ISTANBUL TECHNICAL UNIVERSITY ★ GRADUATE SCHOOL

AN FPGA IMPLEMENTATION OF A RISC-V BASED SOC SYSTEM WITH CUSTOM INSTRUCTION SET FOR IMAGE PROCESSING APPLICATIONS

M.Sc. THESIS

Erfan GHOLIZADEHAZARI

Department of Electronic and Communication Engineering

Electronic Engineering Programme

JULY 2021

ISTANBUL TECHNICAL UNIVERSITY ★ GRADUATE SCHOOL

AN FPGA IMPLEMENTATION OF A RISC-V BASED SOC SYSTEM WITH CUSTOM INSTRUCTION SET FOR IMAGE PROCESSING APPLICATIONS

M.Sc. THESIS

Erfan GHOLIZADEHAZARI
(504171292)

Department of Electronic and Communication Engineering

Electronic Engineering Programme

Thesis Advisor: Prof. Dr. Sıddıka Berna Örs Yalçın

JULY 2021

İSTANBUL TEKNİK ÜNİVERSİTESİ ★ LİSANSÜSTÜ EĞİTİM ENSTİTÜSÜ

GÖRÜNTÜ İŞLEME UYGULAMALARI İÇİN ÖZEL KOMUT SETİNE SAHİP RISC-V TABANLI BİR SOC SISTEMİNİN FPGA GERÇEKLEMESİ

YÜKSEK LİSANS TEZİ

Erfan GHOLIZADEHAZARI
(504171292)

Elektronik ve Haberleşme Mühendisliği Anabilim Dalı

Elektronik Mühendisliği Programı

Tez Danışmanı: Prof. Dr. Sıddıka Berna Örs Yalçın

TEMMUZ 2021

Erfan GHOLIZADEHAZARI, **a M.Sc.** student of ITU **Graduate School** student ID 504171292, successfully defended the **thesis** entitled "**AN FPGA IMPLEMENTATION OF A RISC-V BASED SOC SYSTEM WITH CUSTOM INSTRUCTION SET FOR IMAGE PROCESSING APPLICATIONS**", which he/she prepared after fulfilling the requirements specified in the associated legislations, before the jury whose signatures are below.

Thesis Advisor :	**Prof. Dr. Sıddıka Berna Örs Yalçın** Istanbul Technical University	.....
Jury Members :	**Prof. Dr. Sıddıka Berna Örs Yalçın** Istanbul Technical University	.....
	Prof. Dr. Ender Mete Ekşioğlu Istanbul Technical University	.....
	Prof. Dr. Nizamettin Aydın Yıldız Technical University	.....

Date of Submission : 8 June 2021
Date of Defense : 7 July 2021

To my family

FOREWORD

The base idea of this thesis is about Instruction Extension of RISC-V Processor for Laplacian filter implementation. It mainly consists of design of an embedded processor suitable for different practical applications. The embedded processor is supposed to be implemented in FPGA. Moreover, a complete image processing system including peripherals such as cameras and monitors will be designed around the proposed processor. This thesis is done for upon completion of Master degree at the Electronics and Communication Engineering Department, Istanbul Technical University (ITU).

I would like to thank my advisor, Prof. Dr. Berna Ors for her sincere guidance during my M.Sc studies.

JULY 2021 Erfan GHOLIZADEHAZARI

TABLE OF CONTENTS

Page

ABBREVIATIONS

ALU : Arithmetic Logic Unit
AXI : Advanced Extensible Interface
CPU : Central Processing Unit
DFD : Driver Fatigue Detection
DSP : Digital Signal Processor
FF : Flip Flop
FPGA : Field-programmable gate array
FPS : Frame Per Second
GCC : GNU Compiler Collection
GPIO : General-Purpose Input/Output
GPU : Graphics Processing Unit
HDMI : High-Definition Multimedia Interface
ISA : Instruction Set Architecture
IP : Intellectual Property
LED : Light Emitting Diode
LUT : Look Up Table
MAC : Multiply-Accumulate
RAM : Random Access Memory
RGB : Red-Green-Blue
RISC : Reduced Instruction Set Computer
R/W : Read/Write
SoC : System on Chip
VGA : Video Graphics Array
VHDL : Very High Speed Integrated Circuit Hardware Description Language

SYMBOLS

$ Terminal Command

LIST OF TABLES

Page

LIST OF FIGURES

Page

AN FPGA IMPLEMENTATION OF A RISC-V BASED SOC SYSTEM WITH CUSTOM INSTRUCTION SET FOR IMAGE PROCESSING APPLICATIONS

SUMMARY

Nowadays, reliable and fast hardware systems for image processing are playing an important role in our lives. When it comes to decision making, implemented an image processing algorithm on hardware is fast and reliable. For example, these days driver fatigue detection systems are very important in the transportation sector. Most people are victims of drowsiness while driving, simply after a too-short night's sleep, during long journeys, or altered physical condition. Hence, there is a need for an alarming system implemented in the transportation units to avoid horrible accidents caused by fatigue and yawning of the driver. Embedded systems are developed specifically for the purpose of providing low-cost performance requirements such as speed, accuracy, and reliability. By definition, it contains application-specific hardware and software components. However, as the target system grows, the need for workforce and experience diversity for the design of the system increases. Therefore, the user control of the system becomes more difficult. It is very important that software support for hardware is provided in order to be able to use a system easily, to update it if necessary, and to be able to operate in accordance with other embedded systems. Many applications can be implemented with application-specific inexpensive hardware units. However, the interest still is toward general-purpose, expensive solutions designed to support different applications because of the lack of a software development environment. On the other hand, the application areas of embedded systems can be extended by supporting the embedded systems with appropriate software and by developing efficient signal processing and decision-making algorithms specific to these systems. The increase of embedded systems where researchers and users can easily develop on them means that many applications can become cheap, practical, and capable of responding to changing conditions. One of the cheapest and most convenient methods for performing the instruction set extension that forms the building block of our work is modifying an already designed open-source processor. At this point, it is necessary to follow a design process for designing an embedded system with all its software and hardware components. This necessity requires a combination of hardware, software, and an environment for software development design in order to form a base for the implementation of the specific application chosen in this project. Then, cost-sensitive application software design is the last requirement. An open-source processor is needed in order to extend the instruction set architecture to improve performance. For the SoC Implementation of our project, Nexys 4 DDR FPGA OV7670 CMOS camera module is used. Laplacian filter, which is the most significant filter for detecting fatigue and yawning, is used as an IP core and added to the system. Finally, the instructions of RISC-V are extended to lower the time and improve the performance of our entire system.

Steps for the implementation of the Laplacian filter on SoC is shown below:

1. Modeling the system for image processing and verification of the model
2. Implementation of RISC-V processor on an FPGA
3. Realizing the model of the system for Laplacian filter on the RISC-V processor implemented on the FPGA
4. Extension of the instructions of RISC-V processor for custom kernel convolution and implementation on the FPGA

RISC-V is an open-source extendable instruction set architecture. It is pretty much easy to add custom extensions to the RISC-V processor for performance improvement which makes it a proper architecture for the aim of this project. RISC-V has various cores and SoCs for implementation that supporting diverse extensions. Considering our project, Ibex core is chosen for the project since it is easy to work, modify and easy to understand source codes including ALU, Instruction fetch, Instruction decoder, execute, write back, and other stages. It also implements the standard multiplication extension of RISC-V, which is very useful in image convolution applications. However, Ibex has a disadvantage in that it does not come with a bus interface or any peripherals. In order to easily add and use peripherals like VGA and camera, a modified version of Ibex, called ibex_wb is used. This project supports an extendable Wishbone bus interface module. Using this protocol, any Wishbone compatible peripheral can be connected to the core as an IP easily. The base project includes a wishbone led IP as a slave, which communicates with RAM over wishbone interconnect. The main duty of wishbone interconnect is sorting Masters and Slaves according to their priorities for establishing the connection between a master and a slave in a single moment. The related RISC-V compiler has been installed in Linux operating systems in order to translate the C code to assembly language to initialize the instruction and data memory. This compiler is generating the .vmem file, which is going to be included in the Ibex core project. A test bench is written to test the core and led IP in the first phase of the project. After successful simulation, the Ibex project has been synthesized and implemented in the Nexys4DDR FPGA board. In the next step, Camera and VGA IP have been added as masters with the RAM direct access ability. In order to save all the pixels to RAM, the memory size of the stack region has been increased by modifying link.ld file. After getting the video stream by camera module and illustrating it on a VGA monitor, a Laplacian filter is written in C languages to apply it to the image. The mentioned process has forwarding tasks: camera module captures raw RGB data and writes to RAM, Laplacian filter algorithm processes the raw image and the results of the calculations written back to memory. VGA modules show output images to monitor. In order to compare the software and hardware implementation, the hardware implementation of the Laplacian filter has been designed and implemented in our system by adding a single DSP unit. Considering the processing time, there is 35 times improvement in pure hardware implementation in compared to software implementation of Laplacian filter.

Multiply and addition operation (MAC) is the base operation of every convolution process in image processing applications. Inspiring from the hardware implementation improvements, custom instructions are added to the system. The compiler source

code was edited and rebuilt to make the compiler identify the newly added custom instructions. In the project, the instruction decoder and ALU part are modified according to our instructions. First custom instruction initializing the kernel of a filter. Second custom instruction, implementing the MAC operating in parallel by using 18 DSP units. Later the custom instructions are calling by using the inline assembly method inside the C code. In order to implement the Smooth filter, all the kernel members should be 1/9, however, the Ibex core doesn't support float extension. In order to overcome this problem, initializing kernel custom instruction input multiplied by 16 (shift 4 bits to left). In the ALU part, the result of the MAC operation is divided by 16 (arithmetic shift right preserving sign bit). Considering this method, filters with float kernels are either supported for image processing applications. Filter implementation methods are compared in *chapter*9.

GÖRÜNTÜ İŞLEME UYGULAMALARI İÇİN ÖZEL KOMUT SETİNE SAHİP RISC-V TABANLI BİR SOC SISTEMİNİN FPGA GERÇEKLEMESİ

ÖZET

Günümüzde görüntü işleme için güvenilir ve hızlı donanım sistemleri hayatımızda önemli bir rol oynamaktadır. Karar verme söz konusu olduğunda, donanım üzerinde uygulanan bir görüntü işleme algoritması hızlı ve güvenilirdir. Örneğin bu günlerde sürücü yorgunluk tespit sistemleri ulaşım sektöründe çok önemli. İnsanların çoğu, sadece çok kısa bir gece uykusundan sonra, uzun yolculuklar sırasında veya değişen fiziksel koşullar sırasında, araba kullanırken uyuşukluğun kurbanı oluyor. Bu nedenle, sürücünün yorgunluğundan ve esnemesinden kaynaklanan korkunç kazaları önlemek için nakliye birimlerine uygulanan bir alarm sistemine ihtiyaç vardır. Gömülü sistemler, hız, doğruluk ve güvenilirlik gibi düşük maliyetli performans gereksinimleri sağlamak amacıyla özel olarak geliştirilmiştir. Tanım olarak, uygulamaya özel donanım ve yazılım bileşenleri içerir. Ancak hedef sistem büyüdükçe, sistemin tasarımı için işgücüne ve deneyim çeşitliliğine olan ihtiyaç artmaktadır. Bu nedenle sistemin kullanıcı kontrolü daha zor hale gelir. Bir sistemin rahatlıkla kullanılabilmesi, gerektiğinde güncellenebilmesi ve diğer gömülü sistemlere uygun çalışılabilmesi için donanım için yazılım desteğinin sağlanması çok önemlidir. Birçok uygulama, uygulamaya özel ucuz donanım birimleriyle gerçekleştirilebilir. Ancak yine de ilgi, yazılım geliştirme ortamının olmaması nedeniyle farklı uygulamaları desteklemek için tasarlanmış genel amaçlı, pahalı çözümlere yöneliktir. Öte yandan gömülü sistemlerin uygun yazılımlarla desteklenmesi ve bu sistemlere özel verimli sinyal işleme ve karar verme algoritmaları geliştirilerek gömülü sistemlerin uygulama alanları genişletilebilir. Araştırmacıların ve kullanıcıların kolayca geliştirebilecekleri gömülü sistemlerin artması, birçok uygulamanın ucuz, pratik ve değişen koşullara yanıt verebilecek nitelikte olabileceği anlamına gelir. Çalışmamızın yapı taşını oluşturan komut seti uzantısını gerçekleştirmenin en ucuz ve en uygun yöntemlerinden biri, önceden tasarlanmış bir açık kaynaklı işlemciyi değiştirmektir. Bu noktada tüm yazılım ve donanım bileşenleri ile gömülü bir sistem tasarlamak için bir tasarım sürecini takip etmek gerekir. Bu zorunluluk, bu projede seçilen özel uygulamanın uygulanması için bir temel oluşturmak için bir donanım, yazılım ve yazılım geliştirme tasarımı ortamı kombinasyonunu gerektirir. O halde, maliyete duyarlı uygulama yazılımı tasarımı son gereksinimdir. Performansı artırmak için komut seti mimarisini genişletmek için açık kaynaklı bir işlemciye ihtiyaç vardır. Projemizin SoC Uygulaması için Nexys 4 DDR FPGA OV7670 CMOS kamera modülü kullanılmıştır. Yorgunluk ve esnemeyi tespit etmek için en önemli filtre olan Laplacian filtresi IP core olarak kullanılmakta ve sisteme eklenmektedir. Son olarak, RISC-V'nin komutları, zamanı azaltmak ve tüm sistemimizin performansını iyileştirmek için genişletilmiştir.

Laplacian filtresinin SoC üzerinde uygulanması için adımlar aşağıda gösterilmiştir:

1. Görüntü işleme ve modelin doğrulanması için sistemin modellenmesi
2. RISC-V işlemcinin FPGA üzerinde uygulanması
3. FPGA üzerinde uygulanan RISC-V işlemci üzerindeki Laplacian filtresi için sistem modelini gerçekleştirme
4. Özel çekirdekli konvolüsyonun RISC-V işlemcisi için komut setinin uzatılması ve FPGA üzerinde gerçekleştirilmesi

RISC-V, açık kaynaklı, genişletilebilir bir komut seti mimarisidir. Performans iyileştirme için RISC-V işlemcisine özel uzantılar eklemek oldukça kolaydır ve bu da onu bu projenin amacına uygun bir mimari haline getirir. RISC-V, çeşitli uzantıları destekleyen uygulama için çeşitli çekirdeklere ve SoC'lere sahiptir. Projemiz göz önüne alındığında, çalışması, değiştirilmesi ve ALU, Instruction fetch, Instruction decoder, yürütme, geri yazma ve diğer aşamaları içeren kaynak kodlarının anlaşılması kolay olduğu için proje için Ibex çekirdeği seçilmiştir. Ayrıca, görüntü işleme uygulamalarında çok kullanışlı olan RISC-V'nin standart çarpma uzantısını da uygular. Ancak, Ibex'in bir veri yolu arabirimi veya herhangi bir çevresel birimi ile gelmemesi gibi bir dezavantajı vardır. Vga ve kamera gibi çevresel birimlerini kolayca eklemek ve kullanmak için, Ibex'in ibex_wb adı verilen değiştirilmiş bir sürümü kullanılır. Bu proje, genişletilebilir bir Wishbone veri yolu arabirim modülünü destekler. Bu protokolü kullanarak, herhangi bir Wishbone uyumlu çevresel birimi çekirdeğe bir IP olarak kolayca bağlanabilir. Temel proje, Wishbone ara bağlantısı üzerinden RAM ile iletişim kuran bir slave olarak bir LED IP içerir. LED ara bağlantısının temel görevi, bir ana ve bir köle arasındaki bağlantıyı tek bir anda kurmak için Master ve Slave'leri önceliklerine göre sıralamaktır. Komut ve veri belleğini başlatmak için C kodunu assembly diline çevirmek için Linux işletim sistemlerinde ilgili RISC-V derleyicisi kurulmuştur. Bu derleyici, Ibex çekirdek projesine dahil edilecek olan .vmem dosyasını oluşturuyor. Projenin ilk aşamasında çekirdek ve led IP'yi test etmek için bir test bench yazılır. Başarılı simülasyondan sonra Ibex projesi sentezlendi ve Nexys4DDR FPGA kartında uygulandı. Bir sonraki adımda, RAM doğrudan erişim yeteneği ile Master olarak Kamera ve VGA IP eklenmiştir. Tüm pikselleri RAM'e kaydetmek için link.ld dosyası değiştirilerek yığın bölgesinin bellek boyutu artırıldı. Video akışını kamera modülü ile aldıktan ve VGA monitörde gösterdikten sonra, görüntüye uygulamak için C dillerinde bir Laplacian filtresi yazılır. Bahsedilen işlemin yönlendirme görevleri vardır: kamera modülü ham RGB verilerini yakalar ve RAM'e yazar, Laplacian filtre algoritması ham görüntüyü işler ve hesaplamaların sonuçları belleğe geri yazılır. VGA modülleri, izlenecek çıktı görüntülerini gösterir. Yazılım ve donanım uygulamasını karşılaştırmak için, tek bir DSP birimi eklenerek Laplacian filtresinin donanım uygulaması tasarlanmış ve sistemimize uygulanmıştır. İşlem süresi göz önüne alındığında, Laplacian filtresinin yazılım uygulamasına kıyasla saf donanım uygulamasında 35 kat iyileştirme vardır.

Çarpma ve toplama işlemi (MAC), görüntü işleme uygulamalarında her evrişim işleminin temel işlemidir. Donanım uygulama geliştirmelerinden esinlenerek, sisteme özel talimatlar eklenir. Derleyici kaynak kodu, derleyicinin yeni eklenen özel talimatları tanımlamasını sağlamak için düzenlendi ve yeniden oluşturuldu. Projede komut kod çözücü ve ALU kısmı bizim yönergelerimize göre değiştirilmiştir. Filtre çekirdeğini başlatan ilk özel talimat. İkinci özel talimat, 18 DSP birimi kullanılarak

paralel olarak çalışan MAC'in uygulanması. Daha sonra özel talimatlar, C kodu içinde satır içi derleme yöntemi kullanılarak çağrılır. Düzgün filtreyi uygulamak için tüm çekirdek üyeleri 1/9 olmalıdır, ancak Ibex çekirdeği float uzantısını desteklemez. Bu sorunun üstesinden gelmek için, çekirdek özel komut girişinin başlatılması 16 ile çarpılır (4 bit sola kaydırma). ALU kısmında, MAC işleminin sonucu 16'ya bölünür (aritmetik sağa kaydırma ve işaret bitini koruyarak). Bu yöntem göz önüne alındığında, görüntü işleme uygulamaları için kayan çekirdekli filtreler desteklenir. Filtre uygulama yöntemleri bölüm 9'da karşılaştırılmıştır.

1. INTRODUCTION

1.1 Motivation

The integration of electronic and/or microelectronic systems in the automobile microelectronic systems in the automobile is growing and, in general, has been demanding much support from the computer-aided systems [8]. In fact, there have been many attempts to achieve reliable real-time vehicle driver surveillance systems. However, today's configurable hardware technology is able to support tasks with a very high computational load, such as image processing. One way to achieve very low-cost and real-time automobile systems is to use Hardware Description Languages (HDL) [9], new high-performance and low-cost FPGA [10] devices and implementation in open source cores.

Benefits of using open source cores like RISC-V [11] are that we can add custom instructions [12] in order to improve the execution time of processing. In image processing applications, the convolution process [2] is highly used for image transforms like edge detection [13].

1.2 Background

SoC is a chip/integrated circuit that holds many components of a computer, usually the CPU, memory and GPIO ports [14]. As a hardware platform, Nexys4DDR [15] has been used. An open source core called RISC-V [11] as a microprocessor is used in this project [4]. GPIO IPs are camera [6] and VGA modules [16]. For memory section, the BRAM of the FPGA board has been used for instruction and data memory. Wishbone [5] protocol has been added in order to make a communication data bus among all IPs as masters and slaves. RISC-V Core called 'Ibex' [17] is implemented in the FPGA board in our project. Ibex has a 2-stage pipeline, Instruction Fetch (IF), Instruction Decode and Execute (ID/EX). All instructions require two cycles minimum to pass down the pipeline. One cycle in the IF stage and one in the ID/EX stage.

For image processing applications Laplacian filter [18] and smooth filter [19] has been implemented in our system using convolution method [2]. Convolution method for image processing applications is depicted below:

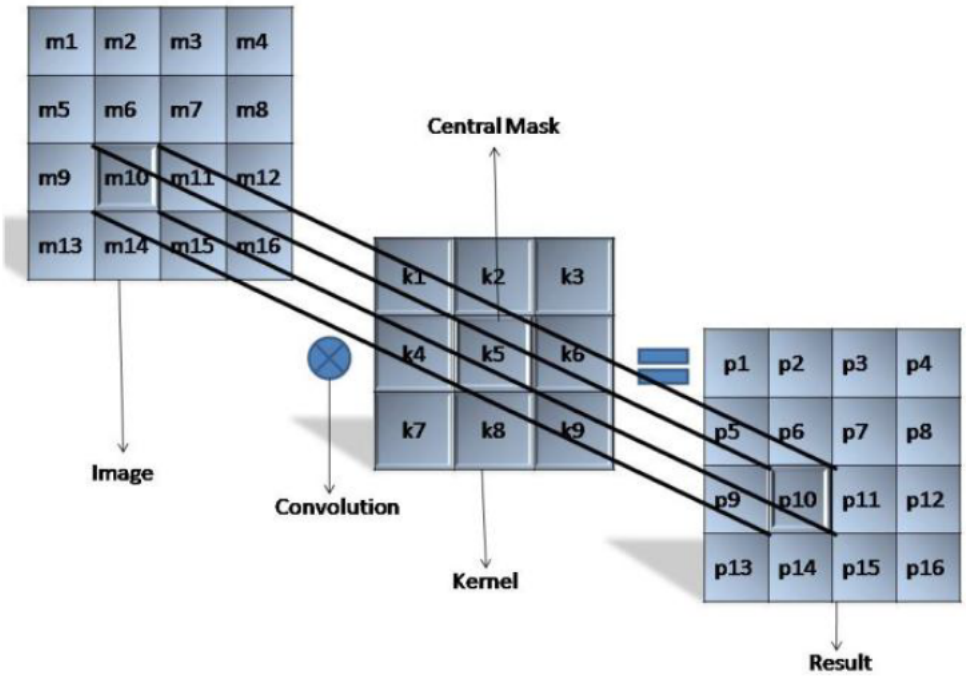


Figure 1.1 : Convolution in Image Processing [2].

Figure 1.1 shows the convolution of the image and the kernel matrices. This image represents the 4×4 matrices and kernel matrices represents the 3×3 and resultant image represents the photo size in 4×4. The equation below represents how the convolution operation should be performed.

$$P10 = (m5 \times k1) + (m6 \times k2) + (m7 \times k3) + (m9 \times k4) + (m10 \times k5) + (m11 \times k6) + (m13 \times k7) + (m14 \times k8) + (m15 \times k9). \quad \textbf{(1.1)}$$

Two common formats for images are RGB and YCbCr [20]. The difference between YCbCr and RGB is that YCbCr represents color as brightness and two color difference signals, while RGB represents color as red, green and blue. In YCbCr, the Y is the brightness (luma), Cb is blue minus luma (B-Y) and Cr is red minus luma (R-Y). In Laplacin filter edge detection [18], for the sudden changes o the pixel values, the

amount of the calculated value in Eq.1.1 getting high and appear as whitish pixels as demonstrated in Figure 1.2.

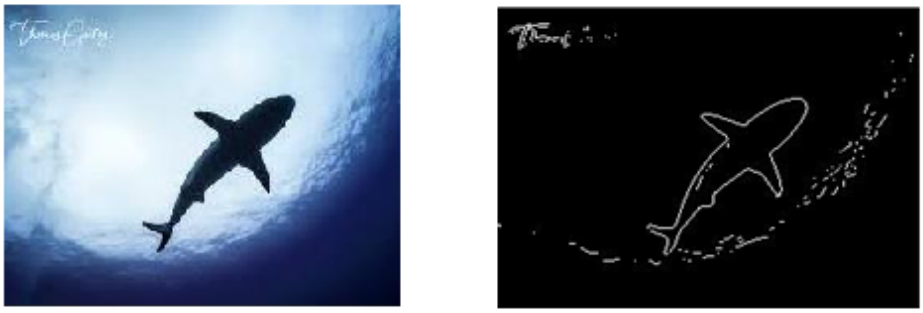


Figure 1.2 : Edge detection [3].

For comparison purposes, mentioned filters has been implemented in hardware and software and the results are shown in Table 9.1.

1.3 Thesis Contribution

This thesis is done in contribution to the joint project which is supported by the Scientific and Technological Research Council of Turkey (TÜBİTAK) and the Ministry of Science, Research & Technology of Iran (MSRT) under the project number of 119N461. In this work, An FPGA Implementation of a RISC-V based SoC system for image processing applications is covered. As a result of performance improvement, instruction set are extended in RISC-V to decrease the processing time of the input image.

1.4 Organization of This Study

The organization of this thesis is as follows. A brief explanation of RISC-V open source core and Laplacian filter is given in Chapter 2. In Chapter 3, we have done a literature review on RISC-V ISAs and extensions with various opcode formats. In chapter 4, all the necessary environment is covered step by step to get the system ready for SoC implementation. Chapter 5 presents the simulation and implementation of an RISC-V open source core called Ibex on Nexys 4 DDR FPGA board. Chapter 6 discusses the communication protocol, Wishbone, and related signals in order to communicate with every IPs over shared bus. In Chapter 7, Wishbone Compatible IPs such as core, camera module and VGA implementation is explained. Chapter 8,

demonstrates the Laplacian filter as an additional IP to compare the results between hardware and software implementation of the filter in our core. In Chapter 9, instruction set of RISC-V is added to the core in order to process the image in less time and Chapter 10 concludes this work and gives future directions on the topic.

2. LITERATURE REVIEW

2-D convolution implementations [2] vary in a wide range which includes ASICs, FPGAs, general-purpose processors, digital signal processors, and graphic processing units. In this work, recent real-time and power-efficient implementations are covered.

2.1 Hardware Architectures for Edge Detector

Laplacian filter implementation is similar to the Sobel filter [21]. A comprehensive study for Sobel filter hardware implementations is given in [21]. FPGA implementations given therein are exploited parallelism by introducing processing elements (PEs) to perform gradients by means of parallel additions and multiplications. Providing 6 (or 12 for 2 PEs) pixels simultaneously to the PEs, and using constant multiplication a significant increase in throughput is observed. However, this design cannot be generalized to be used with other convolution kernels. Moreover, the memory access bottleneck is overlooked.

2.2 Accelerating Local Laplacian Filters on FPGAs

Another work exploiting FPGA parallelism is presented in [22], where a host system (CPU) is responsible to arrange image capture and reconstruction. FPGA convolution contains 9 processing units which employ multiply-accumulate modules implemented on DSP blocks. The host and FPGA communicates through a dual RAM. Despite this additional communication cost, accelerator in [22] can process a 1 megapixel image 7.5 times faster than an 8-core CPU implementation.

2.3 Parallel Laplacian Filter Using CUDA on GPGPU

In [23] a GPU accelerator is used with a CPU host. CPU calls CUDA wrapper, a parallel computing platform and interface for GPU. The input image is transferred from host memory to the accelerator memory allocated initially by the GPU. Similarly,

output image is written on pre-allocated GPU memory before transmitted to the host. The image is processed with multiple (up to 200) threads, this way large image processing is accelerated. However, the transfer of these large images becomes slower but it is not reported in [23].

2.4 FPGA Implementation of Sobel and Laplacian Filters

A real-time filter system implementation on Zynq SoC is given in [20]. Implementing the image resampler, interconnect blocks such as YCbCr to RGB converter on programmable logic speeds up the system. Moreover, the system becomes flexible by implementing the filters on the processor system. On contrary to the other examples, data is kept on the same chip. AXI bus is used for data transfer so HDMI to AXI converters are used as an overload.

2.5 Enhancing FPGA Softcore Processors

The benefits of adding DSP support to an FPGA based softcore processor are discussed in [24]. They designed, implemented, simulated and validated an experimental DSP processor on a Xilinx Virtex-5 XC5VSX50T [25] FPGA chip. New hardware to improve the performance of the processor's DSP implementations had been developed. These hardware development techniques include methods such as a specialist execution unit, modifying the memory architecture, adding an extra address controller, upgrading it with more efficient addressing modes, and extending the instruction set with support for zero overhead loops. These improvements reduced the number of commands used in the inner loop of convolution-based DSP algorithms with a single command. Using these proposed enhancement techniques, they increased the overall performance by an average of 9-fold (800) in FIR filter [24].

2.6 Literature Conclusion

Related works show that, when a hardware accelerator is managed with flexibility enhancing CPU, significant speed up is achieved. However, data transfer between chips and memory access overload should be considered. Next, a custom hardware

IP for Laplacian filter is introduced. This IP is implemented on the same chip with a controlling CPU and dual port data memory.

3. RISC-V

In this work, as the main processor, a RISC-V core [1] is going to be implemented in an FPGA. RISC V processor has a completely open source ISA, so it is possible to change and update the ISA of any version (except frozen versions) [1]. RISC-V allows designers to customize their work by extending instruction set architectures. The simplicity and elegance of the RISC-V Instruction result in SoC designs that are simpler, have fewer logic gates, consume less energy, and ultimately cost less. Some ISA versions of RISC-V are Frozen, which means that they can no longer be changed, however, changes can be done in the form of extensions as shown in Figure 3.1.

Base	*Version*	*Draft Frozen?*
RV32I	2.0	Y
RV32E	1.9	N
RV64I	2.0	Y
RV128I	1.7	N
Extension	Version	Frozen?
M	2.0	Y
A	2.0	Y
F	2.0	Y
D	2.0	Y
Q	2.0	Y
L	0.0	N
C	2.0	Y
B	0.0	N
J	0.0	N
T	0.0	N
P	0.1	N
V	0.7	N
N	1.1	N

Figure 3.1 : RISC-V ISA Versions [1].

3.1 RISC-V ISA

The explanation of each extension explanation is given in the Table 3.1.

Table 3.1 : RISC-V Extension Descriptions [1].

A	Atomic instructions
M	Integer multiplication and division
F	Single precision floating point
D	Double precision floating point
Q	Quad precision floating point
C	Compressed instructions
L	Decimal floating point
B	Bit manipulation
P	Packed SIMD instructions
V	Vector operations
N	User level interrupts
J	Dynamically translated languages
T	Transactional memory

3.1.1 RISC-V base instructions

In the Base versions of RISC-V, ISAs are RV32I, RV64I, RV128I and RV32E. The differences are in the number of bit registers [1]. There is also a G extension which refers to general-purpose and covers all extensions: I, M, F, A, D. This mode is commonly used to implement computer CPUs. The word 'I' has been added to RVxxI to mean operations on integers. So the ISA base does not have Floating-Point and Multiplication operations, and to do this, extensions must be added. RV32I is much more widely used. When designing, ISA must be considered one of the bases and then extensions will be added if needed. The RV32E is a subset of the RV32I that was added to support microcontrollers, also has 16 registers instead of 32. All RV32I base instruction sets can be found in 3.2 [1].

In the base RV32I ISA, there are four core instruction formats (R/I/S/U), as shown in Figure 3.3 [1]. All are a fixed 32 bits in length and must be aligned on a four-byte boundary in memory. An instruction-address-misaligned exception is generated on a taken branch or unconditional jump if the target address is not four-byte aligned. This exception is reported on the branch or jump instruction, not on the target instruction. No instruction-address-misaligned exception is generated for a conditional branch that is not taken.

imm[31:12]				rd	0110111	LUI
imm[31:12]				rd	0010111	AUIPC
imm[20|10:1|11|19:12]				rd	1101111	JAL
imm[11:0]		rs1	000	rd	1100111	JALR
imm[12|10:5]	rs2	rs1	000	imm[4:1|11]	1100011	BEQ
imm[12|10:5]	rs2	rs1	001	imm[4:1|11]	1100011	BNE
imm[12|10:5]	rs2	rs1	100	imm[4:1|11]	1100011	BLT
imm[12|10:5]	rs2	rs1	101	imm[4:1|11]	1100011	BGE
imm[12|10:5]	rs2	rs1	110	imm[4:1|11]	1100011	BLTU
imm[12|10:5]	rs2	rs1	111	imm[4:1|11]	1100011	BGEU
imm[11:0]		rs1	000	rd	0000011	LB
imm[11:0]		rs1	001	rd	0000011	LH
imm[11:0]		rs1	010	rd	0000011	LW
imm[11:0]		rs1	100	rd	0000011	LBU
imm[11:0]		rs1	101	rd	0000011	LHU
imm[11:5]	rs2	rs1	000	imm[4:0]	0100011	SB
imm[11:5]	rs2	rs1	001	imm[4:0]	0100011	SH
imm[11:5]	rs2	rs1	010	imm[4:0]	0100011	SW
imm[11:0]		rs1	000	rd	0010011	ADDI
imm[11:0]		rs1	010	rd	0010011	SLTI
imm[11:0]		rs1	011	rd	0010011	SLTIU
imm[11:0]		rs1	100	rd	0010011	XORI
imm[11:0]		rs1	110	rd	0010011	ORI
imm[11:0]		rs1	111	rd	0010011	ANDI
0000000	shamt	rs1	001	rd	0010011	SLLI
0000000	shamt	rs1	101	rd	0010011	SRLI
0100000	shamt	rs1	101	rd	0010011	SRAI
0000000	rs2	rs1	000	rd	0110011	ADD
0100000	rs2	rs1	000	rd	0110011	SUB
0000000	rs2	rs1	001	rd	0110011	SLL
0000000	rs2	rs1	010	rd	0110011	SLT
0000000	rs2	rs1	011	rd	0110011	SLTU
0000000	rs2	rs1	100	rd	0110011	XOR
0000000	rs2	rs1	101	rd	0110011	SRL
0100000	rs2	rs1	101	rd	0110011	SRA
0000000	rs2	rs1	110	rd	0110011	OR
0000000	rs2	rs1	111	rd	0110011	AND
fm | pred | succ		rs1	000	rd	0001111	FENCE
000000000000		00000	000	00000	1110011	ECALL
000000000001		00000	000	00000	1110011	EBREAK

Figure 3.2 : RV32I Base Instruction [1].

The RISC-V ISA keeps the source (rs1 and rs2) and destination (rd) registers at the same position in all formats to simplify decoding. Except for the 5-bit immediates used in CSR instructions, immediates are always sign-extended, and are generally packed towards the leftmost available bits in the instruction and have been allocated to reduce hardware complexity. In particular, the sign bit for all immediates is always in bit 31 of the instruction to speed sign-extension circuitry.

3.1.2 Integer register register operations

RV32-Integer defines so many arithmetic R-type instructions [1]. All instructions read the rs1-register and rs2-register as a source operands and write back the result into rd register. Type of operation selection has done by funct7 and funct3 as shown in Figure 3.4.

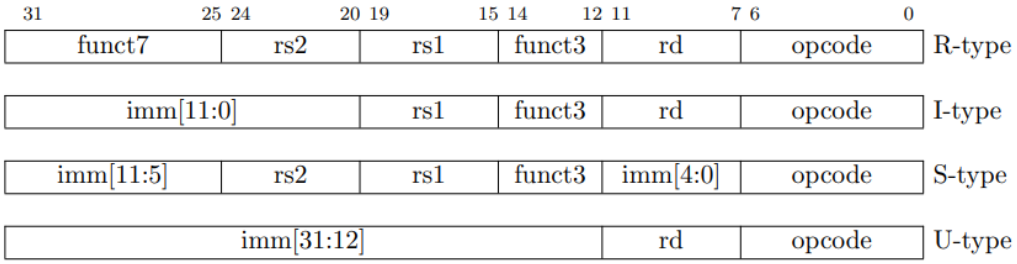


Figure 3.3 : RISC-V Base Instruction Formats [1].

31 25	24 20	19 15	14 12	11 7	6 0
funct7	rs2	rs1	funct3	rd	opcode
7	5	5	3	5	7
0000000	src2	src1	ADD/SLT/SLTU	dest	OP
0000000	src2	src1	AND/OR/XOR	dest	OP
0000000	src2	src1	SLL/SRL	dest	OP
0100000	src2	src1	SUB/SRA	dest	OP

Figure 3.4 : RISC-V Operations [1].

ADD command executes the rs1 and rs2 addition. SUB command executes rs2 from rs1 subtraction. SLT and SLTU executes comparison of signed and unsigned, if (rs1 < rs2) writing 1 to rd, otherwise 0. SLTU rd, x0, if rs2 is nonzero, rs2 sets rd to 1, otherwise rd set to 0. XOR ,AND, and OR execute bitwise-logical operations. SRA, SRL and SLL execute arithmetic right shifts, logical right and logical left by the shift amount held in the lower 5 bits of rs2 on the value in rs1.

4. SETTING UP THE ENVIRONMENT

RISC-V core implementation requirements are as follows:

Table 4.1 : RISC-V Core Implementation Prerequisites.

Linux Ubuntu	16.04
Xilinx Vivado	2019.2
RISC-V GCC	10.2

4.1 Installation Procedure

4.1.1 Installing linux operating system

At first, Ubuntu 16.04 [26] is installed to prepare the required applications. Linux based OS is chosen since complex open source cores are capable of being implemented in Linux rather than Windows, to work properly because of scripts and other tools.

4.1.2 Installing vivado design suite

Next step is to install Vivado Design Suite 2019.2 [27]. For installing the Vivado following steps must be done:

- Make a directory in the opt/ directory and apply write permission :

```
$ sudo mkdir opt/Xilinx
$ sudo chmod –R 777 opt/Xilinx
```

- In the download list of the Xilinx Vivado Design Suite, Download a version with .tar.gz extension file.
- Go to Downloads folder and extract the downloaded file.
- Enter to the extracted directory and install:

```
$ cd Xilinx_Vivado_2019.2_1106_2127
$ sudo ./xsetup
```

After installation process, following command must be added at the end of .bashrc file in /home directory:

```
source /opt/Xilinx/Vivado/2019.2/settings64.sh
```

4.1.3 Installing RISC-V GNU toolchain

Final step, is to install RISC-V GNU toolchain. A suitable compiler toolchain is needed to program the processor. The version required for Ibex is the "RV32IMC" version. This instruction set is the most basic of RISC-V architecture processors. To download the resource from the RISC-V GNU toolchain repository, the following commands can be entered into the terminal opened in the root [28]:

- First of all, there are some necessary package to be installed in the system:

```
$ sudo apt-get install autoconf automake
autotools-dev curl libmpcdev libmpfr-dev
libgmp-dev gawk build-essential bison flex
texinfo gperf libncurses5-dev libusb-1.0-0
libboost-dev
```

- To use gitHub in linux OS, git tool should be installed. following command must be entered tin the terminal:

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

- After installing the required package, RISC-V GNU toolchain can be installed. this is a cross compiler which can convert instructions into machine code or low level code for a processor other than on which it is running. Following commands must be entered to start the compiler installing process:

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

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

A directory is chosen to install the cross compiler. In this project /opt/riscv is selected, now the path /opt/riscv/bin should be added to PATH, ".bashrc" file in home directory, as follows:

```
export PATH=$PATH:/opt/riscv/bin
```

Now, the environment is ready and we can install the compiler:

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

This process can take about 30-40 minutes. when finished, proper installation can be checked by entering the command below:

```
$ which riscv32-unknown-elf-gcc
```

If the process is successful, the versions of the compiler will be shown as Figure 4.1:

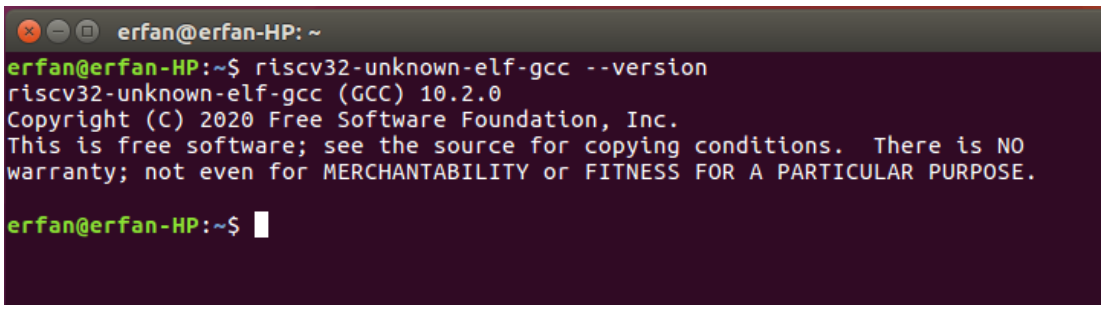


Figure 4.1 : Compiler Proper Installation Check - 32bit Version.

This is the 32-bit version of the compiler which supports "IMC" extension of RISC-V instructions. We are going to install 64-bit with "GC" extensions for further projects of 64-bit RISC-V cores, by following the commands below as well:

```
$ ./configure --prefix=/opt/riscv --with-arch=rv64gc
$ make
```

By calling the following command, we will check for the correct installation as shown in Figure 4.2:

```
$ which riscv64-unknown-elf-gcc
```

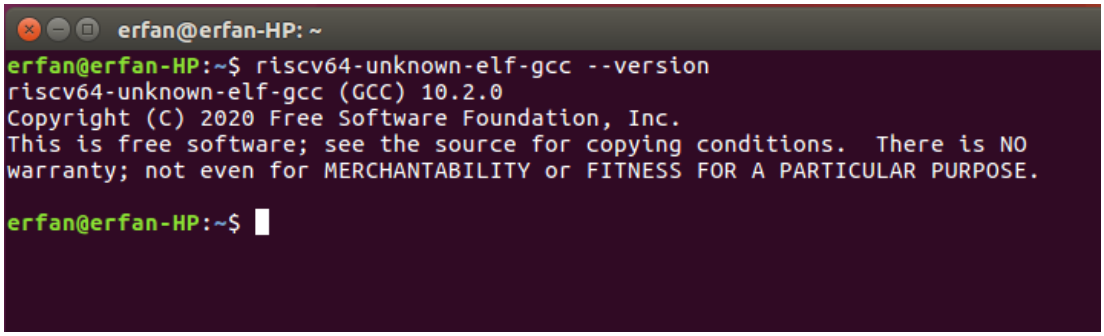


Figure 4.2 : Compiler Proper Installation Check - 64bit Version.

This compiler will generate .vmem file from given C file which is the required instruction data for embedding in RISC-V core [11]. The next chapter, is about Ibex core [4] implementation with a basic C code that will make LEDs to blink.

5. RISC-V OPEN SOURCE IMPLEMENTATION ON FPGA

In this section, we are going to implement an open source RISC-V core on FPGA. There are many available cores and SoCs in RISC-V Foundation [11]. Our candidate for implementation is Ibex core as it supports "IMC" extension and all stages are clear to understand and modify for instruction extension of ISA. Xilinx Vivado is used for RTL simulation and implementation. The FPGA board which is chosen to implement the selected core, is Nexys 4 DDR [15].

5.1 Ibex Core Implementation on NEXYS 4 DDR

In this section, a complete Ibex core implementation steps on FPGA is covered.

5.1.1 Vivado project

In order to get started, Ibex repository from GitHub must be downloaded. Overall block diagram of stages are shown in Figure 5.1. We are going to start from an example which is available in the "/examples/fpga/artya7/rtl/top_artya7.sv" directory. After creating a vivado project, top_artya7.sv is going to be added as a top module. All missing sub modules must be included in the project according to the source hierarchy.

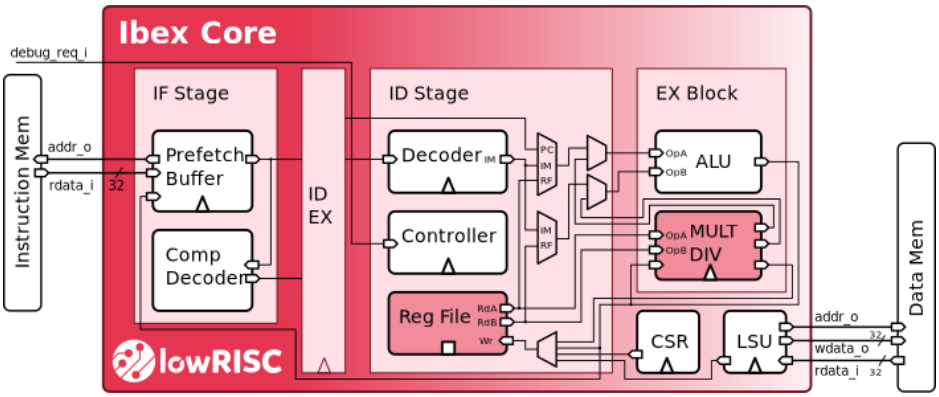


Figure 5.1 : Ibex Overall Structure [4].

As seen in the Figure 5.1, Ibex core needs a physical ram to read instruction and R/W processed data. A piece of verilog code added to the end of ram_1p.sv module for initializing the ram as follows:

```
localparam MEM_FILE = "led.mem";
 initial
  begin
   $display("Initializing SRAM from %s", MEM_FILE);
   $readmemh(MEM_FILE, mem);
  end
```

5.1.2 Create memory file to initialize RAM

In this section, we are going to generate a memory file using RISC-V compiler that installed in previous chapter. In the /examples/sw/led directory there is a C program, which is a simple led blinking code. In this folder, a terminal should be opened and the following command must be entered to generate memory file:

```
$ make CC=/opt/riscv/bin/riscv32-unknown-elf-gcc
```

After the successful operation a memory file called led.vmem must be generated in the directory as depicted in Figure 5.2. We are going to add this file to the created vivado project. As vivado doesn't support .vmem file, file extension kindly converted to .mem and added to our project. The whole project source files can be seen in Figure 5.3

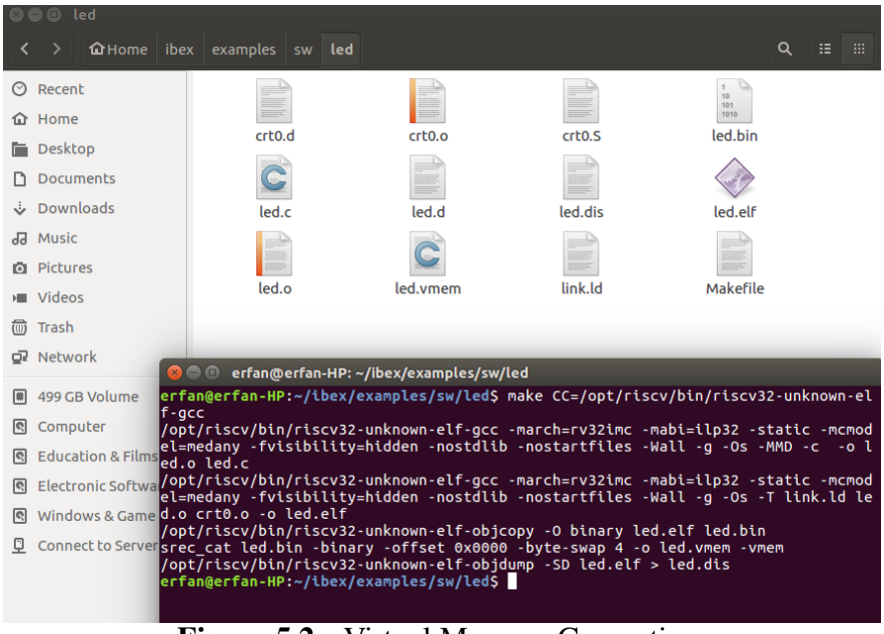


Figure 5.2 : Virtual Memory Generation.

5.1.3 Ibex core simulation

In order to validate the Ibex core and memory file instructions, a test bench is written to test the module. The simulation result is displayed in Figure 5.4

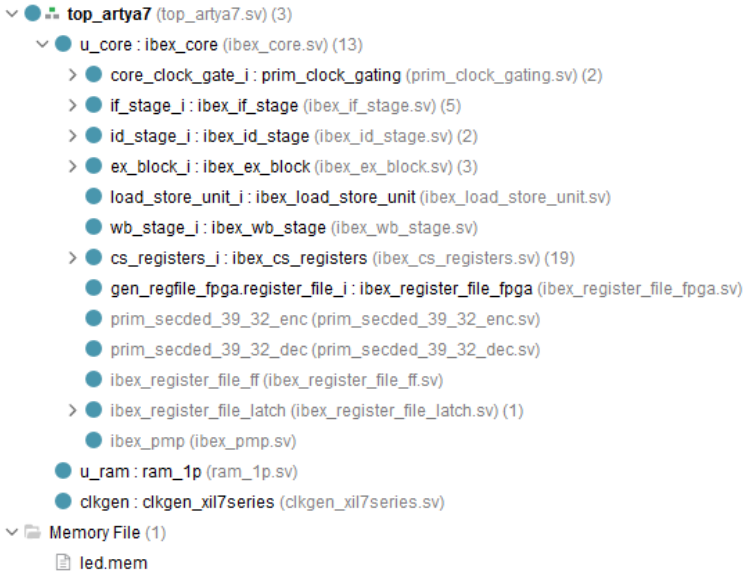


Figure 5.3 : Ibex Core Source Project Hierarchy.

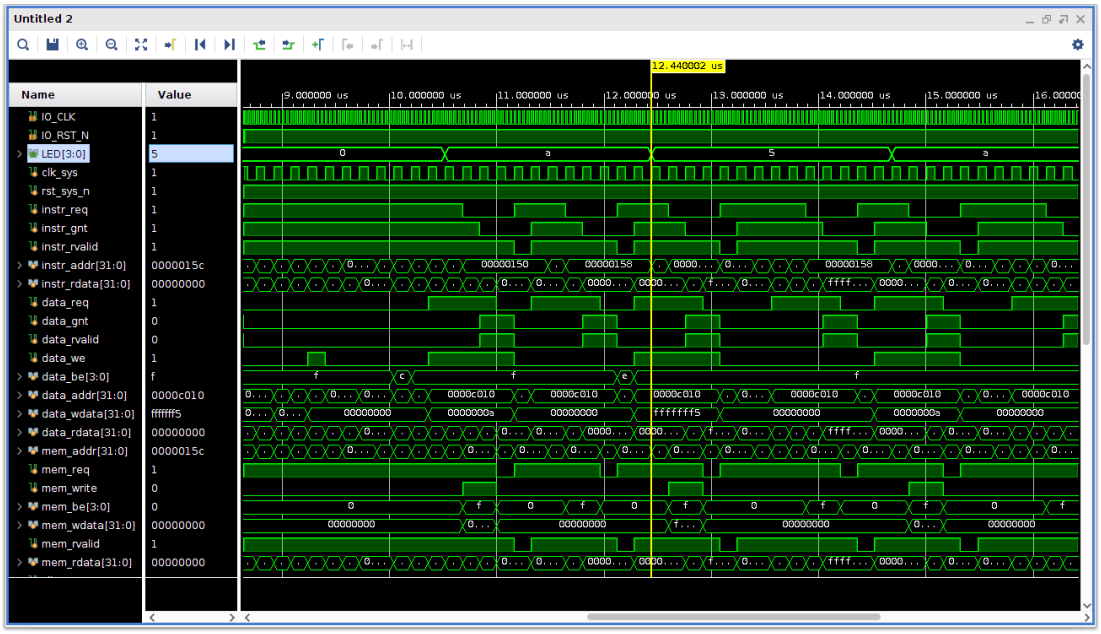
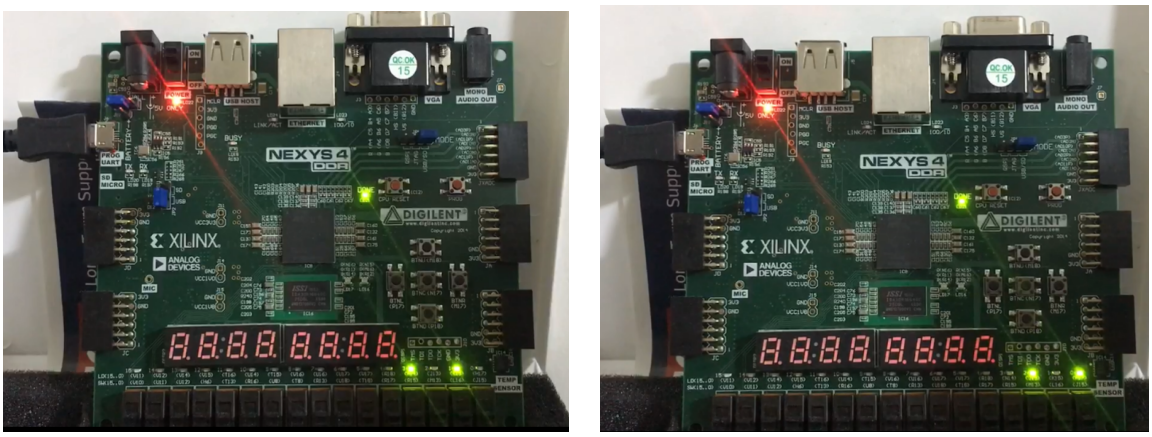


Figure 5.4 : Ibex Core Simulation Result.

Inspecting the simulation results, we can deduce that the instructions are properly executing one after another and the value which is assigning to the LEDs are correct.

5.1.4 Ibex implementation on FPGA

After validation of the core simulation, it is time to jump to the implementation step. Simply, we are going to add .xdc file [29] according to our FPGA board [15] and modify it regarding to the I/Os of our project. After generating bitstream in vivado, board can be programmed with .bit file. The result of a basic LED blinking program is shown in Figure 5.5.



a **b**

Figure 5.5 : (a) 0xA (b) 0x5 Value on LEDs as Written in C Program (Bitwise Not).

6. WISHBONE PROTOCOL

In the previous chapter, an Ibex simulation and implementation on FPGA is covered. However, to communicate with other peripherals such as camera, VGA, ... a computer bus as a communication protocol should be implemented. Our suitable candidate for protocol implementation is the Wishbone interface [30]. The Wishbone B.4 SoC interconnection architecture for portable IP Cores is a flexible design for use with IP cores [5]. This is accomplished by creating a common interface between IP cores that proves the simplicity and reliability of the system. Wishbone signals and their directions are shown in Figure 6.1:

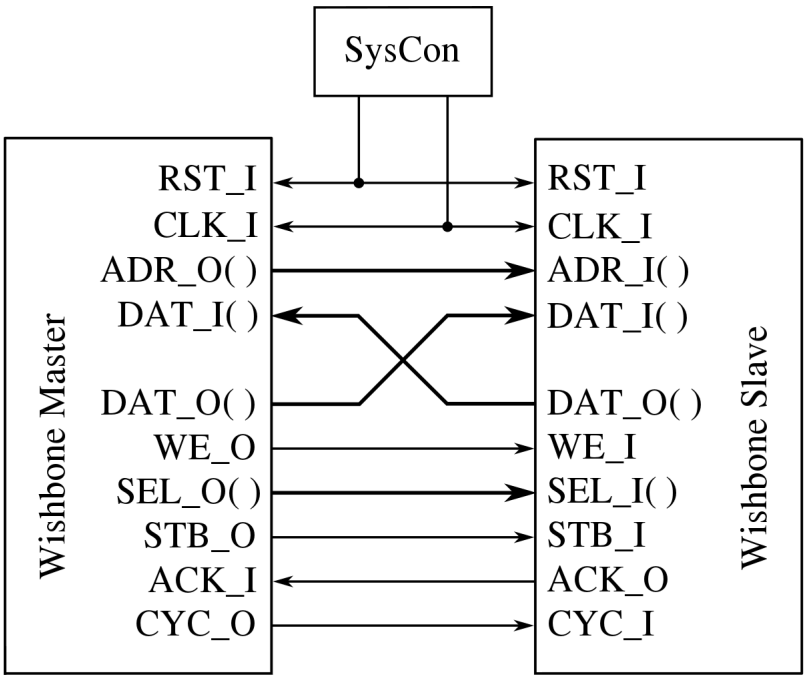


Figure 6.1 : Wishbone Interface [5].

6.1 Read sequence

Read sequence of the bus protocol works as follows, which displayed in Figure 6.2:

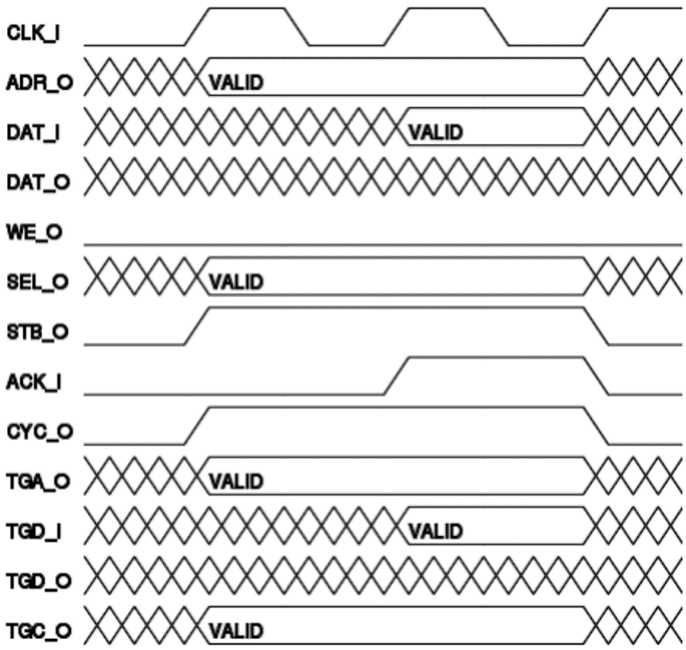


Figure 6.2 : Wishbone Read Sequence [5].

1. Clock Sequence First Phase

    - Master, puts an address in address_output.
    - Master, keeps we_output to zero to represent a read sequence.
    - Master, puts bit-selection select_output to represent where it looks for data.
    - Master, makes cycle_output high in order to represent the start of the sequence.
    - Master, makes strobe_output high to represent the start of the communication.

2. Clock Sequence Second Phase

    - Slave, discovers inputs and accept communication by making acknowledge_input high.
    - Slave, places reliable data in data_input.
    - Master, observes acknowledge_input, and gets ready to take data in data_input.

3. Clock Sequence Third Phase

    - Master, takes data in data_input.

- Master, makes strobe_output and cycle_output low to represent the termination of the sequence.
- Slave, makes acknowledge_input low in response to strobe_output reset.

6.2 Write sequence

Write sequence of the bus protocol works as follows, which displayed in Figure 6.3:

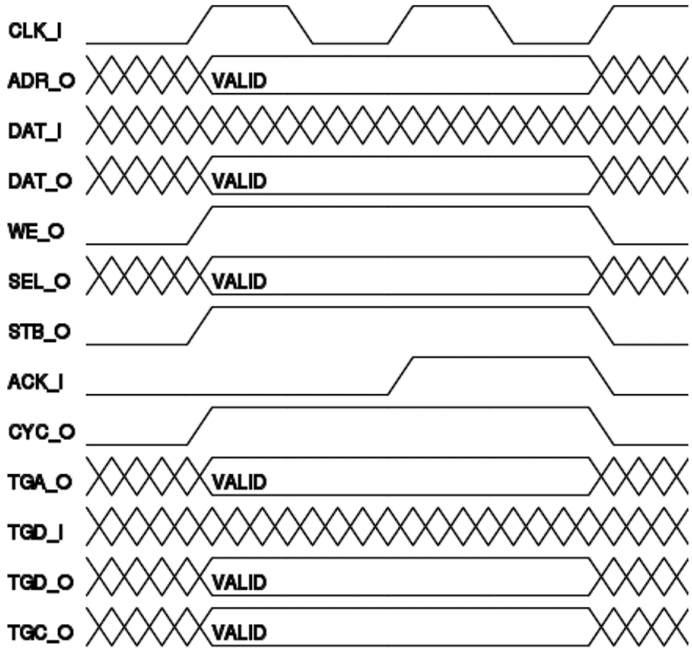


Figure 6.3 : Wishbone Write Sequence [5].

1. Clock Sequence First Phase

   - Master, puts an address in address_output.
   - Master, puts reliable data in data_output.
   - Master, keeps we_output to one to represent a write sequence.
   - Master, puts bit-selection select_output to represent where it looks for data.
   - Master, makes cycle_output high in order to represent the start of the sequence.
   - Master, makes strobe_output high to represent the start of the communication.

2. Clock Sequence Second Phase

- Slave, discovers inputs and accept communication by making acknowledge_input high.
- Slave, gets ready to take reliable data in data_output.
- Master, observes acknowledge_input, and gets ready to terminate the sequence.

3. Clock Sequence Third Phase

- Slave, takes data in data_output.
- Master, makes strobe_output and cycle_output low to represent the termination of the sequence.
- Slave, makes acknowledge_input low in response to strobe_output reset.

6.3 Interconnect

All Masters and Slaves are going to communicate with each other over a shared bus called interconnect. An arbiter controls masters' priority. Slaves are selected based on their addresses which are applied by masters. A communication principle is shown in Figure 6.4

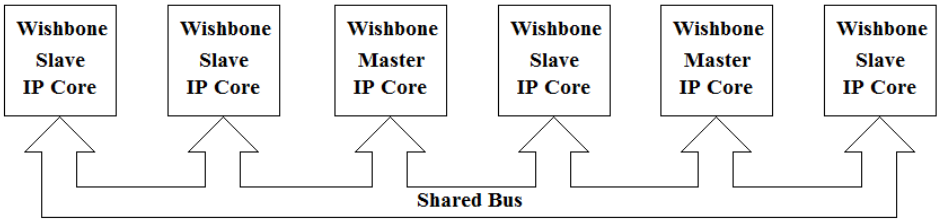


Figure 6.4 : Wishbone Shared Bus over Masters and Slaves [5].

In a single moment, only one master and one slave can link to each other and start to communicate. If another master wants to connect to the same slave, it should wait until the bus gets free by resetting STB_O and CYC_O via previous master.

6.4 System Integration

Point-to-point connection design is a straight interconnection of the master core with a slave core and it is an ordinary to design [30]. However, the system design

using shared-bus connection which is much more complicated and forces design complexities to the system integrator by SoC integration. The data may be a binary address value or a simple data value. Hence buses are used to distribute the data around a system. Multiplexer-based bus decreases the number of pins in a chip, but it requires much more clock pulses to move the data and address information in a system. Therefore, it decreases the performance of the system. To implement logic interconnections, multiplexer logic interconnection is mostly used, as these are easier to route in FPGA devices than that of three-state logic interconnection.

7. WISHBONE COMPATIBLE SoC IMPLEMENTATION

In this chapter, wishbone compatible Ibex core and other peripherals [14] like RISC-V instruction, data memory, camera, and VGA are going to be added as IP cores. Overall structure consists of five masters and a slave as shown in Figure 7.1

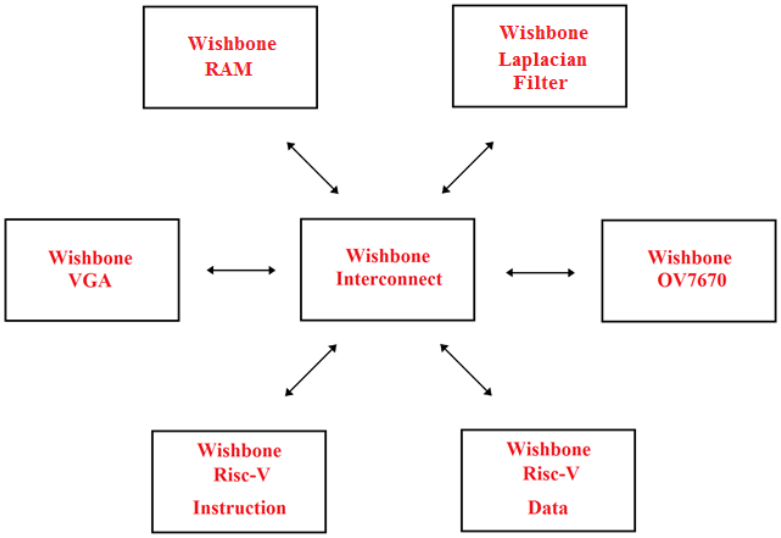


Figure 7.1 : Interconnect Management

Figure 7.2 shows the schematic diagram of the implemented SoC with wishbone which is depicted in Figure 7.1. Wishbone RAM is the only slave and other IPs are acting as master. Wishbone camera module reads data from camera and writes pixels to RAM. Wishbone RISC-V instruction IP reads instructions from memory and executes them sequentially. It also reads/writes data to Wishbone RISC-V data memory and all the data stores in RAM. Wishbone Laplacian IP is added in order to implement the filter as a additional hardware to our system. All masters have direct access to the RAM over wishbone interconnect.

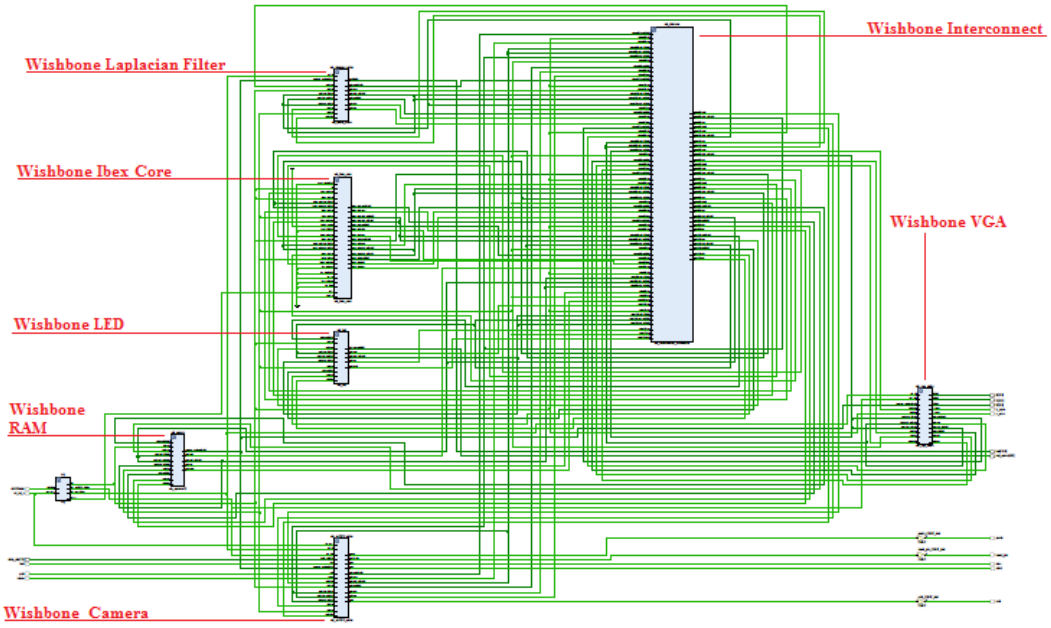


Figure 7.2 : Schematic Diagram of the Implemented SoC with Wishbone Interface.

7.1 Peripherals

After the successful implementation of Ibex core, we are going to add the related peripherals in order to make the other units communicate with each other over a single data bus.

7.1.1 Wishbone ibex core

In order to start, we need to fetch the wishbone compatible source codes from [14]. In the top module, "ibex_soc", we can see the management of masters and slaves. As a default, there are two masters, one for instruction memory, one for data memory, and three slaves as debug mode, RAM, and LED module with their base addresses.

7.1.2 Wishbone LED

A basic led program as a slave is included in the project folder. We are going to implement the module to figure out the system's integrity to develop for further peripheral integration. After generating the .mem file for RAM initialization, we are

going to add it to the verilog project for RAM initialization, which is declared in Chapter5. After implementation, bitstream generation, and programming the whole system worked properly.

7.1.3 Wishbone GPIO

GPIO module is going to be added as another slave to the system. Directions of GPIO can be input or output. System verilog code for wishbone compatible GPIO module is included in *APPENDIXD*. After including related headers from Pulpino [31], using the gpio_init function, GPIO struct is initialized with the base address of GPIO module, then the pin direction is set using gpio_set_direction function. Inputs and outputs can be initialized with gpio_get_input and gpio_set_output functions. To get value from a pin, gpio_set_pin is used, and to set value for a pin, gpio_set_pin and gpio_clear_pin function must be instantiated. C program is written to use the GPIO module as shown in Figure 7.3. Two switches are initialized as inputs, and 5 LEDs are defined as outputs. According to the switches position, four operations as, upcounting, downcounting, bitwise not and reset applied. The result is shown on LEDs.

```
struct gpio gpio0;

int main(int argc, char **argv) {

  volatile uint32_t *var = (volatile uint32_t *) 0x00010000;

  gpio_initialize     (&gpio0, (volatile void *) PLATFORM_GPIO_BASE);
  gpio_set_direction  (&gpio0, 0x00);
  *var = gpio_get_input(&gpio0);


  while (1) {
    usleep(1000 * 200); // 200 ms
    if (*var <0) *var=63;
    if      (gpio_get_input(&gpio0) == 3) {*var = *var + 1;}
    else if (gpio_get_input(&gpio0) == 2) {*var = *var - 1;}
    else if (gpio_get_input(&gpio0) == 1) {*var = ~(*var);}
    else if (gpio_get_input(&gpio0) == 0) {*var =0;}
  }
}
```

Figure 7.3 : GPIO C Code.

7.1.4 Wishbone camera & VGA

7.1.4.1 Camera-VGA IP

In this project, we are going to use OV7670 camera module. The OV7670 camera chip image sensor [6] is a low voltage CMOS sensor that provides the full functionality

of a single chip VGA camera and image processor in a small footprint package. The OV7670 provides full frame, sub-sampled or windowed 8 bit images in a wide range of formats, whoh is controlled via the Serial Camera Control Bus (SCCB) interface. OV7670 camera sensor pinouts are shown in Figure 7.4. This camera module has an image array capable of operating up to 30 FPS, in which, user has full control over image quality, formatting and output data transfer type. All required image processing functions, including exposure control, gamma, white balance, color saturation, hue control and more, are also programmable through the SCCB interface. In addition, OmniVision sensors use proprietary sensor technology to improve image quality by reducing or eliminating common lighting/electrical source of image contamination, such as fixed pattern noise (FPN), smearing, blooming, etc., to produce a clean, fully stable color image. OV7670 camera module has these features:

- Size of Pixel : 3.6 um x 3.6 um
- Resolution : 640 x 480 (VGA), 320x240(QVGA), ...
- Frame rate : 30 fps
- Color format : RGB, YUV(4:2:2) and YCbCr(4:2:2)
- Scan mode : Progressive
- Output : Parallel (16 bit for YCbCr)
- Power supply : max 3.0 V
- Interface : I2C

Now everything is ready and we are going to make the hardware connection as shown in Figure 7.5.

In order to get started, a verilog source code is gotten from [16]. An IP of OV7670 camera module is configured and generated in order to add as an block to our system as shown in Figure 7.6. Necessary changes are done over .xdc file to match the related I/Os to our FPGA board.

After this step, bitstream is generated and the video stream is shown in Figure 7.7 as expected.

PIN NO	NAME	PIN NO	NAME
1	NC	13	MCLK
2	AGND	14	D8
3	SDA	15	DGND
4	AVDD(2.8V)	16	D7
5	SCL	17	PCLK
6	RESET	18	D6
7	VSYNC	19	D2
8	PWDN	20	D5
9	HSYNC	21	D3
10	DVDD(1.8)	22	D4
11	DOVDD(2.8V)	23	D1
12	D9	24	D0

a **b**

Figure 7.4 : (a) OV7670 Camera Module (b) Pinouts of Camera Module [6]

.

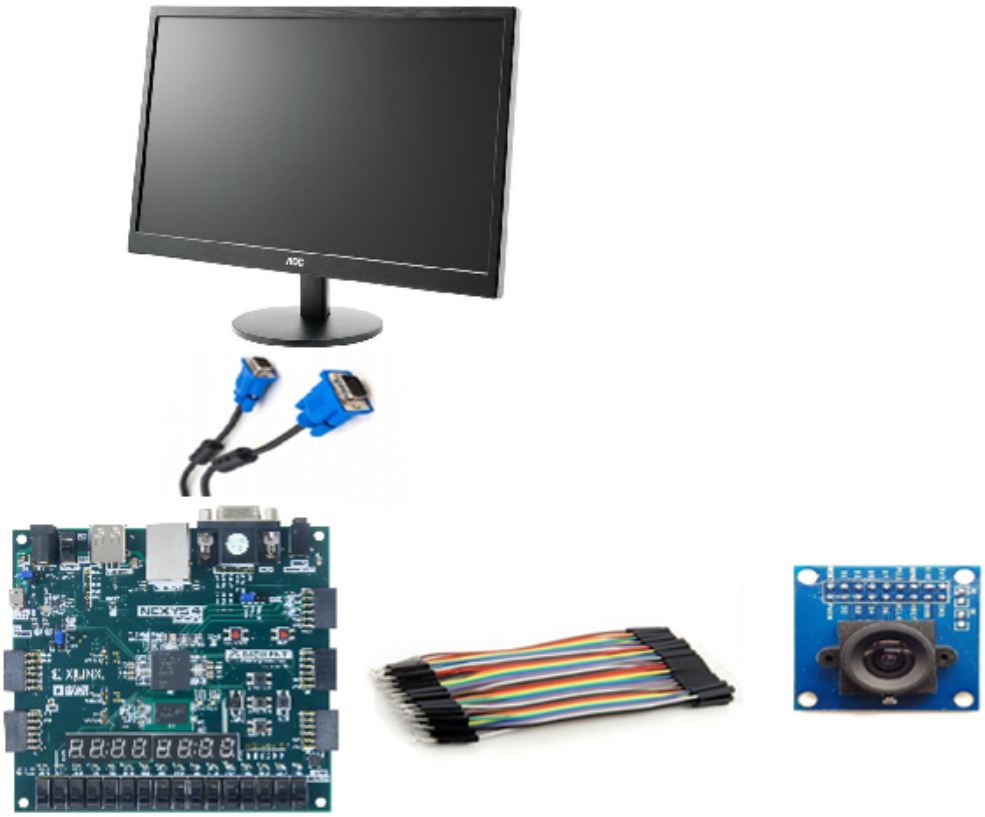


Figure 7.5 : Hardware Connection of System.

7.1.4.2 Example of camera-VGA IP implementation

All the pixels which are coming from camera module, should be stored in memory for further process. Default memory management of the Ibex core is not enough for

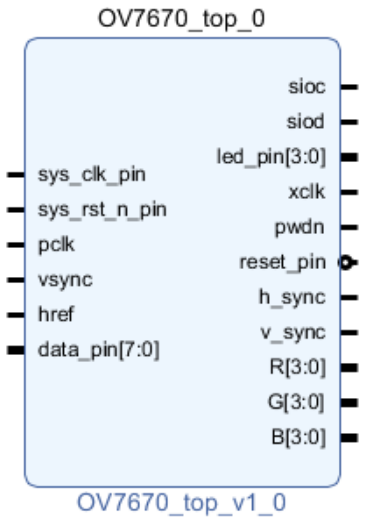


Figure 7.6 : IP Configuration.

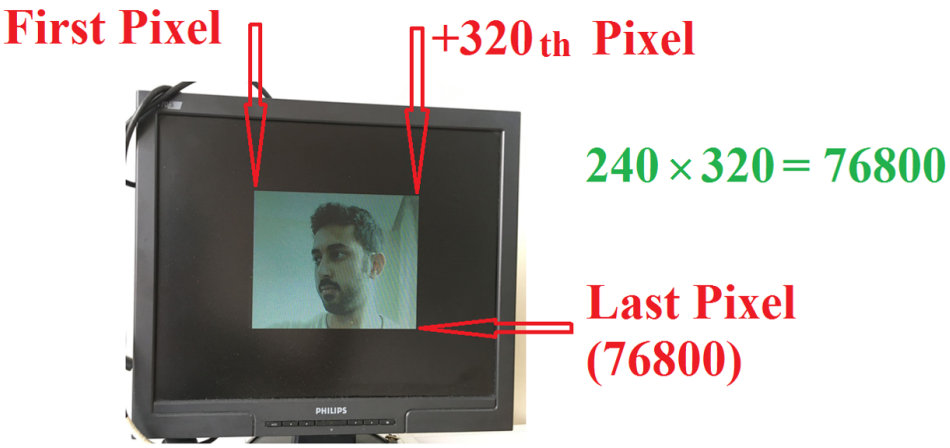


Figure 7.7 : OV7670 QVGA Video Stream.

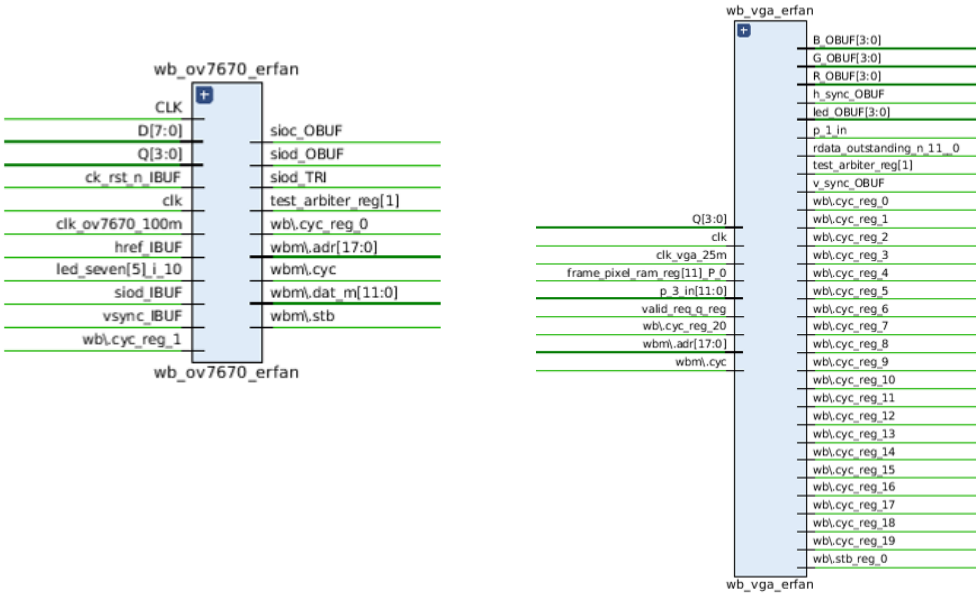
saving the data. Therefore, memory should be increase by altering the memory size in 'link.ld' file as shown in Figure 7.8.

```
MEMORY
{
    rom     : ORIGIN = 0x00000000, LENGTH = 0xC000    /* 48 kB */
    stack   : ORIGIN = 0x0000C000, LENGTH = 0x40000   /* 256 kB */
}

/* Stack information variables */
_min_stack      = 0x2000;   /* 8K - minimum stack space to
reserve */
_stack_len      = LENGTH(stack);
_stack_start    = ORIGIN(stack) + LENGTH(stack);
```

Figure 7.8 : Increase Memory Size to Save All the Pixels.

7.1.4.3 Add wishbone signals to camera-VGA modules

After IP validation, wishbone interface has been added in order to make this IP communicate with other peripherals. RTL schematic and pinouts of Camera and VGA module is shown in Figure 7.9.



a **b**

Figure 7.9 : (a) Wishbone Compatible OV7670 (b) & VGA RTL Schematic.

Now, environment is ready and we can test the modules. A basic C program as a filter is written as shown in Figure 7.10 to read the pixels from data memory and for those whose pixel value is more than $0xAAA$ (Whitish Pixels), red pixels are replaced with original pixels as depicted in Figure 7.11.

```
  delay_loop_ibex(usec_cycles);
  return 0;
}
static int usleep(unsigned long usec) {
  return usleep_ibex(usec);
}

volatile uint32_t *pixel_ram = (volatile uint32_t *) 0xC010;

int main(int argc, char **argv) {

volatile uint32_t  i;

  while (1) {

      for (i = 0 ; i<76800; i = i + 1)
      {
       if (pixel_ram[i]> 0xAAA) pixel_ram[i]=0xF00;
      }

     }

}
```

Figure 7.10 : C Program for Testing The Camera & VGA Modules.

Figure 7.11 : Output of Basic Filter.

8. SoC LAPLACIAN FILTER IMPLEMENTATION

8.1 Laplacian Filter

Laplacian filter is used as an edge detector [32]. Generally 3×3 convolution kernels are used. The center of the kernel is a high value, which is surrounded by smaller and opposite signed values. This way, the edges in the image are emphasized by introducing sudden changes in the output pixel values [32] as described in section 1.2. The output image contains bright tones for the edges and black tones in the surrounding. The sum of the kernel coefficients is kept at zero, therefore gain is avoided.

The edge detection method is an important part of the image processing system [33]. Edge detection techniques are generally used for finding discontinuities in gray level images [32]. To detect consequential discontinuities in the gray level image is the important common approach in edge detection. It is the process used to find the boundaries of objects or textures depicted in the image. Knowing the locations of these boundaries is critical in the process of image enhancement, recognition, restoration, and compression [2]. The edges of the image are considered among the most important features of the image, providing valuable information for human visual perception. There are different methods suggested to improve the edge detection method in real images [34]. With the edge detection method, image processing speed becomes a difficult problem because the larger the images, the larger the data. Among all methods mentioned in [35], the Laplacian operator is used for edge detection for its quality and clarity of edge-detecting as compared in [35].

An extended discrete Laplacian mask

$$G = \begin{pmatrix} 0 & -1 & 0 \\ -1 & 4 & -1 \\ 0 & -1 & 0 \end{pmatrix} \tag{8.1}$$

as described in [32] is used in this work. These kernels are not only used as an edge detection alternative, but the Laplacian pyramids which are first presented in

[13] are preferred for detail manipulation. Hardware accelerators for edge detection applications are investigated in the next section.

8.2 Custom IP Design For Laplacian Filter

In this section, the building blocks of our Laplacian filter are briefly described. In each cycle, required data for the filter is taken from RAM in each cycle. A Laplacian filter kernel convolved by each input pixel comes from the camera module.

The communication flow between the processor and the custom IP has the following steps [5]:

1. The processor should assert the assigned address to the Laplacian filter register (master_arbiter) in order to activate the IP. For the reading procedure, the master presents an address and data on the communication bus.
2. Custom IP negates [WE_O] to indicate a read cycle, asserts [CYC_O] and [STB_O] to start the cycle.
3. RAM decodes inputs, and asserts [ACK_I].
4. Custom IP presents valid data on [DAT_I] and asserts [ACK_I] in response to [STB_O] to indicate valid data.
5. Custom IP negates [STB_O] to indicate end of the data phase.

Received data is multiplied by the kernel constant and stored in add_mul_result register until end of the calculation. Calculations are completed in 9 cycles and the ultimate result is written back to the RAM.

In order to implement a custom IP for Laplacian filter with the mentioned wishbone interface, first we have drawn the block diagram shown in Figure 8.1.

1. Wishbone interconnect controls all masters and slaves. The data flow of the whole system passes through this block.
2. As soon as the IP is activated, the communication cycle starts.
3. State machine is responsible to generate 9 event for each pixel calculation. At the end of 9 states, the result is ready and it will write back to RAM.

4. In calculation unit, input pixel from the RAM is multiplied by kernel element and the multiplication result is accumulated with the previous one, which is stored in conv_result register.

5. All read/write processes require address. In address generator, both addresses are generated for the read and the write operations upon request of the state machine. At the end of the operation for one pixel, which takes 9 cycles, the final result of the calculation is written in add_mul_result. Finally, the content of add_mul_result register is written to the dedicated address of the RAM by using over wishbone protocol.

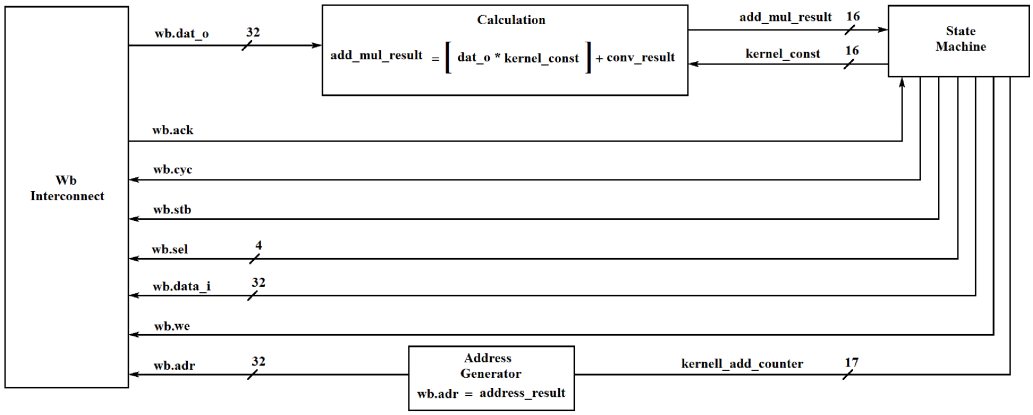


Figure 8.1 : Block Diagram of the Laplacian Filter.

The custom IP is designed with Verilog HDL and implemented using Xilinx Vivado. It mainly consists of a state machine, a calculation unit, an address generator, and a wishbone protocol as shown in Figure 8.3. The state machine controls the wishbone signals, sends kernel elements for calculation, and generates the required address for the read/write processes as shown in 8.2. In calculation unit, there is a multiplication and addition circuit for convolution implementation. Address generator, generates the required address for processed data to be written in RAM. Finally, all the processed data are going to be written in RAM over wishbone protocol.

In order to start the simulation, first we need to write a program that will change picture to data and vise versa. The algorithm below, converts photo (lenna.png [36]) to text

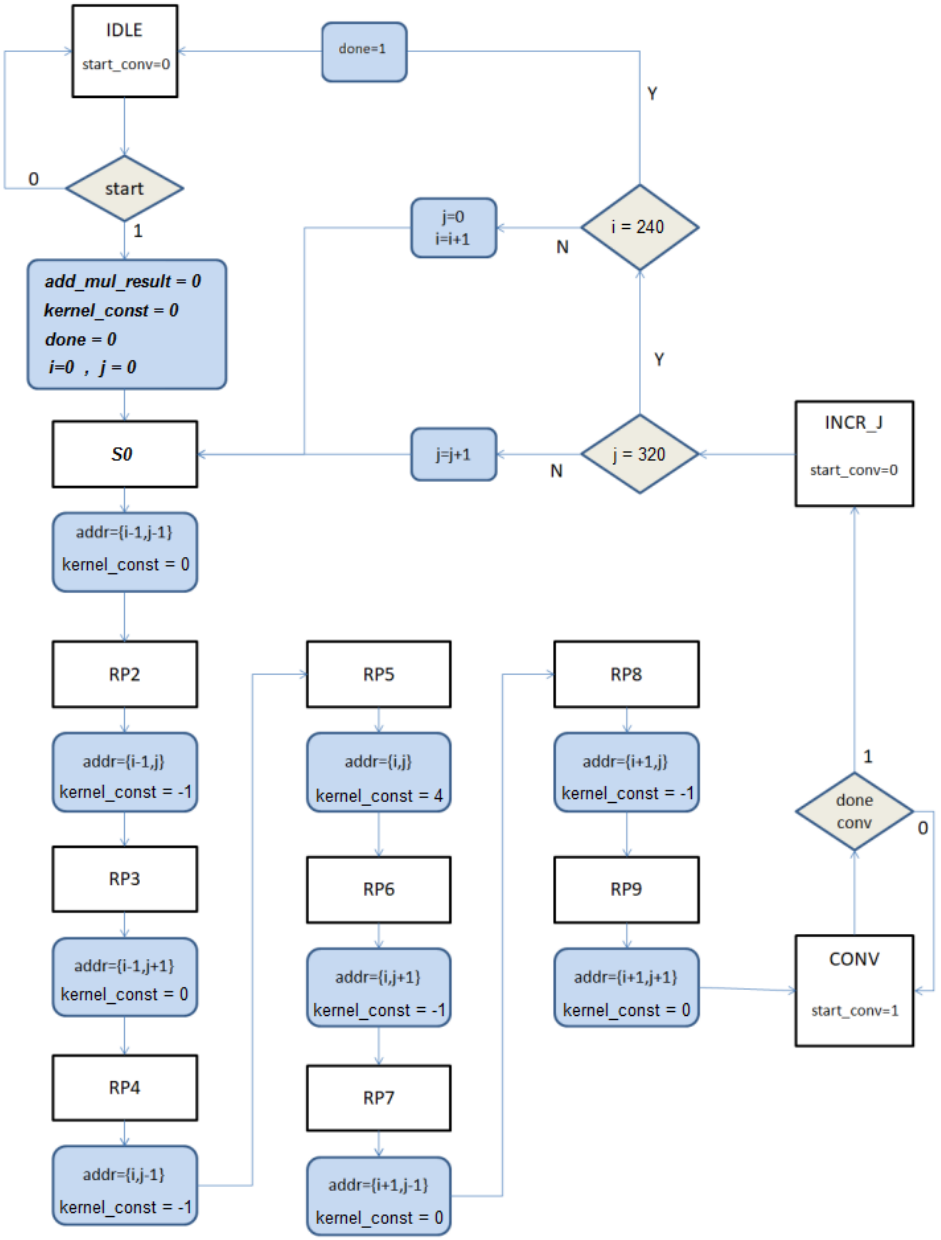


Figure 8.2 : State Machine of Laplacian Filter.

file (data_init.txt) and text file (data_from_verilog.txt) to photo (output.png). In test bench, RAM will initialize with "data_init.txt" and the processed data coming from IP core, will be stored in "data_from_verilog.txt". Finally, this data converted to the output photo. Detailed python code for these conversions are shown in *APPENDIXG*:

We simulated our custom IP Verilog code, the waveform is shown in Figure 8.4. To start a calculation process, a request is sent by the IP to the interconnect when this IP is

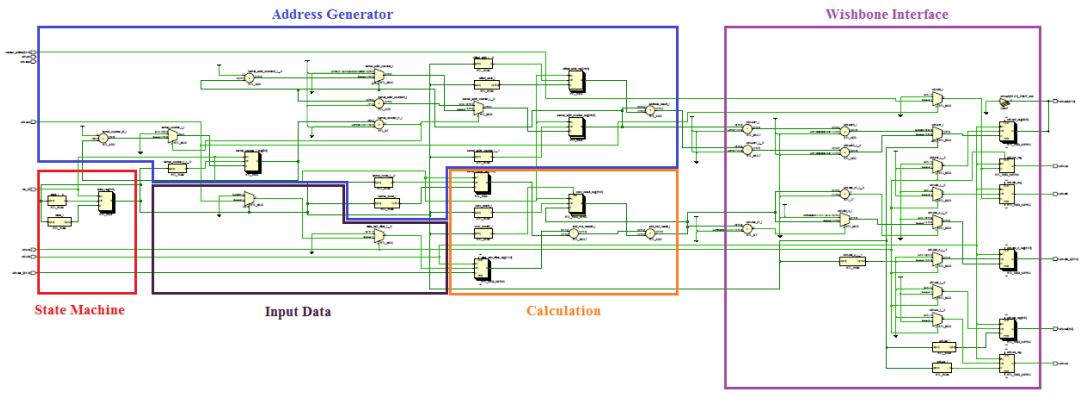


Figure 8.3 : Schematic Diagram of the Laplacian Filter Block.

active. An acknowledge from RAM indicates that data bus is ready for communication. In this phase, the address and read signals are sent over the data bus. A single input data is multiplied by the first element of kernel in the Laplacian filter. The result of this calculation is stored in conv_result register. After this step, the state machine changes the state of the process and generates a new address for the next input data. Then the result is calculated and stored in the conv_result register. Sequentially, these steps are repeated for 9 cycles in order to calculate one pixel of filtered image as shown in Figure 8.4. At the end of the ninth cycle, the calculation has totally done for a single output pixel and ready to be written in RAM. In this step, the state machine generates an offset address to update the address generator for the next pixel and it enables write signal to send data to RAM over wishbone protocol.

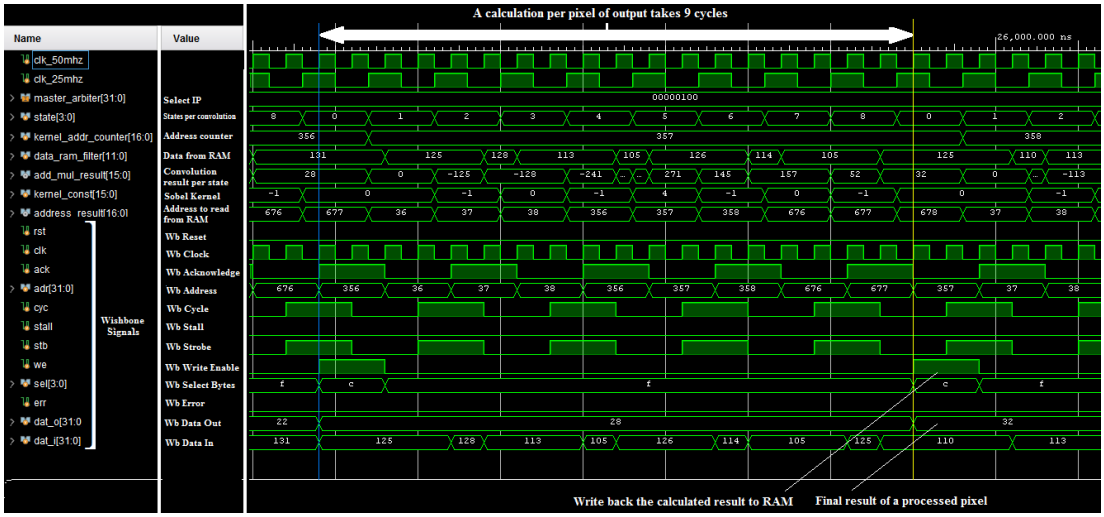


Figure 8.4 : Simulation of Implemented Laplacian Filter.

The design is tested on Lena image [36]. A test bench is written to apply an image to the custom IP input. Timing signals are analyzed and the output image is recorded. The input and output images are shown in Figure 8.5. Considering input image, in the points that brightness changes sharply or it has discontinuities, it appears as whitish pixel in output image which dedicates that the applied filter functions properly.

a **b**

Figure 8.5 : (a) Input and (b) Output Images of the Custom Laplacian Filter IP.

8.2.1 Hardware implementation utilization report

After Laplacian filter implementation, utilization report is generated to check for the implementation parameters as shown in Figure 8.6.

Resource	Utilization	Available	Utilization %
LUT	4900	63400	7.73
FF	2265	126800	1.79
BRAM	91	135	67.41
DSP	2	240	0.83
IO	54	210	25.71
PLL	1	6	16.67

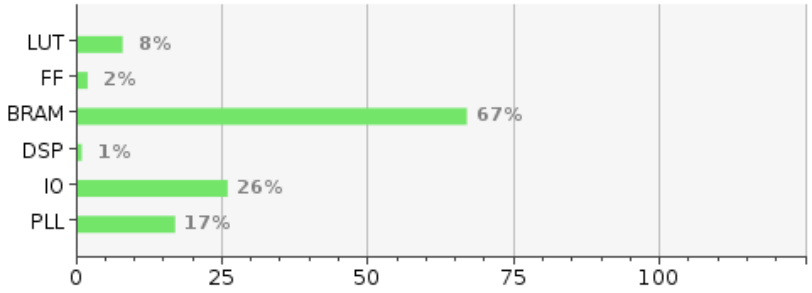


Figure 8.6 : Hardware Utilization Of Laplacian Filter.

8.3 Laplacian Filter Software Implementation

In this section, a software implementation of Laplacian filter has been covered. Data flow of this process is shown in Figure 8.7. Camera module (OV7670), VGA module, data memory and instruction memory act as master, while RAM acts as a slave. All masters have direct access to RAM and they have been activating in the following order: Camera module initializes the RAM with raw red-green-blue (RGB565 format) pixels, RISC-V core processes the image and writes back to memory and VGA module get data from the RAM to display on the monitor. For software implementation of Laplacian filter, a C program is written and compiled using RISC-V GNU toolchain. In this way a memory file is generated to initialize the instruction memory of RISC-V core. As hardware implementation of Laplacian filter, we add Laplacian filter as an IP to the following structure, this new IP acts as a master and has read/write permission directly to the RAM.

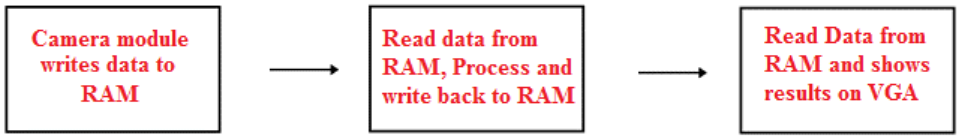


Figure 8.7 : Dataflow of Laplacian Filter Software Implemenatation.

For controlling the IPs of the whole system, a 32-bit memory space is allocated in stack region with 0xC010 starting address. For each IP an arbiter activator is assigned which is going to be controlled in C program. In Figure 8.8, the data flow of Figure 8.7 has been implemented. First of all, the camera module is activated by setting the related arbiter value. Basically, it reads the data from camera and write in memory. Next, is to start the image processing, calculations or other operations on pixels. Processed pixels are going to written back in memory. Finally, VGA module gets active and show the processed image:

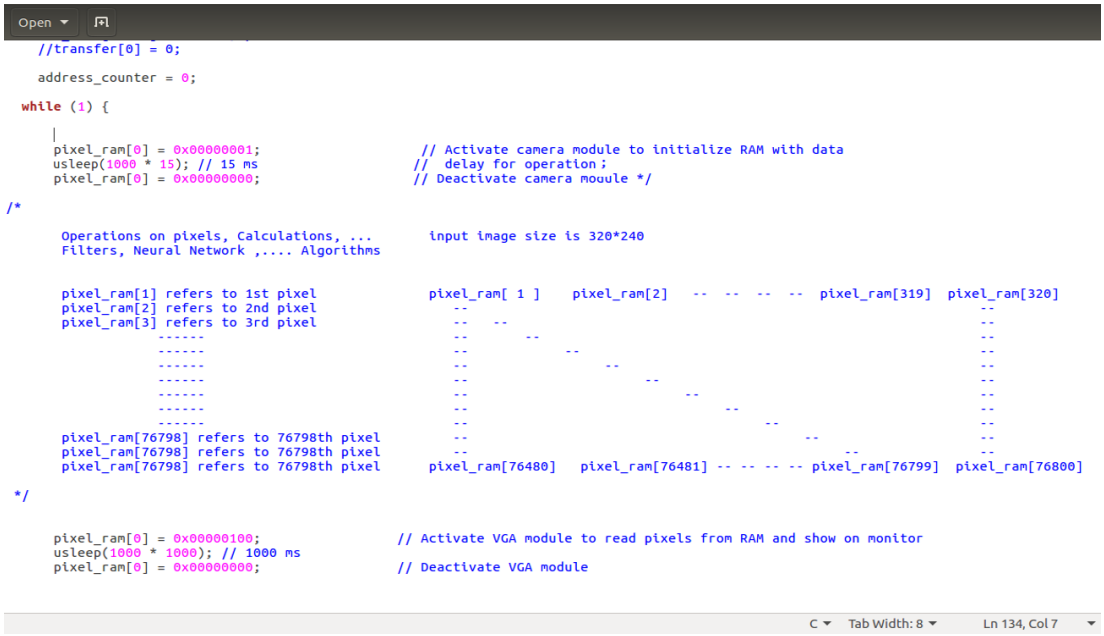


Figure 8.8 : Software Implementation of Image Processing Dataflow.

A simple C program as depicted in Figure 8.9 is written in order to apply the filter to the image coming from the camera module. After the process, data is ready to monitor in the VGA.

```
for (y = 0 ; y<240; y = y + 1)
{
  for (x = 1 ; x<321; x = x + 1)
    {
     pixel_val = 0 ;
     for (j = 0 ; j<3; j = j + 1)
     {
       for (i = 0 ; i<3; i = i + 1)
         {
            pixel_val = pixel_val + (pixel_ram[x+i+((y+j)*320)]&0xFF)*kernel[i][j];

         }
     }
     tester = pixel_ram[x+y*320];
     if (pixel_val>255) pixel_val = 255;
     if (pixel_val<=0) pixel_val = 0;
     pixel_ram[x+y*320] = ((pixel_val)<<16) | (tester&0xFFFF) ;
    }
 }
```

Figure 8.9 : Laplacian C code.

The captured image and the filtered output image are shown in Figure 8.10.

In this implementation, Laplacian filter is implemented without IP. Again, For comparison purposes, utilization report is generated and shown in Figure 8.11. As shown in the Table 8.1, in software implementation, the speed is much lower rather than hardware implementation and the area is smaller. On the other side, in hardware implementation, the speed is faster than in software implementation, however, in this

a **b**

Figure 8.10 : (a) Captured Image from Camera Module (b) Output Image From SoC Implementation.

Table 8.1 : Comparison of Software and Hardware Implementation.

	Slice LUTs	Slice Regis-ters	DSPs	Max. Freq. (Mhz)	Power (W)	Clock Cycles	Time (sec)
Hardware	4900	2265	2	52.55	0.250	1,869,750	0.037
Software	4842	2232	1	53.76	0.242	66,649,500	1.33

case, area is bigger. In hardware implementation, the system uses an extra DSP unit, which causes more energy and power consumption in compared to software implementation.

Resource	Utilization	Available	Utilization %
LUT	4842	63400	7.64
FF	2232	126800	1.76
BRAM	91	135	67.41
DSP	1	240	0.42
IO	54	210	25.71
PLL	1	6	16.67

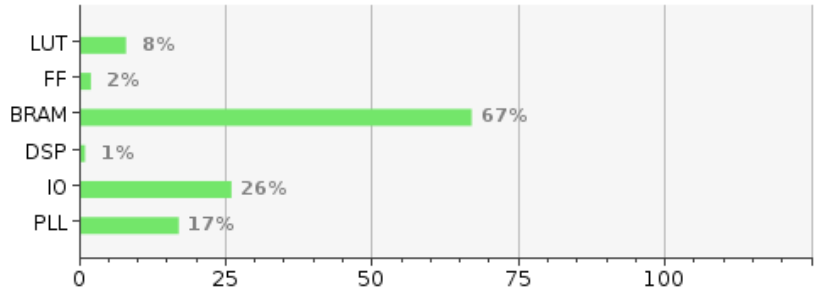


Figure 8.11 : Software Utilization of Laplacian Filter.

9. INSTRUCTION SET EXTENSION OF RISC-V PROCESSOR

9.1 Convolution and Multiply-Accumulate

In order to compare the results between software and hardware implementation, a 10×10 random data is given to the system. To measure the exact processing time, 0x0000 is set to demonstrate the start of the program execution and 0xffff is the finished value of conversion as shown in the simulation results in Figure 9.1 and Figure 9.2.

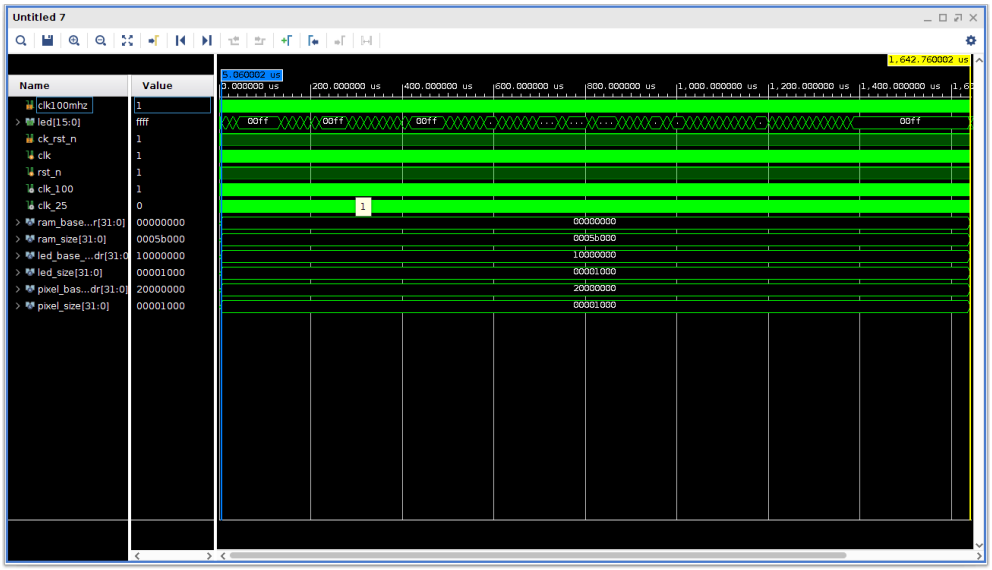


Figure 9.1 : Software Simulation of Random Data in SoC.

As shown in Figure 9.1 and Figure 9.2, in hardware implementation, calculations are taking much lower clock cycles compared to software implementation. There is a 65.6 % improvement in custom instruction implementation in our SoC system as calculated below:

$$\frac{1642ns - 566ns}{1642ns} = 0.65 \tag{9.1}$$

In iterative operations like convolutions, instructions are repeating in each cycle and spend many clock cycles as shown in figures above and Eq.9.1. In order to make the execution time much lower, we are going to add an additional hardware to our system

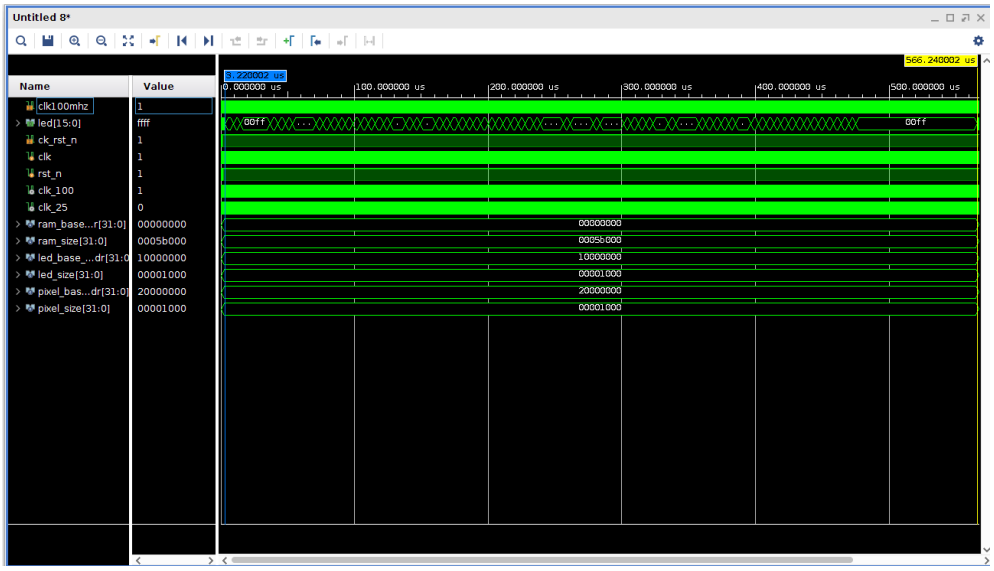


Figure 9.2 : Hardware Simulation of Random Data with Custom Instruction.

in order to make calculations parallel to reduce the processing cycle. In Laplacian filter, a single pixel needs 9 operations to be processed by multiply and addition of kernel with input image as described in section 1.2. This operation needs to be done in serial, and each calculation needs the previous result to be continued as shown in Figure 9.3.

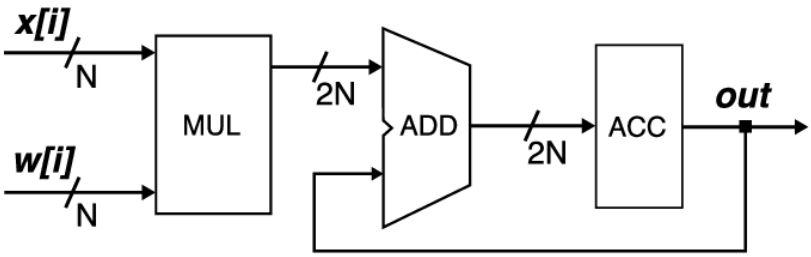


Figure 9.3 : Multiply-Accumulate Operation in Serial [7].

On the other hand, we can add additional hardware in order to make the MAC operation time much lower. In our work, additional multipliers are added and in the output stage, there is a single adder to calculate the output as shown in Figure 9.4.

For Multipliers, DSP units are used. Now, we are going to add this new operations as a custom instruction to our system for increasing the performance of convolution operations.

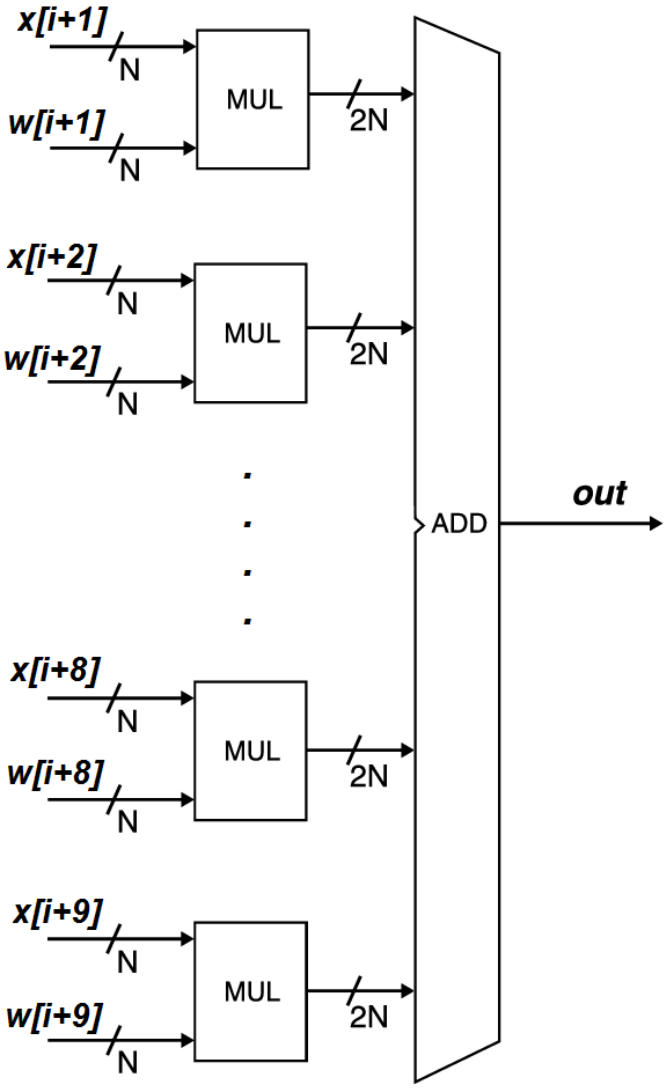


Figure 9.4 : Multiply-Accumulate Operation in Parallel.

9.2 Custom Instruction Addition to the RISC-V Compiler

In order to add a custom instruction extension to our system, the RISC-V compiler, ALU instruction decoder unit must be customized.

9.2.1 Compiler modification

Adding new instruction sets to RISC-V GNU Toolchain, are going to be made by modifying the related files in $riscv-bintuils$. As shown in Figure 9.5, "riscv-opc.c" and "riscv-opc.h" are the main files that are going to be modified by adding new instruction [12].

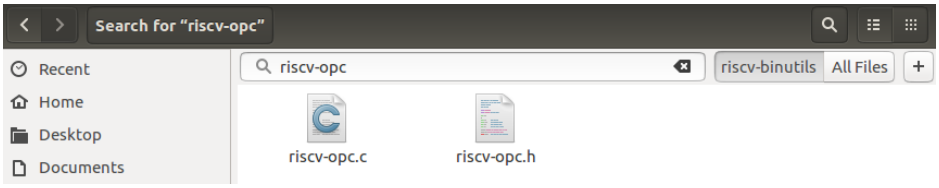


Figure 9.5 : Necessary Files to Modify RISC-V GNU Toolchain.

In the "riscv-opc.c" & "riscv-opc.h", the new instruction must be added to the *opcodes* structure [12]. Instruction format is as follows and applied to the compiler files in Figure 9.6:

name , isa , operands , match , mask , match_func .

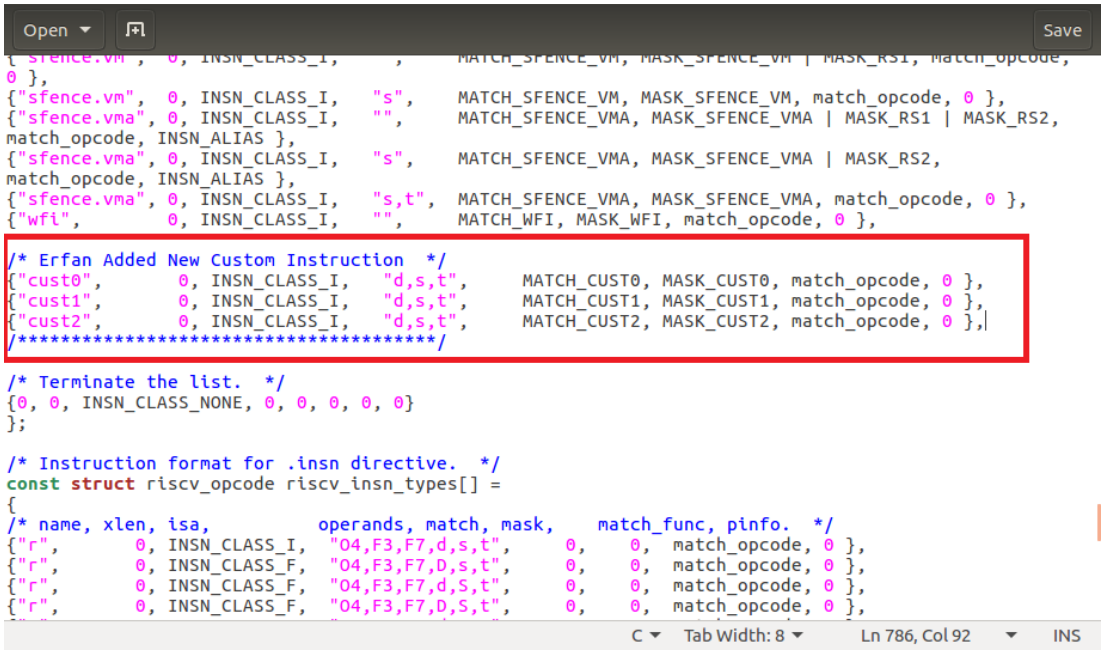


Figure 9.6 : riscv-opc.c Modification for Custom Instruction Addition.

Where, Cust belongs to the Integer ISA and that it is a triadic instruction, meaning that it takes 3 registers whose symbolic names are d,s,t. 'd' being the destination register, 's' and 't' being source registers [12]. MATCH, MASK and match_opcode are about the structure of instructions and it mask and unmask operands with these elements added into the structure. In our work, we need a single instruction, however, for future work we added 3 instructions. As well as "$riscv - opc.c$" modification, we should define the related custom instructions in "$riscv - opc.h$" file. It is necessary to check the newly added instruction shouldn't conflict with the default instructions by their hex value. Newly added custom instructions are shown in Figure 9.7.

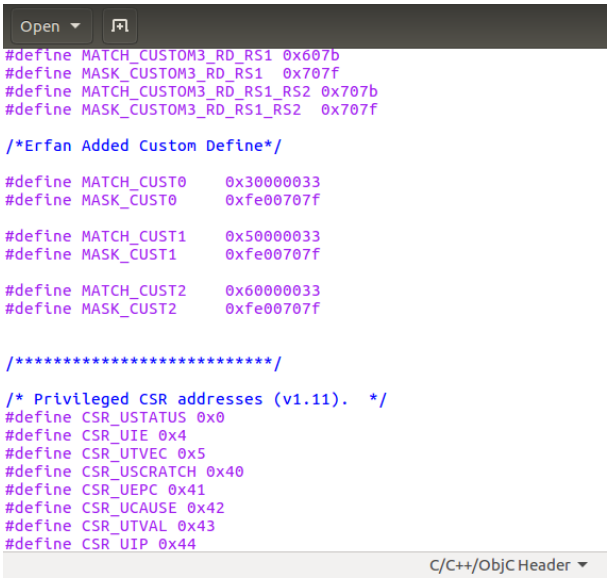


Figure 9.7 : riscv-opc.h Modification for Custom Instruction Addition.

Now, the modified compiler files are ready and we are going to rebuild the GCC GNU toolchain in order to make it recognize the new instructions by applying the following commands in terminal :

```
$ cd /opt/riscv
$ make distclean
$ ./configure --prefix=/opt/riscv [--with-arch=rv32imc
for 32-bit, --with-arch=rv64gc for 64-bit]
$ make
```

9.2.2 Instruction decoder and ALU modification

All the instructions are passing through the instruction decoder and ALU [**?**]. First, the instruction decoder, decodes the instruction which is going to be executed. Then the ALU gets two operands and applies the related function to them. Based on the instruction definition in Figure 9.7, instruction decoder has been modified as depicted in Figure 9.8:

In our system, camera module output is 16-bit (565 RGB format). We are going to convert it to gray scale by summing up the RGB channels. This conversion will give us the 7-bit grayscale image. As ALU gets only two 32-bit operands and our pixels for processing is 7-bit wide each, we are going to compress all 9 pixels for calculation in two operands as shown in Figure 9.9

Final step, is to add the related hardware for our newly added instruction. Considering our kernel for Laplacian filter in eq. 8.1, we can implement -1 as $2'scomplement + 1$

Project Summary | ibex_decoder.sv

_ibex_ext.srcs/sources_1/imports/Desktop/ibex/ibex_decoder.sv

```
418             // RV32I ALU operations
419             {6'b00_0000, 3'b000}: alu_operator_o = ALU_ADD;   // Add
420             {6'b10_0000, 3'b000}: alu_operator_o = ALU_SUB;   // Sub
421             {6'b00_0000, 3'b010}: alu_operator_o = ALU_SLT;   // Set Lower Than
422             {6'b00_0000, 3'b011}: alu_operator_o = ALU_SLTU;  // Set Lower Than Unsigned
423             {6'b00_0000, 3'b100}: alu_operator_o = ALU_XOR;   // Xor
424             {6'b00_0000, 3'b110}: alu_operator_o = ALU_OR;    // Or
425             {6'b00_0000, 3'b111}: alu_operator_o = ALU_AND;   // And
426             {6'b00_0000, 3'b001}: alu_operator_o = ALU_SLL;   // Shift Left Logical
427             {6'b00_0000, 3'b101}: alu_operator_o = ALU_SRL;   // Shift Right Logical
428             {6'b10_0000, 3'b101}: alu_operator_o = ALU_SRA;   // Shift Right Arithmetic
429
430             // Erfan Added Custom instructions
431             {6'b01_1000, 3'b000}: alu_operator_o = ALU_CUST0; // Custom 0:
432             {6'b10_1000, 3'b000}: alu_operator_o = ALU_CUST1; // Custom 1
433             {6'b11_0000, 3'b000}: alu_operator_o = ALU_CUST2; // Custom 2
434
435
436
437             // supported RV32M instructions
438             {6'b00_0001, 3'b000}: begin // mul
439               alu_operator_o        = ALU_ADD;
440               multdiv_operator_o    = MD_OP_MULL;
441               mult_en_o             = RV32M ? 1'b1 : 1'b0;
```

Figure 9.8 : Instruction Decoder Modification.

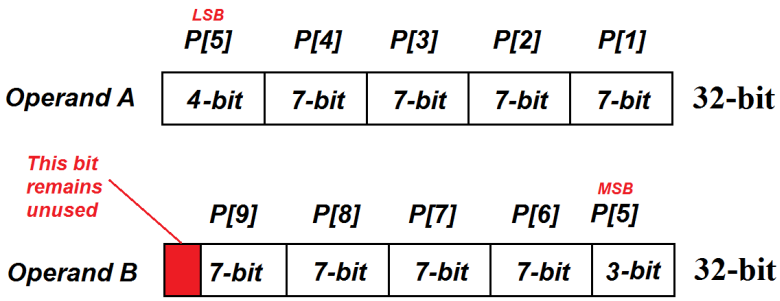


Figure 9.9 : Custom Instruction Operand Management.

and 4 as $2-bit-left-shift$ at our design in Figure 9.4. Modified ALU for this design is shown in Figure 9.10.

9.3 Performance Analysis of Custom Instruction Addition

Inspiring from the added custom instruction for random data simulation, we are going to add another custom instruction to customize the kernel value for each convolution. The first register of the added instruction points to the index of the kernel matrix and the other one refers to the kernel value as shown in Figure 9.11.

Therefore, a custom kernel matrix for convolution processing is going to be available in each calculation as shown in Figure 9.12.

```
always_comb
begin
   if(temp_cust1_result[31]==1'b1) cust1_result=32'b0;
   else if (temp_cust1_result[30:8]!=0) cust1_result = 32'hff;
   else cust1_result = temp_cust1_result;
end

assign temp_cust1_result =
(~temp1_cust1 + 1  ) +
(~temp2_cust1 + 1  ) +
( temp3_cust1      ) +
(~temp4_cust1 + 1  ) +
(~temp5_cust1 + 1  );

assign temp1_cust1 = operand_a_i[13:7];
assign temp2_cust1 = operand_a_i[27:21];
assign temp3_cust1 = {operand_b_i[2:0],operand_a_i[31:28],2'b00};
assign temp4_cust1 = operand_b_i[9:3];
assign temp5_cust1 = operand_b_i[23:17];
)
```

Figure 9.10 : ALU Modification.

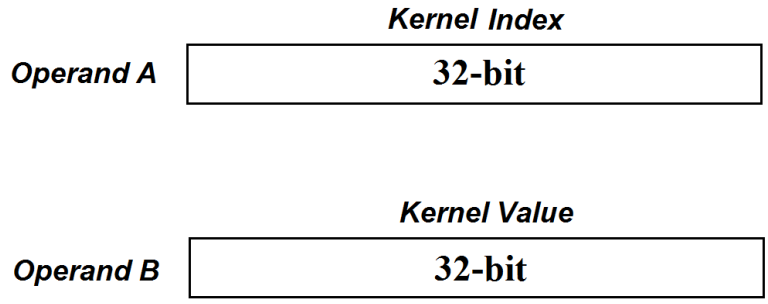


Figure 9.11 : Custom Kernel Operands.

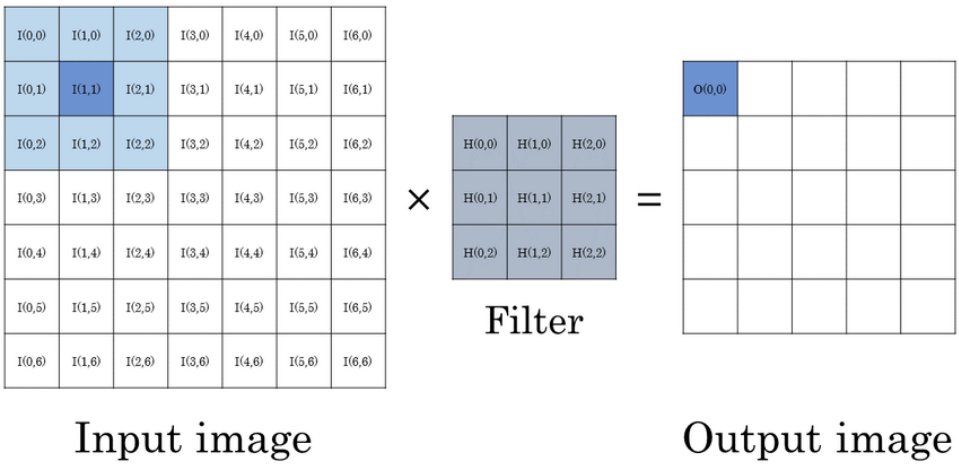


Figure 9.12 : Custom Matrix Kernel.

In this level, we are going to add ALU a sub-module to load the related values for kernel when the custom instruction command comes from the instruction decoder unit

as shown in Figure 9.13 & Figure 9.14. Since the kernel matrix elements are much smaller than 32-bits, overflow won't occur in the result register.

```
`default_nettype wire

import ibex_pkg::*;

module ibex_custom_reg (
    input  logic                 clk_i,
    input  logic                 rst_ni,
    input  ibex_pkg::alu_op_e    cust_operator_i,
    input  logic [31:0]          cust_operand_a_i  ,
    input  logic [31:0]          cust_operand_b_i  ,

    output logic [0:8][31:0]     cust_kernel_val

);

logic [3:0] kernel_id ;

assign  kernel_id = cust_operand_a_i[3:0];

always @( posedge clk_i)
 begin
  if (cust_operator_i == ALU_CUST0)
   cust_kernel_val[kernel_id] = cust_operand_b_i;
 end

endmodule
```

Figure 9.13 : Kernel Matrix Load/Store.

```
logic [31:0] cust1_result;

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

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

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

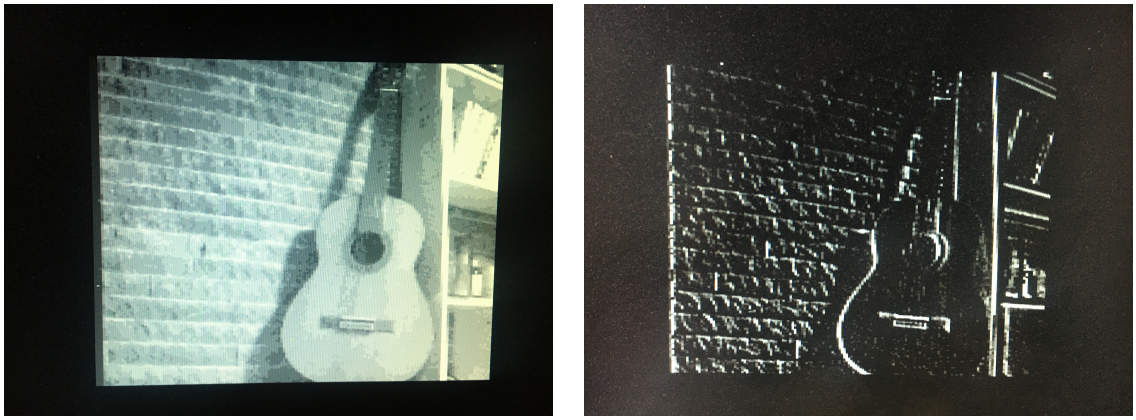
Figure 9.14 : MAC Implementation with Custom Kernel.

In inline assembly method during programming, first the custom0 function should call in order to initialize the kernel, then custom1 function can be used for MAC calculations. Inline method is shown in Figure 9.15.

```
static int custom0(unsigned int a, signed int b) {
  static int result;
  asm volatile( "cust0 %[result1], %[value1], %[value2]\n\t"
   :[result1] "=r" (result) : [value1] "r" (a), [value2] "r" (b));
return result;
}

static int custom1(unsigned int a, unsigned int b) {
  static int result;
  asm volatile( "cust1 %[result1], %[value1], %[value2]\n\t"
   :[result1] "=r" (result) : [value1] "r" (a), [value2] "r" (b));
return result;
}
```

Figure 9.15 : Inline Assembly Function Declaration.

Now, the environment has been set and the system is ready for implementing Laplacian filter using newly added custom instructions. The implementation both in software and hardware are done to compare the timing and hardware usage of the SoC. Input/output pictures are shown in Figure 9.16.



a **b**

Figure 9.16 : (a) Input and (b) Output Images of the Laplacian Filter Implementation.

As an another implementation, we are going to implement the Smooth filter. Smooth filter kernel [19], which is shown below, is used for filter implementation:

$$S = \begin{pmatrix} 1/9 & 1/9 & 1/9 \\ 1/9 & 1/9 & 1/9 \\ 1/9 & 1/9 & 1/9 \end{pmatrix} \tag{9.2}$$

The elements of the smooth filter, are float, therefore we need to make a small change in ALU unit to support for floating point convolution calculation. While initializing the kernel using custom0 function, we are going to insert the value multiplied by 16 and round to nearest integer. For example, in smoothing filter initialization, we are going to insert 2 instead of 1/9 $\left(\frac{16}{9} \approx 2\right)$.

```
// Load Kernel Data
for (j = 0 ; j<3; j = j + 1)
   for (i = 0 ; i<3; i = i + 1)
        custom0( j*3+i   , 2 );
```

Figure 9.17 : Smooth Filter Initialization.

On the ALU side, after multiply and addition process, we are going to shift the result 4 bit to right, preserving the sign bit by using the »> operator.

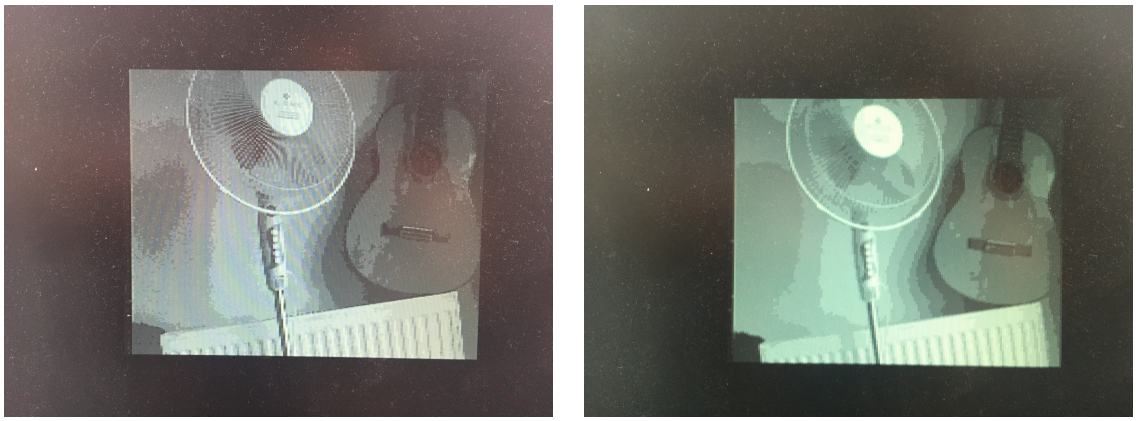
```
logic signed [31:0] cust1_result;
wire signed [31:0] temp_result ;

always_comb
begin
cust1_result = temp_result>>>4;
end

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

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

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

Figure 9.18 : ALU modification to Support Float Filter Kernels.

Smooth filter implementation results are depicted in Figure 9.19.

Table 9.1 shows execution time and area usage of the Laplacian filter mentioned in 3 methods and smooth filter implementation.

Considering the table 9.1, the Laplacian filter which is implemented with custom instruction is taking much less time to be completed compared to the custom kernel implementation, because in custom instruction the conditional comparing of processed pixels are implemented in the SoC and in custom kernel it is embedded in software. The only difference between Laplacian filter & Smooth filter with custom kernel is that in Laplacian filter, the whole process takes a little more time than smooth filter



a **b**

Figure 9.19 : (a) Input and (b) Output Images of the Smooth Filter Implementation.

Table 9.1 : Comparison of Software and Hardware Implementation.

	Slice LUTs	Slice Regis-ters	DSPs	Max. Freq. (Mhz)	Power (W)	Clock Cycles	Time (sec)
Laplacian Filter with Custom Instruction	5197	2233	1	52.05	0.298	23,753,281	0.474
Laplacian Filter With Custom Kernel	5300	2233	19	47.62	0.305	25,106,315	0.501
Smooth Filter With Custom Kernel	5316	2233	19	47.62	0.302	23,753,281	0.474
Laplacian Filter in Software	5263	2232	1	53.76	0.242	66,649,500	1.33

because of mentioned conditional comparing in implementation. Overall, in software implementation the power consumption is less than the hardware implementation of filters because less hardware is used in this implementation compared to hardware implementation and so, the power consumption is less than other implementations. However, in software implementation, the total pixels processing time is much more than other methods.

10. CONCLUSIONS AND FUTURE WORK

In this study, the RISC-V open source core implementation and custom instruction are discussed. The benefits of adding accelerators are introduced and implemented. Laplacian filter, which is one of the important transforms for edge detection, has been studied in software and hardware implementation and compared in terms of execution time and area usage. For an additional filter kernel implementations, a custom instruction is added in order to change the kernel for each MAC calculation of the input pixels. Inspiring from this study, a neural network algorithms are going to be trained and implemented for driver fatigue detection algorithms.

REFERENCES

[1] **RISC-V**, (2019), RISC-V Specification, Volume 1, Unprivileged Spec v.

[2] **Sreenivasulu, M. and Meenpal, T.** (2019). Efficient Hardware Implementation of 2D Convolution on FPGA for Image Processing Application, *2019 IEEE International Conference on Electrical, Computer and Communication Technologies (ICECCT)*, pp.1–5.

[3] **Saini, A. and Biswas, M.** (2019). Object Detection in Underwater Image by Detecting Edges using Adaptive Thresholding, *2019 3rd International Conference on Trends in Electronics and Informatics (ICOEI)*, pp.628–632.

[4] **LowRISC**, Ibex Core, `https://github.com/lowRISC/ibex`.

[5] **Herveille, R.** (2010). WISHBONE System-on-Chip (SoC) Interconnection Architecture for Portable IP Cores, **Technical Report**, OpenCores.

[6] **OmniVision**, (2006), OV7670/OV7171 CMOS VGA (640x480) CameraChip Sensor with OmniPixel Technology.

[7] **Nahmias, M.A.**, **de Lima, T.F.**, **Tait, A.N.**, **Peng, H.T.**, **Shastri, B.J. and Prucnal, P.R.** (2020). Photonic Multiply-Accumulate Operations for Neural Networks, *IEEE Journal of Selected Topics in Quantum Electronics*, *26*(1), 1–18.

[8] **Moreno, F.**, **Aparicio, F.**, **Hernandez, W. and Paez, J.** (2003). A low-cost real-time FPGA solution for driver drowsiness detection, *IECON'03. 29th Annual Conference of the IEEE Industrial Electronics Society (IEEE Cat. No.03CH37468)*, volume 2, pp.1396–1401 Vol.2.

[9] **Chinedu, O.K.**, **Genevera, E.C. and Akinyele, O.O.** (2011). Hardware description language (HDL): An efficient approach to device independent designs for VLSI market segments, *3rd IEEE International Conference on Adaptive Science and Technology (ICAST 2011)*, pp.262–267.

[10] **Monmasson, E.**, **Idkhajine, L. and Naouar, M.W.** (2011). FPGA-based Controllers, *IEEE Industrial Electronics Magazine*, *5*(1), 14–26.

[11] **RISC-V**, RISC-V FOUNDATION,, `https://riscv.org/`.

[12] **Srivastava, N.**, (2017), Adding custom instruction to RISCV ISA, `https://nitish2112.github.io/post/adding-instruction-riscv/`.

[13] **Paris, S.**, **Hasinoff, S. and Kautz, J.** (2011). Local Laplacian Filters: Edge-aware Image Processing with a Laplacian Pyramid, *ACM Trans. Graph.*, *30*, 68.

[14] **pbing**, (2019), RISC-V Ibex core with Wishbone B4 interface, `https://github.com/pbing/ibex_wb`.

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

[16] **Zheng, T.**, (2018), Video stream from OV7670 camera and displays via VGA on Artix7 FPGA, `https://github.com/Tom-Zheng/OV7670_to_VGA_FPGA`.

[17] **LowRISC**, Ibex Core Documentation, `https://ibex-core.readthedocs.io/en/latest/`.

[18] **Yaman, S.**, **Karakaya, B. and Erol, Y.** (2019). Real Time Edge Detection via IP-Core based Sobel Filter on FPGA, *2019 International Conference on Applied Automation and Industrial Diagnostics (ICAAID)*, volume 1, pp.1–4.

[19] **Fernando, S.**, (2018), Smooth Filter Kernel, `https://www.opencv-srf.com/2018/01/filter-images-and-videos.html`.

[20] **Onat, E.** (2017). FPGA implementation of real time video signal processing using Sobel, Robert, Prewitt and Laplacian filters, *2017 25th Signal Processing and Communications Applications Conference (SIU)*, pp.1–4.

[21] **Singh, S.**, **Saurav, S.**, **Saini, R.**, **Saini, A.K.**, **Shekhar, C. and Vohra, A.** (2014). Comprehensive review and comparative analysis of hardware architectures for Sobel edge detector, *International Scholarly Research Notices*, *2014*.

[22] **Khandelwal, S.**, **Choudhury, Z.**, **Shrivastava, S. and Purini, S.** (2020). Accelerating Local Laplacian Filters on FPGAs, *2020 30th International Conference on Field-Programmable Logic and Applications (FPL)*, pp.109–114.

[23] **Almazrooie, M.**, **Abdullah, R.**, **Yi, L.**, **Venkat, I. and Abdalkareem, Z.** (2014). Parallel Laplacian Filter Using CUDA on GPGPU.

[24] **Altameemi, A.A. and Bergmann, N.W.** (2016). Enhancing FPGA softcore processors for digital signal processing applications, *2016 Sixth International Symposium on Embedded Computing and System Design (ISED)*, pp.294–298.

[25] **Xilinx**, Virtex-5, `https://www.xilinx.com/support/documentation-navigation/silicon-devices/mature-products/virtex-5.html`.

[26] **Linux**, Linux Ubuntu 16.04 LTS,, https://releases.ubuntu.com/16.04/.

[27] **Xilinx**, Vivado Design Suite,, https://www.xilinx.com/support/download/index.html/content/xilinx/en/downloadNav/vivado-design-tools/2019-2.html.

[28] **RISC-V**, RISC-V Gnu Toolchain,, https://github.com/riscv/riscv-gnu-toolchain.

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

[30] **Swain, A.K. and Mahapatra, K.** (2010). Design and verification of WISHBONE bus interface for System-on-Chip integration, *2010 Annual IEEE India Conference (INDICON)*, pp.1–4.

[31] **Pulp-Platform**, (2020), An open-source microcontroller system based on RISC-V, https://github.com/pulp-platform/pulpino.

[32] **Choi, W. and Choi, T.S.** (2008). Fast three-dimensional shape recovery in TFT-LCD manufacturing.

[33] **Sharma, A.**, **Ansari, M.D. and Kumar, R.** (2017). A comparative study of edge detectors in digital image processing, *2017 4th International Conference on Signal Processing, Computing and Control (ISPCC)*, pp.246–250.

[34] **Gonzalez, C.I.**, **Melin, P.**, **Castro, J.R.**, **Mendoza, O. and Castillo, O.** (2014). An improved sobel edge detection method based on generalized type-2 fuzzy logic, *Soft Computing*, *20*(2), 773–784, https://doi.org/10.1007/s00500-014-1541-0.

[35] **Suwanmanee, S.**, **Chatpun, S. and Cabrales, P.** (2013). Comparison of video image edge detection operators on red blood cells in microvasculature, *The 6th 2013 Biomedical Engineering International Conference*, pp.1–4.

[36] **Wikipedia**, (2021), Lenna Picture, https://en.wikipedia.org/wiki/Lenna.

APPENDICES

APPENDIX A : Ibex RTL Schematic with LED Outputs
APPENDIX B : Laplacian Filter RTL Schematic
APPENDIX C : RTL Schematic of Ibex core Including Wishbone Protocol and IPs
APPENDIX D : Wishbone Compatible GPIO Module
APPENDIX E : Wishbone Compatible OV7670 Module
APPENDIX F : Wishbone Compatible VGA Module
APPENDIX G : Images to Text and Vise Versa Conversion

APPENDIX A

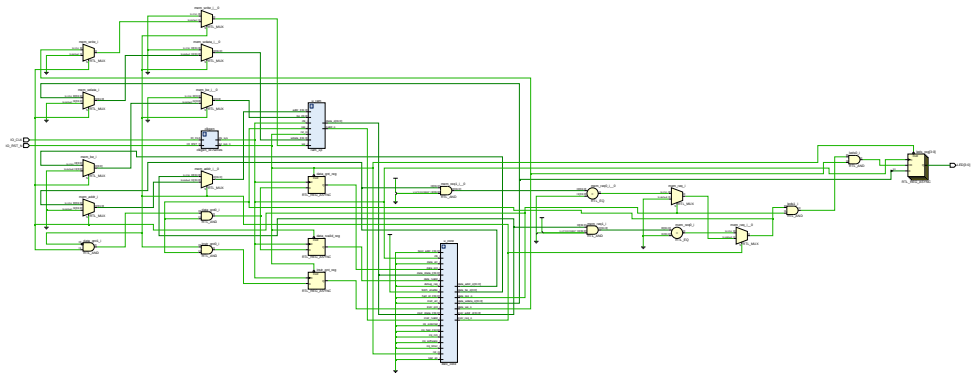


Figure A.1 : Ibex RTL Schematic with LED Outputs

APPENDIX B

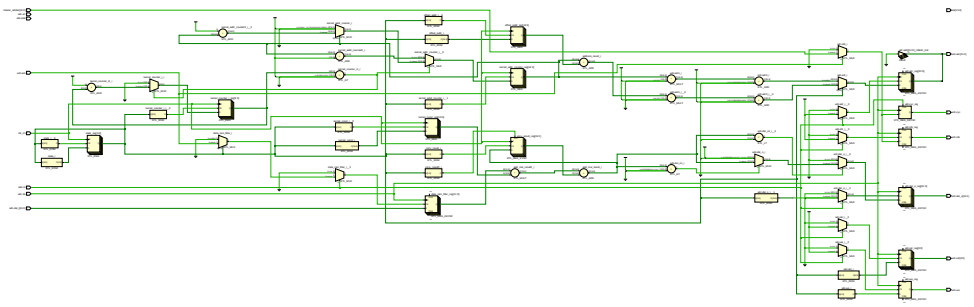


Figure B.1 : Laplacian Filter RTL Schematic

APPENDIX C

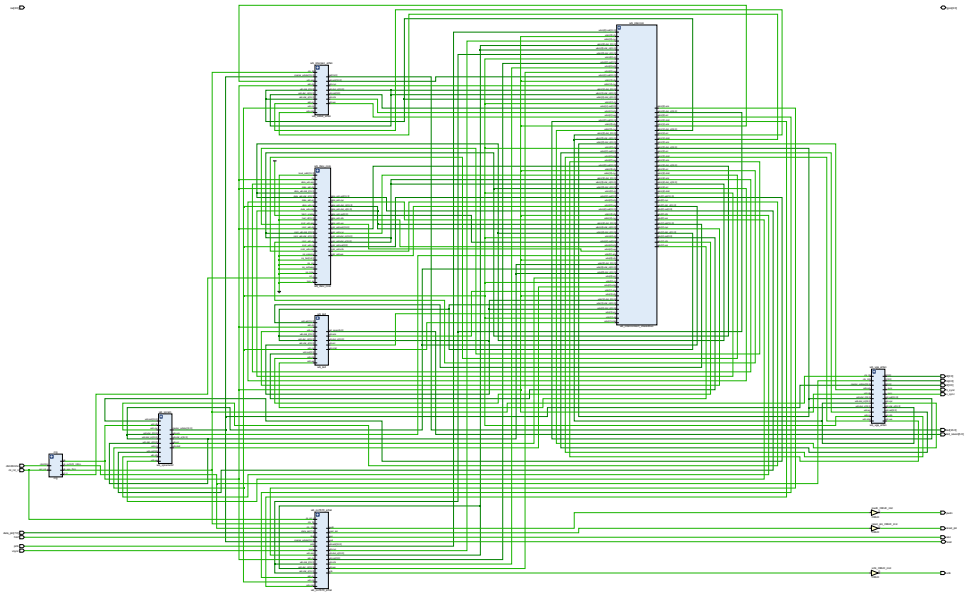


Figure C.1 : RTL Schematic of Ibex core Including Wishbone Protocol and IPs

APPENDIX D

```
module wb_gpio
  #(parameter width = 4)
  (inout  wire  [width-1:0] gpio,
    wb_if.slave              wb);

  logic [width-1:0 ] input_reg;
  logic [width-1:0 ] output_reg = 'b0;
  logic [width-1:0 ] direction_reg = 'b0;

  for (genvar i = 0; i< width; i = i+1) begin
     assign gpio[i] = direction_reg[i] ? output_reg[i] : 1'bz;
     assign input_reg[i] = direction_reg[i] ? 1'b0 : gpio[i];
  end

  logic valid;
  assign valid = wb.cyc && wb.stb;
  assign wb.stall = 1'b0;
  assign wb.err = 1'b0;

  always @(posedge wb.clk or posedge wb.rst)
     if (wb.rst) begin
      direction_reg <= '0;
      output_reg <= '0;
      end

     else if (valid)
      if (wb.we)
        case (wb.adr[3:0])
         4'h4 : output_reg    <= wb.dat_i [width-1:0];
         4'h8 : direction_reg <= wb.dat_i [width-1:0];
         default : ;
        endcase
     else
      case  (wb.adr[3:0])
        4'h0 : wb.dat_o     <= input_reg;
        4'h4 : wb.dat_o     <= output_reg;
        4'h8 : wb.dat_o     <= direction_reg;
        default : ;
      endcase

  always_ff @(posedge wb.clk or posedge wb.rst)
     if (wb.rst)
      wb.ack <= 1'b0;
     else
      wb.ack <= valid & ~ wb.stall;
```

```
endmodule

}
```

APPENDIX E

```
 ov7670_init u_ov7670_init
     (
         .iCLK(clk_100),         //100MHz
         .iRST_N(rst_ov),        //Global Reset
         .I2C_SCLK(sioc),     //I2C CLOCK
         .I2C_SDAT(siod)     //I2C DATA
     );


    ov7670_capture u_ov7670_capture(
         .data_send(1'b1),
         .pclk(pclk),
         .vsync(vsync),
         .href(href),
         .d(data_pin),
         .addr(capture_addr),
         .dout(capture_data),
         .wr(wr)
          );


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

 else
  begin

    if ( master\_arbiter[0]=='h1  )
     begin
       wb.cyc   <= 1'b1;
       wb.stb   <= 1'b1;
     end

     else
     begin
       wb.cyc   <= 1'b0;
       wb.stb   <= 1'b0;
     end
       wb.dat_o  <= capture_data[15:11]+capture_data[10:5]
       +capture_data[4:0];
       wb.adr   <= (capture_addr * 4) + 'hC014;
       wb.we    <= 1'b1;
```

```
      wb.sel   <= 4'b1111;

      if (wb.ack)
        begin
         wb.stb <= 1'b0;
         wb.cyc <=  1'b0;
       end
    end
end
```

APPENDIX F

```
  vga u_vga (
       .clk25(clk_25),
       .vga_red(R),
       .vga_green(G),
       .vga_blue(B),
       .vga_hsync(h_sync),
       .vga_vsync(v_sync),
       .frame_addr(frame_addr),
       .frame_pixel(frame_pixel_ram)
        );

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

else
 begin

   if ( master\_arbiter[16]=='h1)
    begin
      wb.cyc   <= 1'b1;
      wb.stb   <= 1'b1;
    end

   else
   begin
      frame_pixel_ram <= 12'h000;
      wb.cyc   <= 1'b0;
      wb.stb   <= 1'b0;
    end

      wb.adr   <=  (frame_addr*4) + 'hC014;
      wb.we    <= 1'b0;
      wb.sel   <= 4'b1111;

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

  end
```

```
end
```

APPENDIX G

```
import cv2
import csv
import numpy as np
import time

picture_data_counter = 0

img = cv2.imread('lenna.png', cv2.IMREAD_GRAYSCALE)
with open("data_init.txt", "w") as text_file:
    for j in range(240):
      for i in range(320):
         print(img[j,i], file=text_file)



a_file = open("data_from_verilog.txt", "r")

string_without_line_breaks = ""

for line in a_file:

  stripped_line = line.rstrip()
  string_without_line_breaks += (stripped_line) +'\n'
  picture_data_counter = picture_data_counter + 1
  if picture_data_counter == 76800:
     break

a_file.close()

result = np.fromstring(string_without_line_breaks,
dtype=int, sep='\n')

result_reshaped_to_2d = np.reshape(result, (-1,320))

print(type(string_without_line_breaks))
print(type(result))
print(len(result))

print(result.shape)
print(result_reshaped_to_2d.shape)
```

```
cv2.imwrite('output.png',result_reshaped_to_2d)
```

CURRICULUM VITAE

Name Surname: Erfan Gholizadehazari
Place and Date of Birth: Urmia, Iran, 05/08/1995
E-Mail: gholizadehazari18@itu.edu.tr

EDUCATION:

- **B.Sc.:** 2017, Urmia University, Electronics Engineering

HONORS AND REWARDS:

- Ranked 1st among all Electronics Engineering department graduates in B.Sc.
- Ranked 2nd in Tet Proje Pazari Competition, Smart Cities-Smart Projects, Istanbul, Turkey.
- Certificate of skill from Technical and Vocational Training Organization TVTO

PROFESSIONAL EXPERIENCES:

- 2015-2018 Kara Electronics Co. - Embedded Systems Design Engineer
- 2020-2021 Istanbul Technical University - Research Assistant

PUBLICATIONS ON THE THESIS:

- Erfan Gholizadehazari, Tuba Ayhan, and Berna Ors. "An FPGA Implementation of a RISC-V Based SoC System for Image Processing Applications" The 29th IEEE Conference on Signal Processing and Communication Application (2021).