UNIVERSITY OF HARTFORD

COLLEGE OF ENGINEERING, TECHNOLOGY, AND ARCHITECTURE

Microprocessor System Design, Simulation, and Implementation

By

Brice Vadnais

B.S. Computer Engineering

And

Aristotle Nafpliotis

B.S. Computer Engineering

Presented to the Faculty of the College of Engineering,

University of Hartford

As a partial requirement for the Degree of Bachelors of Science in Engineering

May 14th 2018

Acknowledgements

We would like to thank our Professor and Technical Advisor, Ph.D. Krista Hill, for all of her support and encouragement throughout the entire project. Without her and her previous course lessons nothing that was done during this project would have been possible. In the past year we have both learned so much about the process of designing a project from the ground up and what it means to be an engineer.

We would also like to thank the University of Hartford Engineering Department for providing labs and development boards for the testing that was done throughout the entire project design phase.

Finally, we would like to thank all of the faculty and students who encouraged the development of the project and came to listen in on the final presentation.

Table of Contents

List of Figures

List of Tables

Executive Summary

This report covers extensively the design of a custom 8-bit microprocessor system with a 16-bit memory bus. Initially the project starts off with the explanation of the basics involved in how a processor operates and then gradually reveals more and more to what makes the processor unique in the space of all other embedded processors. The instruction set architecture describes what is possible and capable on the processor. The internal modules carry out this architecture description and through careful description no errors are made and the processor can execute code directly from a custom assembly level language. All source code for the processor is written in VHDL and will be released under an open permissive MIT license for future computer engineers and enthusiasts to make improvements as seen fit. This document represents the work of a full semester on a single project. This device is an excellent tool in learning the possibilities of VHDL microprocessors and can be used for a number of specific targeted use cases. The report also covers the testing methodology that allows for such a project to be created. By the end of the report one should be able to replicate this processor with little to no external help.

Chapter 1: Project Overview

Outline, Background, and Overall Objective

Microprocessors are the brains of all computer systems. They tell the rest of the computer what to do when certain functions are being called or what needs to put into memory for later use. Microprocessors are now used in nearly every electronic device as a way of making these devices "intelligent". Some microprocessors are built as a unit to run a personal computer where their tasks mainly focus on the daily use from a user. Other microprocessors are built smaller to be used as the processor for a cell phone and some are built with the functionality of a simple way of interfacing with a variety of other devices to allow them to work together easily.

The way that microprocessors can and are built are very different. For the complex ones found in the personal computers of most people the design is limited by the manufacturing process. This process allows for smaller and smaller complex chip designs to be possible. However, the design for this microprocessor will not be built with any fancy manufacturing techniques but rather by using an FPGA. An FPGA is a Field Programmable Gate Array and its uses are countless. An FPGA can be used to program and build complex circuits while abstracting a lot of the complex detail. This makes an FPGA a great way to build a complex circuit such as a microprocessor. FPGAs are also limited by the number of gates that they are able to construct on one chip however so this does limit what the most complex design will be.

The objective of this project is to design a microprocessor from the ground up, then simulate it thoroughly using the software available, and finally implement the entire design onto an FPGA. The design portion of the project will primarily be done in a program called Xilinx which is used as a means to simulate and program digital circuits. In addition, some of the design will be done on paper or through simulations in other programs, such as the instruction set, because there needs to be grounds on which all the other digital logic is designed. The implementation is straightforward because after all the circuit is designed in Xilinx, it will be flashed to the FPGA allowing for testing to be performed on the microprocessor.

Impact

The main use for small microprocessors are their ease of use with peripheral devices. This project will aim for the same ease of use while also being easy to configure for a variety of devices. Microprocessors are constantly becoming more useful in the world because of the devices that we design. This is also an extremely useful learning tool for seeing what goes into the entire design of a microprocessor. For the user, it will provide a way to load custom programs onto the microprocessor and carry out a variety of simple tasks. In addition, because of the design of the microprocessor, it will be fairly simple to load the processor onto another FPGA to be used for another application.

Context

The microprocessor that is being deigned will be similar to the Arduino microcontroller. The Arduino can be used in a variety of applications that use peripheral devices while still being simple in implementation. The microprocessor we are designing will have some of this functionality while also being better for implementation by the user. One way that these devices are similar will be the way that the user interacts with a peripheral device, such as a temperature sensor. In contrast, the devices will also be different in the way that the user loads the program, which interacts with the sensor, onto the device. The design of our microprocessor will also serve as a tool for learning the complexity that goes into designing a microprocessor-based system from the ground up.

Detailed Use Case

This project's primary use case is as a versatile general-purpose microprocessor built from the ground up. The primary users are us, as we are the ones to benefit from its design and the process of building the microprocessor. The microprocessor allows us to write programs and load them onto the processor through a bootloader. In addition, this functionality can also be used in conjunction with interfacing with peripherals. The peripheral devices that are able to be used need to use a serial protocol that we have implemented. It is then fairly simple to use the processor in embedded type applications and as a data acquisition tool.

Another user for out microprocessor are engineering teams that want to test and configure their own peripheral devices. In this case all that needs to be done are some simple programs written that will interact over the serial protocol to the engineer's peripheral device. This is beneficial because a microprocessor implemented on an FPGA has a significant amount of expandability and modularity. This means that if the specific device needed more input and output, it could just be added to the microprocessors specifications. Also, if more memory or less memory was needed to perform a certain task the engineers could modify the code on the board to fit their specific needs.

The last users this device targets are computer and electronics hobbyists. Electronic hobbyists would like this system because it provides an easy to use real time system that can interact with the electronic sensors and peripherals that are used in many hobbyists' projects. Computer hobbyists would like the system because of its uniqueness in the space of microcontrollers, microprocessors, and single board computers. This device itself would be fun to work with and modify for a variety of uses or just to see what the maximum potential is of an FPGA microprocessor.

Chapter 2: Previous Work

Some of the classes that are required for the computer engineering degree taught us significantly about the uses and possibilities of microprocessors. In fact, one such class introduced the students to a microprocessor that was also built and implemented onto an FPGA. Learning about this processor made the idea of building a processor in a year, less fantasy and more reasonable. Although there are still limits to what is possible in such a time frame. This microprocessor, for example, is still able to do many simple programs such as multiplying, searching, and storing arrays. Its use for many floating-point operations are basically nonexistent.

Inspiration

The main inspiration for this project comes from the nod4 project in the Computer Architecture Course as well as all the things that were learned in the Microprocessor Applications and Computer Systems Laboratory courses. In addition, as computer engineers, building a custom microprocessor has been a passion project that could only fully be realized in the senior capstone project class due to the complexity and length of the project.

Nod4 is a microprocessor that was developed by Ph.D. Krista Hill. It is a simple yet non-trivial microprocessor intended for teaching the basics of computer architecture[CITATION Kri09 \l 1033]. This processor is 8-bits and computes all data with only 8-bits. The memory system is also 8-bits so the addressable space by the user is 256 8-bit values. The processor is described as being simple yet non-trivial for many different reasons. It is simple because there are enough instructions to do basic functions and learn the beginning of assembly, and also because the 8-bits makes encoding and decoding instructions easier to the user. In addition, it is simple just due to the fact that there are not a lot of registers or complex architecture to deal with from a user's perspective. It is non-trivial for a variety of other reasons such as its many useful addressing modes and beneficial architectural features such as a stack and an indexed register.

This entire project became a huge inspiration to the current project. One reason for this was just because it was one of the best explained introductions to computer architecture. By keeping the architecture in a similar fashion to nod4 it was much easier to understand and expand the use ability of our microprocessors. An example of this is the load-store based architecture. This in itself is something critical to the design of the microprocessor. By using this it only improved what can be done with the processor and now, with a much stronger understanding of computer architecture there are many more designs that could be made for a microprocessor.

In addition to the nod4 project, learning about microprocessor applications and computer systems made the process of learning and completing this project much easier. In microprocessor applications, different computer architectures were learned and the class focused on a specific microprocessor. Using this knowledge only enhanced what would be possible when putting everything together for the final project. In addition, in computer systems, there was a microprocessor that was built and simulated piece by piece to learn about microprocessors and finally it was used with a device to interact with peripheral serial devices. This in and of itself is where the concept for a peripheral interfacing microprocessor system came from. The procedure for simulating the computer systems laboratory processor is similar to what is used for our processor system.

Proof of Concept

Before any components would be built we had to first make sure that the project was feasible. This was done by researching other projects that were similar and could prove that the idea was possible. For example, one of the first places that we looked into was the microprocessor that was built in one of our previous classes. This would give us a nice place to start by looking into the way that processors are designed. Also, this design was implemented onto an FPGA so already at this stage in the project we

were well aware that what we were designing and building was possible because it had been done before.

To start with more research on the project, it was important to test some of the fundamental principles of the microprocessor to ensure the likelihood that it would work and be a functional project. One way in which this was done, was by simulating the instruction set that was designed for the microprocessor. This simulation was done in Python, a high-level programming language, which allowed for highly modular code to be designed that would present the instruction set in a way that would allow us to ensure that all the necessary functionality was there. Also, by designing the instruction set in a high-level language, it was easy to add on new instructions as necessary as well as remove others that were not. This process allowed our instruction set to only include the essentials that would make up the microprocessor's architecture.

The instruction set simulator was designed as follows. All of the instructions were designed as a function, within Python, which took one parameter which was the operand (see fig. 1).

```
def lda_imm(opr):
    global regA,pc
    regA = opr['opr']
    pc +=1
def lda_dir(opr):
    global regA,pc
    regA = mem[opr['opr']]
    pc+=1
def ldb_imm(opr):
    global regB,pc
    regB = opr['opr']
    pc+=1
```

Figure 1. Structure of Instruction Set Simulator

An empty matrix was used to simulate the program memory, which could be read or written to according to the instruction given, and other variable names took the place for each of the necessary registers (see fig. 2).

```
pc = 0
regA = 0
regB = 0

halt = False
x = 0
sp = 0
ccreg = 0

mem_size = 100
mem = [0] * mem_size
```

Figure 2. Structure of Registers and Memory in Simulator

Then all the instructions were put into a dictionary which would allow the program running on the simulated instruction set to parse out the function of the instruction and carry out its intended purpose.

```
#look up table for assembly of program
lut = {'lda': {'imm':lda_imm,'dir':lda_dir,'indx':lda_indx}
       'ldb': {'imm':ldb_imm,'dir':ldb_dir,'indx':ldb_indx},
       'add': add_acc,
       'hlt': hlt,
       'jmp': jmp_imm,
       'sta': {'imm':sta_ref,'dir':sta_ref},
       'stb': {'imm':stb_ref,'dir':stb_ref},
       'adda': {'imm':adda_imm,'dir':adda_dir},
       'addb': {'imm':addb_imm,'dir':addb_dir},
       'ldx' : {'imm':ldx_imm,'dir':ldx_dir}
      }
```

Figure 3. Dictionary Built for Look up of Instruction Functions during Assembly

After all of these core principles were built for the simulator. A basic assembler was written so that assembly could be written in a particular syntax that would allow the simulator to operate as if it were given the individual instructions.

```
#Assembly Language Test
#Multiply Program using Repeated Addition
Start:      lda 0x06
            ldb 0x03
            decb
Mul:        adda 0x06
            decb
            cmp
            jne [Mul]
End:        hlt
```

Figure 4. Basic Assembly Program with Syntax

Instructions from the assembly were parsed and then placed into memory along with any data for the operand and with Python, functions were able to be executed after calling them from a dictionary. This is what made Python programming so effective in simulating the instruction set.

One conclusion we drew after designing the instruction set simulator, is that we would not likely need any branch instructions because we were going to use jumps instead. This would allow for simpler implementation and would allow the instructions themselves to be unique in purpose. In general, the instruction set simulator proved to be an effective 'quick and dirty' approach to a prototype that would benefit the design of our microprocessor.

Another conclusion that was drawn, was the ability of the simulator to become more than just an instruction set simulator. In this case many of the functions of an actual microprocessor could be programmed so that the simulator would behave as such. For example, rather than using the actual

names of the instructions in the assembler, there could be encoding done on each of the functions that would actually represent the correct instruction. Seeing as this was not the goal of the simulator, we only made it to simulate instructions and further the design of the microprocessor.

Chapter 3: Engineering Specifications and Design Constraints

The specifications are what set microprocessors apart from each other. Each processor is designed and built for a specific use and our use case is as a versatile general-purpose microprocessor. The design for this microprocessor is done from the ground up. This means that all the critical components for the processor will be designed and implemented by us. Using the detailed use case as well as the things learned in the proof of concept instruction set simulator, it is possible to choose the correct specifications that the microprocessor system should adhere to.

Fundamental Principles

In any microprocessor system there are three main components that make the processor a full system. A controller, a memory system, and a data path are the primary components that are used in our microprocessor. The controller is what sends all the necessary control signals to the memory system and data path in order for certain operations to be performed. The data path is where the movement of data happens within the processor and also where all the computers registers and data is located. The memory system contains all of the instructions that are supposed to be used as well as the data that is being manipulated during operation. A simple block diagram of this computer system is shown below (see fig. 5).

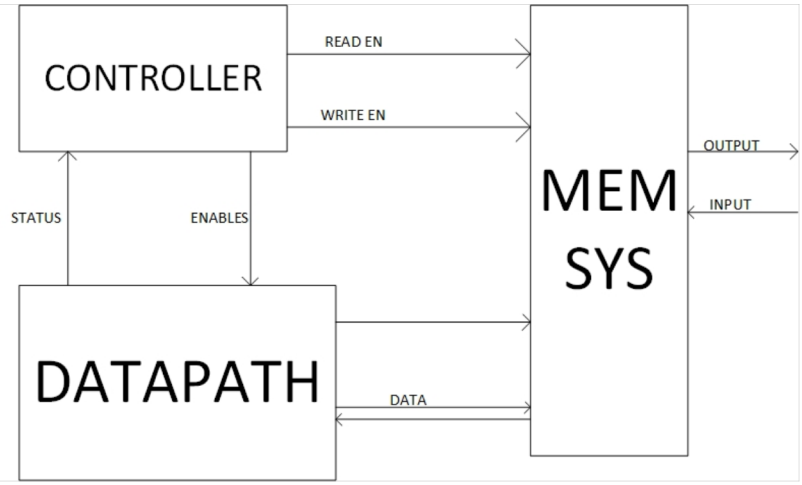


Figure 5. Basic Computer System Overview Block Diagram

These principles were first originally defined by John von Neuman. He described an architecture that had a control unit, an arithmetic logic unit, a memory unit, and input output[CITATION Neu45 \l 1033]. These are all modules that are contained within the processor and the design fully follows this Von

Neuman architecture. Below is an image to show the basic idea and how similar it is to our processors architecture.

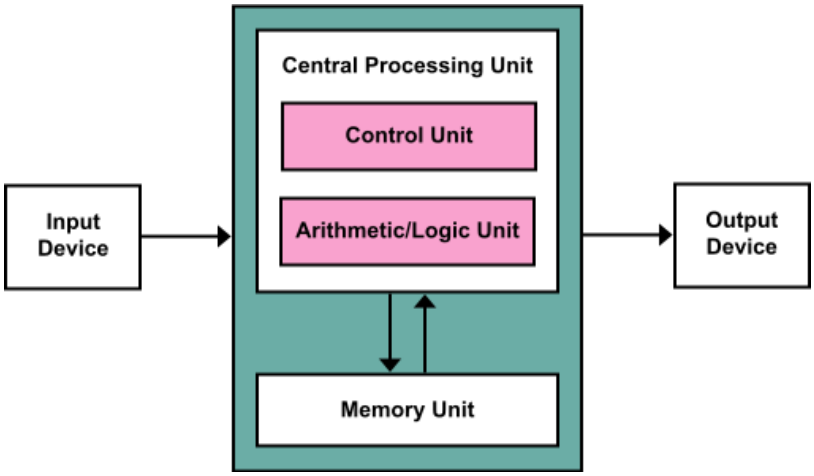


Figure 6. Von Neuman Computer Architecture Block Diagram

Architectural Decisions

Architectural decisions are the design choices made for our project that will impact the way it operates compared to other processor designs. The critical architectural decisions are shown below along with their impact to the overall design of the microprocessor system.

8-bit Accumulator-Based Microprocessor

Using an accumulator-based microprocessor has a big effect on what the processor is capable of and so it is a significant architectural decision. The microprocessor uses 8 bits for each of its two general purpose accumulators and most of the data registers within the data path. This makes it a much more capable processor than that of a 4-bit processor. To our project, this means that the processor has the ability to do more. This also means that the processor has a more complex design. This complex design has the potential to use more space for implementation. Another effect of using an 8-bit accumulator design is the ease of use for some designated processes. The accumulators take the decision of deciding what register to use out of the programmer's hands and instead just forces the operators to all perform on the accumulator register itself. In this circumstance it is easier to use an 8-bit processor for some use cases, rather than a complex microcontroller or microprocessor.

Load/Store Architecture

A load store architecture is pretty self-explanatory to what it actually means with respect to computer architecture. Basically, the load is when data is being written to a register within the data path and store is when the data is being written to somewhere else, such as the memory system. This is an architectural decision because there are many other ways of moving data around within a processor, such as moves. The reason that load and store were chosen is primarily because of our familiarity with the architecture and also because it provides a simple way to interact with the two general purpose accumulators.

16-bit Memory System

The decision was made to use 16-bits of addressable memory for a couple of reasons. One of these reasons is simply that more addressable space leaves more room for programs to run. Also, if the memory system only used 8-bits it would not be able to store the more complex programs that would need to be run by the processor such as the bootloader. Another reason for using 16-bits is increasing the capability of what can be done with the processor. At this scale there is plenty of room for all of the memory system but also room for many peripheral devices. This also becomes expandable as necessary for the application being done. This decision to use 16-bits make the design and architecture much more complex but at the tradeoff of having a much more useful processor and one that satisfies the use case.

Addressing Modes

Every microprocessor has addressing modes that define the way a certain operation will interact with its operand. In this microprocessor there are 5 distinct addressing modes, inherent, immediate, direct, indexed, and stack. These were chosen as an architectural decision because it allows for all the basic needs of a microprocessor to be completed in operation. Inherent addressing means that there is no operand and the operation can be performed with just the operation code. Immediate addressing is when the data is immediately after the operation code. This data is what is used to perform the operation. Direct addressing has the effective address of where the data is to be used for the operation. Indexed addressing contains an offset in the operand that is used to find the effective address of the data that needs to be used to perform an operation. Finally, the stack is used to store values to and from a point in memory.

Jumps Only

There are multiple ways in which microprocessors change the location of where the program counter is pointing. One of these ways is by using branching which involves an offset to the program counter to point it to another spot. Another way is by jumping, this actually replaces the program counter with a new location to point the program counter. It was decided that it would make sense to just use jumps only instead of both jumps and branches because it would simplify the instruction set as well as the complexity of the processor without introducing any redundancy. These jumps would also be useful in setting up the subroutines.

Subroutines

Subroutines provide a way of doing a certain small program within a bigger program. By implementing subroutines into the design, the complexity of the design increases. Subroutines are something that are very useful when programming so the inclusion of them into the design seemed to make a lot of sense. The subroutine return value for the address was stored at the same place in memory so that the subroutine would always work.

Bootloader

One of the impacts of using a bootloader is that it is a more efficient way of programming the microprocessor. The bootloader is used for on the fly changes to the program that is run by the processor over some sort of communication port. This affects the design of the processor because in order to use a bootloader it is necessary to develop the necessary device that allows the communication between an external computer and the processor. For FPGA's it is possible to program the board when it is being designed but it requires that the entire board will need to be flashed each time with all of the

programming information first and each time a new program is used. Allowing a bootloader will significantly speed up this process and allow for a more dynamic programming environment for the user. This makes the use case much more practical for each program to be done by the user on another computer. The downside to using a bootloader is that is a more complex design and also that it is possible for loaded programs to be corrupt if not designed properly. Lastly if no program is sent or there is no program to start, then the processor will lose a lot of functionality by waiting.

Implementing on an FPGA

Implementing the entire circuit onto the FPGA is an architectural decision because it limits what is possible with the microprocessor but also improves the expandability and speed of development. FPGAs prove to be a great way of implementing complex specific integrated circuits because of their ability to have a lot of input and output. In addition, it is impossible to implement an integrated circuit with a significantly cheaper cost than an actual manufactured integrated circuit. This makes it the best choice for designing our custom microprocessor on.

Constraints

When designing a microprocessor there are many constraints that need to be taken into consideration in order for the microprocessor to work per accordingly with the use case. Below list all the possible constraints of the project and what their solution was.

Language for Hardware Modules

One constraint when designing digital logic for a FPGA is choosing which hardware description language to use. The primary options are VHDL or Verilog with support from the largest manufacturers of FPGAs, Altera/Intel and Xilinx. VHDL was chosen because of our familiarity with it, however any hardware description language could have been used in addition to some new niche languages that support translation to VHDL fairly simply. Also, this constraint was cemented in place once the FPGA device target was chosen to be from the Xilinx Spartan Family. These devices in particular are only supported on the legacy software known as ISE and ISE does not support Verilog.

Number of Gates on FPGA

All FPGAs have a limited amount of logic elements that can be used for digital circuit design. Even within the same family of a FPGA chip there are many different package formats that have far more or far less logic elements. The reason for this disparity in packages is that for some applications it is not necessary to have an enormous amount of logic devices. These chips typically will be cheaper. In addition, some large scale specific integrated circuits might need a lot of logic elements and for these designs a larger chip package can be chose to cover the need for that specific applications. These chips end up costing more but still have the same functionality and ease of use as the smaller chips. The way that this was decided for this particular project was first by using a family of chips that were familiar, such as the Spartan 6, and then through many revisions, the final chip was chosen by synthesizing on other packages to reach optimal efficiency.

Mathematic Operations

One decision that needed to be made when constructing this processor is the inclusion or exclusion of certain mathematical operators. This becomes a constraint because it will limit the potentially functionality of the processor but also the complexity of the design and code. Some things that could

have been included would be a multiply and divide instruction. These would have improved the capability of the processor but would have ended up making the design more complex as well as then needing a way to store floating point numbers. In order to keep the processor to a reasonable size and complexity these instructions were ignored due to the fact that they could be remade by using repeated addition or repeated subtraction to calculate in some specific application.

Addressable Data Size and Memory to Hold Complex Programs

The constraint for the data bus's side was determined by seeing what a useful programs size might be. In the case of this microprocessor system, it makes much more sense to allow for a 16-bit memory bus so that much more data could be addressed and used for programming. One way this was determined was by writing simple and complex programs in the instruction set simulator. By using this, the length of the programs could be seen and for something as complex as a bootloader it was obvious that much more data would need to be stored and used rather than something as small as 8-bits. One thing that was kept the same however was the size of the data itself on the bus. This data was all kept to 8-bits to make the speed much faster when transferring data as well as when calculations were being performed.

Power Use and Heat

The power use for the system is something that is a constraint but not something that is used to limit the design of the processor. In particular, the power use would be that from a USB so at most 5V at 1A which would just be 5W. This is similar to many small embedded processors and microcomputers. Heat is something that doesn't necessarily need to be considered into the design either because with the low enough power use of the processor there shouldn't be too much heat produced unless the speed of the processor was increased.

Budget and Time

In any project there are budget and time constraints. The specific constraint for this project is time because there is one full semester to accomplish the use case that was declared for the project. With more time there is definitely more that can be done with the actual project and a lot of improvements can be done. The budget for this project is a constraint to some regard, because without a sponsor, any parts for the project comes out of our own pockets.

Chapter 4: Professional Standards

When designing something, there are standards that you need to follow based on principles of openness, balance, consensus, and due process. These standards can be for a number of things, like technical, safety, regulatory, and societal and market needs. A lot of the standards are provided by Institute of Electrical and Electronics Engineers (IEEE) for our project.

JTAG Standard

A standard that needed to be adhered to in our project are the Joint Test Action Group (JTAG) for Boundary Scan Instructions. The JTAG interface is something that is used in order to load the FPGA board with the code to program it to make it our microprocessor. It is not necessary in the end product but more rather for debugging purposes as well as quick updates to the system design.[CITATION IEE01 \l 1033]

VHDL Standard

In order to program our microprocessor, we need to choose a language. The language we were taught and most familiar with was VHSIC (Very High Speed Integrated Circuit) Hardware Description Language (VHDL). VHDL is the programming language that can be used to implement hardware onto an FPGA. It is a very versatile language and has a lot of features to make programming the board more accessible and an easier process. Specifically, we are using VHDL-93 because that is what our version of ISE Design Suite was compatible with. It is also used for lower end processors mainly for testing and checking and reading signals. [CITATION IEE02 \l 1033]

Peripheral Standard

The IEEE Standards Working Group 1284 is a group of standards that relates to parallel communication specifically for connectivity needs whether it be over PS/2, USB, or via a network connection. One of the standards included in this group is the standard for Bidirectional Parallel Peripheral Interface for Personal Computers. This is what is going to be used in order to communicate with our microprocessor.[CITATION Don18 \l 1033]

SREC Standard

Another standard needed to be followed was the SREC file format type. This standard for SREC file format configures binary information in ASCII hex text form. It is primarily used for programming flash memory in microcontrollers and logic devices. The typical application uses an assembler and converts a program's source code into machine code and outputs the hex file. The file is then imported to put the machine code into non-volatile memory or the target system for loading and execution. The s19 record will be less than or equal to seventy-eight bytes in length. The format for the s-record is 2 bits for type, 2 bits for count, 4, or 9 bits for the address, up to 64 bits for the data, and 2 bits for the checksum. The type field describe the type of record. The count field displays the count of remaining character pairs in the record when it is paired with as a hexadecimal value. The address field displays the address at which the data field is to be loaded into memory. The length of the field depends on the number of bytes necessary to hold the address. The data field represents the memory loadable data. The checksum field display the least significant byte of the ones compliment of the sum of the byte values. These values are represented by the pairs of characters that make up the count, the address, and the data fields. Specifically, our project utilizes the s19 file format because it has sixteen-bit addresses. [CITATION Mot92 \l 1033][CITATION Uni \l 1033]

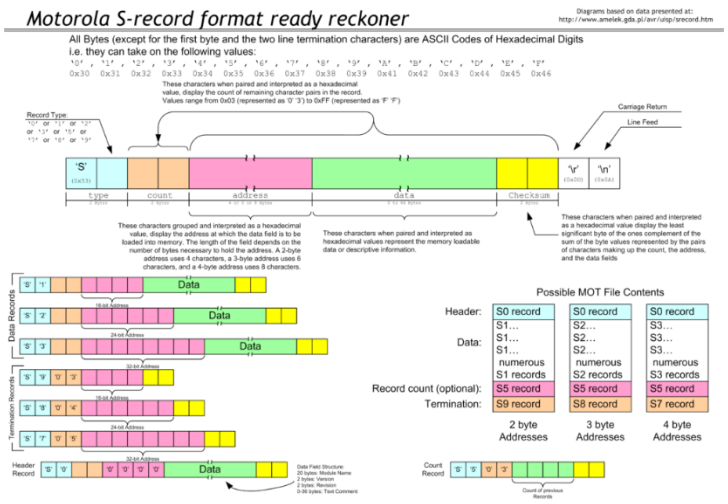


Figure 7. S-Record Format Information[CITATION htt \l 1033]

Chapter 5: Project Plan

In order to get the project started, we had created a projected timeline of what needed to be done in order for this project to be completed. This timeline started off at the beginning of the spring semester. With that in place, work was also expected to be started over winter break which was about one-month long. Once winter break ended, it was time to follow this timeline in order to have this project completed on time.

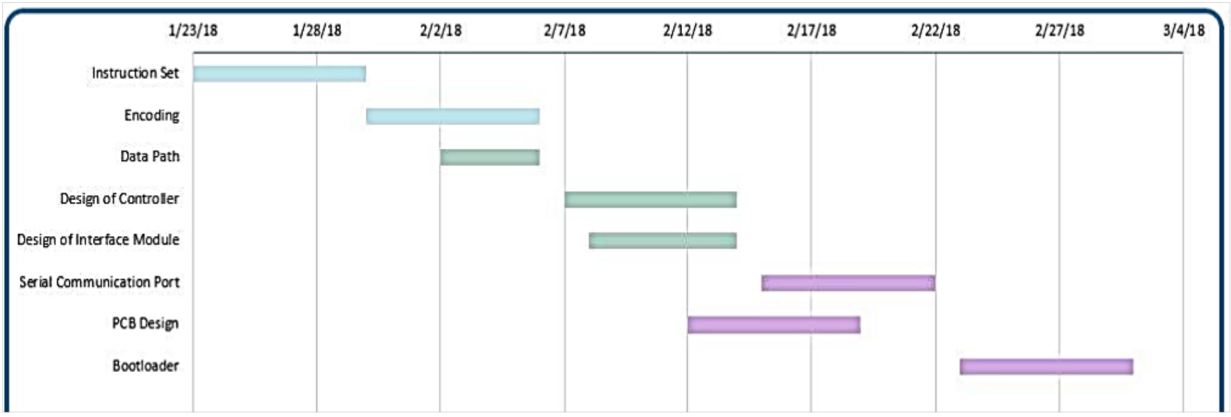


Figure 8. Initial Gantt Chart Project Plan

The first thing on the timeline to complete was the instruction set architecture. By the completion of the instruction set architecture. The goal at the time was to finish the instruction set architecture by only including the necessary instructions for the controller to be started. This was one of the discoveries made because of many duplicate or unnecessary instructions that were removed. As part of the instruction set architecture, the encoding was done for all the instructions. The encoding was crucial to getting the controller started as that was a lengthy process. At that point in time, our project was currently behind by one week. In order to get back on track, when it came to the design of the

controller, as long the design did not last more than 3 weeks, there would still be more than enough time to work on everything necessary for the project.

After the completion of the instruction set architecture and encoding, work on the assembler began, as well as the VHDL code for the multiplexers of the data path. The data path was still being worked on to iron out any uncertainties that may arise during the testing and simulation. The goal of that week in correlation with our projected timeline, was to work on the assembler at the same as the data path. The assembler began to create s19 files from the assembly code. With these components being completed, the design of the controller would be smooth and easy. Something needed to be figured out during this week was how sixteen-bit addresses were going to be handled. This led to certain options and decisions like having two clock cycles per bus cycle to avoid any current and potential errors. This is the critical point in our timeline because we need to be one hundred percent confident in the data path so the controller design will be smooth. This will help us keep on track with our timeline and ensure that finish all the remaining design elements. The current plan was to create the entire data path schematic in Xilinx during this next week. Test benches were then being written for a variety of instructions. This week is where more thought was being put towards the controller and its design using state diagrams. An issue that arose was handling sixteen-bit addresses with the ALU as well as the data out multiplexer. We managed to solve both of these problems in a timely fashion so it did not set us back. The project at this point is making progress and moving forward. In the next week, the controller and memory system should be finished to ensure enough time can be left for simulations and testing.

Next on the projected timeline was the design of the interface module, then serial communication port, and PCB design. In order for us to have a working PCB design it would need to be completed before spring break for it to be shipped in time. This was because we would need to test it and check for any issues in order for us to get another one. In the end we were not able to get working PCB design due to the fact that it was not critical to the design and functionality of the project because a breadboard can still be used in significantly less time. Instead the time went to making final touches on the data path and thinking of the controller in terms of control signals from the data path. Work is also being on the simulations of the data path to ensure that instructions are operating as expected. At this point in time, the project is behind in terms of the projected timeline but moving along at a fair rate. It is definitely clear at this point that more work needs to be done at a faster rate each week to ensure we have a fully operational design.

In this week the final touches were put on the data path and memory system. Currently the work is being done on the controller. This includes the design of the state diagrams as well as VHDL code. The original design of the controller was going to be in microcode but seeing as we know how well the instructions operate, it was simpler to design the whole controller was one big state machine. The controller was then being tested to interact with the memory system and data path. The first thing that was done was the reset to ensure that the processor correctly reads the program start address from the last two bytes in ROM. The controller was then finalized and all instructions were tested. This was done by putting the respective opcodes and operands into ROM to ensure that their operation was correct as expected. At this point in the semester there is not much time left to keep working on this project but everything is going smoothly and assuming that there are minimal problems with the design elements moving forward, such as the bootloader and serial communications, the project will be completed on time.

The assembler is currently being finalized to work with the bootloader. The bootloader will be a modified version of the Boots program but written to operate on this microprocessor. Details are being worked out to have successful implementation of serial communication as well. These are the last few components needed to be finished in the projected timeline. In the next week, the bootloader and assembler were finished up to the point of serial communications. There was errors and issues when trying to implement the BxCom VHDL component code within our memory system. A couple weeks went by which set us behind in order to get a working implementation of our microprocessor. At that point, not enough time was put into getting the serial communication port completed on time by the time that it was to interface it with peripheral devices. We reached the time of the presentation where a simulation was only to be shown of the project.

In order to implement this project, a budget was made to project the cost of our project. Originally, the projected cost of our project was zero dollars. The reason for that was that the University of Hartford already had a lot of the tools and programs necessary for this project to be completed. The cost of the PCB design was expected but because we decided that time was an issue, we did not have to make that purchase. The other expected cost was based on whether or not we wanted to buy our own FPGA board. On average, the cost comes out to $140. Even then, it is not a necessary purchase, but one we made for the reason that there was more space on the board for our microprocessor program.

Chapter 6: Detailed Design

In designing a microprocessor there are many design elements that make up the entire project. Listed below are all the design elements that were implemented on the FPGA microprocessor.

Design Elements

Table 1. Design Elements (not simulation elements)

Instruction Set Architecture	Encoding Scheme
Data Path Design	Memory System Design
Controller Design	Serial Communication Device Design
Assembler	Bootloader
Test Assembly Programs	Implementation on Spartan 6

Instruction Set Architecture

In order to design a proper instruction, set the architectural decisions were used to form an understanding of all the registers that were allowed to be used by the user. By looking at the available registers, it is fairly simple to make the instructions that will correspond to those registers accordingly. In addition, the architectural decisions are what forms the actual instructions to be included in the set. The registers that can be used by the user are listed below.

Table 2. Registers Seen by User

Register	***Description***
A	General Purpose 8-bit Accumulator
B	General Purpose 8-bit Accumulator
CCR	Code Condition Register – Holds Statuses of Flags: Zero, Carry, and Negative
X	16-bit Index Register – Contains Value to be used for offsetting in memory
SP	16-bit Stack Pointer Register – Contains Value of Current Stack Memory Location
PC	16-bit Program Counter – Contains value of the location in memory that the program is at

With the knowledge of all these registers and what they represented it was possible to start deciding what instructions would be included to the actual instruction set. This was also determined by using the instruction set simulator as a guide and tool in the design. The instruction set was also designed using the known addressing modes such as, inherent, immediate, direct, indexed, and stack. The final thing that was taken into consideration was the fact that no branch type instructions were going to be used and instead all types of branching would just be replaced with jumps. This decision was made to simplify the actual instruction set and limit redundancy in the actual microprocessor.

To start off with the design the first instructions to be included were the load and store instructions. These were first written to have immediate data and then would have another instruction that could cover the use case for the other addressing modes. After that more instructions were added such as addition and subtraction of the accumulators. Not only with immediate and direct data but also some inherent instructions that would add the two accumulators together or increment or decrement them as necessary. The other instructions that were included were based off of test programs that were programmed using the instruction set simulator.

The instruction set simulator was extremely useful in learning about what was necessary to include in the processor, as well as what was necessary to exclude. The test programs that were written typically would include basic operations that would be expected to be carried out by the processor to carry out the use case. Then these instructions were added to the list of the actual instruction set. It was determined to be complete once all the basic test programs ran and there were no other complications to it. The set was also slimmed down from any excess or unnecessary instructions just to make sure that the implementation process would go as smoothly as possible.

Listed in Appendix A are all the instructions that were implemented into the microprocessor. In total there are approximately 60 instructions. This makes the instruction set simple enough to use and understand, while complex enough to do especially cool things with the microprocessor, such as talking to peripheral devices.

Encoding Scheme

Encoding is the process of taking the assembly level language instructions and converting them to something that the processor can read and understand and then execute. The encoding for this particular microprocessor was chosen to be fairly arbitrarily. This means that the encoding scheme is not necessarily something that can be figured out by hand but rather something that corresponds to a pattern relative to the design of the microprocessor. The encoding is specifically designed in a way that it is easy to see what addressing mode the instruction mode belongs to but not what it is actually doing unless looking it up in the datasheet.

The encoding scheme is as follows. For the first 4-bits, a value is chosen that is consistent among all the same instructions within an addressing mode. For example, this means that all the instructions that correspond to something such as inherent addressing will start with a 0. That zero of course refers to a hexadecimal value because this is how all the values would be read from the microprocessor and also how data would be encoded to from the assembly files. In table # all the addressing modes are listed were there associated encoding scheme for the first 4-bits of the encoded value.

Table 3. Encoding of First 4-bits and Associated Addressing Mode

Encoding (First 4-bits)	Addressing Mode
0	Inherent
1	Immediate
2	Direct
3	Stack
4	Indexed
5	Jump

The next 4-bits of the encoding scheme were the individual instruction within the specific addressing mode that was being covered. This leaves only 16 possible instructions for each addressing mode. Although this doesn't leave a lot of room for many more instructions per addressing mode, it covers all the possible instructions that were covered in this instruction set. If more instructions need to be added there is plenty of room in 8-bits to encode more instructions for 8-bits there are 256 possible instructions that could be included in the instruction set. The encoding for each individual instruction is shown in Appendix A in both binary and hexadecimal.

Data Path Design

The data path is one of the most important design elements within the microprocessor. It defines the way that data moves around within the actual processor and where instructions will be allowed to operate at any given point. The data path also contains the arithmetic logic unit. This device is what allows the mathematical operations of the processor to be performed as well as any logical operators between its inputs.

The initial design of the data path was created by placing all the possible registers that were to be used onto a diagram. This diagram was then connected up as logically as possible without any code being written. After an initial revision was made there were instruction traces written for the instructions to ensure that they would be able to operate. If the instruction was not able to operate correctly then the design was iterated and revised to accommodate new changes.

Initial revisions for the data path can be seen in Appendix B and C. These revisions were far off from what the actual design would end up being, however they give a good example of where the data paths design started and the methodology for improving it. Many of the core components were there but the design itself needed improvement before it would be able to cover all the instructions necessary. With the third revision of the data path, that can be seen in Appendix D, the actual design was starting to shape up to be usable for most if not all instructions. To simplify the actual look of the data path a block diagram is shown below.

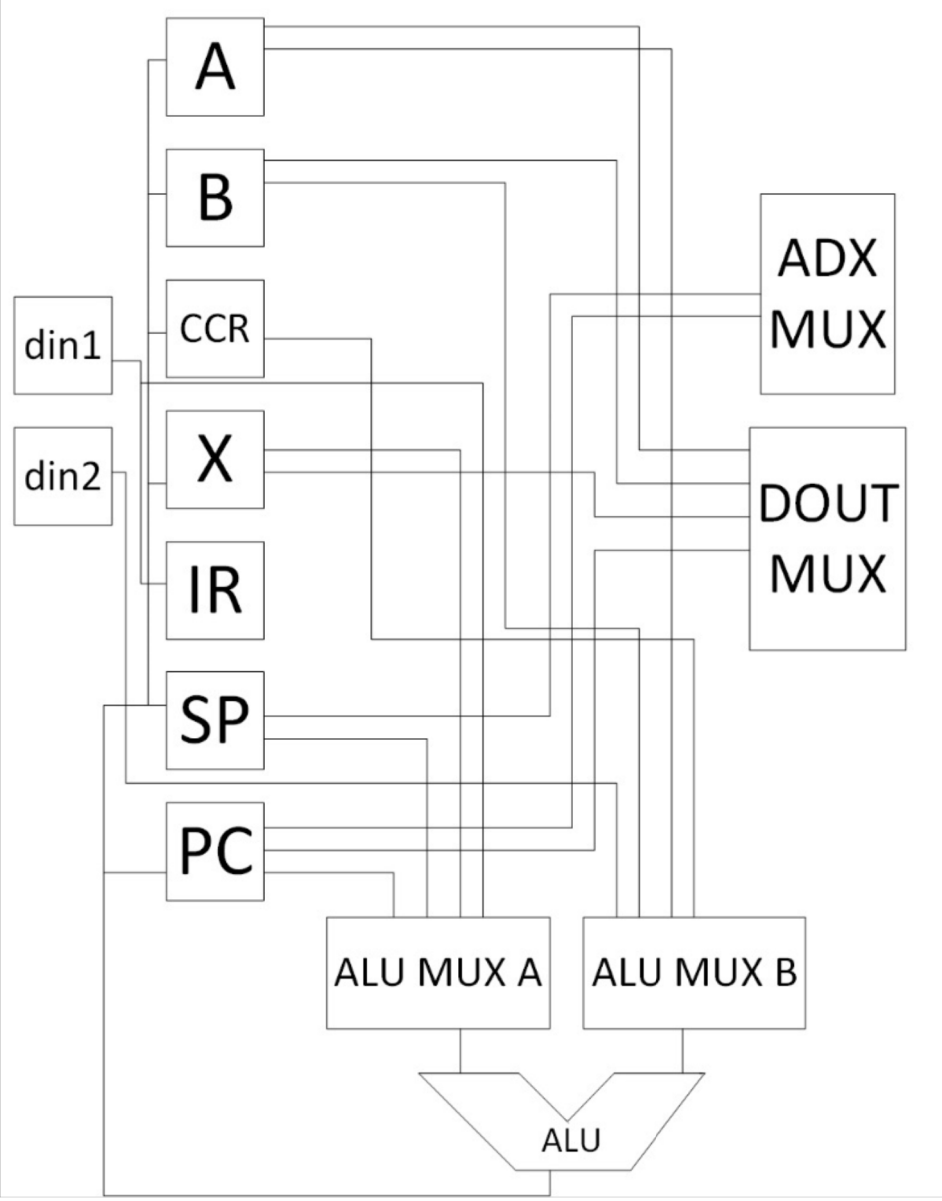


Figure 9. Data Path Block Diagram

Each one of the blocks in the block diagram were then converted into the corresponding VHDL code and connected together to form an actual data path that could be used for simulation and testing. In the figure below the entire final schematic for the data path and in the sections below, the code will be shown and explained to demonstrate the design of the data path.

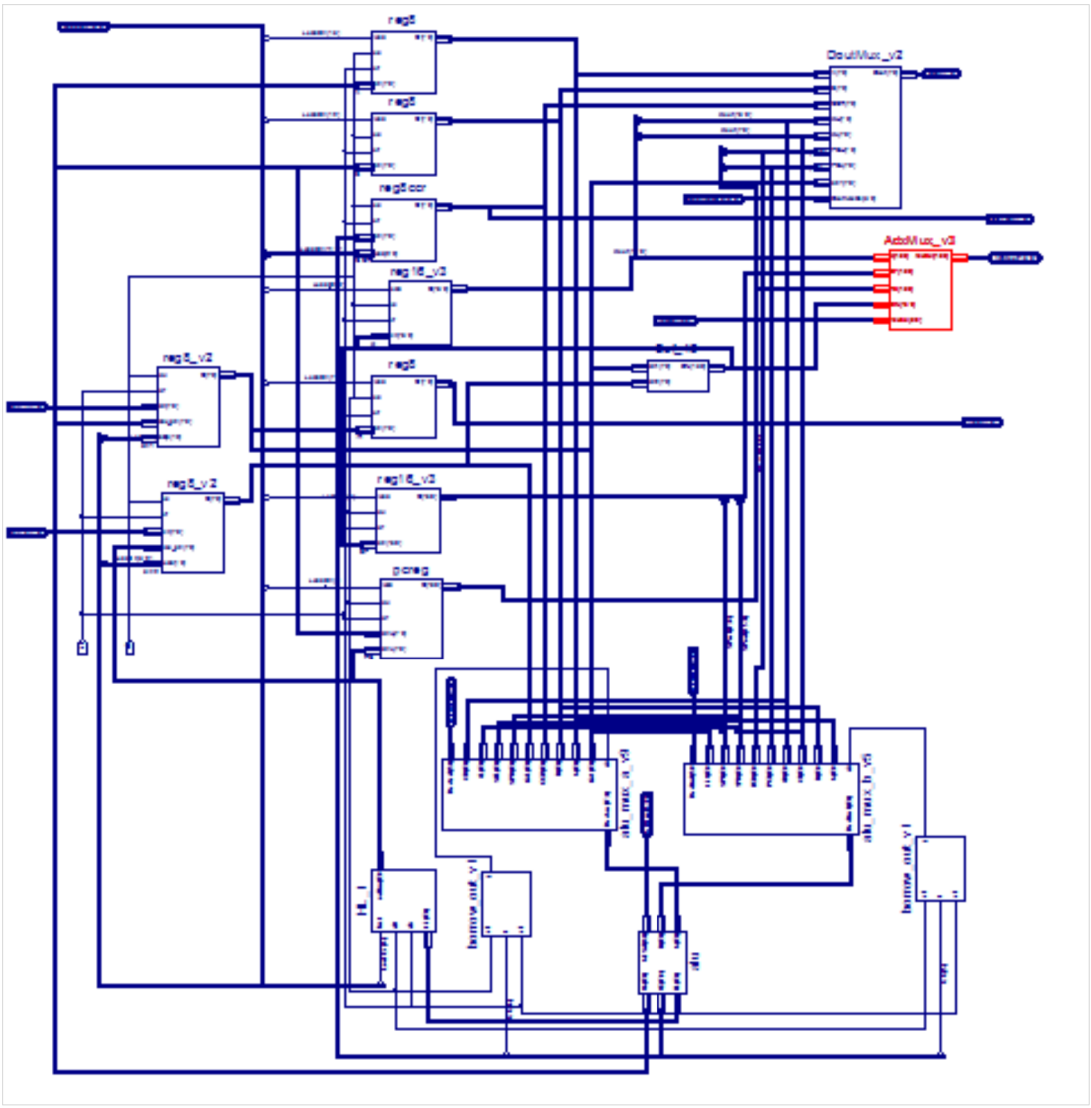


Figure 10. Final Schematic for Data Path

8-bit Register VHDL Design

The 8-bit registers within the data path are the 2 general purpose accumulators, the instruction register, the code condition register, and both din1 and din2. To design the 8-bit registers it was necessary to have a load enable, this input would make it so that the register could only be loaded with 8-bit data when the enable is high. The 8-bit register is also synchronous so data can only be written or read if the clock signal is high. The last thing to do inside of this make the output irrelevant when the register is not being used. This is because all registers and data on the data path is happening simultaneously.

```
entity reg8 is
    Port ( din : in  STD_LOGIC_VECTOR (7 downto 0);
           load : in  STD_LOGIC;
           clk : in  STD_LOGIC;
           clr : in  STD_LOGIC;
           Q : out  STD_LOGIC_VECTOR (7 downto 0));
end reg8;

architecture reg8_arch of reg8 is
  signal QQ,QX : std_logic_vector(7 downto 0);
begin

  -- Memory
  process(clk,clr)
  begin
    if clr = '1' then
      QQ <= x"00";
    elsif clk'event and clk = '1' then
      QQ <= QX;
    end if;
  end process;

  -- Multiplexer and output
  QX <= QQ when  load = '0' else
        din when load = '1' else
        "XXXXXXXX";
  Q <= QQ;
end reg8_arch;
```

Figure 11. VHDL 8-bit General Purpose Register

The code shown in above is what is used for the A, B, and CCR registers however the din1 and din2 registers needed to be designed slightly differently to accommodate other instructions. In din1 and din2 there is an additional bit for the load signals. This makes it so that the value that is being inputted into din1 or din2 can change. The specific input is labeled adx_din because it refers to an address value that would need to be loaded into din1 and din2 from within the data path and not from the memory system, which is where the data normally comes from. The figure below shows how this was implemented in VHDL.

```
entity reg8_v2 is
  port (
          din : in  STD_LOGIC_VECTOR (7 downto 0);
          adx_din : in STD_LOGIC_VECTOR(7 downto 0);
          load : in  STD_LOGIC_VECTOR(1 downto 0);
          clk : in  STD_LOGIC;
          clr : in  STD_LOGIC;
          Q : out  STD_LOGIC_VECTOR (7 downto 0)
  ) ;
end entity ; -- reg8_v2

architecture arch of reg8_v2 is
  signal QQ,QX : std_logic_vector(7 downto 0);
begin
    -- Memory
  process(clk,clr)
  begin
    if clr = '1' then
     QQ <= x"00";
    elsif clk'event and clk = '1' then
    QQ <= QX;
    end if;
  end process;

  -- Multiplexer and output
  QX <= QQ when  load = "00" else
        din when load = "01" else
        adx_din when load = "10" else
      "XXXXXXXX";
  Q <= QQ;
end architecture ; -- arch
```

Figure 12. VHDL din1 and din2 Registers

16-bit Register VHDL Design

The 16-bit registers within the data path are the index register, the stack pointer register, and the program counter. The index and stack pointer registers were designed fairly similar to the 8-bit general

purpose registers and the only thing that was changed was there increase in length to 16-bits. Everything else behaved similarly such as the load enable and a synchronous load. This can be shown in figure below.

```
entity reg16_v3 is
    Port ( din : in STD_LOGIC_VECTOR(15 downto 0);
           load : in  STD_LOGIC;
           clk : in  STD_LOGIC;
           clr : in  STD_LOGIC;
           Q : out  STD_LOGIC_VECTOR (15 downto 0));
end reg16_v3;

architecture reg16_v3_arch of reg16_v3 is
  signal QQ,QX : std_logic_vector(15 downto 0);

begin
  process (clk,clr)
  begin
    if clr = '1' then QQ <= x"0000";
    elsif clk'event and clk = '1' then QQ <= QX;
    end if;
  end process;

  QX <= QQ when load = '0' else
        din when load = '1' else
      "XXXXXXXXXXXXXXXX";
  Q <= QQ;

end reg16_v3_arch;
```

Figure 13. 16-bit General Purpose Register

The program counter followed the same idea as this code however it needed to be a bit different to account for the incrementing that would be done at each fetch. The program counter is 16-bits and the ALU is 8-bits. This meant that something would need to be done in order for the 16-bit value to add an 8-bit value without leaving the program counter in an invalid state, something that should never happen during a processors operation. The solution was to use another register to hold the low byte while the upper byte was added to any possible carry. These were then concurrently placed into the 16-bit register through two 8-bit inputs, one for the high byte and one for the low byte. This is shown in the figure below.

```
entity pcreg is
  port (
            dinH : in  STD_LOGIC_VECTOR (7 downto 0);
            dinL : IN STD_LOGIC_VECTOR(7 downto 0);
            load : in  STD_LOGIC;
            clk : in  STD_LOGIC;
            clr : in  STD_LOGIC;
            Q : out  STD_LOGIC_VECTOR (15 downto 0)
  ) ;
end entity ; -- pcreg

architecture arch of pcreg is
 signal QQ,QX : std_logic_vector(15 downto 0);
begin
  process (clk,clr)
  begin
    if clr = '1' then QQ <= x"0000";
    elsif clk'event and clk = '1' then QQ <= QX;
    end if;
  end process;

  QX <= QQ when load = '0' else
      dinH & dinL when load = '1' else
      "XXXXXXXXXXXXXXXX";
  Q <= QQ;
end architecture ; -- arch
```

Figure 14. PC Register VHDL

8-bit ALU VHDL Design

The ALU was designed in a way so that all the simple 8-bit operations could be performed with no complications. Below in table 4 is all the possible outputs that could come from the ALU and what flags are set in accordance with those operations. A and B refer to the inputs A and B of the ALU and not the Accumulators A and B. Also, there is no Invert B because the ALU only needs to invert values from the A input.

Table 4. Operations Supported by ALU

Operation	Flag Set	Description Internally
A + B	Carry Flag	Sum
A + B + 1	Carry Flag	Increment
A – B	Borrow Flag	Subtract
A – B – 1	Borrow Flag	Decrement
A AND B	'0'	Logical AND
A OR B	'0'	Logical OR
INV A	'0'	Logical Invert

These basic operations would allow for all the instructions to be supported and would provide an easy enough design challenge in constructing an ALU that could do all of this and support the necessary flags. The ALU is set up in a way where all the operations are controlled by a select of 3 bit, this is because there are only 8 operations in the ALU. The final two bits being used in the load select between the outputs of the ALU. Those outputs correspond to the Accumulators, the Hold Low Register, and the Code Condition Register. The figure below shows exactly how this was implemented in VHDL.

```
entity alu is
  port (A,B : IN STD_LOGIC_VECTOR(7 downto 0);
        HL,CC,AA  : OUT STD_LOGIC_VECTOR(7 downto 0);   --Hold Output, Code Condition Register Output, Accumulator output
        AluSel    : IN STD_LOGIC_VECTOR(4 downto 0)
  ) ;
end entity ; -- alu

architecture arch of alu is
  signal Din, AandB, AorB, InvA, InvB, Sum, AAi : STD_LOGIC_VECTOR(7 downto 0);
  signal C : STD_LOGIC_VECTOR(8 downto 0);
  signal zf,nf,cf : STD_LOGIC;
begin
    -- Logic to support subtraction, and, or
  Din   <= A when AluSel(1) = '0' else not A;
  C(0)  <= AluSel(0);
  AandB <= A and B;
  AorB  <= A or  B;
  InvA <= not A;

  -- The adder circuit
  G: for I in 0 to 7 generate
    C(I+1) <= (B(I) and C(I)) or (B(I) and Din(I)) or (C(I) and Din(I));
    Sum(I) <= (B(I) xor Din(I)) xor C(I);
  end generate;

  AA  <= AAi;
  HL <= AAi when AluSel(3) = '1' else
    "XXXXXXXX";

    -- Select the desired result
  AAi <= Sum   when AluSel(4 downto 0) = "00000" else
         Sum when AluSel(4 downto 0) = "00001" else
         not Sum when AluSel(4 downto 0) = "00010" else
         not Sum when AluSel(4 downto 0) = "00011" else
         AandB when AluSel(4 downto 0 ) = "00100" else
         AorB  when AluSel(4 downto 0) = "00101" else
         InvA when AluSel(4 downto 0) = "00110" else
         Sum   when AluSel(4 downto 0) = "01000" else
         Sum when AluSel(4 downto 0) = "01001" else
         not Sum when AluSel(4 downto 0) = "01010" else
         not Sum when AluSel(4 downto 0) = "01011" else
         AandB when AluSel(4 downto 0 ) = "01100" else
         AorB  when AluSel(4 downto 0) = "01101" else
         InvA when AluSel(4 downto 0) = "01110" else
       "XXXXXXXX";

    -- Generate flags
  zf <= '1'  when AAi = x"00" else '0';  -- High when result is zero
  cf <=  C(8) when AluSel(2 downto 1) = "00" else     -- Add
         not C(8) when AluSel(3 downto 1) = "001" else -- Subtract
  '0'; -- Pass for arithmetic                  -- Logical
  nf <= C(8) when AluSel(2 downto 1) = "01" else '0';
    -- Pick output for flags bus
  CC <= zf & cf & nf & "00000" when AluSel(4) = '0' else
    AAi when AluSel(4) = '1' else
      "XXXXXXXX";

end architecture ; -- arch
```

Figure 15. ALU VHDL Code

In general, because there were no complex mathematical instructions included in the instruction set, writing the VHDL for this ALU was far easier than if it were to have multiply or divide or something else of that nature. Leaving the ALU in this state leaves a lot of room for improvement and many more instructions could be implemented fairly easily if they were to be included in the ALU.

Multiplexer VHDL Design

Multiplexers are an essential part of the data path design. They allow for signals to be selected from the data bus to be input into another module such as the ALU or to be put out to the memory system. The multiplexers in this design currently are the ALU A input, the ALU B input, the Data output, and the

address output. The reason why these multiplexers are necessary is because they provide a way for the controller to select between the variety of signals that are available at any given point in the data path. Below are each of the multiplexers and which signals they control.

ALU MUX A

The ALU needs two separate distinct inputs when calculating any operation otherwise it will have an invalid output. This multiplexer is the A input and contains all the signals that would be necessary for the different operations. It contains a lot of signals because of the way that the ALU is set up. This multiplexer only deals with 8-bits as inputs and outputs so it was necessary for any 16-bit values to split them up into a higher byte and a lower byte. The default setting for this multiplexer is zero when nothing is being used or loaded from it. The borrow value is also set within it so that if the subtraction of a value were to cause the value to turn negative, the borrow value could be used in the next operation. Below shows the VHDL code that describes how this multiplexer operates.

```
ENTITY alu_mux_a_v9 IS
    PORT(din1,A,B,CCR,din2,SPH,SPL,XL,XH : IN STD_LOGIC_VECTOR(7 downto 0);
      cin        : IN STD_LOGIC;
        MuxAsel              : IN STD_LOGIC_VECTOR(3 downto 0);
        MuxAout              : OUT STD_LOGIC_VECTOR(7 downto 0));
END alu_mux_a_v9;

ARCHITECTURE muxA_arc of alu_mux_a_v9 IS
  signal cin1 :  STD_LOGIC_VECTOR(7 downto 0);
BEGIN
   cin1      <= "0000000" & cin;
    MuxAout     <= x"00" WHEN MuxAsel = "0000" ELSE
                   din1  WHEN MuxAsel = "0001" ELSE
                   A     WHEN MuxAsel = "0010" ELSE
                   B     WHEN MuxAsel = "0011" ELSE
                   CCR   WHEN MuxAsel = "0100" ELSE
                   cin1   WHEN MuxAsel = "0101" ELSE
                   din2   WHEN MuxAsel = "0110" ELSE
                   SPH    WHEN MuxAsel = "0111" ELSE
                   SPL    WHEN MuxAsel = "1000" ELSE
                   XL     WHEN MuxAsel = "1001" ELSE
                   XH     WHEN MuxAsel = "1010" ELSE
                   "XXXXXXXX";
END muxA_arc;
```

Figure 16. ALU MUX A VHDL Code

The signals that this multiplexer handles are as follows; din1, A, B, CCR, cin1, din2, SPH, SPL, XL, and XH. These were determined through extensive testing of each instruction.

ALU MUX B

This multiplexer is the B input of the ALU and contains all the other signals that are necessary when performing mathematical operations. Its function is fairly similar to that of the A multiplexer and it contains some of the same signals. This multiplexer also splits up the 16-bit data values into their

corresponding 8-bit high and low value, just like the A multiplexer. The default setting for the multiplexer is to output zero. Below shows the VHDL code that describes how this multiplexer operates.

```
ENTITY alu_mux_b_v5 IS
    PORT(A,B,XL,XH,PCL,PCH,SPH,SPL,din1     : IN STD_LOGIC_VECTOR(7 downto 0);
         cin                               : IN STD_LOGIC;
         MuxBsel                           : IN STD_LOGIC_VECTOR(3 downto 0);
         MuxBout                           : OUT STD_LOGIC_VECTOR(7 downto 0));
END alu_mux_b_v5;

ARCHITECTURE muxB_arc of alu_mux_b_v5 IS
  signal cin1 : STD_LOGIC_VECTOR(7 downto 0);
BEGIN
    cin1 <= "0000000" & cin;
    MuxBout <= x"00" WHEN MuxBsel = "0000" ELSE
               A     WHEN MuxBsel = "0001" ELSE
               B     WHEN MuxBsel = "0010" ELSE
               XH    WHEN MuxBsel = "0011" ELSE
               XL    WHEN MuxBsel = "0100" ELSE
               PCH   WHEN MuxBsel = "0101" ELSE
               PCL   WHEN MuxBsel = "0110" ELSE
               SPH   WHEN MuxBsel = "0111" ELSE
               SPL   WHEN MuxBsel = "1000" ELSE
               din1  WHEN MuxBsel = "1001" ELSE
               cin1  WHEN MuxBsel = "1010" ELSE
               "XXXXXXXX";
END muxB_arc;
```

Figure 17. ALU MUX B VHDL Code

The signals that this multiplexer handles are somewhat redundant to those in A but this is to allow many different varieties of inputs to be used in the ALU. The signals are as follows; A, B, XH, XL, PCH, PCL, SPH, SPL, din1, cin1.

DOUT MUX

The data out multiplexer is used to output data from certain registers to the memory system. One thing to note about it is the separation of 16-bit signals into 8-bit bytes. This is done because the memory system can only read and write 8-bit values. One drawback that this will cause is that it will take more clock cycles to output a 16-bit value. The benefit is that everything is still supported as expected and because this processor is not performance oriented then the drawback is not bad at all. The data output sends a default signal of zero when nothing is being loaded into it. Below shows the description of the DOUT multiplexer in VHDL.

```
ENTITY DoutMux_v2 IS
    PORT(A,B,CCR,XH,XL,PCH,PCL,din1  :IN STD_LOGIC_VECTOR(7 downto 0);
         DoutMuxSel  : IN STD_LOGIC_VECTOR(3 downto 0);
         Dout        : OUT STD_LOGIC_VECTOR(7 downto 0));
END DoutMux_v2;

ARCHITECTURE dout_arc OF DoutMux_v2 IS
BEGIN
    Dout    <= x"00" WHEN DoutMuxSel = "0000" ELSE
               A     WHEN DoutMuxSel = "0001" ELSE
               B     WHEN DoutMuxSel = "0010" ELSE
               CCR   WHEN DoutMuxSel = "0011" ELSE
               PCL   WHEN DoutMuxSel = "0100" ELSE
               PCH   WHEN DoutMuxSel = "0101" ELSE
               XL      WHEN DoutMuxSel = "0110" ELSE
               XH  WHEN DoutMuxSel = "0111" ELSE
               din1 WHEN DoutMuxSel = "1000" ELSE
               "XXXXXXXX";
END dout_arc;
```

Figure 18. DoutMux VHDL Code

The signals that were chosen to be output from DOUT are for specific reasons. The A and B are chosen because that is the nature of a load store architecture. The CCR was chosen as a way to store the code condition registers flag values if necessary. The Program Counter was a necessary output because it would store the return value for any subroutine. The index register was chosen to store the value of the index register in memory as was din1 for any values that came into the din1 that needed to be sent back out right away or if the value was an offset.

ADX MUX

The address multiplexer contains all the possible addressed that would be sent out to either the ROM or RAM. The values that are sent to the read are the addresses that contain the program start address, which is always located at 0xFFFE and 0xFFFF. The other addresses are all things that would be read or written to when considering the data in memory. The only thing of note is the address that will always contain the return address for a subroutine. These values are 0x7FFE and 0x7FFF. The reason for using specific values for the return address is so that things such as the stack can be reused as seen fit by the programmer throughout the programming of the microprocessor.

```
ENTITY AdxMux_v3 IS
    PORT(X,SP,PC,DIN        : IN STD_LOGIC_VECTOR(15 downto 0);
        AdxSel          : IN STD_LOGIC_VECTOR(3 downto 0);
        AdxOut          : OUT STD_LOGIC_VECTOR(15 downto 0));
END AdxMux_v3;

ARCHITECTURE adx_arc OF AdxMux_v3 IS
BEGIN
    AdxOut <= x"0000" WHEN AdxSel = "0000" ELSE
              X     WHEN AdxSel = "0001" ELSE
              SP    WHEN AdxSel = "0010" ELSE
              PC    WHEN AdxSel = "0011" ELSE
              DIN   WHEN AdxSel = "0100" ELSE
              x"FFFE" WHEN AdxSel = "0101" ELSE
              x"FFFF" WHEN AdxSel = "0110" ELSE
              x"7FFE" WHEN AdxSel = "0111" ELSE
              x"7FFF" WHEN AdxSel = "1000" ELSE
              "XXXXXXXXXXXXXXXX";
END adx_arc;
```

Figure 19. ADX MUX VHDL Code

The other signals that the address multiplexer handles are the index address, the stack address, the program counter address, and DIN. DIN is the concatenation of din1 and din2, these two registers contain temporary addresses that need to be used for retrieving data from the memory system. One such example is an offset value and the index register. The other signals present are used as necessary from the instruction that is being used.

Hold Low Register and Borrow Out Register VHDL Design

One problem that arose while designing the data path was the invalid state of the program counter that was caused when incremented because the program counter was 16-bits and the ALU was 8-bits. The solution to this problem proved to be very interesting and useful for other values that would be calculated with the 16-bit registers. The Hold Low register stores the value of the lower byte of any 16-bit value that is being used within the ALU. This prevents invalid states because the 16-bit registers can then be loaded with both the high byte, from the output of the ALU, and the low byte, from the output of Hold Low. The Hold Low register is basically the same type of register as all other 8-bit general purpose registers with a load and clear for all of its use case. Below is the code that describes the Hold Low register.

```
entity HL_1 is
   port (
      din   :  in STD_LOGIC_VECTOR(7 downto 0);
      load :   in STD_LOGIC;
      clk : in STD_LOGIC;
      clr : in STD_LOGIC;
      outlow : out STD_LOGIC_VECTOR(7 downto 0)
      );
   end HL_1;

architecture HL_arc of HL_1 is
   signal QQ,QX : std_logic_vector(7 downto 0);
begin

  -- Memory
  process(clk,clr)
  begin
    if clr = '1' then
      QQ <= x"00";
    elsif clk'event and clk = '1' then
      QQ <= QX;
    end if;
  end process;

  -- Multiplexer and output
  QX <= QQ when  load = '0' else
        din when load = '1' else
        "XXXXXXXX";
  outlow <= QQ;
end HL_arc;
```

Figure 20. Hold Low Register VHDL Code

Another thing that needed to be accounted for with the 16-bit values was the possibility for borrows and carries to be used in the addition or subtraction of a number. In order to solve this problem, a simple flip flop was used to obtain the value of the flag for carry or borrow and it was attached as an input into the ALU's multiplexer inputs. Below shows the code for this simple flip-flop with a clear signal.

```
entity borrow_out_v1 is
  port (
   clk,d,clr   : in std_logic;
   q : out std_logic
  ) ;
end entity ; -- borrow_out

architecture arch of borrow_out_v1 is

begin
   process (clk,clr) is
   begin
      if clr = '1' then
         q <= '0';
      else
         if clk'event and clk = '1' then
            q <= d;
         end if;
      end if;
   end process;
end architecture ; -- arch
```

Figure 21. Borrow or Carry Out VHDL Code

Clear Signal

One thing to note about all the VHDL modules within the data path is that they all have a signal to clear them. This was used so that there would never be an invalid state within any of the modules at startup of the microprocessor.

Memory System Design

The memory system is another one of the most critical items when designing a microprocessor. It contains the values of all the instructions that are being used in any program as well as providing a location for other data items to be read and written to. The memory system used in this microprocessor has synchronous read and synchronous write. This means that there needs to be a clock signal in order for the microprocessor to read or write data from the memory system. The way that addresses were differentiated within the memory system was by using enables that corresponded to the address of the device that was being used. In addition to this, there would be a manual signal from the controller that would state whether there was reading or writing being done. Below shows the code for the address enabler within the memory system. This adx_en also determines what the addressable spaces of the memory are.

```
entity adx_en_v3 is
  Port ( adx : in  STD_LOGIC_VECTOR (15 downto 0);
         enrom  : out STD_LOGIC;
         enram  : out STD_LOGIC;
         enout : out STD_LOGIC;
         enbxcom : out STD_LOGIC);
end adx_en_v3;
architecture arch of adx_en_v3 is
begin
  enrom <= '1'   when adx(15 downto 10) = "111111"
                 else '0';
  enram <= '1'   when adx(15 downto 14) = "01"
                 else '0';
  enout <= '1'  when adx(15 downto 4) = x"000"
                 else '0';
  enbxcom <= '1' when adx(15 downto 0) = x"0010" or
                      adx(15 downto 0) = x"001F"
                      else '0';
end arch;
```

Figure 22. adx_en VHDL Code

Addressable Memory Addresses

With the current values that are within the adx_en the addressable space for each memory device can be seen below in table 5.

Table 5. Addressable Memory Addresses

Addressable Space	Total Amount of Space	Memory Device
0x0000 to 0x0008	8 Bytes	Peripheral Output
0x0010 to 0x001F	16 Bytes	Serial Communications Device
0x3000 to 0x7FFF	16k Bytes	RAM – Random Access Memory (Read/Write)
0xFC00 to 0xFFFF	1k Bytes	ROM – Read Only Memory (Read)

The only reserved memory locations that should never be replaced are 0xFFFE, 0xFFFF, 0x7FFE, and 0x7FFF. This is because they contain the value for the program start address and the return address of the subroutines respectively.

ROM VHDL Design

The ROM Is read only memory, in the microprocessor read only is handled by an assortment of enables that only allow the ROM memory device to be read and it allows no data to be written to it. This is fairly simple to implement in VHDL because the device itself just needs to only have an input for the read enable signal and from the address enable. Once both of these signals are high the output can be read from the device during a low clock phase. One other thing that the ROM uses is a look up table, this table contains all the hexadecimal values of the opcodes and operands for the processors operation. Below is the description in VHDL.

```
entity rom_1k is
  Port ( ax  : in  STD_LOGIC_VECTOR (9 downto 0);
         clk : in STD_LOGIC;
         clp : in STD_LOGIC;
         clr : in STD_LOGIC;
         rw  : in  STD_LOGIC_VECTOR (1 downto 0);
         ena : in  STD_LOGIC;
         dy  : out  STD_LOGIC_VECTOR (7 downto 0));
end rom_1k;

architecture arch of rom_1k is
  component ROM1K_tab is
    Port    (ax : in  STD_LOGIC_VECTOR (9 downto 0);
             dy : out STD_LOGIC_VECTOR (7 downto 0));
  end component;

  signal DROM, dy1   : STD_LOGIC_VECTOR(7 downto 0);
  signal oen       : std_logic;
begin

  oen <= '1' when ena = '1' and rw = "10" else '0';

  -- Lookup table
  LUT: ROM1K_tab port map (ax=>ax,dy=>DROM);

  -- Synchronous read
  process(clk,clr,clp)
  begin
    if clr = '1' then
      dy1 <= x"00";
    elsif clk'event and clk = '1' and clp ='0' then
      dy1 <= DROM;
    end if;
  end process;
  -- Output logic
  dy  <= dy1 when oen = '1' else x"00";
end arch;
```

Figure 23. ROM VHDL Code

The figure above shows how the output of the ROM can only show a value when it is the device being read and also that it cannot replace any values internally because there is no input for 8-bit data.

RAM Design

The RAM is random access memory, this memory is volatile meaning that it will reset every time the microprocessor system runs. It can be read and written to without any changes to the code. This is in contrast to the ROM which will save its state every time the processor is re-started. The RAM is designed by using the same type of enables as the ROM however it allows the use of a write signal as well. The read and write signals are never both active at the same time because this would provide an invalid state to the memory and nothing would work properly. Below is the VHDL code that describes how the RAM is actually implemented in this design.

```
architecture arch of ram16k_v2 is
  constant RAM_SIZE : integer := 2**NA;
  subtype REC is std_logic_vector(ND-1 downto 0);
  type RAMTYPE is array(0 to RAM_SIZE-1) of REC;
  --
  signal ADXRAM : integer range 0 to RAM_SIZE-1;
  signal DRAM   : std_logic_vector(ND-1 downto 0);
  signal dtmp   : std_logic_vector(ND-1 downto 0);
  signal RAMS   : RAMTYPE;
  signal rd,wr,oen   : std_logic;
  begin
  rd <= rw(1); wr <= rw(0);
  -- Address to access
  ADXRAM <= CONV_INTEGER(ax);
  -- Reading from RAM
  process(clr,clk,clp)
  begin
    if clr = '1' then
     DRAM   <= (others=>'0');
    elsif clk'event and clk='1' and clp = '0' then
      DRAM   <= RAMS(ADXRAM);
    end if;
  end process;
  -- Output enable
  oen  <= ena and rd;
  dtmp <= DRAM when (oen = '1') else (others=>'0');
  dy   <= dx or dtmp;

  -- Writing to RAM
  process(clk)
  begin
    if clk'event and clk = '1' then
      if wr = '1' and ena = '1' and clp = '0' then
        RAMS(ADXRAM) <= dw;
      end if;
    end if;
  end process;
end arch;
```

Figure 24. RAM VHDL Code

This design for RAM instantiates 2^14 bytes and uses an array to store the data as it comes in from the data path. This method makes it volatile which is how the RAM is supposed to behave. In general, the output code and enable code is very similar to that of the ROM. The RAM is set to 0 when the clear signal is set and this is to make sure that nothing starts with an invalid state within the RAM.

Controller Design

Designing the controller was definitely one of the most challenging aspects of the entire microprocessor design. This is because it controls all the necessary enables throughout the processor that allow

everything to run as expected. All instructions are handled using the controller because it sends the enables that actually cause the instruction to perform as it was described in the instruction set architecture.

The design for the controller ended up being a giant state machine. Using this method, it was fairly simple to work on each piece of the controller to end up with a fully functional device. The controller takes a few inputs from the data path such as the status flags within the code condition register and then it sends out the corresponding signals as necessary. The controller also takes the input from the instruction register so that it can determine what instruction is currently fetched and what needs to be sent out. The controller can't be included due to the length of the code but will be attached in the appendix.

The current design of the controller is as follows. First the system is reset. This causes the controller to look for the program start address, which is located at the last 2 bytes of addressable memory. Then that value is loaded into the program counter. Once that is done the first actual byte of information can be read from ROM using a fetch and increment. The fetch retrieves the value from memory over 2 clock cycles and the increment adds 1 to the program counter. The first byte should always be an instruction unless coded improperly. This instruction is placed into the instruction register. Once there, the controller can check the value and determine what needs to be done next. If the instruction was inherent then the work then the next state involves whatever work is being done, otherwise the memory system is read again to retrieve the next value from the ROM. This cycle is known as the fetch-execute cycle and it is found everywhere in the world of computing.

Currently the design has no pipelining built in but it could be implemented if performance was more of a critical issue. There are also other performance increases that can be done to the controller but seeing as the performance was never the goal of the use case, it has been forgone so that the controller could be fully operable in time.

Controller Signals

The controller signals contain all the enables that are sent to both the data path and the memory system. The signals are as follows; load_en_out(13:0), muxAsel_out(3:0), muxBsel_out(3:0), DoutMuxSel_out(3:0), adx_sel_out(3:0), alu_sel_out(4:0), and rw_en_out(1:0). Below explains what each bit of every signal corresponds to.

Table 6. Load Enables and Device

Load Enables – 14bits	Device
load_en_out(13)	Accumulator A
load_en_out(12)	Accumulator B
load_en_out(11)	Code Condition Register
load_en_out(10)	Code Condition Register
load_en_out(9)	Code Condition Register
load_en_out(8)	X Register
load_en_out(7)	Instruction Register
load_en_out(6)	Stack Pointer Register
load_en_out(5)	Program Counter Register
load_en_out(4)	Hold Low Register
load_en_out(3)	din2 Register
load_en_out(2)	din2 Register
load_en_out(1)	din1 Register
load_en_out(0)	din1 Register

For the devices that have more than one load enable, the table below shows what those extra signals do.

Table 7. Din and CCR Load Enable Outputs

Device Enables	Input	Input Taken
din2(3:2) or din1(1:0)	00	None
	01	din
	10	adx_din
	11	XX
CCR(11:9)	000	None
	001	Load zf and store the rest
	010	Load zf, cf, and store the rest
	011	Load zf, cf, nf, and store the rest
	100	Load register

Table 8. ALU MUX A Control Signal I/O

ALU MUX A Input – 4bits	ALU MUX A Output – 8bits
0000	0x00
0001	din1
0010	A
0011	B
0100	CCR
0101	cin1
0110	din2
0111	SPH
1000	SPL
1001	XL
1010	XH

Table 9. ALU MUX B Control Signal I/O

ALU MUX B Input – 4bits	ALU MUX B Output – 8bits
0000	0x00
0001	A
0010	B
0011	XH
0100	XL
0101	PCH
0110	PCL
0111	SPH
1000	SPL
1001	din1
1010	cin1

Table 10. Dout MUX Control Signal I/O

Dout MUX Select Input – 4bits	Dout MUX Select Output – 8bits
0000	0x00
0001	A
0010	B
0011	CCR
0100	PCL
0101	PCH
0110	XL
0111	XH
1000	din1

Table 11. ADX Select Control Signal I/O

ADX Select Input – 4bits	ADX Select Output – 16bits
0000	0x0000
0001	X
0010	SP
0011	PC
0100	DIN
0101	0xFFFE
0110	0xFFFF
0111	0x7FFE
1000	0x7FFF

Table 12. ALU Control Signal I/O

ALU SEL (2:0)	Operation Performed
000	A + B
001	A + B + 1
010	A – B
011	A – B – 1
100	A AND B
101	A OR B
110	INV A

When ALU SEL (3) = 1, HL is loaded with output; When ALU SEL (4) = 1, CCR is loaded with Flags

Otherwise Accumulators have the output

The last signal is the read and write signal where the second bit refers to read and the first bit refers to write. This corresponds with the actual naming convention to make it easy and both of them cannot be enabled together otherwise an invalid state would be present.

Serial Communications Device Design

The serial communications device is crucial to our project because it is what allows us to communicate with the microprocessor. This would involve sending one bit at a time, sequentially, over a computer bus. The BxCom serial communication port was originally developed for the Sys08 processor to be used with the Boots program. There is no direct support for hardware handshaking but there are simple parallel inputs and outputs. The transmitter component within the device is double-buffered. This allows a character to be inserted while a prior character is being transmitted. Within the BxCom device are a series of signals. The system signals that are write-bus related, are ena, ax, dw, and cb. The ena signal is what enables the device using positive logic. The ax bit are the address bits which writes the data to the device. The rw bit is for read-write select where 10 equals read, 01 equals write, and 00 equals no action. The last signal is the cb or clock bus signal. The original BxCom device uses the clock bus signal to concatenate the clock, clock phase, and clear bits. However, for our system we kept the signals separate. The System signals that return the read-bus are dx, dy, irq, and ira. The dx bit returns the data bus input and the dy bit returns the data bus output. The irq and ira bits are the interrupt request

output and interrupt acknowledge input were originally used in the BxCom device but in this project, they were not needed or necessary. The next set of signals are the serial data and parallel data interface signals. RxD and TxD represent the serial communications port received and transmitted data. The PDI and PDO signals represent the parallel ports, one of which receives data or an input and the other sends data or an output.

The BxCom device uses its ax and rw signals to refer to specific register locations. The device itself has 4 registers: STPX or the Status register and punch commands, CHRS or Configuration and soft-reset register, RDTD or Serial Communications receive and transmit data register, and PDIO or Parallel input and output registers. The status register indicates that the transmit is ready for a character, the receiver has a character, a break pattern was detected, and the transmitter is in idle state. The configuration register writes to configure the device, however if written to the two least significant bits, it will cause the device to have a software reset. The serial communications receive and transmit data register returns the received byte data register value using a read. A write command causes a byte to be transmitted. The parallel input and output registers is where reading returns parallel data. The parallel input data is first sampled with a write to the status register. Writing sends out parallel data and is sampled with a write to the status register as well.

Assembler

The assembler was designed as an afterthought to the original design of the microprocessor. Its purpose was to make programming the processor much faster. The assembler is written entirely in Python and it is used to assemble the custom assembly level language instructions into machine code that can be read by the processor. The file that it assembles the code into is of the s19 format. This format was primarily used because the bootloader takes in files of that format as well as the RomTools tool that was developed by our technical advisor.

To design the assembler the essential parsing was taken out of the instruction set simulator and converted so that it could encode the values as necessary in the assembler. Then any extra information data had to be handled as well so this was added to the assembler's abilities. Finally, the assembler took the encoded string and would write to an s19 file all the data that was necessary for an s19 file format such as the length and the checksum. This would ensure that the code could be read into the processor using the bootloader or using rom tools.

In general, the assembler is a quick and dirty approach to assembly however because it works and improves the speed of programming it is definitely useful for the project. The future of this assembler could have many more features and be improved significantly in compile time. For now, it is a useful tool that was designed to improve the projects use case and usefulness.

The code for the assembler is attached within the appendix but below is some example code and the format in which it is output for the bootloader or ROM to use for the programs execution.

```
#Assembly Language Test

XDEF 0xFC00


ORG 0xFC00

Start:  lda 0x04
        adda 0x1F
        inva
        adda 0x2F
        sta 0x4000
        jsr [TEST]
        ldx 0x400
        sta 0x400
        inca
        sta 0x401
        inca
        sta 0x402
        ldb [X+0]
        ldb [X+1]
        ldb [x+2]
        stb [x+5]
        sta [X+6]


End:    hlt

TEST:   lda 0x06
        ldb 0x03
        lds 0x5012
        psha
        pshb
        rfs
```

Figure 25. Assembly Language Test Code

```
S135FC001004121F07122F26400051FC2016040026040003260401032604024100410141024305420 6FF0000100611031750123031522 2
S105FFFEFC0001
```

Figure 26. Compiled S19 Assembly File

Using RomTools the file is converted into a ROM look up table to be used by the memory system which is shown in the figure below.

```
1  --------------------------------------------------------------------------------
2  -- rom1k_asmtest.vhd
3  -- This is a ROM-table file that describes the contents of a ROM
4  --
5  -- <ROMTX>3<X> - Format version
6  -- <MNAME>rom1k<X> - VHDL module name
7  -- <TNAME>rom1k_tab<X> - VHDL table module name
8  -- <ABITS>10<X> - Address bits for ROM - decimal (value)
9  -- <RIBUS>n<X> - Return bus input (y or n)
10 -- <UNAME>brice<X> - Designer name
11 -- <MDATE>5/4/18<X> - Make date
12 -- <SNAME>rom1k_asmtest.vhd<X> - S19 file name
13 -- </ROMTX>
14 --------------------------------------------------------------------------------
15
16 library IEEE;
17 use IEEE.STD_LOGIC_1164.ALL;
18 use IEEE.STD_LOGIC_ARITH.ALL;
19 use IEEE.STD_LOGIC_UNSIGNED.ALL;
20
21 entity rom1k_tab is
22   Port   (ax : in  STD_LOGIC_VECTOR (9 downto 0);
23           dy : out STD_LOGIC_VECTOR (7 downto 0));
24 end rom1k_tab;
25
26 architecture arch of rom1k_tab is
27   constant LUT_ENTRIES : integer := 1024;
28   subtype  LUT_REC is std_logic_vector(7 downto 0);
29   type LUT_TYPE is array(0 to LUT_ENTRIES-1) of LUT_REC;
30   signal AXLUT : integer range 0 to LUT_ENTRIES-1;
31
32   constant LUTS : LUT_TYPE := (
33 x"10",x"04",x"12",x"1F",x"07",x"12",x"2F",x"26",x"40",x"00",x"51",x"FC",x"20",x"16",x"04",x"00",-- 0000
34 x"26",x"04",x"00",x"03",x"26",x"04",x"01",x"03",x"26",x"04",x"02",x"41",x"00",x"41",x"01",x"41",-- 0001
35 x"02",x"43",x"05",x"42",x"06",x"FF",x"00",x"00",x"10",x"06",x"11",x"03",x"17",x"50",x"12",x"30",-- 0002
36 x"31",x"52",x"00",x"00",x"00",x"00",x"00",x"00",x"00",x"00",x"00",x"00",x"00",x"00",x"00",x"00",-- 0003
37 x"00",x"00",x"00",x"00",x"00",x"00",x"00",x"00",x"00",x"00",x"00",x"00",x"00",x"00",x"00",x"00",-- 0004
38 x"00",x"00",x"00",x"00",x"00",x"00",x"00",x"00",x"00",x"00",x"00",x"00",x"00",x"00",x"00",x"00",-- 0005
```

Figure 27. Compiled Assembly Loaded into ROM LUT

Bootloader

The bootloader was a necessary component in the design of the microprocessor system because it allows the use case to be satisfied. The bootloaders purpose is to allow code to be uploaded to the board on the fly and executed. The specific code that is uploaded is through the serial communications device and the type of file that is accepted is s19. The main benefit in using a bootloader is that the speed of programming the microprocessor is greatly increased. The reason for this is that when code is loaded onto the FPGA it takes a long time to synthesize and load all the hardware descriptions onto the board. While using a bootloader that entire process is skipped and so code can be directly loaded an executed after a one-time upload of the hardware descriptions with the bootloader loaded into ROM.

The bootloader was written as a translation from a precious bootloader design that was provided by our technical advisor. This code had to be functionally the same in order to achieve the same output goals however the code was written for a different architecture entirely. Some things in particular that needed to be overcome were branches and branch clears, as well as immediate move instructions and some other specific pedantic items. The code for the bootloader is attached in the appendix. It is currently written for the custom microprocessor architecture that is used in this project.

Test Assembly Programs

In order to verify the status of the use case for the microprocessor test programs were written to ensure that the final result would be in line with the design goals. This meant including many small test programs during the design phase to ensure the operation was as expected. Some simple programs that were written for example was multiplication via repeated addition. Another example was a program that prints hello world via an ASCII string. Using both of these as an example it was possible to see that the final processor was on the right track to give the results that were wanted.

The test assembly programs were also implemented onto the board and fully simulated to ensure the operation was also as expected. This is discussed more in the Testing Methodology and Results Section.

Implementation on Spartan 6

Implementation was done using the tools currently available within the ISE Design Suite. It refers to the entire hardware description of the device being synthesized and uploaded to the FPGA board. Doing this was essential to the completion of the microprocessor system. It would also allow the processor to finally be used outside of a simulation. Implementation is successful when the code is able to transfer properly without any errors.

Chapter 7: Testing Methodology and Results

In any large-scale project, it is a necessity to keep testing and incrementing design as the project moves forward. For this microprocessor system, testing was done for each and every component individually and then as the blocks grew bigger the testing would be done for the larger schematic. Once everything was put together the final tests that were run ensured that the operation of all the instructions was as expected.

VHDL Testing

The first type of testing that was done was through VHDL testbenches. In this testing phase the VHDL module of each individual block was tested by itself to make sure that the code was giving valid outputs and no errors from any of the possible inputs that would be given. In all of the testbenches that were written a clock signal was given and set to oscillate at some arbitrary value such as 50ns. Then the testbench would begin with a clear signal set to high as if to simulate the state in which the processor would begin following reset.

Testbenches are an extremely powerful tool to use when simulating integrated circuits and all of the tools within ISE make it simple and easy to generate testbenches for the respective module that is being simulated.

Inputs were handled in the testbench by giving the signals specific values at certain periods of time. The one thing that always need to be considered was the state of the clock. This is mainly because the modules within the processor all operate on a rising clock edge. If the code were to be set before the

clock edge than it wouldn't give the expected output. This would become an even greater issue when the timing becomes a greater challenge as is with the memory system.

All of the simulations were done using the tool known as ISIM. ISIM provides a nice graphical user interface that can display a variety of wave signals from all the digital logic that is contained within the simulation. Depending on which module was being tested, signals would be added to the wave window to show what the outputs and inputs were at any given point within the simulation. ISIM also gives the user the ability to run the simulation for any period of time which is convenient when trying to locate a specific error or bug that is found during testing.

Memory System Testing

The memory system was not particularly difficult to test because it had many of the same test signals as previous modules before it. However, after combining it with the data path, writing testbenches required that the timing be perfect to simulate the actual reading or writing of data from the data path to the memory system and vice versa.

While the memory system was contained in its own top-level module it was much easier to design a test bench that would just test the signals and outputs that would be expected from the memory system. Knowing that everything worked on its own, would make the process of connecting it to the controller and data path a much better experience.

Controller Testing

The controller was one of the most critical components in the entire design of the microprocessor and so naturally it utilized the most amount of testing. The testing process for it was iterative for each instruction that was handled. Before any instructions were set to be used a reset state had to be implemented. This process involved writing the code for the controller that would handle the reset, then running the simulation testbench in ISIM and checking the output of waves that should be set. Following reset the program counter was supposed to be loaded with the last two bytes of memory. If this process did not work then the code was modified and the simulation was run again.

Once the reset was set and working then the actual instruction handling code could be written in the controller. This code would do one thing but would go through many states. That is why simulating it in ISIM was necessary. The simulation would allow almost a sort of debugging to see what the current state was as well as the next state the controller was headed to.

Through this iterative process the entire controller was able to be designed and fully operational. It was definitely more time consuming then writing all the code at once but it also ensured that the controller was working properly which would mean that the entire microprocessor system was working properly. Shown below are all the signals that were checked after each run of the simulation.

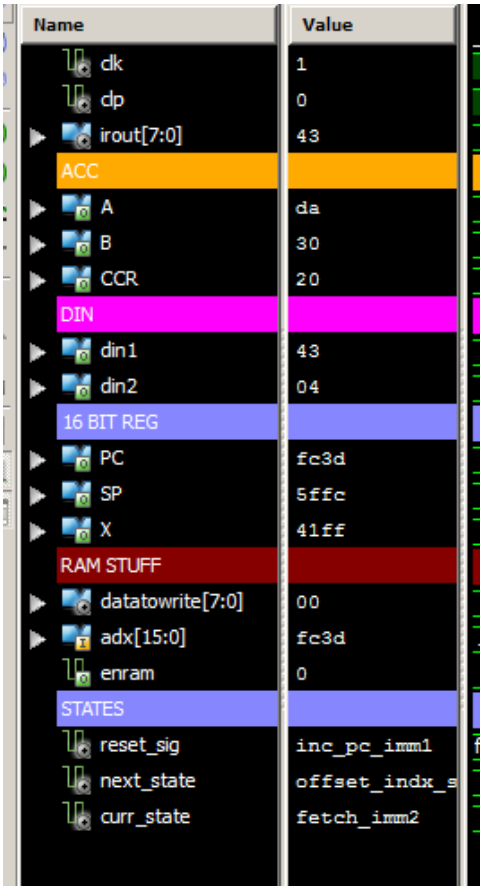


Figure 28. Signals Checked after each Simulation Increment

The controller simulations were considered finished when every single simulation was functional for every type of instruction.

Assembly Code Testing

The idea behind assembly code testing was to make sure that when the board was finally implemented that all the simulation outputs obtained would match their real-world counterparts. In general, the testing for this was much easier than writing a test bench because it would involve all the tools and components that were already built.

The code was written in assembly, then it was put through the assembler to make an s19 file. With this file, RomTools was utilized to make a ROM look up table that would contain all the necessary instructions and data. Finally, a test bench could be made with the basics of just a clock signal and a reset signal. This test bench would be simulated and all the output waves would be observed till the completion of the assembly code and it could be compared against the instruction set simulator to make sure that the output of the program matched that of the simulator. With all of this done, the simulation and testing portion of the project was complete.

Example Simulation of Load A Immediate

This is an example of the simulation that shows accumulator A being loaded with 8-bit immediate data.

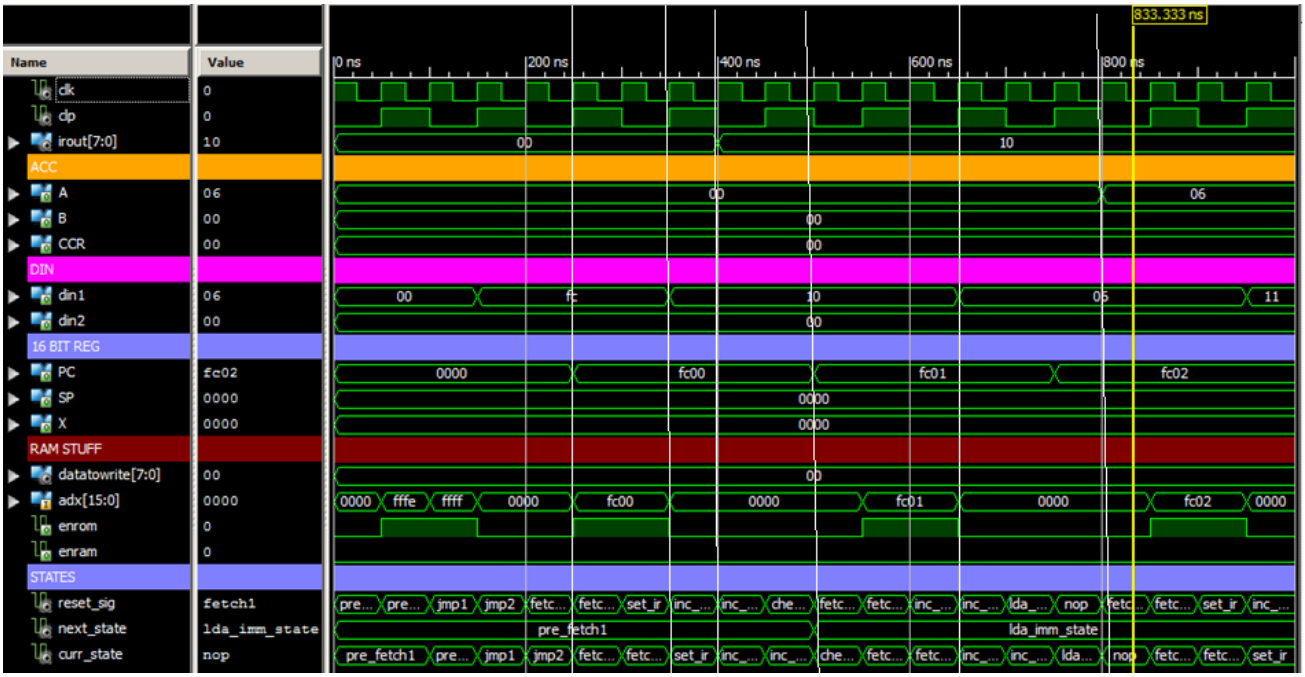


Figure 29. Load A Immediate Simulation

In the first portion of the wave window shown above the reset signal is sent to high and the processor fetches the program start address from the last two bytes in ROM (fffe and ffff seen at signal adx[15:0]) and loads it into the program counter (seen at signal PC with fc00).

Next, the processor fetches the first byte located at the address of the program counter and loads it into the instruction register (seen at time 380ns at signal irout[7:0] = 10). The value in the instruction register is now 0x10. This corresponds to a Load A Immediate instruction.

The processor now knows it needs to fetch another byte from memory so it increments and fetches the next byte (seen at time 500ns PC = fc01).

The value that is retrieved can be seen in the signal din1 as 0x06. This value is then loaded into A (seen at time 800ns) and the program counter can be incremented once again.

This is the fetch execute cycle within the custom microprocessor and just one example of its functionality.

Chapter 8: Conclusion and Future Work

This project proved to be one of the greatest undertakings as an undergraduate within the University of Hartford. It was also the most fun and exhilarating experiences that could be had as a computer engineering student.

The end result was less than what was expected of the use case but the microprocessor system was still an extremely useful learning tool and step into what is possible in the world of embedded computing.

The reason that the project didn't satisfy our use case is because the FPGA was never implemented and tested with a peripheral device. This is partially due to the time constraint of the semester and also because the serial communications device was not finished in time.

The total percentage of completion was about 90% with the last 10% being the peripheral device communications and testing on a physical FPGA.

The final cost was over $150 and it should have been more considering there should have been devices purchased but none of that happened as time went on and the communications device was at a standstill.

In the future, plenty can be done to this project to improve. For one, there are plenty of optimizations that can be made within the data path and the controller. In the data path many extra unused load signals could be removed as well as improvements to the multiplexer inputs to only the necessary data. In the controller, optimization can be made to the fetch and execute cycle as well as pipelining to improve speed and overall usability. Other things could be including more instructions to be supported as well as expanding the ALU to allow for more operations.

Lastly the project is extremely close to where it originally set out to be and getting the processor to work with peripheral devices shouldn't be too hard from where the project is currently at.

All code for this processor will be released under a liberal and permissive open source license such as MIT. This will allow any person in the future to modify the processor and make improvements as seen fit.

References

[1] K. Hill, "Project nod4 - A Simple Yet Non-Trivial Soft-Core Processor," 8 Septemeber 2009. [Online]. Available: http://uhaweb.hartford.edu/kmhill/projects/nod4/. [Accessed 2018].

[2] J. v. Neuman, "First Draft of a Report on the EDVAC," University of Pennsylvania, Pennsylvania, 1945.

[3] IEEE, "IEEE Standard Test Access Port and Boundary Scan Architecture," *IEEE*, vol. 1149.1, no. 2001, 2001.

[4] IEEE Computer Society, "IEEE Standard VHDL Language Reference Manual," *IEEE Standards*, vol. 1076, pp. 1-212, 2002.

[5] Motorola, Programmer's Reference Manual, Motorola, 1992.

[6] Unix MAN Pages, "Motorola S-records," [Online]. Available: http://www.amelek.gda.pl/avr/uisp/srecord.htm.

[7] Wikipedia, "Motorola S-record format ready reckoner," [Online]. Available: https://en.wikipedia.org/wiki/SREC_(file_format)#/media/File:Motorola_SREC_Chart.png.

[8] D. Wright, "Standards Development Working Group," IEEE, [Online]. Available: http://standards.ieee.org/develop/wg/1284.html. [Accessed May 2018].

Appendix

Appendix A: Instruction Set Architecture and Encoding Documentation

INSTRUCTION	BINARY Representation	HEX Representation
Implied/Inherent		
nop	0000 0000	0x00
add_acc	0000 0001	0x01
sub_acc	0000 0010	0x02
inc_a	0000 0011	0x03
inc_b	0000 0100	0x04
dec_a	0000 0101	0x05
dec_b	0000 0110	0x06
inv_a	0000 0111	0x07
inv_b	0000 1000	0x08
cmp	0000 1001	0x09
hlt	1111 1111	0xFF
Immediate		
lda_imm	0001 0000	0x10
ldb_imm	0001 0001	0x11
adda_imm	0001 0010	0x12
addb_imm	0001 0011	0x13
suba_imm	0001 0100	0x14
subb_imm	0001 0101	0x15
ldx_imm	0001 0110	0x16
lds_imm	0001 0111	0x17
anda_imm	0001 1000	0x18
andb_imm	0001 1001	0x19
ora_imm	0001 1010	0x1A
orb_imm	0001 1011	0x1B
Direct		
lda_dir	0010 0000	0x20
ldb_dir	0010 0001	0x21
adda_dir	0010 0010	0x22
addb_dir	0010 0011	0x23
suba_dir	0010 0100	0x24
subb_dir	0010 0101	0x25
sta_dir	0010 0110	0x26
stb_dir	0010 0111	0x27
ldx_dir	0010 1000	0x28
lds_dir	0010 1001	0x29

anda_dir	0010 1010	0x2A
andb_dir	0010 1011	0x2B
ora_dir	0010 1100	0x2C
orb_dir	0010 1101	0x2D

Stack

push_a	0011 0000	0x30
push_b	0011 0001	0x31
pop_a	0011 0010	0x32
pop_b	0011 0011	0x33
push_imm	0011 0100	0x34
push_dir	0011 0101	0x35
push_indx	0011 0110	0x36

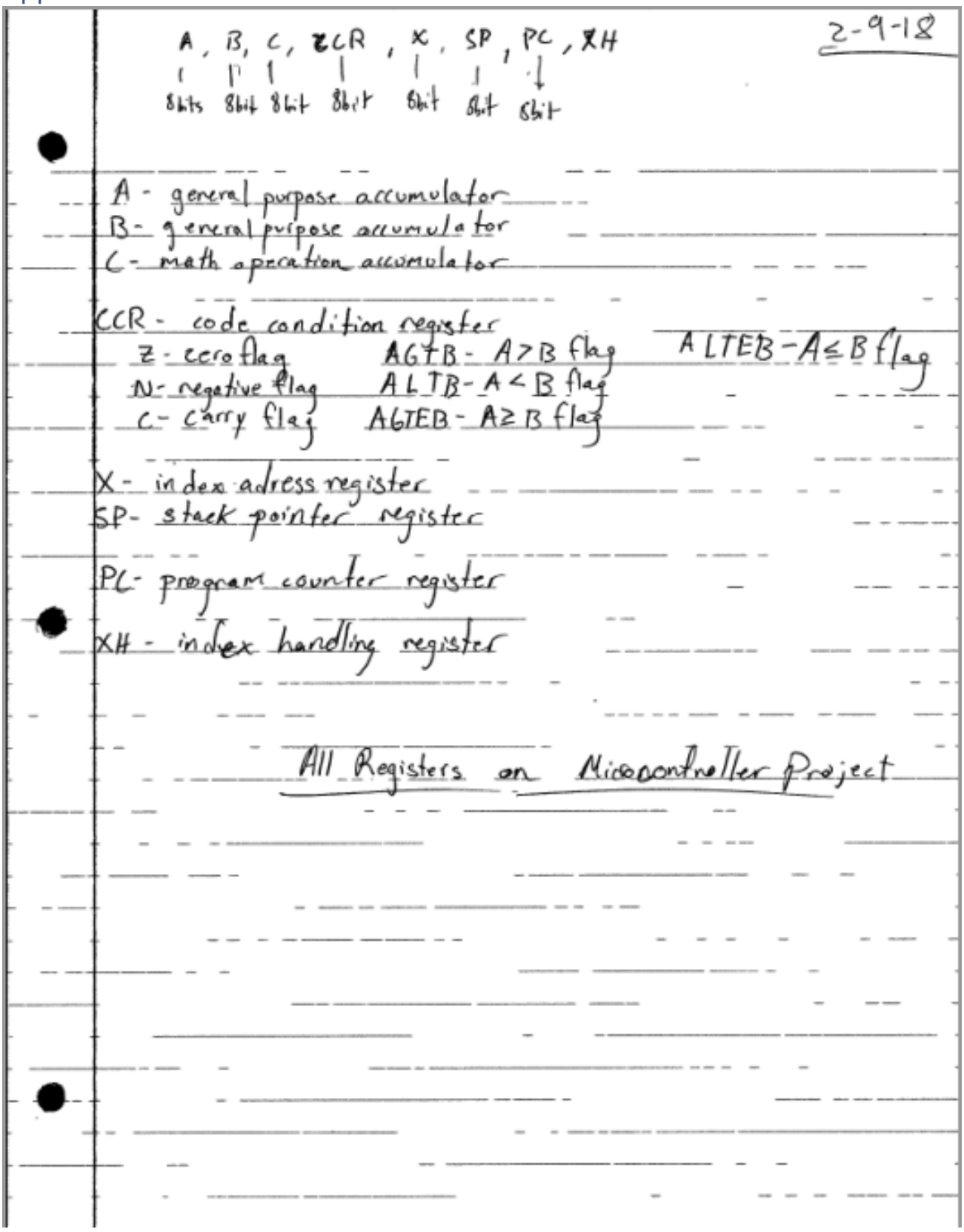
Indexed

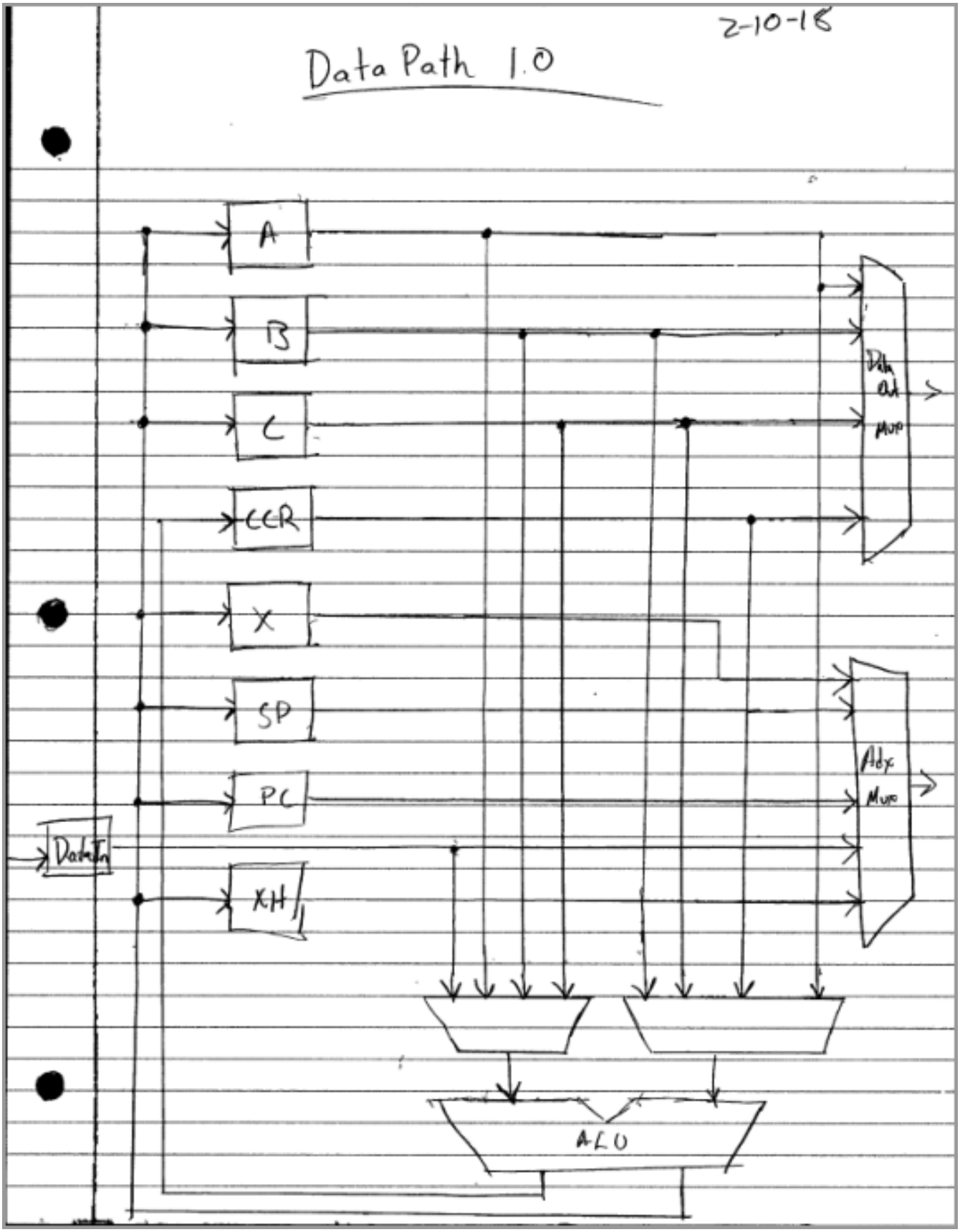
lda_indx	0100 0000	0x40
ldb_indx	0100 0001	0x41
sta_indx	0100 0010	0x42
stb_indx	0100 0011	0x43
adda_indx	0100 0100	0x44
addb_indx	0100 0101	0x45
suba_indx	0100 0110	0x46
subb_indx	0100 0111	0x47
anda_indx	0100 1000	0x48
andb_indx	0100 1001	0x49
ora_indx	0100 1010	0x4A
orb_indx	0100 1011	0x4B
inc_x	0100 1100	0x4C
dec_x	0100 1101	0x4D

Jump

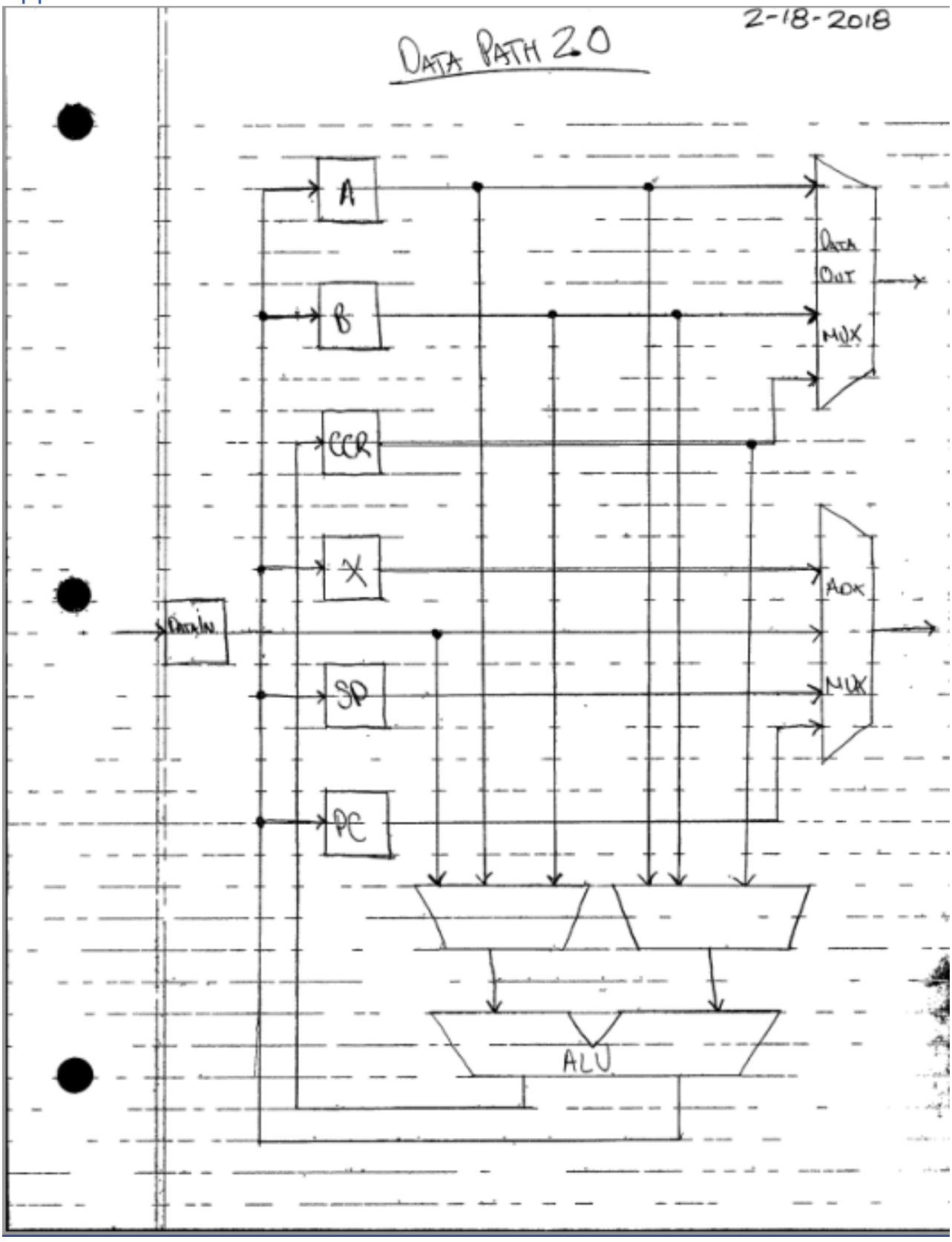
jmp_imm	0101 0000	0x50
jmp_sub	0101 0011	0x51
ret_sub	0101 0100	0x52
jmp_eq	0101 0101	0x53
jmp_neq	0101 0110	0x54
jmp_gt	0101 0111	0x55
jmp_lt	0101 1000	0x56
jmp_gte	0101 1001	0x57
jmp_lte	0101 1010	0x58

Appendix B: Data Path Revision 1.0

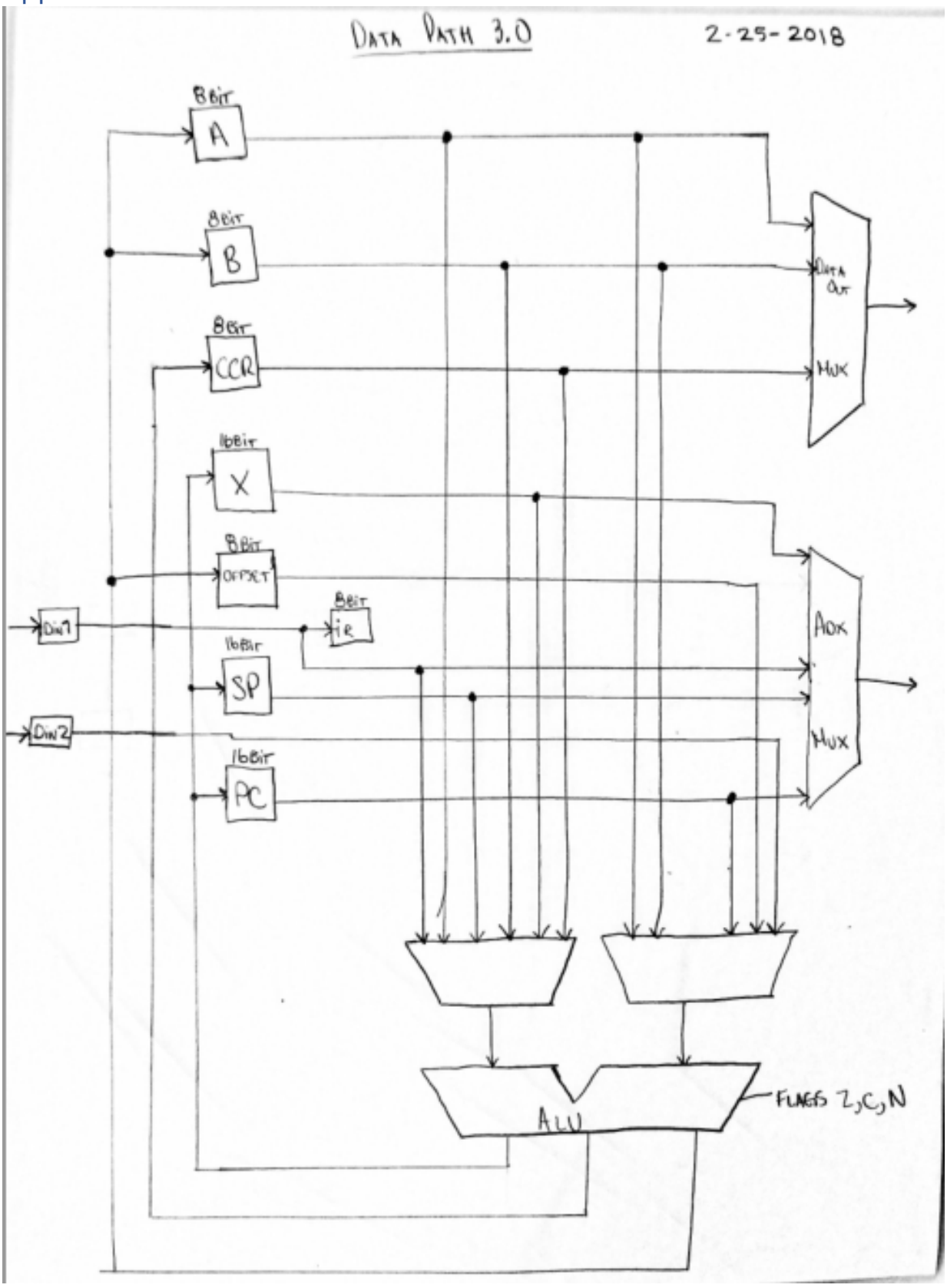



2-10-18
Data Path 1.0
A
B
C
CCR
X
SP
PC
XH
ALU

Appendix C: Data Path Revision 2.0



Appendix D: Data Path Revision 3.0



Appendix E: Controller VHDL Code

```
library ieee ;
    use ieee.std_logic_1164.all ;
    use ieee.numeric_std.all ;

entity cpu_cont_v5 is
  port (
    clk, clp, clr : IN std_logic;
    ir_in : IN std_logic_vector(7 downto 0);
    load_en_out : OUT std_logic_vector(13 downto 0);
    muxAsel_out : out std_logic_vector(3 downto 0);
    muxBsel_out : out std_logic_vector(3 downto 0);
    DoutMuxSel_out : out std_logic_vector(3 downto 0);
    adx_sel_out : out std_logic_vector(3 downto 0);
    alu_sel_out : out std_logic_vector(4 downto 0);
    CCR_in : in std_logic_vector(7 downto 0);
    rw_en_out : out std_logic_vector(1 downto 0)
  ) ;
end entity ; -- cpu_cont_v5

architecture arch of cpu_cont_v5 is
    TYPE State_type is (pre_fetch1,pre_fetch2,jmp1,jmp2,-- for reset
                        fetch1,fetch2,inc_pc1,inc_pc2,set_ir,check_ir, -- for initial setup
                        --indx states
                        fetch_indx1,fetch_indx2,offset_indx_state1,offset_indx_state2,inc_x1,dec_x1,inc_x2,dec_x
                        --jmp states
                        jmp_eq_state,jmp_neq_state,jmp_gt_state,jmp_lt_state,jmp_gte_state,jmp_lte_state,push_pc
                        fetch_ret_sub,
                        --imm states
                        fetch_imm1,fetch_imm2,inc_pc_imm1,inc_pc_imm2,lda_imm_state,ldb_imm_state,adda_imm_state
                        subb_imm_state,fetch_2_bytes1,fetch_2_bytes2,fetch_2_bytes3,fetch_2_bytes4,inc_pc_2b_1,i
                        ldx_imm_state,lds_imm_state,anda_imm_state,andb_imm_state,ora_imm_state,orb_imm_state,
                        --dir states
                        sta_dir_state,stb_dir_state,lda_dir_state,ldb_dir_state,fetch_dir1,fetch_dir2,anda_dir_s
                        orb_dir_state,adda_dir_state,addb_dir_state,suba_dir_state,subb_dir_state,
                        --stack states
                        pre_dec_state1,pre_dec_state2,load_sp_state1,load_sp_state2,push_a_state,push_b_state,po
                        post_inc_state2,post_inc_state1,push_imm_state,push_dir_state,fetch_imm1_s,fetch_dir1_s,
                        lda_imm_s_state,ldb_imm_s_state,
                        --inh states
                        add_acc_inh_state,sub_acc_inh_state,inc_a_inh_state,inc_b_inh_state,dec_a_inh_state,dec_
                        inv_b_inh_state,cmp_state,
                        hlt,nop);
    signal reset_sig : State_type;
    signal next_state :State_type;
    signal curr_state :State_type;
begin
    process(clk,clp,clr,reset_sig)
     begin
    if clr = '1' then
        rw_en_out <= "00";adx_sel_out <= "0000";load_en_out <= "00000000000000";muxAsel_out <= "0000";muxBsel_ou
        reset_sig <= pre_fetch1;
     elsif clk'event and clk ='1' then
          case reset_sig is
            when pre_fetch1 => -- read from m[FFFE]
                curr_state <= pre_fetch1;
                if  clp = '1' then
                    rw_en_out <= "10";  --reading
                    adx_sel_out <= "0101";   --select FFFE from adx mux
                    load_en_out <= "00000000000000";    --not loading anything yet
                    muxAsel_out <= "0000";
                    muxBsel_out <= "0000";
                    DoutMuxSel_out <= "0000";
                    alu_sel_out <= "00000";
                    reset_sig <= pre_fetch2;
                end if;
            when pre_fetch2 =>  -- put m[fffe] into din1 and read m[ffff] for din2
                curr_state <= pre_fetch2;
                if clp = '0' then
                    rw_en_out <= "10";  --reading
                    adx_sel_out <= "0110";   --select FFFF from adx mux
                    load_en_out <= "00000000000001";    --now loading m[FFFE] into din1
                    muxAsel_out <= "0000";
                    muxBsel_out <= "0000";
                    DoutMuxSel_out <= "0000";
                    alu_sel_out <= "00000";
                    reset_sig <= jmp1;
                end if;
            when jmp1 => --Loads the program counter with din1:din2
                curr_state <= jmp1;
                muxAsel_out <= "0110";   --din2
```

```
81                muxBsel_out <= "0000";  --00
82                alu_sel_out <= "01000"; --A+B
83                load_en_out <= "00000000010000";    --Load HL to store low byte
84                DoutMuxSel_out <= "0000";
85                rw_en_out <= "00";
86                adx_sel_out <= "0000";
87                reset_sig <= jmp2;
88            when jmp2 =>
89                if clp = '0' then
90                    curr_state <= jmp2;
91                    muxAsel_out <= "0001";   --din1
92                    muxBsel_out <= "0000";  --00
93                    alu_sel_out <= "00000"; --A+B
94                    load_en_out <= "00000000100000";    --Load PC with both alu output + HL at the same time
95                    DoutMuxSel_out <= "0000";
96                    rw_en_out <= "00";
97                    adx_sel_out <= "0000";
98                    reset_sig <= fetch1;
99                end if;
100
101               -- Essentially sending out FC00 to read first byte of ROM
102 ----ABOVE THIS LINE IS RESET
103           when fetch1 =>
104               curr_state <= fetch1;
105               -- the program counter is now loaded
106               if  clp = '1' then
107                   rw_en_out <= "10";
108                   load_en_out <= "00000000000000";
109                   muxBsel_out <= "0000";
110                   muxAsel_out <= "0000";
111                   alu_sel_out <= "00000";
112                   DoutMuxSel_out <= "0000";
113                   adx_sel_out <= "0011";
114                   reset_sig <= fetch2;
115               end if;
116           when fetch2 =>
117               if clp = '0' then
118                   curr_state <= fetch2;
119                   rw_en_out <= "10";                  -- want to read m[PC]
120                   load_en_out <= "00000000000001";    --Loading din1 with data that was read from the previous
121                   muxBsel_out <= "0000";
122                   muxAsel_out <= "0000";
123                   alu_sel_out <= "00000";
124                   DoutMuxSel_out <= "0000";
125                   adx_sel_out <= "0011";       -- sending PC to memsys
126                   reset_sig <= set_ir;
127               end if;
128           when inc_pc1 =>
129               curr_state <= inc_pc1;
130               load_en_out <= "00000000010000"; -- Loading hold low
131               muxAsel_out <= "0000";           -- choosing 0 for A of ALU
132               muxBsel_out <= "0110";          -- choosing PCL for B of ALU
133               alu_sel_out <= "01001";         -- A+B+1 so that the lowerbyte is incremented and put in the HL
134               rw_en_out <= "00";
135               adx_sel_out <= "0000";
136               DoutMuxSel_out <= "0000";
137               reset_sig <= inc_pc2;
138           when inc_pc2 =>
139               curr_state <= inc_pc2;
140               alu_sel_out <= "00000";         -- A+B
141               muxBsel_out <= "0101";          -- PCH
142               muxAsel_out <= "0101";           -- CIN
143               load_en_out <= "00000000100000";--Load PC
144               rw_en_out <= "00";
145               DoutMuxSel_out <= "0000";
146               adx_sel_out <= "0000";
147               reset_sig <= check_ir;
148           when set_ir =>
149               curr_state <= set_ir;
150               load_en_out <= "00000010000000"; -- Load IR
151               alu_sel_out <= "00000";
152               muxAsel_out <= "0000";
153               muxBsel_out <= "0000";
154               DoutMuxSel_out <= "0000";
155               adx_sel_out <= "0000";
156               rw_en_out <= "00";
157               reset_sig <= inc_pc1;
158           when check_ir =>
159               curr_state <= check_ir;
160               if ir_in = x"10" then
```

```
161                     reset_sig <= fetch_imm1;
162                     next_state <= lda_imm_state;
163                 elsif ir_in = x"11" then
164                     reset_sig <= fetch_imm1;
165                     next_state <= ldb_imm_state;
166                 elsif ir_in = x"12" then
167                     reset_sig <= fetch_imm1;
168                     next_state <= adda_imm_state;
169                 elsif ir_in = x"13" then
170                     reset_sig <= fetch_imm1;
171                     next_state <= addb_imm_state;
172                 elsif ir_in = x"14" then
173                     reset_sig <= fetch_imm1;
174                     next_state <= suba_imm_state;
175                 elsif ir_in = x"15" then
176                     reset_sig <= fetch_imm1;
177                     next_state <= subb_imm_state;
178                 elsif ir_in = x"16" then
179                     reset_sig <= fetch_2_bytes1;
180                     next_state <= ldx_imm_state;
181                 elsif ir_in = x"17" then
182                     reset_sig <= fetch_2_bytes1;
183                     next_state <= lds_imm_state;
184                 elsif ir_in = x"18" then
185                     reset_sig <= fetch_imm1;
186                     next_state <= anda_imm_state;
187                 elsif ir_in = x"19" then
188                     reset_sig <= fetch_imm1;
189                     next_state <= andb_imm_state;
190                 elsif ir_in = x"1A" then
191                     reset_sig <= fetch_imm1;
192                     next_state <= ora_imm_state;
193                 elsif ir_in = x"1B" then
194                     reset_sig <= fetch_imm1;
195                     next_state <= orb_imm_state;
196                 elsif ir_in = x"20" then
197                     reset_sig <= fetch_2_bytes1;
198                     next_state <= lda_dir_state;
199                 elsif ir_in = x"21" then
200                     reset_sig <= fetch_2_bytes1;
201                     next_state <= ldb_dir_state;
202                 elsif ir_in = x"22" then
203                     reset_sig <= fetch_2_bytes1;
204                     next_state <= adda_dir_state;
205                 elsif ir_in = x"23" then
206                     reset_sig <= fetch_2_bytes1;
207                     next_state <= addb_dir_state;
208                 elsif ir_in = x"24" then
209                     reset_sig <= fetch_2_bytes1;
210                     next_state <= suba_dir_state;
211                 elsif ir_in = x"25" then
212                     reset_sig <= fetch_2_bytes1;
213                     next_state <= subb_dir_state;
214                 elsif ir_in = x"26" then
215                     reset_sig <= fetch_2_bytes1;
216                     next_state <= sta_dir_state;
217                 elsif ir_in = x"27" then
218                     reset_sig <= fetch_2_bytes1;
219                     next_state <= stb_dir_state;
220                 elsif ir_in = x"2A" then
221                     reset_sig <= fetch_2_bytes1;
222                     next_state <= anda_dir_state;
223                 elsif ir_in = x"2B" then
224                     reset_sig <= fetch_2_bytes1;
225                     next_state <= andb_dir_state;
226                 elsif ir_in = x"2C" then
227                     reset_sig <= fetch_2_bytes1;
228                     next_state <= ora_dir_state;
229                 elsif ir_in = x"2D" then
230                     reset_sig <= fetch_2_bytes1;
231                     next_state <= orb_dir_state;
232                 elsif ir_in = x"30" then
233                     reset_sig <= pre_dec_state1;
234                     next_state <= push_a_state;
235                 elsif ir_in = x"31" then
236                     reset_sig <= pre_dec_state1;
237                     next_state <= push_b_state;
238                 elsif ir_in = x"32" then
239                     reset_sig <= pop_state1;
```

```
241                 elsif ir_in = x"33" then
242                     reset_sig <= pop_state1;
243                     next_state <= ldb_imm_s_state;
244                 elsif ir_in = x"34" then
245                     reset_sig <= pre_dec_state1;
246                     next_state <= fetch_imm1_s;
247                 elsif ir_in = x"35" then
248                     reset_sig <= fetch_2_bytes1;
249                     next_state <= pre_dec_state1;
250                 elsif ir_in = x"36" then
251                     reset_sig <= fetch_imm1;
252                     next_state <= offset_indx_state1;
253                 elsif ir_in(7 downto 4) = "0100" then
254                     if ir_in = x"4C" then
255                         reset_sig <= inc_x1;
256                     elsif ir_in = x"4D" then
257                         reset_sig <= dec_x1;
258                     else
259                         reset_sig <= fetch_imm1;
260                         next_state <= offset_indx_state1;
261                     end if;
262                 elsif ir_in = x"50" then
263                     reset_sig <= fetch_2_bytes1;
264                     next_state <= jmp1;
265                 elsif ir_in = x"51" then
266                     reset_sig <= fetch_2_bytes1;
267                     next_state <= push_pcl;
268                 elsif ir_in = x"52" then
269                     reset_sig <= fetch_pcl;
270                 elsif ir_in = x"53" then
271                     reset_sig <= fetch_2_bytes1;
272                     next_state <= jmp_eq_state;
273                 elsif ir_in = x"54" then
274                     reset_sig <= fetch_2_bytes1;
275                     next_state <= jmp_neq_state;
276                 elsif ir_in = x"55" then
277                     reset_sig <= fetch_2_bytes1;
278                     next_state <= jmp_gt_state;
279                 elsif ir_in = x"56" then
280                     reset_sig <= fetch_2_bytes1;
281                     next_state <= jmp_lt_state;
282                 elsif ir_in = x"57" then
283                     reset_sig <= fetch_2_bytes1;
284                     next_state <= jmp_gte_state;
285                 elsif ir_in = x"58" then
286                     reset_sig <= fetch_2_bytes1;
287                     next_state <= jmp_lte_state;
288                 elsif ir_in = x"00" then
289                     reset_sig <= nop;
290                 elsif ir_in = x"01" then
291                     reset_sig <= add_acc_inh_state;
292                 elsif ir_in = x"02" then
293                     reset_sig <= sub_acc_inh_state;
294                 elsif ir_in = x"03" then
295                     reset_sig <= inc_a_inh_state;
296                 elsif ir_in = x"04" then
297                     reset_sig <= inc_b_inh_state;
298                 elsif ir_in = x"05" then
299                     reset_sig <= dec_a_inh_state;
300                 elsif ir_in = x"06" then
301                     reset_sig <= dec_b_inh_state;
302                 elsif ir_in = x"07" then
303                     reset_sig <= inv_a_inh_state;
304                 elsif ir_in = x"08" then
305                     reset_sig <= inv_b_inh_state;
306                 elsif ir_in = x"09" then
307                     reset_sig <= cmp_state;
308                 elsif ir_in = x"FF" then
309                     reset_sig <= hlt;
310                 end if;
311 
312 
313             --INDEXED instructions
314             when offset_indx_state1 =>
315                 curr_state <= offset_indx_state1;
316                 load_en_out <= "0000000010000"; -- Loading hold low
317                 muxAsel_out <= "0001";           -- choosing din1 which contains offset
318                 muxBsel_out <= "0100";          -- choosing X1 for B of ALU
319                 alu_sel_out <= "01000";         -- A+B+I so that the lowerbyte is incremented and put in the HL
320                 rw_en_out <= "00";
```

```
321                 adx_sel_out <= "0000";
322                 DoutMuxSel_out <= "0000";
323                 reset_sig <= offset_indx_state2;
324                 if ir_in = x"40" then next_state <= lda_imm_state;
325                 elsif ir_in = x"41" then next_state <= ldb_imm_state;
326                 elsif ir_in = x"44" then next_state <= adda_imm_state;
327                 elsif ir_in = x"45" then next_state <= addb_imm_state;
328                 elsif ir_in = x"46" then next_state <= suba_imm_state;
329                 elsif ir_in = x"47" then next_state <= subb_imm_state;
330                 elsif ir_in = x"48" then next_state <= anda_imm_state;
331                 elsif ir_in = x"49" then next_state <= andb_imm_state;
332                 elsif ir_in = x"4A" then next_state <= ora_imm_state;
333                 elsif ir_in = x"4B" then next_state <= orb_imm_state;
334                 elsif ir_in = x"36" then next_state <= pre_dec_state1;
335                 end if;
336             when offset_indx_state2 =>
337                 curr_state <= offset_indx_state2;
338                 alu_sel_out <= "00000";          -- A+B
339                 muxBsel_out <= "0011";           -- XH
340                 muxAsel_out <= "0101";            -- CIN
341                 load_en_out <= "00000000001010";--Load din1 and din2
342                 rw_en_out <= "00";
343                 DoutMuxSel_out <= "0000";
344                 adx_sel_out <= "0000";
345                 if ir_in = x"42" then
346                     reset_sig <= sta_dir_state;
347                 elsif ir_in = x"43" then
348                     reset_sig <= stb_dir_state;
349                 else
350                     reset_sig <= fetch_indx1;
351                 end if;
352             when fetch_indx1 =>
353                 curr_state <= fetch_indx1;
354                 if clp = '1' then
355                     rw_en_out <= "10";              --reading
356                     load_en_out <= "00000000000000";
357                     muxBsel_out <= "0000";
358                     muxAsel_out <= "0000";
359                     alu_sel_out <= "00000";
360                     DoutMuxSel_out <= "0000";
361                     adx_sel_out <= "0100";          --din1:din2 (this is the offseted X)
362                     reset_sig <= fetch_indx2;
363                 end if;
364             when fetch_indx2 =>
365                 curr_state <= fetch_indx2;
366                 if clp = '0' then
367                     rw_en_out <= "10";
368                     load_en_out <= "00000000000001";    --load din1
369                     muxBsel_out <= "0000";
370                     muxAsel_out <= "0000";
371                     alu_sel_out <= "00000";
372                     DoutMuxSel_out <= "0000";
373                     adx_sel_out <= "0100";           --din1:din2 (this is the offseted X)
374                     reset_sig <= next_state;
375                 end if;
376             when inc_x1 =>
377                 curr_state <= inc_x1;
378                 load_en_out <= "00000000010000"; -- Loading hold low
379                 muxAsel_out <= "0000";           -- choosing 0 for A of ALU
380                 muxBsel_out <= "0100";          -- choosing XL for B of ALU
381                 alu_sel_out <= "01001";         -- A+B+1 so that the lowerbyte is incremented and put in the HL
382                 rw_en_out <= "00";
383                 adx_sel_out <= "0000";
384                 DoutMuxSel_out <= "0000";
385                 reset_sig <= inc_x2;
386             when inc_x2 =>
387                 curr_state <= inc_x2;
388                 alu_sel_out <= "00000";          -- A+B
389                 muxBsel_out <= "0011";           -- XH
390                 muxAsel_out <= "0101";            -- CIN
391                 load_en_out <= "00000000001010";--Load din2 and din1
392                 rw_en_out <= "00";
393                 DoutMuxSel_out <= "0000";
394                 adx_sel_out <= "0000";
395                 reset_sig <= ldx_imm_state;
396             when dec_x1 =>
397                 curr_state <= dec_x1;
398                 load_en_out <= "00000000010000"; -- Loading hold low
399                 muxAsel_out <= "1001";           -- choosing XL for A of ALU
400                 muxBsel_out <= "0000";
```

```
401                 alu_sel_out <= "01011";          -- A-B-I so that the lowerbyte is incremented and put in the HL
402                 rw_en_out <= "00";
403                 adx_sel_out <= "0000";
404                 DoutMuxSel_out <= "0000";
405                 reset_sig <= dec_x2;
406             when dec_x2 =>
407                 curr_state <= dec_x2;
408                 alu_sel_out <= "00010";          -- A - B
409                 muxAsel_out <= "1010";            -- Choosing XH
410                 muxBsel_out <= "1010";           -- cinI
411                 load_en_out <= "0000000001010";--Load din1 and din2
412                 rw_en_out <= "00";
413                 DoutMuxSel_out <= "0000";
414                 adx_sel_out <= "0000";
415                 reset_sig <= ldx_imm_state;
416             --jump instructions
417             when jmp_eq_state =>             -- A = B
418                 if CCR_in(7) = '1' then
419                     next_state <= jmp1;
420                 else
421                     next_state <= nop;
422                 end if;
423                 reset_sig <= next_state;
424             when jmp_neq_state =>            -- A != B
425                 if CCR_in(7) = '0' then
426                     next_state <= jmp1;
427                 else
428                     next_state <= nop;
429                 end if;
430                 reset_sig <= next_state;
431             when jmp_gt_state =>             -- A > B
432                 if CCR_in(7) = '0' then
433                     if CCR_in(5) = '0' then
434                         next_state <= jmp1;
435                     else
436                         next_state <= nop;
437                     end if;
438                 end if;
439                 reset_sig <= next_state;
440             when jmp_lt_state =>             -- A < B
441                 if CCR_in(7) = '0' then
442                     if CCR_in(5) = '1' then
443                         next_state <= jmp1;
444                     else
445                         next_state <= nop;
446                     end if;
447                 end if;
448                 reset_sig <= next_state;
449             when jmp_gte_state =>            -- A >= B
450                 if CCR_in(7) = '0' or CCR_in(7) = '1' then
451                     if CCR_in(5) = '0' then
452                         next_state <= jmp1;
453                     else
454                         next_state <= nop;
455                     end if;
456                 end if;
457                 reset_sig <= next_state;
458             when jmp_lte_state =>            -- A <= B
459                 if CCR_in(7) = '0' or CCR_in(7) = '1' then
460                     if CCR_in(5) = '1' then
461                         next_state <= jmp1;
462                     else
463                         next_state <= nop;
464                     end if;
465                 end if;
466                 reset_sig <= next_state;
467             when push_pcl =>
468                 curr_state <= push_pcl;
469                 if clp = '1' then
470                     muxAsel_out <= "0000";
471                     muxBsel_out <= "0000";
472                     alu_sel_out <= "00000";
473                     load_en_out <= "0000000000000";
474                     DoutMuxSel_out <= "0100";     --sending out PCL
475                     rw_en_out <= "01";           --writing
476                     adx_sel_out <= "0111";        --sending out to address 7FFE
477                     reset_sig <= push_pch;
478                 end if;
479             when push_pch =>
```

```
483                 muxBsel_out <= "0000";
484                 alu_sel_out <= "00000";
485                 load_en_out <= "00000000000000";
486                 DoutMuxSel_out <= "0101";     --sending out PCH
487                 rw_en_out <= "01";           --writing
488                 adx_sel_out <= "1000";       --sending out to address 7FFF
489                 reset_sig <= jmp1;
490             end if;
491         when fetch_pcl =>
492             curr_state <= fetch_pcl;
493             if  clp = '0' then
494                 rw_en_out <= "10";  --reading
495                 adx_sel_out <= "1000";   --select 7FFE from adx mux
496                 load_en_out <= "00000000001000";    --not loading anything yet
497                 muxAsel_out <= "0000";
498                 muxBsel_out <= "0000";
499                 DoutMuxSel_out <= "0000";
500                 alu_sel_out <= "00000";
501                 reset_sig <= fetch_pch;
502             end if;
503         when fetch_pch =>  -- put m[7FFE] into din1 and read m[7FFF] for din2
504             curr_state <= fetch_pch;
505             if clp = '0' then
506                 rw_en_out <= "10";  --reading
507                 adx_sel_out <= "0111";   --select 7FFF from adx mux
508                 load_en_out <= "00000000000001";    --now loading m[7FFE] into din1
509                 muxAsel_out <= "0000";
510                 muxBsel_out <= "0000";
511                 DoutMuxSel_out <= "0000";
512                 alu_sel_out <= "00000";
513                 reset_sig <= fetch_ret_sub;
514             end if;
515         when fetch_ret_sub =>  -- put m[7FFE] into din1 and read m[7FFF] for din2
516             curr_state <= fetch_ret_sub;
517             if clp = '0' then
518                 rw_en_out <= "10";  --reading
519                 adx_sel_out <= "0111";   --select 7FFF from adx mux
520                 load_en_out <= "00000000000100";    --now loading m[7FFE] into din1
521                 muxAsel_out <= "0000";
522                 muxBsel_out <= "0000";
523                 DoutMuxSel_out <= "0000";
524                 alu_sel_out <= "00000";
525                 reset_sig <= jmp1;
526             end if;
527         --fetches for instructions
528         when fetch_imm1 =>
529             curr_state <= fetch_imm1;
530             if  clp = '1' then
531                 rw_en_out <= "10";
532                 load_en_out <= "00000000000000";
533                 muxBsel_out <= "0000";
534                 muxAsel_out <= "0000";
535                 alu_sel_out <= "00000";
536                 DoutMuxSel_out <= "0000";
537                 adx_sel_out <= "0011";
538                 reset_sig <= fetch_imm2;
539             end if;
540         when fetch_imm2 =>
541             if clp = '0' then
542                 curr_state <= fetch_imm2;
543                 rw_en_out <= "10";
544                 load_en_out <= "00000000000001";
545                 muxBsel_out <= "0000";
546                 muxAsel_out <= "0000";
547                 alu_sel_out <= "00000";
548                 DoutMuxSel_out <= "0000";
549                 adx_sel_out <= "0011";
550                 reset_sig <= inc_pc_imm1;
551             end if;
552         when inc_pc_imm1 =>
553             curr_state <= inc_pc_imm1;
554             load_en_out <= "00000000010000"; -- Loading hold low
555             muxAsel_out <= "0000";            -- choosing 0 for A of ALU
556             muxBsel_out <= "0110";           -- choosing PCL for B of ALU
557             alu_sel_out <= "01001";          -- A+B+I so that the lowerbyte is incremented and put in the HL
558             rw_en_out <= "00";
559             adx_sel_out <= "0000";
560             DoutMuxSel_out <= "0000";
```

```
561                 reset_sig <= inc_pc_imm2;
562             when inc_pc_imm2 =>
563                 curr_state <= inc_pc_imm2;
564                 alu_sel_out <= "00000";          -- A+B
565                 muxBsel_out <= "0101";           -- PCH
566                 muxAsel_out <= "0101";            -- CIN
567                 load_en_out <= "00000000100000";--Load PC
568                 rw_en_out <= "00";
569                 DoutMuxSel_out <= "0000";
570                 adx_sel_out <= "0000";
571                 reset_sig <= next_state;
572             when fetch_2_bytes1 =>
573                 curr_state <= fetch_2_bytes1;
574                 if  clp = '1' then
575                     rw_en_out <= "10";
576                     load_en_out <= "00000000000000";
577                     muxBsel_out <= "0000";
578                     muxAsel_out <= "0000";
579                     alu_sel_out <= "00000";
580                     DoutMuxSel_out <= "0000";
581                     adx_sel_out <= "0011";
582                     reset_sig <= fetch_2_bytes2;
583                 end if;
584             when fetch_2_bytes2 =>
585                 if clp = '0' then
586                     curr_state <= fetch_2_bytes2;
587                     rw_en_out <= "10";
588                     load_en_out <= "00000000000001";    --Load din1
589                     muxBsel_out <= "0000";
590                     muxAsel_out <= "0000";
591                     alu_sel_out <= "00000";
592                     DoutMuxSel_out <= "0000";
593                     adx_sel_out <= "0011";
594                     reset_sig <= inc_pc_2b_1;
595                 end if;
596             when inc_pc_2b_1 =>
597                 curr_state <= inc_pc_2b_1;
598                 load_en_out <= "00000000010000"; -- Loading hold low
599                 muxAsel_out <= "0000";            -- choosing 0 for A of ALU
600                 muxBsel_out <= "0110";           -- choosing PCL for B of ALU
601                 alu_sel_out <= "01001";          -- A+B+I so that the lowerbyte is incremented and put in the HL
602                 rw_en_out <= "00";
603                 adx_sel_out <= "0000";
604                 DoutMuxSel_out <= "0000";
605                 reset_sig <= inc_pc_2b_2;
606             when inc_pc_2b_2 =>
607                 curr_state <= inc_pc_2b_2;
608                 alu_sel_out <= "00000";          -- A+B
609                 muxBsel_out <= "0101";           -- PCH
610                 muxAsel_out <= "0101";            -- CIN
611                 load_en_out <= "00000000100000";--Load PC
612                 rw_en_out <= "00";
613                 DoutMuxSel_out <= "0000";
614                 adx_sel_out <= "0000";
615                 reset_sig <= fetch_2_bytes3;
616             when fetch_2_bytes3 =>
617                 curr_state <= fetch_2_bytes3;
618                 if  clp = '1' then
619                     rw_en_out <= "10";
620                     load_en_out <= "00000000000000";
621                     muxBsel_out <= "0000";
622                     muxAsel_out <= "0000";
623                     alu_sel_out <= "00000";
624                     DoutMuxSel_out <= "0000";
625                     adx_sel_out <= "0011";
626                     reset_sig <= fetch_2_bytes4;
627                 end if;
628             when fetch_2_bytes4 =>
629                 if clp = '0' then
630                     curr_state <= fetch_2_bytes4;
631                     rw_en_out <= "10";
632                     load_en_out <= "00000000000100";    --Load din2
633                     muxBsel_out <= "0000";
634                     muxAsel_out <= "0000";
635                     alu_sel_out <= "00000";
636                     DoutMuxSel_out <= "0000";
637                     adx_sel_out <= "0011";
638                     reset_sig <= inc_pc_2b_3;
639                 end if;
```

```
641                 curr_state <= inc_pc_2b_3;
642                 load_en_out <= "00000000010000"; -- Loading hold low
643                 muxAsel_out <= "0000";           -- choosing 0 for A of ALU
644                 muxBsel_out <= "0110";          -- choosing PCL for B of ALU
645                 alu_sel_out <= "01001";         -- A+B+1 so that the lowerbyte is incremented and put in the HL
646                 rw_en_out <= "00";
647                 adx_sel_out <= "0000";
648                 DoutMuxSel_out <= "0000";
649                 reset_sig <= inc_pc_2b_4;
650             when inc_pc_2b_4 =>
651                 curr_state <= inc_pc_2b_4;
652                 alu_sel_out <= "00000";         -- A+B
653                 muxBsel_out <= "0101";          -- PCH
654                 muxAsel_out <= "0101";           -- CIN
655                 load_en_out <= "00000000100000";--Load PC
656                 rw_en_out <= "00";
657                 DoutMuxSel_out <= "0000";
658                 adx_sel_out <= "0000";
659                 reset_sig <= next_state;
660             when fetch_dir1 =>
661                 curr_state <= fetch_dir1;
662                 if  clp = '1' then
663                     rw_en_out <= "10";
664                     load_en_out <= "00000000000000";
665                     muxBsel_out <= "0000";
666                     muxAsel_out <= "0000";
667                     alu_sel_out <= "00000";
668                     DoutMuxSel_out <= "0000";
669                     adx_sel_out <= "0100";
670                     reset_sig <= fetch_dir2;
671                 end if;
672             when fetch_dir2 =>
673                 curr_state <= fetch_dir2;
674                 if clp = '0' then
675                     rw_en_out <= "10";
676                     load_en_out <= "00000000000001";
677                     muxBsel_out <= "0000";
678                     muxAsel_out <= "0000";
679                     alu_sel_out <= "00000";
680                     DoutMuxSel_out <= "0000";
681                     adx_sel_out <= "0100";
682                     reset_sig <= next_state;
683                 end if;
684
685             --INSTRUCTION STATES
686             --Immediate
687             when lda_imm_state =>
688                 curr_state <= lda_imm_state;
689                 muxAsel_out <= "0001";       --selecting din1
690                 muxBsel_out <= "0000";      --00
691                 alu_sel_out <= "00000";     --A+B
692                 load_en_out <= "10000000000000";    --Load A
693                 DoutMuxSel_out <= "0000";
694                 rw_en_out <= "00";
695                 adx_sel_out <= "0000";
696                 reset_sig <= nop;
697             when ldb_imm_state =>
698                 curr_state <= ldb_imm_state;
699                 muxAsel_out <= "0001";       --selecting din1
700                 muxBsel_out <= "0000";      --00
701                 alu_sel_out <= "00000";     --A+B
702                 load_en_out <= "01000000000000";    --Load A
703                 DoutMuxSel_out <= "0000";
704                 rw_en_out <= "00";
705                 adx_sel_out <= "0000";
706                 reset_sig <= nop;
707             when adda_imm_state =>
708                 curr_state <= adda_imm_state;
709                 muxAsel_out <= "0001";       -- din1
710                 muxBsel_out <= "0001";      -- A
711                 alu_sel_out <= "00000";     -- A+B
712                 load_en_out <= "10000000000000"; -- Load A
713                 DoutMuxSel_out <= "0000";
714                 rw_en_out <= "00";
715                 adx_sel_out <= "0000";
716                 reset_sig <= nop;
717             when addb_imm_state =>
718                 curr_state <= addb_imm_state;
719                 muxAsel_out <= "0001";       -- din1
```

```
721                 alu_sel_out <= "00000";      -- A+B
722                 load_en_out <= "0100000000000"; -- load B
723                 DoutMuxSel_out <= "0000";
724                 rw_en_out <= "00";
725                 adx_sel_out <= "0000";
726                 reset_sig <= nop;
727             when suba_imm_state =>
728                 curr_state <= suba_imm_state;
729                 muxAsel_out <= "0010";   --A
730                 muxBsel_out <= "1001";  -- din1
731                 alu_sel_out <= "00010"; -- aluA - aluB
732                 load_en_out <= "1000000000000";    --Load A
733                 DoutMuxSel_out <= "0000";
734                 rw_en_out <= "00";
735                 adx_sel_out <= "0000";
736                 reset_sig <= nop;
737             when subb_imm_state =>
738                 curr_state <= subb_imm_state;
739                 muxAsel_out <= "0011";   --B
740                 muxBsel_out <= "1001";  -- din1
741                 alu_sel_out <= "00010"; -- aluA - aluB
742                 load_en_out <= "0100000000000";    --Load B
743                 DoutMuxSel_out <= "0000";
744                 rw_en_out <= "00";
745                 adx_sel_out <= "0000";
746                 reset_sig <= nop;
747             when ldx_imm_state =>
748                 curr_state <= ldx_imm_state;
749                 muxAsel_out <= "0000";
750                 muxBsel_out <= "0000";
751                 alu_sel_out <= "00000";
752                 load_en_out <= "0000010000000";    --Load X
753                 DoutMuxSel_out <= "0000";
754                 rw_en_out <= "00";
755                 adx_sel_out <= "0000";
756                 reset_sig <= nop;
757             when lds_imm_state =>
758                 curr_state <= lds_imm_state;
759                 muxAsel_out <= "0000";
760                 muxBsel_out <= "0000";
761                 alu_sel_out <= "00000";
762                 load_en_out <= "0000000100000";    --Load SP
763                 DoutMuxSel_out <= "0000";
764                 rw_en_out <= "00";
765                 adx_sel_out <= "0000";
766                 reset_sig <= nop;
767             when anda_imm_state =>
768                 curr_state <= anda_imm_state;
769                 muxAsel_out <= "0001";       --din1
770                 muxBsel_out <= "0001";       --A
771                 alu_sel_out <= "00100";      --aluA and aluB
772                 load_en_out <= "1000000000000";        --Load A
773                 DoutMuxSel_out <= "0000";
774                 rw_en_out <= "00";
775                 adx_sel_out <= "0000";
776                 reset_sig <= nop;
777             when andb_imm_state =>
778                 curr_state <= andb_imm_state;
779                 muxAsel_out <= "0001";       --din1
780                 muxBsel_out <= "0010";       --B
781                 alu_sel_out <= "00100";      --aluA and aluB
782                 load_en_out <= "0100000000000";    --Load B
783                 DoutMuxSel_out <= "0000";
784                 rw_en_out <= "00";
785                 adx_sel_out <= "0000";
786                 reset_sig <= nop;
787             when ora_imm_state =>
788                 curr_state <= ora_imm_state;
789                 muxAsel_out <= "0001";       --din1
790                 muxBsel_out <= "0001";       --A
791                 alu_sel_out <= "00101";      -- aluA or aluB
792                 load_en_out <= "1000000000000";        --Load A
793                 DoutMuxSel_out <= "0000";
794                 rw_en_out <= "00";
795                 adx_sel_out <= "0000";
796                 reset_sig <= nop;
797             when orb_imm_state =>
798                 curr_state <= orb_imm_state;
799                 muxAsel_out <= "0001";       --din1
800                 muxBsel_out <= "0010";       --B
```

```
801                 alu_sel_out <= "00101";     --aluA or aluB
802                 load_en_out <= "0100000000000";    --load B
803                 DoutMuxSel_out <= "0000";
804                 rw_en_out <= "00";
805                 adx_sel_out <= "0000";
806                 reset_sig <= nop;
807
808             --Direct States
809             when sta_dir_state =>
810                 curr_state <= sta_dir_state;
811                 if clp = '1' then
812                     muxAsel_out <= "0000";
813                     muxBsel_out <= "0000";
814                     alu_sel_out <= "00000";
815                     load_en_out <= "0000000000000";
816                     DoutMuxSel_out <= "0001";    --sending out A
817                     rw_en_out <= "01";          --writing
818                     adx_sel_out <= "0100";       --sending out din1:din2
819                     reset_sig <= nop;
820                 end if;
821             when stb_dir_state =>
822                 curr_state <= stb_dir_state;
823                 if clp = '1' then
824                     muxAsel_out <= "0000";
825                     muxBsel_out <= "0000";
826                     alu_sel_out <= "00000";
827                     load_en_out <= "0000000000000";
828                     DoutMuxSel_out <= "0010";    --sending out B
829                     rw_en_out <= "01";          --writing
830                     adx_sel_out <= "0100";       --sending out din1:din2
831                     reset_sig <= nop;
832                 end if;
833             when lda_dir_state =>
834                 curr_state <= lda_dir_state;
835                 muxAsel_out <= "0000";
836                 muxBsel_out <= "0000";
837                 alu_sel_out <= "00000";
838                 load_en_out <= "0000000000000";
839                 DoutMuxSel_out <= "0000";
840                 rw_en_out <= "00";
841                 adx_sel_out <= "0100";       --sending out din1:din2
842                 reset_sig <= fetch_dir1;
843                 next_state <= lda_imm_state;
844             when ldb_dir_state =>
845                 curr_state <= lda_dir_state;
846                 muxAsel_out <= "0000";
847                 muxBsel_out <= "0000";
848                 alu_sel_out <= "00000";
849                 load_en_out <= "0000000000000";
850                 DoutMuxSel_out <= "0000";
851                 rw_en_out <= "00";
852                 adx_sel_out <= "0100";       --sending out din1:din2
853                 reset_sig <= fetch_dir1;
854                 next_state <= ldb_imm_state;
855             when adda_dir_state =>
856                 curr_state <= adda_dir_state;
857                 muxAsel_out <= "0000";
858                 muxBsel_out <= "0000";
859                 alu_sel_out <= "00000";
860                 load_en_out <= "0000000000000";
861                 DoutMuxSel_out <= "0000";
862                 rw_en_out <= "00";
863                 adx_sel_out <= "0100";       --sending out din1:din2
864                 reset_sig <= fetch_dir1;
865                 next_state <= adda_imm_state;
866             when addb_dir_state =>
867                 curr_state <= addb_dir_state;
868                 muxAsel_out <= "0000";
869                 muxBsel_out <= "0000";
870                 alu_sel_out <= "00000";
871                 load_en_out <= "0000000000000";
872                 DoutMuxSel_out <= "0000";
873                 rw_en_out <= "00";
874                 adx_sel_out <= "0100";       --sending out din1:din2
875                 reset_sig <= fetch_dir1;
876                 next_state <= addb_imm_state;
877             when suba_dir_state =>
878                 curr_state <= suba_dir_state;
879                 muxAsel_out <= "0000";
```

```
881                 alu_sel_out <= "00000";
882                 load_en_out <= "00000000000000";
883                 DoutMuxSel_out <= "0000";
884                 rw_en_out <= "00";
885                 adx_sel_out <= "0100";       --sending out din1:din2
886                 reset_sig <= fetch_dir1;
887                 next_state <= suba_imm_state;
888             when subb_dir_state =>
889                 curr_state <= subb_dir_state;
890                 muxAsel_out <= "0000";
891                 muxBsel_out <= "0000";
892                 alu_sel_out <= "00000";
893                 load_en_out <= "00000000000000";
894                 DoutMuxSel_out <= "0000";
895                 rw_en_out <= "00";
896                 adx_sel_out <= "0100";       --sending out din1:din2
897                 reset_sig <= fetch_dir1;
898                 next_state <= subb_imm_state;
899             when anda_dir_state =>
900                 curr_state <= anda_dir_state;
901                 muxAsel_out <= "0000";
902                 muxBsel_out <= "0000";
903                 alu_sel_out <= "00000";
904                 load_en_out <= "00000000000000";
905                 DoutMuxSel_out <= "0000";
906                 rw_en_out <= "00";
907                 adx_sel_out <= "0100";       --sending out din1:din2
908                 reset_sig <= fetch_dir1;
909                 next_state <= anda_imm_state;
910             when andb_dir_state =>
911                 curr_state <= andb_dir_state;
912                 muxAsel_out <= "0000";
913                 muxBsel_out <= "0000";
914                 alu_sel_out <= "00000";
915                 load_en_out <= "00000000000000";
916                 DoutMuxSel_out <= "0000";
917                 rw_en_out <= "00";
918                 adx_sel_out <= "0100";       --sending out din1:din2
919                 reset_sig <= fetch_dir1;
920                 next_state <= andb_imm_state;
921             when ora_dir_state =>
922                 curr_state <= ora_dir_state;
923                 muxAsel_out <= "0000";
924                 muxBsel_out <= "0000";
925                 alu_sel_out <= "00000";
926                 load_en_out <= "00000000000000";
927                 DoutMuxSel_out <= "0000";
928                 rw_en_out <= "00";
929                 adx_sel_out <= "0100";       --sending out din1:din2
930                 reset_sig <= fetch_dir1;
931                 next_state <= ora_imm_state;
932             when orb_dir_state =>
933                 curr_state <= orb_dir_state;
934                 muxAsel_out <= "0000";
935                 muxBsel_out <= "0000";
936                 alu_sel_out <= "00000";
937                 load_en_out <= "00000000000000";
938                 DoutMuxSel_out <= "0000";
939                 rw_en_out <= "00";
940                 adx_sel_out <= "0100";       --sending out din1:din2
941                 reset_sig <= fetch_dir1;
942                 next_state <= orb_imm_state;
943
944             --Stack States
945             when pre_dec_state1 =>
946                 if ir_in = x"35" then
947                     next_state <= fetch_dir1_s;
948                 elsif ir_in = x"36" then
949                     next_state <= push_imm_state;
950                 end if;
951                 curr_state <= pre_dec_state1;
952                 load_en_out <= "00000000010000"; -- Loading hold low
953                 muxAsel_out <= "1000";            -- choosing SPL for B of ALU
954                 muxBsel_out <= "0000";           -- choosing 0 for A of ALU
955                 alu_sel_out <= "01011";          -- A-B-1so that the lowerbyte is decremented and put in the HL r
956                 rw_en_out <= "00";
957                 adx_sel_out <= "0000";
958                 DoutMuxSel_out <= "0000";
959                 reset_sig <= pre_dec_state2;
960             when pre_dec_state2 =>
```

```
963                 muxAsel_out <= "0111";           -- SPH
964                 muxBsel_out <= "1010";          -- CIN
965                 load_en_out <= "0000000001010";--Load din1 and din2
966                 rw_en_out <= "00";
967                 DoutMuxSel_out <= "0000";
968                 adx_sel_out <= "0000";
969                 reset_sig <= load_sp_state1;
970             when load_sp_state1 =>
971                 curr_state <= load_sp_state1;
972                 load_en_out <= "0000000100000"; -- Loading SP
973                 muxAsel_out <= "0000";
974                 muxBsel_out <= "0000";
975                 alu_sel_out <= "00000";
976                 rw_en_out <= "00";
977                 adx_sel_out <= "0000";
978                 DoutMuxSel_out <= "0000";
979                 reset_sig <= next_state;
980             when load_sp_state2 =>
981                 curr_state <= load_sp_state2;
982                 load_en_out <= "0000000100000"; -- Loading SP
983                 muxAsel_out <= "0000";
984                 muxBsel_out <= "0000";
985                 alu_sel_out <= "00000";
986                 rw_en_out <= "00";
987                 adx_sel_out <= "0000";
988                 DoutMuxSel_out <= "0000";
989                 reset_sig <= nop;
990             when post_inc_state1 =>
991                 curr_state <= post_inc_state1;
992                 alu_sel_out <= "01001";          -- Add I to low byte for HL reg
993                 muxAsel_out <= "0000";           -- 00 to add
994                 muxBsel_out <= "1000";          -- SPL
995                 load_en_out <= "0000000010000";--Load HL
996                 rw_en_out <= "00";
997                 DoutMuxSel_out <= "0000";
998                 adx_sel_out <= "0000";
999                 reset_sig <= post_inc_state2;
1000            when post_inc_state2 =>
1001                curr_state <= post_inc_state2;
1002                alu_sel_out <= "00000";          -- aluA + aluB
1003                muxAsel_out <= "0101";           -- CIN
1004                muxBsel_out <= "0111";          -- SPH
1005                load_en_out <= "0000000001010";--Load din1 and din2
1006                rw_en_out <= "00";
1007                DoutMuxSel_out <= "0000";
1008                adx_sel_out <= "0000";
1009                reset_sig <= load_sp_state2;
1010            when push_a_state =>
1011                curr_state <= push_a_state;
1012                if clp = '1' then
1013                    muxAsel_out <= "0000";
1014                    muxBsel_out <= "0000";
1015                    alu_sel_out <= "00000";
1016                    load_en_out <= "00000000000000";
1017                    DoutMuxSel_out <= "0001";    --sending out A
1018                    rw_en_out <= "01";          --writing
1019                    adx_sel_out <= "0010";       --sending out SP
1020                    reset_sig <= nop;
1021                end if;
1022            when push_b_state =>
1023                curr_state <= push_b_state;
1024                if clp = '1' then
1025                    muxAsel_out <= "0000";
1026                    muxBsel_out <= "0000";
1027                    alu_sel_out <= "00000";
1028                    load_en_out <= "00000000000000";
1029                    DoutMuxSel_out <= "0010";    --sending out B
1030                    rw_en_out <= "01";          --writing
1031                    adx_sel_out <= "0100";       --sending out SP
1032                    reset_sig <= nop;
1033                end if;
1034            when pop_state1 =>
1035                curr_state <= pop_state1;
1036                if  clp = '0' then
1037                    rw_en_out <= "10";              --reading
1038                    load_en_out <= "00000000000000";
1039                    muxBsel_out <= "0000";
```

```
1041                    alu_sel_out <= "00000";
1042                    DoutMuxSel_out <= "0000";
1043                    adx_sel_out <= "0010";   --sending out SP
1044                    reset_sig <= pop_state2;
1045                end if;
1046            when pop_state2 =>
1047                curr_state <= pop_state2;
1048                if clp = '0' then
1049                    rw_en_out <= "10";              --reading
1050                    load_en_out <= "00000000000001";
1051                    muxBsel_out <= "0000";
1052                    muxAsel_out <= "0000";
1053                    alu_sel_out <= "00000";
1054                    DoutMuxSel_out <= "0000";
1055                    adx_sel_out <= "0010";          --sending out SP
1056                    reset_sig <= next_state;
1057                end if;
1058            when lda_imm_s_state =>
1059                curr_state <= lda_imm_s_state;
1060                muxAsel_out <= "0001";       --selecting din1
1061                muxBsel_out <= "0000";       --00
1062                alu_sel_out <= "00000";      --A+B
1063                load_en_out <= "10000000000000";    --Load A
1064                DoutMuxSel_out <= "0000";
1065                rw_en_out <= "00";
1066                adx_sel_out <= "0000";
1067                reset_sig <= post_inc_state1;
1068            when ldb_imm_s_state =>
1069                curr_state <= ldb_imm_s_state;
1070                muxAsel_out <= "0001";       --selecting din1
1071                muxBsel_out <= "0000";       --00
1072                alu_sel_out <= "00000";      --A+B
1073                load_en_out <= "01000000000000";    --Load B
1074                DoutMuxSel_out <= "0000";
1075                rw_en_out <= "00";
1076                adx_sel_out <= "0000";
1077                reset_sig <= post_inc_state1;
1078            when fetch_imm1_s =>
1079                next_state <= push_imm_state;
1080                curr_state <= fetch_imm1_s;
1081                if  clp = '1' then
1082                    rw_en_out <= "10";
1083                    load_en_out <= "00000000000000";
1084                    muxBsel_out <= "0000";
1085                    muxAsel_out <= "0000";
1086                    alu_sel_out <= "00000";
1087                    DoutMuxSel_out <= "0000";
1088                    adx_sel_out <= "0011";
1089                    reset_sig <= fetch_imm2;
1090                end if;
1091            when push_imm_state =>
1092                curr_state <= push_imm_state;
1093                if clp = '1' then
1094                    muxAsel_out <= "0000";
1095                    muxBsel_out <= "0000";
1096                    alu_sel_out <= "00000";
1097                    load_en_out <= "00000000000000";
1098                    DoutMuxSel_out <= "1000";    --DINI
1099                    rw_en_out <= "01";          --writing
1100                    adx_sel_out <= "0010";       --sending out SP
1101                    reset_sig <= nop;
1102                end if;
1103            when fetch_dir1_s =>
1104                next_state <= push_imm_state;
1105                curr_state <= fetch_dir1_s;
1106                if  clp = '1' then
1107                    rw_en_out <= "10";
1108                    load_en_out <= "00000000000000";
1109                    muxBsel_out <= "0000";
1110                    muxAsel_out <= "0000";
1111                    alu_sel_out <= "00000";
1112                    DoutMuxSel_out <= "0000";
1113                    adx_sel_out <= "0100";
1114                    reset_sig <= fetch_dir2;
1115                end if;
1116
1117            --Inherent states
1118            when nop =>
1119                curr_state <= nop;
```

```
1123                load_en_out <= "0000000000000";
1124                DoutMuxSel_out <= "0000";
1125                rw_en_out <= "00";
1126                adx_sel_out <= "0000";
1127                reset_sig <= fetch1;
1128            when add_acc_inh_state =>
1129                curr_state <= add_acc_inh_state;
1130                muxAsel_out <= "0010";        --A
1131                muxBsel_out <= "0010";       --B
1132                alu_sel_out <= "00000";      --aluA +aluB
1133                load_en_out <= "0100000000000";         --load into B
1134                DoutMuxSel_out <= "0000";
1135                rw_en_out <= "00";
1136                adx_sel_out <= "0000";
1137                reset_sig <= nop;
1138            when sub_acc_inh_state =>
1139                curr_state <= sub_acc_inh_state;
1140                muxAsel_out <= "0011";        --B
1141                muxBsel_out <= "0001";       --A
1142                alu_sel_out <= "00000";      --aluA -aluB
1143                load_en_out <= "0100000000000";         --load into B
1144                DoutMuxSel_out <= "0000";
1145                rw_en_out <= "00";
1146                adx_sel_out <= "0000";
1147                reset_sig <= nop;
1148            when inc_a_inh_state =>
1149                curr_state <= inc_a_inh_state;
1150                muxAsel_out <= "0010";        --A
1151                muxBsel_out <= "0000";       --00
1152                alu_sel_out <= "00001";      --aluA + aluB + 1
1153                load_en_out <= "1000000000000";         --load A
1154                DoutMuxSel_out <= "0000";
1155                rw_en_out <= "00";
1156                adx_sel_out <= "0000";
1157                reset_sig <= nop;
1158            when inc_b_inh_state =>
1159                curr_state <= inc_b_inh_state;
1160                muxAsel_out <= "0011";        --B
1161                muxBsel_out <= "0000";       --00
1162                alu_sel_out <= "00001";      --aluA + aluB + 1
1163                load_en_out <= "0100000000000";         --load B
1164                DoutMuxSel_out <= "0000";
1165                rw_en_out <= "00";
1166                adx_sel_out <= "0000";
1167                reset_sig <= nop;
1168            when dec_a_inh_state =>
1169                curr_state <= dec_a_inh_state;
1170                muxAsel_out <= "0010";        --A
1171                muxBsel_out <= "0000";       --00
1172                alu_sel_out <= "00011";      --aluA - aluB - 1
1173                load_en_out <= "1000000000000";         --load A
1174                DoutMuxSel_out <= "0000";
1175                rw_en_out <= "00";
1176                adx_sel_out <= "0000";
1177                reset_sig <= nop;
1178            when dec_b_inh_state =>
1179                curr_state <= dec_b_inh_state;
1180                muxAsel_out <= "0011";        --B
1181                muxBsel_out <= "0000";       --00
1182                alu_sel_out <= "00011";      --aluA - aluB - 1
1183                load_en_out <= "0100000000000";         --load B
1184                DoutMuxSel_out <= "0000";
1185                rw_en_out <= "00";
1186                adx_sel_out <= "0000";
1187                reset_sig <= nop;
1188            when inv_a_inh_state =>
1189                curr_state <= inv_a_inh_state;
1190                muxAsel_out <= "0010";        --A
1191                muxBsel_out <= "0000";       --00
1192                alu_sel_out <= "00110";      --INV aluA
1193                load_en_out <= "1000000000000";         --load A
1194                DoutMuxSel_out <= "0000";
1195                rw_en_out <= "00";
1196                adx_sel_out <= "0000";
1197                reset_sig <= nop;
1198            when inv_b_inh_state =>
1199                curr_state <= inv_b_inh_state;
```

```
1201                muxBsel_out <= "0000";      --00
1202                alu_sel_out <= "00110";     --INV aluA
1203                load_en_out <= "01000000000000";        --load B
1204                DoutMuxSel_out <= "0000";
1205                rw_en_out <= "00";
1206                adx_sel_out <= "0000";
1207                reset_sig <= nop;
1208            when cmp_state =>
1209                curr_state <= cmp_state;
1210                muxAsel_out <= "0011";       --A
1211                muxBsel_out <= "0001";      --B
1212                alu_sel_out <= "00010";     -- A-B
1213                load_en_out <= "00100000000000";    --store flags in CCR
1214                DoutMuxSel_out <= "0000";
1215                rw_en_out <= "00";
1216                adx_sel_out <= "0000";
1217                reset_sig <= nop;
1218            when hlt =>
1219                curr_state <= hlt;
1220                muxAsel_out <= "0000";
1221                muxBsel_out <= "0000";
1222                alu_sel_out <= "00000";
1223                load_en_out <= "00000000000000";
1224                DoutMuxSel_out <= "0000";
1225                rw_en_out <= "00";
1226                adx_sel_out <= "0000";
1227            when others =>
1228                reset_sig <= hlt;
1229        end case;
1230        end if;
1231     end process;
1232 end architecture ; -- arch
```

Appendix F: Assembler Code

```
# Brice Vadnais Capstone II ECE 493

# Assembler for Custom Microprocessor Architechture

# One thing to note is all labels need to go in brackets
# Still cant do immediate with labels so that needs to get fixed

# THIS WILL BREAK ONCE THERE ARE TOO MANY INSTRUCTIONS

# 16 bits of memory space
mem = [0] * (2 ** 16)

# pc initialized to 0x0000
pc = 0x0000
org_pc = 0x0000
# Encoding Table or ent
ent = {'nop': '0x00',
       'add': '0x01',
       'sub': '0x02',
       'inca': '0x03',
       'incb': '0x04',
       'deca': '0x05',
       'decb': '0x06',
       'inva': '0x07',
       'invb': '0x08',
       'cmp': '0x09',
       'hlt': '0xFF',
       'lda': {'imm': '0x10', 'dir': '0x20', 'indx': '0x40'},
       'ldb': {'imm': '0x11', 'dir': '0x21', 'indx': '0x41'},
       'adda': {'imm': '0x12', 'dir': '0x22', 'indx': '0x44'},
       'addb': {'imm': '0x13', 'dir': '0x23', 'indx': '0x45'},
       'suba': {'imm': '0x14', 'dir': '0x24', 'indx': '0x46'},
       'subb': {'imm': '0x15', 'dir': '0x25', 'indx': '0x47'},
       'ldx': {'imm': '0x16', 'dir': '0x16'},
       'lds': {'imm': '0x17', 'dir': '0x16'},
       # Direct encoding may change in future for this
       'anda': {'imm': '0x18', 'dir': '0x2A', 'indx': '0x48'},
       # Direct encoding may change in future for this
       'andb': {'imm': '0x19', 'dir': '0x2B', 'indx': '0x49'},
       # Direct encoding may change in future for this
       'ora': {'imm': '0x1A', 'dir': '0x2C', 'indx': '0x4A'},
       # Direct encoding may change in future for this
       'orb': {'imm': '0x1B', 'dir': '0x2D', 'indx': '0x4B'},
       'sta': {'imm': '0x26', 'dir': '0x26', 'indx': '0x42'},
       'stb': {'imm': '0x27', 'dir': '0x27', 'indx': '0x43'},
       'psha': '0x30',
       'pshb': '0x31',
       'popa': '0x32',
       'popb': '0x33',
       'psh': {'imm': '0x34', 'dir': '0x35', 'indx': '0x36'},
       'incx': '0x4C',
       'decx': '0x4D',
       'jmp': {'dir': '0x50'},
       'jsr': {'dir': '0x51'},
       'rts': '0x52',
       'jeq': {'dir': '0x53'},
       'jne': {'dir': '0x54'},
       'jgt': {'dir': '0x55'},
       'jlt': {'dir': '0x56'},
       'jgte': {'dir': '0x57'},
       'jlte': {'dir': '0x58'}
       }


# preparing the asm file
asm_file = open('asmtest.asm', 'r')
memory = []
for lines in asm_file:
    if lines.startswith("#") or lines.startswith("\n"):
        continue
    else:
        asm_code = lines.split()
        for op in asm_code:
            if op == "#":
                continue
            elif isinstance(op, str) and op.startswith("\""):
                str_msg = [ord(c) for c in op]
                for i in str_msg:
```

```
                    memory.append(hex(i))
            else:
                memory.append(op)
# print(memory)
labels = []
final_encoded_mem = ""
temp_mem = []
if memory[0] == 'XDEF':
    temp_mem = list(memory)
    # final_encoded_mem = final_encoded_mem + format(int(temp_mem[1], 16), '04X')
    psa_val = int(temp_mem[1], 16)
    memory.remove(memory[0])
    memory.remove(memory[0])

for x in range(psa_val):
    memory.insert(x, '0')


newlabels = []
len_memory = len(memory)
i_items = 0
while i_items < len_memory:
    if memory[i_items].endswith(":") and memory[i_items + 1] == 'EQU' and memory[i_items + 2].startswith("0x"):
        labels.append(memory[i_items])
        labels.append(int(memory[i_items + 2], 16))
        memory.remove(memory[i_items])
        memory.remove(memory[i_items])
        memory.remove(memory[i_items])
        len_memory -= 3
        i_items = 0
    elif memory[i_items].endswith(":"):
        labels.append(memory[i_items])
        labels.append(i_items)
        memory.remove(memory[i_items])
        len_memory -= 1
        i_items = 0
    elif memory[i_items] == 'ORG' and memory[i_items + 1].startswith("0x"):
        memory.remove(memory[i_items])
        memory.remove(memory[i_items])
        i_items = 0
        len_memory -= 2
    i_items += 1


for stuff in labels:
    if isinstance(stuff, int):
        newlabels.append(stuff)
    else:
        newlabels.append(stuff[:-1])

label_dict = dict(newlabels[i:i + 2] for i in range(0, len(newlabels), 2))
# print(memory)
print(label_dict)

# for items in memory:
# fix the address for jumps/ labels, currently counting org as part of the count
i = -1
while i < len(memory) - 1:
    i += 1
    if memory[i] == '0x0D' and memory[i + 1] == '0x0A':
        xx = memory[i]
        while xx is not "0x00" and xx not in ent:
            final_encoded_mem = final_encoded_mem + format(int(memory[i], 16), '02X')
            if not xx.startswith("0x"):
                break
            else:
                i += 1
                xx = memory[i]
    if memory[i] in ent:
        mem[i] = {}
        mem[i]['opc'] = ent[memory[i]]
        if not isinstance(mem[i]['opc'], dict):
            final_encoded_mem = final_encoded_mem + format(int(mem[i]['opc'], 16), '02X')
        if memory[i] == 'hlt':
            mem[i]['opr'] = pc
            final_encoded_mem = final_encoded_mem + format(mem[i]['opr'], '04X')
    elif memory[i].startswith('0x'):
        if mem[i - 1]['opc']['imm'] == '0x16' or mem[i - 1]['opc']['imm'] == '0x17' \
```

```
158             mem[i - 1].update({'opc': mem[i - 1]['opc']['imm'], 'opr': int(memory[i], 16)})
159             final_encoded_mem = final_encoded_mem + format(int(mem[i - 1]['opc'], 16), '02X')
160             final_encoded_mem = final_encoded_mem + format(mem[i - 1]['opr'], '04X')
161         else:
162             mem[i - 1].update({'opc': mem[i - 1]['opc']['imm'], 'opr': int(memory[i], 16)})
163             final_encoded_mem = final_encoded_mem + format(int(mem[i - 1]['opc'], 16), '02X')
164             final_encoded_mem = final_encoded_mem + format(mem[i - 1]['opr'], '02X')
165     elif memory[i].startswith('['):
166         memory[i] = memory[i].strip('[]')
167         if memory[i].startswith('X') or memory[i].startswith('x'):
168             tmpindx = memory[i].split('+')
169             mem[i - 1].update({'opc': mem[i - 1]['opc']['indx'], 'opr': tmpindx[1]})
170             final_encoded_mem = final_encoded_mem + format(int(mem[i - 1]['opc'], 16), '02X')
171             final_encoded_mem = final_encoded_mem + format(int(mem[i - 1]['opr'], 16), '02X')
172         elif memory[i].startswith('0x'):
173             mem[i - 1].update({'opc': mem[i - 1]['opc']['dir'], 'opr': int(memory[i], 16)})
174             final_encoded_mem = final_encoded_mem + format(int(mem[i - 1]['opc'], 16), '02X')
175             final_encoded_mem = final_encoded_mem + format(int(mem[i - 1]['opr'], 16), '04X')
176         elif isinstance(memory[i], str):
177             mem[i - 1].update({'opc': mem[i - 1]['opc']['dir'], 'opr': label_dict[memory[i]]})
178             final_encoded_mem = final_encoded_mem + format(int(mem[i - 1]['opc'], 16), '02X')
179             final_encoded_mem = final_encoded_mem + format(mem[i - 1]['opr'], '04X')
180         else:
181             continue
182     else:
183         continue
184
185 # print(final_encoded_mem)
186
187
188 def split_list(alist, wanted_parts=1):
189     length = len(alist)
190     return [alist[i * length // wanted_parts: (i + 1) * length // wanted_parts]
191             for i in range(wanted_parts)]
192
193
194 # print split_list(A, wanted_parts=2)
195 # print split_list(A, wanted_parts=8)
196
197 n = 2
198
199 new_enc_mem = [int(final_encoded_mem[i:i + n], 16) for i in range(0, len(final_encoded_mem), n)]
200
201 # splitted_lists = split_list(new_enc_mem, wanted_parts=16)
202 # itemsinsplit = 0
203 # for splits in splitted_lists:
204 #     for items in splits:
205 #         print(splits)
206 encodingmem = []
207 encodingmem1 = []
208 encodingmem2 = []
209 encodingmem3 = []
210 count = 0
211 if len(new_enc_mem) < 255:
212     while count < 250:
213         encodingmem.append(new_enc_mem[count])
214         count += 1
215         if len(new_enc_mem) == count:
216             break
217 elif len(new_enc_mem) > 255:
218     while count < 250:
219         encodingmem.append(new_enc_mem[count])
220         count += 1
221         if len(new_enc_mem) == count:
222             break
223     while count < 510:
224         encodingmem1.append(new_enc_mem[count])
225         count += 1
226         if len(new_enc_mem) == count:
227             break
228
229 s19file = open('asmtest.s19', 'w')
230 if len(new_enc_mem) < 250:
231     encodingmem.insert(0, 0)
232     encodingmem.insert(0, 252)
233     encodingmem.insert(0, len(encodingmem) + 1)
234     encodingmem1.insert(0, len(encodingmem1) + 1)
235     tot = 0
```

```
237         tot += opc_opr
238     checksum_val = format(int(hex(0xFFFFFF - int(hex(tot), 16)), 16), '02X')
239     encodingmem.append(int(checksum_val[-2:], 16))
240     with s19file:
241         s19file.write('S1')
242         for i in encodingmem:
243             s19file.write(format(i, '02X'))
244         s19file.write('\n')
245         s19file.write('S105FFFEFC0001')
246 elif len(new_enc_mem) > 250:
247     encodingmem.insert(0, 0)
248     encodingmem.insert(0, 252)
249     encodingmem1.insert(0, 0)
250     encodingmem1.insert(0, 251)
251     encodingmem.insert(0, len(encodingmem) + 1)
252     encodingmem1.insert(0, len(encodingmem1) + 1)
253     tot = 0
254     for opc_opr in encodingmem:
255         tot += opc_opr
256     checksum_val = format(int(hex(0xFFFFFF - int(hex(tot), 16)), 16), '02X')
257     encodingmem.append(int(checksum_val[-2:], 16))
258     tot = 0
259     for opc_opr in encodingmem1:
260         tot += opc_opr
261     checksum_val = format(int(hex(0xFFFFFF - int(hex(tot), 16)), 16), '02X')
262     encodingmem1.append(int(checksum_val[-2:], 16))
263     with s19file:
264         s19file.write('S1')
265         for i in encodingmem:
266             s19file.write(format(i, '02X'))
267         s19file.write('\n')
268         s19file.write('S1')
269         for i in encodingmem1:
270             s19file.write(format(i, '02X'))
271         s19file.write('\n')
272         s19file.write('S105FFFEFC0001')
273 # print(encodingmem)
274 # print(encodingmem1)
275 # new_enc_mem.insert(0, len(new_enc_mem) + 1)
276 # # print(new_enc_mem)
277 # tot = 0
278 # for opc_opr in new_enc_mem:
279 #     tot += opc_opr
280 # checksum_val = format(int(hex(0xFFFFFF - int(hex(tot), 16)), 16), '02X')
281 # # print(checksum_val)
282 # new_enc_mem.append(int(checksum_val[-2:], 16))
283 # print(new_enc_mem)
284
285 # s19file = open('tests19.txt', 'w')
286
287 # with s19file:
288 #     s19file.write('S1')
289 #     for i in en:
290 #         s19file.write(format(i, '02X'))
291 #     s19file.write('\n')
292 #     s19file.write('S105FFFEFC0001')
```

Appendix G: Bootloader Code

```
# Bootloader for Custom CPU
# Brice Vadnais
# Code translated from Professor Hill, Ph.D. -- Boots08 Program
            XDEF 0xFC00

BxCOM:      EQU 0x0010
# Needs to store at this point in memory

RamStAdx:   EQU 0x4000
# Needs to store this as a point in mempry


RecNum:     EQU 0x00FA
RecType:    EQU 0x00FB
RecCkSum:   EQU 0x00FC
RecByteNum: EQU 0x00FD
RecBase:    EQU 0x00FE

            ORG 0xFC00
Start:      lda 0x01
            ldx [BxCOM]
            ldb [X+3]
            cmp
            jeq [StartBoots]
            jmp [RamStAdx]

StartBoots: lds 0x0180

#Start-up Bootloader

            lda 0x30
            sta [RecNum]
            jsr [BxInit]
            ldx 0xFC96
            jsr [BxPutChar]

#Wait for S character
BxFindS:    jsr [BxGetChar]
            ldb 0x53
            cmp
            jne [BxFindS]

#Ignore S0 record type
            jsr [BxGetChar]
            ldb 0x30
            cmp
            jne [BxFindS2]

            jsr [BxNextRC]
            jsr [BxPutChar]
            jmp [BxFindS]

#Detect S1 and S9 record types
```

```
 54 BxFindS2:   ldb 0x31
 55             cmp
 56             jeq [BxRecType]
 57             ldb 0x39
 58             cmp
 59             jeq [BxRecType]
 60
 61 #Non recognized record type
 62             jsr [BxNextRC]
 63             lda 0x3F
 64             jsr [BxPutChar]
 65             jmp [BxFindS]
 66
 67 #Save record type and get record length
 68 BxRecType:  sta [RecType]
 69             jsr [BxGetByte]
 70             sta [RecCkSum]
 71             suba 0x03
 72             sta [RecByteNum]
 73             cmp
 74             jne [BxRecBase]
 75             lda 0x39
 76             sta [RecType]
 77
 78 #Read record base address
 79 BxRecBase:  jsr      [BxGetByte]
 80             sta      [RecBase]
 81             adda      [RecCkSum]
 82             sta      [RecCkSum]
 83
 84             jsr      [BxGetByte]
 85             ldx      [RecBase]
 86             sta      [X+1]
 87             adda      [RecCkSum]
 88             sta      [RecCkSum]
 89
 90 #Read record cargo bytes
 91 BxRecBytes: ldx     [RecBase]
 92
 93 BxRecByte2: ldb      [RecByteNum]
 94             decb
 95             jgte      [BxRecCks]
 96             jsr      [BxGetByte]
 97
 98             sta      [RecBase]
 99             adda      [RecCkSum]
100             sta      [RecCkSum]
101
102             incx
103             jmp      [BxRecByte2]
104
105
106 #Read checksum and verify
```

```
107 BxRecCks:   jsr      [BxGetByte]
108             adda      [RecCkSum]
109             inca
110             jne      [BxRecFail]
111             jsr      [BxNextRC]
112             jsr      [BxPutChar]
113
114 BxRecCkEnd: lda      [RecType]
115             ldb      0x39
116             cmp
117             jne      [BxFindS]
118
119 #Read record done, transfer execution to code
120             lda      0x2A
121             jsr      [BxPutChar]
122             ldx      0x0000
123             jsr      [BxDelay]
124             jmp      [RamStAdx]
125
126 #Report failed CheckSum
127 BxRecFail:  jsr      [BxNextRC]
128             lda      0x58
129             jsr      [BxPutChar]
130             jmp      [BxFindS]
131
132 BxMsgStart: 0x0D 0x0A "Bootloader" "v0.1" "BV" 0x0D 0x0A 0x00
133
134 #Produce report code for record
135 BxNextRC:   lda      [RecNum]
136             inca
137             ldb      0x39
138             cmp
139             jlte      [BxNextRC2]
140             lda      0x31
141             sta      [RecNum]
142             lda      0x0D
143             jsr      [BxPutChar]
144             lda      0x0A
145             jsr      [BxPutChar]
146             lda      0x30
147 BxNextRC2:  rfs
148
149 #BxInit - Initialize devices and data structures
150 BxInit:     lda 0x03
151             ldx [BxCOM]
152             sta [X+1]
153             rfs
154
155 #BxGetByte - Get two characters, convert to byte
156 #Result byte will be in the A register
157 BxGetByte:  jsr    [BxGetNib]
158 #nsa
159             inca
```

```
160             psha
161             jsr    [BxGetNib]
162             popa
163             inca
164             psha
165             rfs
166 #THIS PROBABLY NEEDS TO BE CHECKED
167
168 #BxGetNib - Get a character, covert from hexd.
169 #Value produced will be in the A register
170 BxGetNib:   jsr    [BxGetChar]
171             ldb     0x39
172             cmp
173             jlte    [BxGetNib1]
174             suba    0x07
175 BxGetNib1:  suba    0x30
176             anda    0x0F
177             rfs
178
179
180 #BxGetChar - Get a char, after waiting on RDRF
181 #Character received will be in the A register
182 BxGetChar:  lda 0x05
183             ldb [BxCOM]
184             sub
185             cmp
186             jeq [BxGetChar]
187             lda 0x20
188             sta [BxCOM]
189             ldx [BxCOM]
190             lda  [X+2]
191             rfs
192
193 #BxPutStr - Transmit null-terminated character string
194 #First character is referred to by HX.
195 BxPutStr:   lda      [X+0]
196             jeq      [BxPutSt1]
197             incx
198             jsr      [BxPutChar]
199             jmp      [BxPutStr]
200 BxPutSt1:   rfs
201
202
203 #BxPutByte - Transmit byte as two printable ASCII chars
204 #Character is in the A register
205 BxPutByte:  psha
206 #nsa
207             jsr      [BxPutNib]
208             popa
209
210
211 #BxPutNib - Convert lower nibble to ASCII & transmit
212 #Lower nibble is in A register
```

```
213 BxPutNib:   anda      0x0F
214             ldb 0x09
215             cmp
216             jlte      [BxPutNib1]
217             adda      0x07
218 BxPutNib1:  adda      0x30
219
220
221 #BxPutChar - Send a char, after waiting on TDRE.
222 #Character to send is in A register
223 BxPutChar:  lda 0x06
224             ldb [BxCOM]
225             sub
226             cmp
227             jeq [BxPutChar]
228             ldx [BxCOM]
229             sta [X+2]
230             rfs
231
232
233 #BxDelay - Simple do-nothing delay. Current BxCOM
234 #lacks a transmission complete flag. 12*HX+5
235 BxDelay:    lda 0x01
236             suba 0x01
237             cmp
238             jeq      [BxDDone]
239             nop
240             jmp      [BxDelay]
241 BxDDone:    rfs
```