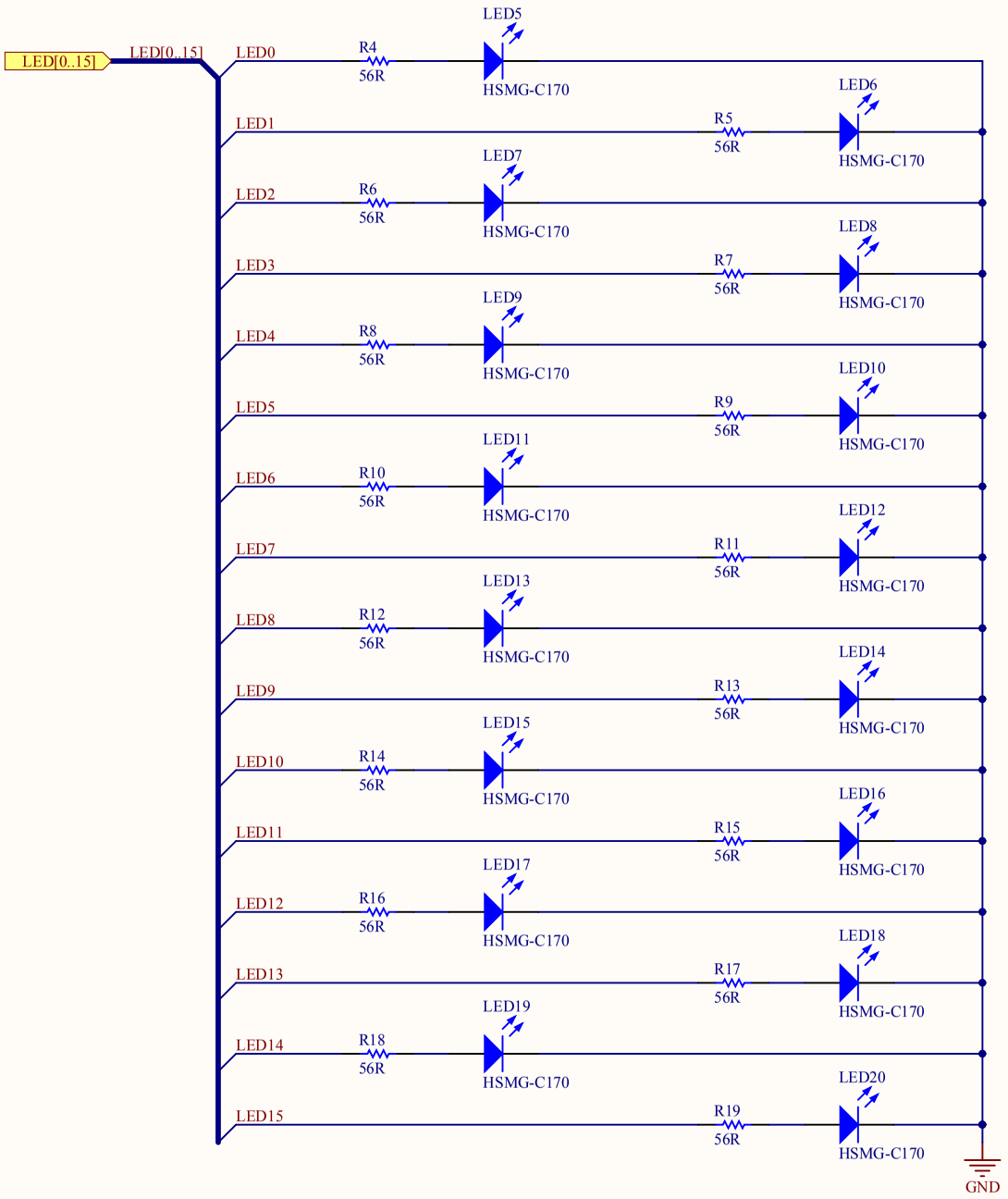




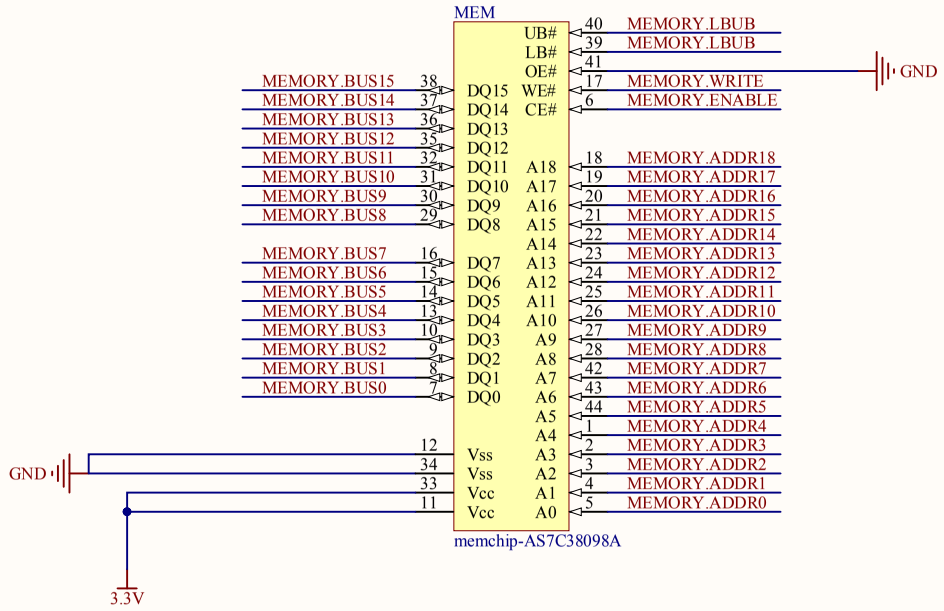
Title: USB		
Size: A4	Number:	Revision: 407fdd40
Date: 17.11.2013		Sheet of
File: C:\Users\...\USBdevice.SchDoc		Drawn By:



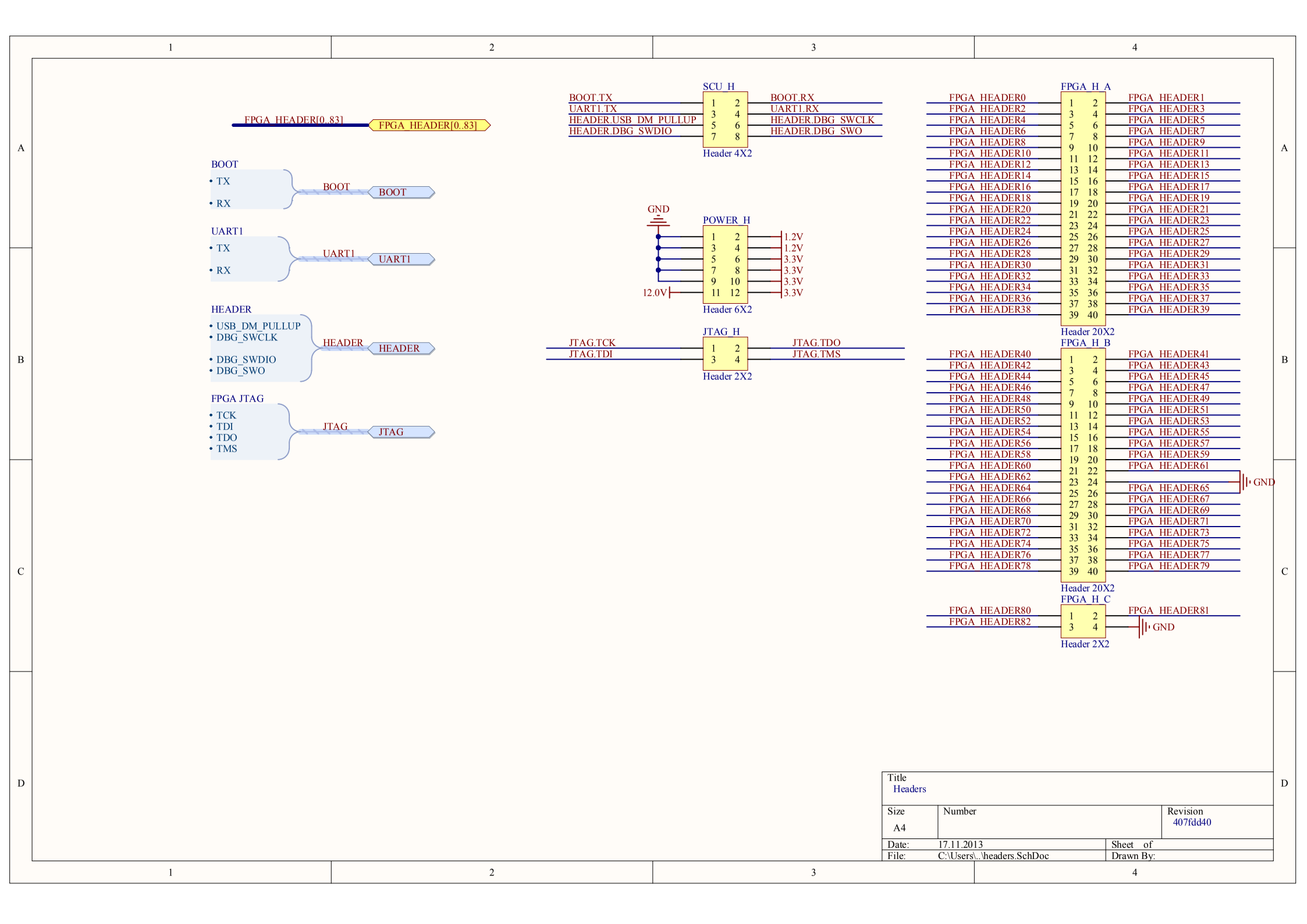
Title: Buttons		
Size: A4	Number:	Revision: 407fdd40
Date: 17.11.2013		Sheet of
File: C:\Users\..\buttons.SchDoc		Drawn By:

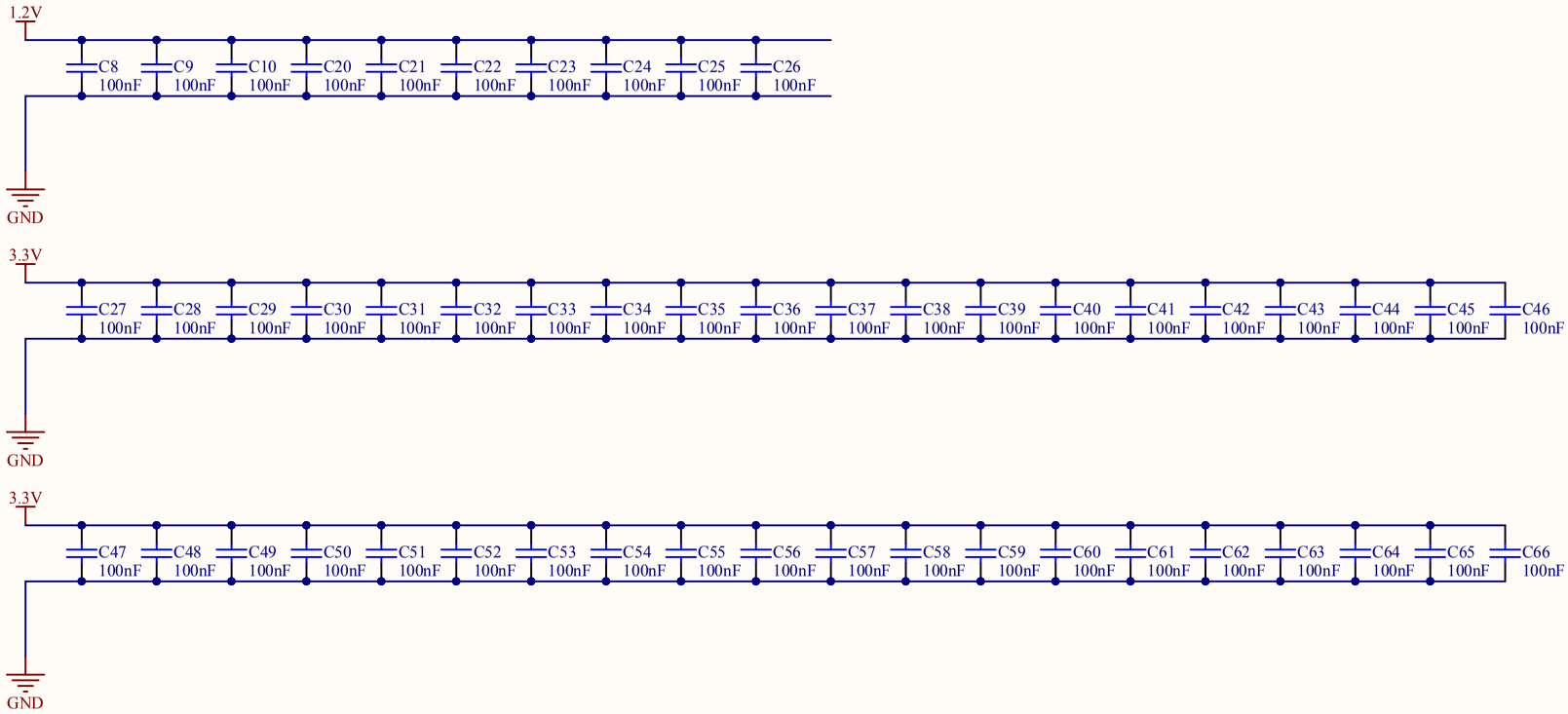


Title: User leds			
Size: A4	Number:	Revision: 407fdd40	
Date: 17.11.2013		Sheet of	
File: C:\Users\..\leds.SchDoc		Drawn By:	

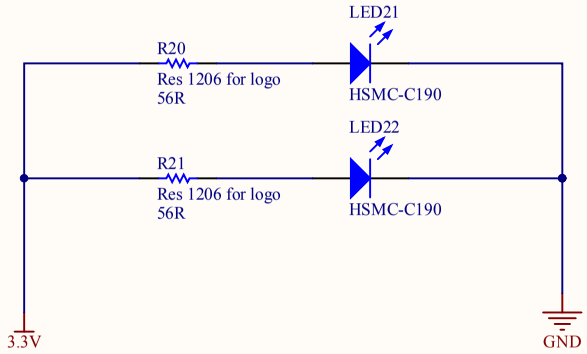


Title: Memorychip		
Size: A4	Number:	Revision: 407fdd40
Date: 17.11.2013	Sheet of	
File: C:\Users\...\memorychip_schematic.SchDoc	Drawn By:	

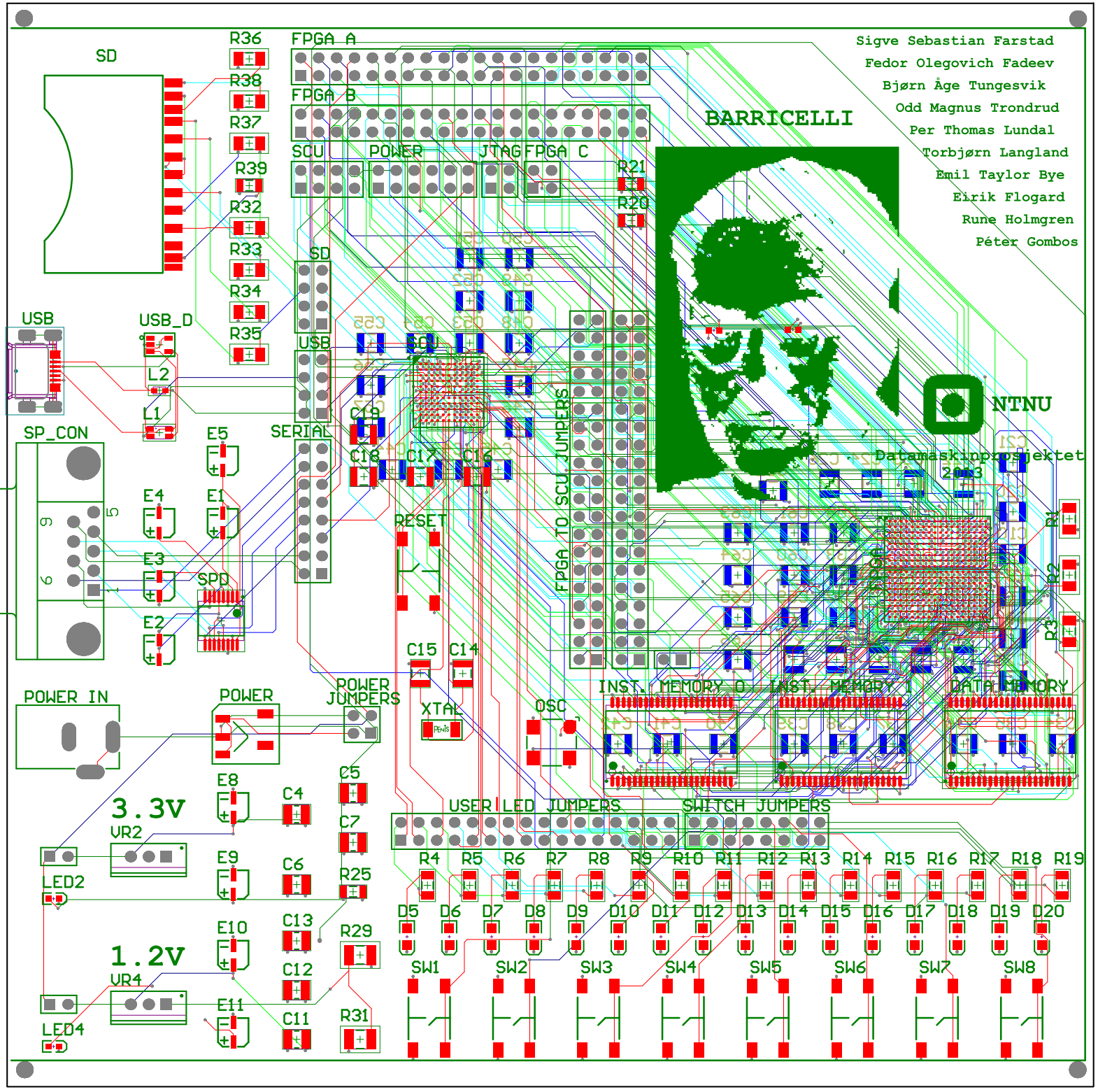

FPGA_HEADER[0..83]
FPGA_HEADER[0..83]
BOOT
• TX
• RX
BOOT
BOOT
UART1
• TX
• RX
UART1
UART1
HEADER
• USB_DM_PULLUP
• DBG_SWCLK
• DBG_SWDIO
• DBG_SWO
HEADER
HEADER
FPGA JTAG
• TCK
• TDI
• TDO
• TMS
JTAG
JTAG
SCU_H
BOOT.TX
UART1.TX
HEADER.USB_DM_PULLUP
HEADER.DBG_SWDIO
1 2
3 4
5 6
7 8
BOOT.RX
UART1.RX
HEADER.DBG_SWCLK
HEADER.DBG_SWO
Header 4X2
GND
POWER_H
1 2
3 4
5 6
7 8
9 10
11 12
12.0V
1.2V
1.2V
3.3V
3.3V
3.3V
3.3V
Header 6X2
JTAG_H
JTAG.TCK
JTAG.TDI
1 2
3 4
JTAG.TDO
JTAG.TMS
Header 2X2
FPGA_H_A
FPGA_HEADER0
FPGA_HEADER2
FPGA_HEADER4
FPGA_HEADER6
FPGA_HEADER8
FPGA_HEADER10
FPGA_HEADER12
FPGA_HEADER14
FPGA_HEADER16
FPGA_HEADER18
FPGA_HEADER20
FPGA_HEADER22
FPGA_HEADER24
FPGA_HEADER26
FPGA_HEADER28
FPGA_HEADER30
FPGA_HEADER32
FPGA_HEADER34
FPGA_HEADER36
FPGA_HEADER38
1 2
3 4
5 6
7 8
9 10
11 12
13 14
15 16
17 18
19 20
21 22
23 24
25 26
27 28
29 30
31 32
33 34
35 36
37 38
39 40
FPGA_HEADER1
FPGA_HEADER3
FPGA_HEADER5
FPGA_HEADER7
FPGA_HEADER9
FPGA_HEADER11
FPGA_HEADER13
FPGA_HEADER15
FPGA_HEADER17
FPGA_HEADER19
FPGA_HEADER21
FPGA_HEADER23
FPGA_HEADER25
FPGA_HEADER27
FPGA_HEADER29
FPGA_HEADER31
FPGA_HEADER33
FPGA_HEADER35
FPGA_HEADER37
FPGA_HEADER39
Header 20X2
FPGA_H_B
FPGA_HEADER40
FPGA_HEADER42
FPGA_HEADER44
FPGA_HEADER46
FPGA_HEADER48
FPGA_HEADER50
FPGA_HEADER52
FPGA_HEADER54
FPGA_HEADER56
FPGA_HEADER58
FPGA_HEADER60
FPGA_HEADER62
FPGA_HEADER64
FPGA_HEADER66
FPGA_HEADER68
FPGA_HEADER70
FPGA_HEADER72
FPGA_HEADER74
FPGA_HEADER76
FPGA_HEADER78
1 2
3 4
5 6
7 8
9 10
11 12
13 14
15 16
17 18
19 20
21 22
23 24
25 26
27 28
29 30
31 32
33 34
35 36
37 38
39 40
FPGA_HEADER41
FPGA_HEADER43
FPGA_HEADER45
FPGA_HEADER47
FPGA_HEADER49
FPGA_HEADER51
FPGA_HEADER53
FPGA_HEADER55
FPGA_HEADER57
FPGA_HEADER59
FPGA_HEADER61
GND
FPGA_HEADER65
FPGA_HEADER67
FPGA_HEADER69
FPGA_HEADER71
FPGA_HEADER73
FPGA_HEADER75
FPGA_HEADER77
FPGA_HEADER79
Header 20X2
FPGA_H_C
FPGA_HEADER80
FPGA_HEADER82
1 2
3 4
FPGA_HEADER81
GND
Header 2X2
Title
Headers
Size
A4
Number
Revision
407fdd40
Date:
17.11.2013
Sheet of
File:
C:\Users\...\headers.SchDoc
Drawn By:



Title: Decoupling		
Size: A4	Number:	Revision: 407fdd40
Date: 17.11.2013		Sheet of
File: C:\Users\..\decoupling.SchDoc		Drawn By:



Title: Barricellis eyes			
Size: A4	Number:		Revision: 407fdd40
Date: 17.11.2013		Sheet of	
File: C:\Users\..\leds_logo.SchDoc		Drawn By:	


Sigve Sebastian Farstad
Fedor Olegovich Fadeev
Bjørn Åge Tungesvik
Odd Magnus Trondrud
Per Thomas Lundal
Torbjørn Langland
Emil Taylor Bye
Eirik Flogard
Rune Holmgren
Péter Gombos
BARRICELLI
NTNU
Datamaskinprosjektet
2003
SD
FPGA A
FPGA B
SCU
POWER
JTAG
FPGA C
USB
USB_D
SERIAL
SP_CON
SPD
RESET
FPGA TO SCU JUMPERS
POWER IN
POWER
POWER JUMPERS
XTAL
OSC
INST. MEMORY 0
INST. MEMORY 1
DATA MEMORY
USER LED JUMPERS
SWITCH JUMPERS
3.3V
VR2
LED2
1.2V
VR4
LED4
SW1
SW2
SW3
SW4
SW5
SW6
SW7
SW8

APPENDIX

C

CASE SCHEMATICS

Figure C.1: Case design

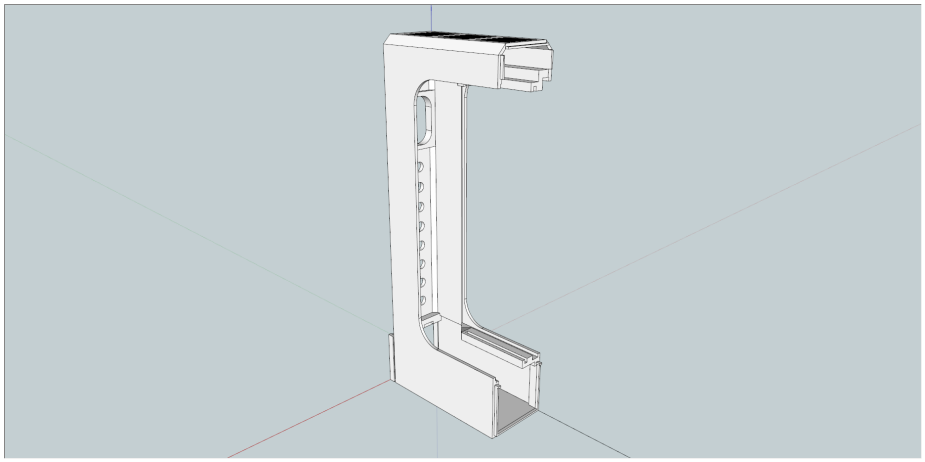


Figure C.2: The front of the case

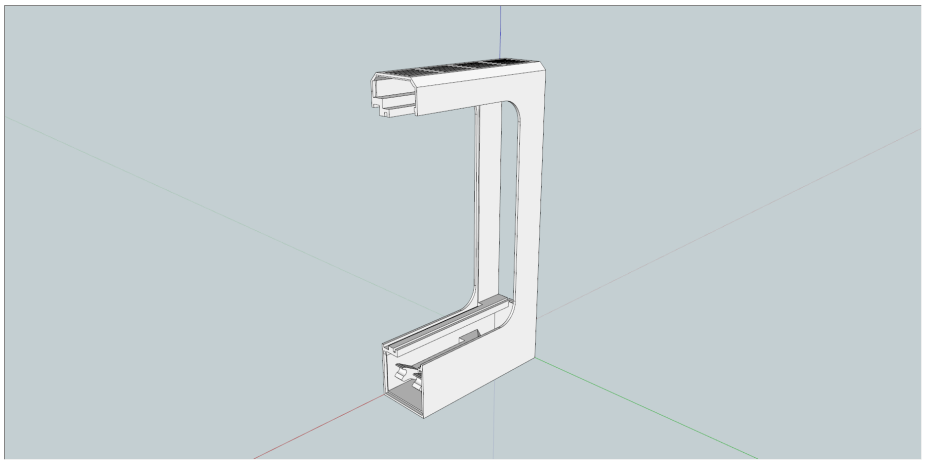


Figure C.3: The back of the case

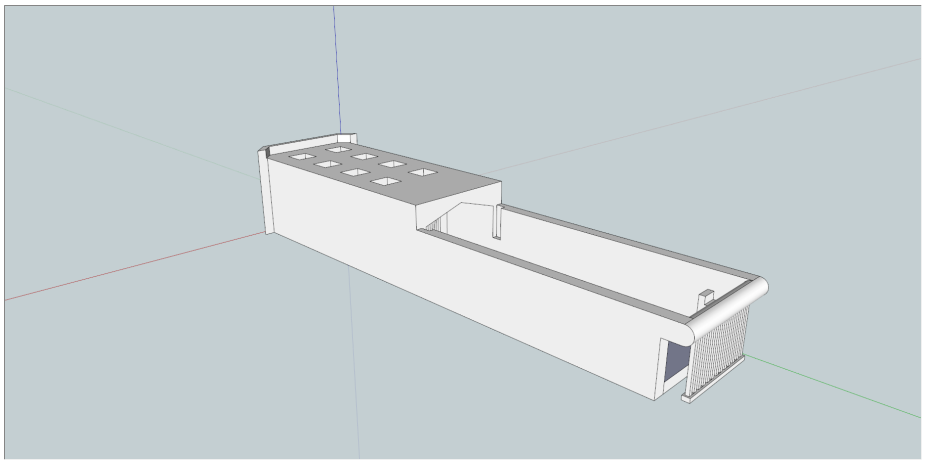


Figure C.4: The keyboard tray

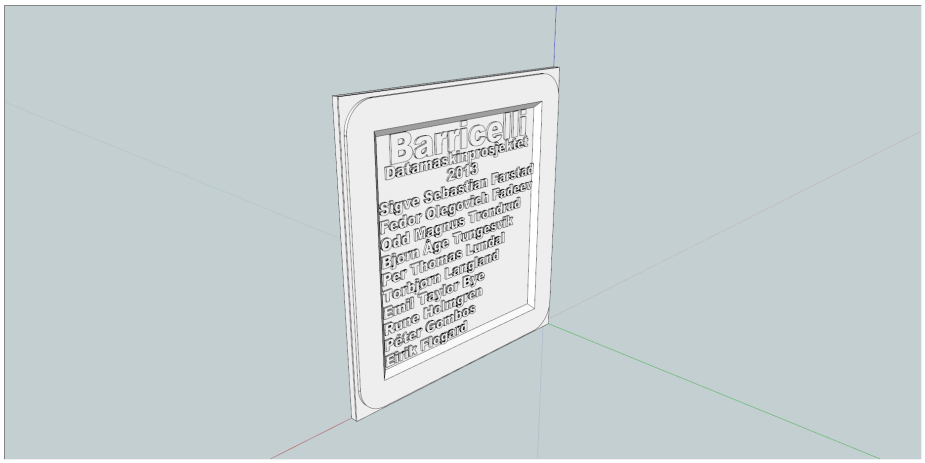


Figure C.5: Side panel with the names of the team members

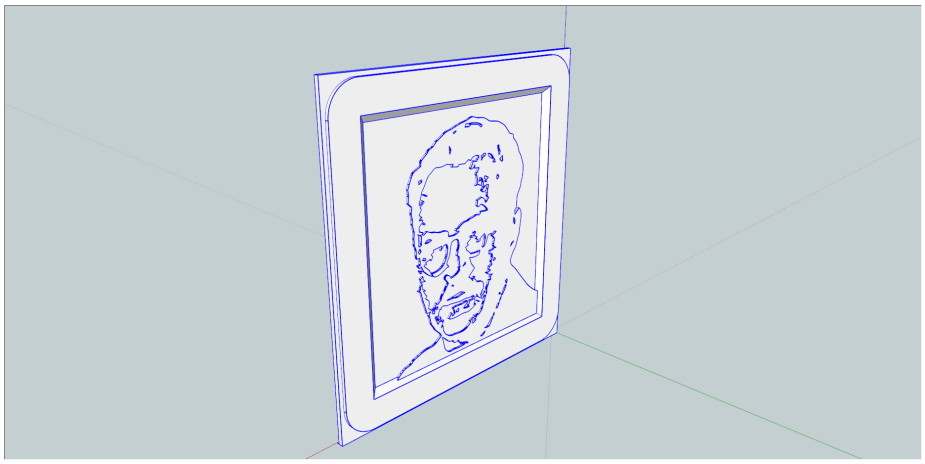


Figure C.6: The side panel with the picture of Nils Aall Barricelli

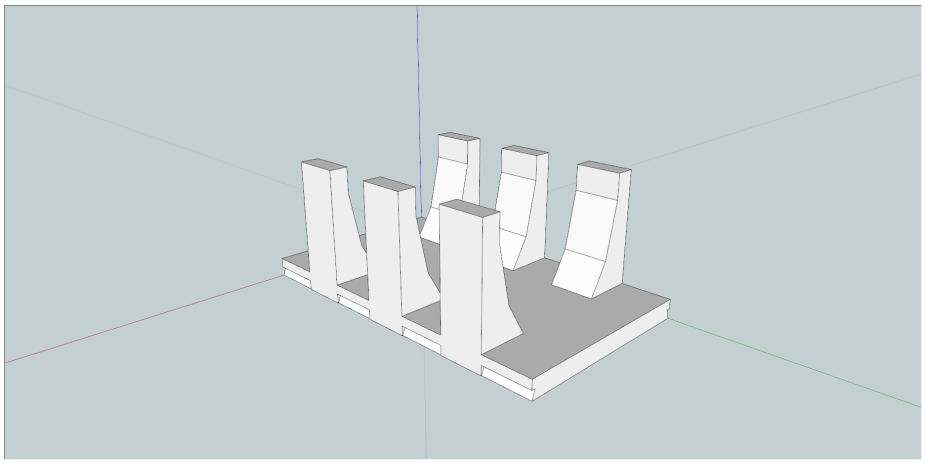


Figure C.7: The support holding the buttons of the keyboard in place

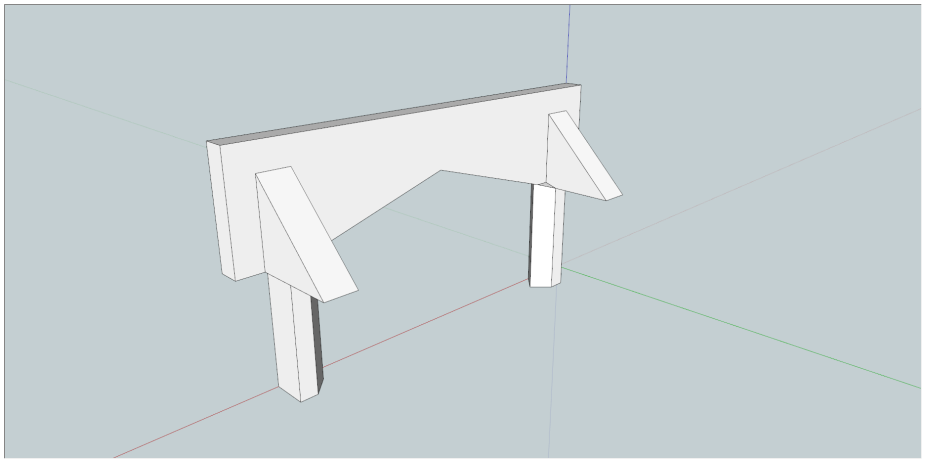


Figure C.8: The plastic edge that stop the keyboard from sliding all the way out

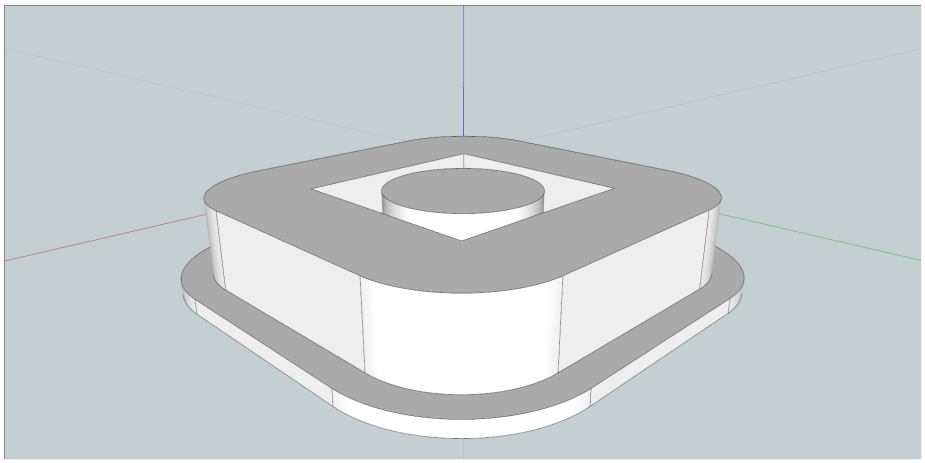


Figure C.9: The blue part of the NTNU logo in front of the case

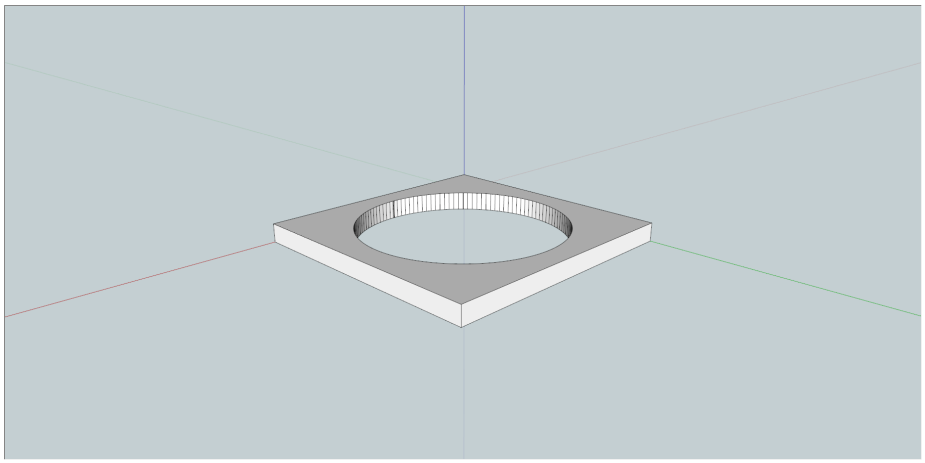


Figure C.10: The blue part of the NTNU logo in front of the case

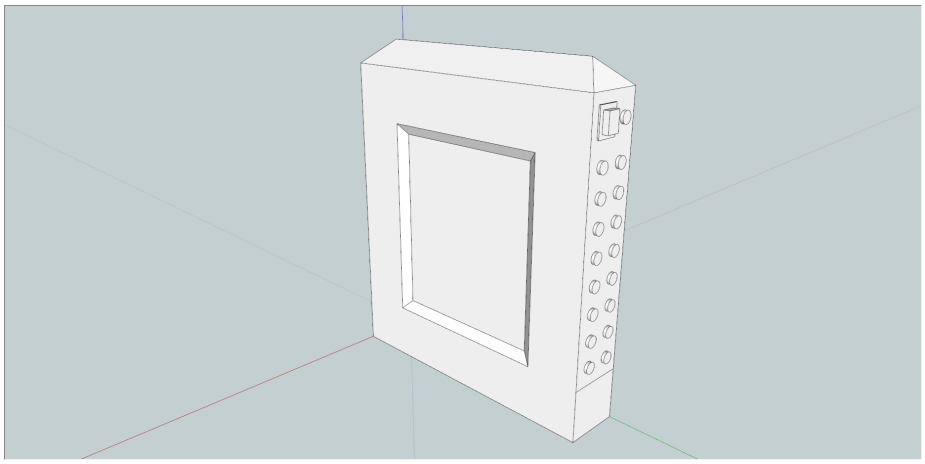


Figure C.11: An early sketch of the case

APPENDIX

D

GALAPAGOS ASSEMBLER LISTING

galapagos–assembler/galapagos/__init__.py

```
import sys
import glob
from assembler import (assemble, ascii_binary_to_real_binary,
                       ascii_binary_to_vhdl_code)


def main():

    if len(sys.argv) == 1:
        print "USAGE: galapagos-as [-d] source.gas [source2.gas ...]"
        print "use --ascii to get output ascii strings of 0s and 1s" \
            "instead of real binary"
        print "use --vhdl to generate fake VHDL ram for testing purposes"
        return

    debug = '-d' in sys.argv

    paths = filter(lambda x: x[0] != '-',
                   [item for sublist in map(glob.glob, sys.argv[1:])
                    for item in sublist])

    for path in paths:
        with open(path, 'r') as f:
            assembly = [line for line in f]
        assembled = assemble(assembly)

        if not assembled:
            return

        address = 1
        ascii_binary = []
        for line in assembled:
            ascii_binary.append(line.toBinary())
            if debug:
                print 'Address: %s' % address
                print 'text: ' + str(line)
                print line.toBinary(debug=debug)
                print
            address += 1

        ascii_binary = ''.join(ascii_binary)

        ascii_mode = '--ascii' in sys.argv
        vhdl_mode = '--vhdl' in sys.argv

        if vhdl_mode:
            ascii_binary_to_vhdl_code(ascii_binary)

        with open(path + '.out', 'w') as f:
            f.write(
                ascii_binary
                if ascii_mode else
                ascii_binary_to_real_binary(ascii_binary)
            )


if __name__ == '__main__':
    main()
```

galapagos–assembler/galapagos/assembler.py

```
from scanner import scanner
from base import Cond
import instructions
import base
import inspect
import re


instruction_map = {}

for name, obj in inspect.getmembers(instructions, inspect.isclass):
    if base.Instruction in inspect.getmro(obj):
        instruction_map[name] = obj


def assemble(lines):

    stripped_lines = strip_comments(map(str.strip, lines))
    tokenized_lines = [(line_number, scanner.scan(line)) for
                       line_number, line in stripped_lines]

    # nop insertion pass
    hot_register = None
    for i, line_tuple in enumerate(tokenized_lines):
        line_number, line = line_tuple
        tokens, rest = line
        token_type, value = tokens[0]
        if token_type == 'operator':
            if value == 'ld':
                hot_register = tokens[1][1]
            else:
                for token in tokens[1:]:
                    token_type, value = token
```

```
                    if token_type == 'register' and value == hot_register:
                        tokenized_lines.insert(i, ([('operator', 'nop')], ''))
                        hot_register = None

    # simple label pass
    labels = {}
    current_address = 1
    for line_number, line in tokenized_lines:
        tokens, rest = line
        token_type, value = tokens[0]
        if token_type == 'label':
            labels[value] = current_address

        elif token_type in ('operator', 'if'):
            current_address += 1

    #actual assemble pass
    assembly = []
    for line_number, line in tokenized_lines:
        tokens, rest = line
        if rest:
            print '[%s]: Tokenizer error near' % line_number, rest
            return

        tokens = iter(tokens)
        condition = Cond.ALWAYS

        try:

            while True:
                token = tokens.next()
                token_type, value = token

                if token_type == 'if':
                    token = tokens.next()
                    token_type, value = token
                    prefix = ''
                    if token == ('operator', 'not'):
                        token = tokens.next()
                        token_type, value = token
                        prefix = 'not_'
                    if token_type == 'condition':
                        condition = Cond.fromString(prefix + value)
                    else:
                        print 'Your if makes no sense. Line number:',
                        print line_number

                elif token_type == 'operator':

                    instruction = value.lower()

                    # Three params
                    if instruction in ['add', 'addi', 'and', 'andi', 'ld',
                                       'mul', 'muli', 'or', 'ori', 'sll',
                                       'slli', 'sra', 'srai', 'srl', 'srli',
                                       'st', 'sub', 'subi', 'xor', 'xori']:
                        _, a = tokens.next()
                        _, b = tokens.next()
                        _, c = tokens.next()
                        instruction = instruction_map[
                            value.capitalize()](a, b, c)

                    # Two params
                    elif instruction in ['jmp', 'ldi', 'sti', 'stg',
                                         'cmp', 'mv', 'neg', 'not']:
                        maybe_label, a = tokens.next()
                        if maybe_label == 'label':
                            b = labels[a]
                            a = 0
                        else:
                            _, b = tokens.next()
                        instruction = instruction_map[value.capitalize()](a, b)

                    # One param
                    elif instruction in ['ldg', 'setg']:
                        _, a = tokens.next()
                        instruction = instruction_map[value.capitalize()](a)

                    # No params
                    elif instruction in ['nop', 'ret']:
                        instruction = instruction_map[value.capitalize()]()

                    # special cases
                    elif instruction == 'call':
                        maybe_label, a = tokens.next()
                        if maybe_label == 'label':
                            b = labels[a]
                            a = 0
                        else:
                            _, b = tokens.next()
                        instruction = instruction_map[value.capitalize()](a, b)

                    if instruction.cond == Cond.UNSET:
                        instruction.cond = condition
                    assembly.append(instruction)

        except StopIteration:
            pass
```

```
    return assembly


def strip_comments(lines):
    newline_delimitor = '!newline_delimitor!'
    line_number_delimitor = '!line_number_delimitor!'
    lines = [str(line_number) + line_number_delimitor + line for
             line_number, line in enumerate(lines)]
    joined = newline_delimitor.join(lines)
    stripped = re.sub('(/\*([^*]|(\*+[^*/]))*\*+/)', '', joined)
    annotated = filter(lambda x: x[1],
                       [line.split(line_number_delimitor)
                        for line in stripped.split(newline_delimitor)])
    return [(int(line_number), line) for line_number, line in annotated]


def ascii_binary_to_real_binary(ascii_binary):
    return ''.join(
        [chr(int('0b'+ascii_binary[i:i+8], 2))
            for i in range(0, len(ascii_binary), 8)]
    )


def ascii_binary_to_vhdl_code(ascii_binary):

    def zero_pad(string, length):
        return string + "0" * max(length - len(string), 0)

    number_of_instructions = len(ascii_binary)/32

    print "type mem is array (0 to " + str(number_of_instructions + 1) + ")",
    print "of std_logic_vector(16-1 downto 0);"
    print ""
    print "constant hi : mem := ("
    print "    0 => \"0000000000000000\","
    i = 0
    for _ in range(number_of_instructions):
        print "    " + str(i + 1) + " => \"" + \
            zero_pad(ascii_binary[i * 32: i * 32 + 16], 16) + "\","
        i += 1
    print "    " + str(i + 1) + " => \"0000000000000000\""
    print ");"

    print "constant lo : mem := ("
    print "    0 => \"0000000000000000\","
    i = 0
    for _ in range(number_of_instructions):
        print "    " + str(i + 1) + " => \"" + \
            zero_pad(ascii_binary[i * 32 + 16: i * 32 + 32], 16) + "\","
        i += 1
    print "    " + str(i + 1) + " => \"0000000000000000\""
    print ");"
```

galapagos–assembler/galapagos/base.py

```

def binary(num, size):
    if(num < 0):
        num = ~num + 1
        representation = bin(num)[2:]
        while(len(representation) < size):
            representation = '1' + representation
    else:
        representation = bin(num)[2:]
        while(len(representation) < size):
            representation = '0' + representation
    return representation


class Cond(object):
    NEVER = 0b0000
    EQUAL = 0b0001
    NOT_EQUAL = 0b0010
    GREATER_EQUAL = 0b0011
    GREATER = 0b0100
    LESS_EQUAL = 0b0101
    LESS = 0b0110
    OVERFLOW = 0b0111
    NOT_OVERFLOW = 0b1000
    ALWAYS = 0b1111
    UNSET = -1

    @staticmethod
    def fromString(string):
        return {
            'equal': Cond.EQUAL,
            'not_equal': Cond.NOT_EQUAL,
            'greater_than': Cond.GREATER,
            'greater_than_or_equal': Cond.GREATER_EQUAL,
            'less_than': Cond.LESS,
            'less_than_or_equal': Cond.LESS_EQUAL,
            'zero': Cond.EQUAL,
            'not_zero': Cond.NOT_EQUAL,
            'positive': Cond.GREATER,
            'positive_or_zero': Cond.GREATER_EQUAL,
            'negative': Cond.LESS,
```

```
42              'negative_or_zero': Cond.LESS_EQUAL,
43              'overflow': Cond.OVERFLOW,
44              'not_overflow': Cond.NOT_OVERFLOW,
45              'never': Cond.NEVER,
46              'always': Cond.ALWAYS,
47          }[string.lower()]
48
49
50  class Instruction(object):
51
52      cond = Cond.UNSET
53      opcode = 0
54
55
56  class RRR(Instruction):
57
58      rd = 0
59      rs = 0
60      rt = 0
61      function = 0
62
63      def __init__(self, rd, rs, rt):
64          super(RRR, self).__init__()
65          self.rd = rd
66          self.rs = rs
67          self.rt = rt
68
69      def toBinary(self, debug=False):
70          return ['', 'cond__opcode__rd_____rs_____N/A____rt_____function\n'][
71              debug] + \
72              binary(self.cond, 4) + \
73              ['', '__'][debug] + \
74              binary(self.opcode, 4) + \
75              ['', '____'][debug] + \
76              binary(self.rd, 5) + \
77              ['', '__'][debug] + \
78              binary(self.rs, 5) + \
79              ['', '__'][debug] + \
80              '00000' + \
81              ['', '__'][debug] + \
82              binary(self.rt, 5) + \
83              ['', '__'][debug] + \
84              binary(self.function, 4)
85
86      def __repr__(self):
87          return self.__class__.__name__.lower() + '_' + str(self.rd) + \
88              ',_' + str(self.rs) + ',_' + str(self.rt)
89
90
91  class RRI(Instruction):
92
93      rd = 0
94      rs = 0
95      immediate = 0
96      function = 0
97
98      def __init__(self, rd, rs, immediate):
99          super(RRI, self).__init__()
100         self.rd = rd
101         self.rs = rs
102         self.immediate = immediate
103
104     def toBinary(self, debug=False):
105         return ['', 'cond__opcode__rd_____rs_____immediate___function\n'][
106             debug] + \
107             binary(self.cond, 4) + \
108             ['', '__'][debug] + \
109             binary(self.opcode, 4) + \
110             ['', '____'][debug] + \
111             binary(self.rd, 5) + \
112             ['', '__'][debug] + \
113             binary(self.rs, 5) + \
114             ['', '__'][debug] + \
115             binary(self.immediate, 10) + \
116             ['', '__'][debug] + \
117             binary(self.function, 4)
118
119     def __repr__(self):
120         return self.__class__.__name__.lower() + '_' + str(self.rd) + \
121             ',_' + str(self.rs) + ',_' + str(self.immediate)
122
123
124 class RI(Instruction):
125
126     rd = 0
127     immediate = 0
128
129     def __init__(self, rd, immediate):
130         super(RI, self).__init__()
131         self.rd = rd
132         self.immediate = immediate
133
134     def toBinary(self, debug=False):
135         return ['', 'cond__opcode__rd_____immediate\n'][debug] + \
136             binary(self.cond, 4) + \
137             ['', '__'][debug] + \
138             binary(self.opcode, 4) + \
139             ['', '____'][debug] + \
140             binary(self.rd, 5) + \
```

```
            ['', '__'][debug] + \
            binary(self.immediate, 19)

    def __repr__(self):
        return self.__class__.__name__.lower() + '_' + \
            str(self.rd) + ',_' + str(self.immediate)
```

galapagos–assembler/galapagos/instructions.py

```
from base import RRR, RRI, RI, Cond, Instruction


class Add(RRR):
    opcode = 0b1000
    function = 0


class Addi(RRI):
    opcode = 0b1100
    function = 0


class And(RRR):
    opcode = 0b1000
    function = 0b0101


class Andi(RRI):
    opcode = 0b1100
    function = 0b0101


class Mul(RRR):
    opcode = 0b1000
    function = 0b0010


class Muli(RRI):
    opcode = 0b1100
    function = 0b0010


class Or(RRR):
    opcode = 0b1000
    function = 0b0100


class Ori(RRI):
    opcode = 0b1100
    function = 0b0100


class Sll(RRR):
    opcode = 0b1000
    function = 0b0111


class Slli(RRI):
    opcode = 0b1100
    function = 0b0111


class Sra(RRR):
    opcode = 0b1000
    function = 0b0011


class Srai(RRI):
    opcode = 0b1100
    function = 0b0011


class Srl(RRR):
    opcode = 0b1000
    function = 0b1000


class Srli(RRI):
    opcode = 0b1100
    function = 0b1000


class Sub(RRR):
    opcode = 0b1000
    function = 0b0001


class Subi(RRI):
    opcode = 0b1100
    function = 0b0001


class Xor(RRR):
    opcode = 0b1000
    function = 0b0110

```

```
88
89  class Xori(RRI):
90      opcode = 0b1100
91      function = 0b0110
92
93
94  class Call(RI):
95      opcode = 0b0011
96
97
98  class Jmp(RI):
99      opcode = 0b0010
100
101
102 class Ld(RRI):
103     opcode = 0b0000
104
105     def __init__(self, rd, rs, immediate):
106         self.rd = rd
107         self.rs = rs
108         self.immediate = immediate
109
110
111 class Ldi(RI):
112     opcode = 0b0100
113
114
115 class St(RRI):
116     opcode = 0b0001
117
118     def __init__(self, rd, rs, immediate):
119         self.rd = rd
120         self.rs = rs
121         self.immediate = immediate
122
123
124 class Sti(RI):
125     opcode = 0b0101
126
127
128 class Ldg(RRR):
129     opcode = 0b1001
130
131     def __init__(self, rd):
132         self.rd = rd
133
134
135 class Setg(RI):
136     opcode = 0b1011
137
138     def __init__(self, rd):
139         self.rd = rd
140
141
142 class Stg(RRR):
143
144     opcode = 0b1010
145
146     def __init__(self, rs, rt):
147         self.rs = rs
148         self.rt = rt
149
150
151 # Pseudo instructions
152
153 class Cmp(Instruction):
154     def __new__(self, rs, rt):
155         return Sub(0, rs, rt)
156
157
158 class Mv(Instruction):
159     def __new__(self, rd, rs):
160         return Ori(rd, rs, 0)
161
162
163 class Neg(Instruction):
164     def __new__(self, rd, rs):
165         return Sub(rd, 0, rs)
166
167
168 class Nop(Instruction):
169     def __new__(self):
170         nop = Add(0, 0, 0)
171         nop.cond = Cond.NEVER
172         return nop
173
174
175 class Not(Instruction):
176     def __new__(self, rd, rs):
177         return Xori(rd, rs, -1)
178
179
180 class Ret(Instruction):
181     def __new__(self):
182         return Jmp(31, 0)
```

galapagos–assembler/galapagos/scanner.py

```
import re


def s_label(scanner, token):
    return ('label', token)


def s_condition(scanner, token):
    return ('condition', token)


def s_register(scanner, token):
    return ('register', int(token[1:]))


def s_int_10(scanner, token):
    return ('int', int(token, 10))


def s_int_16(scanner, token):
    return ('int', int(token, 16))


def s_int_2(scanner, token):
    return ('int', int(token, 2))


def s_operator(scanner, token):
    return ('operator', token)


def s_if(scanner, token):
    return ('if', token)


def s_whitespace(scanner, token):
    return


scanner = re.Scanner([
    ('[\n_\t\v,:]+', s_whitespace),
    (r'\b(r[0-9]+\b)', s_register),
    (r'\bif\b', s_if),
    (r'-?0x[0-9a-fA-F]+', s_int_16),
    (r'-?0b[01]+', s_int_2),
    (r'-?[0-9]+', s_int_10),
    (r'\b(add|addi|and|andi|call|call|jmp|ld|ldi|ret|' +
     r'ldg|mul|muli|or|ori|setg|sll|slli|sra|srai|srl|' +
     r'srli|st|sti|stg|sub|subi|xor|xori|cmp|mv|neg|nop|not)\b', s_operator),
    ('(equal|not_equal|greater_than|greater_than_or_equal|less_than|' +
     r'less_than_or_equal|zero|not_zero|positive|positive_or_zero|' +
     r'negative|negative_or_zero|overflow|not_overflow|never|always)',
     s_condition),
    ('[^_:0-9\n\t\v][^_:\n\t\v]*', s_label),
])
```

APPENDIX

E

DEMONSTRATION PROGRAM LISTINGS

GAS–programs/knapsack/solver.gas

```
/*
 * 0x1-0x40  - 64 x ((weight << 32) + value)
 * 0x41      - weight limit
 * 0x42 + 2n - best genome + best fitness, for each core
 *
 * r1: Best genome position
 * r2: weight limit
 * r3: gene
 * r4: weight used
 * r5: score
 * r6: Item flag (1, 2, 4, ..., 1<<63)
 * r7: Item pointer (1, 2, 3, ..., 64)
 * r8: Item weight
 * r9: Item value
 * r10: gene copy, discarding bits as items are "counted"
 * r11: best fitness so far
 * r12: 0xffffffff
 *
 */

jmp ONE
jmp TWO
jmp THREE
jmp FOUR
jmp FIVE
jmp SIX
jmp SEVEN
jmp EIGHT

ONE:
addi r1, r0, 0x42
jmp START

TWO:
addi r1, r0, 0x44
jmp START

THREE:
addi r1, r0, 0x46
jmp START

FOUR:
addi r1, r0, 0x48
jmp START

FIVE:
addi r1, r0, 0x4a
jmp START

SIX:
addi r1, r0, 0x4c
jmp START

SEVEN:
addi r1, r0, 0x4e
jmp START

EIGHT:
addi r1, r0, 0x50
jmp START

START:
    /* Define settings */
    addi r1, r0, 1
    addi r2, r0, 0b00100
    addi r3, r0, 0b100
    addi r4, r0, 0b01000000

    /* Load settings */
    slli r1, r1, 16
    slli r2, r2, 11
    slli r3, r3, 8
    or r5, r5, r1
    or r5, r5, r2
    or r5, r5, r3
    or r5, r5, r4
    setg r5


ldi r2, 0x41
addi r11, r0, 0
addi r12, r0, -1
srli r12, r12, 32

MAIN_LOOP:
ldg r3
addi r10, r3, 0
addi r4, r0, 0
addi r5, r0, 0
addi r6, r0, 1
addi r7, r0, 1

INNER_LOOP:
and r0, r3, r6
if zero: jmp INNER_LOOP_END
xor r10, r10, r6
```

```
97  ld r8, r7, 0
98  and r9, r8, r12
99  srli r8, r8, 32
100 add r4, r4, r8
101 add r5, r5, r9
102
103 INNER_LOOP_END:
104 cmp r10, r0
105 /* If r10 is zero, we have put all the items in the sack */
106 if equal: jmp MAIN_LOOP_END
107 cmp r4, r2
108 if greater than: jmp MAIN_LOOP_END
109 slli r6, r6, 1
110 addi r7, r7, 2
111 jmp INNER_LOOP
112
113 MAIN_LOOP_END:
114 cmp r4, r2
115 /* If we had too much weight, set value (fitness) to 0 */
116 if greater than: addi r5, r0, 0
117 stg r5, r3
118 cmp r5, r11
119 if less than: jmp MAIN_LOOP
120 mv r11, r5
121 st r3, r1, 0
122 st r5, r1, 1
123 jmp MAIN_LOOP
```

GAS–programs/simple_memtest/memtest.gas

```
1  jmp r0, INIT_ONE
2  jmp r0, INIT_TWO
3  jmp r0, INIT_THREE
4  jmp r0, INIT_FOUR
5
6  INIT_ONE:
7  addi r1,r0,0x1
8  jmp r0, RUN
9
10 INIT_TWO:
11 addi r1,r0,0x5
12 jmp r0, RUN
13
14 INIT_THREE:
15 addi r1,r0,0x9
16 jmp r0, RUN
17
18 INIT_FOUR:
19 addi r1,r0,0xd
20
21 RUN:
22 ld r2,r1,0x0
23 addi r2,r2,0x1
24 st r2,r1,0x0
25
26 DONE:
27 jmp r0, DONE
```

GAS–programs/color–search/color–search.gas

```
1  /*
2   * Color Search
3   *
4   * A simple genetic algorithm searcher that searches for a given color.
5   *
6   * Register overview:
7   *
8   * r1: the red color target
9   * r2: the green color target
10  * r3: the blue color target
11  * r4:
12  * r5: the current individual
13  * r6: the current fitness
14  * r7:
15  * r8: the best individual
16  * r9: the best fitness
17  * r10:
18  * r11: the current red color
19  * r12: the current green color
20  * r13: the current blue color
21  * r14:
22  * r15: total distance
23  * r16: red color distance
24  * r17: green color distance
25  * r18: blue color distance
26  * r19:
27  * r20:
28  * r21: red color mask
29  * r22: green color mask
30  * r23: blue color mask
31  * r24:
32  * r25: -1
33  * r26:
```

```
 * r27:
 * r28:
 * r29:
 * r30:
 * r31:
 */

entry_points:
    /* adding a bunch of jumps, in case of many processors */
    jmp main
    jmp main
    jmp main
    jmp main
    jmp main
    jmp main
    jmp main
    jmp main
    jmp main
    jmp main
    jmp main
    jmp main

main:
    /* Define settings */
    addi r1, r0, 1
    addi r2, r0, 0b00100
    addi r3, r0, 0b100
    addi r4, r0, 0b01000000

    /* Load settings */
    slli r1, r1, 16
    slli r2, r2, 11
    slli r3, r3, 8
    or r5, r5, r1
    or r5, r5, r2
    or r5, r5, r3
    or r5, r5, r4
    setg r5

    /* load color masks */
    addi r21, r0, 0xFF
    slli r22, r21, 8
    slli r23, r22, 8

    /* load goal colors */
    addi r1, r0, 255
    addi r2, r0, 0
    addi r3, r0, 255

    /* load max int */
    subi r25, r0, 1
    srli r25, r25, 1

loop:
    /* load a new unrated individual */
    ldg r5

    /* extract color information from individual */
    and r11, r5, r21
    and r12, r5, r22
    srli r12, r12, 8
    and r13, r5, r23
    srli r13, r13, 16

    /* calculate distance from color */
    sub r16, r1, r11
    if negative: neg r16, r16
    sub r17, r2, r12
    if negative: neg r17, r17
    sub r18, r3, r13
    if negative: neg r18, r18

    /* sum the distances */
    add r15, r16, r17
    add r15, r15, r18

    /* calculate the fitness */
    sub r6, r25, r15

    /* store the gene with the distance sum as the fitness score */
    stg r6, r5

    /* compare to best */
    cmp r6, r9
    if less than: jmp loop

    /* store new */
    mv r8, r5
    mv r9, r6

    /* repeat! :D */
    jmp loop
```

APPENDIX

F

TEST BENCH DOCUMENTATION

F.1 Introduction

This document provides additional documentation and graphics from unit test benches, used for verifying the components in the FPGA. The graphics are selected screenshots taken in ISim.

F.2 Component Tests

F.2.1 Fitness Core

In order to test the fitness cores a set of small assembly programs were created. These programs aims to test various cases in the galapagos architecture. The tests are included here for reference.

RRR-RRI Instructions Simple program testing the use of RRI and RRR instructions

testing/gas–listings/rrr–rri.gas

```
1  /* basic RRI test */
2
3  addi r1, r0, 0xBA
4  slli r1, r1, 8
5  addi r1, r0, 0x12
6  slli r1, r1, 8
7  addi r1, r0, 0x12
8  slli r1, r1, 8
9  addi r1, r0, 0x1C
10 slli r1, r1, 8
11 addi r1, r0, 0xEC
12 slli r1, r1, 8
13 addi r1, r0, 0xC1
14
15 /* r1 should now contain 0xBA1212ICECC1 */
16
17 /* basic RRR test */
18
19 add r2, r1, r0
20
21 /* r2 should now contain 0xBA1212ICECC1 */
```

Store instruction Demonstrating the store instruction

testing/gas–listings/store.gas

```
1  /* store data test */
2
3  addi r1, r0, 1
4  st r1, r0, 0
5
6  /* value in memory at address zero should be 1 */
```

Load instruction Demonstrating the load instruction instruction

testing/gas–listings/load.gas

```
1  /* load test */
2
3  addi r3, r0, 1
4  st r3, r0, 0
5  ld r1, r0, 0
6
7  /* r1 shoudld contain 1 */
```

Conditional execute Demonstrating code with conditionals that evaluate to true.

testing/gas–listings/conditional–executed.gas

```
/* conditional taken test */

addi r1, r0, 1
cmp r1, r0
if not equal: addi r1, r1, 1
/* r1 shoudld be 2 */
```

Conditional non-execute Demonstrating code with conditionals that evaluate to false.

testing/gas–listings/conditional–not–executed.gas

```
/* conditional taken test */

addi r1, r0, 1
cmp r1, r0
if equal: addi r1, r1, 1
/* r1 shoudld be 1 */
```

Branch taken Demonstrating an conditional branch where the conditional is evaluated to true.

testing/gas–listings/branch–taken.gas

```
/* branch taken test */

addi r1, r0, 1
cmp r0, r1
if not equal: jmp end
addi r1, r1, 1
addi r1, r1, 1
addi r1, r1, 1
end:
    /* r1 should be 1 at this point */
    jmp end
```

Branch not-taken Demonstrating an conditional branch where the conditional is evaluated to false.

testing/gas–listings/branch–not–taken.gas

```
/* branch taken test */

addi r1, r0, 1
cmp r0, r1
if equal: jmp end
addi r1, r1, 1
addi r1, r1, 1
addi r1, r1, 1
end:
    /* r1 should be 4 at this point */
    jmp end
```

Store gene Demonstrate the use of the store gene instruction.

testing/gas–listings/store–gene.gas

```
/* store gene test */

addi r1, r0, 1
addi r2, r0, 2
stg r2, r1

/* an individual ('1') with fitness 2 should be sent to the rated pool */
```

Load gene Demonstrate the use of the load gene instruction.

testing/gas–listings/load–gene.gas

```
/* load gene test */

stg r1

/* r1 should contain an unrated individual from the unrated pool */
```

F.2.2 Genetic Pipeline

Selection Core

Crossover Core

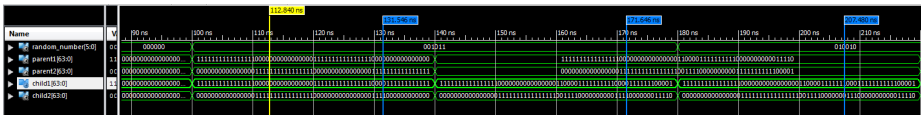


Figure F.1: Crossover Split Simulation Screenshot

Figure F.1 shows the simulation of Crossover Split function, where the blue markers are set just before the crossover begins. Changes in input parents and random_number causes changes in children.

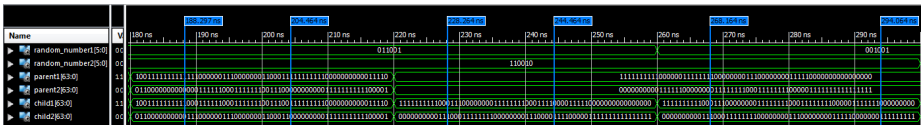


Figure F.2: Crossover Double-Split Simulation Screenshot

Figure F.2 shows the simulation of Crossover Double-Split function, where the blue markers are set alternating before and after crossover. Changes in input parents and any random_number causes changes in children.

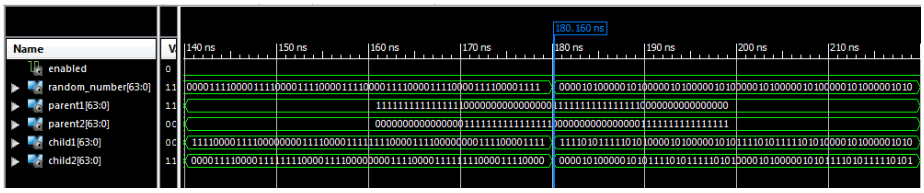


Figure F.3: Crossover XOR Simulation Screenshot

Figure F.3 shows the simulation of Crossover XOR function, where the blue marker is set at a change in the random_number, causing changes on the crossover

in the children. Changes in input parents and any random_number causes changes in children.

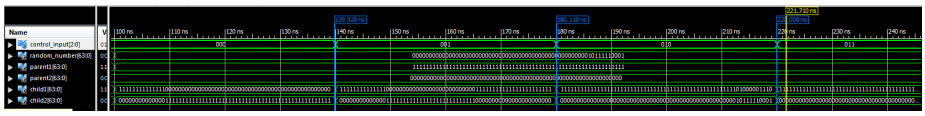


Figure F.4: Crossover Toplevel Simulation Screenshot

Figure F.4 shows the simulation of Crossover toplevel, where the blue markers are set at changes in the control_input, changing from split to double-split, then to XOR, and finally to no crossover at all.

Mutation Core

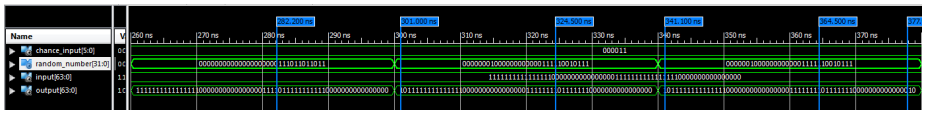


Figure F.5: Mutation Core Simulation Screenshot 1

Figure F.5 shows the simulation of the Mutation Core, where the blue markers are set just before the bits that are mutated in the output. Figure F.6 shows another part of the same simulation, with different main input.

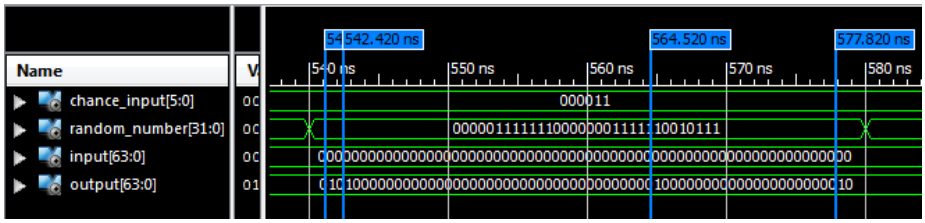


Figure F.6: Mutation Core Simulation Screenshot 2

APPENDIX

G

PCB COMPONENTS

This is the list of components used in the production of Barricelli.

With the exception of the FPGA and Microcontroller, all components were ordered from Farnell and are listed with their farnell product number. The FPGA and microcontroller were ordered from Digi-Key and are listed with their full identifiers instead.

Component	**Farnell product №**	**№required**
Oscillator	1842148	1
Jumpers	4218176	86
Headers	1580053	278
EFM32GG390F1024-BGA112	Ordered from digi-key	1
XC6SLX45-2CSG324I	Ordered from digi-key	1
Serial Port Connector	1653978	1
Serial Port Driver	1287435	1
Power Connector	224960	1
Memory Chip	2103743	3
LED, red	8554510	6
LED, green	5790852	16
micro USB Receptacle	2293751	1
SD Card Receptacle	2226409	1
Voltage Regulator	1685484	1
Voltage Regulator 3.3V	1469037	1
Switch (button)	3801287	8
Switch (toggle)	1524244	1
Transient Voltage Suppressor	1748616	1
ESD Suppressor (30V VCL)	1850152	1
ESD Suppressor (17V VCL)	1850151	1
Resistor 50k R	2057780	7
Resistor 0 R	1653183	1
Resistor 56 R	1738995	18
Resistor 105 R	2139353	1
Resistor 120 R	1470033	1
Resistor 330 R	2333547	3
Resistor 100k R	9240764	1
Capacitor 100 nF	1759297	59
Capacitor 1 uF	1759455	3
Capacitor 4.7 uF	1759444	1
Capacitor (Electrolyte) 100 nF	9697039	5
Capacitor (Electrolyte) 10 uF	2326109	4

BIBLIOGRAPHY

[1] AN0030 - FAT on SD Card. http://www.silabs.com/Support%20Documents/TechnicalDocs/AN0030.pdf.

[2] An0043 - efm32 debug and trace. http://www.silabs.com/Support%20Documents/TechnicalDocs/AN0043.pdf.

[3] AN0045 - USART/UART - Asynchronous mode. http://www.silabs.com/Support%20Documents/TechnicalDocs/AN0045.pdf.

[4] AN0046 - USB Hardware Design Guide. http://www.silabs.com/Support%20Documents/TechnicalDocs/AN0046.pdf.

[5] AN0065 - EFM32 as USB Device. http://www.silabs.com/Support%20Documents/TechnicalDocs/AN0065.pdf.

[6] AS7C38098A 512K X 16 BIT HIGH SPEED CMOS SRAM Datasheet. http://www.alliancememory.com/pdf/sram/fa/as7c38098a.pdf.

[7] Dieharder: A Random Number Test Suite.

[8] EFM32GG990 Datasheet F1024/F512. http://cdn.energymicro.com/dl/devices/pdf/d0046_efm32gg990_datasheet.pdf.

[9] FatFs - Generic FAT File System Module. http://elm-chan.org/fsw/ff/00index_e.html.

[10] Junit. http://junit.org/.

[11] Karma. http://karma-runner.github.io/0.10/index.html.

[12] Previous projects in TDT4295 Computer Design. http://www.idi.ntnu.no/emner/tdt4295/oldproj.

[13] David A.Patterson and John L. Hennessy. *Computer Organization and Design*. Elsvier, Revised 4th edition, 2012.

[14] David Noever. Steady-State vs. Generational Genetic Algorithms : A Comparison of Time Complexity and Convergence Properties. 1992.

[15] David B. Fogel. Nils Barricelli—Artificial Life, Coevolution, Self-Adaptation. http://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=01597062, 2006.

[16] Gunnar Tufte. TDT4295 Computer Design Project Assignment Text 2013. It's Learning.

[17] Juraj Hromkovič. *Algorithmics for Hard Problems*. Springer, 2nd edition, 2004.

[18] Nadia Nedjah and Luiza de Macedo Mourelle. Hardware Architecture for Genetic Algorithms. http://download.springer.com/static/pdf/217/chp%253A10.1007%252F11504894_76.pdf?auth66=1380662030_cebfeff2fc5857025c25324f3efb4818&ext=.pdf. .

[19] Norihiko Yoshida. VLSI Hardware Design for Genetic Algorithms and Its Parallel and Distributed Extensions. 1999.

[20] Paul Graham and Brent Nelson. Genetic Algorithms In Software and In Hardware – A Performance Analysis of Workstation and Custom Computing Machine Implementations. http://www.pvv.org/~gombos/GA%20FPGA%20vs%20Software.pdf.

[21] Raymond Sung and Andrew Sung and Patrick Chan and Jason Mah. Linear Feedback Shift Registers. http://www.ece.ualberta.ca/~elliott/ee552/studentAppNotes/1999f/Drivers_Ed/lfsr.html. .

[22] redisthefastestcolor@tumblr.com. Red is the fastest color. http://redisthefastestcolor.tumblr.com/post/44938966112.

[23] Xunying Zhang and Chen Shi and Fei Hui. FPGA-Based Genetic Algorithm Kernel Design. http://link.springer.com/content/pdf/10.1007%2F978-3-540-74626-3_40.pdf.