<u>**ISTANBUL TECHNICAL UNIVERSITY**</u>
<u>**ELECTRICAL-ELECTRONICS FACULTY**</u>

REALIZATION OF FREQUENCY BASED IMAGE STEGANOGRAPHY USING RISC-V PROCESSOR

<u>**SENIOR DESIGN PROJECT**</u>

Yaşar Utku ALÇALAR
Sinem Başak KAPUCU
Burcu TÜRK

ELECTRONICS AND COMMUNICATION ENGINEERING DEPARTMENT

JUNE 2022

ISTANBUL TECHNICAL UNIVERSITY
ELECTRICAL-ELECTRONICS FACULTY

REALIZATION OF FREQUENCY BASED IMAGE STEGANOGRAPHY USING RISC-V PROCESSOR

SENIOR DESIGN PROJECT

Yaşar Utku ALÇALAR
040170054

Sinem Başak KAPUCU
040180005

Burcu TÜRK
040170052

ELECTRONICS AND COMMUNICATION ENGINEERING DEPARTMENT

Project Advisor: Prof. Dr. Sıddıka Berna ÖRS YALÇIN

JUNE 2022

İSTANBUL TEKNİK ÜNİVERSİTESİ
ELEKTRİK-ELEKTRONİK FAKÜLTESİ

FREKANS TEMELLİ GÖRÜNTÜ STEGANOGRAFİSİNİN RISC-V İŞLEMCİ KULLANILARAK GERÇEKLENMESİ

LİSANS BİTİRME TASARIM PROJESİ

Yaşar Utku ALÇALAR
040170054

Sinem Başak KAPUCU
040180005

Burcu TÜRK
040170052

Proje Danışmanı: Prof. Dr. Sıddıka Berna ÖRS YALÇIN

ELEKTRONİK VE HABERLEŞME MÜHENDİSLİĞİ BÖLÜMÜ

HAZİRAN, 2022

We are submitting the Senior Design Project Report entitled as “REALIZATION OF FREQUENCY BASED IMAGE STEGANOGRAPHY USING RISC-V PROCESSOR”. 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 .

Yaşar Utku ALÇALAR 040170054	U.alcalar ....
Sinem Başak KAPUCU 040180005	[illegible] ....
Burcu TÜRK 040170052	[illegible] ....

To our loved ones,

FOREWORD

We would like to express our gratitude to our project advisor Prof. Dr. Sıddıka Berna Örs Yalçın who helped us throughout the process and guided us through the exciting world of digital electronics and cryptography. We are also thankful for our university faculty for encouraging us in the way of becoming successful engineers.

Finally, we would like to thank deeply to our families who supported our education and helped us become who we are.

June 2022

Yaşar Utku ALÇALAR
Sinem Başak KAPUCU
Burcu TÜRK

TABLE OF CONTENTS

Page

ABBREVIATIONS

2D	: Two-dimensional
ASCII	: American Standard Code for Information Interchange
FPGA	: Field Programmable Gate Array
GNU	: GNU’s Not Unix
HDL	: Hardware Definition Language
IEEE	: The Institute of Electrical and Electronics Engineers
IP	: Intellectual Property
ISA	: Instruction Set Architecture
LED	: Light-Emitting Diode
LFSR	: Linear Feedback Shift Register
Linux	: Lovable Intellect Not Using XP
LSB	: Least Significant Bit
LTS	: Long-Term Support
MATLAB	: Matrix Laboratory
OS	: Operating System
RAM	: Random Access Memory
RISC	: Reduced Instruction Set Computer
ROM	: Read-Only Memory
RSA	: Rivest–Shamir–Adleman
RTL	: Register-Transfer Level
SoC	: System on Chip
SRAM	: Static Random Access Memory

SYMBOLS

$	**:** Ubuntu Terminal Command
$\boldsymbol{G_x}$	**:** Gradient in x-direction
$\boldsymbol{G_y}$	**:** Gradient in y-direction

LIST OF TABLES

Page

LIST OF FIGURES

REALIZATION OF FREQUENCY BASED IMAGE STEGANOGRAPHY USING RISC-V PROCESSOR

SUMMARY

Due to the rapid increase in today's technology and internet usage, the security of information and communication channels is becoming more and more crucial. Therefore, several methods and algorithms have been developed to increase security. Perhaps the most widely heard of these methods is cryptography. The science of cryptography is the encryption of the message with various algorithms in order to protect its confidentiality, while sending a message from a transmitter to a receiver. The main purpose in cryptography is to ensure that this message is uncomeatable, even if the message/information is accessed by third parties who do not have permission to access it.

Another method that can be used to ensure this secure communication is steganography. The main purpose of steganography is to hide the occurrence of a message transmission. In this project, these two methods were used together, and both the existence of transmitted information is hidden and in case its existence is noticed, precautions are taken by encrypting the message in order to ensure the confidentiality of the message. The process in the steganography part is carried out by hiding the text-formatted message into an image-formatted data, this type of steganography is called image steganography.

In this project, a simple Least Significant Bit (LSB) image steganography method was used without including cryptography in the first stage. In this method, each bit of the message is sequentially embedded in the least significant bits of the pixels of an image type data used as a cover image.

Afterwards, an algorithm has been added in which the pixel selection will be made randomly. The Linear Feedback Shift Register (LFSR) algorithm, which can generate pseudo-random numbers, is used. The basic logic of the LFSR method is based on the generation of a new number by XORing some bits of a randomly selected number one by one and writing the value obtained by one bit right shifted into the most significant bit of the number. The generated pseudo-random numbers were chosen as the pixel numbers in which the bits of the message would be embedded. To encrypt the message, the key sequence containing the numbers created with the LFSR algorithm must be known.

For a more efficient algorithm, it is considered to increase the message bit encoded into some pixels from one to two. To prevent visibility due to changing 2 bits in pixel, pixels with high frequency in the image are detected and two bits are embedded in these pixels.

In order to determine the high frequency regions, the cover image was filtered by the two-dimensional convolution method using the Sobel filter. Pixels are marked as high and low frequency using a threshold. To increase the capacity, two thresholds are used instead of one, and the pixels are divided into three as low, medium, and high

frequency. Here, messages containing 1, 2 and 3 bits of information are embedded in these frequency regions, from low to high, respectively.

The mentioned algorithms were first simulated using the MATLAB. After that, the Vivado environment was used for the implementation on the Field Programmable Gate Array (FPGA). Nexys4 DDR board is used in this project. Ibex Core has been added to the Vivado project and its functionality has been tested. Afterwards, each image steganography algorithm was implemented as software using the C programming language and converted into a memory file and added to the Vivado project. Simulations of the algorithms were performed in Vivado, then the pixel values obtained as a result were shown as stego images and the encoded message was decrypted using MATLAB.

To display the encoded images, the image is given to a monitor connected to the card by using the Video Graphics Array (VGA) output of the card. Although the Ibex core has the ability to communicate with the instruction and data memory, the Wishbone protocol is integrated into the Ibex core so that it can communicate with an external unit such as VGA. Thus, separate units of Ibex core, Random Access Memory (RAM), VGA and the "interconnect" module that make the connection between them were created.

The pixels that are result of encoding process are written to the RAM with the relevant code lines in the C code. Then, by activating the VGA module, the pixel values in the RAM were read and the image was displayed on the monitor. In addition, some of the message obtained by decrypting the encoded image in the C code was observed on the Light-Emitting Diodes (LED) on the board.

Finally, the filtering process was transferred from software to hardware for accelerating the process. For this, a module in which image pixel values are read and filtered from RAM and written back to RAM was included in the project and connections with other modules were made. Afterward, the filtered image written to the RAM by the module was given to the monitor via VGA to verify that the module is working properly on the board. Thus, the planned steps in the project have been completed.

FREKANS TEMELLİ GÖRÜNTÜ STEGANOGRAFİSİNİN RISC-V İŞLEMCİ KULLANILARAK GERÇEKLENMESİ

ÖZET

Günümüzdeki teknoloji ve internet kullanımının hızla artışı nedeniyle bilginin ve haberleşme kanallarının güvenliği de gitgide daha da büyük bir önem arz etmektedir. Bu nedenle, güvenliği artırmak için çeşitli yöntemler ve algoritmalar geliştirilmiştir. Bu yöntemlerden belki de en çok duyulanı ve bilineni kriptografidir. Kriptografi bilimi bir vericiden bir alıcıya mesaj gönderilirken mesajın gizliliğini koruyabilmek adına çeşitli algoritmalar ile şifrelenmesidir. Kriptografide asıl amaç mesajın/bilginin erişime izni olmayan üçüncü kişiler tarafından ulaşılması dahilinde bu mesajın güvenliğini sağlamaktır. Bu işlemi çeşitli algoritmalar sayesinde ve mesajı şifreleme anahtarı olmadan çözülemeyecek bir forma dönüştürerek gerçekleştirir.

İletişim güvenliğini sağlamak için kullanılabilecek bir diğer yöntem ise steganografidir. Steganografideki temel amaç ise bir mesaj iletiminin gerçekleştiğinin gizlenmesidir. Bu projede güvenli bir haberleşme ortamı ve veri iletimi sağlamak için bahsedilen bu iki yöntem bir arada kullanılmıştır. Böylece, hem iletilen bir bilginin varlığı gizlenmiş hem de varlığının fark edilmesi durumunda mesaj gizliliğini sağlamak için mesaj şifrelenerek önlem alınmıştır. Steganografi kısmındaki işlem metin formatındaki mesajı görüntü formatındaki bir veriye gizleyerek gerçekleştirilir ki bu tür steganografiye görüntü steganografisi denir.

Projenin ilk aşamasında kriptografi dahil edilmeden basit bir görüntü steganografisi yöntemi kullanılmıştır ve bu yöntemin adı Least Significant Bit (LSB) yöntemidir. Bu metotta, mesaj ikilik forma dönüştürüldükten sonra her bir biti sıralı olarak kapak görüntüsü olarak kullanılan bir görüntü verisinin piksellerinin en az anlamlı bitlerine gömülür. Piksel seçimi görüntünün başlangıç pikselinden başlar ve ikilik mesajın uzunluğu boyunca ardışık olarak devam eder. Ancak bu yöntemde mesajın kapak görüntüsünden çıkarılması oldukça kolaydır. Bu sebeple piksel seçiminin rastgele yapılacağı bir algoritma eklenmesine karar verilmiştir ve bunun sağlayacağı karmaşıklık sayesinde mesajın anahtar olmadan elde edilmesi olasılığının oldukça düşürülmesi hedeflenmiştir. Piksel seçiminin rastgele yapılması için yarı-rastgele sayıların üretebildiği Linear Feedback Shift Register (LFSR) algoritması kullanılmıştır. LFSR yönteminin temel mantığı rastgele seçilen bir sayının bazı bitlerinin tek tek XOR işleminden geçirilmesi ile elde edilen değerin bir bit sağa kaydırılmış sayının en anlamlı bitine yazılmasıyla yeni bir sayı üretilmesine dayanır. Üretilen yarı-rastgele sayılar mesajın bitlerinin gömüleceği piksel numaraları olarak seçilmiş, böylece mesaj görüntünün içine sıralı olmayan şekilde gizlenmiştir. Mesajın şifresinin çözülebilmesi için LFSR algoritması ile oluşturulan sayıların bulunduğu anahtar dizisi bilinmelidir.

Algoritmanın geliştirilmesi amacıyla bir görüntüye gizlenebilecek mesaj kapasitesinin artırmak için yeni bir yöntem arayışına girilmiştir. Daha verimli bir algoritma için bazı piksellerin içine şifrelenen mesaj biti sayısının birden ikiye çıkarılması yöntemi göz önünde bulundurulmuştur. Bir pikselin sadece en az anlamlı biti yerine son iki en az

anlamlı bitlerinin birden değiştirilmesi piksel değerini daha fazla değiştireceğinden insan gözü tarafından algılanabilmesi daha olası hale gelir. Görüntüdeki değişimlerin gözle görülebilme durumunu önlemek amacıyla görüntüdeki frekansı yüksek olan pikseller belirlenmiştir ve iki bitlik mesaj bilgisinin gömüleceği pikseller olarak bu yüksek frekanslı pikseller kullanılmıştır. Çünkü yüksek frekans bölgeleri ani ton değişimlerinin olduğu bölgelere işaret eder ve bu bölgelerde yapılacak olan değişimlerin insan gözü tarafından algılanması daha zordur.

Yüksek frekans bölgelerinin belirlenmesi için kapak görüntüsü Sobel filtresi kullanılarak iki boyutlu konvolüsyon yöntemi ile filtrelenmiştir. Sobel filtresinde iki ayrı kernel kullanılmaktadır. Bu iki ayrı kernel için de konvolüsyon sonuçları hesaplandıktan sonra bu sonuçların mutlak değerleri toplanarak filtre çıkışı elde edilir. Başlangıçta pikseller düşük ve yüksek frekans bölgesi olarak ikiye ayrılmak istendiğinden bu tek bir eşik değer kullanılarak sağlanmıştır. Eşik değerin üstünde kalan pikseller yüksek frekanslı altında kalanlar ise düşük frekanslı olarak işaretlenmiştir. Bir sonraki adımda kapasiteyi daha da artırmak adına bir yerine iki eşik değer kullanılmış ve pikseller düşük, orta ve yüksek frekans olmak üzere üçe ayrılmıştır. Burada düşük frekanslı piksellere 1, orta frekanslı piksellere 2 ve yüksek frekanslı piksellere 3 bitlik bilgi içeren mesaj gömülmüştür.

Bahsedilen algoritmaların ilk olarak MATLAB uygulaması kullanılarak benzetimleri yapılmıştır. Sonrasında bu algoritmaların FPGA üzerinde gerçeklenmesi amacıyla Vivado ortamına geçilmiştir. Fiziksel gerçekleme aşamasında Nexys4 DDR kartı kullanılmıştır. Bu amaçla Vivado projesine Ibex Core eklenmiş ve çalışırlığı test edilmiştir. Sonrasında her bir görüntü steganografisi algoritması C programlama dili kullanılarak yazılımsal olarak gerçeklenmiş ve bellek dosyalarına dönüştürülerek Vivado projesine eklenmiştir. Böylece yazılan algoritmaya ilişkin talimatlar Vivado projesinde okunabilir hale gelmiştir. Bu işlemlerin ardından Vivado'da benzetimler gerçekleştirilmiş ve sonucunda elde edilen piksel değerleri bir ".txt" uzantılı dosyaya kaydedilmiştir. Bu dosyalar MATLAB ortamına taşınarak içinde mesaj da içeren bu pikseller stego-görüntü olarak gösterilmiş ve şifrelenmiş mesaj deşifre edilmiştir. Bu aşama Vivado ortamındaki benzetimlerin doğruluğunu test etmek amacıyla yapılmıştır.

Şifrelenmiş görüntüleri görüntüleyebilmek amacıyla kartın Video Graphics Array (VGA) çıkışını kullanarak görüntü karta bağlanan bir monitöre verilmiştir. Ibex core talimat ve veri belleği ile haberleşme yeteneğine sahip olsa da VGA gibi dış bir birim ile haberleşebilmesi için Ibex core'a Wishbone protokolü entegre edilmiştir. Böylece Vivado projesinde Ibex core, Random Access Memory (RAM), VGA ve bunlar arasındaki bağlantıyı gerçekleştiren “interconnect” birimlerine ait ayrı modüller oluşturulmuştur.

Şifreleme işleminden geçen pikseller C kodundaki ilgili kod satırları ile RAM'e yazılmıştır. Sonrasında VGA modülünün aktifleştirilmesi ile RAM'deki piksel değerleri okunarak görüntünün monitörde görüntülenmesi sağlanmıştır. Buna ek olarak şifrelenmiş görüntü C kodunda deşifre edilerek elde edilen mesajın bir kısmı kart üzerinde bulunan ledlerde gözlenmiştir.

Son olarak süreci hızlandırmak amacıyla filtreleme işlemi yazılımdan domanıma geçirilmiştir. Bunun için görüntü piksel değerlerinin RAM'den okunup filtrelendikten sonra tekrar RAM'e yazıldığı bir modül projeye dahil edilmiş ve diğer modüllerle bağlantıları yapılmıştır. Sonrasında bu modüllerin kart üzerinde çalıştığının doğrulanması için modül tarafından RAM'e yazılan filtrelenmiş görüntü VGA aracılığıyla monitöre verilmiştir. Böylece projedeki yapılması planlanan adımlar tamamlanmıştır.

1. INTRODUCTION

1.1 Purpose of Project

In today's world, information security is one of the most significant elements of digital communication systems. Encrypting the message during the transmission from the transmitter to the receiver will increase the security of the communication environment. For that purpose, various cryptography methods are applied. However, the fact that cryptology methods only prevent the detectability of the transmitted message cannot hide the fact that communication is made. On the other hand, steganography offers a solution to this security problem by hiding the fact that communication is taking place. By using these two sciences together, a highly secure communication environment can be formed.

In our work, the main goal is enhancing a simple LSB steganography algorithm with an LFSR and frequency detection algorithms and implementing them on a RISC-V processor. During the implementation, it is aimed to find the slow-running software parts and speed-up the system by switching those slow-running sections with custom hardware structures.

1.2 Literature Review

1.2.1 Image steganography

As A. Yahya (2019) investigated in his book, steganography aims to ensure the undetectability of the communication by hiding the secret message in data. This data can be in text, audio, image, or video format [1]. For this project, this data is selected as an image, hence image steganography will be studied. Various steganography methods are available and applied, such as spatial domain and frequency domain steganography. Because of its simplicity and uniqueness, one of the spatial domain algorithms frequently used by researchers is The Least Significant Bit (LSB) algorithm. To minimize any visual change or damage in the image, the secret message bit is embedded in the last bit of the selected pixel. Thus, it does not arouse suspicion in anyone [2].

1.2.2 LFSR

The LSB algorithm seems weak considering the detection risk. Thus, in addition to hiding the secret message with the LSB algorithm, the bits of the message should be embedded in the pixels of the image in a random order for the encryption phase of the message. The study of Ghosh et. al (2020) showed that it is better to make use of the random nature of the LFSR algorithm when selecting the pixels to embed than the traditional LSB algorithm. In that way, selecting the pixels to embed a message bit will enhance the security, thus ensuring a more imperceptible transmission of information to be formed [3].

1.2.3 Frequency based algorithms

A more efficient image steganography method could be investigated to hide more information on an image of the same size compared to the LSB steganography method. For this, it is necessary to embed information in the last few bits of some pixels in the image, instead of only the least significant bit. Kalaichelvi et. al (2020) proposed a steganography system using a Canny-Edge detection technique and a modified RSA (Rivest–Shamir–Adleman) algorithm for that purpose. In their study, they applied methods such as filtering the image for noise elimination and determination of the edge points in an image to be able to embed more than 1 bit in a pixel. According to the results, they found that the proposed method has increased the security, undetectability and efficiency of the traditional RSA algorithm. For the steganography method to be used in the project, the most efficient frequency based algorithm to embed more information in an image will be investigated [4].

2. BASIC INFORMATION AND CONCEPTS

2.1 Cryptography

In today's world, there are disadvantages as well as advantages of being so intertwined with technology. Considering that almost all communication is carried out in virtual environments such as the internet, security problems that may arise during information transmission become inevitable. In such environments where communication is established, the search for security increases over time. There are some features that a communication environment must provide to be secure. In order to create an environment where these features are provided, various methods such as cryptography have been produced in time. Cryptography is a method of encryption that aims to protect information as well as the sender and receiver in a transmission. Basically, the data to be transmitted in a communication system is encrypted with the cryptography method and transmitted as encrypted. That way, even if the message is captured without permission and unauthorized access, information leakage is prevented since this encrypted message is incomprehensible.

Keyless encryption, symmetric encryption, or asymmetric encryption methods can be used for encryption in a cryptographic system.

Key generation, generated key distribution and its storage are quite complex operations. One of the most important advantages of this method is that there is no need to use such complex algorithms in keyless encryption. In addition to the difficulty of processing, choosing keyless encryption can be an advantage depending on the situation, as long and complex keys will require more power and memory. Furthermore, one of the advantages of this method is that there is no possibility of any key being revealed.

The symmetric key method uses only one key for encryption and decryption processes. Symmetric key cryptography is relatively simple and therefore requires less computational power. In the asymmetric key method, there are two keys. One of these keys is public and the other is private. The public key is for encryption, and if this key is known, it can transmit information. However, thanks to the secret key, which is only on the receiving side and used for decryption, the transmitted message is prevented from being caught by someone else [5].

2.2 Image Steganography

Steganography also offers a solution to provide a secure communication environment. It aims to hide the existence of confidential information intended to be transmitted by embedding a secret message in an unsuspecting piece of information.

Steganography methods vary depending on the type of data to be used as the cover. For instance, if an audio file is used as a cover while transmitting text-type data, it is called audio steganography.

In Figure 2.1, different security systems and types of steganography methods are shown as a tree schematic.

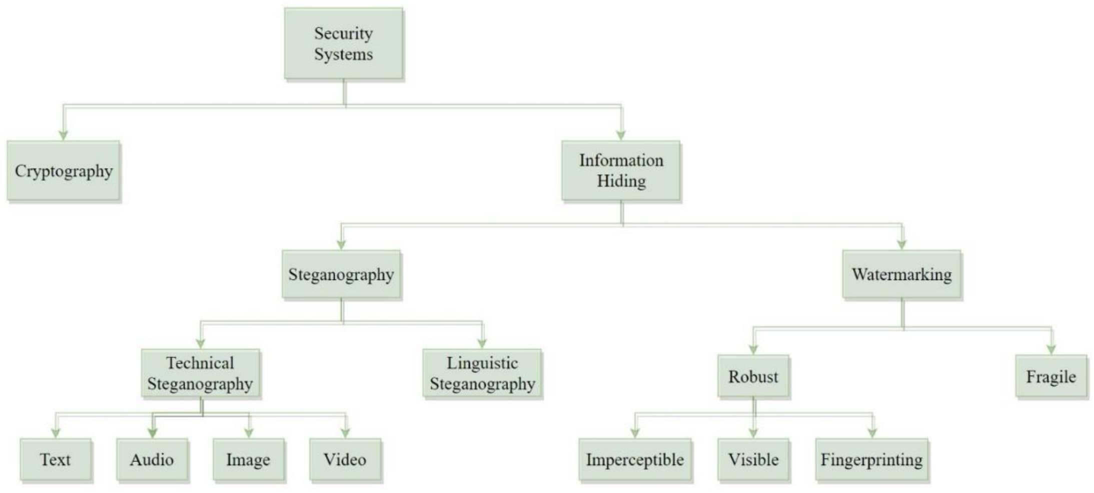


Figure 2.1 : Types of security systems.

Image steganography is the transmission of the data to be transmitted by embedding it in an image. This image, which is irrelevant to the message, is referred to as a cover image.

The final version of this image, in which the message is encoded, is the stego-image. As can be seen in Figure 2.2 and Figure 2.3 below, the difference between stego-image and cover image cannot be distinguished.

Figure 2.2 : Peppers cover image [11].

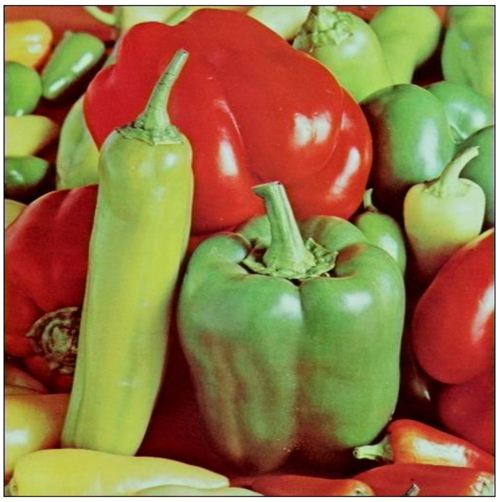


Figure 2.3 : Peppers stego-image.

This encoding process is done as follows: The pixels of the cover image are accessed and the bits of the message that are in binary form or converted to binary form are replaced one by one with the bits of the cover image's pixels. The less significant bits of the pixels are used during this process. Since the changes to be made on the pixel will change the tone of the pixel, making these changes in less significant bits will prevent the human eye from understanding the difference between the picture with embedded information (stego-image) and the original picture (cover image). In order to ensure the security of information transmission, it is preferable to choose the message size to be encoded so that it does not fill all the pixels of the picture. Thanks to the ability to randomly access the pixels of an image, the message can be placed on randomly selected pixels to increase security. This is one of the most important advantages of using this image data type as a cover. In addition, a more secure communication environment can be created by using steganography and cryptography sciences together. Some image steganography techniques and methods as common are shown in Figure 2.4 below [6].

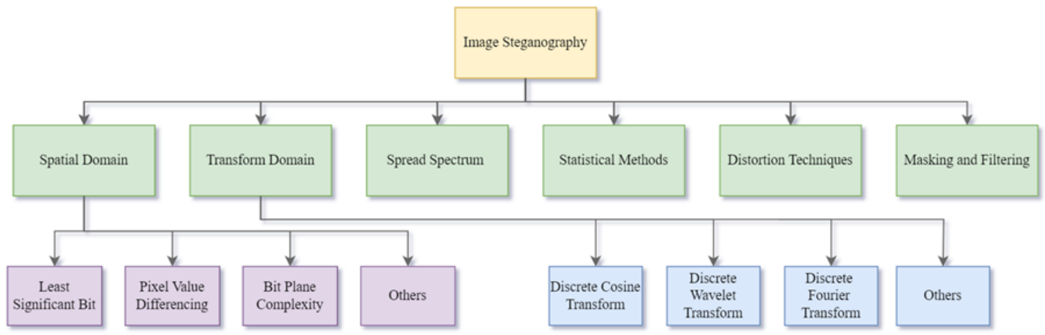


Figure 2.4 : Image steganography techniques and methods.

2.3 RISC-V Processor

RISC-V is a RISC-based instruction set architecture (ISA), which determines the connection between software and hardware, and the most significant feature of this ISA is being open source. Unlike the other ISA designs, it was developed by UC Berkeley, is freely available to everyone, and the fact that the core developed by providing the features in the guide or optionally extended ISAs can be shared on the official site of RISC-V, allows others to benefit from these developments [12]. In this way, many varieties are formed with the contribution of many people, and several cores and SoCs can be found on that site. Some core examples listed on RISC-V's site are shown in Table 2.1 [13]. Ibex core created by lowRISC was used in this project.

Table 2.1 : Table with RISC-V examples.

Name	Supplier	Capability	User spec.	Primary Language
RV32EC_P2	IQonIC Works	RV32	RV32E[M]C/RV32I[M]C	SystemVerilog
RV32IC_P5	IQonIC Works	RV32	RV32I[M][N][A]C	SystemVerilog
rocket	SiFive, UCB Bar	RV32	2.3-draft	Chisel

Table 2.1 (continued) : Table with RISC-V examples.

Name	Supplier	Capability	User spec.	Primary Language
Ibex (formerly Zero-riscy)	lowRISC	RV32	RV32I[M]C/RV32E[M]C	SystemVerilog
Hornet	Yavuz Selim Tozlu	RV32	RV32IM	Verilog

2.4 Xilinx Vivado Environment

Vivado produced by Xilinx is a virtual design and testing environment that allows to make designs using hardware description languages (HDLs) and perform synthesis, implementation and different types of simulations of these designs. In Figure 2.5, the interface of the Vivado environment for a sample project is shown.

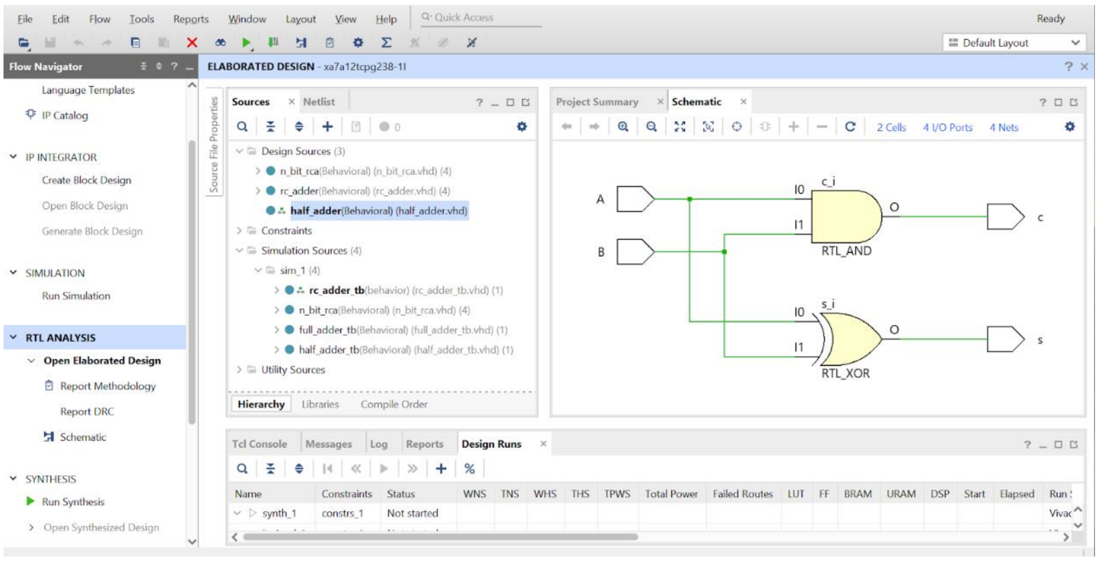


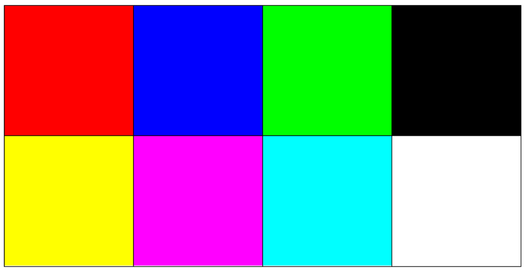
Figure 2.5 : Xilinx Vivado environment.

The Vivado environment was used to check whether the hardware designs to be made in this project work correctly with some processes that are offered by Vivado, and also to modify the hardware design and load them onto the FPGA card after these processes.

3. ALGORITHMS

3.1 LSB Algorithm

The LSB algorithm is one of the most basic image steganography methods. In this method, the least significant bit in an image is changed as part of the secret message, and, due to the fact that the change in the least significant bit will cause a change of only 1 ton in the pixel, it cannot be distinguished by the human eye. The RGB image has 3 layers, and if only LSB is used for encoding, the maximum message size that can be encoded in bits is $MxNx3$ for a colored image. If this image was grayscale, this time, a maximum of MxN bits of messages could be encoded. Since a grayscale image consists of a single layer, more information can be embedded in an RGB image than a grayscale image. The process of encrypting a message for a colored image is explained below, and the original image that is used as the cover image is shown in Figure 3.1.



1111 1111 0000 0000 0000 0000	0000 0000 0000 0000 1111 1111	0000 0000 1111 1111 0000 0000	0000 0000 0000 0000 0000 0000
1111 1111 1111 1111 0000 0000	1111 1111 0000 0000 1111 1111	0000 0000 1111 1111 1111 1111	1111 1111 1111 1111 1111 1111

Figure 3.1 : Cover image and its binary values.

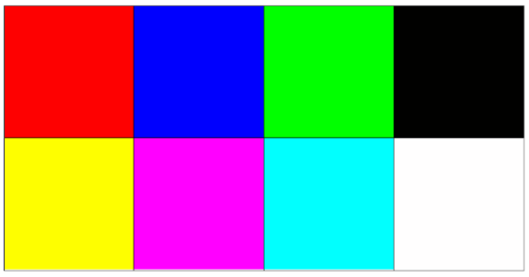
The message will be embedded into the least significant bits of the cover image. There are 8 pixels for this image that is the size of 2x4. In addition, one bit of the message will be hidden in every pixel, owing to the fact that this picture is colored, 3 bits of message can be encoded in each pixel. As a result, a maximum 24-bit message can be encrypted in this image.

For instance, it is assumed that the message to be transmitted is "cat", and considering that each character corresponds to 8 bits, it can be concluded that there is a message with a total length of 24 bits. The binary form of this message is shown in Figure 3.2.

c	a	t
0110 0011	0110 0001	0111 0100

Figure 3.2 : Message in binary form.

The bits of this binary message are encoded by circulating the image pixels one by one, and the stego-image in which the information is embedded is as shown in the image below as Figure 3.3.



1111 1110 0000 0001 0000 0001	0000 0000 0000 0000 1111 1110	0000 0001 1111 1111 0000 0000	0000 0001 0000 0001 0000 0000
1111 1110 1111 1110 0000 0000	1111 1111 0000 0000 1111 1111	0000 0001 1111 1111 1111 1110	1111 1111 1111 1110 1111 1110

Figure 3.3 : Stego-image and its binary values.

Although there is no difference when looking at the images, it can be seen that the stego-image is different from the original with the changes in the pixel values. In this way, a message to be transmitted can be hidden in inconspicuous data and transmitted securely.

3.2 LFSR Algorithm

3.2.1 Motivation for using the LFSR algorithm

As is known, Least Significant Bit Steganography is the most common method to transmit a secret message among other steganography methods. However, due to its simple structure, it is not secure enough for today's world. Therefore, Linear Feedback Shift Register (LFSR) has been used to generate a sequence of pseudo-numbers for selecting random image pixels. That way, the secret message bits were embedded in random cover image pixels to increase the undetectability of the communication instead of being embedded in an order.

3.2.2 Mathematics behind the LFSR algorithm

LFSR has been used in various engineering applications that have a need for creating a random process. It basically allows one to create a pseudo-random binary string of bits while making use of the seed which is a random string. Exact string of bits in the exact order can be obtained on the receiver side if the sender provides the seed value, which makes it pseudo-random. Although this string is generated based on an algorithm, it is unpredictable for someone who does not have the initial seed value [7].

The working principle of any LFSR device is based on the primitive polynomials. To understand the primitive polynomials, one needs to understand the irreducible polynomials which cannot be factored into the product of two polynomials that are in the same field and non-constant. Then, the definition of a primitive polynomial can be given as a polynomial C(x) with degree of n in $\mathbb{F}2[x]$ which is irreducible and divides $x^{2^n-1}-1$.

These polynomials are special in a way that using them while deciding the tap values allows the device to generate 2^n-1 different numbers where n is the number of bits in the seed. Most of the time, there are more than 1 primitive polynomial for a degree.

To give an example, while generating the elements of a binary extension field GF(2m) where m = 8, all the following polynomials can be used [14]:

$$p(x) = x^8 + x^6 + x^5 + x^3 + 1 \tag{3.1}$$

$$p(x) = x^8 + x^6 + x^5 + x^2 + 1 \tag{3.2}$$

$$p(x) = x^8 + x^6 + x^5 + x^1 + 1 \tag{3.3}$$

$$p(x) = x^8 + x^6 + x^3 + x^1 + 1 \tag{3.4}$$

$$p(x) = x^8 + x^6 + x^4 + x^3 + x^2 + x^1 + 1 \tag{3.5}$$

To explain the structure in more detail, the tap values are passed through the XOR operation among themselves to determine the feedback value. These tap values are selected from the location which is equal to the powers of the x variables in the primitive polynomial.

To illustrate, let’s consider the LFSR algorithm in Figure 3.4 that uses the polynomial given in equation 3.1 which is a primitive polynomial for a degree of 8.

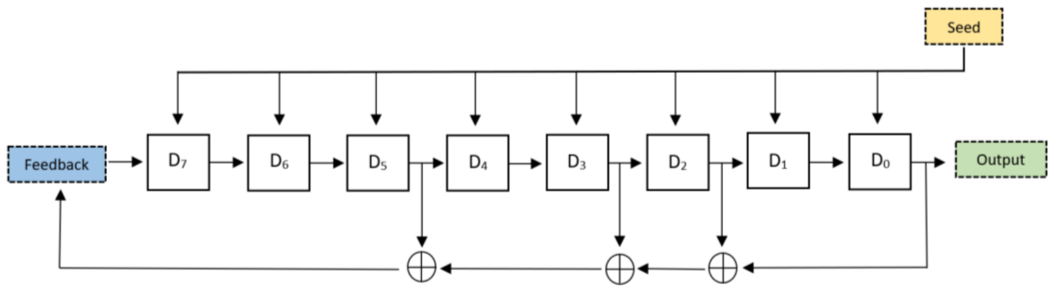


Figure 3.4 : Representation of LFSR algorithm using $p(x) = x^8 + x^6 + x^5 + x^3 + 1$

As seen from the figure, a random seed value which is also known by the receiver is given to the D flip-flops. Then, as the primitive polynomial suggests, values in 8th tab (D_0), 6th tab (D_2), 5th tab (D_3), and 3rd tab (D_5) is gone through the XOR operation and stored in the feedback register to be given to the most significant bit which is the value held by D_7. Then the bits are shifted to the right and the value in the feedback register is written to the D_7. As a side note, "+1" in each primitive polynomial represents the feedback.

3.3 Frequency Detection Based Algorithm

3.3.1 Motivation for using a frequency detection based algorithm

Even though the LSB algorithm that uses a Linear Feedback Shift Register for its pixel selection process is a secure and undetectable algorithm, there is still room for improvement. One of the ways to achieve this improvement is to embed not 1 but 2 secret message bits on the high-frequency pixels. Since these pixels correspond to sharp changes, or simply edges, in an image, it is harder for the human eye to identify any color or tone change in them. These high-frequency regions can be found using edge detection algorithms such as gradient operators (Sobel, Roberts, Prewitt, Frei-Chen), Canny edge detection and Haar based edge detection [8]. Among them, Sobel edge detection method was selected as the algorithm for finding the high-frequency pixels in an image.

3.3.2 Mathematics behind the selected frequency detection based algorithm

Sobel edge detection method utilizes the gradient (first derivative) of a grayscale image to detect the sharp changes. It identifies the image point as an edge if the calculated gradient is above the given threshold value.

The Sobel operator convolves the image with kernels that are 3x3 to find the matrices G_x and G_y which contain horizontal and vertical changes in the image respectively.

$$G_x = \begin{pmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{pmatrix} * \text{Img Mat} \ , \ G_y = \begin{pmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{pmatrix} * \text{Img Mat} \qquad (3.6)$$

These gradient approximations are used for obtaining the final gradient magnitude given below.

$$|G| = \sqrt{{G_x}^2 + {G_y}^2} \qquad (3.7)$$

To make the computation faster, this value can also be calculated with the equation given below.

$$|G| = |G_x| + |G_y| \qquad (3.8)$$

As mentioned earlier, this value is compared with the given threshold, and it is decided whether the selected pixel is an edge or not. Briefly, the Sobel operator approximates the absolute image gradient by making use of the intensity values in a 3×3 region around every pixel of the image. Due to its practicality, it has been used for various hardware and software applications. Grayscale cameraman image with 512x512 resolution and its corresponding absolute gradient magnitude from Sobel operator that has 0.07 threshold value can be seen in Figure 3.5.

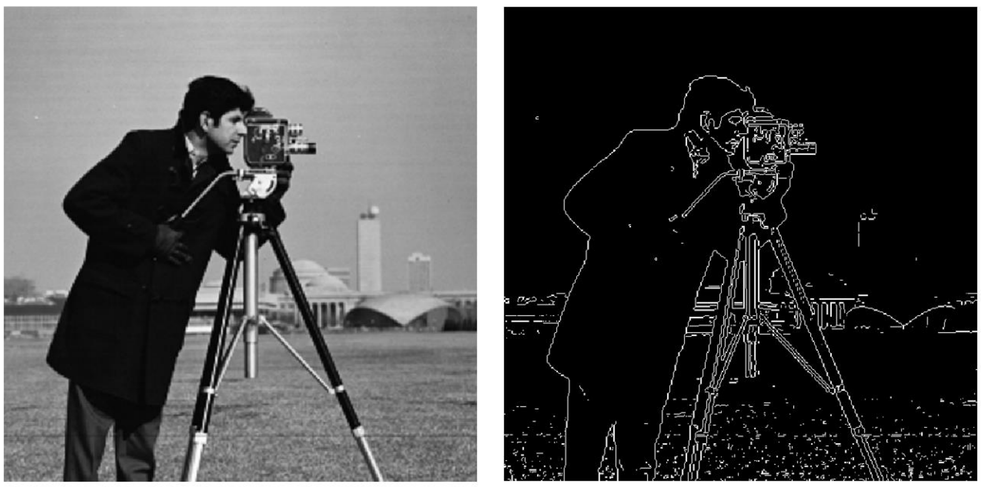


Figure 3.5 : Grayscale image and its gradient magnitude from Sobel operator [15].

3.4 MATLAB Simulation of the Algorithms

3.4.1 LSB based image steganography

In the first step, a MATLAB program is designed and simulated that embeds the secret message in the least significant bits of the pixels of a selected image, starting from the first pixel in order. Grayscale images are used during the process; therefore, each pixel of the image consists of a single palette and takes values between 0-255. Also, images in "bitmap" format are used to avoid jamming in the image and therefore not losing pixel values. After the image is read by the program, its size is determined in order to navigate the pixels of the image matrix. At the same time, the string type secret message was converted into binary type using the ASCII code equivalents of the characters and its length in this form was determined. The program performs the embedding operation in the number of the length of the secret message. The first pixel value of the image is read and the least significant bit is replaced by the first bit of the secret message. Afterwards, the second pixel of the image is passed and the second bit of the secret message is added to this pixel. The process continues in that way until the last bit of the secret message is embedded in the image.

In order to replace the least significant bits with the bits of the secret message, the following line of code is used in a loop for navigating the bits of the image:

```
        "stegoImage(i,j) = candidateImage(i,j) -
mod(candidateImage(i,j),2) + Message(count,1);"
```

Here, “stegoImage” refers to the stego-image matrix to be created, “candidateImage” refers to the original image read by the program, and “Message” refers to the binary form of the secret message. The value in the pixel is modulated by 2 to extract the last bit of the original pixel. Thus, if the value of the last bit is 1, the number 1, and if it is 0, the number 0 is subtracted from the value in the pixel. Then, the value of the relevant bit of the secret message is added.

The original of the image without a message embedded in it and the stego-image created after the secret message is embedded can be seen in Figures 3.6 and 3.7. As with the purpose of image steganography, the message embedded in the image cannot be perceived by the human eye, therefore, the pixels of the stego-image should be checked to see that the embedding process is performed correctly. Since it may take time to check the pixel values one by one, a decryption code has been written to obtain the message embedded in the stego-image.

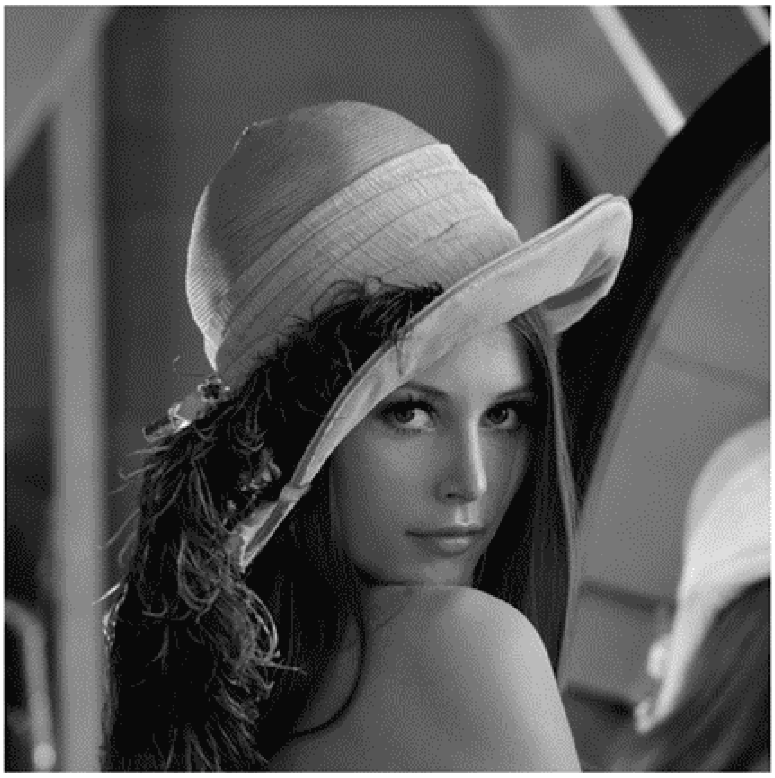


Figure 3.6 : Lena original image [16].

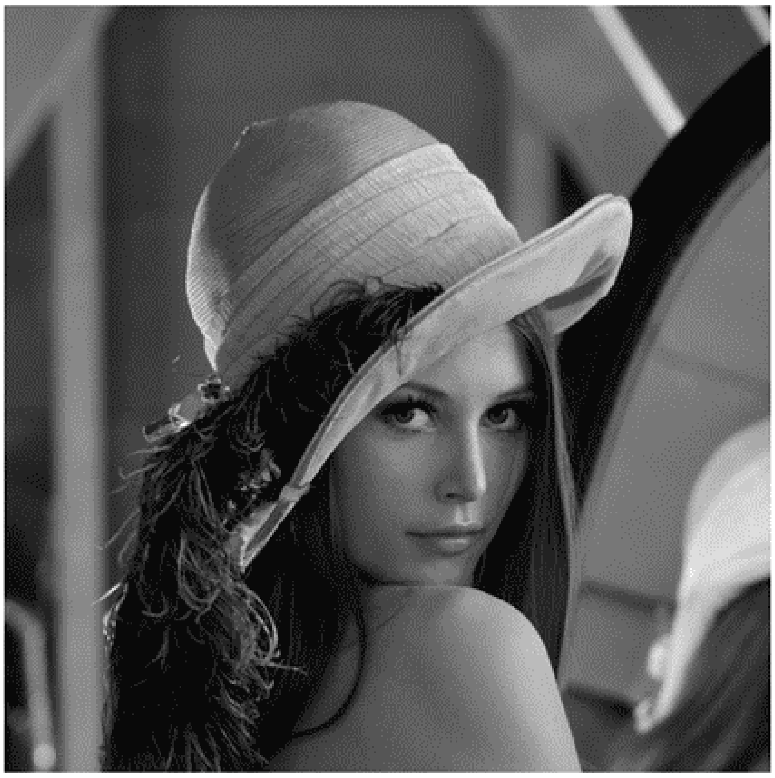


Figure 3.7 : Lena stego-image.

The secret message used in the simulation code is a text in the form of “This is the secret message.”. In order to obtain the secret message, the least significant bit of the value in each pixel was taken by navigating through the stego-image pixels in order. After the process is completed and all the bits are stored in an array, the bits are grouped as octets and converted into characters corresponding to the ASCII code of each group. Thus, the binary type message is converted to text and this text is saved in a file. The content of the text file that is the output of the decryption code is as follows:

“This is the secret message.”

It is seen that the secret message used during the embedding process is the same as the text obtained after decryption. Therefore, it can be said that the embedding and recovery of the message were successfully carried out.

3.4.2 Including LFSR into the algorithm

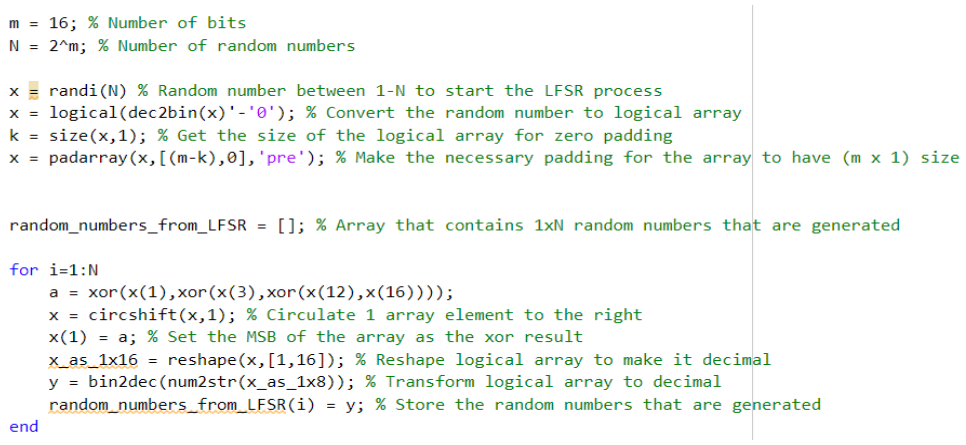
In the project, it is aimed to embed the bits of the message into the image randomly in order to reduce the accessibility of the secret message. With the LFSR algorithm that is used to realize this, a pseudo-random number sequence is produced in this range according to the number of pixels in the image. An image with a size of 256x256 was used during the simulation and the total number of pixels in this image is 65536. Therefore, using the LFSR algorithm, a sequence containing numbers between 1 and 65535 was created.

Since the number 65536 is equal to 216, 16 bits are required to write each number as binary. Numbers are defined as "logical array", since the bits of numbers will be processed with the XOR operator. A starting number must be given to the function for the number generation to begin. For this purpose, a random number in the range of 1 – 65536 was generated and assigned as the first element of the pseudo-random number array. The required number of 0 is added with padding so that the number can be a 16-bit logical array.

There are 14 of the 16th degree irreducible polynomials. One of these polynomials is chosen to implement the LFSR algorithm that generates 16-bit numbers. The relevant polynomial is as follows:

$$p(x) = x^{16} + x^{12} + x^{3} + x^{1} + 1 \qquad (3.9)$$

Due to the chosen polynomial, the 1st, 3rd, 12th, and 16th bits of the selected number, which is the starting number, are processed with the xor operator in the first step, and then this number is shifted to the right one by one with the right shift operator. The next element of the pseudo-random number sequence is produced by assigning the output of the xor operation to the most significant bit of the shifted number. This process is repeated until $2^{16} - 1$ different numbers are generated. At the same time, the new number generated at each stage is converted to decimal format and added to the number sequence. The relevant part of the MATLAB code, in which the pseudo-random number sequence is generated by using xor and right shift, is in Figure 3.8.



```
m = 16; % Number of bits
N = 2^m; % Number of random numbers

x = randi(N) % Random number between 1-N to start the LFSR process
x = logical(dec2bin(x)'-'0'); % Convert the random number to logical array
k = size(x,1); % Get the size of the logical array for zero padding
x = padarray(x,[(m-k),0],'pre'); % Make the necessary padding for the array to have (m x 1) size


random_numbers_from_LFSR = []; % Array that contains 1xN random numbers that are generated

for i=1:N
    a = xor(x(1),xor(x(3),xor(x(12),x(16))));
    x = circshift(x,1); % Circulate 1 array element to the right
    x(1) = a; % Set the MSB of the array as the xor result
    x_as_1x16 = reshape(x,[1,16]); % Reshape logical array to make it decimal
    y = bin2dec(num2str(x_as_1x8)); % Transform logical array to decimal
    random_numbers_from_LFSR(i) = y; % Store the random numbers that are generated
end
```

Figure 3.8 : MATLAB code for LFSR algorithm.

To ensure that the correct number of numbers is generated, a code has been added for the function that implements the LFSR algorithm, counting how many different numbers are generated. As a result, it was seen that 65535 different numbers, therefore all numbers between 1 and 65535 were produced.

The image steganography method was improved by using the designed 16-bit LFSR function in the LSB code. Thus, the secret message is not embedded in the bits of the image in order, but in the order of the numbers in the sequence produced by the LFSR function, and the bits of the message are embedded in a pseudo-random order. In order to add the LFSR algorithm to the LSB code, a change was made in the part where the pixels are selected. After determining the length of the secret message, the element of the random number sequence produced by LFSR was selected as much as the message bit length. To embed, it is necessary to find out which element of the image matrix the generated random numbers correspond to. The row and column of the bit to be selected are calculated using the image dimensions. Then, instead of the least significant bit of the selected matrix element, the relevant bit of the secret message is added. The original of the image without a message embedded in it and the stego-image created after the secret message is embedded, can be seen in Figure 3.9 and Figure 3.10.

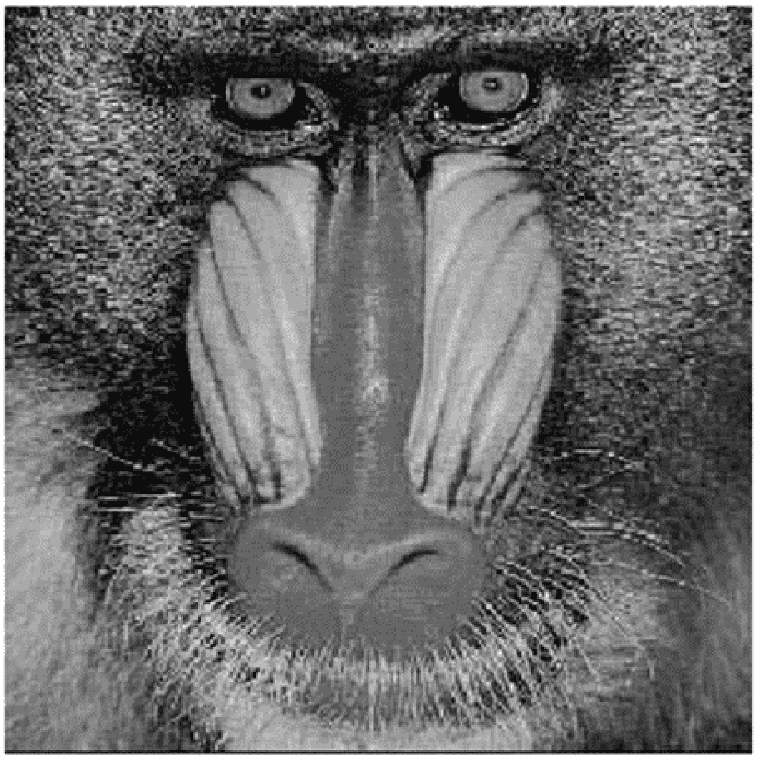


Figure 3.9 : Mandrill original image [17].

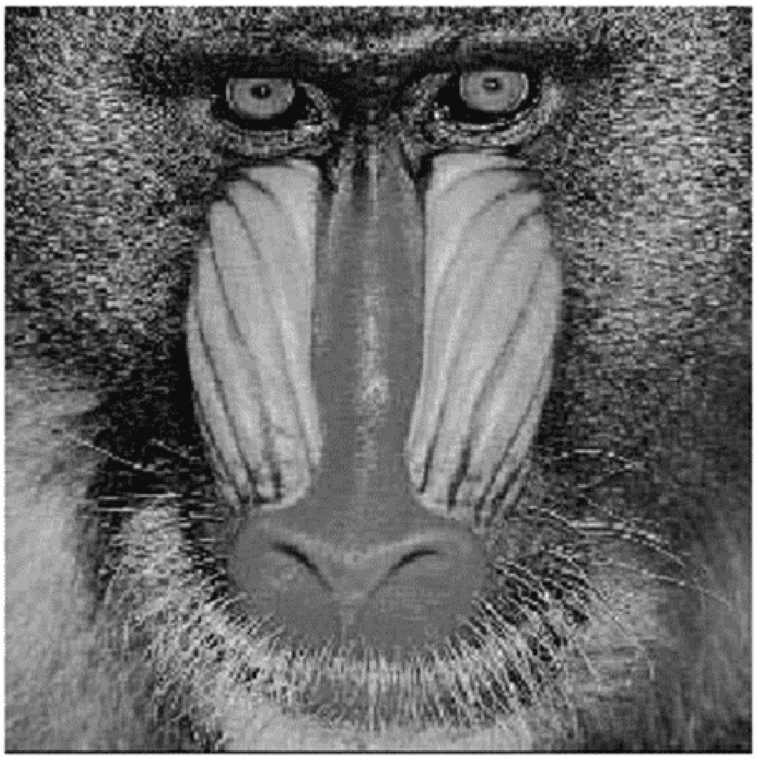


Figure 3.10 : Mandrill stego-image.

To check that the secret message is correctly embedded in the bits produced by LFSR, the method of retrieval of the message was used as before. The least significant bits were selected by navigating the bits in the generated random sequence in order, and then each 8 bits were combined and converted into the character corresponding to the ASCII code. "This is the secret message." was used as the secret message text during the simulation. The text in the output file obtained as a result of the decryption is same as the secret message. Thus, it has been verified that the code for embedding information in the least significant bits of the image using the LFSR algorithm works correctly.

3.4.3 Improving the algorithm with frequency detection

Previous MATLAB project that implements the LSB algorithm which uses a LFSR for its pixel selection process was improved in terms of pixel usage to allow longer secret messages to be transmitted using the same cover image. This was achieved by embedding 2 bits instead of 1 into the pixels that are also the edges of the cover image.

After reading the image, turning it into grayscale, and generating the random index numbers from the LFSR algorithm function, the grayscale image was exposed to the Sobel operator which has a threshold value of 0.07 to detect the edge regions in the cover image having a size of 256x256 using the code line seen below.

```
"BW = edge(candidateImage, 'Sobel', 0.07);"
```

Edge detected image after this operation can be seen in Figure 3.11.

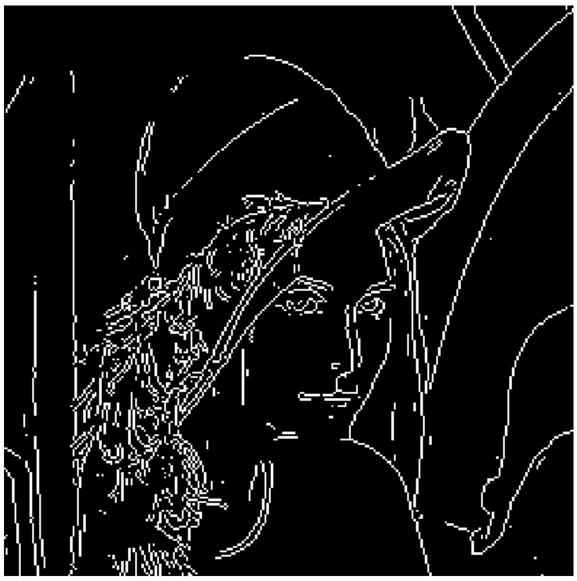
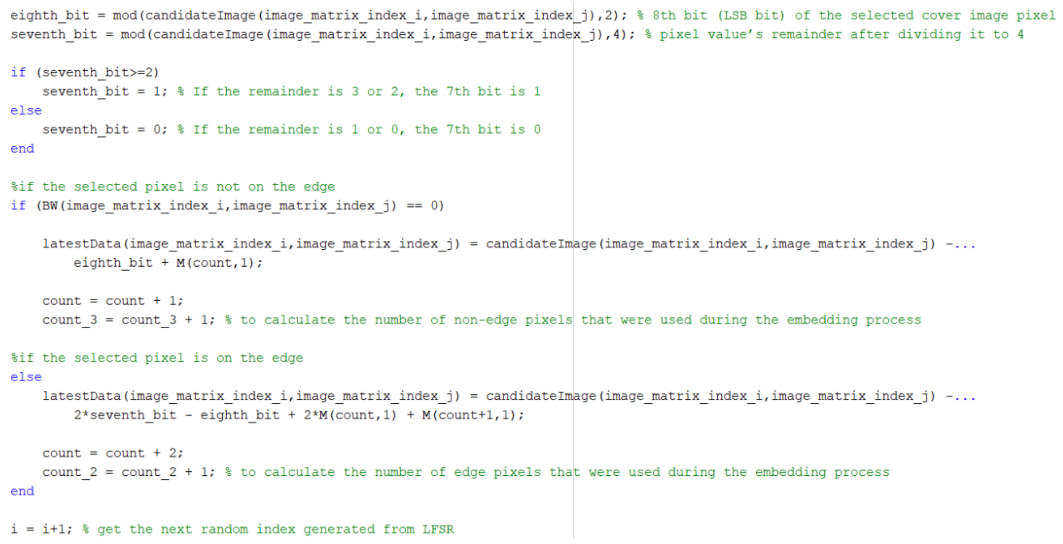


Figure 3.11 : Gradient magnitude of the cover image from Sobel operator.

Afterwards, counters for counting the number of edge and non-edge pixels that were used during the embedding process were defined to validate the code. Then, the pixel indexes generated from the LFSR algorithm were reshaped in order to access the pixels of the cover image as row and column.

Crucial part of this design was to access not only the least significant bit, which is the eighth bit of the selected pixel, but also to the seventh bit of it to swap it with the message bit if necessary. To obtain the seventh bit, pixel value’s remainder after dividing it to 4 is calculated and if it is equal to 3 or 2, it is set as 1. Otherwise, it is set as 0. Then swap between the image pixel bits and the secret message bits was done by first checking whether the selected pixel is an edge or not. The relevant part of the MATLAB code can be seen in Figure 3.12.



```
eighth_bit = mod(candidateImage(image_matrix_index_i,image_matrix_index_j),2); % 8th bit (LSB bit) of the selected cover image pixel
seventh_bit = mod(candidateImage(image_matrix_index_i,image_matrix_index_j),4); % pixel value’s remainder after dividing it to 4

if (seventh_bit>=2)
    seventh_bit = 1; % If the remainder is 3 or 2, the 7th bit is 1
else
    seventh_bit = 0; % If the remainder is 1 or 0, the 7th bit is 0
end

%if the selected pixel is not on the edge
if (BW(image_matrix_index_i,image_matrix_index_j) == 0)

    latestData(image_matrix_index_i,image_matrix_index_j) = candidateImage(image_matrix_index_i,image_matrix_index_j) -...
        eighth_bit + M(count,1);

    count = count + 1;
    count_3 = count_3 + 1; % to calculate the number of non-edge pixels that were used during the embedding process

%if the selected pixel is on the edge
else
    latestData(image_matrix_index_i,image_matrix_index_j) = candidateImage(image_matrix_index_i,image_matrix_index_j) -...
        2*seventh_bit - eighth_bit + 2*M(count,1) + M(count+1,1);

    count = count + 2;
    count_2 = count_2 + 1; % to calculate the number of edge pixels that were used during the embedding process
end

i = i+1; % get the next random index generated from LFSR
```

Figure 3.12 : MATLAB code for embedding process.

After the swap, the next random number generated from the LFSR algorithm was gotten by increasing the necessary index. The original cover image and the stego-image created after the secret message bits were embedded can be seen from Figure 3.13 and Figure 3.14.

Figure 3.13 : Lena original image.

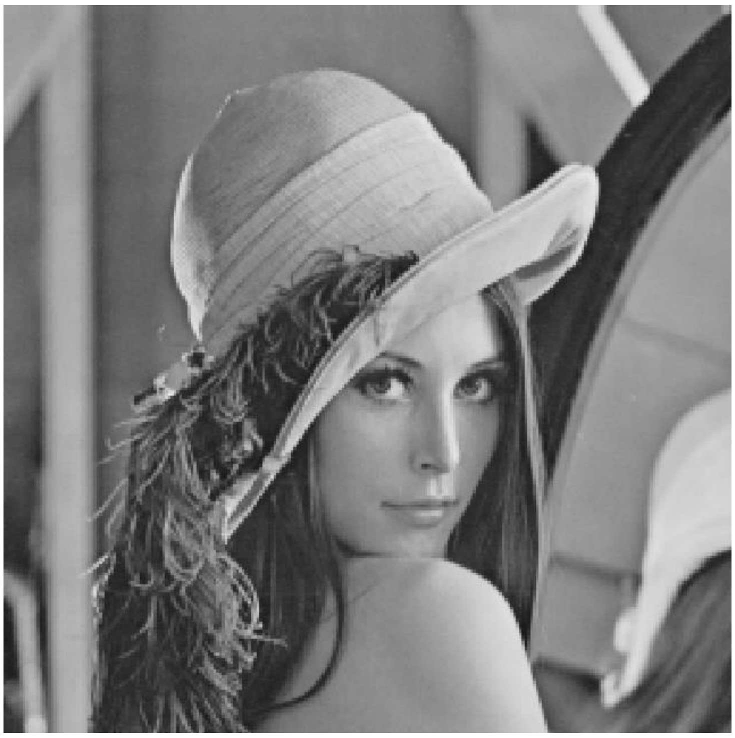


Figure 3.14 : Lena stego-image.

To make a decryption, or simply to obtain the secret message bits that were embedded, image pixel's eighth bit only or the seventh and eighth bits together were taken depending on the pixel being edge or not. After that, all the bits taken were stored in an array where they were grouped as octets afterwards. Similar to before, they were converted into characters corresponding to the ASCII code. This way, binary message were converted into a text seen below.

“This is the secret message.”

It shows that the secret message used during the embedding process matches with the text obtained after decryption. Thus, the system with LSB algorithm which uses a LFSR algorithm for its pixel selection and a Sobel operator to decide the number of bits to embed into the selected pixel is verified. Also, it is seen that after a few re-simulations, the number of edge pixels that were used during the embedding process were between 11-15 depending on the seed value in the LFSR for the secret message above which has 200 bits.

4. IBEX CORE AND WISHBONE PROTOCOL

4.1 Preparing the Work Environment

Necessary systems and applications should be installed to implement the RISC-V core, to perform various simulations, tests and applications regarding the project, and to obtain memory files. That's why the Linux operating system, Xilinx Vivado and RISC-V GNU toolchain were installed.

4.1.1 Installing Linux operating system

First of all, Linux Ubuntu 20.04.1 LTS, the operating system in which the necessary applications will be installed and used, has been installed. The reason for choosing Linux as the operating system is to avoid encountering errors due to installation and problems that may occur during operations such as open source core installation.

4.1.2 Installing Vivado Design Suite for Linux OS

For the installation of the Xilinx Vivado application, Vivado 2020.1 version was downloaded from the Xilinx site. After the download is complete, it should be ensured to go to the downloaded file location in the terminal and make the file executable. Afterwards, the file should be run and the Vivado installation page should be accessed. The codes required for these steps are as follows:

```
$ chmod +x Xilinx_Unified_2020.1_0602_1208_Lin64.bin
$ sudo ./Xilinx_Unified_2020.1_0602_1208_Lin64.bin
```

During the installation of Vivado, a warning was encountered that Vivado does not support versions above Ubuntu 18.6. That is why the version of the Ubuntu operating system is shown as Ubuntu 18.04.4 by using the OS faking method. With this method, Vivado was installed on the current operating system without any problems. After selecting the required features and completing the installation of the Xilinx Vivado program, what needs to be done to run the program is to go to the file location where the program is located and run it with the code below.

```
$ ./vivado
```

4.1.3 Installing RISC-V GNU toolchain

It is necessary to install the RISC-V GNU compiler to create ".mem" memory files to be used in applications in the Vivado environment by compiling C codes.

Firstly, the packages required to use the Toolchain should be downloaded with the following command to be run on the Ubuntu terminal:

```
$ sudo apt-get install autoconf automake autotools-dev curl python3 libmpc-dev libmpfr-dev libgmp-dev gawk build-essential bison flex texinfo gperf libtool patchutils bc zlib1g-dev libexpat-dev
```

Git should be installed with the following command in order to use GitHub in Ubuntu and access the Toolchain repository:

```
$ sudo apt-get install git
```

Then the following command is typed into the terminal to download the repository to the Ubuntu operating system [18]:

```
$ git clone https://github.com/riscv/riscv-gnu-toolchain
```

After the repository download is complete, the following commands should be run in order by going to the file location of the downloaded riscv-gnu-toolchain folder:

```
$ ./configure --prefix=/opt/riscv --with-arch=rv32imc
$ sudo make
```

Depending on the computer's performance and internet speed, this process may take 20-30 minutes. When the process is complete, the toolchain is loaded and ready to use.

4.2 Implementation of Ibex Core

4.2.1 Ibex core overview

RISC-V has numerous cores and SoC’s that are available to use which can be found on the official website of RISC-V [12]. However, because the Ibex core is one of the most verified cores among the others and has been used in the projects and researches prior to our work, it was selected as the RISC-V core to be implemented. It is a 32-bit

CPU core whose primary language is SystemVerilog. Figure 4.1 illustrates the architecture of an Ibex core and its connections with the data memory (physical RAM).

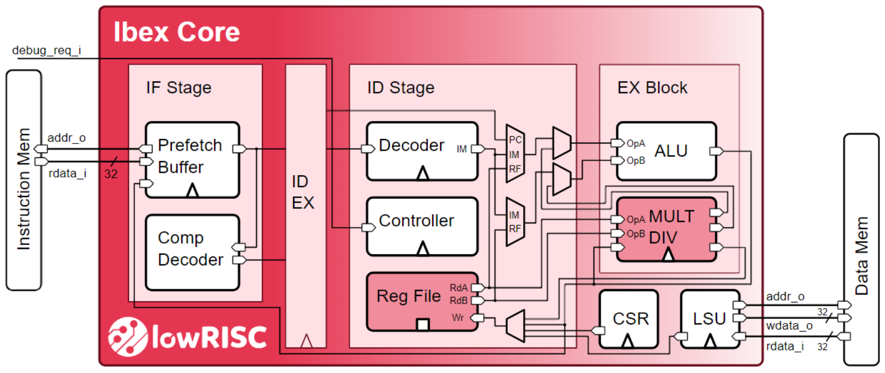


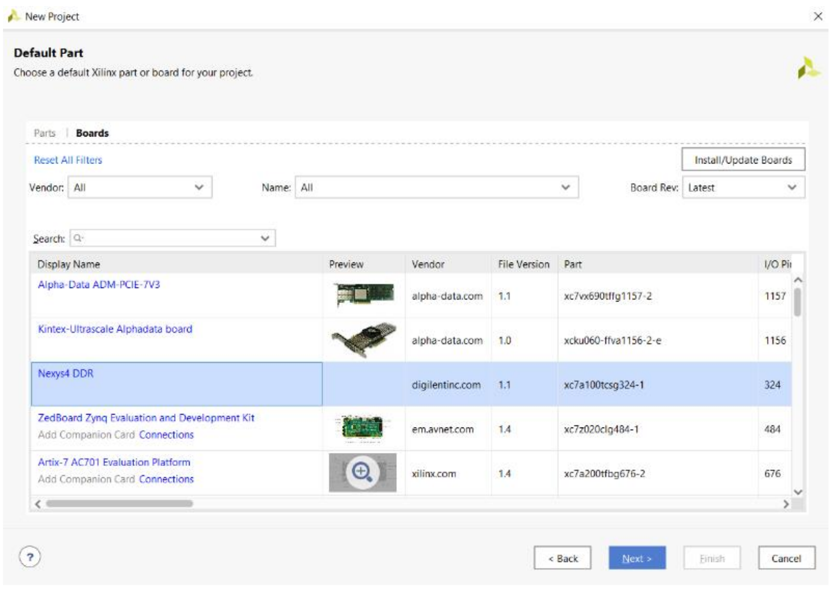
Figure 4.1 : Block diagram for Ibex core [19].

Ibex core will be implemented using the IowRISC's GitHub repository [19]. According to the requirements given in Ibex RISC-V Core SoC Example, fusesoc and srecord were installed using the commands below:

```
$ sudo pip install fusesoc

$ sudo apt-get install srecord
```

4.2.2 Creating a Vivado project

To implement the Ibex core on the Nexys4 DDR Board, a Vivado project was opened to create the bitstream file to program the board. After the directory for the project was given, the board selection was done as Nexys4 DDR [20] as shown in Figure 4.2.



Default Part

Choose a default Xilinx part or board for your project.

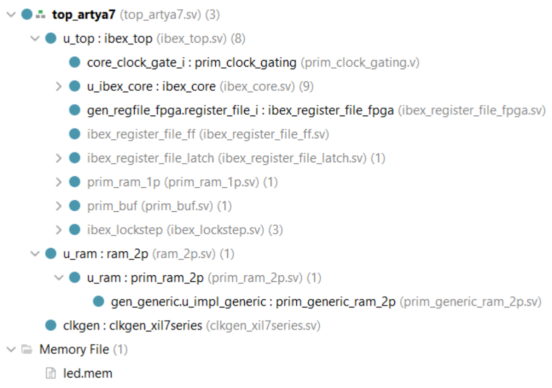
Vendor: All Name: All Board Rev: Latest

Search:

Display Name	Preview	Vendor	File Version	Part	I/O Pi
Alpha-Data ADM-PCIE-7V3		alpha-data.com	1.1	xc7vx690tffg1157-2	1157
Kintex-Ultrascale Alphadata board		alpha-data.com	1.0	xcku060-ffva1156-2-e	1156
Nexys4 DDR		digilentinc.com	1.1	xc7a100tcsg324-1	324
ZedBoard Zynq Evaluation and Development Kit Add Companion Card Connections		em.avnet.com	1.4	xc7z020clg484-1	484
Artix-7 AC701 Evaluation Platform Add Companion Card Connections		xilinx.com	1.4	xc7a200tfbg676-2	676

Figure 4.2 : Board selection for the project.

Then, the example from "/examples/fpga/artya7/rtl/top_artya7.sv" directory will be implemented on Vivado environment to validate the Ibex core. First, the top module "top_artya7.sv" was added as a design and simulation source. Then, every required submodule was added under the top module and the hierarchy shown in Figure 4.3 was obtained.

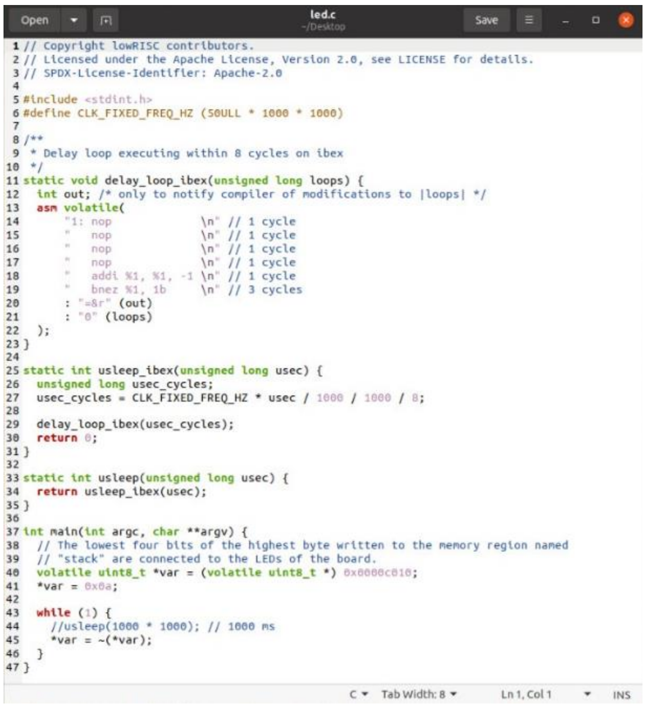


```
top_artya7 (top_artya7.sv) (3)
  u_top : ibex_top (ibex_top.sv) (8)
    core_clock_gate_i : prim_clock_gating (prim_clock_gating.v)
    u_ibex_core : ibex_core (ibex_core.sv) (9)
    gen_regfile_fpga.register_file_i : ibex_register_file_fpga (ibex_register_file_fpga.sv)
    ibex_register_file_ff (ibex_register_file_ff.sv)
    ibex_register_file_latch (ibex_register_file_latch.sv) (1)
    prim_ram_1p (prim_ram_1p.sv) (1)
    prim_buf (prim_buf.sv) (1)
    ibex_lockstep (ibex_lockstep.sv) (3)
  u_ram : ram_2p (ram_2p.sv) (1)
    u_ram : prim_ram_2p (prim_ram_2p.sv) (1)
      gen_generic.u_impl_generic : prim_generic_ram_2p (prim_generic_ram_2p.sv)
  clkgen : clkgen_xil7series (clkgen_xil7series.sv)
Memory File (1)
  led.mem
```

Figure 4.3 : Hierarchy of the Ibex core project.

4.2.3 RAM initialization

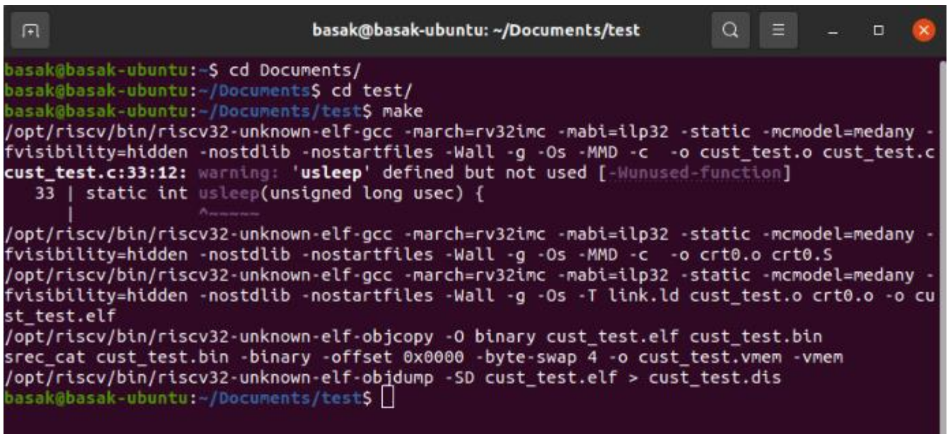
Dual port RAM was chosen rather than a single port RAM to increase the speed. To initialize the dual port memory with a memory file, installed RISC-V GNU toolchain will be utilized. First of all, C code from the "ibex/examples/sw/led" directory shown in Figure 4.4 was used.



```c
// Copyright lowRISC contributors.
// Licensed under the Apache License, Version 2.0, see LICENSE for details.
// SPDX-License-Identifier: Apache-2.0

#include <stdint.h>
#define CLK_FIXED_FREQ_HZ (50ULL * 1000 * 1000)

/**
 * Delay loop executing within 8 cycles on ibex
 */
static void delay_loop_ibex(unsigned long loops) {
  int out; /* only to notify compiler of modifications to |loops| */
  asm volatile(
      "1: nop             \n" // 1 cycle
      "   nop             \n" // 1 cycle
      "   nop             \n" // 1 cycle
      "   nop             \n" // 1 cycle
      "   addi %1, %1, -1 \n" // 1 cycle
      "   bnez %1, 1b     \n" // 3 cycles
      : "=&r" (out)
      : "0" (loops)
  );
}

static int usleep_ibex(unsigned long usec) {
  unsigned long usec_cycles;
  usec_cycles = CLK_FIXED_FREQ_HZ * usec / 1000 / 1000 / 8;

  delay_loop_ibex(usec_cycles);
  return 0;
}

static int usleep(unsigned long usec) {
  return usleep_ibex(usec);
}

int main(int argc, char **argv) {
  // The lowest four bits of the highest byte written to the memory region named
  // "stack" are connected to the LEDs of the board.
  volatile uint8_t *var = (volatile uint8_t *) 0x0000c010;
  *var = 0x0a;

  while (1) {
    //usleep(1000 * 1000); // 1000 ms
    *var = ~(*var);
  }
}
```

Figure 4.4 : Compiled C code for testing the Ibex core.

The C code which blinks the LEDs on the board is compiled using the Ubuntu terminal shown in Figure 4.5 below.



```
basak@basak-ubuntu: ~/Documents/test
basak@basak-ubuntu:~$ cd Documents/
basak@basak-ubuntu:~/Documents$ cd test/
basak@basak-ubuntu:~/Documents/test$ make
/opt/riscv/bin/riscv32-unknown-elf-gcc -march=rv32imc -mabi=ilp32 -static -mcmodel=medany -
fvisibility=hidden -nostdlib -nostartfiles -Wall -g -Os -MMD -c  -o cust_test.o cust_test.c
cust_test.c:33:12: warning: 'usleep' defined but not used [-Wunused-function]
   33 | static int usleep(unsigned long usec) {
      |            ^~~~~~
/opt/riscv/bin/riscv32-unknown-elf-gcc -march=rv32imc -mabi=ilp32 -static -mcmodel=medany -
fvisibility=hidden -nostdlib -nostartfiles -Wall -g -Os -MMD -c  -o crt0.o crt0.S
/opt/riscv/bin/riscv32-unknown-elf-gcc -march=rv32imc -mabi=ilp32 -static -mcmodel=medany -
fvisibility=hidden -nostdlib -nostartfiles -Wall -g -Os -T link.ld cust_test.o crt0.o -o cu
st_test.elf
/opt/riscv/bin/riscv32-unknown-elf-objcopy -O binary cust_test.elf cust_test.bin
srec_cat cust_test.bin -binary -offset 0x0000 -byte-swap 4 -o cust_test.vmem -vmem
/opt/riscv/bin/riscv32-unknown-elf-objdump -SD cust_test.elf > cust_test.dis
basak@basak-ubuntu:~/Documents/test$ 
```

Figure 4.5 : Terminal output after the compile operation.

Obtained .vmem file has been converted to .mem file which is compatible with Vivado. After that, memory was initialized in the top module file as follows:

```
parameter   SRAMInitFile = "led.mem";
```

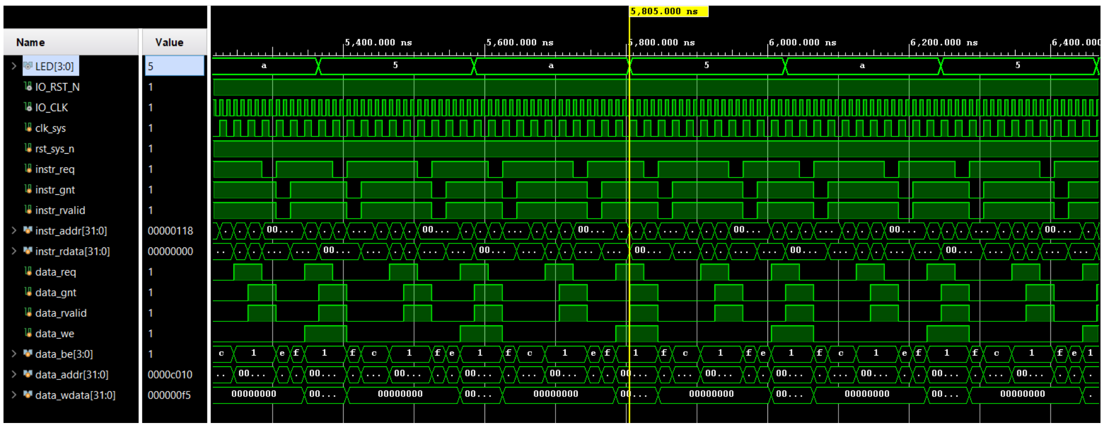
In the submodules, this memory file is transferred using the code below:

```
.MemInitFile(SRAMInitFile);
```

By changing the initialized memory file on the SystemVerilog code, the Ibex core can be programmed to execute any program compiled by the user.

4.2.4 Vivado simulation

After setting up everything, a simple testbench was written to simulate the core. Timing diagram results for that simulation can be seen in Figure 4.6.



```
Name              Value
LED[3:0]          5
IO_RST_N          1
IO_CLK            1
clk_sys           1
rst_sys_n         1
instr_req         1
instr_gnt         1
instr_rvalid      1
instr_addr[31:0]  00000118
instr_rdata[31:0] 00000000
data_req          1
data_gnt          1
data_rvalid       1
data_we           1
data_be[3:0]      1
data_addr[31:0]   0000c010
data_wdata[31:0]  000000f5
```

Figure 4.6 : Timing diagram results in Vivado.

By looking at the results, it is seen that the correct values were assigned to the LED output, hence the Ibex core is working as expected.

4.2.5 Implementation on the FPGA

After seeing that the simulation is done correctly, synthesis and implementation were run and to generate a bitstream for the Nexys4 DDR Board, the constraint file was downloaded [21]. Later, used pins in the .xdc file were uncommented and renamed. Finally, bitstream file that has a .bit extension was generated and uploaded to the board. The results can be seen from the LEDs that are on Nexys4 DDR board in Figure 4.7 and Figure 4.8.

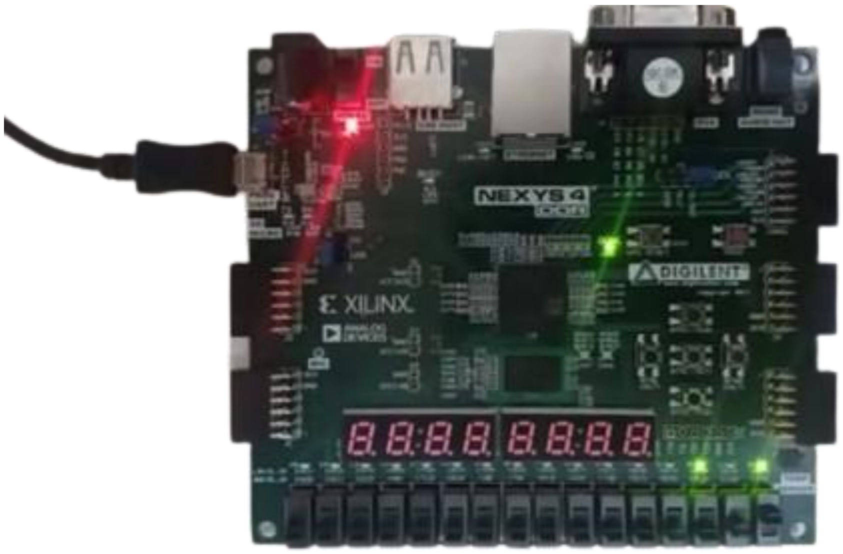


Figure 4.7 : 0x5 value on the LEDs.

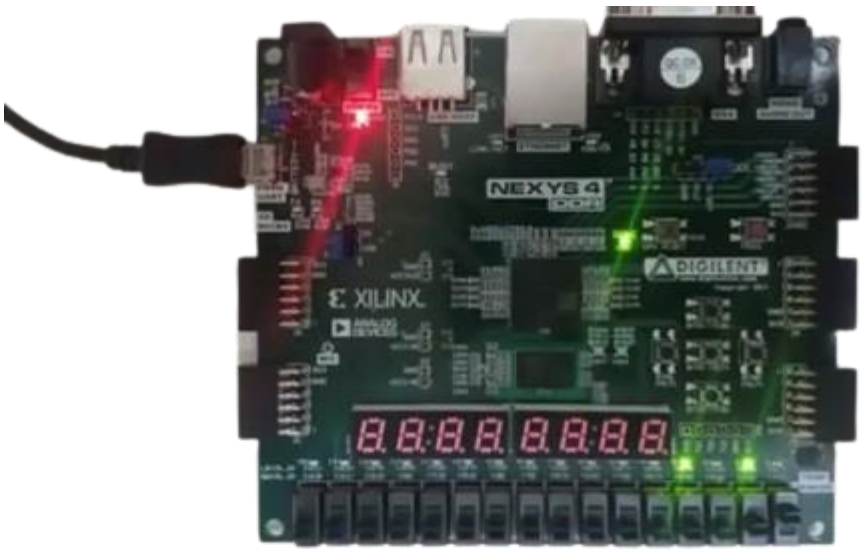


Figure 4.8 : 0xa value on the LEDs.

4.3 Integrating Wishbone Protocol with Ibex Core

4.3.1 Wishbone protocol

The system created in the project needs to communicate with external units such as the RAM and the VGA. Ibex core is capable of communicating with instruction memory and data memory, but does not have an interface to communicate with peripherals. To achieve this, a computer bus protocol must be implemented. That is why, Wishbone, an open-source computer bus that allows different parts in a circuit to communicate with each other, has been added to the project. The Ibex core version, which was tested before, was also connected to other units via Wishbone bus. Wishbone signals and their connections are as in Figure 4.9.

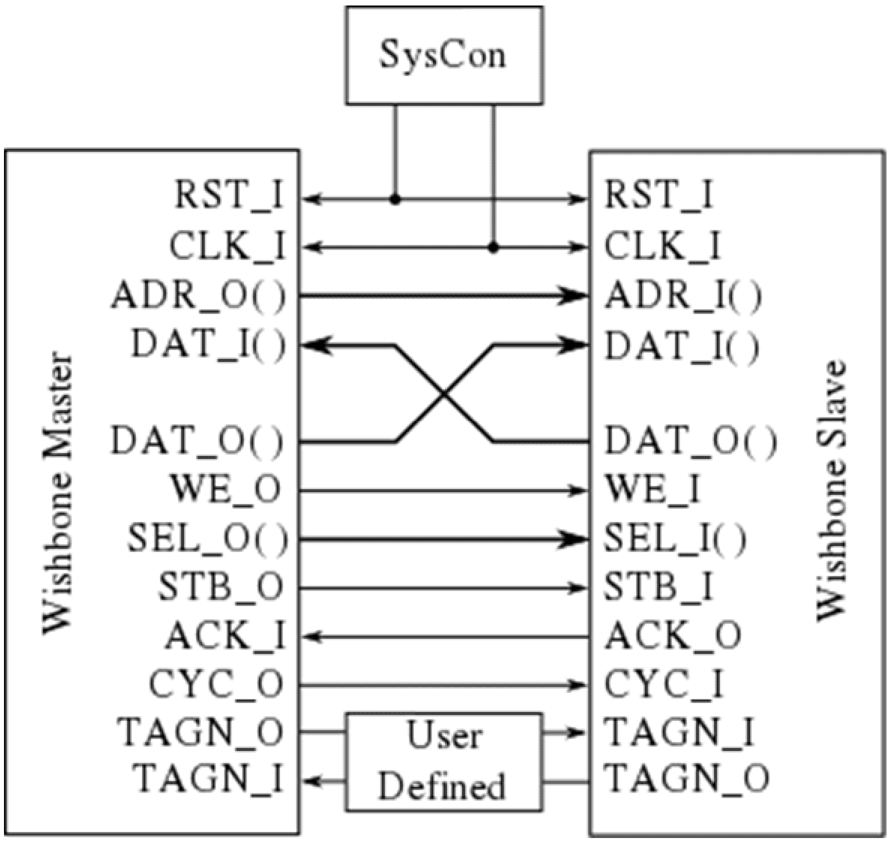


Figure 4.9 : Master and slave Wishbone's interfaces [22].

A shared bus named interconnect is the unit that allows master and slaves to communicate. Slaves are selected according to the addresses in the masters, whose priorities are determined by the arbiter. Share bus connection is shown in Figure 4.10 [10].

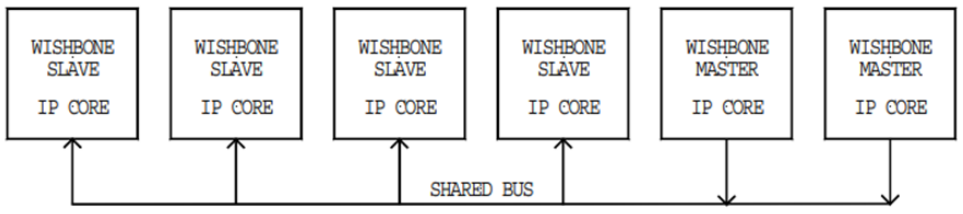


Figure 4.10 : Shared bus interconnection [10].

A handshaking protocol is used between the Master and Slave interfaces. Using this protocol allows controlling the data transfer rate. An example of the implementation of the handshaking protocol is shown in Figure 4.11 [10].

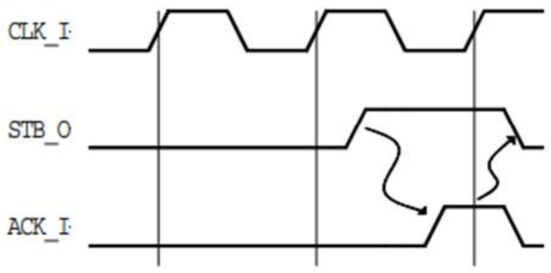


Figure 4.11 : Local bus handshaking protocol [10].

4.3.2 Adding an Ibex core into the Wishbone and existing peripherals

Ibex core version, which has been tested and checked to be working before, has been added to the Vivado project, which includes five masters and slaves. The schematic of the five masters and slaves is shown in Figure 4.12.

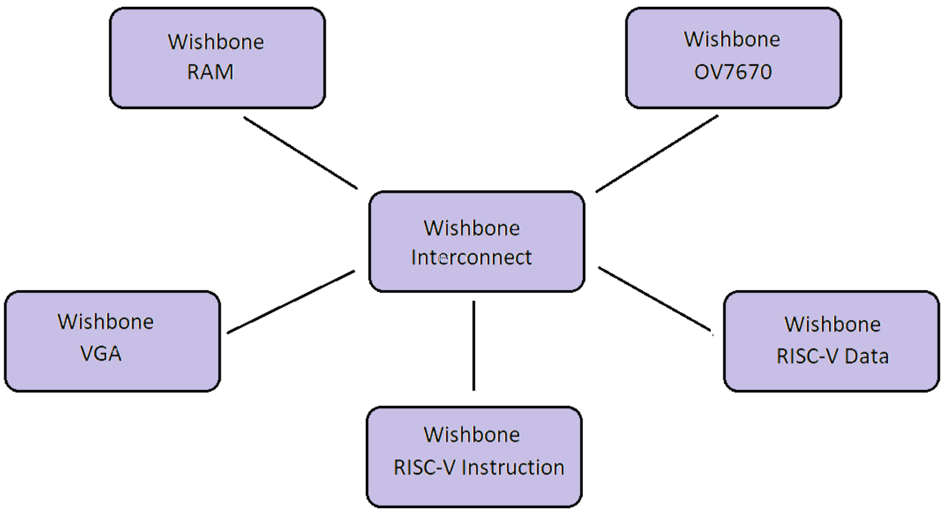
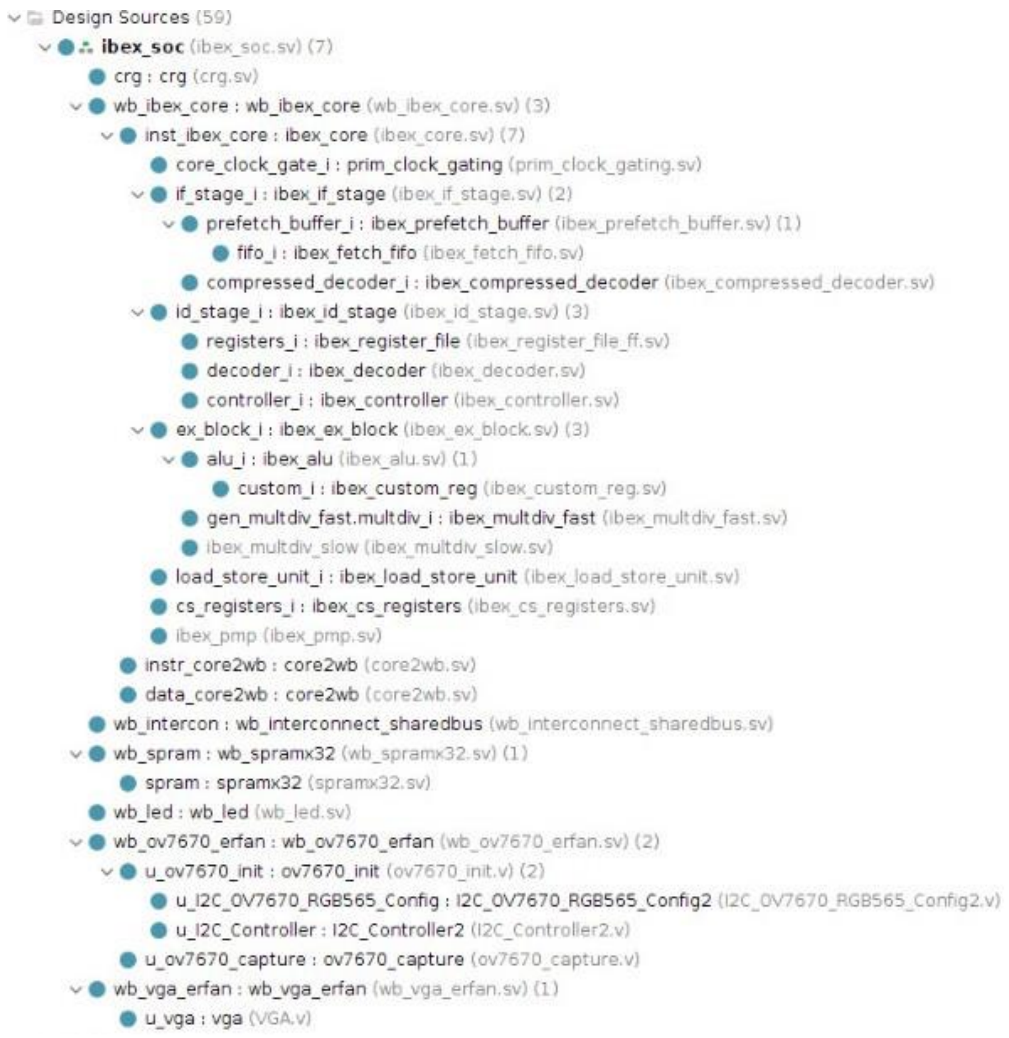


Figure 4.12 : Interconnect management.

In the Vivado project, one of the modules called under the top module named “ibex_soc” is the Wishbone Ibex core module named “wb_ibex_core”. This module is required for the Ibex core to be able to communicate with the other peripherals. At the stage of connecting the previously tested Ibex core to the project, it was observed that the Ibex core module defined in “wb_ibex_core” had less input and output signals than the previously tested Ibex core. Since there are no suitable places to connect these signals, it was decided to add the Ibex core defined in “wb_ibex_core” module into the project and continue with this version in the next steps [10]. After adding all necessary modules, the hierarchy is obtained as in Figure 4.13.



```
Design Sources (59)
  ibex_soc (ibex_soc.sv) (7)
      crg : crg (crg.sv)
      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_ff.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) (1)
                custom_i : ibex_custom_reg (ibex_custom_reg.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)
          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_led : wb_led (wb_led.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_Config2.v)
            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)
```

Figure 4.13 : Vivado Wishbone project hierarchy.

The system includes two masters and three slaves. Masters are for data memory and installation memory. Image pixel values will be provided in C codes and will be accessed through the Wishbone RAM module. There are also camera and VGA modules in the system so that the FPGA card can communicate with the camera and access the image. However, it is not planned to use the camera module just yet. A schematic of all peripheral connections can be seen in Figure 4.14.

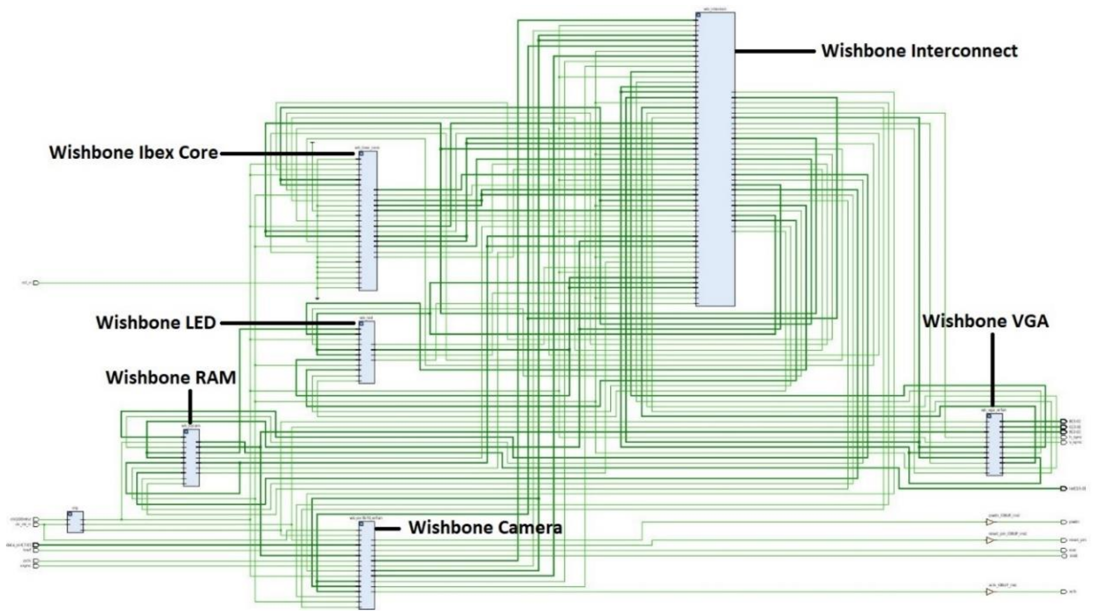


Figure 4.14 : Schematic of the implemented SoC with Wishbone interface.

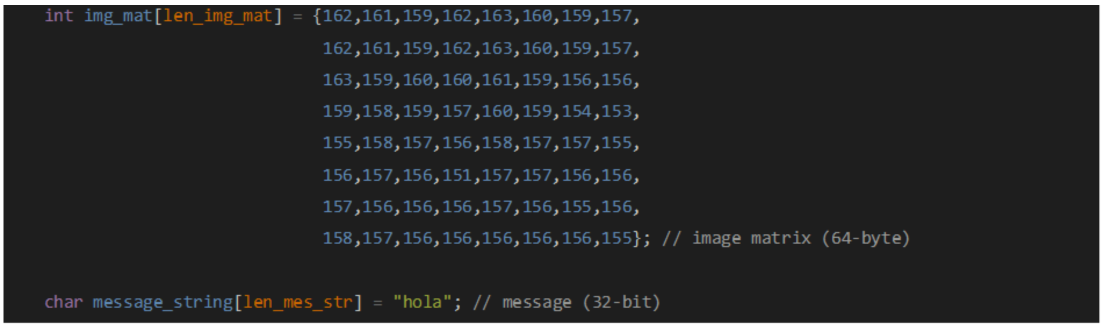
5. FPGA IMPLEMENTATION

5.1 Iplementation of Image Steganography with LSB

5.1.1 C code generation for programming Ibex

In order to perform the steganography process in the embedded system, the LSB algorithm, which was previously written in MATLAB code, must be translated into C code. No library should be used in this C code, it should be built with basic operations.

First of all, an 8x8 grayscale image matrix and secret message are defined as input as shown in Figure 5.1.



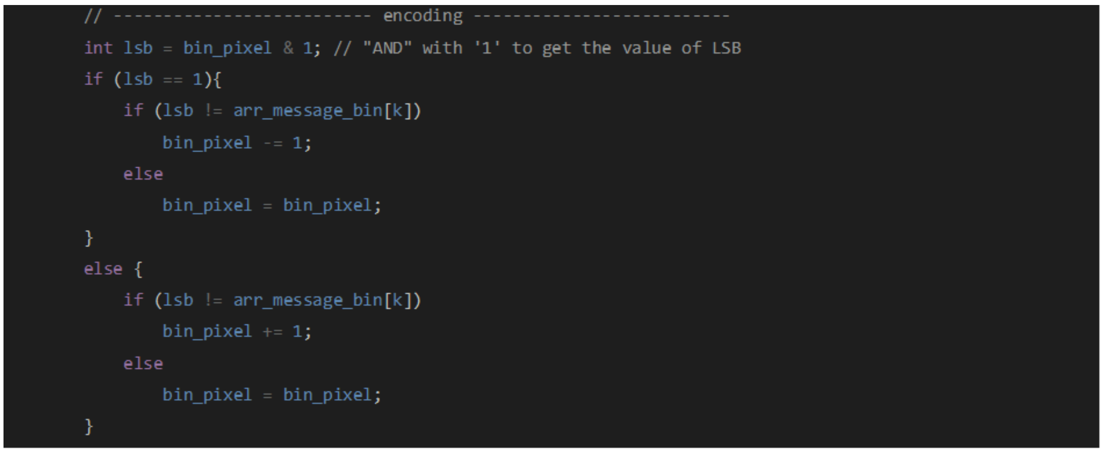
```
int img_mat[len_img_mat] = {162,161,159,162,163,160,159,157,
                            162,161,159,162,163,160,159,157,
                            163,159,160,160,161,159,156,156,
                            159,158,159,157,160,159,154,153,
                            155,158,157,156,158,157,157,155,
                            156,157,156,151,157,157,156,156,
                            157,156,156,156,157,156,155,156,
                            158,157,156,156,156,156,156,155}; // image matrix (64-byte)

char message_string[len_mes_str] = "hola"; // message (32-bit)
```

Figure 5.1 : Definitions of image matrix and secret message in C code.

The reason for choosing a small size image like 8x8 is to make it easier to follow the process while testing the code.

If these values are known, the code that embeds the message using the LSB method is designed. This code basically circulates the pixels in the image matrix starting from the beginning and replaces them with the secret message bits in binary form. For this message, the length of the message is 32 bits and the size of the image is 64 bytes. One bit of message content will be embedded in each pixel. As a result, half of the image would be replaced by the message, and the rest would remain in its original form.

In this design, first, sequentially selected pixels from the matrix and the hidden message are converted into binary form. Embedding the message to the least significant bit of the pixel is shown in Figure 5.2.



```
// ------------------------- encoding -------------------------
int lsb = bin_pixel & 1; // "AND" with '1' to get the value of LSB
if (lsb == 1){
    if (lsb != arr_message_bin[k])
        bin_pixel -= 1;
    else
        bin_pixel = bin_pixel;
}
else {
    if (lsb != arr_message_bin[k])
        bin_pixel += 1;
    else
        bin_pixel = bin_pixel;
}
```

Figure 5.2 : Encoding part of C code.

Here "lsb" refers to the last bit of the pixel. The "AND" operation is used to extract the last bit of the 8-bit pixel. Because if an “AND” operation is performed for an 8-bit value and "00000001", all bits of this 8 bit will be zero except for the least significant bit. Therefore, the pixel value in binary form, named "bin_pixel" in the design, and the value ‘1’ will be processed with the “AND” operator, and the result will be the least significant bit of the image pixel.

The next step is to compare the kth bit in the message with the value of LSB and assign the value of the pixel according to the situation. If these two values are the same, the pixel does not change. However, if it is different, the LSB of the pixel is replaced with the message bit.

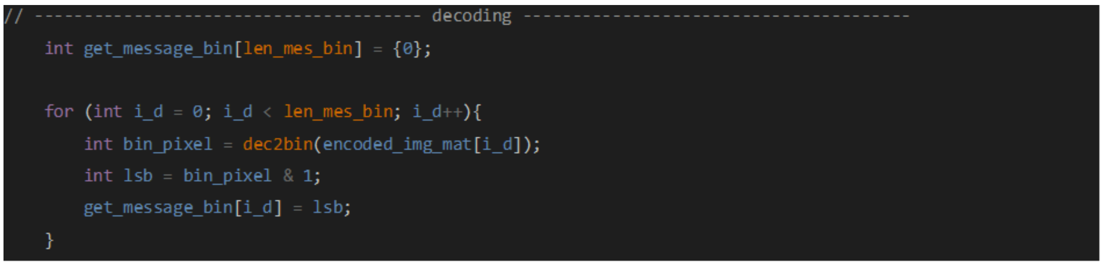
Table 5.1 shows how this code works for the first 8 pixels.

Table 5.1 : Encoding process for first 8 bits of the image.

Pixel Values (decimal)	Pixel Values (binary)	Message in Binary Form (“h”)	Encrypted Pixel Values (binary)	Encrypted Pixel Values (decimal)
162	10100010	0	10100010	162
161	10100001	1	10100001	161
159	10011111	1	10011111	159
162	10100010	0	10100010	162
163	10100011	1	10100011	163
160	10100000	0	10100000	160
159	10011111	0	10011110	158
157	10011101	0	10011100	156

Thus, the message embedding process is carried out. Furthermore, similar steps with message encoding are followed for message decoding. The least significant bit of the image matrix containing the message is obtained with the help of the "AND" operation, also in this matrix, pixels are read as the length of the message and the least significant bit of each one is stored into an array.

The code equivalent of the transactions mentioned above is also shown below in Figure 5.3.



```
// ------------------------------------- decoding -------------------------------------
    int get_message_bin[len_mes_bin] = {0};

    for (int i_d = 0; i_d < len_mes_bin; i_d++){
        int bin_pixel = dec2bin(encoded_img_mat[i_d]);
        int lsb = bin_pixel & 1;
        get_message_bin[i_d] = lsb;
    }
```

Figure 5.3 : Decoding part of C code.

After these operations, necessary actions are taken to convert this message from binary form to character, and the secret message is obtained from the encoded image in the end.

Subsequent to testing the C code with a small size image and message in terms of message embedding and decoding operations, the feasibility of the operation was retested under new conditions by giving a larger image matrix and message input to this code, which works without any problems.

As input, the matrix of the 256x256 grayscale image shown in Figure 5.4 and the new message "This is the secret message." string is given.

$$\begin{array}{cc} & \begin{array}{ccc} \textit{Column 1} & & \textit{Column 256} \end{array} \\ \begin{array}{c} \textit{Row 1} \\ \\ \textit{Row 256} \end{array} & \begin{bmatrix} 162 & \cdots & 141 \\ \vdots & \ddots & \vdots \\ 44 & \cdots & 106 \end{bmatrix} \end{array}$$

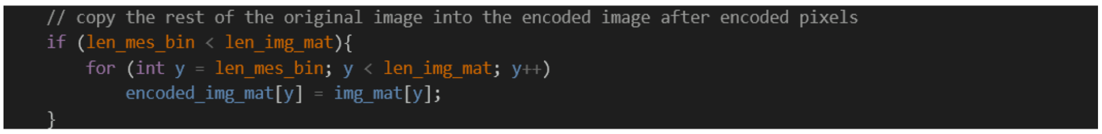
Figure 5.4 : 256x256 grayscale cover image and its matrix.

With these new inputs, the C code was retested and the message was first encoded in the image and then decoded from the image. Subsequently, message below was obtained and it was confirmed that the code works without any problems.

“This is the secret message.”

When this code, which works correctly, is included in the project on Vivado, the desired output could not be obtained. Even though the operation is done correctly, it is a problem for the project that some simple operations cannot be read from the behavioral simulation, because as a result of this situation, pixel values cannot be obtained at the output and this is naturally undesirable. For these reasons, the C code has been rearranged in order to eliminate such problems. The main purpose of these corrections is to observe that these operations are done in the simulation by placing the for loops where simple operations are made into the more complex for loops.

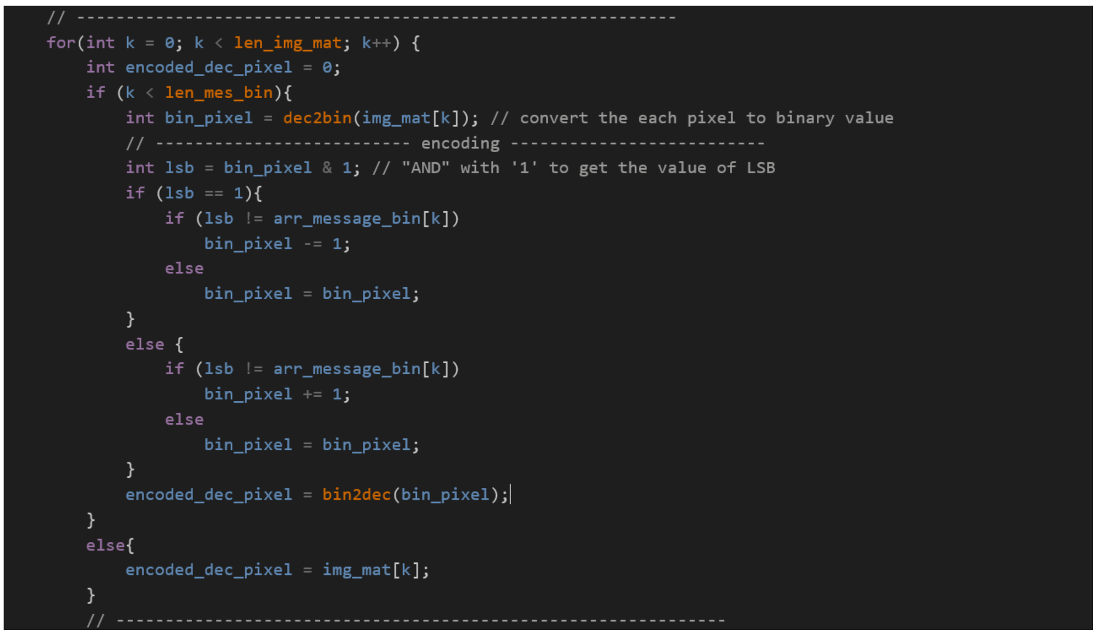
In this case, the outputs of the loop outside this embedding loop, which copies the non-message part of the image from the original matrix to the encoded matrix, do not appear in the simulation. The operation performed in this loop is shown in Figure 5.5, and as can be seen from the figure, this is a quite simple assignment operation.



```
// copy the rest of the original image into the encoded image after encoded pixels
if (len_mes_bin < len_img_mat){
    for (int y = len_mes_bin; y < len_img_mat; y++)
        encoded_img_mat[y] = img_mat[y];
}
```

Figure 5.5 : Simple loop.

Therefore, the loop in which message embedding is performed as shown above is rearranged as shown in Figure 5.6.

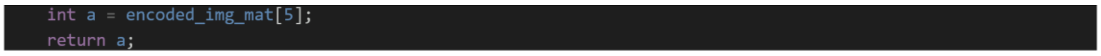


```
// -------------------------------------------------------------
for(int k = 0; k < len_img_mat; k++) {
    int encoded_dec_pixel = 0;
    if (k < len_mes_bin){
        int bin_pixel = dec2bin(img_mat[k]); // convert the each pixel to binary value
        // ------------------------- encoding -------------------------
        int lsb = bin_pixel & 1; // "AND" with '1' to get the value of LSB
        if (lsb == 1){
            if (lsb != arr_message_bin[k])
                bin_pixel -= 1;
            else
                bin_pixel = bin_pixel;
        }
        else {
            if (lsb != arr_message_bin[k])
                bin_pixel += 1;
            else
                bin_pixel = bin_pixel;
        }
        encoded_dec_pixel = bin2dec(bin_pixel);
    }
    else{
        encoded_dec_pixel = img_mat[k];
    }
    // -------------------------------------------------------------
```

Figure 5.6 : Edited complex loop.

The change made here is basically that the operation that was done outside of this loop before, which is the copying of the bits that will not be embedded in any message, into the encoded matrix, is included in the more complex loop. Only an if block is inserted to specify whether to place the embedded or direct original pixel into the new matrix without much change in the way the loop works. This if block performs this operation by checking whether the message is finished with the help of an index.

It is also necessary to return any pixel of the image to the end of the main() function in order to see the outputs in the simulation. The arbitrarily chosen return value is shown in Figure 5.7 below.



```
    int a = encoded_img_mat[5];
    return a;
```

Figure 5.7 : Code lines for return of the main().

5.1.2 Simulation results on Vivado

After testing the accuracy of the C code of the image steganography algorithm using LSB, the next step was to test this algorithm by adding it to the project created in the Vivado program. To achieve this, the C code written must be compiled with the GNU tool chain and a memory file containing the instructions contained in the C code must be obtained as a result. The extension of the memory file with the extension ".vmem", which was created as a result of the compilation, was changed to ".mem" and added to the Vivado project as a source file. In the project, by adding the file name to the module related to RAM, the memory file is read and the algorithm written with the instructions received is performed.

In order to see the pixel values desired to be obtained during the simulation, the data output signal of Ibex Core was added to the simulation, and the data values were checked when the signal representing the write enable in the module took the value of 1. A part of Vivado simulation results for image steganography algorithm using LSB can be seen in Figure 5.8.

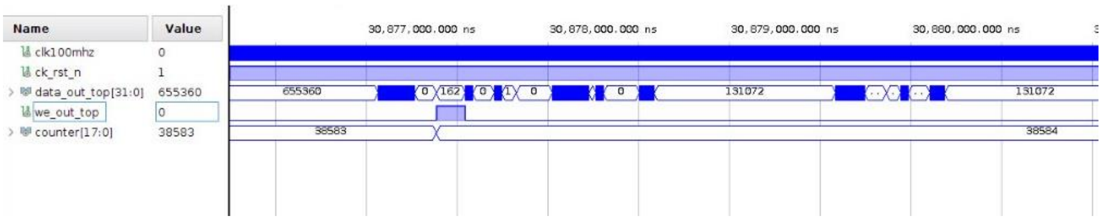


Figure 5.8 : Simulation results for image steganography algorithm using LSB.

At the same time, in order to check the accuracy of the obtained encoded image pixels, an addition made to the test bench code is provided to write the pixel values to a created file name “output.txt”.

The pixels in the "output.txt" file obtained after the simulation is run are arranged line by line, with one pixel value per line. The MATLAB program was used to obtain the image from the values in this text file. Thus, after the file was read, it was converted into a 256x256 matrix and its output in image format was observed. The image shown in figure 3.5 was chosen as the image on which the embedding process will be applied. The figure for the encoded image on which the LSB steganography algorithm is applied can be seen in Figure 5.9.

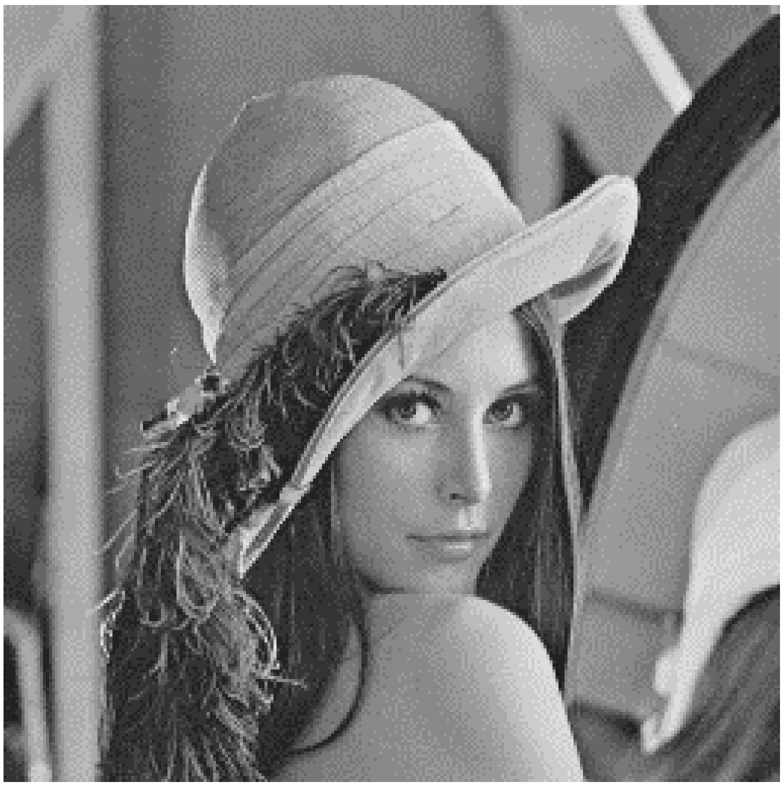


Figure 5.9 : Encoded image using LSB image steganography.

As can be observed from the results, there was no visible change in the image after the embedding process. However, it should be tested that the embedding process is performed correctly in the image obtained as the output of the Vivado program. For this purpose, the pixels of the image are navigated in order, starting from the first pixel, in a loop that repeats as long as the length of the binary form of the embedded message, and the least significant bits are stored in an array. The elements in this array consisting of 0 and 1 values were grouped as eight, and the ASCII code equivalents of these 8-

bit numbers were found and converted into text. As a result of this decryption MATLAB code, the text is below obtained:

“This is the secret message.”

Since the message intended to be embedded and the message obtained after decryption are the same, it has been verified that the encryption process has been completed correctly.

5.1.3 Physical visualization of the implementation

After the simulations were made in the Vivado and MATLAB environments, the algorithm has been verified. Then, the visualization of the verified algorithm was made using a VGA cable and a monitor. To be able to use the VGA output of the Nexys4 DDR board, the previously implemented Wishbone protocol must be utilized. As stated in the section 4.3.1, the reason for this is that the RISC-V processor cannot communicate with external modules such as SRAM, VGA, or a user-made custom IP without a computer bus protocol called Wishbone.

The process for visualizing the implementation physically can be divided into two subprocesses: C code alteration and the Wishbone modification. These sub-processes are described in the following paragraphs in the order in which they are given.

As known, in memory-mapped peripheral registers, use of volatile pointers to volatile variables are very convenient. Thus, a 32-bit register named “pixel” was memory mapped at address 0x11010 in the C code using the code line below.

❖ volatile uint_32t *pixel = (volatile uint32_t *) 0x11010;

By defining it volatile, compiler optimization was prevented and thus, it is ensured that the desired pixel values are written to the desired RAM addresses starting from 0x11010. Encoding was done as before (as explained in the section 5.1.1) by initializing a matrix with cover image pixels and embedding the secret message bits in the least significant bits of these matrix elements in an order. Only change in the embedding section of the code was made at the end when the element values of the encoded image matrix were copied into the “pixel”.

After that, by changing the first element in “pixel”, VGA module was activated, and process was suspended for 5 seconds using a usleep() function defined in the code before the decryption section. Activation of VGA module will be explained in the further paragraphs.

At the end of the C code, the decryption was made by taking the least significant bits of the first 16 “pixel” values and collecting them into an array. Since there are only 16 user LEDs on the Nexys4 DDR Board, it is possible to display only 2 letters on the LEDs at the same time considering each letter needs to be represented by 8-bits. In order to demonstrate the reliability of the algorithm, showing the first 2 embedded letters would be enough to validate the decryption. After collecting the LSBs into an array, a simple code block summing all 16 binary number with increasing 2’s power to display every bit in the LEDs output in an order was written. To illustrate, if bits 1, 0 and 1 are wanted to be shown in LED2, LED1 and LED0 respectively, the summation below needs to be done before giving a value to the LEDs output.

$$sum = 1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 = 5 = (101)_2$$

If this summation value is given to the LEDs output, LED2 and LED0 will light up.

The written C code for the activation of the VGA module, usleep() function and the decryption section can be seen in Figure 5.10.

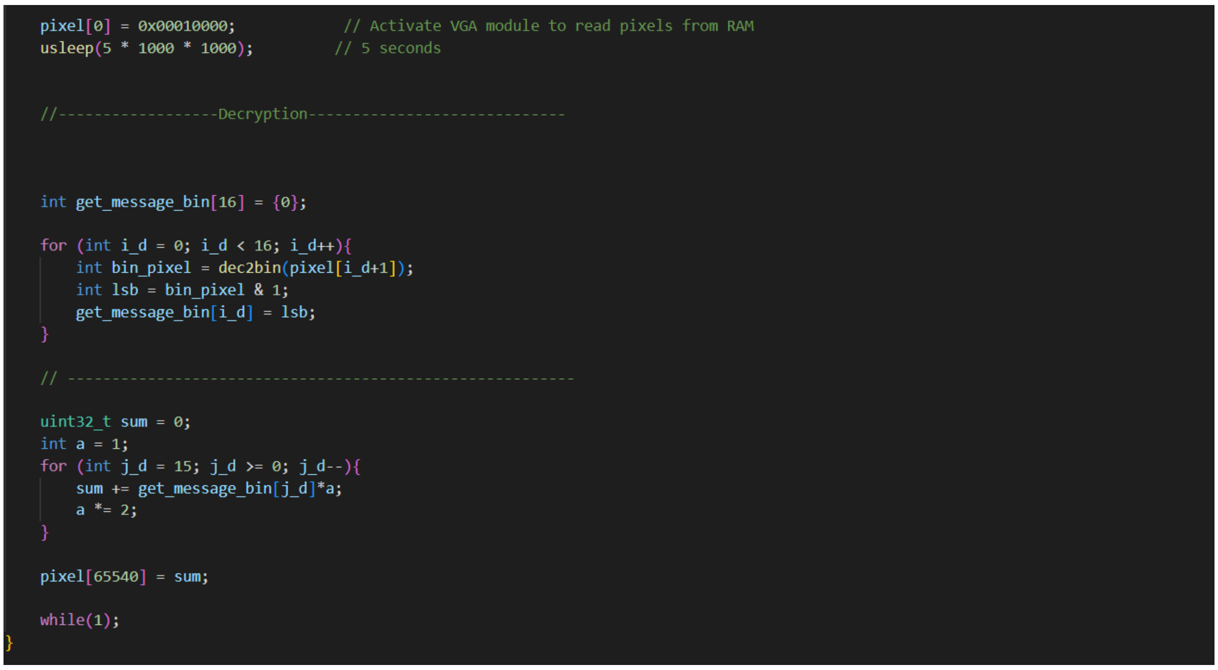

```c
    pixel[0] = 0x00010000;            // Activate VGA module to read pixels from RAM
    usleep(5 * 1000 * 1000);         // 5 seconds


    //-----------------Decryption----------------------------



    int get_message_bin[16] = {0};

    for (int i_d = 0; i_d < 16; i_d++){
        int bin_pixel = dec2bin(pixel[i_d+1]);
        int lsb = bin_pixel & 1;
        get_message_bin[i_d] = lsb;
    }

    // --------------------------------------------------------

    uint32_t sum = 0;
    int a = 1;
    for (int j_d = 15; j_d >= 0; j_d--){
        sum += get_message_bin[j_d]*a;
        a *= 2;
    }

    pixel[65540] = sum;

    while(1);
}
```

Figure 5.10 : C code written for physical visualization on LEDs and monitor.

A critical part in the C code modification was the reduction of the number of bits used in the representation of the cover image. To make the C code compatible with the VGA output of the Nexys4 DDR Board, image represented with 8 bits (0-255 scale) was converted to 4-bits (0-15 scale) since board’s VGA output has 12-bit (every channel in the RGB has 4 bits output). After altering the C code, usual steps were followed to create the memory file and initialize the RAM.

In the Vivado project, VGA module was introduced as a master while the SRAM module was introduced as a slave. Since the default memory management defined by the Ibex is not enough to compile the code, ROM size was increased to 0x11000 which equals to 68kB using the linker script lines given below to prevent the ROM overflow.

```
MEMORY
{
   rom      : ORIGIN = 0x00000000, LENGTH = 0x11000 /* 68 kB */
   stack    : ORIGIN = 0x00011000, LENGTH = 0x64000 /* 400 kB */
}
```

Therefore, to make the single port 32-bit RAM in the Vivado project compatible with the generated memory file, RAM size in the Vivado project was also increased to 0x75000 which equals to 468kB. In the VGA’s wishbone interface module, the VGA module was controlled with an input called “master_arbiter”. In the RAM module section given below, this value was initialized with the data input at the address 0x11010 which is the first value of “pixel” or in other terms: pixel[0].

```
        else if(valid && wb.we && (wb.adr[addr_width + 1 : 2]=='h4404))
            master_arbiter <= wb.dat_i;
```

Thus, by assigning values to the first element of “pixel”, master modules in the wishbone were controlled. For this case, by assigning 0x00010000 to it, VGA module was activated because of the code lines below in the VGA’s wishbone interface.

```
        if ( master_arbiter[16]=='h1)
        begin
            wb.cyc      <= 1'b1;
            wb.stb      <= 1'b1;
        end
```

After the activation of the VGA module, the input pixel values were read depending on the acknowledge signal coming from the interconnect shared bus. In the VGA module, these 4-bit pixel values were given into the VGA output channels using the code section below in order to display the image as grayscale in the reserved 256x256 space.

```
        vga_red   <= frame_pixel[3:0];
        vga_green   <= frame_pixel[3:0];
        vga_blue  <= frame_pixel[3:0];
```

At the end, encoded image using LSB algorithm was given to the monitor. Related monitor output can be seen in Figure 5.11.

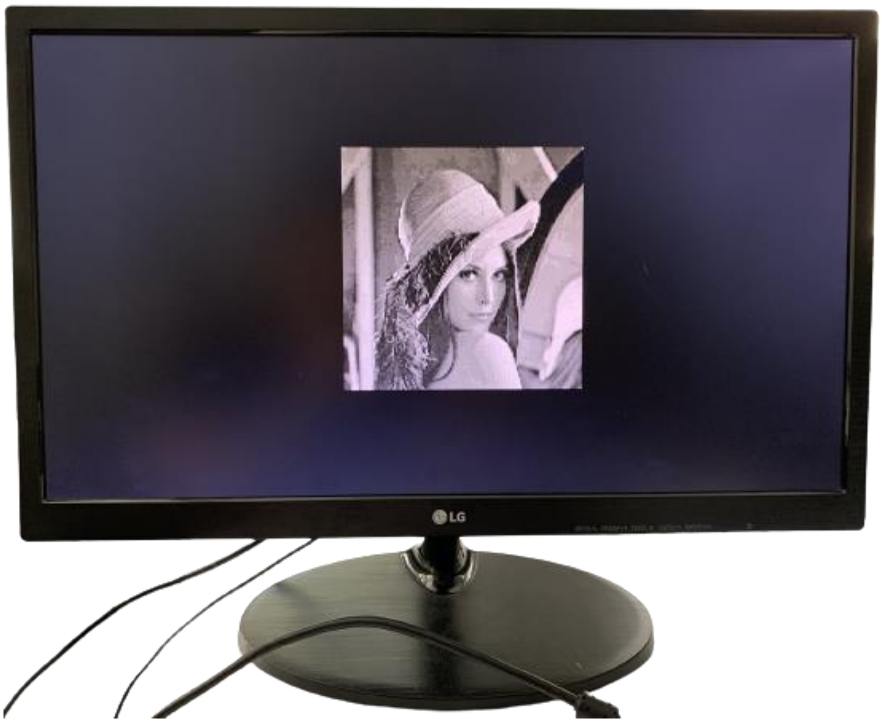


Figure 5.11 : Encoded image using LSB algorithm given to the monitor.

Then, as seen from the code section given in Figure 5.10, algorithm waits 5 seconds before it makes any decryption. Then, calculated sum value was given to the first “pixel” address after the encoded image matrix values were written to the required addresses.

Finally, sum value written to RAM was given to the LEDs output using the code section below in SRAM module.

```
        else if(valid && wb.we && (wb.adr[addr_width + 1 : 2]=='h14408))
            led <= wb.dat_i [15:0];
```

Display of the ASCII letters “T” and “h” in binary format on the board’s LED outputs can be seen in Figure 5.12.

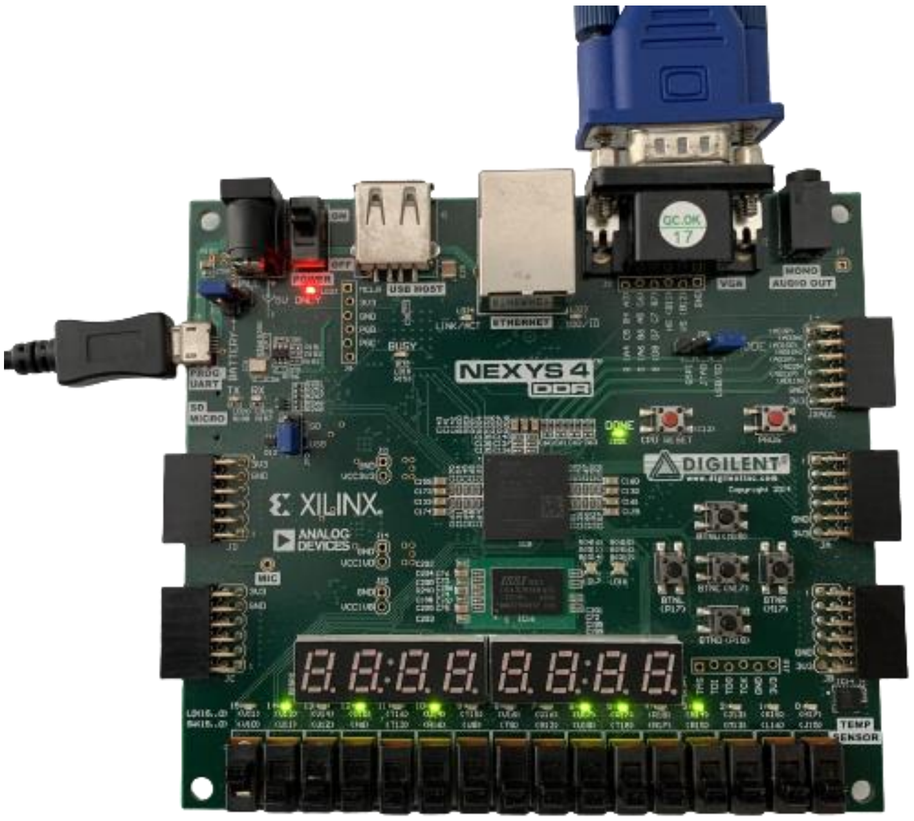


Figure 5.12 : Display of the letter “T” and “h” in binary format on the LEDs.

5.2 Iplementation of Image Steganography with LSB Using LFSR Algorithm

5.2.1 C code generation for programming Ibex

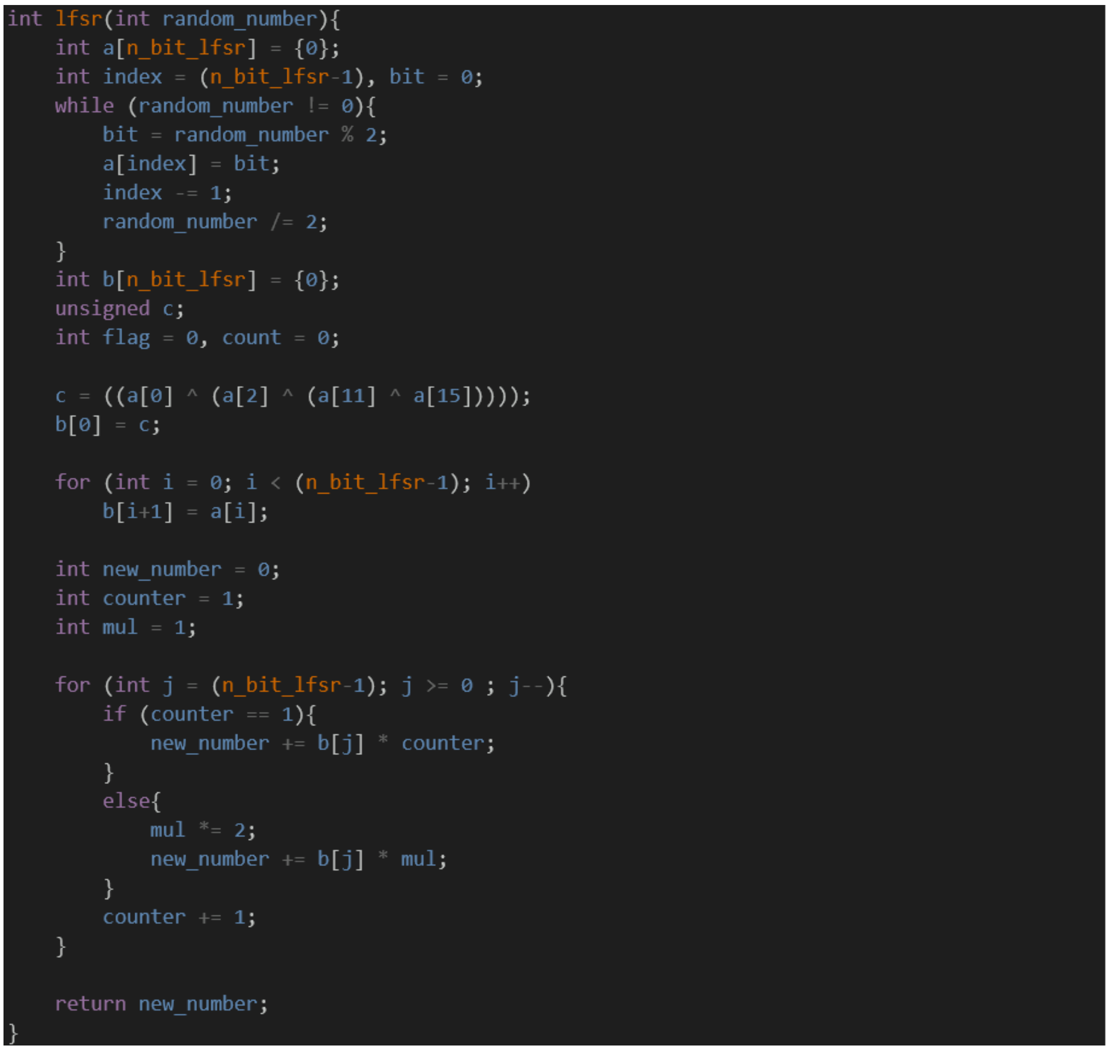
LSB with LFSR was designed by modifying the C code of the previous LSB method. The innovation LFSR algorithm brings to the conventional LSB method, as explained in the previous titles, is the ability to select pixels randomly.

For this project, the LFSR algorithm is designed as 16-bit, and thereby, at most $2^{16} = 65536$ different numbers can be produced. In order to reach this maximum value, as stated in the previous titles, an irreducible polynomial should be used. XORing the bits selected from the 16-bit number according to this polynomial will let one to produce the maximum amount of different numbers.

The irreducible polynomial used in this design is given below.

$$p(x) = x^{16} + x^{12} + x^{3} + x^{1} + 1 \tag{5.1}$$

Section that performs the LFSR algorithm can be seen in Figure 5.13.



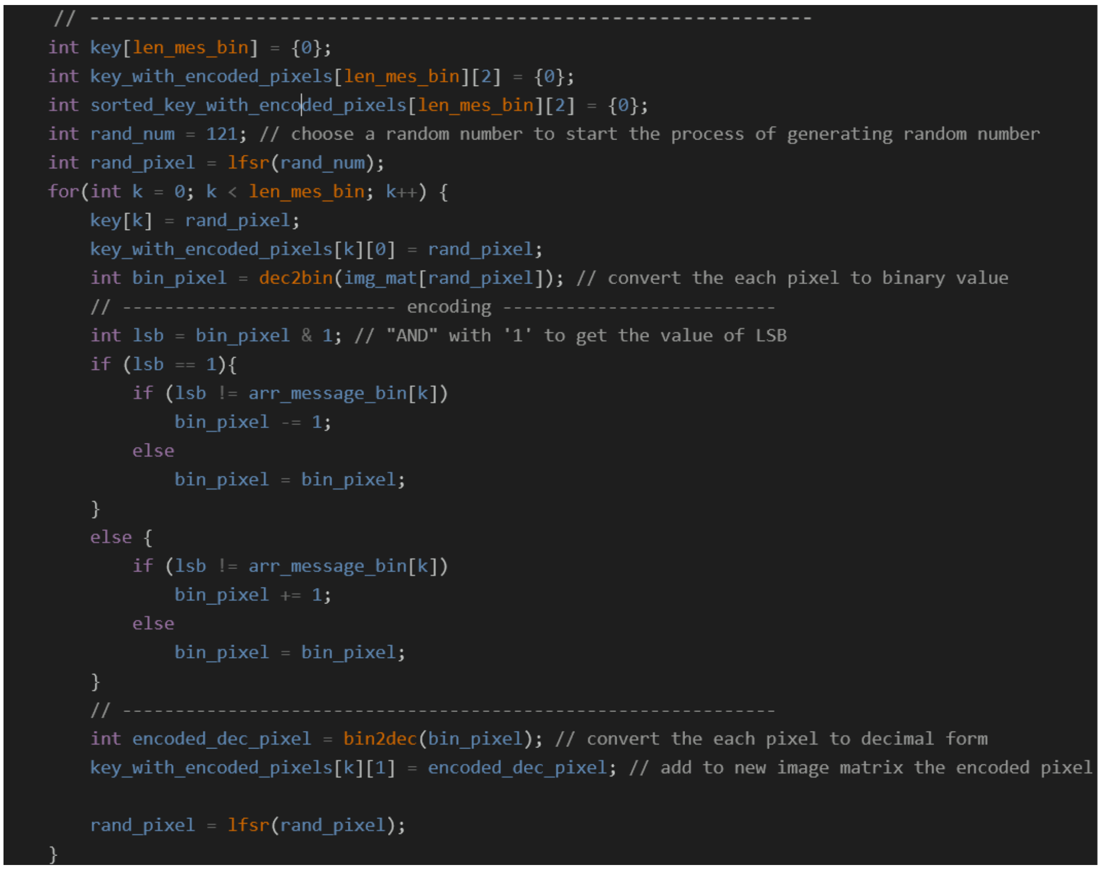
```c
int lfsr(int random_number){
    int a[n_bit_lfsr] = {0};
    int index = (n_bit_lfsr-1), bit = 0;
    while (random_number != 0){
        bit = random_number % 2;
        a[index] = bit;
        index -= 1;
        random_number /= 2;
    }
    int b[n_bit_lfsr] = {0};
    unsigned c;
    int flag = 0, count = 0;

    c = ((a[0] ^ (a[2] ^ (a[11] ^ a[15]))));
    b[0] = c;

    for (int i = 0; i < (n_bit_lfsr-1); i++)
        b[i+1] = a[i];

    int new_number = 0;
    int counter = 1;
    int mul = 1;

    for (int j = (n_bit_lfsr-1); j >= 0 ; j--){
        if (counter == 1){
            new_number += b[j] * counter;
        }
        else{
            mul *= 2;
            new_number += b[j] * mul;
        }
        counter += 1;
    }

    return new_number;
}
```

Figure 5.13 : The function for the LFSR algorithm.

This function takes a pixel value in the decimal form at its input and converts this value to binary form and turns it into an array. Then it takes the 16th (MSB), 14th, 5th, and 1st (LSB) bits of this array and XORs it, and assigns this value to a variable. It copies these values to another array by shifting each bit to the right and assigns the value, which is obtained by XORing some bits and defined earlier, to the most significant bit of the new array.

Finally, this function outputs the decimal equivalent of the new number obtained. Thus, the LFSR algorithm is implemented.

One of the changes made in the rest of the code was made around the for loop where the embedding operation is performed, and these changes in the C code can be seen from Figure 5.14.



```c
    // -------------------------------------------------------------
    int key[len_mes_bin] = {0};
    int key_with_encoded_pixels[len_mes_bin][2] = {0};
    int sorted_key_with_encoded_pixels[len_mes_bin][2] = {0};
    int rand_num = 121; // choose a random number to start the process of generating random number
    int rand_pixel = lfsr(rand_num);
    for(int k = 0; k < len_mes_bin; k++) {
        key[k] = rand_pixel;
        key_with_encoded_pixels[k][0] = rand_pixel;
        int bin_pixel = dec2bin(img_mat[rand_pixel]); // convert the each pixel to binary value
        // ------------------------- encoding -------------------------
        int lsb = bin_pixel & 1; // "AND" with '1' to get the value of LSB
        if (lsb == 1){
            if (lsb != arr_message_bin[k])
                bin_pixel -= 1;
            else
                bin_pixel = bin_pixel;
        }
        else {
            if (lsb != arr_message_bin[k])
                bin_pixel += 1;
            else
                bin_pixel = bin_pixel;
        }
        // -------------------------------------------------------------
        int encoded_dec_pixel = bin2dec(bin_pixel); // convert the each pixel to decimal form
        key_with_encoded_pixels[k][1] = encoded_dec_pixel; // add to new image matrix the encoded pixel

        rand_pixel = lfsr(rand_pixel);
    }
```

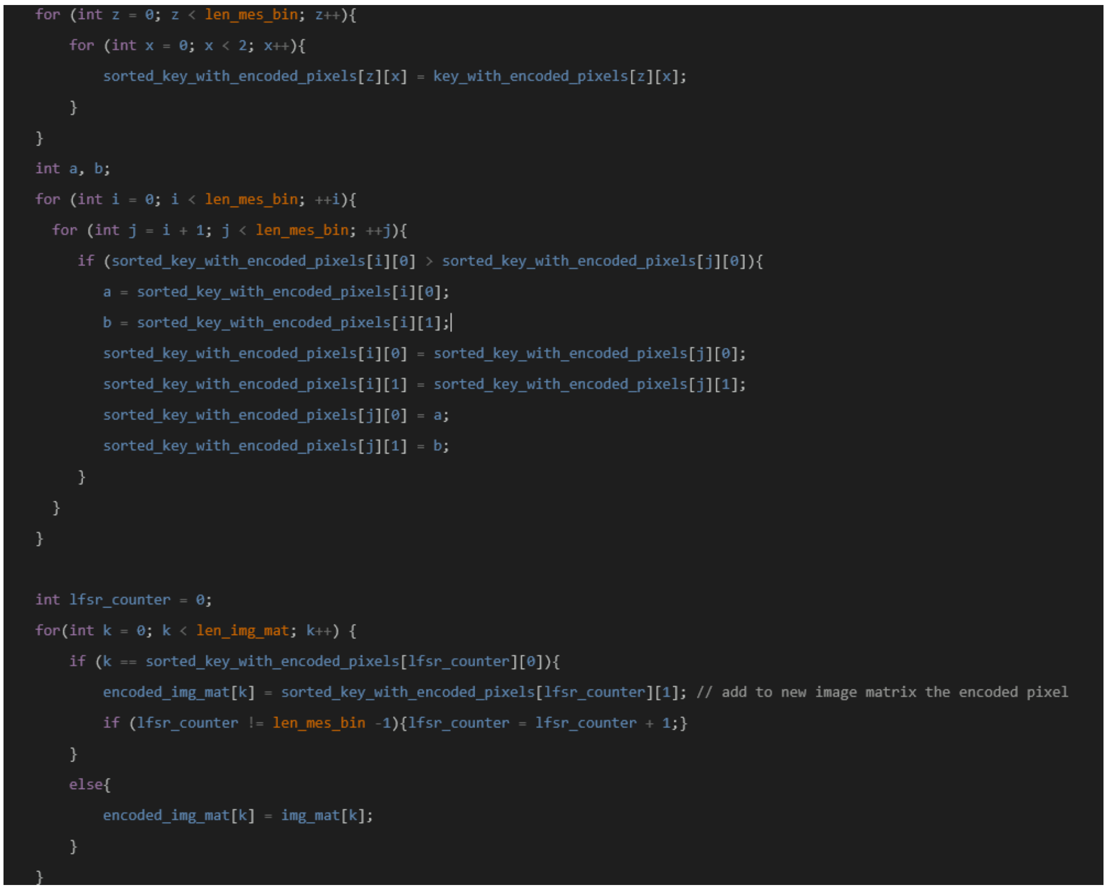
Figure 5.14 : Edited encoding part of C code.

Here a random pixel is selected before entering the loop. For this, first, a randomly determined number is given to the input of the lfsr() function given in Figure 5.13 and the value at the output of this function is processed as the first random pixel, then the random pixel value generation is done inside the loop.

In addition, the message bits are embedded into selected pixel's least significant bits and the encoded pixel values that are converted from binary to decimal are collected in a 2D-array.

With the process of recording each generated random pixel in a key sequence, the key of the encryption algorithm is also determined. This part is crucial for testing the design and decoding the secret message.

In addition, the sequentiality of the key array has an important place in observing pixel values in behavioral simulation. For this reason, lines of code that perform the sorting operation have been added to the code as shown in Figure 5.15 below.



```
for (int z = 0; z < len_mes_bin; z++){
    for (int x = 0; x < 2; x++){
        sorted_key_with_encoded_pixels[z][x] = key_with_encoded_pixels[z][x];
    }
}
int a, b;
for (int i = 0; i < len_mes_bin; ++i){
  for (int j = i + 1; j < len_mes_bin; ++j){
     if (sorted_key_with_encoded_pixels[i][0] > sorted_key_with_encoded_pixels[j][0]){
        a = sorted_key_with_encoded_pixels[i][0];
        b = sorted_key_with_encoded_pixels[i][1];
        sorted_key_with_encoded_pixels[i][0] = sorted_key_with_encoded_pixels[j][0];
        sorted_key_with_encoded_pixels[i][1] = sorted_key_with_encoded_pixels[j][1];
        sorted_key_with_encoded_pixels[j][0] = a;
        sorted_key_with_encoded_pixels[j][1] = b;
     }
  }
}

int lfsr_counter = 0;
for(int k = 0; k < len_img_mat; k++) {
    if (k == sorted_key_with_encoded_pixels[lfsr_counter][0]){
        encoded_img_mat[k] = sorted_key_with_encoded_pixels[lfsr_counter][1]; // add to new image matrix the encoded pixel
        if (lfsr_counter != len_mes_bin -1){lfsr_counter = lfsr_counter + 1;}
    }
    else{
        encoded_img_mat[k] = img_mat[k];
    }
}
```

Figure 5.15 : Sorting part of C code.

The operation done here is the bubble sort method, which has a simple working logic. With this algorithm, the purpose is to sort the contents of the key array, as stated before, so that even though the message is randomly embedded in pixels, the hidden message can be accessed understandably because it is sequential.

The operating logic of the bubble sort algorithm is based on comparing two elements. It compares two consecutive elements and swaps them if the former element is greater than the latter, otherwise, it does not do any operation and passes the next one. However, it does this with a loop inside the loop, as it cannot fully perform this enumeration operation by returning the array only once. And with these two nested loops, it can sort all elements of the array.

5.2.2 Simulation results on Vivado

After testing the accuracy of the C code of the LSB image steganography algorithm using LFSR, this algorithm is tested on Vivado project by using a similar process mentioned in section 5.1.2.

The C code was compiled with the GNU tool chain and a memory file was obtained as a result. The extension of the memory file changed from “.vmem” to “.mem” and the file added to the Vivado project as a source file. In the project, by adding the file name to the module related to RAM, the memory file is read and the algorithm written with the instructions received is performed.

Data output of the Ibex Core and write enable signal was added as signals in order to observe the result pixels on the simulation. A part of Vivado simulation results for LFSR based image steganography algorithm can be seen in Figure 5.16. As previous, pixel values of encoded image are written to “output.txt” file that created in test bench.

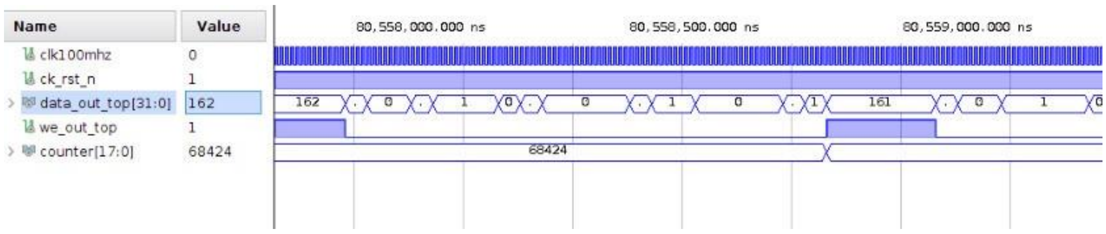


Figure 5.16 : Simulation results for LFSR based image steganography algorithm.

Using the MATLAB code mentioned in the section 5.1.2, pixel values in “output.txt” file are converted into a 256x256 matrix and it is obtained as an image format. The image shown in figure 3.5 was chosen as the image on which the embedding process will be applied. The figure for the encoded image on which the LSB steganography algorithm using LFSR is applied can be seen in Figure 5.17.

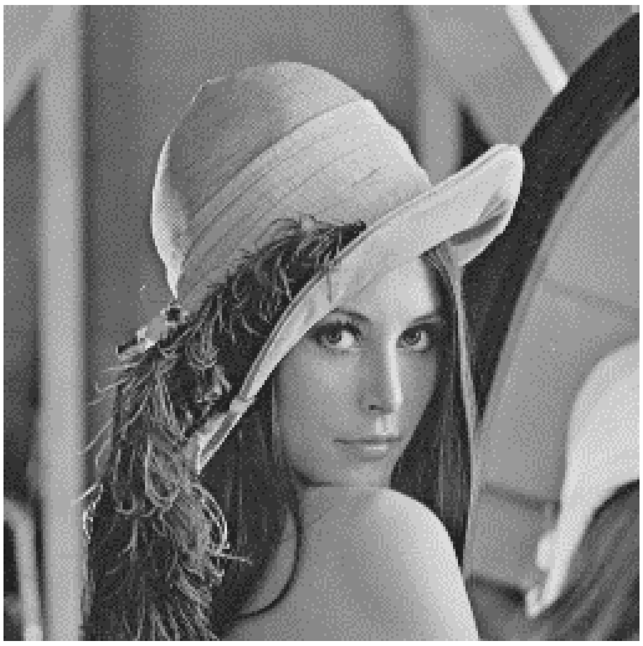


Figure 5.17 : Encoded image using LFSR based image steganography.

When the original and encoded image images are compared, there is no noticeable difference between them. However, it should be tested that the embedding process is performed correctly in the image obtained as the output of the Vivado program.

In order to test that the embedding process is performed correctly in the image obtained as the output of the Vivado program, the key produced in C code and random pixel numbers generated with LFSR can be used. The pixels of the image represented by each element of the key are selected in order, and the least significant bits of these pixel values are stored in an array. The elements in this array are converted into text by using their ASCII code equivalents. As a result of this decryption MATLAB code, the text below is obtained:

“This is the secret message.”

Since the message intended to be embedded and the message obtained after decryption are the same, it has been verified that the encryption process for LFSR based image steganography algorithm has been completed correctly.

5.2.3 Physical visualization of the implementation

After the simulations were made in the Vivado and MATLAB environments successfully, the verification of the algorithm has been done. Later, the verified algorithm was visualized again using a VGA cable and a monitor. For using the Nexys-4 DDR board’s 12-bit VGA output, previously implemented Wishbone protocol and its master-slave structure must be utilized since RISC-V processor cannot communicate with external modules without it.

The process of visualization the implementation physically was done in the same manner as in section 5.1.3. First, C code was modified according to the algorithm and then, the Wishbone that was altered in the section 5.1.3 was used as it is.

Like previous, a 32-bit register named “pixel” was memory mapped at address 0x11010. Encoding was done as explained in the section 5.2.1 by initializing a matrix with cover image pixels, creating a random number sequence using the LFSR algorithm and embedding the secret message bits in the least significant bits of the selected pixels from the LFSR algorithm. The only change made in the embedding part of the code was to copy the cover image pixels to “pixel” and replace the generated pixel indices with message embedded versions in their least significant bits.

Later, as explained in section 5.1.3, by changing the first element in “pixel”, VGA module was activated. Then, the process was suspended for 5 seconds using a usleep() function defined in the code before the decryption section.

At the end of the C code, the decryption was made by taking the least significant bits of the first 16 embedded bits and collecting them into an array. In this decryption algorithm, previously stated key sequence was utilized.

Like before, after collecting the LSBs into an array, a simple code block was written for summing all 16 binary number with increasing 2’s power to display every bit in the 16-bit LED output of the Nexys4 DDR Board. Also, the number of bits used in the representation of the cover image was reduced from 8 bits to 4 bits for making the C code compatible with the VGA output of the FPGA.

After changing the C code, memory file was created using the linker script with an increased ROM size and the RAM was initialized using the created memory file in the modified Vivado project.

Finally, bitstream was generated, FPGA was programmed and by programming it, encoded image using LFSR algorithm was given to the monitor. Related monitor output can be seen in Figure 5.18.

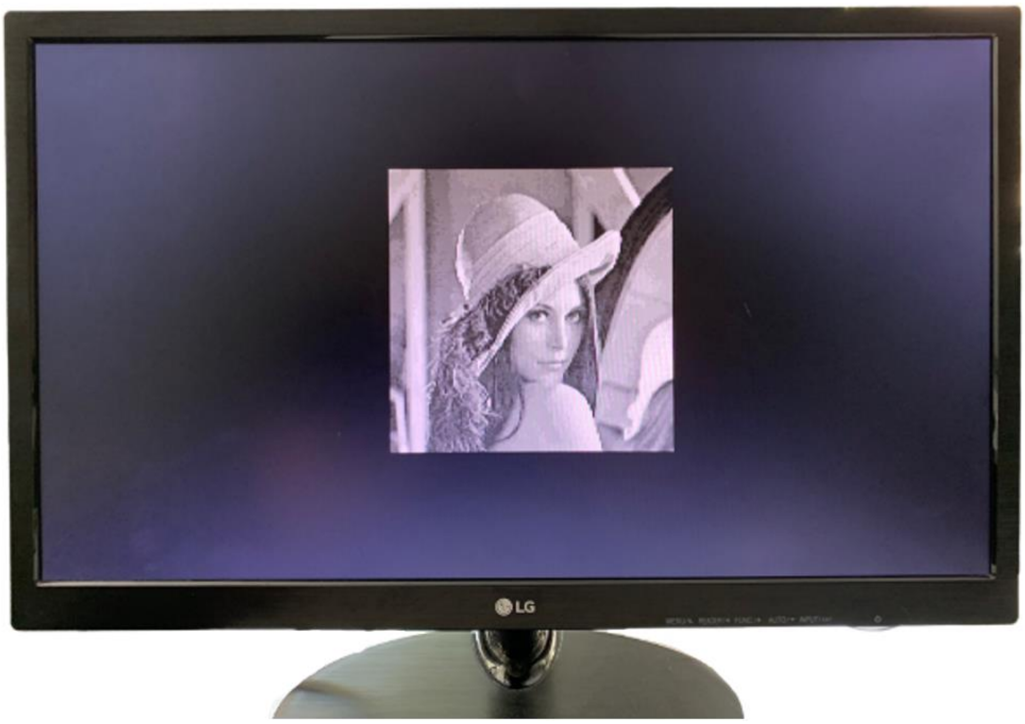


Figure 5.18 : Encoded image using LFSR algorithm given to the monitor.

Display of the ASCII letters “T” and “h” in binary format on the board’s LED outputs for the system using LFSR algorithm can be seen in Figure 5.19.

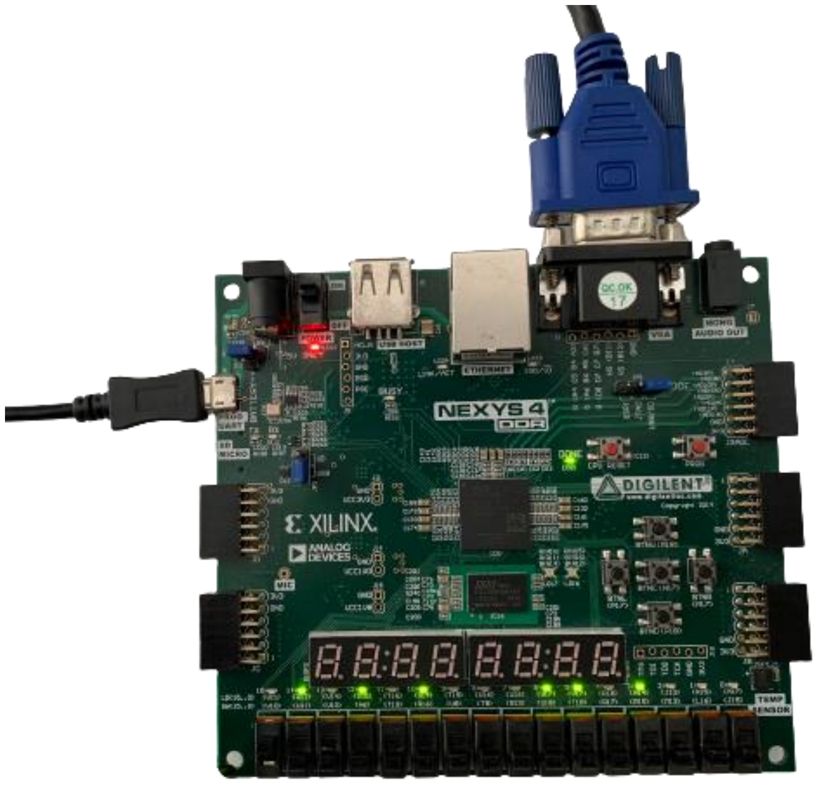


Figure 5.19 : Display of the letter “T” and “h” in binary format on the LEDs.

By looking at the LEDs, one can say that cryptography algorithm works as expected.

5.3 Iplementation of Image Steganography with Frequency Based Algorithm

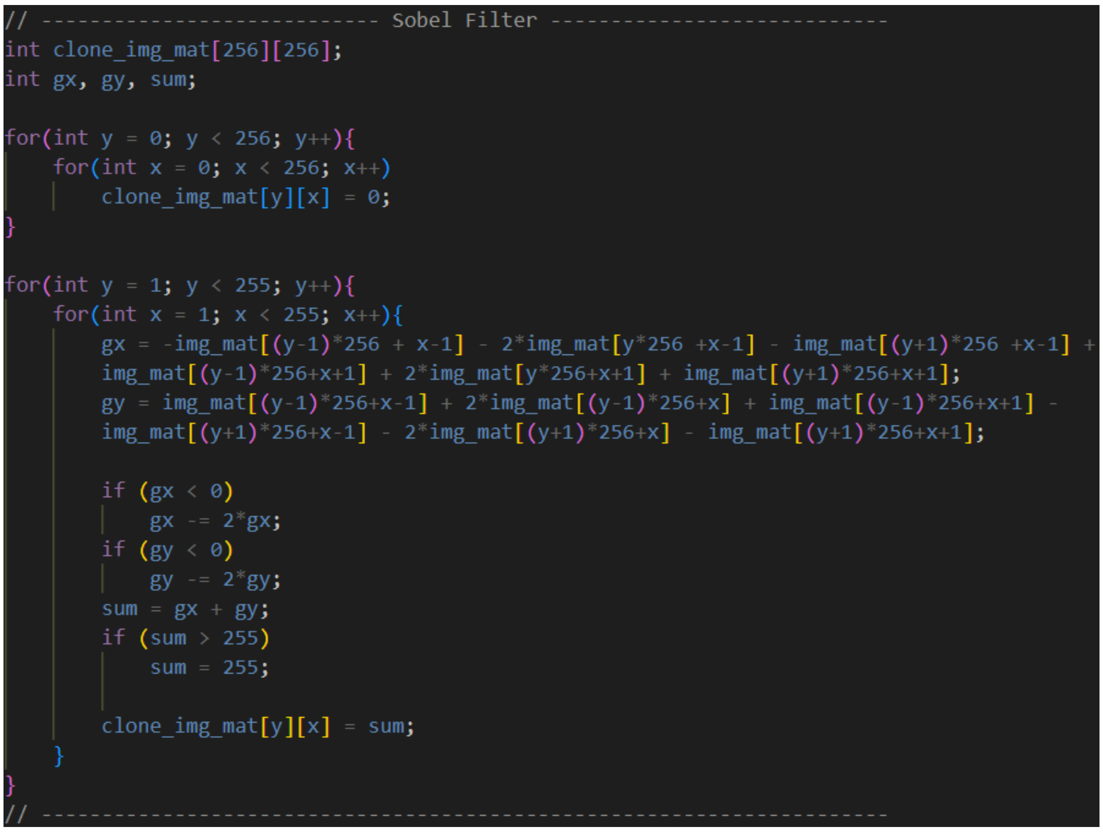
5.3.1 C code generation for programming Ibex

Steganography was previously performed using the LSB method with an LFSR for selecting random cover image pixels. While embedding information related to secret message with these methods, as mentioned before, information was placed only in the least significant bit of the pixels.

Now the goal is to push the boundaries a little bit more and store information in more than one bit in a pixel. Of course, embedding information from the least significant bit to more significant pixels will bring the possibility of distorting the image. Determination for choosing pixels to hide more information is also done by detecting the high-frequency regions in the image. For this detect operation, the Sobel filter, whose mathematics was examined in previous titles, was used.

In this section, the implementation of this filter on C code will be discussed. This code design was made by adding onto the LSB method using the LFSR algorithm for its pixel selection. First of all, the code lines where the Sobel filter implementation is performed will be mentioned, and then the added code lines will be examined in order to fully realize the image steganography process with the frequency-based algorithm.

First, a zero matrix with the dimensions of 256x256 was created and the filtered pixel values were written into this matrix. While filtering the image, two loops were opened to convolve the cover image pixels with the filter coefficients given in Equation 3.6. As stated in section 3.3.2, Sobel filter was implemented by finding the gradients G_x and G_y first. Then, absolute values of these gradients were summed and if it exceeds the maximum level of 255 for an 8-bit image, 255 was written to the new matrix, otherwise, the sum is written. The lines of code shown in Figure 5.20 below are used for these filtering operations.

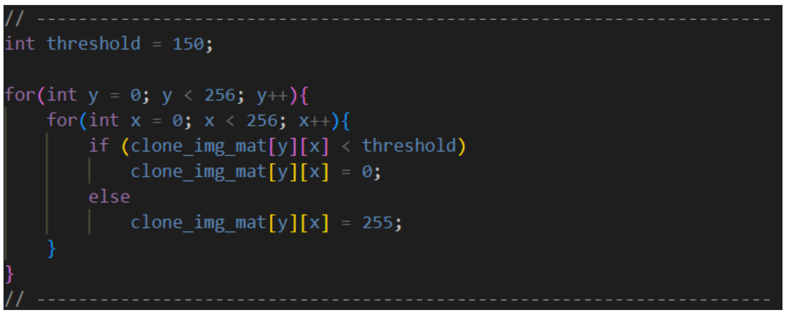


```c
// --------------------------- Sobel Filter ---------------------------
int clone_img_mat[256][256];
int gx, gy, sum;

for(int y = 0; y < 256; y++){
    for(int x = 0; x < 256; x++)
        clone_img_mat[y][x] = 0;
}

for(int y = 1; y < 255; y++){
    for(int x = 1; x < 255; x++){
        gx = -img_mat[(y-1)*256 + x-1] - 2*img_mat[y*256 +x-1] - img_mat[(y+1)*256 +x-1] +
        img_mat[(y-1)*256+x+1] + 2*img_mat[y*256+x+1] + img_mat[(y+1)*256+x+1];
        gy = img_mat[(y-1)*256+x-1] + 2*img_mat[(y-1)*256+x] + img_mat[(y-1)*256+x+1] -
        img_mat[(y+1)*256+x-1] - 2*img_mat[(y+1)*256+x] - img_mat[(y+1)*256+x+1];

        if (gx < 0)
            gx -= 2*gx;
        if (gy < 0)
            gy -= 2*gy;
        sum = gx + gy;
        if (sum > 255)
            sum = 255;

        clone_img_mat[y][x] = sum;
    }
}
// -----------------------------------------------------------------
```

Figure 5.20 : Code lines for filtering operation.

Then, the pixel values obtained after the filtering process was saturated according to a selected threshold. For this purpose, value of 150 was preferred and if the pixel values in the matrix are below 150, 0 for that pixel; otherwise, 255 is assigned.

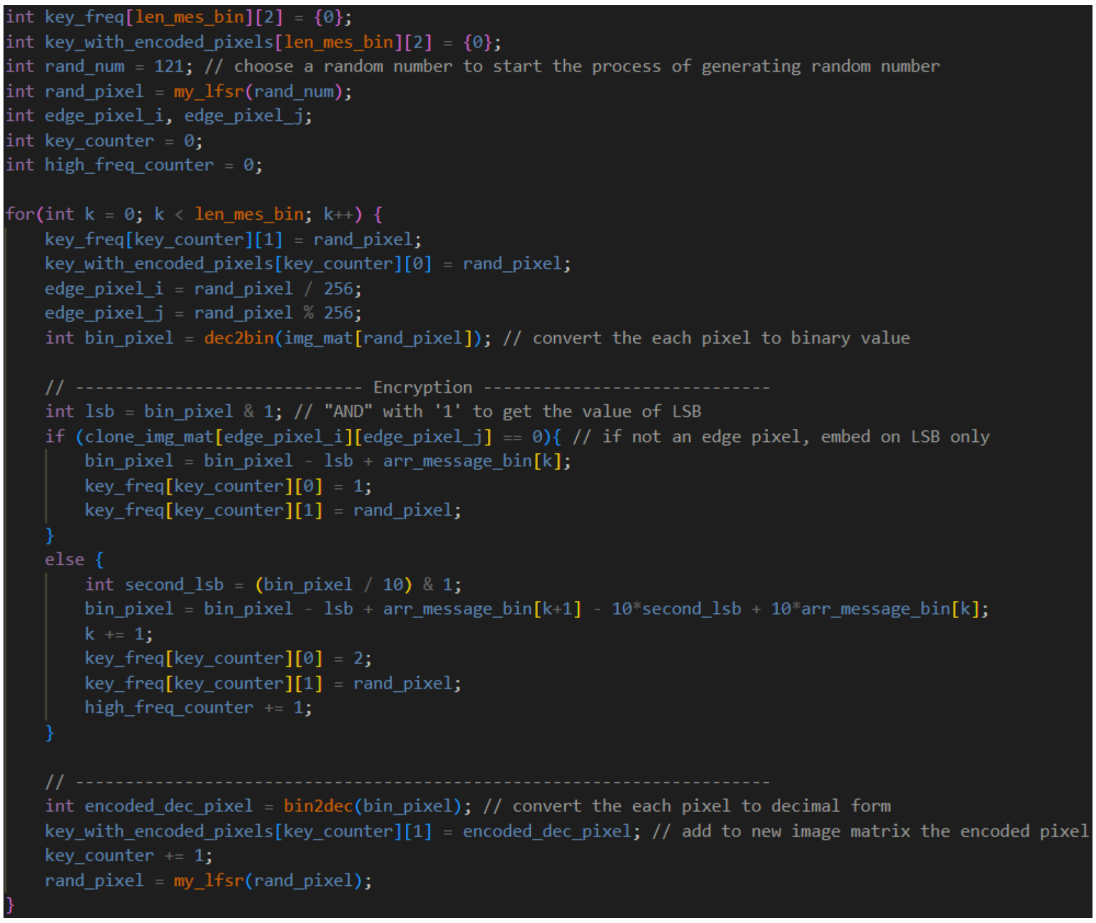
This process is given in Figure 5.21 below.



```
// ---------------------------------------------------------------------
int threshold = 150;

for(int y = 0; y < 256; y++){
    for(int x = 0; x < 256; x++){
        if (clone_img_mat[y][x] < threshold)
            clone_img_mat[y][x] = 0;
        else
            clone_img_mat[y][x] = 255;
    }
}
// ---------------------------------------------------------------------
```

Figure 5.21 : Thresholding the Sobel image.

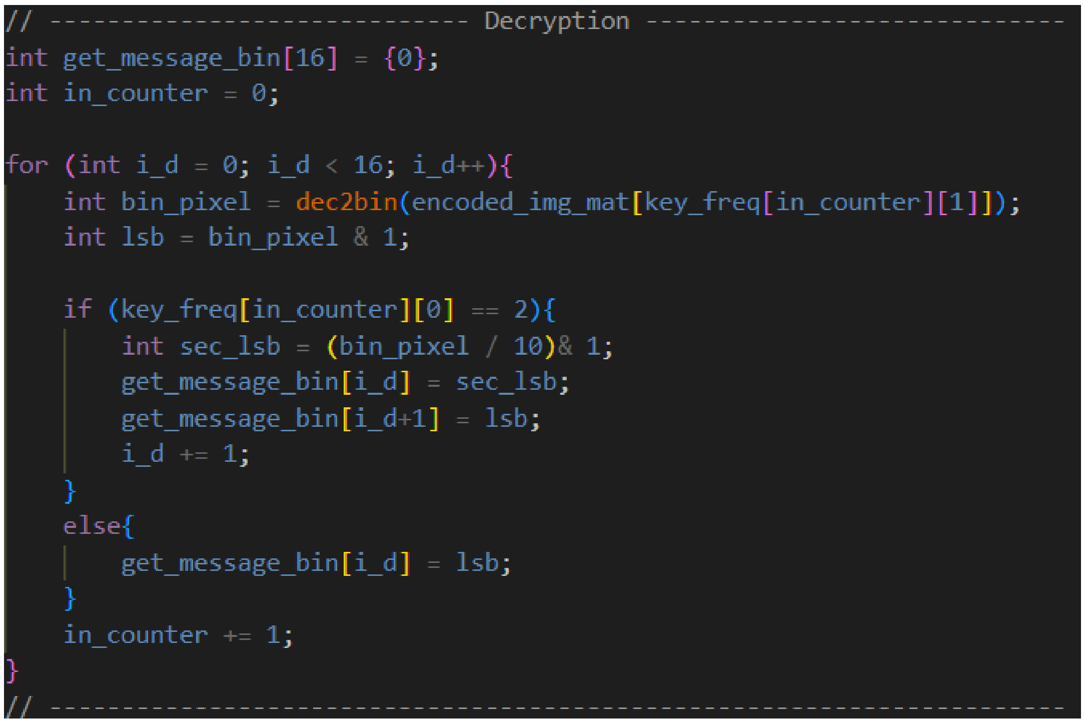
After filtering and detecting high-frequency regions, there are some differences in the encryption part in addition to the previous ones in this code design. If value of the pixel selected with the LFSR method is 0, then this means that the selected pixel is the edge point in the image. Therefore, if the value is 0, the message will be written to the last 2 bits of the image. If not, then LSB method will be employed. These processes and the necessary arrangements made in addition to this can be seen in Figure 5.22.



```
int key_freq[len_mes_bin][2] = {0};
int key_with_encoded_pixels[len_mes_bin][2] = {0};
int rand_num = 121; // choose a random number to start the process of generating random number
int rand_pixel = my_lfsr(rand_num);
int edge_pixel_i, edge_pixel_j;
int key_counter = 0;
int high_freq_counter = 0;

for(int k = 0; k < len_mes_bin; k++) {
    key_freq[key_counter][1] = rand_pixel;
    key_with_encoded_pixels[key_counter][0] = rand_pixel;
    edge_pixel_i = rand_pixel / 256;
    edge_pixel_j = rand_pixel % 256;
    int bin_pixel = dec2bin(img_mat[rand_pixel]); // convert the each pixel to binary value

    // ---------------------------- Encryption ----------------------------
    int lsb = bin_pixel & 1; // "AND" with '1' to get the value of LSB
    if (clone_img_mat[edge_pixel_i][edge_pixel_j] == 0){ // if not an edge pixel, embed on LSB only
        bin_pixel = bin_pixel - lsb + arr_message_bin[k];
        key_freq[key_counter][0] = 1;
        key_freq[key_counter][1] = rand_pixel;
    }
    else {
        int second_lsb = (bin_pixel / 10) & 1;
        bin_pixel = bin_pixel - lsb + arr_message_bin[k+1] - 10*second_lsb + 10*arr_message_bin[k];
        k += 1;
        key_freq[key_counter][0] = 2;
        key_freq[key_counter][1] = rand_pixel;
        high_freq_counter += 1;
    }

    // ----------------------------------------------------------------------
    int encoded_dec_pixel = bin2dec(bin_pixel); // convert the each pixel to decimal form
    key_with_encoded_pixels[key_counter][1] = encoded_dec_pixel; // add to new image matrix the encoded pixel
    key_counter += 1;
    rand_pixel = my_lfsr(rand_pixel);
}
```

Figure 5.22 : Encryption for the algorithm using the frequency detection method.

In addition to all these, necessary arrangements have been made in the decryption part, which is written only for testing the C code, to collect data from the 2nd bit in high-frequency pixels, and it has been verified that the code performs the encryption process correctly. Also, the decryption part can be viewed in Figure 5.23.



```
// ------------------------------ Decryption ----------------------------
int get_message_bin[16] = {0};
int in_counter = 0;

for (int i_d = 0; i_d < 16; i_d++){
    int bin_pixel = dec2bin(encoded_img_mat[key_freq[in_counter][1]]);
    int lsb = bin_pixel & 1;

    if (key_freq[in_counter][0] == 2){
        int sec_lsb = (bin_pixel / 10)& 1;
        get_message_bin[i_d] = sec_lsb;
        get_message_bin[i_d+1] = lsb;
        i_d += 1;
    }
    else{
        get_message_bin[i_d] = lsb;
    }
    in_counter += 1;
}
// ---------------------------------------------------------------------
```

Figure 5.23 : Decryption for the algorithm using the frequency detection method.

5.3.2 Simulation results on Vivado

After testing the accuracy of the C code of the image steganography algorithm which detects high frequency pixel, and the algorithm is verified on Vivado project by using a similar process explained in sections 5.1.2 and 5.2.2.

The extension of the memory file with the extension ".vmem", which was obtained after the frequency-based C code was compiled using the GNU toolchain, was changed to ".mem" and added to the Vivado project as source file. Afterwards, this file was added to the RAM module in the project where the instructions were read, so that the instructions of the codes written in the C code were executed by the Ibex core.

Data output of the Ibex Core and write enable signal were added to the simulation signals to see the result pixels. A part of Vivado simulation results for frequency-based image steganography algorithm can be seen in Figure 5.24. With the arrangements made in the testbench code, the pixel values of the encoded image are saved in the "output.txt" file.

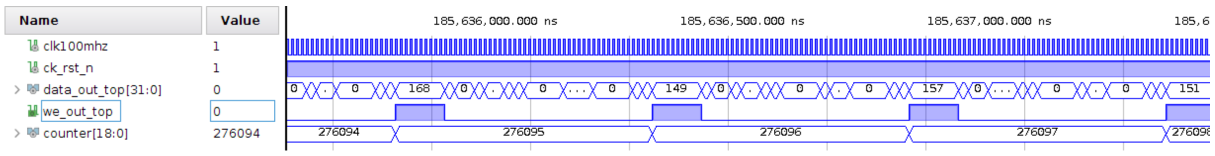


Figure 5.24 : Vivado simulation results for frequency-based algorithm.

To see the encoded image obtained, the MATLAB code is used, where the "output.txt" file is read, converted into a 256x256 square matrix, and displayed. The figure for the encoded image on which the frequency-based steganography algorithm is applied can be seen in Figure 5.25.

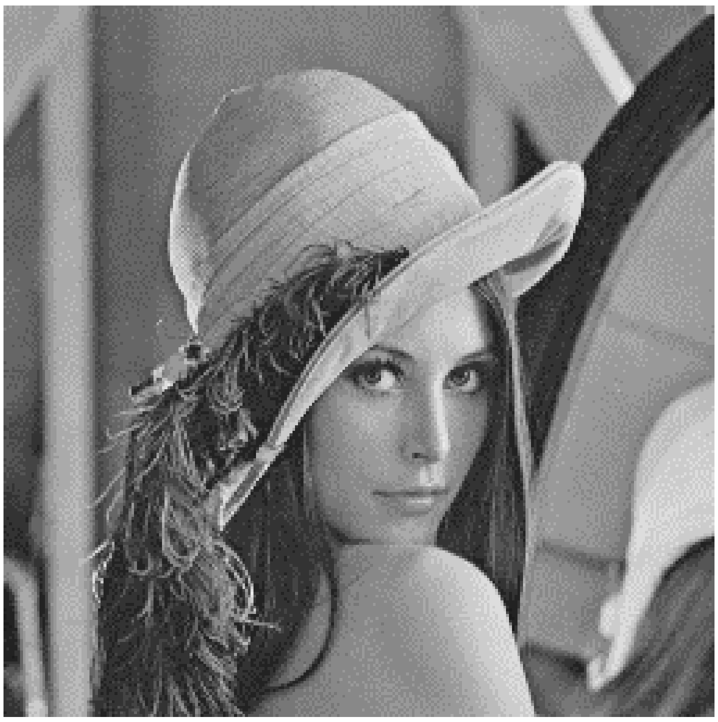


Figure 5.25 : Encoded image using LFSR based image steganography.

When the output image is considered, there is no visible change compared to the original image. However, it is necessary to test whether the embedding process is carried out correctly.

A decryption code was written using the MATLAB environment to check that the embedding process was done correctly. The code first finds the pixel values and reads the "output.txt" file and converts it to a 256x256 square matrix form. Then, using the keys produced with C code, the pixels with embedded messages are visited in the order of the key. In the frequency-based C code, in addition to the C code where only the LFSR algorithm is used, another key is created in which high frequency regions are added. In this way, if a message embedded pixel has a high frequency, the first and second least significant bit of the pixel value is extracted, otherwise only the first least significant bit is extracted.

By adding the bit values extracted from the pixels end to end, a sequence is obtained and this sequence is grouped as octet. By finding the ASCII code equivalent of each 8-bit value group, the binary message is converted to text. As a result of this decryption MATLAB code, the text below is obtained:

```
"This is the secret message."
```

Since the message planned to be embedded and the message obtained after decryption are the same, it has been verified that the encryption process for frequency-based image steganography algorithm has been completed correctly.

5.3.3 Physical visualization of the implementation

As seen from the section 5.3.2, simulations in Vivado and MATLAB environments were done successfully. After that, the algorithm was verified, and the validated algorithm was then visualized using a VGA cable and a monitor. To use the 12-bit VGA output of the Nexys4 DDR card, the previously implemented Wishbone protocol and master-slave structure must be used, as the RISC-V processor cannot communicate with external modules without it.

The physical visualization of the application was done in the same way as in sections 5.1.3 and 5.2.3. First, the C code was modified according to the algorithm, and then Wishbone, which was changed in section 5.1.3, was used as is in the Vivado project.

Like previous, a 32-bit register named “pixel” was memory mapped at address 0x11010. Encoding was done as described in section 5.3.1 by initializing a matrix with the cover image pixels, generating a random number sequence using the LFSR algorithm, and embedding the secret message bits in the last one or two bits of the selected pixels depending on whether the selected pixel in the LFSR algorithm is an edge point or not. Sobel filtered image before thresholding was given to the monitor by changing the first element in the "pixel" to activate the VGA module. Then, using the usleep() function defined in the code, the process was suspended for 5 seconds to give enough time to see the monitor output. Sobel filter output after generating the bitstream and programming the FPGA can be seen in Figure 5.26.

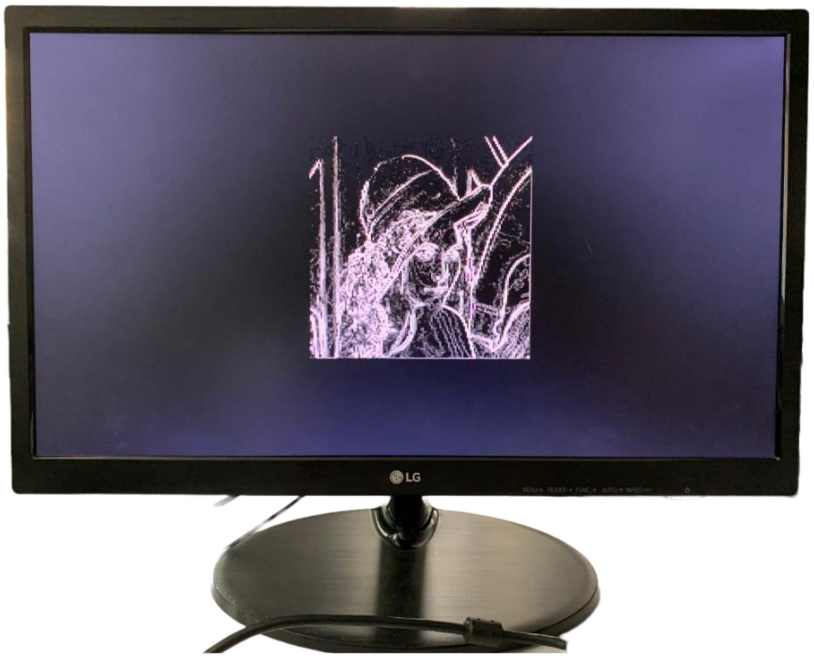
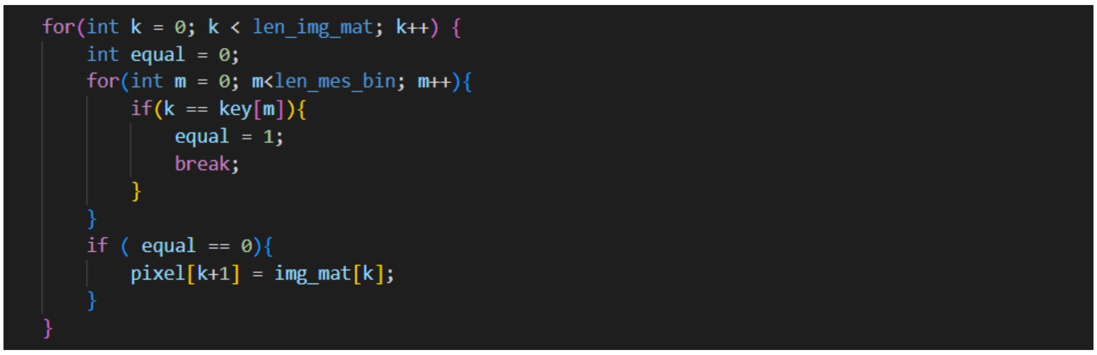


Figure 5.26 : Sobel filter output given to the monitor.

For the embedding part, the only change made in the code was to copy the Sobel filtered image pixels to “pixel” and replace the pixel values in the generated pixel indices with message embedded versions in their last one or two bits. During the embedding, cover image pixels were taken, and necessary embedding operations were made. Also at the final stage, the pixel values in the unselected pixel indices were replaced with their original value from the cover image. The code block that provides this process can be seen from Figure 5.27.



```
for(int k = 0; k < len_img_mat; k++) {
    int equal = 0;
    for(int m = 0; m<len_mes_bin; m++){
        if(k == key[m]){
            equal = 1;
            break;
        }
    }
    if ( equal == 0){
        pixel[k+1] = img_mat[k];
    }
}
```

Figure 5.27 : Replacing the pixel values in the unselected pixel indices.

Then, VGA module was activated again and usleep() function was used for suspending the process for 5 seconds to give enough time for the decryption.

Encoded image using frequency detection algorithm can be seen in Figure 5.28.

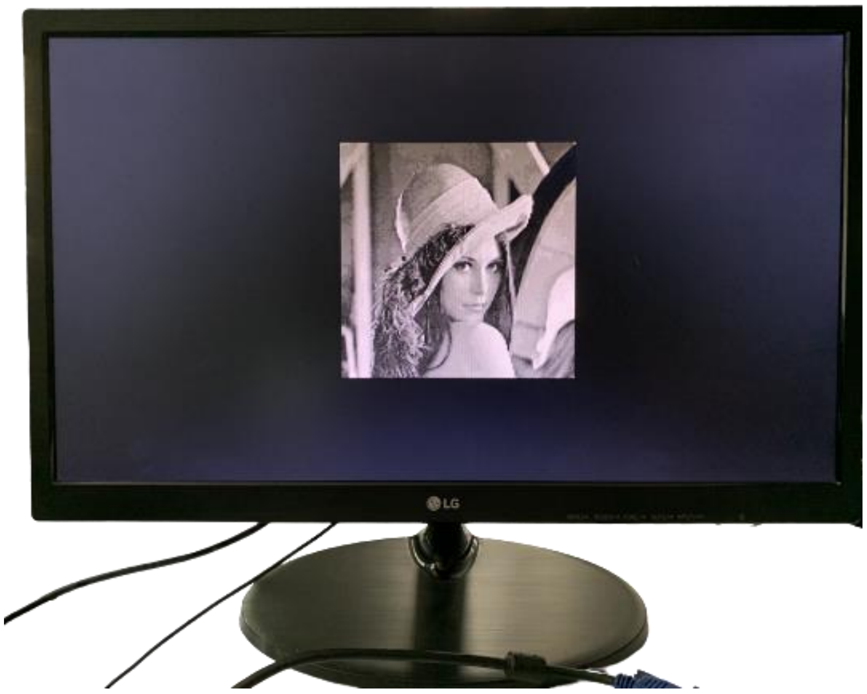
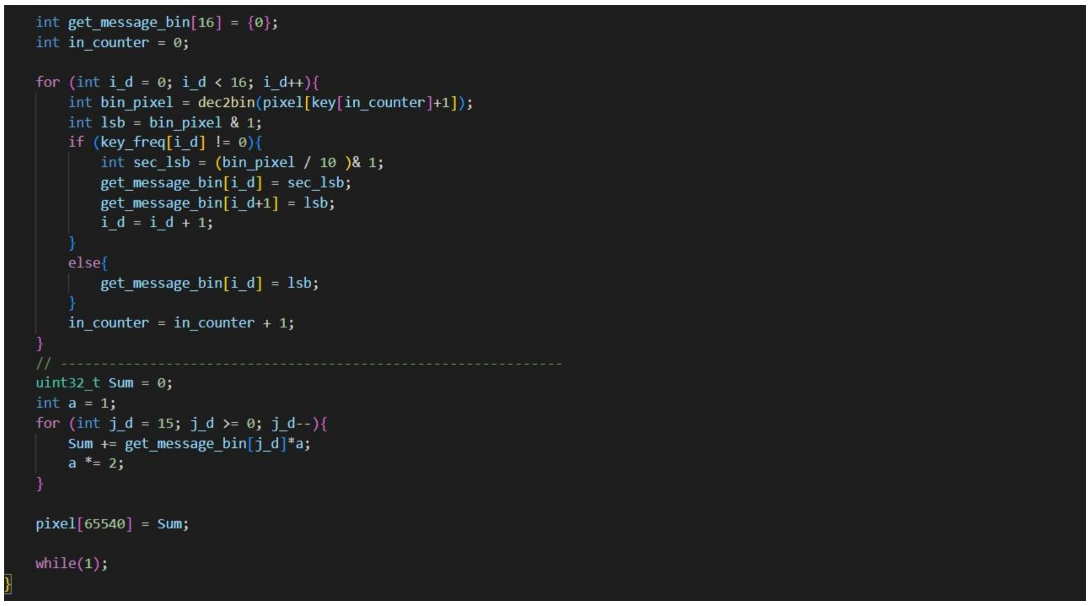


Figure 5.28 : Encoded image using frequency based algorithm on the monitor.

At the end of the C code, the decryption was made by obtaining the first 16 embedded bits and collecting them into an array. In this decryption algorithm, previously stated key sequence was utilized. After collecting the necessary 16 bits to represent the first 2 letters of the secret message, the previous operations were repeated.

The code block for decryption process can be seen from Figure 5.29.



```
    int get_message_bin[16] = {0};
    int in_counter = 0;

    for (int i_d = 0; i_d < 16; i_d++){
        int bin_pixel = dec2bin(pixel[key[in_counter]+1]);
        int lsb = bin_pixel & 1;
        if (key_freq[i_d] != 0){
            int sec_lsb = (bin_pixel / 10 )& 1;
            get_message_bin[i_d] = sec_lsb;
            get_message_bin[i_d+1] = lsb;
            i_d = i_d + 1;
        }
        else{
            get_message_bin[i_d] = lsb;
        }
        in_counter = in_counter + 1;
    }
    // ------------------------------------------------------------
    uint32_t Sum = 0;
    int a = 1;
    for (int j_d = 15; j_d >= 0; j_d--){
        Sum += get_message_bin[j_d]*a;
        a *= 2;
    }

    pixel[65540] = Sum;

    while(1);
}
```

Figure 5.29 : Representing the first two letters of the secret message on LEDs.

Also, to make the C code compatible with the VGA output, the number of bits used in the representation of the cover image has been reduced from 8 bits back to 4 bits. After updating the C code, memory file with ".vmem" extension was created using the linker script that has a higher ROM size value than default and the RAM module was initialized using the created memory file in the modified Vivado project used in the sections 5.1.3 and 5.2.3.

Displayed ASCII letters "T" and "h" in binary format on the Nexys4 DDR board's 16-bit LED output for the system using frequency detection algorithm in addition to the LFSR can be seen from Figure 5.30.

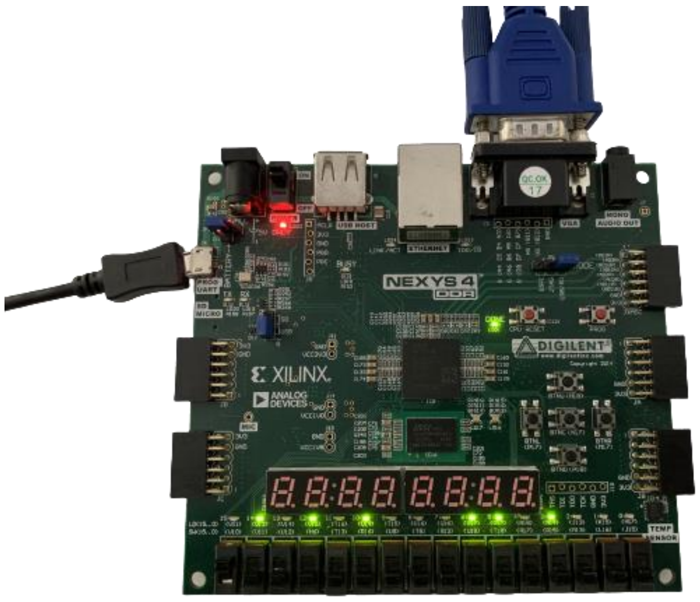


Figure 5.30 : Display of the letter "T" and "h" in binary format on the LEDs.

5.4 Improving the Frequency Based Algorithm With Two Thresholds

5.4.1 Modifying the C code for programming Ibex

In the previous titles, message embedding was performed on the least significant two bits of the high-frequency pixels. This title will examine how this method can be further developed. As mentioned before, it is aimed to increase the number of bits to be embedded for a pixel by determining more threshold values for a better result. To be specific, an extra threshold has been assigned in addition to the previous frequency-based image steganography code. Besides, the pixel is 0 if the sum of the two filter outputs is less than the smaller threshold, and 127 if it is between the two threshold values; if it is greater than either of these thresholds, it takes the 255. Necessary arrangements for this have been made in the encryption and decryption parts of the code, too. Furthermore, message embedding has been tested to be correct, and Appendix A is also visible for the entire code.

5.4.2 Simulations for the algorithm

After testing the C code of the frequency-based image steganography algorithm in the previous sections, tests and simulations of this algorithm are also carried out in the Vivado environment. This testing process is done similarly to the testing process of previous algorithms.

First of all, the C code is compiled using the GNU toolchain and a memory file is obtained. This file is included in the Vivado project. In addition, in order to be able to read this memory file in the file associated with the RAM in the project, the file name is written on the relevant line and the algorithm operation is performed with the reading of the file.

In order to read the pixel values in the simulation, write enable signal and ibex core's data output signal are added to the timing diagram, as was done before. And a little part of this simulation can be seen in Figure 5.31. Furthermore, as a result of this simulation, the image pixels in which the message is embedded in the final state are written to the "output.txt" file specified in the testbench.

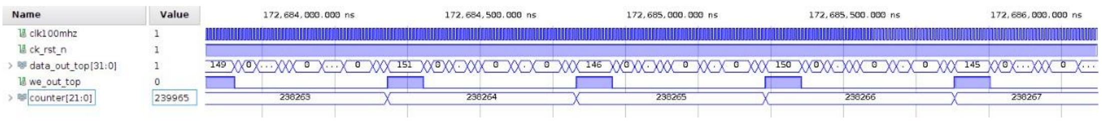


Figure 5.31: Simulation results for two thresholds case.

For this algorithm, like the others, the image in Figure 5.3 is used and the values of the encoded image pixels of this image are obtained as a result of Vivado simulation. Then, these pixel values are read with the MATLAB code mentioned in section 5.1.2 and displayed as an image with the dimensions of 256×256, and the image shown in Figure 5.32 below is obtained in the MATLAB output.

In other words, the following figure shows the stego image in which the message is embedded with the frequency-based image steganography method.

Figure 5.32: Encoded image for two thresholds case.

Since this message embedding process is done at an invisible level, the information encoded in this image must be decrypted. For this process, MATLAB codes in Appendix B are used. To perform the decryption operation, a key array containing the order information of the pixels selected by the LFSR method by the C code is required. Besides, a key array is required which specifies how many bits are embedded in which pixels as more information is embedded in high-frequency pixels. In the MATLAB code, pixels are visited one by one according to the elements of this LFSR key, and 1, 2, or 3-bit messages are collected from the pixels according to the high-frequency key. As a result of these processes, the output obtained when the collected messages are converted to ASCII and then converted to character value is as follows:

```
“This is the secret message.”
```

As a consequence, the frequency-based image steganography algorithm with two thresholds performed in the Vivado environment is verified to work correctly.

Apart from these, the cover image used in this algorithm is only 8-bit, a 4-bit image is not used. As it is known, in previous algorithms, the image was reduced to 4 bits and the operations in the monitoring phase with VGA were carried out. However, since there are two threshold values in this algorithm, working with a 4-bit image is not convenient in terms of security and detectability. For this reason, only an 8-bit image was worked on, and since an 8-bit image cannot be given to VGA, that step was not carried out. Therefore, tests and simulations were performed on MATLAB, C programming, and Vivado environment only.

5.5 Custom IP Design for Frequency Detection

5.5.1 Custom IP architecture in Vivado

As explained in the previous sections, all the changes made in the image so far have been written in C code using software. To develop and accelerate the project, it was decided to transfer some of the processes performed in software to hardware. For this reason, a Verilog code has been written to implement the Sobel filter hardware, the software algorithm of which is explained in section 5.3.1.

The new module named "itublaze_sobel" was connected to the system as a master, and thus, proper communication was ensured with the interconnect module that makes the inter-module connections. The inputs and outputs of the filter module, which are cross sectioned from the RTL schematic, are shown in Figure 5.33.

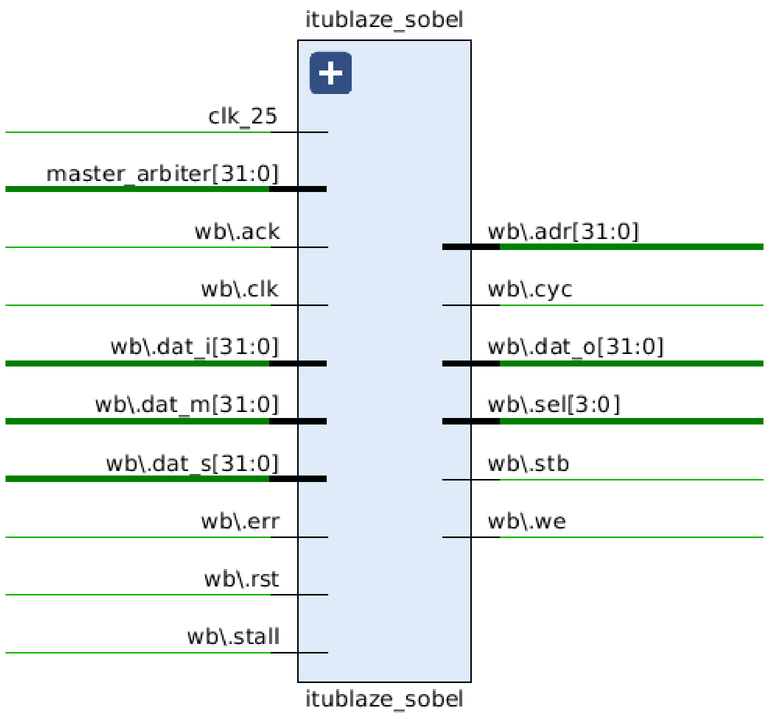


Figure 5.33 : Filter module from RTL .

RTL schematic including filter module can be seen in Appendix C. The task of this filter module is to apply 2D convolution to the image separately with two kernels using two separate 3x3 kernels required for the Sobel filter, and to mark various frequency regions in the image by adding the absolutes of the values obtained as a result of the convolution. Then, as in the software part, the filtered image will be reconstructed to create high and low frequency values according to the threshold value set as 9. If the pixel value is greater than or equal to 9, the pixel will be white, otherwise it will change to black.

For the module to be activated and start its operations, the input named master arbiter must have received the value defined for the filter module, as in the VGA module. The master arbiter value that will activate the filter module has been selected as 0x00000100. The master arbiter signal is a signal defined at the address of the first element of the array, which is defined by the pixel name in the C code. Therefore, assigning the value 0x00000100 to the first element of the pixel array in the C code activates the filter module. After the pixel values of the image to be filtered are written to the RAM, it is sufficient to activate the filter module as mentioned above, because this module is designed to read the image from the RAM and write it to the same addresses in the RAM after filtering.

A total of nine states are used to implement the operations to be performed in this module. Since the kernels used for filtering are in the form of 3x3 matrix, it is necessary to read nine-pixel values to calculate the filter output of a pixel. During these nine states, the relevant values are read from the RAM and the filter output value is calculated for a pixel. When the process is completed at the end of nine states, it is returned to the first state named "zero" and the process of writing the calculated value to RAM is performed in this state. As mentioned, when the filter module is activated using the "master_arbiter" input, the case block with the states starts to run. In the first state the convolution result is reset, and then the (0,0) indexed pixel of the 256x256 image is read by using the address variable defined in the module. This pixel value read is multiplied by the (0,0) indexed values of the two kernels to be used in the Sobel filter, and these values are saved separately. When it is passed to the next state, this time the (0,1) indexed pixel of the image is read and added to the results saved in the previous state by multiplying the kernels with the same indexed elements. By proceeding in this way, the kernel result in the 3x3 area in the upper left corner of the image is calculated at the end of the ninth state. Then it is returned to the first state and the absolute values of the convolution results calculated for the two separate kernels are summed and its value after threshold is written to the address corresponding to the index of the image (1,1) in RAM. If the calculated value less than 8, the value of 0 is written to the RAM, otherwise value of 15 is written. The code of the part where the final pixel value is written to RAM can be seen in Figure 5.34. Here, "add_mul_result_final" is the calculated filter output value and "wb.dat_o" is the module output to be written to RAM. In addition, in the state named "zero", the "wb.we" signal which means write enable, is given a value of 1 to ensure that the write operation is performed.

```
if (state==zero)
begin
   wb.adr            <=  (kernel_addr_counter * 4) + 'h11014;
   wb.we             <= 1'b1;
   wb.sel             <= 4'b1111;


  if (add_mul_result_final<8)

     wb.dat_o[15:0]    <= 0;

  else

     wb.dat_o[15:0]    <= 15;


  if (wb.ack)
    begin
       data_ram_filter  =   wb.dat_i[11:0];
       wb.stb          <=   1'b0;
       wb.cyc          <=   1'b0;

    end
  end

else
 begin
   wb.adr      <=  (address_result * 4) + 'h11014;
   wb.we       <= 1'b0;
   wb.sel      <= 4'b1111;
```

Figure 5.34 : Verilog code for writing calculated values to RAM.

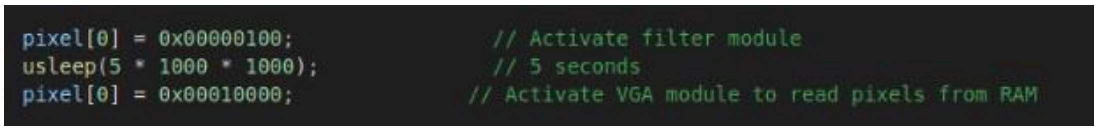
To calculate the second kernel result, kernels are shifted to the right by one column so that the indexes of the kernels (0,0) correspond to the pixels of the image (0,1) and the same operations are repeated afterwards. With this method, a 2D convolution is applied on the entire image by navigating the kernels over the image. The Verilog code written for the filter module is available in Appendix D.

5.5.2 Visualization of the filtered image using a custom IP

In this section, it will be explained how the output of the filter module is given to the VGA output and displayed on the monitor. As explained in section 5.1.1, the filter module reads the original image pixels from RAM and rearranges them according to the threshold value after the filtering process, and then writes the output image to

RAM. And for this module to start working, the "master_arbiter" input should take the value of 0x00000100. A C code has been written so that the Custom IP output can be displayed on the monitor.

In the C code, starting from the 1 indexed address of the RAM, 256x256 4-bit image pixels are written to the RAM, respectively. Afterwards, for the filter module to work, 0x00000100 is written to 0 indexed address of the RAM, so that the "master_arbiter" gets this value and the filter module is activated. A certain amount of time is waited by using the usleep() function for the filtering process to be completed. Then, the VGA module is activated by giving the value 0x00010000 to the 0 indexed address of the RAM. Thus, the filtered image written in RAM is displayed on the monitor. The related part of code in which the described operations are performed is shown in Figure 5.35.



```
pixel[0] = 0x00000100;              // Activate filter module
usleep(5 * 1000 * 1000);            // 5 seconds
pixel[0] = 0x00010000;            // Activate VGA module to read pixels from RAM
```

Figure 5.35 : C code for monitoring filter module output.

After the C code was compiled and the memory file was obtained, this file is added to the Vivado project and bitstream is produced. After the generated bitstream is loaded onto the card, the image obtained on the monitor is given in Figure 5.36.

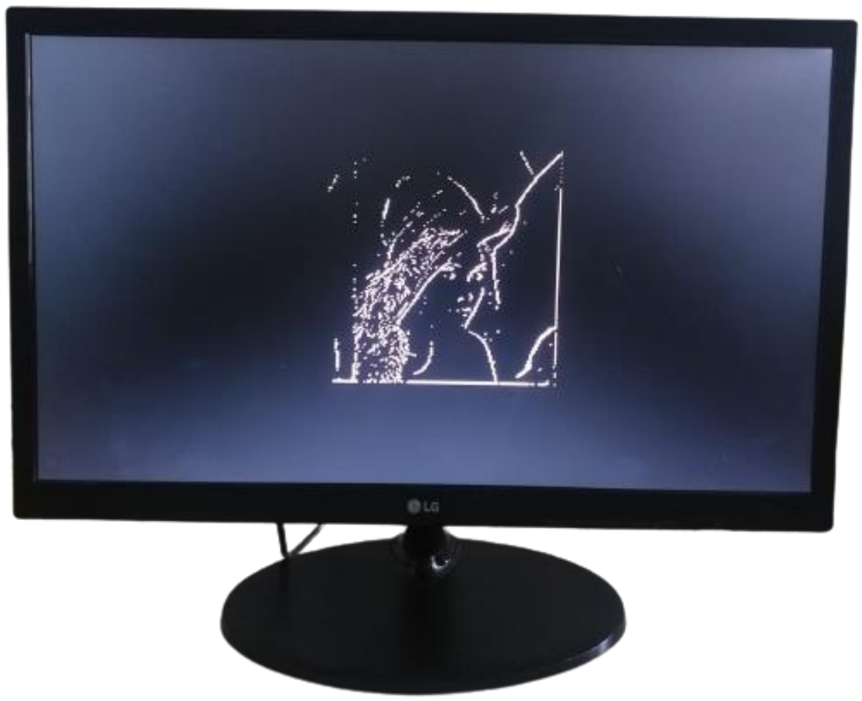


Figure 5.36 : Filter module output given to the monitor.

As can be seen, high-frequency pixels forming the edge regions in the original image are white, while the regions low frequency are black. This shows that the filter module added to the project is working properly and as desired.

The image steganography algorithm using high frequency regions and the display of the results on monitors and LEDs were explained in section 5.3. Frequency based image steganography results can be obtained by using Custom IP if the part in which the original image is filtered is removed in the C code written for this part, and a line of code with the filter module activated is added instead.

6. REALISTIC CONSTRAINTS AND CONCLUSIONS

6.1 Practical Application of this Project

For privacy reasons, the undetectability of the secret message while communicating has always been a significant issue for the establishments and institutions where privacy is a priority. In this project, it was aimed to improve one of the common methods used for this purpose named as LSB algorithm to more advanced algorithms where cryptography and steganography are used together and to use RISC-V processor to speed up the process. The methodology which was followed is based on the MATLAB simulation of the created algorithms, synthetization of the C codes, hardware replacement of the slow-running filtering part of the software application, and FPGA implementation of the RISC-V based processors. The combination of steganography and cryptography will strengthen security since the secret message is not only encrypted, but the existence of the communication is also hidden.

6.2 Realistic Constraints

6.2.1 Social, environmental and economic impact

Information shared in an environment mutually, importance for secure communication is increasing day by day. Despite the development of methods that provide security, cyber attacks against them are also increasing. Therefore, new methods for secure communication continue to be sought and developed. For this reason, a study that provides both the invisibility and security of information transition will be of great benefit to the society in this area. Furthermore, using an open-source processor which is cost-free and developing the usage areas of it will have an economic impact on future studies.

6.2.2 Cost analysis

The cost of the project consists of the FPGA development board, the monitor used for display and the cables required for the connections. The FPGA board is provided by the faculty, other products and computers used during the project belong to the students themselves. The free versions for the Vivado environment and the license provided by the university for MATLAB were used. In addition, the operating system necessary for the applications is also free as it uses open sources.

6.2.3 Standards

The work to be done in this project complies with several different standards. The hardware implementation will be IEEE compliant. In addition, while writing the application of image steganography algorithms, it is aimed to comply with the C programming language standards. Furthermore, engineering code of conduct was taken into consideration during the studies.

6.2.4 Health and safety concerns

The products used in the project are not harmful to any human being. Likewise, the applications made during the project do not pose any safety and health risks.

6.3 Future Work and Recommendations

In the project, studies were carried out to develop the image steganography method that can be used to provide a secure communication system. The process, which was completed at the last stage of the project, of dividing the frequency regions in the image into three and embedding different numbers of messages into them can be further developed to determine how many bits of messages will be embedded according to the pixel frequency. Thus optimizing the total message length that can be embedded in the image. In addition, it has been tested in the project that the message is embedded in the image using the relevant algorithms and the message is correctly extracted from the image. However, in order to test a proper communication system, the transmitter part where the message is embedded in the image and the receiver part where the message is extracted from the image can be realized separately and real-time communication between these two parties can be achieved.

REFERENCES

[1] A. Yahya, "Introduction to Steganography," in Steganography techniques for digital images, Cham: Springer International Publishing, 2019, pp. 1–2.

[2] M. K. Harahap and N. Khairina, "Dynamic steganography least significant bit with stretch on pixels neighborhood," *Journal of Information Systems Engineering and Business Intelligence*, vol. 6, no. 2, p. 151, 2020.

[3] D. Ghosh, A. K. Chattopadhyay, K. Chanda, and A. Nag, "A Secure Steganography Scheme Using LFSR" Advances in Intelligent Systems and Computing, p. 716, 2019.

[4] V. Kalaichelvi, P. Meenakshi, P. Vimala Devi, H. Manikandan, P. Venkateswari, and S. Swaminathan, "A stable image steganography: A novel approach based on modified RSA algorithm and 2–4 least significant bit (LSB) technique," Journal of Ambient Intelligence and Humanized Computing, vol. 12, no. 7, pp. 7239–7242, 2020.

[5] M. A. Al-Shabi, "A survey on symmetric and asymmetric cryptography algorithms in information security," International Journal of Scientific and Research Publications (IJSRP), vol. 9, no. 3, p. 1, 2019.

[6] A. Yahya, "Steganography Techniques," in Steganography techniques for digital images, Cham: Springer International Publishing, 2019, p. 14.

[7] D. A. Singer, "Secrets of Linear Feedback Shift Registers," p. 3.

[8] S. R. Joshi and R. Koju, "Study and comparison of edge detection algorithms," 2012 Third Asian Himalayas International Conference on Internet, 2012.

[9] "Specification for the: WISHBONE System-on-Chip (SoC) Interconnection Architecture for Portable IP Cores", pp. 24-41, Sep. 2002.

[10] Erfan Gholizadehazari, "An FPGA Implementation of a RISC-V Based SoC System with Custom Instruction Set for Image Processing Applications", Istanbul Technical University, Science and Technology Institute, July 2021, Turkey.

[11] "Figure 2. original image (512x512 pixels). - researchgate.net." [Online]. Available: https://researchgate.net/figure/Original-image-512x512-pixels_fig2_308838117. [Accessed: 13-Oct-2021].

[12] RISC-V, 13-Dec-2021. [Online]. Available: https://riscv.org/. [Accessed: 10-Jan-2022].

[13] "RISC-V Exchange: Cores & socs - RISC-V international," *RISC-V*, 16-Sep-2021. [Online]. Available: https://riscv.org/exchange/cores-socs/. [Accessed: 10-Nov-2021].

[14] A. Partow, Primitive polynomial list. [Online]. Available: https://www.partow.net/programming/polynomials/index.html. [Accessed: 22-Oct-2021].

[15] A Unified Proximity Algorithm with Adaptive Penalty for Nuclear Norm Minimization - Scientific Figure on ResearchGate. [Online]. Available: https://www.researchgate.net/figure/Original-512-512-images-Lena-Pirate-Cameraman-with-full-rank-first-column_fig1_336446112. [Accessed: 15-Mar-2022].

[16] Lena Söderberg (256x256 JPEG image, 77 kbytes). [Online]. Available: https://www.researchgate.net/figure/Lena-Soederberg-256x256-JPEG-image-77-Kbytes_fig8_259296904. [Accessed: 05-Oct-2021].

[17] Mandrill image on synthetic surface - researchgate.net. [Online]. Available: https://www.researchgate.net/figure/Mandrill-Image-on-Synthetic-Surface_fig1_3410901. [Accessed: 05-Oct-2021].

[18] 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: 15-Nov-2021].

[19] lowRISC, GitHub. [Online]. Available: https://github.com/lowRISC/ibex. [Accessed: 10-Nov-2021].

[20] Digilent, Nexys 4 DDR, https://reference.digilentinc. com/reference/programmable-logic/nexys-4-ddr/reference-manual.

[21] Digilent, Nexys 4 DDR XDC File,, https://github.com/Digilent/ digilent-xdc/blob/master/Nexys-4-DDR-Master.xdc.

[22] "File:wishbone Interface.svg," Wikipedia. [Online]. Available: https://en.wikipedia.org/wiki/File:Wishbone_Interface.svg. [Accessed: 12-Dec-2021].

APPENDICES

APPENDIX A

```
#define len_img_mat 256*256 // the length of the img_mat
#define len_mes_str 27 // the length of the string message
#define len_mes_bin 8*len_mes_str // the length of the binary message
#define n_bit_lfsr 16

int dec2bin(int dec_val)
{
    int bin = 0, osn = 1, c = 0;
    while (dec_val!= 0){
        c = dec_val % 2;
        bin += c * osn;
        osn *= 10;
        dec_val = dec_val/2;
    }
    return bin;
}

int bin2dec(int bin_val){
    int dec = 0;
    int subtrahend = 10000000;
    for (int j = 0; j < 8; j++){
        int counter = 0;
        int digit_val = 1;

        while (bin_val >= subtrahend){
            bin_val -= subtrahend;
            counter++;
        }

        for (int i = 0; i < (7-j); i++)
            digit_val *= 2;

        dec += digit_val * counter;

        subtrahend /= 10;
        }
    return dec;
}

int lfsr(int random_number){
    int a[n_bit_lfsr] = {0};
    int index = (n_bit_lfsr-1), bit = 0;
    while (random_number != 0){
        bit = random_number % 2;
        a[index] = bit;
        index -= 1;
        random_number /= 2;
    }
    int b[n_bit_lfsr] = {0};
    unsigned c;
    int flag = 0, count = 0;

    c = ((a[0] ^ (a[2] ^ (a[11] ^ a[15]))));
    b[0] = c;

    for (int i = 0; i < (n_bit_lfsr-1); i++)
        b[i+1] = a[i];
```

```c
    int new_number = 0;
    int counter = 1;
    int mul = 1;

    for (int j = (n_bit_lfsr-1); j >= 0 ; j--){
        if (counter == 1){
            new_number += b[j] * counter;
        }
        else{
            mul *= 2;
            new_number += b[j] * mul;
        }
        counter += 1;
    }
        return new_number;
}


int main() {
    int img_mat[len_img_mat] = {162, 162, 162, 163, 165,…}
    // image matrix
    int encoded_img_mat[len_img_mat] = {0}; // encoded image matrix

    // ---------------------------- Sobel Filter ----------------------------
    int clone_img_mat[256][256];
    int gx, gy, sum;

    for(int y = 0; y < 256; y++){
        for(int x = 0; x < 256; x++)
            clone_img_mat[y][x] = 0;
    }

    for(int y = 1; y < 255; y++){
        for(int x = 1; x < 255; x++){
            gx = -img_mat[(y-1)*256 + x-1] - 2*img_mat[y*256 +x-1] - img_mat[(y+1)*256
+x-1] +
            img_mat[(y-1)*256+x+1] + 2*img_mat[y*256+x+1] + img_mat[(y+1)*256+x+1];
            gy = img_mat[(y-1)*256+x-1] + 2*img_mat[(y-1)*256+x] + img_mat[(y-
1)*256+x+1] -
            img_mat[(y+1)*256+x-1] - 2*img_mat[(y+1)*256+x] - img_mat[(y+1)*256+x+1];

            if (gx < 0)
                gx -= 2*gx;
            if (gy < 0)
                gy -= 2*gy;
            sum = gx + gy;
            if (sum > 255)
                sum = 255;

            clone_img_mat[y][x] = sum;
        }
    }
    // ----------------------------------------------------------------------
    int threshold_1 = 85;
    int threshold_2 = 170;

    for(int y = 0; y < 256; y++){
        for(int x = 0; x < 256; x++){
            if (clone_img_mat[y][x] < threshold_1)
                clone_img_mat[y][x] = 0;
            else if (threshold_1 <= clone_img_mat[y][x] < threshold_2)
                clone_img_mat[y][x] = 127;
            else
                clone_img_mat[y][x] = 255;
        }
    }

    // ---------------------------------------------------------------------
    char message_string[len_mes_str] = "This is the secret message."; //message (27 x 8
= 216-bit)

    // ---------------------------------------------------------------------
    int message_ascii[len_mes_str] = {0};
    int message_binary[len_mes_str] = {0};

    for (int i = 0; i < len_mes_str; i++)
        message_ascii[i] = message_string[i];

    for (int j = 0; j < len_mes_str; j++)
        message_binary[j] = dec2bin(message_ascii[j]);

    // ---------------------------------------------------------------------
    int arr_message_bin[len_mes_bin] = {0};
    int index_counter = 0;

    for (int m = 0; m < len_mes_str; m ++){
        int char_bin = message_binary[m];
        int subt = 10000000;
        for (int x = 0; x < 8; x++){
            int message_bit = 0;
            if (char_bin >= subt){
                char_bin -= subt;
                message_bit = 1;
            }
            arr_message_bin[index_counter] = message_bit;
            index_counter++;
            subt /= 10;
        }
    }

    // ---------------------------------------------------------------------
    int key_freq[len_mes_bin][2] = {0};
    int key_with_encoded_pixels[len_mes_bin][2] = {0};
    int rand_num = 121; // choose a random number to start the process of generating
random number
    int rand_pixel = lfsr(rand_num);
    int edge_pixel_i, edge_pixel_j;
    int key_counter = 0;
    int high_freq_counter = 0;

    for(int k = 0; k < len_mes_bin; k++) {
        key_freq[key_counter][1] = rand_pixel;
        key_with_encoded_pixels[key_counter][0] = rand_pixel;
        edge_pixel_i = rand_pixel / 256;
        edge_pixel_j = rand_pixel % 256;
        int bin_pixel = dec2bin(img_mat[rand_pixel]);

        // ----------------------------- Encryption -----------------------------
        int lsb = bin_pixel & 1; // "AND" with '1' to get the value of LSB
        if (clone_img_mat[edge_pixel_i][edge_pixel_j] == 0){ // if not an edge pixel,
embed on LSB only
            bin_pixel = bin_pixel - lsb + arr_message_bin[k];
            key_freq[key_counter][0] = 1;
            key_freq[key_counter][1] = rand_pixel;
        }
        else if (clone_img_mat[edge_pixel_i][edge_pixel_j] == 127){
            int second_lsb = (bin_pixel / 10) & 1;
            bin_pixel = bin_pixel - lsb + arr_message_bin[k+1] - 10*second_lsb +
10*arr_message_bin[k];
            k += 1;
            key_freq[key_counter][0] = 2;
            key_freq[key_counter][1] = rand_pixel;
            high_freq_counter += 1;
        }
        else{
            int second_lsb = (bin_pixel / 10) & 1;
            int third_lsb = (bin_pixel / 100) & 1;
            bin_pixel = bin_pixel - lsb + arr_message_bin[k+2] - 10*second_lsb +
10*arr_message_bin[k+1] - 100*third_lsb + 100*arr_message_bin[k];
            k += 2;
            key_freq[key_counter][0] = 3;
            key_freq[key_counter][1] = rand_pixel;
            high_freq_counter += 2;
        }

        // ---------------------------------------------------------------------
        int encoded_dec_pixel = bin2dec(bin_pixel);
        key_with_encoded_pixels[key_counter][1] = encoded_dec_pixel;
        key_counter += 1;
        rand_pixel = lfsr(rand_pixel);
    }

    int len_sorted_key = len_mes_bin - high_freq_counter;
    int sorted_key_with_encoded_pixels[len_sorted_key][2];

    for (int z = 0; z < len_sorted_key; z++){
        for (int x = 0; x < 2; x++)
            sorted_key_with_encoded_pixels[z][x] = key_with_encoded_pixels[z][x];
    }

    int a, b;
    for (int i = 0; i < len_sorted_key; ++i){
        for (int j = i + 1; j < len_sorted_key; ++j){
            if (sorted_key_with_encoded_pixels[i][0] >
sorted_key_with_encoded_pixels[j][0]){
                a = sorted_key_with_encoded_pixels[i][0];
                b = sorted_key_with_encoded_pixels[i][1];
                sorted_key_with_encoded_pixels[i][0] =
sorted_key_with_encoded_pixels[j][0];
                sorted_key_with_encoded_pixels[i][1] =
sorted_key_with_encoded_pixels[j][1];
                sorted_key_with_encoded_pixels[j][0] = a;
                sorted_key_with_encoded_pixels[j][1] = b;
            }
        }
    }

    int lfsr_counter = 0;
    for(int k = 0; k < len_img_mat; k++) {
        if (k == sorted_key_with_encoded_pixels[lfsr_counter][0]){
            encoded_img_mat[k] = sorted_key_with_encoded_pixels[lfsr_counter][1];
            if (lfsr_counter != len_mes_bin -1){lfsr_counter = lfsr_counter + 1;}
        }
        else
            encoded_img_mat[k] = img_mat[k];
    }
    int o = encoded_img_mat[5];
    return o;
}
```

APPENDIX B

```
clc, clear, close all;

% --- Reading the txt file that includes pixel values
fileID1 = fopen('output_file_256_freq_two_th.txt','r');
IN = fscanf(fileID1, '%d\n');

% --- Reformat input image to a 256x256 matrix
stego_img = uint8(transpose(reshape(IN,[256 256])));
figure,
imshow(stego_img, [0 255]);
title('Stego Image')

fclose(fileID1);

% --- Decryption for frequency based algotrithm
fileID2 = fopen('key_freq_1.txt','r'); % get the key of random pixel produced using
LFSR
random_indexes = fscanf(fileID2, '%d\n');
len = length(random_indexes);
fclose(fileID2);

fileID3 = fopen('key_freq_0.txt','r'); % get the key of frequency key
key_freq = fscanf(fileID3, '%d\n');
fclose(fileID3);

M = 'secret.txt'; % Message is read in order to get its size
secret = fopen(M,'rb'); % open secret file
[M,L] = fread(secret,'ubit1'); % read secret file as bin array

[n1,m1] = size(stego_img);

ct = 1;
temp = M;
% --- navigating through FLSF pixel numbers in order to get the secret message
for k = 1:len
    pixel_number = random_indexes(k) + 1;

    % --- find the indexes of image to which the FLSR numbers correspond
    if mod(pixel_number,n1)==0
        image_matrix_index_i_2 = floor(pixel_number/n1);
        image_matrix_index_j_2 = n1;
    else
        image_matrix_index_i_2 = floor(pixel_number/n1)+1;
        image_matrix_index_j_2 = mod(pixel_number,n1);
    end

    % --- store the extracted bits in temp array
    if key_freq(k,1) == 1
        temp(ct,1) =
mod(stego_img(image_matrix_index_i_2,image_matrix_index_j_2),2);
    else
        binary_pixel =
dec2bin(stego_img(image_matrix_index_i_2,image_matrix_index_j_2)); % convert the
pixel value to the binary form
        if key_freq(k,1) == 3
            third_lsb = bin2dec(binary_pixel(end-2)); % take the second last
element of the array
            temp(ct,1) = third_lsb;
            ct = ct + 1;
        end
        second_lsb = bin2dec(binary_pixel(end-1)); % take the second last element
of the array
        temp(ct,1) = second_lsb;
        ct = ct + 1;
        temp(ct,1) =
mod(stego_img(image_matrix_index_i_2,image_matrix_index_j_2),2);
    end
    ct = ct + 1;
end

% --- converting extracted bit array into a text
c = 0;
j = 8;

fileID = fopen('output_freq_two.txt', 'w'); % produces text will be written into a
txt file

% --- group bits by 8, and convert it to char
for i = 1:L
    c = c + temp(i,1)*(2^(j-1));
    j = j -1;
    if j == 0
        j = 8;
        fwrite(fileID, c, 'char');
        c = 0;
    end
end
```

APPENDIX C

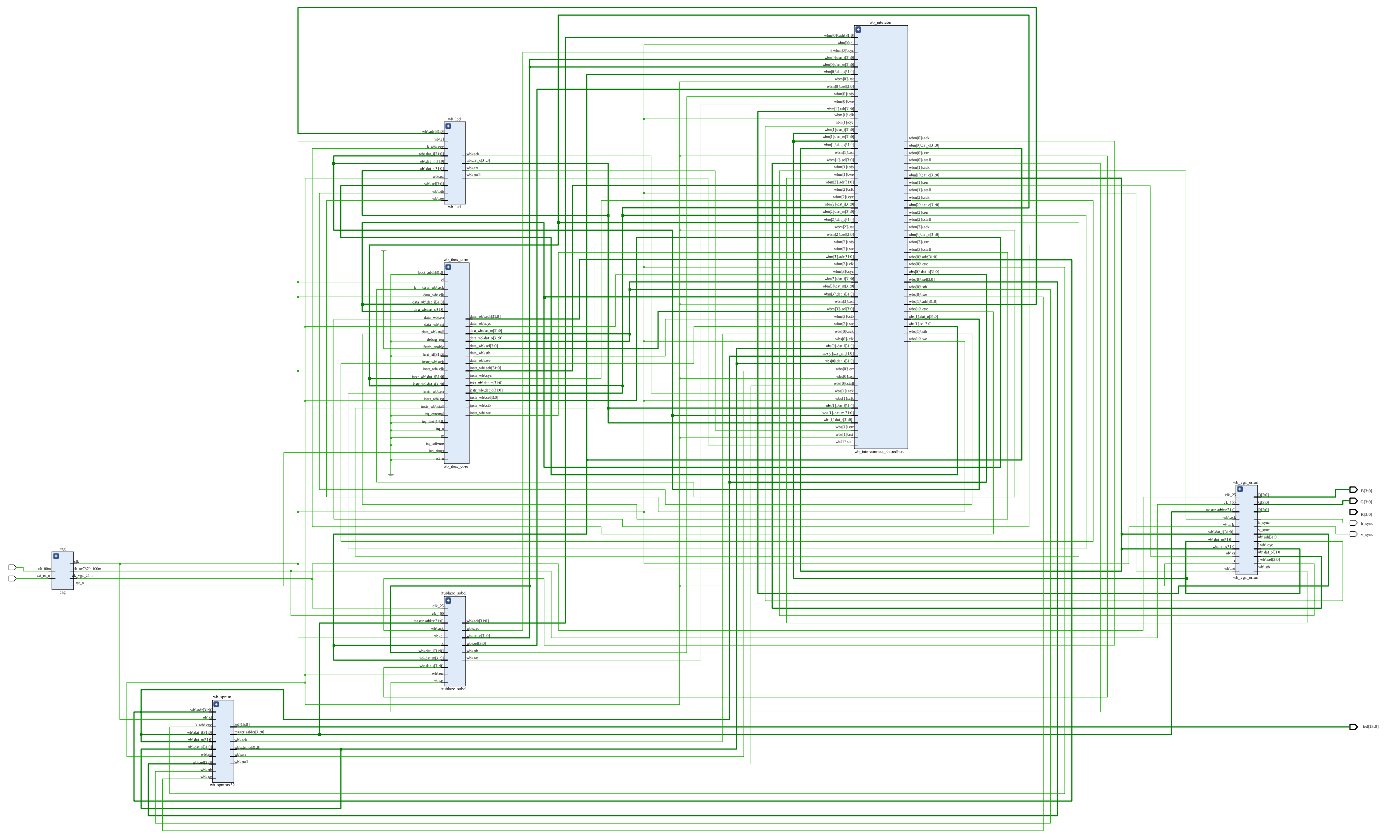


Figure C.1 : RTL schematic including filter module.

APPENDIX D

```
/* ItuBlaze Added wishbone Sobel Filter module */

`default_nettype none

module wb_sharpen_erfan(

   input  wire  [31:0] master_arbiter,
   input wire clk_25,
   input wire clk_100,

   wb_if.master         wb
   );


reg  [3:0]  state=0;
reg  signed [15:0] conv_result_x;
reg  signed [15:0] conv_result_y;
reg  [16:0] kernel_addr_counter = 'd257;
reg  [16:0] kernel_counter;
reg  [9:0]  kernel_counter_i;
reg  [16:0] adress_ram_filter;
reg  [11:0] data_ram_filter;

wire signed [15:0] add_mul_result_x;
wire signed [15:0] add_mul_result_y;
wire signed [15:0] add_mul_result_final;
reg  signed [15:0] kernel_const_x;
reg  signed [15:0] kernel_const_y;
wire  [16:0] address_result;
reg  signed [16:0] offset_addr;

parameter zero=0, one=1, two=2, three=3, four=4, five=5,
six=6, seven=7, eight=8;


reg [3:0] clock_out_filter;



 always @(posedge wb.clk or posedge wb.rst)
  begin
   if (wb.rst)
       wb.cyc <= 1'b0;
 else
  begin

    if ( master_arbiter[8]=='h1) // if master_arbiter =
0x00000100
     begin
       wb.cyc      <= 1'b1;
       wb.stb      <= 1'b1;
     end
    else
    begin
       wb.cyc      <= 1'b0;
       wb.stb      <= 1'b0;
     end

    if (state==zero)
    begin
       wb.adr            <=  (kernel_addr_counter * 4) +
'h11014;
       wb.we             <= 1'b1;
       wb.sel             <= 4'b1111;


      if (add_mul_result_final<8)

         wb.dat_o[15:0]    <= 0;

      else

         wb.dat_o[15:0]    <= 15;


      if (wb.ack)
        begin
           data_ram_filter   =   wb.dat_i[11:0];
           wb.stb           <=   1'b0;
           wb.cyc           <=   1'b0;

        end
      end

    else
     begin
       wb.adr      <=  (address_result * 4) + 'h11014;
       wb.we       <= 1'b0;
       wb.sel      <= 4'b1111;

      if (wb.ack)
        begin
           data_ram_filter   =   wb.dat_i[11:0];
           wb.stb           <=   1'b0;
           wb.cyc           <=   1'b0;
        end
      end

   end
 end


assign add_mul_result_x = (data_ram_filter * kernel_const_x) +
conv_result_x;
assign add_mul_result_y = (data_ram_filter * kernel_const_y) +
conv_result_y;
assign add_mul_result_final = (add_mul_result_x[15] & (-
1)*add_mul_result_x) + (~add_mul_result_x[15] &
add_mul_result_x) + (add_mul_result_y[15] & (-
1)*add_mul_result_y) + (~add_mul_result_y[15] &
add_mul_result_y) ;

assign  address_result = kernel_addr_counter + offset_addr;

always @(posedge clk_25)
 begin
   case (state)
                  zero:
                     begin
                        conv_result_x = 0;
                        conv_result_y = 0;

                        if ( kernel_counter_i > 254)
                          begin
                            kernel_counter_i = 0;
                            kernel_addr_counter =
kernel_addr_counter + 1;
                          end
                        else
                         begin
                            kernel_counter_i =
kernel_counter_i + 1 ;
                             if (kernel_addr_counter ==
'd65281)
                                 kernel_addr_counter =
'd65281;
                             else
                                 kernel_addr_counter =
kernel_addr_counter + 1;
                          end

                        offset_addr = -257;
                        kernel_const_x = -1 ;
                        kernel_const_y = 1 ;
                        state = one;
                     end

                  one:
                     begin
                        offset_addr = -256;
                        conv_result_x = add_mul_result_x;
                        conv_result_y = add_mul_result_y;
                        kernel_const_x = 0 ;
                        kernel_const_y = 2 ;
                        state = two;
                     end

                  two:
                      begin
                        offset_addr = -255;
                        conv_result_x = add_mul_result_x;
                        conv_result_y = add_mul_result_y;
                        kernel_const_x = 1;
                        kernel_const_y = 1;
```

```
      state = three;
    end

three:
    begin
      offset_addr = -1;
      conv_result_x = add_mul_result_x;
      conv_result_y = add_mul_result_y;
      kernel_const_x = -2 ;
      kernel_const_y = 0 ;
      state = four;
    end

four:
    begin
      offset_addr = 0;
      conv_result_x = add_mul_result_x;
      conv_result_y = add_mul_result_y;
      kernel_const_x = 0 ;
      kernel_const_y = 0 ;
      state = five;
    end

five:
    begin
      offset_addr = 1;
      conv_result_x = add_mul_result_x;
      conv_result_y = add_mul_result_y;
      kernel_const_x = 2 ;
      kernel_const_y = 0 ;
      state = six;
    end

six:
    begin
      offset_addr = 255;
      conv_result_x = add_mul_result_x;
      conv_result_y = add_mul_result_y;
      kernel_const_x = -1 ;
      kernel_const_y = -1 ;
      state = seven;
    end

seven:
    begin
      offset_addr = 256;
      conv_result_x = add_mul_result_x;
      conv_result_y = add_mul_result_y;
      kernel_const_x = 0 ;
      kernel_const_y = -2 ;
      state = eight;
    end

eight:
    begin
      offset_addr = 257;
```

```
                            conv_result_x = add_mul_result_x;
                            conv_result_y = add_mul_result_y;
                            kernel_const_x = 1 ;
                            kernel_const_y = -1 ;
                            state = zero;
                          end


                 endcase

         end

     endmodule

     `resetall
```

CURRICULUM VITAE

Name Surname	**:** Yaşar Utku Alçalar
Place and Date of Birth	**:** İstanbul / 15.11.1999
E-Mail	**:** utkualcalar17@gmail.com

Yaşar Utku Alçalar completed his primary and high school education in İstanbul. He is currently a senior year student at Electronics and Communication Engineering in Istanbul Technical University. He completed his internships in ASELSAN A.Ş. and Istanbul Technical University Embedded System Design Laboratory.

CURRICULUM VITAE

Name Surname	**:** Sinem Başak Kapucu
Place and Date of Birth	**:** Istanbul / 13.03.1999
E-Mail	**:** basak.kapucu.13@gmail.com

Sinem Başak Kapucu completed her primary and high school education in İstanbul. She is currently a senior year student at Electronics and Communication Engineering in Istanbul Technical University and works part-time in ABE Technology. She completed her internships in ABE Technology and Istanbul Technical University Embedded System Design Laboratory.

CURRICULUM VITAE

Name Surname	**:** Burcu Türk
Place and Date of Birth	**:** Tokat / 16.07.1999
E-Mail	**:** bturk5407@gmail.com

Burcu Türk completed her primary and high school education in Tokat. She is currently a senior year student at Electronics and Communication Engineering in Istanbul Technical University and works part-time in Turkish Aerospace Industries. She completed her internships in ASELSAN A.Ş. and Istanbul Technical University Embedded System Design Laboratory.