

ISTANBUL TECHNICAL UNIVERSITY
ELECTRICAL-ELECTRONICS FACULTY

**INSTRUCTION EXTENSION OF RISC-V PROCESSOR FOR DRIVER
FATIGUE DETECTION SYSTEM AND IMPLEMENTATION**

SENIOR DESIGN PROJECT

Elif DİNÇ
Gülce BAYSAL
Merve KILIÇ

**ELECTRONICS AND COMMUNICATION ENGINEERING
DEPARTMENT**

JUNE 2022

ISTANBUL TECHNICAL UNIVERSITY
ELECTRICAL-ELECTRONICS FACULTY

**INSTRUCTION EXTENSION OF RISC-V PROCESSOR FOR DRIVER
FATIGUE DETECTION SYSTEM AND IMPLEMENTATION**

SENIOR DESIGN PROJECT

Elif DİNÇ
040170040

Gülce BAYSAL
040170051

Merve KILIÇ
040170078

**ELECTRONICS AND COMMUNICATION ENGINEERING
DEPARTMENT**

Project Advisor: Prof. Dr. Sıddıka Berna Örs Yalçın

JUNE 2022

İSTANBUL TEKNİK ÜNİVERSİTESİ
ELEKTRİK-ELEKTRONİK FAKÜLTESİ

**SÜRÜCÜ YORGUNLUK TESPİT SİSTEMİ VE UYGULAMASI İÇİN RISC-V
İŞLEMCİSİ KOMUT SETİNİN GENİŞLETİLMESİ**

LİSANS BİTİRME TASARIM PROJESİ

Elif DİNÇ
040170040

Gülce BAYSAL
040170051

Merve KILIÇ
040170078

Proje Danışmanı: Prof. Dr. Sıddıka Berna Örs Yalçın

ELEKTRONİK VE HABERLEŞME MÜHENDİSLİĞİ BÖLÜMÜ

HAZİRAN, 2022

We are submitting the Senior Design Project Report entitled as “INSTRUCTION EXTENSION OF RISC-V PROCESSOR FOR DRIVER FATIGUE DETECTION SYSTEM AND IMPLEMENTATION”. The Senior Design Project Report has been prepared as to fulfill the relevant regulations of the Electronics and Communication Engineering Department of Istanbul Technical University. We hereby confirm that we have realized all stages of the Senior Design Project work by ourselves and we have abided by the ethical rules with respect to academic and professional integrity.

Elif DİNÇ
040170040

.....

Gülce BAYSAL
040170051

.....

Merve KILIÇ
040170078

.....

FOREWORD

Firstly; we would like to thank our project advisor Assoc. Prof. Dr. Sıddıka Berna Örs Yalçın, who did not spare us her interest and always stood by us with her advice and assistance whenever we had difficulties. Also, we would like to thank our friends Okan Yağız and Yavuz Sultan Çakırca who worked with us on the TÜBİTAK project for their support. Lastly, we are grateful to our families who have been with us not only during this project but throughout our entire education life.

June 2022

Elif DİNÇ
Gülce BAYSAL
Merve KILIÇ

TABLE OF CONTENTS

	<u>Page</u>
FOREWORD	iv
TABLE OF CONTENTS	vii
ABBREVIATIONS	ix
LIST OF FIGURES	xi
SUMMARY	xiii
ÖZET	xv
1. INTRODUCTION	17
1.1 Motivation	17
1.2 Objectives	17
1.3 Thesis Contribution	18
1.4 Organization of This Study	18
2. LITERATURE REVIEW	19
3. RISC-V	20
3.1 Instruction Set Architecture:	20
3.2 RISC-V History	21
3.3 Advantages of RISC-V	21
3.4 RISC-V Assembly	22
3.5 Custom Instruction Set Extensions of RISC-V Processor	23
3.5.1 Multiplier and accumulator operator	23
3.5.2 Convolution and multiply-accumulate	24
3.5.3 Customization of RISC-V	25
4. IBEX CORE AND WISHBONE SYSTEM-ON-CHIP INTERCONNECTION ARCHITECTURE	27
4.1 Benefits of Ibex	27
4.2 Wishbone Protocol Explanation	28
4.3 Wishbone Signals	29
4.3.1 Single READ / WRITE cycles	31
4.3.2 Block READ / WRITE cycles	31
4.3.3 RMW cycle	31
5. IMAGE PROCESSING	33
5.1 Image Analysis	33
5.1.1 Colors	35
5.2 Image Filtering	36
5.2.1 Linear filters	37
5.2.1.1 Laplacian filter	37
5.3 Image Filtering and Custom IP Design	38
6. IMPLEMENTATION OF IBEX CORE AND TESTING HARDWARE	41
6.1 Installing Vivado	41
6.2 Setting Up RISC-V GCC Toolchain	42
6.3 Installing, Synthesizing and Implementing Ibex Core	43
6.4 Connecting GPIO	47
6.5 Connecting UART	48

7. APPLICATION OF IMAGE PROCESSING ALGORITHMS	50
7.1 Laplacian Filter and Design of its Custom IP	50
7.2 Custom Instruction Added Module	52
7.3 Connecting Camera and VGA.....	55
8. ACCESSING ONBOARD DDR RAM.....	60
8.1 What is DDR RAM?	60
8.2 Bootloader	60
8.3 Accessing DDR RAM with MicroBlaze	63
8.3.1 MicroBlaze	63
8.3.2 Steps to access DDR with MicroBlaze	62
8.4 Accessing DDR RAM via Wishbone Interface.....	69
8.4.1 Vector extension approach.....	69
8.4.2 Wishbone to AXI bridge	72
9. CONCLUSION.....	80
9.1 Results.....	80
9.2 Progress of the Project.....	80
9.3 Cost Analysis.....	81
9.4 Future Work and Recommendations.....	82
REFERENCES	83
APPENDICES	89
APPENDIX A	90
APPENDIX B.....	92
CURRICULUM VITAE.....	94
CURRICULUM VITAE.....	95
CURRICULUM VITAE.....	96

ABBREVIATIONS

ABI	: Application Binary Interface
ALU	: Arithmetic Logic Unit
AXI	: Advanced Extensible Interface
BSP	: Board Support Package
CMYK	: Cyan, Magenta, Yellow & Key
CPU	: Central Processing Unit
DFD	: Driver Fatigue Detection
DDR	: Double Data Rate
DMA	: Direct Memory Access
DSP	: Digital Signal Processor
EDK	: Embedded Development Kit
FIFO	: First In First Out
FPGA	: Field-Programmable Gate Array
FPN	: Fixed Pattern Noise
GCC	: GNU Compiler Collection
GPIO	: General Purpose Input/Output
HDL	: Hardware Description Language
HSV	: Hue, Saturation & Value
IP	: Intellectual Property
ISA	: Instruction Set Architecture
LED	: Light Emitting Diode
MAC	: Multiplier and Accumulator
MIG	: Memory Interface Generator
PMP	: Physical Memory Protection
RAM	: Random Access Memory
RGB	: Red-Green-Blue
RISC	: Reduced Instruction Set Computer
RMW	: Read-Modify-Write

RTL : Register Transfer Level
SCCB : Serial Camera Manage Bus
SDK : Software Development Kit
SDRAM : Synchronous Dynamic Random-Access Memory
SoC : System on Chip
UART : Universal Asynchronous Receiver-Transmitter
USB : Universal Serial Bus
VGA : Video Graphics Array
YCbCr : Luminance, Chroma Blue & Chroma Red
YUV : Luminance, Chrominance1 & Chrominance2

LIST OF FIGURES

	<u>Page</u>
Figure 3.1 : Level of abstraction in ISA.....	20
Figure 3.2 : RISC-V membership growth.....	20
Figure 3.3 : Register names in RISC-V.	23
Figure 3.4 : Serial structure of MAC operator.	24
Figure 3.5 : Parallel structure of MAC operator.	25
Figure 4.1 : The block diagram of the small parametrization with a 2-stage pipeline	27
Figure 4.2 : Master and slave Wishbone's interfaces.	28
Figure 4.3 : Tag types	93
Figure 4.4 : Wishbone shared bus	93
Figure 4.5 : Wishbone direction of data flow	93
Figure 4.6 : Block read cycle	32
Figure 4.7 : Block write cycle	32
Figure 5.1 : Representation of pixels	34
Figure 5.2 : RGB color model	35
Figure 5.3 : The CbCr plane at constant luma $Y'=0.5$	36
Figure 5.4 : Convolution of an 7x7 Image and a 3x3 Kernel	38
Figure 5.5 : Convolution of an 4x4 Image and a 3x3 Kernel	39
Figure 5.6 : Edge detection example	39
Figure 6.1 : Xilinx website.....	41
Figure 6.2 : Vivado interface.	42
Figure 6.3 : Github code for RISC-V toolchain.....	43
Figure 6.4 : Github code of Ibex core.	44
Figure 6.5 : Design sources of the Ibex core.	44
Figure 6.6 : The C code named led.c.	45
Figure 6.7 : Continuation of the C code named led.c.	45
Figure 6.8 : Simulation screenshot with led.mem file.	46
Figure 6.9 : FPGA results.	47
Figure 6.10 : UART connections.	48
Figure 6.11 : UART definitions on the Ibex_soc module.....	49
Figure 6.12 : The resulting RTL schematic.	49
Figure 7.1 : Laplacian filter.....	52
Figure 7.2 : Operands defined in ALU.	53
Figure 7.3 : Arranges in instruction decoder.....	53
Figure 7.4 : Custom 0 module.....	54
Figure 7.5 : Custom 1 module.....	55
Figure 7.6 : The OV7670 camera module.	56
Figure 7.7 : Key specifications of OV7670 camera module.....	56
Figure 7.8 : VGA and camera connections in the hierarchy.....	57
Figure 7.9 : Filter code.....	58
Figure 7.10 : The assigned address for controlling the IPs.....	59
Figure 7.11 : Test result.	59
Figure 8.1 : Comparison of DDR SDRAM and SDRAM	61
Figure 8.2 : DDR controller block diagram.....	62

Figure 8.3: MicroBlaze IP configuration.	64
Figure 8.4: Clocking wizard configuration.	65
Figure 8.5: Completed block design.	66
Figure 8.6: Address editor screen.....	67
Figure 8.7: Linker script screen.	68
Figure 8.8: New linker script screen.	68
Figure 8.9: Vector coprocessor block diagram.	70
Figure 8.10: The UART code.....	71
Figure 8.11: The test code.	71
Figure 8.12: FPGA implementation output.....	72
Figure 8.13: Planned block diagram.	73
Figure 8.14: Package IP of Ibex with Wishbone project.	74
Figure 8.15: Package IP of Wishbone-AXI bridge.	74
Figure 8.16: Adding repositories to the IP catalog.	74
Figure 8.17: New LED output signals.....	75
Figure 8.18: Final block design.....	76
Figure 8.19: The C code for the LEDs.....	77
Figure 8.20: Block design with MIG	79

INSTRUCTION EXTENSION OF RISC-V PROCESSOR FOR DRIVER FATIGUE DETECTION SYSTEM AND IMPLEMENTATION

SUMMARY

Today, image processing applications are used in practice, playing a role in the solution of many problems, constantly being developed and appearing in many areas of life. These applications, which basically consist of algorithms that allow us to extract meaningful expressions from the image we have, are reliable and have a structure that allows to reach the goal quickly. Security and face recognition can be given as examples of areas where image processing applications are used the most today. While talking about safety, besides being protected from environmental factors in human life, situations arising from human negligence and needing a warning system should also be mentioned. At this point, driver drowsiness detection applications come to the fore. Today, many fatal or damaging accidents are caused by drivers driving when they are not able to adequately meet their sleep needs. Tiredness and fatigue are many indicators of drivers who are exposed to constantly changing environmental conditions during long journeys and suffer from insomnia; Many unfortunate accidents can be prevented with detection and warning systems that can be designed taking these indicators into account. The most current application for the design of this type of detection mechanism is embedded systems due to its reliability and inherent speed. Embedded systems are targeted personal designs that perform tasks directly assigned to them. They were developed by combining a special software and hardware parts for the application to be developed. Besides the hardware, the greatest importance of the software component is that this system can work in accordance with other embedded systems, and that it provides support for adapting and updating according to changing conditions. An embedded system hardware can serve different purposes with different algorithms by developing and modifying the accompanying software component. In the implementation of the driver drowsiness detection system, which is the general purpose of our project, it is aimed to transform an embedded system into a system for the desired purpose with image processing algorithms. For this reason, an open-source processor has been used in order to develop the desired programs and design Custom Intellectual Property (IP)'s. In addition, it is aimed to design a system at the optimum level in terms of cost, speed and usability. The most important element needed for usability is memory and storage space. A certain amount of data can be kept in the memory (RAM) that a processor can access while performing its functions. The data on which we will use image processing algorithms in our project consists of pictures of sleepy-drowsy drivers. Nexys 4 DDR FPGA, the card we use in our project, does not have the memory necessary to store the data set we have and the program we will develop. It is aimed to obtain this extra memory we need with the external memory Double Data Rate (DDR) on the card we use. While solving the memory problem, which is the main purpose of the project, several methods have been found to enable the communication protocol and DDR to communicate seamlessly. Wishbone communication protocol was used throughout the project with Ibex, the open-source processor used, but it turned out that the Memory Interface Generator (MIG) needed to access the mentioned extra memory, DDR, can only be connected to the processor via Advanced Extensible Interface (AXI) connection. Accordingly, the method found is a structure called "bridge", which provides access to DDR by providing AXI-

Wishbone connection. In order to understand the communication protocol thoroughly, various simulations and implementations were made by providing General Purpose Input Output (GPIO) and Universal Synchronous Asynchronous Receiver Transmitter (UART) connections, and finally, image processing algorithms were implemented. With the communication protocol used, camera connection and filter applications were made with the captured image. Laplacian filter has been implemented with a processor with an extended instruction set, and in case the required memory is accessed, a system that works very fast and efficiently in driver drowsiness detection will be obtained.

The path followed in the design of this system, which detects driver fatigue-drowsiness, is given in order:

1. Implementation of the open-source Ibex processor
2. Making the various connections by making the Wishbone protocol compatible with the processor
3. Testing GPIO, UART and camera connections
4. Providing Wishbone-AXI-MIG-DDR connection with bridge method in order to access DDR
5. Making image processing applications with the developed Laplacian filter

At the end of the project, Ibex was successfully implemented and competence was gained in providing connections with the Wishbone protocol. At first, the Light Emitting Diodes (LED) were operated under the control of the Wishbone protocol with a simple LED activating code, and the UART connection was seen in the Register Transfer Level (RTL) schematics. Camera-Video Graphics Array (VGA) connections are also made with the Wishbone protocol. An implementation of the Laplacian filter with extended instruction set is made. As a result of many methods tried to access the extra memory DDR required by the data set, which is the main purpose of the project, it was decided that “bridge” application was the most useful one and simulation studies are continuing on it.

SÜRÜCÜ YORGUNLUK TESPİT SİSTEMİ VE UYGULAMASI İÇİN RISC-V İŞLEMCİSİ KOMUT SETİNİN GENİŞLETİLMESİ

ÖZET

Günümüzde görüntü işleme uygulamaları pratikte kullanılarak birçok sorunun çözümünde rol almakta, sürekli olarak geliştirilmekte ve hayatın birçok alanında karşımıza çıkmaktadır. Temel olarak elimizde var olan görüntüden anlamlı ifadeler çıkarmamıza yarayan algoritmalarından oluşan bu uygulamalar güvenilir ve amaca hızlıca varmayı sağlar yapıdadır. Görüntü işleme uygulamalarının günümüzde en çok kullanıldığı alanlara güvenlik ve yüz tanıma örnek verilebilir. Güvenlikten bahsederken insan hayatında çevresel faktörlerden korunmanın yanında, insanın kendi ihmalkarlığından doğan ve bir uyarı sistemine ihtiyaç duyulan durumlardan da bahsedilmelidir. Bu noktada sürücü uyku-uyanıklık tespiti uygulamaları öne çıkar. Günümüzde birçok ölümcül ya da büyük hasarlara yol açan kazalar, sürücülerin uyku ihtiyaçlarını yeterince karşılayamamış olmaları durumunda araç sürmelerinden kaynaklanır. Uzun yolculuklar sırasında sürekli farklılık gösteren çevresel koşullara maruz kalan ve uykusuzluk çeken sürücülerin yorgunluk, halsizlik durumlarının birçok göstergesi olup; bu göstergeler dikkate alınarak tasarlanabilecek tespit ve uyarı sistemleriyle birçok elim kaza önlenir. Bu tip tespit mekanizmalarının tasarımı için en güncel uygulama, güvenilirliği ve yapısal hızı nedeniyle gömülü sistemlerdir. Gömülü sistemler, direkt olarak kendisine atanmış görevleri yerine getiren hedefe yönelik kişisel tasarımlardır. Geliştirilecek olan uygulama adına özel bir yazılım ve donanım parçalarının birleştirilmesiyle geliştirilmişlerdir. Donanımın yanında yazılım bileşeninin en büyük önemi bu sistemin diğer gömülü sistemlere elverişli çalışabilmesi, değişen koşullara göre adapte olup güncellenebilmesi adına bir destek sağlamasıdır. Bir gömülü sistem donanımı, beraberindeki yazılım bileşeni geliştirilip değiştirilerek farklı algoritmalarla birlikte farklı amaçlara hizmet edebilir. Projemizin genel amacı olan sürücü uyku-uyanıklık tespiti sisteminin gerçekleştirilmesinde de bir gömülü sistemi görüntü işleme algoritmalarıyla birlikte istenilen amaca yönelik bir sisteme dönüştürmek hedeflenmiştir. Bu sebeple, istenilen programları geliştirebilmek ve Custom IP'ler tasarlayabilmek adına açık kaynaklı bir işlemci ile çalışılmıştır. Bununla birlikte maliyet, hız ve kullanılabilirlik açısından en optimum seviyede bir sistem tasarlamak amaçlanmıştır. Kullanılabilirlik konusunda ihtiyaç duyulan en önemli unsur hafıza ve depolama alanıdır. Bir işlemcinin, işlevlerini yerine getirirken erişebildiği hafızada (RAM) belirli bir miktarda veri tutulabilir. Projemizde görüntü işleme algoritmalarını üzerinde kullanacak olduğumuz veri, uykulu-uyuşuk durumdaki sürücülerin resimlerinden oluşmaktadır. Projemizde kullandığımız kart olan Nexys 4 DDR FPGA, sahip olduğumuz veri setinin ve geliştireceğimiz uygulamanın saklanabilmesi adına gerekli olan hafızaya sahip değildir. İhtiyaç duyduğumuz bu ekstra hafızayı yine kullandığımız kartın üzerinde bulunan harici hafıza DDR ile elde etmek hedeflenmiştir. Projenin en temel amacı olan hafıza sorununu çözerken, kullanılan haberleşme protokolü ve DDR'ı sorunsuz iletişime sokabilmek için birkaç yöntem bulunmuştur. Kullanılan açık kaynaklı işlemci olan Ibex ile proje boyunca Wishbone haberleşme protokolü kullanılmış, ancak bahsedilen ekstra hafıza olan DDR'a erişmek için ihtiyaç duyulan MIG (Memory Interface Generator)'ın işlemciye yalnızca AXI bağlantısı ile bağlanabildiği ortaya çıkmıştır. Buna göre bulunan yöntem köprü adı verilen ve AXI-Wishbone bağlantısını sağlayarak DDR'a erişebilmeyi sağlayan bir yapıdır. Haberleşme protokolünün iyice

anlaşılması için GPIO ve UART bağlantıları sağlanarak çeşitli simülasyonlar ve gerçeklemler yapılmış, en sonunda da görüntü işleme algoritmaları gerçekleştirilmiştir. Kullanılan haberleşme protokolüyle birlikte kamera bağlantısı ve alınan görüntüyle filtre uygulamaları yapılmıştır. Komut seti genişletilmiş bir işlemci ile birlikte Laplacian filtresi gerçekleştirilmiştir ve ihtiyaç duyulan hafızaya erişilmesi durumunda sürücü uyku-uyanıklık tespiti konusunda oldukça hızlı ve verimli çalışan bir sistem elde edilecektir.

Sürücü uyku-uyuşukluk tespiti yapan bu sistemin tasarımında izlenen yol sırasıyla verilmiştir:

1. Açık kaynaklı IbeX işlemcisinin gerçekleştirilmesi
2. Wishbone protokolünün işlemciyle uyumlu hale getirilip çeşitli bağlantıların yapılması
3. GPIO, UART ve kamera bağlantılarının test edilmesi
4. DDR'a erişebilmek adına 'bridge' yöntemiyle Wishbone-AXI-MIG-DDR bağlantısının sağlanması
5. Geliştirilen Laplacian filtresi ile görüntü işleme uygulamalarının yapılması

Projenin sonunda IbeX başarıyla gerçekleştirilmiş, Wishbone protokolü ile bağlantıları sağlama konusunda yetkinlik kazanılmıştır. En başta basit bir led çalıştırma koduyla Wishbone protokolü kontrolünde ledler çalıştırılmış, UART bağlantısı RTL şematiklerinde görülmüştür. Kamera-VGA bağlantıları da Wishbone protokolüyle birlikte gerçekleştirilmiştir. Laplacian filtresinin, genişletilmiş komut setiyle birlikte bir uygulaması yapılmıştır. Projenin temel amacı olan veri setinin ihtiyaç duyduğu ekstra hafıza DDR'a erişme konusunda denenen birçok yöntem sonucunda en kullanışlı olanın Bridge uygulaması olduğuna karar verilmiştir ve üzerinde simülasyon çalışmaları yapılmaya devam edilmektedir.

1. INTRODUCTION

1.1 Motivation

Using hardware systems for image processing by artificial intelligent algorithms is an important topic. However; the different way of representation of the numbers in hardware systems can result in faulty than acceptable level when it is implemented with RISC-V processor [1]. In this project, it is aimed to extend the instruction set of RISC-V processor by changing the arithmetic logic unit of the processor with using image processing and artificial intelligent algorithms [1].

1.2 Objectives

Main objective of the project is to implement the artificial intelligent algorithms for image processing to the open-source RISC-V processor [1]. We aimed to save time and cost and decrease the complexity in system-on-chip designs by using open-source processor with applying specific extensions. Our first objective was to synthesize and simulate a specific RISC-V core and to fully understand how it is written [1]. To be able to extend its instructions we aimed to clearly see how it works and applies the existing instructions. Our second objective was to implement a Driving Fatigue Detection (DFD) system [3] which had written in C programming language [2] on the RISC-V processor [1] using artificial intelligent algorithms. We tried to make this system work properly on our processor without any number representation faults. To determine which algorithm to use to make changes in arithmetic logic unit of our open-source processor we found the most used function by analyzing the program. Then we implemented the decided function to the arithmetic logic unit. To reach a bigger memory capacity which makes the DFD system [3] work more properly, we used an external memory called DDR RAM [4]. After editing the instruction set, adding them

to the arithmetic logic unit and connecting a DDR RAM [4] to our system we connected a Video Graphics Array (VGA) [5] module lastly.

1.3 Thesis Contribution

The construction phase of the project, which is the subject of this thesis, has been advanced with the contributions of The Scientific and Technological Research Council of Turkey (TUBITAK) and the Ministry of Science, Research & Technology of Iran (MSRT). The project had the project number ‘119N461’.

1.4 Organization of This Study

Project’s stages are connected sequentially. That is why we took place in each stage together. First, we implemented Ibex based core [6] on Vivado [42] and on board. In the next stage, we prepared a documentation for to prevent mistakes in our future work. Afterwards, connected Direct Memory Access (DMA) to Microblaze since we wanted to access DDR RAM [4]. We added a Wishbone interface [7] to Ibex. Then, we started to integrate VGA and camera components to Ibex via. Wishbone interface. The image taken from camera sent to VGA screen. Then, we implemented the image processing algorithms needed for driver fatigue detection system as software only on Ibex processor [6].

2. LITERATURE REVIEW

There are other studies that detect driver inattention, such as this project we carried out to reduce the number of accidents due to driver distraction. A wide variety of methodologies are used [8]. Existence of many studies can be observed that train artificial neural networks or use machine learning and image processing as we do. For example, by image processing, smoking or eating behaviors can be detected. The first of the two points where our project differs is the eyes where we focus from driver behavior and the second is choosing RISC-V to do FPGA implementation [1]. Firstly, distraction can be best determined from the eye movement data since other behaviors are not necessarily mean driver distraction. Therefore, our methodology will be giving the best output to determine critical situations. Secondly, in this project, we preferred FPGA implementation of RISC-V processor over other Instruction Set Architecture designs because of the advantages in complexity and cost [1]. RISC-V enables user to change and update versions of Instruction Set Architecture, minimize logic gates and energy consumption, customize their work by instruction extension as we plan to do in this project [9] [10]. When we need to choose a based core in RISC-V, we choose Ibex-based core [6]. It is fast, small, open-source processor and it supports I, E, M, C and B extensions. Generally, our methodology will both protect the driver in the most effective way and do this in the simplest and cheapest way.

3. RISC-V

In this project, RISC-V has been used as the main processor [1]. The implementation process has been done on an FPGA. RISC-V is an instruction set architecture [1]. It is a well-organized Instruction Set Architecture divided into categories and extensions.

3.1 Instruction Set Architecture:

Instruction Set Architecture (ISA) [12] defines the software interface for hardware. ISA is a specification used to understand how to program a hardware that has to do implementation. A single ISA can have many implementations and is a specification may be for general purpose microprocessors, Digital Signal Processors (DSP's) and specialized hardware operations. Because the software for an ISA can be reused, an ISA creates its own software ecosystem. This is the reason why x86 is dominant on servers and arm is more dominant in mobile. ISA defines everything visible to software [11]. It includes the set of instructions and how they behave the data types registers addressing modes and more such as how the memory model works protection levels that accommodates system software.

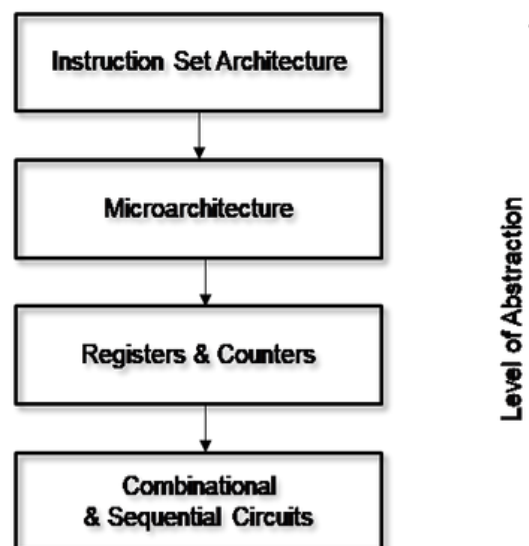


Figure 3.1: Level of abstraction in ISA [12]

3.2 RISC-V History

In 2010 at Berkeley University Computer Architecture group, Kryste Asanovic started the design as a 3-month summer project to use in their next set of projects. The ones that the project team used were too complex and they needed a simpler one. Andrew Waterman, Yunsup Lee, David Patterson and Kryste Asanovic himself were the principal writers of the RISC-V [1]. In May 2014, the user specifications were released after many iterations. That way, RISC-V became the fifth ISA designed in Berkeley. As you can see from the graph, the usage of RISC-V is increasing every day [13].

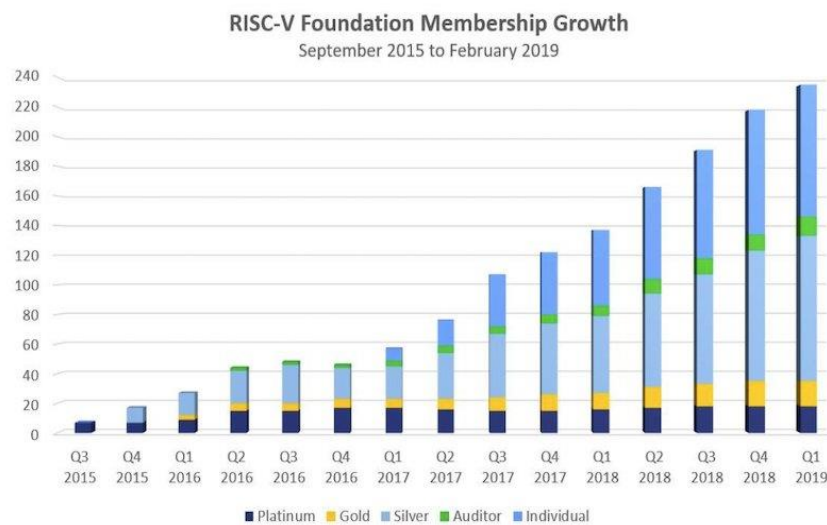


Figure 3.2: RISC-V membership growth [14]

3.3 Advantages of RISC-V

RISC-V is seen as a disruptive innovation among Instruction Set Architecture (ISA)'s and is used by many individuals/companies [1]. There are many reasons why it is so preferred. RISC-V was designed with a clean slate and it is an open source that can be downloaded from git repositories easily and used to design hardware free of Intellectual Property (IP) and licensing restrictions. Since companies can build their own RISC-V easily, it created a new business model and its development became really fast thanks to the open-source nature of this project. Needless to say, RISC-V architecture is on par with modern Central Process Unit (CPU)s in terms of performance, code density and power consumption. It is also royalty-free, has a standard maintained by the non-profit RISC-V foundation and is suitable for all computing systems [15].

3.4 RISC-V Assembly

To understand RISC-V basic assembly code generation, the logic behind an Instruction Set Architecture (ISA) should be covered [1]. ISA defines how to program a machine with the instructions, stateful elements that define the context of a process that is the value of the registers in the memory for a given point in the execution of a program. This notion of the progress is important to the operating system. System programs provide a layer of abstraction and modularity between application programs and ISA. This helps code reuse and application portability. The stateful elements of the computer that program sees are program counter, register file and memory. The program counter is a pointer to an instruction address in RISC-V [17]. This could be 32 or 64 bits. The register file contains the registers defined by the architecture such as the 32 general purpose RISC-V registers from x0 through x32 [17]. Some ISAs also define condition codes separately from the registers to store information about recent instruction results such as integer overflow, inequalities and zero valued results. Memory is logically viewed as an array of bytes that contain both code and data. The logical view is usually divided into several sections or segments. This is usually done at least to separate the program code usually called text read-only data, stack data and program heap data. The base RISC-V architecture uses a little-endian byte order which means the little end of the word comes first in memory [1]. So, the least significant byte is located at the first lowest memory address of the word containing it. A program is a sequence of machine instructions and data created by compiling high level source and assembling assembly language source into object code that is linked to create the program. The format of machine instructions is defined by the architecture which provides a one-to-one mapping to readable assembly instructions. This includes the opcodes and the operands of each instruction. So, a given line of assembly corresponds to one machine instruction and vice versa. A program is created through a sequential process that uses a compiler to convert source code into binary object code and then a link editor or linker brings together multiple object code files to create the program. At runtime, a loader which is usually part of the operating system brings the program into memory so that it can begin execution on the processor. There are 32 general purpose registers of the RISC-V architecture that can be seen in the Figure 3.3 with their Application Binary Interface (ABI) names [17]. These are the registers that a program uses explicitly for integer operations. There is another set of registers for

floating point operations. The general-purpose registers are numbered prefixed with an x from x0 to x31. Unlike other assembly languages, RISC-V tools do not use any prefixes other than the x for the register names [16].

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 3.3: Register names in RISC-V [17]

3.5 Custom Instruction Set Extensions of RISC-V Processor

In order to increase the performance, adding custom instructions was necessary. Since RISC-V is an extendable instruction set architecture, we were able to add custom instructions [1].

3.5.1 Multiplier and accumulator operator

Multiplier and accumulator (MAC) operator [19] is a special digital signal processor hardware unit known as multiplier and accumulator. It simply calculates the product of two number and adds them to an accumulator. This part of the project was done on the MAC (Stunning Vivado Flash Scripts collector) unit used in the project. Since the MAC unit will be used by many structures, its performance is of critical importance for the whole system. It also has fine points in the design. At this point, studies were carried out on the parallel striking model and the aim was to decrease the operation

time of MAC by adding a hardware. As you can see from the figure, it has a serial structure. The serial structure means that each input needs the previous output [18].

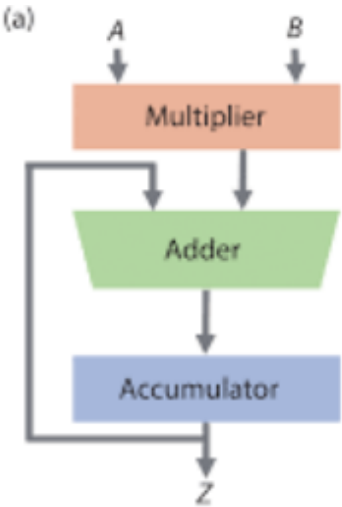


Figure 3.4: Serial structure of MAC operator [19]

3.5.2 Convolution and multiply-accumulate

Sometimes, results that we got from hardware and software can differ from each other. We decided to pre-run the tests to avoid this causing a problem. According to the simulation results, we observed that hardware calculations are taking much more time than the software calculations [15].

Convolution is an iterative operation repeating the instructions, allows us to observe the cycles and their timing [15]. An additional hardware would solve the problem by lowering the execution time.

In the previous section, it can be seen that MAC unit has a serial structure. However, by adding a hardware, we got a new parallel structure. It is shown in figure. Digital Signal Processing (DSP) tools are used for the multipliers. In order to optimize convolution operators, stated operations will be added as custom instructions [18].

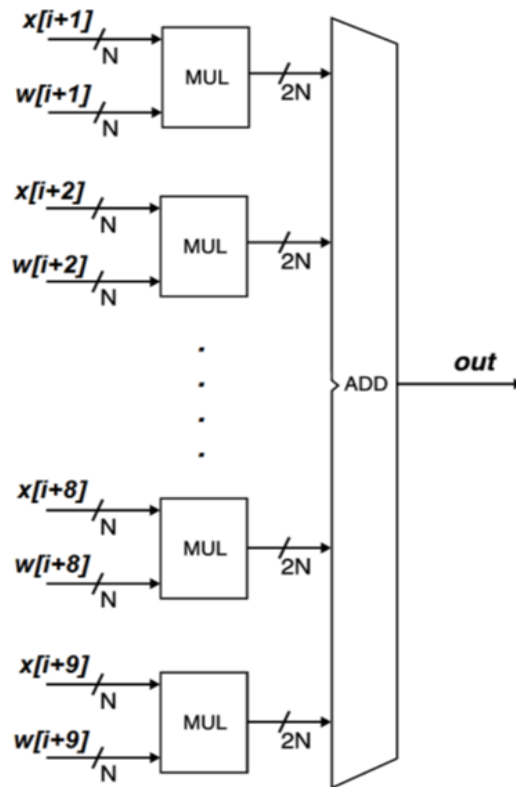


Figure 3.5: Parallel structure of MAC operator [19]

3.5.3 Customization of RISC-V

Customizations means adding a new custom instruction on top of the instruction set which we aimed in the previous part. Every RISC-V needs to support the basic instruction sets but it do not prevent us from adding our own custom instruction [1]. In this project, Arithmetic Logic Unit (ALU) instruction decoder unit needed to be customized [15]. The scope of changes needed for customizing RISC-V and instruction path in a processor should be known in order to add the custom IP. To tame the combinations of functionality that can be applied, a nomenclature is defined to designate them in the currently approved privilege ISA specification.

With the increase in the number of extensions, the standard now allows extensions to be named with a single "Z" followed by an alphabetical name and an optional version number.

In order for the multiplication to be added to the processor, there must be an unused instruction in the application. For this project, the extraction command, which was determined not to be used by looking at the assembly code of the filter code, was chosen. The subtraction command has been converted to the multiplication command, by changing the arithmetic logic unit [15]. After this point, the processor will perform multiplication instead of the instruction that comes as subtraction. The point that should not be missed here is that if we multiply two thirty-two-bit numbers, the result will be sixty-four bits, and this will cause problems since our processor is a thirty-two-bit processor. To prevent this, the multiplied numbers were chosen to be sixteen bits. This is not a problem, since numbers with a size of sixty-four bits will not be used in practice [15]. In order to test the new state of the processor, simple multiplication operations were performed first. As confirmed by the results, the subtraction command has been changed to make the multiplication command.

For the future image processing work, we decided to customize kernel value as we did with adding a custom instruction [15]. We repeat the same processes for each convolution.

Shortly, in order to increase the performance, we extended the custom instructions by characterizing the application, developing new custom instructions, characterizing new application, optimizing the model in this order.

4. IBEX CORE AND WISHBONE SYSTEM-ON-CHIP INTERCONNECTION ARCHITECTURE

An Ibex core is simply a 2-stage in-order 32-bit RISC-V processor core [1]. It is designed to be small and efficient [6]. It has U-Mode, M-Mode, Physical Memory Protection (PMP) and it has been written in system Verilog. It has been initiated by ETH Zurich and developed by lowRISC which is a non-profit organization [20]. It has mostly 'I' but optionally 'E' extension and control and status register access. In the Figure 4.1, a basic block diagram of Ibex is shown.

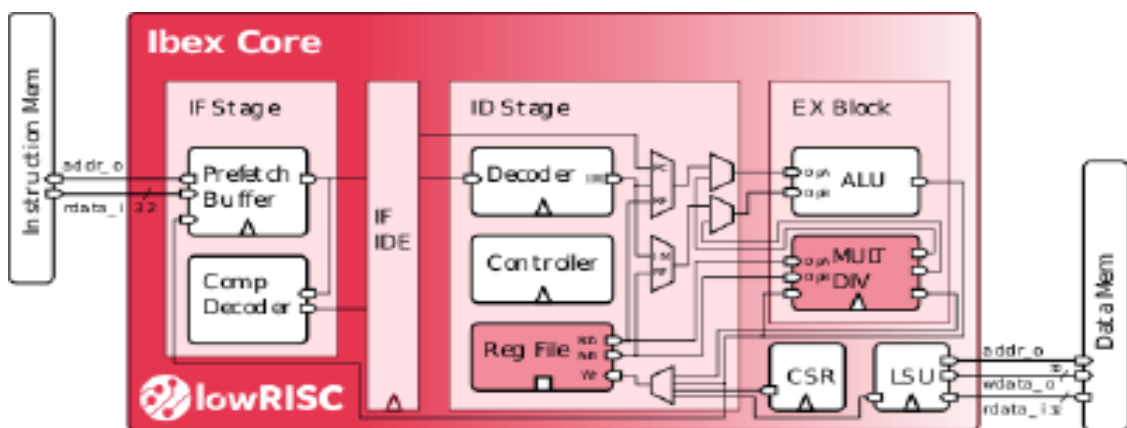


Figure 4.1: The block diagram of the small parametrization with a 2-stage pipeline [21]

4.1 Benefits of Ibex

It is an open source which encourage new hires, peer-to-peer help and this leads a quick progress towards innovative developments. LowRISC [20] the company has many documentations stating the progress and final results. Also, they are really open to communication and they share all of the tools, scripts, codes [15]. That is why, it is quite easy to solve problems with Ibex core [6]. Ibex core is also easy to modify and has understandable source codes. It supports various code structures and is very useful for image processing applications which we worked on. So, even though RISC-V [1] has many cores and System on Chips (SoCs) that can be implemented on FPGA, we decided to work with Ibex throughout the project.

4.2 Wishbone Protocol Explanation

Wishbone Protocol is an open standard protocol [7]. Most of the communication protocols enables user to communicate with a certain chosen device not like Wishbone for which no payment is required. It allows to have a topology; a bunch of devices talk to each other. It is not logical to have a bunch of custom interfaces stringed together to talk to different designs rather than having a guaranteed communication protocol like Wishbone Protocol [15]. Using a common protocol, it is possible to take and put a video decoder and it will be able to talk on the same bus. It gives the flexibility to reuse the designs without a need of reinventing them.

Interfaces are important to capture these protocols rather the signals and bundle them into an easily expendable structure. A basic Wishbone protocol consists a Wishbone master and Wishbone slave [7]. They are connected as in the figure below. It has two common signals that both units see the reset in the clock and these come from your system controller. For example, to indicate that data has been captured, acknowledge signal is sent.

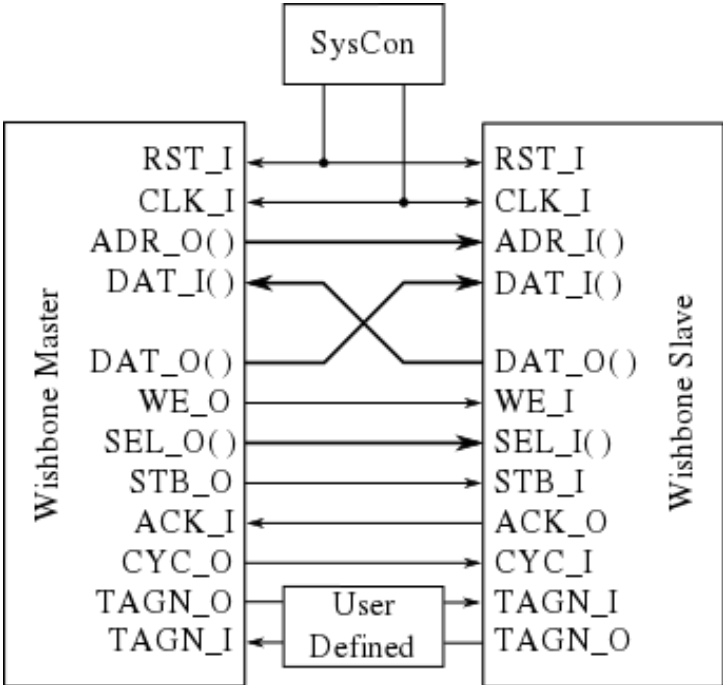


Figure 4.2: Master and slave Wishbone's interfaces [22]

These pins actually all the same but have different directions. System Verilog allows for the concept of an interface and the Wishbone protocol can be extended with new

signals so interface grows and everybody can see it [7]. In order to propagate these changes manually to each of the designs, we use interfaces. Interfaces are an abstract data type that actually act like a module that sit in between user and user's design. There is a set of common signals and they usually exist withing the interface. In order to give them directions and to actually associate them with a particular input or output, we create what are called mod ports. Mod ports allow us to define the direction of data. So, we say that there is an address object inside the Wishbone. Interfaces have a very strong structure which strongly checks whether or not the assigning inputs to outputs correctly that Verilog do not have. A lot of people misconnect the signals therefore a protocol that guarantees us that we do not have problem with the misconnections in this project was very advantageous. Wishbone also allow us to propagate common signals outside of the port definitions or rather along with the mod port definition.

4.3 Wishbone Signals

Ibex core was implemented however in order to communicate with cameras, screens or computer buses we needed a communication protocol [6]. Wishbone Protocol is an open source and logic bus that provides communication between integrated circuits via signals [7]. It transfers signal information (clock cycles, high/low levels) instead of electrical knowledge. Wishbone interface is flexible and compatible with Ibex and provides communication between all IP's, masters and slaves. Portable IP cores were used for SoC design to provide reliability. In order to communicate, all the components share a bus called interconnect [16]. In the Figure 4.3, signals and signal directions of the Wishbone and tag types interface can be observed.

Tag Types

Description	Master		Slave	
	TAG TYPE	Associated with	TAG TYPE	Associated with
Address tag	TGA_O()	ADR_O	TGA_I()	ADR_I()
Data tag input	TGD_I()	DAT_I()	TGD_I()	DAT_I()
Data tag output	TGD_O()	DAT_O()	TGD_O()	DAT_O()
Cycle tag	TGC_O()	Bus Cycle	TGC_I()	Bus Cycle

Figure 4.3: Tag types [23]

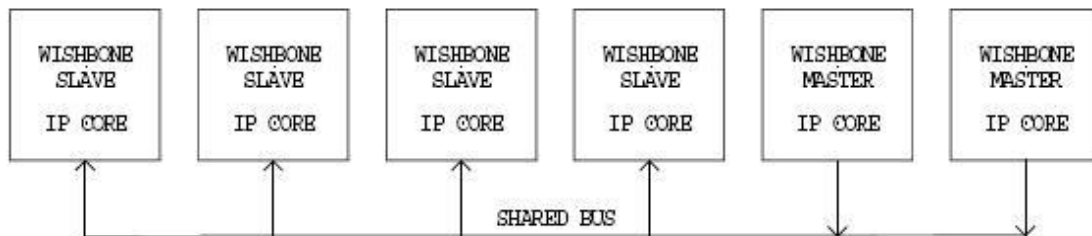


Figure 4.4: Wishbone shared bus [22]

Shared Bus technique is an advantageous technique since shared communication systems are compact systems. It requires fewer logic gates and routing resources. However, while being compact and less costly shared bus can be slower than other configurations [15].

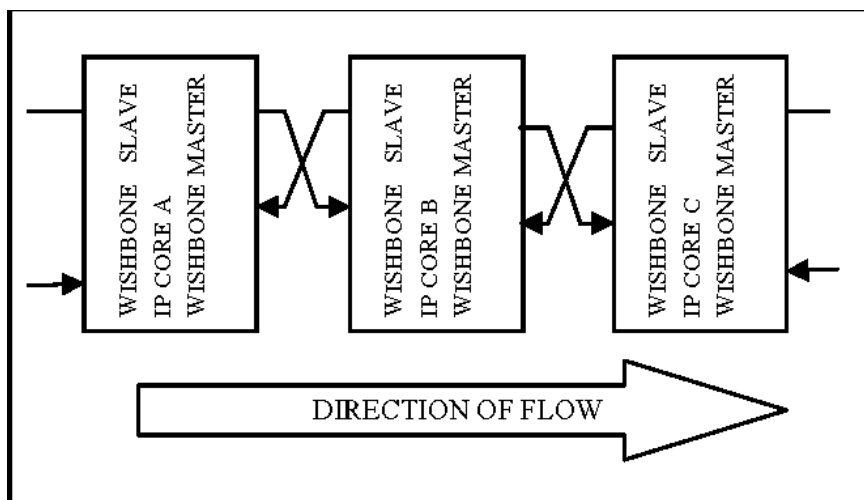


Figure 4.5: Wishbone direction of data flow [22]

As it can be seen in the 4.5, Wishbone interconnection has a parallelism property which enables us to execute more than one function [7].

4.3.1 Single READ / WRITE cycles

In order to make data transfers on Wishbone interconnect, these cycles are used [7]. At one time, these Single Read/Write cycles perform one data transfer.

4.3.2 Block READ / WRITE cycles

Multiple data transfers are performed by the BLOCK transfer cycles. They're similar to single READ and WRITE cycles, but with a few tweaks to accommodate multiple transfers.

The interface basically conducts SINGLE READ/WRITE cycles throughout BLOCK cycles, as explained above. The BLOCK cycles, on the other hand, have been somewhat altered so that these distinct cycles (known as phases) are joined to produce a single BLOCK cycle. When many MASTERS are utilized on the connection, this capability is quite handy. If the SLAVE is a shared (dual port) memory, for example, an arbiter for that memory can determine when one MASTER is through with it so that another can access it [24].

4.3.3 RMW cycle

For indivisible semaphore actions, the RMW (read-modify-write) cycle is employed. A single read data transmission is made throughout the first half of the cycle. A write data transmission is executed in the second half of the cycle [11]. During both half of the cycle, the [CYC O] signal is asserted [16].

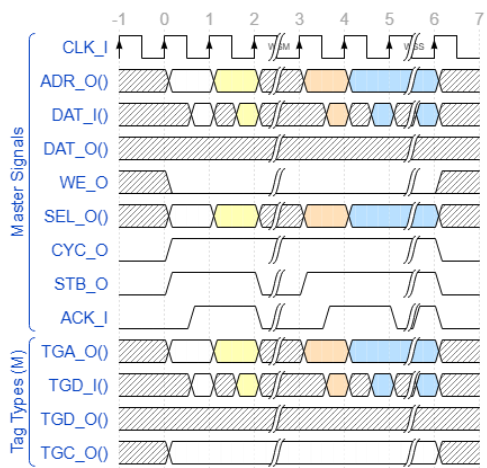


Figure 4.6: Block read cycle [24]

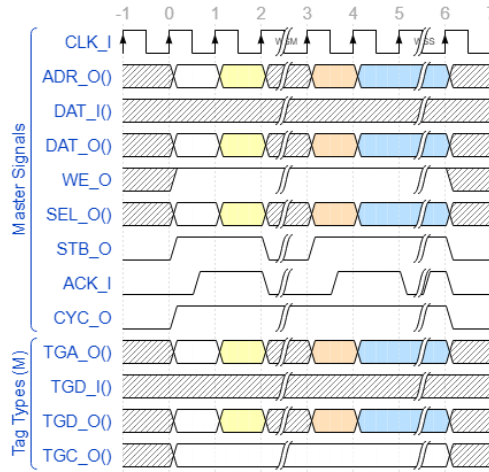


Figure 4.7: Block write cycle [24]

After these steps, Wishbone added Ibex core were tested with a basic LED code. The expected result was observed on the LEDs on the FPGA board.

5. IMAGE PROCESSING

The alteration of photographs using digital computers is known as digital image processing. In the previous few decades, its popularity has skyrocketed. Its uses vary from medicine to entertainment, with geological processing and remote sensing thrown in for good measure [25].

Digital image processing [25] is a broad field that includes both digital signal processing and picture-specific approaches. A function $f(x, y)$ of two continuous variables x and y can be considered a picture. It must be sampled and turned into a matrix of numbers in order to be processed digitally. Because a computer can only represent numbers with finite precision, they must be quantized before being represented digitally. The handling of those finite precision numbers is what digital image processing is all about.

There are many types that an image can be processed with. Image restoration which is basically reverting vitiated images is one of these types. Image enhancement is another type in which heuristic techniques are used to obtain beneficial information from an image. There is an image processing type called image compression which reduces the cost for storage or transmission of an image. The most commonly used and the type that our project is all about is image analysis. All of these classes of image processing have basically similar techniques and differ according to their intended use.

5.1 Image Analysis

Image analysis is a broad word that refers to procedures and strategies for interpreting and parametrizing data from an image or series of images. Obtaining error estimates for the generated parameters and assessing picture dependability are also part of these techniques [26].

Pixels (short for Picture Element) are used to divide an image; each pixel represents the smallest single point on the screen. A pixel can only be one color at a time, and the color of each individual pixel must be saved in binary when a picture is saved. The

more bits required to store each pixel, the greater the number of colors possible. Each pixel is represented by a set of bits.

In actuality, photographs are significantly more complex than what is seen, with color depth and image resolution impacts. The color depth refers to the number of bits per pixel. It has a significant impact on the image's color quality. Since the 0 symbolizes black and the 1 represents white in photographs, only two colors can be utilized if each pixel is encoded with one bit (2^1). Each pixel requires additional bits to represent more than two colors in an image. The minimum needed color depth from the number of colors in an image is determined as 2^n , in which n represents the number of bits. For instance, an image with 8 bits can demonstrate 2^8 colors.

An image is a collection of square pixels ordered in columns and rows in a matrix. Each pixel in an 8-bit greyscale image has an assigned intensity that runs from 0 to 255. A grey scale image is similar to a black and white image, but the term emphasizes that it will contain a variety of shades of grey [27].

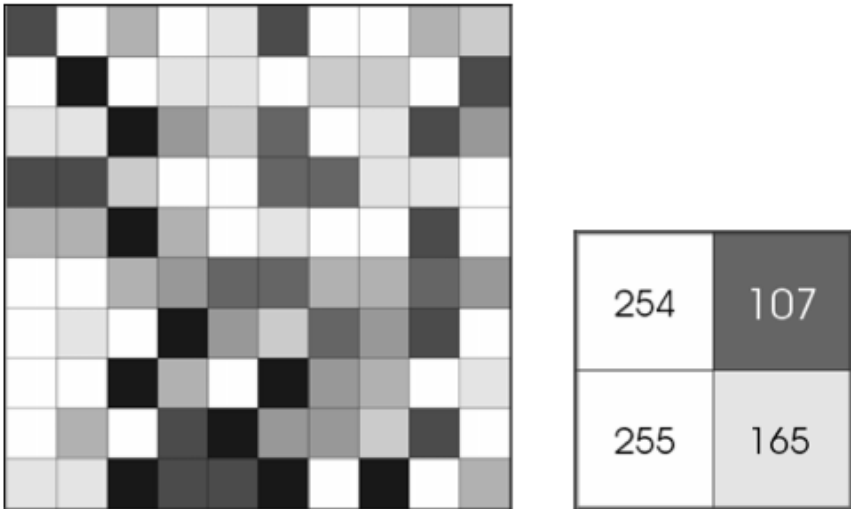


Figure 5.1: Representation of pixels [27]

When the definition of a color depth is considered, the figure above which is a grayscale image has 8-bit color depth and $2^8 = 256$ greyscales. When a true color image is considered which has 24-bit color depth, by the definition it has 2^{24} (more than sixteen million) colors.

5.1.1 Colors

The goal of color spaces in image processing is to make it easier to specify colors in a consistent manner. Different forms of color spaces are utilized in a variety of professions, including hardware, animation, and other fields. The goal of the color model is to make color requirements more standard. Different forms of color models are utilized in a variety of industries, including hardware, animation, and other applications.

There are different types of color spaces which are used in specify colors of images in processing. Red, Green, Blue (RGB), Luminance, Chorama Blue & Chorama Red (CMYK) and Hue, Saturation & Value (HSV) are some of those color spaces. The most common used one is RGB color model in digital image processing [28]. RGB is referred to red, green and blue colors. These three colors are main color components of this model. The proportional ratio of these three colors produces all other colors as seen in Figure 5.2. The main color space used in image processing part of this project is RGB model.

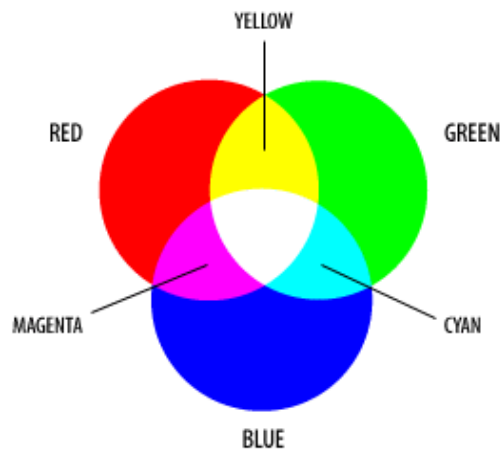


Figure 5.2: RGB color model [28]

There is another common color space called Luminance, Chrominance1 & Chrominance2 (YCbCr) [29] which is also named YUV in which slightly different spaces are taken. The luma component of the color is represented by Y. The color's brightness is measured in luma. This refers to the color's light intensity. Cb and Cr are the blue-luminance and red-luminance components of

the chroma component, respectively which means Cb is blue and Cr is red related to the green component. Since the Y component is more sensitive to the human eye, it must be more accurate, whereas Cb and Cr are less sensitive.

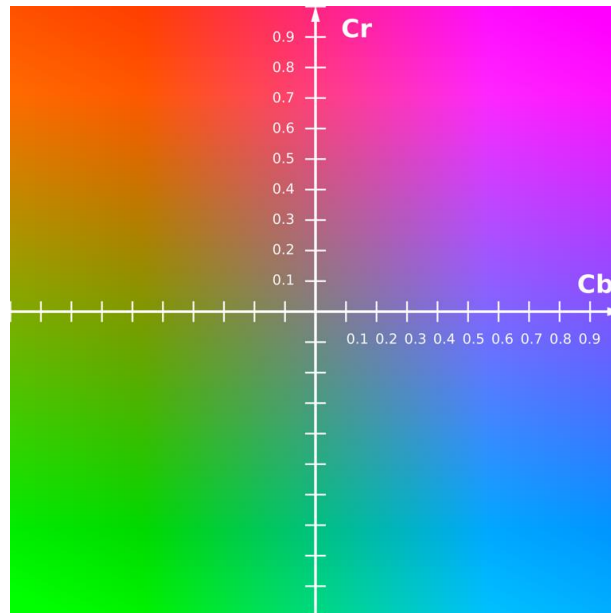


Figure 5.3: The CbCr plane at constant luma $Y'=0.5$ [29]

The YCbCr color scheme allows the computer to exploit these eye sensitivities to eliminate any unneeded features during image compression. YCbCr is useful for compressing images and video-type images, which need less data and demand less storage space [30].

5.2 Image Filtering

Image filtering is changing the appearance of an image by altering the colors of the pixels [31]. Digital images are enhanced and modified using filtering algorithms. They are also used to blurring, sharpening, edge detection and noise reduction applications on images. Image filtering techniques can be divided into two as in spatial and frequency domain. Edge detection filters are used in the low frequencies whereas smoothing filters are used in high frequencies. Spatial domain image filtering

techniques can be linear or non-linear and filters belong to these classes have the required characteristics.

The more commonly used method for filtering is spatial filtering [32]. Spatial filtering directly applies on the pixels. A linear filter can be created using convolution, which is just the linear sum of data in a sliding window. In the Fourier domain, it's equal to multiplying the spectrum by an image. Whereas convolution or Fourier multiplication cannot be used to create a non-linear filter. To give an example; Laplacian, Gaussian and Neighborhood Average filters are linear filters. As an example of non-linear filter Median filters are the most used ones.

5.2.1 Linear filters

Linear filtering is the most basic and fastest type of filtering. It replaces each pixel with a linear combination of its neighbors, and the linear combination is prescribed using a convolution kernel [33].

A kernel is a tiny matrix which is used for blurring, edge detection, sharpening and other tasks of image processing [34]. A convolution of this matrix and the digital image is used to achieve this. The convolution kernel in linear image processing is simply represented as follows:

$$y(t) = \int_{-\infty}^{\infty} h(r) \cdot x(t - r) dr \quad (5.1)$$

where $x(n)$ is the input signal and the $h(n)$ is the impulse response.

5.2.1.1 Laplacian filter

A Laplacian filter [15] is used for edge detection in image filtering. The second derivatives of a digital image are computed using a Laplacian filter, which measures the rate at which the first derivatives change. This determines whether a change in neighboring pixel values is caused by an edge or is part of a continuous progression. Negative values in upper and lower triangles are centered within the matrix, are common in Laplacian filter kernels. A high value is at the kernel's center, surrounded by lesser and oppositely signed values. By introducing abrupt changes in the output pixel values, the image's edges are highlighted [35]. The corners have either a 0 or a 1

value. The edges of the output image have bright tones, while the rest of the image has black tones. Gain is avoided since the sum of the kernel coefficients is hold at zero. In the array below, a 3x3 kernel for a Laplacian filter is represented.

$$G = \begin{bmatrix} 0 & -1 & 0 \\ -1 & 4 & -1 \\ 0 & -1 & 0 \end{bmatrix}$$

For image processing part of our project, Laplacian filter is implemented by using convolution operation. Each element of the image array is a pixel and each pixel is placed in the middle of a new matrix. The absent values of the newly formed matrix are taken as zeros. The formed matrix and the kernel are subjected to a convolution operation. The result is the new value of that pixel. These operations are represented in the Figure 5.4.

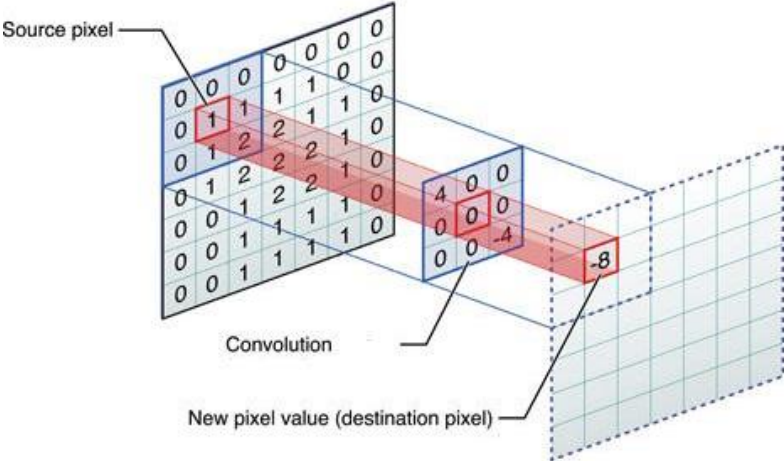


Figure 5.4: Convolution of an 7x7 Image and a 3x3 Kernel [36]

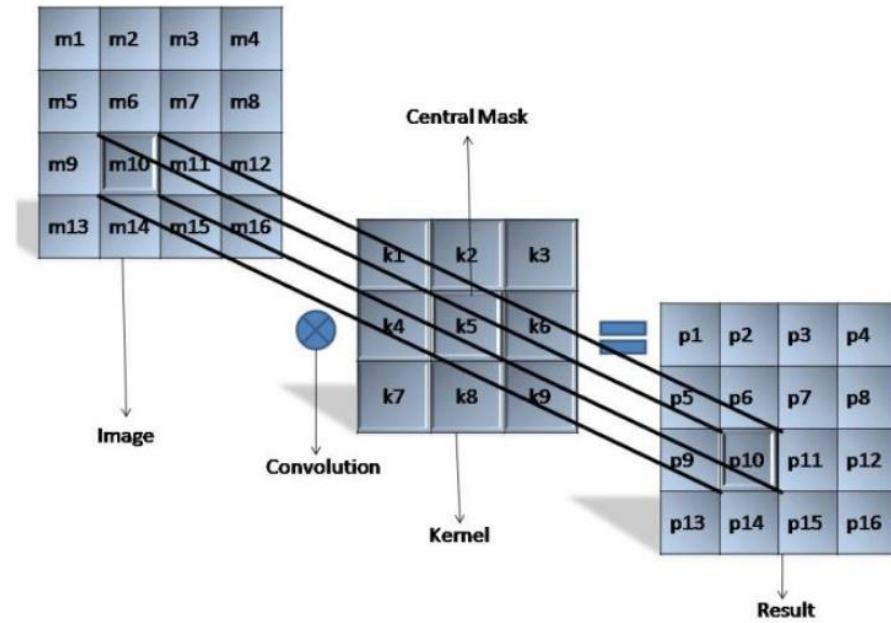


Figure 5.5: Convolution of an 4x4 Image and a 3x3 Kernel [37]

The image in Figure 5.5 is represented by the 4x4 matrix and the kernel is represented by 3x3 matrix. The matrix that results represent the image size in 4x4. The convolution process should be conducted as shown in the equation below.

$$P_{10} = (m_5 \times k_1) + (m_6 \times k_2) + (m_7 \times k_3) + (m_9 \times k_4) + (m_{10} \times k_5) + (m_{11} \times k_6) + (m_{13} \times k_7) + (m_{14} \times k_8) + (m_{15} \times k_9) \quad (5.2)$$

When using the Laplacian filter to identify abrupt changes in pixel values, the amount of the computed value in equation 5.2 increases, resulting in white pixels, as shown in Figure 5.6.



Figure 5.6: Edge detection example [38]

5.3 Image Filtering and Custom IP Design

Image processing and other calculational focused tasks used in electronics devices need massive amounts of data processing and take a lot of power. As a result, specialized IP design can be critical in data-intensive operations [39].

The convolution operation underlying the filters used in image processing consists of matrix multiplications. The power and capacity required for the number and functionality of these processes should be controllable by the user. The main purpose of Custom IP design [40] is to customize the filter design we use in the image processing part of our project and to manage it with commands. We aim to keep the multiplication operations in the convolution operation as a custom instruction set defined in the Arithmetic Logic Unit (ALU) of the open-core Ibex processor [6], instead of spending unnecessary time and power in a certain calculation cycle. In this way, we expect to perform the filtering we want by simply calling these commands.

The data we have within the scope of the Driving Drowsiness Detection (DFD) [3] project consists of the positions and states of the mouth and eyes of the drivers in certain waking states. With the edge detection filter, the mouth-eye images of the drivers in the mentioned data will be detected with the Laplacian filter and kept in the Block RAM [41], so that the processor will perform the necessary operations. With the command set we have expanded, it is expected from Laplacian filter to perform the operations, to make high-accuracy detections and to realize the purpose of the project.

6. IMPLEMENTATION OF IBEX CORE AND TESTING HARDWARE

In this part of our thesis, first of all, the design environment we use and its installation are mentioned in detail. Then, the processor in our system, Ibex [6], and the reasons for choosing this processor are mentioned. In order to better understand Ibex and to understand its working mechanism, the tests and applications we have done together with the communication protocol we use are also included in this section. All processes are explained in order and the results are also shown.

6.1 Installing Vivado

The software package, Xilinx Vivado Suite, is the platform where we performed all our hardware-related operations throughout the project [42]. From the ‘Xilinx Unified Installer’ located in the official Xilinx website shown in Figure 6.1, we downloaded the program by logging in to our Xilinx accounts. While downloading, it is important to download one with .tar.gz extension file. After completing the installation process, we started to create an FPGA project which we chose NEXYS 4 DDR as in the Figure 8.2 [43]. We added sources, inputs and outputs of the project and move to the processes about our RISC-V [1] processor code called ‘Ibex’.

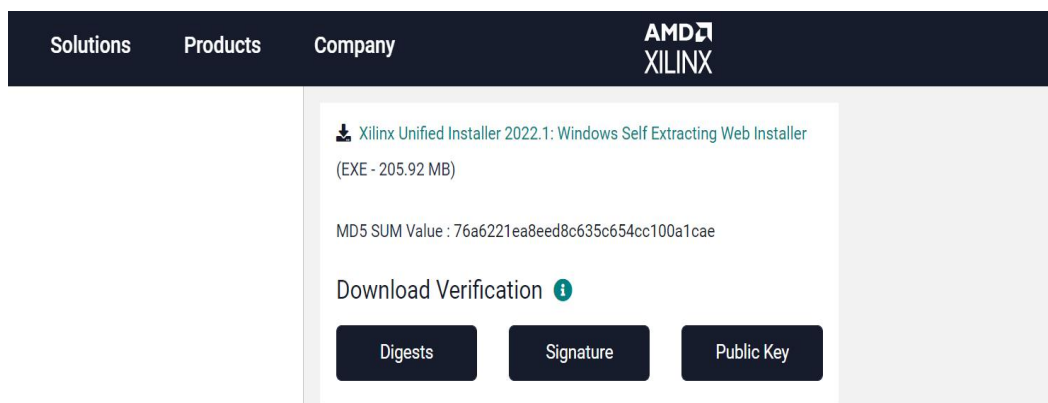


Figure 6.1: Xilinx website

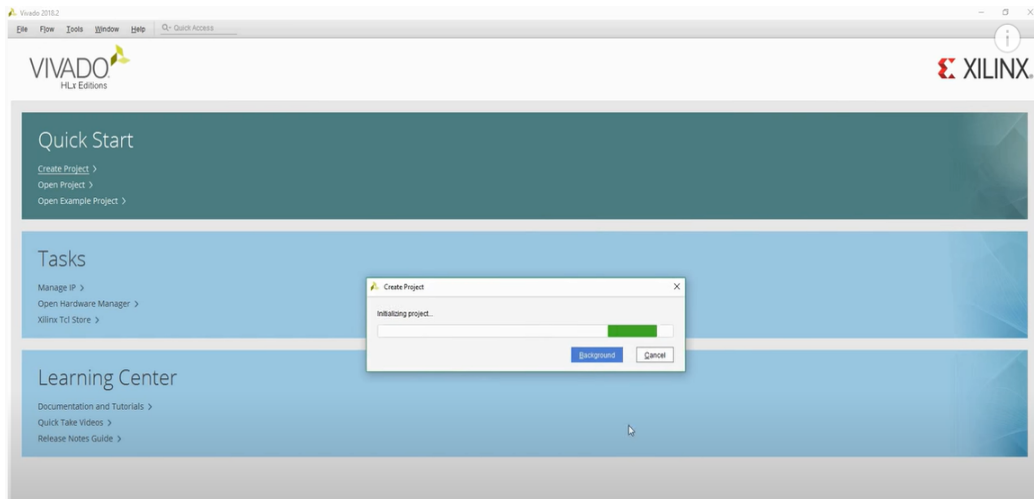


Figure 6.2: Vivado interface

6.2 Setting Up RISC-V GCC Toolchain

A compiler and linker, together a toolchain, is needed for the project. The suitable toolchain for "RV32IMC" is available on github. The github project called 'RISC-V GNU Compiler Toolchain' [44] which includes GCC (GNU Compiler Collection) is compiled for this part of the project [15]. It has also a C library for RISC-V and we were able to change the architecture regarding our project [1]. For example, the bit number was not compatible with our project, we could easily change an argument for a different bit number. We first cloned the github project and installed the required packages. It is a cross compiler that can convert instructions from the processor it is running on to machine code or low-level code for another processor. For projects with 64-bit RISC-V core, the 32-bit RISC-V that we downloaded was not enough, and we repeated this process for 64-bit. A .vmem file from a C file is generated and while blinking LED's on FPGA we will use this file to write a simple C code.

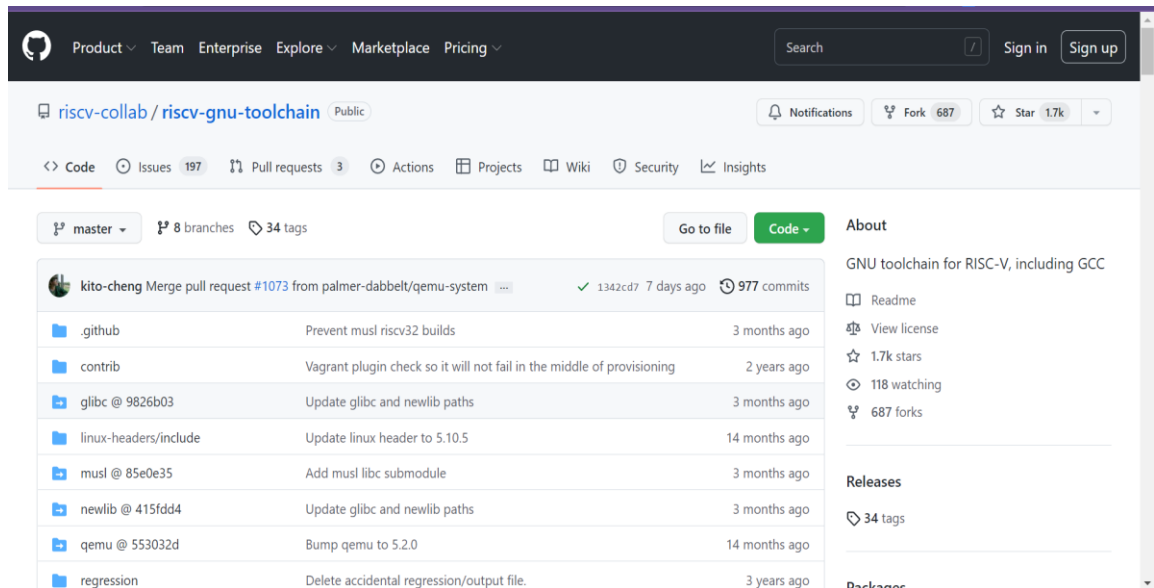


Figure 6.3: Github code for RISC-V toolchain

6.3 Installing, Synthesizing and Implementing Ibex Core

The first thing that we did was to synthesize and implement the Ibex core on Vivado and try to understand the several configuration parameters to meet the needs of various application scenarios that Ibex offers [6]. We took the Ibex repository from GitHub and we chose the example in the directory in which the top module is `top_artya7.sv` module. Then we added all the other modules with respect to the source hierarchy as seen in Figure 6.5. There is an issue in Ibex that it requires a physical ram to read instructions and perform read/write operations with data. With some changes in Ibex core modules for initializing the ram. Then we generated a memory file with a RISC-V GNU compiler to run it in the project that we created with Ibex Core [1]. We generated the memory file as an elf file in the Linux system and then converted it into a `.mem` file to be able to use it in the project in Vivado.

To clearly see that our core is successfully realized we compiled a C code that counts up to fifteen. This code is written to be run on the card, and if examined carefully, “`usleep (1000 * z000); // 1000 ms`” line can be seen. This line of code provides a 1 second delay so that the change in LEDs can be observed on the card. If RTL is to be simulated, this delay time needs to be reduced because a one-second delay for the simulator means simulating for very long periods of time. For example, if the line in the code that we mentioned before is changed, the changes in the LEDs can be easily

observed in the simulation. The C code that we compiled is shown in Figure 6.6 and Figure 6.7.

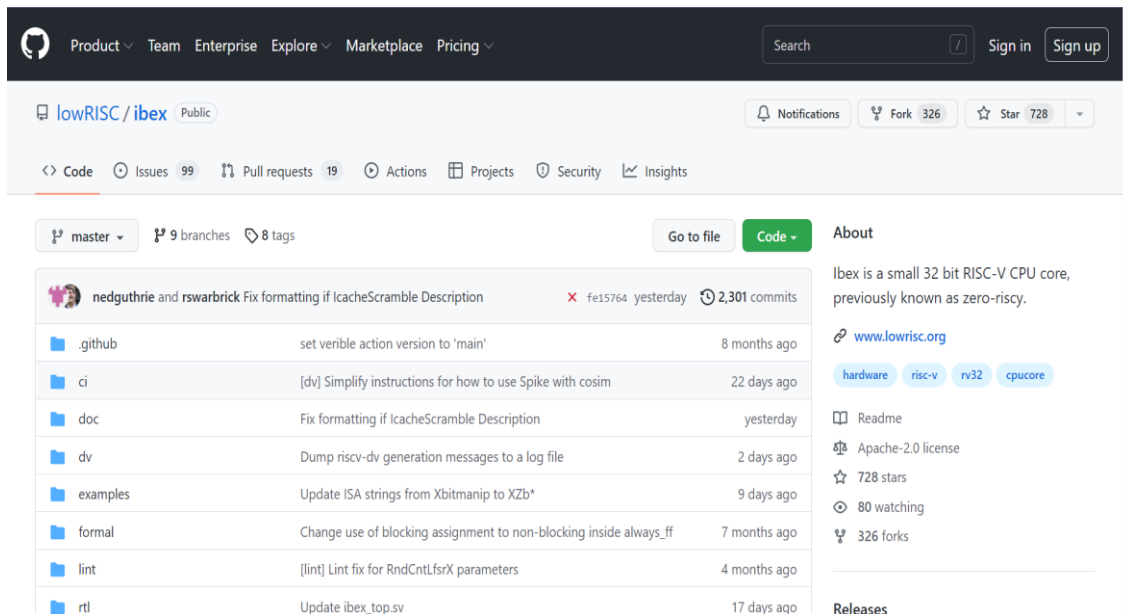


Figure 6.4: Github code of Ibex core

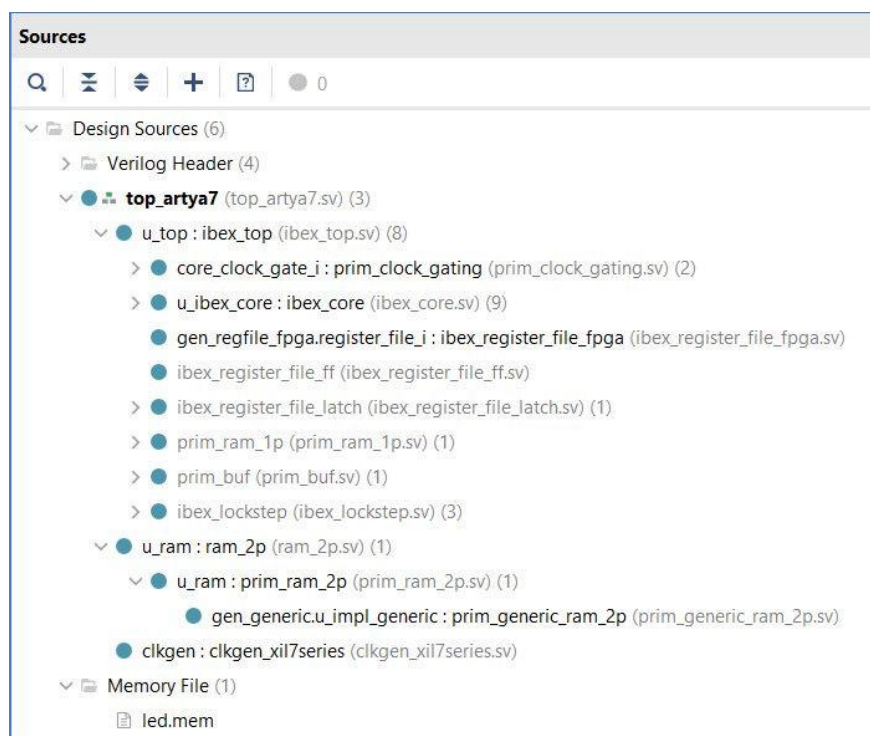


Figure 6.5: Design sources of the Ibex core

After the led.mem file is included to the project the simulation is performed and as seen in the Figure 6.8, the Ibex core [6] could read and write the instructions properly.

```

C led.c  ×
Users > gulcebaysal > Downloads > led > C led.c
1  // Copyright lowRISC contributors.
2  // Licensed under the Apache License, Version 2.0, see LICENSE for details.
3  // SPDX-License-Identifier: Apache-2.0
4
5  #include <stdint.h>
6  #define CLK_FIXED_FREQ_HZ (50ULL * 1000 * 1000)
7
8  /**
9   * Delay loop executing within 8 cycles on ibex
10  */
11  static void delay_loop_ibex(unsigned long loops) {
12      int out; /* only to notify compiler of modifications to |loops| */
13      asm volatile(
14          "1: nop          \n" // 1 cycle
15          "  nop          \n" // 1 cycle
16          "  nop          \n" // 1 cycle
17          "  nop          \n" // 1 cycle
18          "  addi %1, %1, -1 \n" // 1 cycle
19          "  bnez %1, 1b   \n" // 3 cycles
20          : "=&r" (out)
21          : "0" (loops)
22      );
23  }

```

Figure 6.6: The C code named led.c

```

25  static int usleep_ibex(unsigned long usec) {
26      unsigned long usec_cycles;
27      usec_cycles = CLK_FIXED_FREQ_HZ * usec / 1000 / 1000 / 8;
28
29      delay_loop_ibex(usec_cycles);
30      return 0;
31  }
32
33  static int usleep(unsigned long usec) {
34      return usleep_ibex(usec);
35  }
36
37  int main(int argc, char **argv) {
38      // The lowest four bits of the highest byte written to the memory region named
39      // "stack" are connected to the LEDs of the board.
40      volatile uint8_t *var = (volatile uint8_t *) 0x0000c010;
41      volatile uint8_t *add = (volatile uint8_t *) 0x0000c011;
42      int i;
43      i = 0;
44      *add = 0x01;
45      *var = 0x0a;
46
47      while (1) {
48
49          if(i<=15){
50              *var = *var + *add;
51              i = i+1;
52              usleep(10 * 10); // 1000 ms
53          }else
54              i = 0;
55      }
56  }
57  }
58
59  // usleep(1000 * 1000); // 1000 ms
60  // *var = ~(*var);

```

Figure 6.7: Continuation of the C code named led.c

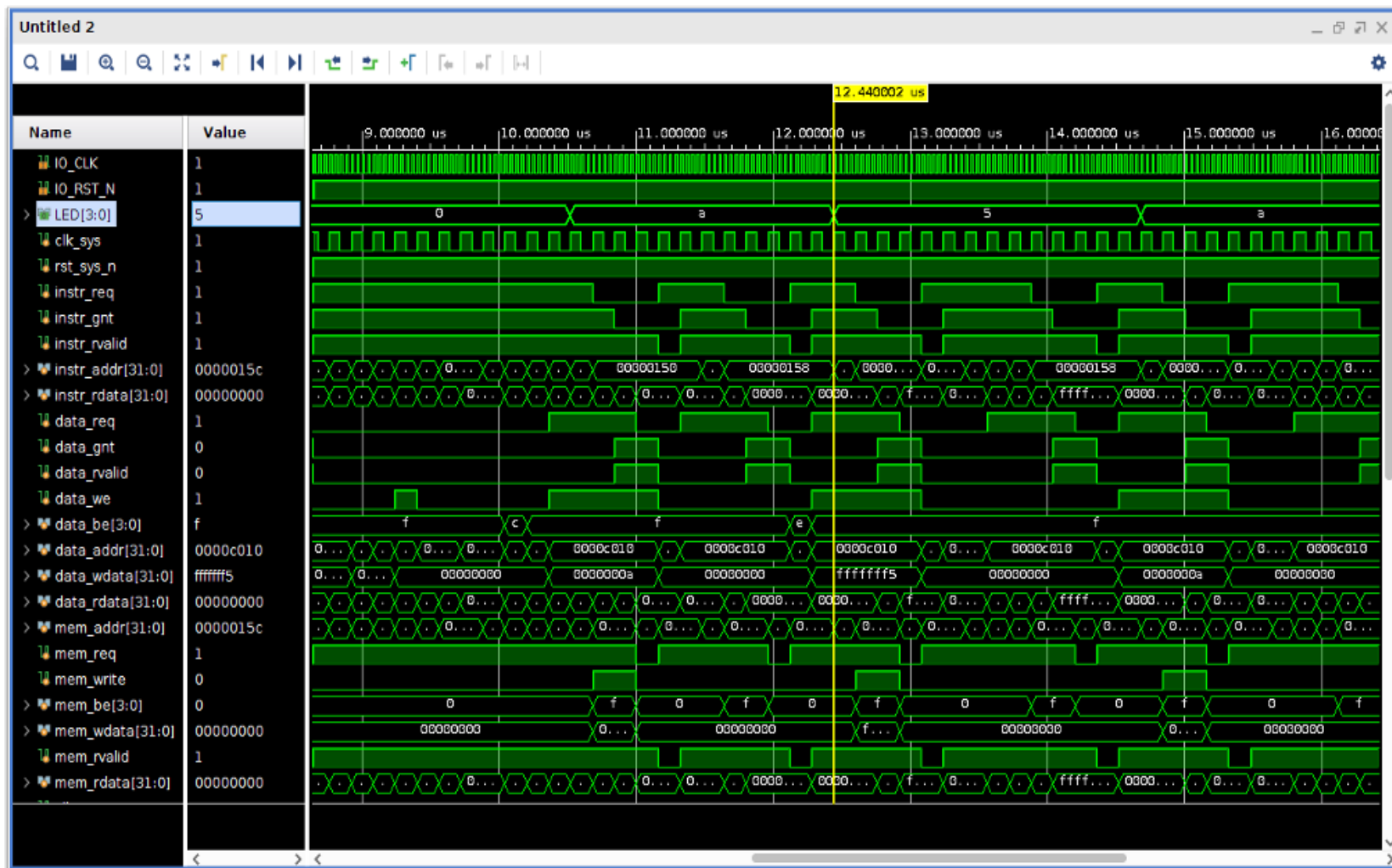


Figure 6.8: Simulation screenshot with led.mem file

Seeing that the Ibex Core [6] is working properly from the simulation, we decided to move to add a related peripheral in order to provide a communication. We fetched the Wishbone compatible source codes to the top module. After adding the Wishbone protocol, we decided to move to the implementation step that would be performed in a FPGA board. We did not need to change anything in the top module but simply add an .xdc file compatible with FPGA [15]. Then, we assigned inputs and outputs according to the project inputs and outputs. We generated bitstream and displayed it on the FPGA. The results were correct on the FPGA as well as in the simulation. A section from the video where FPGA lights up the LEDs are in the figure below.



Figure 6.9: FPGA results

Finally, in order not to repeat the problems we will experience while performing this implementation, we will prepare a documentation explaining all the steps that lead us to a running RISC-V [1] processor on Xilinx Vivado [42].

6.4 Connecting GPIO

A simple LED code was compiled and added to the project in order to ensure that the Ibex processor and the innovations made in the project work correctly at every step [6]. Necessary tests were done by observing the LEDs on the FPGA board. While

doing this test, the General-Purpose Input/Output (GPIO) [73] needed to control the LEDs. A GPIO port handles both incoming and outgoing digital signals. GPIO added to the first version of the project without the Wishbone interface connections yet, and it worked successfully. In the following steps, a GPIO connection was also needed for the Ibex with the Wishbone interface connected.

6.5 Connecting UART

UART, or Universal Asynchronous Receiver-Transmitter [72], is one of the most used device-to-device communication protocols. When properly configured, UART can communicate with a wide range of serial protocols. Based on the idea that boot loading can be done by accessing the terminal using the UART connection, it was decided to make a UART connection to the project. All Wishbone signal connections are made with the `wbuart` wrapper module.

```

29     wbuart #(
30         .INITIAL_SETUP           (INITIAL_SETUP),
31         .LGFLEN                  (LGFLEN),
32         .HARDWARE_FLOW_CONTROL_PRESENT (HARDWARE_FLOW_CONTROL_PRESENT)
33     )wbuart (
34
35         .i_clk                    (wb.clk),
36         .i_rst                    (wb.rst),
37         .i_wb_cyc                 (wb.cyc),
38         .i_wb_we                  (wb.we),
39         .i_wb_addr                (addr),
40         .i_wb_data                (wb.dat_i),
41         .i_wb_sel                 (wb.sel),
42         .i_wb_stb                 (wb.stb),
43         .o_wb_ack                 (wb.ack),
44         .o_wb_stall               (wb.stall),
45         .o_wb_data                (wb.dat_o),
46         // .i_uart_rx             (i_uart_rx),
47         .o_uart_tx                (o_uart_tx),
48         // .i_ots_n               (i_ots_n),
49         .o_rts_n                  (o_rts_n),
50         .o_uart_tx_int            (o_uart_tx_int),
51         .o_uart_rx_int            (o_uart_rx_int),
52         .o_uart_txfifo_int        (o_uart_txfifo_int),
53         .o_uart_rxfifo_int        (o_uart_rxfifo_int)
54     );
55
56     assign wb.err = 1'b0;
57
58 endmodule

```

Figure 6.10: UART connections

Afterwards, necessary changes were made on the `Ibex_soc` module. UART base address and size address information were added. While there were 4 masters and 2 slaves connected to Ibex [6] before, there were 4 masters and 3 slaves.


```

101     wb_interconnect_sharedbus
102     #(.numm      (4),
103       .nums      (3),
104       .base_addr ('{ram_base_addr, led_base_addr,uart_base_addr}),
105       .size      ('{ram_size, led_size,uart_size })) //,
106     wb_intercon
107     (.*);
108
109     wb_spramx32 # (ram_size) wb_spram(.wb(wbs[0]),.*);
110
111     wb_led      wb_led
112     (.wb(wbs[1]),
113      .*);
114
115     ////////////////////////////////////////////////////
116
117     wb_ov7670_erfan  wb_ov7670_erfan
118     (.wb(wbm[0]),
119      .*);
120
121     /*wb_sharpen_erfan  wb_sharpen_erfan
122     (.wb(wbm[1]),
123      .*); */
124
125     // wb_vga_erfan  wb_vga_erfan
126     //     (.wb(wbm[1]),
127     //     .*);
128
129     wb_wbuart_wrap wb_wbuart_wrap
130     (.wb(wbs[2]),
131      .*);

```

Figure 6.11: UART definitions on the Ibex_soc module

The resulting RTL schematic was as in Figure 6.12.

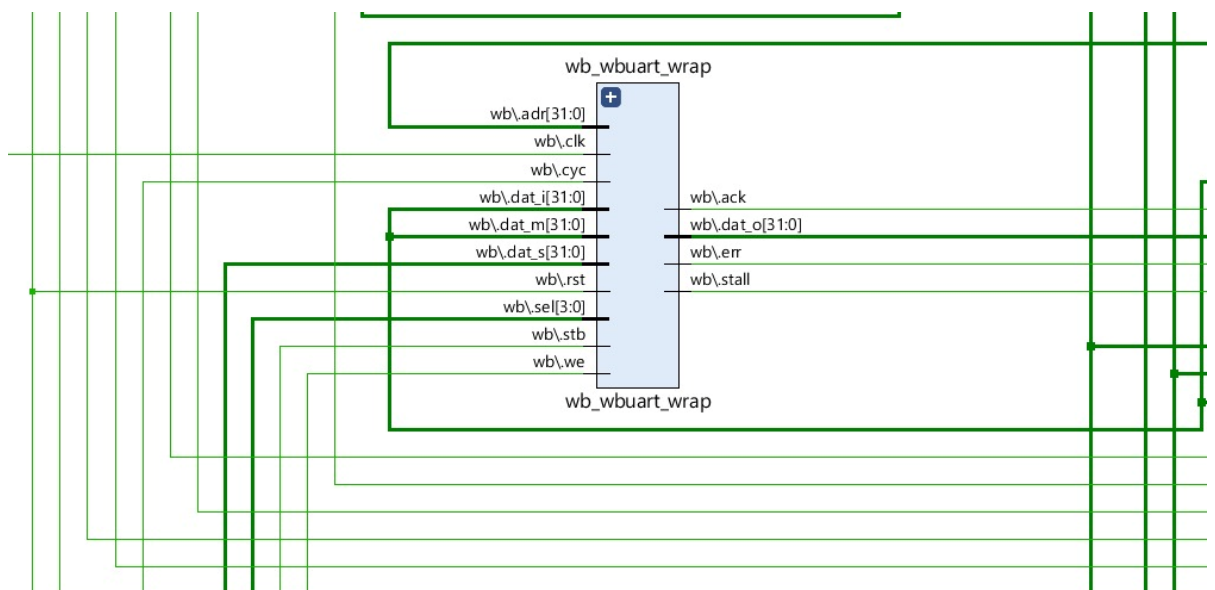


Figure 6.12: The resulting RTL schematic

7. APPLICATION OF IMAGE PROCESSING ALGORITHMS

In this part of the project, we will talk about how we implement the filtering applications that we mentioned in the image processing part. For edge detection [38], we concluded that we should put the data-image we have in the convolution process with the Laplacian filter. We discussed how the Laplacian filter [15] we have works and what kind of image we expect to get as a result. The convolution operations that this filter we have in order to perform edge detection consisted of multiplication operations, and we could add these operations to the ALU of our open-source processor with custom instructions. We aim to show that we have implemented a simple edge detection filter using this processor whose instruction set is extended according to our/the project's needs.

Secondly, we explain how we have obtained the image we need in order to apply image processing algorithms. The camera that we will obtain the image we aim to filter is the ov7670 [45] camera module. After acquiring this module, we aimed to project the image onto the screen with a Video Graphics Array (VGA) [5] connection. Simply in this process: the camera sends the data to the FPGA and the FPGA receives this data and sends it to the screen via VGA. The necessary filtering of the instruction set added in the processor takes place while the data is in the FPGA. As a result, we have seen that we can apply the edge detection filter to the image we have.

7.1 Laplacian Filter and Design of its Custom IP

In this section, it will be explained how all the blocks that make up the Laplacian filter that we will use for edge detection are designed and described. The data to be convoluted with the Laplacian filter consists of the image we will get from the camera. This image will be retrieved from RAM every cycle, and every pixel from the camera module will be convoluted. Although there is a possibility of encountering a problem such as insufficient memory as the size of the image and the number of data increases, the storage capacity of the RAM will be sufficient for the image from the camera module while designing the filter at this stage.

The communication between the processor and the designed custom IP will be provided by the Wishbone interface used in the project. To talk about how an entire communication network works [46] [15]:

- All of the procedure starts with the master (master_arbiter) demonstrating an address and the data on the bus for reading process. For IP to get enabled the processor must assert the allocated address to the Laplacian filter register which is mentioned as the master.
- To signify a read cycle, Custom IP cancels [WE_O] and asserts [CYC_O] and [STB_O] which begins the cycle.
- RAM asserts [ACK_I] after decoding inputs.
- Custom IP asserts [ACK_I] and delivers valid data on [DAT_I] regarding [STB_O] for specifying valid data.
- To mark the end of the data period, Custom IP cancels [STB_O].

The rest of the process is about the convolution. The Kernel constant is multiplied with the data received. The resultant data is kept in add_mul_register throughout the whole multiplication process. The calculations rest for 9 cycles since the product of all elements of the image and kernel occurs nine times. Finally, the final outcome is written back to the Random-Access Memory after the 9th cycle. The input image pixels which are stored in RAM is convoluted with the Kernel and the result is accrued with the result that is calculated and was stored in con_result register before [15].

As mentioned before, for communication cycle to begin the IP must be excited. This whole process is controlled by the Wishbone [7]. Wishbone interconnect is also used when the ultimate result stored in add_mul_register is sent back to the allocated address in RAM [7]. All of the procedures above are based on addresses which are created by the address generator. The block diagram of the designed Custom IP is in the Figure 7.1.

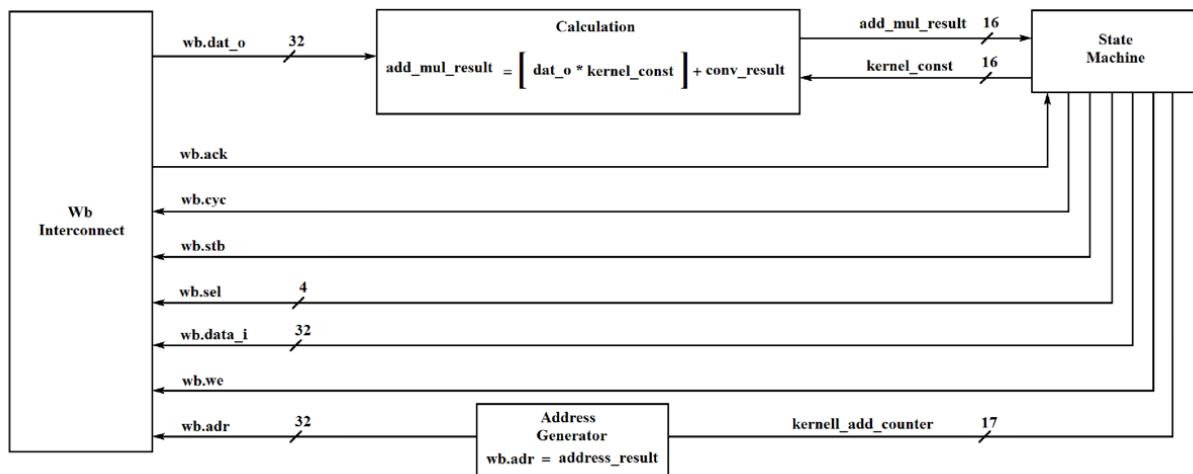


Figure 7.1: Laplacian filter [15]

7.2 Custom Instruction Added Module

As mentioned before, the open-source processor selection is very important for this project. We can customize the ALU of the processor in a way by preparing the instruction set that will perform all the convolution operations of the designed Laplacian filter faster and more effectively. RISC-V [1] gives us the freedom to customize and expand the instruction set by adding the command we want. Adding new commands is done through the RISC-V GNU Toolchain [44]. The two basic files that need to be modified when adding new commands to RISC-V are "riscv-opc.c" and "riscv-opc.h". Commands should be added with the "opcodes" [47] structure in these two files. Opcodes are microprocessor operation codes that carry out operations such as addition, multiplication, and division. A command must be written in a specific structure. The rule for the command we chose is as follows [48]: name, isa, operands, match, mask, match_func.

Respectively, the name of the instruction to be added, the instruction set model it is from, the registers to be used, and the structure of the instruction when these elements are added. Only two commands are used in our project and they are Custom 0 and Custom 1 command.

The new commands created are directly related to the ALU and the instruction decoder. The instruction decoder reads the next instruction from memory and transmits the individual components to the appropriate destinations [49]. The ALU takes two operands and executes the corresponding operation to them after the instruction decoder decodes the instruction that will be processed in the beginning. Because of this relationship, both the ALU and the instruction decoder must be arranged as they appear in Figure 7.2 and Figure 7.3 respectively.

```

9  module ibex_alu (
10     input  ibex_pkg::alu_op_e operator_i,
11     input  logic [31:0]      operand_a_i,
12     input  logic [31:0]      operand_b_i,
13
14     input  logic [32:0]      multdiv_operand_a_i,
15     input  logic [32:0]      multdiv_operand_b_i,
16
17     input  logic             multdiv_en_i,
18
19     output logic [31:0]      adder_result_o,
20     output logic [33:0]      adder_result_ext_o,
21
22     output logic [31:0]      result_o,
23     output logic             comparison_result_o,
24     output logic             is_equal_result_o
25 );

```

Figure 7.2: Operands defined in ALU

```

unique case ({instr[30:25], instr[14:12]})
// RV32I ALU operations
{6'b00_0000, 3'b000}: alu_operator_o = ALU_ADD; // Add
{6'b10_0000, 3'b000}: alu_operator_o = ALU_SUB; // Sub
{6'b00_0000, 3'b010}: alu_operator_o = ALU_SLT; // Set Lower Than
{6'b00_0000, 3'b011}: alu_operator_o = ALU_SLTU; // Set Lower Than Unsigned
{6'b00_0000, 3'b100}: alu_operator_o = ALU_XOR; // Xor
{6'b00_0000, 3'b110}: alu_operator_o = ALU_OR; // Or
{6'b00_0000, 3'b111}: alu_operator_o = ALU_AND; // And
{6'b00_0000, 3'b001}: alu_operator_o = ALU_SLL; // Shift Left Logical
{6'b00_0000, 3'b101}: alu_operator_o = ALU_SRL; // Shift Right Logical
{6'b10_0000, 3'b101}: alu_operator_o = ALU_SRA; // Shift Right Arithmetic

```

Figure 7.3: Arranges in instruction decoder

The ALU only works with 32-bit operands. With this in mind, it is necessary to adjust the changes we will make to the ALU and the instruction decoder. The image we aim to get from the camera is in RGB format and 16 bits. It will be

necessary to convert this image to grayscale by collecting all R, G, B channels. When we do this, we expect to have a 7-bit grayscale image. In order to be able to process every 9 pixels we have in two operands; it is necessary to define the layouts appropriately.

The custom0 should be called to define the designed Kernel, then the custom1 function should be called to perform the necessary convolution operations as in Figure 7.4 and Figure 7.5 respectively. As seen in Figure 7.5, each 7-bit element in the image matrix is multiplied by the kernel and the result is added. The Laplacian filter model for which the result is calculated has been explained in detail in the previous section.

```

ibex_custom_reg.sv
C:/custom_filter yavuz/custom_filter/project_20.srcs/sources_1/new/ibex_custom_reg.sv
🔍 📁 ⏪ ⏩ ✂ 📄 🗑 ✖ // 📊 💡
1
2  `default_nettype wire
3
4  import ibex_pkg::*;
5
6  module ibex_custom_reg (
7      input  logic          clk_i,
8      input  logic          rst_ni,
9      input  ibex_pkg::alu_op_e  cust_operator_i,
10     input  logic [31:0]  cust_operand_a_i ,
11     input  logic [31:0]  cust_operand_b_i ,
12
13     output logic [0:8] [31:0]  cust_kernel_val
14
15 );
16
17 logic [3:0] kernel_id ;
18
19 assign kernel_id = cust_operand_a_i[3:0];
20
21 always @( posedge clk_i)
22 begin
23     if (cust_operator_i == ALU_CUST0)
24         cust_kernel_val[kernel_id] = cust_operand_b_i;
25 end
26
27 endmodule

```

Figure 7.4: Custom 0 module

```

////////////////////////////////
// Custom 1 //
////////////////////////////////

logic [31:0] cust1_result;

assign cust1_result =
(operand_a_i[6:0] * load_kernel[0] ) +
(operand_a_i[13:7] * load_kernel[1] ) +
(operand_a_i[20:14] * load_kernel[2] ) +
(operand_a_i[27:21] * load_kernel[3] ) +

((operand_b_i[2:0],operand_a_i[31:28]) * load_kernel[4] )+

(operand_b_i[9:3] * load_kernel[5] ) +
(operand_b_i[16:10] * load_kernel[6] ) +
(operand_b_i[23:17] * load_kernel[7] ) +
(operand_b_i[30:24] * load_kernel[8] ) ;

```

Figure 7.5: Custom 1 module

7.3 Connecting Camera and VGA

After installing Ibex core and making sure it works correctly, there were two main elements needed to be able to observe the driver behavior, which is the main purpose of the project. The first of these was a camera to take images from the driver inside the vehicle. The second component is Video Graphics Array, or VGA [5], which will reflect the image taken with the camera to the image screen.

The OV7670 Camera Module [45] is a first in first out (FIFO) camera module that comes in a variety of pin configurations from various manufacturers. The OV7670 can output full frame, windowed 8-bit images in a variety of formats. This camera module features an image array that can operate at 30 frames per second and gives the user complete control over image quality. Serial Camera Manage Bus (SCCB), an I2C interface with a maximum clock frequency of 400KHz, is used to control the OV7670 image sensor [45]. The SCCB interface allows you to program all of the necessary image processing operations. Furthermore, OmniVision detectors employ patented sensor technology to increase quality of the image by decreasing or eliminating typical lighting/electrical sources of image corruption, like fixed pattern noise (FPN), blurring, fading, and so on, in order to generate a clean, totally consistent color image [15].

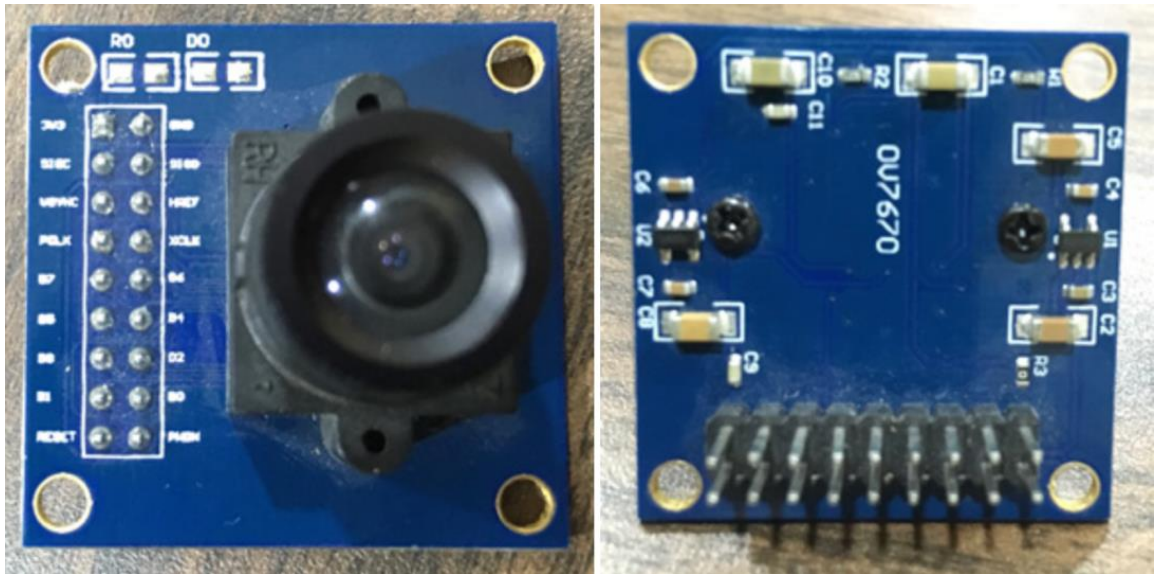


Figure 7.6: The OV7670 camera module [45]

	Active Array Size	640 x 480
Power Supply	Digital Core	1.8VDC \pm 10%
	Analog	2.45V to 3.0V
	I/O	1.7V to 3.0V ^a
Power Requirements	Active	60 mW typical (15fps VGA YUV format)
	Standby	< 20 μ A
Temperature Range	Operation	-30°C to 70°C
	Stable Image	0°C to 50°C
	Output Formats (8-bit)	<ul style="list-style-type: none"> • YUV/YCbCr 4:2:2 • RGB565/555/444 • GRB 4:2:2 • Raw RGB Data
	Lens Size	1/6"
	Chief Ray Angle	25°
	Maximum Image Transfer Rate	30 fps for VGA
	Sensitivity	1.3 V/(Lux • sec)
	S/N Ratio	46 dB
	Dynamic Range	52 dB
	Scan Mode	Progressive
	Electronics Exposure	Up to 510:1 (for selected fps)
	Pixel Size	3.6 μ m x 3.6 μ m
	Dark Current	12 mV/s at 60°C
	Well Capacity	17 K e
	Image Area	2.36 mm x 1.76 mm
	Package Dimensions	3785 μ m x 4235 μ m

Figure 7.7: Key specifications of OV7670 camera module [50]

The ov7670 module was determined as the camera to be used in this project. Various modules were added to the project, including the definitions of VGA and the camera to be used, and pixel definitions.

VGA stands for Video Graphics Array [5] and was created in 1987 by IBM [51] as a common screen standard. VGA color display panels have a resolution of 640×480 pixels, a frame rate of 60 Hz, and can display up to 16 colors at once. 256 colors are displayed when the resolution is reduced to 320×200 [5]. The camera is connected to the FPGA board and the data is transferred to the screen via VGA cable. VGA is important because it provides the transmission between the camera module in our hand and the image we project onto the screen.

In order to make the necessary Wishbone connections, `wb_ov7670_erfan` and `wb_vga_erfan` modules [15] were correctly added to the project as master as seen in Figure 7.8 [7]. All of the design source files in hierarchy can be found in Appendix A. In order to test that the system we created can receive and process the correct image, a C code to filter the image with custom instructions was compiled and added to the project. A part of the mentioned filter code can be seen below. The full version of the code can be found in the Appendix B.

- ▼ ● `ibex_soc (ibex_soc.sv) (7)`
 - `crg : crg (crg.sv)`
 - > ● `wb_ibex_core : wb_ibex_core (wb_ibex_core.sv) (3)`
 - `wb_intercon : wb_interconnect_sharedbus (wb_interconnect_sharedbus.sv)`
 - > ● `wb_spram : wb_spramx32 (wb_spramx32.sv) (1)`
 - `wb_led : wb_led (wb_led.sv)`
 - > ● `wb_ov7670_erfan : wb_ov7670_erfan (wb_ov7670_erfan.sv) (2)`
 - > ● `wb_vga_erfan : wb_vga_erfan (wb_vga_erfan.sv) (1)`

Figure 7.8: VGA and camera connections in the hierarchy

```

--
76 // Load Kernel Data - 1st
77 /*for (j = 0 ; j<3; j = j + 1)
78     for (i = 0 ; i<3; i = i + 1)
79         custom0( j*3+i , kernel[i][j] );*/
80 custom0( 0 , -1 );
81 custom0( 1 , 0 );
82 custom0( 2 , 1 );
83 custom0( 3 , -2 );
84 custom0( 4 , 0 );
85 custom0( 5 , 2 );
86 custom0( 6 , -1 );
87 custom0( 7 , 0 );
88 custom0( 8 , 1 );
89
90
91 pixel[0] = 0x00000001; // Activate camera module to initialize RAM with data
92 usleep(1000 * 15); // 15 ms
93 pixel[0] = 0x00000000;
94
95 pixel[0] = 0x00000100;
96
97 for (j = 0 ; j<240; j = j + 1)
98 {
99     for (i = 1 ; i<321; i = i + 1)
100     {
101         pixel_val = 0 ;
102         data1=(pixel[320*(j+1)+i+1]<<28)|(pixel[320*(j+1)+i]<<21)|(pixel[320*j+i+2]<<14)|(pixel[320*j+i+1]<<7)|(pixel[320*j+i]) ;
103         data2=(pixel[320*(j+2)+i+2]<<24)|(pixel[320*(j+2)+i+1]<<17)|(pixel[320*(j+2)+i]<<10)|(pixel[320*(j+1)+i+2]<<3)|(pixel[320*(j+1)+i+1]>>4) ;
104
105         pixel_val = custom1(data1,data2);
106
107         tester = pixel[i+j*320];
108         if (pixel_val>255) pixel_val = 255;
109         if (pixel_val<0) pixel_val = 0;
110         pixel[i+j*320] = ((pixel_val)<<16) | (tester&0xFFFF) ;
111

```

Figure 7.9: Filter code

A 32-bit memory space in stack region with 0xC010 beginning address is assigned for managing the IPs of the entire system as seen in Figure 7.10. Each IP is given an arbitrator activator, which will be handled by a C programming application.

```
57 volatile uint32_t *pixel = (volatile uint32_t *) 0xC010;
58
59 int main() {
60
61 volatile uint32_t *var = (volatile uint32_t *) 0x10000000;
62 volatile int32_t data1=0;
63 volatile int32_t data2=0;
64 volatile int32_t tester = 0;
65 volatile int32_t i=0,j=0,x=0,y=0,pixel_val=0;
66
```

Figure 7.10: The assigned address for controlling the IPs

As a result of the test, it was seen that the image was taken and processed as desired with the all-hardware equipment is supplied as well. In this way, an environment was created in which the image of the driver would be taken and processed.

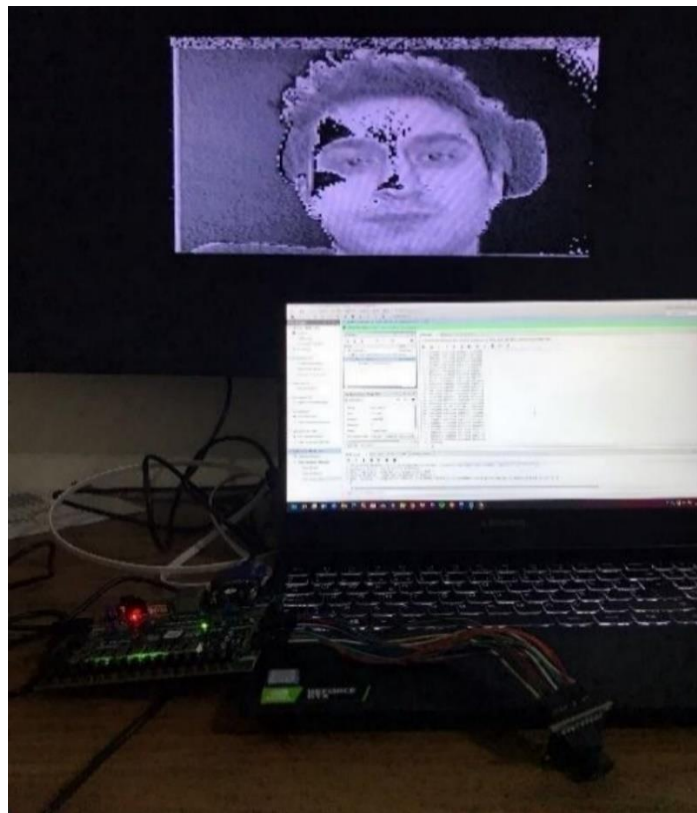


Figure 7.11: Test result

8. ACCESSING ONBOARD DDR RAM

In this part of our thesis, the steps we follow to access the Double Data Rate Random-Access Memory (DDR RAM) [53], which is hardware on the board we will use, are explained in order to eliminate the memory deficiency, which is the main problem we aim to solve in our project. Before the Wishbone communication protocol, we used at the beginning, our methods of accessing DDR RAM by making simple tests with Vivado's [42] own communication protocol and processor MicroBlaze [54], and then our process of repeating the same operations with the Wishbone protocol [7] were explained. The various methods we have tried to access DDR RAM and the shortcomings and advantages of these methods are explained.

The driver's image must be taken and the acquired image must be processed in order to create the system that detects driver fatigue, which is the project's major goal. A massive dataset is created by taking continuous images from the driver. Additionally, image data consumes a significant amount of storage space. Moreover, storing the artificial neural network code that will be added in order to identify the image and make it meaningful in future researches, requires a considerable amount of memory. Since the operating idea of artificial neural networks is to update the weights at each step, these vast and continually rising amounts of weights consume a significant quantity of memory. Synchronous Dynamic Random-Access Memory (SDRAM) [55] is insufficient to hold all of this data, instructions, and core code. As a consequence, the requirement to access and store data on an additional memory resource has emerged. This memory will be provided via onboard DDR RAM. The main reason why DDR RAM is preferred to store this data, which is mentioned as being too big to store on SDRAM, is the availability of DDR on the FPGA [56] board that will be used to set up the system. This prevents the need for an additional component, which would raise the cost and complicate the system.

8.1 What is DDR RAM?

Every processor in an electronic device needs memory to store data variables and addresses for subsequent processes. Data and addresses are kept in Random Access Memory (RAM) [41], and the address and data variables can be accessed from anywhere in the memory. As a result, the processor is able to access the data more

rapidly than the ROM [41]. RAM, the most well-known type of computer memory, is referred to as "random access memory" since any memory cell can be accessed directly if the row and column that intersect at that cell are known. Different varieties of RAM are available to meet the demands of technology advancements. The increased speed of processors necessitates exceptionally fast memory accesses. Double Data Rate Synchronous Dynamic Random-Access Memory, superior known as DDR SDRAM or DDR RAM for brief, is one of the forms of RAM developed to meet this purpose [53]. DDR RAM differs from ordinary RAM in that it can send data on both the rising and falling edges of each clock signal, whereas regular RAM can only send data on the rising edge of each clock signal. Thus, the quantity of data that can be delivered in the same length of time has been doubled, implying that it now works twice as quickly [53].

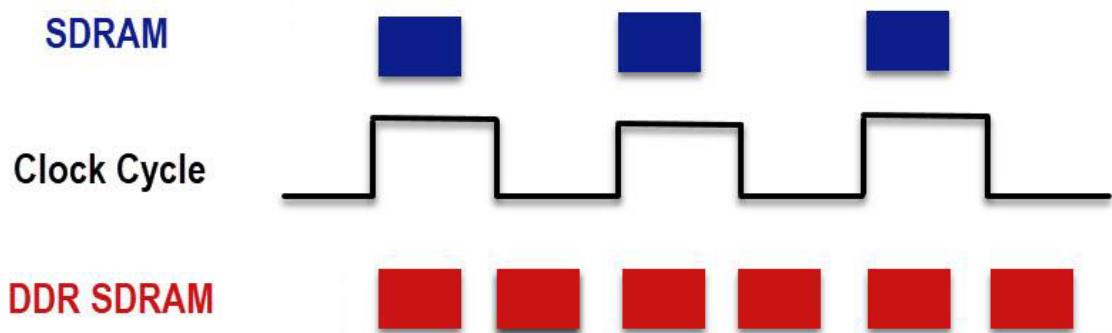


Figure 8.1: Comparison of DDR SDRAM and SDRAM [55]

DDR SDRAM sends data at 266 MHz instead of 133 MHz, to illustrate the difference between the two data transmission rates. Moreover, DDR is able to work with 16, 32 and 64 bits of data widths [53]. The SDRAM accesses may be readily controlled thanks to the DDR SDRAM memory controller, which takes the user's commands and executes them on the DDR. The data flow between the integrated processor and the DDR SDRAM is synchronized using the controller [41]. The memory controller conducts the operations such as READ, WRITE, and REFRESH and sends the data to memory [41].

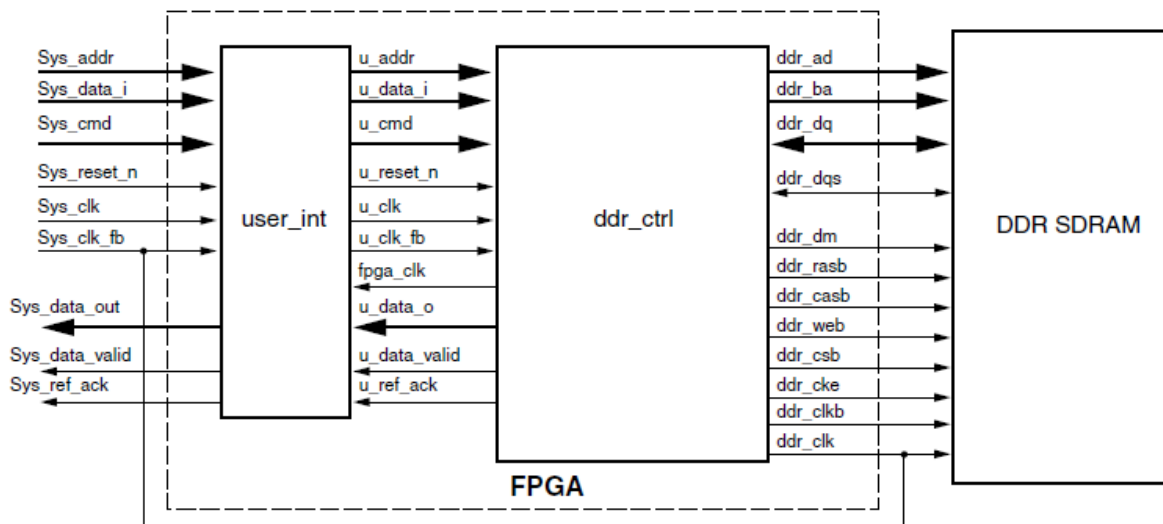


Figure 8.2: DDR controller block diagram [41]

DDR RAM is often preferred for processes such as image processing and signal processing, which generally need more capacity [57]. It is also an excellent choice when additional memory capacity is required for any hardware application due to its combination of low cost and speed. Several devices, including the most recent graphics processing cards, currently employ DDR RAM in various forms [57].

8.2 Bootloader

As a result of making the DDR [53] a module attached to the project, the onboard DDR can be physically accessible by providing the proper signal connections. As a consequence, the system is able to save data received by the microprocessor in DDR RAM. However, the instruction memory, or the core instructions that run the program, must be saved on external memory in the following phases. Since this is not a direct action, an alternative approach is necessary. The procedure that must be performed here is known as 'booting' or 'boot loading' [59]. When a new application needs to be imported into the rest of program memory, a bootloader is utilized as a distinct program in program memory. For load the application, the bootloader will use a serial port or some other methods. A bootloader will constantly run every time the computer is restarted whether a new software is to be loaded or if the application is to be run. A bootloader may include primitive operations that the program can use [59].

8.3 Accessing DDR RAM with MicroBlaze

After it was determined that an extra memory would be needed and DDR [53] was chosen as the component that would meet this demand, what had to be done was to provide access to DDR over the installed core. Due to a lack of illuminating sources and an inability to understand the working principle of accessing DDR signals via the Wishbone interface [7], this process was first accomplished using another processor. MicroBlaze [54] was chosen for this study since it is a processor that is widely utilized in a variety of applications, has a wealth of tutorials, and is simple to use. One of the reasons why MicroBlaze is preferred is that it is a 32-bit processor similar to the Ibex [6] core used in the project. By accessing DDR RAM with MicroBlaze, it was aimed to recognize the working mechanism of the system, its implementation steps and to test its work on the FPGA board.

8.3.1 MicroBlaze

There are two main types of microprocessors which can be used in Xilinx FPGAs with the Xilinx Embedded Development Kit (EDK) software tools [54]. Soft-core embedded microprocessors and hard-core embedded microprocessors are available. MicroBlaze is included in this classification as one of the soft-core embedded microprocessors. The MicroBlaze is a virtual microprocessor constructed by merging code units known as cores within a Xilinx FPGA [54]. The benefit of this method is that you only have as much microprocessor as you require. Additionally, the project can be customized according to particular requirements. MicroBlaze is a 32-bit microprocessor built with the Harvard RISC [58] architecture. It is customized for use in Xilinx FPGAs. MicroBlaze microprocessor has a parallel pipeline structure with three stages consisting of Fetch, Decode and Execute [54]. To summarize briefly, each stage takes one clock cycle to complete. Thus, when the given instruction is completed, three clock cycles have passed. Each stage is activated during each clock cycle, so three instructions can be transmitted at the same time from each pipeline stage. It runs the 32-bit instruction and data bus at full speed. Hence, the program runs and provides simultaneous access to both on-chip memory and externally supplied memory. Its general structure consists of 32 general purpose registers, a shift unit and two levels of

interrupt. This simple structure can be easily shaped according to the aim to be achieved in the established project and can be made useful. Thanks to this flexibility of use, the needed component area cost can be reduced while obtaining the required performance [54].

8.3.2 Steps to access DDR with MicroBlaze

MicroBlaze is used among the soft-core embedded microprocessor models available in the Design Tool of the already installed Xilinx Vivado [42]. In addition, the Software Development Kit (SDK) and Xilinx Vitis [60], which is used to perform the booting process, are among the tools used. One of the points to note here was that the SDK and Vivado should be the same version to avoid any errors that may arise. Following the steps shown in [61], the processes described below were carried out. First of all, a new project was opened on Vivado and a block design was created. MicroBlaze [54], AXI [62] GPIO [63], UartLite [64], AXI QUAD SPI [65], Memory Interface Generator (MIG) [66] components were added to the new block design.

As the first step, MicroBlaze IP was added to the design and configured as in the Figure 8.3.

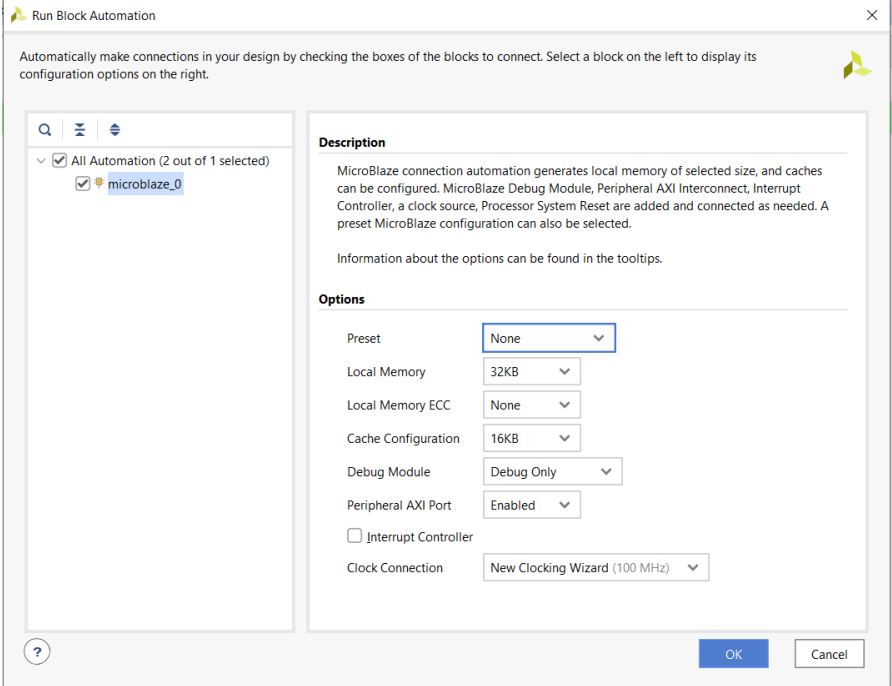


Figure 8.3: MicroBlaze IP configuration

In the next step, Clocking Wizard and the AXI QUAD SPI [65] blocks were added to the design and configured. Reset type was chosen as active low reset.

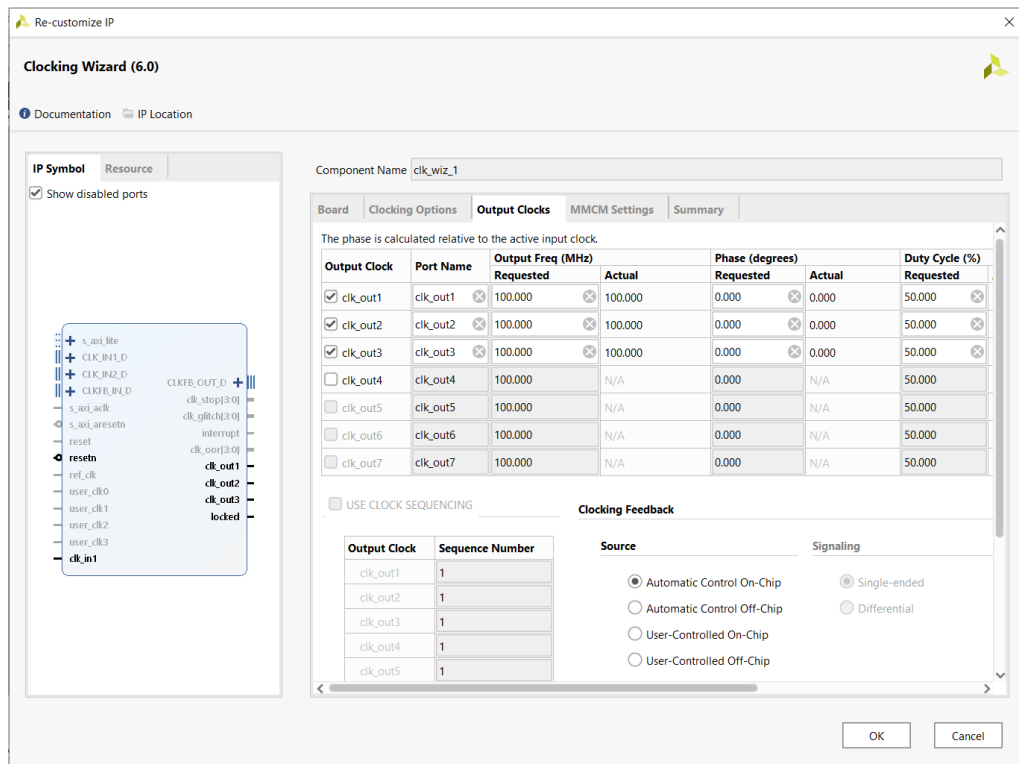


Figure 8.4: Clocking wizard configuration

Later, UartLite [64] and the Memory Interface Generator blocks were added to the design. Block automation was run for the MIG block. AXI GPIO [63] IP was added to the design and configured. After this process is done, connection automation was run for the AXI GPIO block. The final view of the completed block design was as in the Figure 8.5.

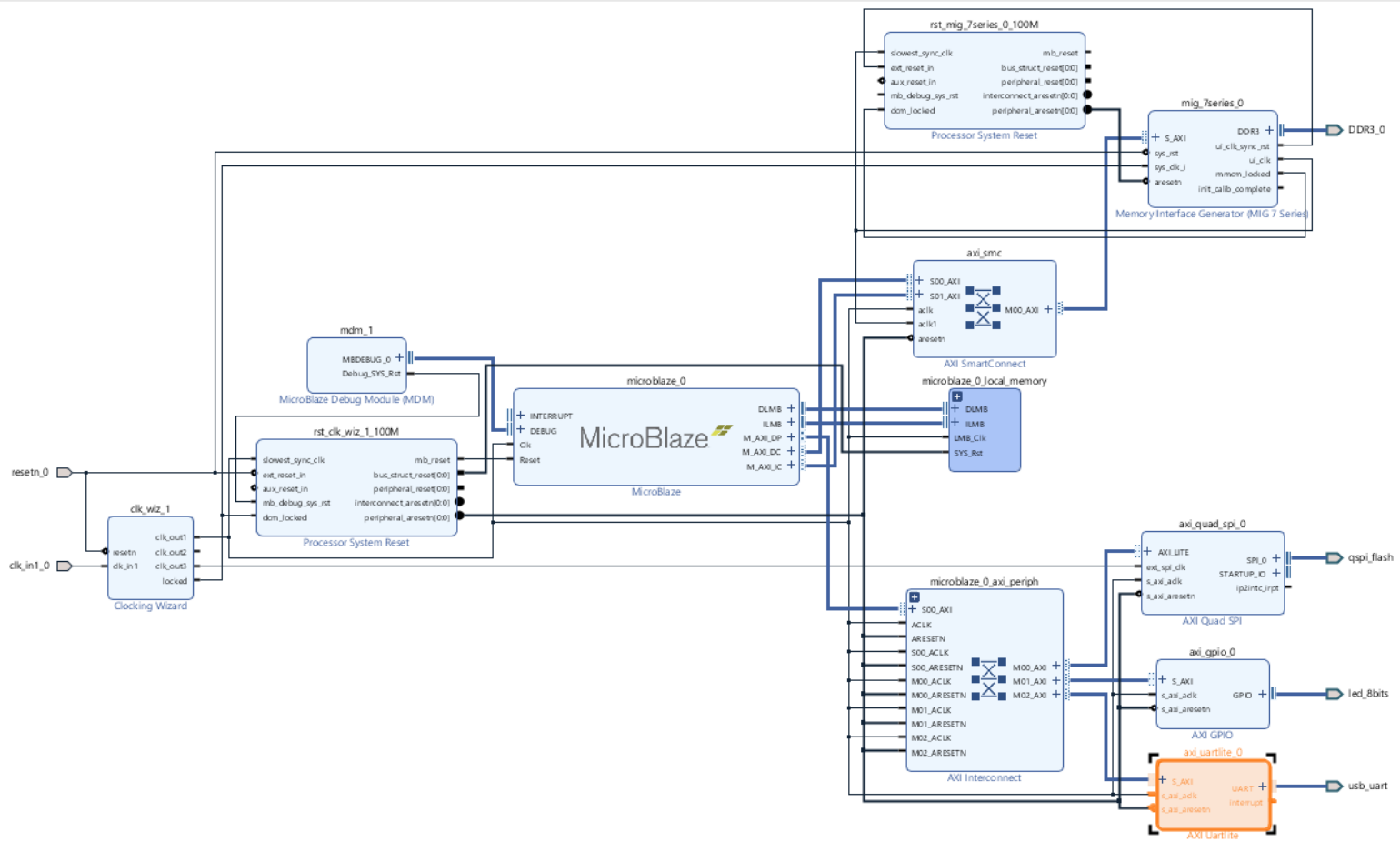


Figure 8.5: Completed block design

Cell	Slave Interface	Base Name	Offset Address	Range	High Address
microblaze_0					
Data (32 address bits : 4G)					
axi_gpio_0	S_AXI	Reg	0x4000_0000	64K	0x4000_FFFF
axi_quad_spi_0	AXI_LITE	Reg	0x44A0_0000	64K	0x44A0_FFFF
axi_uartlite_0	S_AXI	Reg	0x4060_0000	64K	0x4060_FFFF
microblaze_0_local_memory/dlmb_bram_if_cntlr	SLMB	Mem	0x0000_0000	32K	0x0000_7FFF
mig_7series_0	S_AXI	memaddr	0x8000_0000	512M	0x9FFF_FFFF
Instruction (32 address bits : 4G)					
microblaze_0_local_memory/ilmb_bram_if_cntlr	SLMB	Mem	0x0000_0000	32K	0x0000_7FFF
mig_7series_0	S_AXI	memaddr	0x8000_0000	512M	0x9FFF_FFFF

Figure 8.6: Address editor screen

After the synthesis and implementation steps by adding the appropriate constraint file, the bitstream file was created. Hardware was exported to SDK.

Thus, all connections were made and DDR RAM [53], an external memory, was added to the MicroBlaze [54] processor. As a result, physical access to DDR RAM, which is currently on the FPGA board, is provided. The next step was boot loading to access the external memory so that the data could be stored in DDR RAM. Firstly, a new Xilinx Vitis [60] project was created and configured the Board Support Package (BSP) settings to perform the bootloader. The offset value suitable for the FPGA board used is written to the blconfig.h file. Since the Nexys Video FPGA board [52] is used in this application, the appropriate offset value is 0x00C00000. Checked if the mapping of the bootloader is into MicroBlaze properly by looking at the linker script.

Linker Script: lscript.ld em.mss

Section Name	Memory Region
.text	microblaze_0_local_memory_ilmb_bram_if_cntlr_Mem_micro...
.init	microblaze_0_local_memory_ilmb_bram_if_cntlr_Mem_micro...
.fini	microblaze_0_local_memory_ilmb_bram_if_cntlr_Mem_micro...
.ctors	microblaze_0_local_memory_ilmb_bram_if_cntlr_Mem_micro...
.dtors	microblaze_0_local_memory_ilmb_bram_if_cntlr_Mem_micro...
.rodata	microblaze_0_local_memory_ilmb_bram_if_cntlr_Mem_micro...
.sdata2	microblaze_0_local_memory_ilmb_bram_if_cntlr_Mem_micro...
.sbss2	microblaze_0_local_memory_ilmb_bram_if_cntlr_Mem_micro...
.data	microblaze_0_local_memory_ilmb_bram_if_cntlr_Mem_micro...
.got	microblaze_0_local_memory_ilmb_bram_if_cntlr_Mem_micro...
.got1	microblaze_0_local_memory_ilmb_bram_if_cntlr_Mem_micro...
.got2	microblaze_0_local_memory_ilmb_bram_if_cntlr_Mem_micro...
.eh_frame	microblaze_0_local_memory_ilmb_bram_if_cntlr_Mem_micro...
.jcr	microblaze_0_local_memory_ilmb_bram_if_cntlr_Mem_micro...
.gcc_except_table	microblaze_0_local_memory_ilmb_bram_if_cntlr_Mem_micro...
.sdata	microblaze_0_local_memory_ilmb_bram_if_cntlr_Mem_micro...
.sbss	microblaze_0_local_memory_ilmb_bram_if_cntlr_Mem_micro...
.tdata	microblaze_0_local_memory_ilmb_bram_if_cntlr_Mem_micro...
.tbss	microblaze_0_local_memory_ilmb_bram_if_cntlr_Mem_micro...
.bss	microblaze_0_local_memory_ilmb_bram_if_cntlr_Mem_micro...
.heap	microblaze_0_local_memory_ilmb_bram_if_cntlr_Mem_micro...
.stack	microblaze_0_local_memory_ilmb_bram_if_cntlr_Mem_micro...

Figure 8.7: Linker script screen

Since the LEDs on the FPGA will be used to test the created system, a simple counter code was used to observe the LED outputs. The BSP settings of the simple LED counter code which is created using the SDK were configured. Checked if the mapping of the application file is into MIG memory region by looking at the new linker script.

Section Name	Memory Region
.text	mig_7series_0_memaddr
.init	mig_7series_0_memaddr
.fini	mig_7series_0_memaddr
.ctors	mig_7series_0_memaddr
.dtors	mig_7series_0_memaddr
.rodata	mig_7series_0_memaddr
.sdata2	mig_7series_0_memaddr
.sbss2	mig_7series_0_memaddr
.data	mig_7series_0_memaddr
.got	mig_7series_0_memaddr
.got1	mig_7series_0_memaddr
.got2	mig_7series_0_memaddr
.eh_frame	mig_7series_0_memaddr
.jcr	mig_7series_0_memaddr
.gcc_except_table	mig_7series_0_memaddr
.sdata	mig_7series_0_memaddr
.sbss	mig_7series_0_memaddr
.tdata	mig_7series_0_memaddr
.tbss	mig_7series_0_memaddr
.bss	mig_7series_0_memaddr
.heap	mig_7series_0_memaddr
.stack	mig_7series_0_memaddr

Figure 8.8: New linker script screen

Finally, .elf file was generated and the FPGA board was programmed. The program, which aims to provide access to DDR RAM [53] with the MicroBlaze processor [54] using the bootloader, worked as expected. Although this study with MicroBlaze did not directly contribute to the main purpose of the project, it helped to understand the principle of storing data on DDR by accessing external memory and bootloader principle.

8.4 Accessing DDR RAM via Wishbone Interface

The Wishbone [7] interface was established in the early stages of the project to interconnect components in order to provide simultaneous access to several components over the Ibex [6] core. It was necessary to connect the relevant Wishbone signals to DDR [53] signals to enable physical access to DDR SDRAM. However, because this is a procedure that cannot be completed directly, various ways have been tried for this purpose. Since DDR is often connected to the processor via the Memory Interface Generator (MIG) [66] interface, the MIG connection must be formed first using Wishbone [7]. Memory Interface is a free software tool used to generate memory controllers and interfaces for Xilinx FPGAs. Since MIG is utilized for Xilinx products, it can only be connected to other modules with Advanced Extensible Interface (AXI) [62]. As a result, it became required to create the proper environment for adding the MIG module by first constructing Wishbone-AXI connections, then adding the MIG and connecting the DDR RAM. While working with MicroBlaze, DDR was easily accessible due to the large number of explanatory resources available and MicroBlaze's availability among Xilinx products as a block that can be directly connected to the AXI, however this was difficult to do when working with Ibex due to lack of resources. Numerous alternative approaches were considered in order to complete this crucial stage, which will supply the extra memory required to perform the project's main goal.

8.4.1 Vector extension approach

Vector extension of RISC-V is clearly for machine learning and the instruction used in this extension can really make the run time of the code smaller. The main reason to use this technique is for its fetch part. Imagine you want to add 2 vectors with 64

elements each 8 bit. In Ibex [6] you go one by one to multiply each bit but with vector extension you can fetch all the vectors once and do the computation. So, at the end there will not be any need to add MAC instruction like we added to Ibex since we added V extension to the core.

To apply vector extension to Ibex we need a vector coprocessor. In the GitHub source that is Ibex with vector extension also includes a coprocessor called Vicuna [67]. The block diagram of this vector coprocessor is seen in Figure 8.9.

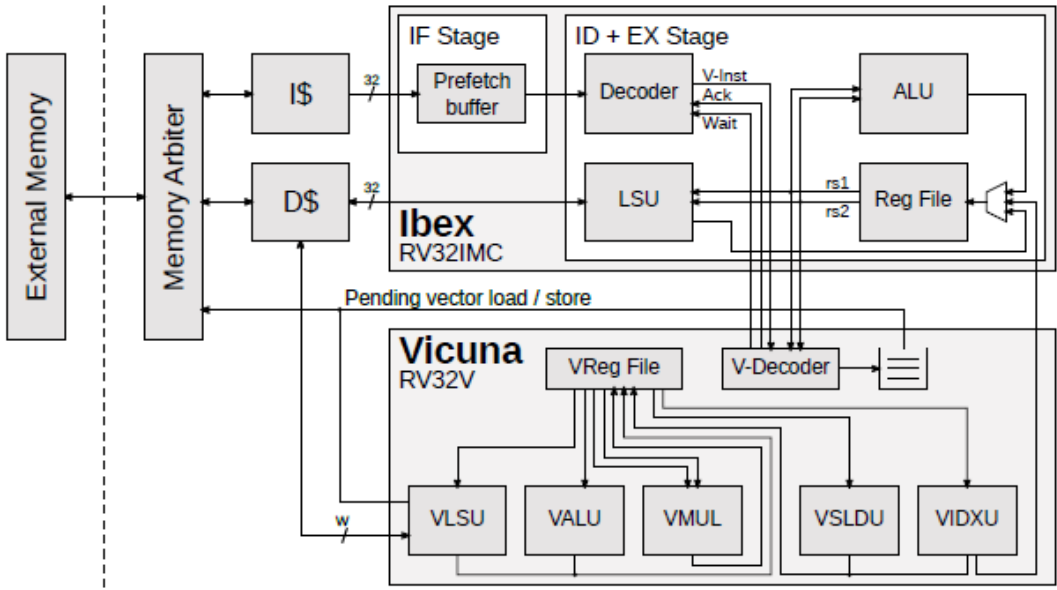
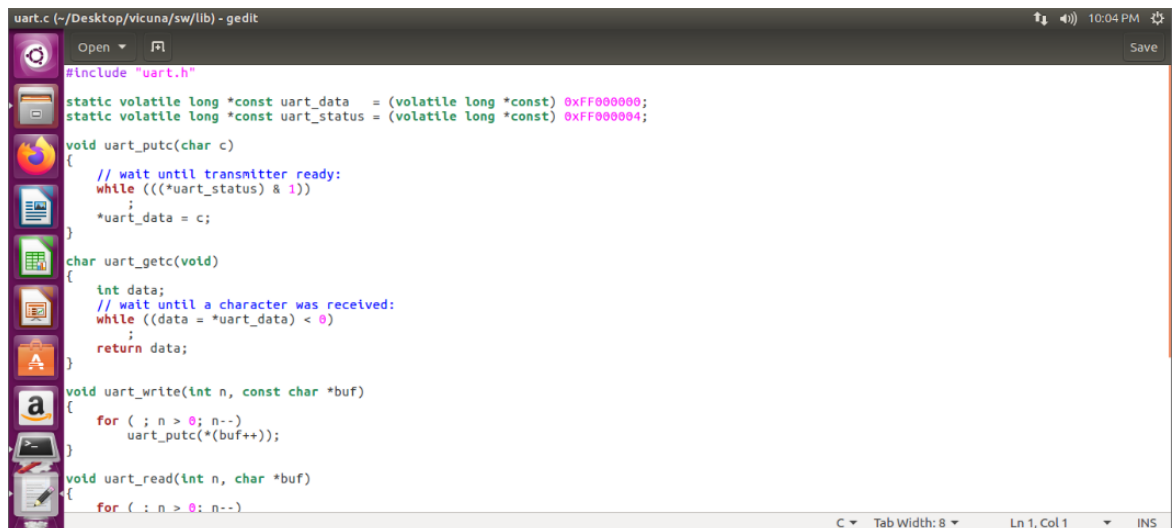


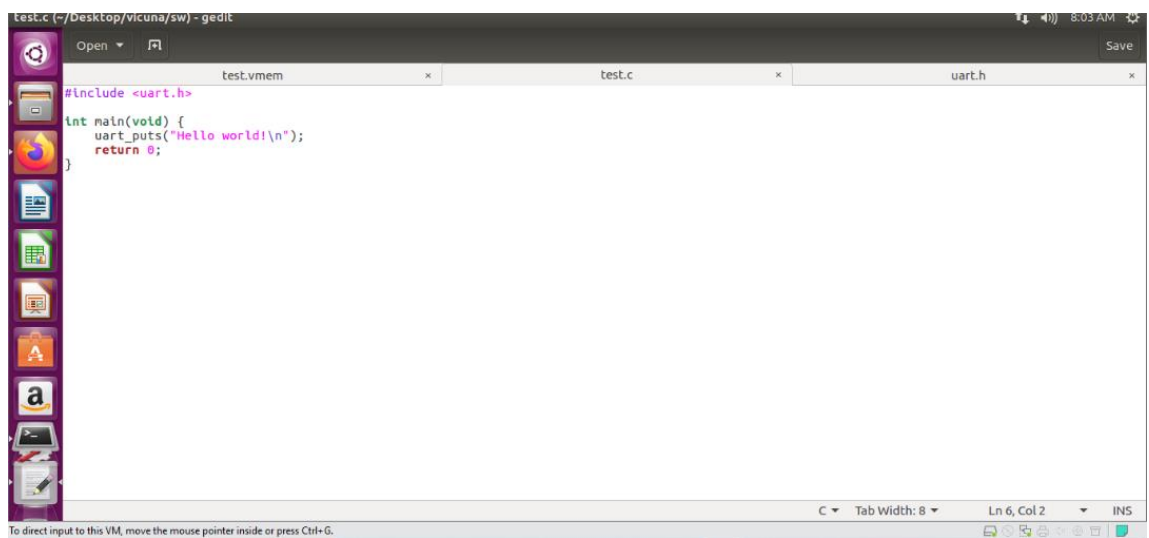
Figure 8.9: Vector coprocessor block diagram [68]

To test and see the output on the screen if the Vicuna implementation works, the UART [71] is used. The UART code and the test code is seen in Figures 8.10 and 8.11 respectively.



```
uart.c (~/Desktop/vicuna/sw/lib) - gedit
Open Save
#include "uart.h"
static volatile long *const uart_data = (volatile long *const) 0xFF000000;
static volatile long *const uart_status = (volatile long *const) 0xFF000004;
void uart_putc(char c)
{
    // wait until transmitter ready:
    while ((*uart_status & 1))
    ;
    *uart_data = c;
}
char uart_getc(void)
{
    int data;
    // wait until a character was received:
    while ((data = *uart_data) < 0)
    ;
    return data;
}
void uart_write(int n, const char *buf)
{
    for (; n > 0; n--)
        uart_putc(*(buf++));
}
void uart_read(int n, char *buf)
{
    for (; n > 0; n--)
```

Figure 8.10: The UART code



```
test.c (~/Desktop/vicuna/sw) - gedit
Open Save
test.vmem x test.c x uart.h x
#include <uart.h>
int main(void) {
    uart_puts("Hello world!\n");
    return 0;
}
C Tab Width: 8 Ln 6, Col 2 INS
To direct input to this VM, move the mouse pointer inside or press Ctrl+G.
```

Figure 8.11: The test code

As seen in Figure 8.12, from the terminal the output “Hello World” was observed.



Figure 8.12: FPGA implementation output

Finally, with the help of the source it is succeeded to boatload the BRAM of the vector extended Ibex but cannot reach DDR unfortunately.

8.4.2 Wishbone to AXI bridge

As a result of the research performed to find solutions to solve the problem of accessing DDR RAM [53], a solution proposal was found that will provide the connection between Wishbone [7] and the AXI [62] interface. It was explained in the previous sections that a direct connection cannot be established with the Wishbone interface to transfer the data held in the processor memory to DDR SDRAM, and this process requires a MIG [66] module. Since the AXI interface is also required to connect with the MIG, the first step would be to combine the Wishbone signals with the corresponding AXI signals. As the Wishbone interface built on Ibex contains large and complex modules, attempts to do this directly did not yield any results. As a consequence of the problem-solving research, it was discovered that a structure known as a 'bridge', which connects Wishbone to AXI and AXI to Wishbone, was appropriate for this operation. Because re-establishing this structure would take a significant amount of effort and knowledge, an open source bridge was employed instead. It is tried to establish this connection using the module named “wb2axi” from GitHub [69]. As mentioned in this resource, this bridge was using Wishbone in its pipelined mode and was fine with our project so far, as well [7].

The basic concept underlying the idea of accessing DDR using the bridge was to achieve the result by connecting the required modules respectively. The major purpose was expected to be accomplished by connecting Wishbone-AXI, AXI-MiG, and MiG-DDR, sequentially. This concept can be better understood with the help of the block diagram in Figure 8.13.

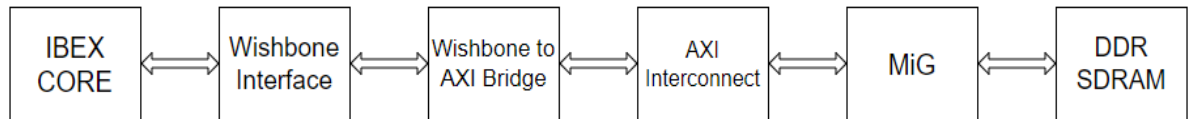


Figure 8.13: Planned block diagram

Since the interface used in our project was Wishbone [7], it was not possible for us to add a custom IP by choosing from Vivado's IP Catalog and to see all the modules that are created until now as a block design. It is thought that making the correct signal connections one by one by directly adding the selected bridge modules to the project will increase the possibility of making mistakes. As a result, in order to test the accuracy of the connections and achieve a more meaningful design image, design was packaged piece by piece and combined it into a new project. For this purpose, the project used so far in Vivado was packaged with the 'Create and Package New IP' option under the 'Tools' tab and obtained as a single block as seen in Figure 8.14. Likewise, the Wishbone-AXI bridge code [69] from Github was installed in a newly opened project in Vivado and packaged into a block. Later, a new project was opened on Vivado and a new block design was created. These blocks must be selectable over the IP Catalog in order to add the packaged projects to this design. For this reason, each one was added as a separate IP Repository as seen in Figure 8.16, allowing it to be added to the project through the IP catalog.

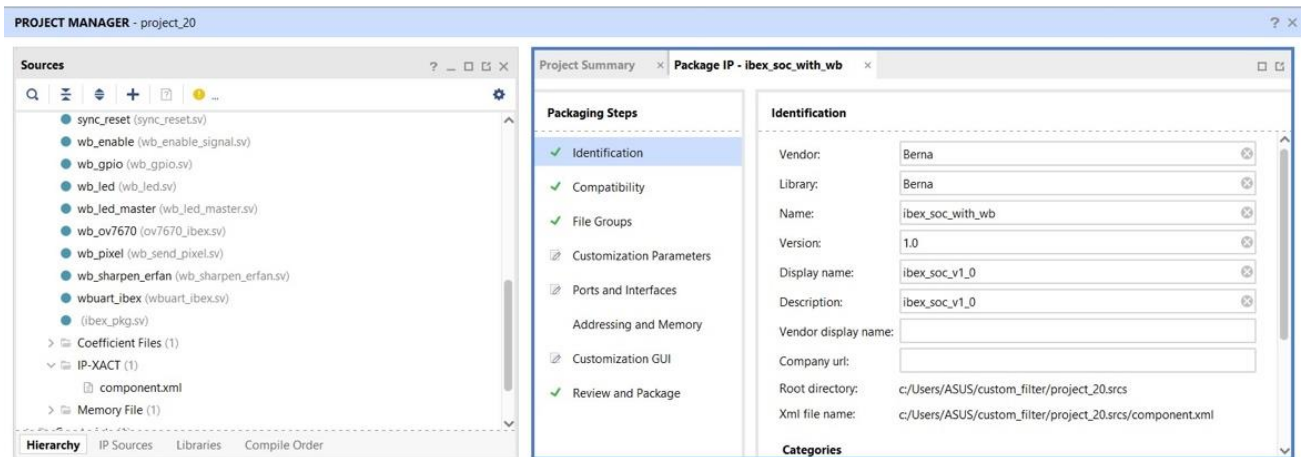


Figure 8.14: Package IP of Ibex with Wishbone project

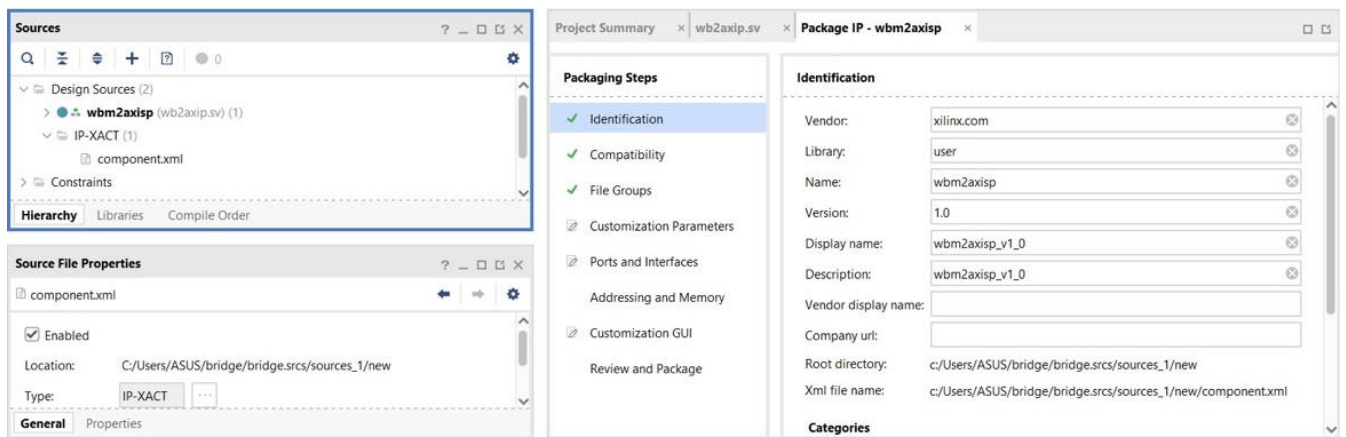


Figure 8.15: Package IP of Wishbone-AXI bridge

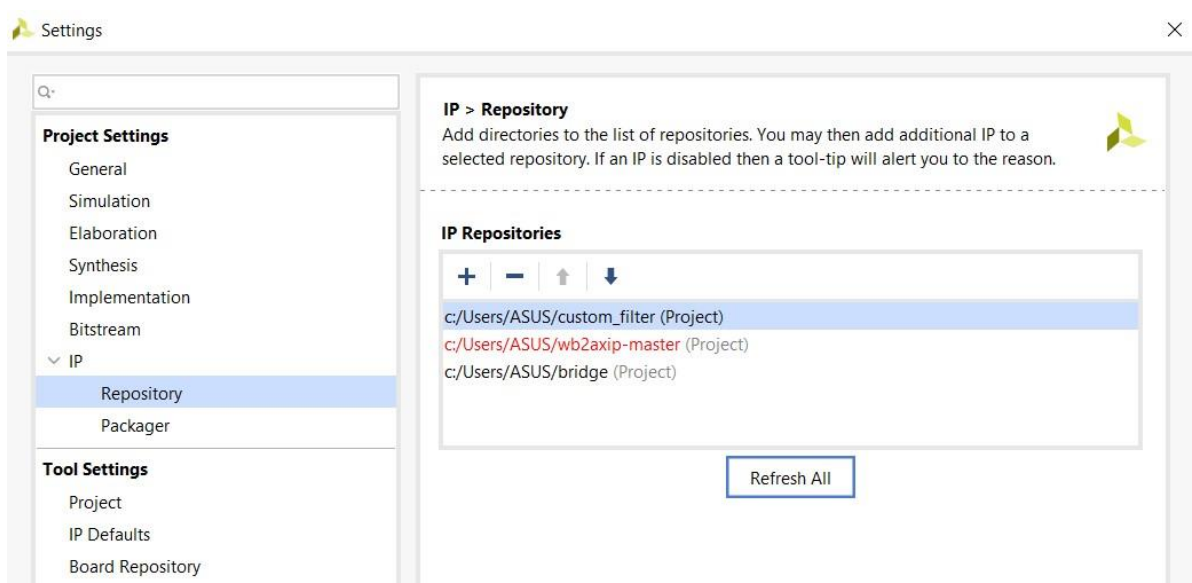


Figure 8.16: Adding repositories to the IP catalog

After the blocks obtained by packaging were added to the block design, the AXI Interconnect [62] block selected from the IP catalog was added. Before adding the MIG [66] block, it was decided to test whether the system works as desired with the LEDs on the FPGA by obtaining LED output over GPIO [72]. Additionally, in order to obtain LED data as output, the connections of the LED module defined as wbs[1] on the original project as slave were matched with the new output signals defined on the top module. These connections can be seen in Figure 8.17. The project was repackaged with the changed top module and the new package was added to the block design, thus the LED outputs were obtained directly.

```
assign clk_b = clk;
assign rst_b = rst_n;
assign ack_b = wbs[1].ack;
assign adr_b = wbs[1].adr;
assign cyc_b = wbs[1].cyc;
assign stall_b = wbs[1].stall;
assign stb_b = wbs[1].stb;
assign we_b = wbs[1].we;
assign sel_b = wbs[1].sel;
assign err_b = wbs[1].err;
assign dat_i_b = wbs[1].dat_m;
assign dat_o_b = wbs[1].dat_s;
```

Figure 8.17: New LED output signals

The AXI GPIO [63] block was also added to the design and the necessary connections were made one by one. The final block design was as in Figure 8.18.

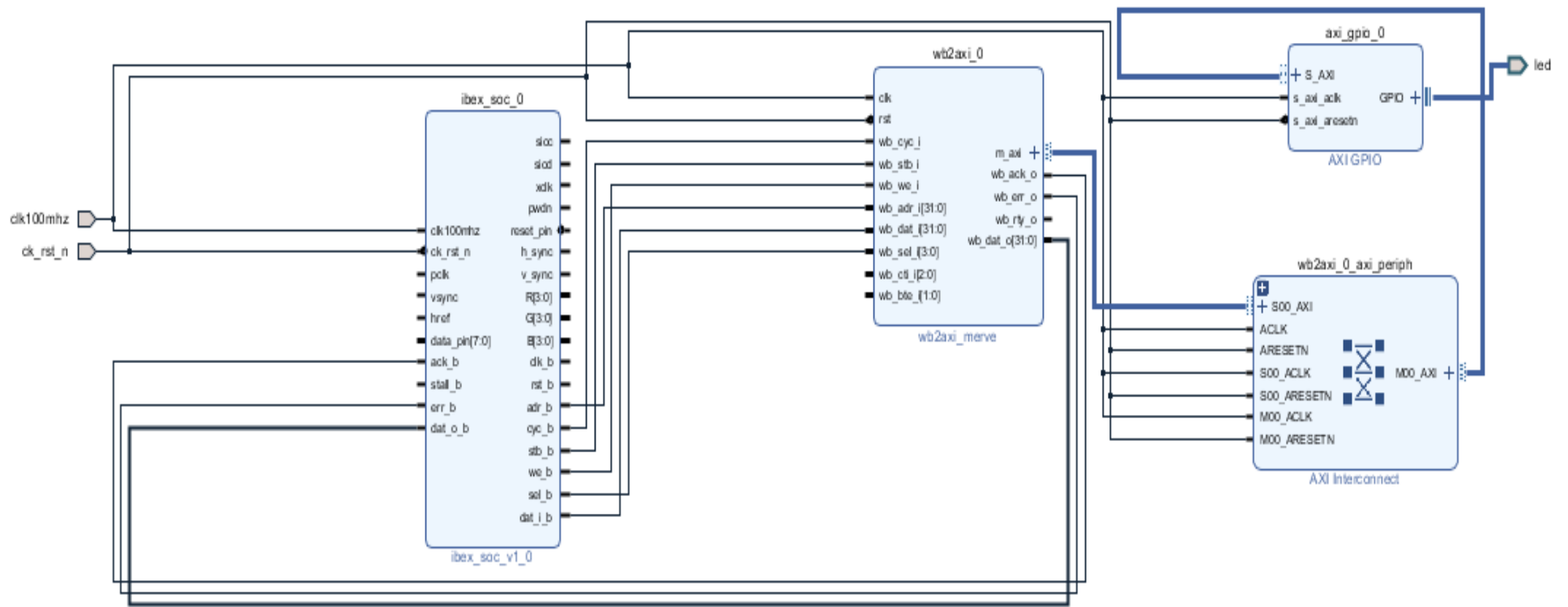


Figure 8.18: Final block design

Afterwards, the wrapper file was automatically created by right-clicking on the block design and selecting 'Create HDL Wrapper' option. Lastly, the recompiled C code that is used to test the GPIOs in the first version of Ibex project with Wishbone and it is expected to obtain the same results in the AXI GPIO [63] output this time. The C code written for the LEDs is in Figure 8.19.

```

Users > gulcebaysal > Downloads > led > C led.c
1  // Copyright lowRISC contributors.
2  // Licensed under the Apache License, Version 2.0, see LICENSE for details.
3  // SPDX-License-Identifier: Apache-2.0
4
5  #include <stdint.h>
6  #define CLK_FIXED_FREQ_HZ (50ULL * 1000 * 1000)
7
8  /**
9   * Delay loop executing within 8 cycles on ibex
10  */
11  static void delay_loop_ibex(unsigned long loops) {
12      int out; /* only to notify compiler of modifications to |loops| */
13      asm volatile(
14          "1: nop          \n" // 1 cycle
15          "  nop          \n" // 1 cycle
16          "  nop          \n" // 1 cycle
17          "  nop          \n" // 1 cycle
18          "  addi %1, %1, -1 \n" // 1 cycle
19          "  bnez %1, 1b   \n" // 3 cycles
20          : "=&r" (out)
21          : "0" (loops)
22      );
23  }
24
25  static int usleep_ibex(unsigned long usec) {
26      unsigned long usec_cycles;
27      usec_cycles = CLK_FIXED_FREQ_HZ * usec / 1000 / 1000 / 8;
28
29      delay_loop_ibex(usec_cycles);
30      return 0;
31  }
32
33  static int usleep(unsigned long usec) {
34      return usleep_ibex(usec);
35  }

```

Figure 8.19: The C code for the LEDs

After performing the synthesis and implementation steps on Vivado, a .bit file was created and the FPGA board was programmed. Unfortunately, the desired result was not observed on the FPGA board. In this application made to test the system, if the project created with the bridge could be observed to work correctly, the next step would be to provide access to DDR by adding MIG IP instead of AXI GPIO connection. The block diagram planned for this stage is as in Figure 8.20. Unfortunately, this target could not be achieved due to an incomprehensible error in the generated design.

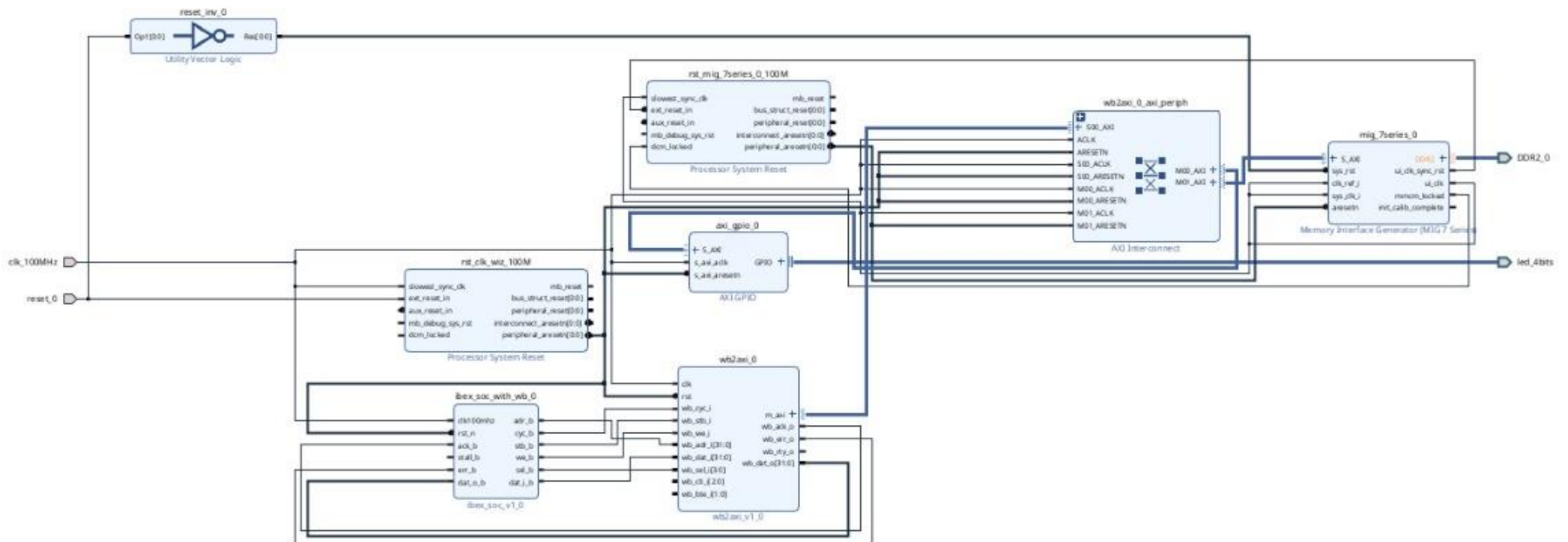


Figure 8.20: Block design with MIG

9. CONCLUSION

The RISC-V [1] open-source core implementation and custom instruction are discussed in this paper. One of the most significant transforms for edge detection Laplacian Filter has been reviewed. To solve one of the biggest problems in the project, the memory problem, we aimed to access DDR RAM and worked on it. We tested the extended instructions and included to our project in order to increase performance.

9.1 Results

In the final version of the project, necessary connections such as camera, VGA [51], GPIO, UART between Ibex [6], the processor we chose, and the communication protocol Wishbone [7] have been made and tested to work. Custom IP, designed for the filtering process where image processing algorithms are applied, was transferred to our system and it was seen that this filtering process was successful with the image taken from the camera. During this test, the extended instruction set of the RISC-V processor and the operations performed by the Custom IP were expressed in the ALU of the processor, thus bringing speed and efficiency to the system. In order to solve the memory problem, which is the main problem of our project, various connections were made with the bridge [69] method in between AXI-Wishbone. We are currently working on proving that this connection is successfully established and that DDR can be accessed with this method, together with a simulation.

9.2 Progress of the Project

We reached all the results we got throughout the project step by step. We aimed to use our time efficiently by proceeding in a systematic and programmed manner. At the beginning of the project, we examined the applications and implementations of driver fatigue detection systems with a detailed literature review. We aimed to design a

system in which we can use the artificial neural network code that can detect driver fatigue in the most effective way with the image processing system we have. The biggest problem we encountered in this process was the memory problem. The size of our dataset, which we will apply the image processing algorithms to, consists of the photos taken from the in-car camera, was not suitable for keeping in Block RAM. For this reason, we decided to use the DDR external memory, which is already on the board, Nexys 4 DDR FPGA [43], we chose. Since the communication protocol and open-source processor we use did not ensure that it would be suitable for us to access DDR, we sought different methods. In this process, as we explained in our thesis, we turned to methods such as accessing DDR without MIG and dealing with the Vector Extension project. Since we couldn't get the efficiency, we wanted from any of them, we made research about the bridge structure, which is a new method that we have not tried before. We predicted that this structure would give us the efficiency we wanted in terms of accessing MIG and therefore DDR, thanks to its ability to interconnect communication protocols. We focused on making all connections and addresses. We argue that this method is the solution to the memory problem that wastes us time, and we continue our attempts to make simulation studies on this structure.

9.3 Cost Analysis

Within the scope of the whole project, the time we allocate for the progress of the project and the achievement of certain results consists of two semesters, the fall and spring semesters. The FPGA card we use throughout the process we are working on our project is Nexys 4 DDR [43]. Access to this card, which is the first product that we can show as an expense within the scope of our project, has been quite easy since we already had the card. Xilinx Vivado [42], the implementation and design environment we use, gives a free right to anyone who wants to use it, so there was no cost in this part. The camera ov7670 [50], which we acquired in order to create a simulation of the driver detection system, was purchased to facilitate our work. In addition, the screen on which the image will be projected and the VGA [51] cable, which plays a role in the image transmission between the FPGA-camera-screen, were accessed from the GSTL Lab [70] in Istanbul Technical University.

9.4 Future Work and Recommendations

The first future work that can be done is extending the instruction set of the processor used for the implementation of detection filter even more in order to increase filter's performance. Secondly, for the accessing DDR RAM [53], 'bridge' structure can be further investigated and implemented. To connect two interfaces and use the benefits DDR RAM offers 'bridge' structure seems to be the best solution.

The most useful recommendation we can give is to concentrate on addressing and fixing missing connections while working on the bridge structure. When the correct structure is established, a system whose accuracy can be tested with a simple simulation code will be obtained and access to DDR RAM, which is the problem of the project, will be provided. Finally, the system should be developed and made suitable for in-vehicle use.

REFERENCES

- [1] “V,” RISC, 24-Jan-2022. [Online]. Available: <https://riscv.org/>. (accessed: Jun 4, 2022).
- [2] “ISO/IEC 9899:2018,” ISO, 13-Oct-2020. [Online]. Available: <https://www.iso.org/standard/74528.html>. (accessed Jun 4, 2022).
- [3] **G. Sikander and S. Anwar**, “Driver Fatigue Detection Systems: A Review,” *IEEE Transactions on Intelligent Transportation Systems*, vol. 20, no. 6, pp. 2339–2352, 2019.
- [4] **G. A. Thorpe**, “The design of a memory controller for DDR SDRAM,” 2021.
- [5] “Video graphics array,” Wikipedia, 01-Jun-2022. [Online]. Available: https://en.wikipedia.org/wiki/Video_Graphics_Array. (accessed: June 4, 2022).
- [6] lowRISC, “LowRISC/Ibex: Ibex is a small 32 bit RISC-v CPU core, previously known as Zero-riscy.,” GitHub. [Online]. Available: <https://github.com/lowRISC/Ibex>. (accessed Jun 1, 2022).
- [7] **M. Sharma**, “Wishbone bus architecture - A survey and comparison,” *International Journal of VLSI Design & Communication Systems*, vol. 3, no. 2, pp. 107–124, 2012.
- [8] **W. Zheng, Q. Q. Zhang, Z. H. Ni, Z. G. Ye, Y. M. Hu, and Z. J. Zhu**, “Distracted driving behavior detection and identification based on improved Cornernet-Saccade,” 2020 IEEE Intl Conf on Parallel & Distributed Processing with Applications, Big Data & Cloud Computing, Sustainable Computing & Communications, Social Computing & Networking (ISPA/BDCLOUD/SocialCom/SustainCom), 2020.
- [9] **W. Hussein and M. S. El-Seoud**, “Improved driver drowsiness detection model using relevant eye image’s features,” 2017 European Conference on Electrical Engineering and Computer Science (EECS), 2017.
- [10] **F. Mizoguchi, H. Nishiyama, and H. Iwasaki**, “A new approach to detecting distracted car drivers using eye-movement data,” 2014 IEEE 13th International Conference on Cognitive Informatics and Cognitive Computing, 2014.
- [11] “Extending RISC-v ISA with a custom instruction set extension,” Design And Reuse. [Online]. Available: <https://www.design-reuse.com/articles/46237/extending-risc-v-isa-with-a-custom-instruction-set-extension.html>. (accessed Jun 1, 2022).
- [12] **E. Staff**, “A quick introduction to instruction set architecture and extensibility,” Embedded.com, 14-Apr-2021. [Online]. Available: <https://www.embedded.com/a-quick-introduction-to-instruction-set-architecture-and-extensibility/>. (accessed Jun 1, 2022).
- [13] “History of RISC-V”, riscv.org, (2020, October 17). [Online]. Available: <https://riscv.org/about/history/#:~:text=The%20RISC-V%20Foundation%20>. (accessed May 30, 2022).

- [14] “V Foundation membership exceeds 100 percent growth over the past year - RISC-V International,” RISC, 01-Oct-2020 [Online]. Available: <https://riscv.org/announcements/2019/02/risc-v-foundation-membership-exceeds-100-percent-growth-over-the-past-year/>. (accessed Jun 1, 2022).
- [15] **E. Gholizadehazari, T. Ayhan, and B. Ors**, “An FPGA implementation of a RISC-V based SOC system for Image Processing Applications,” 2021 29th Signal Processing and Communications Applications Conference (SIU), 2021.
- [16] **A. Waterman, Y. Lee, D. A. Patterson, and K. Asanovi**, “The RISC-V instruction set manual. volume 1: User-level ISA, version 2.0,” 2014. Available: <https://web.eecs.utk.edu/~smarz1/courses/ece356/notes/assembly/>. (accessed Jun 1, 2022).
- [17] **S. Marz**, “RISC-V Assembly Language”. [Online]. Available: web.eecs.utk.edu. (accessed Jun 1, 2022).
- [18] **P. Nannipieri, S. Di Matteo, L. Zulberti, F. Albicocchi, S. Saponara, and L. Fanucci**, “A RISC-V post quantum cryptography instruction set extension for number theoretic transform to speed-up crystals algorithms,” IEEE Access, vol. 9, pp. 150798–150808, 2021.
- [19] **J. Chen, J. Li, Y. Li, and X. Miao**, “Multiply accumulate operations in memristor crossbar arrays for analog computing,” Journal of Semiconductors, vol. 42, no. 1, p. 013104, 2021.
- [20] “Open to the core,” lowRISC. [Online]. Available: <https://lowrisc.org/>. (accessed Jun 4, 2022).
- [21] “Pipeline details,” Pipeline Details - Ibex Documentation 0.1.dev50+g9b68b5e.d20220601 documentation. [Online]. Available: https://Ibex-core.readthedocs.io/en/latest/03_reference/pipeline_details.html. (accessed Jun 4, 2022).
- [22] “Wishbone (computer bus),” Wikiwand. [Online]. Available: [https://www.wikiwand.com/en/Wishbone_\(computer_bus\)](https://www.wikiwand.com/en/Wishbone_(computer_bus)). (accessed Jun 1, 2022).
- [23] “Introduction to the Wishbone bus interface a Systemonchip,” SlideToDoc.com. [Online]. Available: <https://slidetodoc.com/introduction-to-the-Wishbone-bus-interface-a-systemonchip/>. (accessed Jun 1, 2022).
- [24] “Wishbone classic bus cycle” WISHBONE Classic Bus Cycle - WISHBONE B3. [Online]. Available: https://Wishbone-interconnect.readthedocs.io/en/latest/03_classic.html. (accessed Jun 1, 2022).
- [25] **E. A. B. da Silva and G. V. Mendonça**, “Digital Image Processing,” in The Electrical Engineering Handbook, W.K. Chen, Ed., Boston: Elsevier Academic Press, 2005, pp. 891–910. [Online]. Available: <https://www.sciencedirect.com/science/article/pii/B9780121709600500645>. (accessed May 28, 2022).

- [26] **G.B. Taylor and C.L. Carilli and R.A. Perley**, “Synthesis Imaging in Radio Astronomy II”, in ASP Conference Series, vol. 180, 1999, pp.301-319. [Online]. Available: <https://adsabs.harvard.edu/pdf/1999ASPC..180..301F>
- [27] “Introduction to image processing” [esahubble.org. https://esahubble.org/static/projects/fits_liberator/image_processing.pdf](https://esahubble.org/static/projects/fits_liberator/image_processing.pdf) (accessed May 21,2022).
- [28] **P. Pedamka**, “RGB Color Model”, [educba.com. https://www.educba.com/rgb-color-model/](https://www.educba.com/rgb-color-model/) (accessed May 21, 2022).
- [29] “YCbCr”, [wikipedia.org. https://en.wikipedia.org/wiki/YCbCr](https://en.wikipedia.org/wiki/YCbCr) (accessed May 25, 2022).
- [30] **F. Max**, “RGB vs YCbCr: What are the Main Differences” [10scopes.com. https://10scopes.com/rgb-vs-ycbcr/#what-is-ycbcr](https://10scopes.com/rgb-vs-ycbcr/#what-is-ycbcr) (accessed May 21, 2022).
- [31] **I. Stirb, L. Deligiannidis and H. R. Arabnia**, “Highlight image filter significantly improves optical character recognition on text images,” in Emerging trends in image processing, computer vision, and pattern recognition, Amsterdam: Morgan Kaufmann, 2015, pp. 131–147.
- [32] **A. Getis and D. A. Griffith**, “Comparative spatial filtering in regression analysis,” *Geographical Analysis*, vol. 34, no. 2, pp. 130–140, 2002.
- [33] **R. Chandel and G. Gupta**, “International Journal of Advanced Research in Computer Science and Software Engineering,” Image Filtering Algorithms and Techniques: A Review, vol. 3, no. 10, pp. 198–202, Oct. 2013. [Online]. Available: https://www.researchgate.net/profile/Gaurav-Gupta-53/publication/325681876_Image_Filtering_Algorithms_and_Techniques_A_Review/links/5b1e1ab0aca272021cf585c9/Image-Filtering-Algorithms-and-Techniques-A-Review.pdf
- [34] “Kernel (image processing)” [wikipedia.org. https://en.wikipedia.org/wiki/Kernel_\(image_processing\)](https://en.wikipedia.org/wiki/Kernel_(image_processing)) (accessed May 22,2022).
- [35] **W.-J. Choi and T.-S. Choi**, “Fast three-dimensional shape recovery in TFT-LCD manufacturing,” *Applications of Digital Image Processing XXXI*, 2008.
- [36] **M. Basavarajaiah**, “6 Basic Things to Know About Convolution”, [medium.com. https://medium.com/@bdhuma/6-basic-things-to-know-about-convolution-daef5e1bc411](https://medium.com/@bdhuma/6-basic-things-to-know-about-convolution-daef5e1bc411) (accessed May 26, 2022).
- [37] **T. Siriburanon, S. Kondo, K. Kimura, T. Ueno, S. Kawashima, T. Kaneko, W. Deng, M. Miyahara, K. Okada, and A. Matsuzawa**, “A 2.2 GHz -242 DB-FOM 4.2 MW ADC-PLL using digital sub-sampling architecture,” *IEEE Journal of Solid-State Circuits*, vol. 51, no. 6, pp. 1385–1397, 2016.
- [38] **A. Saini and M. Biswas**, “Object detection in underwater image by detecting edges using adaptive thresholding,” 2019 3rd International Conference on Trends in Electronics and Informatics (ICOEI), 2019.

- [39] **A. Sengupta and R. Chaurasia**, “Hardware IP cores for image processing functions,” in *Advances in image and Data Processing using VLSI Design*, vol. 1, IOP Publishing Ltd, pp. 7–14.
- [40] **S. Malipatil, A. Gour, and V. Maheshwari**, “Custom IP design for fault-tolerant digital filters for high-speed imaging devices,” *Inventive Computation and Information Technologies*, pp. 305–314, 2021.
- [41] **S. Latti**, "Design and Implementation of DDR SDRAM Controller for Embedded Processors", *International Journal For Research & Development in Technology*, vol. 6, no. 2349-3585, p. 133, 2016. Available: <https://www.ijrdt.org/upload/6313724Design%20and%20Implementat ion%20of%20DDR%20SDRAM%20Controller%20for%20Embedded %20Processors.pdf>.
- [42] “Adaptable. intelligent,” Xilinx. [Online]. Available: <https://www.xilinx.com/>. (accessed Jun 4, 2022).
- [43] “Nexys 4 DDR,” Nexys 4 DDR - Digilent Reference. [Online]. Available: <https://digilent.com/reference/programmable-logic/nexys-4-ddr/start>. (accessed June 4, 2022).
- [44] RiscV-Collab, “RISCV-Collab/RISCV-GNU-toolchain: GNU Toolchain for RISC-V, including GCC,” GitHub. [Online]. Available: <https://github.com/riscv-collab/riscv-gnu-toolchain>. (accessed Jun 1, 2022).
- [45] **A. Pandit**, “How to Use OV7670 Camera Module with Arduino”, [circuitdigest.com](https://circuitdigest.com/microcontroller-projects/how-to-use-ov7670-camera-module-with-arduino). <https://circuitdigest.com/microcontroller-projects/how-to-use-ov7670-camera-module-with-arduino> (accessed May 25, 2022).
- [46] **R. Herveille**, “WISHBONE System-on-Chip (SoC) Interconnection Architecture for Portable IP Cores”, OpenCores, 2010. [Online]. Available: <https://opencores.org/howto/wishbone>
- [47] “Opcode”, [wikipedia.org](https://en.wikipedia.org/wiki/Opcode). <https://en.wikipedia.org/wiki/Opcode> (accessed May 25, 2022).
- [48] **N. Srivastava**, “Adding custom instruction to RISCV ISA”, <https://nitish2112.github.io/post/adding-instruction-riscv/> (accessed 25 May 2022)
- [49] “Microprocessor Design/Instruction Decoder”, [wikibooks.org](https://en.wikibooks.org/wiki/Microprocessor_Design/Instruction_Decoder). https://en.wikibooks.org/wiki/Microprocessor_Design/Instruction_Decoder (accessed May 25, 2022).
- [50] “OV7670/OV7171 CMOS VGA (640x480) CameraChip Sensor with OmniPixel Technology”, OmniVision, 2006. [Online]. Available: https://web.mit.edu/6.111/www/f2016/tools/OV7670_2006.pdf
- [51] VGA video driver V6.0.10.284 - IBM system X3610 (7942), 07-Jun-2008. [Online]. Available: <https://www.ibm.com/support/pages/vga-video-driver-v6010284-ibm-system-x3610-7942>. (accessed Jun 5, 2022).

- [52] "Nexys Video," Nexys Video - Digilent Reference. [Online]. Available: <https://digilent.com/reference/programmable-logic/nexys-video/start>. (accessed June 10, 2022).
- [53] **P. Bibay, A. Sahu and V. Chandra**, "Design and Implementation of DDR SDRAM Controller using Verilog", *International Journal of Science and Research*, vol. 2, no. 1, pp. 320-321, 2013. Available: <https://www.ijsr.net/archive/v2i1/IJSROFF130201036.pdf>.
- [54] **R. Jesman, F. Vallina and J. Saniee**, "MicroBlaze Tutorial Creating a Simple Embedded System and Adding Custom Peripherals Using Xilinx EDK Software Tools", *Ecasp.ece.iit.edu*, 2006. [Online]. Available: http://ecasp.ece.iit.edu/tutorials/microblaze_tutorial.pdf. (accessed May 31, 2022).
- [55] **S. Thornton**, "What is DDR (Double Data Rate) Memory and SDRAM Memory", *Microcontrollertips.com*, 2022. [Online]. Available: <https://www.microcontrollertips.com/understanding-ddr-sdram-faq/>. (accessed May 31, 2022).
- [56] **I. Kuon, R. Tessier and J. Rose**, *FPGA Architecture: Survey and Challenges*. Hanoven, MA, USA: Now Publishers. 2007, pp. 1-7.
- [57] **D. Pyatkov**, "DDR RAM CONTROLLER FOR THE CYCLONE II FPGA", *People.ece.cornell.edu*, 2008. [Online]. Available: <https://people.ece.cornell.edu/land/courses/eceprojectsland/STUDENTPROJ/2007to2008/dp239/Denis-MEng-Final-nocode.pdf>.
- [58] **P. Trivedi and R. P. Tripathi**, "Design & analysis of 16 bit RISC processor using low power pipelining," in *International Conference on Computing, Communication & Automation*, 2015, pp. 1294-1297, doi: 10.1109/CCA.2015.7148575.
- [59] **M. Siegesmund**, "Preprocessor Directives", in *Embedded C Programming: Techniques and Applications of C and PIC MCUS*, M. Siegesmund, Ed. 2014, p. 35.
- [60] "Vitis Software Platform", Xilinx. [Online]. Available: <https://www.xilinx.com/products/design-tools/vitis/vitis-platform.html>. (accessed Jun 6, 2022).
- [61] **M. Skreen**, "How to Store Your SDK Project in SPI Flash - Digilent Reference", *Digilent.com*, 2019. [Online]. Available: <https://digilent.com/reference/learn/programmable-logic/tutorials/htsspif/start>.
- [62] Xilinx Customer Community. [Online]. Available: https://support.xilinx.com/s/article/1053914?language=en_US. (accessed Jun 4, 2022).
- [63] "AXI General Purpose IO", Xilinx. [Online]. Available: https://www.xilinx.com/products/intellectual-property/axi_gpio.html. (accessed Jun 6, 2022).
- [64] "Documentation Portal", *Docs.xilinx.com*. [Online]. Available: <https://docs.xilinx.com/v/u/en-US/pg142-axi-uartlite>. (accessed Jun 6, 2022).

- [65] "AXI Quad SPI", Xilinx. [Online]. Available: https://www.xilinx.com/products/intellectual-property/axi_quadspi.html#overview. (accessed Jun 6, 2022).
- [66] "Memory Interface", Xilinx. [Online]. Available: <https://www.xilinx.com/products/intellectual-property/MIG.html>. (accessed May 31, 2022).
- [67] "GitHub - vproc/vicuna: RISC-V Zve32x Vector Coprocessor", GitHub. [Online]. Available: <https://github.com/vproc/vicuna>. (accessed May 31, 2022).
- [68] **M. Platzer and P. Puschner**, (7-9 Jul. 2021). Vicuna: A Timing-Predictable RISC-V Vector Coprocessor for Scalable Parallel Computation. Presented in 33rd Euromicro Conference on Real-Time Systems. [Online] Available: <https://drops.dagstuhl.de/opus/volltexte/2021/13932/pdf/LIPICs-ECRTS-2021-1.pdf>
- [69] "GitHub - ZipCPU/wb2axip: Bus bridges and other odds and ends", GitHub. [Online]. Available: <https://github.com/ZipCPU/wb2axip>. (accessed May 31, 2022).
- [70] "İTÜ Gömülü Sistem Tasarım Laboratuvarı (GSTL)" [Online]. Available: <https://www.gstl.itu.edu.tr/hakkimizda/>. (accessed Jun 8, 2022).
- [71] **E. Pena, M.G. Legaspi**, "UART: A hardware communication protocol understanding universal asynchronous receiver/transmitter," UART: A Hardware Communication Protocol Understanding Universal Asynchronous Receiver/Transmitter | Analog Devices. [Online]. Available: <https://www.analog.com/en/analog-dialogue/articles/uart-a-hardware-communication-protocol.html>. (accessed June 9, 2022).
- [72] **C. Wootton**, "General purpose input/output (GPIO)," Samsung ARTIK Reference, pp. 235–288, 2016.

APPENDICES

APPENDIX A: Design Source Files in Hierarchy

APPENDIX B: Filter Code

APPENDIX A

```
Design Sources (19)
├── ibex_soc (ibex_soc.sv) (5)
│   ├── wb_ibex_core : wb_ibex_core (wb_ibex_core.sv) (3)
│   │   ├── inst_ibex_core : ibex_core (ibex_core.sv) (7)
│   │   │   ├── core_clock_gate_i : prim_clock_gating (prim_clock_gating.sv)
│   │   │   ├── if_stage_i : ibex_if_stage (ibex_if_stage.sv) (2)
│   │   │   │   ├── prefetch_buffer_i : ibex_prefetch_buffer (ibex_prefetch_buffer.sv) (1)
│   │   │   │   │   ├── fifo_i : ibex_fetch_fifo (ibex_fetch_fifo.sv)
│   │   │   │   │   └── compressed_decoder_i : ibex_compressed_decoder (ibex_compressed_decoder.sv)
│   │   │   ├── id_stage_i : ibex_id_stage (ibex_id_stage.sv) (3)
│   │   │   │   ├── registers_i : ibex_register_file (ibex_register_file.sv)
│   │   │   │   ├── decoder_i : ibex_decoder (ibex_decoder.sv)
│   │   │   │   └── controller_i : ibex_controller (ibex_controller.sv)
│   │   │   ├── ex_block_i : ibex_ex_block (ibex_ex_block.sv) (3)
│   │   │   │   ├── alu_i : ibex_alu (ibex_alu.sv)
│   │   │   │   ├── gen_multdiv_fast_multdiv_i : ibex_multdiv_fast (ibex_multdiv_fast.sv)
│   │   │   │   └── ibex_multdiv_slow (ibex_multdiv_slow.sv)
│   │   │   ├── load_store_unit_i : ibex_load_store_unit (ibex_load_store_unit.sv)
│   │   │   ├── cs_registers_i : ibex_cs_registers (ibex_cs_registers.sv)
│   │   │   └── ibex_pmp (ibex_pmp.sv)
│   │   └── ibex_pmp (ibex_pmp.sv)
│   ├── instr_core2wb : core2wb (core2wb.sv)
│   ├── data_core2wb : core2wb (core2wb.sv)
│   ├── wb_intercon : wb_interconnect_sharedbus (wb_interconnect_sharedbus.sv)
│   ├── wb_spram : wb_spramx32 (wb_spramx32.sv) (1)
│   │   └── spram : spramx32 (spramx32.sv)
│   ├── wb_ov7670_erfan : wb_ov7670_erfan (wb_ov7670_erfan.sv) (2)
│   │   ├── u_ov7670_init : ov7670_init (ov7670_init.v) (2)
│   │   │   ├── u_I2C_OV7670_RGB565_Config : I2C_OV7670_RGB565_Config2 (I2C_OV7670_RGB565)
│   │   │   └── u_I2C_Controller : I2C_Controller2 (I2C_Controller2.v)
│   │   └── u_ov7670_capture : ov7670_capture (ov7670_capture.v)
│   ├── wb_vga_erfan : wb_vga_erfan (wb_vga_erfan.sv) (1)
│   │   └── u_vga : vga (vga.v)
│   ├── wbuart (wbuart.v) (4)
│   │   ├── rx : rxuart (rxuart.v)
│   │   ├── rxfifo : ufifo (ufifo.v)
│   │   ├── txfifo : ufifo (ufifo.v)
│   │   └── tx : txuart (txuart.v)
│   └── dm_top (dm_top.sv) (3)
│       └── i_dm_csrs : dm_csrs (dm_csrs.sv) (1)
```



Figure A.1: Design Source Files in Hierarchy

APPENDIX B

```
1  #include <stdint.h>
2
3  #define CLK_FIXED_FREQ_HZ (50ULL * 1000 * 1000)
4
5  /**
6   * Delay loop executing within 8 cycles on ibex
7   */
8  static void delay_loop_ibex(unsigned long loops) {
9      int out; /* only to notify compiler of modifications to |loops| */
10     asm volatile(
11         "1: nop          \n" // 1 cycle
12         "   nop          \n" // 1 cycle
13         "   nop          \n" // 1 cycle
14         "   nop          \n" // 1 cycle
15         "   addi %1, %1, -1 \n" // 1 cycle
16         "   bnez %1, 1b   \n" // 3 cycles
17         : "=r" (out)
18         : "0" (loops)
19     );
20 }
21 static int usleep_ibex(unsigned long usec) {
22     unsigned long usec_cycles;
23     usec_cycles = CLK_FIXED_FREQ_HZ * usec / 1000 / 1000 / 8;
24
25     delay_loop_ibex(usec_cycles);
26     return 0;
27 }
28 static int usleep(unsigned long usec) {
29     return usleep_ibex(usec);
30 }
31
32
33 static int custom0(unsigned int a, signed int b) {
34     static int result;
35     asm volatile( "cust0 %[result1], %[value1], %[value2]\n\t"
36                 :[result1] "=r" (result) : [value1] "r" (a), [value2] "r" (b));
37     return result;
38 }
39
40 static int custom1(unsigned int a, unsigned int b) {
41     static int result;
42     asm volatile( "cust1 %[result1], %[value1], %[value2]\n\t"
43                 :[result1] "=r" (result) : [value1] "r" (a), [value2] "r" (b));
44     return result;
45 }
46
47 static int custom2(unsigned int a, unsigned int b) {
48     static int result;
49     asm volatile( "cust2 %[result1], %[value1], %[value2]\n\t"
50                 :[result1] "=r" (result) : [value1] "r" (a), [value2] "r" (b));
51     return result;
52 }
53
54
55
56
57 volatile uint32_t *pixel = (volatile uint32_t *) 0xC010;
58
59 int main() {
60
61     volatile uint32_t *var = (volatile uint32_t *) 0x10000000;
62     volatile int32_t data1=0;
63     volatile int32_t data2=0;
64     volatile int32_t tester = 0;
65     volatile int32_t i=0,j=0,x=0,y=0,pixel_val=0;
66
67
68
69     int kernel[3][3] =
70     {
71         { 0 , -1 , 0},
72         {-1 , 4 , -1},
73         { 0 , -1 , 0}
74     };

```

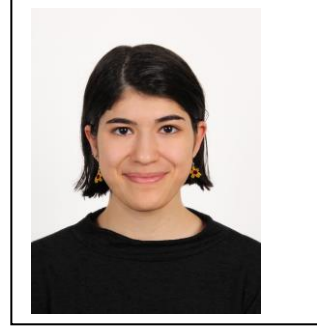
```

75
76 // Load Kernel Data - 1st
77 /*for (j = 0 ; j<3; j = j + 1)
78     for (i = 0 ; i<3; i = i + 1)
79         custom0( j*3+i , kernel[i][j] );*/
80 custom0( 0 , -1 );
81 custom0( 1 , 0 );
82 custom0( 2 , 1 );
83 custom0( 3 , -2 );
84 custom0( 4 , 0 );
85 custom0( 5 , 2 );
86 custom0( 6 , -1 );
87 custom0( 7 , 0 );
88 custom0( 8 , 1 );
89
90
91 pixel[0] = 0x00000001; // Activate camera module to initialize RAM with data
92 usleep(1000 * 15); // 15 ms
93 pixel[0] = 0x00000000;
94
95 pixel[0] = 0x00000100;
96
97 for (j = 0 ; j<240; j = j + 1)
98 {
99     for (i = 1 ; i<321; i = i + 1)
100     {
101         pixel_val = 0 ;
102         data1=(pixel[320*(j+1)+i+1]<<28)|(pixel[320*(j+1)+i]<<21)|(pixel[320*j+i+2]<<14)|(pixel[320*j+i+1]<<7)|(pixel[320*j+i]) ;
103         data2=(pixel[320*(j+2)+i+2]<<24)|(pixel[320*(j+2)+i+1]<<17)|(pixel[320*(j+2)+i]<<10)|(pixel[320*(j+1)+i+2]<<3)|(pixel[320*(j+1)+i+1]>>4) ;
104
105         pixel_val = custom1(data1,data2);
106
107         tester = pixel[i+j*320];
108         if (pixel_val>255) pixel_val = 255;
109         if (pixel_val<0) pixel_val = 0;
110         pixel[i+j*320] = ((pixel_val)<<16) | (tester&0xFFFF) ;
111
112     }
113 }
114
115
116 pixel[0] = 0x0000010000; // Activate VGA module to read pixels from RAM
117 while(1);
118 }

```

Figure B.1: Filter Code

CURRICULUM VITAE



Name Surname : Elif Dinç
Place and Date of Birth : Ankara - 06/06/1999
E-Mail : dince17@itu.edu.tr

EDUCATION

- **B.Sc.:** Istanbul Technical University – Electronics and Communication Engineering (2017-2022)

PROFESSIONAL EXPERIENCE

- 01.09.2020-01.10.2021, Part Time Team Member, ITU Çekirdek
- 15.06.2019-15.03.2020, Laboratory Assistant, ITU Industrial Automation Laboratory

CURRICULUM VITAE



Name Surname : Gülce Baysal
Place and Date of Birth : Manisa - 29/01/1999
E-Mail : baysalg17@itu.edu.tr

EDUCATION

- **B.Sc.:** Istanbul Technical University – Electronics and Communication Engineering (2017-2022)

PROFESSIONAL EXPERIENCE

- 17.06.2019-12.07.2019, Design and Quality Assurance (DQA) Researcher and Test Assistant, Vestel Electronics
- 02.08.2021-18.09.2021, Responsible Researcher in the Electronic System Development Unit, BAYKAR
- 11.10.2021 – 01.06.2022, Part-Time Working Student, Siemens Advanta

CURRICULUM VITAE



Name Surname : Merve Kılıç
Place and Date of Birth : Samsun - 02/02/1999
E-Mail : kilicmer17@itu.edu.tr

EDUCATION

- **B.Sc.:** Istanbul Technical University – Electronics and Communication Engineering (2017-2022)

PROFESSIONAL EXPERIENCE

- 10.08.2021-07.09.2021, Digital Circuit Design Intern, TUBITAK BILGEM
- 22.11.2022-30.05.2022, Part-Time Working Student, TUBITAK BILGEM