ISTANBUL TECHNICAL UNIVERSITY
ELECTRICAL-ELECTRONICS FACULTY

APPLICATION-SPECIFIC EXTENSION OF THE INSTRUCTION SET OF RISCV PROCESSOR

SENIOR DESIGN PROJECT

Adem EREN
Mehmet Emre YAĞAR

ELECTRONICS AND COMMUNICATION ENGINEERING DEPARTMENT

JULY, 2020

ISTANBUL TECHNICAL UNIVERSITY
ELECTRICAL-ELECTRONICS FACULTY

APPLICATION-SPECIFIC EXTENSION OF THE INSTRUCTION SET OF RISCV PROCESSOR

SENIOR DESIGN PROJECT

Adem EREN
(040170719)

Mehmet Emre YAĞAR
(040170711)

ELECTRONICS AND COMMUNICATION ENGINEERING DEPARTMENT

Project Advisor: Assoc. Prof. Dr. Sıddıka Berna ÖRS YALÇIN

JULY, 2020

We are submitting the Senior Design Project Report entitled as "**APPLICATION-SPECIFIC EXTENSION OF THE INSTRUCTION SET OF RISCV PROCESSOR**". The Senior Design Project Report has been prepared as to fulfill the relevant regulations of the Electronics and Communication Engineering Department of Istanbul Technical University. We hereby confirm that we have realized all stages of the Senior Design Project work by ourselves and we have abided by the ethical rules with respect to academic and professional integrity .

Adem EREN
(040170719)

Mehmet Emre YAĞAR
(040170711)

Project Advisor: **Assoc. Prof. Dr. Sıddıka Berna ÖRS YALÇIN**

FOREWORD

Thanks to our assistant mentor Assoc. Prof. Dr. Sıddıka Berna Örs Yalçın who helped us find this project and assisted us in all our failures, to achieve total success and at least do our best to sustain successfully our project. We are also grateful to Latif Akçay, who has opened our way with the knowledge he has in the field of riscv architecture. We would like to thank our dear friend Mehmet Doğan, who always supported and helped us in this project, and also Ebru Nur Yağar, Gülsün Yağar, Hüseyin Yağar and Eren family who did not withhold their endless support from us.

July 2020

Adem Eren
Mehmet Emre Yağar

TABLE OF CONTENTS

Page

ABBREVIATIONS

ALU	**:** Arithmetic Logic Unit
CPU	**:** Central Processing Unit
FPGA	**:** Field Programmable Gate Array
FPU	**:** Floating Point Unit
IDE	**:** Integrated Development Environment
ISA	**:** Instruction Set Architecture
JDK	**:** Java Development Kit
OS	**:** Operating System
PC	**:** Personal Computer
PK	**:** Proxy Kernal
RISC	**:** Reduced Instruction Set Computing
RTL	**:** Register Transfer Level
SoC	**:** System on Chip
VHDL	**:** Very High –Speed Integrated Circuit Hardware Description Language

LIST OF FIGURES

Page

APPLICATION-SPECIFIC EXTENSION OF THE COMMAND SET FOR OPEN SOURCE PROCESSORS

SUMMARY

Time is a phenomenon that should not be wasted for human life anymore. Therefore, the planned works must be completed quickly, regularly and accurately. Processors are called electronic devices, or their brains, that control the operation of computer units and the data flow between them, performing data processing tasks. Producing processors is an activity that few companies in the world can do, so there is no common processor structure. For this reason, these companies sell the processors and software they designed at a very high price. Open source processors, which is a term that we hear a lot today, and that we will even hear in the future, enable people to design their own processors and use them in their own machines. RISC-V is an open, free ISA that allows for a new era of processor innovation via the open standard collaboration. Born in academia and researchat the University of California, Berkeley, RISC V ISA provides architecture with a new level of free, extensible software and hardware freedom, opening the way for computing design and innovation in the next 50 years.

Speed is a very important concept for processors, and with the development of technology, efforts to increase performance are increasing. In this project, Ariane core, which is one of the RISC-V processors, will be implemented and an extension on ALU is aimed to increase performance of the core. For the implementation of this project, Ariane core works on Linux OS and to complete implementation modular multiplication algorithm is used. For the extension planned to be made on ALU, the most appropriate extension will be made on the ALU by first determining the appropriate instructions selected in our algorithm, writing with system verilog encoding, and finally a bit file will be produced to control the increase in performance.

In conclusion, the outcome of this project also give results how to make implementation on the processor and making extension of ALU . Also it is clear that, there is not so much studies on this area and no developed processors done in the Turkey so these open source processors are a leading process and advantageous to follow the technological deveelopments in this area.We can clearly say that working with these open source processors will give us time, resources, work, expense, and prototyping advantages.

AÇIK KAYNAKLI İŞLEMCİLERDE KOMUT SETİNİN UYGULAMAYA ÖZEL GENİŞLETİLMESİ

ÖZET

Zaman artık insan hayatı için boşa harcanmaması gereken bir olgudur. Bu nedenle planlanan çalışmalar hızlı, düzenli ve doğru bir şekilde tamamlanmalıdır. İşlemcilere, bilgisayar birimlerinin çalışmasını ve aralarındaki veri akışını kontrol ederek veri işleme görevleri gerçekleştiren elektronik cihazlar veya beyinleri denir. İşlemci üretmek, dünyadaki birkaç şirketin yapabileceği bir faaliyettir, bu nedenle ortak işlemci yapısı yoktur. Bu nedenle, bu şirketler tasarladıkları işlemcileri ve yazılımları çok yüksek bir fiyata satmaktadırlar. Bugün çok duyduğumuz ve gelecekte duyacağımız bir terim olan açık kaynaklı işlemciler, insanların kendi işlemcilerini tasarlamalarını ve kendi makinelerinde kullanmalarını sağlıyor. RISC-V, açık standart işbirliği yoluyla yeni bir işlemci yeniliği çağına izin veren açık ve ücretsiz bir ISA'dır. Akademi doğumlu ve California Üniversitesi, Berkeley'de yeniden araştırma yapan RISC V ISA, mimariye ücretsiz, genişletilebilir yazılım ve donanım özgürlüğü sunarak önümüzdeki 50 yıl içinde tasarım ve yeniliği hesaplamanın yolunu açıyor.

Hız, işlemciler için çok önemli bir kavramdır ve teknolojinin gelişmesiyle performansı artırma çabaları artmaktadır. Bu projede, RISC-V işlemcilerinden biri olan Ariane çekirdeği uygulanacak ve çekirdeğin performansını arttırmak için ALU üzerinde bir genişletme yapılması hedefleniyor. Bu projenin uygulanması için, Linux işletim sisteminde “Batch mode” ile Ariane çekirdeği oluşturulmuş ve uygulamanın tamamlanması için modüler çarpma algoritması kullanılmıştır. ALU'da yapılması planlanan genişletme için, öncelikle algoritmamızda seçilen uygun komutlar belirlenerek, sistem verilog kodlaması ile yazılarak, en uygun genişletme ALU'da yapılacak ve son olarak, artışı kontrol etmek için bir bit dosyası üretilecektir.

Sonuç olarak, bu projenin çıktısı hem işlemci üzerinde nasıl uygulama yapılacağı hem de ALU'nun nasıl genişletileceği ile ilgili sonuçlar vermektedir. Ayrıca, bu alanda çok fazla çalışma yapılmadığı ve Türkiye'de gelişmiş bir işlemcinin yapılmadığı açıktır, bu nedenle bu açık kaynak işlemcileri bu alandaki teknolojik gelişmeleri takip etmek için öncü bir süreçtir ve avantajlıdır. bu açık kaynaklı işlemciler bize zaman, kaynak, iş, gider ve prototip avantajları sağlayacaktır.

1. INTRODUCTION

The aim of this graduation project is the reducing process time for specific application by using the method which is the extension of arithmetic logic unit. There is two different processor architecture in the literature: Complex Instructşon Set Computer(CISC) and Reduced Instruction Set Computer(RISC). In this project Ariane chip was designed from RISC architecture which was choosen because of open source availability. Later, ALU extension will be implemented and performance comparison tests will be done in simulation waveforms by using GTK waveform simulater. For the test program, modular multiplication algorithm will be implemented on Ariane chip. In this section, general information has been explained about CISC and RISC architecture and then the Ariane chip will be examined in detail.

1.1 General Information And Concepts

In this section, firstly, a description will be made about the processor architectures and then detailed information will be given about the software and tools required for the realization of the project. Each section will include a comprehensive overview of all the tools and software used in the project. What each is and what the purpose serves in the project will be specified ..

1.1.1 CISC architecture

A complex instruction set computer is a computer where single instructions can perform various low-level operations such as memory loading, arithmetic operation, and memory store, or are performed in single instructions via multi-step processes or mode addressing, as its name suggests "Complex Instruction Set."

Based on the description of program compilers, the CISC machines have good actions; as the variety of creative instructions can be obtained in one set of instructions. They conceive compound instructions in a simple set of instructions. They achieve low-level processes, which makes it easier to have large address nodes and additional types of data in a machine's hardware.

1.1.2 RISC Architecture

In the RISC architecture, the set of instructions are reduced and each instruction here is expected to achieve very small function. Because of the instruction sets in this

architacture are plain and simple, complex codes can be implemented easily. Each instruction is about the same length; these are wound together in a single operation to get compound tasks done.

1.1.2.1 RISC-IMAC Instruction Set

RISC-V was originally developed to support research and education in computer architecture, but we hope now it is also a standard free and open architecture for the industry.

IMAC stands for integer, multipilication, atomic and compressed instruction set. Ariane ISA is consist of the combination of these instruction sets.[1]

RV64I Base Instruction Set (in addition to RV32I)

imm[11:0]		rs1	110	rd	0000011	LWU
imm[11:0]		rs1	011	rd	0000011	LD
imm[11:5]	rs2	rs1	011	imm[4:0]	0100011	SD
000000	shamt	rs1	001	rd	0010011	SLLI
000000	shamt	rs1	101	rd	0010011	SRLI
010000	shamt	rs1	101	rd	0010011	SRAI
imm[11:0]		rs1	000	rd	0011011	ADDIW
0000000	shamt	rs1	001	rd	0011011	SLLIW
0000000	shamt	rs1	101	rd	0011011	SRLIW
0100000	shamt	rs1	101	rd	0011011	SRAIW
0000000	rs2	rs1	000	rd	0111011	ADDW
0100000	rs2	rs1	000	rd	0111011	SUBW
0000000	rs2	rs1	001	rd	0111011	SLLW
0000000	rs2	rs1	101	rd	0111011	SRLW
0100000	rs2	rs1	101	rd	0111011	SRAW

RV64M Standard Extension (in addition to RV32M)

0000001	rs2	rs1	000	rd	0111011	MULW
0000001	rs2	rs1	100	rd	0111011	DIVW
0000001	rs2	rs1	101	rd	0111011	DIVUW
0000001	rs2	rs1	110	rd	0111011	REMW
0000001	rs2	rs1	111	rd	0111011	REMUW

RV64A Standard Extension (in addition to RV32A)

00010	aq	rl	00000	rs1	011	rd	0101111	LR.D
00011	aq	rl	rs2	rs1	011	rd	0101111	SC.D
00001	aq	rl	rs2	rs1	011	rd	0101111	AMOSWAP.D
00000	aq	rl	rs2	rs1	011	rd	0101111	AMOADD.D
00100	aq	rl	rs2	rs1	011	rd	0101111	AMOXOR.D
01100	aq	rl	rs2	rs1	011	rd	0101111	AMOAND.D
01000	aq	rl	rs2	rs1	011	rd	0101111	AMOOR.D
10000	aq	rl	rs2	rs1	011	rd	0101111	AMOMIN.D
10100	aq	rl	rs2	rs1	011	rd	0101111	AMOMAX.D
11000	aq	rl	rs2	rs1	011	rd	0101111	AMOMINU.D
11100	aq	rl	rs2	rs1	011	rd	0101111	AMOMAXU.D

Figure 1.1 : RISCV instruction sets

1.1.2.2 RISC-V assembler and compiler toolchain

In the project, we are evaluating the RISC-V open source instruction set architecture and ariane core with our own implementation. We need a software, as named RISC-V GNU Toolchain, to compile, assemble and link our source files in order to execute in both a simulator and FPGA. Because we are using Ariane core, our GCC flow is a little bit different from that figure with some additional terminal commands. In the project, we will be using the Verilator with the riscv-fesvr for simulation results.

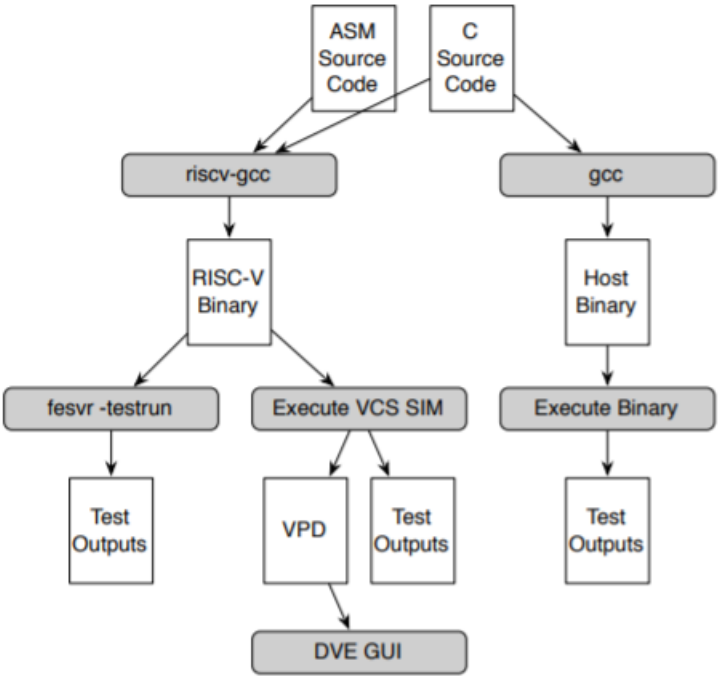


Figure 1.2: RISC-V assembler and compiler toolchain

1.1.2.3 Ariane core view

PULP is a silicon-proven Parallel Ultra Low Power platform that has an aim to get high energy qualification. The platform is established as clusters of RISC-V cores that share a tightly-coupled data memory. This platform cooperate with ETH Zürich (Integrated Systems Laboratory) and University of Bologna (Energy-effecient

Embedded Systems) to find out modern and effecient cores for high energy effeciency and low power processing.[2] Ariane core;

- 6-Stage, single issue, out-of-order write-back, in-order-commit
- 64-bit RISC-V instruction set with in-order CPU
- It implements I(Integer instructions), M(Multiplication and Division instructions), A(Atomic instructions) and C(Compressed Instructions).
- Also implemets three privilege levels M, S, U to fully support a Unix-like operating system.
- Configurable size
- Seperate TLBs
- Hardware PTW(Page Table Walker)
- Branch-Prediction(branch target and branch history table)
- The first important aim is to reduce the critical path length

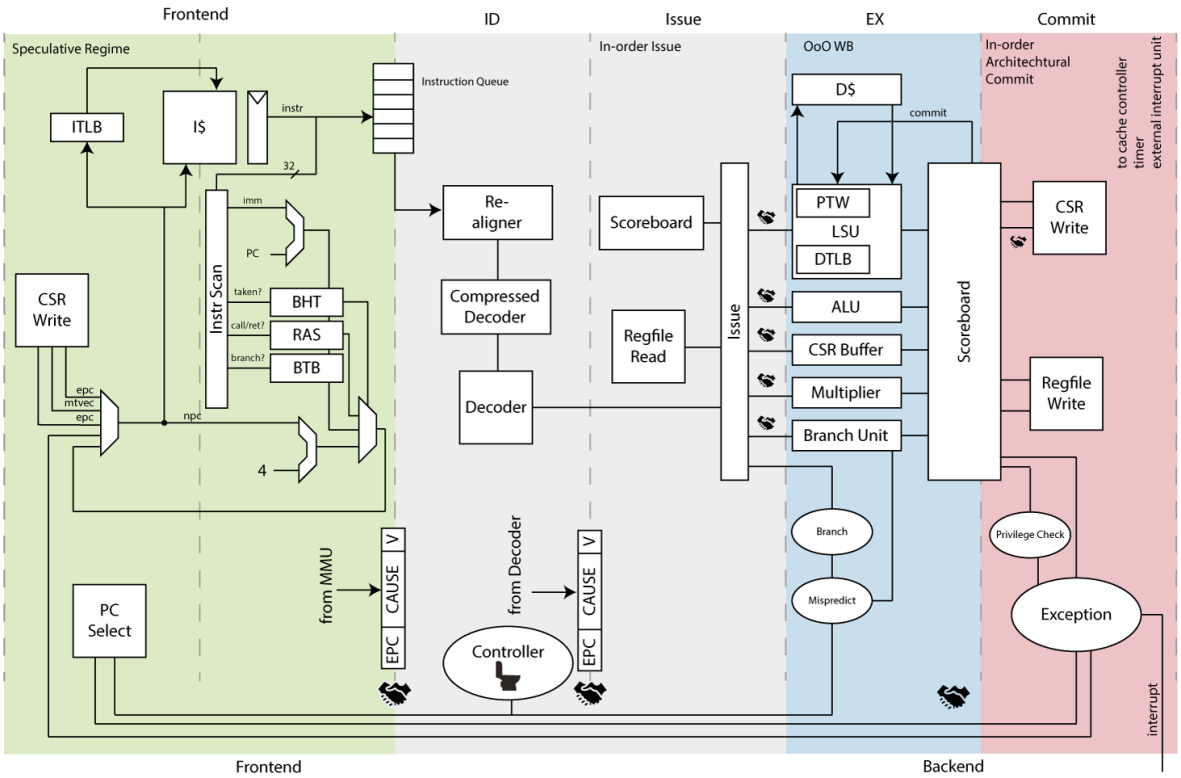


Figure 1.3 Ariane core overview

As the Ariane core architecture, we would like to give some informations in the “Issue” part. The purpose of the issue stage is to receive the decoded instructions and issue them to the different functional units. The issue stage also keeps track of all the instructions issued, the functional unit status and collects the write-back data from the execution stage and additionally, it includes the register file for the CPU. By using the data structure, known as scoreboard, it exactly knows the instructions are issued, the functional unit they are in and the register they are going to write back to. We can splits the execution process in the four parts; issue, read operands, write-back and execute. Except the second part(read operands), issue handles the other parts.

Actually, we are going deeper of the structure with getting information on execute stage and then ALU(Arithmetic Logic Unit).

The execution stage is a logical process encapsulating each of the functional units (FUs). Each FU must be capable of its operation separately of any other unit. Also every functional unit retains a correct signal to give the valid signal output data, and a ready signal to inform the issue logic whether or not it can accept a new request. Execute stage contains CSR buffer and divide-multiply unit and ALU, branch unit and load store unit(LSU).

The most important part of the core in this project is the ALU structure. The arithmetic logic unit (ALU) is a specific piece of hardware that implements addition, subtraction, comparisons, and shifts of 32 and 64-bit. It always completes its process in one cycle and therefore includes no full-state entities. It also receives an operator, together with the two operands, that informs it which operation to perform.[2]

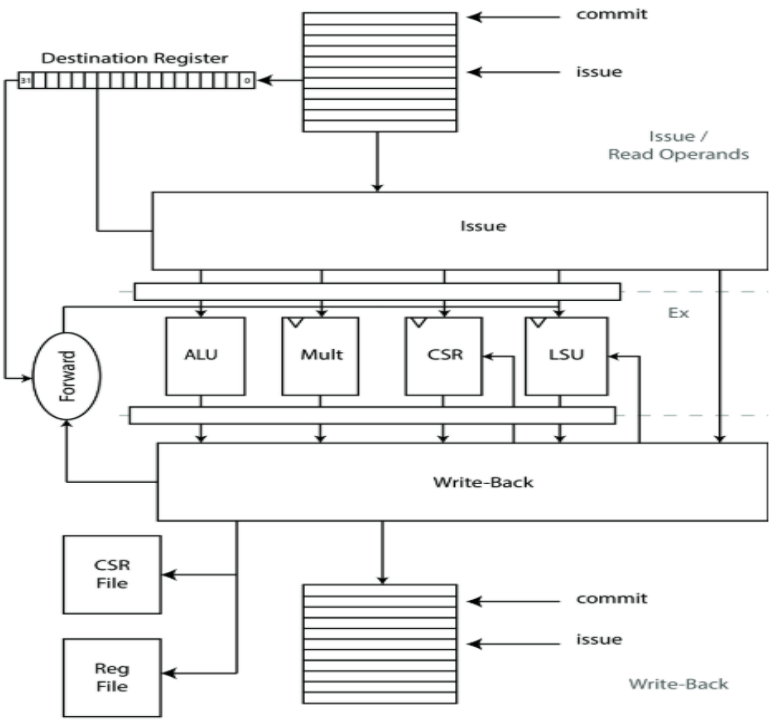


Figure 1.4 : Issue and execute stage of ariane

1.1.3 Modular multiplication algorithm

If only certain people are required to access information, we need to encrypt it. Many algorithms have been developed due to security concerns in the encryption area. Various encryption operations can be performed using the Modular Multiplication algorithm. The modification to be made in Ariane Core's ALU will be made to make this algorithm work faster. There are some explanations about this algortihm according to the following code.

Step1: INPUT → X,Y,M

Step2: OUTPUT→ RESULT

Step3: RESULT = X*Y

Step4: M << N

Step5: for i=N down to 0 do

if (RESULT >= M)

RESULT = RESULT - M

M = M >> 1

endfor

Step6: return RESULT

The modular multiplication algorithm, the general scheme of which is given, can be written in different high-level languages. The algorithm can also be updated to avoid overflow. In this graduation project, a “C” code was written to test the Ariane core. Additional commands have been added to prevent overflow in this c code.

This algorithm can be written in C language as in the Figure 1.8.

```
int main(int a, int b, int mod)
{
    int res = 0; // result
    a = a % mod;
    while (b > 0)
    {
        // If b is odd, add 'a' to result
        if (b % 2 == 1)
            res = (res + a) % mod;

        // Multiply 'a' with 2
        a = (a * 2) % mod;

        // Divide b by 2
        b /= 2;
    }

    // Return result
    return res % mod;
}
```

Figure 1.5 : The c code for the test of ariane RISCV core.

Riscv compiler converted this c code into machine code. Then the "verilator" program was used to test in Ariane core. The resulting test file was simulated with the "spike"

program. The result based on the entered values was calculated as expected. Each of these phases will be described in detail in the implementation section.

After testing core processes, the waveform of signals in Ariane core will be shown by using verilator and GTK wave simulation programs. For this purpose the assembly code is more advantageous to use because of speed. Therefore the assembly codes of this algorithm is needed to written. After this operation, The simulation tests could be observed in GTK wave.

1.1.4 General flow of project

In this section, the steps of the project will be discussed in detail. First of all the operational system which in the implementation of Ariane core is decided to Windows because of the simplicity. But later, there is a change of operational system with Linux. Because the creator of Ariane core has designed in "batch mode". The meaning of that we need to use terminal in Linux so we need an internal or external disk to install the Ubuntu 18.04 operating system. Ariane core can be implemented only for Genesys 2 board. Also in order to get the Genesys 2 board files, you need to get the fully licenced version of the Vivado not the Webpack version but because of the lack of the Genesys 2 board in the laboratory, we will work on the project in the simulation level. The simulation tool that is used in that project is GTK Wave simulator associated with Verilator tool. How they are installed and worked will be explained in the next chapter. Also, in order to compile the core we will need the some software programs. After all needed installation tools are downloaded, we write the C code of the Modular Multiplication algorithm. With converting the C code into assembly code, we will edit the bootrom of the Ariane core. Then, we can control the signal on the GTK Wave simulation whether it simulate correctly or not. The final process is to check the algorithm that which part is more repeated. Because the most repeated part of the algorithm occupies the processors most. Then we will edit the ALU of the Ariane core with adding some logical operands and make the performance measurement.

1.2 Literature Review

In this section, the researches in literature that are relevent with the RISC-V core and extension of the command set will be explained in detail.

1.2.1 Literature review at Istanbul Technical University

- **(Güngör, Öndeş, Sarı, Uçkun; 2018)** ***"Instruction Set Extension Of Some Processors For Secure Iot Implementations"***

The next generation Internet of Things (IoT) technologies are primarily concerned with security. Therefore the cryptology was chosen as the application area in the research. The key theme of this study is the use of AES(Advanced Encryption Standard) and present cryptology algorithms in open source processors, with instartcion set extension (ISE) and to improve protection for IoT applications for these processors. The Pulpino RI5CY, part of the ETH Zurich, is at the center of the project. The one of developer in this project, implement this core on Xilinx's Nexys4 DDR Development Board and then found it worked through checking. After that point, the main theme of the AES thesis application was successfully implemented using the instruction set extention method.[3]

- **(Sürer, Bartu; 2019)** ***"Implementatıon Of A Soc By Usıng Lowrısc Processor On An Fpga For Image Fılterıng Applıcatıons"***

The purpose of this project is using the Image Processing Algorithm with instruction set extension in open source processors and make some filters such as, Gauss blur filter, sharpening filter which is applied to images with pure C code without using libraries. The project manager is Bartu Sürer decided to make research on the lowRISC core and used Nexys DDR4 FPGA board which is produced by Xilinx and then made observations that whether the Image Filtering is working correctly or not. At the end of the project, the result of the implementation is that, lowRISCV system on chip could be used on FPGA board as well as prototyping and testing on FPGA board correctly. [4]

1.2.2 Literature review at Turkey

No thesis research on this subject in our country has been done so far. But a member of the RISC-V group, listed as silver, is one of the largest defense industry firm in Turkey ASELSAN.

1.2.3 Global Literature review

- **(F. Zaruba and L. Benini; 2019)** ***"The Cost of Application-Class Processing: Energy and Performance Analysis of a Linux-Ready 1.7-GHz 64-Bit RISC-V Core in 22-nm FDSOI Technology"***

This research mentions that promoting application-class execution involves an unavoidable reduction in energy-efficiency and more computational performance is cost-effectively improved, instead of high-frequency operations, with instruction extensions. They made efficiency analysis about power consumption based on the Ariane core on a functional unit level. In addition to this there is a comparison with other RISC-V 64-Bit cores . The data of this comparison about speed, required area, power consumption, IPC, energy per operation is available in this research. In the light of such information Ariane has a big advantageous in energy efficiency and performance in own class. The performance tests can be examined by using Verilator and QuestaSim simulation or after the generating bitstream on the FPGA[5].

	Ariane	**Rocket [14]**	**Boom 2-w [27]**	**Shakti [28]**
Bits	64	32/64	64	64
User Spec	IMC	IMAFDC	IMAFD	IMAFD
Priv. Spec	1.11	1.11	1.11	1.10
Tech	GF 22 nm	TSMC 45 nm	TSMC 45 nm	22 nm
Speed	1.7 GHz	1.6 GHz [29]	1.5 GHz [27]	800 MHz
Area	$0.3\,mm^2$	$0.5\,mm^2$	$1.7\,mm^2$ [27]	$0.29\,mm^{2*}$
Power	52 mW	125 mW [30]	300 mW [30]	90 mW
IPC	0.87^a	$0.95^‡$	$1.45^‡$	0.9^1
Energy/Op	52 pJ	100 pJ [30]	133 pJ [30]	122 pJ

Figure 1.6 : RISC-V core comparisons[5]

2. IMPLEMENTING ARIANE PROCESSOR

2.1 Gathering Required Environment

In this section all software and tools requirements will be explained in particularly.

2.1.1 Ubuntu operating system

Firstly, the operationg system of “Ubuntu 18.04” should be installed on the external or internal disk and there should be at least 250GB memory space in disk. Internal disk is preferred because of making process takes less time.

Also, Ubuntu should be installed in English version. Installation of Ubuntu in Turkish bring problems because of Turkish characters.

Firstly we will write a few commands on terminal as the following way.

$sudo apt-get update

The command "sudo apt-get update" is used to retrieve information about the packages from all configured sources.

$ *sudo apt-get upgrade*

$ *sudo apt install build-essential*

To install available updates of all currently installed packages on the device from sources configured via sources.list file, you run "sudo apt-get upgrade."

Also in order to get the sources file from github we need to install “git” with following command

$ *sudo apt-get install git*

Because we do not use FPGA board, we do not need Vivado program exactly because we will use GTKWave as a simulator. But in order to get bit file from bitstream we need Xilinx Vivado 2018.2 version and it should be installed with licenced version because Genesys 2 board files only could be seen in licenced version. After installation, we should write a command in “.bashrc” file in order to open vivado with command in terminal.

$source /opt/Xilinx/Vivado/2018.1/settings64.*sh*

$vivado

2.1.2 RISC-V GNU toolchain

GNU Compilar Toolchain is the RISC-V C++ and C cross-compiler. It has two modes: Newlib/Elf toolchain and Linux-Elf/glibc toolchain but we will use the first one, Newlib toolchain because we do not perform on Linux, we will use machine codes directly. The fact that we work on Linux operating system for implementation of required software and tools. However Ariane core would not be implemented with a operating system, as a pure structure.

We will obtain its sources or repository with writing that command in terminal.[6]

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

This will download the repository as “riscv-gnu-toolchain” at home directory.

Also in order to build the toolchain, we need to download some additional packages. With writing this command,[7]

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

Now, we should enter the toolchain directory,

$ *cd riscv-gnu-toolchain*

In order to configure the Newlib cross-compiler, we will pick an installation path then build it in that directory.

$*./configure --prefix=/opt/toolchain --with-arch=rv64imac --with-abi=lp64*

Then finally writing that command will install the toolchain in the “/opt/toolchain” path.

$ *make*

2.1.3 RISC-V tools

RISC-V tools contains a set of RISC-V simulators, compiler and other tools.

Riscv-fesvr: RISC-V Frontend Server

Riscv-isa-sim: ISA simulator(Spike)

Riscv-qemu: Higher-performance ISA simulator

Riscv-pk: RISC-V Proxy Kernel

We can download the RISC-V tools repository with these commands[8]:

$ *git clone https://github.com/riscv/riscv-tools*

$ *cd riscv-tools*

$ *git submodule update --init –recursive*

$ *export RISCV=/opt/riscv/riscv-tools*

$ *export PATH=${PATH}:${RISCV}/bin*

$ *./build.sh*

Also in Ubuntu system, some packages are needed and with writing a command as:

$ *sudo apt-get install autoconf automake autotools-dev curl libmpc-dev libmpfr-dev libgmp-dev libusb-1.0-0-dev gawk build-essential bison flex texinfo gperf libtool patchutils bc zlib1g-dev device-tree-compiler pkg-config libexpat-dev*

There is an important requirement that riscv-tools needs a compiler with GCC>=4.8 so with writing this command we can handle it:

$ *CC=gcc-5 CXX=g++-5 ./build.sh*

2.1.4 Verilator

Verilator is a free open source software tool that translates Verilog to C++ or SystemC into a cycle-accurate behavioral model. In the sense of general design flows ASIC and FPGA and for performance and power analysis, researchers use Verilator to create new co-simulation environments. Verilator is also a common tool for student dissertations. Not only does Verilog Convert HDL to C++ or SystemC. Alternatively, Verilator compiles the code into a much faster-optimized thread-divided model, packaged inside a C++/SystemC / C++-under-Python module. Verilator is called up with parameters similar to GCC(GNU Compiler Colection). Verilator reads the Verilog or SystemVerilog code, performs lint checks, and optionally puts assertion checks and covers. The code "Verilated" produces single or multi-threaded.cpp and.h files. The outcome is a compiled Verilog model, which also works 100 times faster on one thread than Verilog's interpreted simulators.

Verilator can be conveniently mounted by writing the following order to the terminal.[9]

$ *apt-get install verilator*

However, as certain applications and software needed for verilator are out of date it may not be possible to successfully install them. So installing the verilator via Git would be more useful. In our work, we loaded by the following way[1].

First of all requirements are needed to install as the following three lines. They may be give an error but this is not a serious issue, please continue to type codes.

$*sudo apt-get install git make autoconf g++ flex bison*

$*sudo apt-get install libfl2*

$*sudo apt-get install libfl-dev*

Secondly the repository of verilator is needed to download in it's own source.

$git clone https://git.veripool.org/git/verilator # Only first time

Every time you need to type the following two lines when the installation of verilator for C shell and bash. If there is taken an error, do not get any trouble and just pass this commands.

$*unsetenv VERILATOR_ROOT*

$*unset VERILATOR_ROOT*

Now enter the directory of verilator.

$*cd verilator*

Ensure that git is an updated repository

$git pull # Make sure git repository is up-to-date

$git tag # See what versions exist

$git checkout master # Use development branch (e.g. recent bug fixes)

$git checkout stable # Use most recent stable release

$git checkout v{version} # Switch to specified release version

$autoconf # Create ./configure script

$./configure

$make

$sudo make install

2.1.5 GTKWave simulator

GTKWave is an analysis tool used to test simulation models on Verilog or VHDL. This is not intended to run interactively with simulation except for interactive VCD viewing, but rather relies on a postmortem approach through the use of dumpfiles. It supports various dumpfile formats but we will use vcd format file in that project.

GTKWave was designed to perform debug tasks on a chip on large systems and was used as an offline substitute for third party debug tools in this ability.

For Verilog, GTKWave allows users to debug simulation results at both the net level by providing a bird's eye view of multiple signal values over varying time periods and also at the RTL level by annotating the signal values back into the RTL for a given time step.

In order to compile and install the simulator, the following steps should be followed:

Un-tar the source code into any temprorary directory and enter the file and make configure.[10]

$ *./configure*

$ *make*

This will take some time depends on the PC and then

$ *su*

This will ask the password, with writing the password by user, write this command

$ *make install*

Wait for finishing installation then GTKWave will be installed.

2.2 Running User-Space Applications

The Ariane processor can be tested in two ways. The first can be achieved with Bootrom inside the processor. The second can be done directly with a high-level language with instruction memory.[2]

We will use Makefile in the repository of Ariane to build the Verilator model of Ariane and “work-ver” file will be created.

$ make verilate

Then we will build the Verilator model of core in order to obtain vcd format file and “work-ver” file will be debugged with this command;

$ make verilate DEBUG=1

2.2.1 Assembly level test

As seen in the simulation. Instruction set takes the instructions from bootrom in order so we can say that when we make some operations on core. Instruction set works correctly.

Also another check about core is program counter and we will control that when the instruction set gets the instructions program counter(pc) increases or not. As clearly seen program counter increases gradually on the top of the simulation signals.

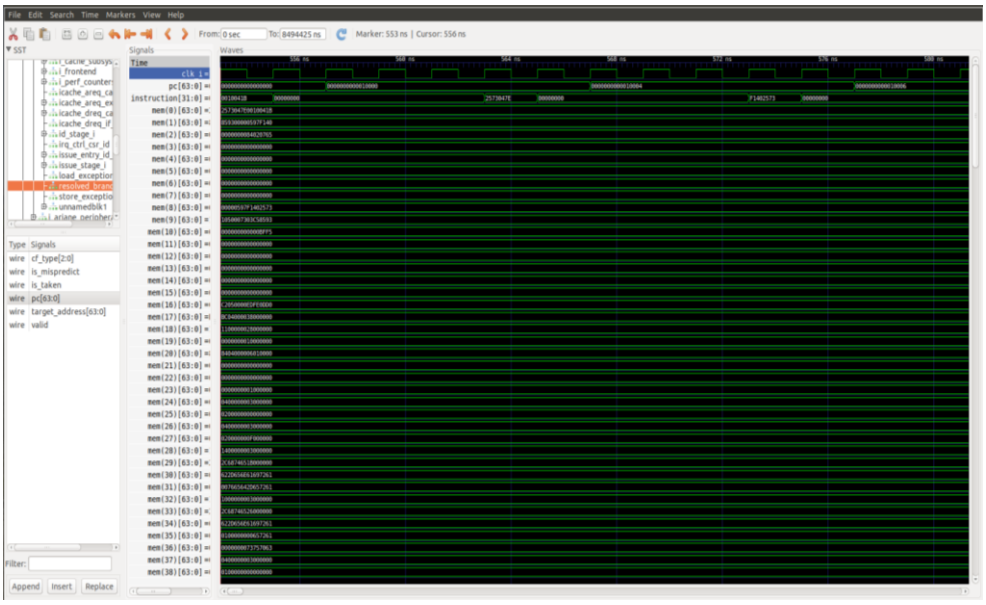


Figure 2.1: Bootrom addresses and their values

We have seen that instruction set gets the instructions from bootrom and now we will write a small addition operation of assembly code in bootrom file and will check

whether the Ariane core do this operation correctly. In the directory; /Ariane/boorom/bootrom.S

Here is the bootrom file content and we have added a set of commands as written in red.

```
#define DRAM_BASE 0x80000000
.section .text.start, "ax", @progbits
.globl _start
_start:
  lui s3 , 4
  lui s4 , 3
  add s5 , s3 , s4
  li s0, DRAM_BASE
  csrr a0, mhartid
  la a1, _dtb
  jr s0
.section .text.hang, "ax", @progbits
.globl _hang
_hang:
  csrr a0, mhartid
  la a1, _dtb
1:
  wfi
  j 1b
.section .rodata.dtb, "a", @progbits
.globl _dtb
.align 5, 0
_dtb:
.incbin "ariane.dtb"
```

Figure 2.2: The bootrom of ariane core

After writing assembly commands we will compile it and see simulation result and want to see the result of addition of 3+4=7.

In the directory of ariane/bootrom , write this command in terminal

$ make all

It will give the output as;

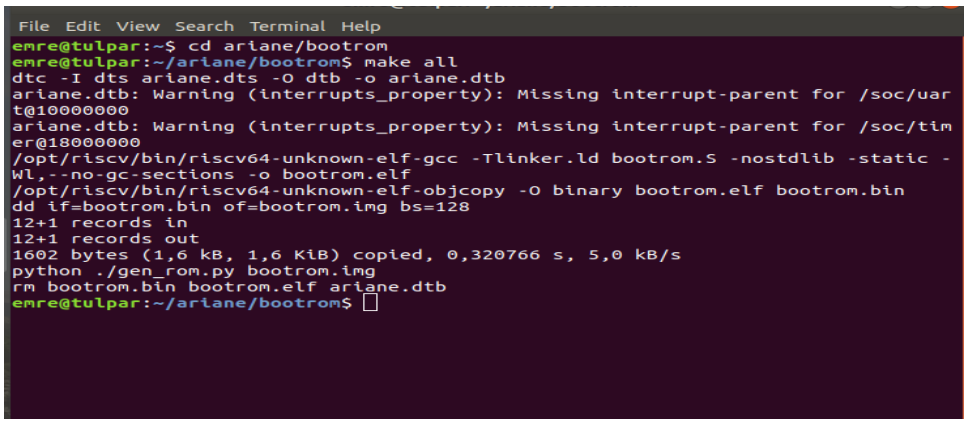


Figure 2.3: Linux terminal after modification of bootrom

Now it is time to create vcd format file to see the simulation result and we will write the same commands on terminal as done before.

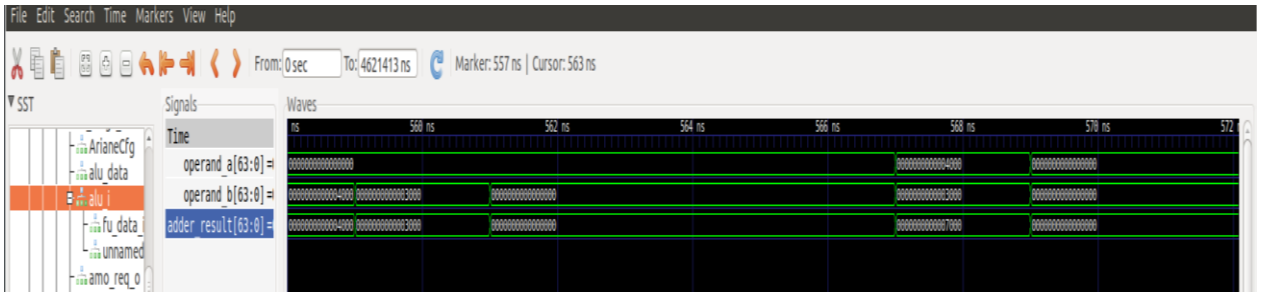


Figure 2.4: Simulation result of addition operation

Here is the simulation result and as seen clearly, “operand_a” and “operand_b” take the values properly and then makes addition operation and shows the result in “adder_result” signal as 7.

There are two way to test Ariane core. First way is to write some assembly commands on bootrom file but the aim of project is not writing Assembly code and it is hard to write this way each time so there is an another way to make operations on the core.

2.2.2 C level test

Let’s write a simple C code to test Ariane processor as the following;

```
$ echo '
#include <stdio.h>

int main(int argc, char const *argv[]) {
printf("Hello Ariane!\\n");
return 0;
}' >$ ariane.c
```

Then riscv compiler will be used to compile this code and convert this file into elf format file.

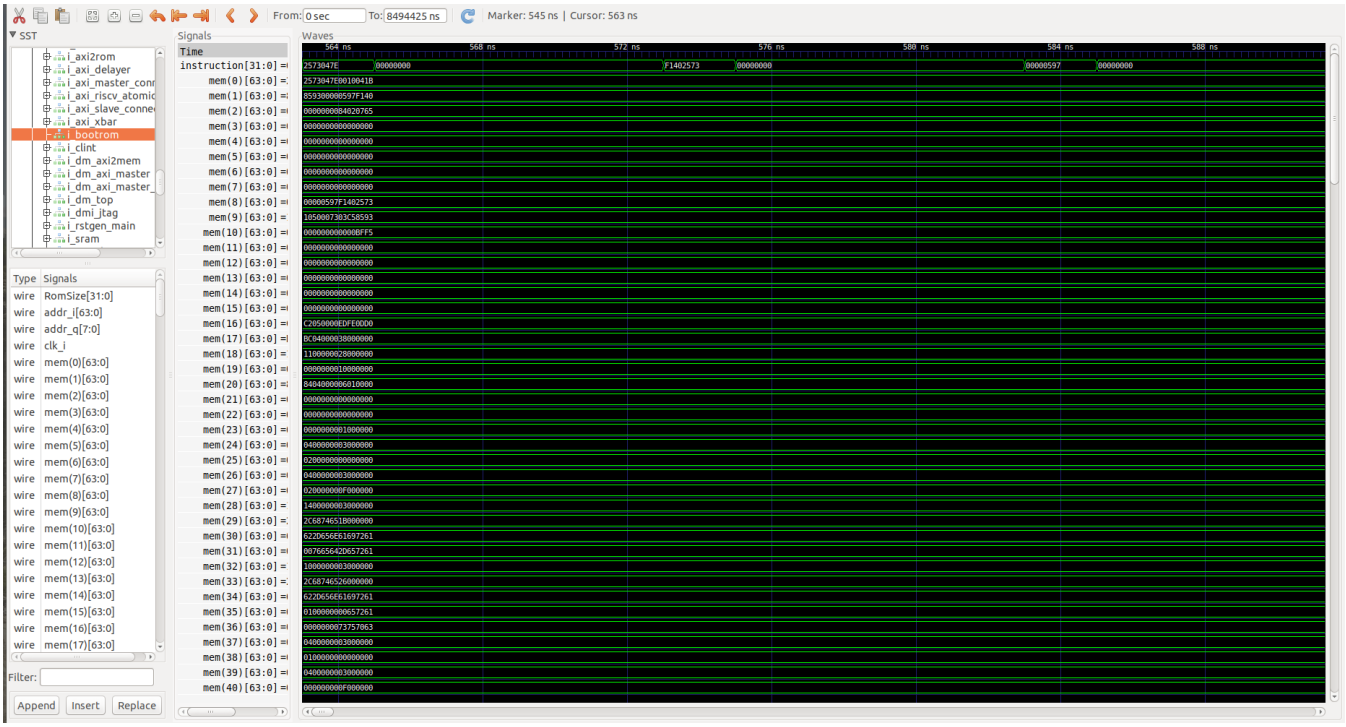
$ riscv64-unknown-elf-gcc ariane.c -o ariane.elf

Now it's time to create vcd format to work with the simulation.

$ make verilate DEBUG=1

$ work-ver/Variane_testharness -v ariane.vcd $RISCV/riscv64-unknown-elf/bin/pk ariane.elf

This process take about 30 minutes because it's size is about 20 GB and also if you work with the numbers not extra functions like printf it takes less minutes and smaller



size. The size is about 12GB.

Figure 2.5: The simulation of hello.c on the ariane processor

This simulation is the interface of GTKWAVE Simulator and we will trace the signals to check whether the core do calculations or operations correctly or not.

We will write a C code then compile it with riscv gcc compiler and obtain the vcd format file and will try to observe the simulation results. Now we will use modular multiplication algorithm with different values in order to complete "Ariane core

Implementation". There will be three values for the same code to ensure that code is working correctly on the core.

Firstly we will write our C code in the Ariane file then compile it and convert it into elf format and finally create the vcd format file and the logfile will be obtained to follow the instructions and We will stop after we reach the desired instruction. Here is the required commands written on the terminal.[2]

$ riscv64-unknown-elf-gcc C_name.c -o arbitrary_name.elf

$ make verilate DEBUG=1

$ work-ver/Variane_testharness -v arbitrary_name.vcd $RISCV/riscv64-unknown-elf/bin/pk arbitrary_name.elf

$ spike-dasm < trace_hart_00.dasm > logfile.txt

(In order to follow the instructions. Log file is created with this command.)

There are two example of C level test for modular multiplication algorithm with different input values.

The first example is as follows;

Let's make another implementation with different values; we have "a =225 " "b=17" and "mod=39" so the result will be rem[(225*17)/39]=8 so will see the result in the simulation.

```c
#include <stdio.h>

int main(int argc, char const *argv[])
{
    int res = 0; // Initialize result
int a = 225;
int b = 17;
int mod = 39;
    a = a % mod;
    while (b > 0)
    {
        // If b is odd, add 'a' to result
        if (b % 2 == 1)
            res = (res + a) % mod;

        // Multiply 'a' with 2
        a = (a * 2) % mod;

        // Divide b by 2
        b /= 2;
    }

    // Return result
    return res % mod;

}

```

Figure 2.6: C code for modular multiplication algorithm for first example

```
550843          2286637 0x1014c U (0x0000f422) c.sdsp   s0, 40(sp)
550844          2286637 0x1014e U (0x00001800) c.addi4spn s0, sp, 48
550845          2286638 0x10150 U (0x000087aa) c.mv      a5, a0
550846          2286639 0x10152 U (0xfcb43823) sd        a1, -48(s0)
550847          2286640 0x10156 U (0xfcf42e23) sw        a5, -36(s0)
550848          2286641 0x1015a U (0xfe042623) sw        zero, -20(s0)
550849          2286642 0x1015e U (0x0e100793) li        a5, 225
550850          2286645 0x10162 U (0xfef42423) sw        a5, -24(s0)
550851          2286645 0x10166 U (0x000047c5) c.li      a5, 17
550852          2286647 0x10168 U (0xfef42223) sw        a5, -28(s0)
550853          2286647 0x1016c U (0x02700793) li        a5, 39
550854          2286655 0x10170 U (0xfef42023) sw        a5, -32(s0)
550855          2286674 0x10174 U (0xfe842703) lw        a4, -24(s0)
550856          2286675 0x10178 U (0xfe042783) lw        a5, -32(s0)
550857          2286681 0x1017c U (0x02f767bb) remw      a5, a4, a5
550858          2286683 0x10180 U (0xfef42423) sw        a5, -24(s0)
550859          2286683 0x10184 U (0x0000a095) c.j       pc + 100
550860          2286696 0x101e8 U (0xfe442783) lw        a5, -28(s0)
550861          2286698 0x101ec U (0x00002781) c.addiw a5, 0
550862          2286699 0x101ee U (0xf8f04ce3) blt       zero, a5, pc - 104
550863          2286706 0x10186 U (0xfe442703) lw        a4, -28(s0)
550864          2286707 0x1018a U (0x41f7579b) sraiw     a5, a4, 31
```

Figure 2.7: Trace file of modular multiplication algorithm for first example

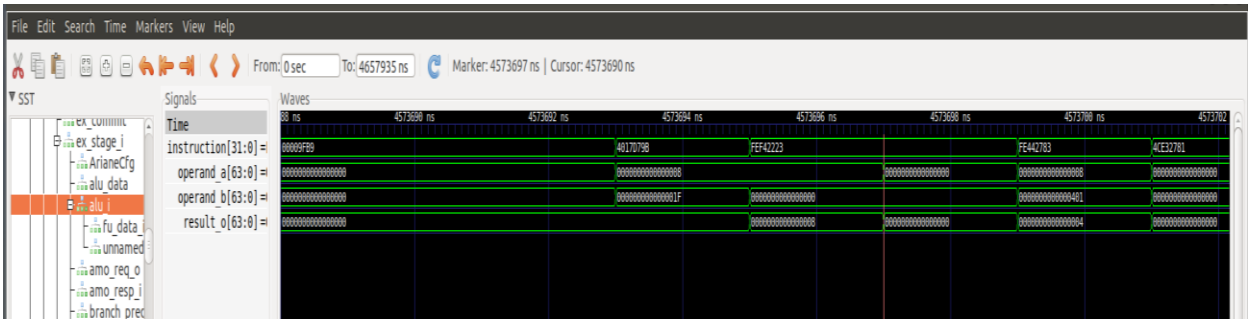


Figure 2.8: The simulaiton of modular multiplication algorithm for first example

The second example is as follows;

We have “a =11 “ “b=14” and “mod=10” so the result will be rem[(11*14)/10]=4 so will see the result in the simulation.

As seen, our C code starts when the “000047ad” instruction has come and our values are loaded properly and when the “ret” command has come as shown in the second figure as “00008082” instruction, the modular multiplication code finishes and get the result.

```
548626          2286444 0x1015e U (0x000047ad) c.li     a5, 11
548627          2286446 0x10160 U (0xfef42423) sw       a5, -24(s0)
548628          2286447 0x10164 U (0x000047b9) c.li     a5, 14
548629          2286449 0x10166 U (0xfef42223) sw       a5, -28(s0)
548630          2286449 0x1016a U (0x000047a9) c.li     a5, 10
548631          2286451 0x1016c U (0xfef42023) sw       a5, -32(s0)
548632          2286459 0x10170 U (0xfe842703) lw       a4, -24(s0)
548633          2286468 0x10174 U (0xfe042783) lw       a5, -32(s0)
548634          2286472 0x10178 U (0x02f767bb) remw     a5, a4, a5
548635          2286474 0x1017c U (0xfef42423) sw       a5, -24(s0)
548636          2286474 0x10180 U (0x0000a095) c.j      pc + 100
548637          2286487 0x101e4 U (0xfe442783) lw       a5, -28(s0)
548638          2286489 0x101e8 U (0x00002781) c.addiw a5, 0
548639          2286490 0x101ea U (0xf8f04ce3) blt      zero, a5, pc - 104
548640          2286497 0x10182 U (0xfe442703) lw       a4, -28(s0)
548641          2286498 0x10186 U (0x41f7579b) sraiw    a5, a4, 31
548642          2286500 0x1018a U (0x01f7d79b) srliw    a5, a5, 31
548643          2286501 0x1018e U (0x00009f3d) c.addw   a4, a5
548644          2286503 0x10190 U (0x00008b05) c.andi   a4, 1
548645          2286504 0x10192 U (0x40f707bb) subw     a5, a4, a5
548646          2286506 0x10196 U (0x00002781) c.addiw a5, 0
548647          2286507 0x10198 U (0x0000873e) c.mv     a4, a5
548648          2286508 0x1019a U (0x00004785) c.li     a5, 1
548649          2286509 0x1019c U (0x00f71f63) bne      a4, a5, pc + 30
548650          2286523 0x101ba U (0xfe842783) lw       a5, -24(s0)
548651          2286526 0x101be U (0x0017979b) slliw    a5, a5, 1
548652          2286528 0x101c2 U (0x00002781) c.addiw a5, 0
548653          2286529 0x101c4 U (0x0000873e) c.mv     a4, a5
548654          2286536 0x101c6 U (0xfe042783) lw       a5, -32(s0)
548655          2286539 0x101ca U (0x02f767bb) remw     a5, a4, a5
548750          2286783 0x101be U (0x0017979b) slliw    a5, a5, 1
548751          2286785 0x101c2 U (0x00002781) c.addiw a5, 0
548752          2286786 0x101c4 U (0x0000873e) c.mv     a4, a5
548753          2286793 0x101c6 U (0xfe042783) lw       a5, -32(s0)
548754          2286798 0x101ca U (0x02f767bb) remw     a5, a4, a5
548755          2286800 0x101ce U (0xfef42423) sw       a5, -24(s0)
548756          2286802 0x101d2 U (0xfe442783) lw       a5, -28(s0)
548757          2286803 0x101d6 U (0x01f7d71b) srliw    a4, a5, 31
548758          2286804 0x101da U (0x00009fb9) c.addw   a5, a4
548759          2286806 0x101dc U (0x4017d79b) sraiw    a5, a5, 1
548760          2286808 0x101e0 U (0xfef42223) sw       a5, -28(s0)
548761          2286814 0x101e4 U (0xfe442783) lw       a5, -28(s0)
548762          2286816 0x101e8 U (0x00002781) c.addiw a5, 0
548763          2286817 0x101ea U (0xf8f04ce3) blt      zero, a5, pc - 104
548764          2286826 0x101ee U (0xfec42703) lw       a4, -20(s0)
548765          2286827 0x101f2 U (0xfe042783) lw       a5, -32(s0)
548766          2286830 0x101f6 U (0x02f767bb) remw     a5, a4, a5
548767          2286832 0x101fa U (0x00002781) c.addiw a5, 0
548768          2286833 0x101fc U (0x0000853e) c.mv     a0, a5
548769          2286840 0x101fe U (0x00007422) c.ldsp   s0, 40(sp)
548770          2286840 0x10200 U (0x00006145) c.addi16sp sp, 48
548771          2286841 0x10202 U (0x00008082) ret
548772          2286841 0x100fc U (0x0000a221) c.j      pc + 264
548773          2286842 0x10204 U (0x00001141) c.addi   sp, -16
548774          2286843 0x10206 U (0x00004581) c.li     a1. 0
```

Figure 2.9: The trace file of modular multiplication for second example

Now we will observe the simulation whether the result is obtained correctly or not. Actually as seen in the simulation, when the “80826145” has come, “ret” command is obtained and it gives us the result as “4”.

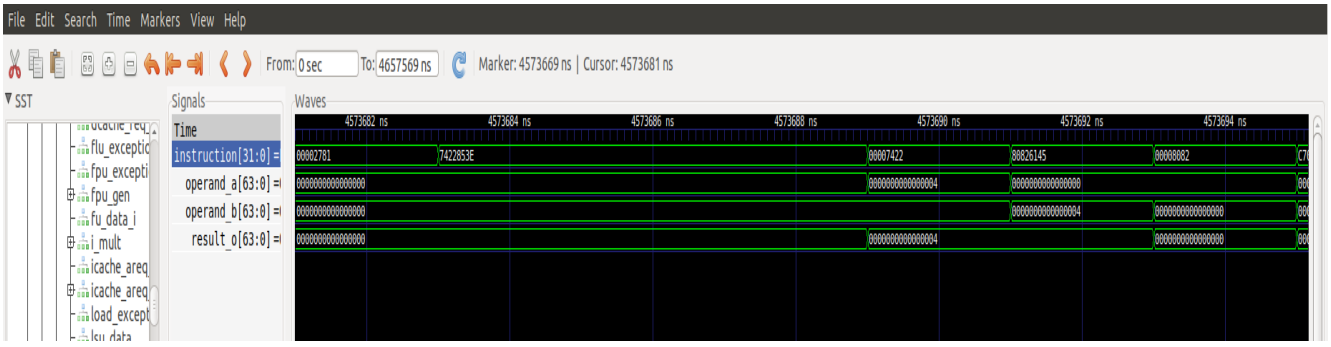


Figure 2.10: The simulation of modular multiplication for second example

3. ALU INSTRUCTION EXTENSION

To add a new instruction to the Ariane core's ALU, we first had to determine the new instruction’s function. Since this instruction will be done to improve the modular multiplication algorithm for better performance, we first performed this process three times with different input values in the processor. Each time we created a trace file and obtained every process that took place in the processor. After generating trace file, this text file will be analyzed to decide which two instructions will be choosen to replace with new instruction. This text analyzer program will be written in Java in this project.

3.1 Trace File

Trace file is a text file that includes the instruction mnemonics in Assembly Language. These mnemonics show the operations the processor performs during the process. To generate a trace file of a C code, which is modular multiplication algorithm in our case, the following steps should be done.

First of all, The Elf file is obtained from the modular multiplication algorithm in C file. To do that following command is typed to Linux terminal.

$ riscv64-unknown-elf-gcc mod_mult.c -o mod_mult.elf

Secondly, ariane core is verilated on mode DEBUG=1. Hence if it necessary to simulate the “.vcd” file can be procured from verilator model. Note that, the environmental variable must be specified before.

$ RISCV=/opt/riscv/toolchain

$ make verilate DEBUG=1

Thirdly, the elf file, which was generated in first step,is used with the Verilator model to run the RISC-V ELF by the following command.

$ work-ver/Variane_testharness $RISCV/riscv64-unknown-elf/bin/pk mod_mult.elf

The verilator model will produce trace logs. The instruction mnemonics can be obtained from this trace logs into a text file.[2]

$ spike-dasm < trace_hart_00.dasm > logfile.txt

3.2 Text Analyzer

Text Analyzer program is used to calculate and list how much instruction pairs are in the text. This program takes a text file as input and gives a text as output on console. However, there is no need to graphical user interface to do these processes. However, Java JDK and Eclipse IDE is required to write console application. There are many other language and environment options to write text analyzer program. For instance, Python language and Pycharm IDE also could be choosen. In our case, Java and Eclipse IDE is used to code this program because of our competences.

An example output of text analyzer program as follows:

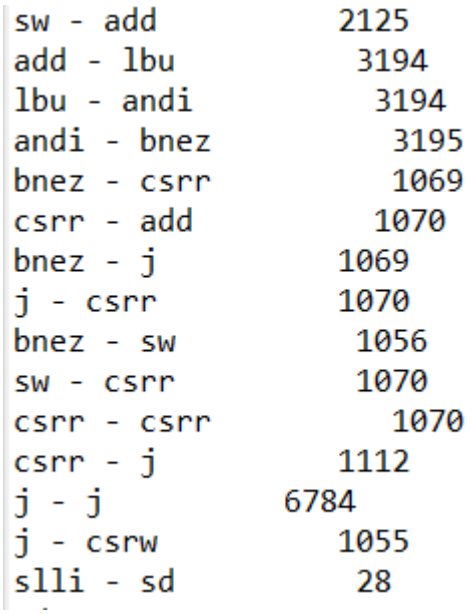


Figure 3.1: The example output of text analyzer

By using text analyzer program, three different trace files were analyzed in this program. As a result the most repitative and appropriate instruction pair is “srliw - c.addw”. One of the results can be seen in figure.

```java
package text_analysis;
import java.io.File;
import java.util.ArrayList;
import java.util.Scanner;
public class Main_func{
public static void main(String[] args) throws Exception
{
    // pass the path to the file as a parameter
    File file =
    new File("C:\\Users\\adem\\Desktop\\deneme.txt");
    ArrayList<String> lines_txt = new ArrayList<String>();
    ArrayList<String> instr = new ArrayList<String>();
    ArrayList<String> instr_operation = new ArrayList<String>();
    ArrayList<String> instr_pairs = new ArrayList<String>();

    //lines are taken from input file
    Scanner sc = new Scanner(file);
    while (sc.hasNextLine()) {
        lines_txt.add(sc.nextLine());
    }
    //instruction mnemonics are taken from lines_txt arraylist.
    int paranthesis_index = 0;
    for (int i = 0; i < lines_txt.size(); i++) {
            paranthesis_index = lines_txt.get(i).indexOf(")");
            instr.add((lines_txt.get(i).substring(paranthesis_index+2, lines_txt.get(i).length())).toString());
    }
    //Our concern is obtaining just the names of instruction.
    //The following lines will take names into insr_operation arraylist.
    int white_space_index =0;
    for (int k = 0; k < instr.size(); k++) {
        if (instr.get(k).contains(" ")) {
            white_space_index= instr.get(k).indexOf(' ');
            instr_operation.add(instr.get(k).substring(0, white_space_index));
            }
        else {
            instr_operation.add(instr.get(k));
            }
    }
    //Now, how many times is found the each instruction pairs will be calculated.
    int instr_counter [] = new int[instr_operation.size()];
    int count = 0;
    int operation = 0;
    for (int j = 0; j < instr_operation.size()-2; j++) {
        String str_1 = instr_operation.get(j);
        String str_2 = instr_operation.get(j+1);
        int flag_continue = 0;
        //continue if the same pair searched int the text before.
        for (int p = 0; p < j; p++) {
            if (str_1.equals(instr_operation.get(p)) && str_2.equals(instr_operation.get(p+1))) {
                flag_continue = 1;} }
        if(flag_continue==1) {
            continue;
        }
        for (int m = 0; m < instr_operation.size()-1; m++) {

            if(instr_operation.get(m).equals(str_1) && (instr_operation.get(m+1).equals(str_2))) {
                    count++;}}
        instr_counter[operation] = count;
        instr_pairs.add(str_1+" - "+str_2+"          "+count);
        count = 0;
        operation++;
    }
    //write pairs and counts of them in the text will be written on the console.
    for (int w = 0; w < instr_pairs.size(); w++) {
        System.out.println(instr_pairs.get(w));
    }
}
}
```

Figure 3.2: Text analyzer software in java

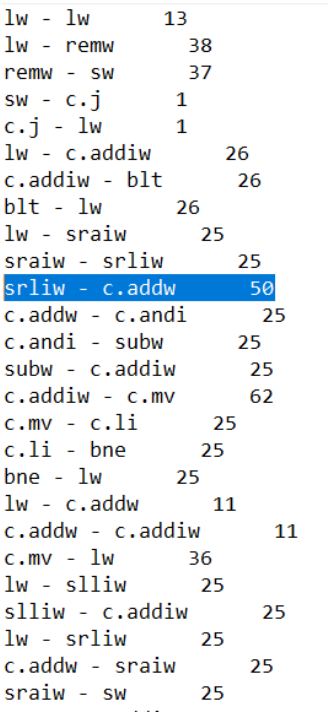


Figure 3.3: The output of text analyzer for trace file of modular multiplication

3.3 Instruction Implementation on ALU

It is known to create an instruction that performs the function of the instruction pair we found in the previous step. We have named this instruction CUSTOM1. Since this instruction will be a combination of Shift Right Logical (SRL) and Addition(ADD) intruction, the system verilog code has been written similar to the SRL instruction, and the output of the srl instruction has been entered in the ADD instruction. Since the SRL instruction is known to always have a 31-bit shift done, there is no need for additional input.[11]

srliw a5, a5, 31

addiw a4, a5

} CUSTOM1 a4, a5, a4

If we consider CUSTOM1 mnemonic, we can say that a5 and a4 are the inputs of instruction and a4 is also the output of instruction. Firstly, a5 will be shifted 31-bit right and then will be added to a4. The result of addition will be saved to a4 register again.

```
assign shift_amt = fu_data_i.operand_b;
assign shift_left = (fu_data_i.operator == SLL) | (fu_data_i.operator == SLLW);
assign shift_arithmetic = (fu_data_i.operator == SRA) | (fu_data_i.operator == SRAW);
// right shifts, we let the synthesizer optimize this
logic [64:0] shift_op_a_64;
logic [32:0] shift_op_a_32;
// choose the bit reversed or the normal input for shift operand a
assign shift_op_a    = shift_left ? operand_a_rev   : fu_data_i.operand_a;
assign shift_op_a32  = shift_left ? operand_a_rev32 : fu_data_i.operand_a[31:0];
assign shift_op_a_64 = { shift_arithmetic & shift_op_a[63], shift_op_a};
assign shift_op_a_32 = { shift_arithmetic & shift_op_a[31], shift_op_a32};
assign shift_right_result     = $unsigned($signed(shift_op_a_64) >>> shift_amt[5:0]);
assign shift_right_result32   = $unsigned($signed(shift_op_a_32) >>> shift_amt[4:0]);
// bit reverse the shift_right_result for left shifts
genvar j;
generate
  for(j = 0; j < 64; j++)
    assign shift_left_result[j] = shift_right_result[63-j];

  for(j = 0; j < 32; j++)
    assign shift_left_result32[j] = shift_right_result32[31-j];

endgenerate
assign shift_result = shift_left ? shift_left_result : shift_right_result[63:0];
assign shift_result32 = shift_left ? shift_left_result32 : shift_right_result32[31:0];
```

Figure 3.4: The hardware of shift instructions on ALU

Figure 3.4 shows the hardware of shift operations on ALU. We need to make similar hardware for the first part of CUSTOM1. There are two different result for shift operations at the end. Because Ariane is 64-bits processor. Hence, there might be calculations for 64-bits numbers. Also another situation is different kinds of shift operations can be done. In the hardware it has been managed with control signals such as shift_left. However, we will just focus the hardware of SRL.

```
// prepare operand a
assign adder_in_a    = {fu_data_i.operand_a, 1'b1};
// prepare operand b
assign operand_b_neg = {fu_data_i.operand_b, 1'b0} ^ {65{adder_op_b_negate}};
assign adder_in_b    =  operand_b_neg ;
// actual adder
assign adder_result_ext_o = $unsigned(adder_in_a) + $unsigned(adder_in_b);
assign adder_result       = adder_result_ext_o[64:1];
assign adder_z_flag       = ~|adder_result;
```

Figure 3.5: The hardware of addition instruction

Figure 3.5 shows the ADD instruction hardware. Now, the combination of these two instruction hardware will give us the hardware of the new instruction.

```
//CUSTOM instruction for modulo multiplication
  assign shift_right_result32_forCustom   = $unsigned($signed(shift_op_a_32) >>> 31);
  assign shift_result32_forCustom = shift_left ? shift_left_result32 : shift_right_result32_forCustom[31:0];

  assign adder_in_a2    = {{{32{shift_result32_forCustom[31]}}, shift_result32_forCustom[31:0]}, 1'b1};
  assign operand_b_neg2 = {fu_data_i.operand_b, 1'b0} ^ {65{adder_op_b_negate}};
  assign adder_in_b2    =  operand_b_neg2 ;
  assign adder_result_ext_o2 = $unsigned(adder_in_a2) + $unsigned(adder_in_b2);
  assign custom_result       = adder_result_ext_o2[64:1];
  assign adder_z_flag       = ~|custom_result;
```

Figure 3.6: The hardware of CUSTOM1 instruction

The hardware of CUSTOM1 instruction is coded as above picture in System Verilog. After the modification of ALU, we need to make sure the truth of hardware. Therefore we verilate Ariane processor again with these commnad in the ariane class:

$ make verilate DEBUG=1

Because of the verilator model was produced correctly, the modification of ALU is successful. After the modification process on ALU, we wanted to generate an executable file to test the Ariane processor and see its simulation at GTKwave. However, the compiler did not detect the new instruction. Because after adding a new instruction to ALU, the compiler needs to be modified accordingly. We take the following error during the test.[12]

Figure 3.7: Compiler unrecognized opcode error

To sum up, we have to modify riscv compiler after extension of ALU. The compiler has to know what is new in the ALU.

4. CONCLUSION AND FUTURE WORK

Ariane implements the 64-bit RISC-V instruction set which is a 6 stage, in order CPU and single issue. Also Ariane chip is designed for RISCV architecture due to its open source availability. The first process done in the project is to complete the implementation of Ariane core with modular multiplication algorithm and check whether it works correctly or not. After implementation and analyzing the simulation resuults, the next step is to make extension on ALU with adding some costom instructions.

In conclusion, we completed implementation of core and ALU extension in correct way but we could not do the reducing process time for modular multiplication because of Riscv GNU Toolchain. The problem is that compiler does not identify the instructions we add as CUSTOM1 in the ALU and there should be made some changes in complier as well that's why the modular multiplication algorithm with extended ALU could not be compiled accordingly and bitsteam of the core could not completed as well.

The first future work is the speed analysis of the modular multiplication algorithm can be done by generating the bitstream file by removing the compiler error with the proper extension to be made in riscv gcc compiler. Secondly, in any area such as complex cryptology algorithms, artificial intelligence studies and even image processing studies, in terms of performance , better results can be obtained by making proper extensions in the future of this study.

REFERENCES

[1] **A. Waterman, Y. Lee, D. A. Patterson, and K. Asanovi**, “The RISCV instruction set manual: User-level ISA, version 2.0,” Dept. Elect. Eng. Comput. Sci., Univ. California, Berkeley, Berkeley, CA, USA, Tech. Rep. UCB/EECS-2016-118, 2014, vol. 1.

[2] Openhwgroup/cva6. GitHub. (2020). Retrieved 15 July 2020, from https://github.com/openhwgroup/cva6.

[3] **C. B. Güngör, Y. Öndeş, T. T. Sarı and B. Uçkun**, "Instructoıon Set Extension of Some Processors for Secure IoT Implementation", Istanbul, 2018.

[4] **B. Sürer**, "Implementation Of A Soc By Using Lowrisc Processor On An Fpga For Image Filtering Applications”, Istanbul, 2019.

[5] **F. Zaruba and L. Benini**, "The Cost of Application-Class Processing: Energy and Performance Analysis of a Linux-Ready 1.7-GHz 64-Bit RISC-V Core in 22-nm FDSOI Technology," in IEEE Transactions on Very Large Scale Integration (VLSI) Systems, vol. 27, no. 11, pp. 2629-2640, Nov. 2019, doi: 10.1109/TVLSI.2019.2926114.

[6] "riscv/riscv-gnu-toolchain", GitHub, 2020. [Online]. Available: https://github.com/riscv/riscv-gnu-toolchain. [Accessed: 18- Jan- 2020].

[7] **Xu Lin, Y**. (2019). An Architectural Journey into RISC Architectures for HPC Workloads [Ebook] (pp. 34-35). Ying Hao Xu Lin. Retrieved 15 November 2019, from https://upcommons.upc.edu/handle/2117/131610.

[8] "riscv/riscv-tools", GitHub, 2020. [Online]. Available: https://github.com/riscv/riscv-tools. [Accessed: 18- Jan- 2020].

[9] "Documentation - Verilator - Veripool", *Veripool.org*, 2020. [Online]. Available: https://www.veripool.org/projects/verilator/wiki/Documentation. [Accessed: 1- Feb- 2020].

[10] "GTKWave", *Gtkwave.sourceforge.net*, 2020. [Online]. Available: http://gtkwave.sourceforge.net/. [Accessed: 5- Feb- 2020].

[11] **S. Payvar, E. Pekkarinen, R. Stahl, D. Mueller-Gritschneder and T. D. Hämäläinen**, "Instruction Extension of a RISC-V Processor Modeled with IP-XACT," 2019 IEEE Nordic Circuits and Systems Conference (NORCAS): NORCHIP and International Symposium of System-on-Chip (SoC), Helsinki, Finland, 2019, pp. 1-5, doi: 10.1109/NORCHIP.2019.8906975.

[12] **G. Tagliavini, S. Mach, D. Rossi, A. Marongiu and L. Benini,** "Design and Evaluation of SmallFloat SIMD extensions to the RISC-V ISA," 2019 Design, Automation & Test in Europe Conference & Exhibition (DATE), Florence, Italy, 2019, pp. 654-657, doi: 10.23919/DATE.2019.8714897.

CURRICULUM VITAE

Name Surname **:** Adem Eren

Place and Date of Birth **:** Denizli 09.07.1996

E-Mail **:** erena17@itu.edu.tr

Adem Eren finished primary school at Kızılcasöğüt primary school in Denizli and high school at Şevkiye Özel Anatolian Teacher High School in Denizli. He is currently senior year student at Electronics and Communication Engineering in Istanbul Technical University Electrical-Electronics Faculty. He completed his interships at Arçelik A.Ş. in Tekirdağ and ASELSAN A.Ş. in Ankara.

Name Surname **:** Mehmet Emre Yağar

Place and Date of Birth **:** Konya 09.02.1995

E-Mail **:** yagar17@itu.edu.tr

Mehmet Emre Yağar finished primary school at Barboros primary school in Konya and high school at Konya Karatay Toki Anatolian High School in Konya. He is currently senior year student at Electronics and Communication Engineering in Istanbul Technical University Electrical-Electronics Faculty. He completed his interships at Türk Telekom. in Konya and TÜBİTAK in Gebze.