DESIGNING A RISC-V MICROCONTROLLER FOR FPGA

By

OMAR ABDULWAHAB AL-NANAIH 1740046

FARIS MOAGEB AL-ZAHRANI 1741237

NAWAF FAHAD BARBOUD 1742472

TEAM NO.: 13

FALL-2020/21 INTAKE,

SPRING-2020/21 REGISTRATION

Project Advisor

DR.MOHAMMAD AWEDH

CHECKED AND APPROVED (ADVISOR): 

Project Customer

Dr.Saud Wasly

DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING

FACULTY OF ENGINEERING

KING ABDULAZIZ UNIVERSITY

JEDDAH – SAUDI ARABIA

MAY 2021 G – SHAWWAL 1442 H

DESIGNING A RISC-V MICROCONTROLLER FOR FPGA

By

OMAR ABDULWAHAB AL-NANAIH 1740046

FARIS MOAGEB AL-ZAHRANI 1741237

NAWAF FAHAD BARBOUD 1742472

A senior project report submitted in partial fulfillment of the requirements for the degree of

BACHELOR OF SCIENCE

IN

ELECTRICAL AND COMPUTER ENGINEERING

Approved by (SDP Evaluator)

……

SIGNATURE ....

DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING

FACULTY OF ENGINEERING

KING ABDULAZIZ UNIVERSITY

JEDDAH – SAUDI ARABIA

FALL 2020/21 INTAKE, SPRING-2020/21 REGISTRA

بسم الله الرحمن الرحيم

والصلاة والسلام على خير الورى عدَّ الحصى والرمل والثرى

وعلى آله وأُصحابه ومن تبع هداهم واهتدى

نحمد الله العلي القدير الذي قدَّرنا على إنهاء هذا المشروع في الوقت المحدد

ABSTRACT

DESIGNING A RISC-V MICROCONTRLOOER FOR FPGA

A microcontroller is an integrated circuit that is used to control other electronic devices/components. It contains a microprocessor, memory, other peripherals, and I/O ports. It is widely used in the field of embedded systems, robotics, or any field that requires fast and real-time responses. Developers, students, and startup companies that are interested in developing a microcontroller for a specific use suffer from the downside of license fees looking at most of them have limited resources. Companies that are known in the field of embedded systems such as ARM or Intel requires significant fees for their architecture, tools, and development environment. because of these restrictions, the process of learning for students will be hindered, make research/development more costly for researchers and startup companies. This project aims to develop a fully functional open-source microcontroller that will grant access to any student, researcher, developer, or startup company and give them the ability to make any modifications and adjustments that will fit their specified goal without worrying about the downside of paying any licensing fees. By doing this, students can explore and learn more about the internal design of a microcontroller and allow startups to modify the design to incorporate it into their designs. There are a couple of solutions to approach this problem. The solution that was chosen is the second solution which is to design a microcontroller from scratch using the MyHDL library (library in Python) following the WISHBONE architecture. The decision was made using the weighted decision matrix which helped in making sure the customer objectives and need are to be attained by the end of the project. From that the process of creating the product has started, the first thing was to check the baseline design that was made and follow up on the assumption made about the project. Next the modules of the components were created through trial and error a final working solution was found for each module. When the modules were finish, they were taken to the implementation phase to make sure they were working and undergone every test to make sure they were working, this was done on the most essential component and the final product (CPU, RAM, ROM and final product). From there several tests were conducted on the final product and achieved the needed result that identified that the product is working correctly. Lastly the most unique achievement of the project was the system builder for its need for the users.

ACKNOLEDGMENT

First and foremost, we thank Allah for helping and guiding us to the end of the project and finish implementing. We also thank our families who try and give us encouragement through the difficulties that we have faced through the whole term. We need not forget to give gratitude to each person that helped us reach the finish line of the project. This goes without saying that the guidance and help that we received from our advisor Dr.Mohammed Awedh has been a big push to finish our project. And to Dr.Saud Wasly who is the customer of the project for giving us this great opportunity at a new and exciting project that tested us and helped us become better engineers , and also guided us to understanding his views in the project that helped us in the longevity of the project. Because of this guidance and support from the people who helped us, inshallah our work accepted and guide us to achieve more success.

TABLE OF CONTENT

LIST OF TABLES

LIST OF FIGURES

CHAPTER – 1 INTRODUCTION

1.1 ABOUT THE PROJECT

1.1.1 *Relevant Information*

When developing a microcontroller, certain aspects need to be looked into, such as the microcontroller design and its microarchitecture, the Instruction set architecture (ISA) to be used in the microcontroller design, and what components will this microcontroller contains (CPU, RAM, ROM, etc.). These are all important aspects of making a microcontroller design.

Every microcontroller has a CPU that will perform arithmetic, logic, control, storing and loading, input, and output operations. These operations are based on something called an ISA, which is an abstract of a computer. It's a set of machine language commands and instructions that will get executed by the CPU.

An example of one of the most recent ISAs is the RISC-V ISA (David A. Patterson, May 12, 2017), which is an open-source reduced instruction set architecture, which means that it is made available for free, and can be redistributed and modified to the user's requirements. RISC-V ISA is an instruction set architecture used in modern devices and its valuable and unique because of its reduced set of instructions. In the RISC-V architecture its instructions can be executed in a single clock cycle. And one of the advantages of the RISC-V ISA is that it supports a user's own custom instruction set architecture extensions.

There are many ways of designing and implementing electronic systems, but one of the most well-known methods is using a Hard Description Language (HDL). An HDL is a language used to describe electrical and digital systems. HDLs are useful when designing complex circuits, it can help with debugging, testing, functionality, and the behavior of the circuit. One of the common and well known HDLs is Verilog, it was standardized by the IEEE in 1364. Verilog is mostly used for

hardware modeling, and design verification at the Register Transfer Level (RTL). We will use Verilog which is a hardware description language on a Field Programmable Gate Array.

One of the main issues for students, researchers, or start-ups when it comes to creating their microcontroller design and/or extending a design, large companies that manufacture these microcontroller designs hold ownership of the design. So, when students and developers want to create their design, extend other designs or simply use it for educational purposes, they will need to pay licensing fees to that company first, and an extra fee for the tape-out process as well as the fees for the microarchitecture and warranty fees.

And for start-ups, these licensing fees are even more expensive as they need to develop their design of a microcontroller and print their developed designs in mass. This poses a problem for students, researchers, and startups because even if they pay the licensing fees there are somethings that they cannot change in the design, this means that the design will not be truly theirs. This causes an issue as it will not help the students in increasing their knowledge on microcontrollers neither will the researcher be able to extend the design unless paying the hefty licensing fees.

Lastly, even if the hefty licensing fees are paid, there is still an issue of designing and printing the designing which also cost money, so there is a need for an easy-to-use system builder that will build the whole design of the microcontroller and can simulate it to make sure this is the preferred design without printing it and wasting money on faulty design. the purpose of the system builder is to simulate the design and test it using the user’s own implementation.

1.2 TERMINOLOGY

ISA: Instruction Set Architecture. It's the bridge between hardware and software, a set of instructions in software to be executed by hardware. [1]

FPGA: Field-Programmable Gate Array. A reprogrammable hardware circuit that contains an array of logic blocks that can be configured by the user to do specific tasks. [2]

CPU: Central Processing Unit. An electrical circuit that can perform logic, arithmetic, control operations, and execute instructions (which represents computer programs). [3]

ROM: Read-Only Memory. A non-volatile computer memory that stores instructions for the CPU to execute. [4]

RAM: Random Access Memory. A volatile computer memory that stores data and allows random access to all addresses in the address space. [5]

UART: Universal Asynchronous Receiver/Transmitter. A hardware device capable of transmitting and receiving electrical signals. [6]

I/O: Input/Output. A socket in a computer that cables connect to, allows for communication between the computer and the outside world. [7]

LPT: Parallel port. It's an interface used to transfer electrical signals in parallel. through engineering design. [8]

HDL: in computer engineering, hardware components can be described using what's called "Hardware Description Language". It is used to describe the behavior of electrical and digital circuits. [9]

Verilog HDL: it is one of the most well-known HDLs, it is used to design digital systems. [10]

RTL: register-transfer level, it is a design abstraction of models that describes the data flow from register to another inside digital circuits. [11]

1.3 BACKGROUND

A microcontroller is an integrated circuit (IC) designed by electrical engineers to do take control of electrical systems. Microcontrollers can be considered as small computers on a metal oxide semiconductor (MOS); they contain many different internal components including one or more CPU, ROM, RAM, UART, and many other peripherals to achieve the functionality of a computer and to be used for general purpose applications. Microcontrollers are used to automatically control electrical systems and devices, and they can be found in most big and complex devices like medical appliances, automobile engines, and heavy machinery, to the small and simple devices like an RC car, remote controller, or a keyboard.

How much does Arm Flexible Access cost?

	Entry	Standard
Access fee	**$75k** per annum **$0 for startups***	**$200k** per annum
License fees (due on project manufacture)	**Calculated per project based on IP used**	
Royalties	**Calculated per project and paid per unit shipped**	

*Startups with <$5M funding, <1M annual revenue, privately held

Figure 1 ARM architecture access cost (annually)

Microcontroller manufacturers that are considered reliable and produce high-performance microcontrollers usually ask for a licensing fee for their products (hardware or software). These licensing fees can vary depending on the product of course, but most of the time, small businesses, start-up companies, students, or researchers can't afford them. Big semiconductor and software design companies like ARM provide the latest technologies and designs when it comes to CPUs or different ARM IPs, they offer a wide variety of the fastest, most cost and power efficient products on the market, and they are one of the leading companies when it comes to processor technology. But unfortunately, as we can see in figure 1 the cost of a license for an ARM processor and the price for some additional ARM IPs. As you can see, $200K annually is a lot of money for a student or a researcher, and even some small companies [12].

Further background research was made to make sure on the availability of micro architecture compared to the more known architectures, such as ARM and x86

microcontrollers. Now to state that the intel architecture which are the x86, and Arm Limited offer the ARM architectures. The main difference between both these architectures and the RISC-V architecture, is that RISC-V is open source, and the more known architectures are not.

These architectures that are not open source they ask for fees for the Instruction set Architecture, and fees for the microarchitecture itself and warranty fees, this is a major drawback especially for the start-ups and students/developers to pay these fees because they cannot afford at this level, as seen in figure 1. Different from RISC-V which offers and open-source IP license, that offer its own instruction set architecture, microarchitecture for free and no warranty fees also. [13]

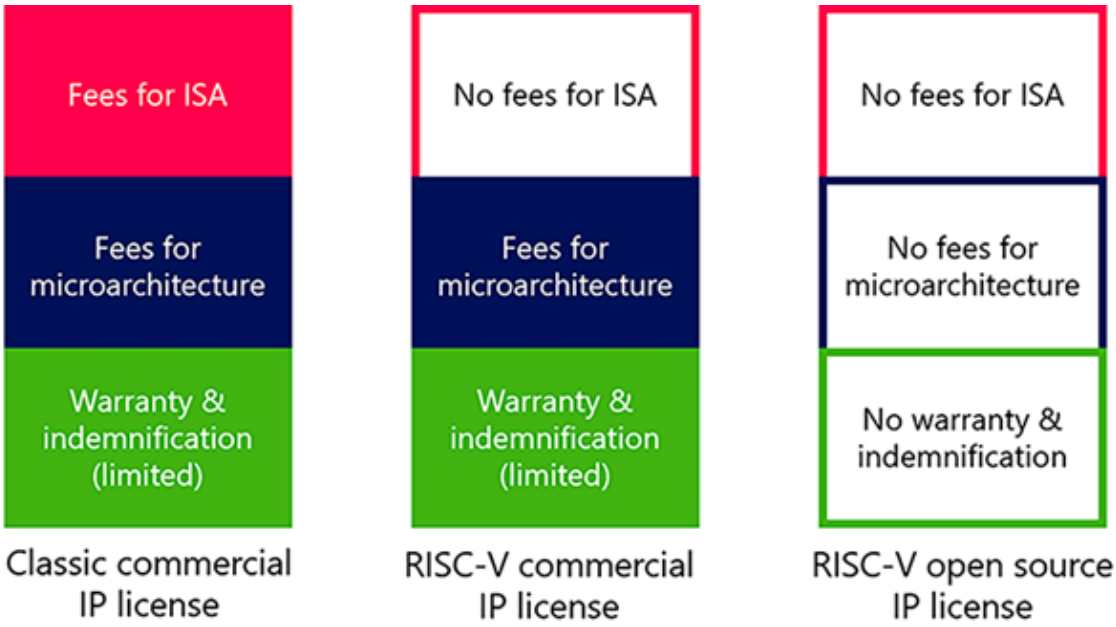


Figure 2 This image shows the difference between classic microarchitectures and RISC-V

In figure 2 it shows the difference between the known commercial architecture (on the left) and the RISC-V open-source architecture that is giving resources it has for free (on the right).

CHAPTER – 2 CONCEPTUAL DESIGN

2.1 SITUATION DESCRIPTION

Nowadays microcontrollers are used in our everyday lives, such as in cars, hospitals, or any electronic devices, that is why the importance of microcontrollers has become more significant in our generation. Most microcontrollers come in small sizes as it is usually used and put in big systems, this allows the microcontroller to control the system, checking its inputs and output, and making sure everything runs smoothly.

A common microcontroller includes several components that are needed to make sure it runs and handles things properly, such as CPU, RAM, ROM, and several I/O ports that are contained inside the microcontroller chip [6]. Now, most of the large companies that develop these microcontrollers sell them to the public for normal use, but the problem arises when an individual such as a student or researcher wants to change the design of the microcontroller or extend it for their research or educational purposes, the company will allow them to extend or change the design but, they will only give them a trial or very limited access to their designs and programs. This is a major problem for start-up companies that want to create their design of a microcontroller for their business, it discourages start-ups. There are only a few open-source microarchitectures that are implemented in microcontrollers for these companies to use without licensing fees, this poses an issue for students that want to learn, researchers that want to extend the design, start-up companies that want to build a design that is theirs. Also, designing the internal architecture of microcontrollers can be difficult for most students and startups since they lack the experience or don’t have the time to design

microcontrollers from scratch. For this reason, some students, which we consider to be future engineers may be discouraged to start designing microcontrollers.

2.2 DEFINING THE PROBLEM

For this section of the report, we will go through the process of achieving a very specific definition of the problem. First of all, an initial problem statement for the problem will be used as our base definition, and then use two different statement-restatement analysis techniques will be applied to generate multiple statements that will help us get a more specific and more defined statement.

2.2.1 STATEMENT-RESTATEMENT

To get a more concise description of the problem, we'll go through two techniques to get a better statement based on the initial problem statement. First of all, we'll start with the Paraphrasing technique. Paraphrasing the initial statement generate new statements with different perspectives. Secondly, Broaden the Focus technique. In this technique, we'll restate the problem definition in a larger context, and this will help get a better view of the problem from a distance. Lastly, Redirecting the Focus technique. In this technique, we'll try and look at solving the initial problem but from a different angle.

Baseline Problem Statement:

- Students, developers, and startup companies can't afford microcontroller architecture design licensing fees.

Paraphrasing:

- Students, developers, and startup companies struggle with microcontroller licensing fees.
- Microcontroller licensing fees are too expensive for students, developers, and startup companies.
- Students, developers, and startup companies are facing the problem of not being able to afford licensing fees to acquire microcontroller designs, tools and development environments.

Broaden the focus:

- Can an open-source microcontroller development environment for students, developers, and startup companies solve the problem of expensive licensing fees?

- Design an open-source microarchitecture with a microcontroller system builder program for students, developers, and startup companies with no licensing fees, ISA and warranty fees.

  Final problem statement:

  Now after we used two different techniques to come up with different problem statements, our final problem statement is:

- Students, developers, and startup companies are facing the problem of not being able to afford licensing fees, ISA and warranty fees to acquire microcontroller designs, tools, and development environments.

2.3 PROJECT OBJECTIVES

The project's objectives are dependent on the customers' technical goals that the he/she has specified. In the table below is the list of high- and low-level objectives

Table 1 project objectives

High Level Objectives	Low Level Objectives
allow developers to use the internal components of the microcontroller (CPU, memory, serial port… etc.) in their separate designs	design and open-source fully functional microcontroller
Make the process of customizing a microcontroller design simpler	make an optimized microarchitecture design that can achieve high performance
to Implement a method to make the functional simulation a lot easier and reduce the time consumed by designers in this step	the microcontroller design to be implemented on an FPGA (Field Programmable Gate-Array)

these objectives are that the projects use the RISC-V architecture, to design and implement a CPU microarchitecture that can fulfill the subset of the instructions, also to design a bus system to add peripherals to the system, design the I/O ports and UART for the system. In abiding by these customer objectives, we developed the higher and lower objectives from the must and wants of the customer. In developing this project, we can achieve the customers' goals of creating a KAU-branded RISC-V based microcontroller that contains some peripherals which will make it function as a microcontroller system, as the project is aimed to be open-source in the future it will help student, researcher, and start-up that can use this design and implement their components as much as they want.

2.4 APPLICABLE ENGINEERING STANDARDS

All applicable engineering standards are to be outlined and their application in the project discussed. It is important to consider the relative engineering standards (technical, safety etc.) in defining all the constraints in the Product Design Specifications, especially the musts.

In this project we will follow three engineering standers that covers performance requirements, verification and test procedure. For performance requirements, we will follow the “Specification for the WISHBONE System-on-Chip (SoC) Interconnection Architecture” [14] WISHBONE architecture has a set of rules that forms the basic framework of its specifications. These rules ensure the success of our microarchitecture design and will allow us to achieve all the objectives of the WISHBONE architecture. We as designers will make sure to follow these rules when we start designing the components of the system, for example making sure that each component has all the needed signals required to achieve good 2 communication between different components. The architecture also offers a set of recommendations to increase the overall performance of the system, as these specifications are designed to achieve high performance it is still possible to have poor performance depending on the experience of the designers.

for the verification standers, we followed “IEEE Std 1012™-2016 IEEE Standard for System, Software, and Hardware Verification and Validation” [15]. One of the implementations we will use in this standard for our project is Requirement Evaluation in this section we need to evaluate the requirements of the project for its correctness, consistency, accuracy, readability, and testability. In each step of the project development, we must verify that the software requirement satisfies the systems requirements. As well we implement the hazard analysis to determine what software requirements that contribute to each system hazard. In this standard, we will implement the requirement evaluation techniques that will lead us in the correct direction of verifying our design, one of the points is the Completeness (9.2 Activity) which will be implemented, this is where we verify the functionality of the requirements and the performance of the project to make sure it is in line with the projects needs and the users need to implement the project correctly. Also, another example of one of the verification standards that will be implemented is the software integration (9.5 Activity), to perform the integration, we need to verify that these

software components are integrated in the correct way, so we need to analyze their test results. for the test procedure stander, we followed “IEEE Std 1364™-2005 IEEE Standard for Verilog® Hardware Description Language” [16]. As our project includes designing different electrical components the microcontroller using the hardware description language Verilog, we are going to be using different procedures in this standard to test and analyze different aspects of the finished design. These procedures include procedural statements that governs or controls the simulation to manipulate different variables, and these statements differ for different procedures, for example we have blocking and non-blocking procedural assignment, continuous procedural assignments, force and release, timing controls and many more. All these procedures will make sure our finished product is done properly to ensure that we create a good microarchitecture design.

2.5 REALISTIC CONSTRAINTS

Table 2 Project Constraints

Project Constraints	
Constraint #1	The project’s budget must not exceed 3000SR
Constraint #2	Frequency of the system check must be at least 10MHz
Constraint #3	The project must be completed before April 2021

2.6 PRODUCT DESIGN SPECIFICATIONS (PDS)

As we’ve stated earlier, we are working on finding a solution for the problem a lot of students, developers and startup companies are facing, and for this, we are designing a fully functional microcontroller, and testing the final design on an FPGA. And after that, we’ll also be making a system builder program that will help everyone create their design for their microcontroller. We’ll first go through the in-scope and out-of-scope specifications for this project. After that, we’ll look into any assumptions about this project, risks that we may face during the process, and how we’ll handle them.

2.6.1 MUST AND WANTS

In the next table (1), we can see a summarization of the compulsory in-scope specifications (musts) and out-of-scope specifications (wants) for this project. The

must are the items that are in tune with the objectives of the project as it affects the final design of the project, as for the want they have desired outcome from the design.

Table 3 musts and wants

Musts	All the files, documents, and designs must be published as open-source
	The bus system must follow the Wishbone bus architecture.
	Follow the Google developer documentation approach.
	The CPU must be able to execute all RV32I, RV32M RISC-V instructions.
Wants	To be improved and developed further by future students
	CPU can execute RV32A, RV32F, RV32D RISC-V instructions
	The design is easily extendable.
	A user-friendly interface for the System Builder program

2.6.2 ASSUMPTIONS

- RISC-V reduced instruction set will have all the necessary functionalities of other large ISAs'.
- The MyHDL library in python can be used to convert code to Verilog code.
- Good pre-existing knowledge about RISC-V instruction set architecture.
- The tools used such as the design synthesis tool for FPGA are free of charge.

2.6.3 RISKS and REMEDIES

- **RISK #1**: The design of the microcontroller cannot reach the needed performance

  REMEDY #1: The microarchitecture of the system is optimized
- **RISK #2:** MyHDL does not generate proper Verilog code

  REMEDY #2: To generate proper Verilog code, a custom MyHDL can be developed to generate the code properly
- **RISK #3:** The system experiences Propagation delay or time skew issues between the component of the system

  REMEDY #3: To try and fix the delay, the system could run multiple timing simulations to find the core of the delay.
- **RISK #4**: Different components in the system are slower than the CPU.

REMEDY #4: The CPU could enter a wait state or function at lower frequencies.

2.7 LITERATURE REVIEW

Microcontrollers became a significant creation to the world when it was first invented between the years 1970 and 1971. Intel is now the world's biggest and most valued semiconductor chip manufacturer, and they continue to develop the most high-tech chips. An engineer called Gary Boone built the first microcontroller, which was a single integrated circuit chip which can hold all the essential circuits for a calculator, but without the display and keypad. Since then, the sales and the number of microcontrollers has increased and reached a point that in today's world it is found in our home and in the most common technology that we use today.

FPGA Implementation of 8-bit RISC Microcontroller for Embedded Systems [14]**:**

This paper documents the prototype 8-bit RISC Microcontroller, which has a 16-bit address bus, it contains a decoder, ALU and an interrupt control unit with some added peripherals. This project wants to try and use RISC to reduce the number of instructions and make all the clock cycled unified for each instruction that will be executed. in this design, it creates a controller that contains 8-bits for the data bus and 6-bits for the internal address bus, this system for the controller architecture consists of a core, peripherals, programmer, memory driver, and external memory units.

This microcontroller design (Figure 3) contains a peripheral unit as seen in the controller architecture block diagram, it consists of a USART, a 16-bit timer-counter, and I/O Ports. All these peripherals are connected to the internal main data bus. There is also the core which is one of the significant units of the microcontroller, this is where the instructions are decoded and then executed, almost every component is driven by this unit.

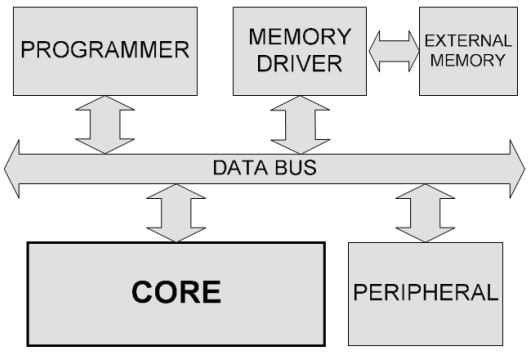


Figure 3 AGH University microcontroller block diagram [14]

As seen in the diagram above, the core is one of the most significant units in all microcontrollers. The core (CPU) is where all instructions are decoded then executed. one of the components inside the Core is the General Purposes Register, it Is made up of 32 8-bit registers, they act as a source for ALU operations, by doing this it is deemed the accumulator needless which in turn reduced the number of data being transmitted with the Random-Access Memory. The transfer functions such as the read and writes to the peripherals, are realized by the GPRU unit. The ALU (Figure 4) is where all the data modification operations are executed it has four main inner components: the adder/subtractor. The logical unit, bit unit, and multiplier.

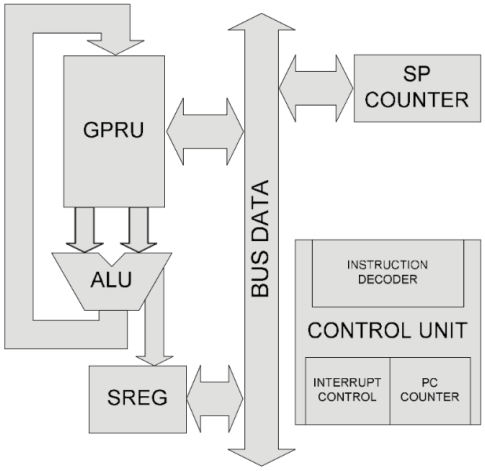


Figure 4 AGH University microcontroller core [13]

The project that was seen in this paper, where they developed the necessary components to perform an FPGA implementation of an 8-bit RISC microcontroller, this project has the common them to our projects which is to create a RISC-V Based microcontroller using FPGA. They both share the same views in developing an industrial microcontroller that will be of use in the future, as you see both projects use RISC-V to be implemented using FPGA, because RISC-V is open-source ISA and has no licensing fees to pay, and because it reduces the number of needed instructions as it is one of the ways to improve performance.

The main differences between our projects and the one in the paper are, firstly the microcontroller in this project will be a 32-bit implementation of the microcontroller which will improve the CPU performance and have a better processing speed than the 8-bit microcontroller. And we will use the WISHBONE architecture as our bus system which will be 32 bits as opposed to that project's 16 bits. All these differences make the project much more advanced and better suited for future innovation and/or implementation and the fact that every part of the project is open-source will help make the project better.

FPGA Implementation of 32-bit RISC-V Processor with Web-Based Assembler-Disassembler [15]**:**

This paper describes the implementation of a 32-RISC-V microprocessor. This is design using Verilog HDL and implemented on one of the FPGA's such as the DE12-115 board. And a web-based assembler-disassembler is created and was published for this project. The implementation starts with the user to generate machine code, that is downloaded onto the RISC-V processor with the use of Universal Asynchronous Receiver Transmitter. The tools are developed on a website. The processor is completely functional that implements the RV32I base integer instruction set.

In this design, they built a single cycle design, so various operations are running in a single cycle at the same time. There is a control signal which shall determine the progression that will produce the output. Then the instruction will be fetched from the instruction memory. The whole system process will start in the decoding stage and control stage.

As for the microarchitecture of the system, it consists of fetching, decoding, and control stages. In the fetch stage, it determines the value of the current PC and predicts the next PC. This PC value is kept in the 32-bit registers and then it

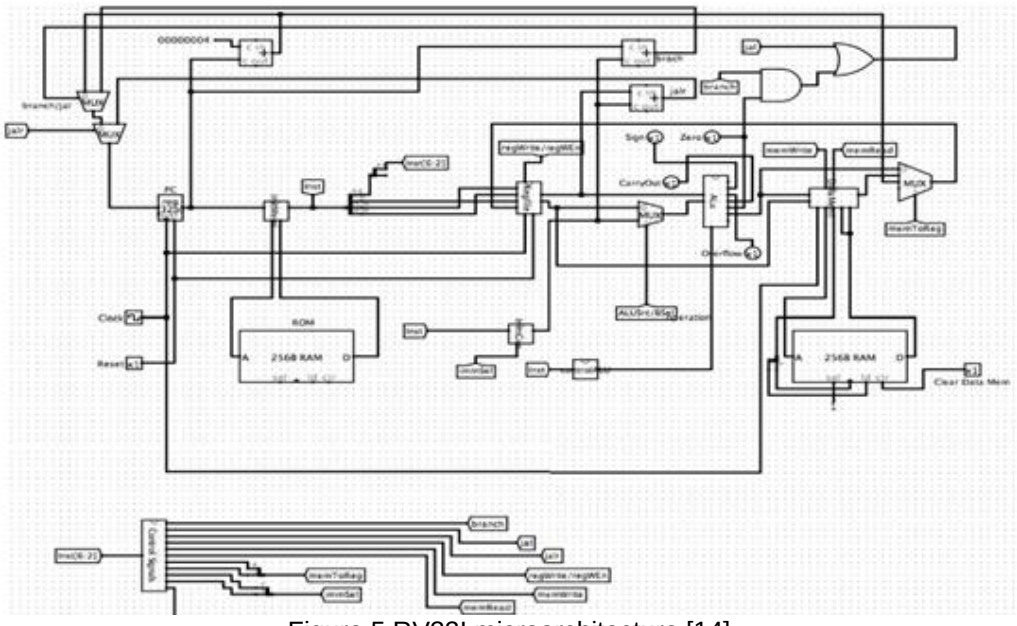


Figure 5 RV32I microarchitecture [14]

calculates the PC's next value (PC+4). Then in the decoding stage, the instructions taken from instruction memory are resolved, this will then determine what signals

will be sent to the control stage. Depending on this design the are 12 outputs from the control stage. Below is the diagram of the microarchitecture of the RV32I ISA.
The study that was made to develop the Web-based Assembler-Disassembler, to help implement the RV32I microarchitecture design using an FPGA. This project has some similarities to our project, as it implements the RV32I instructions but our project implements more of these instructions, such as RV32M instructions. This design is very viable, as it implements the processor microarchitecture with few modules, and as opposed to our projects it will implement a microcontroller design as a whole with several modules such as the UART module and the PWM module. The design of our project is more complex as it adds more features and modules. Also, our design will include a system builder that will implement the custom design of the user's microcontroller design.

2.8 THEORETICAL BACKGROUND AND ANALYSIS

To find the best alternative design, we will use some methods to analyze the alternatives theoretically, financially, and finally we will apply a technique called the weighted decision matrix.

Theoretical analysis:

in our design, we are aiming to find a solution that will implement a bus technology that allow an easy intercommunication between the peripherals and the master of the bus. Additionally, since our project have a lot of part that requires a significant time on development, so if the suggested solution use a certain development technique/tool it will be a plus for that suggested alternative.

cost analysis:

in the cost analysis, we will list all the essential parts and the estimated labor hours for each alternative solution. Finally, we will compare all the suggested alternatives using the total price (include the essential parts with addition to the labor hours) and then chose the cheapest alternative.

weighted decision matrix:

The Weighted Decision Matrix which is a quantitative technique that puts different solutions against several criteria the objective of the project depends on. in order to implement this technique, we are going to set some criteria in order to measure each

alternative. The measures are Performance, Customizable, Cost, Ease of Development and Portability.

2.9 ANALYZING ALTERNATIVE SOLUTIONS

2.9.1 SOLUTION 1

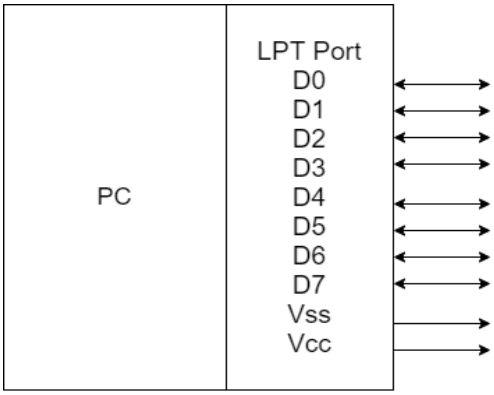


Figure 6 solution 1 block diagram

Since most of our projects will be done in software, and the only component we might be needing is an FPGA, we thought maybe we're better off if we continued and done everything in software and got rid of all hardware components. So, for the first solution, we aim to design an open-source program that allows the user to design and emulate the functionality of a microcontroller using the user's PC to communicate with the external components by using the parallel port (LPT port) shown in Figure 6. The microcontroller will be based on RISC-V Instruction set architecture. All the internal components of a microcontroller will be designed and implemented in software, and this can all be done in a program we'll create using the hardware description language library MyHDL in Python. Not only this, but a system builder will also be built to make things even easier for the inexperienced user. The system builder program will help them by making the process of designing a custom controller much easier by just dragging and dropping the components you need using the program, and it will automatically provide the code for the whole system.

2.9.1.1 COMPONENTS

Hardware components:

- LPT Port

Software components:

- RISC-V ISA
- CPU

- RAM
- ROM
- Bus
- I/O Ports
- Timer

2.9.1.2 WORKING OF THE SYSTEM

This solution aims at designing a virtual microcontroller that acts and functions as a microcontroller. The virtual microcontroller will be implemented based on a RISC-V ISA, and the user will be able to use the LPT port to communicate with the external components. The in-scope items of the system work with this solution as it can follow the google developer approach in documentation as well as follow the wishbone architecture, and can execute all RISC-V RV32I, RV32M, instructions. But as for the out-of-scope items or wants, the system may be impractical to be developed further for it is not easy to be carried around and may not be used in real projects.

2.9.1.3 PROS AND CONS

Table 4 solution 2 pros and cons

Pros	Cons
Everything is done in software	Not portable
Allow more Optimization for specific applications	Bad performance
-	No FPGA implementation

2.9.1.4 COST ANALYSIS

This table (3) presents the Cost of the components that make up the project itself:

Table 5 solution 1 cost analysis

Item	Quantity	Cost (SAR)
Computer	1	1200
LPT Cable	1	25
Labor	-	3*(2SAR/h*6h*5d*8w) = 1440
	Total Cost	2665

2.9.2 SOLUTION 2

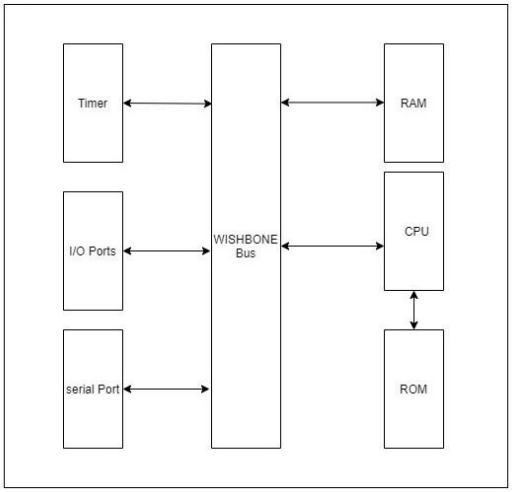


Figure 7 solution 2 block diagram

For the second solution, we aim to create our design from scratch, we will be designing a microcontroller using the open-source RISC-V Instruction Set Architecture (Figure 5), this will include but not limited to: central processing unit, memory, I/O ports, and some other internal peripherals like timers for example. By designing everything from scratch, we're not bound to use certain programming languages, we have the freedom to use Python, C, or C++. Thus, we will be using a Python library called MyHDL [16] which allows us to write low-level code in a high-level language (python). using MyHDL will provide us with the benefits of easier functional simulation and will reduce the time and effort needed to achieve the desired design. Along with the design of the components, we will create a system builder that will help with the process of creating a custom microcontroller. Therefore, the user will have the ability to replace one of the components in our design with his own without going through the trouble of modifying the rest of the design. Finally, we chose the WISHBONE bus architecture to be the architecture used when building our bus system. This architecture is open-source which is important looking at our objectives. Additionally, in terms of performance, the WISHBONE bus tends to gain an upper edge over other competitors because it provides for connecting circuit functions together in a way that is simple, flexible, and portable due to its synchronous design.

2.9.2.1 COMPONENTS

Hardware components:

- FPGA DE10_LITE

Software components:

- RISC-V ISA
- CPU
- RAM

- ROM
- Bus
- I/O Ports
- Timer

2.9.2.2 WORKING OF THE SYSTEM

This solution aims to design a microcontroller using the RISC-V reduced instruction set, it will contain a microcontroller main component that a basic microcontroller has such as CPU, RAM, ROM, timers, etc. All these components are connected using a memory-mapped bus system and will follow the Wishbone bus architecture. Also, along with the microcontroller design, this project aims to create a system builder that helps the users when trying to create a custom microcontroller. This solution satisfies all in-scope items that the customers specified, as it will follow the Wishbone Bus architecture, able to execute all the RV32I, RV32M RISC-V instructions, every file, document, and design can be published as open-source and follows Google Developer documentation approach. As for the wants for out-of-scope items the solution, it has all the capabilities to achieve the wants of the system design.

2.9.2.3 PROS AND CONS

Table 6 solution 2 pros and cons

Pros	Cons
Hardware implementation (FPGA DE10_LITE)	It takes more time to develop since we're starting from scratch
High Performance	-
More Optimized	-
Can be done in multiple programming languages	-
Functional simulation can be done easily	-

2.9.2.4 Cost Analysis

This table (5) presents the Cost of the components that make up the project itself:

Table 7 solution 2 cost analysis

Item	Quantity	Cost (SAR)
FPGA DE10_LITE	1	260
Labor	-	3*(2SAR/h*6h*5d*12w) = 2160
	Total Cost	2420

2.9.3 SOLUTION 3

For the third solution, we'll take an already established open-source microprocessor architecture and add other components (memory, I/O ports, etc.) that we build using the library MyHDL to any hardware description language to create a microcontroller that meets the desired objectives and specifications. since the customer requested that the microcontroller to be based on RISC-V ISA, we will use a microprocessor from the "OpenRISC Project". Additionally, we will design the rest of the microcontroller components which include but are not limited to memory, I/O ports, timers, etc.

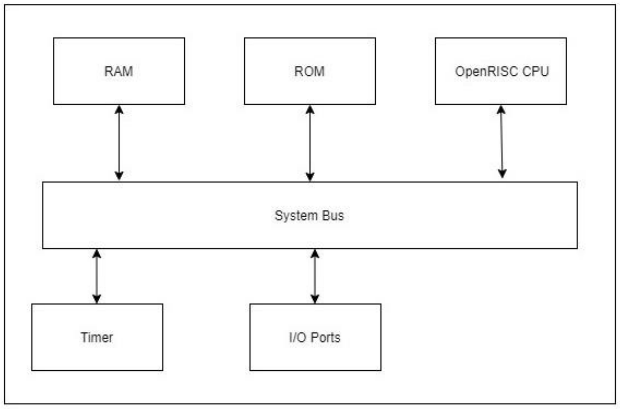


Figure 8 solution 3 block diagram

2.9.3.1 COMPONENTS

Hardware components:

- FPGA DE10_LITE

Software components:

- RISC-V ISA
- OpenRISC CPU
- RAM
- ROM
- Bus

- I/O Ports
- Timer

2.9.3.2 WORKING OF THE SYSTEM

For this solution, we are going to be using an already established CPU architecture as our base design for the CPU. This can be done using the multiple designs provided by the OpenRISC [17] community. This solution works exactly similar to solution 2 but the main difference is the CPU, which is the most complicated component in the system. Of course, this design of the microcontroller will have all the main components such as the main memory (RAM), a program memory (ROM), Timer, and more. This solution will meet all the in-scope items, and the out-of-scope items as well.

2.9.3.3 PROS AND CONS

Table 8 solution 3 pros and cons

Pros	Cons
Hardware Implementation (FPGA DE10_LITE)	Not Optimized
High performance	Functional simulation is not easy
Ready to use components	Only supports C and C++
-	Requires more time due to lack of knowledge in OpenRISC

2.9.3.4 COST ANALYSIS

This table (7) presents the Cost of the components that make up the project itself:

Table 9 solution 3 cost analysis

Item	Quantity	Cost (SAR)
FPGA DE10_LITE	1	260
Labor Cost	-	3*(2SAR/h*6h*5d*8w) = 1440
	Total Cost	1700

2.9.4 OPTIMAL ALTERNATIVE DESIGN

Approaching solution number 1 it is clear that it only satisfies one of the three lower-level objectives, this is a big sign that this solution is not the solution we're looking for. Also, when analyzing the cost of this solution, we found that it depends

on the price of the computer, which can differ from user to user, but we chose the cheapest computer to start with. Furthermore, solution number 1 will out-perform every other solution in the performance criteria since it will use the computers' CPU developed by big known companies like AMD or Intel.

As for the second and third solutions, these two solutions satisfy all objectives and costs at a reasonable price at the same time. But, the difference between these two comes down to the pros and cons of each solution. As we've shown previously, the third solution has too many cons and it's more than its pros. But the reason we're not going with this solution is that it uses only two languages, C and C++. This alone will affect the completion time of this solution since we're unfamiliar with these languages, and it will also affect the process of functional simulations.

We're currently left off with the second solution, and as we declared previously, it satisfies all the lower/ higher-level objectives, it solely has one con and 5 pros, stays inside the constraints and therefore the total price of the solution is reasonable. For these reasons, we concluded that the most effective solution to our problem would be the second solution. Finally, we'll discuss a more detailed way of deciding the optimal solution using the well-known Weighted Decision matrix as it a quantitative way of evaluating the alternative solutions.

2.9.4.1 ALTERNATIVE SOLUTION EVALUATION

We needed to find a method to choose the best solution from several solutions available. We used a Weighted Decision Matrix [18] which is a quantitative technique that puts different solutions against several criteria the objective of the project depends on.

Table 10 weighted decision matrix

		Solution 1		Solution 2		Solution 3	
Criteria	Weights	Score (1-5)	Weighted score	Score (1-5)	Weighted score	Score (1-5)	Weighted score
Performance	4	3	12	5	20	4	16
Customizable	5	5	25	5	25	4	20
Cost	3	3	9	3	9	4	12
Ease of Development	3	5	15	5	15	3	9
Portability	2	1	2	5	10	5	10
		Total	63	Total	79	Total	67

Considering table 8 of the weighted decision matrix, the alternative solutions were evaluated based on certain criteria that relate to the project's objective (higher and lower) and for each criterion, a weight is given. After analyzing the Weighted decision matrix, we found that alternative solution number 2, that has the highest score out of all alternative solutions. It is clear that alternative solution 2 complies with all the objectives of the project and is contains all the in-scope items as well as can achieve all the out-of-scope items.

2.9.4.2 JUSTIFICATION

First of all, one of the most important aspects of the project is that it is an open-source project, so considering the first in-scope specification which is to publish all the design as open-source. Second, the second solution can implement any bus system, so it can follow the wishbone bus architecture which checks off the second in-scope item (musts).

This design also can follow the Google developer documentation approach, this is for other developers or people who are interested in the design so they can understand the code written. Lastly, the design is created so that it can have a CPU that will be able to execute all the RV32I, RV32M RISC-V instructions.

This satisfies all the in-scope items generated from the customers' requirements. Now looking at the out-of-scope items, this is the wants that the project can achieve if needed but is not a must. Considering the wants shown in the table in section 2.4, the project will be well documented and has a level of flexibility that will make the

future improvement easier, by that we achieved the first out-scope specification. Additionally, as mentioned before, this project has a system builder and that makes it easily extendable (achieve the second point of the out-scope specification). Finally, the system builder that we aim to design will have a friendly UI (User Interface), and by doing that we will fulfill the last out-scope specification.

2.10 MATURING BASELINE DESIGN

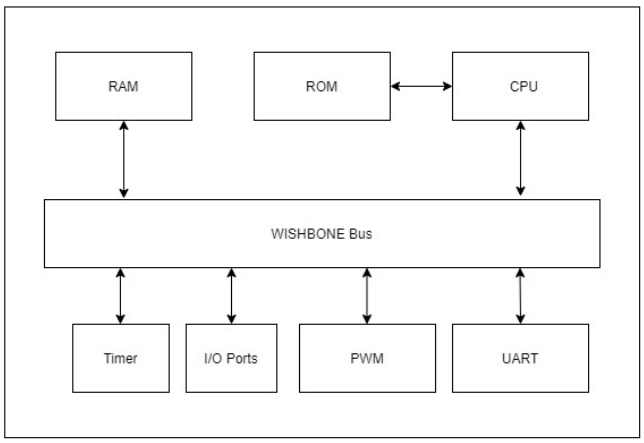


Figure 9 baseline design block diagram

When creating the Weighted Decision Matrix, we gave each criterion a weight and for each solution a score for that criteria, after this step, we multiply the criteria's weight by the solution's score in that criteria. This step is done for every criterion and lastly, the weights multiplied by the score to give a total. The lowest total in our set of solutions was Solution 1. Solution 3 has the second-best total out of the three alternative solutions. Lastly based on the Total score that was calculated using the Weighted Decision Matrix, Solution 2 has the best score and is the chosen Alternative solution based on the Total Score.

Based on the evaluation analysis, solution 2 was the best and optimal solution alternative and is our baseline design. Solution 2 is a Design of a microcontroller using the open-source RISC-V Instruction Set Architecture along with our system builder, and design a bus system following the wishbone bus architecture.

Solution 2 was chosen over the rest of the alternatives based on the Objective Analytical Evaluation method. Using Solution 2 it can deliver an optimized microarchitecture design that will achieve high performance, as well as it can be implemented on an FPGA, and because RISC-V is an open-source ISA we can achieve the objective of designing a functional open-source microcontroller design. With the extent that solution 2 gives the developers the option of completely designing their separate components of the microcontroller, and with the design of

the system builder it will make the process of customizing the design easier and that will lead to an optimized design for the developers' needs. Opposite of the other solution where you do not have full control of the total design of the components.

We made two modifications to the baseline solution; we replaced the additional serial port with a UART since it uses serial communication, and added a PWM module. By doing that, we improved our baseline solution since most base microcontrollers developed by big companies (Arduino UNO, etc.) now have at least one UART and PWM modules.

CHAPTER – 3 PRODUCT BASELINE DESIGN

3.1 BLOCK DIAGRAM

As we've mentioned previously, we chose Solution 1 to be our optimum solution. In figure 10 we can see the block diagram of the baseline design after doing some minor adjustments (More in section 2.7). the microcontroller will have eight different components, CPU to do all the logical and arithmetic operations, ROM to store instructions, RAM to store temporary data, UART [19] for asynchronous data transmission and reception, PWM to send out pulse-modulated signals, Timer, I/O Ports as a general synchronous input/output ports and a WISHBONE Bus system to connect all the components. More about each component in the next section.

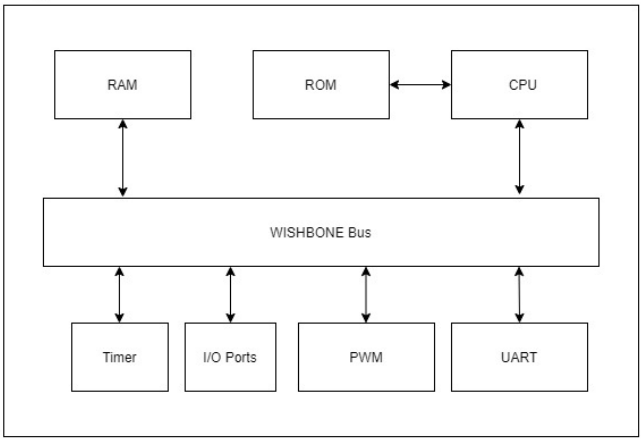


Figure 10 final baseline design block diagram

3.2 SYSTEM DESCRIPTION

The main goal of this project is to make a fully functional open-source microcontroller, in this section we'll go into detail on how we're going to be designing, implementing, and testing each component. To start things off, every component will be following the "Specification for the WISHBONE System-on-Chip (SoC) Interconnection Architecture" [14] standard when it comes to the signals coming from or going to the peripheral. In addition, all design specification and verification procedures will follow the "IEEE Std 1012™-2016 IEEE Standard for System, Software, and Hardware Verification and Validation" [15] From section 3.1, we saw

that we have eight components inside the microcontroller, CPU, ROM, RAM, UART, PWM, Timer, I/O Port, and a WISHBONE Bus system. To go over how the system works, we'll give a brief explanation of how everything works, and how they're connected. First of all, the most important component in the system is the CPU, and since it's the MASTER that will take control of every other component, we'll talk about it first. The CPU is the brain of the microcontroller, it will do all needed operations to run any program given by the user. It will take instructions from the ROM, one by one, execute it and send out all necessary signals and flags to other components. ROM, it's the place where programs are stored.

The main purpose of the ROM is to send instructions that makeup computer programs to the CPU. RAM, is the component where temporary data is stored, it's used by the CPU to store data temporarily while the program is running. UART, it's a type of input/output module but the main difference is that it's asynchronous, meaning it doesn't need to use the same clock as other components. It is used to send and receive data to allow the microcontroller to communicate with external components. PWM, it's a module that's responsible for sending out pulse width modulated signals. It is used to control the power provided to external peripherals. Timers, as the name suggests, is a component that acts as a timer, its only purpose is to count up with each clock cycle.

I/O Port, its similar to the UART, but it's synchronized with the clock of the microcontroller. It is used to send and receive data to allow communication between the microcontroller and external peripherals. Finally, the WISHBONE Bus system. The Bus is the connection between the CPU and all other components, its main purpose that it connects everything.

3.2.1 BASELINE DESIGN COMPONENTS SPECIFICATIONS

3.2.1.1 CPU

In figure 11 we can see the entire data path and control unit of the CPU; it consists of many components connected to form the CPU. The diagram also shows the interaction between the CPU and the instruction memory (ROM) and the data memory (RAM). We'll go on how they work together, and go in-depth on each component later on. The first element in the data path is the PC, it points at the

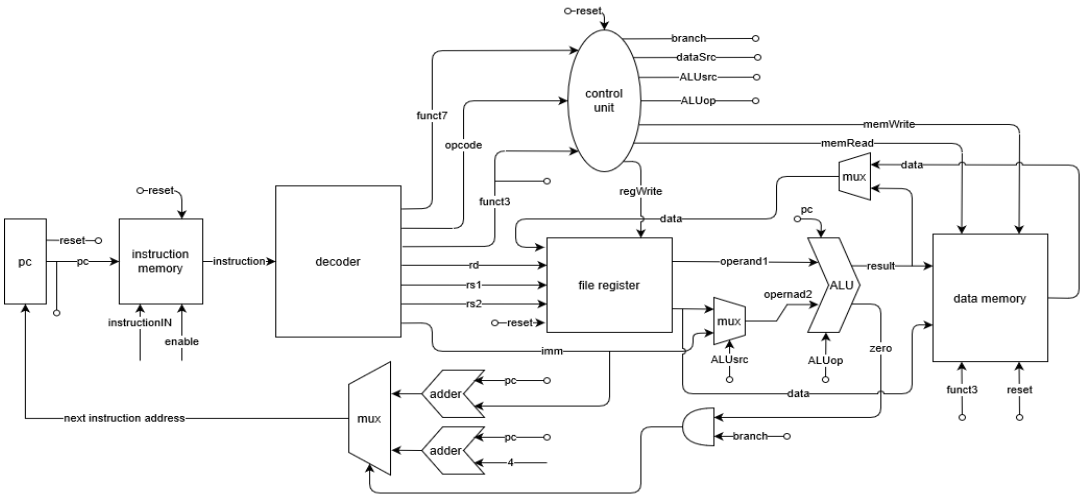


Figure 11 CPU Datapath

current address in the instruction memory. The instruction memory then takes the address from the PC and outputs the corresponding instruction to the decoder. The decoder then takes the instruction and decodes it into seven different signals, each signal having a different meaning. As shown in figure 11, signals named funct7, funct3, and opcode all goes to the control unit, while rd, rs1, and rs2 goes to the file register, and the last signal imm goes to the multiplexer after the file register. Signals going to the control unit determine the output of the control unit, which is to send control six signals to the rest of the components. As for the file register, it's a place to store data inside the CPU instead of storing it in the RAM or ROM. Storing data in the file register makes operations way faster since it's inside the CPU itself. Now that we have selected which operation do and which data to do these operations on, the data coming from the file register will move to the next component which is the ALU. The ALU is the component that does the logical and arithmetical operations on its two inputs that comes from the file register. After that, if we wish to store the data at RAM, the control signal has two enable writing to the RAM, and if we wish

to read data from the RAM, the control signal enables reading from the RAM. What we just discussed is the process of executing one instruction, after the CPU finishes executing one instruction, it increments the PC to move to the next instruction. More on each component inside the CPU in the next sections.

3.2.1.1.1 DECODER

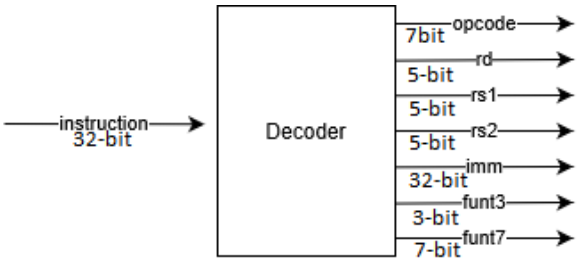


Figure 12 decoder diagram

In any computer or a binary system, any instruction you want the CPU to perform you'd write it in the assembler, then it will get translated to machine code, this translation is done by the assembler itself. By doing so, you're giving the CPU information on exactly what to do. Now let's say the CPU got the information in machine code, what's next? How does it understand this line of zeros and ones? Here comes the decoder. The decoder is a translator for the CPU, it will read 32-bit instructions written in machine code, and using internal circuitry, it will decode this instruction into some useful variables, each variable has its unique value. For example, the first and most important variable is the opcode, which is short for operation code, the opcode is useful for identifying what's the current instruction, whether it's an addition, subtraction, or any other instruction. Another example would be the two operands, rs1 and rs2, these two operands are mostly used with mathematical and logical operations, but can also be used for branches and calls instructions. As displayed in the next table, we can see each unique variable coming from the decoder, and what it does:

Table 11 decoder input/output

Opcode	Operation code
Rd	The register destination operand. It gets the result of the operation.
Rs1	The first register source operand.
Rs2	The second register source operand.
Imm	An immediate value
Funct3	An additional opcode field.
Funct7	An additional opcode field.

3.2.1.1.2 File Register

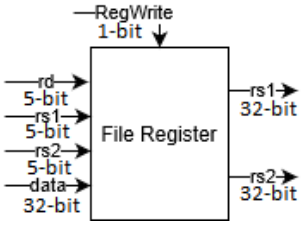


Figure 13 file register block diagram

After the decoding stage, decoded information will get passed to the next component to use, but usually, this data will not be received by one component, it can go through many wires to reach different destinations inside the CPU, one of the destinations is the what we are going to be talking about in this part of the report. As you can guess from the title of this paragraph, we are talking about a component called File Register, you might find this name familiar because we ‘ve already mentioned it previously, now let’s talk about it in detail. File Register is abasically 32-bit x 32 memory elements, it is used to store data temporarily until it's removed by instruction or the power is off, otherwise, it will be stored in the registers. File register is a very useful component, a lot of instruction would be so difficult or even impossible to perform. For example, let’s say we want to compare the summation of two number with a certain number, x, and y and we want to compare it with z, first, we’ll use the add instruction add them together, and then compare them with z, but we will need to execute another instruction to be able to compare it with the result from the previous operation, this is where we can find a problem, the result is lost since we didn’t store it in a memory element, and here comes the importance of registers. Results from operations can be saved in register addressed from x0 – x31, and future instructions can access them using their unique address for more mathematical, logical, store, or load operations. In the next table we will include every input and output for the file register and what it’s used for:

Table 12 file register input/output

Input	Rd	Address of the destination register
	Rs1	Address of the first operand register
	Rs2	Address of the second operand register
	Data	Data from the write-back stage
	RegWrite	Register write enable signal
Output	Rs1	Data stored in the first operand register
	Rs2	Data stored in the second operand register

3.2.1.1.3 ALU

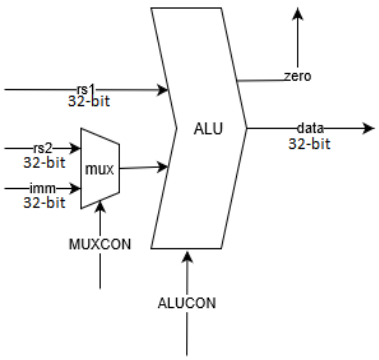


Figure 14 ALU block diagram

Mathematical and logical operations are the most important functionalities of a CPU, fast and accurate calculations are desired when creating any kind of system. We use computers to do complex calculations to eliminate the human error factor from the equation, some applications can't allow any kind of miscalculations, even small ones. For this reason, computer engineers needed to design a digital circuit that can do multiple mathematical and logical operations very fast with high precision and with little to no errors. ALUs, or Arithmetic Logical Unit, is what these engineers wanted to achieve, it's a component inside the CPU has two main inputs (operands) and an ALU control signal (operation), with these three signals the internal circuitry inside the ALU can produce two output signals, the first one is the data signal, which represents the result from the operation, whether it's an addition, subtraction, multiplication or division, the result will be outputted as the resulting signal. The next signal is the zero signal, this is a 1-bit signal that will have the value of 1 if the result from the current operation is 0, otherwise, it will be 1. We also can see that there's a multiplexer right before the second operand, this multiplexer is placed at the second operand because some instruction requires an immediate value, and some require a value stored in one of the registers, and since immediate values don't come from file registers, we needed a different path for them, and this multiplexer chooses between these two paths, depending on the MUXCON signal coming from the control unit (discussed in part-2 from this project. As shown in the next table, each input to the ALU system has a different functionality:

Table 13 ALU input/output

Input	Rs1	First operand value, which comes from file register
	Rs2	Second operand value, which comes from file register
	MUXcon	Multiplexer control signal chooses the second operand
	Imm	Immediate value, which comes from the decoder
	ALUcon	ALU control signal chooses the operation
Output	Zero	Zero flag signal
	Data	Result of the operation signal

Now that we know how the ALU works, we will also display how the ALU is controlled. As shown previously, the ALU has two control signals, one to choose the operand and one to choose the operation. The operand control signal is a 1-bit signal, that can only choose between two data buses, either the value will come from a file register, or the decoder as an immediate value, which doesn't make it complicated. But on the other hand, the ALU control signal controls the whole system, it's a 6-bit signal that can select 64 different operations, but in our case, we have only had 45 different instructions, so the rest is left unused, or we can also add some instructions if we wanted to, so we added a halt and bubble instructions but they don't need the ALU.

3.2.1.1.4 ADDER

The adder is a combinational circuit that will take two signals as an input to the module and will output the summation of them as an output signal. This assignment has two adders, one will be between the out from pc and constant value of 4, and the other will be between the pc and the Instruction memory. The first adder is made to add the pc by 4. Although adders can be constructed for many number representations, such as binary-coded decimal or excess-3, the most common adders operate on binary numbers. The second adder is used to calculate conditional branches and unconditional branches (call), it will take the value of the current pc and add an immediate value to it to calculate the jump amount from the current instruction addresses in the instruction memory. Also, the adders of the CPU will use binary only because it is the language of the computer. The adders that the CPU use are full adders, not half adders. In the next table we will be displaying all inputs and output to these adders:

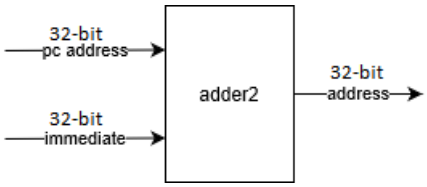


Figure 15 adder 2 block diagram

Table 14 Adder Inputs/Outputs

Input	Pc address	The current address stored in the PC module
	immediate	An immediate value used to calculate the jump amount
	4	Constant
Output	address	The new address for the PC module

3.2.1.1.5 MUX

The multiplexer is like the control, it will choose which input has to pick to make it pass to the next step. The CPU uses the multiplexer to connect two wires, it Can't just join wires together with no control because it will occur, so it has to use a multiplexer control the inputs. The controlling process will be dependent on the selector and it is the most significant bit. The mux has (2^n) inputs and has n select lines for the (2^n) inputs. The mux is called a multiple-input, single-output switch because as we said it takes (2^n) and it produces one output only. This project contains 3 multiplexers and they are:

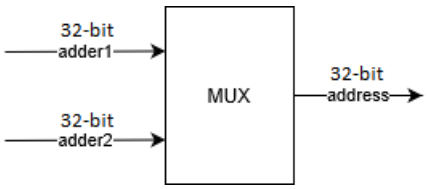


Figure 18 mux2 design

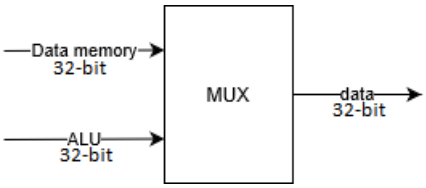


Figure 17 mux1 design

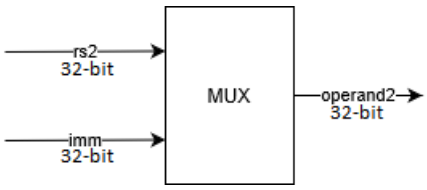


Figure 16 mux3 design

1- Mux1 between adder1 (pc + imm) and adder2 (pc + 4).
2- Mux2 between rs2 and imm.
3- Mux3 between the output from Instruction memory and (out from File registers) that will out operand 2 for the ALU

And finally, in the next table we will display all inputs and output for all three multiplexers:

Table 15 Multiplexer Inputs/Outputs

Input	adder1	The result from the first adder
	adder2	The result from the second adder
	data memory	Data coming from the data memory module
	ALU	Data coming from the ALU module
	rs2	Rs2 value
	imm	Immediate value
Output	address	The new address for the PC module
	Data	Data going back to the file register module
	operand2	The second operand for the ALU module

3.2.1.1.8 CONTROL UNIT

The last and one of the most important components inside the CPU is control unit component. This component is responsible for data flow throughout the whole system, it receives the opcode, funct3, and funct7 as its input, and inside, depending on these signals, it will output 7 different variables (figure 18), and these variables are; WE, branch, register write, ALU operation, data source, ALU source and STB (Table 15). All the ALU operation codes are written in the next table, they are divided depending on the type of instruction:

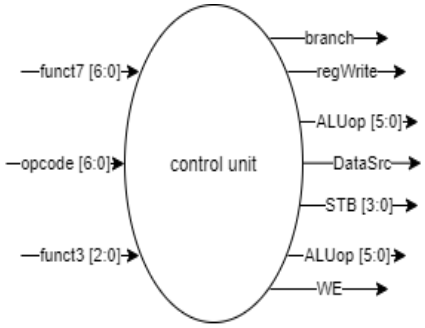


Figure 19 control unit block diagram

Table 16 control unit input/output

Input	Funct7	First operand value, which comes from file register
	Funct3	Second operand value, which comes from file register
	Opcode	Multiplexer control signal chooses the second operand
Output	Branch	Branch indication signal
	regWrite	Register File write enable signal
	ALUop	ALU operation selection signal
	STB	Optional 4-bit signal used with outside of the CPU modules
	DataSrc	Register File data source selection signal
	WE	Write enable signal going outside of the CPU
	ALUsrc	Selects between imm or rs2 signals as the second ALU operand

Table 17 Control Unit opcode types

type	instruction	opcode
I-type	LB	000000
	LH	000001
	LW	000010
	LBU	000011
	LHU	000100
	SLLI	000101
	SRLI	000110
	SRAI	000111
	ADDI	001000
	XORI	001001
	ORI	001010
	ANDI	001011
	SLTI	001100
	SLTIU	001101
	JALR	001110

S-type	SB	001111
	SH	010000
	SW	010001
R-type	SLL	010010
	SRL	010011
	SRA	010100
	ADD	010101
	SUB	010110
	XOR	010111
	OR	011000
	AND	011001
	SLT	011010
	SLTU	011011
	MUL	011100
	MULH	011101
	MULHSU	011110
	MULHU	011111
	DIV	100000
	DIVU	100001
	REM	100010
	REMU	100011
U-type	LUI	100100
	AUIPC	100101
J-type	JAL	100110
B-type	BEQ	100111
	BNE	101000
	BLT	101001
	BGE	101010
	BLTU	101011
	BGEU	101100

3.2.1.2 RAM

This block represents a memory element inside the Microcontroller, it's supposed to store and hold on to data until it's changed, overwritten or we shut the system down. Each memory element can store up to 8-bits, so data represented in more than 8-bit will be stored in different memory elements next to each other. This module has five inputs and one output.

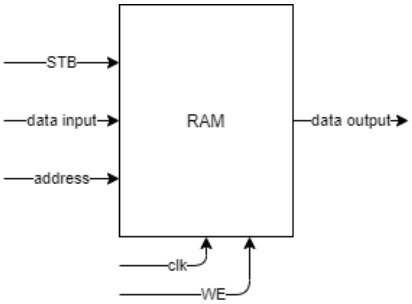


Figure 20 RAM block diagram

Inputs are, clock, WE, address, and data input, And the output is the data output signal. All these signals together control data memory to be able to store, receive, and transfer data. In the next table we will display the functionality of each signal and how it affects the flow of data in the system:

Table 18 RAM input/output

Input	clock	Clock signal used to control the flow of data
	WE	Used to enable write/read
	STB	Used to select between the number of bytes (8-16-32)
	address	Address inside the data memory
	Data input	Data to be stored in the memory
Output	Data output	Output data from the memory to the CPU

3.2.1.3 ROM

This is a module that represents a memory that is used to store instructions for the CPU to execute. The ROM will output the previously initialized instructions to the decoder.

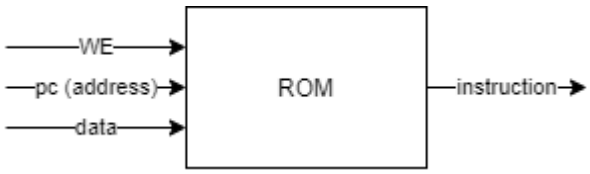


Figure 21 ROM block diagram

The instruction memory is a combinational element so it doesn't need a clock, and it will send instructions depending on the address coming from the PC module. This module has three inputs and only one output, the first input is the WE signal, it is used as an ON/OFF switch for the instruction memory, but for the type of memory (ROM), we won't be writing any data, so this signal is always low. The second signal is the pc (or the address), it is used to select which instruction to execute after each clock cycle. Lastly, the data input signal. This signal is not used since we're designing a read-only memory, but it is needed because the FPGA used in this

project doesn't support ROM memory blocks but has RAM memory blocks. In the next table we will be displaying all inputs and outputs for the instruction memory:

Table 19 ROM input/output

Input	data	Constant zero
	pc address	Address coming from the PC module
	WE	Write/read signal for the data memory. Always low
Output	instruction	Binary instructions

3.2.1.4 PWM

In figure 23, we can see the block diagram of the PWM module, and its inputs and outputs. For the inputs, we have the data input, which represents the period time and the active time of the PWM signal. STB, enables the module and starts the generation, reset resets everything. Next, we have the data path of the PWM module, and for the design, we decided that it needs only 2 main signals coming from PWM controller and 3 coming from the control unit. It will

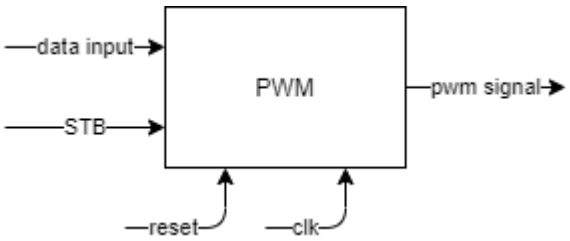


Figure 23 PWM block diagram

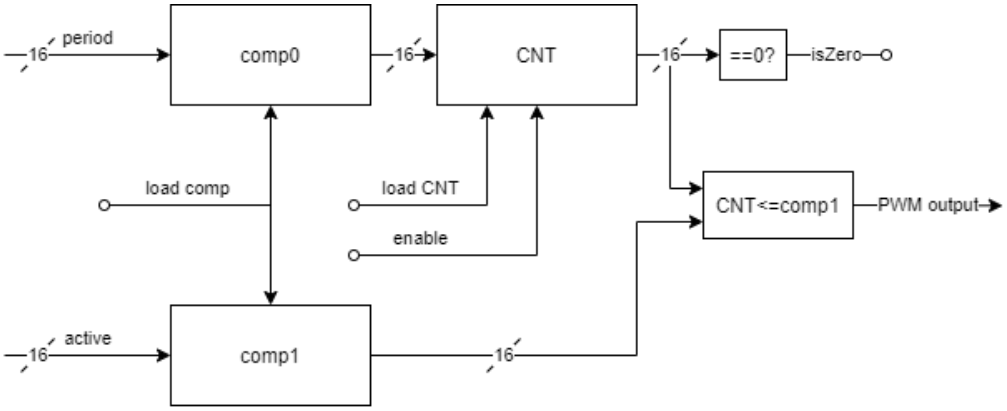


Figure 22 PWM Datapath

receive the period and the active times, also, load signals for the countdown register (CNT) and the two comp registers, and finally, an enable signal for the CNT to start counting down. This design will also output 2 signals, the PWM signal and a flag signal that indicates that the CNT value is nearing 0. And finally, in the next table we'll summarize all the input and output signals to the PWM module:

Table 20 PWM input/output

input	Clock	The clock used to control the flow of data
	Data	Contains the period and the active time
	Reset	Reset the PWM module
	STB	Enables the module and starts it
output	PWM signal	Pulse modulated signal

3.2.1.5 UART

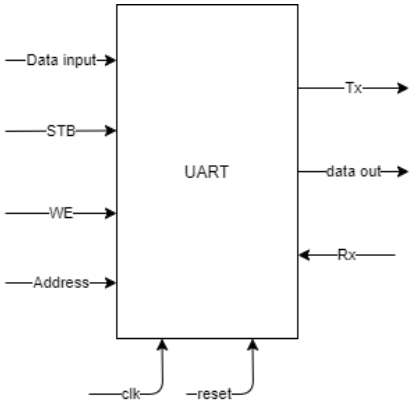


Figure 24 UART block diagram

As we've mentioned previously, the UART is responsible for transmitting and receiving, to and from the microcontroller. The UART like every other component is connected to the Bus inside the microcontroller, it receives control signals to whether to transmit or receive data. The Bus is also used to receive data from the CPU to send to external peripherals and to send data back to the CPU. Inside the UART there are four main components, a control unit, baud rate generator, Tx and Rx modules. The Tx used for transmitting data, it holds 8-bit data provided by the user to send outside the microcontroller through one of its ports. Now for Rx module, it is used to receive data from outside the microcontroller through one of its port, and store it inside the UART unit new data arrives or the CPU receives it. As for the control unit, it controls the flow of data inside the module, it waits for flags or signals coming from the CPU, all four modules, and the Baud Rate Generator to produce signals accordingly. Lastly, the Baud Rate Generator, allows the UART to send/receive data asynchronously by changing the baud rate to match the baud rate of the external peripheral or device.

Table 21 UART input/output

Input	Data input	Data coming from the CPU for the UART to transmit
	STB	Signal to enable the UART
	WE	Signal to enable transmitting data from the UART
	Address	Address signal coming from the CPU
	Clock	Clock signal
	Reset	Reset signal
	Rx	Data coming from the Rx Pin
Output	Data output	Data stored in the UART (Rx data, Rx flag, Tx flag)
	Tx	Data transmitted from the UART

3.2.1.6 Timer

The timer module is the easiest in the system, it only consists of two components, the control unit, and a count-up-register. The control unit will receive an initial value from the CPU to set the value of the count-up-register, otherwise, it will start at 0. Also, the control unit can control the number of clock cycles required for each count up register increments. As for the count up register, it's a register that will increment its value every clock cycle or so, depending on the control unit.

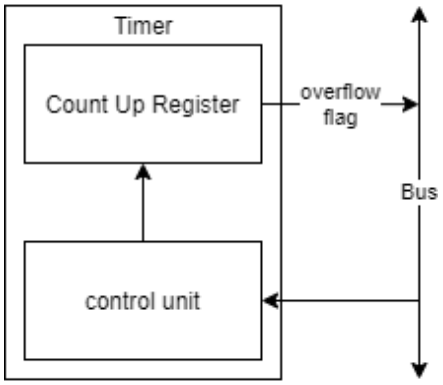


Figure 25 Timer Module Block Diagram

3.2.1.7 I/O Ports

For the general input/output ports, we decided to go with two simple input/output ports designs. As shown in figure 26, this is output port and it has three inputs (STB, data in, clock) and it has one output (data out). The module is responsible for sending data outside the microcontroller. As for the input port, it receives data from outside the microcontroller and saves it until the CPU is ready to receive it. These two allow the CPU to communicate the peripherals outside microcontroller synchronously. We decided to go with two different ports instead of one bidirectional port because MyHDL doesn't support bidirectional (in-out) signals.

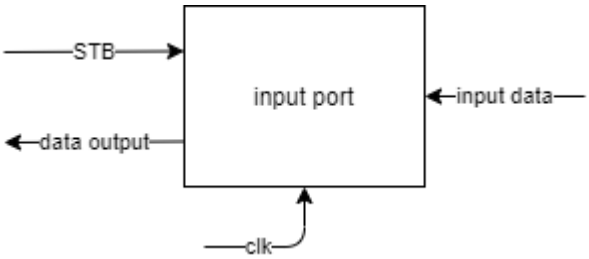


Figure 27 input port

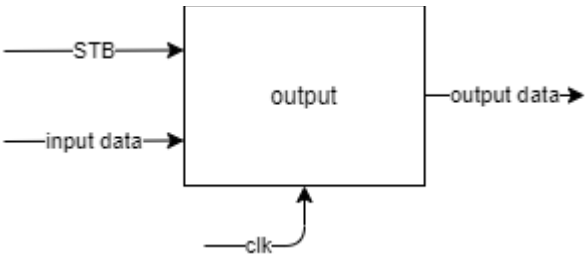


Figure 26 output port

3.2.1.8 SBA Bus System

As for the Bus System, its whole purpose is to connect different components with each other, and connect external components with the internal components inside the microcontroller. The Bus System follows the Simple Bus Architecture, so it has its signals, and has the same number of buses, which are: input bus, output but, address bus and data bus. The input bus is responsible for transferring data from all the slaves to the CPU. The output bus is responsible for transferring data from the CPU to all the slaves. The address bus will carry the address of the desired module/peripheral. And, the data bus will carry the data that the CPU wants one of its slaves to have. Also, the bus has multiple supporting/optional signals, which are: STB, WE and RST. The STB signal is a 4-bit that carries any options needed by any of the peripherals. The WE signal is basically an enable signal for writing or reading to/from any of the peripherals. Last, the RST signal resets every component in the microcontroller. In addition, the bus system has the SBA decoder. Which will map the address given to it by the CPU to one of its slaves, and will only enable this particular slave.

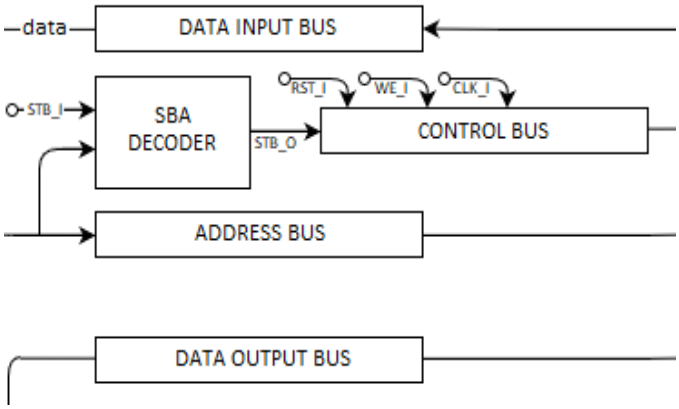


Figure 28 Bus System

3.2.3 FLOWCHARTS FOR SOFTWARE BLOCKS

3.2.3.1 CPU FLOWCHART

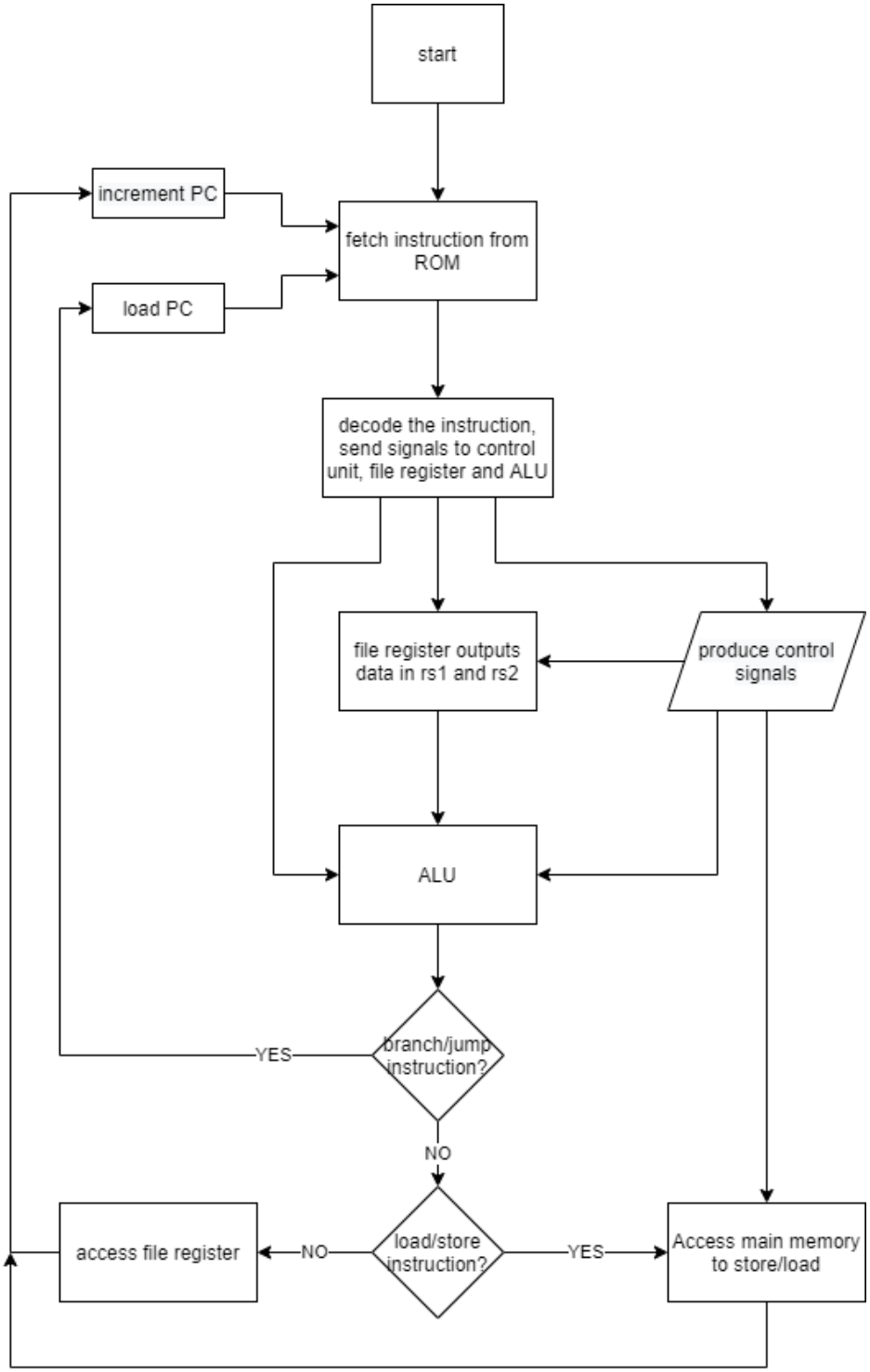


Figure 29 CPU Flowchart

3.2.3.2 SYSTEM FLOWCHART

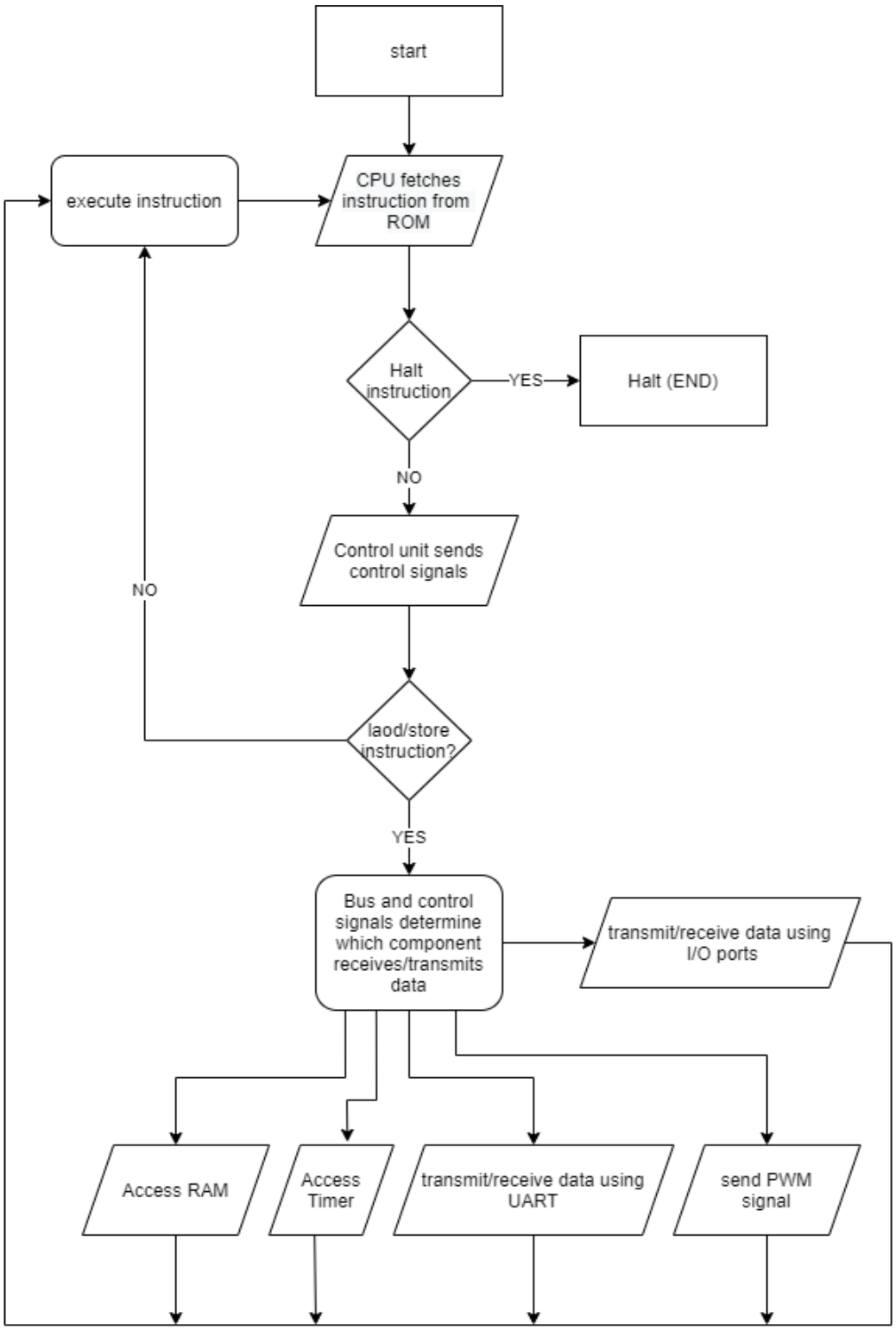


Figure 30 System Flowchart

3.2.5 POSSIBLE AESTHETICS

We aim to use a python library called QT5 to design the user interface of the system builder. This library is easy to use and can produce attractive designs that will make the final look of the program professional and convenient to the user.

3.2.6 INPUT/OUTPUT SPECIFICATIONS

Since most of the work is done in software, and the only hardware component for this solution is the FPGA, the input/output specification depends on the specifications of the FPGA board itself. For our project, we managed to get one of Intel's FPGAs, the DE10-Lite [20] board. According to the datasheet, the input to the board is 5V DC from USB or external power supply and can receive/transmit a 3.3V signal through its I/O ports.

3.2.7 OPERATING INSTRUCTIONS

Instruction to use the microcontroller is straight forward and easy to follow, not like most microcontrollers out there.

1- Write your RICS assembly program or in any programming language that can be translated to RISC assembly language using libraries or plug-ins.
2- Copy your RICS assembly code and use a RICS-V assembler to translate it to machine code, and paste it into the instructions.txt file, and use bin_to_8hexfile.py to generate the memory initialization files(.mif).
3- Use the system builder to select the number of peripherals needed and to generate the SBA bus, SBA decoder and top-level design.
4- Convert the python top level code (SBA_SYS_GENERATED.py) to Verilog HDL.
5- Copy the memory initialization code(e.g. $readmemh(bankname.mif, MemoryBankName), and paste it in the generated Verilog code, after deleting the ROM initialization loops.
6- Use Quartus or Xilinx to compile the design and upload the sof file to the FPGA.

3.3 SIMULATION RESULTS

After designing the internal components of the microcontroller, we started working on the main components in the system, the CPUs control unit, ALU, decoder. After a lot of trial and error, we came up with an initial simulation for these parts in python. In this section, we'll be showcasing the simulation result for each component.

3.3.1 CONTROL UNIT SIMULATION

```
# R-TYPE
opcode.next = 0b0110011
funct3.next = 0b0
funct7.next = 0b0
yield clk.posedge
print('opcode: %s | funct3: %d | funct7: %d | ALUop: %d | memRead: %d | \nmemWrite: %d | regWrite: %d '
      '| branch: %d | ALUsrc: %d | dataSrc: %d' % (bin(opcode,7),funct3,funct7,ALUop,memRead,memWrite,
                                                   regWrite,branch,ALUsrc,dataSrc))
# I-TYPE (LOAD)
opcode.next = 0b0000011
funct3.next = 0b0
funct7.next = 0b0
yield clk.posedge
print('\nopcode: %s | funct3: %d | funct7: %d | ALUop: %d | memRead: %d | \nmemWrite: %d | regWrite: %d
      '| branch: %d | ALUsrc: %d | dataSrc: %d' % (bin(opcode,7),funct3,funct7,ALUop,memRead,memWrite,
                                                   regWrite,branch,ALUsrc,dataSrc))
yield clk.posedge
print('\nopcode: %s | funct3: %d | funct7: %d | ALUop: %d | memRead: %d | \nmemWrite: %d | regWrite: %d
      '| branch: %d | ALUsrc: %d | dataSrc: %d' % (bin(opcode,7),funct3,funct7,ALUop,memRead,memWrite,
                                                   regWrite,branch,ALUsrc,dataSrc))
```

```
opcode: 0110011 | funct3: 0 | funct7: 0 | ALUop: 0 | memRead: 0 |
memWrite: 0 | regWrite: 0 | branch: 0 | ALUsrc: 0 | dataSrc: 0

opcode: 0000011 | funct3: 0 | funct7: 0 | ALUop: 21 | memRead: 0 |
memWrite: 0 | regWrite: 1 | branch: 0 | ALUsrc: 0 | dataSrc: 0

opcode: 0000011 | funct3: 0 | funct7: 0 | ALUop: 0 | memRead: 1 |
memWrite: 0 | regWrite: 1 | branch: 0 | ALUsrc: 1 | dataSrc: 1
<class 'myhdl._SuspendSimulation'>: Simulated 30 timesteps
```

Figure 31 CU simulation

In figure 31, we can see the result from the simulation of the CPUs control unit. For simplicity, we are only showing two instructions and what is the output from the control unit on three different clock cycles. We first tested an R-type instruction and observed the output from the control unit, and after that, we tested an I-type instruction and observed the output printed on the console as well, and the result came as we expected, the control unit sent out the correct signals.

3.3.2 ALU SIMULATION

In the previous figure, we can see the result from the simulation of the ALU. For this simulation, we decided to use three different operations and print the result on the console. The first instruction was a simple addition between the values 2 and 7, and as displayed in figure 32, we can see the result is 9 (in binary). After that, we tested the SRA instruction, which is supposed to shift the binary equivalent of the number 15 two digits to the right, and as shown in the figure, the value of 15 is shifted two

```
@instance
def stimulus():

    # ADD instruction
    operand1.next = 7
    operand2.next = 2
    control.next = 0b010101
    print('before> ', bin(operand1, 32), bin(operand2, 32), bin(out, 32))
    yield delay(10)
    print('after> ', bin(operand1, 32), bin(operand2, 32), bin(out, 32))
    print()

    # SRA instruction
    operand1.next = 15
    operand2.next = 2
    control.next = 0b010100
    print('before> ', bin(operand1, 32), bin(operand2, 32), bin(out, 32))
    yield delay(10)
    print('after> ', bin(operand1, 32), bin(operand2, 32), bin(out, 32))
    print()

    # MUL instruction
    operand1.next = 50
    operand2.next = 40
    control.next = 0b011100
    print('before> ', bin(operand1, 32), bin(operand2, 32), bin(out, 32))
    yield delay(10)
    print('after> ', bin(operand1, 32), bin(operand2, 32), bin(out, 32))
    print()
```

```
before>  00000000000000000000000000000000 00000000000000000000000000000000 00000000000000000000000000000000
after>  00000000000000000000000000000111 00000000000000000000000000000010 00000000000000000000000000001001

before>  00000000000000000000000000000111 00000000000000000000000000000010 00000000000000000000000000001001
after>  00000000000000000000000000001111 00000000000000000000000000000010 11000000000000000000000000000011

before>  00000000000000000000000000001111 00000000000000000000000000000010 11000000000000000000000000000011
after>  00000000000000000000000000110010 00000000000000000000000000101000 00000000000000000000011111010000
```

Figure 32 ALU simulation

digits to the right as expected. Lastly, we tried multiplication, we multiplied 15 by 2, and as shown in the output, the result is correct (30).

3.3.3 DECODER SIMULATION

For the decoder, we tested out two different instructions. The first instruction was an R-type instruction, and the second one was an I-type instruction. The goal of this simulation is to see whether the decoder will decode the raw binary instruction and separate the signals into their proper form or an error will happen. And after looking at the result, we can see that the decoder is working as expected.

```
@instance
def simulate():
    yield delay(1)  # starter delay
    # print default values
    print(bin(num, width=32))
    print('rd:',int(rd),'rs1:',int(rs1),'rs2:',int(rs2),'imm:',int(imm))
    print()

    # this should be r type instruction
    # add x27,x30,x11
    num.next = 0b00000000101111110000110110110011
    yield delay(10)
    print(bin(num, width=32))
    print('rd:',int(rd),'rs1:',int(rs1),'rs2:',int(rs2),'imm:',int(imm))
    print()

    # this should be i type instruction
    #srai x22, x12,30
    num.next = 0b01000001111001100101101100010011
    yield delay(10)
    print(bin(num, width=32))
    print('rd:',int(rd),'rs1:',int(rs1),'rs2:',int(rs2),'imm:',int(imm))
    print()
```

```
opcode: 0000000
invalid opcode
00000000000000000000000000000000
rd: 0 rs1: 0 rs2: 0 imm: 0

opcode: 0110011
next instruction is r type
00000000101111110000110110110011
rd: 27 rs1: 30 rs2: 11 imm: 0

opcode: 0010011
next instruction is i type
01000001111001100101101100010011
rd: 22 rs1: 12 rs2: 0 imm: 30
```

Figure 33 decoder simulation

3.3.4 Register File

```
@instance
def test():
    rs1Address.next = 10  # rs1 address is set to 10
    rs2Address.next = 11  # rs2 address is set to 11

    writeEnable.next = 1  # enable the writing mode
    rdAddress.next = 10  # update the value of rdAddress to 10
    data.next = 200  # the input data to the register at rd is 200
    yield clock.negedge  # wait for the clock
    print('Register File output signals:\nrs1: %d rs2: %d' % (int(rs1Out), int(rs2Out)))

    # test the file register by trying to write 450 at address at 11
    writeEnable.next = 1  # enable the writing mode
    rdAddress.next = 11  # update the value of rdAddress to 10
    data.next = 450  # the input data to the register at rd is 450
    yield clock.negedge  # wait for the clock
    print('Register File output signals:\nrs1: %d rs2: %d' % (int(rs1Out), int(rs2Out)))
```

Figure 34 register file simulation setup

As for the Register File, we will test if the storing and outputting mechanisms work as expected. First, we changed the values for the register1 address to 10, and register2 address to 11. By doing so, the output signals from the Register File will always output the contents of both these registers. Moving on, we enabled the writing mode, and selected address 10 as the destination address for the input data, which is 200. After that, we did the same but changed the input value to 450, and the destination address to 11.

From figure 35, we can see the result of the previous operations done on the Register File. From the first operation, we can see that the value 200 is stored at

```
now 5 registers:
register 0 : 0
register 1 : 0
register 2 : 0
register 3 : 0
register 4 : 0
register 5 : 0
register 6 : 0
register 7 : 0
register 8 : 0
register 9 : 0
register 10 : 0
register 11 : 0
register 12 : 0
end
Register File output signals:
rs1: 200 rs2: 0
```

```
now 15 registers:
register 0 : 0
register 1 : 0
register 2 : 0
register 3 : 0
register 4 : 0
register 5 : 0
register 6 : 0
register 7 : 0
register 8 : 0
register 9 : 0
register 10 : 200
register 11 : 0
register 12 : 0
end
Register File output signals:
rs1: 200 rs2: 450
```

Figure 35 register file simulation output

address 10 in the Register File. Also, at address 11, we can see that the value 450

is also stored. Finally, the output of the Register File depends on the values of register1 address and register2 address, as we've mentioned earlier.

3.3.5 PWM

```
module test_bench();

// inputs
reg [31:0] data;
reg reset, clk;
reg [3:0] STB;

// outputs
wire PWM_output;

SBA_PWM pwm(data, reset, clk, STB, PWM_output);

// initialize values to 0
initial begin
    data = 0;
    reset  = 0;
    clk    = 0;
    STB = 4'b1000;  // module enabled but stopped
    #1;
end

// clock generator
always begin
    clk = ~clk;
    #3;
end

// test
initial begin
    #6
    data = 32'b0000_0000_0001_0100_0000_0000_0000_1010;
// period=20;
// active=10;
    STB = 4'b1010;  // module enabled
    #500;
end
endmodule
```

Figure 37 PWM simulation setup in Verilog

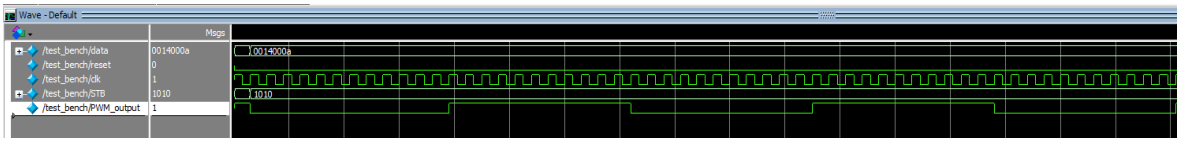


Figure 36 PWM simulation result in ModelSim

For the PWM module, it works by setting two main parameters, the active time and the period time. The active time represents the time when the signal is set to high during the period. The period is the total time of the cycle. For this simulation, chose to set the period time to be 20, and the active time to be 10. Basically, the pulse modulated signal is going to be half active and half inactive as shown in figure 37.

In figure 36, we can see the result of this simulation in ModelSim. To start setting up the PWM module, we set the data input signal to 0014_000A, which means that the active time is 10 (A in hex) and the period time is 20 (14 in hex). And, by setting the STB signal to 1010, we activate the module. The PWM module produced the signal we expected, the full period takes 20 clock cycles, and 10 clock cycles are set to high, and the rest to low.

3.3.6 UART

The UART module has two main functionalities, send data, and receiving data. For the first simulation, we'll start by testing if the UARTs Rx module works as expected. in figure 38, we can see the initial values given to the Rx module from outside the microcontroller. We start by sending the start bit (high signal) to indicate the start of a byte. After that we start sending the byte we want to send. At the end, we produce the stop bit signal, to indicate the ending of the byte. In figure 38, we can see that the Rx module stared receiving and storing the data coming from the port successfully.

```
Rx.next = 0  # start bit
yield delay(64)
Rx.next = 0
yield delay(64)
Rx.next = 1
yield delay(64)
Rx.next = 1
yield delay(64)
Rx.next = 0
yield delay(64)
Rx.next = 0
yield delay(64)
Rx.next = 1
yield delay(64)
Rx.next = 1
yield delay(64)
Rx.next = 0
yield delay(64)
Rx.next = 1  # stop bit
yield delay(64)
```

```
start
35 *** RX 1: start bit value: 0 ***
99 *** RX 2: received value: 0 ***
163 *** RX 2: received value: 1 ***
227 *** RX 2: received value: 1 ***
291 *** RX 2: received value: 0 ***
355 *** RX 2: received value: 0 ***
419 *** RX 2: received value: 1 ***
483 *** RX 2: received value: 1 ***
547 *** RX 2: received value: 0 ***
611 *** RX 3: stop bit value: 1 ***
```

Figure 38 Rx simulation setup and result

As for the Tx module, the transmitting module, the test is fairly simple. in figure 39, we can see the initial value of the input signal is 01100110. After that, the only thing we need to wait for is the clock to start transmitting a single byte of data. in figure 39, we can see that the Tx module started sending a start bit before it started transmitting the byte. This indicates to the receiver that a byte will be transmitted after this bit. After that, the Tx module started transmitting the actual byte bit by bit, with a delay between each bit following the specified baud rate. At the end, the Tx module has to generate the stopping bit signal, to indicate the completion of the transmitting operation.

```
@instance
def run():
    print('start')
    TxData.next = 0b01100110
    yield clk.negedge
```

```
start
63 *** TX 1: start bit value: 0 ***
191 *** TX 2: bit value: 0 ***
255 *** TX 2: bit value: 1 ***
319 *** TX 2: bit value: 1 ***
383 *** TX 2: bit value: 0 ***
447 *** TX 2: bit value: 0 ***
511 *** TX 2: bit value: 1 ***
575 *** TX 2: bit value: 1 ***
639 *** TX 3: last bit value: 0 ***
703 *** TX 4: stop bit value: 1 ***
```

Figure 39 Tx simulation setup and result

3.3.7 I/O Port

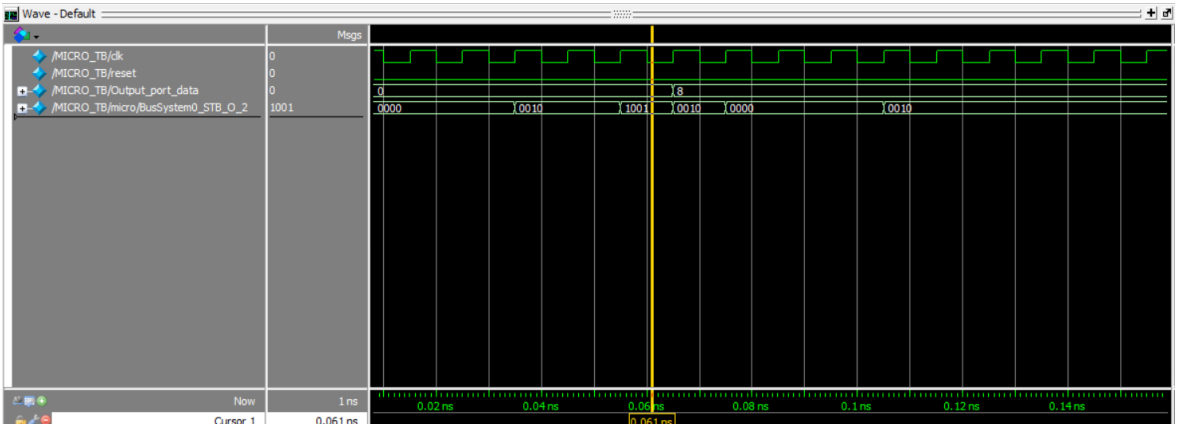


Figure 40 output port simulation in ModelSim

As the name suggests, the I/O ports are just simple ports responsible for inputting and outputting data into and out of the CPU. In this simulation, we tested the capabilities of both of these ports individually. First, we tested the input port, as it will be receiving data from outside the CPU. As shown in figure 41, we can see the data coming from the port to the input module. This is a simple test as we expected. Next on, we tested the output port, to see if it capable of outputting the byte we assigned to it. And as you can see in figure 40, the CPU sent 1 byte of data to the module, and so the module took this data and as if it forwards it to the pin, for other devices outside the microcontroller can receive it.

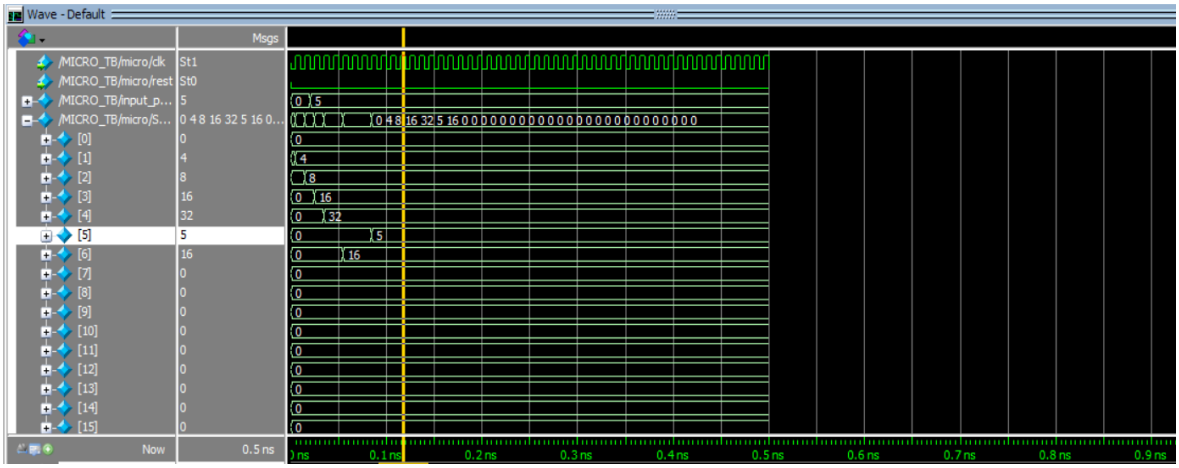


Figure 41 input port simulation result

3.3.8 Timer

From figure 42, we can see the sittings for testing the timer module. For testing reasons, we set the timer to be an 8-bit timer. So, the timer will start counting from the base value given to it by the CPU (or from zero if none was given) unit it reaches the maximum limit of the counter, and then the counter resets back to zero, and a flag is risen.

```
1   module timer_tb();
2
3   // input
4   reg [7:0] base;
5   reg clk;
6   reg reset;
7   reg [3:0] STB;
8
9   // output
10  wire flag;
11
12  timer tm(STB, base, clk, reset, flag);
13
14  always
15  #5 clk <= ~clk;
16
17  initial begin
18  clk = 0;
19  reset = 0;
20  base = 8'h00;
21  STB = 1'b0;
22
23  #10;
24  base = 8'hFC;
25  STB = 4'b1010;
26  #10;
27
28  end
29
30  endmodule
```

Figure 42 Timer simulation setup

For this simulation, the base value for the timer is set to 252. So, the counter should start counting from 252 until it reaches the value 255 (maximum value for 8-bit counter). After that, a flag indicating that the timer reached its maximum value should be set to high. In figure 43, we can see the result from this simulation using ModelSim. At first, the initial value of the counter changed from zero to 252, and then, after each clock cycle, the counter increments unit it reaches the maximum value (255), the counter goes back to zero, and the flag is set to high.

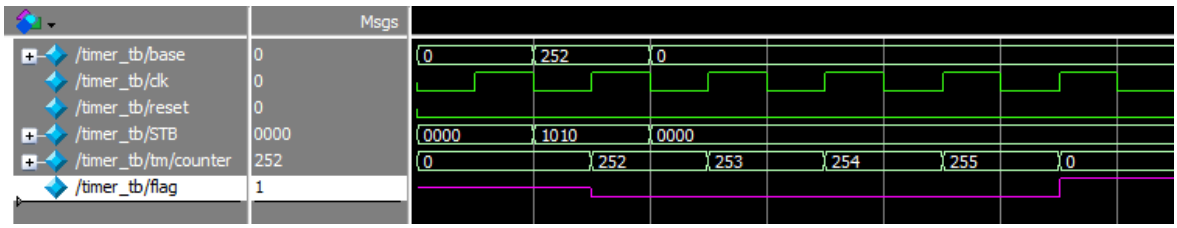


Figure 43 Timer simulation result

3.3.9 ROM

The ROM component, is the place where instructions should be stored before the start of the microcontroller. For this test, we will start testing the ROM module and see if its capable of loading instructions. As you can see in figure 44, we started by initializing the internal memory elements to the specified instruction values from a mif file. As a result, we should see all instructions inside the ROM at the start of the simulation. Next, we started playing with the input address, changing it every few time units. As a result, we expect the output of the ROM to change accordingly.

```
11  // Instantiate the Unit Under Test (UUT)
12  ROM ROM1(WE, DATA_IN, ADDRESS_IN, DATA_OUT);
13
14  initial begin
15  WE = 0;
16  ADDRESS_IN = 0;
17  DATA_IN = 0;
18
19  # 5;
20  ADDRESS_IN = 4;
21
22  # 5;
23  ADDRESS_IN = 8;
24
25  # 5;
26  ADDRESS_IN = 12;
27
28  # 5;
29  ADDRESS_IN = 16;
30  end
```

Figure 44 ROM simulation setup

In figure 45, we can see that all banks inside the ROM got initialized with the instructions from mif file. So, when an address is given to the ROM, the output is a combination of all signals coming from these banks to form a full instruction. Going back to the figure, we can see that with the change of the address, the output of the ROM is a combination of the 8-bit signal coming from each bank to form a full 32-bit instruction as expected.

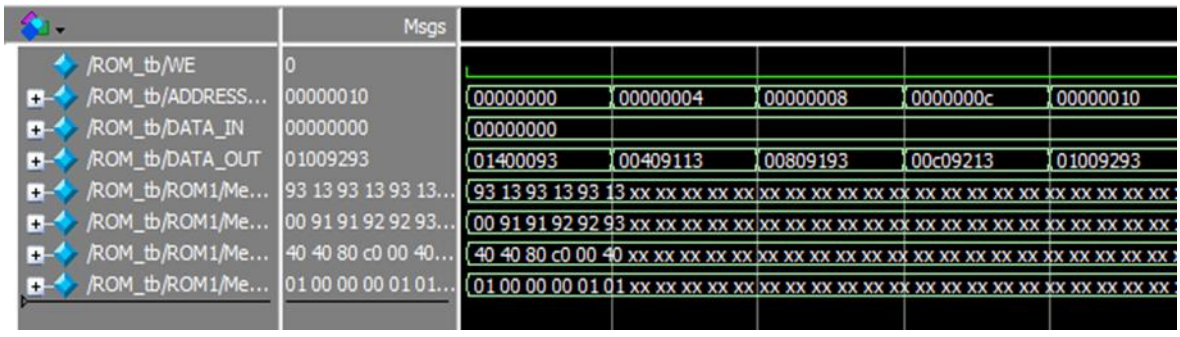


Figure 45 ROM simulation result

3.3.10 RAM

For the random-access memory, the test is fairly similar to the ROM test we finished previously. The difference between the two modules is that the ROM has a fixed zero as input data and write enabling signal. in RAM, both of these signals come from the CPU, and are not fixed like in ROM. So, to start testing the RAM, we started first by storing data into it, as seen in figure 47. After that, we selected those same addresses to load data from them back to the CPU. In figure 46, it shows the simulation from the previous store and load operations. First, we can see that we enabled the STB signal to indicate the we want to use the RAM module. After that, we provided the RAM with the address, data we need to store and WE signal. After we stored data in ROM, we then changed the WE signal to indicate that we now want to load from the ROM instead of writing into it. By looking at the output of the RAM, we can see that the result matches the contents of the memory elements with the address we provided.

```
11  // Instantiate the Unit Under Test (UUT)
12  ROM ROM1(WE, DATA_IN, ADDRESS_IN, DATA_OUT);
13
14  initial begin
15  WE = 0;
16  ADDRESS_IN = 0;
17  DATA_IN = 0;
18
19  # 5;
20  ADDRESS_IN = 4;
21
22  # 5;
23  ADDRESS_IN = 8;
24
25  # 5;
26  ADDRESS_IN = 12;
27
28  # 5;
29  ADDRESS_IN = 16;
30  end
```

Figure 47 RAM simulation setup

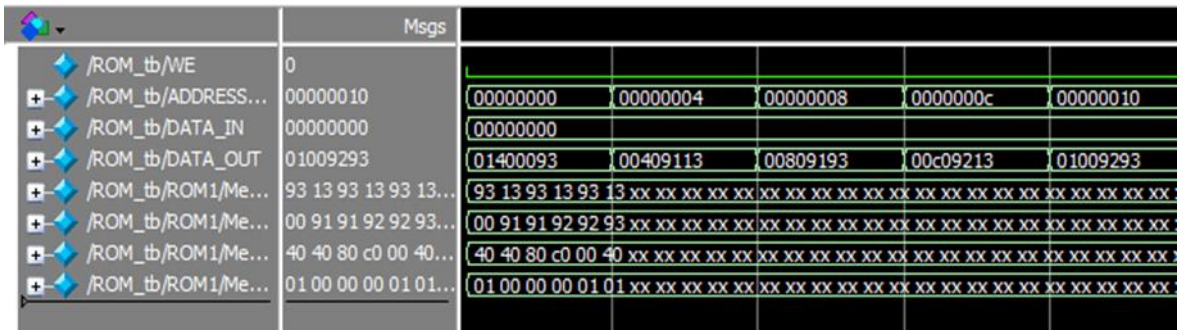


Figure 46 RAM simulation result

CHAPTER – 4 IMPLEMENTATION

In this section we will introduce the parts of our project that are essential for the project to be completed which means that without it the product will not work at all. Having this in mind, we have four essential tasks(blocks) that are shown in the baseline design of the project that was shown in chapter three of this report. The four essential blocks are the Central Processing Unit (CPU), Random Access Memory (RAM or data memory), Read Only Memory (RAM or instruction memory) and lastly the Bus System that follow the wishbone bus architecture.

4.1 ESSENTIAL TASK 1 / CPU

The first essential block in a design as seen in our baseline design is the Central Processing Unit, or CPU for short. The CPU is one of (if not the most) important module/component in the project. It is considered to be the heart and brain of the microcontroller. In general, what a CPU does is deal with the execution of instructions, coming from the ROM (essential task #2). Doing some arithmetic and logical operations, calculations, sending and receiving data from other modules, and controlling the flow of data in the microcontroller to achieve a desired functionality. The process of designing, creating and testing the architecture of the CPU will be displayed in the next three sub-sections (trials). Each one will talk about the progression of any modification done to the internal design, data path, signals or the architecture in general.

4.1.1 First trial – Multiple Stage CPU

The first initial design of the CPU was as follows; the CPU is separated into 5 sections, instruction fetch, instruction decoding and register file access, instruction execution, memory access, and finally write back. This design is inspired by the basic CPU architecture introduced in the book: “Computer Organization and Design RISC-V Edition”. We first picked this design since we were familiar with it (taught in course EE-361), and it seemed to provide a high level of concurrency when

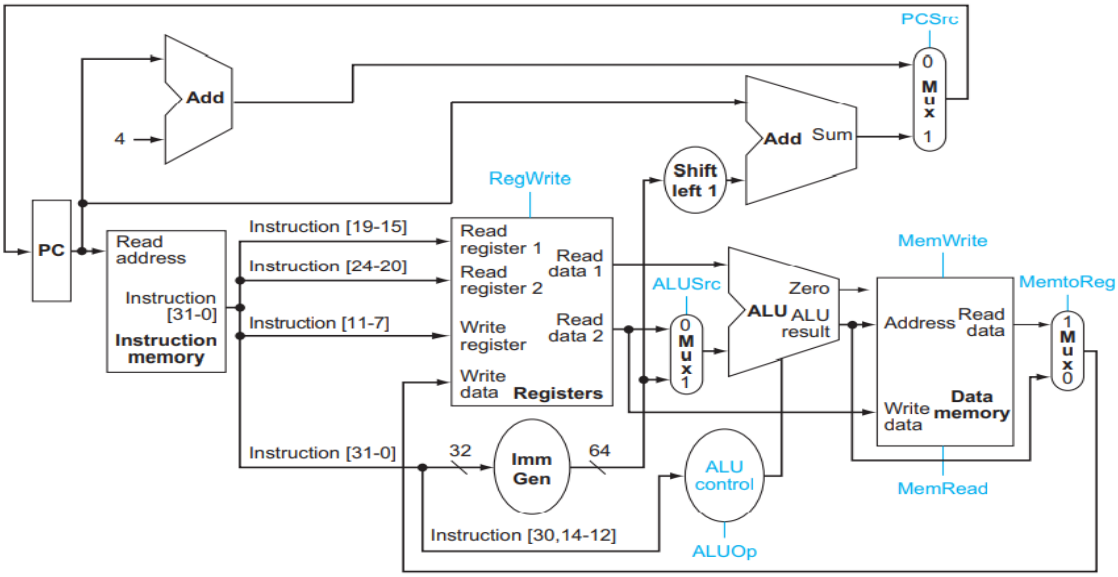


Figure 48 RISC-V pipelined CPU design

executing instructions. The block diagram of the CPU is shown in figure 48.

This design is a good design when parallelism is the most important aspect of the CPU design. Since we’re dealing with sections in this design, we can execute multiple instructions simultaneously. Each instruction can get access to one of the sections one after the other. By doing so, we can get up to five instruction to be executed at the same time. This seems to be a great design at first, but as we dive deeper, we can see a lot of problems emerging. We expected to find some problems of course, but as the number of problems increased, the design of the CPU is going to be far more complex than the one displayed in figure 48. For example, instructions that can cause a branch to happen. As we’ve said before, multiple instructions will get into the CPU at one, now imagine this scenario, a comparison instruction got

into the CPU, now what will happen if the branch decision is false (branch is not taken)? The CPU will keep running and nothing wrong will happen. Now let's say a branch did occur, what will happen? The CPU will have to flush the data (instructions) inside of it before it can start executing again. Another example, let's say there's two instructions, one directly after the other inside the CPU. The first one is a loading instruction, and the second is any arithmetic instruction. The problem is, the second instruction will get different parameter/operands than the ones loaded from the memory, since the first instruction won't have enough time to successfully load the data coming from outside the CPU into the Register File. These are just two of many problems (hazards) we faced during the time we worked on designing the internal architecture of the CPU. After searching online and reading some chapters from the book "Computer Organization and Design RISC-V Edition" we found some solutions to some of the problems, but at the same time a lot of the them came with some kind of drawback or obvious flaws, and at the end we decided to move on from this five stage/section design and find a different one.

4.1.2 Second trial – Single Cycle CPU

For the second attempt at designing the internal architecture of the CPU, we used the same architecture introduced to us previously, but with some modifications. the previous design focused more on concurrency and parallelism. This alone introduced a lot of hazards and invited a lot of problems. To fix this issue, we decided to go with a Single Cycle CPU Architecture. This approach/design got rid of all the

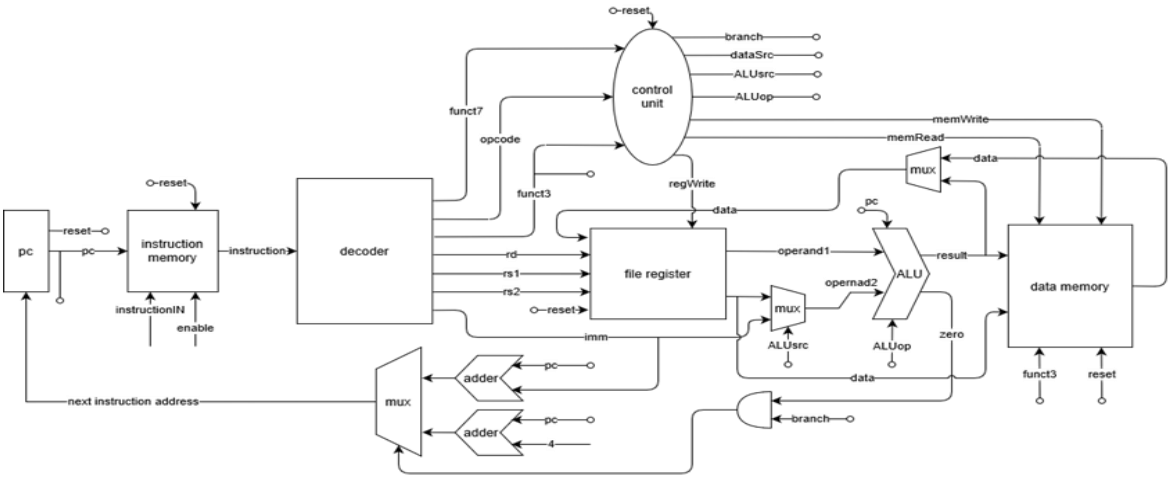


Figure 49 single cycle RISC-V CPU

problems that came with the multiple stage CPU architecture, since one instruction will get executed at a time. Obviously, this approach will have no parallelism or any kind of concurrency, but the difference in the time it takes a multiple stage CPU to execute one instruction versus the single cycle architecture is not that much. For this design, we decided that each instruction will get access to all stages at once, there's separation between the sections. To do so, all internal components of the CPU need to use combinational logic (no clock needed to get an output). Using combinational logic to design the internal components, it will make the CPU fast, as it will only take one clock cycle to execute an instruction. As shown in figure 49, we can see the initial design of the single cycle CPU. the CPU fast, as it will only take one clock cycle to execute an instruction. As shown in figure 49, we can see the initial design of the single cycle CPU.

4.1.3 Third trial – Register File

After we've displayed the initial testing for some of the parts inside the CPU in chapter 3, we have two main components left, the Register File and the Program Counter. We'll start off with the register file. The register file is essentially a small memory to store data of information inside the CPU. The register file consists of 32 memory elements, each one is 32-bit in length, and as shown in figure 50, we can see the name of each element (register) and what it is used for. Of course, this is the recommended use for these register (except for the x0 register, always gives value of zero). Now moving on, we designed the register file in python and used the MyHDL library for the Verilog conversion. What we noticed at first, the file register

Register	ABI Name	Description	Saver
x0	zero	Hard-wired zero	—
x1	ra	Return address	Caller
x2	sp	Stack pointer	Callee
x3	gp	Global pointer	—
x4	tp	Thread pointer	—
x5	t0	Temporary/alternate link register	Caller
x6–7	t1–2	Temporaries	Caller
x8	s0/fp	Saved register/frame pointer	Callee
x9	s1	Saved register	Callee
x10–11	a0–1	Function arguments/return values	Caller
x12–17	a2–7	Function arguments	Caller
x18–27	s2–11	Saved registers	Callee
x28–31	t3–6	Temporaries	Caller
f0–7	ft0–7	FP temporaries	Caller
f8–9	fs0–1	FP saved registers	Callee
f10–11	fa0–1	FP arguments/return values	Caller
f12–17	fa2–7	FP arguments	Caller
f18–27	fs2–11	FP saved registers	Callee
f28–31	ft8–11	FP temporaries	Caller

Figure 50 RISC-V Register File registers description

takes one cycle to output the contents of any register, since it was inspired by the register file showcased in the “Computer Organization and Design RISC-V Edition”. This does not go with the idea of a single cycle CPU. What we had to do is, redesign the output mechanism so that as soon as the signal (Addresses) of the desired register is delivered to the register file, the output should simultaneously get out of the file register. To do so, we changed the logic of the output module inside the register file to be combinational. As you can see in figure 51, at first, we stored some values into the register file (200 and 500 at address 21 and 22 respectively), and then selected these registers to be give out their contents. And as you can see, the output is delivered as soon as the signal arrives, and doesn’t need a clock cycle to give out the contents of a register.

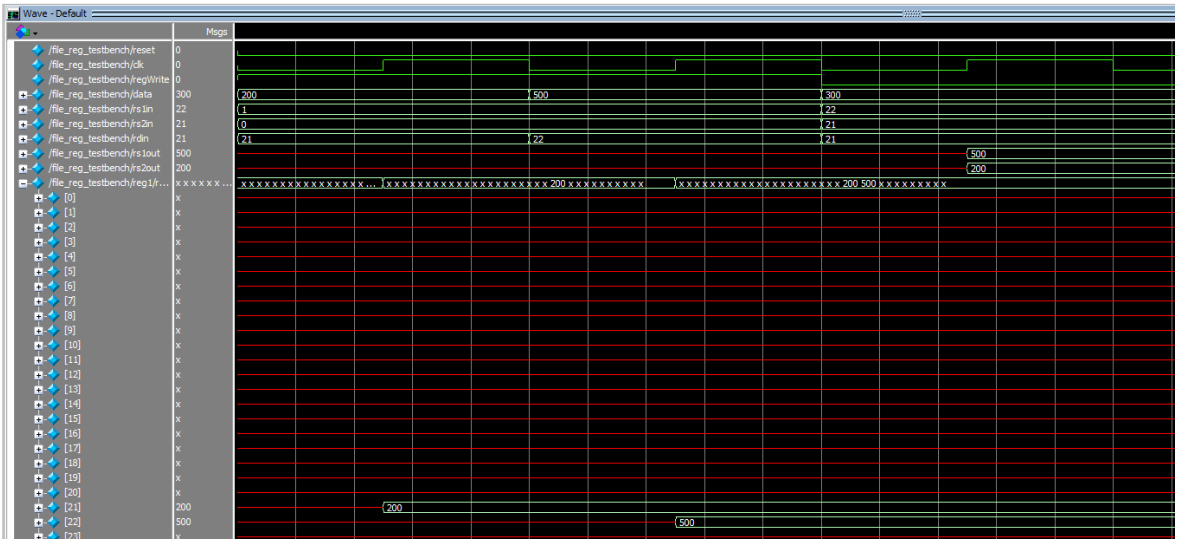


Figure 51 ROM simulation

4.2 ESSENTIAL TASK 2 / RAM

Now moving to the second essential task in the project, deigning the RAM module. For this task, we wanted to design a memory module that will allow random access to all its element at all time for the CPU. To do so, we made a simple module shown in figure 52, that has N amount of memory elements, with seven input signals, and one output signal.

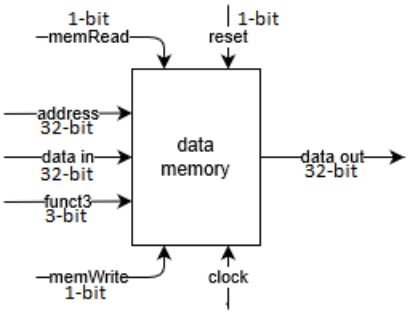


Figure 52 RAM block diagram

4.2.1 First trial – Regular RAM

After finishing the design and doing some tests, we figured out that there were some problems regarding this design. First problem is; this design does not go well with the Wishbone architecture, since the architecture we only have one WE signal responsible for reading/writing. The second and the biggest problem, is that this design performs poorly when trying to store data in it. The reason is, each element consists of 8 bits, and the data coming from the CPU is 32-bit long. So, when trying to store a 32-bit long data, we needed to repeat the store data instruction three more times after the initial store instruction. Also, we need to store 8-bit data at a time and change the address each time we store a byte! This design will obviously bottle neck the system, and we need to do something about it.

4.2.2 Second trial – Banked RAM

To fix the previous problem, we decided to use the concept of Banks to the RAM module. This concept was first introduced to us in the course EE-366, where we used the well-known PIC microcontroller. Banks are designed to separate the memory into N number of banks, in our case we went with four. Each bank has its own set of memory elements of 8-bit length. As shown in figure 53, we can see that the memory is separated into four different blocks, called banks, and each bank gets a part of the data to store. For example, if the CPU sends a 32-bit data, each bank

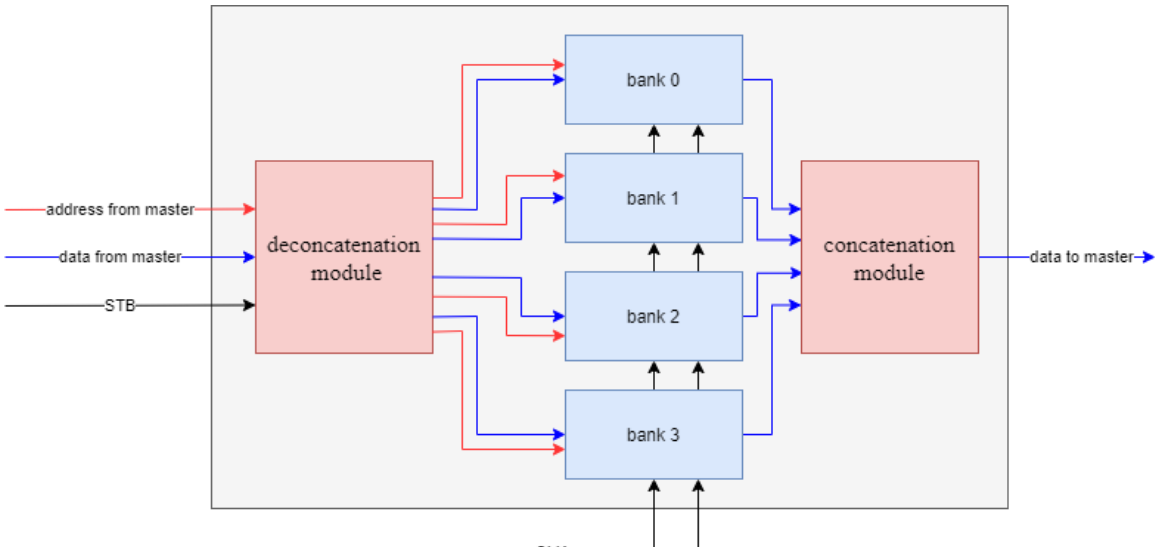


Figure 53 Banked RAM design

will get 8-bits to store. This can be done by the decontamination module before all banks. It works by getting a 32/16/8-bit data and an address at first, separating the data and giving each bank of memory a slice of this data. The second job to do is to shift the address given to the memory by a constant of 2. This is done to get rid of unnecessary bits, and to ignore any number that is not a multiple of four. For example, we want to store data at address 5, when we shift the value of 5 (0101) 2 bits to the right, it becomes (0001), which is the second element in each bank. By going with this design, we can avoid the problem of having to go through four different clock cycles just to store one word of data. The only drawback to this design, is that the user will have be careful just to use addresses that can be divided by four, otherwise it will get addressed to nearest multiplication of four.

4.3 ESSENIAL TASK / ROM

Rom, or Random-Access Memory, is the place where all the instruction are stored as zeros and ones ready to be executed by the CPU. With each clock cycle, a new instruction is outputted from the ROM to the CPU. Which instruction to output? It depends on the value provided by program counter module. Each signal from the program counter points at a certain instruction in the ROM. Each memory element in the ROM is 8-bits in length, and with each clock cycle, a constant value of 4 is added to the previous program counter value, or with any branch instruction, a specified address is calculated.

4.3.1 First trial – Simple ROM

In this project, we built the ROM as simple as possible, without sacrificing any of its functionalities. It is supposed to store machine code instructions provided by the user/programmer. These instructions are translated from the human-readable RISC-V assembly language to machine code. The translation can be done using any free RISC-V interpreters found online. In figure 54, we can see the design of the ROM module, and as we can see that the inputs/outputs are the same as we

Figure 54 simple ROM block diagram

discussed in previous chapters. When we initially designed the ROM, we did it in python, and the results from the simulations gave a positive output. But the problems occurred when we converted the code to Verilog. The first problem is that the generated code from python to Verilog, suggested that this a module with NO memory elements, which is completely wrong. And the second problem is, the MyHDL library does not support memory initialization files (.mif files).

4.3.2 Second trial –ROM using Banked RAM

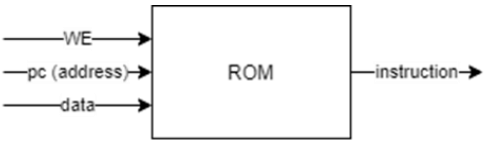


Figure 56 banked ROM

At the second attempt, we redesigned the ROM module to get generated correctly by the library, and to get synthesized correctly by Quartus. To do so, we changed the input signals of the ROM module and added two more signals, WE and Data signal. The problem is, the FPGA we're using doesn't have ROM, it only has RAM in it. So, we decided to implement a ROM module using RAM blocks, that's why we added these two signals. Otherwise, the synthesizer will look at as if it's just a combination of register, and it will use internal LUTs as the building block for the ROM. Obviously, the WE and Data signals are always set to zero, so the write won't ever get enabled. The new design of the ROM can be viewed in figure 56. As for the second problem,

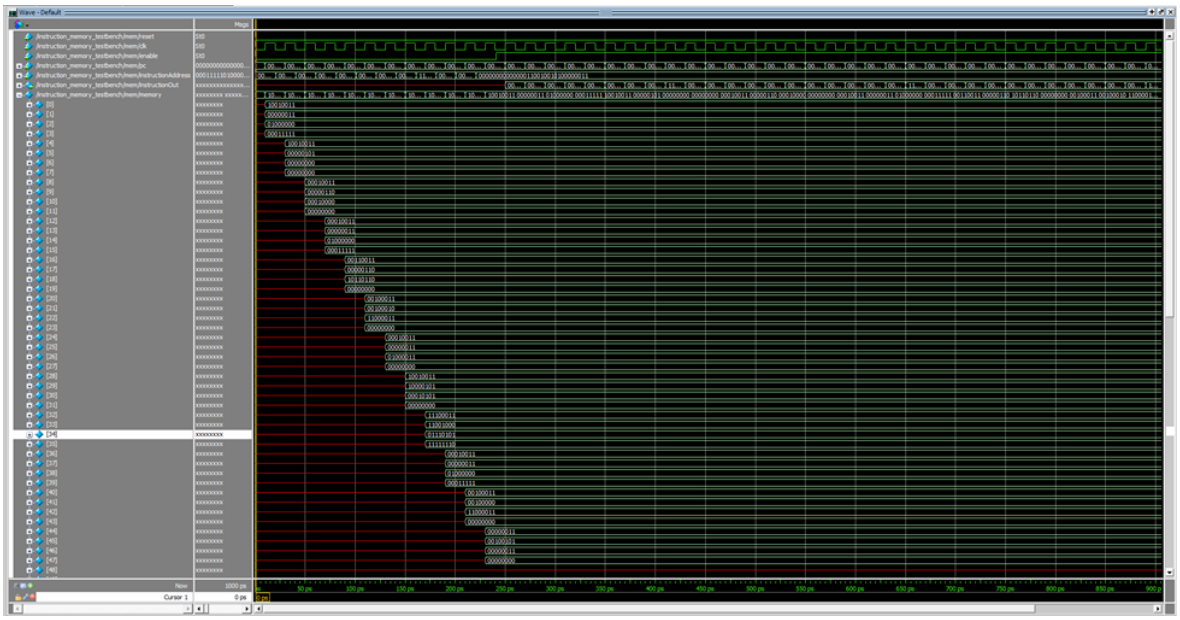


Figure 55 Banked ROM simulation using ModelSim

we couldn’t find any solution other than adding the memory initialization instruction by hand, since MyHDL library doesn’t support it. The mif file can be initialized as shown in figure 55. By using this code, we can upload the machine code instructions to the memory before the CPU starts executing instructions. In figure 55, we can see from the ModelSim simulation, the ROM contains the data (instructions) provided by the memory initialization Verilog code shown previously.

4.4 FINAL PRODUCT

As we come to the final section in this chapter, we are happy to showcase the final product. In general, we aimed to design a fully functional RISC-V microcontroller on an FPGA, and in this section, we’ll display our final result. To start things off, we connected all different modules with each other, as displayed in figure 58. We started off with designing the microcontrollers peripherals, components and controllers in Python using MyHDL library. During the designing process, we tested all of them in Python to make the best of MyHDL capabilities. After that, we used the MyHDL to generate Verilog code that will later synthesized by Quartus to be loaded into the MAX 10 FPGA. As you can see in figure 57,

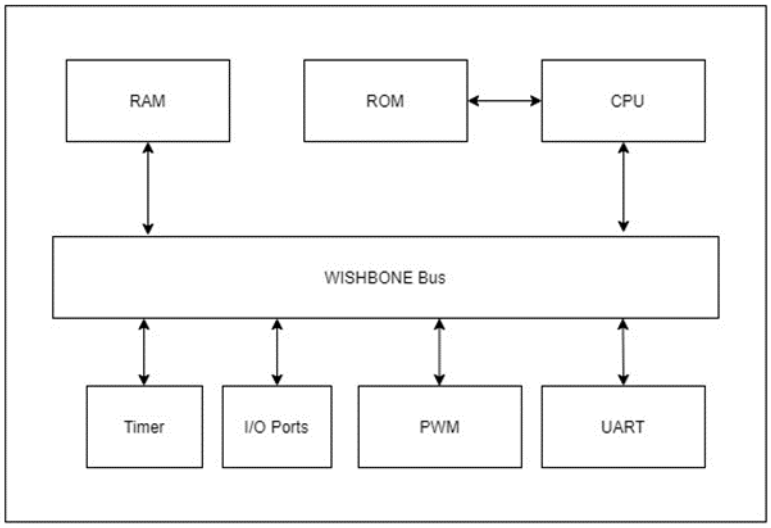


Figure 58 block diagram of the whole microcontroller system

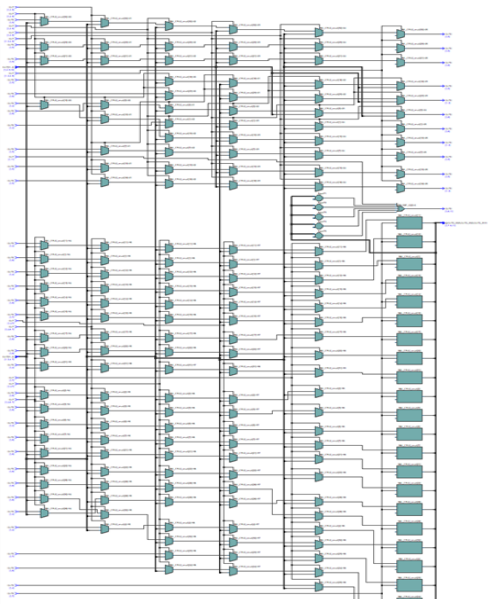


Figure 57 result from RTL viewer

the result from the RTL viewer doesn’t look anything like the block diagram of the design (CPU and RAM only, for simplicity), and this is due to the Python to Verilog conversion done by MyHDL since the whole design is put into one module. The result from synthesizing the Verilog code, and running it on an FPGA, has the same functionality as the microcontroller in figure 59. In figure 59, we connected each module we created in Python ourselves to see if it makes any difference when connecting them each one in a separate module in Verilog. The result from running multiple simulations on both the generated code and the code we wrote ourselves are exactly the same, the only difference is the RTL diagram. At the end, the final design of the microcontroller proved to be a success. The microcontroller contains a Central Processing Unit, a Random-Access Memory, Read-Only Memory, Universal Asynchronous Transmitter/Receiver, Timer, I/O Ports, Pulse Width Modulator and a Wishbone Bus System. All these parts are connected together to

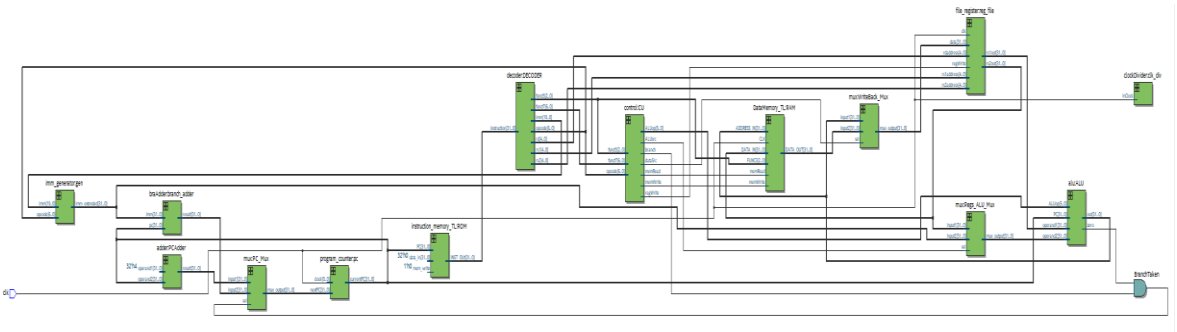


Figure 59 microcontroller RTL viewer result for separated modules

form the microcontroller.

CHAPTER – 5 RESULTS, DISCUSSION, AND CONCLUSION

5.1 RESULTS AND DISCUSSION

This section will display the result of multiple testing and experimentations done on the final product. The aim of this section is to validate the end product by using different methods and programs that will use the full potential of the microcontroller. Also, at each sub-section, we'll display the inputs and outputs of each procedure and what is the conclusion we get from them.

5.1.1 Test 1 – UPCOUNTER

In this sub-section, a validation test is made to make sure can do even the simplest task, such as a counter. The counter will count from 0 to F in hexadecimal, the numbers will be displayed on the 7 segment Display and LEDs on the FPGA. This test will validate the functionality of the CPU and its internal components. In figure 60, it is showing a RISC-V interpreter, the interpreter will simulate a functionality of a RISC-V CPU. The first instruction is an add immediate instruction. This instruction will add the value of 15 to x1 register in the register file. Then, the CPU will get inside

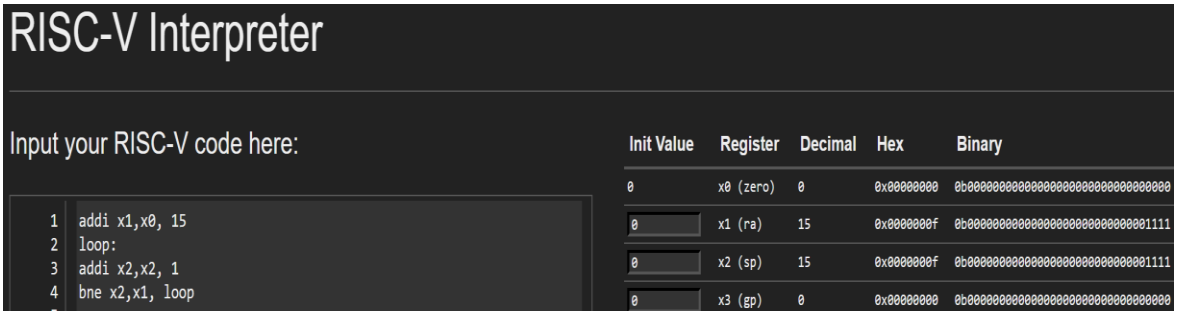


Figure 60 RISC-V interpreter simulation result

a loop. Inside the loop, a constant of 1 is added to the contents of register x2, and saved in register x2. This will continue until the branch not equal (BNE) instruction on line 4 is satisfied to get out of the loop. To make sure the final product is getting the correct results, the same instruction will be used in the product, and the value of the Write Back signal will be displayed on the 7-segment. The images below will show the FPGA's 7-segment display working for every 8 counts.

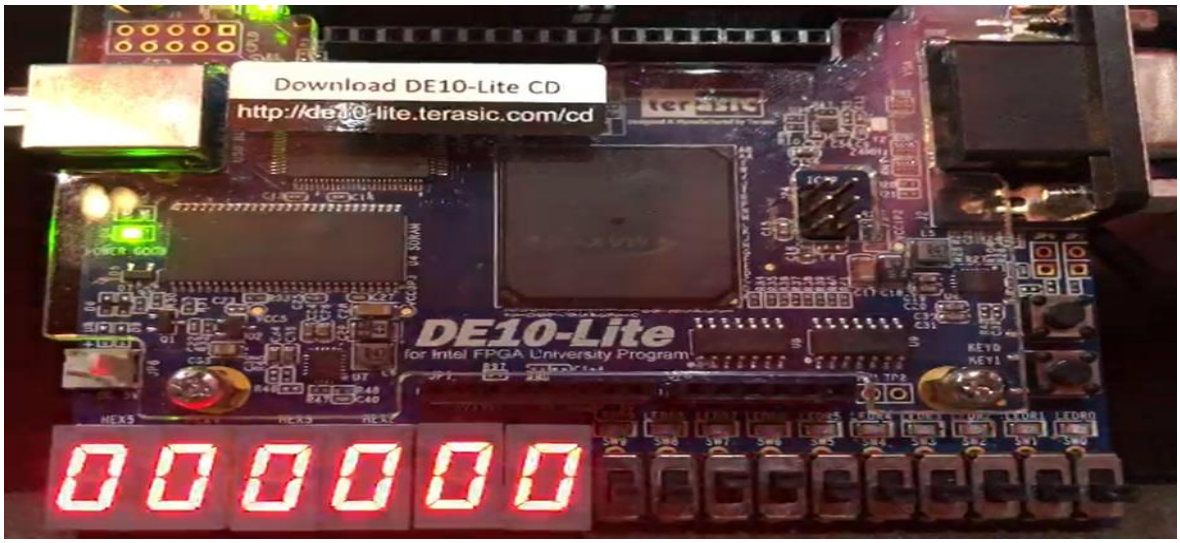


Figure 61 counter on FPGA (1)

The first image is showing the 7-segment displaying starting the count from 0.

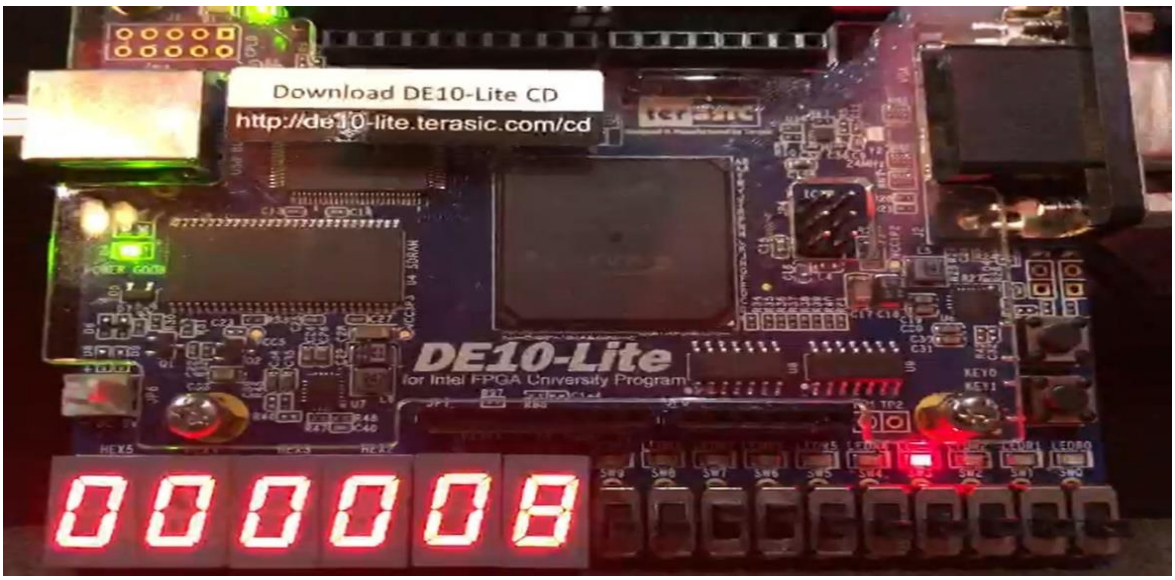


Figure 62 counter on FPGA (2)

The second image shows the 7-segment display counted to 8 in hexadecimal.

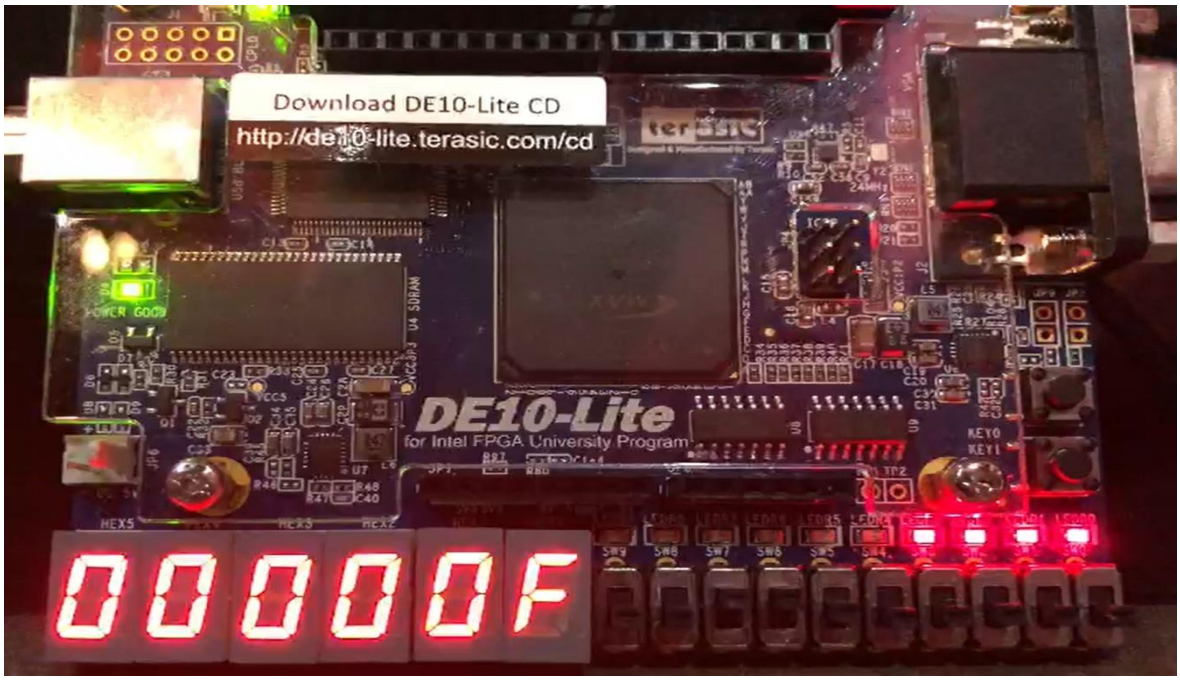


Figure 63 counter on FPGA (3)

Lastly, the third image shoes that the program has finish and counted to F in hexadecimal.

5.1.2 Test 2 – Store/Load, Loops, Comparisons

In this sub-section, several instructions will be tested both arithmetic and logical (figure 64), what these set instructions will do is first add a constant value and store it in register x7, add 0 to register x11 and x6, constant of 1 added to register x12, then it enters in a loop, in the loop it will adding the values in register x11 and x12 and save it back to register x12, next register x6 will store a word with an offset of 4 then increment by a value of 4. Also, register x11 is then incremented by 1 after these instructions there is a branch is less than instruction (BLT) will check if the value in register x11 is less than the value

```
1  addi x7,x0,500
2  addi x11,x0,0
3  addi x12 ,x0,1
4  addi x6,x0,0
5  loop1:
6  add x12,x12,x11
7  sw x12,4(x6)
8  addi x6,x6,4
9  addi x11,x11,1
10 blt x11,x7,loop1
11 addi x6,x0,1
12 sw x12,0(x6)
13 lw x10,0(x6)
14
15 loop2:
16 blt x0,x10, loop2
```

Figure 64 test 2 code

0	x6 (t1)	1	0x00000001	0b00000000000000000000000000000001
0	x7 (t2)	500	0x000001f4	0b00000000000000000000000111110100
0	x8 (s0/fp)	0	0x00000000	0b00000000000000000000000000000000
0	x9 (s1)	0	0x00000000	0b00000000000000000000000000000000
0	x10 (a0)	0	0x00000000	0b00000000000000000000000000000000
0	x11 (a1)	500	0x000001f4	0b00000000000000000000000111110100
0	x12 (a2)	124751	0x0001e74f	0b00000000000000011110011101001111

Figure 65 test 2 interpreter result

of register x7. These are the results when the RISC-V interpreter Is used to verify and achieve the correct results when using the instructions in figure 65. After making sure that the results appeared correctly in the interpreter without any errors. Now

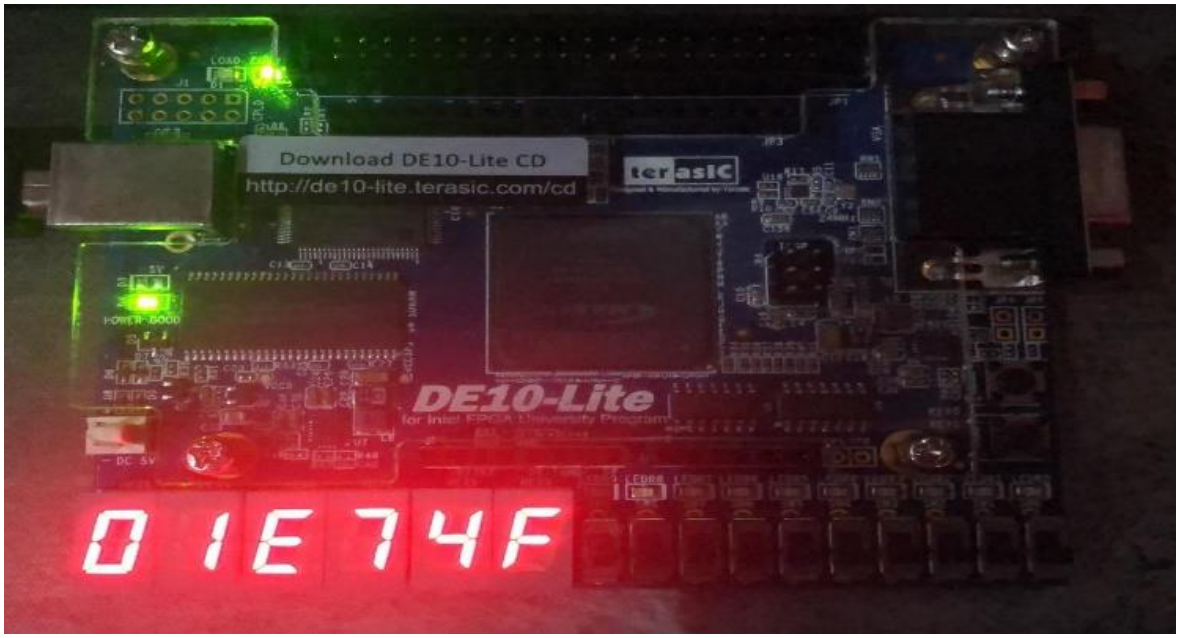


Figure 66 test 2 FPGA result of write back signal

what's left is to make sure that by using the same instructions we'll get the same the result on the FPGA. In figure 66, we can see the result of running the same code on our microcontroller. From the 7-segment, we can see the number 1E74F. which is 124751 in hex. From this, we can say that the communication between the CPU, SBA Bus and RAM is done perfectly since the result matches the result from the interpreter.

5.1.3 Test 3 – Timer, PWM, UART and I/O Port

For this section, we'll be testing the peripherals inside the microcontroller. First, we'll start with the PWM module. this module is expected to output a pulse modulated signal with a specified time period and an active period. The time period represents the time of the whole period (low and high). On the other hand, the active time represents the time of the high signal alone. For this simulation, we configured the period to be 20, and the active time to be 9. As you can see in figure 67, the countdown counter starts from the value of the period (14 hex = 20 decimal), and starts counting down until it reaches zero. Also, one of the LEDs indicates the state of the PWM module, and in this case it's in the running state, meaning the counter will keep decrementing.

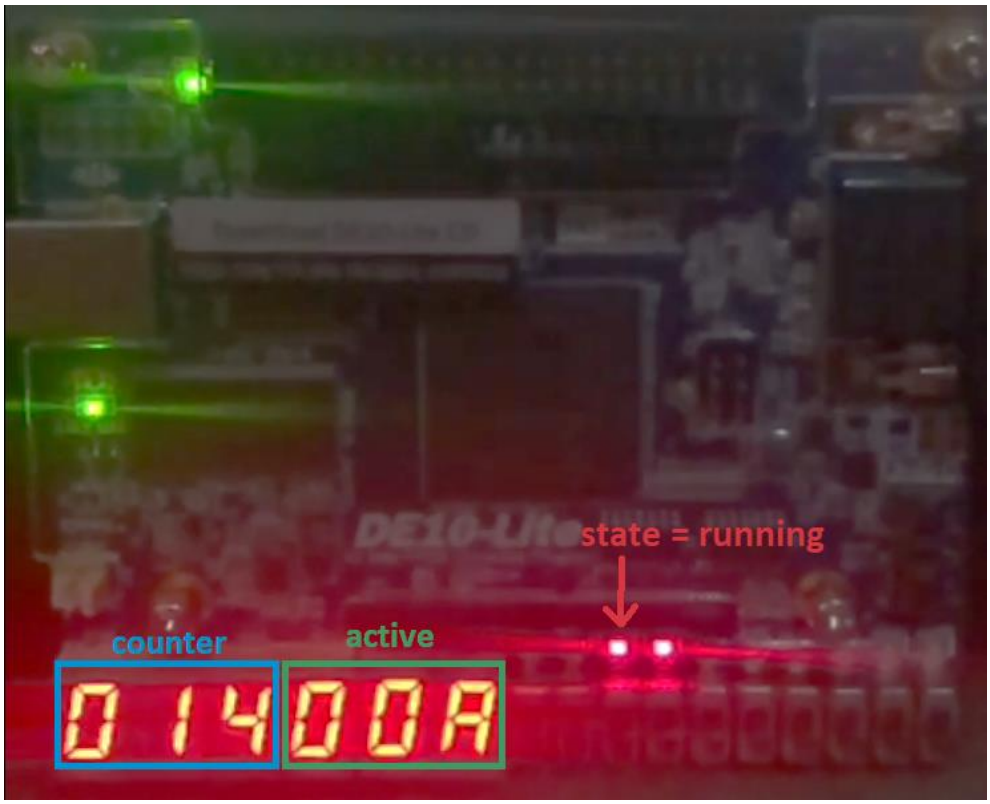


Figure 67 PWM simulation #1

At the same time, when the counter reaches the value of 9, the PWM signal becomes high, as you can see in figure 69. lastly, the counter will reset after it reaches zero indefinitely until the CPU orders it to stop or a reset is issued.

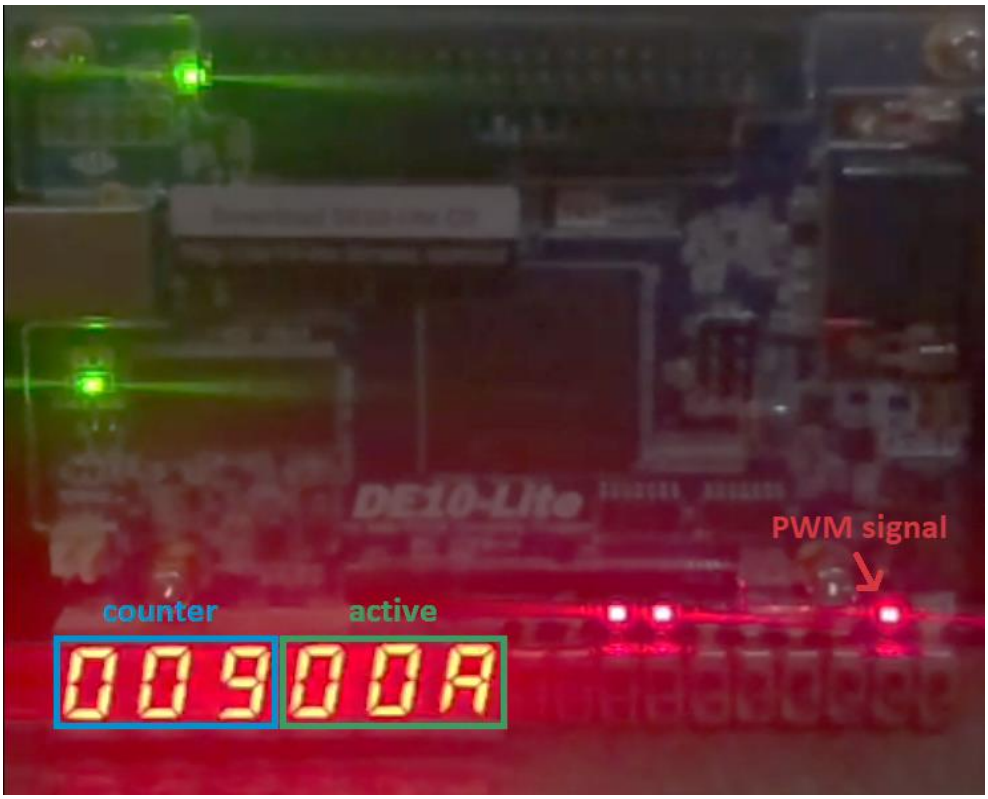


Figure 69 PWM simulation #2

Next, we have the output port. For the output port, the CPU generated the value of 8 to be outputted from the microcontroller. The functionality of this port is simple, the CPU sends an 8-bit data to the output port, then the output port just has to store it,

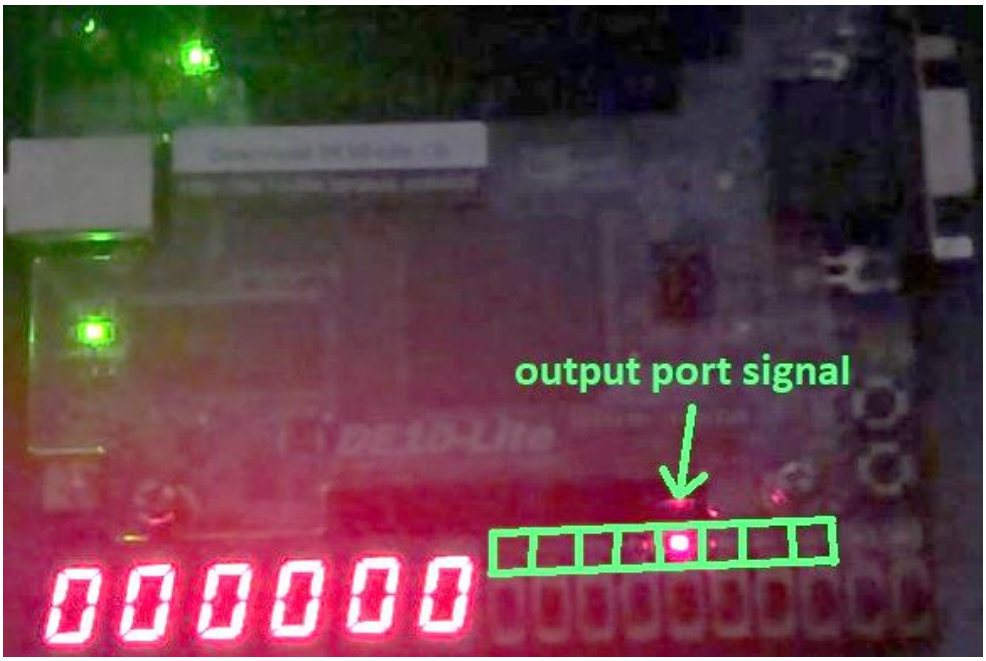


Figure 68 I/O port simulation

and transfer it outside of the microcontroller. As you can see in figure 68, the

highlighted 8 bits represent the signals coming from the output port. The fourth bit from the right is high, while the rest are set to zero, this means that the value coming from the output port is 8.

Furthermore, the Timer module is fairly simple, the Timer contains an 8-bit counter, and it starts counting from the value that the CPU provides. The CPU specifies a base value for the timer to start from. For this experiment, the CPU sent the value of AA (170 decimal) to the Timer module. so, it is expected from the timer to start counting from this value until it reaches FF. After that, the Timer flag should be high, and the counter resets back to 0. From figure 70, we can see that the initial value of both the counter and the register that holds the initial value, and both of them at the start hold the same value.

Figure 70 Timer Simulation #1

After that, the counter will start to increment until it reaches the value of FF and it resets, and the Timer flag will be high, as in figure 71.

Figure 71 Timer simulation #2

Lastly, the UART module. for this experiment, we want the UART to receive data from the CPU, containing the baud rate, and the message that we want to send outside the microcontroller. In figure 72, we can see that the CPU sent a 32-bit data to the UART that contains both the baud rate initial value to be 2, and the message

Figure 72 UART simulation #1

to send it AA (1010 1010 binary). At first, the Tx module will wait for ticks coming from the Baud Rate Generator to start shifting one bit at a time starting from the LSB in the register that holds the value becomes zero. In figure 73, we can see that one

Figure 73 UART simulation #2

bit got shifted and outputting from the Tx module by at the value of the Tx signal. Also, the value of the Tx register becomes 55 (0101 0101), which means that the LSB is out from the Tx module.

Lastly, in figure 74, we can see that after the Tx module sends the whole 8-bit message, the Tx module will set the Tx signal back to 1, which means that the Tx module sent the whole 8-bit message and this is the stop bit.

Figure 74 UART simulation #3

5.1.4 Test 4 – System Builder

Designing a system builder that will simplify the customization of the microcontroller is one of our main objectives in this project. Our design for the system builder requires minimum user interaction, all the users should do is choose the number of modules he needs from each peripheral starting from RAM, PWM, output ports, input ports, TIMER and UART respectfully. After inputting the requested number of peripherals, the system builder will generate a python file named “SBA_SYSTEM_GENERATED.py” which contain the code for the top-level module in addition to the decoder file “SBA_DECODER_GENERATED.py” and the bus system file “SBA_BUS_GENERATED”. Since the project is using RISC-V instruction set architecture, interacting with the peripherals will be via using store and load instructions only so the user will need the addresses for each peripheral. The system builder will provide the user with the address of each peripheral via a comment in the decoder python code (figure 75).

```
elif ADDR_I == 1028:
    STB_O.next = concat(intbv(8)[6:], STB_I[4:1])  # output_port_0 || address = 1028
elif ADDR_I == 1029:
    STB_O.next = concat(intbv(16)[6:], STB_I[4:1])  # timer_0 || address = 1029
elif ADDR_I == 1030:
    STB_O.next = concat(intbv(32)[6:], STB_I[4:1])  # pwm_0 || address = 1030
```

Figure 75 address of peripherals

Requirements:

1- The user system should have python 3 installed
2- The user system must have MyHDL library installed
3- The System_Builder.py file must be in the same directory of the CPU and the peripherals files

Constraints:

1- The system must not contain more than **one** RAM
2- The system must not contain more than **one** UART

How to use:

In order to build the system builder properly, Move the “System_Builder.py” file to the same directory as the CPU and peripherals, Run the system builder script via command line (cmd) using the following command in figure 76. After providing the name of the script “System_Builder.py” the user should choose the number of modules he needs from each peripheral starting from RAM, PWM, output ports, input ports, TIMER and UART respectfully. i..e, python System_Builder.py (RAM) (number of PWM) (number of output ports) (number of input ports) (number of timers) (UART).

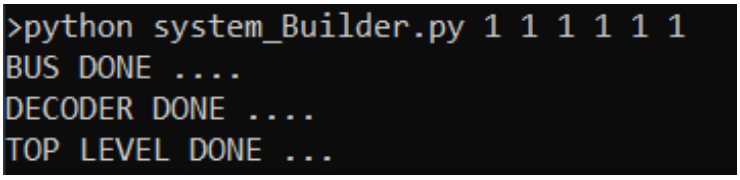


Figure 76 the result in the cmd

5.2 EVALUATION OF SOLUTIONS

5.2.1 Technical Aspects

During this project, our main aim is to make sure that all the customers’ needs and objectives are satisfied, from low level to high level objectives. With this in mind, the work shown in chapters 3 and 4, from simulations to actual implementation on actual hardware (FPGA), displayed the state we’re at after we’ve completed the project. Based on the results from all testing and validation procedures done on the final product, we satisfied all three high level objectives and all three low level objectives. The testing in chapter 4 displayed the functionality of different components and peripherals inside the microcontroller. The CPU managed to handle different programs, which used different types of RISC-V instructions. The ROM showed its capability of storing instructions from mif files for later execution by the CPU. RAM displayed the ability of storing and outputting correct data provided by the CPU. The Timer peripheral also showed successful implementation of a timer that will count from a certain value until it reaches the maximum value of its internal counter (8-bit in our case). As for the UART module, from the simulations done on this module, we can say that it displayed the ability of transmitting and receiving

data asynchronously. With the I/O modules, both of them showed their capability of inputting/outputting data to/from the microcontroller. Lasty, the PWM also showed that it can send pulse modulated signal from its output port according to a specific timing configuration. All in all, all these components working together form the microcontroller. Furthermore, the system builder we designed resulted in a successful creation of the bus system, top level module and the SBA decoder. The only thing that needs improving regarding the system builder, is adding a friendly user interface. As for the microcontroller, the current design met all the requirements of the customer.

5.2.2 Environmental Impacts

After researching on the FPGA which is the only physical component in the design. One of the important positive impacts of the FPGA is that it is reprogrammable, which allows it to be reused and can be used more than one time for various tasks. Because it can be reprogramed it has the aspect of flexibility for prototyping. For this reason, there is no need to manufacture different prototypes, this, in turn, can reduce the excess waste of electronic devices. This means that when an FPGA is used it can prolong the end of life of these integrated circuits compared to other integrated circuits e.g., ASIC. Looking at the negative impacts of the FPGA, it was found that in the FPGA is an Integrated Circuit, which means that these Integrated Circuits are manufactured in factories. When manufacturing these components, they expose human beings who work in a factory to dangerous chemicals. Also, at the end of life for the components, these combined integrated circuits are generally dumped in poor countries which means hazardous materials are released on that location and negatively impact the environment around it [14].

5.2.3 Safety Aspects

5.2.2.1 FPGA

The project is mainly done in software, other than the FPGA, every component needed to complete this project is designed, tested and validated completely in software with no hardware intervention. As for the safety of this project, the project can be used by anyone, with or without experience with microcontroller or FPGAs. The only electrical part as we said is the FPGA, and according to Intel's documentation that came with the FPGA, it only needs 5V input to start the MAX 10 FPGA device, and the maximum voltage it can produce is 3.3V, which is considered safe.

5.2.4 Financial Aspects

The project does not have many hardware components, as it only contains one hardware component which is the Field Programmable Gate Array or FPGA. The FPGA has a market price of 260 Saudi Riyals which is the average price of the components, but it may differ given there are different types of FPGAs. The rest of the project work is in software, so it is free, and every software tool being used is free of licensing fees. Considering the labor cost, it takes most of the project cost, as it is 2 Saudi riyals per hour, and when measured with the total length of the project and multiplied by the number of workers, it accumulates to be 2160 Saudi Riyals.

Table 21 final project cost analysis

Item	Quantity	Cost (SAR)
FPGA DE10-Lite	1	260
Labor	-	3*(2SAR/h*6h*5d*12w) = 2160
Open-source RISC-V Instruction Set Architecture	Free	0
Open-source WISHBONE bus Architecture	Free	0
	Total Cost	2420

The effect that this project will have on the market of microcontroller design is expected to be significant, as it brings a new path of the microcontroller to life, as we think in the short term, people will not be used to it and will not want to leave their already made designs. Although, they will cost more money as opposed to our project since it will be free. But in the long term, we see this project can change the direction of the development of microcontrollers and lead them all to be open source with no licensing fees at all. Because this project is open-source so there is no actual business aspect as the design will be published online as open-source, so there is no profit.

5.2.5 Social Impacts

Considering the prices of other microcontroller designs that are available, our product is free, which not true for most microcontrollers. Considering the capabilities of the project in changing the market of microcontroller designs, this project can improve the local market in Saudi Arabia, as well as help start-up companies. They can use this product to help create their own companies and base it on this new and open-source microarchitecture. Also, our microcontroller is KAU branded, so it will help future students in developing their microcontroller designs.

5.2.6 Global Impact

As for the global impact of the design of the microcontroller, this design is the first of its kind here in Saudi Arabia. Also, it is one of the few fully functional open-source microcontrollers that combines ease of customization and easier functional simulation methods. Furthermore, it will impact small companies that are required to pay fees for microcontroller designs and tools. For small companies, they cannot modify microcontroller designs to make their separate design and use it for their start-ups. this is where the impact of the products comes in and helps these companies and future engineers around the world.

5.3 CONCLUSION

In the long run, start-up companies, universities and researchers, will move from the most common way of working with microcontrollers. Working with big microcontroller companies that require fees to use their microarchitectures, designs and tools can be expensive for most of their users. As we've shown in previous chapters, these fees can cost thousands of dollars annually, and most start-ups or students can't afford to pay such amount. In addition, for students who aren't familiar with microcontrollers, building and designing one can be a tedious task.

For this reason, we as computer engineers worked with our customers' specifications and requirements to find a suitable solution for these problems. As a result, we decided to start to build our own open-source RISC-V ISA based microcontroller design with its own system builder program. The solution consists of multiple designs for different components inside the microcontroller, which includes a fully functioning CPU, RAM and ROM, and some additional peripherals such as a UART, Timer, I/O Ports and a PWM module. Together, these components combined form the microcontroller. As for the way of designing, testing and validating the final product, we used Python, with MyHDL library for the ease of use when performing different types of simulations.

Altogether, we achieved the main objective of our project, which is to design an open-source RISC-V microcontroller and testing it on actual hardware (FPGA). At the end, we fulfilled all the low- and high-level objectives and met all the requirements set by the customer. Many simulations and validation procedures (visit chapter 3 and 4) were conducted to make sure the final product works as expected.

Finally, we would recommend for future students, researches, start-ups or anyone interested in improving or working with our project to look into making the main processor more powerful by redesigning it and making it pipelined instead of single cycle design. Also, since this microcontroller has different peripherals, an interrupt system should also be implemented. Lasty, we would also recommend adding more peripherals to the microcontroller as it would more functionalities to the system to be used for more applications.

REFERENCES

[1] P. Koopman, "Instruction Set Architecture (ISA)," [Online]. Available: http://www.cs.kent.edu/~durand/CS0/Notes/Chapter05/isa.html.

[2] intel, "intel FPGAs Resources," [Online]. Available: https://www.intel.com/content/www/us/en/products/details/fpga/resources/overview.html.

[3] cs, "How Computers Work: The CPU and Memory," [Online]. Available: How Computers Work: The CPU and Memory.

[4] Tutorialspoint, "Computer - Read only Memory," [Online]. Available: https://www.tutorialspoint.com/computer_fundamentals/computer_rom.htm.

[5] "Memory," [Online]. Available: https://erg.abdn.ac.uk/users/gorry/eg2068/course/mem.html. [Accessed 17 5 2021].

[6] S. Campbell, "Circuit Basics," [Online]. Available: https://www.circuitbasics.com/basics-uart-communication/. [Accessed 17 5 2021].

[7] Pearson, "Pearson It Certification," [Online]. Available: https://www.pearsonitcertification.com/articles/article.aspx?p=1681059#:~:text=I%2FO%20ports%20allow%20for,ports%20include%20USB%20and%20FireWire.. [Accessed 17 5 2021].

[8] "Firewall," [Online]. Available: http://www.firewall.cx/networking-topics/cabling-utp-fibre/120-network-parallel-cable.html. [Accessed 17 5 2021].

[9] "mepits," [Online]. Available: https://www.mepits.com/tutorial/143/vlsi/hardware-description-language. [Accessed 17 5 2021].

[10] "javaTpoint," [Online]. Available: https://www.javatpoint.com/verilog. [Accessed 17 5 2021].

[11] "Semiconductor Engineering," [Online]. Available: https://semiengineering.com/knowledge_centers/eda-design/definitions/register-transfer-level/. [Accessed 17 5 2021].

[12] ARM , "ARM Licensing," ARM Company, [Online]. Available: https://www.arm.com/why-arm/how-licensing-works. [Accessed 4 12 2020].

[13] J. Shepard, "MicrocontrollerTips," [Online]. Available: https://www.microcontrollertips.com/risc-v-vs-arm-vs-x86-whats-the-difference/. [Accessed 16 5 2021].

[14] R. Gal, A. Golda, M. Frankiewicz and A. Kos, "ieeexplore.ieee.org," 8 9 2011. [Online]. Available: http://0o10a685w.y.https.ieeexplore.ieee.org.kau.proxy.deepknowledge.io/stamp/stamp.jsp?tp=&arnumber=6015934. [Accessed 5 12 2020].

[15] E. Gür, Z. E. Sataner, Y. H. Durkaya and S. Bayar, "ieeexplore.ieee.org," 20 6 2019. [Online]. Available: https://ieeexplore.ieee.org/document/8742406?denied=. [Accessed 5 12 2020].

[16] myHDL, "myHDL," [Online]. Available: http://docs.myhdl.org/en/stable/manual/preface.html. [Accessed 6 12 2020].

[17] OpenRISC, "opwnRISC project overview," [Online]. Available: https://openrisc.io/. [Accessed 6 12 2020].

[118] Valentin, "airfocus blog," [Online]. Available: https://airfocus.com/blog/weighted-decision-matrix-prioritization/.

[119] T. Instruments, "TI," [Online]. Available: https://www.ti.com/lit/ug/sprugp1/sprugp1.pdf?ts=1607250396247&ref_url=https%253A%252F%252Fwww.google.com%252F. [Accessed 6 12 2020].

[120] Intel, "Intel," [Online]. Available: https://www.intel.com/content/dam/altera-www/global/en_US/portal/dsn/42/doc-us-dsnbk-42-2912030810549-de10-lite-user-manual.pdf. [Accessed 3 12 2020].

[121] S. S. B. L. Margaret Rouse, 9 2019. [Online]. Available: https://internetofthingsagenda.techtarget.com/definition/microcontroller. [Accessed 1 12 2020].

[122] WISHBONE, "cdn.opencores.org," 7 9 2002. [Online]. Available: https://cdn.opencores.org/downloads/wbspec_b3.pdf. [Accessed 15 11 2020].

[123] IEEE, "ieeexplore.ieee.org," 29 9 2017. [Online]. Available: https://ieeexplore.ieee.org/document/8055462. [Accessed 15 11 2020].

[124] IEEE, "ieeexplore.ieee.org," 7 4 2006. [Online]. Available: https://ieeexplore.ieee.org/document/1620780?denied=. [Accessed 15 11 2020].

[125] D. Prof. Ron Lasky, "dartmouth.edu," [Online]. Available: http://www.dartmouth.edu/~cushman/courses/engs171/ElectronicsIndustry.pdf.

[26] J. L. H. David A. Patterson, Computer Organization and Design RISC-V, Morgan Kaufmann; 1st edition, May 12, 2017.

[27] J. K. J. B. K. L. Charles H. Roth, Digital Systems Design Using Verilog, Cengage Learning, 2016.

[28] M. S. P.Fiedler, "Realiabilty and Safety Issues of FPGA Based Design," p. 6, 2012.

APPENDIX – A: VALIDATION PROCEDURES

EXPERIMENT #1:

Introduction

In this report, a major part of the project is chosen and tested using a proper workplan that will test all the scenarios that could happen in this major part to make sure it is working as theorized. The major part that is experimented on in the report is the Data Memory or for whatis more known for as the RAM. The Random-Access Memory in any system is a vital part is an important part in the microcontroller that is used to store values that are then later used in the microcontroller. Throughout this report the findings that are achieved and the validation of these results will be shown in this report.

Background

As was shown in our baseline design, one of the main components that will be validated and experimented on is shown in this report. The RAM (DataMemory) is used to store the values given to it and will stay in the data memory until it is either overwritten or replaced (Figure 1). Then depending on the need of the stored values in the CPU,these stored values will be sent throughout the system where it is needed. The objective of the experimentis to make sure that the values are being

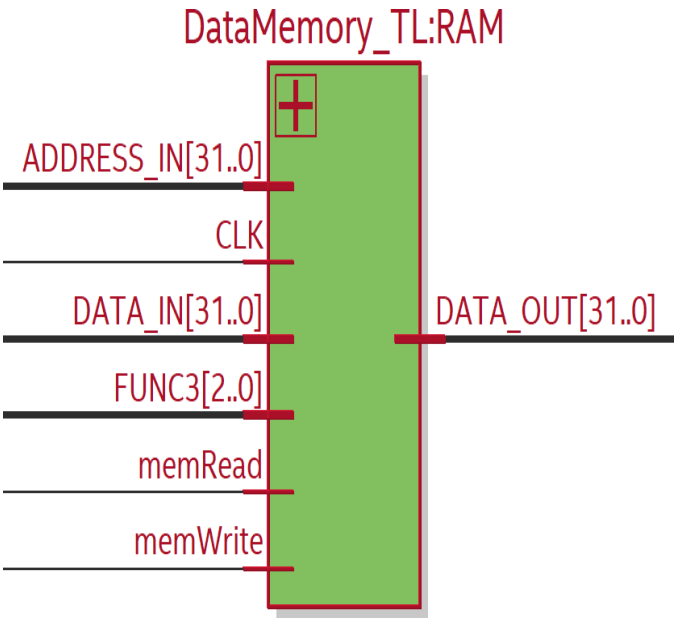


Figure 1 Data memory Design in Quartus

stored in the data memory are stored correctly and when leaving the data memory are outputting the correct data. The data memory is simulated in the Quartus IDE using Verilog HDL (Hardware Description Language). It has 6 inputs and 1 output, the inputs are the clock, DATA_IN, memRead, memWrite, FUNC3 and ADDRESS_IN. As for the output it's only onewhich is called DATA_OUT.

Experiment

In this section of the report what will be shown is the tools used in the experiment and its specifications. The work plan that is detailed containing all the necessary steps for the experiment. This experiment will allow us to validate that results that are achieved in the data memory to make sure thevalues are stored correctly. The experiment is not to test the interpreter but to test the correctness of the modules that we created in both pythons and Verilog HDL.

Objectives

- To correctly convert the Python code that uses MyHDL library to Verilog HDL, to make sure that no modification are needed in the converted code
- To use the RISC-V interpreter to get the result correctly as possible and then compare it to the results of our own design of the RAM.
- Compare the python and Verilog ram modules against the RISC-V interpreter to check the accuracy of the result.

Tools

- Quartus IDE: is a design software that provides a design environment that adapts to specificuser design.
- Verilog HDL: Verilog itself is an HDL, it is used for describing electronic circuits and systems. It is used for simulating the user specific design.
- Documentation of Verilog HDL
- Python MyHDL library [1]
- RISC-V interpreter [2]
- ModelSim [3]

Work Plan

In order to check the behavior of our design, we will build a test-bench of the design for the Data Memory and to make sure that the result that are outputted are correct a RISC-V interpreter will be used. In this plan we will go through the procedure step by step to clearly show the experimental design.

Plan:

1. From the Created Data memory module that was created using MyHDL library in python a testbench was created to test the modules design,
2. Run the testbench with the instruction needed to store and load the files.
3. Take note of the results in the python.
4. Now use a RISC Interpreter found online to put the instruction of the store and load and takenote of the results.
5. Convert the data memory modules made in python to Verilog HDL
6. Upload the data memory module and its testbench to the Quartus IDE
7. Compile the design after uploading the modules to the Quartus IDE
8. After the design is compiled in the assignments tab through its simulation setting will initializethe testbench in order to simulate the design.
9. The test bench has an order in which to test the data
    - First we will initialize the inputs
    - Then we will start testing by changing the values of the inputs
    - the inputs will that influence the Simulation are:
        - FUNC3
        - memWrite
        - DATA_IN
        - ADDRESS_IN
        - memRead
10. we will apply 2 writing values that come from data in to test that these values are stored
11. Lastly use the ModelSim to simulate the design

Following this work plan we will be able to simulate our design and make sure that the data that is being inputted through these variables mentioned in work plan. The step that are taken in the work plan are tgeprocedures taken to create and perform

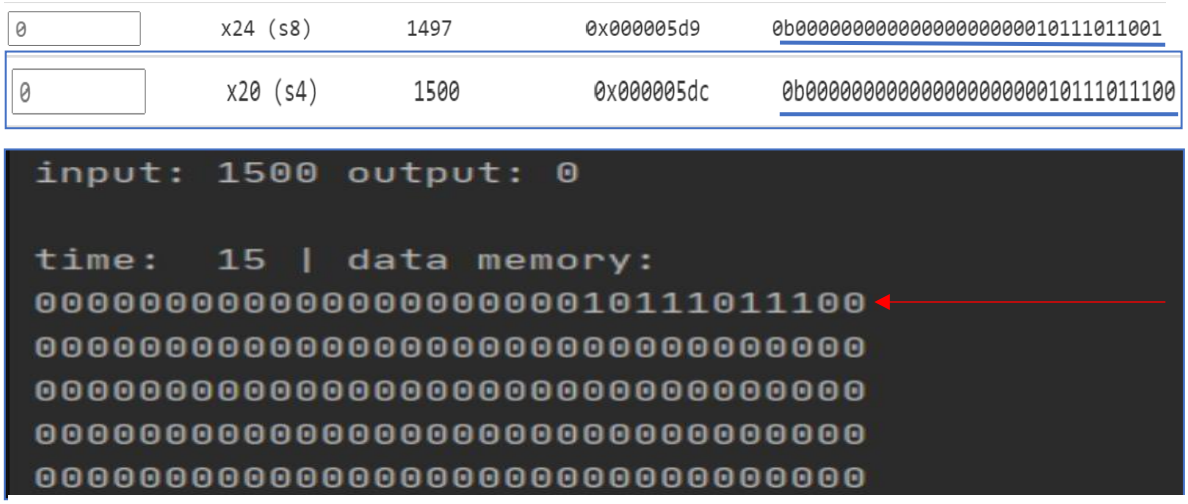
the experiment, this is very important because these values needto be correct as assumed before the experimentation.

Results

In this section we will go through the results of the work plan going one by one and analyses the simulation result from python and from Quartus. In these results we will compare them withthese definite results from the RISC-V interpreter. We will first compare the simulation from python and the RISC-V interpreter to make sure that the result that python is generating are correct. Next, we will compare the simulation using Quartus IDE and ModelSim which is usedfor the simulation and compare the results with the RISC-V interpreter to check if the results are correct. This will help in making sure that the conversion from MyHDL to Verilog HDL iscorrect and that the simulations are working as expected.

RISC-V Interpreter vs Python MyHDL

In this sub-section of results, we will show the results of the RISC-V interpreter and compare itwith the results in python. The IDE which is used for python is PyCharm. The instruction thatwill be tested for are the store word and store byte.



As shown in figure 2 the block that is in white is the result from the RISC-V interpreter and the black block is from the PyCharm IDE. The store instruction will store the value of 1500 into register 20. As seen in the white block which is for the RISC-V interpreter the value of 1500 is stored in the register 20. When we look to the generated results using python, we find that the value 1500 has been stored in the data memory. In this next figure we use another store operation but instead of storing a word a store byte wasexecuted.

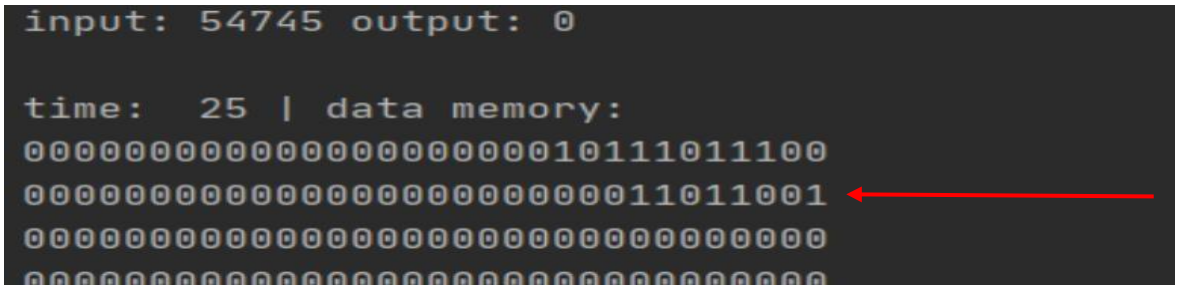


Figure 3 another comparison between RISC-V interpreter and python

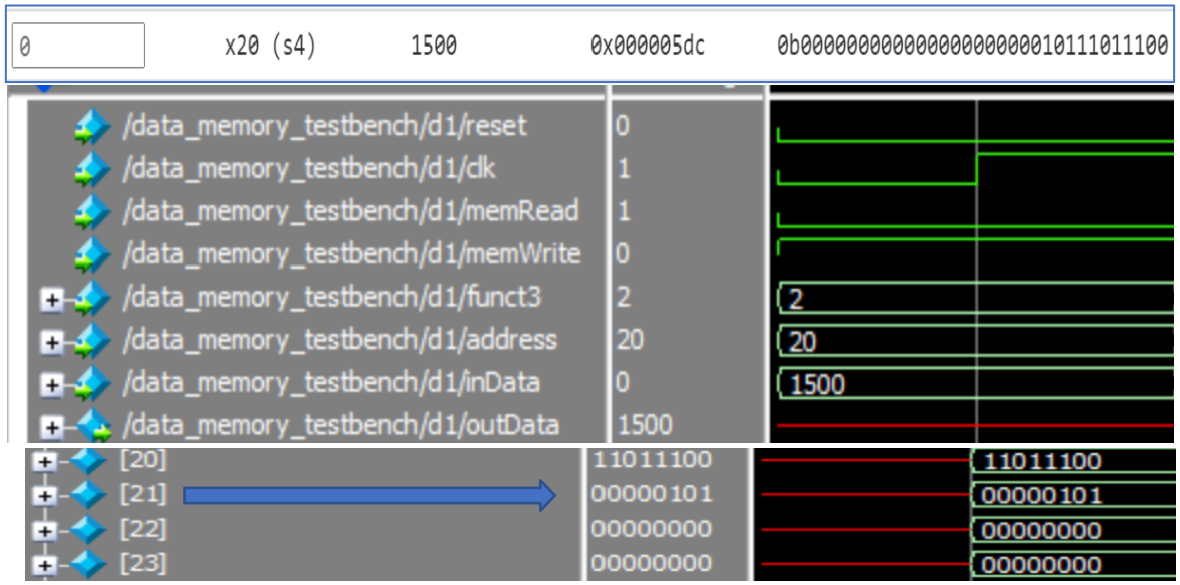
As shown in figure 3 the block that is in white is the result from the RISC-V interpreter and the black block is from the PyCharm IDE. The store instruction will store the value of 54745 into register 24. As seen in the white block which is for the RISC-V interpreter the value of 54745 is stored in the register 24. When we look to the generated results using python, we findthat the value 54745 has been stored in the data memory.

Discussion

As seen in this simulation in python it is storing these values that have been given and shownin binary, and these results are the same as with the RISC-V Interpreter.

RISC-V Interpreter vs Verilog HDL in Quartus

In this section we will discuss the same operations done previously with PyCharm. The simulation will be correct if the conversion from myHDL to Verilog HDL is completely correct. This section will show the simulation results from the Quartus IDE using ModelSim in Quartus to show the simulation results. The simulation will



show the store operations done and make sure they are stored in the correct registers.

Figure 4 Comparison between RISC-V interpreter and Quartus IDE using ModelSim

in figure 4 the white block represents the RISC-V interpreter and black block represent ModelSim showing the results in a waveform. When func3 is 010 that instruction will be to store word, and the address to store the value is 20 and the value that will be stored is 1500. Asseen in the blue arrow this is how the 1500 are stored.

In the below figure we will test the store byte instruction where we will compare the result with the RISC-V interpreter and the results in the ModelSim simulation.

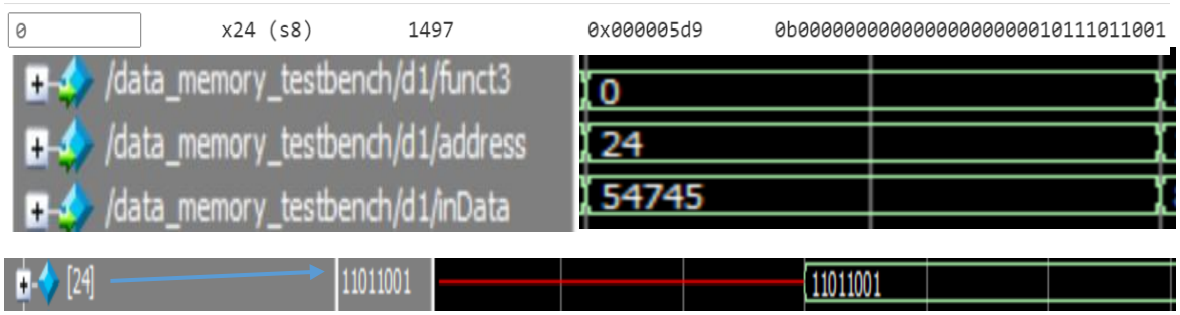


Figure 5 RISC-V interpreter comparison with ModelSim simulation

In figure 5 the white block represents the RISC-V interpreter and black block represent ModelSim showing the results in a waveform. When func3 is 000 that instruction will be to store byte, and the address to store the value is 24 and the value that will be stored is 1500. Asseen in the blue arrow this is how the value 54745 are stored.

Conclusion

In conclusion, the results that are shown the simulation that are mentioned in the report have been correct, this means that the modules that was created in PyCharm and then converted to Verilog HDL and synthesized in Quartus for a function simulation have been correct. That assumption made previously about whether the values will be stored correctly have been verified by validating the results. This is done through work plan that have been set to make through the experiment and compare with factual results in the RISC-V interpreter. This concludes the validation and experimentation of the Data Memory. The data we achieved fromthis experiment helped the simulation.

Engineering Standards

IEEE Std 1364™-2005 IEEE Standard for Verilog® Hardware Description Language [4]:

- The design of the data memory in Python follows the IEEE Standard for Verilog designstandards.

- The Simulation done follows the IEEE Standard Verilog validation and verificationstandards

References

[1]: myHDL, "myHDL," [Online]. Available:

http://docs.myhdl.org/en/stable/manual/preface.html. [2]:

https://www.cs.cornell.edu/courses/cs3410/2019sp/riscv/interpreter/#

[3] : https://www.intel.com/content/www/us/en/software/programmable/quartus-prime/modelsim.html

[4] : https://www.eg.bucknell.edu/~csci320/2016-fall/wp-content/uploads/2015/08/verilog-std1364-2005.pdf

EXPERIMENT #2

INTRODUCTION

In this report, we'll be displaying the simulations, testing and validations methods used on the CPU part in the project. These methods were used using different simulation tools to ensure the validity of the result across different environments. In this report we'll give a brief background information about the project as a whole and how the CPU contributes in the workings of the project, the objectives of the testing done on this part, the methods and simulation tools used, the workplan, display of the results from different procedures and levels of testing, and finally a discussion at the end going over everything done so far.

BACKGROUND

As we've displayed in previous reports, we are aiming to design a microcontroller (simply put). Microcontrollers has CPUs in them to control everything done inside the microcontroller. Our CPU contains seven main parts (as shown in figure 1), and some extra adders and multiplexers. The main parts are: program counter, instruction memory (ROM), decoder, control unit, file register and an arithmetic logical unit. The testing was done on all these parts as a whole using two

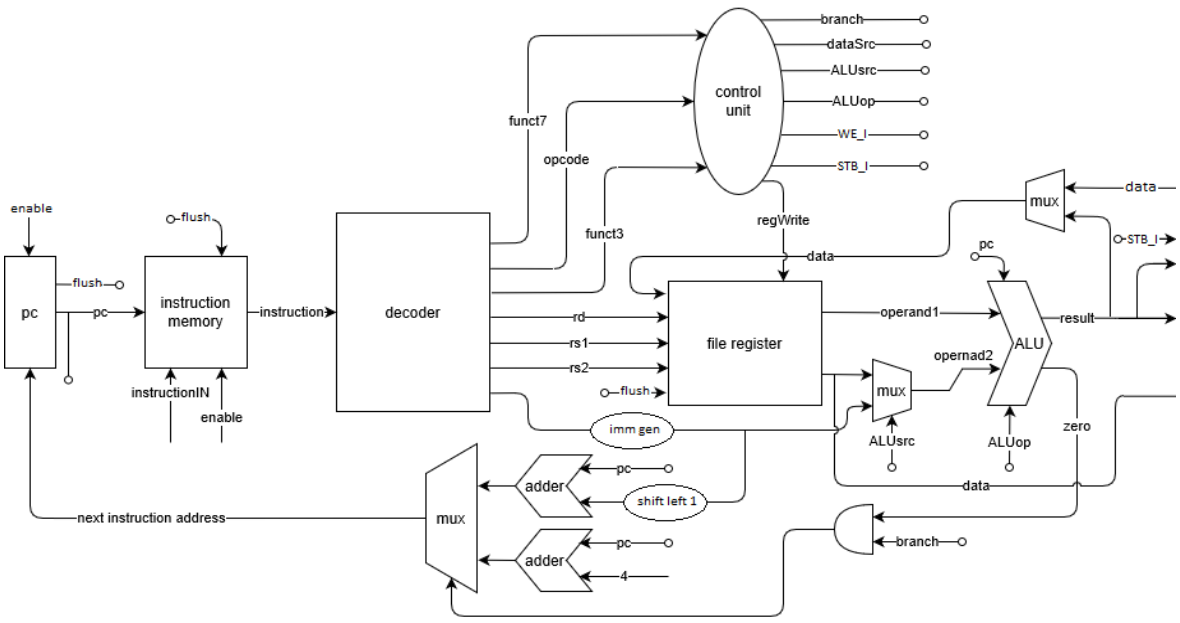


Figure 2 CPU internal components block diagram

different programming languages and simulation tools to validate the results of the CPU, and make sure the results are the same across different environments.

OBJECTIVE

- Make sure the Python to Verilog conversion for all the internal parts is accurate and the generated code works with no further intervention or modifications by the user.
- Use RISC-V interpreters to accurately see the contents of the register file and compare it with the result from the simulation.
- Run the same programs on the CPU in Python and Verilog and compare the results.

TOOLS

- Python
- PyCharm [1]
- MyHDL library [2]
- Verilog
- Quartus [3]
- ModelSim [4]
- RISC-V interpreter [5]

WORKPLAN

1. Connect different components made by all team members that defines the CPU (ROM, Decoder, Register File… etc.) and other extra needed components (Adders and Multiplexers) in one Python file.
2. Create a testbench section using the MyHDL library in Python that will run a complicated code that will use as many instructions as possible.
3. Print out the contents of the Register File on the run tool window in PyCharm
4. Use a RISC-V interpreter and run the same code previously used to test the functionality of the CPU.
5. Compare the results (data) stored in the Register File from the first simulation (in Python) with the second results (data) displayed in the Register File by the interpreter.

6. Validate the results from CPU created in Python with results from the RISC-V interpreter.
7. Write Python to Verilog conversion code after connecting, testing and validating the CPU.
8. Import the generated Verilog code from the generated text file by the MyHDL library to Quartus project.
9. Synthesize the generated Verilog CPU code using Quartus, check the similarity of the generated RTL block diagram with hand drawn block diagram of the CPU.
10. Run the same code used previously on the interpreter and the Python CPU on the generated CPU and check the contents of the Register File and internal signals using the ModelSim simulation tool.
11. Validate the result from the ModelSim simulator by comparing it with both the result from the interpreter and the Python CPU.

```
37    clock_driver = clock_generator(clock)
38
39    pc_driver = program_counter(clock, nextPC, enable, currentPC, reset)
40
41    pcAdder_driver = adder(4, currentPC, normPC)
42
43    BranchAdder_driver = braAdder(currentPC, imm, branchValue)
44
45    AndGate_Driver = And(zero, branch, branchTaken)
46
47    PCMUX = mux(branchTaken, normPC, branchValue, nextPC)
48
49    inst_memory_driver = inst_memory(reset, clock, enable, currentPC, instructionIN, instruction)
50
51    decoder_driver = decoder(instruction, opcode, rd, rs1_in, rs2_in, imm, funct3, funct7)
52
53    control_unit_driver = control(reset, clock, opcode, funct3, funct7, ALUop, memRead, memWrite, regWrite, branch,
54                                  ALUsrc,
55                                  dataSrc)
56
57    file_register_driver = file_register(reset, clock, regWrite, data, rs1_in, rs2_in, rd, rs1_out, rs2_out)
58
59    operand2MUX_driver = mux(ALUsrc, rs2_out, imm, operand2)
60
61    ALU_driver = alu(currentPC, rs1_out, operand2, ALUop, result, zero)
62
63    dataMUX_driver = mux(dataSrc, result, data_from_mem, data)
64
65    data_mem_driver = data_mem(reset, clock, memRead, memWrite, funct3, result, rs2_out, data_from_mem)
```

Figure 3 python implementation of a CPU internal modules connected together

RESULTS

After going through the previous eleven points (steps), we can see the result for all of them in this section. First things first, we connected all the components in one python file (figure 2). Each module is shown explicitly in a single line, with its input and output signals. For example, we can see that the decoder module has

eight input/output signals connected to it. All these modules together function together to create the CPU.

```
addi x7,x0,500
addi x11,x0,0
addi x12 ,x0,1
addi x6,x0,500
loop1:add x12,x12,x11
sw x12,4(x6)
addi x6,x6,4
addi x11,x11,1
blt x11,x7,loop1
addi x6,x0,500
sw x12,0(x6)
lw x10,0(x6)
```

Figure 5 RICS-V assembly program

```
register 0 : 0
register 1 : 0
register 2 : 0
register 3 : 0
register 4 : 0
register 5 : 0
register 6 : 500
register 7 : 500
register 8 : 0
register 9 : 0
register 10 : 124751
register 11 : 500
register 12 : 124751
```

Figure 4 data stored in the file register after the completion of the program

Now that we connected all the modules, we will run a piece of code on this CPU to test if it's working properly. To do so, we wrote a simple program (RISC-V assembly programming language) that will have different operations and loops in it to test it on a variety of instructions. In figure 4, we can see the program, and in the figure next to it (figure 3) we can see the result (data) stored in the file register after the completion of the program displayed on the PyCharm run tool window.

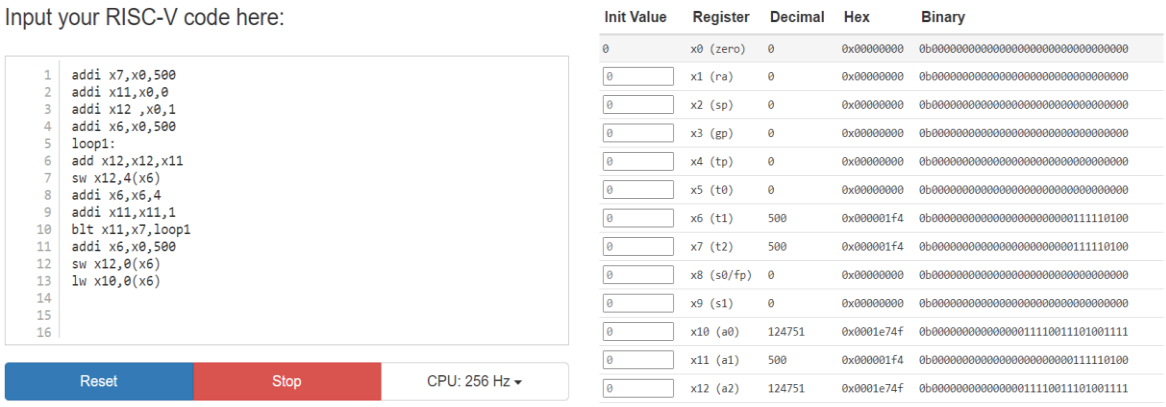


Figure 6 RISC-V interpreter result

After testing this program on our CPU, we have to validate the results. To do so, we used an online RICS-V interpreter [5]. In figure 5, we can see the input code to the interpreter and the data stored in the file register of the interpreted CPU.

From the previous figures (figure 3 and 5), we can compare the contents of both file registers and look at the data stored in each of them to see if they're similar to each other or not. And as a result, we can see that the contents of both file register are the exact same. 500 in register x6, x7 and x11, 124751 in both register x10 and x12.

Now things will get more complication, we've come to the Python to Verilog conversion part. It is not mention in the MyHDL library documentation but the conversion may cause some errors in the Verilog code. These errors may include: wrong memory element conversion, setting a variable as a register instead of a wire and vice versa. To avoid such errors, we had to do some simple adjustments to some components in Python to make sure the output of the conversion is what we expect. In figure 6, we can see a comparison between the file register in Python and the converted Verilog code. Blue bounding boxes in the left represent the

Figure 7 comparison between the original Python code and the generated Verilog code generated by the MyHDL library

input/output of the module. In right, we can see that the MyHDL library knows which signals are input and which are outputs automatically. The clock, reg_write, data, rs1in, rs2in and rdin are set as inputs to the file register, and rs1out and rs2out are set as output signals. And we can also notice that it also created a memory element named "register" which represents the registers inside the register file.

After we've confirmed that the conversion works as expected, we import the Verilog to Quartus program to use its features and functionalities to validate the converted code for the CPU, and to make sure it works as expected. In this step we'll copy the

generated code to a Quartus project, run the Analysis and Synthesis, Fitter, Assembler and the Timing Analysis tools to get different results used to validate the correct functionality of the generated code. In the next figure (figure 7) we can see the register-transfer level block diagram generated by the Analysis and Synthesis tool.

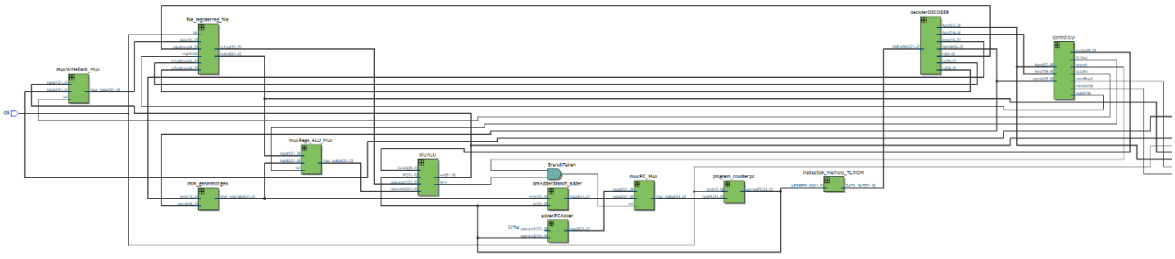


Figure 8 CPU block diagram using Quartus RTL viewer

By looking at the previous figure, we can see the similarities between the generated CPU block diagram and the hand drawn block diagram in figure 1. All the connections between the components are included in the generated block diagram and all modules are there as well. The only difference is the organization of the modules, which is not important.

For the next step, we'll be testing the functionality of the CPU using the ModelSim simulation tool [4]. We will run the same code (figure 4) previously used to test the Python CPU and used in the RISC-V interpreter. In the next figure (figure 8) we can see the result from the ModelSim simulation.

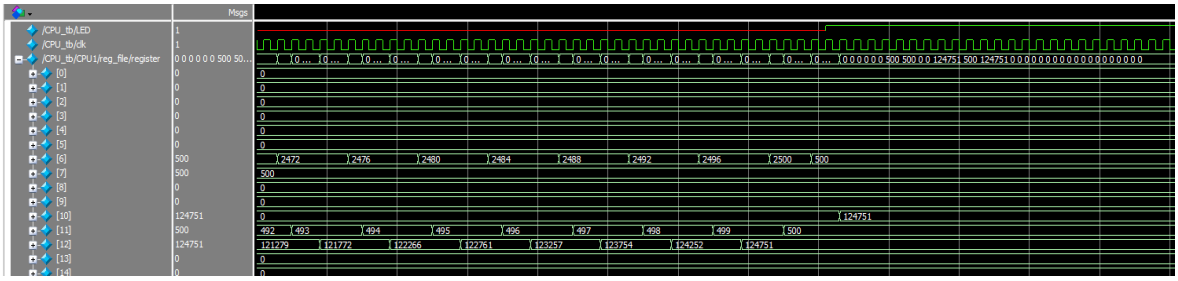


Figure 9 ModelSim simulation of the CPU running the same code in figure 4

The file register in the previous figure shows the contents of each register, and as we can see, the contents of register x6, x7 and x11 is 500, and the contents of register x10 and x12 is 124751. This means that the Verilog CPU code generated by the MyHDL [2] runs as expected, since the contents of the file register is the same as the file register in the Python CPU.

CONCLUSION

Taking everything into account, the result from all simulations and testing methods used to validate the functionality of the CPU across different environments were as expected. Functional simulations in Python, Verilog and the result from the RISC-V interpreter gave the same result, the data stored in the file register after the completion of the code were the same, and the RTL viewer tool generated the right block diagram for a CPU. Finally, it may be concluded, that the validation and experimentation procedures are considered a success, and we can move to next stage in the project.

ENGINEERING STANDARDS

IEEE Std 1364™-2005 IEEE Standard for Verilog® Hardware Description Language [6]:

- All design procedures done Python were made with the IEEE Standard for Verilog design standards in mind.
- All procedures used to validate the correctness of the functional, timing simulations and behavioural modeling follow the IEEE Standard for Verilog.

REFERENCES

[1] https://www.jetbrains.com/pycharm/

[2] http://www.myhdl.org/

[3] https://www.intel.com/content/www/us/en/software/programmable/quartus-prime/overview.html

[4] https://www.intel.com/content/www/us/en/software/programmable/quartus-prime/model-sim.html

[5] https://www.cs.cornell.edu/courses/cs3410/2019sp/riscv/interpreter/

[6] https://www.eg.bucknell.edu/~csci320/2016-fall/wp-content/uploads/2015/08/verilog-std-1364-2005.pdf

EXPERIMENT #3

INTRODUCTION

Testing and validation are one of the most important steps in engineering design. Thus, in this report we are going to test and validate different essential component of our project. In order to ensure the result and the correctness of the outputs, we are going to test the selected component in each stage/level of the project. starting by applying functional simulation on the Python level by using MyHDL, then importing the generated code and testing it on Quartus Prime (Synthesizing, RTL schematic, etc.). In this report, we aim to test the control unit which is considered of the internal modules of the CPU and it is one of curtail parts of our project.

BACKGROUND

our project simply aims to develop a RISC-V Microcontroller for FPGA. Microcontrollers contain a CPU, a set of peripherals and a bus system to communicate between the CPU and other peripherals. The CPU is considered the most important part of the system since it is going to perform logical and arithmetic operations, control other components, coordinate and drive the bus system. Thus, in case of the controlling and driving the bus system will be the responsibility of the control unit since we have single master (CPU).

OBJECTIVE

- Ensuring a successful conversion of the design code written in python using MyHDL to synthesizable Verilog code while ensuring that the IEEE standers for Verilog are met. [1]
- Simulate the Control Unit using the Python library MyHDL and ensuring that the output of the simulation and the expected output are identical.
- Simulate the Verilog code of the Control Unit using Quartus Prime and ensure that the RTL Schematic diagram and the waveform simulation outputs are as expected.

TOOLS

- Python
- PyCharm
- MyHDL [2]
- Quartus Prime [3]
- Verilog
- ModelSim [4]
- RARS [5]

WORK PLAN

1- Connect the internal components of the CPU along with the Control Unit in Python using the library MyHDL.

2- Write a RISC-V assembly code to simulate the whole system and convert it into binary and include it in a test bench that simulate the system in Python.

3- If the system function as expected i.e. the Register File contain the expected values at the end of the simulation, then all internal of the CPU is functioning as expected.

4- After the success of the previous step, we will convert the Python code to Verilog code using MyHDL and make sure that the code is synthesizing and it shows the right RTL Schematic diagram.

5- Write a test bench in converted Verilog and simulate the design using ModelSim, by checking the waveform of the simulation we will make sure if the Control Unit is outputting the correct signals to the rest of the Microcontroller components.

RESULTS

After connecting the internal components of the CPU with the Control Unit, we ran the test bench on the Python level to test if the control Unit is generating the right control signals for the other components of the CPU.

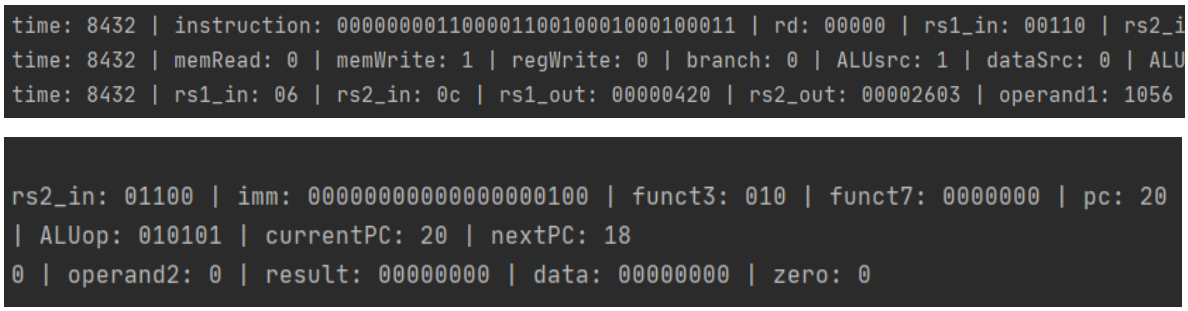


Figure 10: Control Signals for S type instruction

In the above figure (figure 1), shows the generated signals for the instruction “sw x12,4(x6)”. The control unit is responsible for generating the memRead, memWrite, regWrite, Branch, ALUsrc, dataSrc and ALUop signals for the rest of the components. The expected values of the memWrite, ALUsrc and ALUop should be 1, 1, and 010101 respectfully and simulation show that the Control Unit generated them properly.

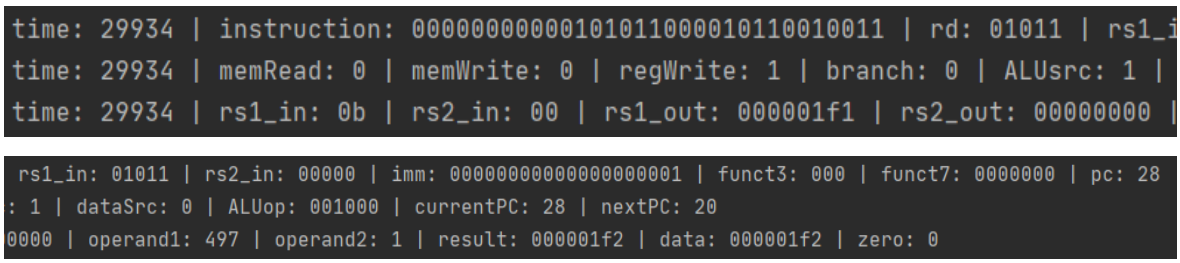


Figure 11: ADDI instruction

The above figure (figure 2), shows the functional simulation result of an I-type instruction “addi x11,x11,1”. The expected values of the regWrite, ALUsrc, dataSrc and ALUop should be 1, 1, 0, 001000 respectfully and Control Unit generated them properly.

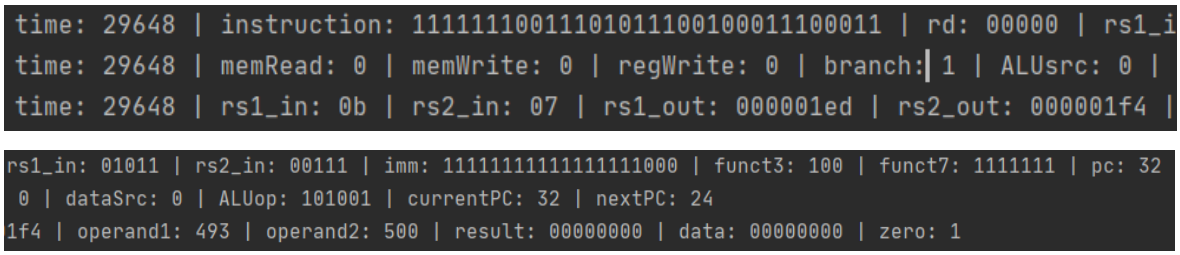


Figure 12: BLT instruction

The above figure (figure 3) shows the results of the functional simulation of the Control Unit of an intrsuction of type B “blt x11,x7,loop1”. The expected values of the branch, regWrite, ALUsrc, dataSrc and ALUop should be 1, 0, 0, 0, 101001 respectfully and the simulation of the control unit result match the expected values which conclude that the Control Unit is working as expected.

After testing the correctness of the of the Control Unit design, the next step is to test the correctness of the code conversion process. As you can see in figure 4, by comparing the netlist of each code (Python on the left and Verilog on the right) are the same which indicate the code is generated properly to a certain extent. In the upcoming sections of the report, we are going to apply a test bench to ensure that the logic in the Verilog code is compatible with the Python code.

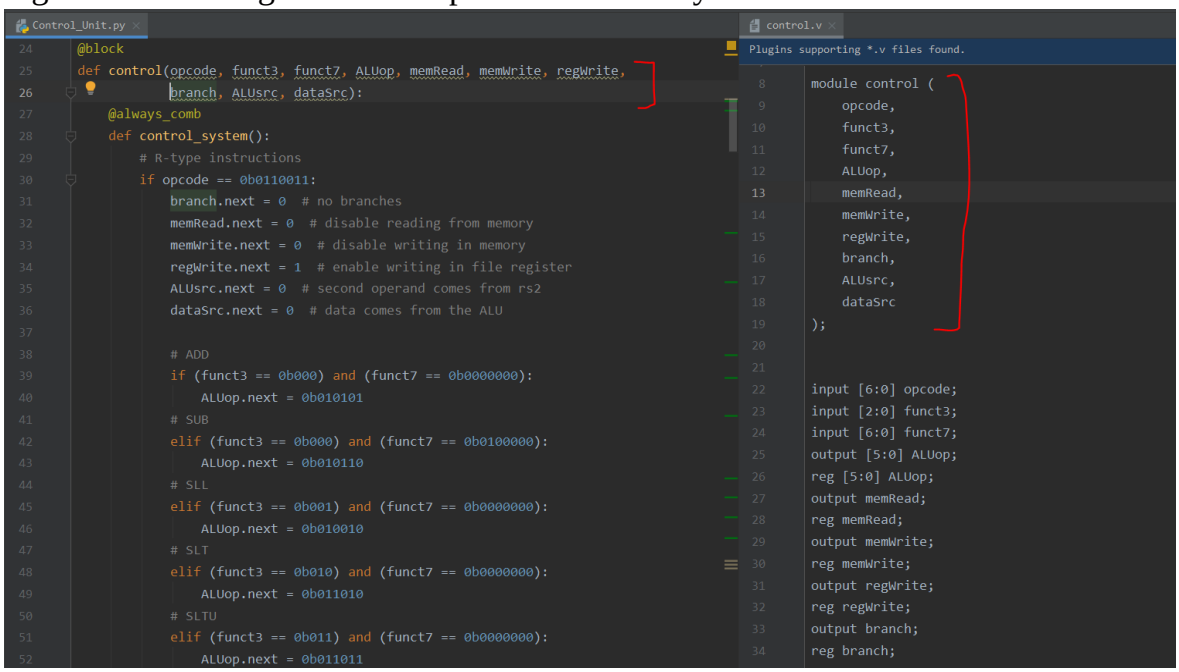


Figure 15: netlist comparison

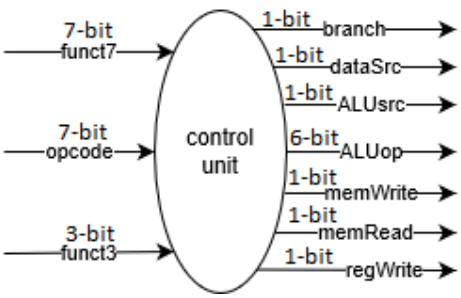


Figure 13 : expected diagram

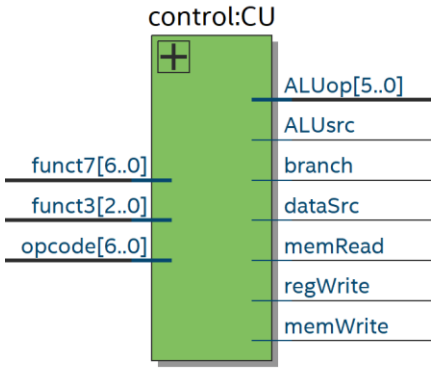


Figure 14 : RTL diagram

To ensure that the converted code is working as expected, we will synthesize the generated code and compare the expected diagram of the Control Unit and the RTL Schematic diagram. As you can see in figure 5 and 6 below, the input signals in the expected and the generated Verilog code is the same.

Finally, after converting the MyHDL code, synthesizing it and inspecting the RTL diagram, we need to ensure the logic of the Verilog code and that it follows the standers of IEEE for Verilog .in order to test the logic of the converted code and making sure it functions as expected, we wrote a test bench that will test the generated signals for R-type instructions, I-type instructions and B-type instructions which covers the base integer instructions.

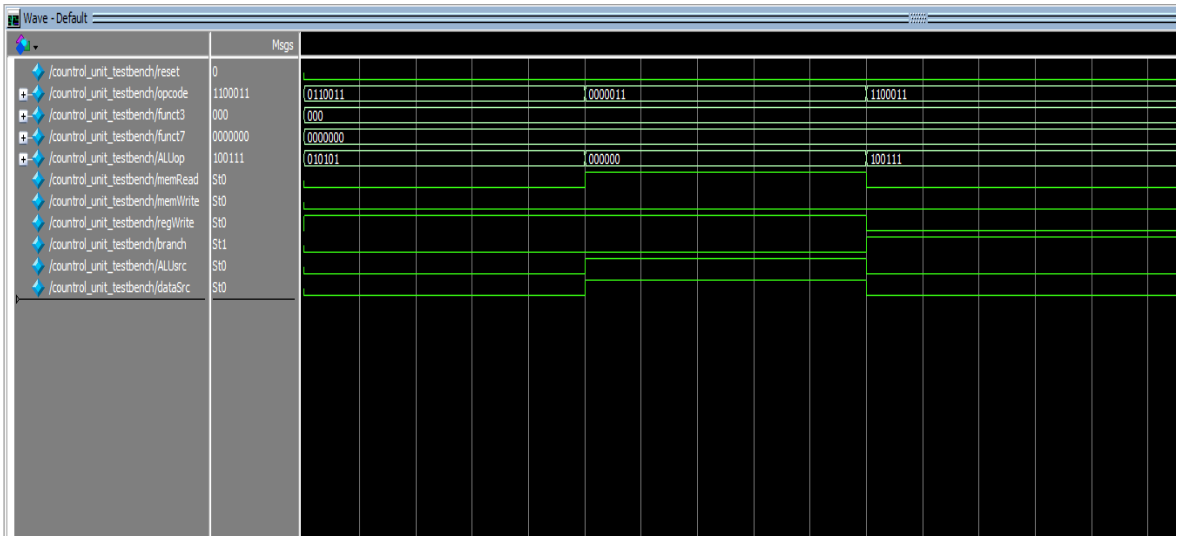


Figure 16: waveform from ModelSim

As you can see in the above waveform simulation, the first opcode references the R-type instructions and corresponds to the ADD instruction is specific. The generated signals ALUop, memRead, memWrite, regWrite, branch, ALUsrc and dataSrc values are 010101, 0, 0, 1, 0, 0 and 0 respectfully and they indicate that the signals for the R-type instructions are generated properly. Additionally, the second opcode references the I-type instruction specifically the LW instruction. The generated signals ALUop, memRead, memWrite, regWrite, branch, ALUsrc and dataSrc values are 000000, 1, 0, 1, 0, 1 and 1 respectfully which indicate that the Control Unit is generating the correct signals for the I-type instruction. Finally, the las opcode references the B-type instruction specifically beq instruction. The generated signals ALUop, memRead, memWrite, regWrite, branch, ALUsrc and dataSrc values are

100111, 0, 0, 0, 1, 0 and 0 respectfully which indicate that the Control Unit is generating the correct signals for the B-type instruction

ENGINEERING STANDARDS

- All the written code and design methods in the python level were conducted following the standers set by the organization that developed MyHDL.
- The converted Verilog code was test and inspected to make sure it follow the IEEE Std 1364™-2005 IEEE Standard for Verilog® Hardware Description Language

CONCLUSION

In conclusion, the result of all the simulations and the test that we ran on the control unit showed that it functions as expected and it result and produce the correct number of signals with the right values. Additionally, the Control Unit plays a huge rule in synchronizing the rest of the components inside the CPU and in the test proven that the design of our Control Unit performed this task correctly. Finally, we can conclude from this report that the Control Unit is functioning without no logical errors and without no synchronization problems.

REFERENCES

[1]https://www.eg.bucknell.edu/~csci320/2016-fall/wp-content/uploads/2015/08/verilog-std1364-2005.pdf

[2] http://www.myhdl.org/

[3] https://www.intel.com/content/www/us/en/software/programmable/quartus

[4]https://www.intel.com/content/www/us/en/software/programmable/quartus-prime/modelsim.html

[5] https://github.com/TheThirdOne/rars

EXPERIMENT FINAL PRODUCT

Introduction

Since our project is big and has a lot of parts and different components, we needed to test and validate each component individually before integrating them together to for our final product. In the case of huge projects such as ours, components may work individually as expected but when they are integrated together to form the final product some issues and errors may arise. In this final validation experiment, we will test the final product (Microcontroller) which is form of the components validated in the previous experiments in addition to other components (peripherals) such as the PWM, UART and the Timer.

Objectives

1- Ensure the connected peripherals are receiving and sending the data from the master (CPU) via the bus system without any errors.

2- Ensure that the peripheral is working as expected, outputting the expected values at the right time without any delays that can cause problems

Tools

- Python
- PyCharm
- MyHDL
- Quartus Prime
- Verilog
- DE10-lite FPGA board
- RARS

Work Plan

1- Converting the top-level python code to Verilog, the top-level must contain the peripherals that need to be tested
2- Write an assembly code to be loaded in the CPUs' ROM in order to simulate the microcontroller.
3- Convert the assembly code to binary text and load it in the ROM
4- Run the whole design (Quartus Prime)
5- Use Quartus Primes' programmer to program the FPGA with the design

Results

after implementing the first mentioned step in the work plan, we used the following code to test the PWM.

```
addi x10, x0, 20
slli x10, x10, 16
addi x10, x10, 10
sw x10, 1029(x0)
```

This code will load the PWM with the value of "1400A" which will make the "period = 20" and the "active = 10" (in HEX), making the duty cycle equal to 50%.

After loading the CPUs' ROM with the previous code and programing the FPGA with the design, we recorded a video, and we are going to discuss them step by step to make sure that the PWM is working as expected. Before we start with the pictures, keep in mind the following format and LED assignments:

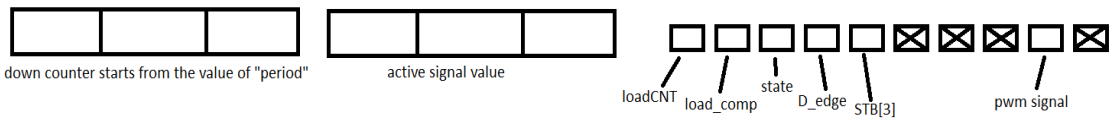


Figure 17: LEDs and 7-Segment assignment

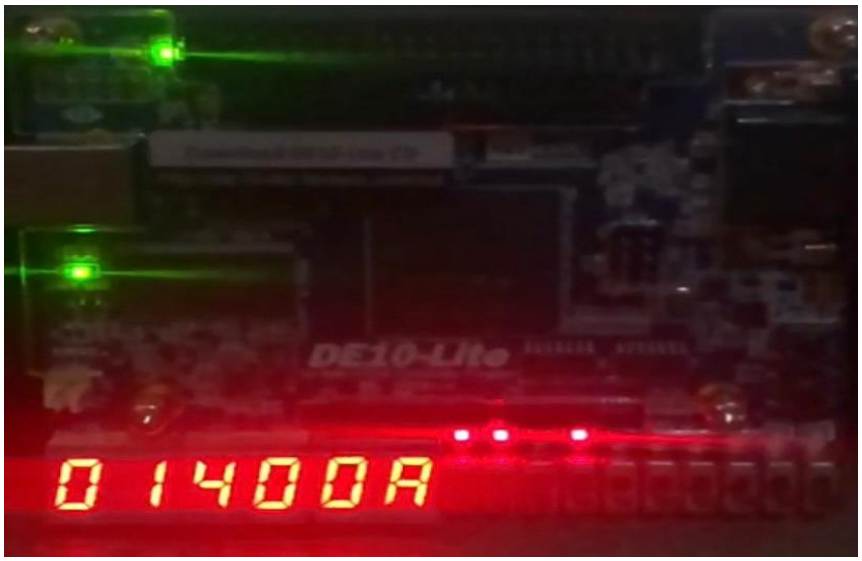


Figure 18: PWM after receiving the data

As you can see in figure 3, At the beginning of the program, the PWM received the data correctly which are 14 (HEX) for the period and A for the active (HEX).

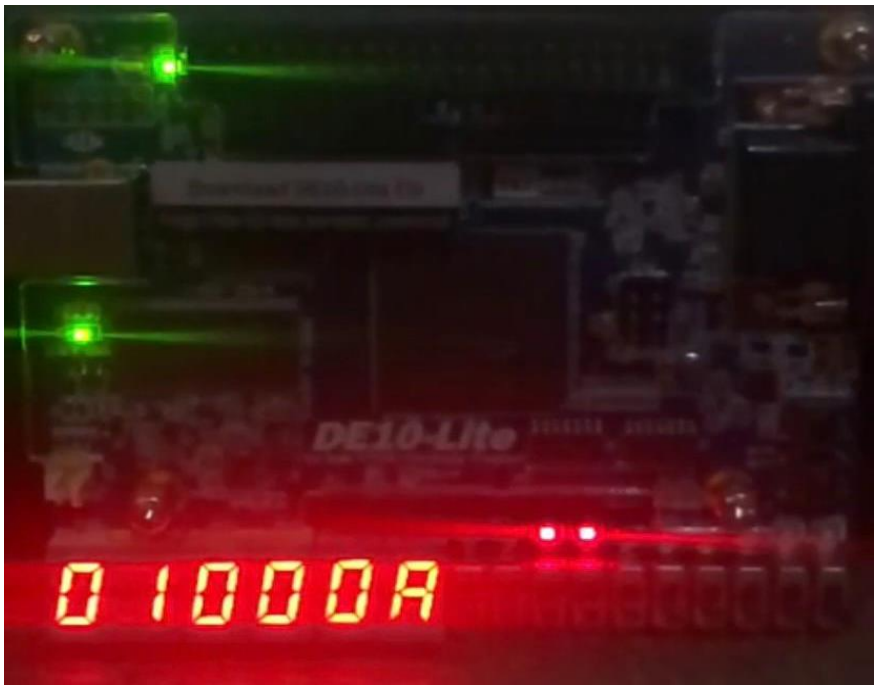


Figure 19: the counter (10) is decresing

As you can see in figure 4, counter will start to count down (it was 14 at the start) and it will keep in counting down until it is less than the value in the “active” register.

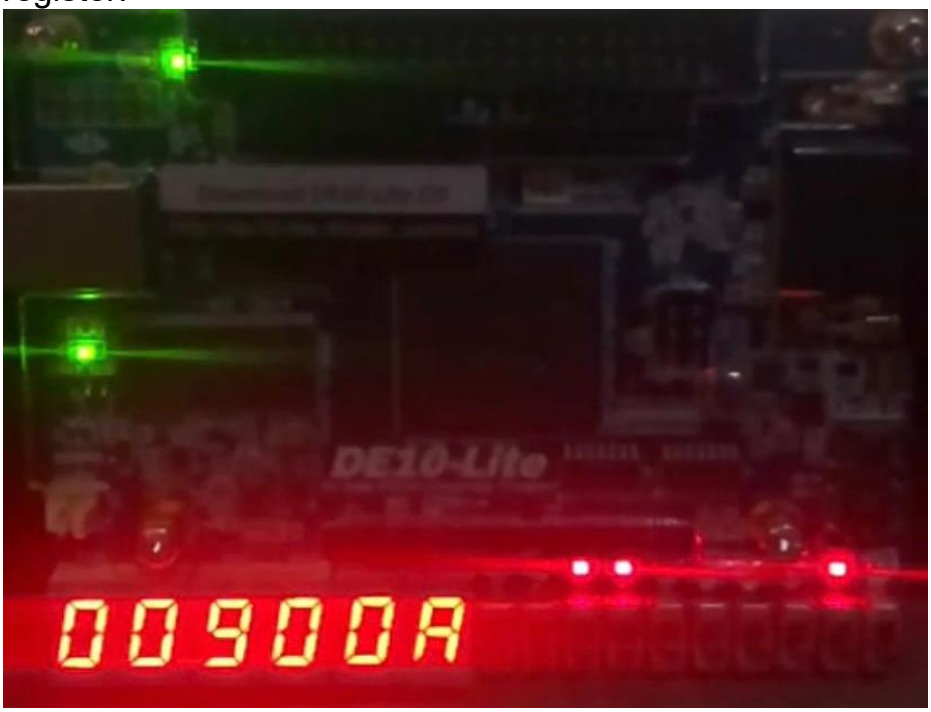


Figure 20: the PWM signal is set to high

As you can see in figure 5, once the value in the counter became less that the value in the “active” register, the PWM signal is raised to high (the second LED from the right).

Figure 21: the value of the counter has been reseted

After that the value will keep on decreasing until it resets to start over again (see figure 6)

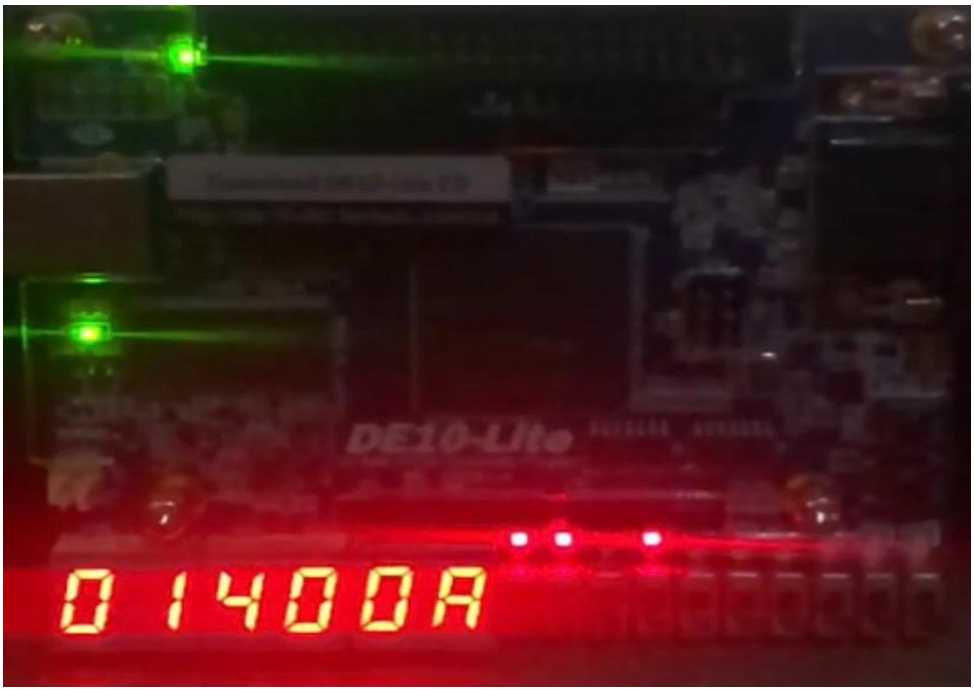


Figure 22: the counter is loaded again

As you can see in figure 7, After the value of the period has been reseted, the counter will load the value of the period again, you can verifiy that by the left most LED which refers to the signal “loadCNT” which it will enable the counter to load the value of the period again afte its completion.

Conclusion

After completing this experimet, we succufully acomplised each of the objectives that we aimed for. The communication between the CPU and the PWM via the bus system is proved to be working correctly and the PWM is performing its job as excpected.

APPENDIX – B: SELF ASSESSMENT CHECKLIST

Use student outcomes (SOs 1 - 7) rubrics to fill the following table. Each member needs to fill in this table; it is important to enable the department to know to what degree the EE programs have been able to achieve the required KPIs of each SO. Please use the following grading letters:

E: Exemplary, **S:** Satisfactory, **D:** Developing, and **U:** Unsatisfactory.

Table 22 Self-Assessment Table

Student Outcome (SO)	**Key Performance Index (KPI)**	**Self-assessment (E, S, D, or U)**		
		M1	**M2**	**M3**
1. an ability to identify, formulate, and solve complex engineering problems by applying principles of engineering, science, and mathematics	1.1. Problem Identification	E	E	E
	1.2. Problem formulation	E	S	E
	1.3. Problem solving	S	E	E
2. an ability to apply engineering design to produce solutions that meet specified needs with consideration of public health, safety, and welfare, as well as global, cultural, social, environmental, and economic factors	2.1. Design Problem Definition	E	E	E
	2.2. Design Strategy	S	S	E
	2.3. Conceptual Design	E	E	S
3. an ability to communicate effectively with a range of audiences	3.1. Effective Written Communication	E	E	E
	3.2. Effective Oral Communication	E	E	E
4. an ability to recognize ethical and professional responsibilities in engineering situations and make informed judgments, which must consider the impact of engineering solutions in global, economic, environmental, and societal contexts	4.1. Recognition of Ethical and Professional Responsibility	E	E	E
	4.2. Consideration of Impact of Engineering Solutions	E	S	S
5. an ability to function effectively on a team whose members together provide leadership, create a collaborative and inclusive environment, establish goals, plan tasks, and meet objectives	5.1. Effective Team Interactions	E	E	E
	5.2. Use of Project Management Techniques	S	E	S
6. an ability to develop and conduct appropriate experimentation, analyze and interpret data, and use engineering judgment to draw conclusions	6.1. Developing Appropriate Experiment	E	S	E
	6.2. Conducting Appropriate Experiment	S	E	S
	6.3. Analysis and interpretation of Experiment Data and Drawing Conclusions	S	E	E
7. an ability to acquire and apply new knowledge as needed, using appropriate learning strategies	7.1. Effective Access of information	E	E	E
	7.2. Ability to learn and apply new knowledge independently	E	E	E