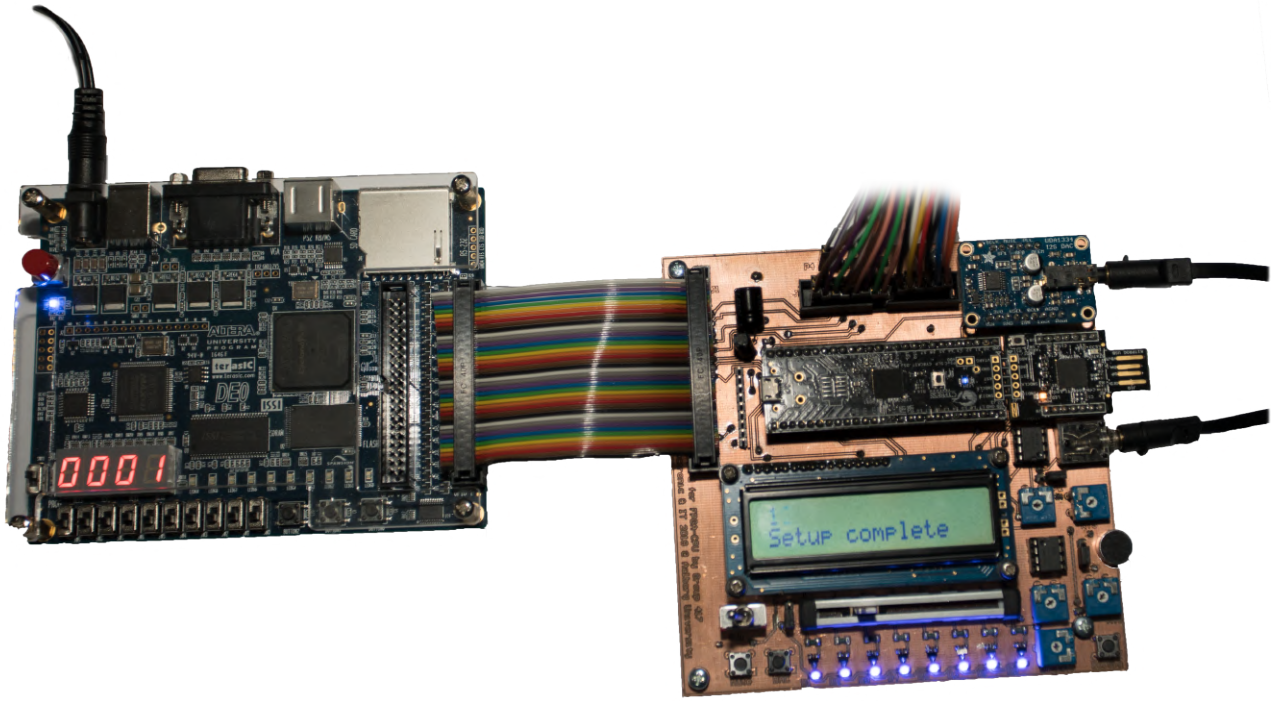
Softcore Microprocessor

- P4 -



Project Report

Group 417

Aalborg University
Department of Electronic Systems
Fredrik Bajers Vej 7B
DK-9220 Aalborg

Copyright © Aalborg University 2018

This report was written in LaTeX and has been shared online in-between the group members using ShareLatex licensed to all students at Aalborg University.

Department of Electronic Systems
Fredrik Bajers Vej 7
DK-9220 Aalborg Ø
http://es.aau.dk

AALBORG UNIVERSITY
STUDENT REPORT

Title:
Softcore Microprocessor

Theme:
Design of Digital Systems

Project Period:
Spring Semester 2018

Project Group:
Group 417

Participants:
Frederik Skyt Dæncker Rasmussen
Magnus Bøgh Borregaard Christensen
Max Væhrens
Mikkel Damgård Hardysøe
Jakob Krarup Thomsen
Peter Kjær Fisker

Supervisor:
Flemming Christensen

Report Page Numbers: 79

Appendix Page Numbers: 172

Date of Completion:
May 30, 2018

Abstract:

The purpose of this report is to document the design and construction of a CPU capable of processing digital audio, mainly by filtering. A custom RISC CPU architecture is designed and implemented in VHDL on an FPGA. Among others, the designed architecture implements stack functionality, non-sequential execution of code in the form of jumps, and hardware interrupts. Additionally, a hardware DSP accelerator is implemented in the form of a FIR filter.
To complement the synthesized CPU, an instruction set is crafted, and an accompanying assembler is created to facilitate the creation of software. To effectively convert analog audio signals to digital data, an ADC is implemented on a PSoC, and the sampled data is transmitted to the CPU by the means of an I^2S Bus. To allow the CPU to convert digital data back to the analog domain, an external DAC is used. Additionally, to mitigate the effect of noise on the system, a PCB for the peripheral systems is built.
Tests of the system showed, that while the CPU is able to process data, communicate with peripherals, and filter a live audio stream in a basic interrupt driven set up, additional code in the main loop halts the system. Two instructions were also identified to function incorrectly. Further effort is required to identify and resolve these issues, including resolving inconsistencies in the synthesis tool. Despite this, it is concluded that the system met its specifications in a very acceptable manner. Therefore, it is deemed that the project has been very successful, while still presenting several challenges to overcome.

The content of this report is freely available, but publication (with reference) may only be pursued after agreement is made with the authors.

Preface

This report is about a project which aims to create a CPU which specializes in audio processing. This project takes place on the fourth semester of the Electronics and IT bachelor at Aalborg University, during the Spring semester of 2018. The main theme of the third semester is *Design of Digital Systems*. The author of this report is a project group made up of students, and the students are all listed on the title page. During the project one supervisor have guided the group.

The first part of this report establishes the scope of the project by introducing an overview of the system, the subsystems and then ending with a problem statement. The next part covers the requirements of the product and the design and hardware choices made in the initial design phase. Thereafter, a more detailed system overview based on the requirements is presented. The third part contains all of the system design chapters where each of the subsystems are designed. The fourth part is the conclusion of the report. This part is initialized with a complete system test, which has the purpose of verifying that all of the subsystems function together. Then a discussion is made about the project and especially the developed system. Finally the fifth part contains the appendices, this includes short analysis of relevant topics, test protocols and journals, large scale figures and full code implementation.

First off, the group would like to thank Aalborg University and the Department of Electronics Systems for setting a good study environment for Electronics and IT students, and further allowing ambitious projects to come to life. We would also like to thank the following people:
Our supervisor: Flemming Christensen for supervision and feedback on the report.
Flemming Bøgh Christensen for thorough feedback on the report.

The following software have been used for this project:

- Quartus 13.1 Was used for FPGA development.
- ModelSim 10 used for simulations/tests.
- PSoC Creator 4.2 was used for the development on the PSoC.
- Microsoft Visual Studio Community 2015 version 14.0 with the C-compiler was used for the development of the Assembler
- GPP Open Source tool was used for preprocessing of VHDL files.
- Digilent WaveForms was used with the Analog Discovery 2 USB Oscilloscope was used to record wave forms.
- GitKraken was used for version control.

List of abbreviations

(ADC) Analog-to-Digital Converter
(ALU) Arithmetic Logic Unit
(CISC) Complex Instruction Set Computing
(CPU) Central Processor Unit
(CU) Control Unit
(DAC) Digital-to-Analog Converter
(DFT) Direct Fourier Transform
(DMA) Direct Memory Access
(DSP) Digital Signal Processor
(FFT) Fast Fourier transform
(FIR) Finite impulse response
(FIFO) First-In-First-Out
(FPGA) Field Programmable Gate Array
(HDL) Hardware Descriptive Language
(I/O) Input/Output
(I^2S) Inter-IC Sound
(IC) Integrated Circuit
(IIR) Infinite Impulse Response
(ISA) Instruction Set Architecture
(ISR) Interrupt Service Routine
(LCD) Liquid-Crystal Display
(MIPS) Microprocessor without Interlocked Pipeline Stages
(PC) Program Counter
(PROM) Programmable Read-Only Memory
(PSoC) Programmable System on Chip
(RAM) Random Access Memory
(RISC) Reduced Instruction Set Computer
(SPI) Serial Peripheral Interface
(TD) Transaction Descriptor
(UI) User Interface
(VGA) Video Graphics Array
(VHDL) VHSIC (Very High Speed Integrated Circuit) Hardware Description Language

Contents

Part I

Project Scope

Introduction

This report aims to document the design and development of a CPU implemented on a Field Programmable Gate Array (FPGA). The focus of the CPU implementation is directed at signal processing and especially audio signals. Specifically, it is desired to implement audio filters that will allow, e.g. equalization of an audio signal.

The focus will primarily be the digital aspects of audio processing. This is because it is the only way to process audio signals on a CPU, however it is also because the main topic of this semester is *Design of Digital Systems.*

In conjunction with the development of the CPU, other peripheral subsystems are to be implemented as well. Most importantly are, an Analog-to-Digital Converter (ADC) and Digital-to-Analog Converter (DAC) which are required to allow the analog audio signals to be processed digitally and then converted back to analog audio. A series of internal peripherals are also needed to enable communication between the CPU and external peripherals.

Additionally, the CPU will have user interfaces such as buttons and a display. A more detailed overview of the system and all its subsystems will be given in chapter 1.

Before designing and developing this system, the scope of the project and a preliminary study is required to uncover the various aspects and challenges which lie in the process of creating the specified system. The preliminary study in its entirety can be found in appendix A to B. The following chapter introduces the general overview of the system which is to be developed during the project.

Chapter 1

System Introduction

The end-goal of this project is to develop a system to process audio signals. To accomplish this, the system can be divided into two overall parts; The Central Processing Unit (CPU) and peripheral subsystems. The CPU must digitally process the analog input collected by some of the peripheral subsystems and afterwards transfer the signal to other peripheral subsystems that convert the signals back to analog to be played through a speaker. This is also illustrated in figure 1.1.

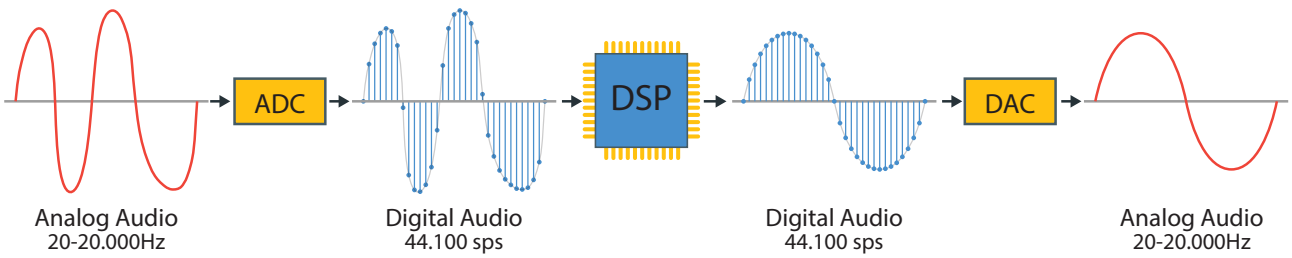


Figure 1.1: Illustration of the proposed system. An analog signal gets digitized and processed through a DSP-module and returned as a different digitized signal.

The following sections will go into detail describing the CPU and the necessary peripheral subsystems. To give an overview of the system and all its subsystems, a list has been compiled which can be seen below.

1. CPU.
    - Internal control system
    - Arithmetic Logic unit
    - Signal processing module
    - Memory
    - Bus for ADC & DAC
    - Hardware interrupts
    - Input driver for buttons on the FPGA board
    - Seven segment display peripheral
2. Peripherals
    - ADC
    - DAC
    - Input buttons
    - Output display

1.1 Central Processing Unit

The CPU can carry out instructions to process data. The CPU consists of several blocks which interconnect with each other, the registers and Arithmetic Logic Unit (ALU) being two of these. This section will describe the central blocks of the CPU which is going to be designed, starting with the ALU. These subsystems of the CPU will be described at a higher level of abstraction to assure the overview of the whole CPU is not lost.

1.1.1 Arithmetic Logic Unit

The arithmetic logic unit also known as the ALU is the core of the CPU. The ALU is two units. An arithmetic unit and a logic unit [1]. As a single unit, it is expressed as in the figure 1.2.

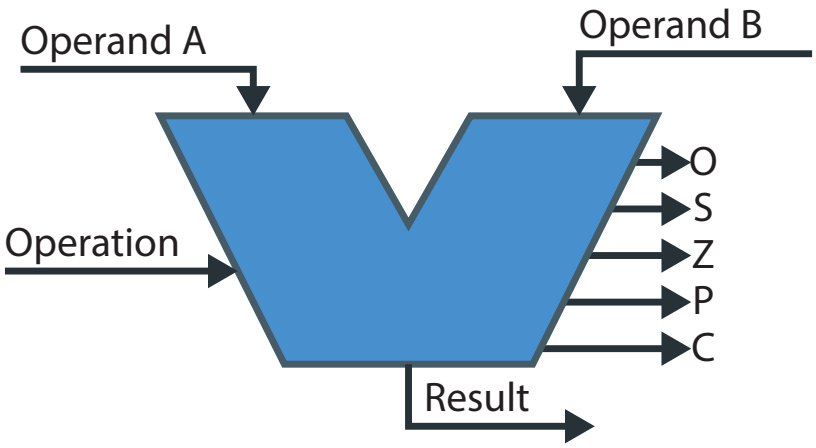


Figure 1.2: Illustration of an ALU and its inputs and outputs

An ALU is an abstraction of a lot of logic gates to a single operating unit. Figure 1.2 is an example of an ALU. It takes two input operands, A and B, on which it performs calculations based on Boolean algebra. Additionally, to these two operands it also requires an operation code (opcode) which defines what operation to perform inside the ALU. Different operations could be to add the two operands, subtract the operands, or let the operands pass through. As for outputs, the ALU has one output and several flags. These flags are only represented with a single bit which means that the flags can only be true or false.

1.1.2 Data Storage

To store values for later use or to allow more complex operations, the CPU must be able to store data. Usually a CPU has internal registers for temporary/intermediate storage and external memory for more permanent storage [2]. Both kinds of memory described in this section is volatile memory and thus only saves the data if power is available.

Registers

A register is an internal storage location, which is used to save values that are being processed [2, p. 56]. For example, value A is to be added to value B, so A and B are loaded into separate registers. Afterwards, the values are added in the ALU, saved in a register. Then if needed the values can be stored in the memory.

There are other registers in a CPU than the ones used to store data for different operations. For example, the program counter register contains the memory address of the current operation being performed. What types and how many registers there are in a CPU is up to the manufacturer and is based on what the CPU is designed for. Internal registers are usually the fastest type of memory, due to their location close to the other components of the CPU [2, p. 56].

Memory

Memory is used to store data for a longer time compared to registers. Since a Harvard architecture is used there are two kinds of memory, instruction memory and data memory. Instruction memory is used for storing all instructions while data memory is used to store all the data that the programmer wants to store [2]. It is possible for the programmer to store and load data inside the data memory with a store and load instruction.

1.1.3 Instructions

A CPU uses instructions to determine what operation to perform next. These instructions are stored as binary machine code, which translates to a specific operation. The instruction contains all the information needed to do a certain operation. The instruction contains an operation code and the registers and values that should be used to do the operation. The opcode describes what the CPU should do, and the rest of the instruction tells the CPU what it should use to complete operation.

The instruction set of a CPU is usually based on what the CPU is used for. If the CPU just has to add numbers and be able to save them, a small instruction set is needed. For more general use or more advanced operations, more instructions are needed. Secondly the instruction set is also based on the hardware limitations of the CPU. For example, if the CPU has a dedicated multiplication block, an instruction to multiply two numbers is needed. The instructions are interpreted by the control unit (CU) and then carried out by the required hardware.

1.1.4 Control Unit

The CU is the part of the CPU that decodes machine code into operations as well as keep track of the location of the code currently being executed. The CU control all the signal paths in the CPU [2, p. 55-56]. While the CU does not perform any processing, it wires the rest of the system in such a way, that instructions can be carried out.

1.1.5 Hardware Interrupts

A hardware interrupt is a hard-wired connection from an input to the CPU, which allows for interruptions in the currently running code. The purpose of such an interrupt is to service external modules,

such as an ADC when it has data that needs reading. An interrupt works by saving the location in the currently running code, before jumping to an interrupt service routine (ISR). The CPU has to be instructed where the code for an ISR is located in memory, same as labels for jumps but written in specific interrupt hardware. An ISR is simply a piece of code that dictates what happens when an interrupt occurs. For example, for an ADC, the interrupt could be that a word of data is ready to be read. The ISR for this this interrupt would then be to read and save the data, from the ADC, to memory.

1.2 Peripherals

This section will explain in detail the different peripherals that is desired to work with the CPU.

1.2.1 ADC and DAC

For the CPU to process audio signals, the signal must be converted from analog to digital first. This is done using an analog to digital converter (ADC). An ADC samples the input signal and converts it to digital format, which the CPU can manipulate.

To output the digital audio signal, it must be converted back into analog form. This is done using a digital to analog converter (DAC). For more information on sampling of signals, see appendix A.

1.2.2 User Interface

To be able to communicate with the CPU and for the CPU communicate with the user, a UI must be implemented. The following UI could be implemented.

- An array of seven-segment-displays to display numeric values.
- Buttons to be able to give simple inputs and adjust simple settings.
- Screen to have a live feed of what is happening with the signals.
- Keyboard to interact with the CPU in a more efficient way.

At least one of the inputs and a feedback method from the list above should be implemented for an UI.

1.3 Digital Filters

For the CPU to execute signal processing it is desired to have signal filters. Since this project is focused on the digital aspects of audio processing, it has been chosen to have digital filters. The advantage of digital filters rather than analog filters is that it can be easily configured without having to change the components. More information about digital filters can be found in appendix B.

1.4 Direct Fourier Transform

A Direct Fourier Transform (DFT) block could be implemented to convert from time domain to frequency domain. [3, p.11] The DFT could be an interesting way to visualize the signals being processed in the system. This would be most interesting if a screen were implemented to visualize the result in real time.

1.5 Problem Statement

From the analysis, it can be concluded that digital processing is a viable route to go, for audio applications. As such, it is decided to design a CPU. The CPU should be capable to perform audio processing. To better understand both how a CPU works, and digital filters etc. The next part of this report will attempt to answer the following questions:

- How is a CPU designed, and how can one be implemented on an FPGA?
- What functions should this CPU have to be best suited to process audio?
- How can audio processing be done on an FPGA in conjunction with the CPU?

Part II

Requirements & System Design

Chapter 2

Product Specifications

In this chapter, the different specifications of the product will be established. These requirements are established based on careful consideration of the product. The different specifications for the product will be divided into three groups: Functional requirement, Design choices, and Hardware choices.

2.1 Functional Requirements

To specify the expectations of the system, the table 2.1 about the functional requirements will make an outline of what is desired from the product. The requirements are listed in order of prioritization.

Table 2.1: The functional requirements

Priority	Specification of the requirement	The justification
1.	A functional CPU that can execute instructions.	The CPU is the core of the project and must be implemented for the whole system to work.
2.	RAM.	RAM is needed to store the program and the output of the program.
3.	Capable of performing arithmetic and logical operations.	The logical and arithmetic operations are needed for the CPU to perform operations.
4.	Instruction set architecture (ISA).	An ISA is needed to define the operations that the CPU can perform.
5.	Simple UI	UI for simple user interaction with the CPU
6.	Hardware interrupts.	Hardware interrupts are desired because it enables the CPU to react on demand which is useful for features such as UI.
7.	Line-level analog I/O from the product sampled at 44,1 kHz.	Sound signals are presented as analog signals and thus analog I/O is required. It is desired that the system is compatible with other systems. Therefore, the standard for digital audio is followed [4].
8.	Assembler for the designed ISA.	An assembler is desired since it significantly improves the programming experience and efficiency for on the CPU and prevent errors.
9.	Digital filter.	Filtering is an essential part of signal processing and thus a digital filter is desired for this.
10.	Advanced UI.	More UI would increase the user experience of the system.
11.	DFT block.	The DFT block would let it visualize the signals and their frequencies live.
12.	I/O to external storage.	It is desired to be able to store and load programs and important data on non-volatile memory.

2.2 Design Choices

With the functional requirements in place, this section will focus on the various design choices that have been made. The design choices and a short explanation on why the choices are made can be found in table 2.2.

Table 2.2: Design choices

Nr.	The design choice	The justification
1.	I^2S Bus to handle the connection between the digital audio devices.	The I^2S standard is used since it is designed specifically for audio, and compatible devices are widely available.
2.	Fixed point is chosen over floating point.	This is done because it is less complex. And the dynamic range of the floating point is not needed.
3.	RISC is chosen over CISC.	RISC is chosen because of its simplicity. The programs that will be written will not be complex enough to warrant CISC. It is more desirable to have a less complex CPU than less complex assembly code.
4.	Assembler written in C-code.	The assembler will make the programming of the CPU significantly easier. And it will be written in C because of previous experience with C-coding, as well as being taught C in the current semester
5.	A seven-segment-display.	It is ideal for displaying simple outputs.
6.	VGA connection and driver.	To show a range of graphical information to the user, such as an FFT. VGA is chosen since the FPGA board used already has a connector for this.
7.	ADC with a Programmable Read-Only Memory or henceforth known as a PSoC.	Because of the lessons and experience with the PSoC. The PSoC also has an I^2S module. Also, the PSoC was available at the time.
8.	I^2S supported DAC.	It is needed to convert the audio back to an analog signal, to play it on a speaker. The DAC must support I^2S to be used with the CPU.
9.	32-bit instructions.	These are used to allow up to three registers per instruction, or up to two registers and a 16-bit integer value. This significantly simplifies the CU part of the CPU, at the cost of slightly longer instructions.
10.	SPI protocol.	Used for non-audio serial communication. The protocol is simple, compatible with most I/O systems such as SD card readers, as well as being well documented.
11.	No pipelining.	While a pipelined design can significantly increase the maximum clock frequency of a CPU, it also increases the design complexity. Therefore, it is decided to implement a non-pipelined design.
12.	Internal peripheral control is done via memory mapped I/O.	This means the peripheral drivers are treated the same way as data memory is. This allows for easy addition of peripherals in the future.
13.	Harvard architecture.	It is implemented to help speed up the CPU. The added complexity is easily outweighed by the benefits of essentially being able to access two memory locations per clock cycle. The Harvard architecture is ideal when dealing with filtering, since a lot of data must be loaded into the filters, which can be done parallel to loading instructions.
14.	Representing data using big endian	This CPU is designed for learning. For this reason, it is desirable to use the endianness most intuitive to humans, in this case being big endian. Since there are no compilers or higher-level languages available for the CPU, then the code that will be written for the CPU will be bare-bones and close to the metal.

A lot of these design choices are made with the deadline of this project in mind. Therefore, not every choice is the so called "best" choice for the given product but instead the best for the given situation. The choices are based on earlier experience, knowledge from the sources that was used in the analysis section, and how much time there is to finish the project.

2.3 Hardware Choices

To implement the system according to the requirements, a hardware platform will have to be decided upon. There are many viable options and not just one right choice. But the choices were made based on the requirements, what was available and on what would provide the best development experience.

2.3.1 The FPGA

For the FPGA the DE0 board from Altera [5] was the preferred platform. This was based on the development tool chain available and the development-board that was available was preferred to the alternative at the time which was a Papilio DUO [6]. Most important is the higher number of logic elements, multipliers and onboard memory. The more logic elements the more logic can be implemented, but more important is the filters which are the most limited based on logic elements can be improved with higher number of logic elements and multipliers. And finally, the Altera FPGA use the Quartus IDE for the development which was preferred to the ISE Design Suite used for the Papilio DUO.

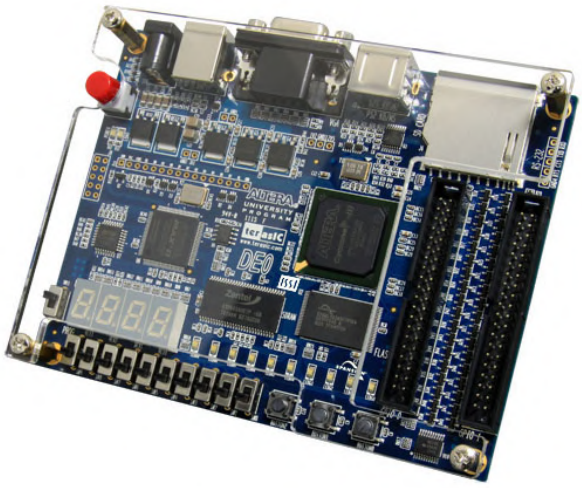


Figure 2.1: The DE0 board from Altera. [7]

2.3.2 DAC

The DAC used for this project is an Adafruit I^2S Stereo Decoder - UDA1334A Breakout. This was chosen based on the simple interface and that it is a complete package, ready to use.

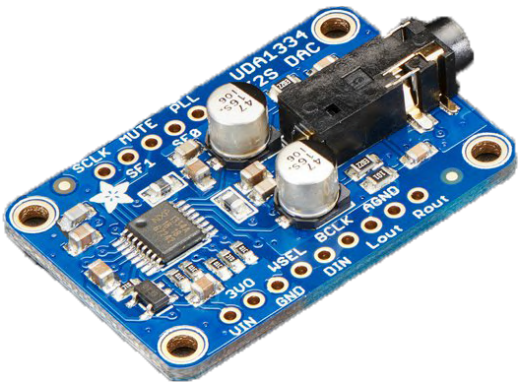


Figure 2.2: Adafruit I^{2}S Stereo Decoder - UDA1334A Breakout[8]

2.3.3 ADC and PSoC

For the ADC it was decided to use the PSoC development board as it has an ADC module and an I^{2}S interface module. This was decided because it was easily available, and because it could provide some experience with developing on a PSoC and using the PSoC Creator IDE.

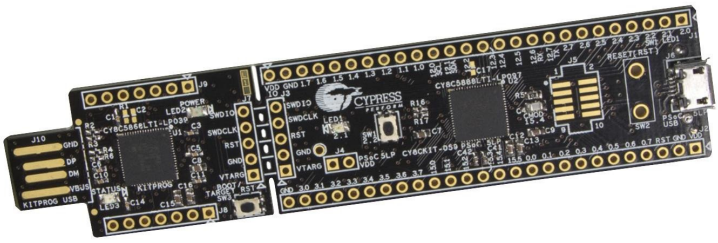


Figure 2.3: CY8CKIT-059 Development Board. [9]

2.3.4 Printed Circuit Board

For all the peripheral systems it is beneficial to place these components on a single board, therefor a Printed Circuit Board (PCB) will be designed. Because all the subsystems placed on the PCB contain mixed analog- and digital signals, the PCB should be designed to reduce noise as much as possible. To facilitate testing of the individual systems and the complete systems, pinouts should be placed for both analog and digital input and output of the whole system.

With specific hardware modules chosen and overall system requirements determined, the design process of the system is ready to begin. In the following chapters the overall system is going to described and afterwards split up into different subsystems to be designed individually.

Chapter 3

System Design Overview

This chapter will cover the design of the overall system and the individual parts that will be designed and their purposes. First an overview diagram of the system will be presented. Afterwards the purpose and functionality of the various parts will be individually addressed.

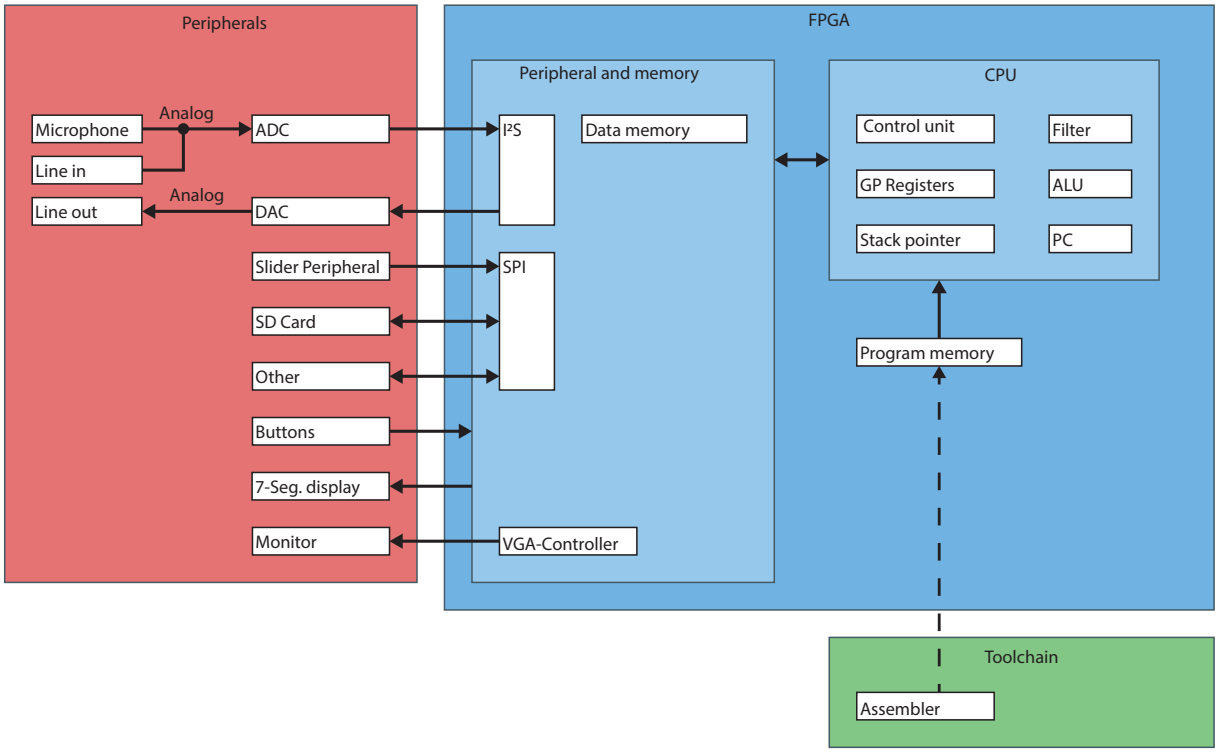


Figure 3.1: Full system overview

Figure 3.1 presents an overview of the full system, which is designed based on the requirements in part II. The CPU is the main focus of the project. But for the CPU to be useful it needs I/O. All the blocks on figure 3.1 will be described in the following sections.

3.1 CPU

The CPU is the core of the project and does practically all the data processing. The processor reads the program memory and based on instructions, it processes the data from the data memory, and

interacts with the peripherals by the peripheral interfaces. As illustrated on figure 3.1, it consists of six primary parts. The design of the CPU is described in chapter 4

3.1.1 Control Unit

The CU controls all the internal connections in the CPU. Based on the incoming instruction from the program memory the CU is programmed to turn on or off the necessary connections inside the CPU. The CU does not perform any processing but rather routes signals using combinational logic. It maps a type of instruction to a configuration of the internal connections. The other components in the CPU will the perform the instruction when connected as specified in the CU. Another part of the control of the CPU is implementation of interrupts. When an interrupt is triggered the internal connections are also changed so that the CPU perform the specified interrupt service routine. An overview of the CU is found in table 4.2.

3.1.2 General-purpose Registers

The general-purpose registers are registers that are available to the programmer. By referring to a register in an instruction the programmer can read or write to the registers. These registers have no specific purpose but to store temporary values that are directly available to the CPU in the current instruction.

3.1.3 Program Counter

The program counter (PC) is a register that holds the address of the next instruction for the CPU in the instruction memory. Unlike the general-purpose registers, the PC is not directly accessible by the programmer. The PC is incremented every time an instruction is read by the CPU. The PC is also used for jumps and interrupts. Both of which is implemented by simply loading the address where to jump to.

3.1.4 Stack Pointer

The stack pointer is used for keeping track of interrupted processes to be able to continue the process after the interrupt. If the interrupt service routine uses the same registers or change the flags, the interrupted process can behave different than intended. Thus, it is often desired to save current context. The context consists of all the registers, flag registers and most importantly the PC. It is not always necessary to save all the context depending on the architecture of the CPU, which can also save time as it takes time to save each bit of context. The stack pointer thus keeps track of the last address onto which the context was saved or "pushed" into memory. That makes it possible load or "pop" the context from memory to return to this place at any time.

3.1.5 Arithmetic Logic Unit

The arithmetic logic unit (ALU) is where all arithmetic and logic processing occur in the CPU. The ALU can perform arithmetic and bitwise operations on the two inputs. The inputs, depending on the routing from the CU, will be from the general-purpose registers or an immediate value from

an instruction. Then based on the opcode from the CU the ALU will perform one of the specified operations and set its output and status flags. The status flags are set based on the result of the operation and can be used for logic such as conditional jumps. Even though, all the ALU can do is to perform the specified operations, it is also used indirectly by the CU in most of the instructions that does not directly do arithmetic or bitwise operations. The design of the ALU is described in chapter 5

3.1.6 Filters

The dedicated filters are implemented to significantly reduce the clock cycles required for filtering audio. It is the main application of the CPU, and while it is possible to make pure software filters with basic ALU operations, it takes a lot of clock cycles and is not very efficient. Thus, to be able to process audio in real time to a satisfactory degree, it is necessary to implement dedicated filters which also free the CPU for other processes. The filters will be implemented as a peripheral that is controlled by instructions. The design of the filters is described in chapter 6

3.2 Memory and Peripheral Interfaces

The memory and peripherals let the CPU have access to data it can process and is an I/O interface. The memory makes it possible to process larger amount of data. And the peripherals let the CPU communicate with other devices to receive input and interact with the world. The CPU needs I/O as it would be pointless without it. The CPU must process audio; thus, it needs an audio interface. The digital audio interface is using the I^2S protocol. For general purpose serial communication, the SPI protocol will be used. And for some of the onboard peripherals custom drivers will be used. A list of the distinct parts is compiled bellow.

- **Program memory:** The program memory will contain the instructions and will be directly accessed by the CPU. The program memory is connected to the PC and it will output the instruction on the PC's address. It is single port memory and will not be changed during operation, thus it will be read only access.
- **Data memory:** The data memory is for general purpose storage during operation. The memory can be read from and written to. All the memory can be accessed by the programmer.
- **Inter-IC Sound:** Inter-IC Sound (I^2S) is a protocol for transferring audio as serial data. It will have an input and output module that will convert between the serial data of I^2S signal and the 16-bit parallel bus of the CPU. The input module will be the interface between the ADC and the CPU. The output module will be an interface between the CPU and DAC.
- **Serial Peripheral Interface bus:** The Serial Peripheral Interface bus (SPI) will be used for communication between the CPU and other peripherals. SPI is a serial protocol made to communicate with multiple peripherals using only one bus.
- **Buttons and Seven-Segment-Display:** The onboard buttons and seven-segment-displays will be accessed by custom drivers. These will be for the most basic inputs and outputs. The seven-segment-display will behave as memory and depending on control signals it will display the data as hexadecimal or decimal. The buttons will have interrupts to the CPU and a memory address with flags indicating if each button is pressed.

- **VGA Controller:** The VGA peripheral will be able to receive commands to draw graphics to a frame buffer which will be displayed on a monitor following the standard for the VGA interface.

The design of the internal peripherals will be described in chapter 7

3.3 Peripherals

To let the CPU interact with the user and the surroundings, peripheral devices are connected via the aforementioned internal interfaces, which are connected to the I/O pins of the chip. The peripherals have many different uses, but in general they are for I/O. But most important for this system is the analog interfaces that allows for analog audio I/O. A list of the peripherals for the system is compiled bellow.

- **Line in and out:** Line-level signals will be used for analog input and output. Line level is compatible with most audio peripherals and the connector will be and AUX-connector.
- **Microphone:** The microphone will be available as an alternative input and will be implemented on the PCB.
- **ADC:** The analog to digital converter is used as an analog input for the CPU. It converts the analog input to an I^2S signal that the CPU can use with the help of the I^2S input module.
- **DAC:** The digital to analog converter is used to convert the digital I^2S signal generated by the I^2S output module of the CPU to an analog audio signal.
- **Slider Peripheral:** The slider is implemented on the PSoC and will be used as a UI that allows for gradual adjustments, it will use the SPI interface.
- **SD Card Reader:** The SD card reader will add the option to store data to non-volatile memory and also to load data from the SD card that can be edited on other devices.
- **Onboard Seven-Segment-Display and buttons:** The onboard buttons and display will be able to communicate directly with the CPU via custom made driver modules.

3.4 Toolchain

Additional external tools are used in conjunction with the CPU, to make use and design easier. These are called the toolchain. Currently the toolchain only consists of an assembler.

3.4.1 Assembler

To program the CPU, a series of instructions must be given via the program memory. These instructions must be at the lowest level possibly - binary machine code. The CPU is designed to use 32-bit instructions. To write this by hand would not only take too much time, but also be very prone to errors, due to the how difficult binary can be to read.

Using an assembler, the binary machine code can be generated from assembly language instructions. This language consists of written text, which makes it easy to read and write. The assembly language

used will draw inspiration from known languages but will mostly be specific to the designed CPU. The design of the assembler is described in chapter 8

Having gained an overview of the system and its subsystems, the design and development of the individual parts can begin. The next chapters will dive deeper into the workings of the distinct parts of the system in an effort to explain the design process and subsequent testing of said parts.

Part III

Design

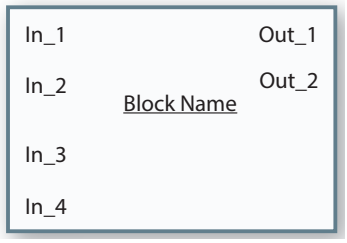
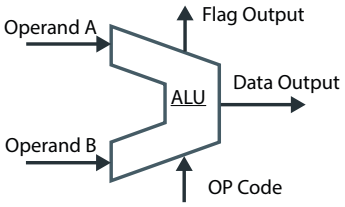
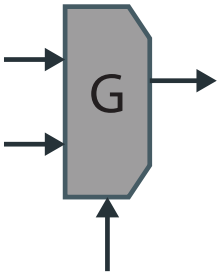
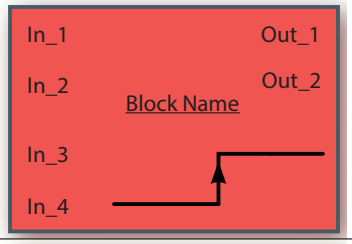
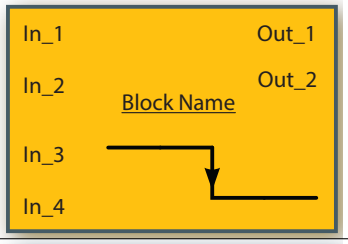
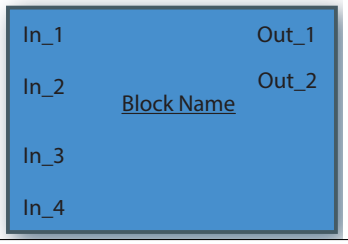
Chapter 4

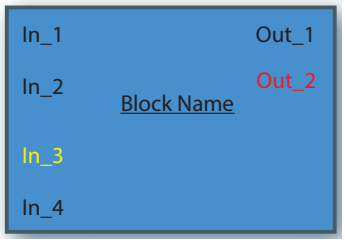
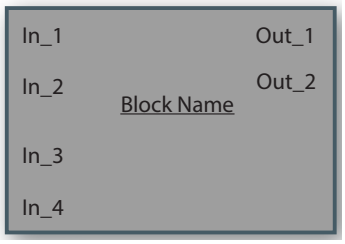
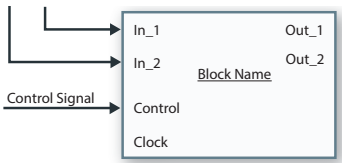
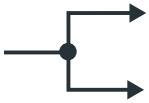
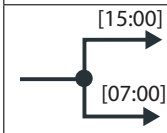
Design of CPU Architecture

This chapter describes the CPU's internal architecture and details the control circuitry connecting all the individual sub-components together. As mentioned in section 2.2, the designed architecture adheres to the principles of the Harvard architecture, and must be able to access both instruction and data memory during a single clock cycle. Additionally, the architecture does not have a pipeline, and thus an abundance of asynchronous logic is used to perform data processing. This design choice simplifies the architecture in certain areas, but will however limit the maximum clock frequency, due to the long propagation delays compared to a pipelined approach.

Due to the overall complexity of the architecture, the chapter covers the architecture from an abstracted view, and does not focus on the details of the VHDL implementation. The VHDL code of the entire implementation can be found in appendix P.5 to P.10 and appendix P.17 to P.18. As a guide the architecture presentation is described in stages where each stage expands upon the previous with the inclusion of additional control logic. Thus, the chapter starts with describing the core of the architecture, only able to perform elemental data processing, and finishes with the entire designed architecture including filtering capabilities, jumps, stack control, and interrupt service routines (ISR). Furthermore, to enhance readability this chapter heavily relies on block diagrams, and to facilitate these diagrams, a description of blocks, shapes, conventions, and colors of these diagrams follow.

Table 4.1: Diagram key

Symbol	Description
	Module: Left side is for inputs, right is for outputs, and the underlined word(s) is the name of the module.
	ALU: Accepts two operands and an opcode as inputs. The outputs are the computed data and condition flags.
	Routing logic: Accepts inputs to be manipulated, condition input in the side, and outputs to the front (right on the shown figure).
	Red Blocks: Synchronous hardware, dependent on rising edge.
	Yellow Blocks: Synchronous hardware, dependent on falling edge.
	Blue Blocks: Asynchronous hardware, only dependent on input.

Symbol	Description
	Port's color: Decides the synchronicity of that port. Supersedes block color.
	Grey Modules: External devices that are not part of the FPGA.
	Routing: Clock signal routing is implied. Control signal is not routed but only labeled. All other signals are fully routed.
	Colored squares: Latches sensitive to the clock according to its color.
	Constants: Represent a constant output value.
	Expression: Operation to be performed on an input.
	Dots: Routing branches.
	Text above data line: Indicate which signals are routed.

4.1 Core Architecture

In the following section the block diagram of the CPU architecture in its stripped-down form will be shown.

This basic implementation of the architecture was partially inspired by the works of Patterson et al. [10]. It can execute all arithmetic, logic, and filtering instructions. It is also capable of memory/peripheral access with the LOAD and STORE instruction. It is, however, limited to purely sequential execution (i.e. no jumps or interrupts), nor does it support stack operations. The diagram contains the following modules:

- **Register File** containing 30 general purpose registers. The Register Source 1 and 2 inputs accepts five bits (32 combinations) indexing the register data to be outputted to Out_1 and Out_2 respectively. Likewise Register Destination also accepts a 5-bit input, however this port indexes the register to have data written to it. The port receiving the data to be written to the RD register is labeled DATA_I. The PC-IN port is used to route the data from the external PC register into the register file, to ensure that the rest of the CPU can access its content. Lastly, the Write-ENable port controls whether data can be written to the indexed RD register. Write is enabled by setting the port high. Notice that the register file is an asynchronous block. The exceptions to this are the three ports associated with writing, RD, W-EN, and DATA_I, which are colored red, meaning that new data can only be written to a register on a rising clock edge.
- **Peripheral and Memory Controller (PMC)** manages access to the DRAM and the various peripherals. Reading and writing data to the controller is determined by the input state of the Read-ENable input ports. These two input ports will never be enabled simultaneously. For writing data to the controller, the CPU must provide an address and data to the controllers ADDR and DATA_IN port respectively. For reading, the CPU must only provide and address, and the controller will then ensure to output the data at the desired address to the DATA_OUT port. For a more in-depth description of the inner workings of the memory and peripheral controller, refer to chapter 7
- **The ALU** is present as an asynchronous block. For more info see chapter 5.
- **A Digital Filter** is present as a synchronous block. Note that this is the only synchronous data processing module. This is undesirable but was necessary due limitations in the filters VHDL implementation. For more info see chapter 6.
- **Instruction Memory** contains the 32-bit instructions to be executed. The instruction to be outputted is indexed by the ADDR input port. The Read enable port is permanently set high and write enable port is set low.
- **The Program Counter (PC)** register indexes the next instruction to be executed.
- **The Control Unit** is effectively a large multiplexer, mapping a range of output signals to each individual opcode. Each output control signal affects the operation of the CPU for the duration of the execution cycle. See table 4.2 for a complete description of each controls signals purpose.
- **CLOCK** represents the physical clock on the DE0 Board.
- **The PLL** is an internal block. Using the external physical clock, it is used to derives a new clock frequency more suitable for the CPU's timing requirements. The second PLL connected to the PMC is used to generate the clock for the internal I^2S IN clock.
- **7SEG DISPLAY** is the array of seven segment displays present on the DE0 Board.

- **Button** represents the three external buttons located on the FPGA board.
- **I²S OUT** represents the external I²S DAC used to create analog signals. See section 7.1 for more information.
- **I²S IN** is the external ADC sending audio to the CPU through I²S. See section 7.1 for more information.
- **Conditional Routing blocks** route signals through the architecture. In the VHDL implementation, they are primarily represented with "IF" or "CASE" statements.

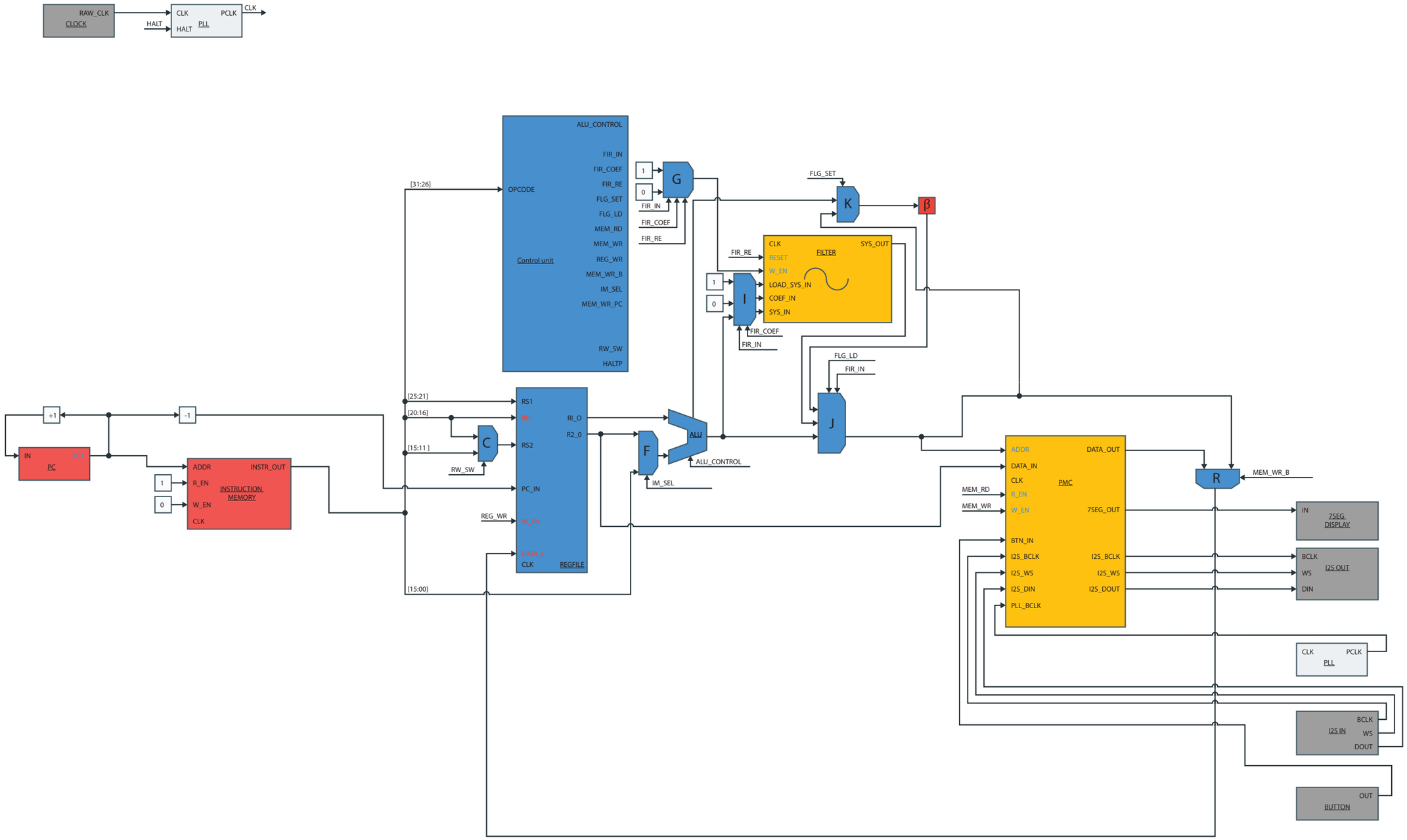


Figure 4.1: Core Architecture

Table 4.2: Control Unit Output Description. Includes example for the "ADDI" instruction

Size	Name	Description	ADDI Example
6	ALU_CONTROL	Code to be sent to the ALU, informing it of the arithmetic/-logical operation it should perform.	000010
3	JUMP_CONTROL	Controls if jumping is enabled/disabled, as well as what kind of conditions the CPU should check for.	000
1	FIR_IN	Filter data input. If enabled, loads data into the filter, and the filter outputs data to the main data line.	0
1	FIR_COEF	Filter coefficient input. If enabled, loads a coefficient into the filter.	0
1	FIR_RE	Filter reset. If enabled, resets the filter.	0
1	FLG_SE	Set Flags. If enabled, sets control flags to the first five LSB's of the 32-bit instruction.	0
1	FLG_LD	Load Flags. If enabled, send flag values to the main data line.	0
1	MEM_RD	Memory Read. If enabled, allows memory/peripheral data to be read and send to CPU.	0
1	MEM_WR	Memory Write. If enabled, allows the CPU to write the contents of the main data line to memory/peripherals.	0
1	REG_WR	Register Write. If enabled, allows CPU to write data to the registers.	1
1	MEM_WR_B	Memory Writeback. If enabled, allows memory data to be written to the registers (NOTE: MEMRD must also be enabled).	0
1	IM_SEL	Immediate Select. If enabled, sends instruction immediate to ALU Operand B, instead of the second register file output	1.
1	PUSH	Push. If enabled, pushes data from the main data line to the stack and increments the stack pointer.	0
1	POP	Pop. If enabled, pops data from the stack and decrements the stack pointer.	0
1	RW_SW	Read Register Write Register Switch. Allows the use of RS1, RS2 and an immediate in an instruction.	0
1	MEM_WR_PC	Memory Write to PC. If enabled, write memory/peripheral data to the program counter. Used to pop data back to the PC.	0
1	HALTP	Halt Processor. If enabled, stops the processor clock. Only used in the HALT instruction.	0

To convey how data is passed through the system, figure 4.2 shows a timing diagram, illustrating how signals propagate through the circuit during and after rising and falling clock edges. Note that this diagram does not describe all aspects of the system (for example, reading and writing to flags is not present), nor are the drawn propagation delays accurate, as no timing measurements have been performed. Its purpose is only to further the understanding of the architecture.

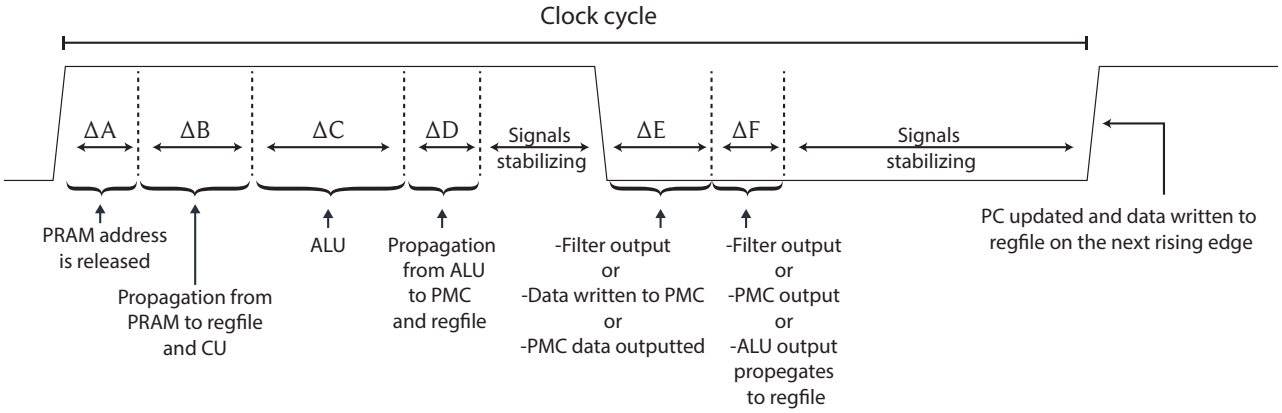


Figure 4.2: Simplified Architecture timing diagram

At the start of a rising edge, the instruction memory loads an instruction based on the program counter (ΔA). This instruction then propagates to the CU and register file, where control and data signals are set (ΔB). This data flows into the ALU, which, based on the control signals, performs a calculation using the provided data (ΔC). The ALU's output proceed to propagate to the filter, PMC, and regfile (ΔD). For the remainder of the time where the clock is set high, no more data is processed, and the signals will spend the duration stabilizing on the various input/output ports. When the clock then returns to logic low, several actions may take place, depending on the previously set control signals. For example, if the "MEM_WR" control signal has been set, data will be written to the PMC. Other actions include: reading data from the PMC or the filter (ΔE). If the PMC or filter generated new data during Δ E, then data will propagate to the register file. For the remainder of the logic low clock cycle, data will stabilize at the input/output port. Then finally, when a new clock cycle begins, data will be written to the register file (that is if the "REG_WR" is set), and the entire process starts again.

Now that the architectures information flow has been defined, each individual part of the architecture will be described in detail.

Starting from the left of Figure 4.1, the PC is shown. The PC stores the address of the instruction to be executed in the following cycle. As seen on the figure, the PC index the location of the next instruction for the program memory to fetch. The PC output is also connected to the register file, to enable the rest of the system to access its value. This connection decrements the PCs value, since it was deemed desirable to retrieve the address of the currently executing instruction, as opposed to the address of the following instruction. Finally, the PC increments its value on every clock cycle.

Next is the instruction memory which outputs 32-bit instructions for the system to interpret and execute. Based on the architectures designed data paths, these instructions are designed to have one of two formats, as shown in table 4.3 and 4.4:

Table 4.3: Immediate Format

32 Bit			
[31:26] (6 bits)	[25:21] (5 bits)	[20:16] (5 bits)	[15:0] (16 bits)
Opcode	Source Register 1	Destination Register	Immediate

Table 4.4: Dual Source Format

32 Bit				
[31:26] (6 bits)	[25:21] (5 bits)	[20:16] (5 bits)	[15:11] (5 bits)	[10:0] (11 bits)
Opcode	Source Register 1	Destination Register	Source Register 2	EMPTY

The six opcode bits represents the fundamental instruction to be executed. They are routed into the CU which then in turn maps them to a predetermined set of control signal outputs. The remaining 26 bits merely acts as parameters for the instruction, and do not change the nature of an instruction execution. If an executing instruction is in the immediate format, it can at maximum accommodate a source register ("RS1"), a destination register ("RD"), and an immediate as input parameters. On the other hand, when using the dual source format, an instruction can accept up to two source registers ("RS1", "RS2"), and a destination register ("RD"). It shall be noted that it is not required for all three input parameters to be present in an instruction. For example, there are instructions which only accepts an immediate as an input, such as FIRCOI (save an immediate as a filter coefficient).

The only exception to the conditions above is the STORE instruction. STORE accepts a piece of data to be written to memory, as well as an address indexed by the contents of a register combined with an immediate value (see Appendix F for more info). thus, to operate properly, it requires two source registers as well as an immediate. Additionally, it does not write any data back to the registers, and it is not needed to index a destination register. However, these input requirements do not match with any of the two described instruction formats. Routing block C is used to overcome this shortcoming. The CU maps the STORE opcode to the "RW_SW" control signal. When this signal is set high, routing block C routes instruction bits [20:16], which under any other circumstances indexes the "RD" register, to the "RS2" input port. This is required since the bits otherwise used to enter "RS2" overlaps with bits used to represent an immediate. This remapping of bits [20:16] to the "RS2" port ensures that STORE can accept all the required input parameters.

After a short propagation delay, the Register file will output the contents of the indexed registers to R1_O and R2_O. Here R1_O is routed directly into the ALU's operand A port. This is not the case with R2_O. This port, in conjunction with the instruction bits representing an immediate, is routed into routing block F. Depending on the state of the "IM_SEL" control signal, this block either routes R2_O or the immediate into the ALU's operand B port. Based on the "ALU_CONTROL" signals, the ALU will perform a specific computation, using either one or both of operands a and b as input parameters. The status flags derived from this computation are routed to routing block K, which in turn routes them into latch β, where they will be latched on the next rising edge. The result of the computation is routed into routing block I and J.

If the "FIR_COEF" control signal is set high, routing block I will route the ALU's output into the "COEF_IN" port and it will set the "LOAD_SYS_IN" port low. This will save one coefficient to the filter. If on the other hand the "FIR_IN" control signal is set high, the "LOAD_SYS_IN" port will be set high and the ALU's output will be routed into "SYS_IN". This operation will result in a data sample being stored in the filter. If neither of the control signals are set high, the routing block will set LOAD_SYS_IN low, and no new data will be routed to the filter.

To complement routing block I, routing block G controls whether the new data can be written to the filter. If either the "FIR_IN", "FIR_COEF", or the "FIR_RE" control signals are enabled, the routing block will set the "W_EN" port high, and therefore it is possible to write the data, routed by block I, to the filter. If none of the listed controls signals are enabled, the port will be set low.

The ALU's computational output, the filter's output, and the latched status flags (i.e. the flags result-

ing from the previous cycles ALU computation) are routed to routing block J. This block essentially allows the system to select which one of the three inputs is to be routed into the remaining sections of the CPU. The status flags will be passed on if the "FLG_LD" control signal is enabled, the filter data is passed if "FIR_IN" is enabled, and the ALU's output is passed along in all other cases.

The output from routing block J is routed to the "ADDR" port of the PMC, as well as routing block K and R. Routing block K control whether the ALU's control flags, or routing block J's output is routed to the flag latch β. By default, the ALU flags are routed to the latch β, however the "SETFLAG" opcode sets the "FLG_SET" control signal, thereby allowing a programmer to directly set flag values.

Routing block J's output is also routed into the PMC's "ADDR" port. This determines memory/peripheral address to be indexed in the PMC. If the "MEM_WR" control signal is set, output from the register files "R2_O" port will be written to the indexed address on the next falling clock edge. If on the other hand the "MEM_RD" control signal is set, the contents of the indexed address will, on a falling clock edge, be send to the PMC's "DATA_OUT" port. This port is then in turn is connected to routing block R.

In a similar fashion to block J, block R determines which of the two input signals, block J and the PMC's output data, to be routed into the register files "DATA_I" port. In most cases block J's output will be routed back into the register file. However, in certain cases, such as with the "LOAD" instruction, the "MEM_WRITE_B" control signal will be set, allowing the PMC's data output to we written to a register.

If the register files "W_EN" port is set high by the "REG_WR" control signal, then the data preset on the "DATA_I" port will be written to the register indexed on the "RD" port, on the following rising clock edge.

4.2 Implementing Stack and Jumps

To expand upon the functionality of the previous described design, this next iteration, shown in figure 4.3, implements branching and stack functionality. Note that the blocks from the core architecture, previously shown in figure 4.1, are made slightly transparent, to make it easier to identify the additions to the architecture.

These changes implement a variety of new routing blocks, as well as stack pointer controller. This module contains an internal stack pointer (SP) register pointing to the top of the stack. When a POP or PUSH instruction is executed, the SP will be used to index the PMC address of interest, and the SP will be decremented (if executing "POP") or incremented (if executing "PUSH". The module has three control signal input, one data input, and one data output. The "ADDR-IN" port is connected to the same data path connecting the register files "DATA-I" path and, given that the "W-EN" port is also set high, allows the CPU to overwrite the value of the internal stack pointer register. To enable the "W-EN" port, the bits entering the register files "RD" port (bits [20:16]) must have an equivalent unsigned integer value of 30. Routing block E handles this condition. The block also ensures that the "W-EN" port will not be able to be set high if the "RW_SW" control signal is enabled. This safety precaution is required to ensure that the stack pointer is not accidentally overwritten in during a STORE instruction. The addition of routing block E allows programmers to write data to the stack pointer, just as they would any of the other 30 general purpose registers. The "POP" and "PUSH" input ports are enabled/disabled by the control lines of the same name. The state of these ports dictates whether the stack pointer increments or decrements. If the "POP" input port is set high, the "ADDR-OUT" port will be set to the SP's current value, and the SP will then subsequently

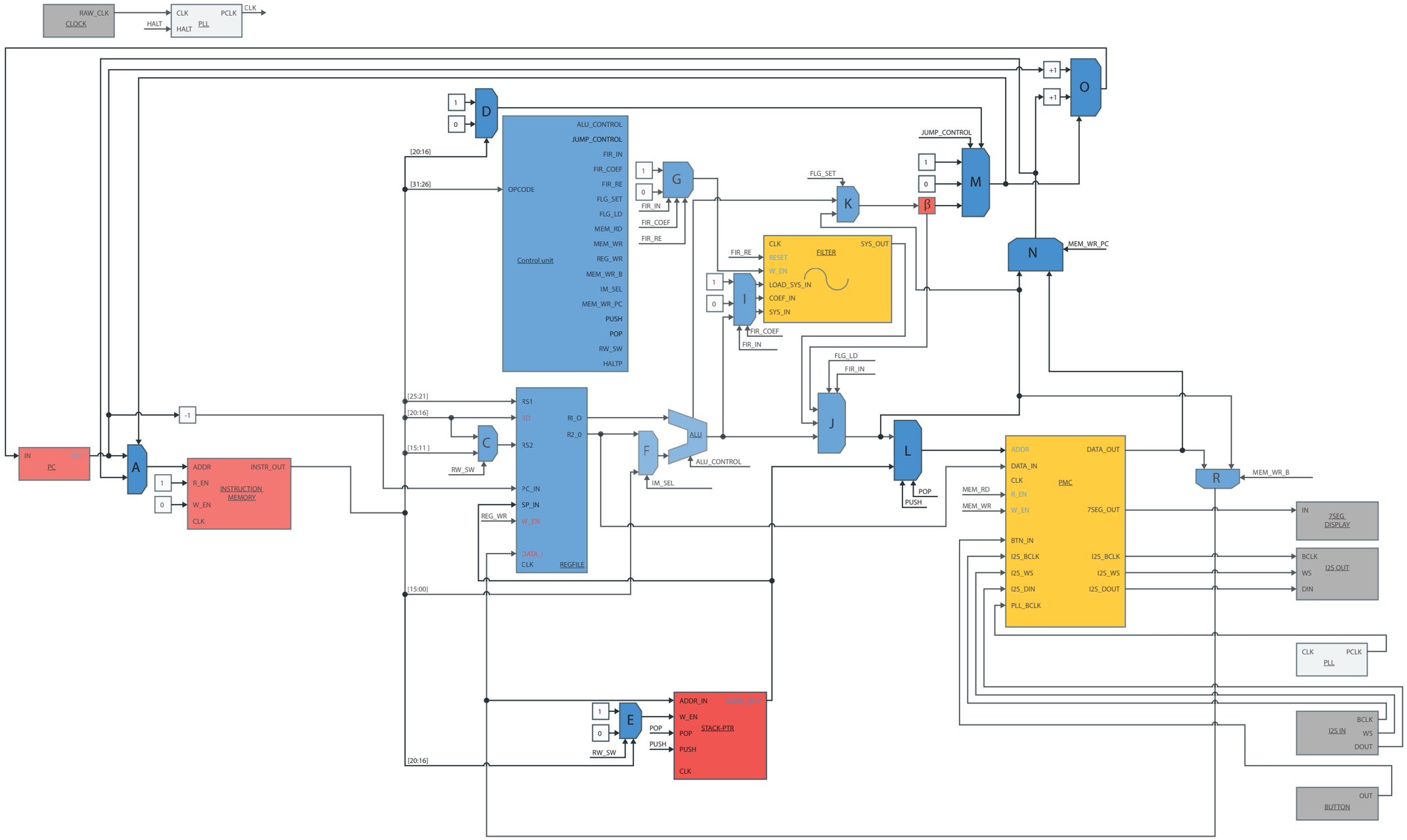


Figure 4.3: Architecture with Jump and Stack functionality

be decremented on the following rising clock. If, however, the "PUSH" input port is set high, the "ADDR-OUT" port will be set to the SP's current value plus 1, and the SP value will permanently be incremented on the next rising clock edge. Outputs are handled this way to ensure that when popping the stack, the top of the stack gets to be outputted before the SP's value changes, and when pushing data, the data is stored on the new value of the stack top. If neither of the "POP" and "PUSH" signals are set high, the "ADDR-OUT" output port is set to the current value of the stack.

The "ADDR-OUT" output port is connected to the register file to allow programmers to index the SP, so its value can appear on either of the register files output ports. More critically, together with the output of routing block J, the "ADDR-OUT" port is also connected to routing block L. This block decides whether it is the SP or the J block that gets to address the PMC. The SP is only routed to the PMC's "ADDR" port when either the "POP" or "PUSH" control signals are set high.

To support non-sequential instruction execution, unconditional and conditional jumping logic has also been implemented. This is achieved using routing block N, M, O and A. As explained previously, by default the CPU increments the PC every clock cycle. To enable the use of jumps the CPU must be capable of switching between either incrementing the PC or setting the PC to some alternative address. To accomplish this, routing block M uses the "JUMP_CONTROL" signals, the control signal generated by block D and the state of the latched flags to enable or disable jumping. If block M enables jumping block A and O will route the new instruction address, provided by block N, to the PC and instruction memory.

The alternative PC address provided by block N is derived from data in the core architecture. The routing block accepts data from the PMC's "DATA_OUT" port as well as block J's output. If the "Mem_WR_PC" control signal is set high, the PMC's data will be used as the alternative address, otherwise block J's output will be used. To determine if a jump to the alternative address is to take place, block M combines the "JUMP_CONTROL" signals with the latched flag values, and routing block D's output. Routing block D allows the programmer to directly set the value of the PC, mirroring the functionality of block E for the SP register. Like block E, it uses the "RD" index bits to determine whether to set its output as high or low. If the "RD" bits correspond to 31, i.e. the index value of the PC register, the output is set high, otherwise it will be low. The "JUMP_CONTROL" signals are concatenated with block D's output, and this bit combination is then interpreted as a single unsigned integer number, where each possible value corresponds to a distinct branch condition. Table 4.5 maps these values to their corresponding jump conditions:

Table 4.5: Block D and "JUMP_CONTROL" signals concatenated implications. The binary representation column represents a 4-bit number, with the "BLOCK D" sub column representing the MSB

Binary Representation		Integer Value	Meaning
Block D	JUMP_CONTROL		
0	000	0	Do not jump
0	001	1	Unconditional jump
0	010	2	Jump if equal
0	011	3	Jump if less than
0	100	4	Jump if not equal
0	101	5	Jump if parity
0	110	6	N/A (Do not jump)
0	111	7	N/A (Do not jump)
1	000	8	Unconditional jump
1	001-111	9-15	N/A (Do not jump)

Block M then uses this mapping in conjunction with the latched flags to determine if the jump should (outputs high) or should not (outputs low) be taken. Figure 4.4 shows the VHDL code used to determine block M's output:

```
WITH to_integer(unsigned(blockD & JUMP_CONTROL)) SELECT blockM <=
   '0'                   WHEN 0,       --do not jump
   '1'                   WHEN 1,       --unconditional jump
   zero_flag_latch       WHEN 2,       --jump if equal
   carry_flag_latch      WHEN 3,       --jump if less than
   NOT zero_flag_latch   WHEN 4,       --jump if not equal
   parity_flag_latch     WHEN 5,       --jump if parity
   '1'                   WHEN 8,       --jump if "RD" indexes 31
   '0'                   WHEN OTHERS; --do not jump
```

Figure 4.4: VHDL pseudo code illustrating the case structure determining the output of block M.

If block M's output is low, then a jump will not be taken, and thus block O will allow the PC to be incremented, and block A will let the PC be routed to the instruction memory's "ADDR" port, thereby continuing sequential execution. If on the other hand block M's output is high, then a jump will be taken. This will result in the alternative address, outputted by block N, to be routed through block A, and entering the instruction memory's "ADDR" port. Meanwhile, the PC will be populated with the alternative address plus 1. This ensures that the next instruction to be loaded will be the address specified by block M, and that the PC is ready to continue sequential execution by indexing the correct address on the following clock cycle.

4.3 Interrupt Implementation

Finally, to complete the narrative of the designed architecture, the interrupt functionality is described. The associated figure, figure 4.5, also illustrates the complete CPU architecture. As in figure 4.3, blocks present in previous illustrations of the architecture, are made slightly transparent. A non-transparent version of the architecture can be found in appendix XX.

When an interrupt occurs, the CPU will allow the current executing instruction to complete execution. On the following clock cycle, it will then block the next instruction from executing, and instead inject at PUSH PC instruction, to push the PC to the stack. At the same time the CPU will enable an unconditional jump to the ISR's program address, provided by the PMC. The ISR will then begin execution on the next clock cycle. If interrupt nesting has been disabled, then the CPU will not be able to act on a new interrupt while executing the ISR. To re-enable interrupts, and return to the main programs execution, the ISR must call a POP PC instruction.

To integrate interrupts into the CPU architecture, routing block B, P, and Q have been added. The PMC gets three more output ports and one input port. Block N and M have also been expanded, with the addition of a data and a control signal.

When the PMC detects an interrupt has occurred, it will output the interrupt state, the address of the ISR, and whether nesting is enabled, to the "INT_OUT", "INT_ADDR", and the "INT_NEST_EN" output ports, on the falling edge immediately following the interrupt. If the "INT_NEST_EN" port is high, interrupt nesting is enabled, and routing block P will ensure that the PMC's "INT_EN" input port will remain high, thereby ensuring that an interrupt can still occur. If instead "INT_NEST_EN" is low, then nesting is disabled, and block P will set "INT_EN" low to block future interrupts from calling ISR's. Additionally, block P will also set its connected latch γ high, to store the fact that interrupts are currently disabled from occurring.

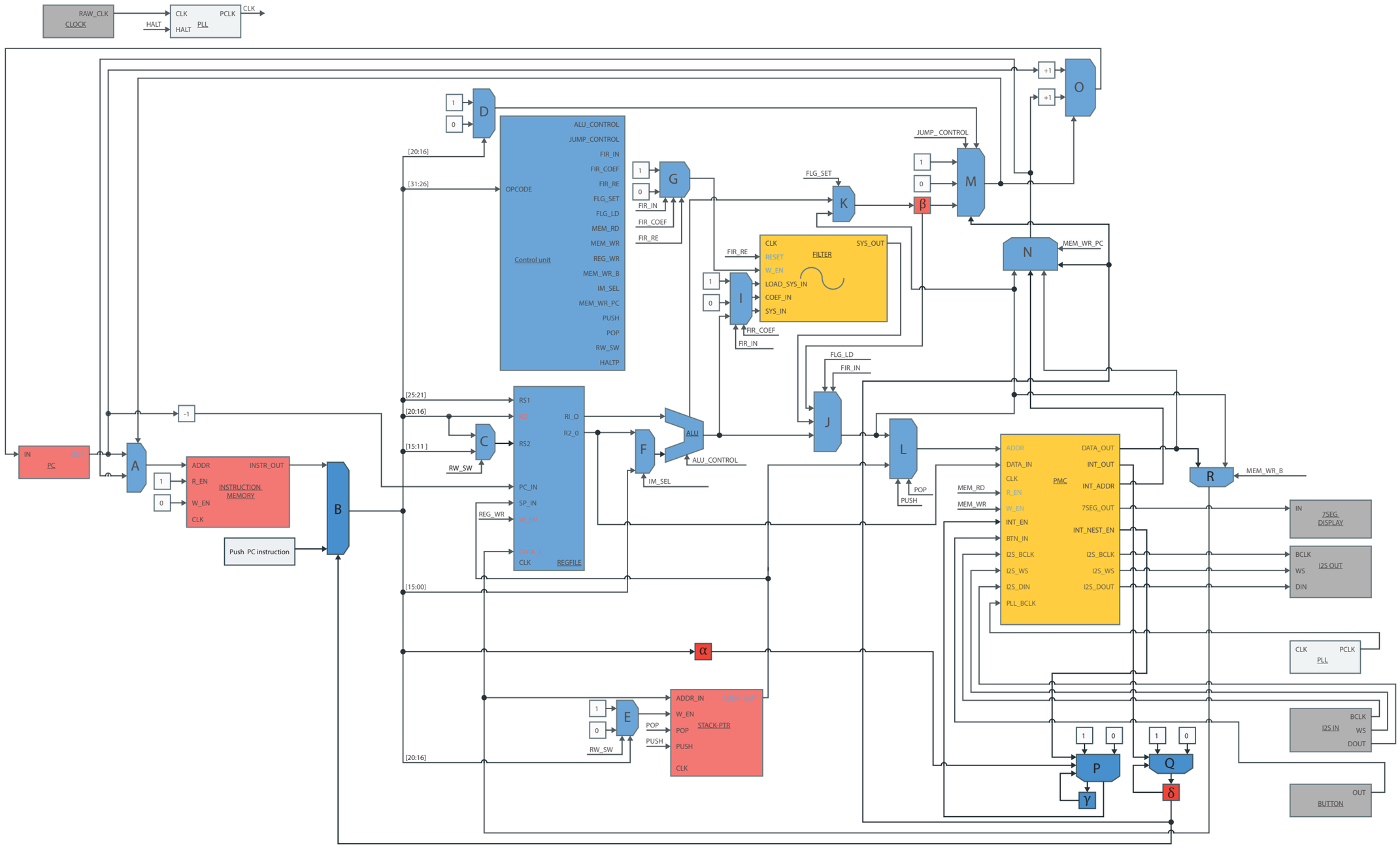


Figure 4.5: Complete Architecture

To allow the current executing instruction to finish execution, the interrupt logic remains idle until the following rising clock edge. When this occurs, the "INT_OUT" signal is latched to the latch δ connected to block Q. This signal is then routed to different areas of the architecture. Following the pattern of starting from the left, the latched signal is routed into routing block B. This blocks the instruction memory's output, and instead routes a predefined "PUSH PC" instruction to the register file and CU. This instruction pushes the PC to the stack and will allow a program to POP the PC from the stack when it has completed executing its ISR, allowing the CPU to resume execution from where it left off. During the execution of the "PUSH PC" instruction, the latch β, connected to routing block M, causes the block to enable an unconditional jump. This jump will set the PC to the ISR's address, as block N has been expanded to route the "INT_ADDR" port of the PMC to block O and A, when the interrupt latch α is set high. On the following cycle, the CPU will be executing the ISR, and block Q's latch δ will set itself low again, readying itself for another interrupt to occur. If nesting is disabled, block P will wait for the previously occurring instruction (latched in the central latch α) to be a "POP PC" instruction. This event corresponds to exiting the ISR, and block P will subsequently re-enable interrupts.

Note that block P checks for the previously executed instruction as a safety precaution. If it checked the currently executing instruction, and a new interrupt occurred at the same time as "POP PC" occurred, then the interrupt would push the instruction address of the last instruction of the ISR instead of the previously popped PC.

This concludes the description of the CPU's architecture. While it would be appropriate to perform a test of the architecture itself, it is difficult (if not impossible) to perform a comprehensive and all inclusive test, as there practically exists countless permutations of instructions. Additionally, to function as intended, the architecture must be able to communicate with external peripherals. Due to these circumstances it is decided that the functionality of the architecture will be tested and evaluated in conjunction with the rest of the system, during the accept test described in chapter 11. In the next chapters, starting with the ALU, key blocks from the designed architecture will be dissected.

Chapter 5

Design of ALU

This chapter will describe the design of the ALU module, mentioned in section 3.1.5. The ALU module is implemented on the FPGA in VHDL and enables the CPU to compute different arithmetic and logical operations.

5.1 Operations in the ALU

The ALU module consist of 18 different operations, each with a unique opcode.

A complete list of the operations can be found in table 5.1.

Table 5.1: Table with available operations in the ALU

Operation Name	Description	Opcode
ADD	Adds the two input operands	0x0001
SUB	Subtracts the two operands	0x0002
MUL	Multiplies the two operands,	0x0003
AND	ANDs the two operands	0x0004
OR	ORs the two operands	0x0005
XOR	XORs the two operands	0x0006
NEA	NEGATES operand A	0x0007
NEB	NEGATES operand B	0x0008
NOA	NOT's operand A	0x0009
NOB	NOT's operand B	0x000A
LSL	Logic Shift Left	0x000B
LSR	Logic Shift Right	0x000C
ASR	Arithmetic Shift right	0x000D
PSA	Passes operand A	0x000E
PSB	Passes operand B	0x000F
ICA	Increments operand A	0x0010
ICB	Increments operand B	0x0011
NOP	Does nothing, does not change flags	0x0012

In addition to the operations, the ALU can set up to five different flags. These flags are *Overflow Flag*, *Signed Flag*, *Zero Flag*, *Parity Flag* and *Carry Flag*. For a more detailed description of the different operations and flags, consult appendix D.

5.2 ALU Design

The ALU module has three inputs and six outputs, as seen in figure 5.1. The inputs consist of two operands and a signal that determines which operation to compute. The outputs consist of a result and the five flags mentioned earlier.

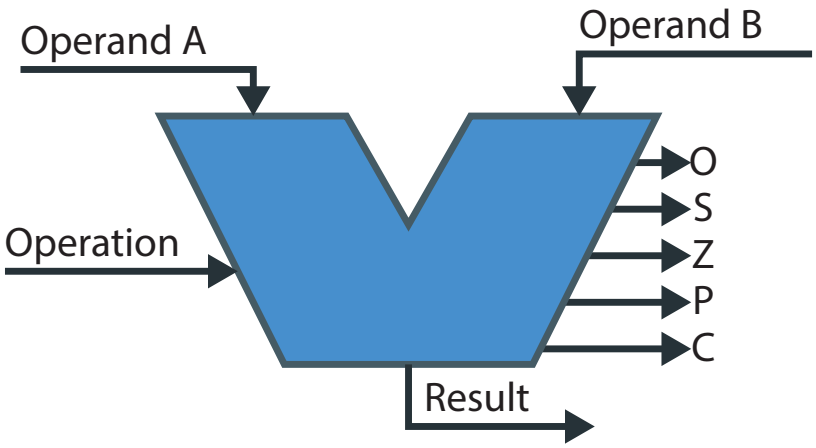


Figure 5.1: Illustration of an ALU and its inputs and outputs

The ALU module is designed in VHDL using a case structure that chooses what action to perform based on the signal on the operation input. The module uses ripple logic, and as such does not depend on a clock to function, instead the result stabilizes at the output when the input signals have propagated through the module.

To illustrate the case structure, some pseudo code is shown in figure 5.2

Most of the operations and flags are straightforward to implement in the VHDL language, for example all the bitwise logical operations and the bit shift operations are already implemented in the language. Due to this, only the operations where distinct considerations were necessary will be described in this section. The VHDL implementation of the ALU module can be found in appendix P.6

Addition, Subtraction and Increment

The first non-trivial operations to design are the addition, subtraction and increment operations. Here it is necessary to typecast the operands as unsigned to ensure that the operations results in a carry when applicable.

Multiplication

Binary multiplication can be somewhat difficult to implement in an effective manner, especially if the internal multipliers on the FPGA are to be used. Therefore, it was decided to use IP cores available in the Quartus software tool, to implement the multiplier. [11] The multiplier module inside the ALU module has two inputs and one output. The inputs are 16 bits each, and the output is 32 bits, which means that the result must be truncated before reaching the output of the FPGA. As the ALU uses

```
1        Arithmetic : PROCESS (Operands) IS
2           BEGIN
3              CASE operation IS
4                 WHEN ADD =>
5                       ... Code that adds ...
6                 WHEN SUB =>
7                    ... Code that subtracts ...
8                 WHEN MUL =>
9                     ... Code that multiplies ...
10                ...
11                More operations
12                ...
13                WHEN PAS => --
14                   ... Code that passes operand A ...
15             WHEN OTHERS =>
16          END CASE;
17       END PROCESS;
```

Figure 5.2: VHDL pseudo code illustrating the case structure design of the ALU.

ripple logic, the operands are always routed to the multiplier independent of the chosen operation, but the output of the multiplier is only routed to the output of the ALU when the multiplication operation is chosen.

Negation

Negation is the operation that changes the sign of a number, either from positive to negative or vice versa. When using two's complement, this is done by flipping all the bits and the adding one. An example of this is illustrated in figure 5.3. The negate operation results in overflow when trying to negate the largest negative number, due to how numbers are represented in two's compliment.

$$10_{10} \xrightarrow{\text{to binary}} 00001010_2 \xrightarrow{\text{flip bits}} 11110101_2 \xrightarrow{\text{add 1}} 11110110_2 \xrightarrow{\text{to decimal}} -10_{10}$$

Figure 5.3: Negation using two's complement binary numbers.

Arithmetic Right Shift

As mentioned earlier, the shift operations are already implemented in the VHDL language, but the arithmetic right shift is a bit different from the logical shifts. When shifting arithmetic to the right the sign-bit is used to fill from the left, instead of zeroes. Thus, the operand has to be typecast as signed for the VHDL synthesizer to perform the shift correctly.

Overflow Flag

The overflow flag is set when a signed number is too big or too small to be contained within the available data type. For addition overflow occurs when adding two numbers with same sign results in a number with a different sign. For subtraction overflow occurs when subtracting two numbers with different signs results in a number with the same sign as the subtrahend. [12] These conditions can be shown with truth tables and underlying logic operations can be solved for using a Karnaugh map. To solve the Karnaugh maps, a CAS tool was used. Truth tables for addition and subtraction is illustrated in table 5.2 and 5.3

Table 5.2: Truth table for overflow flag during addition

Sign of operand A	Sign of operand B	Sign of result	Overflow
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	0

Table 5.3: Truth table for overflow flag during subtraction

Sign of operand A	Sign of operand B	Sign of result	Overflow
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	0

5.3 ALU Test

To document that the designed ALU module functions as intended, a test has been conducted. The test is described in the test journal found in appendix H, and thus only the most important notes are presented here.
The test is performed with a simulation tool and goes through edge cases for each operation. Afterwards the results are compared to expected outcomes generated by a calculator to determine if the ALU works.

Chapter 6

Design of Digital FIR Filter

This section will describe the design of the digital filter mentioned in chapter 3. The filter is implemented on the FPGA in VHDL and enables the CPU to process different given signals by removing unwanted frequencies. The VHDL implementation of the filter can be found in appendix P.8.

6.1 Design and Implementation

As mentioned in appendix B, digital filters can be roughly divided into two types, FIR and IIR filters. The difference between them is the use of feedback in the calculation of filtered samples. The output of an FIR filter depends only on earlier input samples, and thus does not implement any feedback from the output as is the case with IIR filters. This results in FIR filters being unconditionally stable. [3] Even though an IIR filter can achieve better performance than an FIR filter with the same order, due in part to the stability of the FIR filter, the FIR filter has been chosen instead of the IIR filter. [13]

For a FIR filter of length L, the implementation can be represented as a series of multiplications and additions as illustrated in figure 6.1.

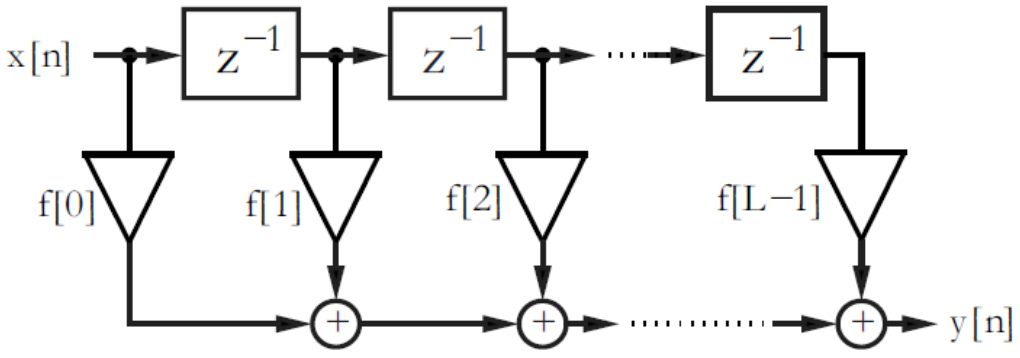


Figure 6.1: Illustration of direct form FIR filter. L represents the number of taps in the filter. [13]

The form in figure 6.1 is called direct form, as it corresponds to evaluating the difference equation associated with the filter, directly. Each multiplier corresponds to a filter coefficient and are usually referred to as "tap weights" or "taps". [13]

When implementing FIR filters on an FPGA, often the transposed form of the filter is preferred to the direct form. The transposed form of the filter is constructed from the direct form by:

- Exchanging the input and output

- Inverting the direction of the signal flow
- Substituting all adders with forks, and vice versa. [13]

When transposing the filter form, the transfer function of the filter does not change. The transposed form is preferred because it has a shorter critical path than the direct form. In the direct form the signal must propagate through all the adders in the system before it reaches the output, whereas in the transposed form, the signal must go through only one adder, the one right before the output. This allows the transposed form to execute faster than the direct form. In addition to this, the transposed form has implicit pipelining due to the delay blocks between the adders. The transposed form of figure 6.1 is illustrated in figure 6.2.

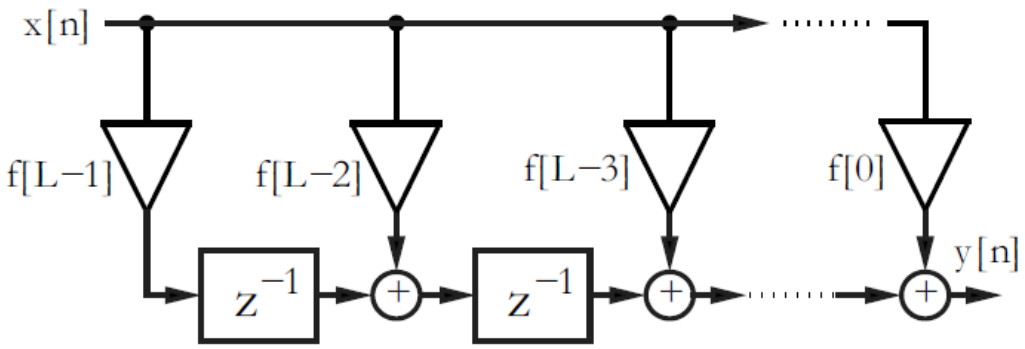


Figure 6.2: Illustration of transposed form FIR filter. L represents the number of taps in the filter. [13]

Filter coefficients
The filter coefficients fed into the multipliers together with the latest sample defines the transfer function of the filter. Generating FIR filters coefficients is outside the scope of this report, however let it be noted that for testing purposes the coefficients were generated using MATLAB's "Filter Designer" application. The coefficients delivered by this application are represented as 16-bit fixed-point numbers as this is the representation chosen for the design of the CPU.

6.2 Code Example

The filter is implemented as a programmable filter in VHDL, based on a program found in [13]. This allows for changing the coefficients during run-time and thus the behavior of the filter. To support this, two processes are defined in the VHDL code, one for loading either coefficients or data and one that performs the convolution of the coefficients and the input signal. The first process is called "Load", and pseudo-code describing this can be seen in figure 6.3.

```
PROCESS (clk, reset, coefficient_in, coefficient_array, system_input)
BEGIN  ------> Load data or coefficients
   IF reset = '1' THEN
        --... clear data and coefficients reg ...
   ELSIF falling_edge(clk) THEN
      IF load_input = '0' THEN
          -- Store coefficient in register
         coefficient_array(TAPS- 1) <= coefficient_in;
         --Coefficients shift one
         FOR I IN TAPS-2 DOWNTO 0 LOOP
         coefficient_array(I)<= coefficient_array(I+ 1);
         END LOOP;
      ELSE
          -- Get one data sample at a time
         input_data_temp <=system_input;
      END IF;
   END IF;
END PROCESS Load;
```

Figure 6.3: VHDL code illustrating the "Load" process in the filter.

The "Load" process does one of three actions depending on the status of the *reset* and *load_input* signals. If reset is set, the registers containing last input sample and the filter coefficients are overwritten with zeroes. If *load_input* is low, a coefficient is loaded into an array and the array is then shifted once to make room for the next coefficient. If *load_input* is high, one data sample is loaded into an array, to be processed in the SOP process.

```
SOP : PROCESS (clk, reset, adder_array)-- Compute sum-of-products
BEGIN
   IF reset = '1' THEN
        --... clear tap registers ...
   ELSIF falling_edge(clk) THEN
      FOR I IN 0 TO TAPS - 2 LOOP -- Compute the transposed
         adder(I) <= x[n]* coef(I) + adder(I+1); -- filter adds
      END LOOP;
      -- First TAP has only a register
      adder(TAPS - 1) <= x[n] * coef(TAPS - 1)
      END IF;
   full_output <= adder(0);
END PROCESS SOP;
```

Figure 6.4: VHDL code illustrating the "SOP" process in the filter.

The second process is called "SOP", and pseudo-code for this process is illustrated in figure 6.4. As with the "Load" process it is possible to reset the registers in the process to zeroes. If reset is not set, the process calculates the filtered output based on the transposed structure mentioned earlier.

The filter has an adder array where the outputs from each individual adder are saved after each

sample goes through. The process works by calculating all the values from the adders, starting from the adder closest to the output and then updating the values in the adder array with the new sums. As each sum depends on the sum of the next adder, the effect of convolution is achieved. When the sum of the successive products forming the result sample by sample has been calculated, it must be scaled down by 2^{15} because of the quantization of the filter coefficients used in the filter. The coefficients are quantized by 2^{15}, one bit less than the output, based on the design found in [13]. To demonstrate the principle behind the implementation, an example is given where an equation for the output is derived based on a filter with a finite number of taps.

Output of filter with four taps
A filter with $L = 4$ taps is illustrated in figure 6.5. As a guide, array indexes have been added to show which values are stored where in the adder array. The result, y[n], can be written as

$$y[n] = a(1) + c(0) \cdot x[n] \tag{6.1}$$

where a(1) is the delayed output from the earlier adder. If all the adders are decomposed into their operands, y[n] can eventually be written as:

$$\begin{aligned} y[n] &= a(1) + c(0) \cdot x[n] && (6.2)\\ &= a(2) + c(1) \cdot x[n-1] + c(0) \cdot x[n] && (6.3)\\ &= a(3) + c(2) \cdot x[n-2] + c(1) \cdot x[n-1] + c(0) \cdot x[n] && (6.4)\\ &= c(3) \cdot x[n-3] + c(2) \cdot x[n-2] + c(1) \cdot x[n-1] + c(0) \cdot x[n] && (6.5)\\ &= \sum_{i=0}^{L-1} c\,(i) \cdot x\,[n-i] && (6.6)\\ & && (6.7) \end{aligned}$$

which is exactly the convolution sum associated with an FIR filter.

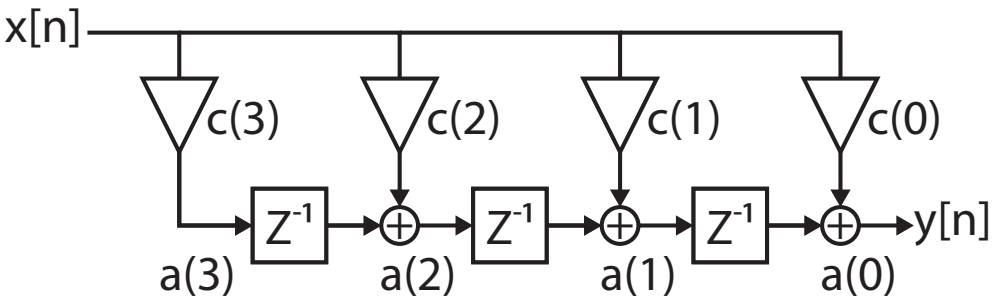


Figure 6.5: Illustration of transposed form FIR filter with four taps

6.3 Capabilities and Limitations of the Filter

The filter has some inherent limitations due to the way the filter coefficients are calculated. The application in MATLAB gives coefficients represented in floating point, which gives a higher resolution than fixed-point. When quantizing the coefficients, a round off error occurs, which can result in a somewhat different transfer function than expected.

Another limitation with the filter is its ability to filter frequencies far below the sampling frequency.

When trying to filter frequencies below 1% of the sampling frequency, the poles of the filter will be clustered very close to each other which in turn, with the quantization problems noted earlier, makes it hard to maintain the needed transfer function. [14]
To counter this effect a method called down-sampling can be used. This method decreases the effective sampling frequency by discarding some of the samples. If the sampling frequency is decreased enough, the effect of the error is smaller. However, to avoid substantial amounts of aliasing, low-pass filtering is necessary before discarding the samples. [15]

6.4 Filter Test

In appendix I, a test journal of the tests performed on the designed filter can be found. In these tests, the filter was first tested to ensure the coefficients were loaded correctly. This test also checked the impulse response of the filter, and both tests passed without issues.

In addition to the first tests, the cutoff frequency and filtering capabilities of the filter were also tested. The cutoff frequency of the filter was compared to what was expected with the given coefficients. The filter was designed for 400 Hz and showed the same on a frequency response. The filter was also able to filter a 300 Hz tone from a mix of a 300 Hz and a 2 kHz tone.

Chapter 7

Design of Peripheral and Memory Controller

This chapter will describe the design of the peripheral and memory controller (PMC). The PMC is in charge of controlling all external data. The internal peripheral control is achieved by memory mapping the internal peripheral I/O. An overview of the PMC is seen on figure 7.1 As seen on the figure both memory and the internal peripherals are accessed by an address. The memory can be both written and read, but peripherals can only be written or read. Beyond the memory mapping the PMC also controls the interrupts from the peripherals. This is done in the Interrupt controller. The internal peripherals are implemented on the FPGA as VHDL components. The VHDL implementation can be found in appendix P.11. The modules are connected to the CPU and enables it to communicate with the peripherals through the GPIO pins of the FPGA.

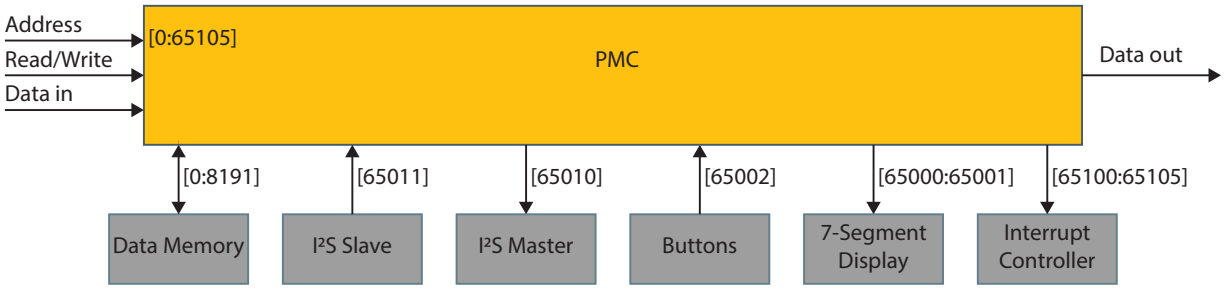


Figure 7.1: Illustration of the memory mapping of the PMC

7.1 I²S internal Peripherals

This section will cover the design of the internal peripherals made for the I²S signals. The I²S protocol is used to transfer the digital sound data between the CPU, ADC and DAC. The I²S interface coded for the FPGA can receive and send a 16-bit word length I²S signal. The interface is coded as two modules. Both modules consist of a base module and a wrapper. The base modules are designed to receive and send a stereo signal. Since this project focus on mono sound, a wrapper has been made for the modules. The input module wrapper only uses the input from the right channel, and the output wrapper send the mono signal out on both channels. Thus, the CPU only needs to process a mono signal. This was done to simplify the system. A conversion to stereo support can be done with minimal adjustments to the module. The full VHDL implementation can be found in appendix P.12 to P.15.

7.1.1 I²S Protocol

The I²S modules are designed to adhere to the I²S protocol designed by Phillips Semiconductors. [16] The I²S protocol use three signal connections and are for one-way communication. The three connections are:

- **Word Select(WS):** The WS connection is used for triggering a new word, and for selecting the left (LOW) and right (HIGH) channel.
- **Continuous Serial Clock (SCK):** The SCK connection is the clock used for logic. On the rising edge the data will be read on the slave. On the falling edge the master will change the data and WS signals.
- **Serial Data (SD):** The serial data connection is used to send the serial data.

A timing diagram of the I²S can be seen on figure 7.2

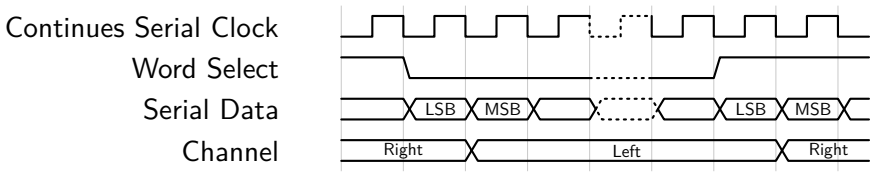


Figure 7.2: I²S timing diagram[16]

7.1.2 I²S Tests

Tests were conducted on the I²S peripherals. A test inspecting the signals in a simulation can be found in appendix J. This test is a simulation of the module where normal operation and edge cases where tested. The conclusion is based on inspection of the resulting wave forms. The test showed that the module behaved as intended and had no apparent flaws in the simulation.

7.2 Onboard Peripherals

The FPGA board has several onboard peripherals. Two of which are used for simple I/O. There are three buttons and four seven segment displays. For these internal peripherals, VHDL module have been made to interface with components.

7.2.1 Input Buttons

The three buttons have been implemented with debouncing logic and an interrupt to the CPU. The test of the debouncing logic can be found in appendix K. The VHDL implementation can be found in appendix P.16 The button module has five I/O ports:

- **CLK:** The clock is used to run the debounce logic.
- **BTN:** This signal is routed to the button connections on the board.

- **DATA_OUT:** The data out is the debounced signals for the buttons.
- **INT:** The interrupt is set high when there is a change in the data out signal.
- **INTR:** The interrupt reset signal is used to reset the interrupt.

To access the state of the buttons at DATA_OUT the memory address 65002 is used.

7.2.2 Seven-Segment-Displays

The seven-segment-displays are used for simple output of a 16-bit word in hexadecimal. To access the memory that is shown on the display, the CPU has to write it to memory address 65000. To control the dots on the displays, the four LSBs of the memory accessed by address 65001 is used. The output can be converted to decimal. To convert the binary to decimal, the binary-coded decimal format is used, and the double dabble method is used for this conversion, the method is explained in appendix E. The VHDL implementation of the seven-segment displays can be found in appendix P.19.

7.2.3 Seven-Segment-Displays Tests

The onboard peripherals were tested to make sure the implementation was successful. The buttons were tested in appendix K and the seven-segment-display in appendix L. The both tests were successful, and the implementation is concluded to be successful as well.

7.3 Memory

The memory is used for temporary storage of data. The memory is implemented as an array 16-bit words as seen in code example 7.3. The array is called RAM, which is also the function that it serves. The address is simply the index in the RAM array. The data and program memory have both been implemented as an array of 16-bit words as seen in figure 7.3. The entire VHDL implementation of the memory blocks can be found in appendix P.17.

7.4 Interrupt Controller

The interrupt controller administrates the interrupt inputs from the different peripherals. If an interrupt has been triggered it will signal to the CPU that an interrupt has occurred, and it will make the address of the interrupt service routine available to the CPU. If desirable, it is possible to enable nested interrupts. Interrupt reset signals are also provided by the interrupt controller. These signals instruct peripherals that the interrupt has been received, and they should set their interrupt signals low. It is possible to change the address of an ISR. To set the ISR address several registers can be written to. These registers can be accessed just as any data memory or peripheral I/O, with an address. The VHDL implementation of the interrupt controller can be found in appendix P.18.

As is the case for the overall system architecture, due to the PMC's reliance on internal and external peripherals to function as intended, it is difficult to perform isolated tests of this module. Therefore, the PMC will be evaluated as part of the system as a whole, during the accept tests, described in chapter 11.

```
ARCHITECTURE falling OF Memory IS
        -- Total 516k memory bits 8k*32 = 256k
        --we use 50% for DataMemory and 50% for ProgramMemory
   TYPE ram_type IS
       ARRAY (WORD_COUNT-1 DOWNTO 0) OF
       std_logic_vector(WORD_SIZE-1 DOWNTO 0);
   SIGNAL RAM : ram_type := (OTHERS => x"0000");
BEGIN

   ----------------------------------------
   -- MemoryReadWrite:
   -- Reads and writes data from the data RAM
   ----------------------------------------
   MemoryReadWrite : PROCESS (clk)
   BEGIN
      IF (falling_edge(clk)) THEN -- Start when the clock rises
         IF write_enable = '1' THEN -- Write enable
         --write Data In bus into RAM array at position address
            RAM(to_integer(unsigned(address))) <= data_in;

         END IF;
         IF read_enable = '1' THEN -- Read enable
         -- writes RAM array at the address position into Data Out bus.
            data_out <= RAM(to_integer(unsigned(address)));
         END IF;
         -- if not writing or reading set all to high impedance to make
         -- sure nothing unintended happens
         IF write_enable = '0' AND read_enable = '0' THEN
            data_out <= (OTHERS => 'Z');
         END IF;
      END IF;
   END PROCESS;
END falling;
```

Figure 7.3: VHDL implementation of the memory

Chapter 8

Design of the Assembler

This chapter will cover the design of the assembler that was designed with the purpose of making it easier writing programs for the CPU. First some basics of the assembly language are introduced, followed by what kind of assembly language the desired assembler will contain. After this the chapter will cover the actual design of the assembler and end with an example containing a few instructions. The source code for the assembler can be found in appendix Q.

8.1 Assembly Language

The assembly language is the most similar to the machine language in form and content and is built of statements that represents instructions [17]. It is decided to divide assembly language into three groups of instructions and analyze what to keep and what to discard. The three groups are: Machine instructions, Assembler instructions, Macro instructions.

8.1.1 Machine Instructions

Machine instructions are instructions written to the specific CPU that is in use. These instructions do one specific task at a time such as making a jump instruction or an ALU instruction using register and or memory. The machine instructions map directly over to machine code. Machine code is a series of bits that is the framework of all instructions in the CPU. These bits are divided into groups of meaning where the opcode is usually the first few bits in the instruction [18].

In table 8.1, an example of a 32-bit machine instruction from the MIPS architecture, can be found, that shows how different instructions are split up in different series of bits.

Table 8.1: Examples of machine instructions from MIPS architecture [18]

Number of bits	6	5	5	5	6	5
Instruction example 1.	opcode	register source	register source 2	register destination	shift (shamt)	funct
Instruction example 2.	opcode	register source	register source 2	Immediate		

8.1.2 Assembler Instructions

"An assembler instruction is a request to the assembler to do certain operations during the assembly of a source module; for example, defining data constants, reserving storage areas, and defining the end of the source module." As written in "Assembler instructions" (2014) [17]. However, the assembler does not translate the assembler instructions into object coding.

The assembler instructions are not to be confused with macro instructions. The assembler instructions do change the written code when compiled and therefore it is not the same as a macro.

8.1.3 Macro Instructions

A macro is a single line of code that is written in the assembler that results in multiple lines of code which executes a specified instruction sequence. This makes it easier for the for programmer to write multiple functions that are repeated in the coding process without having to write each individual instruction again. [17]

8.1.4 Selection of the Assembler

Based on of the previous subsections, a list of desired functions has been made for the assembler.

- The most important functionality of the assembler is that it can output machine code for the CPU. This machine code must be binary and correspond to the written assembly code.
- It is desired to have "jump to label" instructions inside of the CPU and therefore assembler instructions are also wanted. Assembler instructions can rewrite the labels and process them as line numbers. Therefore, they can secure that jump instructions will always point at the correct line number no matter where it is written in the program. This is done without having the programmer worry about if the assembler gives the jump instruction the correct integer.
- There is no appeal to have macro instructions in the assembler since it is not expected to be writing large programs that contains enough repeatable pieces of instructions to justify the implementation. It is also not expected to have the need for writing the same set of instructions multiple times for the desired programs to work.

8.2 The Assembler

The following subsections will describe the development of the assembler and the thoughts behind the programming of it.

8.2.1 Analysis of the Assembler

The core requirement for the assembler is that it can convert assembly language code into binary machine instructions. To do this satisfactorily, the assembler must be able to receive an input, convert it and then output the code again. For this, in- and output files can be used. The output must be 32-bit binary machine code.

In addition to converting text-based code into binary code, it would be ideal if the assembler could also convert integer immediates into binary values. This will make writing the code easier, which is the point of an assembler. As well as making it easier, it will also help reduce errors, since integers are easier to read for the programmer when coding.

On top of this, it is also desired to allow for loops within the assembly code. This will be done via the use of jump instructions in conjunction with labels. A jump instruction could simply be: "JMP

10" which would tell the CPU to jump to line 10 in the code. To make coding more intuitive, as well as allow for moving the loops around without having to change every jump instruction, labels are used. A label will simply correspond to a line-number, and as such a jump can be written as "JMP <label>".

Lastly, it would be beneficial to allow some form of commenting in the assembly code. This will help with maintaining an overview of the code, as well as making it easier to distribute, in- and externally.

8.2.2 Coding the Assembler

The first step in coding an assembler, is to choose a coding language. C is ideal for this purpose, since it is capable of reading, converting and outputting text. In addition to this, it is also lightweight, portable and taught in the current semester.

With C chosen as the language, the next task is to implement the desired functionality. The assembler must be able to recognize assembly code and convert it to machine code. In C, the string library helps with manipulating and comparing strings, which are arrays of chars. Since the assembly code comes with a predictable syntax, it is easier to predict what input will be received. Using knowledge of assembly, a few rules can be set:

1. ONLY one opcode per line of code.
2. Between zero and three operands per line of code.
3. If a line contains a colon ":", it must be a label.
4. If a line contains two forward slashes "//", it must be a comment.

Considering this, the first step is to determine what the current line of code contains. If the line is blank, nothing is to be done. If two forward slashes are found, the text behind it must be ignored. This is because comments can be both inline and on its own. If the line contains a colon, a label is present and must be saved together with its corresponding line number. Labels are done before the actual assembly, since the labels are needed in assembling jump instructions. This will be described later in this section.

If none of these are the case, actual assembly instructions must be present. To determine which kind of instruction it is, and what operands it contains, a series of checks are to be done.

Coding the Assembler: Input and Output

Before the assembler can convert assembly code into machine code, it must be able to read and write code. To make it easier, it is decided to use files for this task. Some analysis of the FPGA and IDE shows that either .hex or .mif files can be used for the CPU. For input, a .asm file used since this is the standard file for assembly. All code will be encoded in ASCII format, and as such, any file can be used if this is respected.

The first thing to implement is the input since this is needed to give the assembler something to convert. The input file is loaded using a file pointer and:

```
fIn = fopen("filename.ext","r")
```

which will open the input file in read-only mode. When the file is opened, the function:

```
fgets(s1, bufSize, fIn)
```

will read a line of up to bufSize characters from fIn and save it in s1. This is ideal since converting one line of code at a time is easier.

For output, the format .mif is used since it has configurable bit depth and widths. Bit depth is the amount of addresses in memory, while bit width is the width of each address. The output file is opened the same way as the input file, however in write-only mode. The converted code is written to the file using:

```
fprintf(fOut,"%s", s1)
```

which will output s1 to fOut. In addition to the machine code, a few lines must be written before and after the code, for the .mif format to work [19] An example of .mif machine code can be seen in figure 8.1. The machine code in figure 8.1, is the assembled code from figure 8.2.

```
DEPTH = 1024;
WIDTH = 32;
ADDRESS_RADIX = UNS;
DATA_RADIX = BIN;

CONTENT
BEGIN

0    :    01110100000000000000000101000001;
1    :    00110100001000010000000000000001;
2    :    01110000001000000000000001100100;
3    :    10010000000000000000000000000001;
4    :    10011000000000000000000000000000;
[5..1023]  :  00000000000000000000000000000000;
END;
```

Figure 8.1: Memory initialization file example.

Some outputs require different output orders, regarding how the code is read. An example is the following assembler code:

```
ADDR $r1 $r2 $r3
```

Is written with following structure:

```
OPCODE RS1 RS2 RD
```

But must be output in binary with the following structure:

```
OPCODE RS1 RD RS2
```

Since the assembler converts the four input words of the assembly instruction, it will output them in the wrong order. To circumvent this, an output array is used. For every line of code, the input words are saved one by one, in the order they are written. When the line has been converted and saved to the array, the assembler compares the first index of the output array with the known opcode outputs. Since the output order is determined by the opcode, this allows the assembler to output the machine code correctly.

Coding the Assembler: Labels

Since labels must be known before the rest of the code can be compiled, these are handled first on their own. The assembler goes through the entire input document and checks for labels. When a label is found, by using strstr to search for colons, the label name and line number is saved in two arrays, with the same index. When a label is used in the code, it is compared to an array of known label names, and the corresponding line number will be used in the actual machine code.

In figure 8.2, an example of how a label can be used to create a loop. In this case, 10 is moved to register one, register one is incremented by one, and if register one is less than 100, it will loop.

```
1 MOVI 10 $r1
2 Loop:
3 ADDI $r1 1 $r1//comment
4 CMPI $r1 100
5 JMPLE #Loop
6 HALT
```

Figure 8.2: An example of assembler code using labels. It's an assembler implementation of a for loop looping 10 times.

Coding the Assembler: Opcodes

Checking which opcode is present is accomplished by using the string library. Since only one opcode is written per line of code, it is only needed to compare the input with an array of known opcode names. Using the function:

```
strstr(s1, s2)
```

two char arrays are easily compared, where s1 is checked whether or not it contains s2. With the opcode known, the corresponding 6-bit value is saved in an output array. This array is used to output the machine code in different orders, since some instructions vary or need zero-padding.

Once the opcode is known, the next step is to figure out which operands to expect. The instructions are split into four groups:

1. Immediates.
2. Registers.
3. Memory.
4. Jumps.

Coding the Assembler: Identifying Operands

Once the opcode and the instruction type is known, the next step is to figure out how many, and which operands are present. Since the assembly language has operands split by white spaces, these are used to check for operands. This is done by saving the location of all spaces in the string, subtracting the current space from the next space and checking if the values is greater than one. If this is the case, an operand must be present, and it is loaded into a temporary string.

The operand type and which operands are presented defines what to do with the operand. The operands are also partly determined by the instruction type. The operands are described below in order of complexity.

Registers only has to be compared with an array of known register names. This is done the same way as when checking opcodes, just one at a time since multiple register can be present.

A jump can either take a register, an immediate or a label as operand. To check which operand it is, two types of jump opcodes are used, as well as some other checks. There are both JMP and JMPR opcodes. The first is for use with immediates and labels, the second is for registers. To distinguish between labels and immediates, a number symbol " #" is used for labels, e.g. JMP #label.

The second most complex is the immediate instruction type. These can take either immediates, signed and unsigned, and labels. The two are distinguished using the number symbol "#", same as for the jump instructions. Signed and unsigned are distinguished by using a dash "-".

Finally, the most complex instruction type is memory instructions. These can take either an immediate, a register or a register plus an immediate as input. To distinguish the three, it is first checked if a plus sign "+" is present. If this is not the case, the assembler checks for a dollar sign "$", since these are used for registers. If neither are present, a lone immediate must be present. In memory instructions, memory addresses are written inside square brackets "[ ]", which are to be ignored by the assembler.

Coding the Assembler: Converting Operands

Once the type of operand is known, and the operand is stored in a temporary string, it must be output correctly. For registers, the corresponding 5-bit value is saved to the output array, the same way as the opcode.

For immediate values, the number shall be converted to binary and then saved for output. To do this, the string is converted to an integer using:

```
int1 = atoi(s1)
```

which will save s1 as int1 if s1 only contains numbers. Afterwards, the function:

```
itoa(s1,int1,int2)
```

can be used to convert int1 back into a s1, with a desired radix int2. This allows for conversion to binary, with the downside that the binary value will only be as many bits as is necessary. For example, 5 would be converted to 101, which means it must be padded with zeros in 16-bit unsigned format, which is the desired immediate length. If the number is negative, using itoa will output two's complement at 32-bit length, since the program is compiled for 32-bit. To achieve the desired 16-bit, the bottom 16-bit are saved for output.

Labels are handled in the same way as register, when it comes to checking which is used. Once the label name is found in the array of known labels, the corresponding line number is converted to binary, same as for immediates, and saved for output.

Coding the Assembler: Additional features

To make assembly easier for the user, as well as to make the assembler more robust, a series of functionalities are implemented. The first is to allow the user to specify in- and output file names, as well as the desired memory depth. This can be done either via arguments in command prompt, or after launching the executable. The assembler will also prompt the user whether they want to create the output file, should this not exist.

The assembler will also print all outputs, as well as some debugging information, in the command prompt window. This helps with debugging potential errors in the assembly code.

8.2.3 Assembler Example

To give an overview over how the assembler is used, as well as demonstrating its capability, this section will contain a small example. The example will be a simple assembly program, which will use the different functionalities of the assembler. The assembly code to be assembled can be seen in figure 8.3.

```
1 MOVI 0 $r1
2
3 Main:
4 STORE $r1 [65001] //Saves r1 to memory 65001,
5                   //which is the LED display.
6 ADDI $r1 1 $r1    //Increments r1 and saves in r1.
7 CMPI $r1 100      //Compares r1 with 100.
8 JMPLE #Main       //If r1 is less than 100, jump to Main.
9 HALT              //Else halt program.
```

Figure 8.3: Simple assembly program that will count from 0 to 100 and display the current number on the sevensegment display.

Assembling the code from figure 8.3 using the assembler will be done by launching the executable, to show how this works. The input file will be Main.asm and the output file will be MemInit.mif. The memory depth of the CPU is set to 1024. The assembler will also be instructed to create the output file. The command prompt output of the assembler can be seen in figure 8.4.

```
Zero arguments given...
Did you run the program as .exe? Y/N: y

Would you like to give arguments now? Y/N: y

Write <input-filename.extension>: Main.asm

Write <output-filename.extension>: MemInit.mif

Write <memory depth>: 1024
1 label(s) were found
Label-check done...

011101 00000 00001 0000000000000000

Label Found
011111 11101 00001 1111110111101001

001101 00001 00001 0000000000000001
011100 00001 00000 0000000001100100
100100 00000 00000 0000000000000010
100110 00000 00000 00000 00000 000000
Assembly successful. Press enter to exit...
```

Figure 8.4: Example code being assembled using the assembler, opened as executable.

The assembled code is output in .mif format, and can be seen in figure 8.5.

```
DEPTH = 1024;
WIDTH = 32;
ADDRESS_RADIX = UNS;
DATA_RADIX = BIN;

CONTENT
BEGIN

0    :    01110100000000010000000000000000;
1    :    01111111101000011111110111101001;
2    :    00110100001000010000000000000001;
3    :    01110000001000000000000001100100;
4    :    10010000000000000000000000000001;
5    :    10011000000000000000000000000000;
[6..1023] : 00000000000000000000000000000000;
END;
```

Figure 8.5: Simple machine code program that will count from 0 to 100 and display the current number on the seven segment display.

8.3 Assembler Test

To ensure the assembler works as intended a final test is performed on the assembler as a stand-alone component. The actual test journal from the assembler tests can be found in appendix M and therefore only the most important notes are presented in this section. The test and verification uncovered a considerable number of errors, as are documented in the test journal in appendix M. The assembler was tested again with the same set of tests, after the errors were corrected. The test results were perfect, and the assembler worked as intended. With this the assembler is ready to be a part of the product.

Chapter 9

Design of the PSoC Module

The PSoC and the connected peripherals have the purpose of fulfilling several product requirements and design choices specified in chapter 2. To meet these requirements, there is a need for both external circuit connected to the PSoC as well as an internal hardware design within the PSoC. The source code for the PSoC can be found in appendix R.

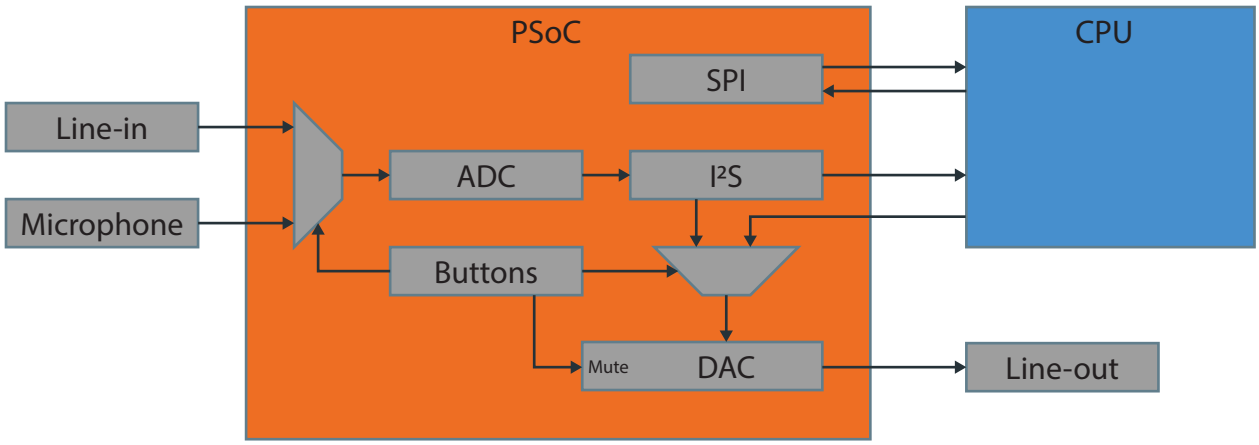


Figure 9.1: Overview of the PSoC features

The object of the internal circuits in the PSoC is to sample an analog signal and convert it into the I^2S bus before the signal can be transferred to the FPGA. Additionally, the PSoC should also be able to communicate with the FPGA via the SPI interface, this requires the implementation of an SPI slave. The internal hardware and their connections to the FPGA can be seen on figure 9.1.

The object of the external circuit is to implement a line-in and microphone input for the ADC. Additionally, it is required to be able to toggle between the two analog inputs and between the I^2S output of the PSoC and that of the FPGA. The toggling is implemented with multiplexers in the PSoC internal hardware and these are individually controlled by mechanical buttons.

First the internal hardware design and software programming of the PSoC will be described followed by the external circuit.

9.1 PSoC Hardware and Software design

The PSoC hardware and software design is split into three modules, each with their own purpose. The first module is the delta sigma ADC and I^2S output, the second module is the SAR ADC and SPI Bus and the final module is the User Interface Circuit. Each of these modules will be described in the three following sections.

9.1.1 Delta Sigma ADC and I^2S Output

The primary purpose of this module is to sample the analog sound input and then output every discrete sample value through an I^2S bus. The top-level design of this module is shown in figure 9.3
The delta sigma ADC is configured as single ended and to sample 16-bit words at a sample-rate of 44,100 Hz. The input range of the ADC is set to 0 V to 2,5 V.
The Direct Memory Access (DMA) is connected through its DMA request pin to the ADC's End of Conversion (EOC pin), this pin is set high on the rising edge of the ADC, which signals the end of a single analog to digital conversion.

The Direct memory access controller is used in the PSoC to transfer data directly between memory and peripherals. The function of the DMA is based on Transaction Descriptors (TD). A TD contains all the information about the transaction that is to be transferred. To start a transaction, a hardware request must be specified. A request can be any digital signal and is often a signal intended as an interrupt signal for a CPU. However, if a continuous task is to transfer a piece of sampled data, the DMA can be utilized to alleviate workload from the CPU.
If the word length of the data that is going to be transferred is the same for both the source and destination, then the DMA only requires a single transaction for each transferred word. However, if the source and destination specify different word lengths, a workaround is required, because a direct memory transfer is not suitable.
The latter is exactly the case for the DMA transaction between the 16-bit delta sigma ADC and the 8-bit First-In-First-Out (FIFO) buffer in the I^2S master. The I^2S master requires to transfer 8-bit before another 8-bit can be transferred and thus finalizing the ADC sample transfer.

A solution to this issue is to have the first TD transfer the samples from the ADC to a memory buffer. Then each byte can be routed sequentially with two additional TDs to the I^2S master. The I^2S master will signal when its FIFO buffer is empty, thus enabling each TD transfer into the buffer. The Transfer Descriptor loop functions by every TD pointing to the next TD and at the last TD it points back to the first one. An illustration of the data transfer inside the DMA for the above example is shown in figure 9.2.

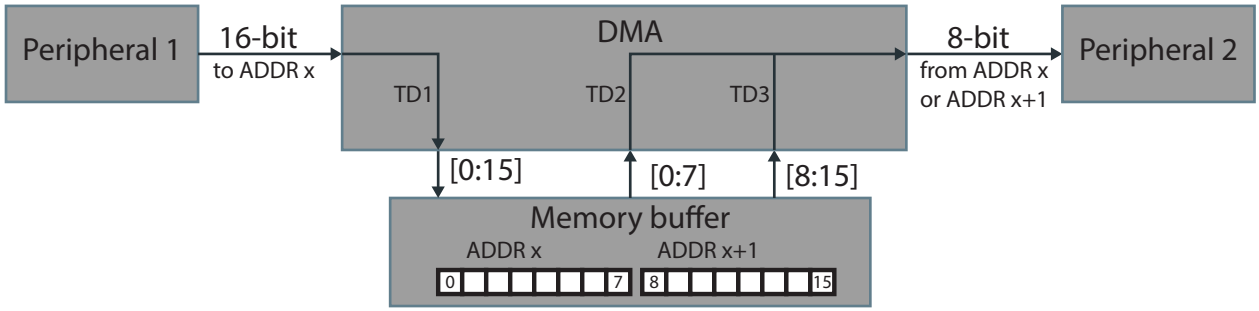


Figure 9.2: Illustration of the use of the DMA and a memory buffer to transfer data from a 16-bit peripheral to a peripheral with an 8-bit FIFO-buffer

Once the DMA is connected to the I^2S master and continuously transfers data, the I^2S master will output each byte through its Serial Data Output (SDO pin) in a 16-bit word. The Serial Clock (SCK pin) outputs the given frequency which the I^2S master is set to, for 16-bit words and 44,1 kHz sample rate this output will be 2,8224 MHz. Finally, the Word Select (WS pin) will switch between high and low depending on which audio channel is being transferred.

Additional features which have been added to this module consists of several input and output multiplexers. The input multiplexers are designed to switch between the input signals of the delta sigma ADC, this allows for both a line-in input and a microphone input for the same ADC. The multiplexer switching is triggered by a button, switching between high and low states. The button input also has debouncing hardware to stabilize the input and a toggle flip flop component which allows a tactile switch to function as a toggle switch.
Likewise, for the I^2S output there is a multiplexer which switches between which output is send to the I^2S DAC. The first input for this multiplexer is the direct output of the I^2S master, the other input is the I^2S output of the FPGA module. Like the first set of multiplexers, this multiplexer is controlled by a debounced toggle switch.

9.1.2 Successive Approximation ADC and SPI Bus

This PSoC module has the purpose of both sampling an analog signal, which is the voltage drop across a potentiometer, and enabling an SPI interface between the PSoC and the FPGA. The top-level design of this module is shown in figure 9.4. The analog sampling is executed by the Successive Approximation Register (SAR) ADC which samples 12-bit with a default sampling rate of 55,556 Hz in single ended mode. The input range of the SAR ADC is 0 V to 5 V which matches the variable output range of the potentiometer circuit. Like the previous PSoC module, this ADC has a DMA connected to the EOC pin which feeds the 12-bit ADC output into a 16-bit buffer in RAM.

The purpose of the SPI slave is to transfer the sampled ADC data to the FPGA module. The SPI slave is configured to 8-bit transfer in full duplex mode with a sampling rate of 1000 kbps. A DMA is connected through its hardware request to the SPI slaves TX interrupt pin, the configuration of this DMA is similar to the DMA connected to the I^2S master component in the previous module. However, the SPI slave differs slightly for two reasons; first it requires that a zero byte is transferred before any data can be transferred, secondly the SPI slave requires that the connected SPI master first transfers a byte to the slave before the slave starts transferring any data.

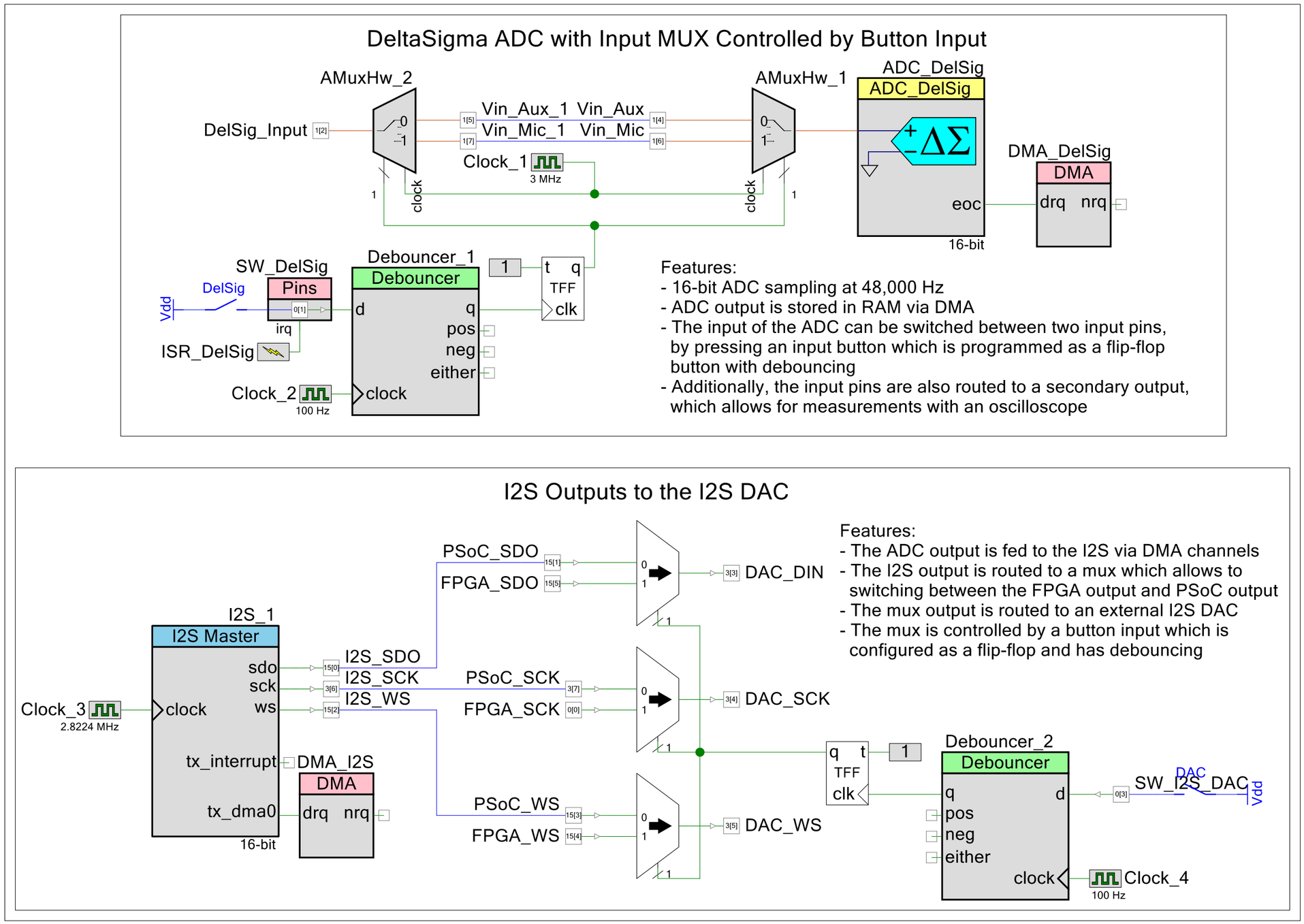


Figure 9.3: Top design of the hardware of the first PSoC module

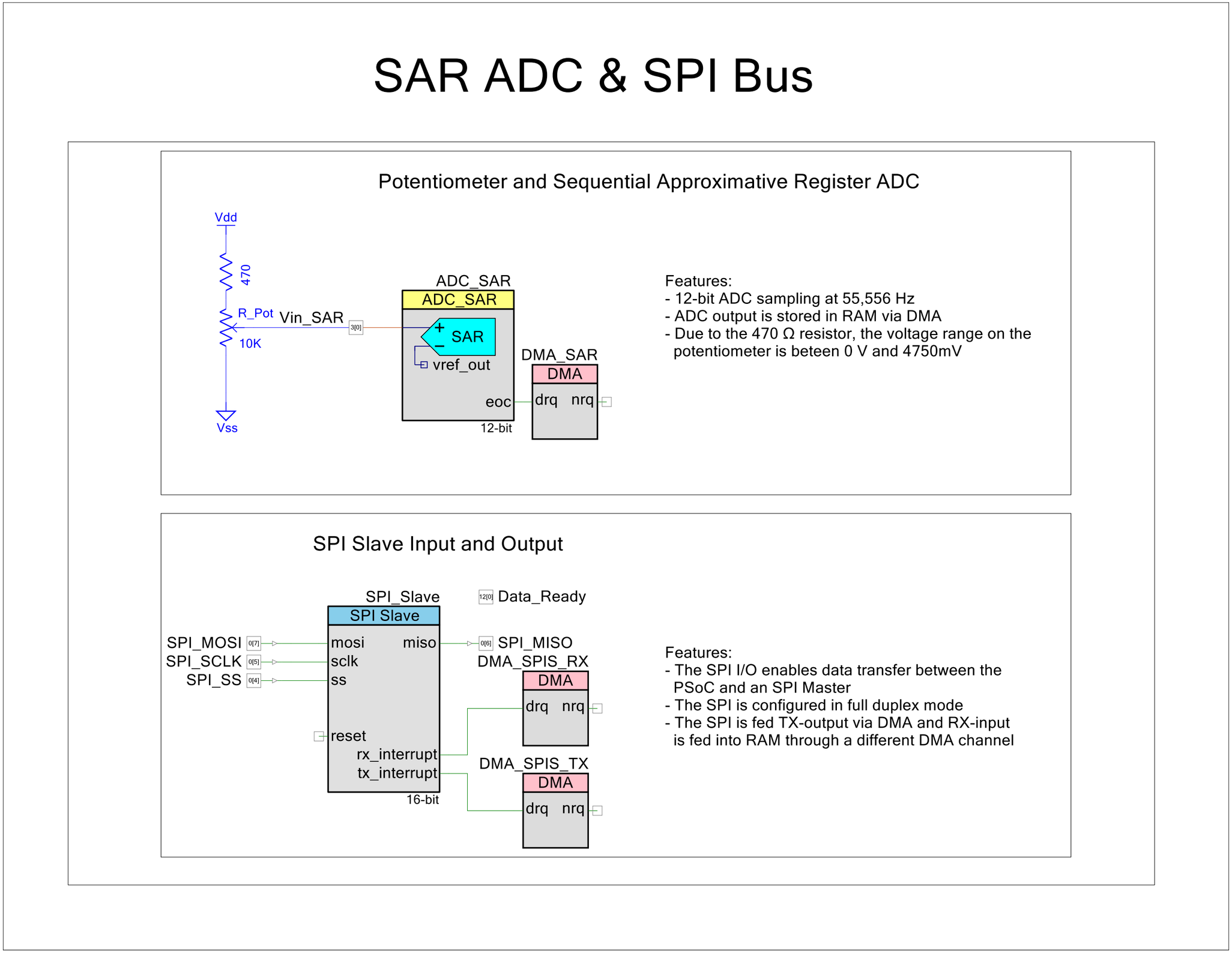


Figure 9.4: Top design of the hardware of the second PSoC module

9.1.3 User Interface Circuit

This PSoC module consists of the remaining user interface components for the PSoC system. This includes the following: a mute button for the DAC, a character LCD and eight individually controlled LEDs. The top-level design of this module is shown in figure 9.5. The mute button is configured like the two other toggle switches used in the first PSoC module. However, instead of outputting a high or low signal to internal hardware this button will output 0 V or 5 V to the mute input of the DAC depending on the toggle state.
The character LCD is a two by sixteen-character display with individually addressable pixels. The LCD is programmed in software to display the status of the three input buttons. And finally, once programmed into the SPI component, the LCD will also be able to display any received data from the FPGA. The eight LEDs are individually addressable through their respective output pin. Currently, the LEDs are programmed to emit light along the sliding potentiometer depending on the sliders position.

9.2 PSoC Peripherals Design

The external peripherals connecting to the PSoC consists of a microphone circuit, line-in circuit, tactile buttons, LCD, potentiometers and LEDs.
The microphone and line-in circuits are very similar, they both consist of variable signal amplification and variable voltage shift. The amplification itself is realized by two non-inverting operational amplifiers powered by single rail power from the PSoC. The amplification and voltage shift can be adjusted by the turn of four potentiometers, one for each voltage shift and amplifier for both line-in and the microphone.
The three buttons are simple circuits inspired by an Arduino example [20]. When the tactile buttons are actuated they will send 5 V to the designated pin through a wire or PCB trace.

9.3 PSoC Tests

The test results of the complete system in appendix N confirms a successful design of the PSoC peripherals. The results show that the 16-bit ADC, I^2S master and I^2S DAC can convert an analog signal to digital and back to analog without introducing much distortion. This means that the design of this module meets the required specifications.
However, for the microphone circuit there is a lot of noise introduced at the input. At the output it is almost impossible to tell where the input sine wave is located along the frequency axis. The conclusion of this problem is that the microphone circuit is designed poorly. However, due to the test setup, where the microphone was located below a small speaker and with a lot of background noise, this will also introduce a lot of noise to the input. Aside from the microphone the implementation is a success.

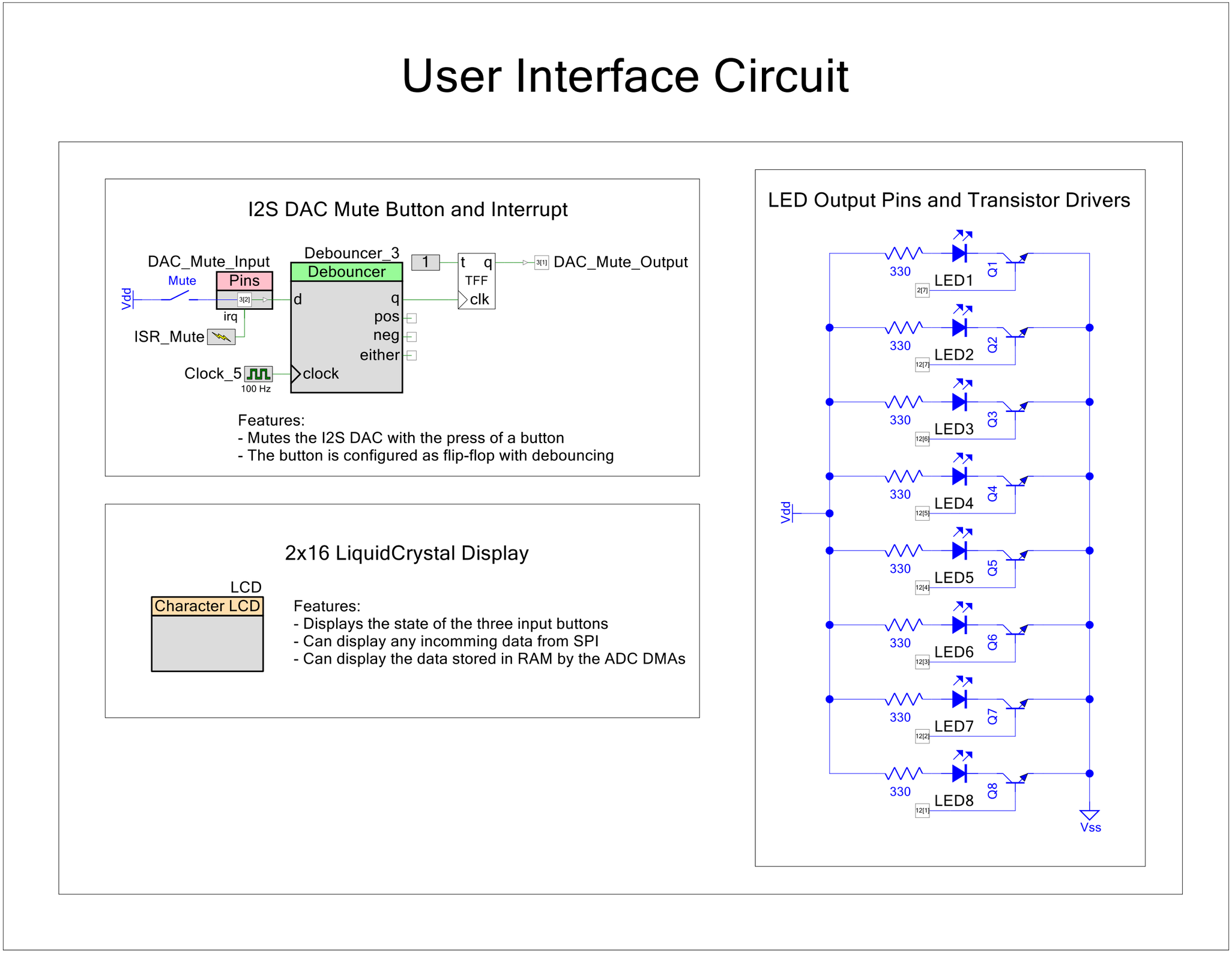


Figure 9.5: Top design of the hardware of the third PSoC module

Chapter 10

Design of the Breakout Printed Circuit Board

The breakout PCB is the final piece of hardware which has been designed. The purpose of this PCB is to place all the modules excluding the FPGA on a single PCB and most importantly improve the performance of all the connected modules. This chapter will cover the design of the PCB which includes reducing noise in the system thus improving performance and enabling the entire system to be tested thoroughly.

10.1 PCB Design Techniques and Implementations

In PCB design, reducing noise is critical to achieving reliable performance. In fact, there are so many different design aspects that all of them will not be covered in this chapter, nor were every aspect implemented in the PCB design. However, a decent amount of proper PCB design techniques has been utilized and they are summarized in the following list:

- Trace impedance
- Component placement
- Separate analog and digital grounding
- Ground planes
- Return loops
- Bypass and bulk capacitors
- Crosstalk mitigation
- Guard traces
- Three-width spacing

The theory behind the above design techniques will be described in the following sections. Additionally, these design techniques will be implemented onto a breakout PCB.

PCB Trace Impedance

Every PCB trace has an impedance, and specifically the resistance in this impedance can cause noise in a given system. Especially analog signals routed to high bit-depth ADCs are susceptible to offset error from trace resistance. [21, p. 12.5]
In this project two ADCs are used, a 16-bit delta sigma ADC with an input resistance of 74 kΩ and a 12-bit SAR ADC with an input resistance of 148 kΩ. [22, p. 89] For a 16-bit ADC, one LSB error is equivalent to an offset error of 0,0015 % and for a 12-bit ADC it is 0,0244 %.
For the implemented ADCs, one LSB error will occur when the total trace resistance is 1,08 Ω for the DeltaSigma ADC and 36,112 Ω for the SAR ADC. [21, p. 12.5]
Furthermore, this results in a lot of headroom for the trace length with a 0,3 mm wide trace for the analog signals before any offset error occurs. However, it is still desired to keep the trace length as

low as possible to improve overall performance.

Crosstalk Mitigation

Noise due to cross talk occurs when one aggressor signal (often a digital signal) affects a victim signal (often an analog signal) without the two signals being connected. The two most common types of cross talk are capacitive coupling and inductive coupling. [23, p. 6]
Capacitive coupling occurs when two different signal traces are routed directly above one another on different layers of a PCB. To mitigate this crosstalk, the two different signal traces should be routed at a ninety-degree angle when the traces need to cross each other. [23, p. 6]
Inductive coupling occurs when two or more different signals become magnetically coupled. To mitigate this, the traces should be placed as close to a ground plane as possible so that the ground plane absorbs the magnetic coupling. [23, p. 7] Additionally, if the traces are placed with a distance to each other of three times the width of the largest trace 70 % of the magnetic coupling is reduced. [23, p. 7]
Finally, guard traces can be utilized to reduce coupling between two signals. This mitigation is simply implemented by placing a ground trace in-between any parallel routed traces. Guard traces can be very easily implemented by placing a copper plane connected to ground around all signal traces in a given layer. [23, p. 7]

Ground Planes and Separate Analog and Digital Grounding

Grounding in circuits is very important and should not be left untouched when designing PCBs.
When designing a PCB with mixed signals it is advantageous to create separate ground connections for analog and digital signals. For example, this is important when using an ADC with a sensitive sensor or line-in connection, because a shared return loop with digital or high current components will result in an offset error. [23, p. 4]

With these and previous considerations in mind, the PCB has been designed so that all the analog signal components are placed as close as possible to each other and to the respective ADC pinouts on the PSoC kit. Furthermore, the analog ground is a copper plane on both sides of the PCB that spans the area of all the analog signal components and connects to the analog ground pinout of the I^2S DAC.
The rest of the components, both user interface and digital I/O, are placed as close as possible to their respective pinout and a copper plane on both sides of the PCB is connected to ground.

Bypass and Bulk Capacitors

Bypass and bulk capacitors are utilized to stabilize power supply voltages which are also affected by noise.
Bypass capacitors are placed near the power supply pin of each individual component and connects to ground. Bypass capacitors remove high-frequency noise which stems from the supply [23, p. 12]. These capacitors are often ceramic capacitors with capacitance ranging from 1 nF to 100 nF. [23, p. 12]
Bulk capacitors are placed near the regulators or as close as possible to the power supply output on the board. Bulk capacitors remove low-frequency noise and help stabilize power supplies for longer periods of time. [23, p. 12] Bulk capacitors are usually between 1 µF to 100 µF.

10.2 PCB test

Based on the above considerations a PCB has been designed and constructed. The PCB bill of material (figure G.1) and layout, both with and without copper planes (figure G.3 and G.2 respectively), can be found in Appendix G. Furthermore, during the soldering of the PCB all the connections have been white-box tested and verified. The PCB was also utilized in the test of the PSoC peripherals in appendix N, and based on this test it is deemed that the hardware on the PCB works as intended.

With the PCB constructed, each individual subsystem has been designed, developed and tested to some extent. It is now possible to combine the subsystems into one and run a complete system test. In the following part, chapter 11 will document this test and eventually lead to a conclusion determining to which degree the system follows the requirements stated in chapter 2.

Part IV

Accept Test and Conclusion

Chapter 11

Accept Test

With all parts of the system having been designed, they must all be tested together. This chapter will focus on setting up and performing tests that will verify if the system either works as intended, is flawed or does not work at all. The verification is done by investigating if the system meets the design specifications from chapter 2.

11.1 CPU Test

To test the CPU, a program that filters incoming audio while running a main loop is programmed in the assembler. The main loop of the program is used to test all the opcodes, while ensuring the interrupt functionality works as well.

11.1.1 Test Procedure

When testing the CPU, the following setup will be used:

1. The CPU is clocked at 1 MHz.
2. The ADC is set to sample at 44,1 KHz.
3. The CPU internal I^2S master's output signal is set to 44,1 kHz.
4. The CPU is configured to have 8192 Words of program memory, and 8192 words of data memory.
5. The ADC samples the AUX line-in.

The program template used to perform the test can be found in figure 11.1. Note that the code is just a template, and it is populated with meaningful instructions for each individual test.

When testing the functionality of the opocodes, the program contains a main loop with randomly chosen operands, which results in a final value. The main loop is programmed to display the final result of the calculations on the seven-segment-display. These calculations were checked through by hand, and the final value will be compared with the one on the display.

```
// Set up Interrupt service routines and other initialization code

//I2S ISR example
MOVI #I2SISR $r2 //Store the address for the ISR in the interrupt controller
STORE $r2 [65102]
MOVI 512 $r2 //Configure the interrupt controller to enable the ISR and disable nesting
STORE $r2 [$r1+1]

//Initiate Main Loop
MAINLOOP:
//
//place main loop code here
//
JMP #MAINLOOP //Returns to start of main loop

//ISR code goes here

//I2S ISR example
I2SISR:
// <-- place ISR code here
POP $pc //POP pc to return to main loop
```

Figure 11.1: Code overview of the accept test program.

First, all opcodes were tested without any loops or interrupts configured. Secondly, the audio filtering and interrupt functionalities were tested without a main loop containing code. Finally, an extensive program containing all instructions in a main loop, combined with button and I^2S ISR, was used to test the full functionality.

11.1.2 Test Results and Analysis

The CPU is able to filter the incoming data and transferring it to the DAC when there is no main loop running simultaneously. It is unknown why this is the case, since the CPU is able to let the audio pass through unfiltered while running a main loop. It appears the FIRSAI and FIRSAR opcodes do not work, while a main loop contains any other instruction than NOP instructions.

In addition to filter opcodes not working when a main loop is running, the NEGR and NEGI opcodes do not work properly either. This is because the control structure maps these to the ALU's invert command, rather than negate. The fix for this issue appears to be trivial, however, when implemented the CPU ceases to function. The cause of this is currently unknown, though it is theorized to be because of a synthesis error.

Furthermore, the JUMPNQ opcode does not work either, however it is known to have worked as intended in previous iterations of the CPU. The reason for this is unknown and will not be further examined due to time constraints.

11.2 Noise Tests

Since the designed CPU is primarily made for audio processing, it is also desirable to test and quantify output noise generated throughout the system's various parts.

To comprehensively test the systems generation of noise, a variety of measurements have been performed. The test aimed to quantify noise, and thus spectral measurements were made.

11.2.1 Test Procedure

To perform the test, the following test setup was used:

- Measurements were performed with the Analog Discovery 2. Waveforms Version 3.7.22 was used as the software interface.
- To isolate the system from noise originating from the electrical grid, battery power (in the form of a laptop) was used to power the PSOC and its peripheral subsystem, while USB power from a desktop was used to power the FPGA.
- All measurements were performed on the DAC's right channel output.
- The CPU on the FPGA was clocked at 1 MHz.
- The frequency spectra were measured in the interval of 0 - 22.05 KHz (i.e. Half the sampling frequency of 44.1 KHz). Each measured spectrum consists of 4096 points, and results in a sampling resolution of 5.38 Hz.
- For all measurements, 500 measurements were made, and the spectra show the average of the measurements. This procedure filters random noise.
- Amplitude is given as dBV with respect to 1 Volt.
- The only load present on the output channel is the Analog Discovery 2. The Discovery has an input impedance of 1MOhm || 24pF.
- Unless otherwise specified, the FPGA CPU is assumed to be programmed to receive I^2S data from its I^2S slave and send it straight through to its internal I^2S master.

In total, seven tests were conducted, and they are enumerated from "A" to "G". Each test, and its purpose, is listed below:

To begin the tests, several baseline measurements were made:

A When no power is supplied to any part of the system (including the DAC).

B When power is applied to the DAC, but it receives no input signals.

C When a 441 Hz, equal to $\frac{44100}{100}$ (one hundredth the sampling frequency), digital sine wave using the full 16-bit dynamic range is generated in the FPGA and send to the DAC.

Following these baseline measurements, different areas of the system were short circuited to identify the major sources of noise:

D Grounding the AUX line-in to the system. This test aims to measure the total noise generated throughout the entire system.

E Making a "software short circuit" in the PSoC ADC, resulting in the PSoC only sending '0' through its I^2S connection to the FPGA. This test will provide insight into noise generated in the ADC, CPU, and DAC.

F Making a "software short circuit" in the FPGA CPU, i.e. programming the CPU to only output '0' through its I^2S connection to the DAC. This test will quantify noise produced by the FPGA and the DAC.

G Grounding the AUX Line in to the system and programming the CPU to low-pass filter the signal. The filter matches the filter tested in appendix I. This test investigates if filtering affects the generated noise.

11.2.2 Test Results and Analysis

Starting with the baseline tests. Test A, measuring DAC output when power is not supplied, establishes the analog discovery's noise floor to be about 90 dBV. Test results are shown in figure 11.2. Similarly, test B, measuring the DAC's output when power is supplied, but input is lacking, sets its noise floor at just below 70 dBV. Test results are shown figure 11.3.

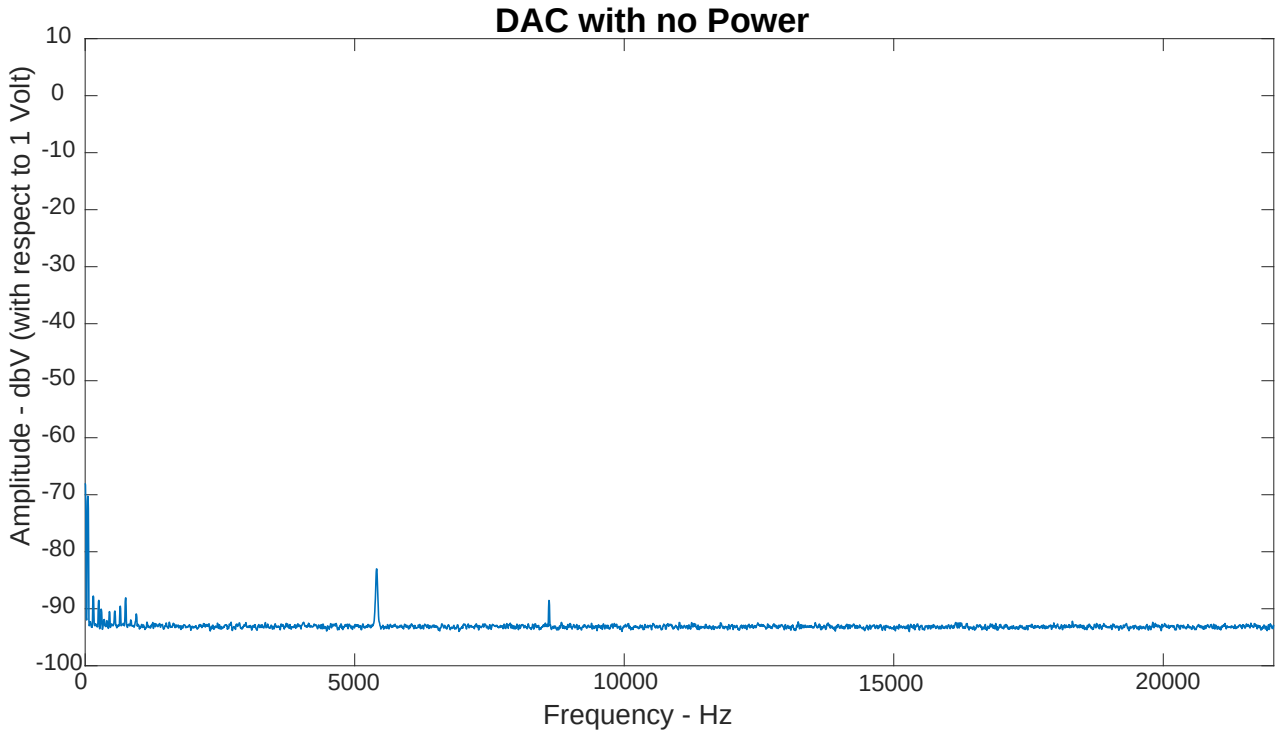


Figure 11.2: Results of test A. Output spectrum of DAC when no power is applied to it

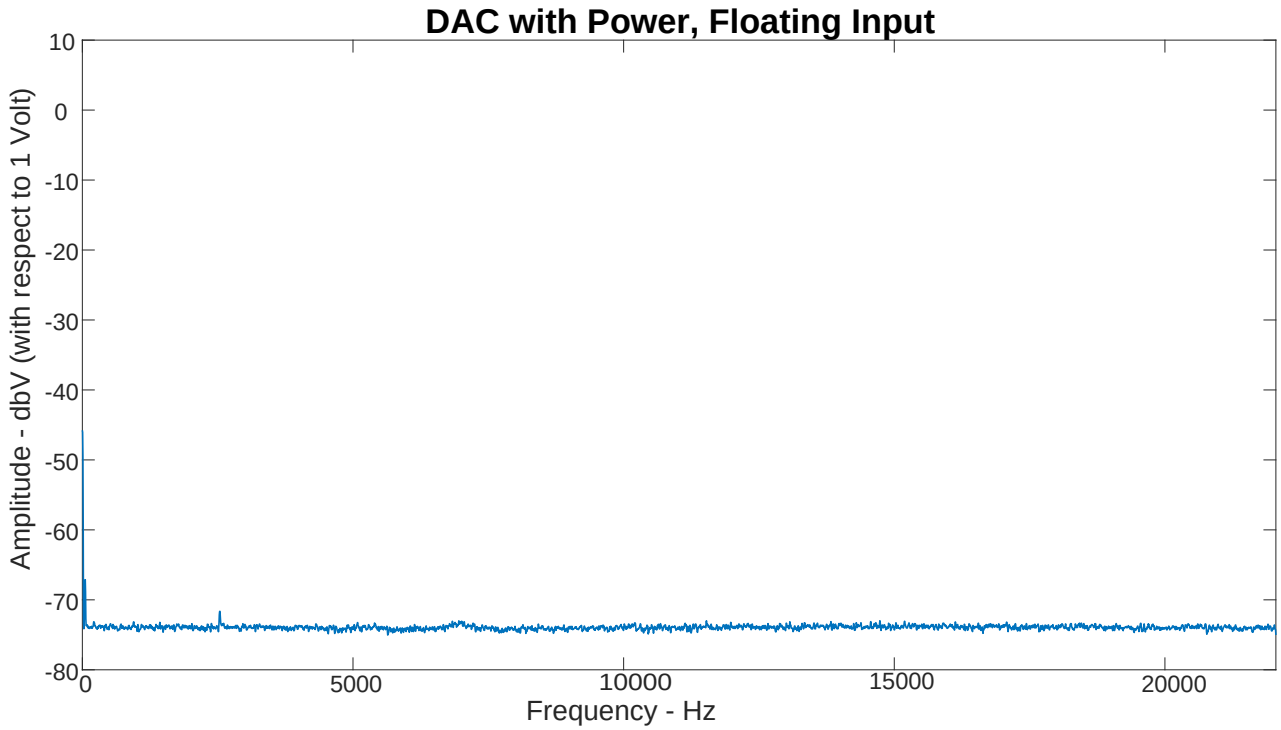


Figure 11.3: Results of test B. Output spectrum of DAC when power, but no input, is applied to it

Test C, results illustrated in figure 11.4, clearly shows that the generated test tone peaks at around 0 dBV. Additionally, "spikes" of varying amplitude is present between 2.5 and 18 KHz. Curiously, these spikes are placed at either ~340 Hz or~680 Hz apart.

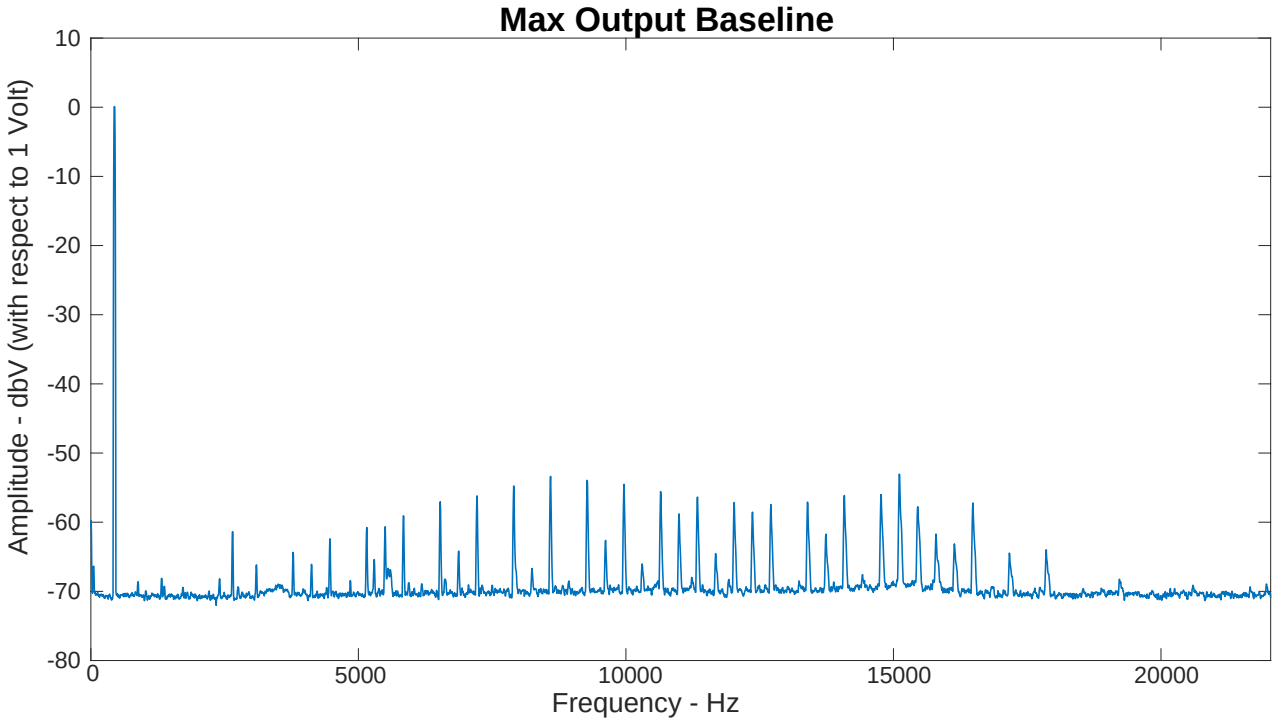


Figure 11.4: Results of test C. Output spectrum of DAC when a 441 Hz full scale tone is send to its input

Moving on to the systematic noise tests. Like in the previous test, in test D, results are shown in

figure 11.5, several spikes placed at either ~340 Hz or ~680 Hz apart, are present between 2.5 and 18 KHz. This indicates that the observed noise in test C is not an artifact caused by the test tone, but rather a constant noise of the system.

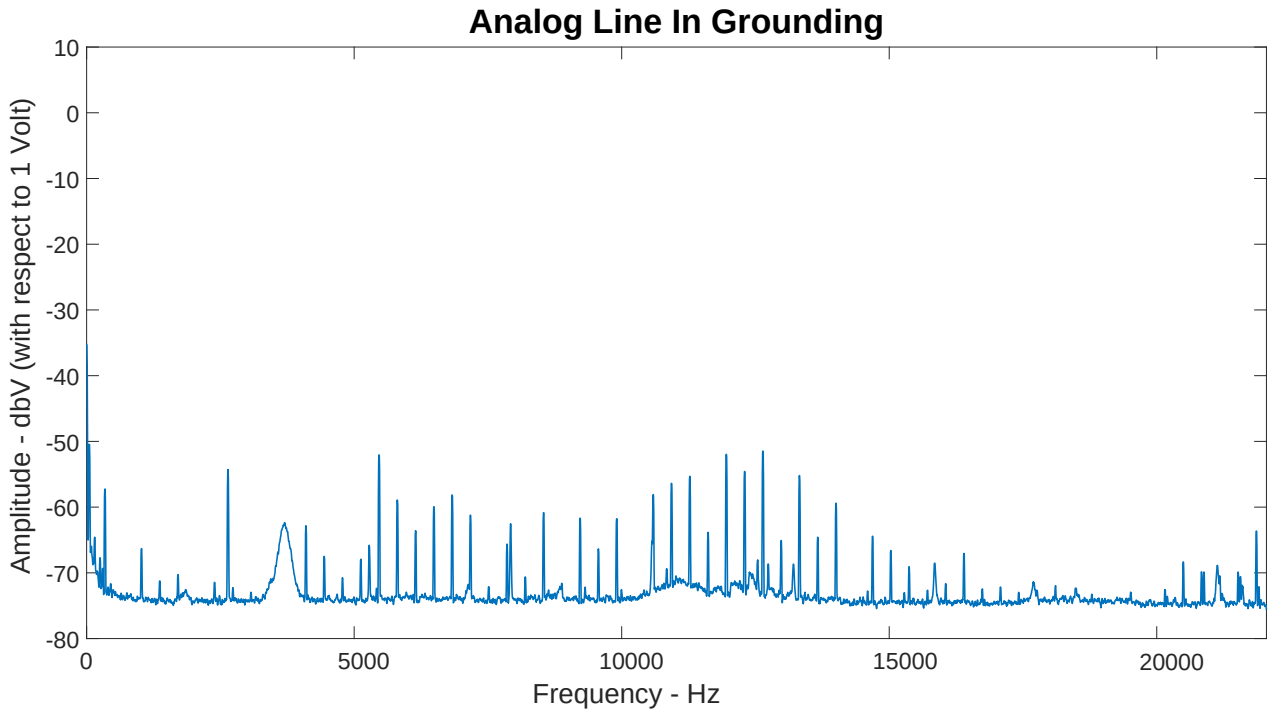


Figure 11.5: Results of test D. Output spectrum of DAC when the front end of the system is grounded

To try and isolate the source of the noise, the PSoC is digitally grounded (thus the analog input is completely ignored by the system) and test E is performed. Figure 11.6 displays the collected data. When compared to figure 11.5, the overall noise appears to be lower, however the overall noise pattern remains.

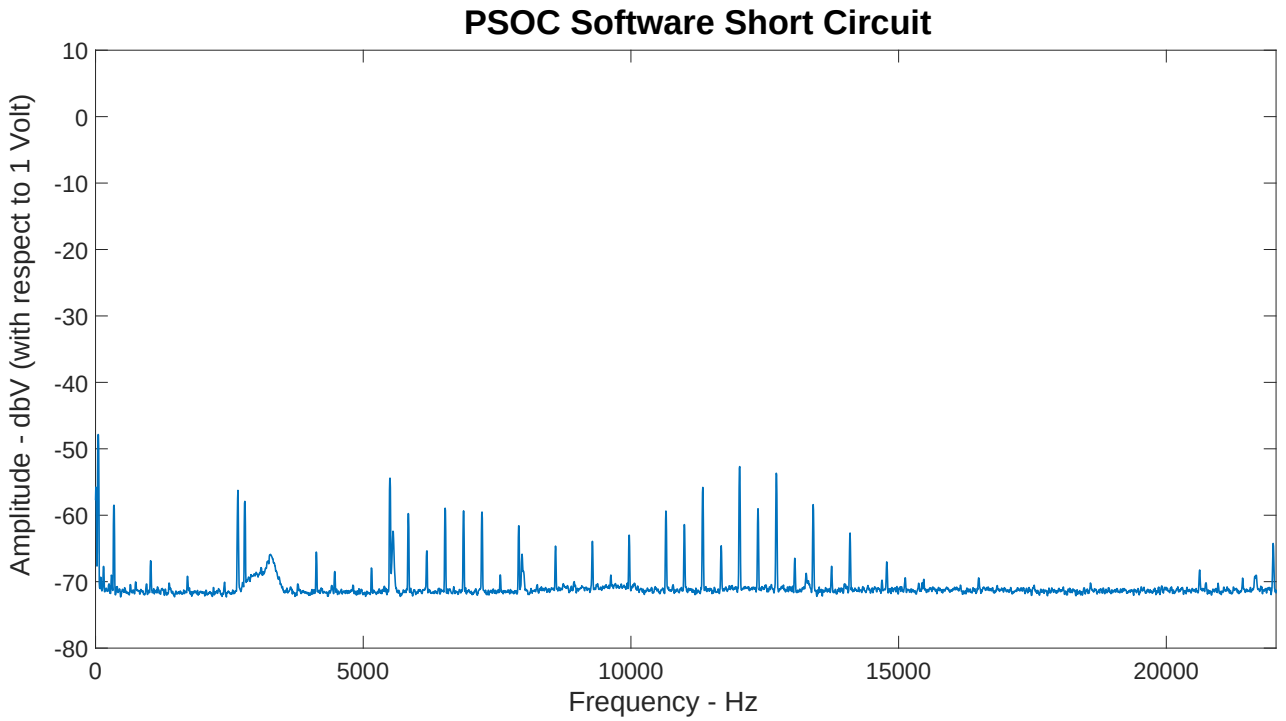


Figure 11.6: Results of test E. Output spectrum of DAC when PSoC outputs raw zeroes. This data is routed through the FPGA before reaching the DAC

To further isolate system subcomponents, test F is performed, and the FPGA is digitally grounded to isolate itself and the DAC from the PSoC's ADC and the analog input. The results are shown in figure 11.8, and are almost identical to the results of test E. These results imply that the PSoC's ADC is an insignificant source of noise in the system.

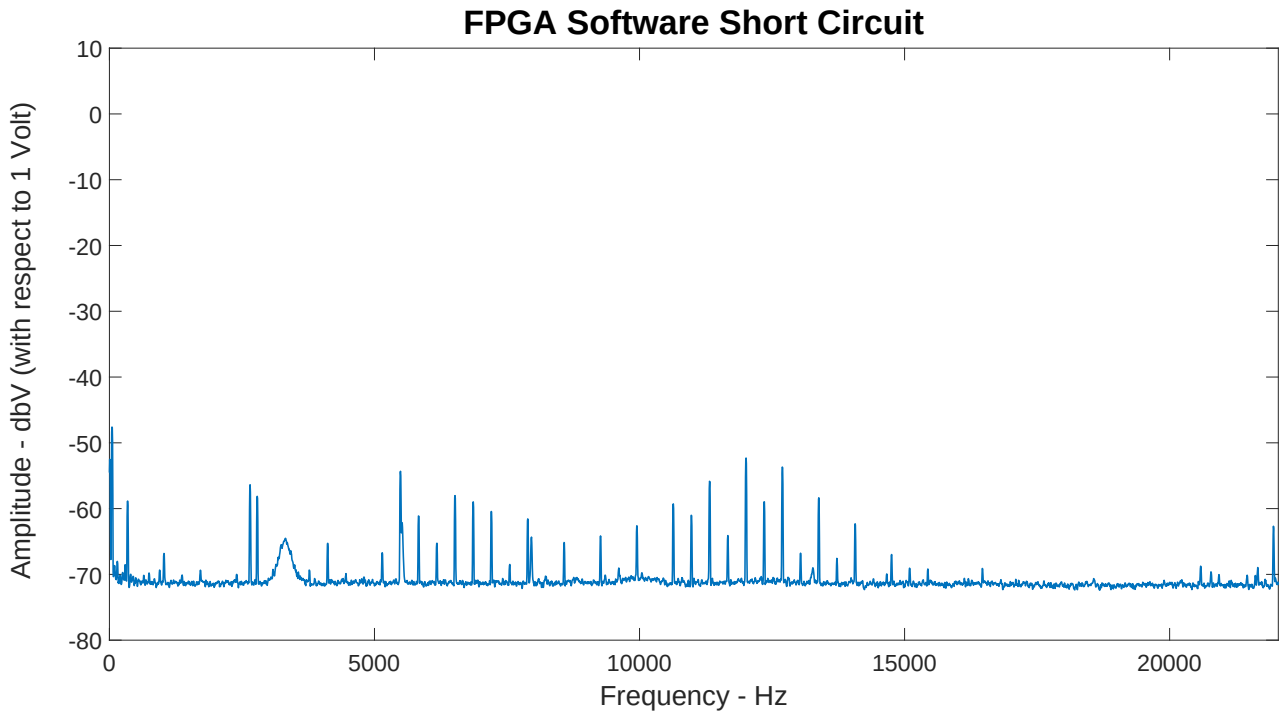


Figure 11.7: Results of test F. Output spectrum of DAC when FPGA outputs zeroes to the DAC

Finally test G was performed. In this test the analog input was shorted, like in test D, however for this test the CPU was programmed to filter the input data. Results are shown in figure 11.8. These results appear to be very similar to the previous tests, and it is not apparent that any filtering has occurred. This strongly suggests that little noise is generated in the analog input stage, in the ADC, or in the internals of the FPGA.

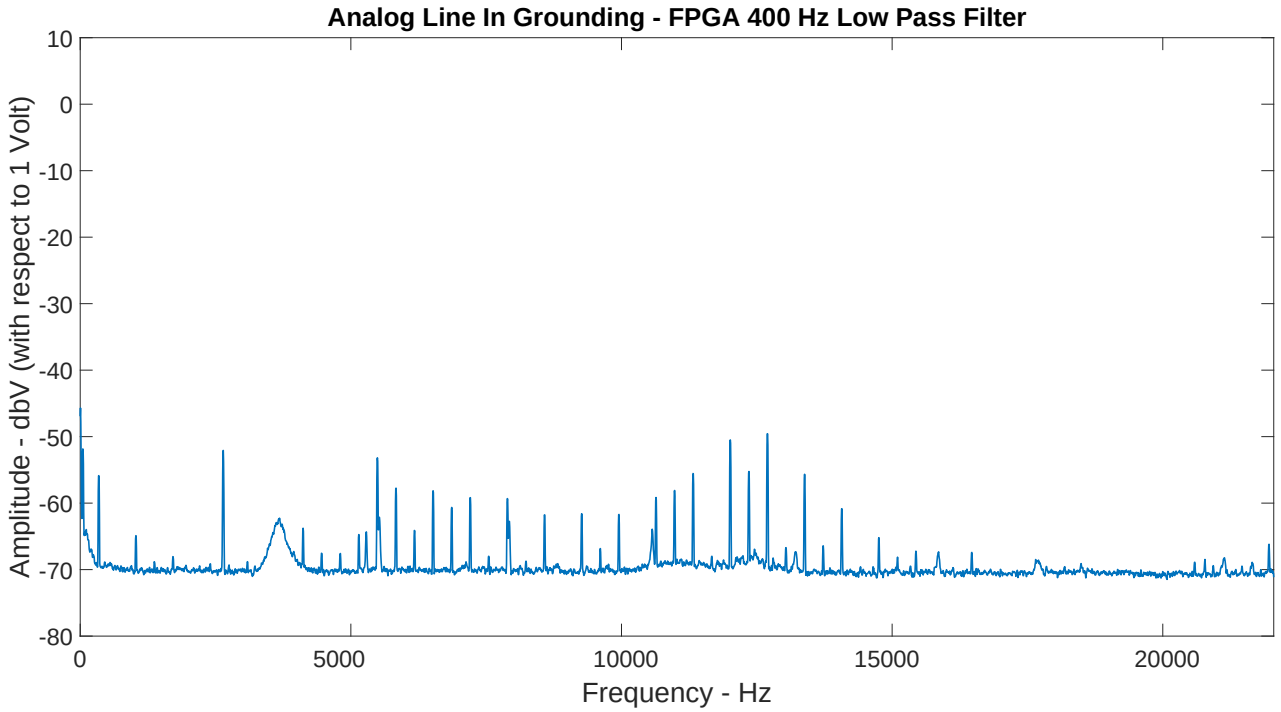


Figure 11.8: Results of test G. Output spectrum of DAC when the front end of the system is grounded, and the FPGA is programmed to low-pass filter signals above 400 Hz

While more rigorous tests are required to define the exact noise parameters of the system, at this stage, most tests indicate that the majority of noise originates in the output of the FPGA, the PSoC connections connecting the FPGA's I^2S slave to the DAC, or the DAC itself. Do note that while test B, figure 11.3, showed that the DAC produced no noise in and of itself, it is still possible that the peripheral digital system causes it to induce noise.

On another note, a sporadic "crackling" noise has been observed on the systems output stage at various, supposedly random, times. Unfortunately, it was not possible to replicate this phenomenon during testing.

11.3 Conclusion of Accept Test

The executed accept tests proves that the designed CPU predominantly works as specified in chapter 2. Additionally, the noise produced by the system has been quantified, and it is determined that the system only introduces a negligible amount of noise into the sampled audio stream. However, issues were discovered during the tests, which cannot be explained without examining the CPU using tools which are not currently available. While these issues do not affect the individual functionality of the CPU, the full potential of the designed CPU is limited by them.

Chapter 12

Discussion

In this chapter the developed system is compared to the requirements of the system, and the degree to which these have been fulfilled is investigated. In addition to this, the results from the accept test will be discussed in relation to the expected results. Lastly, some of the problems encountered both during and after the design of the system and hypothetical solutions to these, will be discussed.

Functional Requirements

In chapter 2 both functional requirements and design choices for the system were set. However, not all of the functional requirements were implemented in the final system because of time constraints. It was decided to prioritize the development of the CPU and the peripherals directly associated with data acquisition and/or data processing, higher than the other peripherals. Therefore, the last three requirements; Advanced UI, DFT block and I/O data stream were not implemented in the final system.

Accept Test

The accept test was performed using some of the modules that had not prior been tested individually. The reason for this is that the untested subsystems were either too complicated to test in a reasonable manner, e.g. the architecture of the CPU, or that the subsystems relied on other subsystems to work as intended. Performing a system test in this way can lead to some issues, since it makes it hard to determine where errors originate from.

The accept test was split into two parts; a CPU test and a noise test. The first test was of the CPU architecture, to verify that the CPU meet the product specifications. This test showed that the CPU could filter an incoming audio signal, though only if no other instructions were performed at the same time. Additionally, it was discovered that an error in the internal routing of the CPU resulted in negate operations being performed as NOT operations. However, while repairing this, the CPU ceased to function properly and it is theorized that a synthesis error is to blame which is discussed later.

The second test is of the complete system to determine the noise sources in the system, with the intention to discover if one module contributed with more noise than others. This test showed that the majority of the noise is introduced by either the output of the FPGA, the PSoC multiplexers connecting the FPGA's I^2S output to the DAC, or the DAC itself. The pattern of the noise was noted to be somewhat periodical, with "spikes" being either 340 Hz or 680 Hz apart. The significance or origin of the reoccurring "spikes" between 2.5 and 18 KHz are not currently understood, however all "spikes" are located below -50 dBV, and are negligible for the intended usage of the system.

Problems Encountered

Synthesis issues have plagued the project throughout the development period. These issues usually cause a complete system failure (i.e. the CPU does not execute code) when supposedly insignificant changes are made to the VHDL code. This problem resurfaced during the accept test, as it was discovered that the "NEGI" and NEGR" instructions were assigned to the ALU's invert opcode, instead of its negate operation. Steps were taken to try and solve the incorrect mapping, however issues caused by the synthesis prohibited these changes. At this stage it is unknown what causes these synthesis related issues, however it is believed that the Quartus 13.1 is at least partially at fault. Another possibility is that the authors inexperience with synthesizable hardware have caused them to miss important details, resulting in the VHDL in and off itself being unstable and/or sub-par.

Another related issue was a lack of insight into the internal behaviour of the synthesized circuit. This made it difficult to identify whether a problem was caused by the written VHDL, or by internal timing requirements. In hindsight it would have been advantageous to have performed gate level/analog simulations of the CPU. The data gathered from these simulations could potentially have saved many hours of development.

Chapter 13

Conclusion

The purpose of this project is to obtain knowledge and insight about embedded systems, their architecture, construction interfaces and programming. Additionally, it is also expected that some understanding of their advantages and limitations is to be gained. As a route to learn about digital and embedded systems, designing a CPU is an ideal choice.

A CPU is a fairly general-purpose system, so in order to help guide the project, as well as the design of the CPU, a specific purpose is chosen. Audio processing is chosen as the primary application of the CPU, so in essence the project is to design a DSP. To further guide the project, and to set a baseline for later tests, a set of product specifications is developed in chapter 2.

During the development of the distinct subsystems and modules, individual tests were conducted to ensure that the product specifications were met on a subsystem level, before the final accept test was performed. Any subsystem that was not easily tested on its own, was tested as part of the system test. All individual tests passed without significant errors, and the identified errors were corrected before the final accept test.

Despite the bugs and issues faced during the acceptance tests, the tests proved that the created system is able to move, process, and filter data while simultaneously acting upon interrupt request to support the real time requirements of the system. Furthermore, it performs all this without introducing a significant amount of noise into the existing audio stream. As such the system is by and large deemed functional and usable for its specified purpose, albeit with limitations.

As mentioned in the discussion, not all the functional requirements were implemented in the final system. The incompleted requirements are advanced UI, a DFT block and I/O data stream. However, these requirements are of the lowest priority while the rest of the requirements are implemented and tested. Therefore, the requirements are met to a desired satisfaction.

In conclusion, the product fulfills the majority of the design specifications, and is able to perform its intended application. In addition to a functional product, the project as a whole is also a success, in that a significant learning in the domains of digital and embedded systems were achieved. Not least the implementation of an assembler, tailored to the design of a custom instruction set architecture, upon which the successful implementation of an interrupt driven real time application system is accomplished, is in its own right viewed as a significant achievement.

Bibliography

[1] "Arithmetic logic unit (alu)," https://study.com/academy/lesson/arithmetic-logic-unit-alu-definition-design-function.html, 2018.

[2] A. S. Tanenbaum and T. Austin, *Structured Computer Organization*, 6th ed. Pearson, 2013.

[3] D. Liu, *Embedded DSP Processor Design*, 1st ed. Morgan Kaufmann, 2008.

[4] *AES5-2008 (r2013)*. Audio Engineering Society, Inc., 2008.

[5] "Altera de0 board manual," https://www.altera.com/content/dam/altera-www/global/en_US/portal/dsn/42/doc-us-dsnbk-42-5804152209-de0-user-manual.pdf, 2018.

[6] "Papilio duo board manual," http://papilio.cc/index.php?n=Papilio.PapilioDUOHardwareGuide#Overview, 2018.

[7] "Altera de0 board," http://www.terasic.com.tw/cgi-bin/page/archive.pl?Language=English&No=364, 2018.

[8] "Adafruit uda1334 i2s dac," https://www.adafruit.com/product/3678, 2018.

[9] "Cypress semiconductor cy8ckit-059 development board," http://dk.farnell.com/cypress-semiconductor/cy8ckit-059/dev-board-psoc-5-prototyping/dp/2476010, 2018.

[10] J. L. Patterson, David A.;Hennessy, *Computer Organization and Design*. Morgan Kaufmann, 2014.

[11] Altera, *LPM Quick Reference Guide*, 1996.

[12] Imperial College London, "Arithmetic Operations on Binary Numbers," https://www.doc.ic.ac.uk/~eedwards/compsys/arithmetic/index.html, 2002.

[13] U. Meyer-Baese, *Digital Signal Processing with Field Programmable Gate Arrays*. Springer Berlin Heidelberg, 2014, iSBN: 9783642453090.

[14] "The effect of coefficient quantization on the performance of a digital filter," www.allaboutcircuits.com/technical-articles/effect-coefficient-quantization-performance-digital-filter/, 12/10/2017.

[15] "Very low frequency filtering: Do it right using downsampling," https://allsignalprocessing.com/very-low-frequency-filtering/, 2015.

[16] P. Semiconductors, "I2s bus specification," https://www.sparkfun.com/datasheets/BreakoutBoards/I2SBUS.pdf, 1986.

[17] "Assembler language," https://www.ibm.com/support/knowledgecenter/SSLTBW_2.1.0/com.ibm.zos.v2r1.asma400/asmr102112.htm, 2014, last seen: 16-5-2018.

[18] "Mips32® i7200 multiprocessing system programmer's guide," https://s3-eu-west-1.amazonaws.com/downloads-mips/I7200/I7200+product+launch/MIPS_I7200_Programmers_Guide_01_20_MD01232.pdf, 20/04/2018.

[19] Quartus, "Memory initialization file," https://www.mil.ufl.edu/4712/docs/mif_help.pdf, 2018.

[20] "Arduino tutorial: Button," https://www.arduino.cc/en/Tutorial/Button, 2015.

[21] H. Zumbahlen, *Basic Linear Design, Analog Devices.* Newnes/Elsevier, 2007.

[22] Cypress Semiconductor Corporation, *PSoC 5LP CY8C58LP Family Datasheet*, 2017.

[23] Cypress, *PSoC 3, PSoC 4, and PSoC 5LP Mixed-Signal Circuit Board Layout Considerations.* Cypress Semiconductor Corporation, 2017.

[24] T. W. Kirkman jr., *Introduction to digital filters*, 2016, last seen: 18-05-2018.

[25] S. W. Smith, *The Scientist and Engineer's Guide to Digital Signal Processing*, 2nd ed. California Technical, 1999.

[26] A. V. Oppenheim, *Discrete-Time Signal Processing*, 3rd ed. Pearson, 2010.

[27] The Institute of Electrical and Electronics Engineers, "Ieee standard for floating-point arithmetic," *IEEE Std 754-2008*, pp. 1–70, 2008.

[28] Codekaizen, "Ieee 754r half floating point format," https://en.wikipedia.org/wiki/File:IEEE_754r_Half_Floating_Point_Format.svg, 2008, 03-03-18.

[29] R. E. Bryant, *Computer systems : a programmer's perspective*, 2nd ed. Pearson, 2011.

[30] C. M. Burnett, "Figures illustrating various logical shifts," https://en.wikipedia.org/wiki/Bitwise_operation, 2006.

[31] A. P. Godse and D. A. Godse, *Digital Techniques.* Technical Publications Punes, 2018, iSBN: 9788184314014.

Part V

Appendix

Appendix A

Analog to Digital Conversion

A processor is a digital device and can handle digital information. The input and output signals that the processor is supposed to process are analog signals, and thus, a way to convert between analog and digital signals is required. For these conversions an Analog-to-Digital Converter (ADC) and Digital-to-Analog Converter (DAC) is needed. The ADC and DAC will be implemented as the analog interface to the system as seen on figure A.1. When converting from an analog to a digital signal, some information is lost due to quantization. And when converting from digital to analog it can only approximate of the original signal. This introduces a series of aspects to consider when performing the conversion. The primary types of analog signals, that are to be processed by the system, are audio signals. Focusing on audio signals narrows the requirements of the technical specifications of the ADC and DAC.

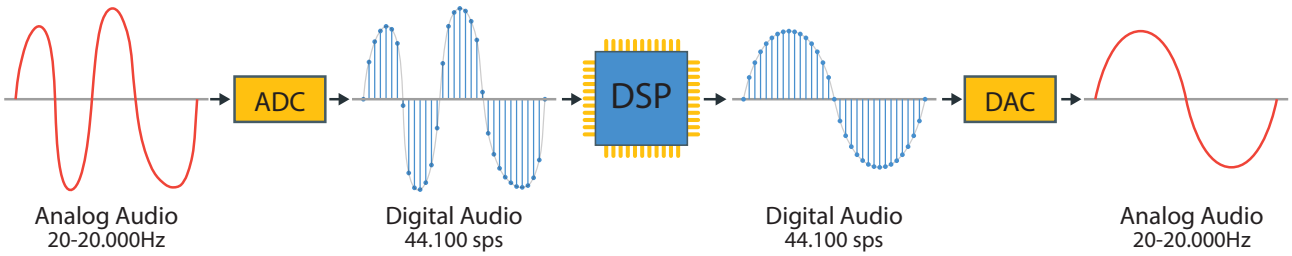


Figure A.1: Illustration of the use of DAC's and ADC's for the system.

For transferring the digital data between the A/D converters (ADC and/or DAC) and the CPU, a bus is required. To avoid having a wire for each bit, a serial bus can be used. A serial bus sends the bits sequentially and thus will have to send multiple bits through one wire per sample. For sound specifically, a common transfer protocol is the I^2S-bus. The I^2S bus requires at least three connections: Continuous serial clock (SCK), word select (WS) and serial data(SD), as seen on figure

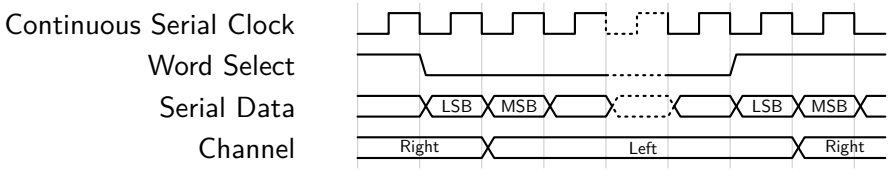


Figure A.2: I^2S timing diagram

The SCK is for controlling the bit-rate. The WS is for selecting one of the two channels and is commonly used for stereo audio. The SD is used to send the data as a bit-stream with the most significant bit (MSB) first. Since the most significant bit is sent first, the sender does not have to

consider the bit-depth of the receiver. The receiver can simply just take in the MSB and as many bits as it can handle. Then cut off the rest of the bits, thereby truncating the signal. If it can handle more bits than is sent, it will simply assume the rest of the bits are zero [16].

A.1 Sampling

To convert an analog signal to a digital signal, the analog signal is sampled at a constant rate called the sample-rate. The result is a sequence of samples, a discrete-time signal. When the amplitude of each of these samples is discrete, the signal is a digital signal. The primary properties of the digital signal are the sample-rate and the bit-depth. The bit-depth is the number of bits used to represent the amplitude of the signal. With discrete time and amplitude, an analog signal can be approximated as a digital signal as seen on figure A.3.

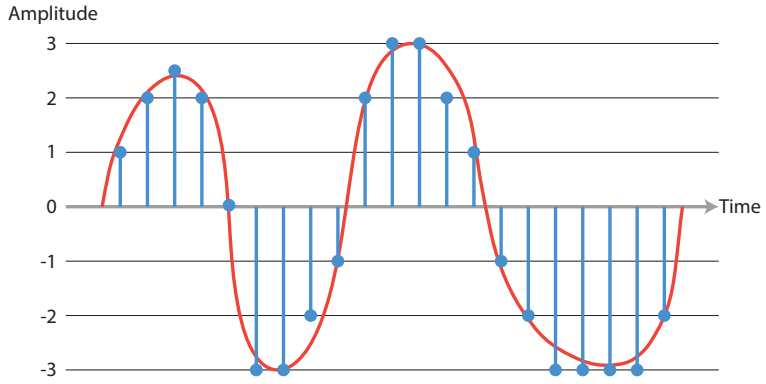


Figure A.3: analog signal (red) sampled in discrete time and amplitude(blue).

A.1.1 Sample-rate

The key factor when considering the sampling-rate is the Nyquist-Shannon sampling theory. This theory states that the sampling-rate must be twice that of the highest frequency wished to detect in a signal. Thus, to represent the exact maximum audible frequency (20 kHz) a sampling rate of at least 40 kHz is necessary.

A.1.2 Bit-depth

A higher bit-depth can translate to a higher amplitude resolution. One of the most significant effects of higher bit-depth is the effect of quantization. Quantization is the process of sampling the continuous analog signal to a discrete digital signal. The difference of the analog and digital signal introduced by quantization, is called the quantization error. Quantization error is present as noise in the digital signal. The noise introduced by the quantization error has a magnitude equal to the LSB. Thus, noise will be reduced by increasing the bit-depth.

Appendix B

Introduction to Digital Signal Processing

In signal processing it is often desired to modify or alter the characteristics of some signal, either in the time or the frequency domain. This is typically done by implementing digital filters, the most common of which are called linear time-invariant (LTI) filters.

In the last few decades, digital filters have begun to replace the use of traditional analog filters due to the several advantages they offer. Some of these advantages are *programmability*, *stability* and *versatility*. In contrast to an analog filter, a digital filter can be programmed, which means that the filter can easily be changed with software and thus does not need a change in hardware if a different characteristic is desired. Similarly, because the characteristics of a digital filter depends only on data stored in memory, it does not suffer from the dependencies of time and temperature known from analog filters. This greatly increases the stability of the filter. Lastly it is possible to obtain very accurate results from digital filters when applying either low or high frequencies, where analog filters tend to get inaccurate when handling very low frequencies. [24]

Digital filters work by applying mathematical operations on discrete samples, usually in the time domain, of some signal. The theory behind sampling can be read in section A.1.

LTI filters can be roughly grouped into two types depending on their impulse response, namely Finite Impulse Response (FIR) or Infinite Impulse Response (IIR). [25, p.11]

B.1 FIR Filters

The FIR filter, also known as a non-recursive filter, generates an output based on one or more earlier samples of a signal. The output of an FIR filter with length L, or order $N = L - 1$, from a series of input samples $x[n]$ is given by a finite sum (a convolution), namely:

$$y[n] = \sum_{k=0}^{L-1} f\,[k] \cdot x\,[n-k]$$

where $f[k]$ are the filter coefficients, also known as the filters impulse response. [13]

It is generally assumed that sampling starts at $t = 0$ with sample $x[0]$ and thus for the cases when $k > n$ the samples are not defined and are usually taken as 0. [24, p.4]

To give an example of an FIR filter, one of the simplest FIR filters, the *Moving Average* (MA) filter, will be described. As the name suggests, the output of the MA filter is the average of some

number of earlier samples. This corresponds to the case where all the filter coefficients are equal to the reciprocal of the length of the filter, which can be expressed as:

$$y[n] = \frac{1}{L} \cdot \sum_{k=0}^{L-1} x[n-k]$$

It is evident that if the samples suddenly change greatly in value, the resulting output will stay somewhat stable and thus the MA filter is in fact a simple low pass filter. [25]

When designing FIR filters, the simplest method is the *window method*. In broad terms this method consists of acquiring an ideal impulse response for the type of filter desired, then truncating it using so called *windows*. This is done by convoluting the ideal impulse response with the Fourier transform of the window. Through history, distinct types of windows have been devised, and each type results in distinct characteristics of the filter. [26]

B.2 IIR Filters

The IIR filter generates an output based not only on the previous samples but the previous outputs as well which is why it is also known as a recursive filter. As the name suggests the impulse response of an IIR is infinite, and this is because every output contributes to the next and thus can the output never reach zero after an input different from zero. The output of an IIR filter with length L, or order $N = L - 1$, from a series of input samples $x[n]$ is given by two finite sums, namely:

$$y\left[n\right] = \sum_{l=0}^{L-1} b\left[l\right] \cdot x\left[n-l\right] \; + \sum_{l=0}^{L-1} a\left[l\right] \cdot y\left[n-l\right] \tag{B.1}$$

where $b[l]$ are the feedforward coefficients, $a[l]$ are the feedback coefficients and L is the length of the filter. [13] Due to the use of feedback in the IIR filter, it is important to be aware of the risk of accumulating errors such as truncation errors. Therefore, this must be kept in mind when designing an IIR filter which is to be used with fixed point numbers.

A simple example of an application with an IIR filter is a simple "Lossy Integrator" used to smooth a noisy signal. [13] A first order implementation of such a filter can be written as:

$$y\left[n+1\right] = \frac{3}{4} \cdot y\left[n\right] + x\left[n\right] \tag{B.2}$$

The next output thus depends on the previous input and output samples as can be seen in equation (B.2). The impulse response of the filter characterized by equation (B.2) can be seen in figure B.1, and it is evident that to obtain the same functionality from an FIR filter, it would need to have a much larger order.
When designing IIR filters, one common method is deriving the discrete-time IIR filter coefficients based on some ideal continuous-time filter. This can be done by using a transform on the impulse response or the system function of the continuous-time filter. As with the FIR filter, there are some well-defined ideal filters to choose from, depending on the application, such as Chebyshev, Butterworth or elliptic filters.

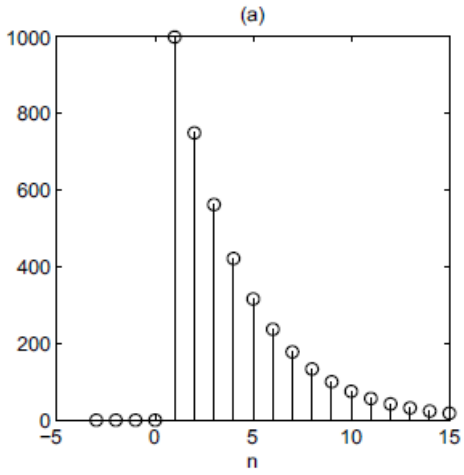


Figure B.1: Simulation of lossy integrator with a = 3/4. (a) Impulse response for x[n] = 1000[n]. [DSP with FPGA]

Appendix C

Fixed-Point vs. Floating-Point

In contrast to conventional mathematics, where the base 10 system is primarily used, digital circuitry almost exclusively uses the base 2 number system (binary) to represent and convey information. When dealing with numerical information, there exists two fundamentally different ways to represent numbers, using either fixed-point or floating-point representations.

C.0.1 Fixed-point

In a fixed-point representation, the number of digits to the left or right of the radix point (commonly known as the decimal point in base 10) is well defined. In binary the least significant bit (LSB), i.e. the digit furthest to the right, has a relative weight of 1, and every subsequent digit has a weight of twice the previous digit. Thus, an N-bit binary number, X, will have an unsigned fixed-point value given by:

$$X = 2^{\text{bias}} \sum_{n=0}^{N-1} x_n 2^n \tag{C.1}$$

where x_n is the nth binary digit of X, and the bias is a predetermined value which dictates the placement of the radix point [13, sec. 2.2]. Using the principles of equation C.1, table C.2 demonstrates how an unsigned fixed-point binary number is converted to base 10:

Table C.1: Shows the relative and absolute values of each digit of an unsigned fixed-point binary number (10111.011), as well as how it is converted to its base 10 representation.

Relative Value:	128	64	32	16	8		4	2	1	
Absolute Value:	16	8	4	2	1		$\frac{1}{2}$	$\frac{1}{4}$	$\frac{1}{8}$	
Binary Number:	**1**	**0**	**1**	**1**	**1**	.	**0**	**1**	**1**	
Base 10 representation:	16	+0	+4	+2	+1		0	$+\frac{1}{4}$	$+\frac{1}{8}$	= 23.375

Since digital circuits can only represent two states, it is impossible to represent the radix point. Thus, unsigned fixed-points numbers are only represented in circuitry according to their relative value. This conveniently corresponds to their numerical value when the radix is placed to the right of the LSB. While it is possible to have a predetermined radix point, this is redundant for computations, as the placement of the radix point only acts as a scaling factor. Thus, when discussing fixed-point binary numbers in a computational context, it is generally assumed that fixed-point integers, where the radix is directly to the right of the LSB, are being used. Therefore, going forward, when referring to integers it conforms to the above definition.

The above only deals with unsigned representation of integer numbers. To represent positive and negative numbers, a signed number representation is required. A variety of signed number representations exist, the most common of which is the two's complement representation [13, sec. 2.2]. In an N-bit two's complement integer, the values from 0 up to $2^{N-1} - 1$ function identically to unsigned integers. However, from 2^{N-1} to 2^N the values will count from a minimum of -2^N to -1 [13, sec. 2.2]. For example, to convert 1110 from two's complement to a decimal representation, one would first convert the number using unsigned principles, so 1110 becomes 14. Since 14 is larger than 2^{N-1} ($2^3 = 8$), the number must be negative. Since the number is negative 14 must be subtracted with 2^N, or 16, resulting in -2. The two's complement system has gained widespread popularity due to the fact that two's complement adheres to many of the same arithmetic rules as unsigned numbers [13, sec. 2.2].

C.0.2 Floating-point

Contrary to fixed-point numbers, where the radix point is placed at a fixed position, the radix point in floating-point numbers "float" around. For example, a 16-bit ("Half-precision") floating-point as defined by the IEEE 754-2008[27] standard has the following format:

- Sign (s) : 1 bit
- Exponent (e): 5 bits
- Mantissa (m): 10 bits

Figure C.1 illustrates the format:

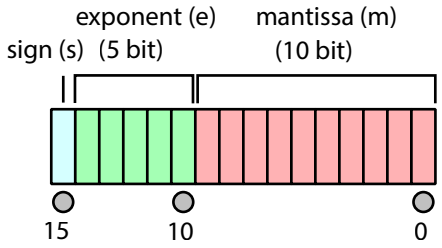


Figure C.1: Illustration of how a 16-bit IEEE 754 floating-point would be laid out in memory [28]

As seen in the figure C.1, a floating-point consists of three distinct components:

1. The sign bit decides the sign of the number, 0 corresponds to '+' and 1 to '-'
2. The exponent
3. The mantissa

Thus floating-points closely resemble scientific notation, this is further demonstrated by the equation used for converting a float to its standard binary representation: [13]

$$\text{binary} = (-1)^s \cdot 1.m \cdot 2^{e-\text{bias}} \tag{C.2}$$

where the bias is:

$$\text{bias} = 2^{\text{bits In Exponent}-1} - 1 \tag{C.3}$$

As an example, it is shown how to convert a 16-bit binary number, 0101100011110101, to base 10 using the IEEE 754 standard.

First, the number is split into its three components, the sign bit (s), the exponent (e), and the mantissa (m):

$$\underbrace{0}_{\text{s}}\ \underbrace{10110}_{\text{e}}\ \underbrace{0011110101}_{\text{m}}$$

To simplify calculations, the 'normal' binary representation of the number is found using equation C.2. The equation is split into three separate components, A, B and C:

$$\text{binary} = \underbrace{(-1)^s}_{\text{A}} \cdot \underbrace{1.m}_{\text{B}} \cdot \underbrace{2^{e-\text{bias}}}_{\text{C}}$$

A, B and C are then calculated:

- A: $(-1)^s = -1^0 = 1$
- B: $1.m = 1.0011110101_2$
- C: $2^{e-\text{bias}} = 2^{10110_2 - 01111_2} = 2^{22_{10} - 15_{10}} = 2^7_{10}$

Where the bias $= 2^{5-1} - 1 = 15$

The product of A, B and C is calculated:

$$\text{A} \cdot \text{B} \cdot \text{C} = 1.0011110101_2 \cdot 2^7_{10} = 10011110.101_2$$

In the above equation, the radix point of $1.m$ is simply moved seven points to the right.

Finally, the resultant product is converted to base 10:

Table C.2: Conversion from 10011110.101_2 to base 10

Value:	128	64	32	16	8	4	2	1		$\frac{1}{2}$	$\frac{1}{4}$	$\frac{1}{8}$	
Binary:	**1**	**0**	**0**	**1**	**1**	**1**	**1**	**0**	.	**1**	**0**	**1**	
Base 10:	128	+0	+0	+16	+8	+4	+2	+0		$+\frac{1}{2}$	+0	$+\frac{1}{8}$	= 158.625

It should be noted that the floating-point format has five different coding types to accommodate zero, infinity, Not a Number, as well as a special denormalized mode, they are as follows: [13, sec. 2.2]

Table C.3: Five possible floating-point coding types

Exponent e	Mantissa m	Meaning
All-zeroes	All-Zeroes	± 0
All-Ones	All-Zeroes	$\pm\infty$
All-Ones	Nonzero	NaN
All-zeroes	Nonzero	Denormalized: $(-1)^s \cdot 0.m \cdot 2^{e-\text{bias}}$
1<e <E_{max}	Any	Normalized:$(-1)^s \cdot 1.m \cdot 2^{e-\text{bias}}$

Due to the way floating-point are implemented, they exhibit very different characteristics compared to a fixed-points implementation. First of all, they allow a very high dynamic range, only being limited by the size of the exponent. For the IEEE 16-bit floating-point where five bits are dedicated to the exponent, it is possible to achieve a dynamic range of 240 dB. In comparison, a 16-bit unsigned integer only has a dynamic range of about 96 dB. However, to achieve this massive dynamic range, floating-points sacrifice resolution. In an integer number representation, the resolution is fixed across the entire range of possible values to one. Thus, the difference between smallest possible value and the next smallest value is one, and the same applies for the largest and the second largest value. For floating-points, the resolution is not uniform across the entire range of possible values. In fact, the resolution is initially very high for the smallest representable values, and then it gradually worsens as the value of the number increases. For example, in IEEE 16-bit floating-point the highest achievable resolution between two numbers is 2^{-24} (applies for values less than 2^{-13}). On the other hand, for values between 2^{14} and 2^{15}, 16 is the highest achievable resolution, therefore $2^{14} + 1$ would still be represented as 2^{14} as there is no way to represent $2^{14} + 1$ in IEEE 16-bit floating-point. While this decrease in absolute resolution might initially seem poor, it turns out it is often a good compromise as the relative value of large numbers is not as affected by a resolution of 16, compared to a smaller number. One could say that fixed-point integers have similar absolute resolution, while floating-point numbers have a similar relative resolution. Figure C.2 illustrates the resolution difference between signed integers and a 6-bit float:

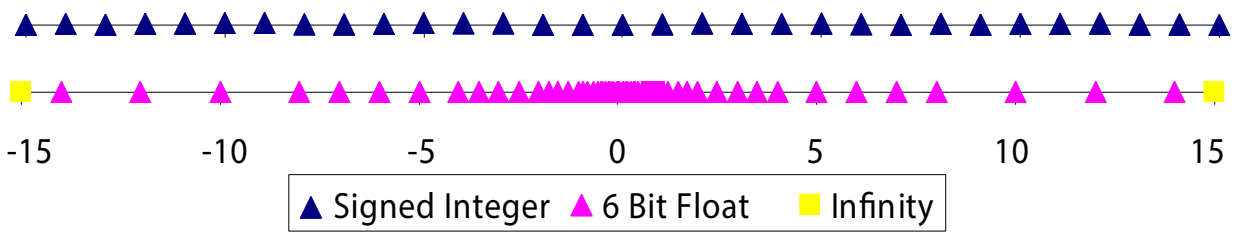


Figure C.2: Resolution comparison between a signed integer and a 6-bit floating-point in IEEE format (with 3-bit exponent) [29]

Appendix D

ALU User Manual

This user manual is for the ALU component of the CPU. It contains a table of in- and outputs, a description of each of its functions and how to use them.

D.1 Introduction

The ALU is used in the processor to perform both simple arithmetic and bitwise operations on binary numbers.

In table D.1 can the list of the ALU's inputs and outputs be found.

Table D.1: List of the ALU's inputs and outputs

Port Name	Description	Comments
OperandA	Data input	This port is 16 bits wide.
OperandB	Data input	This port is 16 bits wide.
clk	Clock input	Input port for the clock to the ALU.
Operation	Data input	This port is 16 bits wide. Contains the opcode used to determine the operation of the ALU
Flags	Data output	This port is 5 bits wide. Contains information about flags raised during operation. Each bit represents a different flag. From msb to lsb; Overflow, Signed, Zero, Parity and Carry.
Result	Data output	This port is 16 bits wide.

D.2 Operations of the ALU

In this section, each of the operations of the ALU will be described A list of featured operations and their operation codes can be found in table D.2.

Table D.2: Table with available operations in the ALU

Operation Name	Description	Opcode
ADD	Adds the two input operands	0x0001
SUB	Subtracts the two operands	0x0002
MUL	Multiplies the two operands,	0x0003
AND	ANDs the two operands	0x0004
OR	ORs the two operands	0x0005
XOR	XORs the two operands	0x0006
NEA	NEGATES operand A	0x0007
NEB	NEGATES operand B	0x0008
NOA	NOT's operand A	0x0009
NOB	NOT's operand B	0x000A
LSL	Logic Shift Left	0x000B
LSR	Logic Shift Right	0x000C
ASR	Arithmetic Shift right	0x000D
PSA	Passes operand A	0x000E
PSB	Passes operand B	0x000F
ICA	Increments operand A	0x0010
ICB	Increments operand B	0x0011
NOP	Does nothing, does not change flags	0x0012

The following subsections will contain a short description of the operations in the ALU.

D.2.1 ADD

The ADD operation adds the two input operands and sends the result as an output. In addition to the result, one or several flags may be raised if certain conditions are met. The available flags and their conditions can be found in table D.6.

D.2.2 SUB

The SUB operation subtracts OperandB from OperandA and sends the result as an output. In addition to the result, one or several flags may be raised if certain conditions are met.

D.2.3 MUL

The MUL operation multiplies OperandA with OperandB. The multiplier makes signed multiplication of A and B. In addition to the result, one or several flags may be raised if certain conditions are met.

D.2.4 AND

The AND operation uses the logical AND operation which compares the bits of the two operands.

Table D.3: AND operation

Input A	Input B	Output
10010011	01010100	00010000

D.2.5 OR

The OR operation uses the logical OR operation and compares the bits of the two operands.

Table D.4: OR operation

Input A	Input B	Output
10010011	01010100	11010111

D.2.6 XOR

The XOR operation uses the XOR operation and compares the bits of the two operands. The XOR is similar to the OR operation, but when both inputs have a 1, the output will be 0 instead of 1.

Table D.5: XOR operation

Input A	Input B	Output
10010011	01010100	11000111

D.2.7 NEA and NEB

The NEA and NEB operations negates operandA and operandB respectively. Negating corresponds to multiplying the number by -1.

D.2.8 NOA and NOB

The NOA and NOB operations uses the logical NOT operation on operandA and operandB respectively.

D.2.9 LSL and LSR

LSL and LSR are logical shifts. A logical shift is a bitwise operation that shifts all the bits of its operand. A logical shift does not preserve a number's sign bit, thus every bit in the operand is simply moved a given number of bit positions, and the vacant bit-positions are filled, in this case with zeros. LSL is a logical shift left and LSR is a logical shift right.

The operations shift the bits in operandA the amount of operandB

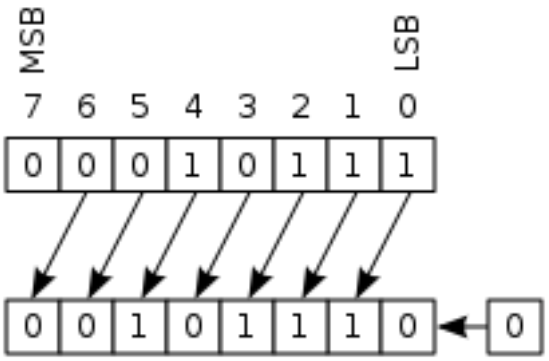


Figure D.1: Logical left shift one bit. [30]

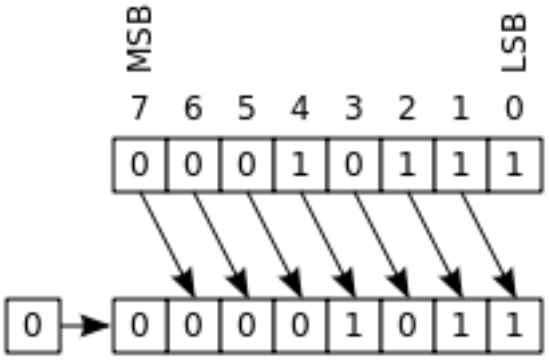


Figure D.2: Logical right shift one bit. [30]

D.2.10 ASR

ASR is an arithmetical right shift. It is a bitwise operation that shifts all the bits of its operand; every bit in the operand is moved a given number of bit positions, and the vacant bit-positions are filled in. Instead of being filled with all 0s, as in logical shift, when shifting to the right, the leftmost bit is replicated to fill in all the vacant positions, this is also known as sign extension.

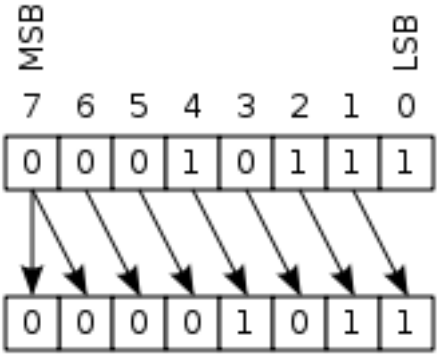


Figure D.3: Arithmetically right shift one bit. [30]

D.2.11 PSA and PSB

The PSA and PSB operations lets operandA and operandB respectively pass directly through the ALU. Conditional flags get raised if their conditions are met.

D.2.12 ICA and ICB

The ICA and ICB operations increments operandA and operandB respectively by one. Conditional flags get raised if their conditions are met.

D.2.13 NOP

The NOP operation does nothing. It is just a operation to make sure the CPU does nothing. Conditional flags are not changed during the NOP operation.

D.2.14 Flags

In addition to the arithmetic and logical operations the ALU also supports five control flags. A list and description of the flags can be seen in table D.6.

Table D.6: Table containing a list and description of the condition flags available in the ALU

Flag type	Description
Overflow	The overflow condition indicates that a number is too big or too small to be contained within the available data type, when the number is interpreted as a signed number. For addition with signed numbers in 2's compliment, overflow occurs when adding two numbers with same sign results in a number with a different sign. For subtraction with signed numbers in 2's compliment, overflow occurs when subtracting two numbers with different signs results in a number with the same sign as the subtrahend. [12]
Sign	The sign condition indicates that the MSB of the result is '1' and thus the number is negative when interpreted as a signed number.
Zero	The zero condition indicates that the result is equal to zero.
Parity	The parity condition indicates if the amount of '1's in the result is odd or even. For an odd amount of '1's, the parity flag is set high.
Carry	The carry condition indicates that a number is too big or too small to be contained within the available data type, when the number is interpreted as an unsigned number. [12]

Appendix E

Binary-Coded Decimal Conversion by the Double Dabble Method

To convert the 16-bit binary number to Binary-coded decimal (BCD) the double dabble method is used [31]. Why this algorithm works will not be explained but the method is as follows for n-bit word:

The bits needed for the BCD representation is given by:

$$\text{required BCD bits} = n + 4 \cdot \text{ceiling}(\frac{n}{3}) \quad \text{[bits]} \quad \text{(E.1)}$$

Where:

n is bits in word that is converted from [bits]

Then the original binary number is shifted left into a buffer that the size calculated with the formula above. For each 4-bit section of the buffer starting from the right the following check is done: If the 4-bit group has a value greater than 4, then add 3. Once it has been shifted and checked n times. the buffer now contains the BCD representation of the original number. The VHDL implementation can be seen on figure E.1. It should be noted that in the example the bits available for the BCD is only 16. While the required bits for a 16-bit word would be:

$$\text{required BCD bits for 16-bit word} = 16 + 4 \cdot \text{ceiling}(\frac{16}{3}) = 22 \quad \text{[bits]} \quad \text{(E.2)}$$

```
-------------------------------------------
-- b2bcd:
-- converts a number to binary coded decimal
-- Note that it is only possible to represent numbers up to
-- 9999 with 16 available output BITS
-------------------------------------------
b2bcd : PROCESS (binary, clr)
   VARIABLE temp : std_logic_vector(31 DOWNTO 0);
BEGIN
   temp := x"00000000";
   IF clr = '0' THEN
      temp(18 DOWNTO 3) := binary;

      FOR i IN 0 TO 12 LOOP
         IF temp(19 DOWNTO 16) > 4 THEN
            temp(19 DOWNTO 16) := temp(19 DOWNTO 16) + 3;
         END IF;
         IF temp(23 DOWNTO 20) > 4 THEN
            temp(23 DOWNTO 20) := temp(23 DOWNTO 20) + 3;
         END IF;
         IF temp(27 DOWNTO 24) > 4 THEN
            temp(27 DOWNTO 24) := temp(27 DOWNTO 24) + 3;
         END IF;
         IF temp(31 DOWNTO 28) > 4 THEN
            temp(31 DOWNTO 28) := temp(31 DOWNTO 28) + 3;
         END IF;
         temp(31 DOWNTO 1) := temp(30 DOWNTO 0);
      END LOOP;
   END IF;
   bcd <= temp(31 DOWNTO 16);
END PROCESS b2bcd;
```

Figure E.1: Implementation of the double dabble method.

Appendix F

Assembly Manual

To help programmers program the CPU an Assembler manual is made available.

F.1 Hardware Overview

F.1.1 Instruction Timing

The clock cycle of the CPU is variable from 10kHz to 20 MHz.

All instructions take only one clock cycle in execution time.

F.1.2 Registers

There are 28 16-bit general purpose registers, $r1 through $r28.

Three special registers, the zero register which contains a zero ($zero) (though it is possible to overwrite this value), the program counter ($pc), and the stack pointer ($sp) is accessible in the assembler.

There are five flags zero (Z), overflow (O), carry (C), parity (P), and signed (S). The flags can be accessed with the GETFLAG and SETFLAG commands. It is also possible to push and pop flags from the stack with PUSHFLAGS and POPFLAGS.

F.1.3 Memory Architecture

Memory is separated into two types, program memory and data memory. Program memory is used to store the program and uses 32-bit words, and therefore all program instructions must be 32 bits long.

Data memory is used to store data during program execution. The data memory uses 16-bit words. Data memory is accessed through STORE and LOAD instructions. Data memory can also be accessed through POP and PUSH, which uses the value in the stack pointer register to address the top of the stack. When pushing to the stack the stack pointer increments, while it decrements when popping the stack. Memory can be accessed using immediates, registers, or both. When a register is used to access a memory word, the value inside of the register is used for the address.

F.1.4 Peripheral Access

The I/O is controlled with LOAD and STORE instructions at specific memory addresses. I/O access is handled in the same way as data memory. The following table lists the peripherals memory addresses, their functionality, as well as their READ/WRITE permissions:

Table F.1: Peripherals memory addresses, their functionality, and their READ/WRITE per-missions

ADDRESS	R/W	PERIPHERAL	DESCRIPTION
0-8191	R/W	DRAM	Read/Write data to the DRAM
65000	W	Seven Segment Display	Data to be displayed
65001	W	Seven Segment Display	Configure the display; Last six LSB turn decimal signs on when high, the seventh LSB enable binary coded decimal when high
65002	R	Push Buttons	Button input data. Each button corresponds to their respective bit. e.g. The second push button correspond to the second LSB
65010	W	I^2S master	I^2S data to be send
65011	R	I^2S lave	Read I^2S data
65100	W	Interrupt Controller	Program Address to jump to when a button interrupt occurs
65101	W	Interrupt Controller	Button Interrupt Configuration. The tenth LSB enables the interrupt, the ninth LSB enables interrupt nesting
65102	W	Interrupt Controller	Program Address to jump to when an I^2S slave interrupt occurs
65103	W	Interrupt Controller	I^2S slave Interrupt Configuration. The tenth LSB enables the interrupt, the ninth LSB enables interrupt nesting
65104	W	Interrupt Controller	Program Address to jump to when an I^2S master interrupt occurs
65105	W	Interrupt Controller	I^2S Master Interrupt Configuration. The tenth LSB enables the interrupt, the ninth LSB enables interrupt nesting

F.2 Assembler Overview

F.2.1 Use of Assembler

The assembler can either be run from a terminal or as a standalone executable. The assembler takes three arguments: the input assembly file, the .mif file to be created, and the amount of data memory words the CPU has been configured for.

The input file must exist for the assembler to run, however the output file can be created during assembly. The assembler will prompt if it should create the output file. The data memory size argument can be left blank, in that case the assembler will assume a memory size of 1024 words.

The assembler will accept any ASCII formatted text file, e.g. .txt or .asm. For output, .mif is recommended since the assembler outputs the binary code in this format.

The assembler will also output debugging information in the console window, to check for any possible mistakes during assembly.

F.2.2 Assembler Directives

There is no support for placing instructions at specific memory addresses. All instructions will be placed in the order they are written.

Labels

Jump instructions such as "JMPEQ" needs labels to know where to jump to. The label after an opcode must begin with hash "#". Labels always represents the next opcode in the assembly code. This indicates that if a jump is made to a label the jump will be written to the next opcode under the label instead of the label itself.

Comments

The assembler supports line comments. Comments are initiated by a double slash "//".

F.2.3 Immediate Values

Immediate values can be given as a number or a label. Labels as immediates are given the same value as the line number of the next opcode written in the assembly code. Immediates cannot exceed 16-bit values. All immediates are unsigned by default, to make any immediate signed simply set a negative sign "-" in front of the immediate. Since immediates can both be unsigned and signed the number 65535 has the same bit-value as the number -1. There is no error handling for writing overflow and underflow immediates in the assembler.

F.3 Instruction Set

The following tables will show the correct syntax of the different instructions and all other need to know information about the assembler instructions. Here is a list that explains the information that

is given in the following section.

- Syntax - Here is the syntax of what is the correct way to write the instructions in the assembler.
- Description - Here is a brief description of what the instruction does.
- Operation - This is a mathematical explanation of the instructions function.
- O - The Overflow flag.
- S - The signed flag.
- Z - The zero flag
- P - The parity flag (even number of bits)
- C - The carry flag
- Encoding [31:0] - The encoding of what the assembler outputs from 31 down to 0 bits to the CPU (a total of 32 bits). the sign 'o' stands for operation (opcode), 'r' stands for RS1 (register source 1), 's' stands for RS2 (register source 2), 'd' stands for RD (register destination), and 'i' stands for immediate (numeric value). '0' is just a zero in the output.

F.3.1 Memory Word Access

Syntax	Description	Operation	Encoding [31:0]
LOAD $RD [$RS1+IMM]	Load indirect, plus direct value	RD = mem[RS1+IMM]	011110.ddddd.rrrrr.iiiiiiiiiiiiiiii
LOAD $RD [$RS1]	Load indirect	RD = mem[RS1]	011110.ddddd.rrrrr.0000000000000000
LOAD $RD [IMM]	Load direct	RD = mem[IMM]	011110.ddddd.11101.iiiiiiiiiiiiiiii
STORE $RS1 [$RD+IMM]	Store indirect, plus direct value	mem[RD+IMM] = RS1	011111.rrrrr.ddddd.iiiiiiiiiiiiiiii
STORE $RS1 [$RD]	Store indirect	mem[RD] = RS1	011111.rrrrr.ddddd.0000000000000000
STORE $RS1 [IMM]	Store direct	mem[IMM] = RS1	011111.11101.ddddd.iiiiiiiiiiiiiiii

F.3.2 I/O Access

Syntax	Description	Operation	Encoding [31:0]
STORE $RS1 [65000]	Stores the value in RS1 to the sevenseg-driver	sevenseg = RS1	011111.11101.rrrrr.iiiiiiiiiiiiiiii
STORE $RS1 [65001]	Stores the value in RS1 to the sevenseg-control		011111.11101.rrrrr.iiiiiiiiiiiiiiii
LOAD $RD [65002]	Loads the value from the buttondriver into RD	RD= buttondriver	011110.ddddd.11101.iiiiiiiiiiiiiiii

F.3.3 Register Access

Syntax	Description	Operation	Encoding [31:0]
MOV $RS1 $RD	Moves RS1 into RD	RD = RS1	011011.rrrrr.ddddd.00000.00000.00000
MOVI $RD IMM	Moves an immediate into RD	RD = IMM	011101.00000.ddddd.iiiiiiiiiiiiiiii

F.3.4 Logical Operations

Syntax	Description	Operation	Encoding [31:0]
ANDR $RS1 $RS2 $RD	AND register	RD = RS1 & RS2	000101.rrrrr.ddddd.sssss.00000.000000
ORR $RS1 $RS2 $RD	OR register	RD = RS1 | RS2	000110.rrrrr.ddddd.sssss.00000.000000
XORR $RS1 $RS2 $RD	XOR register	RD = RS1 ^RS2	000111.rrrrr.ddddd.sssss.00000.000000
ANDI $RS1 IMM $RD	AND immediate	RD = RS1 & IMM	010001.rrrrr.ddddd.iiiiiiiiiiiiiiii
ORI $RS1 IMM $RD	OR immediate	RD = RS1 | IMM	010010.rrrrr.ddddd.iiiiiiiiiiiiiiii
XORI $RS1 IMM $RD	XOR immediate	RD = RS1 ^IMM	010011.rrrrr.ddddd.iiiiiiiiiiiiiiii

F.3.5 Arithmetic Operations

Syntax	Description	Operation	Encoding [31:0]
ADDR $RS1 $RS2 $RD	Adds two registers	RD = RS1+RS2	000001.rrrrr.ddddd.sssss.00000000000
SUBR $RS1 $RS2 $RD	Subtracts two registers	RD = RS1 - RS2	000011.rrrrr.ddddd.sssss.00000000000
NEGR $RS1 $RD	Negates register	RD = -RS1	000100.rrrrr.ddddd.0000000000000000
MULT $RS1 $RS2 $RD	Multiplies two registers	RD = RS1 x RS2	001000.rrrrr.ddddd.sssss.00000000000
CMP $RS1 $RS2	Compares two registers	RS1 - RS2	011010.rrrrr.00000.sssss.00000000000
ADDI $RS1 IMM $RD	Adds a register and an immediate	RD = RS1 + IMM	001101.rrrrr.ddddd.iiiiiiiiiiiiiiii
SUBI $RS1 IMM $RD	Subtracts a register and an immediate	RD = RS1 - IMM	001111.rrrrr.ddddd.iiiiiiiiiiiiiiii
NEGI IMM $RD	Negates immediate	RD = -IMM	010000.00000.ddddd.iiiiiiiiiiiiiiii
MULTI $RS1 IMM $RD	Multiplies a register and an immediate	RD = RS1 x IMM	010100.rrrrr.ddddd.iiiiiiiiiiiiiiii
CMPI $RS1 IMM	Compares a register and an immediate	RS1 - IMM	011100.rrrrr.00000.iiiiiiiiiiiiiiii

F.3.6 Shift Operations

Syntax	Description	Operation	Encoding [31:0]
LSLR $RS1 $RS2 $RD	RS1 logical shift left with the value in RS2	RD = RS1 <<RS2	001001.rrrrr.ddddd.sssss.00000.000000
LSRR $RS1 $RS2 $RD	RS1 logical shift right with the value in RS2	RD = RS1 >>RS2	001010.rrrrr.ddddd.sssss.00000.000000
RASR $RS1 $RS2 $RD	RS1 arithmetical right shift with the value in RS2	RD = RS1 >>RS2 (With sign extension)	001011.rrrrr.ddddd.sssss.00000.000000
LSLI $RS1 IMM $RD	RS1 logical shift left with the immediate	RD = RS1 <<IMM	010101.rrrrr.ddddd.iiiiiiiiiiiiiiii
LSRI $RS1 IMM $RD	RS1 logical shift right with the immediate	RD = RS1 >>IMM	010110.rrrrr.ddddd.iiiiiiiiiiiiiiii
RASI $RS1 IMM $RD	RS1 arithmetical right shift with the immediate	RD = RS1 >>IMM (With sign extension)	010111.rrrrr.ddddd.iiiiiiiiiiiiiiii

F.3.7 Stack Control

Syntax	Description	Operation	Encoding [31:0]
PUSH $RD	Pushes the PC to the stack	RD = PC	100001.00000.ddddd.00000.00000.000000
POP $RS1	POPs the pc from the stack	PC = RS1	100000.00000.00000.rrrrr.00000.000000
PUSHFLAGS	Pushes the flags to the stack	STACK = Flags	110100.00000.00000.00000.000000
POPFLAGS	POPs the flags from the stack	Flags = STACK	110101.00000.00000.00000.000000

F.3.8 FIR Instructions

Syntax	Description	Operation	Encoding [31:0]
FIRCOR $RS1	Stores the value of RS1 as a filter coefficient	Filtercoef = RS1	100111.rrrrr.00000.00000.00000.000000
FIRCOI IMM	Store the value of an immediate as a filter coefficient	Filtercoef = IMM	101000.00000.00000.iiiiiiiiiiiiiiii
FIRSAR $RS1 $RD	Stores RS1 value as filter sample and stores the filters output in RD	Filtercoef = RS1, RD = Filterout	101010.rrrrr.ddddd.00000.00000.000000
FIRSAI IMM $RD	Stores an immediate as filter sample and store the filters output in RD	Filtercoef = IMM, RD = Filterout	101011.00000.ddddd.iiiiiiiiiiiiiiii
FIRCORESET	Resets the filter		101001.00000.00000.00000.00000.000000

F.3.9 Flag Control

Syntax	Description	Operation	Encoding [31:0]
GETFLAG $RD	Saves the values of the flags in register	RD = Flags	110010.00000.ddddd.00000.00000.000000
SETFLAG IMM	Set flags with an immediate	Flags = immediate	110011.00000.00000.iiiiiiiiiiiiiiii

F.3.10 Program Flow Control

Syntax	Description	Operation	Endcoding [31:0]
JMP #Label	Jumps to the targeted label	pc = Label +1	100010.rrrrr.00000.00000.00000.000000
JMPNQ #Label	Jumps if last operation negative to the targeted	pc = Label +1	100101.rrrrr.00000.00000.00000.000000
JMPLE #Label	Jump if last operation was not zero to the targeted label	pc = Label +1	100100.rrrrr.00000.00000.00000.000000
JMPEQ #Label	Jump if last operation was zero to the targeted label	pc = Label +1	100011.rrrrr.00000.00000.00000.000000
JMPPA #Label	Jump if parity flag is set to the targeted label	pc = Label +1	110000.rrrrr.00000.00000.00000.000000

Syntax	Description	Operation	Endcoding [31:0]
JMPR $RS1	Jump to a value inside a register	pc = RS1	101100.rrrrr.00000.00000.00000.000000
JMPNQR $RS1	Jumps if last operation was negative to a value inside a register	pc = RS1	101101.rrrrr.00000.00000.00000.000000
JMPLER $RS1	Jump if last operation was not zero to a value inside a register	pc = RS1	101110.rrrrr.00000.00000.00000.000000
JMPEQR $RS1	Jump if last operation was zero to a value inside a register	pc = RS1	101111.rrrrr.00000.00000.00000.000000
JMPPAR $RS1	Jump if parity flag is set to a value inside a register	pc = RS1	110001.rrrrr.00000.00000.00000.000000

F.3.11 Miscellaneous

Syntax	Description	Operation	Encoding [31:0]
NOP	Does nothing for one clock cycle		000000.00000.00000.00000.00000.000000
HALT	Stops the CPU		100110.00000.00000.00000.00000.000000

Appendix G

PCB Layout

Comment	Description	Designator	Footprint	LibRef	Quantity
Phonejack Stereo SW	3-Conductor Jack with 2 break contacts (normals) and 2 auxiliary make contacts	AUX	Stereo connector - 35RAPC4BH3	Phonejack Stereo SW	1
Cap	Capacitor	C1, C2, C3, C4, C5, C6, C7, C8, C11, C12, C15, C16	6-0805_N	Cap	12
Cap Pol2	Polarized Capacitor (Axial)	C9, C10, C13, C14	RAD-0.1	Cap Pol2	4
LED	SMD LED	D1, D2, D3, D4, D5, D6, D7, D8	SMD_LED	LED	8
FSM14J	Pushbutton Switches Tactile 6x6mm 160g	DAC, SW_DelSig, SW_Mute	FSMJ	FSM14J	3
GPIO1		J1	HDR2X20	Altera_DE0_GPIO1	1
Header 9	Header, 9-Pin	J2	HDR1X9	Header 9	1
Analog_Discovery_2	Inverted footprint!	J3	HDR2X15	Analog_Discovery_2	1
2x16		LCD	2x16_LiquidCrystal_Display	2x16_LiquidCrystalDisplay	1
Condenser microphone	CMA Series 9.7 mm Electret Condenser Omni-Directional Pin Mount 3 V Microphone	MIC	CMA-4544PF-W	CMA-4544PF-W	1
Header 3	Header, 3-Pin	P1, P2, P3	HDR1X3	Header 3	3
Header 8	Header, 8-Pin	P4	HDR1X8	Header 8	1
PSoc 5lp		PSoC	PSOC-5LP	PSoc 5lp	1
BFR520	NPN Bipolar Transistor	Q1, Q2, Q3, Q4, Q5, Q6, Q7, Q8	SOT23-WIDE	BFR520	8
SMD_Res	MCR18	R1, R2, R3, R4, R5, R6, R7, R8, R9, R10, R11, R12, R13, R14, R15, R16, R17, R18	SMD_Res_MCR18	SMD_Res	18
RPot	Potentiometer	RT1, RT2, RT3, RT4, RT6	RPOT	RPot	5
RPot	Potentiometer	RT5	Sliding_Potentiometer	RPot	1
1823.6101	Salecom T8013 SPDT, 3A 250VAC & 5A 125VAC	SW1	T80-T	T80-T	1
OPA350UA	Semiconductors and Actives, Amplifiers, Buffers, Operational Amplifiers _General Purpose_	U1, U2	DIP8	OPA350UA	2
Adafruit_UDA1334_I2S_DAC	Adafruit I2S Stereo Decoder - UDA1334A	U3	Adafruit_UDA1334_I2S_DAC	Adafruit_UDA1334_I2S_DAC	1

Figure G.1: PCB Bill of Materials

G.1 System interfaces

G.1.1 I²S

For the audio data in the system the I²S protocol is used. This protocol is used because it is a very common serial interface for audio. Because I²S is a common interface it was easy to find a DAC and ADC supporting the protocol. The ADC used is the one in the PSOC. The DAC is an ADA Fruit I²S Stereo decoder model UDA1224A. The I²S interface on FPGA the coded in VHDL according to the original I²S bus specification and adjusted for the needs in this project.

VHDL I²S interface

The I²S interface coded for the FPGA can receive and send a 16-bit depth I²S signal. The Interface is coded as two modules. Both base modules are made to receive and send a stereo signal. Since this project focus on mono sound, a wrapper has been made for the modules. The input module wrapper only uses the input from the right channel, and the output wrapper send the mono signal out on both channels. Thus, the CPU only needs to handle a mono signal. This was done to make focus on the objective and simplify data flow handling. However minimal changes can be made to support both stereo and higher bit accuracies if so wished.

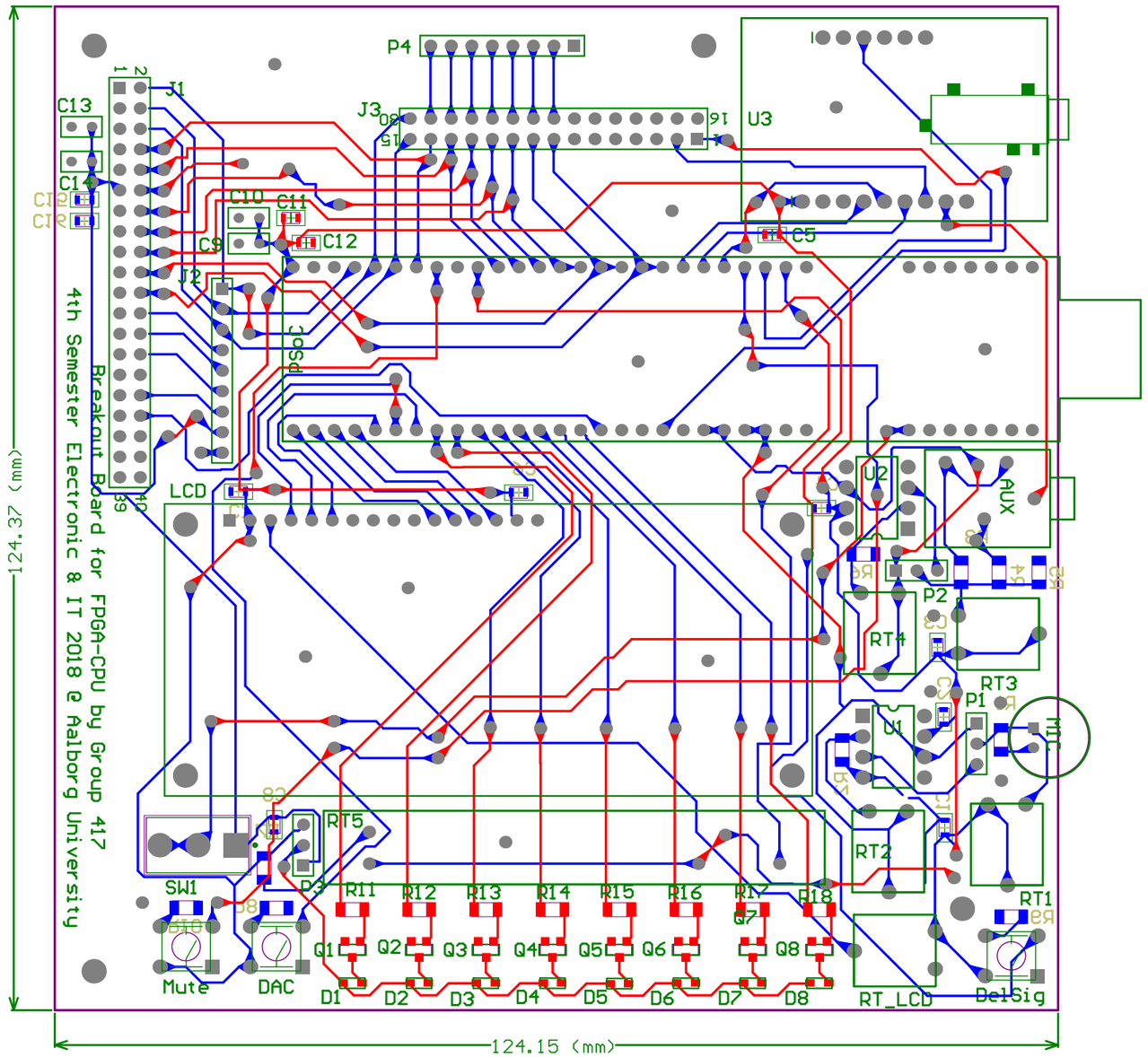


Figure G.2: PCB layout without ground planes
The green lines and text signify component overlay footprints) and text, the red lines are top traces routing, the blue lines are bottom layer traces and the grey circles are holes in the PCB

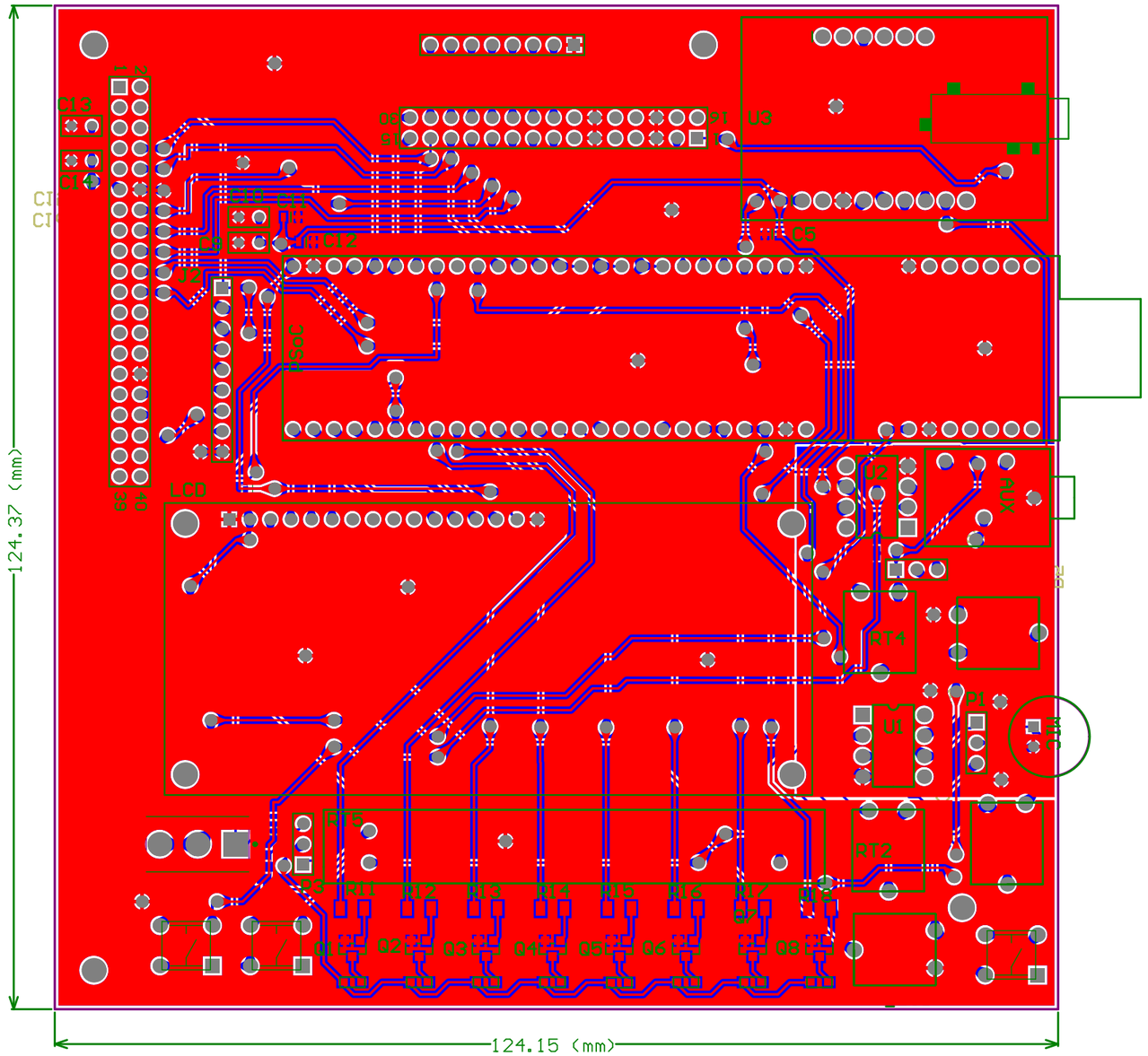


Figure G.3: Top-view of the PCB with ground planes

Appendix H

Test Journal: The ALU

This test journal is focused on the ALU and will go through all the procedures that was executed in the test. The first part will focus on the purpose of the test and what limitations, setups, and background knowledge there is inside the test frame. There will also be an equipment list.

At the end of the test journal there will be an execution list with all the executed tests and a section about the results. The test journal will end with a conclusion of the test.

H.1 Test Purpose

The purpose of this test is to ensure that the different operations of the ALU are executed correctly inside the ALU module.

H.2 Test Frame

This test will be executed using a simulation tool run on a computer, based on the assumption that the results from the simulation test bench corresponds to the results to be expected when the module is implemented in hardware.

The test includes only the ALU module without connection to the rest of the CPU.

H.2.1 Test Setup, Equipment and Test Procedure

The test setup consists of a test bench, a test protocol and a simulation tool. The test bench is written in VHDL and determines the inputs to the ALU module during the simulation. The test protocol contains a list of all the tested operations, the inputs and the expected outputs. The expected outputs are calculated on a separate calculator where applicable or by handwritten calculations. After running the simulation, the simulation tool is set to show the states of the various inputs, outputs and signals

during the simulation. The output data is then noted and compared to the expected outputs found in the test journal.

Equipment:

- Windows 10 64-bit
- ModelSim ALTERA STARTER EDITION 10.1d

H.3 Description of Executed Tests

The test of the ALU runs through all the available operations in the ALU. A list of these can be found in table H.1.

Table H.1: Table with available operations in the ALU

Operation Name	Description	Opcode
ADD	Adds the two input operands	0x0001
SUB	Subtracts the two operands	0x0002
MUL	Multiplies the two operands,	0x0003
AND	ANDs the two operands	0x0004
OR	ORs the two operands	0x0005
XOR	XORs the two operands	0x0006
NEA	NEGATES operand A	0x0007
NEB	NEGATES operand B	0x0008
NOA	NOT's operand A	0x0009
NOB	NOT's operand B	0x000A
LSL	Logic Shift Left	0x000B
LSR	Logic Shift Right	0x000C
ASR	Arithmetic Shift right	0x000D
PSA	Passes operand A	0x000E
PSB	Passes operand B	0x000F
ICA	Increments operand A	0x0010
ICB	Increments operand B	0x0011
NOP	Does nothing, does not change flags	0x0012

Each operation is tested in expected edge cases, to test the module as thoroughly as possible with the least amount of cases. Due to the similarities between some of the operations, such as the logic operators, some of the operations are given the same inputs. The chosen cases, with inputs and expected outputs can be found in appendix H.3

H.4 Test Results

The results from the test can be seen in the table in appendix H.3. For educational purposes an example with the results for one of the operations is shown in this section.

The example shows the test of the ADD operation in the ALU and its results. The inputs, expected outputs and simulation outputs from the tests can be seen in table H.2 and the waveform from the simulation tool can be seen in figure H.1.

Table H.2: Table with results from test of ADD operation.

Adder						O-S-Z-P-C	
	Test nr.	Operand_a	Operand_b	Expected output	Simulation output	Expected flags	Simulation flags
Zero-input	1	x"0000"	x"0000"	x"0000"	x"0000"	0-0-1-0-0	0-0-1-0-0
Two positive input in range	2	x"0000"	x"2000"	x"2000"	x"2000"	0-0-0-1-0	0-0-0-1-0
Two unsigned input, carry test	3	x"F000"	x"1000"	x"0000"	x"0000"	0-0-1-0-1	0-0-1-0-1
Two signed positive input, overflow test	4	x"7000"	x"5000"	x"C000"	x"C000"	1-1-0-0-0	1-1-0-0-0
Two signed negative input, in range	5	x"E000"	x"B000"	x"9000"	x"9000"	0-1-0-0-1	0-1-0-0-1
Two signed negative input, carry test	6	x"F000"	x"F000"	x"E000"	x"E000"	0-1-0-1-1	0-1-0-1-1
Two signed negative input, overflow test	7	x"F000"	x"8000"	x"7000"	x"7000"	1-0-0-1-1	1-0-0-1-1

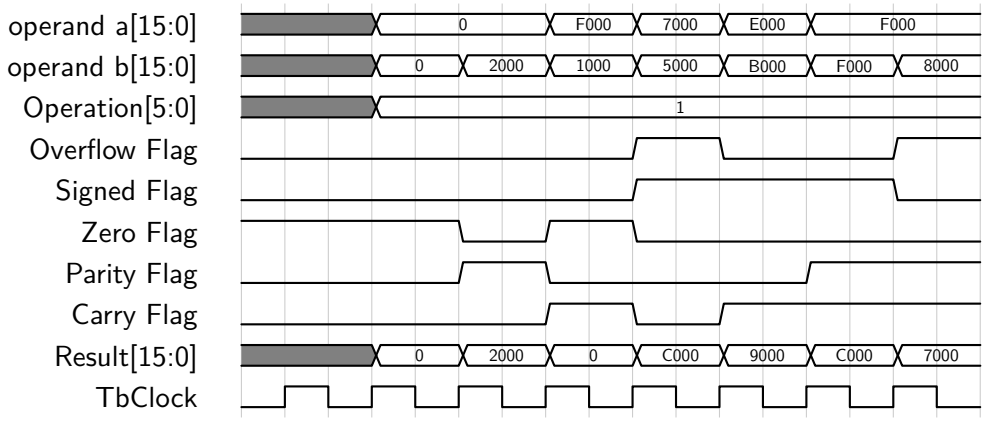


Figure H.1: Illustration of the output from the simulation tool. The operation input is set as "1", corresponding to the "ADD" operation.

H.5 Conclusion

As the table in appendix H.3 shows, in each of the different cases the simulation output is the same as the expected outputs, likewise with the flags. Thus, it is concluded with reasonable certainty that the operations in the ALU module are functional.

Table H.3: ALU test protocol

Adder OPCODE 1	Test number	Input1 hex	input2 hex	Expected Output hex	output	Expected flag	Flag
zero-input	1	0000	0000	0000	0000	0-0-1-0-0	0-0-1-0-0
two positive inputs inside scope	2	0000	2000	2000	2000	0-0-0-1-0	0-0-0-1-0
two unsigned positive inputs, carry test	3	F000	1000	0000	0000	0-0-1-0-1	0-0-1-0-1
two signed positive numbers, overflow test	4	7000	5000	C000	C000	1-1-0-0-0	1-1-0-0-0
two signed negative numbers inside scope	5	E000	B000	9000	9000	0-1-0-0-1	0-1-0-0-1
two signed negative numbers, carry test	6	F000	F000	E000	E000	0-1-0-1-1	0-1-0-1-1

two signed negative numbers, overflow test	7	F000	8000	7000	7000	1-0-0-1-1	1-0-0-1-1
Subtractor OPCODE 2						O-S-Z-P-C	
	Test number	Input1 hex	input2 hex	Expected Output hex	output	Expected flag	Flag
zero-input	1	0000	0000	0000	0000	0-0-1-0-0	0-0-1-0-0
two unsigned positive inputs, inside scope 20-10	2	0014	000A	000A	000A	0-0-0-0-0	0-0-0-0-0
two unsigned positive inputs, carry test (10 - 20)	3	000A	0014	FFF6	FFF6	0-1-0-0-1	0-1-0-0-1
signed negative - unsigned (-10 - 20)	4	FFF6	0014	FFE2	FFF2	0-1-0-0-0	0-1-0-0-0
two signed negative numbers inside scope (-10 - (-20))	5	FFF6	FFEC	000A	000A	0-0-0-0-0	0-0-0-0-0
two signed negative numbers, carry test (-20 - (-10))	6	FFEC	FFF6	FFF6	FFF6	0-1-0-0-1	0-1-0-0-1
two signed negative numbers, overflow test	7	8000	7000	1000	1000	1-0-0-1-0	1-0-0-1-0
Multiplier OPCODE 3						O-S-Z-P-C	
	Test number	Input1 hex	input2 hex	Expected Output hex	output	Expected flag	Flag
zero-input	1	0000	0000	0000	0000	0-0-1-0-0	0-0-1-0-0
two positive inputs inside scope	2	7000	0001	7000	7000	0-0-0-1-0	0-0-0-1-0
two unsigned positive inputs, carry test	3	FFFE	0002	FFFC	FFFC	1-1-0-0-1	1-1-0-0-1
two signed positive numbers, overflow test	4	7000	0005	3000	3000	1-0-0-0-1	1-0-0-0-1
two signed negative numbers inside scope	5	E000	FFFE	4000	4000	0-0-0-1-0	0-0-0-1-0
two signed negative numbers, overflow test	6	C000	FFFA	8000	8000	1-1-0-1-1	1-1-0-1-1
AND OPCODE 4						O-S-Z-P-C	
	Test number	Input1 hex	input2 hex	Expected Output hex	output	Expected flag	Flag
zero input	1	0000	0000	0000	0000	0-0-1-0-0	0-0-1-0-0
Four alternated ones and zeros	2	0000	5000	0000	0000	0-0-1-0-0	0-0-1-0-0

Four alternated ones and zeros	3	0000	A000	0000	0000	0-0-1-0-0	0-0-1-0-0
All bits equal to 1	4	0000	FFFF	0000	0000	0-0-1-0-0	0-0-1-0-0
Two equal inputs	5	F000	F000	F000	F000	0-1-0-0-0	0-1-0-0-0
OR OPCODE 5						O-S-Z-P-C	
	Test number	Input1 hex	input2 hex	Expected Output hex	output	Expected flag	Flag
zero-input	1	0000	0000	0000	0000	0-0-1-0-0	0-0-1-0-0
Four alternated ones and zeros	2	0000	5000	5000	5000	0-0-0-0-0	0-0-0-0-0
Four alternated ones and zeros	3	0000	A000	A000	A000	0-1-0-0-0	0-1-0-0-0
All bits equal to 1	4	0000	FFFF	FFFF	FFFF	0-1-0-0-0	0-1-0-0-0
Two equal inputs	5	F000	F000	F000	F000	0-1-0-0-0	01000
XOR OPCODE 6						O-S-Z-P-C	
	Test number	Input1 hex	input2 hex	Expected Output hex	output	Expected flag	Flag
zero-input	1	0000	0000	0000	0000	0-0-1-0-0	00100
Four alternated ones and zeros	2	0000	5000	5000	5000	0-0-0-0-0	00000
Four alternated ones and zeros	3	0000	A000	A000	A000	0-1-0-0-0	01000
All bits equal to 1	4	0000	FFFF	FFFF	FFFF	0-1-0-0-0	01000
Two equal inputs	5	F000	F000	0000	0000	0-0-1-0-0	00100
Negate A Opcode 7						O-S-Z-P-C	
	Test number	Input1 hex	input2 hex	Expected Output hex	output	Expected flag	Flag
zero-input	1	0000	0000	0000	0000	0-0-1-0-0	00100
From smallest positive to smallest negative	2	1000	0000	F000	F000	0-1-0-0-0	01000
From largest negative to largest positive	3	8000	0000	8000	8000	1-1-0-1-0	11010
From second largest negative number to largest positive number	4	9000	0000	7000	7000	0-0-0-1-0	00010
From smallest negative to smallest positive	5	F000	0000	1000	1000	0-0-0-1-0	00010
	6	0000	F000	0000	0000	0-0-1-0-0	00100
	7	1000	F000	F000	F000	0-1-0-0-0	01000
	8	8000	F000	8000	8000	1-1-0-1-0	11010

	9	9000	F000	7000	7000	0-0-0-1-0	00010
	10	F000	F000	1000	1000	0-0-0-1-0	00010
Negate B Opcode 8						O-S-Z-P-C	
	Test number	Input1 hex	input2 hex	Expected Output hex	output	Expected flag	Flag
zero-input	1	0000	0000	0000	0000	0-0-1-0-0	00100
From smallest positive to smallest negative	2	0000	1000	F000	F000	0-1-0-0-0	01000
From largest negative to largest positive	3	0000	8000	8000	8000	1-1-0-1-0	11010
From second largest negative number to largest positive number	4	0000	9000	7000	7000	0-0-0-1-0	00010
From smallest negative to smallest positive	5	0000	F000	1000	1000	0-0-0-1-0	00010
	6	F000	0000	0000	0000	0-0-1-0-0	00100
	7	F000	1000	F000	F000	0-1-0-0-0	01000
	8	F000	8000	8000	8000	1-1-0-1-0	11010
	9	F000	9000	7000	7000	0-0-0-1-0	00010
	10	F000	F000	1000	1000	0-0-0-1-0	00010
NOT A Opcode 9						O-S-Z-P-C	
	Test number	Input1 hex	input2 hex	Expected Output hex	output	Expected flag	Flag
zero-input	1	0000	0000	FFFF	FFFF	0-1-0-0-0	01000
Four alternated ones and zeros	2	5000	0000	AFFF	AFFF	0-1-0-0-0	01000
Four alternated ones and zeros	3	A000	0000	5FFF	5FFF	0-0-0-0-0	00000
All bits equal to 1	4	FFFF	0000	0000	0000	0-0-1-0-0	00100
Same as above, though input 2 changes	5	0000	F000	FFFF	FFFF	0-1-0-0-0	01000
	6	5000	F000	AFFF	AFFF	0-1-0-0-0	01000
	7	A000	F000	5FFF	5FFF	0-0-0-0-0	00000
	8	FFFF	F000	0000	0000	0-0-1-0-0	00100
NOT B Opcode 10						O-S-Z-P-C	
	Test number	Input1 hex	input2 hex	Expected Output hex	output	Expected flag	Flag
zero-input	1	0000	0000	FFFF	FFFF	0-1-0-0-0	01000
Four alternated ones and zeros	2	0000	5000	AFFF	AFFF	0-1-0-0-0	01000

Four alternated ones and zeros	3	0000	A000	5FFF	5FFF	0-0-0-0-0	00000
All bits equal to 1	4	0000	FFFF	0000	0000	0-0-1-0-0	00100
Same as above, though input 2 changes	5	F000	0000	FFFF	FFFF	0-1-0-0-0	01000
	6	F000	5000	AFFF	AFFF	0-1-0-0-0	01000
	7	F000	A000	5FFF	5FFF	0-0-0-0-0	00000
	8	F000	FFFF	0000	0000	0-0-1-0-0	00100
Logic shift left Opcode 11						O-S-Z-P-C	
	Test number	Input1 hex	input2 hex	Expected Output hex	output	Expected flag	Flag
zero-input	1	0000	0000	0000	0000	0-0-1-0-0	00100
Shift zero times to the left	2	0001	0000	0001	0001	0-0-0-1-0	00010
Shift one time to the left	3	0001	0001	0002	0002	0-0-0-1-0	00010
Shift completely to the left	4	0001	000F	8000	8000	0-1-0-1-0	01010
Shift completely to the left + 1	5	0001	0010	0000	0000	0-0-1-0-0	00100
Logic shift right Opcode 12						O-S-Z-P-C	
	Test number	Input1 hex	input2 hex	Expected Output hex	output	Expected flag	Flag
zero-input	1	0000	0000	0000	0000	0-0-1-0-0	00100
Shift zero times to the right	2	8000	0000	8000	8000	0-1-0-1-0	01010
Shift one time to the right	3	8000	0001	4000	4000	0-0-0-1-0	00010
Shift completely to the right	4	8000	000F	0001	0001	0-0-0-1-0	00010
Shift completely to the right +1	5	8000	0010	0000	0000	0-0-1-0-0	00100
Arithmetic shift right Opcode 13						O-S-Z-P-C	
	Test number	Input1 hex	input2 hex	Expected Output hex	output	Expected flag	Flag
zero-input	1	0000	0000	0000	0000	0-0-1-0-0	00100
Shift zero times to the right	2	8000	0000	8000	8000	0-1-0-1-0	01010
Shift one time to the right	3	8000	0001	C000	C000	0-1-0-0-0	01000
Shift completely to the right	4	8000	000F	FFFF	FFFF	0-1-0-0-0	01000
Shift completely to the right +1	5	8000	0010	FFFF	FFFF	0-1-0-0-0	01000
Shift one time to the right (test of sign extension)	6	4000	0001	2000	2000	0-0-0-1-0	00010

Shift completely to the right (test of sign extension)	7	4000	000E	0001	0001	0-0-0-1-0	00010
Shift completely to the right +1 (test of sign extension)	8	4000	000F	0000	0000	0-0-1-0-0	00100
PASS A Opcode 14						O-S-Z-P-C	
	Test number	Input1 hex	input2 hex	Expected Output hex	output	Expected flag	Flag
zero-input	1	0000	0000	0000	0000	0-0-1-0-0	00100
Random number in scope	2	5000	0000	5000	5000	0-0-0-0-0	00000
All bits equal to 1	3	FFFF	0000	FFFF	FFFF	0-1-0-0-0	01000
Same as above, though input 2 changes	4	0000	F000	0000	0000	0-0-1-0-0	00100
	5	5000	F000	5000	5000	0-0-0-0-0	00000
	6	FFFF	F000	FFFF	FFFF	0-1-0-0-0	01000
PASS B Opcode 15						O-S-Z-P-C	
	Test number	Input1 hex	input2 hex	Expected Output hex	output	Expected flag	Flag
zero-input	1	0000	0000	0000	0000	0-0-1-0-0	00100
Random number in scope	2	0000	5000	5000	5000	0-0-0-0-0	00000
All bits equal to 1	3	0000	FFFF	FFFF	F000	0-1-0-0-0	01000
Same as above, though input 2 changes	4	F000	0000	0000	0000	0-0-1-0-0	00100
	5	F000	5000	5000	5000	0-0-0-0-0	00000
	6	F000	F000	F000	F000	0-1-0-0-0	01000
INC A Opcode 16						O-S-Z-P-C	
	Test number	Input1 hex	input2 hex	Expected Output hex	output	Expected flag	Flag
zero-input	1	0000	0000	0001	0001	0-0-0-1-0	00010
All bits equal to 1	2	FFFF	0000	0000	0000	0-0-1-0-1	00101
Largest positive signed, overflow test	3	7FFF	0000	8000	8000	1-1-0-1-0	11010
INC B Opcode 17						O-S-Z-P-C	
	Test number	Input1 hex	input2 hex	Expected Output hex	output	Expected flag	Flag

zero-input	1	0000	0000	0001	0001	0-0-0-1-0	00010
All bits equal to 1	2	0000	FFFF	0000	0000	0-0-1-0-1	00101
Largest positive signed, overflow test	3	0000	7FFF	8000	8000	1-1-0-1-0	11010
NOP Opcode 18							
	Test number	Input1 hex	input2 hex	Expected Output hex	output	Expected flag	Flag
zero-input	1	0000	0000	0001			
All bits equal to 1	2	0000	FFFF	0000			
Random number in scope	3	0000	4000	4001			

Appendix I

Test Journal: FIR Filter

I.1 Test Purpose

The purpose of this test is to ensure the filter works as designed, as well as to make sure it works in conjunction with the CPU. Furthermore, it is desired to see how the filter coefficients behave when rounded, since the CPU is fixed point.

I.2 Test Frame

The tests will be done partly via simulation and partly through the CPU. By simulating the filter, it is not certain that it will be true to the real implementation, however due to how much easier it is to simulate, it is chosen to be done this way.

When using the CPU to test subsystems, it is important to note that the CPU might not work entirely correctly, so it could be a source of error.

Theoretical Background

Refer to appendix B for the theory behind a FIR filter, as well as digital signal processing.

Test setup and Equipment

- Softcore CPU.
- Digilent Analog Discovery 2.
- Digilent Waveforms software.
- Quartus II 64-bit software.
- Altera ModelSim software.

I.3 Executed Tests

Four tests will be executed to test the functionality of the FIR filter module. The tests include both black and white box tests. The tests are as follows:

1. Loading coefficients.
2. Impulse response.
3. Comparison of filter cutoff frequency with designed cutoff.
4. Comparison of filtered and unfiltered audio signals on oscilloscope.

The first two tests are white box tests, in that they require an understanding of the internal workings of the filter. These tests will be done using the Altera ModelSim software to simulate normal filter operation. The input will be a normalized impulse, to check the impulse response. this test will also allow for checking of correct coefficient loading.

The last two tests will be executed using the CPU. These tests will be done by using an oscilloscope to check the output of the filter. To check the cutoff frequency, a frequency sweep will be done using an oscilloscope. To check the difference in unfiltered and filtered signals, a combination of sinusoidal signals will be given as input, and the output will be checked using an oscilloscope.

I.4 Test Results

Coefficient and Impulse Test

These two tests were simulated and done together, since this was possible. This is because the coefficients must be loaded correctly for the impulse test to be done.

In figure, I.1 a timing diagram of the relevant values of the simulation can be found.

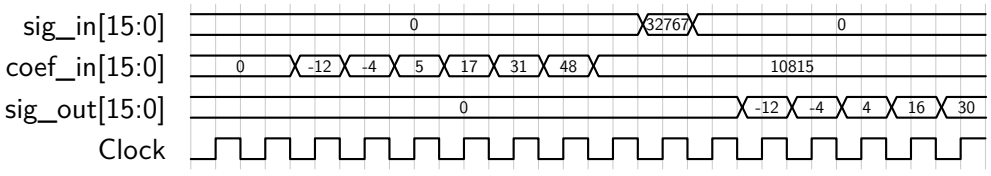


Figure I.1: Simulation output from simulating the impulse response.

Figure I.1 illustrates how the coefficients are loaded in to begin with. This is done via the coef_in signal, which loads the values -12 to 48. The value 10815, which comes after, is undefined in the test and filter and is irrelevant to the operation.

Once the last coefficient is loaded in, the input signal of 32767 followed by zeros is given via the sig_in signal. With the input given, the filter will begin outputting filtered values via the sig_out signal. Since it is an impulse test, the filter coefficients should be output in order, however some are rounded down. This is due to using fixed point values with division.

Cutoff Frequency Test

To make sure the FIR filter works as intended, it must follow the designed filter values for cutoff frequencies. To test this, a frequency sweep of a 400 Hz low-pass filter was done. The result of this test can be seen in figure I.2.

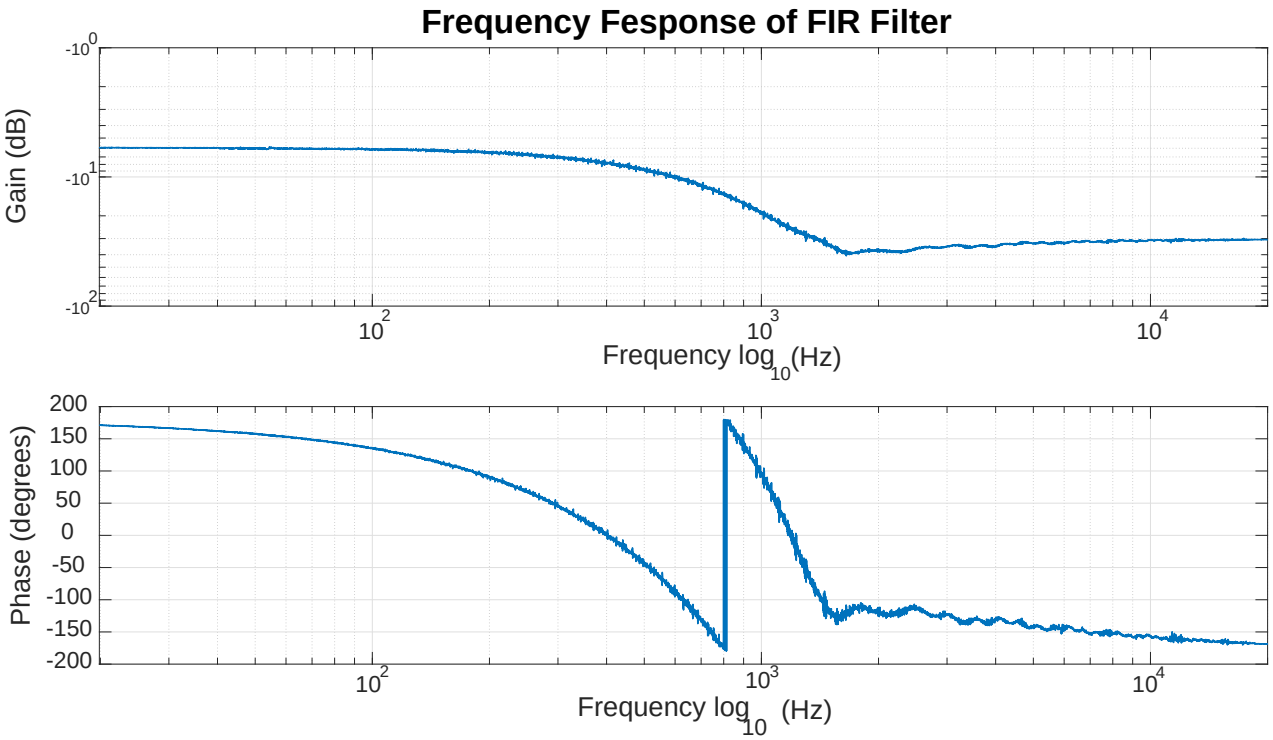


Figure I.2: Frequency response of 400 Hz low-pass filter implemented on the CPU

The filter begins to cut off at around 400 Hz, which is intended. It is however worth noting that the filter has a slight reduction in amplitude compared to the input signal.

Filtered Signal Test

In addition to testing the cutoff frequency of the filter, it is also desired to test the filter's ability to filter signals. To test this, two sine waves, one at 300 Hz and one at 2 kHz, were used as input (blue), and run through the same 400 Hz filter as the previous test. In figure I.3, the filtered output (red) can be seen. The offset in voltage is due to the PSoC not being able to sample signals below 0 V, so the input is DC shifted.

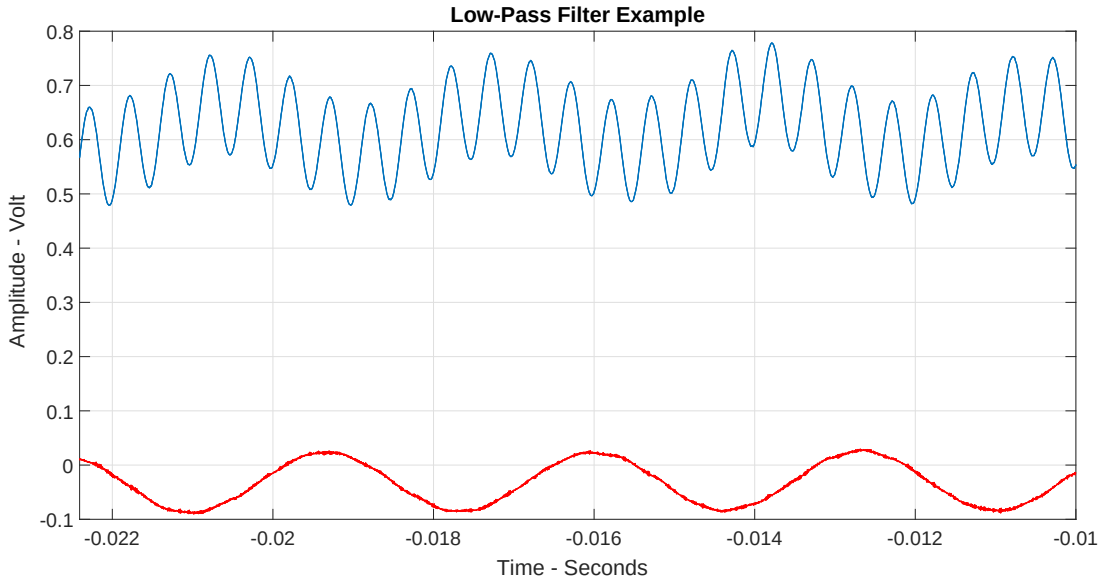


Figure I.3: A 400 Hz low-pass filter used to separate a 300 Hz tone from a 2 kHz tone

I.5 Sources of Error and Other Uncertainties

Due to the nature of the test environments, unexpected errors can occur. Using the CPU can potentially cause errors that are not from the filter. Simulating the filter can be untrue to the actual implementation, so this must be kept in mind as well.

Appendix J

Test Journal: I²S Mono VHDL Module

This test journal will cover the tests conducted on the I²S VHDL modules. The tests are simulated tests of the I²S modules to get an insight into the internal workings which cannot be made once implemented on the FPGA.

J.1 Test Purpose

The purpose of this test is to check the behavior of the I²S VHDL module by simulation. This test will focus on the following features of the I²S modules:

- The same signal that is sent in on the right channel go out on both channels.
- The word select on the output changes on the falling edge of the clock cycle one cycle before the LSB.
- When the bit depth of the signal is less than the 16-bit, the module assume that the rest of the 16-bit is zero.
- When the bit depth of the incoming signal is more than the 16-bit, it will truncate the signal to cut off the LSBs.
- The interrupt from the input module go high when a word is ready in its buffer.
- The interrupt of the input module resets on the next rising edge of its clock when the interrupt reset is set to high

J.2 Test Frame

This test is conducted based on the assumptions that the simulation is going to behave like the physical implementation. And that if the focus points of the test are all behaving as intended, the module will be fully functional. While the entire modules where simulated, only the important signals where logged as wave forms. The important wave forms are input I²S signals, and the I²S output signals where logged. The also the interrupt signals and the internal buffers where logged.

Theoretical Background

For theory on the I²S see the description 7.1 and datasheet G.1 of the I²S protocol.

Test setup, equipment and test procedure

- Windows 10 64-bit
- ModelSim ALTERA STARTER EDITION 10.1d

J.3 Executed Tests

To test the points a testbench was made. The testbench tests a module that pass the I^2S signal from the input module directly to the output module. First the testbench sends numbers from 0 to 5. Afterwards the maximum values and minimum values are send. Then bursts with lower bit-depth is send. And finally bursts with too high a bit-depth.

J.4 Test Results and Data Processing

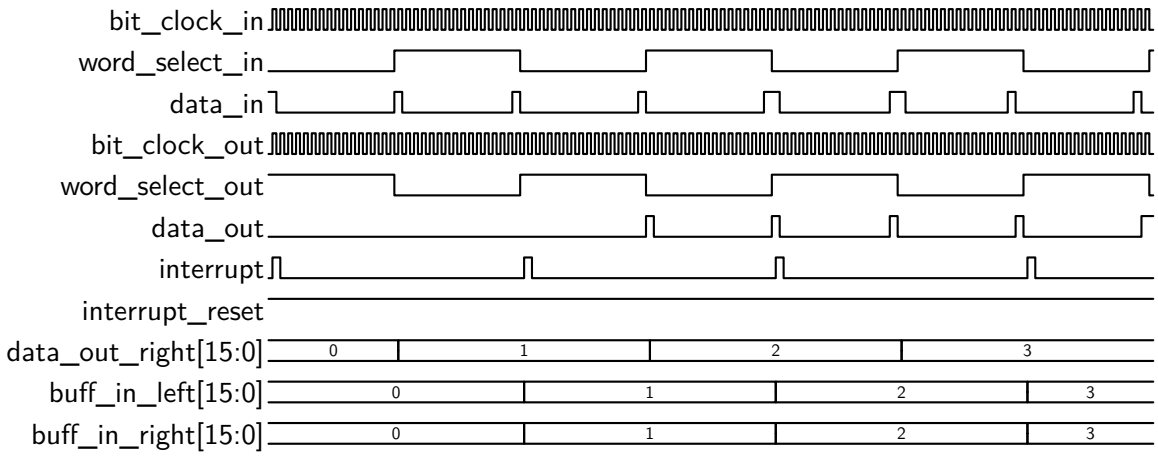


Figure J.1: Standard operation of the I^2S. The output is delayed by three word lengths. But perfectly replicated

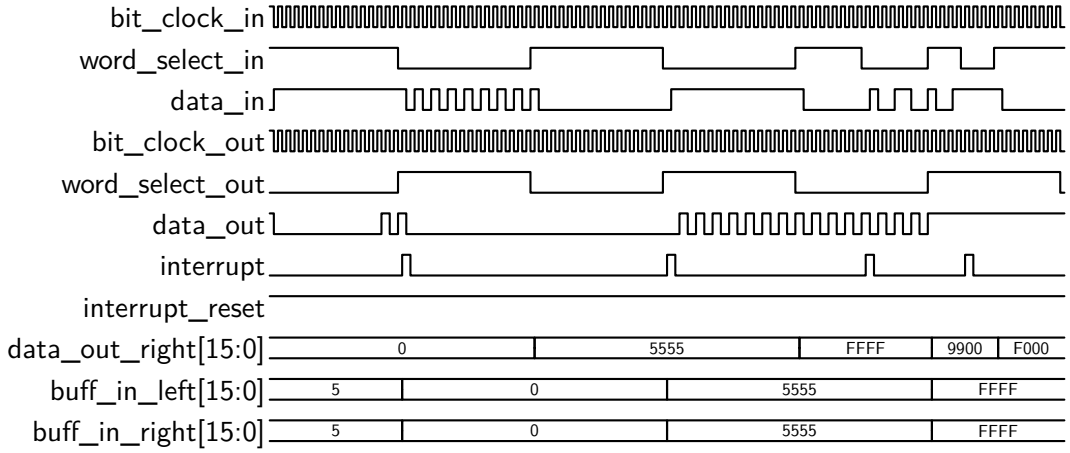


Figure J.2: The I^2S recieving 2 edgecasse of values, all ones and all zero's

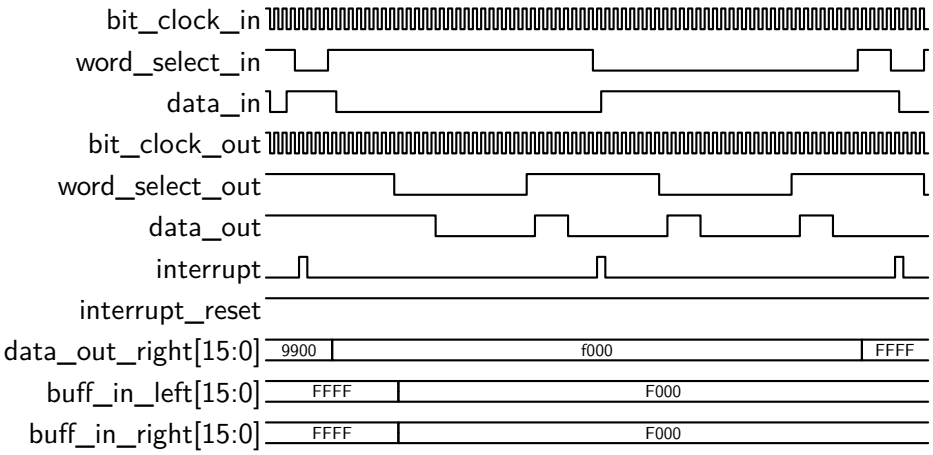


Figure J.3: The I^2S receiving small burst and filling the LSBs with zeroes, and receiving a larger than intended word and cutting off the LSBs

J.5 Test Results

By inspecting the waveforms from the simulation, it is apparent that all of the features of the I^2S modules are working as intended.

Appendix K

Test Journal: The Buttons

To make sure the button works as intended a test have been conducted.

K.1 Test purpose

The purpose of this test is to make sure that the buttons works as intended and will not cause any errors. The button driver must be able to define when they are pressed and the amount of times.

K.2 Test Frame

The button driver is implemented on the FPGA to work with the CPU. But it is not the CPU which sends signals to the buttons. Therefore, it is tested by simulating a button press to the driver.

It is known that by conducting the test this way there will not be any testing for actually button presses. Should any errors later be discovered were the button driver could be a part, will it be inspected again.

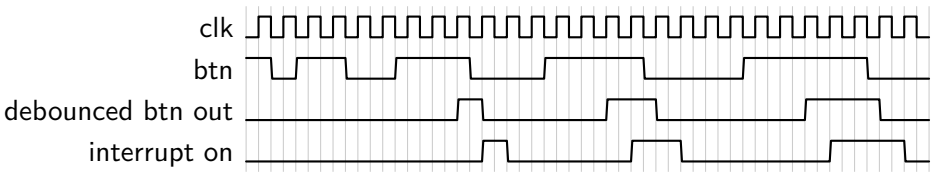
Test Setup and Equipment

- Quartus.
- Test program Quartus.

K.3 Executed Tests

The button driver will only accept a button push were the button is pushed down for longer than 3 clock cycles. Therefore, the test that is executed will have first two "fake" button pushes which are not long enough for the driver to accept them. After this multiplier pushes are made to see how the driver handles pushes coming fast after each other.

Figure K.1: Test signal from button driver



K.4 Test Results and Data Processing

Test was without errors and therefore successful. Here are the test signals that was conducted on the driver:

The clk signal is the clock. The btn is the simulated button, high for pushed and low for being idle. The debounced btn out is a signal to make sure that no pushes may count as two. The interrupt on is the signal that is sent to the CPU.

K.5 Sources of Error and Other Uncertainties

There should not be any errors around the button driver since it is tested now. However, it is not tested how it works together with the CPU and how it works with real life button pushes. This will though be tested in a full system test.

Appendix L

Test Journal: Seven-Segment-Displays

This test journal will describe the tests conducted to ensure correct functionality of the seven-segment-display driver.

L.1 Test Purpose

The purpose of this test is to make sure the seven-segment-display driver works as intended. The driver must allow for changing display settings from decimal to hexadecimal, display decimal dots and obviously display the data it is given.

L.2 Test Frame

The seven-segment-display driver is implemented on the FPGA to work with the CPU. As such, the driver will be tested by using the CPU to input values for the seven segment to display.

A downside to doing it this way, is that any errors found in the test can be caused by the CPU not working as intended. Should this be the case will the driver be inspected on its own later.

Test Setup and Equipment

- FPGA
- Softcore CPU

L.3 Executed Tests

Since the driver can be tested using the CPU, the test can be written in assembly code. The test aims to test all 16 values of each seven-segment-display, as well as the decimal dot functionality. The program will first display 0 on all the displays. When a button is pressed, the CPU will increment all displays by one, for a total of 16 times to test all hexadecimal values. Afterwards, the display will display all decimals dots. Finally, the same will be done with decimal values.

L.4 Test Results

The test concluded with no errors. All ten decimal values and 16 hexadecimal values displayed correctly. The decimal dots also worked as intended.

L.5 Sources of Error and Other Uncertainties

A known source of error could be using the CPU to perform the test, since the CPU might not work as intended. Using the CPU also means using the assembler to write a test program. The assembler has been tested, see appendix M, and showed no errors, however it is still a possible error source.

Appendix M

Test Journal: The Assembler

This test journal is focused on the assembler and will go through all the procedures that was executed in the test. The first part will focus on the purpose of the test and what limitations, setups, and background knowledge there is inside the test frame. There will also be an equipment list.

At the end of the test journal there will be an execution list with all the executed tests, a section about the results and what to anticipate from the assembler henceforth. The test journal will end in a discussion about errors and other uncertainties about the assembler.

M.1 Test Purpose

When the CPU gives back an error there can be more than one reason for the error occurring. To eliminate the assembler from being a possible factor in the errors that can occur in the CPU a test protocol is conducted.

M.2 Test Frame

The test is conducted with the knowledge of what is behind the user interface. This indicates that the test protocol is not a 'black box' test. Not every single possible input is tested. Instead, only edge cases and instructions containing all opcodes are tested. There will also be some random checks, a full range test of all opcodes, and a full range test of all registers. Only some immediates that is close to the extremes will be tested. The reasons for these tests are further described in the section M.3.

The program will only be tested on one computer because the executable file (.exe) for the program is expected to work on all up to date windows computers.

This test frame is constructed with the knowledge in the assembler manual appendix F.

Theoretical Background

It is assumed that the correct syntax is used when testing the functionality of intended features. When testing unintended features, correct syntax may not be used.

To fully comprehend the test journal, it is recommended to read the assembler manual F. In the manual there will be an explanation about immediates that can only contain values inside 16-bit. It will also contain the information about the correct syntax and expected output.

Test Setup and Equipment

- Windows 10 pro 64-bit, version 10.0.16299 Build 16299.
- Microsoft Visual Studio community 2015, version: 14.0.25431.01 Update 3. Microsoft .NET Framework, version 4.7.02556

M.3 Executed Tests

The following tests were executed to ensure correct behavior of the assembler. Some unintended functionalities were also tested, to ensure a sturdier assembler. A more detailed explanation of these tests will be given after the list of tests.

- Opcodes - All opcodes were tested to ensure correct binary output and output order.
- Registers - All registers were tested to ensure they corresponded to their intended binary values.
- Immediates - All known extreme values of the immediates were tested. Both max and min values for signed and unsigned integers.
- Jumps and labels - Labels were tested to make sure they had the correct jump values. Jumps were checked to make sure they handled label names and placing correctly.
- Comments - To make sure the assembler ignored comments correctly, these were tested with used syntax such as opcodes and registers.
- White space and overflow in input size - These were tested to make sure the assembler correctly handles lines without code or comments in. Additionally, lines with more characters than the input buffer can handle was tested, mostly to check behavior.

Opcode Tests

Each opcode was written in an input document, with correct syntax (registers, immediates etc.). The document was assembled, and each line was checked against the assembly language manual F.

Register Tests

All registers were tested by using the ADDR opcode. The reason for this is to make the test more efficient since its only purpose is to check if the registers are given the right bit values. The ADDR opcode takes three registers, and all registers are written in numerical order. The output was then checked to make sure the register values were counting in the same manner.

Immediate Tests

To ensure immediate values worked properly, edge cases of these were tested. The highest and lowest signed and unsigned values were tested for 16-bit numbers. That means 65536 should be overflow, 65535 should be all 1-bits, 0 should be all 0-bits, -32768 should be underflow, -32767 should be 1

followed by all 0-bits. Minus zero was also tested to check how the assembler would handle the case of a input looking like this '-0'.

Jump and Label Tests

Since labels and jumps are used together, these were tested together. First, to make sure labels worked as intended, similar label names were given, e.g. label1 and Label1. Additionally, multiple labels for the same line were given, as well as redefining labels later in the code. Labels were also tested with names of opcodes and registers, to ensure they were not treated as code.

For jumps, similar tests were conducted to ensure the correct label value was used when assembled.

Comment Tests

Since comments are to be ignored by the compiler, all defined special characters were tested after the comment symbol, e.g. //$r1. Comments can both be put in line with code and on its own, so both were tested.

M.3.1 White Space and Overflow of input Buffer Tests

To test how robust the assembler is, white spaces were put in front of and in between opcodes and operands. Additionally, newlines were filled with white spaces to check how these were handled.

Since the assembler is not supposed to handle inputs over the buffer size, the test of this was purely to see how the assembler treated the input. both comments and actual lines of code were written to be over the buffer size.

M.4 Test Results and Data Processing

Only focusing on the test results in section M.6, will this section conclude on what caused errors and what did went as expected in the assembler test. Sources of errors that can create problems for the user are discussed in the M.5 section.

Opcode Test Results

Most opcodes assembled correctly, however NEGI, LOAD and STORE showed some issues. NEGI did not output the desired destination register. LOAD did not have the correct output order for a memory address given by [REG+IMM]. STORE did not have the correct output order for a memory address given by only [IMM].

Register Test Results

Nothing noteworthy, everything outputted in the right results. No error occurred, and no special cases emerged.

Immediate Test Results

As expected overflow and underflow immediates results in errors. These errors are however not always traceable since the output will result in a new value because of overflow and underflow. The new value will not easily be spotted unless every value is specifically inspected in binary to make sure that every number is outputted correctly. This will be reviewed in the M.5 section about errors and other uncertainties.

The max unsigned value is outputted correctly, and so is the minimum and maximum signed values.

There was an error when zero was used as an immediate. Another error also occurred when '-0' was used as an input.

Jump and Label Test Results

If two labels are defined on separate lines, with the same name, only the first label will be used. All special characters can be used in labels, also the ones used in the assembly language. A label cannot contain an opcode written the same way as in the actual language. Labels cannot exceed 50 characters, which is expected since the label string is 50 characters long. More than one unique label works for the same line of code. In other words, two labels can be placed right after each other and therefore refer to the same line of code.

Comment Test Results

Comments, both in line with code and on its own, work as intended. Comments, including the code it might be in line with, may not exceed 100 characters. This is expected.

White Space and Overflow of Input Buffer Test Results

Using more than one white space between the operands of a line of code works as intended. White spaces can break label definitions if they are at the end of a label, e.g. "Label1 ". Putting white spaces before the opcode does not assemble correctly. White spaces inside the "[ ]" of a memory instruction does not work properly. When using blank lines to segment code, errors can occur if these lines contain white spaces. Using horizontal tabs to indent code does not work. Horizontal tabs inside code will not work since it is a different character to spaces, and the assembler uses spaces to separate operands.

M.5 Sources of Error and Other Uncertainties

Since it is code that is being tested, no external error sources are considered. From an understanding of the code, some known possible errors are described as follows.

White Spaces

A known source of possible errors comes from using horizontal tabs as indentation. Since these are different than white spaces, in ASCII code, this can break the assembler. This is because some

functionalities are based on the use of white spaces to separate e.g. registers and opcodes.

Overflow and Underflow

As expected overflow and underflow immediates results in errors. However currently there are no overflow and underflow handling, so it is up to the user to make sure that these errors do not occur.

M.6 Test table

This table is a compilation of all the tests that was conducted regarding the opcode test. This is done to give an example. The table consist of three columns. The first is the tested input. The Second is the expected output. The third is the actual output. All errors are marked red in the actual output column and the error is explained in the last column "Error Notes".

Table M.1: All tests conducted in the opcode test

#	Input	Expected output	Actual output	Error Notes
1	ADDI $r1 20 $r2	001101 00001 00010 0000000000010100	001101 00001 00010 0000000000010100	
2	SUBI $r1 20 $r2	001111 00001 00010 0000000000010100	001111 00001 00010 0000000000010100	
3	NEGI 20 $r1	010000 00000 00001 0000000000010100	010000 00000 0000000000010100	RD is not output resulting in less than 32-bit instruction
4	ANDI $r1 20 $r2	010001 00001 00010 0000000000010100	010001 00001 00010 0000000000010100	
5	XORI $r1 20 $r2	010011 00001 00010 0000000000010100	010011 00001 00010 0000000000010100	
6	ORI $r1 20 $r2	010010 00001 00010 0000000000010100	010010 00001 00010 0000000000010100	
7	MULTI $r1 20 $r2	010100 00001 00010 0000000000010100	010100 00001 00010 0000000000010100	
8	LSLI $r1 20 $r2	010101 00001 00010 0000000000010100	010101 00001 00010 0000000000010100	
9	LSRI $r1 20 $r2	010110 00001 00010 0000000000010100	010110 00001 00010 0000000000010100	
10	RASI $r1 20 $r2	010111 00001 00010 0000000000010100	010111 00001 00010 0000000000010100	
11	CMPI $r1 20	011100 00001 00000 0000000000010100	011100 00001 00000 0000000000010100	
12	MOVI $r1 20	011101 00000 00001 0000000000010100	011101 00000 00001 0000000000010100	
13	ADDR $r1 $r2 $r3	000001 00001 00011 000100 0000000000	000001 00001 00011 000100 0000000000	
14	SUBR $r1 $r2 $r3	000011 00001 00011 000100 0000000000	000011 00001 00011 000100 0000000000	
15	NEGR $r1 $r2	000100 00001 00010 000000 0000000000	000100 00001 00010 000000 0000000000	
16	ANDR $r1 $r2 $r3	000101 00001 00011 000100 0000000000	000101 00001 00011 000100 0000000000	
17	XORR $r1 $r2 $r3	000111 00001 00011 000100 0000000000	000111 00001 00011 000100 0000000000	
18	ORR $r1 $r2 $r3	000110 00001 00011 000100 0000000000	000110 00001 00011 000100 0000000000	
19	MULT $r1 $r2 $r3	001000 00001 00011 000100 0000000000	001000 00001 00011 000100 0000000000	
20	LSLR $r1 $r2 $r3	001001 00001 00011 000100 0000000000	001001 00001 00011 000100 0000000000	
21	LSRR $r1 $r2 $r3	001010 00001 00011 000100 0000000000	001010 00001 00011 000100 0000000000	
22	RASR $r1 $r2 $r3	001011 00001 00011 000100 0000000000	001011 00001 00011 000100 0000000000	
23	CMP $r1 $r2	011010 00001 00000 000100 0000000000	011010 00001 00000 000100 0000000000	
24	MOV $r1 $r2	011011 00001 00010 00000 00000000000	011011 00001 00010 00000 00000000000	
25	JMP #label1	100010 00000 00000 0000000000000001	100010 00000 00000 0000000000000001	
26	JMPNQ #label2	100101 00000 00000 0000000000000010	100101 00000 00000 0000000000000010	
27	JMPLE #label3	100100 00000 00000 0000000000000011	100100 00000 00000 0000000000000011	
28	JMPEQ #label4	100011 00000 00000 0000000000000100	100011 00000 00000 0000000000000100	
29	NOP	000000 00000 00000 00000 00000 000000	000000 00000 00000 00000 00000 000000	
30	LOAD $r1 [$r2]	011110 00010 00001 00000 00000 000000	011110 00010 00001 00000 00000 000000	
31	LOAD $r1 [$r2+20]	011110 00010 00001 0000000000010100	011110 00010 0000000000010100 00001	The immediate comes before register
32	LOAD $r1 [20]	011110 11101 00001 0000000000010100	011110 11101 00001 0000000000010100	
33	STORE $r1 [$r2]	011111 00010 00001 00000 00000 000000	011111 00010 00001 00000 00000 000000	
34	STORE $r1 [$r2+20]	011111 00010 00001 0000000000010100	011111 00010 00001 0000000000010100	
35	STORE $r1 [20]	011111 11101 00001 0000000000010100	011111 00001 11101 0000000000010100	RS1 and RD are switched
36	POP $r1	100000 00000 00001 0000 00000 0000000	100000 00000 00001 0000 00000 0000000	
37	PUSH $r1	100001 00000 00000 00001 00000 000000	100001 00000 00000 00001 00000 000000	
38	HALT	100110 00000 00000 00000 00000 000000	100110 00000 00000 00000 00000 000000	
39	FIRCOR $r1	100111 00001 00000 00000 00000 000000	100111 00001 00000 00000 00000 000000	
40	FIRCOI 20	101000 00000 00000 0000000000010100	101000 00000 00000 0000000000010100	
41	FIRSAR $r1 $r2	101010 00001 00010 00000 00000 000000	101010 00001 00010 00000 00000 000000	
42	FIRSAI 20 $r1	101011 00000 00001 0000000000010100	101011 00000 00001 0000000000010100	
43	FIRCORESET	101001 00000 00000 00000 00000 000000	101001 00000 00000 00000 00000 000000	
44	JMPR $r1	101100 00001 00000 00000 00000 000000	101100 00001 00000 00000 00000 000000	
45	JMPNQR $r1	101101 00001 00000 00000 00000 000000	101101 00001 00000 00000 00000 000000	
46	JMPLER $r1	101110 00001 00000 00000 00000 000000	101110 00001 00000 00000 00000 000000	
47	JMPEQR $r1	101111 00001 00000 00000 00000 000000	101111 00001 00000 00000 00000 000000	

Appendix N

Test Journal: PSoC System

This test journal will describe the tests conducted to ensure correct functionality of the complete PSoC system.

N.1 Test Purpose

The purpose of this test is to make sure the PSoC system works as intended. To do so, each module as described in section 9.1 will be tested to see if they function at once. For this test the custom PCB which has been designed in section 10 will be used throughout the test.

N.2 Test Frame

The complete PSoC design is programmed onto the device and all the appropriate connections between PCB and Analog Discovery 2 is connected. These connections are:

- Both positive scope channels are connected, the first one to the input of the delta sigma ADC and the second channel to the analog output of the I^2S DAC
- Ground
- Waveform generator is connected to the line-in connection

Test Setup and Equipment

The equipment used is:

- PSoC 5LP (CY8C5888LTI-LP097), KitProg version 2.2
- Adafruit UDA1334 I^2S DAC
- Custom PCB
- Digilent Analog Discovery 2 (Box no. 2179-00)

N.3 Executed Tests

To verify that the whole PSoC design works, every component must function at any given time during the test but not necessarily simultaneously with another component. The following are the executed tests:

- Test of the line-in delta sigma ADC sampling of a sine wave and the same output through the I^2S DAC
- Test of the microphone delta sigma ADC sampling and the same output through the I^2S DAC
- Test of sampling of a 12-bit number with the SAR ADC
- Test of LCD character addressing and displaying test
- Test of the three input buttons
- Test of the five hardware multiplexers
- Test of the eight LEDs reacting to the 12-bit ADC sample

N.4 Test Results

The test concluded with no apparent errors. Both inputs for the delta sigma ADC have been sampled and passed through to the I^2S DAC. The results from the Analog Discovery two can be found in section N.4.1. The input and output of the line-in test can be seen on figure N.1 and N.2 respectively. The input and output of the microphone test can be seen on figure N.3 and N.4 respectively.
This test also demonstrated that the first pair of multiplexers and button input completes its function. The input button for the I^2S and the trio of multiplexers also fulfilled their functionality. The mute button also worked by setting the mute pin of the I^2S DAC high when it was toggled as mute.
For good measure, all three of the buttons' toggle state were also displayed on the LCD which also helped verify debouncing hardware.
The SAR ADCs functionality was also verified by outputting its value to the LCD. The eight LEDs also lit up correctly depending on the given value that the SAR ADC was sampling.

N.4.1 Test Data

Presented below are the test result of both delta sigma ADC inputs. For both the line-in and the microphone there is a first figure which shows the input and a second figure which shows the output.

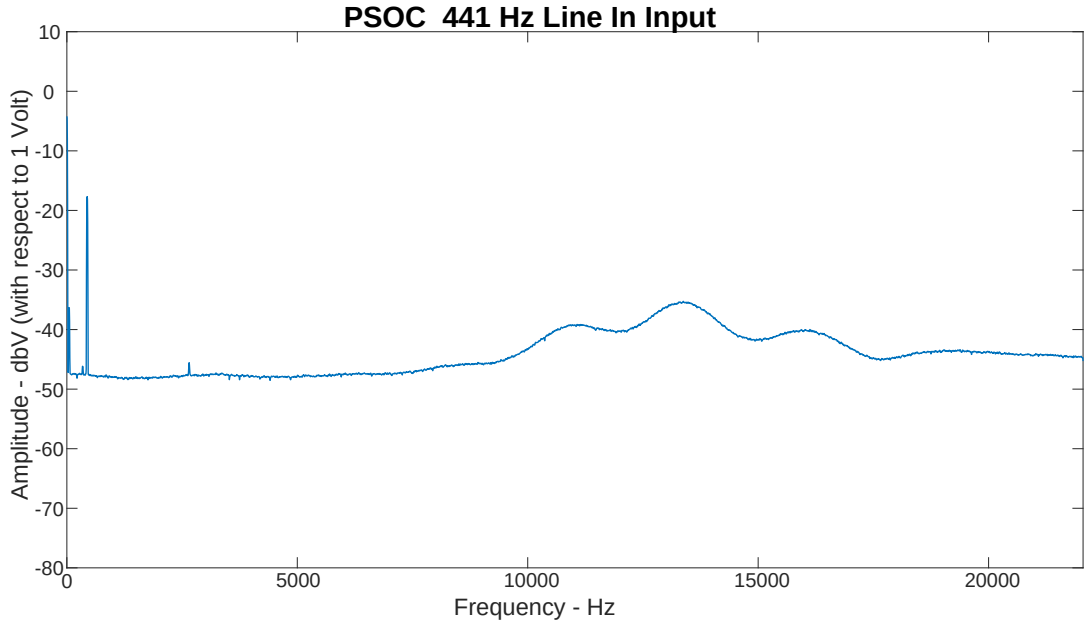


Figure N.1: FFT analysis of the 441 Hz sine wave which is the input of the line in.

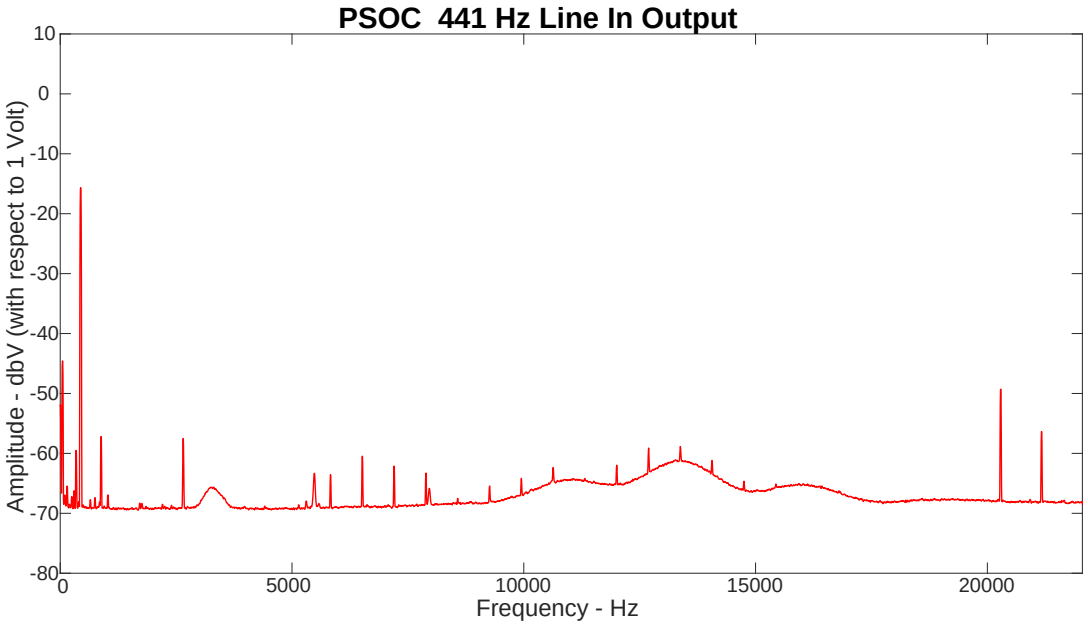


Figure N.2: FFT analysis of the output of the I^2S DAC with the ADC input shown in figure N.1

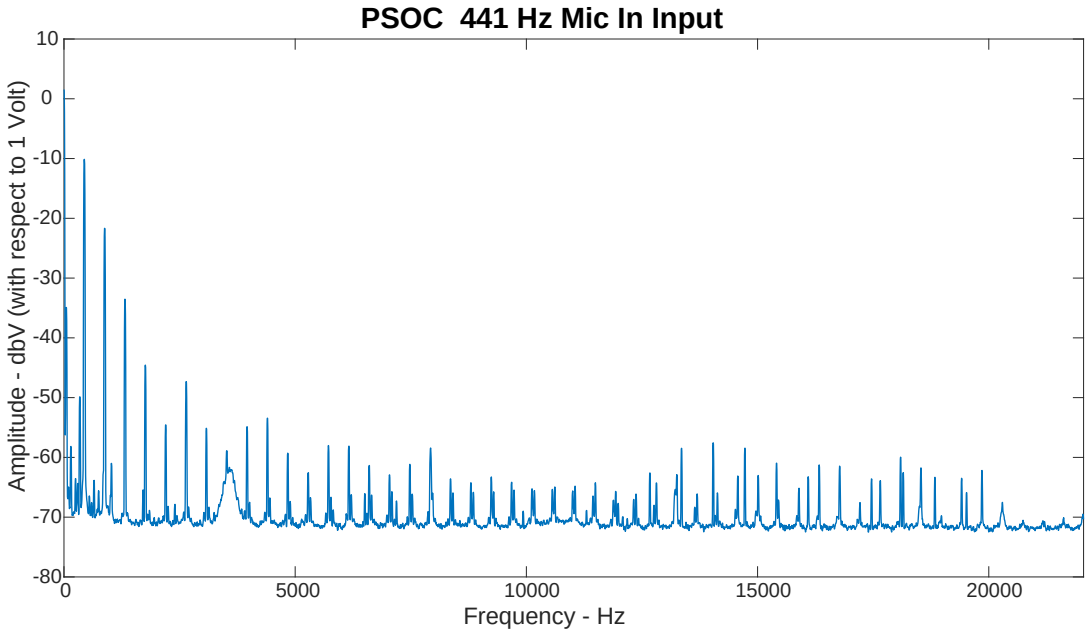


Figure N.3: FFT analysis of the 441 Hz sine wave which is the input of the microphone

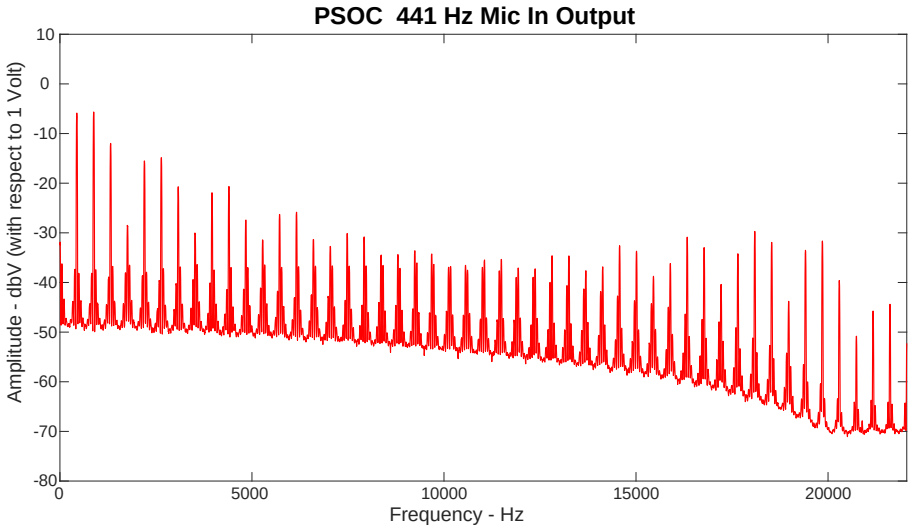


Figure N.4: FFT analysis of the output of the I^2S DAC with the ADC input shown in figure N.3

Appendix O

Full System Architecture

Figure is shown on the following page:

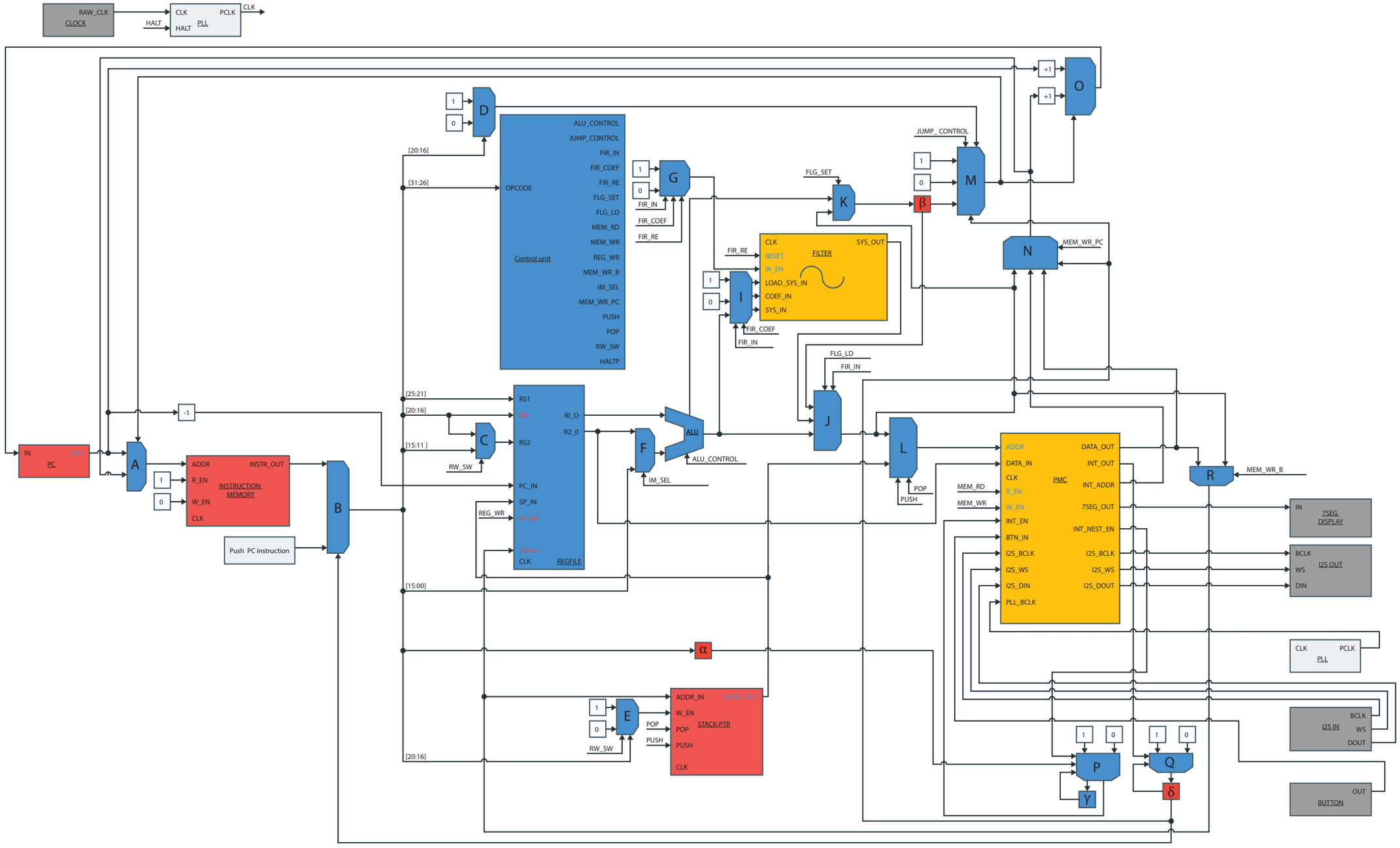


Figure O.1: The complete system Architecture

Appendix P

FPGA Code

The VHDL code for synthesizing the CPU is shown in the followincg sections. The below hierarchy tree shows the file structure of the project, and illustrates which files are dependent on one another. File name in italics are not including in the appendix, as they make use of Altera's IP core modules. Additionally, it should be noted that all written VHDL code needs to be parsed through a C pre processor before it can be synthesized. This is necessary since non-vhdl directives such as "defines" and "include" are used throughout the code. The first four top level files shown in the below hierarchy tree (Config, ControlConfig, FIRConfig, and I2SConfig) are not VHDL files, but are header files where configuration parameters are defined.

- Config
- ControlConfig
- FIRConfig
- I2SConfig
- Master
  - ALU
    - *Multiplier*
  - Control
  - Filter
  - *PLLSysClock*
  - *PLLI2S*
  - *ProgramRAM*
  - RegisterFile
  - Stack
  - PMC
    - I2SMonoIn
      - I2SSlave
    - I2SMonoOut
      - I2SMaster
    - Button
    - DataRAM
    - Interrupt
    - SevenSegmentDisplay
      - BinaryCodedDecimal

P.1 Config

```
#define INSTRUCTION_SIZE 32 //Size of the instruction
#define CONTROL_SIZE 24 //Amount of Control Signals
#define WORD_SIZE 16 //Size of each word
#define ADDRESS_SIZE 16 //Size of each address???
#define WORD_COUNT 8192 //How many words reserved in DRAM
#define PC_SIZE 13 //How many bits are used to represent PRAM (e.g. 10 bits is 1024)
#define ZERO_PAD "000" //Used for padding PC_SIZE to make sure that it fits in WORD_SIZE
```

P.2 Control

```
//ALU Operations
#define ADC 1 //ADD Carry
#define ADD 2 //ADD
#define SUB 3 //Subtract
#define MUL 4 //Multiply
#define OGG 5 //AND
#define ELL 6 //OR
#define XEL 7 //XOR
#define IKA 8 //Negates Operand A
#define IKB 9 //Negates Operand B
#define NOA 10 //Not Operand A
#define NOB 11 //Not Operand B
#define LSL 12 //Logical Left Shift
#define LSR 13 //Logical Right Shift
#define ASR 14 //Arithmetic Right Shift
#define PAS 15 //Pass Operand A
#define PBS 16 //Pass Operand B
#define ICA 17 //Increment Operand A
#define ICB 18 //Increment Operand B
#define NAA 19 //NOP

//Types of Branches
#define NB 0 //No Branching
#define BR 1 //Branch
#define EQ 2 //If Equal
#define LE 3 //If Less Than
#define NQ 4 //If Not Equal
#define PA 5 //If parity

//Control Signals Enabled
#define FIRIN_E '1' //Load Filter data
#define FIRCO_E '1' //Load Filter coefficient
#define FIRRE_E '1' //Filter reset
#define FLGSE_E '1' //Set Flags
#define FLGLO_E '1' //Load flags
#define MEMRD_E '1' //Memory Read
#define MEMWR_E '1' //Memory Write
#define REGWR_E '1' //Register Write
#define MEMRB_E '1' //Memory to Register Write
#define IMSEL_E '1' //Immediate Select
#define PUSHO_E '1' //PUSH
```

```
42 #define POP_E '1'   //POP
43 #define RWSWI_E '1' //Read/Write Switch
44 #define MW2PC_E '1' //Memopry to PC
45 #define HALTP_E '1' //HALT CPU Execution
46
47
48 //Control Signals Disabled
49 #define FIRIN_D '0'
50 #define FIRCO_D '0'
51 #define FIRRE_D '0'
52 #define FLGSE_D '0'
53 #define FLGLO_D '0'
54 #define MEMRD_D '0'
55 #define MEMWR_D '0'
56 #define REGWR_D '0'
57 #define MEMRB_D '0'
58 #define IMSEL_D '0'
59 #define PUSHO_D '0'
60 #define POP_D '0'
61 #define RWSWI_D '0'
62 #define MW2PC_D '0'
63 #define HALTP_D '0'
64
65 //Opcodes
66 #define NOP 0 //NOP
67 #define ADDR 1 //Add Register_A and Register_B
68 #define ADDCR 2 //Add Register_A and Register_B and carry
69 #define SUBR 3 //Subtract Register_B from Register_A
70 #define NEGR 4 //Negate Register_A
71 #define ANDR 5 //AND Register_A and Register_B
72 #define ORR 6 //OR Register_A and Register_B
73 #define XORR 7 //XOR Register_A and Register_B
74 #define MULTR 8 //Multiply Register_A and Register_B
75 #define LSLR 9 //Left Logical Shift Register_A, Register_B times
76 #define LSRR 10 //Right Logical Shift Register_A, Register_B times
77 #define RASR 11 //Right Arithmetic Shift Register_A, Register_B times
78 #define ADDI 13 //Add Register_A and an Immediate
79 #define ADDCI 14 //Add Register_A and an Immediate and Carry
80 #define SUBI 15 //Subtract an Immediate from Register_A
81 #define NEGI 16 //Negate an Immediate
82 #define ANDI 17 //AND Register_A and an Immediate
83 #define ORI 18 //OR Register_A and an Immediate
84 #define XORI 19 //XOR Register_A and an Immediate
```

```
85 #define MULTI 20 //Multiply Register_A and an Immediate
86 #define LSLI 21 //Left Logical Shift Register_A, Immediate times
87 #define LSRI 22 //Right Logical Shift Register_A, Immediate times
88 #define RASI 23 //Right Arithmetic Shift Register_A, Immediate times
89 #define CMP 26 //Compare Register_A with Register_B
90 #define MOV 27 //Move Register to Register
91 #define CMPI 28 //Compare Register_A with an immediate
92 #define MOVI 29 //Move an Immediate to a Regster
93 #define LOAD 30 //Load the data of a specific memory address into a register
94 #define STORE 31 //Store the contents of a register into memory
95 #define POP 32 //Pop the Top of the Stack into a Register
96 #define PUSH 33 //Push a register to the top of the stack
97 #define JMP 34 //Unconditional Jump
98 #define JMPEQ 35 //Jump if equal
99 #define JMPLE 36 //Jump if less than
100 #define JMPNQ 37 //Jump if not equal
101 #define HALT 38 //HALT CPU
102 #define FIRCOR 39 //Set Fir coefficient with register
103 #define FIRCOI 40 //Set Fir coefficient with immediate
104 #define FIRRESET 41 //Reset Fir Filter
105 #define FIRSAR 42 //Fir input from register
106 #define FIRSAI 43 //Fir input from immediate
107 #define JMPR 44 //Jump Register
108 #define JMPNQR 45 //Jump not equal register
109 #define JMPLER 46 //Jump less than register
110 #define JMPEQR 47 //Jump Equal Register
111 #define JMPPA 48 //Jump Parity
112 #define JMPPAR 49 //Jump Parity Register
113 #define GETFLAG 50 //Get flag values and save to register
114 #define SETFLAG 51 //Set flag values
115 #define SETFLAGR 54 //Set flag values from register
```

P.3 FirConfig

```
#define INOUT_BIT_WIDTH 16 //Data bit depth

#define COEFFICIENT_WIDTH 16 //Coefficient Bit depth

#define MULTIPLIER_WIDTH 32 //Multiplier bit depth (must be INOUT_BIT_WIDTH + COEFFICIENT_WIDTH of bit depth)

//Adder width = MULTIPLIER_WIDTH+log2(TAPS)-1
#define ADDER_WIDTH 32

//Should be multiples of two, to generate integer values of adder width
#define TAPS 64

//For test bench
#define INPUT_SAMPLES 401
```

P.4 I2SConfig

```
1 #define DATA_WIDTH 16 //How many bits for one piece of data
```

P.5 Master

```
1  #include "Config.hvhd"
2  #include "FIR_Config.hvhd"
3
4  --Definition of control lines groupings
5  #define ALU_CONTROL 23 DOWNTO 18
6  #define JUMP_CONTROL 17 DOWNTO 15
7  #define FLAG_SET 14
8  #define FLAG_LOAD 13
9  #define FIR_LOAD_SAMPLE 12
10 #define FIR_LOAD_COEFFICIENT 11
11 #define FIR_RESET 10
12 #define MEMORY_READ 9
13 #define MEMORY_WRITE 8
14 #define REGISTER_WRITE 7
15 #define MEMORY_WRITE_BACK 6
16 #define IMMEDIATE_SELECT 5
17 #define PUSH 4
18 #define POP 3
19 #define SWITCH_READ_WRITE 2
20 #define MEMORY_TO_PC 1
21 #define HALT 0
22
23 --Definition of instruction lines groupings
24 #define OPCODE 31 DOWNTO 26
25 #define REGISTER_READ_INDEX_1 25 DOWNTO 21
26 #define REGISTER_READ_INDEX_2 15 DOWNTO 11
27 #define REGISTER_WRITE_INDEX_1 20 DOWNTO 16
28 #define IMMEDIATE 15 DOWNTO 0
29
30 #define PUSH_PC "10000100000000001111100000000000"
31 #define POP_PC "10000000000111110000000000000000"
32
33 LIBRARY IEEE;
34 USE IEEE.STD_LOGIC_1164.ALL;
35 USE IEEE.STD_LOGIC_unsigned.ALL;
36 USE IEEE.NUMERIC_STD.ALL;
37 ENTITY Master IS
38     PORT
39     (
40         clk    : IN std_logic;                          --Device hardware clock
41         btn    : IN std_logic_vector(2 DOWNTO 0);       --Device's 3 available push buttons (note: active low)
```

```
42          sseg   : OUT std_logic_vector(31 DOWNTO 0);  --Seven segment display control signals (8 signals for each of the four
                displays)
43
44          led    : OUT std_logic_vector(9 DOWNTO 0) := (OTHERS => '0');    --Signals for controlling onboard LED's
45
46          --I2S input
47          bclk   : IN std_logic  := '0';               --External input bitclock signal
48          ws     : IN std_logic  := '0';               --External input word select signal
49          Din    : IN std_logic  := '0';               --External Data input
50
51          --I2S output
52          bclkO  : OUT std_logic := '0';               --Bitclock output
53          wsO    : OUT std_logic := '0';               --Word select output
54          DOut   : OUT std_logic := '0'                --data output
55
56
57      );
58  END ENTITY Master;
59
60  ARCHITECTURE Behavioral OF Master IS
61      SIGNAL btn_inverted : std_logic_vector(2 DOWNTO 0) := (OTHERS => '0'); --For stroing inverted button signal
62
63      --Control and instruction registers
64      SIGNAL control_signals : Std_logic_vector(CONTROL_SIZE-1 DOWNTO 0) := (OTHERS => '0'); --Control signals generated by the
            control block
65      SIGNAL instruction : std_logic_vector(INSTRUCTION_SIZE-1 DOWNTO 0) := (OTHERS => '0'); --instruction sent to the system
66      SIGNAL pram_data_out : std_logic_vector(INSTRUCTION_SIZE-1 DOWNTO 0) := (OTHERS => '0'); --instruction oututted by the
            program memory
67
68      --Wires
69      SIGNAL operand_a, operand_b, alu_output : std_logic_vector(WORD_SIZE-1 DOWNTO 0); -- ALU inputs and output wires
70      SIGNAL stack_controller_out : std_logic_vector(WORD_SIZE-1 DOWNTO 0); --Stackpointer address wires
71      SIGNAL register_writeback : std_logic_vector(WORD_SIZE-1 DOWNTO 0); --Wires to connect ALU/Memory output to register
            writeback logic
72      SIGNAL dram_address_index : std_logic_vector(WORD_SIZE-1 DOWNTO 0); --Wires connected to the memory controllers address
            port
73      SIGNAL r2_w1_switch : std_logic_vector(4 DOWNTO 0); --Size '5' to be able to index register. Is used to switch between
            indexing Read_2 and Write_1 register
74
75      --PRAM Signals
76      SIGNAL PC : std_logic_vector(PC_SIZE-1 DOWNTO 0) := (OTHERS => '0'); --Program Counter
77      SIGNAL pram_address_index : std_logic_vector(PC_SIZE-1 DOWNTO 0) := (OTHERS => '0'); --Wires connected to the PRAM's
            address port
```

```
78  SIGNAL PC_ALT : std_logic_vector(PC_SIZE-1 DOWNTO 0) := (OTHERS => '0'); --Alternative new PC (e.g. ALU/Memory Output), used when changing the PC (for jumps)
79  SIGNAL interrupt_address : std_logic_vector(PC_SIZE-1 DOWNTO 0) := (OTHERS => '0'); --Interrupt address, address that the ISR points to
80  SIGNAL pDataIn : std_logic_vector(INSTRUCTION_SIZE-1 DOWNTO 0); --Not used in current implementation, used to write to program memory
81  SIGNAL pDataOut : std_logic_vector(INSTRUCTION_SIZE-1 DOWNTO 0); --Program Memory instruction output
82  SIGNAL previous_instruction : std_logic_vector(INSTRUCTION_SIZE-1 DOWNTO 0);
83  SIGNAL pram_write_enable : std_logic := '0'; --Program memory write enable disabled in current implementation
84  SIGNAL pram_read_enable : std_logic := '1'; --Program memory read enable always on in current implementation
85
86  --DRAM Signals
87  SIGNAL dram_data_out : std_logic_vector(WORD_SIZE-1 DOWNTO 0); --Data RAM output data
88  SIGNAL dram_data_in : std_logic_vector(WORD_SIZE-1 DOWNTO 0); --Data RAM Input data, either source_register_2_output, or R2O-2 (needed for proper interrupt implementation)
89
90  SIGNAL fir_coefficient_in : std_logic_vector(INOUT_BIT_WIDTH-1 downto 0) := (others=>'0'); --Coefficients to be send to filter
91  SIGNAL fir_data_in : std_logic_vector(INOUT_BIT_WIDTH-1 downto 0) := (others=>'0'); --audio data to send to the filter to be filtered
92  SIGNAL fir_data_out : std_logic_vector(INOUT_BIT_WIDTH-1 downto 0) := (others=>'0'); --fir output data (i.e. filtered data)
93  SIGNAL fir_load_data_enable : std_logic := '0'; --Saves coefficients to the filter when low, saves data when high
94  SIGNAL fir_write : std_logic := '0'; --Write data control signal for the filter
95
96  SIGNAL processing_output : std_logic_vector(WORD_SIZE-1 DOWNTO 0); --ALU/filter total output
97
98  SIGNAL source_register_2_output : STD_logic_vector(WORD_SIZE-1 DOWNTO 0); --2nd indexed register output, needed as a buffer to be able to switch bewtween register_2 and Immediate input to the alu
99
100 SIGNAL jmp_enable : std_logic := '0'; --Is '0' when PC increments by 1, is set to '1' when jump occours
101 SIGNAL pc_overwrite, sp_overwrite : std_logic := '0'; --SP and PC are special registers, and PC/sp_overwrite needs to be '1' to be able to change their values
102
103 SIGNAL pc_register_file_input : std_logic_vector(WORD_SIZE-1 DOWNTO 0); --Routes PC+1 into the register file
104
105 SIGNAL interrupt_cpu : std_logic := '0'; --Interrupt signal for the CPU
106 SIGNAL Interrupt_latch : std_logic := '0'; --Latch for the interrupt signal, ensures signal stays on for an additional clock cycle    (Varaible can not use small letters for some reason)
107 SIGNAl interrupt_enable : std_logic := '1'; --Enables interrupts when high, disables when low
108 SIGNAL interrupt_nest_enable : std_logic := '1'; --Enable nested interrupts when high, disabled when low
109 SIGNAL interrupt_nest_enable_latch : std_logic := '0'; --Latches "interrupt_nest_enable" signal
110
```

```
111     --FLAGS
112     SIGNAL parity_flag                      : std_logic                     := '0';
113     SIGNAL signed_flag                      : std_logic                     := '0';
114     SIGNAL overflow_flag                    : std_logic                     := '0';
115     SIGNAL zero_flag                        : std_logic                     := '0';
116     SIGNAL carry_flag                       : std_logic                     := '0';
117
118     --FLAG LATCHES
119     SIGNAL parity_flag_latch                : std_logic                     := '0';
120     SIGNAL signed_flag_latch                : std_logic                     := '0';
121     SIGNAL overflow_flag_latch              : std_logic                     := '0';
122     SIGNAL zero_flag_latch                  : std_logic                     := '0';
123     SIGNAL carry_flag_latch                 : std_logic                     := '0';
124
125     SIGNAL sys_clk : std_logic; --Clock that controls the system, can either be assigned to the normal clock (for simulation),
            or pll_tmp_clk
126     SIGNAL pll_clk : std_logic; --PLL Clock
127     SIGNAL pll_clk_i2s : std_logic; --PLL Clock
128     SIGNAL pll_lock : std_logic; --PLL lock signal
129     SIGNAL pll_i2s_lock : std_logic;
130     SIGNAL pll_tmp_clk : std_logic; --Is assigned the pll_clk when pll_lock is detected
131     SIGNAL pll_i2s_tmp_clk : std_logic; --I2S clock set when PLL lock signal is set
132     SIGNAL clk_counter : std_logic_vector(2 DOWNTO 0); --Clock divider, used to switch LED (works as a clock heart beat)
133
134     SIGNAL FLAG_FITLER : std_logic_vector(1 downto 0); --delete
135 BEGIN
136
137     PLL : ENTITY work.PLL(SYN)
138         PORT MAP(
139             inclk0 => clk,
140             c0 => pll_clk,
141             locked => pll_lock
142         );
143
144     PLL_i2s : ENTITY work.PLL_i2s(SYN)
145     PORT MAP(
146         inclk0 => clk,
147         c1 => pll_clk_i2s,
148         locked => pll_i2s_lock
149     );
150     MEMCNT : ENTITY work.MemoryController
151         PORT MAP(
152             write_enable => control_signals(MEMORY_WRITE),
```

```
153             read_enable => control_signals(MEMORY_READ),
154             address => dram_address_index,
155             data_in => dram_data_in,
156             data_out => dram_data_out,
157             CLK => sys_clk,
158             btn => btn_inverted,
159             seven_seg_control_signals => sseg,
160             interrupt_address => interrupt_address,
161             interrupt_cpu => interrupt_cpu,
162             interrupt_enable => interrupt_enable,
163             interrupt_nest_enable => interrupt_nest_enable,
164             i2s_bit_clk => bclk,
165             i2s_bit_clk_2 => pll_i2s_tmp_clk,
166             --i2s_bit_clk_2 => bclk,
167             i2s_word_select => ws,
168             i2s_data_in => Din,
169             i2s_bit_clk_out => bclkO,
170             i2s_word_select_out => wsO,
171             i2s_data_out => DOut
172         );
173
174     CONTROLLER : ENTITY work.Control(Behavioral)
175         PORT MAP(
176             opcode => instruction(OPCODE),
177             control_signals => control_signals
178         );
179
180     STACK : ENTITY work.Stack(Behavioral)
181         PORT MAP(
182             pop => control_signals(POP),
183             push => control_signals(PUSH),
184             clk => sys_clk,
185             address_out => stack_controller_out,
186             address_in => register_writeback,
187             write_back => sp_overwrite
188         );
189
190     ALU : ENTITY work.ALU(Behavioral)
191         PORT MAP(
192             operation => control_signals(ALU_CONTROL),
193             operand_a => operand_a,
194             operand_b => operand_b,
195             result => alu_output,
```

```
196             parity_flag => Parity_Flag,
197             signed_flag => Signed_Flag,
198             overflow_flag => Overflow_Flag,
199             zero_flag => Zero_Flag,
200             carry_flag => Carry_Flag
201         );
202     FIR : ENTITY work.Filter(Behavioural)
203         PORT MAP(
204         clk     => sys_clk,
205         load_system_input => fir_load_data_enable,
206         reset   => control_signals(FIR_RESET),
207         system_input    => fir_data_in,
208         coefficient_in => fir_coefficient_in,
209         system_output  => fir_data_out,
210         write_enable => fir_write
211         );
212
213     PRAM : ENTITY work.MemAuto(SYN)
214         PORT MAP(
215             data => pDataIn,
216             q => pram_data_out,
217             address => pram_address_index,
218             wren => pram_write_enable,
219             rden => pram_read_enable,
220             clock => sys_clk
221         );
222     REGS : ENTITY work.RegistryInternal(Behavioral)
223         PORT MAP(
224             read_register_a_index => instruction(REGISTER_READ_INDEX_1),
225             write_register_index => instruction(REGISTER_WRITE_INDEX_1),
226             read_register_b_index => r2_w1_switch,
227
228             register_file_data_in => register_writeback,
229
230             register_file_data_out_a => operand_a,
231             register_file_data_out_b => source_register_2_output,
232
233             pc_value_input => pc_register_file_input,
234             sp_value_input => stack_controller_out,
235
236             write_enable => control_signals(REGISTER_WRITE),
237
238             clk => sys_clk
```

```
239         );
240
241
242
243
244     dram_data_in <= source_register_2_output; --Sets DRAM input to RS2
245
246     ---------------------------------------------
247     -- ImmediateOperand:
248     -- If the Immediate control signal is set,
249     -- then the ALU's B operand is assigned the lowest 16 bit of the executing instruction
250     -- as opposed to the register files 2nd output.
251     -- This is done to allow register-immediate arithmatic,
252     -- as opposed to only register-register arithmatic
253     ---------------------------------------------
254     ImmediateOperand : WITH control_signals(IMMEDIATE_SELECT) SELECT operand_b <= -- Selects Register output 2 or Immediate
255     source_register_2_output WHEN '0',
256     instruction(IMMEDIATE) WHEN '1',
257     source_register_2_output WHEN OTHERS;
258
259     ---------------------------------------------
260     -- SwitchReadWriteIndex:
261     -- Under normal operation the "read_register_b_index" is indexed by "instruction(15 DOWNTO 11)",
262     -- except when excuting a "store" instruction (this sets "SWITCH_READ_WRITE" high). In that case
263     -- it is indexed by "instruction(20 DOWNTO 16)"
264     --
265     -- This is necessary, as the "store" instruction is the only instruction requiring an immediate value as well
266     -- as the data from two registers (and no destination registers to writeback data to).
267     --
268     -- To elaborate, the standard grouping of instruction signals used for indexing "read_register_b_index" overlaps with with
269     -- the immediate signal grouping "instruction(15 DOWNTO 0)" and can therfore not be used. Instead the grouping
270     -- conventionally used for index the write index "instruction(20 DOWNTO 16)" is used since no writeback is required
271     -- for the "store" instruction
272     ---------------------------------------------
273     SwitchReadWriteIndex : WITH control_signals(SWITCH_READ_WRITE) SELECT r2_w1_switch <=
274     instruction(REGISTER_WRITE_INDEX_1) WHEN '1',
275     instruction(REGISTER_READ_INDEX_2) WHEN OTHERS;
276
277     PROCESS(control_signals(FIR_LOAD_SAMPLE),control_signals(FIR_LOAD_COEFFICIENT),control_signals(FIR_RESET))
278     BEGIN
279         IF(control_signals(FIR_LOAD_SAMPLE) = '1' OR control_signals(FIR_LOAD_COEFFICIENT) = '1' OR control_signals(FIR_RESET)
                = '1') THEN
280             fir_write <= '1';
```

```
281         ELSE
282             fir_write <= '0';
283         END IF;
284     END PROCESS;
285 
286     PROCESS(control_signals(FIR_LOAD_SAMPLE),control_signals(FIR_LOAD_COEFFICIENT), alu_output, operand_a, operand_b)
287     BEGIN
288         IF(control_signals(FIR_LOAD_SAMPLE) = '1') THEN
289             fir_load_data_enable <= '1';
290             fir_data_in <= alu_output;
291         ELSIF(control_signals(FIR_LOAD_COEFFICIENT) = '1') THEN
292             fir_load_data_enable <= '0';
293             fir_coefficient_in <= alu_output;
294         ELSE
295             fir_load_data_enable <= '0';
296         END IF;
297     END PROCESS;
298 
299     WITH control_signals(FLAG_LOAD DOWNTO FIR_LOAD_SAMPLE) SELECT processing_output <=
300         fir_data_out WHEN "01",
301         fir_data_out WHEN "11", --replicated since the cpu doesn't function without this
302         "00000000000" & carry_flag_latch & parity_flag_latch & signed_flag_latch & overflow_flag_latch & zero_flag_latch WHEN
                "10",
303         alu_output WHEN OTHERS;
304 
305     ---------------------------------------
306     -- MemoryRegisterWrite:
307     -- Process controls what data is written to the register file.
308     -- If "MEMORY_WRITE_BACK" is high, the dram output is written,
309     -- otherwise the ALU output is written
310     ---------------------------------------
311     MemoryRegisterWrite : WITH control_signals(MEMORY_WRITE_BACK) SELECT register_writeback <=
312     dram_data_out WHEN '1',
313     processing_output  WHEN OTHERS;
314 
315     ---------------------------------------
316     -- PcOvewriteEnable:
317     -- Sets "pc_overwrite" if 31 (the register index corresponding to the PC register) is indexed TODO: Implement like SP?
318     ---------------------------------------
319     PcOvewriteEnable : WITH to_integer(unsigned(instruction(REGISTER_WRITE_INDEX_1))) SELECT pc_overwrite <=
320     '1' WHEN 31,
321     '0' WHEN OTHERS;
322 
```

```
323     ------------------------------------------
324     -- SpOvewriteEnable:
325     -- Sets "sp_overwrite" if 30 (the register index corresponding to the SP register) is indexed
326     ------------------------------------------
327     SpOvewriteEnable : PROCESS (instruction(REGISTER_WRITE_INDEX_1), control_signals(SWITCH_READ_WRITE))
328     BEGIN
329         IF (to_integer(unsigned(instruction(REGISTER_WRITE_INDEX_1))) = 30 AND control_signals(SWITCH_READ_WRITE) /= '1') THEN
330             sp_overwrite <= '1';
331         ELSE
332             sp_overwrite <= '0';
333         END IF;
334     END PROCESS;
335
336
337     ------------------------------------------
338     -- DramAddressIndex:
339     -- If POP/PUSH instruction is exectuing, then index memory based on Stack Controller
340     -- Otherwise index based on ALU output
341     ------------------------------------------
342     DramAddressIndex : PROCESS (control_signals(POP), control_signals(PUSH), stack_controller_out, processing_output )
343         VARIABLE TMP : std_logic_vector(1 DOWNTO 0);
344     BEGIN
345         TMP := control_signals(POP) & control_signals(PUSH);
346         IF (to_integer(UNSIGNED(TMP)) > 0) THEN
347             dram_address_index <= stack_controller_out;
348         ELSE
349             dram_address_index <= processing_output ;
350         END IF;
351     END PROCESS;
352
353     ------------------------------------------
354     -- JumpEnable:
355     -- enables/disables "jmp_enable" based
356     -- on various control signals
357     ------------------------------------------
358     JumpEnable : WITH to_integer(unsigned(Interrupt_latch & pc_overwrite & control_signals(JUMP_CONTROL))) SELECT jmp_enable
            <= --Controls branching/changing PC
359     '0' WHEN 0, --If not signals are high, then do not enable branching
360     '1' WHEN 1, --unconditional jump
361     zero_flag_latch WHEN 2, --jmpeq
362     carry_flag_latch WHEN 3, --jmple
363     NOT zero_flag_latch WHEN 4, --jmpnq
364     parity_flag_latch WHEN 5,
```

```
365     '1' WHEN 8, --Jump (change pc) when pc_overwrite is set
366     '1' WHEN 16, --Jump When Interrupt_latch is set
367     '0' WHEN OTHERS;
368
369     ---------------------------------------------
370     -- SetAlternativePc:
371     -- Sets pc_alt based on interrupt status and the MEMORY_TO_PC control line
372     ---------------------------------------------
373     SetAlternativePc : PROCESS (Interrupt_latch, control_signals(MEMORY_TO_PC), dram_data_out(PC_SIZE-1 DOWNTO 0),
            processing_output (PC_SIZE-1 DOWNTO 0))
374     BEGIN
375         IF (Interrupt_latch = '1') THEN
376             pc_alt <= interrupt_address;
377         ELSIF (control_signals(MEMORY_TO_PC) = '1') THEN
378             pc_alt <= dram_data_out(PC_SIZE-1 DOWNTO 0);
379         ELSE
380             pc_alt <= processing_output (PC_SIZE-1 DOWNTO 0);
381         END IF;
382     END PROCESS;
383
384     ---------------------------------------------
385     -- PramAddressSource:
386     -- Choses instruction to be loaded based on branching.
387     -- Loads "pc_alt" if "jump_enabled" is high
388     -- Otherwise load "pc"
389     ---------------------------------------------
390     PramAddressSource : WITH jmp_enable SELECT pram_address_index <=
391         pc_alt WHEN '1', --Load pc alt if jump enabled
392         pc WHEN OTHERS; --otherwise load pc
393
394
395
396
397     ---------------------------------------------
398     -- InstructionSource:
399     -- Changes instruction source if interrupt is detected
400     ---------------------------------------------
401     InstructionSource : PROCESS (Interrupt_latch, pram_data_out)
402     BEGIN
403         IF (Interrupt_latch = '1') THEN
404             instruction <= PUSH_PC; --if interrupt is set, then sets the output instruction to PUSH PC (bypassing PRAM output)
405         ELSE
406             instruction <= pram_data_out; --Otherwise set the output instruction to PRAM output
```

```
407         END IF;
408     END PROCESS;
409
410
411     ---------------------------------------------
412     -- SetPc:
413     -- Chose new value of PC based on branching/interrupts
414     ---------------------------------------------
415     SetPc : PROCESS (sys_clk) --Chose new value of PC based on branching
416     BEGIN
417         IF (rising_edge(sys_clk)) THEN
418             IF (jmp_enable /= '1') THEN
419                 pc <= std_logic_vector(unsigned(pc) + 1); --Most common mode, simply increments PC every clock cycle
420             ELSE
421                 pc <= std_logic_vector(unsigned(pc_alt) + 1); --If jump was performed, then set PC to "PC_ALT+1" (NOT "PC_ALT
                        ", as that is directly passed to the PRAM address index
422             END IF;                                           --If "PC_ALT" were to be passed direcly, then the instruction
                    would run twice)
423         END IF;
424     END PROCESS;
425
426     ---------------------------------------------
427     -- EnableInterrupts:
428     -- Enables/disables interrupts based on "interrupt_nest_enable" (send from the interrupt peripheral)
429     ---------------------------------------------
430     EnableInterrupts : PROCESS (instruction, interrupt_nest_enable)
431     BEGIN
432         IF (interrupt_nest_enable = '0' AND interrupt_nest_enable_latch = '0') THEN --latching to ensure that the IF statement
                doesn't run an infinite loop when "interrupt_nest_enable" is set low
433             interrupt_enable           <= '0';
434             interrupt_nest_enable_latch <= '1';
435         ELSIF (previous_instruction = POP_PC) THEN --Resets interrupts when POP_PC is executed
436             interrupt_enable           <= '1';
437             interrupt_nest_enable_latch <= '0';
438         END IF;
439     END PROCESS;
440
441     ---------------------------------------------
442     -- LatchInterrupt:
443     -- latches interrupt_cpu and sets the PC top be pushed to the stack
444     ---------------------------------------------
445     LatchInterrupt : PROCESS (sys_clk)
446     BEGIN
```

```
447         IF (rising_edge(sys_clk)) THEN
448             IF (interrupt_cpu = '1') THEN
449                 Interrupt_latch <= '1';
450             ELSIF (Interrupt_latch = '1') THEN --Disables latc on the following clock cycle
451                 Interrupt_latch <= '0';
452             END IF;
453         END IF;
454     END PROCESS;
455 
456     ----------------------------------------
457     -- FlagLATCH:
458     -- Latches Flags in temp register, neccesarry due to clock timing
459     ----------------------------------------
460     FlagLATCH : PROCESS (sys_clk)
461     BEGIN
462         IF (rising_edge(sys_clk)) THEN
463             CASE control_signals(FLAG_SET) is
464             WHEN '1' =>
465                 zero_flag_latch     <= processing_output(0);
466                 overflow_flag_latch <= processing_output(1);
467                 signed_flag_latch   <= processing_output(2);
468                 parity_flag_latch   <= processing_output(3);
469                 carry_flag_latch    <= processing_output(4);
470             WHEN OTHERS =>
471                 zero_flag_latch     <= zero_flag;
472                 overflow_flag_latch <= overflow_flag;
473                 signed_flag_latch   <= signed_flag;
474                 parity_flag_latch   <= parity_flag;
475                 carry_flag_latch    <= carry_flag;
476             END CASE;
477         END IF;
478     END PROCESS;
479 
480     pc_register_file_input <= ZERO_PAD & std_logic_vector(unsigned(PC) - 1);
481 
482     PROCESS(sys_clk)
483     BEGIN
484         IF(rising_edge(sys_clk)) THEN
485             previous_instruction <= instruction;
486         END IF;
487     END PROCESS;
488 
489     ----------------------------------------
```

```
490     -- SysClockSelect:
491     -- Controls "sys_clk" source
492     ------------------------------------------------
493     SysClockSelect : WITH control_signals(HALT) SELECT sys_clk <=
494     pll_tmp_clk WHEN '0',
495     '0' WHEN OTHERS;
496
497     ------------------------------------------------
498     -- WaitForPllLock:
499     -- Only allows pll_clk to be accessible when a PLL lock is ensured
500     ------------------------------------------------
501     WaitForPllLock : WITH pll_lock SELECT pll_tmp_clk <=
502         pll_clk WHEN '1',
503         '0' WHEN OTHERS;
504
505     WaitForPllI2sLock : WITH pll_i2s_lock SELECT pll_i2s_tmp_clk <=
506         pll_clk_i2s WHEN '1',
507         '0' WHEN OTHERS;
508     ------------------------------------------------
509     -- ClockCount:
510     -- Counts system clock rising edges
511     ------------------------------------------------
512     ClockCount : PROCESS (pll_clk)
513     BEGIN
514         IF (rising_edge(pll_clk)) THEN
515             clk_counter <= std_logic_vector(unsigned(clk_counter) + 1);
516         END IF;
517     END PROCESS;
518
519     btn_inverted <= NOT btn;
520
521 END ARCHITECTURE Behavioral;
```

P.6 ALU

```
1  #include "Control.hvhd"
2  #include "Config.hvhd"
3
4  ----------------------------------------------------------------------------------
5  ----- Module Name: ALU
6  --
7  --
8  ----- Description:
9  -- This is a simple ALU.
10 -- It has:
11 -- OPERATIONS:
12 -- Add
13 -- Sub
14 -- Multiplier
15 -- AND
16 -- OR
17 -- XOR
18 -- Negate A
19 -- Negate B
20 -- Logic shift left
21 -- Logic shift right
22 -- Arith shift right
23 -- Pass through
24 -- ICA Increments A
25 -- ICB Increments B
26 -- NOP
27 --
28 --The ALU does not depend on a clock signal
29 --
30 -- FLAGS:
31 -- Zero flag
32 -- Overflow flag
33 -- Signed flag
34 -- Parity flag
35 -- Carry flag
36 ----------------------------------------------------------------------------------
37
38 LIBRARY IEEE;
39 USE IEEE.STD_LOGIC_1164.ALL;
40 USE IEEE.NUMERIC_STD.ALL;
41 USE ieee.numeric_std.ALL;
```

```
42
43
44 ENTITY ALU IS
45
46     PORT (
47         operand_a, operand_b : IN std_logic_vector(15 DOWNTO 0); -- Operands 1 and 2
48         operation            : IN std_logic_vector(5 DOWNTO 0); --operation to perform
49
50         overflow_flag        : OUT std_logic; -- Flag raised when overflow is present
51         signed_flag          : OUT std_logic; -- Flag raised when negative result
52         zero_flag            : OUT std_logic; -- Flag raised when result is zero
53         parity_flag          : OUT std_logic; -- Flag raised when number of 1's in result is odd.
54         carry_flag           : OUT std_logic; -- Flag raised when carry is present
55
56         result               : OUT std_logic_vector(15 DOWNTO 0) --Output based on the two operands and operation port
57     );
58 END ENTITY ALU;
59
60 ARCHITECTURE Behavioral OF ALU IS
61
62     SIGNAL mult_temp     : std_LOGIC_VECTOR(31 DOWNTO 0); -- Used to store results from multiplier
63     SIGNAL temp          : std_logic_vector(16 DOWNTO 0); -- Used to store results inside the module.
64
65 BEGIN
66
67     --Altera Multiplier IP Core
68     multiplier : ENTITY work.Multiplier_1
69         PORT MAP(
70             dataa  => operand_a,                    -- Input 1 to the multiplier
71             datab  => operand_b,                    -- Input 2 to the multiplier
72             result => mult_temp                     -- Output of the multiplier
73         );
74
75
76     ---------------------------------------
77     -- Arithmetic:
78     -- Performs ALU operations.
79     --
80     -- Uses the two operands in combination
81     -- with the operation signal to calculate
82     -- an output
83     ---------------------------------------
84     Arithmetic : PROCESS (operand_a, operand_b, operation, temp, mult_temp) IS
```

```
85          VARIABLE Parity : std_logic;
86      BEGIN
87          temp          <= (OTHERS => '0');
88          IF (to_integer(unsigned(operation)) = NAA) THEN
89
90          ELSE
91              CASE to_integer(unsigned(operation)) IS
92                  WHEN ADD => -- Returns operand_a + operand_b
93                      temp   <= std_logic_vector(unsigned("0" & operand_a) + unsigned(operand_b)); -- Operands typecast as
                            unsigned to calculate carry correctly.
94                                                                                                    -- A '0' is appended to one
                                                                                                         operand before the
                                                                                                         operation is done
95                                                                                                    -- to fit the signal length
                                                                                                         of temp. The MSB contains
                                                                                                         the carry.
96                      result <= temp(15 DOWNTO 0);
97
98                  WHEN SUB => -- Returns operand_a - operand_b
99                      temp   <= std_logic_vector(unsigned("0" & operand_a) - unsigned(operand_b));
100                     result <= temp(15 DOWNTO 0);
101
102                 WHEN MUL => -- Returns operand_a * operand_b
103                     temp   <= ("0" & (mult_temp(15 DOWNTO 0)));                                   -- When opcode for mult is
                            chosen, output from multiplier is
104                                                                                                   -- routed to the output of
                                                                                                         the ALU.
105                                                                                                   -- A '0' is appended after
                                                                                                         the operation is done to
                                                                                                         fit the result
106                                                                                                   -- in the signal temp. It's
                                                                                                         done after the operation
                                                                                                         as carry is
107                                                                                                   -- determined using mult_temp
                                                                                                          instead of temp
108                     result <= temp(15 DOWNTO 0);
109
110                 WHEN OGG => -- Returns operand_a AND operand_b
111                     temp   <= ("0" & (operand_a AND operand_b));
112                     result <= temp(15 DOWNTO 0);
113
114                 WHEN ELL => -- Returns operand_a OR operand_b
115                     temp   <= ("0" & (operand_a OR operand_b));
```

```
116             result <= temp(15 DOWNTO 0);
117
118         WHEN XEL => -- Returns operand_a XOR operand_b
119             temp   <= ("0" & (operand_a XOR operand_b));
120             result <= temp(15 DOWNTO 0);
121
122         WHEN IKA => -- Negates operand A
123             temp   <= std_logic_vector("0" & ((NOT signed(operand_a)) + "0000000000000001")); -- Negation is done by
                    flipping all bits and then adding 1.
124             result <= temp(15 DOWNTO 0);
125
126         WHEN IKB => -- Negates operand A
127             temp   <= std_logic_vector("0" & ((NOT signed(operand_b)) + "0000000000000001"));
128             result <= temp(15 DOWNTO 0);
129
130         WHEN NOA => -- Returns NOT operand_a
131             temp   <= ("0" & (NOT (operand_a)));
132             result <= temp(15 DOWNTO 0);
133
134         WHEN NOB => -- Returns NOT operand_a
135             temp   <= ("0" & (NOT (operand_b)));
136             result <= temp(15 DOWNTO 0);
137
138         WHEN LSL => -- Logic Shift Left operand_a by operand_b number of bits. Fill with "0"
139             temp   <= std_logic_vector("0" & (shift_left(unsigned(operand_a), to_integer(unsigned(operand_b)))));
140             result <= temp(15 DOWNTO 0);
141
142         WHEN LSR => -- Logic Shift Right operand_a by operand_b number of bits. Fill with "0"
143             temp   <= std_logic_vector("0" & (shift_right(unsigned(operand_a), to_integer(unsigned(operand_b)))));
144             result <= temp(15 DOWNTO 0);
145
146         WHEN ASR => -- Arithmetic Shift right operand_a by operand_b number of bits. Fill with "sign-bit"
147             temp   <= std_logic_vector("0" & (shift_right(signed(operand_a), to_integer(unsigned(operand_b))))); --
                    Operand A is typecast as signed to enable sign extension
148             result <= temp(15 DOWNTO 0);
149
150         WHEN ICA => -- Increments operand_a
151             temp   <= std_logic_vector(unsigned("0" & operand_a) + 1);
152             result <= temp(15 DOWNTO 0);
153
154         WHEN ICB => -- Increments operand_b
155             temp   <= std_logic_vector(unsigned("0" & operand_b) + 1);
156             result <= temp(15 DOWNTO 0);
```

```
157
158             WHEN PAS => -- Lets operand_a pass through the ALU
159                 temp   <= ("0" & operand_a);
160                 result <= operand_a;
161
162             WHEN PBS => -- Lets operand_b pass through the ALU
163                 temp   <= ("0" & operand_b);
164                 result <= operand_b;
165
166             WHEN OTHERS =>
167
168         END CASE;
169
170         Parity := '0';
171         carry_flag    <= '0';
172         parity_flag   <= '0';
173         signed_flag   <= '0';
174         overflow_flag <= '0';
175         zero_flag     <= '0';
176
177         IF (to_integer(unsigned(operation)) = ADD) THEN
178             overflow_flag <= ((operand_a(15)) OR (temp(15))) AND ((NOT (operand_b(15))) OR(NOT (temp(15)))) AND ((NOT (
                operand_a(15))) OR ((operand_b(15)))); -- Overflow flag is calculated using logic determined by solving a
                Karnaugh map.
179
180         ELSIF (to_integer(unsigned(operation)) = SUB) THEN
181             overflow_flag <= ((operand_a(15)) OR (operand_b(15))) AND ((NOT (operand_b(15))) OR((temp(15)))) AND ((NOT (
                operand_a(15))) OR (NOT (temp(15))));
182         ELSIF (to_integer(unsigned(operation)) = ICA) THEN
183             overflow_flag <= ((operand_a(15)) OR (temp(15))) AND ((NOT ('0')) OR(NOT (temp(15)))) AND ((NOT (operand_a(15)
                )) OR (('0')));
184         ELSIF (to_integer(unsigned(operation)) = ICB) THEN
185             overflow_flag <= ('0' OR (temp(15))) AND ((NOT (operand_b(15))) OR(NOT (temp(15)))) AND ((NOT ('0')) OR ((
                operand_b(15))));
186         ELSIF (to_integer(unsigned(operation)) = IKA) THEN
187             IF (operand_a = x"8000") then                                                    -- For negation, overflow is
                set when the operand is set as the largest negative number.
188                 overflow_flag <= '1';
189             end if;
190         ELSIF (to_integer(unsigned(operation)) = IKB) THEN
191             IF (operand_b = x"8000") then
192                 overflow_flag <= '1';
193             end if;
```

```
194             ELSIF (to_integer(unsigned(operation)) = MUL) THEN
195                 IF (to_integer(unsigned(mult_temp(31 DOWNTO 16))) /= 0) THEN                    -- For multiplication,
                        overflow is set when the 16 MSB's are different from zero.
196                     overflow_flag <= '1';
197                 END IF;
198             END IF;
199
200 --          ------------ Kan disse m ske skrives sammen med dem ovenfor?? ----------------
201             IF (to_integer(unsigned(operation)) = MUL) THEN
202                 IF (to_integer(unsigned(mult_temp(31 DOWNTO 16))) /= 0) THEN
203                     carry_flag <= '1';
204                 END IF;
205             ELSIF (to_integer(unsigned(operation)) = SUB) THEN
206                 if (to_integer(unsigned(operand_b))>to_integer(unsigned(operand_a))) then
207                     carry_flag <= '1';
208                 end if;
209             ELSE
210                 carry_flag  <= temp(16);
211             END IF;
212 --          ------------------------------------------------------------------------------
213             signed_flag <= temp(15);
214
215             IF (temp(15 DOWNTO 0) = "0000000000000000") THEN
216                 zero_flag <= '1';
217             END IF;
218
219             FOR I IN 0 TO 15 LOOP
220                 Parity := Parity XOR temp(I);
221             END LOOP;
222             parity_flag <= Parity;
223
224         END IF;
225
226     END PROCESS;
227
228 END ARCHITECTURE Behavioral;
```

P.7 Control

```
1  #include "Control.hvhd"
2  #include "Config.hvhd"
3  ----------------------------------------------------------------------------------
4  --Module Name: Control Unit
5  --
6  --Description:
7  --Accepts a 6 bit opcode and returns control signales used to control the functionality of the CPU
8  --Control Signal definitions can be found in Control.hvhd
9  --The Control unit does not depend on a clock signal
10 --
11 --The Control lines have the following uses:
12 --
13 ----------------------------------------------------------------------------------
14
15 LIBRARY IEEE;
16 USE IEEE.STD_LOGIC_1164.ALL;
17 USE IEEE.NUMERIC_STD.ALL;
18 ENTITY Control IS
19     PORT (
20         opcode : IN std_logic_vector(31 DOWNTO 26); --6 bit opcode (e.g ADDI)
21         control_signals : OUT std_logic_vector(CONTROL_SIZE-1 DOWNTO 0) --Control Output
22     );
23 END ENTITY Control;
24
25 ARCHITECTURE Behavioral OF Control IS
26 BEGIN
27     --One massive mux/look up table. Is used since the input is mutually exclusive
28     WITH to_integer(unsigned(opcode)) SELECT control_signals <=
29
30     --Register-Register ALU Control
31     std_logic_vector(to_unsigned(ADD, 6)) & std_logic_vector(to_unsigned(NB, 3)) & FLGSE_D & FLGLO_D & FIRIN_D & FIRCO_D &
           FIRRE_D & MEMRD_D & MEMWR_D & REGWR_E & MEMRB_D & IMSEL_D & PUSHO_D & POP_D & RWSWI_D & MW2PC_D & HALTP_D WHEN ADDR,
32     std_logic_vector(to_unsigned(ADC, 6)) & std_logic_vector(to_unsigned(NB, 3)) & FLGSE_D & FLGLO_D & FIRIN_D & FIRCO_D &
           FIRRE_D & MEMRD_D & MEMWR_D & REGWR_E & MEMRB_D & IMSEL_D & PUSHO_D & POP_D & RWSWI_D & MW2PC_D & HALTP_D WHEN ADDCR,
33     std_logic_vector(to_unsigned(SUB, 6)) & std_logic_vector(to_unsigned(NB, 3)) & FLGSE_D & FLGLO_D & FIRIN_D & FIRCO_D &
           FIRRE_D & MEMRD_D & MEMWR_D & REGWR_E & MEMRB_D & IMSEL_D & PUSHO_D & POP_D & RWSWI_D & MW2PC_D & HALTP_D WHEN SUBR,
34     std_logic_vector(to_unsigned(NOA, 6)) & std_logic_vector(to_unsigned(NB, 3)) & FLGSE_D & FLGLO_D & FIRIN_D & FIRCO_D &
           FIRRE_D & MEMRD_D & MEMWR_D & REGWR_E & MEMRB_D & IMSEL_D & PUSHO_D & POP_D & RWSWI_D & MW2PC_D & HALTP_D WHEN NEGR,
35     std_logic_vector(to_unsigned(OGG, 6)) & std_logic_vector(to_unsigned(NB, 3)) & FLGSE_D & FLGLO_D & FIRIN_D & FIRCO_D &
           FIRRE_D & MEMRD_D & MEMWR_D & REGWR_E & MEMRB_D & IMSEL_D & PUSHO_D & POP_D & RWSWI_D & MW2PC_D & HALTP_D WHEN ANDR,
36     std_logic_vector(to_unsigned(ELL, 6)) & std_logic_vector(to_unsigned(NB, 3)) & FLGSE_D & FLGLO_D & FIRIN_D & FIRCO_D &
```

```
        FIRRE_D & MEMRD_D & MEMWR_D & REGWR_E & MEMRB_D & IMSEL_D & PUSHO_D & POP_D & RWSWI_D & MW2PC_D & HALTP_D WHEN ORR,
37      std_logic_vector(to_unsigned(XEL, 6)) & std_logic_vector(to_unsigned(NB, 3)) & FLGSE_D & FLGLO_D & FIRIN_D & FIRCO_D &
        FIRRE_D & MEMRD_D & MEMWR_D & REGWR_E & MEMRB_D & IMSEL_D & PUSHO_D & POP_D & RWSWI_D & MW2PC_D & HALTP_D WHEN XORR,
38      std_logic_vector(to_unsigned(MUL, 6)) & std_logic_vector(to_unsigned(NB, 3)) & FLGSE_D & FLGLO_D & FIRIN_D & FIRCO_D &
        FIRRE_D & MEMRD_D & MEMWR_D & REGWR_E & MEMRB_D & IMSEL_D & PUSHO_D & POP_D & RWSWI_D & MW2PC_D & HALTP_D WHEN MULTR,
39      std_logic_vector(to_unsigned(LSL, 6)) & std_logic_vector(to_unsigned(NB, 3)) & FLGSE_D & FLGLO_D & FIRIN_D & FIRCO_D &
        FIRRE_D & MEMRD_D & MEMWR_D & REGWR_E & MEMRB_D & IMSEL_D & PUSHO_D & POP_D & RWSWI_D & MW2PC_D & HALTP_D WHEN LSLR,
40      std_logic_vector(to_unsigned(LSR, 6)) & std_logic_vector(to_unsigned(NB, 3)) & FLGSE_D & FLGLO_D & FIRIN_D & FIRCO_D &
        FIRRE_D & MEMRD_D & MEMWR_D & REGWR_E & MEMRB_D & IMSEL_D & PUSHO_D & POP_D & RWSWI_D & MW2PC_D & HALTP_D WHEN LSRR,
41      std_logic_vector(to_unsigned(ASR, 6)) & std_logic_vector(to_unsigned(NB, 3)) & FLGSE_D & FLGLO_D & FIRIN_D & FIRCO_D &
        FIRRE_D & MEMRD_D & MEMWR_D & REGWR_E & MEMRB_D & IMSEL_D & PUSHO_D & POP_D & RWSWI_D & MW2PC_D & HALTP_D WHEN RASR,
42
43      --Register-Immediate ALU Control
44      std_logic_vector(to_unsigned(ADD, 6)) & std_logic_vector(to_unsigned(NB, 3)) & FLGSE_D & FLGLO_D & FIRIN_D & FIRCO_D &
        FIRRE_D & MEMRD_D & MEMWR_D & REGWR_E & MEMRB_D & IMSEL_E & PUSHO_D & POP_D & RWSWI_D & MW2PC_D & HALTP_D WHEN ADDI,
45      std_logic_vector(to_unsigned(ADC, 6)) & std_logic_vector(to_unsigned(NB, 3)) & FLGSE_D & FLGLO_D & FIRIN_D & FIRCO_D &
        FIRRE_D & MEMRD_D & MEMWR_D & REGWR_E & MEMRB_D & IMSEL_E & PUSHO_D & POP_D & RWSWI_D & MW2PC_D & HALTP_D WHEN ADDCI,
46      std_logic_vector(to_unsigned(SUB, 6)) & std_logic_vector(to_unsigned(NB, 3)) & FLGSE_D & FLGLO_D & FIRIN_D & FIRCO_D &
        FIRRE_D & MEMRD_D & MEMWR_D & REGWR_E & MEMRB_D & IMSEL_E & PUSHO_D & POP_D & RWSWI_D & MW2PC_D & HALTP_D WHEN SUBI,
47      std_logic_vector(to_unsigned(NOA, 6)) & std_logic_vector(to_unsigned(NB, 3)) & FLGSE_D & FLGLO_D & FIRIN_D & FIRCO_D &
        FIRRE_D & MEMRD_D & MEMWR_D & REGWR_E & MEMRB_D & IMSEL_E & PUSHO_D & POP_D & RWSWI_D & MW2PC_D & HALTP_D WHEN NEGI,
48      std_logic_vector(to_unsigned(OGG, 6)) & std_logic_vector(to_unsigned(NB, 3)) & FLGSE_D & FLGLO_D & FIRIN_D & FIRCO_D &
        FIRRE_D & MEMRD_D & MEMWR_D & REGWR_E & MEMRB_D & IMSEL_E & PUSHO_D & POP_D & RWSWI_D & MW2PC_D & HALTP_D WHEN ANDI,
49      std_logic_vector(to_unsigned(ELL, 6)) & std_logic_vector(to_unsigned(NB, 3)) & FLGSE_D & FLGLO_D & FIRIN_D & FIRCO_D &
        FIRRE_D & MEMRD_D & MEMWR_D & REGWR_E & MEMRB_D & IMSEL_E & PUSHO_D & POP_D & RWSWI_D & MW2PC_D & HALTP_D WHEN ORI,
50      std_logic_vector(to_unsigned(XEL, 6)) & std_logic_vector(to_unsigned(NB, 3)) & FLGSE_D & FLGLO_D & FIRIN_D & FIRCO_D &
        FIRRE_D & MEMRD_D & MEMWR_D & REGWR_E & MEMRB_D & IMSEL_E & PUSHO_D & POP_D & RWSWI_D & MW2PC_D & HALTP_D WHEN XORI,
51      std_logic_vector(to_unsigned(MUL, 6)) & std_logic_vector(to_unsigned(NB, 3)) & FLGSE_D & FLGLO_D & FIRIN_D & FIRCO_D &
        FIRRE_D & MEMRD_D & MEMWR_D & REGWR_E & MEMRB_D & IMSEL_E & PUSHO_D & POP_D & RWSWI_D & MW2PC_D & HALTP_D WHEN MULTI,
52      std_logic_vector(to_unsigned(LSL, 6)) & std_logic_vector(to_unsigned(NB, 3)) & FLGSE_D & FLGLO_D & FIRIN_D & FIRCO_D &
        FIRRE_D & MEMRD_D & MEMWR_D & REGWR_E & MEMRB_D & IMSEL_E & PUSHO_D & POP_D & RWSWI_D & MW2PC_D & HALTP_D WHEN LSLI,
53      std_logic_vector(to_unsigned(LSR, 6)) & std_logic_vector(to_unsigned(NB, 3)) & FLGSE_D & FLGLO_D & FIRIN_D & FIRCO_D &
        FIRRE_D & MEMRD_D & MEMWR_D & REGWR_E & MEMRB_D & IMSEL_E & PUSHO_D & POP_D & RWSWI_D & MW2PC_D & HALTP_D WHEN LSRI,
54      std_logic_vector(to_unsigned(ASR, 6)) & std_logic_vector(to_unsigned(NB, 3)) & FLGSE_D & FLGLO_D & FIRIN_D & FIRCO_D &
        FIRRE_D & MEMRD_D & MEMWR_D & REGWR_E & MEMRB_D & IMSEL_E & PUSHO_D & POP_D & RWSWI_D & MW2PC_D & HALTP_D WHEN RASI,
55
56      --Nop
57      std_logic_vector(to_unsigned(NAA, 6)) & std_logic_vector(to_unsigned(NB, 3)) & FLGSE_D & FLGLO_D & FIRIN_D & FIRCO_D &
        FIRRE_D & MEMRD_D & MEMWR_D & REGWR_D & MEMRB_D & IMSEL_D & PUSHO_D & POP_D & RWSWI_D & MW2PC_D & HALTP_D WHEN NOP,
58
59
60      --Register Compare and Move
61      std_logic_vector(to_unsigned(SUB, 6)) & std_logic_vector(to_unsigned(NB, 3)) & FLGSE_D & FLGLO_D & FIRIN_D & FIRCO_D &
```

```
        FIRRE_D & MEMRD_D & MEMWR_D & REGWR_D & MEMRB_D & IMSEL_D & PUSHO_D & POP_D & RWSWI_D & MW2PC_D & HALTP_D WHEN CMP,
62      std_logic_vector(to_unsigned(PAS, 6)) & std_logic_vector(to_unsigned(NB, 3)) & FLGSE_D & FLGLO_D & FIRIN_D & FIRCO_D &
        FIRRE_D & MEMRD_D & MEMWR_D & REGWR_E & MEMRB_D & IMSEL_D & PUSHO_D & POP_D & RWSWI_D & MW2PC_D & HALTP_D WHEN MOV,
63
64      --Immediate Compare and Move
65      std_logic_vector(to_unsigned(SUB, 6)) & std_logic_vector(to_unsigned(NB, 3)) & FLGSE_D & FLGLO_D & FIRIN_D & FIRCO_D &
        FIRRE_D & MEMRD_D & MEMWR_D & REGWR_D & MEMRB_D & IMSEL_E & PUSHO_D & POP_D & RWSWI_D & MW2PC_D & HALTP_D WHEN CMPI,
66      std_logic_vector(to_unsigned(PBS, 6)) & std_logic_vector(to_unsigned(NB, 3)) & FLGSE_D & FLGLO_D & FIRIN_D & FIRCO_D &
        FIRRE_D & MEMRD_D & MEMWR_D & REGWR_E & MEMRB_D & IMSEL_E & PUSHO_D & POP_D & RWSWI_D & MW2PC_D & HALTP_D WHEN MOVI,
67
68      --Memory Control
69      std_logic_vector(to_unsigned(ADD, 6)) & std_logic_vector(to_unsigned(NB, 3)) & FLGSE_D & FLGLO_D & FIRIN_D & FIRCO_D &
        FIRRE_D & MEMRD_E & MEMWR_D & REGWR_E & MEMRB_E & IMSEL_E & PUSHO_D & POP_D & RWSWI_D & MW2PC_E & HALTP_D WHEN LOAD,
70      std_logic_vector(to_unsigned(ADD, 6)) & std_logic_vector(to_unsigned(NB, 3)) & FLGSE_D & FLGLO_D & FIRIN_D & FIRCO_D &
        FIRRE_D & MEMRD_D & MEMWR_E & REGWR_D & MEMRB_D & IMSEL_E & PUSHO_D & POP_D & RWSWI_E & MW2PC_D & HALTP_D WHEN STORE,
71
72      --Stack Control
73      std_logic_vector(to_unsigned(ICA, 6)) & std_logic_vector(to_unsigned(NB, 3)) & FLGSE_D & FLGLO_D & FIRIN_D & FIRCO_D &
        FIRRE_D & MEMRD_D & MEMWR_E & REGWR_E & MEMRB_D & IMSEL_D & PUSHO_E & POP_D & RWSWI_D & MW2PC_D & HALTP_D WHEN PUSH,
74      std_logic_vector(to_unsigned(NOP, 6)) & std_logic_vector(to_unsigned(NB, 3)) & FLGSE_D & FLGLO_D & FIRIN_D & FIRCO_D &
        FIRRE_D & MEMRD_E & MEMWR_D & REGWR_E & MEMRB_E & IMSEL_D & PUSHO_D & POP_E & RWSWI_D & MW2PC_E & HALTP_D WHEN POP,
75      --Branch Control
76      std_logic_vector(to_unsigned(PBS, 6)) & std_logic_vector(to_unsigned(BR, 3)) & FLGSE_D & FLGLO_D & FIRIN_D & FIRCO_D &
        FIRRE_D & MEMRD_D & MEMWR_D & REGWR_D & MEMRB_D & IMSEL_E & PUSHO_D & POP_D & RWSWI_D & MW2PC_D & HALTP_D WHEN JMP,
77      std_logic_vector(to_unsigned(PBS, 6)) & std_logic_vector(to_unsigned(EQ, 3)) & FLGSE_D & FLGLO_D & FIRIN_D & FIRCO_D &
        FIRRE_D & MEMRD_D & MEMWR_D & REGWR_D & MEMRB_D & IMSEL_E & PUSHO_D & POP_D & RWSWI_D & MW2PC_D & HALTP_D WHEN JMPEQ,
78      std_logic_vector(to_unsigned(PBS, 6)) & std_logic_vector(to_unsigned(LE, 3)) & FLGSE_D & FLGLO_D & FIRIN_D & FIRCO_D &
        FIRRE_D & MEMRD_D & MEMWR_D & REGWR_D & MEMRB_D & IMSEL_E & PUSHO_D & POP_D & RWSWI_D & MW2PC_D & HALTP_D WHEN JMPLE,
79      std_logic_vector(to_unsigned(PBS, 6)) & std_logic_vector(to_unsigned(NQ, 3)) & FLGSE_D & FLGLO_D & FIRIN_D & FIRCO_D &
        FIRRE_D & MEMRD_D & MEMWR_D & REGWR_D & MEMRB_D & IMSEL_E & PUSHO_D & POP_D & RWSWI_D & MW2PC_D & HALTP_D WHEN JMPNQ,
80      std_logic_vector(to_unsigned(PBS, 6)) & std_logic_vector(to_unsigned(PA, 3)) & FLGSE_D & FLGLO_D & FIRIN_D & FIRCO_D &
        FIRRE_D & MEMRD_D & MEMWR_D & REGWR_D & MEMRB_D & IMSEL_E & PUSHO_D & POP_D & RWSWI_D & MW2PC_D & HALTP_D WHEN JMPPA,
81
82      --Branch Control (Jump to registers)
83      std_logic_vector(to_unsigned(PAS, 6)) & std_logic_vector(to_unsigned(BR, 3)) & FLGSE_D & FLGLO_D & FIRIN_D & FIRCO_D &
        FIRRE_D & MEMRD_D & MEMWR_D & REGWR_D & MEMRB_D & IMSEL_D & PUSHO_D & POP_D & RWSWI_D & MW2PC_D & HALTP_D WHEN JMPR,
84      std_logic_vector(to_unsigned(PAS, 6)) & std_logic_vector(to_unsigned(EQ, 3)) & FLGSE_D & FLGLO_D & FIRIN_D & FIRCO_D &
        FIRRE_D & MEMRD_D & MEMWR_D & REGWR_D & MEMRB_D & IMSEL_D & PUSHO_D & POP_D & RWSWI_D & MW2PC_D & HALTP_D WHEN JMPEQR,
85      std_logic_vector(to_unsigned(PAS, 6)) & std_logic_vector(to_unsigned(LE, 3)) & FLGSE_D & FLGLO_D & FIRIN_D & FIRCO_D &
        FIRRE_D & MEMRD_D & MEMWR_D & REGWR_D & MEMRB_D & IMSEL_D & PUSHO_D & POP_D & RWSWI_D & MW2PC_D & HALTP_D WHEN JMPLER,
86      std_logic_vector(to_unsigned(PAS, 6)) & std_logic_vector(to_unsigned(NQ, 3)) & FLGSE_D & FLGLO_D & FIRIN_D & FIRCO_D &
        FIRRE_D & MEMRD_D & MEMWR_D & REGWR_D & MEMRB_D & IMSEL_D & PUSHO_D & POP_D & RWSWI_D & MW2PC_D & HALTP_D WHEN JMPNQR,
87      std_logic_vector(to_unsigned(PAS, 6)) & std_logic_vector(to_unsigned(PA, 3)) & FLGSE_D & FLGLO_D & FIRIN_D & FIRCO_D &
```

```
        FIRRE_D & MEMRD_D & MEMWR_D & REGWR_D & MEMRB_D & IMSEL_D & PUSHO_D & POP_D & RWSWI_D & MW2PC_D & HALTP_D WHEN JMPPAR,
88
89      --Stop Process
90      std_logic_vector(to_unsigned(NAA, 6)) & std_logic_vector(to_unsigned(NB, 3)) & FLGSE_D & FLGLO_D & FIRIN_D & FIRCO_D &
            FIRRE_D & MEMRD_D & MEMWR_D & REGWR_D & MEMRB_D & IMSEL_D & PUSHO_D & POP_D & RWSWI_D & MW2PC_D & HALTP_E WHEN HALT,
91
92      --Filter instructions
93      std_logic_vector(to_unsigned(PAS, 6)) & std_logic_vector(to_unsigned(NB, 3)) & FLGSE_D & FLGLO_D & FIRIN_D & FIRCO_E &
            FIRRE_D & MEMRD_D & MEMWR_D & REGWR_D & MEMRB_D & IMSEL_D & PUSHO_D & POP_D & RWSWI_D & MW2PC_D & HALTP_D WHEN FIRCOR,
94      std_logic_vector(to_unsigned(PBS, 6)) & std_logic_vector(to_unsigned(NB, 3)) & FLGSE_D & FLGLO_D & FIRIN_D & FIRCO_E &
            FIRRE_D & MEMRD_D & MEMWR_D & REGWR_D & MEMRB_D & IMSEL_E & PUSHO_D & POP_D & RWSWI_D & MW2PC_D & HALTP_D WHEN FIRCOI,
95      std_logic_vector(to_unsigned(NAA, 6)) & std_logic_vector(to_unsigned(NB, 3)) & FLGSE_D & FLGLO_D & FIRIN_D & FIRCO_D &
            FIRRE_E & MEMRD_D & MEMWR_D & REGWR_D & MEMRB_D & IMSEL_D & PUSHO_D & POP_D & RWSWI_D & MW2PC_D & HALTP_D WHEN FIRRESET
            ,
96      std_logic_vector(to_unsigned(PAS, 6)) & std_logic_vector(to_unsigned(NB, 3)) & FLGSE_D & FLGLO_D & FIRIN_E & FIRCO_D &
            FIRRE_D & MEMRD_D & MEMWR_D & REGWR_E & MEMRB_D & IMSEL_D & PUSHO_D & POP_D & RWSWI_D & MW2PC_D & HALTP_D WHEN FIRSAR,
97      std_logic_vector(to_unsigned(PBS, 6)) & std_logic_vector(to_unsigned(NB, 3)) & FLGSE_D & FLGLO_D & FIRIN_E & FIRCO_D &
            FIRRE_D & MEMRD_D & MEMWR_D & REGWR_E & MEMRB_D & IMSEL_E & PUSHO_D & POP_D & RWSWI_D & MW2PC_D & HALTP_D WHEN FIRSAI,
98
99      --Flag Operations
100     std_logic_vector(to_unsigned(NAA, 6)) & std_logic_vector(to_unsigned(NB, 3)) & FLGSE_D & FLGLO_E & FIRIN_D & FIRCO_D &
            FIRRE_D & MEMRD_D & MEMWR_D & REGWR_E & MEMRB_D & IMSEL_D & PUSHO_D & POP_D & RWSWI_D & MW2PC_D & HALTP_D WHEN GETFLAG,
101     std_logic_vector(to_unsigned(PBS, 6)) & std_logic_vector(to_unsigned(NB, 3)) & FLGSE_E & FLGLO_D & FIRIN_D & FIRCO_D &
            FIRRE_D & MEMRD_D & MEMWR_D & REGWR_D & MEMRB_D & IMSEL_E & PUSHO_D & POP_D & RWSWI_D & MW2PC_D & HALTP_D WHEN SETFLAG,
102     std_logic_vector(to_unsigned(PAS, 6)) & std_logic_vector(to_unsigned(NB, 3)) & FLGSE_E & FLGLO_D & FIRIN_D & FIRCO_D &
            FIRRE_D & MEMRD_D & MEMWR_D & REGWR_D & MEMRB_D & IMSEL_D & PUSHO_D & POP_D & RWSWI_D & MW2PC_D & HALTP_D WHEN SETFLAGR
            ,
103     (OTHERS => '0') WHEN OTHERS;
104 END ARCHITECTURE Behavioral;
```

P.8 Filter

```
#include "FIR_Config.hvhd"
#define MULTEXTEND product_array(I)(MULTIPLIER_WIDTH - 1)


LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.std_logic_signed.ALL;
USE IEEE.NUMERIC_STD.ALL;
USE ieee.numeric_std.ALL;
-- ----------------------------------------------------------
ENTITY Filter IS ------> Interface
    PORT
    (
        clk                     : IN STD_LOGIC := '0'; -- System clock
        reset                   : IN STD_LOGIC := '0';  -- Asynchron reset
        load_system_input       : IN STD_LOGIC := '0'; -- Load/run switch
        system_input            : IN STD_LOGIC_VECTOR(INOUT_BIT_WIDTH - 1 DOWNTO 0) := (OTHERS => '0'); -- system_input
        coefficient_in          : IN STD_LOGIC_VECTOR(COEFFICIENT_WIDTH - 1 DOWNTO 0) := (OTHERS => '0'); -- Coefficient data
            input
        system_output           : OUT STD_LOGIC_VECTOR(INOUT_BIT_WIDTH - 1 DOWNTO 0) := (OTHERS => '0');  -- System output
        write_enable            : IN STD_LOGIC := '0'
    );

END Filter;
-- ----------------------------------------------------------
ARCHITECTURE Behavioural OF Filter IS

    TYPE coefficient_array_type     IS ARRAY (0 TO TAPS - 1) OF STD_LOGIC_VECTOR(COEFFICIENT_WIDTH - 1 DOWNTO 0);
    TYPE product_array_type         IS ARRAY (0 TO TAPS - 1) OF STD_LOGIC_VECTOR(MULTIPLIER_WIDTH - 1 DOWNTO 0);
    TYPE adder_array_type           IS ARRAY (0 TO TAPS - 1) OF STD_LOGIC_VECTOR(ADDER_WIDTH - 1 DOWNTO 0);

    SIGNAL coefficient_array        : coefficient_array_type := (others => (others => '0'));    -- Coefficient array
    SIGNAL product_array            : product_array_type := (others => (others => '0'));        -- Product array
    SIGNAL adder_array              : adder_array_type := (others => (others => '0'));      -- Adder array

    SIGNAL input_data_temp          : STD_LOGIC_VECTOR(INOUT_BIT_WIDTH - 1 DOWNTO 0) := (OTHERS => '0');
    SIGNAL full_output              : STD_LOGIC_VECTOR(ADDER_WIDTH - 1 DOWNTO 0) := (OTHERS => '0');
    SIGNAL truncated_output         : STD_LOGIC_VECTOR(ADDER_WIDTH - 1 DOWNTO 0) := (OTHERS => '0');

BEGIN
    Load : PROCESS (clk, reset, coefficient_in, coefficient_array, system_input)
```

```
41  BEGIN
42                                          ------> Load data or coefficients
43      IF reset = '1' THEN             -- clear data and coefficients reg.
44          IF(write_enable = '1') THEN
45              input_data_temp <= (others=>'0');
46              FOR K IN 0 TO TAPS - 1 LOOP
47                  coefficient_array(K) <= (others=>'0');
48              END LOOP;
49          END IF;
50      ELSIF falling_edge(clk) THEN
51          IF(write_enable = '1') THEN
52              IF load_system_input = '0' THEN
53                  coefficient_array(TAPS - 1) <= coefficient_in; -- Store coefficient in register
54                  FOR I IN TAPS - 2 DOWNTO 0 LOOP -- Coefficients shift one
55                      coefficient_array(I) <= coefficient_array(I + 1);
56                  END LOOP;
57              ELSE
58                  input_data_temp <= system_input; -- Get one data sample at adder_array time
59              END IF;
60          END IF;
61      END IF;
62  END PROCESS Load;
63
64  SOP : PROCESS (clk, reset, adder_array, product_array)-- Compute sum-of-products
65  BEGIN
66      IF reset = '1' THEN -- clear tap registers
67          IF(write_enable = '1') THEN
68              FOR K IN 0 TO TAPS - 1 LOOP
69                  adder_array(K) <= (OTHERS => '0');
70              END LOOP;
71          END IF;
72      ELSIF falling_edge(clk) THEN
73          IF(write_enable = '1') THEN
74              FOR I IN 0 TO TAPS - 2 LOOP -- Compute the transposed
75                  adder_array(I) <= std_logic_vector(resize(signed(product_array(I)),adder_array(0)'length)) + adder_array(I
                        + 1); -- filter adds
76              END LOOP;
77              adder_array(TAPS - 1) <= std_logic_vector(resize(signed(product_array(TAPS-1)),adder_array(0)'length)); --
                    First TAP has only adder_array register
78          END IF;
79      END IF;
80      full_output <= adder_array(0);
81  END PROCESS SOP;
```

```
82
83      -- Instantiate TAPS multipliers
84      MulGen : FOR I IN 0 TO TAPS - 1 GENERATE
85          product_array(i) <= coefficient_array(i) * input_data_temp;
86      END GENERATE;
87
88      truncated_output <= std_logic_vector(shift_right(unsigned(full_output), 15));
89      system_output <= truncated_output(15 downto 0);
90  END Behavioural;
```

P.9 RegisterFile

```
#include "Config.hvhd"
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.numeric_std.ALL;

ENTITY RegistryInternal IS
    PORT
    (
        --Index of the two registers to be read
        read_register_a_index     : IN std_logic_vector(4 DOWNTO 0);
        read_register_b_index     : IN std_logic_vector(4 DOWNTO 0);

        write_register_index      : IN std_logic_vector(4 DOWNTO 0); --Index of register to write data to

        register_file_data_in     : IN std_logic_vector(WORD_SIZE-1 DOWNTO 0); --data to be written to a register

        --Data outputs
        register_file_data_out_a : OUT std_logic_vector(WORD_SIZE-1 DOWNTO 0);
        register_file_data_out_b : OUT std_logic_vector(WORD_SIZE-1 DOWNTO 0);

        --Input data from the PC/SP register
        pc_value_input           : IN std_logic_vector(WORD_SIZE-1 DOWNTO 0);
        sp_value_input           : IN std_logic_vector(WORD_SIZE-1 DOWNTO 0);

        write_enable             : IN std_logic; --Write enable signal

        clk                      : IN std_logic  --System clock
    );

END RegistryInternal;

ARCHITECTURE Behavioral OF RegistryInternal IS

    TYPE register_type IS ARRAY (29 DOWNTO 0) OF std_logic_vector(WORD_SIZE-1 DOWNTO 0);
    SIGNAL reg : register_type := (OTHERS => x"0000");
BEGIN

    ------------------------------------------
    -- RegIndex:
    -- Indexes, reads, and outputs the correct register
    -- data.
```

```
42      -- Note that the Program counter and Stack Pointer Register
43      -- are handled differently that the other 30 general purpose registers
44      -- since the physical register is not located inside the register file
45      ------------------------------------------
46      RegIndex : PROCESS (read_register_a_index, read_register_b_index, write_register_index, write_enable, pc_value_input, reg)
            IS
47
48      BEGIN
49          IF to_integer(unsigned(read_register_a_index)) = 31 THEN --pc to outOne
50              register_file_data_out_a <= pc_value_input;
51          ELSIF to_integer(unsigned(read_register_a_index)) = 30 THEN --sp to outOne
52              register_file_data_out_a <= sp_value_input;
53          ELSE
54              register_file_data_out_a <= reg(to_integer(unsigned(read_register_a_index)));
55          END IF;
56
57          IF to_integer(unsigned(read_register_b_index)) = 31 THEN --pc to outTwo
58              register_file_data_out_b <= pc_value_input;
59          ELSIF to_integer(unsigned(read_register_b_index)) = 30 THEN --sp to outTwo
60              register_file_data_out_b <= sp_value_input;
61          ELSE
62              register_file_data_out_b <= reg(to_integer(unsigned(read_register_b_index)));
63          END IF;
64      END PROCESS;
65
66      ------------------------------------------
67      -- WriteReg:
68      -- Writes data to the 30 General purpose registers
69      -- SP and PC writes are handled externally
70      ------------------------------------------
71      WriteReg : PROCESS (clk) IS
72      BEGIN
73          IF (rising_edge(clk)) THEN
74              IF (write_enable = '1') THEN
75                  IF (to_integer(unsigned(write_register_index)) <= 29) THEN
76                      reg(to_integer(unsigned(write_register_index))) <= register_file_data_in;
77                  END IF;
78              END IF;
79          END IF;
80      END PROCESS;
81
82  END Behavioral;
```

P.10 Stack

```
1  #include "Config.hvhd"
2
3  ----------------------------------------------------------------------------------
4  --Module Name: Stack Controller
5  --
6  --
7  --Description:
8  --Controls the Stack Pointer register's output, as well as a
9  --automatically incrementing/decrementing the stack pointer
10 --when executing push/pop instructions.
11 --
12 --The Stack Controller depends on the rising edge of the system clock
13 ----------------------------------------------------------------------------------
14
15 LIBRARY IEEE;
16 USE IEEE.STD_LOGIC_1164.ALL;
17 USE IEEE.NUMERIC_STD.ALL;
18
19 ENTITY Stack IS
20     PORT
21     (
22         address_out : OUT std_logic_vector(WORD_SIZE-1 DOWNTO 0); --Stack Pointer output
23         address_in  : IN std_logic_vector(WORD_SIZE-1 DOWNTO 0); --New value of stack pointer (only applicable when "
               write_back" is set high)
24         write_back  : IN std_logic := '0'; --Control port, set high when performing a manual stack point overwrite
25         pop         : IN std_logic := '0'; --Control port, set high when a "pop" instruction occours (should be connected to
               the "pop" control port)
26         push        : IN std_logic := '0'; --Control port, set high when a "push" instruction occours (should be connected to
               the "push" control port)
27         clk         : IN std_logic --CPU clock signal (should be connected to the System Clock)
28     );
29 END Stack;
30 ARCHITECTURE Behavioral OF Stack IS
31     SIGNAL sp : std_logic_vector(WORD_SIZE-1 DOWNTO 0) := (OTHERS => '0');
32
33 BEGIN
34
35     ----------------------------------------------
36     -- ChangeSP:
37     -- Changes the value of the stack pointer on rising clk
38     --
```

```
39      -- Decrements sp if "pop" is executing
40      -- Increments sp if "push" is executing
41      -- Sets sp to an arbitrary value if "address_in" is high
42      ------------------------------------------
43      ChangeSP : PROCESS (clk) --Process changes the value of the stack pointer
44      BEGIN
45          IF (rising_edge(clk)) THEN
46              IF (pop = '1') THEN
47                  sp <= std_logic_vector(unsigned(sp) - 1); --Decrement when pop
48              ELSIF (push = '1') THEN
49                  sp <= std_logic_vector(unsigned(sp) + 1); --Increment when push
50              ELSIF (write_back = '1') THEN
51                  sp <= address_in; --Manual Overwrite
52              END IF;
53          END IF;
54
55      END PROCESS;
56      --If a "push" instruction is being executed, the stack controller outputs the "stack pointer + 1"
57      --to ensure that the pushed value is stored at the new location at the top of the stack.
58      --The sp itself is changed on the next rising edge (see the ChangeSP : Process)
59
60      --If a pop instruction is being executed, the stack controller outputs the current value of the stack pointer
61      --Since this corresponds to the value of the current top. The sp is changed on the next rising edge (see the ChangeSP :
            Process)
62      --To correspond to the new address of the top of the stack.
63      WITH push SELECT address_out <=
64          std_logic_vector(unsigned(sp) + 1) WHEN '1',
65          sp WHEN OTHERS;
66  END Behavioral;
```

P.11 PMC

```
1  #include "Config.hvhd"
2  LIBRARY IEEE;
3  USE IEEE.STD_LOGIC_1164.ALL;
4  USE IEEE.NUMERIC_STD.ALL;
5
6  ENTITY MemoryController IS
7
8      PORT
9      (
10         write_enable               : IN STD_LOGIC;
11         read_enable                : IN STD_LOGIC;
12         address                    : IN STD_LOGIC_vector (WORD_SIZE-1 DOWNTO 0);
13         data_in                    : IN STD_LOGIC_vector (WORD_SIZE-1 DOWNTO 0);
14         data_out                   : BUFFER STD_LOGIC_vector (WORD_SIZE-1 DOWNTO 0) := (OTHERS => '0');
15         clk                        : IN STD_LOGIC;
16         btn                        : IN std_LOGIC_vector(2 DOWNTO 0);
17         seven_seg_control_signals  : OUT std_LOGIC_vector(31 DOWNTO 0);
18
19         --Interrupts
20         interrupt_address          : OUT std_logic_vector(PC_SIZE - 1 DOWNTO 0);
21         interrupt_cpu              : OUT std_logic;
22         interrupt_enable           : IN std_logic := '0';
23         interrupt_nest_enable      : OUT std_logic;
24
25         --I2S
26         i2s_bit_clk                : IN std_logic  := '0';
27         i2s_bit_clk_2              : IN std_logic  := '0';
28         i2s_word_select            : IN std_logic  := '0';
29         i2s_data_in                : IN std_logic  := '0';
30         i2s_bit_clk_out            : OUT std_logic := '0';
31         i2s_word_select_out        : OUT std_logic := '0';
32         i2s_data_out               : OUT std_logic := '0'
33     );
34 END MemoryController;
35
36 ARCHITECTURE Behavioral OF MemoryController IS
37     SIGNAL seven_seg_data, seven_seg_configuration, btn_data, interrupt_data                    : std_LOGIC_vector(
           WORD_SIZE-1 DOWNTO 0) := (OTHERS => '0');
38     SIGNAL dram_address                                                                         : std_logic_vector(
           WORD_SIZE-1 DOWNTO 0);
39     SIGNAL interrupt_controller_address_index                                                   : std_logic_vector(2
```

```
        DOWNTO 0);
    SIGNAL interrupt_btn_reset_signal, interrupt_i2s_reset_signal, interrupt_i2s_out_reset_signal       : STD_LOGIC;
    SIGNAL dram_data_out                                                                               : std_logic_vector(
        WORD_SIZE-1 DOWNTO 0) := (OTHERS => '0');
    SIGNAL btn_interrupt, i2s_interrupt, i2s_out_interrupt                                             : std_logic
                                := '0';
    SIGNAL write_enable_dram                                                                           : std_logic
                                := '0';
    SIGNAL read_enable_dram                                                                            : std_logic
                                := '0';
    SIGNAL i2s_mono_in_data_out                                                                        : std_logic_vector(
        WORD_SIZE-1 DOWNTO 0) := (OTHERS => '0');
    SIGNAL i2s_mono_out_data_in                                                                        : std_logic_vector(
        WORD_SIZE-1 DOWNTO 0) := (OTHERS => '0');
BEGIN

    SevenSegmentDisplayDriver : ENTITY work.ssgddriver
        PORT MAP
        (
            input_data                => seven_seg_data,
            clr                       => seven_seg_configuration(5),
            bcd_enable                => seven_seg_configuration(4),
            dot_control               => seven_seg_configuration(3 DOWNTO 0),
            seven_seg_control_signals => seven_seg_control_signals
        );

    ButtonDriver : ENTITY work.btndriver
        PORT
        MAP(
        debounced_btn_out => btn_data(2 DOWNTO 0),
        clk               => clk,
        clr               => '0',
        btn               => btn,
        interrupt_on      => btn_interrupt,
        interrupt_reset   => interrupt_btn_reset_signal
        );

    MemoryDriver : ENTITY work.Memory(falling)
        PORT
        MAP(
        data_in      => data_in,
        data_out     => dram_data_out,
        clk          => clk,
```

```
76          write_enable => write_enable_dram,
77          read_enable  => read_enable_dram,
78          address      => dram_address
79          );
80
81      I2SMonoIn : ENTITY work.I2SMonoIn(Behavioral)
82          PORT
83          MAP(
84          bit_clock        => i2s_bit_clk,
85          word_select      => i2s_word_select,
86          data_in          => i2s_data_in,
87          data_out         => i2s_mono_in_data_out,
88          interrupt        => i2s_interrupt,
89          interrupt_reset => interrupt_i2s_reset_signal
90          );
91
92      I2SMonoOut : ENTITY work.I2SMonoOut(Behavioral)
93          PORT
94          MAP(
95          clk              => i2s_bit_clk_2,
96          data_in          => i2s_mono_out_data_in,
97          bit_clock_out      => i2s_bit_clk_out,
98          word_select_out => i2s_word_select_out,
99          data_out         => i2s_data_out,
100         interrupt_out    => i2s_out_interrupt,
101         interrupt_out_reset => interrupt_i2s_out_reset_signal
102         );
103     InterruptDriver : ENTITY work.Interrupt(Behavioral)
104         PORT
105         MAP(
106         interrupt_btn              => btn_interrupt,
107         interrupt_btn_reset        => interrupt_btn_reset_signal,
108         interrupt_i2s              => i2s_interrupt,
109         interrupt_i2s_reset        => interrupt_i2s_reset_signal,
110         interrupt_i2s_out          => i2s_out_interrupt,
111         interrupt_i2s_out_reset    => interrupt_i2s_out_reset_signal,
112         write_enable               => write_enable,
113         clk                        => clk,
114         internal_register_address => interrupt_controller_address_index,
115         data_in                    => interrupt_data,
116         interrupt_address          => interrupt_address,
117         interrupt_cpu              => interrupt_cpu,
118         interrupt_enable           => interrupt_enable,
```

```
119         interrupt_nest_enable     => interrupt_nest_enable
120         );
121
122     ------------------------------------------
123     -- MainReadWrite:
124     -- Process takes care of reading from all peripherals
125     -- aswell as writing to all peripherals, except DRAM
126     --
127     -- While much of this process's syntax/logic may seem strange,
128     -- illogical and obscure, it was required for quartus to be able to
129     -- synthesize the design   \_(   )_/
130     ------------------------------------------
131     MainReadWrite : PROCESS (clk, btn_data, dram_data_out)
132     BEGIN
133         IF (to_integer(unsigned(address)) = 65000 AND write_enable = '1') THEN -- sevensegdriver data
134             IF (falling_edge(clk)) THEN
135                 seven_seg_data <= data_in;
136             END IF;
137
138         ELSIF (to_integer(unsigned(address)) = 65001 AND write_enable = '1') THEN -- Sevensegdriver control
139             IF (falling_edge(clk)) THEN
140                 seven_seg_configuration <= data_in;
141             END IF;
142
143         ELSIF (to_integer(unsigned(address)) = 65010 AND write_enable = '1') THEN
144             IF (falling_edge(clk)) THEN
145                 i2s_mono_out_data_in <= data_in;
146             END IF;
147
148         ELSIF (65100 <= to_integer(unsigned(address)) AND write_enable = '1') THEN
149             IF (falling_edge(clk)) THEN
150                 interrupt_controller_address_index <= std_logic_vector(to_unsigned(to_integer(unsigned(address) - 65100),
                        interrupt_controller_address_index'length));
151                 interrupt_data                     <= data_in;
152             END IF;
153
154         --The nested if statements are required for quartus allow synthesis.
155         ELSIF (to_integer(unsigned(address)) >= 65000 AND read_enable = '1') THEN -- ButtonDriver Data
156             IF (falling_edge(clk)) THEN
157                 IF (to_integer(unsigned(address)) = 65002) THEN
158                     data_out <= btn_data;
159                 ELSIF (to_integer(unsigned(address)) = 65011) THEN
160                     data_out <= i2s_mono_in_data_out;
```

```
161                 END IF;
162             END IF;
163
164         ELSIF (read_enable = '1') THEN
165             data_out <= dram_data_out;
166
167         ELSE
168             data_out <= (OTHERS => 'Z');
169         END IF;
170
171     END PROCESS;
172
173     ------------------------------------------
174     -- DramWrite:
175     -- Process takes care of writing ram
176     -- again required due to synthesis limitations
177     ------------------------------------------
178     DramWrite : PROCESS (address, write_enable, read_enable)
179     BEGIN
180         IF (to_integer(unsigned(address)) <= WORD_COUNT-1) THEN
181             IF (write_enable = '1') THEN
182                 dram_address      <= address;
183                 write_enable_dram <= '1';
184             ELSE
185                 write_enable_dram <= '0';
186             END IF;
187
188             IF (read_enable = '1') THEN
189                 dram_address     <= address;
190                 read_enable_dram <= '1';
191             ELSE
192                 read_enable_dram <= '0';
193             END IF;
194         ELSE
195             write_enable_dram <= '0';
196             read_enable_dram  <= '0';
197
198         END IF;
199     END PROCESS;
200
201 END Behavioral;
```

P.12 I2SMonoIn

```
1  #include I2S_Config.hvhd
2
3  LIBRARY ieee;
4  USE ieee.std_logic_1164.ALL;
5
6  ENTITY I2SMonoIn IS
7      PORT
8      (
9          bit_clock       : IN std_logic; --bitclock in
10         word_select     : IN std_logic; --wordselect in
11         data_in         : IN std_logic; --data in
12         data_out        : OUT std_logic_vector(DATA_WIDTH-1 DOWNTO 0); -- data recieved from the i2s (little endian)
13         interrupt       : OUT std_logic; --interupt. is set high when data is available at data_out
14         interrupt_reset : IN std_logic --interupt reset. This should be set high when the data has been read from data_out,
               and will reset interrupt at the next clock.
15     );
16 END I2SMonoIn;
17
18 ARCHITECTURE Behavioral OF I2SMonoIn IS
19
20 BEGIN
21     Input : ENTITY work.i2sDriverIn
22     PORT MAP
23         (
24             bit_clock             => bit_clock,
25             word_select           => word_select,
26             data_in               => data_in,
27             data_out_right        => data_out,
28             interrupt_right       => interrupt,
29             interrupt_reset_right => interrupt_reset
30             --Ports for input
31         );
32 END Behavioral;
```

P.13 I2SSlave

```
#include I2S_Config.hvhd


LIBRARY ieee;
USE ieee.std_logic_1164.ALL;

ENTITY i2sDriverIn IS
    PORT
    (
        bit_clock             : IN std_logic := '0';-- Bitclock in
        word_select           : IN std_logic := '0';-- Worselect
        data_in               : IN std_logic := '0';-- Data in
        data_out_left         : OUT std_logic_vector(DATA_WIDTH -1 DOWNTO 0) := (OTHERS => '0'); -- One full word of data out
        data_out_right        : OUT std_logic_vector(DATA_WIDTH -1 DOWNTO 0) := (OTHERS => '0');
        interrupt_left        : OUT std_logic := '0';-- Interupt out. Is set high when a new word is ready
        interrupt_right       : OUT std_logic := '0';
        interrupt_reset_left  : IN std_logic  := '0';-- Interupt reset. Set this high to reset the interupt. Should be high
            untill intterupt put is low again.
        interrupt_reset_right : IN std_logic  := '0'
        --Ports for input
    );
END i2sDriverIn;

ARCHITECTURE i2sDriverIn OF i2sDriverIn IS
    SIGNAL lr      : std_logic := '1';--Internal wordselect, short for 'left right'. Right channel is active when '1'
    SIGNAL cnt     : INTEGER   := 0; -- Bit counter
    SIGNAL outBuff : std_logic_vector (DATA_WIDTH -1 DOWNTO 0) := (OTHERS => '0') ;-- The initial buffer for the serial data

BEGIN
    data_input : PROCESS (bit_clock, word_select)--The main process
        VARIABLE vcnt     : INTEGER                                  := 0; --Variable for the cnt signal
        VARIABLE voutBuff : std_logic_vector (DATA_WIDTH -1 DOWNTO 0) := x"0000";--variable for the output buffer
    BEGIN
        IF rising_edge(bit_clock) THEN

            --load variables
            vcnt     := cnt;
            voutBuff := outBuff;

            --reset interupts
            IF interrupt_reset_left = '1' THEN
```

```
41              interrupt_left <= '0';
42          END IF;
43          IF interrupt_reset_right = '1' THEN
44              interrupt_right <= '0';
45          END IF;
46
47          IF vcnt < (DATA_WIDTH ) THEN
48
49              voutBuff(DATA_WIDTH-1-vcnt) := data_in; -- Read data to buffer
50
51              vcnt            := vcnt + 1; --increment counter
52
53          END IF;
54
55          IF lr = NOT word_select THEN -- At this point the data change channel
56
57              --Loading the buffer the apropiate output
58              --The logic for truncating the data have been removed because it caused problems, another implementation is
                    possible if needed
59              IF lr = '1' THEN
60                  data_out_left <= voutbuff;
61
62              ELSE
63                  data_out_right <= voutbuff;
64
65              END IF;
66
67              voutbuff := (others=>'0'); -- Rady the buffer for the next word
68
69              lr <= word_select; --change the internal worselect
70
71              vcnt := 0; --reset counter
72
73              IF word_select = '0' THEN -- the buffer is now ready interrupt is set high.
74                  interrupt_right <= '1';
75
76              ELSE
77                  interrupt_left <= '1';
78              END IF;
79
80          ELSE
81          END IF;
82          -- The variables are saved in the signals
```

```
83              cnt     <= vcnt;
84              outBuff <= voutBuff;
85          END IF;
86      END PROCESS;
87  END i2sDriverIn;
```

P.14 I2SMonoOut

```
1  #include I2S_Config.hvhd
2
3  LIBRARY ieee;
4  USE ieee.std_logic_1164.ALL;
5  USE IEEE.STD_LOGIC_unsigned.ALL;
6  USE IEEE.NUMERIC_STD.ALL;
7
8  ENTITY I2SMonoOut IS
9      PORT
10     (
11
12         clk             : IN std_logic; -- Clock that will be used for logic and will be directly used as output bitclock. Can
               be directly connected to a syncronized bitclock input from the i2s input signal.
13         data_in         : IN std_logic_vector(DATA_WIDTH-1 DOWNTO 0);-- The data input. The word that should be loaded into
              the buffers should be loaded to these inputs when the interupts are set to high, and will be loaded at next
              falling_edge.
14         bit_clock_out   : OUT std_logic; -- Output bitclock for the output i2s signal.
15         word_select_out : OUT std_logic; --Output wordselect for the output i2s signal.
16         data_out        : OUT std_logic; -- Output serial data for the i2s signal.
17
18         interrupt_out   : OUT std_logic := '0';
19         interrupt_out_reset : IN std_logic := '0'
20     );
21 END I2SMonoOut;
22
23 ARCHITECTURE Behavioral OF I2SMonoOut IS
24     SIGNAL interrupt_reset_left, interrupt_reset_right, interrupt : std_logic;
25     SIGNAL interrupt_reset                                        : std_logic                                    := '0';
26     SIGNAL data_in_temp                                           : std_logic_vector(DATA_WIDTH-1 DOWNTO 0) := x"0000";
27     SIGNAL interrupt_count                                        : std_logic_vector(0 DOWNTO 0) := "0";
28 BEGIN
29
30     --Handles Interrupts
31     PROCESS (clk,data_in, interrupt_reset)
32     BEGIN
33         IF(rising_edge(clk)) THEN
34             IF (interrupt_reset = '1') THEN
35                 interrupt <= '0';
36                 IF(interrupt_count = "1") THEN
37                 interrupt_out <= '1';
38                 END IF;
```

```
39              ELSIF(interrupt_out_reset = '1') THEN
40                  interrupt_out <= '0';
41              ELSE
42                  data_in_temp <= data_in;
43                  interrupt    <= '1';
44              END IF;
45          END IF;
46      END PROCESS;
47
48      Output : ENTITY work.i2sDriverOut
49          PORT MAP
50          (
51              clk                   => clk,
52              interrupt_left        => interrupt,
53              interrupt_right       => interrupt,
54              interrupt_reset_left  => interrupt_reset_left,
55              interrupt_reset_right => interrupt_reset_right,
56              data_in_left          => data_in_temp,
57              data_in_right         => data_in_temp,
58              bit_clock             => bit_clock_out,
59              word_select           => word_select_out,
60              data_out              => data_out
61              --Ports for Output
62          );
63      interrupt_reset <= interrupt_reset_left AND interrupt_reset_right;
64
65      PROCESS(interrupt_reset)
66      BEGIN
67          IF(rising_edge(interrupt_reset)) THEN
68              interrupt_count <= std_logic_vector(unsigned(interrupt_count) + 1);
69          END IF;
70      END PROCESS;
71  END Behavioral;
```

P.15 I2SMaster

```
1  #include I2S_Config.hvhd
2
3  LIBRARY ieee;
4  USE ieee.std_logic_1164.ALL;
5
6  ENTITY i2sDriverOut IS
7      PORT
8      (
9
10         clk                    : IN std_logic;-- Clock that will be used for logic and will be directly used as output bitclock
               . Can be directly connected to a syncronized bitclock input from the i2s input signal.
11         interrupt_left         : IN std_logic  := '0';-- Interupt for when  new signals are stable at input. should be set high
               when a new word should be loaded into the output buffer.
12         interrupt_right        : IN std_logic  := '0';
13         interrupt_reset_left   : OUT std_logic := '0';--Interupt reset. Input is set to high when the input has been loaded to
               the input buffer.
14         interrupt_reset_right  : OUT std_logic := '0';
15         data_in_left           : IN std_logic_vector(DATA_WIDTH-1 DOWNTO 0);-- The data input buses. The word that should be
               loaded into the buffer should be loaded to these inputs when the interupts are set to high.
16         data_in_right          : IN std_logic_vector(DATA_WIDTH-1 DOWNTO 0);
17         bit_clock              : OUT std_logic;-- Output bitclock for the output i2s signal.
18         word_select            : OUT std_logic;--Output wordselect for the output i2s signal.
19         data_out               : OUT std_logic -- Output serial data for the i2s signal.
20
21     );
22 END i2sDriverOut;
23
24 ARCHITECTURE i2sDriverOut OF i2sDriverOut IS
25     SIGNAL buff_in_left       : std_logic_vector(DATA_WIDTH-1 DOWNTO 0) := (1 => '1', OTHERS => '1');--buffers for the input,
           they are loaded when the interupt are set high
26     SIGNAL buff_in_right      : std_logic_vector(DATA_WIDTH-1 DOWNTO 0) := (1 => '1', OTHERS => '1');
27     SIGNAL bit_counter        : INTEGER                                 := 0; --Counter for the bits
28     SIGNAL left_right_select  : std_logic                               := '0'; --Internal wordselect, short for 'left right'
           left channel is active when '1'
29 BEGIN
30
31     bit_clock <= clk; --bitclock is using the same clock as the logic
32
33     data_output : PROCESS (clk) --The main process
34     BEGIN
35         IF falling_edge(clk) THEN -- The logic is perfomed at the falling edge so that the signals are stable to read at the
```

```
        rising clock.

        IF bit_counter < (DATA_WIDTH-1 + 1) THEN
            IF left_right_select = '1' THEN --Write from the active buffer
                data_out <= buff_in_left(DATA_WIDTH -1- bit_counter);
            ELSE
                data_out <= buff_in_right(DATA_WIDTH -1- bit_counter);
            END IF;

        END IF;

        IF bit_counter + 1 >= (DATA_WIDTH-1 + 1) THEN --Change channel
            word_select       <= NOT left_right_select;
            left_right_select <= NOT left_right_select;

        END IF;
    END IF;

END PROCESS;

data_out_cnt : PROCESS (clk)
BEGIN
    IF rising_edge(clk) THEN -- The counting and buffering are done on the rising edge as they dont affect the active data
         out. (This is not necesary but was done to try and fix another problem that wasn't a problem)
        bit_counter <= bit_counter + 1;

        IF bit_counter + 1 >= (DATA_WIDTH-1 + 1) THEN --Change channel

            bit_counter <= 0;
        END IF;
        IF (interrupt_left = '1') AND(bit_counter = DATA_WIDTH-1) THEN --Update the word that is ready
            buff_in_left         <= data_in_left;
            interrupt_reset_left <= '1';
        END IF;
        IF (interrupt_right = '1') AND(bit_counter = DATA_WIDTH-1) THEN
            buff_in_right         <= data_in_right;
            interrupt_reset_right <= '1';
        END IF;

        IF (interrupt_left = '0') THEN --Update the word that is ready
            interrupt_reset_left <= '0';
        END IF;
        IF (interrupt_right = '0') THEN
```

```
77                  interrupt_reset_right <= '0';
78              END IF;
79
80          END IF;
81
82      END PROCESS;
83
84  END i2sDriverOut;
```

P.16 Button

```
#include "Config.hvhd"

LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
--Debouncer for 3 buttons
--Debounces the buttons 'btn' and debounced output 'debounced_btn_out' by checking 'btn' for 3 clock cycles
ENTITY btndriver IS
    PORT
    (
        clk               : IN STD_LOGIC; --Clock used for debouncing
        clr               : IN STD_LOGIC; --Clear
        btn               : IN STD_LOGIC_vector (2 DOWNTO 0)  := (OTHERS => '0'); --Button inputs
        debounced_btn_out : OUT STD_LOGIC_vector (2 DOWNTO 0) := (OTHERS => '0'); --Debounced button output
        interrupt_on      : OUT std_logic                     := '0';   --Send interrut signal out
        interrupt_reset   : IN std_logic                      := '0'    --Resets interrupt signal when set high
    );
END btndriver;

ARCHITECTURE Behavioral OF btndriver IS
    SIGNAL Q0, Q1, Q2  : std_logic_vector (2 DOWNTO 0) := (OTHERS => '0'); -- Debounce registers
    SIGNAL dbtn_buffer : std_logic_vector(2 DOWNTO 0)  := (OTHERS => '0');
    SIGNAL int_toggle  : std_logic                     := '0';
BEGIN

    ----------------------------------------
    -- Debounce:
    -- Debounces a button press by ensuring the button
    -- has been pressed for 3 clock cycles, thus
    -- valiting the button press
    ----------------------------------------
    Debounce : PROCESS (clk)
    BEGIN
        IF (rising_edge(clk)) THEN
            IF (clr = '1') THEN --Clear
                Q0 <= (OTHERS => '0');
                Q1 <= (OTHERS => '0');
                Q2 <= (OTHERS => '0');

            ELSE --Shift registers for each button
                Q0(2)          <= btn(0);
                Q0(1 DOWNTO 0) <= Q0(2 DOWNTO 1);
```

```
42                 Q1(2)           <= btn(1);
43                 Q1(1 DOWNTO 0) <= Q1(2 DOWNTO 1);
44                 Q2(2)           <= btn(2);
45                 Q2(1 DOWNTO 0) <= Q2(2 DOWNTO 1);
46             END IF;
47         END IF;
48     END PROCESS;
49     --If all values in shift regisers are on the output will be on.
50     dbtn_buffer(0) <= Q0(1) AND Q0(2) AND Q0(0);
51     dbtn_buffer(1) <= Q1(1) AND Q1(2) AND Q1(0);
52     dbtn_buffer(2) <= Q2(1) AND Q2(2) AND Q2(0);
53
54     ---------------------------------------------
55     -- Interrupt:
56     -- Sets interrupt signal high when any combination of button is pressed
57     -- Disables when "int_toggle" is set high.
58     -- An interrupt signal can then only be set once all buttons are released
59     ---------------------------------------------
60     Interrupt : PROCESS (clk,dbtn_buffer, int_toggle)
61     BEGIN
62         IF rising_edge(clk) THEN
63             IF (dbtn_buffer = "000" OR dbtn_buffer = "ZZZ" OR dbtn_buffer = "UUU" OR int_toggle = '1') THEN
64                 interrupt_on <= '0';
65             ELSE
66                 interrupt_on <= '1';
67             END IF;
68         END IF;
69     END PROCESS;
70
71     ---------------------------------------------
72     -- InterruptReset:
73     -- Sets "int_toggle"
74     -- Sets "int_toggle" when "interrupt_reset" is high
75     -- Resets "int_toggle" back to zero once all buttons are released
76     ---------------------------------------------
77     InterruptReset : PROCESS (interrupt_reset, dbtn_buffer)
78     BEGIN
79             IF (interrupt_reset = '1') THEN
80                 int_toggle <= '1';
81             ELSIF (dbtn_buffer = "000") THEN
82                 int_toggle <= '0';
83             END IF;
84     END PROCESS;
```

```
85      debounced_btn_out <= dbtn_buffer;
86  END Behavioral;
```

P.17 DataRAM

```
#include "Config.hvhd"
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.numeric_std.ALL;

ENTITY Memory IS
    PORT
    (
        data_in       : IN STD_LOGIC_VECTOR (WORD_SIZE-1 DOWNTO 0); --Data in
        data_out      : OUT STD_LOGIC_VECTOR (WORD_SIZE-1 DOWNTO 0) := (OTHERS => '0'); --Data Out
        address       : IN STD_LOGIC_VECTOR (WORD_SIZE-1 DOWNTO 0); --address bus
        write_enable  : IN STD_LOGIC; -- Write Enable
        read_enable   : IN STD_LOGIC; -- Read Enable
        clk           : IN STD_LOGIC -- Clock
    );

END Memory;

ARCHITECTURE falling OF Memory IS

    TYPE ram_type IS ARRAY (WORD_COUNT-1 DOWNTO 0) OF std_logic_vector(WORD_SIZE-1 DOWNTO 0); -- Total 516k memory bits 8k*32
        = 256k we use 50% for DataMemory and 50% for ProgramMemory
    SIGNAL RAM : ram_type := (OTHERS => x"0000");
BEGIN

    ----------------------------------------
    -- MemoryReadWrite:
    -- Reads and writes data from the data RAM
    ----------------------------------------
    MemoryReadWrite : PROCESS (clk)
    BEGIN
        IF (falling_edge(clk)) THEN -- Start when the clock rises
            IF write_enable = '1' THEN -- Write enable
                RAM(to_integer(unsigned(address))) <= data_in; --write Data In bus into RAM array at position address
            END IF;
            IF read_enable = '1' THEN -- Read enable
                data_out <= RAM(to_integer(unsigned(address))); -- writes RAM array at the address position into Data Out bus.
            END IF;
            IF write_enable = '0' AND read_enable = '0' THEN --if not writing or reading set all to high impedance to make
                sure nothing unintended happens
                data_out <= (OTHERS => 'Z');
```

```
40                END IF;
41            END IF;
42        END PROCESS;
43 END falling;
```

P.18 Interrupt

```
1  #include "Config.hvhd"
2
3  #define DEFAULT_BEHAVIOUR interrupt_cpu <= '0'; \
4  interrupt_btn_reset_latch <= '0';\
5  interrupt_i2s_reset_latch <= '0';\
6  interrupt_i2s_out_reset_latch <= '0';\
7  interrupt_nest_enable <= '1';\
8  \
9
10 LIBRARY IEEE;
11 USE IEEE.STD_LOGIC_1164.ALL;
12 USE IEEE.STD_LOGIC_unsigned.ALL;
13 USE IEEE.NUMERIC_STD.ALL;
14
15 ENTITY Interrupt IS
16     PORT (
17         interrupt_btn : IN std_logic;                          --Is set high by the button peripheral to interrupt the CPU
18         interrupt_btn_reset : INOUT std_logic;                 --Is set high by the interrupt controller to reset the button
               peripheral's interrupt signal
19
20         interrupt_i2s : IN std_logic := '0';                   --Is set high by the i2s peripheral to interrupt the CPU
21         interrupt_i2s_reset : INOUT std_logic := '0';          --Is set high by the interrupt controller to reset the i2s
               peripheral's interrupt signal
22
23         interrupt_i2s_out : IN std_logic := '0';
24         interrupt_i2s_out_reset : INOUT std_logic := '0';
25
26         interrupt_enable : IN std_logic := '1';                --Enables/Disables interrupts entirely
27         interrupt_nest_enable : OUT std_logic := '1' ;         --Tells the Main logic that whether nesting interrups are allowed.
               MUST ben high by default
28         write_enable :IN std_logic;                            --Write enable to allow changes to the interrupt controllers
               configuration registers
29         clk : IN std_logic;                                    --System Clock
30
31         internal_register_address : IN std_logic_vector(2 downto 0);     --Addresses the configuration registers
32         data_in : IN std_logic_vector(ADDRESS_SIZE-1 downto 0);           --Input data to be written to the configuration
               registers
33         interrupt_address : OUT std_logic_vector(PC_SIZE-1 downto 0);          --Address of the first instruction of an
               interrupt service routine
34         interrupt_cpu : OUT std_logic                                    --Signals the CPU that an interrupt has occoured
35     );
```

```
36 END Interrupt;
37
38 ARCHITECTURE Behavioral OF Interrupt IS
39  TYPE register_type IS ARRAY (5 DOWNTO 0) OF std_logic_vector(PC_SIZE - 1 DOWNTO 0);
40     SIGNAL REG : register_type := (others => (others => '0'));
41
42     --These signals are configurable by the programmer
43     SIGNAL interrupt_btn_enable : std_logic := '0';          --Enables/Disables Button Interrupts
44     SIGNAL interrupt_btn_nest_enable : std_logic := '0';     --Enables/Disables nesting of Button Interrupts
45     SIGNAL interrupt_btn_priority : integer := 0;            --Priority of the Button interrupt (currently not implemented)
46
47     SIGNAL interrupt_btn_latch : std_logic := '0';           --Latches Button Interrupts
48     SIGNAL interrupt_btn_reset_latch : std_logic := '0';     --Used to make sure that resat interrupts can not run until their
           interrupt is set low again
49                                                              --(so we aren't relying on a peripherals ability to quickly reset
                                                                    its interrupt)
50
51     --Same functionaly as the button signals, just for
52     --the i2s peripheral
53     SIGNAL interrupt_i2s_enable : std_logic := '0';
54     SIGNAL interrupt_i2s_nest_enable : std_logic := '0';
55     SIGNAL interrupt_i2s_priority : integer := 0;
56
57     SIGNAL interrupt_i2s_latch : std_logic := '0';
58     SIGNAL interrupt_i2s_reset_latch : std_logic := '0';       --Used to make sure that resat interrupts can not run until
           their interrupt is set low again
59                                                              --(so we aren't relying on a peripherals ability to quickly reset
                                                                    its interrupt)
60
61     SIGNAL interrupt_i2s_out_enable : std_logic := '0';
62     SIGNAL interrupt_i2s_out_nest_enable : std_logic := '0';
63     SIGNAL interrupt_i2s_out_priority : integer := 0;
64
65     SIGNAL interrupt_i2s_out_latch : std_logic := '0';
66     SIGNAL interrupt_i2s_out_reset_latch : std_logic := '0';
67
68     SIGNAL false_signal : std_logic := '0';
69 BEGIN
70
71     ---------------------------------------------
72     -- InterruptCpu:
73     -- Process wait for an interrupt signal from a
74     -- peripheral, and then sends an interrupt
```

```
75      -- signal to the CPU along with an accompanying
76      -- ISR address, and a bool informing the CPU
77      -- if nesting is allowed
78      ------------------------------------------
79      InterruptCpu : process(clk)
80      BEGIN
81          if(falling_edge(clk)) Then
82              IF(interrupt_enable = '1') Then
83                  IF(false_signal = '1') THEN
84
85
86                  ELSIF( Interrupt_btn_latch  = '1') Then
87                      IF(Interrupt_btn_enable = '1') THEN                    --if button enabled and interrupted
88                           interrupt_cpu <= '1';                             --send iterrupt signal to CPU
89                          Interrupt_btn_reset_latch <= '1';                  --send signal to reset button interrupt
90                          interrupt_address <= REG(0);                       --Send ISR address to CPU
91                          interrupt_nest_enable <= Interrupt_btn_nest_enable; --Send enable state to CPU
92                      ELSE --If ISR not enabled, then do not interrupt
93                          interrupt_cpu <= '0';
94                          interrupt_btn_reset_latch <= '1';
95                          interrupt_i2s_reset_latch <= '0';
96                          interrupt_i2s_out_reset_latch <= '0';
97                          interrupt_nest_enable <= '1';
98                      END IF;
99
100                 --Same procedure for the other ISR
101                 ELSIF( interrupt_i2s_latch  = '1') Then
102                     IF(interrupt_i2s_enable = '1') THEN
103                         interrupt_cpu <= '1';
104                         interrupt_i2s_reset_latch <= '1';
105                         interrupt_address <= REG(2);
106                         interrupt_nest_enable <= interrupt_i2s_nest_enable;
107                     ELSE
108                         interrupt_cpu <= '0';
109                         interrupt_btn_reset_latch <= '0';
110                         interrupt_i2s_reset_latch <= '1';
111                         interrupt_i2s_out_reset_latch <= '0';
112                         interrupt_nest_enable <= '1';
113                     END IF;
114                 ELSIF( interrupt_i2s_out_latch  = '1') Then
115                     IF(interrupt_i2s_out_enable = '1') THEN
116                         interrupt_cpu <= '1';
117                         interrupt_i2s_out_reset_latch <= '1';
```

```
118                     interrupt_address <= REG(4);
119                     interrupt_nest_enable <= interrupt_i2s_out_nest_enable;
120                 ELSE
121                     interrupt_cpu <= '0';
122                     interrupt_btn_reset_latch <= '0';
123                     interrupt_i2s_reset_latch <= '0';
124                     interrupt_i2s_out_reset_latch <= '1';
125                     interrupt_nest_enable <= '1';
126                 END IF;
127 
128             ELSE
129                 DEFAULT_BEHAVIOUR
130             END IF;
131         ELSE
132             DEFAULT_BEHAVIOUR
133         End IF;
134     END IF;
135 end process;
136 
137 ------------------------------------------
138 -- BtnResetLatching:
139 -- Latches the reset signal to the button peripheral
140 -- Used to make sure that the button interrupts can not run until their interrupt is set low again
141 --
142 -- In practice this means that interrupts are dependent on changes (e.g rising /falling edges)
143 -- from the received interrupt signals. This ensures that the CPU does not rely on a
144 -- peripherals ability to quickly reset its interrupt
145 ------------------------------------------
146 --BtnResetLatching : INTERRUPT_ RESET(interrupt_btn)
147 BtnResetLatching : PROCESS(clk)
148 BEGIN
149     IF(falling_edge(clk)) THEN
150         IF(interrupt_btn_reset_latch = '1') THEN
151             interrupt_btn_latch <= '0';
152         ElSIF(interrupt_btn = '1') THEN
153             interrupt_btn_latch <= '1';
154         END IF;
155         IF (interrupt_btn_reset_latch = '1') THEN
156             interrupt_btn_reset <= '1';
157         ELSIF(interrupt_btn_reset = '1' AND interrupt_btn = '0') THEN
158             interrupt_btn_reset <= '0';
159         END IF;
160     END IF;
```

```
161     END PROCESS;
162
163     ------------------------------------------
164     -- I2sResetLatching:
165     -- Latches the reset signal to the I2S peripheral
166     -- Used to make sure that the i2s interrupts can not run until their interrupt is set low again
167     --
168     -- In practice this means that interrupts are dependent on changes (e.g rising /falling edges)
169     -- from the received interrupt signals. This ensures that the CPU does not rely on a
170     -- peripherals ability to quickly reset its interrupt
171     ------------------------------------------\
172     I2sResetLatching : PROCESS(clk)
173     BEGIN
174         IF(falling_edge(clk)) THEN
175             IF(interrupt_i2s_reset_latch = '1') THEN
176                 interrupt_i2s_latch <= '0';
177             ElSIF(interrupt_i2s = '1' AND interrupt_i2s_reset = '0') THEN
178                 interrupt_i2s_latch <= '1';
179             END IF;
180             IF (interrupt_i2s_reset_latch = '1') THEN
181                 interrupt_i2s_reset <= '1';
182             ELSIF(interrupt_i2s_reset = '1' AND interrupt_i2s = '0') THEN
183                 interrupt_i2s_reset <= '0';
184             END IF;
185         END IF;
186
187     END PROCESS;
188
189         I2sOutResetLatching : PROCESS(clk)
190     BEGIN
191         IF(falling_edge(clk)) THEN
192             IF(interrupt_i2s_out_reset_latch = '1') THEN
193                 interrupt_i2s_out_latch <= '0';
194             ElSIF(interrupt_i2s_out = '1' AND interrupt_i2s_out_reset = '0') THEN
195                 interrupt_i2s_out_latch <= '1';
196             END IF;
197             IF (interrupt_i2s_out_reset_latch = '1') THEN
198                 interrupt_i2s_out_reset <= '1';
199             ELSIF(interrupt_i2s_out_reset = '1' AND interrupt_i2s_out = '0') THEN
200                 interrupt_i2s_out_reset <= '0';
201             END IF;
202         END IF;
203
```

```
204     END PROCESS;
205
206
207     ----------------------------------------------
208     -- WriteProccess:
209     -- Writes data to the configuration registers
210     ----------------------------------------------
211     WriteProccess : PROCESS (clk) IS
212     BEGIN
213         IF(rising_edge(clk)) THEN
214             IF (write_enable = '1') THEN
215                 REG(to_integer(unsigned(internal_register_address))) <= data_in(PC_SIZE-1 downto 0);
216             END IF;
217         END IF;
218     END PROCESS;
219
220     --Maps registers to control signals
221     interrupt_btn_enable     <= REG(1)(9);
222     interrupt_i2s_enable     <= REG(3)(9);
223     interrupt_i2s_out_enable <= REG(5)(9);
224
225
226     interrupt_btn_nest_enable       <= REG(1)(8);
227     interrupt_i2s_nest_enable       <= REG(3)(8);
228     interrupt_i2s_out_nest_enable   <= REG(5)(8);
229
230
231
232
233     interrupt_btn_priority      <= to_integer(unsigned(REG(1)(7 downto 0)));
234     interrupt_i2s_priority      <= to_integer(unsigned(REG(3)(7 downto 0)));
235     interrupt_i2s_out_priority  <= to_integer(unsigned(REG(5)(7 downto 0)));
236
237
238
239 END Behavioral;
```

P.19 SevenSegmentDisplay

```
1
2  --Macro mapping digits (0-F) to the Seven segment display on the Terasic DE0
3  #define SEVEN_SEGMENT_MAP "1000000" WHEN "0000", \
4  "1111001" WHEN "0001", \
5  "0100100" WHEN "0010", \
6  "0110000" WHEN "0011", \
7  "0011001" WHEN "0100", \
8  "0010010" WHEN "0101", \
9  "0000010" WHEN "0110", \
10 "1111000" WHEN "0111", \
11 "0000000" WHEN "1000", \
12 "0010000" WHEN "1001", \
13 "0001000" WHEN "1010", \
14 "0000011" WHEN "1011", \
15 "1000110" WHEN "1100", \
16 "0100001" WHEN "1101", \
17 "0000110" WHEN "1110", \
18 "0001110" WHEN "1111", \
19 "0000000" WHEN OTHERS;
20
21 LIBRARY IEEE;
22 USE IEEE.STD_LOGIC_1164.ALL;
23 USE IEEE.STD_LOGIC_unsigned.ALL;
24 USE IEEE.NUMERIC_STD.ALL;
25
26 --Seven segment display driver
27 --This driver is for a 4 panel sevensegment display with all segments directly connected
28
29 ENTITY ssgddriver IS
30     PORT
31     (
32         clr                         : IN STD_LOGIC; --Clear
33         bcd_enable                  : IN std_LOGIC; --Enable BCD format
34         input_data                  : IN STD_LOGIC_vector (15 DOWNTO 0) := (OTHERS => '0'); --Hex data in 4 nibbles
35         dot_control                 : IN STD_LOGIC_vector (3 DOWNTO 0); --Dots data
36         seven_seg_control_signals : OUT STD_LOGIC_vector (31 DOWNTO 0) --Segments connections (31| dot4 - 7seg4 - dot3 - 7seg3
               - dot2 - 7seg2 - dot1 - 7seg1 |0)
37     );
38 END ssgddriver;
39
40 ARCHITECTURE Behavioral OF ssgddriver IS
```

```
41
42      COMPONENT b2bcd IS --When input is changed the output is set to the BCD format using the 'shift and add 3' algorithm
43          PORT
44          (
45              clr     : IN std_logic; --Clear
46              binary  : IN std_logic_vector (15 DOWNTO 0); --Binary data in
47              bcd     : OUT std_logic_vector (15 DOWNTO 0) --BCD formatet output
48          );
49      END COMPONENT;
50
51      SIGNAL bcd      : std_logic_vector (15 DOWNTO 0);
52      SIGNAL display  : std_logic_vector (15 DOWNTO 0);
53
54  BEGIN
55      con : COMPONENT b2bcd
56          PORT MAP
57          (
58              clr     => clr,
59              binary  => input_data,
60              bcd     => bcd
61          );
62
63          display <=
64              bcd WHEN bcd_enable = '1' ELSE
65              input_data;
66
67          --Sets the first display
68          seven_seg_control_signals(7)                                        <= dot_control(0);
69          WITH display(3 DOWNTO 0) SELECT seven_seg_control_signals(6 DOWNTO 0) <=
70          SEVEN_SEGMENT_MAP
71
72          --Sets the second display
73          seven_seg_control_signals(15)                                         <= dot_control(1);
74          WITH display(7 DOWNTO 4) SELECT seven_seg_control_signals(14 DOWNTO 8) <=
75          SEVEN_SEGMENT_MAP
76
77          --Sets the third display
78          seven_seg_control_signals(23)                                           <= dot_control(2);
79          WITH display(11 DOWNTO 8) SELECT seven_seg_control_signals(22 DOWNTO 16) <=
80          SEVEN_SEGMENT_MAP
81
82          --Sets the fourth display
83          seven_seg_control_signals(31)                                            <= dot_control(3);
```

```
84          WITH display(15 DOWNTO 12) SELECT seven_seg_control_signals(30 DOWNTO 24) <=
85          SEVEN_SEGMENT_MAP
86
87      END Behavioral;
```

P.20 BinaryCodedDecimal

```
1  LIBRARY ieee;
2  USE ieee.std_logic_1164.ALL;
3  USE ieee.std_logic_unsigned.ALL;
4
5  ENTITY b2bcd IS
6      PORT
7      (
8          clr    : IN std_logic;
9          binary : IN std_logic_vector (15 DOWNTO 0); --Input value
10         bcd    : OUT std_logic_vector (15 DOWNTO 0) --Binary coded decimal representation of that value
11     );
12 END b2bcd;
13 ARCHITECTURE a OF b2bcd IS
14 BEGIN
15
16     ----------------------------------------
17     -- b2bcd:
18     -- converts a number to binary coded decimal
19     -- Note that it is only possible to represent numbers up to
20     -- 9999 with 16 available output BITS
21     -- Uses double-dabble algorithm
22     ----------------------------------------
23     b2bcd : PROCESS (binary, clr)
24         VARIABLE temp : std_logic_vector(31 DOWNTO 0);
25     BEGIN
26         temp := x"00000000";
27         IF clr = '0' THEN
28             temp(18 DOWNTO 3) := binary;
29
30             FOR i IN 0 TO 12 LOOP
31                 IF temp(19 DOWNTO 16) > 4 THEN
32                     temp(19 DOWNTO 16) := temp(19 DOWNTO 16) + 3;
33                 END IF;
34                 IF temp(23 DOWNTO 20) > 4 THEN
35                     temp(23 DOWNTO 20) := temp(23 DOWNTO 20) + 3;
36                 END IF;
37                 IF temp(27 DOWNTO 24) > 4 THEN
38                     temp(27 DOWNTO 24) := temp(27 DOWNTO 24) + 3;
39                 END IF;
40                 IF temp(31 DOWNTO 28) > 4 THEN
41                     temp(31 DOWNTO 28) := temp(31 DOWNTO 28) + 3;
```

```
42                  END IF;
43                  temp(31 DOWNTO 1) := temp(30 DOWNTO 0);
44              END LOOP;
45          END IF;
46          bcd <= temp(31 DOWNTO 16);
47      END PROCESS b2bcd;
48  END a;
```

Appendix Q

Assembler Code

```
1    #include <stdio.h>
2  #include <string.h>
3  #include <stdlib.h>
4  #include <stdbool.h>
5  #include <stdint.h>
6
7  //array size for input buffer
8  #define bufferSize 100
9
10 //opcode types for operand identification
11 #define IMMEDIATE 1
12 #define REGISTER 2
13 #define MEMORY 3
14 #define JUMP 4
15
16 //pointers for IO files
17 FILE *fpIn;
18 FILE *fpOut;
19
20 int main(int argc, char *argv[]) //main takes 2 arguments, both are names of the in- and output files respectively
21 {
22     //the buffer array is used to store either the arguments of main or filenames input from the user.
23     //This is because the program can be run from a command line with arguments or as a .exe without.
24     char buffer[3][100];
25     int memoryDepth = 8192;
26     //The arguments are counted in argc and if there are 2 (3 because argc counts as an argument)
27     //normal execution will occur. Else the program will prompt for input or exit
28     if (argc == 1)
29     {
```

```
30          printf("Zero arguments given...");
31          printf("\nDid you run the program as .exe? Y/N: ");
32
33          if (getchar() == 'y' || getchar() == 'Y')
34          {
35              printf("\nWould you like to give arguments now? Y/N: ");
36              getchar();
37
38              if (getchar() == 'y' || getchar() == 'Y')
39              {
40                  printf("\nWrite <input-filename.extension>: ");
41                  getchar();
42                  fgets(buffer[0], 100, stdin);
43                  printf("\nWrite <output-filename.extension>: ");
44                  fgets(buffer[1], 100, stdin);
45                  printf("\nWrite <memory depth>: ");
46                  fgets(buffer[2], 100, stdin);
47
48                  strtok(buffer[0], "\n");
49                  strtok(buffer[1], "\n");
50                  strtok(buffer[2], "\n");
51                  memoryDepth = atoi(buffer[2]);
52              }
53              else
54              {
55                  printf("Press enter to exit...");
56                  getchar();
57                  return(0);
58              }
59          }
60          else
61          {
62              printf("No? Try again with more arguments.\nPress enter to exit...");
63              getchar();
64              return(0);
65          }
66
67
68      }
69      else if (argc == 2)
70      {
71          printf("Argument given is: %s. One more argument is needed.\n", argv[1]);
72          getchar();
```

```
        return(0);
    }
    else if (argc == 3)
    {
        printf("Arguments given are: %s & %s.\nDefault memory depth will be used (1024)\n", argv[1], argv[2]);
        strcpy(buffer[0], argv[1]);
        strcpy(buffer[1], argv[2]);
    }
    else if (argc == 4)
    {
        printf("Arguments given are: %s, %s & %s.\n", argv[1], argv[2], argv[3]);
        strcpy(buffer[0], argv[1]);
        strcpy(buffer[1], argv[2]);
        strcpy(buffer[2], argv[3]);

        memoryDepth = atoi(buffer[2]);
        if (memoryDepth <= 0)
        {
            printf("Memeory depth is not a number.\nPress enter to exit...");
            getchar();
            return(0);
        }
        else
        {
            printf("Memeory depth will be %d.\n", memoryDepth);
        }
    }
    else
    {
        printf("Too many arguments given...\nPress enter to exit...");
        getchar();
        return(0);
    }

    char temp[bufferSize]; //temporary input array for reading a line of assembly code at a time
    int opCount; //used to count the number of operands in a given line to make is less "hard coded"
    int opcType; //used to define what type of opcode is read
    int lineCount = 0; //used to keep track of label/jump locations as well as outputting to .mif format

    char labelName[100][50]; //array used to store label names
    int labelLine[100]; //array used to store the linenumber corresponding to a given label name
    int labelindex = 0; //every time a label is stored, the index of where to store the next label increments
    bool label; //used to keep track of if a line contains a label
```

```
116
117     int outputIndex = 0; // This is used to determine where in the output array the next output is stored.
118     int outputType = 0; // This is used to determine what order to print the output in.
119     char output[7][17];// Array with all outputs. used to output in different order depending on opcode type and special cases
120     int immMem = 0; //used to keep track of a specific case for memory addresses which contain $rs+imm
121
122     int regJump = 0;
123     fpIn = fopen(buffer[0], "r"); //opens input file in read-only mode
124
125     if (!fpIn) //read file doesnt exist
126     {
127         printf("Read file does not exist.\nPress enter to exit...");
128         getchar();
129         return(0);
130     }
131
132 #pragma region LabelFinder
133     //Finds all labels before actual assembly occurs which allows for jumps in the actual assembly
134     while (1)
135     {
136         if (!fgets(temp, bufferSize, fpIn))//end of file
137         {
138             lineCount = 0; //resets linecount for use in assembly stage
139             printf("%d label(s) were found\n", labelindex); //mostly for debugging
140             printf("Label-check done...\n\n");
141             break;
142         }
143
144         for (int i = 0; i < bufferSize; i++) //comment remover
145         {
146             if (temp[i] == '/' && temp[i + 1] == '/') //comment is found and replaced by a new line and stringend
147             {
148                 temp[i] = '\n';
149                 temp[i + 1] = '\0';
150                 break;
151             }
152         }
153
154         if (strstr(temp, ":")) //if temp contains ":" then a label is present and it's name and line number is saved
155         {
156             for (int i = 0; i < bufferSize; i++)
157             {
158                 if (temp[i] == ':')
```

```
159                 {
160                     break;
161                 }
162                 labelName[labelindex][i] = temp[i];
163                 labelName[labelindex][i + 1] = '\0';
164                 strcat(labelName[labelindex], " "); //appends a space to the end of labelname in case two labels start with
                        the same name.
165                                                     // eg. label and label1. both would contain "label" but not "label "
166             }
167             labelLine[labelindex] = lineCount;
168             labelindex++;
169         }
170         if (temp[0] != '\n' && !strstr(temp,":")) //if the line is not a newline or contains a label, linecount is incremented
171         {
172             lineCount++;
173         }
174 
175     }
176 #pragma endregion
177 
178     fclose(fpIn); //closes input file to reopen it from the beginning
179     fpIn = fopen(buffer[0], "r"); //opens input file in read only mode
180 
181     if (!fpIn) //read file doesnt exist
182     {
183         printf("Read file does not exist...");
184         getchar();
185         return(0);
186     }
187 
188     fpOut = fopen(buffer[1], "r"); //opens output file in read mode, this is to check if it exists
189 
190     if (!fpOut) //read file doesnt exist, asks if user would like to create file
191     {
192         printf("Write file does not exist. Create write file? Y/N: ");
193         if (getchar() == 'y' || getchar() == 'Y') //create file
194         {
195             fpOut = fopen(buffer[1], "w");
196         }
197         else //exit program
198         {
199             printf("Press enter to exit...");
200             getchar();
```

```
201             return(0);
202         }
203 
204     }
205     else //if file exists, open file in write only mode.
206     {
207         fpOut = fopen(buffer[1], "w");
208     }
209 
210 #pragma region MIF prep
211     //outputs to the start of the outputfile. These lines are needed to load a .mif onto the cpu
212     fprintf(fpOut, "DEPTH = %d;\n", memoryDepth);
213     fprintf(fpOut, "WIDTH = 32;\n");
214     fprintf(fpOut, "ADDRESS_RADIX = UNS;\n");
215     fprintf(fpOut, "DATA_RADIX = BIN;\n\n");
216     fprintf(fpOut, "CONTENT\n");
217     fprintf(fpOut, "BEGIN\n\n");
218 #pragma endregion
219 
220     while (1)
221     {
222         strcpy(output[5], "00000");  //used to pad output with 5 zeros
223         strcpy(output[6], "000000");  //used to pad output with 6 zeros
224 
225         label = false; //resets if line contains label
226         opCount = 0; //clears operandCount
227 
228         for (int i = 0; i < bufferSize; i++)
229         {
230             temp[i] = '\0'; //clears temp since fgets does not
231         }
232 
233         if (!fgets(temp, bufferSize, fpIn))//reads a line from the input file for a maximum of 100 chars. breaks if no line is
                read
234         {
235             break; //breaks the while loop if no more lines are present in input file
236         }
237 
238         strcat(temp, " "); //adds space at the end of line to allow operandFinder to work properly if line doesnt end with ' '
                or '\n'
239 
240         for (int i = 0; i < bufferSize; i++) //replaces all new lines with spaces since spaces are used to find operands
241         {
```

```
242             if (temp[i] == '\n')
243             {
244                 temp[i] = ' ';
245                 break;
246             }
247         }
248 
249 #pragma region commentFinder
250         //checks temp for comments and replaces the comment with two spaces and a string end.
251         //also replaces the rest of the string with char \0, which is the string terminator char.
252         for (int i = 0; i < bufferSize; i++)
253         {
254             if (temp[i] == '/' && temp[i + 1] == '/')
255             {
256                 //printf("comment found"); //debugging
257                 for (int u = i; u < bufferSize-i; u++)
258                 {
259                     temp[u] = '\0';
260                 }
261                 temp[i] = ' ';
262                 temp[i + 1] = ' ';
263                 temp[i + 2] = '\0';
264                 break;
265             }
266         }
267 #pragma endregion
268 
269 #pragma region labelFinder
270         //Checks for labels in the input array and sets the label bool to true.
271         //This ensures that the line is ignore in assembly
272         if (strstr(temp, ":"))
273         {
274             printf("Label Found");
275             label = true;
276             opcType = 0;
277         }
278 #pragma endregion
279 
280 #pragma region MIF LinePrint
281         //outputs some required syntax for .mif format before the actual binary opcode
282         for (int i = 0; i < bufferSize; i++)
283         {
284             if (temp[i] != ' ' && temp[i] != '\n' && temp[i] != '\0' && temp[i] != 9)
```

```
285             {
286                 opcType = 5;
287                 break;
288             }
289             else
290             {
291                 opcType = 0;
292             }
293         }
294 
295         if (temp[0] == ' ' && temp[1] == ' ' && temp[2] == '\0') //checks for empty line
296         {
297             opcType = 0;
298         }
299         if (temp[0] == '\0') //checks for empty string as well
300         {
301             opcType = 0;
302         }
303         //else if (label == false) //if no label exists, the opcode type is set to non zero but not 1-4
304         //{
305         //  opcType = 5;
306         //}
307 
308         if (label == false && opcType != 0) // if opcode is present, outputs required syntax
309         {
310             //allows for cleaner formatting of output
311             if (lineCount < 10)
312             {
313                 fprintf(fpOut, "%d   :   ", lineCount);
314             }
315             else if (lineCount < 100)
316             {
317                 fprintf(fpOut, "%d  :   ", lineCount);
318             }
319             else if (lineCount < 1000)
320             {
321                 fprintf(fpOut, "%d :   ", lineCount);
322             }
323             else if (lineCount < 10000)
324             {
325                 fprintf(fpOut, "%d:   ", lineCount);
326             }
327         }
```

```
328 #pragma endregion
329
330 #pragma region opcodeFinder
331
332 #pragma region opcArr
333         //array that contains all opcodes. used for comparison with input
334         char opcArr[100][15];
335         strcpy(opcArr[0], "ADDI "); //ADD immediate
336         strcpy(opcArr[1], "SUBI "); //substract immediate
337         strcpy(opcArr[2], "ADDCI "); // Add carry immediate
338         strcpy(opcArr[3], "NEGI "); // Negate immediate
339         strcpy(opcArr[4], "ANDI "); // and immediate
340         strcpy(opcArr[5], "XORI "); // XOR immediate
341         strcpy(opcArr[6], "ORI "); // or immediate
342         strcpy(opcArr[7], "MULTI "); // multiply immediate
343         strcpy(opcArr[8], "LSLI "); // logic shift left immediate
344         strcpy(opcArr[9], "LSRI "); // Logic shift Right immediate
345         strcpy(opcArr[10], "RASI "); // right arithmetic shift immediate
346         strcpy(opcArr[11], "CMPI "); // compare immediate
347         strcpy(opcArr[12], "MOVI "); // Move immediate
348
349         strcpy(opcArr[13], "ADDR "); // ADD register
350         strcpy(opcArr[14], "ADDC "); // ADD carry
351         strcpy(opcArr[15], "SUBR "); // substract register
352         strcpy(opcArr[16], "NEGR "); // NEGATE register
353         strcpy(opcArr[17], "ANDR "); // AND register
354         strcpy(opcArr[18], "XORR "); // XOR register
355         strcpy(opcArr[19], "ORR "); // OR register
356         strcpy(opcArr[20], "MULT "); // Multiply register
357         strcpy(opcArr[21], "LSLR "); // Logic shift Left register
358         strcpy(opcArr[22], "LSRR "); // Logic shift right register
359         strcpy(opcArr[23], "RASR "); // right arithmetic shift register
360         strcpy(opcArr[24], "CMP "); // compare Register
361         strcpy(opcArr[25], "MOV "); // Move Register
362
363         strcpy(opcArr[26], "JMP "); // Jump label
364         strcpy(opcArr[27], "JMPNQ "); // jump NOT EQUAL TO
365         strcpy(opcArr[28], "JMPLE "); // Jump Less label
366         strcpy(opcArr[29], "JMPEQ "); // Jump equal label
367
368         strcpy(opcArr[30], "NOP "); // No operation
369
370         strcpy(opcArr[31], "LOAD "); // Load
```

```
371         strcpy(opcArr[32], "STORE "); // store
372         strcpy(opcArr[33], "POP "); // POP
373         strcpy(opcArr[34], "PUSH "); // Push
374 
375         strcpy(opcArr[35], "HALT "); // HALT
376 
377         strcpy(opcArr[36], "FIRCOR "); // FIRCOR
378         strcpy(opcArr[37], "FIRCOI "); // FIRCOI
379         strcpy(opcArr[38], "FIRSAR "); // FIRSAR
380         strcpy(opcArr[39], "FIRSAI "); // FIRSAI
381         strcpy(opcArr[40], "FIRCORESET "); // FIRCORESET
382 
383         strcpy(opcArr[41], "JMPR "); // Jump Register
384         strcpy(opcArr[42], "JMPNQR "); // jump not equal to register
385         strcpy(opcArr[43], "JMPLER "); // Jump Less Register
386         strcpy(opcArr[44], "JMPEQR "); // Jump equal Register
387         strcpy(opcArr[45], "JMPPA "); // Jump parity immediate
388         strcpy(opcArr[46], "JMPPAR "); // Jump parity Register
389 
390         strcpy(opcArr[47], "GETFLAG "); // get flags to reg
391         strcpy(opcArr[48], "SETFLAG "); // set flags immediate
392         strcpy(opcArr[49], "PUSHFLAGS "); // push flags
393         strcpy(opcArr[50], "POPFLAGS "); // pop flags
394         strcpy(opcArr[51], "SETFLAGR "); // set flags from register
395 #pragma endregion
396 
397 #pragma region opcArrBin
398         //opcodes written in binary for each corresponding opcode
399         char opcArrBin[100][10];
400         strcpy(opcArrBin[0], "001101"); // ADDI
401         strcpy(opcArrBin[1], "001111"); // SUBI
402         strcpy(opcArrBin[2], "001110"); // ADDCI
403         strcpy(opcArrBin[3], "010000"); // NEGI
404         strcpy(opcArrBin[4], "010001"); // ANDI
405         strcpy(opcArrBin[5], "010011"); // XORI
406         strcpy(opcArrBin[6], "010010"); // ORI
407         strcpy(opcArrBin[7], "010100"); // MULTI
408         strcpy(opcArrBin[8], "010101"); // LSLI
409         strcpy(opcArrBin[9], "010110"); // LSRI
410         strcpy(opcArrBin[10], "010111"); // RASI
411         strcpy(opcArrBin[11], "011100"); // CMPI
412         strcpy(opcArrBin[12], "011101"); // MOVI
413 
```

```
414         strcpy(opcArrBin[13], "000001"); // ADDR
415         strcpy(opcArrBin[14], "000010"); // ADDC
416         strcpy(opcArrBin[15], "000011"); // SUBR
417         strcpy(opcArrBin[16], "000100"); // NEGR
418         strcpy(opcArrBin[17], "000101"); // ANDR
419         strcpy(opcArrBin[18], "000111"); // XORR
420         strcpy(opcArrBin[19], "000110"); // ORR
421         strcpy(opcArrBin[20], "001000"); // MULT
422         strcpy(opcArrBin[21], "001001"); // LSLR
423         strcpy(opcArrBin[22], "001010"); // LSRR
424         strcpy(opcArrBin[23], "001011"); // RASR
425         strcpy(opcArrBin[24], "011010"); // CMP
426         strcpy(opcArrBin[25], "011011"); // MOV
427 
428         strcpy(opcArrBin[26], "100010"); // JMP
429         strcpy(opcArrBin[27], "100101"); // JMPNQ
430         strcpy(opcArrBin[28], "100100"); // JMPLE
431         strcpy(opcArrBin[29], "100011"); // JMPEQ
432 
433         strcpy(opcArrBin[30], "000000"); // NOP
434 
435         strcpy(opcArrBin[31], "011110"); // LOAD
436         strcpy(opcArrBin[32], "011111"); // STORE
437         strcpy(opcArrBin[33], "100000"); // POP
438         strcpy(opcArrBin[34], "100001"); // PUSH
439 
440         strcpy(opcArrBin[35], "100110"); // HALT
441 
442         strcpy(opcArrBin[36], "100111"); // FIRCOR
443         strcpy(opcArrBin[37], "101000"); // FIRCOI
444         strcpy(opcArrBin[38], "101010"); // FIRSAR
445         strcpy(opcArrBin[39], "101011"); // FIRSAI
446         strcpy(opcArrBin[40], "101001"); // FIRCORESET
447 
448         strcpy(opcArrBin[41], "101100"); // JMPR
449         strcpy(opcArrBin[42], "101101"); // JMPNQR
450         strcpy(opcArrBin[43], "101110"); // JMPLER
451         strcpy(opcArrBin[44], "101111"); // JMPEQR
452         strcpy(opcArrBin[45], "110000"); // JMPPA
453         strcpy(opcArrBin[46], "110001"); // JMPPAR
454 
455         strcpy(opcArrBin[47], "110010"); // GETFLAG
456         strcpy(opcArrBin[48], "110011"); // SETFLAG
```

```
457         strcpy(opcArrBin[49], "110100"); // PUSHFLAGS
458         strcpy(opcArrBin[50], "110101"); // POPFLAGS
459         strcpy(opcArrBin[51], "110110"); // POPFLAGS
460 #pragma endregion
461
462         //Compares the temporary input array with an array of possible opcodes to find which one is present.
463         //Since only one opcode is present per line, no position check is needed
464         for (int i = 0; i < 100; i++)
465         {
466             if (strstr(temp, opcArr[i]))
467             {
468                 strcpy(output[outputIndex], opcArrBin[i]);
469                 outputIndex++;
470                 //sets the opcode type depending on what opcode is found in temp
471                 //opcodes are sorted by type in opcArr and opcArrBin
472
473                 if (i <= 12 || i == 37 || i == 39 || i == 48)
474                 {
475                     opcType = IMMEDIATE;
476                 }
477                 if (i > 12 && i <= 25)
478                 {
479                     opcType = REGISTER;
480                 }
481                 if (i == 36 || i == 38 || i == 47 || i == 51)
482                 {
483                     opcType = REGISTER;
484                 }
485                 if (i > 25 && i <= 29)
486                 {
487                     opcType = JUMP;
488                 }
489                 if (i >= 41 && i <= 46)
490                 {
491                     opcType = JUMP;
492                 }
493                 if (i >= 30 && i <= 35)
494                 {
495                     opcType = MEMORY;
496                 }
497                 if (i == 40)
498                 {
499                     opcType = MEMORY;
```

```
500                 }
501                 if (i >= 49 && i <= 50)
502                 {
503                     opcType = MEMORY;
504                 }
505                 break;
506             }
507         }
508 #pragma endregion
509 
510 #pragma region operandFinder
511         //Spaces finder - finds and saves the location of spaces in input.
512         //This is done to find the distance between spaces to find operands
513         //This also allows the program to ignore multiple whitespaces for nicer assembly formatting
514         int spaces[100]; //array to store the location of spaces in temp
515         int spaceIndex = 0; //index where to save space position in spaces
516         int spaceDist; //used to save the distance between
517 
518         memset(&spaces[0], 0, sizeof(spaces)); //resets the spaces array
519 
520         //finds and saves the location of all spaces and newlines since both serve the same purpose
521         for (int i = 0; i < bufferSize; i++)
522         {
523             if (temp[i] == ' ' || temp[i] == '\n')
524             {
525                 spaces[spaceIndex] = i;
526                 spaceIndex++;
527             }
528         }
529 
530         //Operand counter
531         for (int i = 0; i < 100; i++)
532         {
533             if (spaces[i + 1] - spaces[i] > 1) //if spacedistance is more than 1 an operand is preset
534             {
535                 opCount++;
536             }
537         }
538 
539 #pragma region opArr
540         //array that contains all register's names
541         char opArr[32][10];
542         strcpy(opArr[0], "$r0 ");
```

```
543         strcpy(opArr[1], "$r1 ");
544         strcpy(opArr[2], "$r2 ");
545         strcpy(opArr[3], "$r3 ");
546         strcpy(opArr[4], "$r4 ");
547         strcpy(opArr[5], "$r5 ");
548         strcpy(opArr[6], "$r6 ");
549         strcpy(opArr[7], "$r7 ");
550         strcpy(opArr[8], "$r8 ");
551         strcpy(opArr[9], "$r9 ");
552         strcpy(opArr[10], "$r10 ");
553         strcpy(opArr[11], "$r11 ");
554         strcpy(opArr[12], "$r12 ");
555         strcpy(opArr[13], "$r13 ");
556         strcpy(opArr[14], "$r14 ");
557         strcpy(opArr[15], "$r15 ");
558         strcpy(opArr[16], "$r16 ");
559         strcpy(opArr[17], "$r17 ");
560         strcpy(opArr[18], "$r18 ");
561         strcpy(opArr[19], "$r19 ");
562         strcpy(opArr[20], "$r20 ");
563         strcpy(opArr[21], "$r21 ");
564         strcpy(opArr[22], "$r22 ");
565         strcpy(opArr[23], "$r23 ");
566         strcpy(opArr[24], "$r24 ");
567         strcpy(opArr[25], "$r25 ");
568         strcpy(opArr[26], "$r26 ");
569         strcpy(opArr[27], "$r27 ");
570         strcpy(opArr[28], "$r28 ");
571         strcpy(opArr[29], "$zero ");
572         strcpy(opArr[30], "$sp ");
573         strcpy(opArr[31], "$pc ");
574
575 #pragma endregion
576
577 #pragma region opArrBin
578         // registers in binary
579         char opArrBin[32][10];
580         strcpy(opArrBin[0], "ERROR"); // DOES NOT WORK ON CPU BUT DOES EXIST (TECHNICALLY)
581         strcpy(opArrBin[1], "00001"); // R1
582         strcpy(opArrBin[2], "00010"); // R2
583         strcpy(opArrBin[3], "00011"); // R3
584         strcpy(opArrBin[4], "00100"); // R4
585         strcpy(opArrBin[5], "00101"); // R5
```

```
586         strcpy(opArrBin[6], "00110"); // R6
587         strcpy(opArrBin[7], "00111"); // R7
588         strcpy(opArrBin[8], "01000"); // R8
589         strcpy(opArrBin[9], "01001"); // R9
590         strcpy(opArrBin[10], "01010"); // R10
591         strcpy(opArrBin[11], "01011"); // R11
592         strcpy(opArrBin[12], "01100"); // R12
593         strcpy(opArrBin[13], "01101"); // R13
594         strcpy(opArrBin[14], "01110"); // R14
595         strcpy(opArrBin[15], "01111"); // R15
596         strcpy(opArrBin[16], "10000"); // R16
597         strcpy(opArrBin[17], "10001"); // R17
598         strcpy(opArrBin[18], "10010"); // R18
599         strcpy(opArrBin[19], "10011"); // R19
600         strcpy(opArrBin[20], "10100"); // R20
601         strcpy(opArrBin[21], "10101"); // R21
602         strcpy(opArrBin[22], "10110"); // R22
603         strcpy(opArrBin[23], "10111"); // R23
604         strcpy(opArrBin[24], "11000"); // R24
605         strcpy(opArrBin[25], "11001"); // R25
606         strcpy(opArrBin[26], "11010"); // R26
607         strcpy(opArrBin[27], "11011"); // R27
608         strcpy(opArrBin[28], "11100"); // R28
609         strcpy(opArrBin[29], "11101"); // ZERO
610         strcpy(opArrBin[30], "11110"); // SP
611         strcpy(opArrBin[31], "11111"); // PC
612 #pragma endregion
613
614         if (opcType == IMMEDIATE) //immediate opcode type
615         {
616             for (int i = 0; i < opCount; i++)
617             {
618                     spaceDist = spaces[i + 1] - spaces[i];
619                     if (spaceDist > 1) //if spacedistance is greater than one, an operand must be present
620                     {
621                         char opTemp[100]; //temporary array for storing one operand at a time
622
623                         for (int j = 0; j < spaceDist; j++) //saves operand from the current space location +1 to the next
                                space location -1
624                         {
625                             opTemp[j] = temp[spaces[i] + 1 + j];
626                         }
627                         opTemp[spaceDist] = '\0';
```

```
628
629                 //checks what kind of operand is present in the current opTemp
630                 if (strstr(opTemp, "$")) //register operand
631                 {
632                     for (int u = 0; u < 100; u++)
633                     {
634                         if (strstr(opTemp, opArr[u])) //checks which register is contianed in opTemp
635                         {
636                             strcpy(output[outputIndex], opArrBin[u]); //saves the binary of the contianed register in
                                output
637                             outputIndex++;
638                             break;
639                         }
640                     }
641                 }
642                 else if (strstr(opTemp, "#")) //label immediate. Allows for manipulation of labels in immediate
                    opcodes
643                 {
644                     for (int i = 0; i < 100; i++)
645                     {
646                         opTemp[i] = opTemp[i + 1]; //shifts the optemp one space to the left since it begins with a
                            space
647                                                    //since the labelname array does not.
648
649                         if (opTemp[i] == '\0') //string terminator char
650                         {
651                             opTemp[i - 1] = '\0'; //ends the optemp string and stops the shift
652                             break;
653                         }
654
655                     }
656
657                     for (int i = 0; i < 100; i++) //compares the optemp with the labelnames to find out which one is
                        to be printed
658                     {
659                         if (strstr(labelName[i], opTemp)) //if labelname[i] contains optemp
660                         {
661                             int num = labelLine[i]; //saves the labelline to an int
662                             char padString[17]; //creates a temporary string which is needed to pad the value to 16
                                bits,
663                                                 //after it is converted to binary
664                             int zeros = 0; //int for storing the amount of padding zeros
665
```

```
666                     _itoa(num, padString, 2); //saves num as a string in padstring in radix 2 (binary)
667
668                     //checks for the end of padstring and uses this location to find out how many padding
                            zeros are needed.
669                     //Afterwards it creates a temporary char array since the padding has to be done from the
                            right,
670                     //the binary value is then appended onto the temporary string which creates a 16 bit
                            string
671                     //the temporary string is then saved in the original one and saved for output
672                     for (int j = 0; j < 17; j++)
673                     {
674                         if (padString[j] == '\0')
675                         {
676                             zeros = 16 - j;
677                             char padTemp[17];
678                             for (int u = 0; u < zeros; u++)
679                             {
680                                 padTemp[u] = '0';
681                             }
682                             padTemp[zeros] = '\0';
683
684                             strcat(padTemp, padString);
685                             strcpy(padString, padTemp);
686                         }
687
688                     }
689                     strcpy(output[outputIndex], padString);
690                     outputIndex++;
691                     break;
692                 }
693             }
694         }
695
696         else // if previous statements were not true, the immediate value is present
697         {
698             //same immediate handling as previously
699             int num = atoi(opTemp);
700
701             if (num >= 0) //positive
702             {
703                 int zeros = 0;
704                 char padString[17];
705                 _itoa(num, padString, 2);
```

```
706                             for (int j = 0; j < 17; j++)
707                             {
708                                 if (padString[j] == '\0')
709                                 {
710                                     zeros = 16 - j;
711                                     char padTemp[17];
712                                     for (int u = 0; u < zeros; u++)
713                                     {
714                                         padTemp[u] = '0';
715                                     }
716                                     padTemp[zeros] = '\0';
717 
718                                     strcat(padTemp, padString);
719                                     strcpy(output[outputIndex], padTemp);
720                                     outputIndex++;
721                                     break;
722                                 }
723                             }
724                         }
725                         else //negative
726                         {
727                             char padString[33]; //32 bit string since program is 32 bit and negative numbers are converted
                                    to that
728                             _itoa(num, padString, 2);
729                             char padTemp[17];
730                             for (int u = 16; u < 33; u++)
731                             {
732                                 padTemp[u - 16] = padString[u];
733                             }
734                             strcpy(output[outputIndex], padTemp);
735                             outputIndex++;
736                         }
737                     }
738                 }
739             }
740         }
741 
742         if (opcType == REGISTER)
743         {
744             for (int i = 0; i < opCount + 1; i++)
745             {
746                 spaceDist = spaces[i + 1] - spaces[i];
747                 if (spaceDist > 1) //operand is present
```

```
748                 {
749                     //reads operand into temp array
750                     char opTemp[100];
751                     for (int j = 0; j < spaceDist; j++)
752                     {
753                         opTemp[j] = temp[spaces[i] + 1 + j]; //checks from one position to the right of a space and for space
                                distance -1
754                     }
755                     //compares operand with register array and prints found register as binary
756                     for (int u = 0; u < 100; u++)
757                     {
758                         if (strstr(opTemp, opArr[u]))
759                         {
760                             strcpy(output[outputIndex], opArrBin[u]);
761                             outputIndex++;
762                             break;
763                         }
764                     }
765                 }
766             }
767         }
768 
769         if (opcType == MEMORY)
770         {
771             for (int i = 0; i < opCount; i++)
772             {
773                 spaceDist = spaces[i + 1] - spaces[i];
774                 if (spaceDist > 1) //operand must be present
775                 {
776                     //reads operand into temp array
777                     char opTemp[100];
778                     for (int j = 0; j < spaceDist; j++)
779                     {
780                         opTemp[j] = temp[spaces[i] + 1 + j]; //checks from one position to the right of a space and for space
                                distance -1
781                     }
782 
783                     if (i == 1) //if i=1 the operand containing the memory address is present since its written OPC REG [MEM]
784                     {
785                         //Checks for and saves the location of the + for [REG+IMM]
786                         if (strstr(opTemp,"+"))
787                         {
788                             immMem = 1; //sets the immediate memory for correct output
```

```
789                 int plusLoc = 0;
790                 char regTemp[10]; //temp array to store the register part of the operand
791                 for (int j = 0; j < 100; j++)//plusfinder
792                 {
793                     if (opTemp[j] == '+')
794                     {
795                         plusLoc = j; //saves the location of the plus which splits the REG and IMM
796                         break;
797                     }
798                 }
799                 //loads the registry into regtemp to compare it to opArr
800                 for (int j = 0; j < plusLoc; j++)
801                 {
802                     if (j == plusLoc - 1)
803                     {
804                         regTemp[j] = '\0'; //ends the string so string functions can be used
805                     }
806                     else
807                     {
808                         regTemp[j] = opTemp[j + 1];
809                     }
810 
811                 }
812                 strcat(regTemp, " "); //appens a space on the end of regtemp to allow it to compare to opArr
                        properly
813 
814                 for (int u = 0; u < 100; u++)
815                 {
816                     if (strstr(regTemp, opArr[u]))
817                     {
818                         strcpy(output[outputIndex], opArrBin[u]); //copies found registry to output in binary
819                         outputIndex++;
820                         break;
821                     }
822                 }
823 
824 
825                 //loads the immediate part of the operand into a temporary array and outputs it as binary the same
                        as the other immedates
826                 char immTemp[10];
827                 for (int j = plusLoc; j < 100; j++)
828                 {
829                     if (opTemp[j + 1] != ']')
```

```
            {
                immTemp[j - plusLoc] = opTemp[j + 1];
            }
            else
            {
                immTemp[j - plusLoc] = '\0';
                break;
            }
        }
        //converts the immediate value into binary and saves it to output
        int num = atoi(immTemp);
        char padString[17];
        int zeroes = 0;
        _itoa(num, padString, 2);
        for (int j = 0; j < 17; j++)
        {
            if (padString[j] == '\0')
            {
                zeroes = 16 - j;
                char padTemp[17];
                for (int u = 0; u < zeroes; u++)
                {
                    padTemp[u] = '0';
                }
                padTemp[zeroes] = '\0';

                strcat(padTemp, padString);
                strcpy(padString, padTemp);
            }
        }
        strcpy(output[outputIndex], padString);
        outputIndex++;
    }
    //if no [REG+IMM] is given, and $ is present, it must be [REG] and is treated as normal register
        operand
    //except the "[]" has to be ignored
    else if(strstr(opTemp,"$"))
    {
        immMem = 0; //sets the immMem to register only for correct output
        char regTemp[20];
        for (int j = 0; j < 20; j++)
        {
            if (opTemp[j + 1] != ']')
```

```
872                     {
873                         regTemp[j] = opTemp[j + 1];
874                     }
875                     else
876                     {
877                         regTemp[j] = '\0';
878                         strcat(regTemp, " ");
879                         break;
880                     }
881                 }
882                 //compares regtemp with opArr and saves the correct registry in binary to output
883                 for (int u = 0; u < 100; u++)
884                 {
885                     if (strstr(regTemp, opArr[u]))
886                     {
887                         strcpy(output[outputIndex], opArrBin[u]);
888                         outputIndex++;
889                         break;
890                     }
891                 }
892             }
893             else //neither [REG+IMM] nor [REG] so it must be [IMM]
894             {
895                 immMem = 2; //sets immMem for correct output
896                 //loads the immediate value into a temp array and converts it to binary for output like other
                        immediates
897                 char immTemp[10];
898                 for (int j = 1; j < 100; j++)
899                 {
900                     if (opTemp[j] != ']')
901                     {
902                         immTemp[j-1] = opTemp[j];
903                     }
904                     else
905                     {
906                         immTemp[j - 1] = '\0';
907                         break;
908                     }
909                 }
910                 int num = atoi(immTemp);
911                 char padString[17];
912                 int zeroes = 0;
913                 _itoa(num, padString, 2);
```

```
                        for (int j = 0; j < 17; j++)
                        {
                            if (padString[j] == '\0')
                            {
                                zeroes = 16 - j;
                                char padTemp[17];
                                for (int u = 0; u < zeroes; u++)
                                {
                                    padTemp[u] = '0';
                                }
                                padTemp[zeroes] = '\0';

                                strcat(padTemp, padString);
                                strcpy(padString, padTemp);
                                break;
                            }

                        }
                        strcpy(output[outputIndex], padString);
                        outputIndex++;
                        //Exception for this immediate handling is that it has to output register 29 as [REG+IMM],
                        // since register 29 is a zero register which becomes [0+IMM]
                        strcpy(output[outputIndex], opArrBin[29]); //zero reg
                        outputIndex++;
                    }

                }
                else //outputs the REG part of the opcode. Same as earlier register output
                {
                    for (int u = 0; u < 100; u++)
                    {
                        if (strstr(opTemp, opArr[u]))
                        {
                            strcpy(output[outputIndex], opArrBin[u]);
                            outputIndex++;
                            break;
                        }
                    }
                }
            }
        }
    }

```

```
957         if (opcType == JUMP)
958         {
959             for (int i = 0; i < opCount + 1; i++)
960             {
961                 spaceDist = spaces[i + 1] - spaces[i];
962                 if (spaceDist > 1) //if spacedist > 1 an operand must be present
963                 {
964                     //reads operand into temp array
965                     char opTemp[100];
966                     for (int j = 0; j < spaceDist; j++)
967                     {
968                         opTemp[j] = temp[spaces[i] + 1 + j]; //checks from one position to the right of a space and for space
                                distance -1
969                         if (j == spaceDist - 1)
970                         {
971                             opTemp[j + 1] = '\0';
972                         }
973                     }
974                     if (strstr(opTemp, "#")) //label
975                     {
976                         //compares the found operand with the known labelnames to see which is called.
977                         //when the correct labelname is found, the corresponding labeline is saved to output
978                         //as binary, the same as other immediates are handled
979                         regJump = 0; //no reg in jump
980                         for (int u = 0; u < 100; u++)
981                         {
982                             if (strstr(opTemp, labelName[u]))
983                             {
984                                 char padString[17];
985                                 _itoa(labelLine[u], padString, 2);
986                                 int zeros = 0;
987                                 for (int j = 0; j < 17; j++)
988                                 {
989                                     if (padString[j] == '\0')
990                                     {
991                                         zeros = 16 - j;
992                                         char padTemp[17];
993                                         for (int u = 0; u < zeros; u++)
994                                         {
995                                             padTemp[u] = '0';
996                                         }
997                                         padTemp[zeros] = '\0';
998
```

```
                                    strcat(padTemp, padString);
                                    strcpy(padString, padTemp);
                                }
                            }

                            strcpy(output[outputIndex], padString);
                            outputIndex++;
                            break;
                        }
                    }
                }
                else if (strstr(opTemp,"$")) //reg
                {
                    //compares operand with register array and prints found register as binary
                    regJump = 1; //reg in jump
                    for (int u = 0; u < 100; u++)
                    {
                        if (strstr(opTemp, opArr[u]))
                        {
                            strcpy(output[outputIndex], opArrBin[u]);
                            outputIndex++;
                            break;
                        }
                    }
                }
            }
        }
    }

#pragma endregion

#pragma region output
    //outputs the saved output arrays in the required format for each opcode type
    //in total there has to be 32 bits which requires some to be padded with zeros
    //outputs to both the output file, as well as the console window for easier debugging
    if (strstr(output[0], opcArrBin[12])) //Type MOVE immediate
    {
        printf("%s %s %s %s", output[0], output[5], output[2], output[1]);
        fprintf(fpOut, "%s%s%s%s", output[0], output[5], output[2], output[1]);
    }

    if (strstr(output[0], opcArrBin[11])) //Type CMP immediate
    {
```

```
1042            printf("%s %s %s %s", output[0], output[1], output[5], output[2]);
1043            fprintf(fpOut, "%s%s%s%s", output[0], output[1], output[5], output[2]);
1044        }
1045
1046        if (strstr(output[0], opcArrBin[25])) //Type MOVE Registers
1047        {
1048            printf("%s %s %s %s %s %s", output[0], output[1], output[2], output[5], output[5], output[6]);
1049            fprintf(fpOut, "%s%s%s%s%s%s", output[0], output[1], output[2], output[5], output[5], output[6]);
1050        }
1051
1052        if (strstr(output[0], opcArrBin[24])) //Type CMP Registers
1053        {
1054            printf("%s %s %s %s %s %s", output[0], output[1], output[5], output[2], output[5], output[6]);
1055            fprintf(fpOut, "%s%s%s%s%s%s", output[0], output[1], output[5], output[2], output[5], output[6]);
1056        }
1057
1058        if (strstr(output[0], opcArrBin[30]))//NOP
1059        {
1060            printf("%s %s %s %s %s %s", output[0], output[5], output[5], output[5], output[5], output[6]);
1061            fprintf(fpOut, "%s%s%s%s%s%s", output[0], output[5], output[5], output[5], output[5], output[6]);
1062        }
1063
1064        for (int q = 0; q < 11; q++)
1065        {
1066            if (q == 3)
1067            {
1068                q++;
1069            }
1070            if (strstr(output[0], opcArrBin[q])) //Type Immediates
1071            {
1072                printf("%s %s %s %s", output[0], output[1], output[3], output[2]);
1073                fprintf(fpOut, "%s%s%s%s", output[0], output[1], output[3], output[2]);
1074            }
1075        }
1076
1077        if (strstr(output[0],opcArrBin[16]))//NEGR
1078        {
1079            printf("%s %s %s %s %s %s", output[0], output[1], output[2], output[5], output[5], output[6]);
1080            fprintf(fpOut, "%s%s%s%s%s%s", output[0], output[1], output[2], output[5], output[5], output[6]);
1081        }
1082
1083        for (int q = 13; q < 24; q++)
1084        {
```

```
            if (q == 16)
            {
                q++;
            }
            if (strstr(output[0], opcArrBin[q])) //Type Registers
            {
                printf("%s %s %s %s %s %s", output[0], output[1], output[3], output[2], output[5], output[6]);
                fprintf(fpOut, "%s%s%s%s%s%s", output[0], output[1], output[3], output[2], output[5], output[6]);
            }
        }

        if (strstr(output[0], opcArrBin[3]))//NEGI
        {
            printf("%s %s %s %s", output[0], output[5], output[2], output[1]);
            fprintf(fpOut, "%s%s%s%s", output[0], output[5], output[2], output[1]);
        }

        if (opcType == JUMP)
        {
            if (regJump == 0)//label
            {
                printf("%s %s %s %s", output[0], output[5], output[5], output[1]);
                fprintf(fpOut, "%s%s%s%s", output[0], output[5], output[5], output[1]);
            }
            else if (regJump == 1) //reg
            {
                printf("%s %s %s %s %s %s", output[0], output[1], output[5], output[5], output[5], output[6]);
                fprintf(fpOut, "%s%s%s%s%s%s", output[0], output[1], output[5], output[5], output[5], output[6]);
            }
        }
        if (strstr(output[0], opcArrBin[31]))//LOAD
        {
            if (immMem == 1)// reg+imm
            {
                printf("%s %s %s %s", output[0], output[2], output[1], output[3]);
                fprintf(fpOut, "%s%s%s%s", output[0], output[2], output[1], output[3]);
            }
            else if (immMem == 2) //imm
            {
                printf("%s %s %s %s", output[0], output[3], output[1], output[2]);
                fprintf(fpOut, "%s%s%s%s", output[0], output[3], output[1], output[2]);
            }
            else //reg
```

```
1128            {
1129                printf("%s %s %s %s %s %s", output[0], output[2], output[1], output[5], output[5], output[6]);
1130                fprintf(fpOut, "%s%s%s%s%s%s", output[0], output[2], output[1], output[5], output[5], output[6]);
1131            }
1132        }
1133        if (strstr(output[0], opcArrBin[32]))//STORE
1134        {
1135            if (immMem == 1)// reg+imm
1136            {
1137                printf("%s %s %s %s", output[0], output[2], output[1], output[3]);
1138                fprintf(fpOut, "%s%s%s%s", output[0], output[2], output[1], output[3]);
1139            }
1140            else if (immMem == 2) //imm
1141            {
1142                printf("%s %s %s %s", output[0], output[3], output[1], output[2]);
1143                fprintf(fpOut, "%s%s%s%s", output[0], output[3], output[1], output[2]);
1144            }
1145            else //reg
1146            {
1147                printf("%s %s %s %s %s %s", output[0], output[2], output[1], output[5], output[5], output[6]);
1148                fprintf(fpOut, "%s%s%s%s%s%s", output[0], output[2], output[1], output[5], output[5], output[6]);
1149            }
1150        }
1151
1152            if (strstr(output[0], opcArrBin[33])) // POP
1153            {
1154                printf("%s %s %s %s %s %s", output[0], output[5], output[1], output[5], output[5], output[6]);
1155                fprintf(fpOut, "%s%s%s%s%s%s", output[0], output[5], output[1], output[5], output[5], output[6]);
1156            }
1157            if (strstr(output[0], opcArrBin[34])) // Push
1158            {
1159                printf("%s %s %s %s %s %s", output[0], output[5], output[5], output[1], output[5], output[6]);
1160                fprintf(fpOut, "%s%s%s%s%s%s", output[0], output[5], output[5], output[1], output[5], output[6]);
1161            }
1162        if (strstr(output[0],opcArrBin[35]))//Halt
1163        {
1164            printf("%s %s %s %s %s %s", output[0], output[5], output[5], output[5], output[5], output[6]);
1165            fprintf(fpOut, "%s%s%s%s%s%s", output[0], output[5], output[5], output[5], output[5], output[6]);
1166        }
1167        if (strstr(output[0], opcArrBin[36])) //FIRCOR
1168        {
1169            printf("%s %s %s %s %s %s", output[0], output[1], output[5], output[5], output[5], output[6]);
1170            fprintf(fpOut, "%s%s%s%s%s%s", output[0], output[1], output[5], output[5], output[5], output[6]);
```

```
1171            }
1172            if (strstr(output[0], opcArrBin[37])) //FIRCORI
1173            {
1174                printf("%s %s %s %s", output[0], output[5], output[5], output[1]);
1175                fprintf(fpOut, "%s%s%s%s", output[0], output[5], output[5], output[1]);
1176            }
1177            if (strstr(output[0], opcArrBin[38])) //FIRSAR
1178            {
1179                printf("%s %s %s %s %s %s", output[0], output[1], output[2], output[5], output[5], output[6]);
1180                fprintf(fpOut, "%s%s%s%s%s%s", output[0], output[1], output[2], output[5], output[5], output[6]);
1181            }
1182            if (strstr(output[0], opcArrBin[39])) //FIRSAI
1183            {
1184                printf("%s %s %s %s", output[0], output[5], output[2], output[1]);
1185                fprintf(fpOut, "%s%s%s%s", output[0], output[5], output[2], output[1]);
1186            }
1187            if (strstr(output[0], opcArrBin[40]))//NOP
1188            {
1189                printf("%s %s %s %s %s %s", output[0], output[5], output[5], output[5], output[5], output[6]);
1190                fprintf(fpOut, "%s%s%s%s%s%s", output[0], output[5], output[5], output[5], output[5], output[6]);
1191            }
1192            if (strstr(output[0], opcArrBin[47])) //GETFLAG
1193            {
1194                printf("%s %s %s %s %s %s", output[0], output[5], output[1], output[5], output[5], output[6]);
1195                fprintf(fpOut, "%s%s%s%s%s%s", output[0], output[5], output[1], output[5], output[5], output[6]);
1196            }
1197            if (strstr(output[0], opcArrBin[48])) //SETFLAG
1198            {
1199                printf("%s %s %s %s", output[0], output[5], output[5], output[1]);
1200                fprintf(fpOut, "%s%s%s%s", output[0], output[5], output[5], output[1]);
1201            }
1202            if (strstr(output[0], opcArrBin[49]) || strstr(output[0], opcArrBin[50]))//PUSHFLAGS/POPFLAGS
1203            {
1204                printf("%s %s %s %s %s %s", output[0], output[5], output[5], output[5], output[5], output[6]);
1205                fprintf(fpOut, "%s%s%s%s%s%s", output[0], output[5], output[5], output[5], output[5], output[6]);
1206            }
1207            if (strstr(output[0], opcArrBin[51])) //SETFLAGR
1208            {
1209                printf("%s %s %s %s %s %s", output[0], output[1], output[5], output[5], output[5], output[6]);
1210                fprintf(fpOut, "%s%s%s%s%s%s", output[0], output[1], output[5], output[5], output[5], output[6]);
1211            }
1212        #pragma endregion
1213
```

```
#pragma region MIF endLinePrint
        //resets the output variables before next line is read and converted
        outputIndex = 0;
        memset(output, 0, sizeof(output));

        //newline to make sure each output is on seperate lines - mostly for neatness and easier debugging
        printf("\n");

        //if no label is present and the opcType is not 0 (which indicates blank line or comment)
        //the required line endings is printed to the output file.
        //these are required for .mif format
        if (label == false && opcType != 0)
        {
            fprintf(fpOut, ";");
            fprintf(fpOut, "\n");
            lineCount++;
        }
#pragma endregion

    }

    //after the final line is read, the .mif format requires the remaining memory addresses to be populated by zeros
    //followed by an "END;"
    fprintf(fpOut, "[%d..%d] : 00000000000000000000000000000000;", lineCount, memoryDepth - 1);
    fprintf(fpOut, "\nEND;\n");

    //both files are closed
    fclose(fpIn);
    fclose(fpOut);

    //console prints success
    printf("Assembly successful. Press enter to exit...");
    //enter to exit
    getchar();
    return 0;
}
```

Appendix R

PSoC Code

```
/* Project library includes */
#include "project.h"
#include <DMA_DelSig_dma.h>
#include <DMA_I2S_dma.h>
#include <DMA_SAR_dma.h>
#include <ADC_DelSig.h>
#include <SPI_Slave.h>

/* Project Defines */
#define FALSE  0 /* Boolean definitions for integers, added to make the code more readable */
#define TRUE   1
#define DMA_flag 1 /* When this flag is high (1) DMA is enabled and while it is low (0) DMA is disabled */
#define BUFFER_SIZE 16 /* Buffer size of the SPI */
#define STORE_TD_CFG_ONCMPLT 1

void DMA_DelSig_Config(void);
void DMA_I2S_Config(void);
void DMA_SAR_Config(void);
void DMA_SPIS_RX_Config(void);
void DMA_SPIS_TX_Config(void);

/* Variable declarations for DMA_DelSig */
uint8 DMA_DelSig_Chan;
uint8 DMA_DelSig_TD[2];
/* Variable declarations for DMA_I2S */
uint8 DMA_I2S_Chan;
uint8 DMA_I2S_TD[4];
/* Variable declarations for DMA_SAR */
uint8 DMA_SAR_Chan;
```

```
uint8 DMA_SAR_TD [1];
/* Variable declarations for the DMA_SPIS_Rx */
uint8 rxChannel;
uint8 rxTD;
uint16 rxBuffer [BUFFER_SIZE];

/* Variable declarations for the DMA_SPIS_Tx */
uint8 txChannel;
uint8 txTD[4];
uint16 txBuffer [BUFFER_SIZE-1];

/* Declaration of a 16-bit union */
union buffer
{
    uint16 in; /* Input of the union */
    uint8 out[2]; /* Output of the union */
};
union buffer buf; /* Instancing a union-buffer which is used when DMA is disabled */
union buffer I2Sbuffer0, I2Sbuffer1; /* Instancing two union buffers which is for I2S used when DMA is enabled */
uint8 adc_sar_buffer[2] = {1,8}; /* Instances a buffer array consisting of two byes which is to be used for the SAR ADC */

/* This function shuts off all the LEDs */
void LED_ShutOff()
{
    LED1_Write( FALSE );
    LED2_Write( FALSE );
    LED3_Write( FALSE );
    LED4_Write( FALSE );
    LED5_Write( FALSE );
    LED6_Write( FALSE );
    LED7_Write( FALSE );
    LED8_Write( FALSE );
}
/* This function turns on a select number of LEDs depending on the input number */
/* The order at which the LEDs are turned on is predetermined */
void LED_On(uint8 n)
{
    switch(n)
    {
        case 1:
            LED1_Write( TRUE );
            LED2_Write( FALSE );
            LED3_Write( FALSE );
```

```
73              LED4_Write( FALSE );
74              LED5_Write( FALSE );
75              LED6_Write( FALSE );
76              LED7_Write( FALSE );
77              LED8_Write( FALSE );
78              break;
79          case 2:
80              LED1_Write( TRUE );
81              LED2_Write( TRUE );
82              LED3_Write( FALSE );
83              LED4_Write( FALSE );
84              LED5_Write( FALSE );
85              LED6_Write( FALSE );
86              LED7_Write( FALSE );
87              LED8_Write( FALSE );
88              break;
89          case 3:
90              LED1_Write( TRUE );
91              LED2_Write( TRUE );
92              LED3_Write( TRUE );
93              LED4_Write( FALSE );
94              LED5_Write( FALSE );
95              LED6_Write( FALSE );
96              LED7_Write( FALSE );
97              LED8_Write( FALSE );
98              break;
99          case 4:
100             LED1_Write( TRUE );
101             LED2_Write( TRUE );
102             LED3_Write( TRUE );
103             LED4_Write( TRUE );
104             LED5_Write( FALSE );
105             LED6_Write( FALSE );
106             LED7_Write( FALSE );
107             LED8_Write( FALSE );
108             break;
109         case 5:
110             LED1_Write( TRUE );
111             LED2_Write( TRUE );
112             LED3_Write( TRUE );
113             LED4_Write( TRUE );
114             LED5_Write( TRUE );
115             LED6_Write( FALSE );
```

```
116             LED7_Write( FALSE );
117             LED8_Write( FALSE );
118             break;
119         case 6:
120             LED1_Write( TRUE );
121             LED2_Write( TRUE );
122             LED3_Write( TRUE );
123             LED4_Write( TRUE );
124             LED5_Write( TRUE );
125             LED6_Write( TRUE );
126             LED7_Write( FALSE );
127             LED8_Write( FALSE );
128             break;
129         case 7:
130             LED1_Write( TRUE );
131             LED2_Write( TRUE );
132             LED3_Write( TRUE );
133             LED4_Write( TRUE );
134             LED5_Write( TRUE );
135             LED6_Write( TRUE );
136             LED7_Write( TRUE );
137             LED8_Write( FALSE );
138             break;
139         case 8:
140             LED1_Write( TRUE );
141             LED2_Write( TRUE );
142             LED3_Write( TRUE );
143             LED4_Write( TRUE );
144             LED5_Write( TRUE );
145             LED6_Write( TRUE );
146             LED7_Write( TRUE );
147             LED8_Write( TRUE );
148             break;
149         default:
150             LED_ShutOff();
151             break;
152     }
153 }
154
155 char lastInputString[16];
156 /* This is a function which simplifies writing on the LCD into a single line of code, the inputs are:
157  * rowSelect selects which row of the LCD is selected, zero is the first row and one is the second row
158  * columnSelect selects which column of the LCD is used for the first character of the input string
```

```
159  * zero is the first column, one is the second column, etc.
160  * printString is the input string, this is a 16 byte character array.
161  */
162 void LCD_WriteLine(uint8 rowSelect, uint8 columnSelect, char printString[16])
163 {
164     if(printString != lastInputString) /* Checks if the function call has the same input string as the previous function call,
         i.e. if it's from a loop. */
165     {
166         LCD_Position(0,0); /* Sets the position to the first row and first column */
167         LCD_PrintString("                "); /* Clears the entire row of characters */
168         LCD_Position(1,0); /* Sets the position to the second row and the first column */
169         LCD_PrintString("                "); /* Clears the entire row of characters */
170         LCD_Position(rowSelect,columnSelect); /* Selects the input row and column */
171         LCD_PrintString(printString); /* Writes the input string on the LCD */
172 
173         for (uint8 i = 0; i < 15; i++) /* For-loop which runs between zero and 15, which are the 16 input values of the
             character arrays*/
174         {
175              lastInputString[i] = printString[i]; /* Copies the input string into a global variable which is stored for the
                 next use of this functon */
176         }
177     }
178 }
179 
180 uint16 ADC_SAR_ReadValue; /* Variable to store the current read ADC SAR value during its ISR */
181 uint16 ADC_SAR_Output; /* Variable to store ADC SAR average result */
182 #define BUFFSIZE 100 /* Buffer size which determines how many samples there is for each average */
183 uint16 ringBuffer[BUFFSIZE] = {0}; /* Array which stores all the samples */
184 uint8 ringPointer = 0;
185 uint8 fullAverageFlag = 0;
186 /*
187  * Calculates the sum of all the numbers in the input array
188  * and then divides the sum by the input integer thus calculating
189  * the average value of all the samples in the array.
190 */
191 uint16 sampleAverage(uint16 arr[], uint16 sampleAmount)
192 {
193     uint32 totalSamples = 0; /* creates a local variable which holds the total sum */
194     for (uint32 i = 0; i < sampleAmount; i++) /* Creates a for-loop which adds all the samples together and stores it in a
         single sum */
195     {
196          totalSamples += arr[i];
197     }
```

```
198     uint16 averageResult = totalSamples/sampleAmount; /* Calculates the average of the sum and the input integer */
199     return (uint16) averageResult;
200 }
201 /*
202  * Function which takes a 16-bit value and places its in a buffer,
203  * for each function call a pointer to the position of the buffer will be incremented
204  * and then an average value of the whole buffer will be made.
205  * After 100 function calls the output will be an average of 100 samples
206  * and the process loops by resetting the pointer of the buffer position.
207 */
208 uint16 updateSARBuffer(uint16 newSample)
209 {
210     ringBuffer[ringPointer] = newSample; /* Stores the input sample in the buffer at the location which the pointer points at
        */
211     if(ringPointer < BUFFSIZE) /* Checks whether or not the ringPointer should be incremented or reset to zero */
212     {
213         ringPointer = ringPointer + 1; /* Increments the ringPointer so that it is ready for the next function call */
214     }
215     else
216     {
217         ringPointer = 0;
218         fullAverageFlag = 1;
219     }
220     /*ringPointer = ringPointer < BUFFSIZE ? ringPointer + 1 : 0;*/
221     if(fullAverageFlag == 1) /* Checks whether or not the ringBuffer is full, to determine what value to divide the total sum
        with when calculating the average */
222     {
223         return sampleAverage(ringBuffer, BUFFSIZE); /* Returns an average value of the whole ringBuffer */
224     }
225     else
226     {
227         return sampleAverage(ringBuffer, (ringPointer - 1)); /* Returns an average of the ringBuffer up until the current
            amount of samples */
228     }
229
230 };
231
232 /* Interrupt Service Routine (ISR) for the ADC_SAR internal EOC Interrupt */
233 CY_ISR(ADC_SAR_ISR_LOC)
234 {
235     ADC_SAR_ReadValue = ADC_SAR_CountsTo_mVolts(ADC_SAR_GetResult16()); /* Reads a 12-bit sample on the ADC_SAR and stores it
        in the variable */
236     //ADC_SAR_Flag = 1u; /* Sets the flag high */
```

```
}
uint8 flag = 1u;
uint8 whileVariable = 0u;
int main(void)
{
    CyGlobalIntEnable; /* Enable global interrupts. */

    /* Initializes the three DMA configurations */
    DMA_SAR_Config();
    DMA_SPIS_RX_Config();
    DMA_SPIS_TX_Config();

    SPI_Slave_Start(); /* Initializes the SPI Slave */
    SPI_Slave_WriteTxDataZero(0x00); /* Writes zero to the hardware register of the SPI_Slave */

    CyDmaChEnable(rxChannel, STORE_TD_CFG_ONCMPLT); /* Enables the DMA channel of DMA_SPIS_RX */
    CyDmaChEnable(txChannel, STORE_TD_CFG_ONCMPLT); /* Enables the DMA channel of DMA_SPIS_TX */

    /* If DMA is selected for the I2S part of the program, then this code block initializes the two DMA configurations */
    #if DMA_flag
    DMA_DelSig_Config();
    DMA_I2S_Config();
    #endif

    I2S_1_Start(); /* Initializes the I2S */

    ADC_DelSig_Start(); /* Initializes the ADC */
    ADC_DelSig_StartConvert(); /* Starts the ADC conversion of the ADC_DelSig */

    ADC_SAR_Start(); /* Initializes the ADC */
    ADC_SAR_IRQ_StartEx(ADC_SAR_ISR_LOC); /* Initializes the internal End of Conversion (EOC) Interrupt of the ADC_SAR */
    ADC_SAR_StartConvert(); /* Starts the ADC conversion of the ADC_SAR */


    LCD_Start();
    LCD_WriteLine(0,0,"PSoC LP5");
    LCD_WriteLine(1,0,"Setup complete");

    I2S_1_EnableTx(); /* Enables the I2S transmitter */
    for(;;)
    {
        ADC_SAR_Output = updateSARBuffer(ADC_SAR_ReadValue);
        LCD_Position(0,0);
```

```
280         LCD_PrintNumber(ADC_SAR_Output);
281         LCD_PrintString("    ");
282         /* This code block contains the data transfer between ADC and I2S without using DMA */
283         #if !DMA_flag
284         //uint8 whileVariable = 0u; /* flag condition for the while-loop */
285         //uint8 flag = 1u; /* Flag condition deciding which decides which I2S channel is used */
286         while(whileVariable == 0u) /* Starts a while-loop which runs until data has been written to both I2S channel */
287         {
288             if(I2S_1_ReadTxStatus(CH0)&I2S_1_TX_FIFO_0_NOT_FULL) /* Checks if the I2S TX FIFO buffer is not full, i.e. if I2S
                    is ready to send data */
289             {
290                 if(flag == 1) /* Checks if the loop is ready to send data on the first channel */
291                 {
292                     flag = 0; /* Changes the condition, so the first channel is the next one which will be written to */
293                     buf.in = ADC_DelSig_GetResult16(); /* Reads the ADC result as a 16-bit number and stores it in a union-
                            buffer */
294                     I2S_1_WriteByte(buf.out[flag], 0); /* Writes the first 8-bits of the union-buffer to the I2S TX FIFO
                            buffer */
295                 }
296                 else /* flag == 0 */
297                 {
298                     flag = 1; /* Changes the condition, so the second channel is the next one which will be written to */
299                     I2S_1_WriteByte(buf.out[flag], 0); /* Writes the second 8-bits of the union-buffer to the I2S TX FIFO
                            buffer */
300                     whileVariable = 1; /* Changes the while-loop condition so the while-loop is complete */
301                 }
302             }
303         }
304         #endif
305
306         //if(ADC_SAR_Flag != 0u)
307         {
308             if(ADC_SAR_IsEndConversion(ADC_SAR_RETURN_STATUS))
309             {
310                 //ADC_SAR_Output = ADC_SAR_CountsTo_mVolts(ADC_SAR_GetResult16());
311                 if(ADC_SAR_Output >= 0 && ADC_SAR_Output < 100)
312                 {
313                     LED_On(1);
314                 }
315                 else if (ADC_SAR_Output >= 101 && ADC_SAR_Output < 1400)
316                 {
317                     LED_On(2);
318                 }
```

```
                else if (ADC_SAR_Output >= 1401 && ADC_SAR_Output < 2400)
                {
                    LED_On(3);
                }
                else if (ADC_SAR_Output >= 2401 && ADC_SAR_Output < 3500)
                {
                    LED_On(4);
                }
                else if (ADC_SAR_Output >= 3501 && ADC_SAR_Output < 4000)
                {
                    LED_On(5);
                }
                else if (ADC_SAR_Output >= 4001 && ADC_SAR_Output < 4300)
                {
                    LED_On(6);
                }
                else if (ADC_SAR_Output >= 4301 && ADC_SAR_Output < 4500)
                {
                    LED_On(7);
                }
                else if (ADC_SAR_Output >= 4501 && ADC_SAR_Output < 4800)
                {
                    LED_On(8);
                }
                else
                {
                    // Do nothing
                    LED_ShutOff();
                }
            }
            //ADC_SAR_Flag = 0u;
        }
    }
}
/*
 * This DMAs hardware request input is connected to the end of conversion output of the DelSig ADC
 * The DMA is set to transfer two bytes per burst and requires a single hardware request per burst
 * Inside the DMA channel there is two TDs which loops between each other, i.e. TD[0] -> TD[1] -> TD[0] ....
 * Both of the TD transfer a 16-bit sample from the ADC to a each of their own 16-bit union buffer
 * Finally, TD[0] is set as the initial TD and thereafter the DMA channel is enabled.
 */
void DMA_DelSig_Config(void)
{
```

```
    /* Defines for DMA_DelSig */
    #define DMA_DelSig_BYTES_PER_BURST 2
    #define DMA_DelSig_REQUEST_PER_BURST 1
    #define DMA_DelSig_SRC_BASE (CYDEV_PERIPH_BASE)
    #define DMA_DelSig_DST_BASE (CYDEV_SRAM_BASE)

    /* DMA_DelSig Configuration */
    DMA_DelSig_Chan = DMA_DelSig_DmaInitialize(DMA_DelSig_BYTES_PER_BURST, DMA_DelSig_REQUEST_PER_BURST,
                                    HI16(DMA_DelSig_SRC_BASE), HI16(DMA_DelSig_DST_BASE));
    /* Transaction Descriptor (TD) allocation */
    DMA_DelSig_TD[0] = CyDmaTdAllocate();
    DMA_DelSig_TD[1] = CyDmaTdAllocate();

    /* TD configuration settings */
    CyDmaTdSetConfiguration(DMA_DelSig_TD[0], 2, DMA_DelSig_TD[1], CY_DMA_TD_INC_DST_ADR);
    CyDmaTdSetConfiguration(DMA_DelSig_TD[1], 2, DMA_DelSig_TD[0], CY_DMA_TD_INC_DST_ADR);

    /* Set source and destination address of each TD */
    CyDmaTdSetAddress(DMA_DelSig_TD[0], LO16((uint32)ADC_DelSig_DEC_SAMP_PTR), LO16((uint32)&I2Sbuffer0.in));
    CyDmaTdSetAddress(DMA_DelSig_TD[1], LO16((uint32)ADC_DelSig_DEC_SAMP_PTR), LO16((uint32)&I2Sbuffer1.in));

    /* TD initialization */
    CyDmaChSetInitialTd(DMA_DelSig_Chan, DMA_DelSig_TD[0]);

    /* Enable the DMA channel */
    CyDmaChEnable(DMA_DelSig_Chan, 1);
}

/*
 * This DMA has its hardware request input connected to the TX interrupt of the I2S master
 * The DMA is set to transfer a single byte per burst and requires a single hardware request per burst
 * Inside the DMA channel there is four TDs which loop between each other, similar to the DelSig DMA channel.
 * The first TD transfers a single byte from the output of the aformentioned union buffer,
 * thereafter the next TD will transfer the second byte of the same buffer, the last two TDs repeat this finallizing the loop.
 * All of the TDs transfer their single byte to the I2S FIFO-buffer once a FIFO-not-full interrupt has been received.
 * Finally, TD[0] is set as the initial TD and thereafter the DMA channel is enabled.
 */
void DMA_I2S_Config(void)
{
    /* Defines for DMA_I2S */
    #define DMA_I2S_BYTES_PER_BURST 1
    #define DMA_I2S_REQUEST_PER_BURST 1
    #define DMA_I2S_SRC_BASE (CYDEV_SRAM_BASE)
```

```
405     #define DMA_I2S_DST_BASE (CYDEV_PERIPH_BASE)
406
407     /* DMA_I2S Channel configuration */
408     DMA_I2S_Chan = DMA_I2S_DmaInitialize(DMA_I2S_BYTES_PER_BURST, DMA_I2S_REQUEST_PER_BURST,
409                                          HI16(DMA_I2S_SRC_BASE), HI16(DMA_I2S_DST_BASE));
410
411     /* (TD) allocation */
412     DMA_I2S_TD[0] = CyDmaTdAllocate();
413     DMA_I2S_TD[1] = CyDmaTdAllocate();
414     DMA_I2S_TD[2] = CyDmaTdAllocate();
415     DMA_I2S_TD[3] = CyDmaTdAllocate();
416
417     /* TD configuration settings */
418     CyDmaTdSetConfiguration(DMA_I2S_TD[0], 1, DMA_I2S_TD[1], CY_DMA_TD_INC_SRC_ADR);
419     CyDmaTdSetConfiguration(DMA_I2S_TD[1], 1, DMA_I2S_TD[2], CY_DMA_TD_INC_SRC_ADR);
420     CyDmaTdSetConfiguration(DMA_I2S_TD[2], 1, DMA_I2S_TD[3], CY_DMA_TD_INC_SRC_ADR);
421     CyDmaTdSetConfiguration(DMA_I2S_TD[3], 1, DMA_I2S_TD[0], CY_DMA_TD_INC_SRC_ADR);
422
423     /* Set source and destination address of each TD */
424     CyDmaTdSetAddress(DMA_I2S_TD[0], LO16((uint32)&I2Sbuffer0.out[1]), LO16((uint32)I2S_1_TX_CH0_F0_PTR));
425     CyDmaTdSetAddress(DMA_I2S_TD[1], LO16((uint32)&I2Sbuffer0.out[0]), LO16((uint32)I2S_1_TX_CH0_F0_PTR));
426     CyDmaTdSetAddress(DMA_I2S_TD[2], LO16((uint32)&I2Sbuffer1.out[1]), LO16((uint32)I2S_1_TX_CH0_F0_PTR));
427     CyDmaTdSetAddress(DMA_I2S_TD[3], LO16((uint32)&I2Sbuffer1.out[0]), LO16((uint32)I2S_1_TX_CH0_F0_PTR));
428
429     /* TD initialization */
430     CyDmaChSetInitialTd(DMA_I2S_Chan, DMA_I2S_TD[0]);
431
432     /* Enable the DMA channel */
433     CyDmaChEnable(DMA_I2S_Chan, 1);
434 }
435
436 void DMA_SAR_Config()
437 {
438     /* Defines for DMA_1 */
439     #define DMA_1_BYTES_PER_BURST 2
440     #define DMA_1_REQUEST_PER_BURST 1
441     #define DMA_1_SRC_BASE (CYDEV_PERIPH_BASE)
442     #define DMA_1_DST_BASE (CYDEV_SRAM_BASE)
443
444     /* DMA Configuration for DMA_1 */
445     DMA_SAR_Chan = DMA_SAR_DmaInitialize(DMA_1_BYTES_PER_BURST, DMA_1_REQUEST_PER_BURST,
446                                          HI16(DMA_1_SRC_BASE), HI16(DMA_1_DST_BASE));
447
```

```
448     /* (TD) allocation */
449     DMA_SAR_TD[0] = CyDmaTdAllocate();
450 
451     /* TD configuration settings */
452     CyDmaTdSetConfiguration(DMA_SAR_TD[0], 2, DMA_SAR_TD[0], 0);
453 
454     /* Set source and destination address of each TD */
455     CyDmaTdSetAddress(DMA_SAR_TD[0], LO16((uint32)ADC_SAR_SAR_WRK0_PTR), LO16((uint32)adc_sar_buffer));
456 
457     /* TD initialization */
458     CyDmaChSetInitialTd(DMA_SAR_Chan, DMA_SAR_TD[0]);
459 
460     /* Enable the DMA channel */
461     CyDmaChEnable(DMA_SAR_Chan, 1);
462 };
463 
464 void DMA_SPIS_TX_Config()
465 {
466     /* Defines for DMA_TX */
467     #define DMA_SPIS_TX_BYTES_PER_BURST 1
468     #define DMA_SPIS_TX_REQUEST_PER_BURST 1
469     #define DMA_SPIS_TX_SRC_BASE (CYDEV_SRAM_BASE)
470     #define DMA_SPIS_TX_DST_BASE (CYDEV_PERIPH_BASE)
471 
472     /* DMA Configuration for DMA_TX */
473     txChannel = DMA_SPIS_TX_DmaInitialize(DMA_SPIS_TX_BYTES_PER_BURST, DMA_SPIS_TX_REQUEST_PER_BURST,
474                                           HI16(DMA_SPIS_TX_SRC_BASE), HI16(DMA_SPIS_TX_DST_BASE));
475 
476     /* (TD) allocation */
477     txTD[0] = CyDmaTdAllocate();
478     txTD[1] = CyDmaTdAllocate();
479 
480     /* TD configuration settings */
481     CyDmaTdSetConfiguration(txTD[0], 1, txTD[1], 0);
482     CyDmaTdSetConfiguration(txTD[1], 1, txTD[0], 0);
483 
484     /* Set source and destination address of each TD */
485     CyDmaTdSetAddress(txTD[0], LO16((uint32)&adc_sar_buffer[1]), LO16((uint32)SPI_Slave_TXDATA_PTR));
486     CyDmaTdSetAddress(txTD[1], LO16((uint32)&adc_sar_buffer[0]), LO16((uint32)SPI_Slave_TXDATA_PTR));
487 
488     /* TD initialization */
489     CyDmaChSetInitialTd(txChannel, txTD[1]);
490 
```

```
491     /* Enable the DMA channel */
492     CyDmaChEnable(txChannel, 1);
493 };
494 
495 void DMA_SPIS_RX_Config(void)
496 {
497     /* Defines for DMA_SPIS_RX */
498     #define DMA_SPIS_RX_BYTES_PER_BURST 2
499     #define DMA_SPIS_RX_REQUEST_PER_BURST 1
500     #define DMA_SPIS_RX_SRC_BASE (CYDEV_PERIPH_BASE)
501     #define DMA_SPIS_RX_DST_BASE (CYDEV_SRAM_BASE)
502 
503     /* DMA_SPIS_RX Channel configuration */
504     rxChannel = DMA_SPIS_RX_DmaInitialize(DMA_SPIS_RX_BYTES_PER_BURST, DMA_SPIS_RX_REQUEST_PER_BURST,
505                                           HI16(DMA_SPIS_RX_SRC_BASE), HI16(DMA_SPIS_RX_DST_BASE));
506 
507     /* TD allocation */
508     rxTD = CyDmaTdAllocate();
509 
510     /* TD configuration settings */
511     CyDmaTdSetConfiguration(rxTD, BUFFER_SIZE, CY_DMA_DISABLE_TD, TD_INC_DST_ADR);
512 
513     /* Set source and destination address of each TD */
514     CyDmaTdSetAddress(rxTD, LO16((uint32) SPI_Slave_RXDATA_PTR), LO16((uint32) rxBuffer));
515 
516     /* TD initialization */
517     CyDmaChSetInitialTd(rxChannel, rxTD);
518 }
519 
520 /* [] END OF FILE */
```