ISTANBUL TECHNICAL UNIVERSITY
ELECTRICAL-ELECTRONICS FACULTY

ACCELERATION OF THE FALCON POST QUANTUM DIGITAL SIGNATURE ALGORITHM IN HARDWARE

SENIOR DESIGN PROJECT

Bora KIRAN
Cenker ÇETİNKAYA

ELECTRONICS AND COMMUNICATION ENGINEERING DEPARTMENT

Uygundur
Berna Örs Yalçın
15.08.2025

AUGUST 2025

ISTANBUL TECHNICAL UNIVERSITY
ELECTRICAL-ELECTRONICS FACULTY

ACCELERATION OF THE FALCON POST QUANTUM DIGITAL SIGNATURE ALGORITHM IN HARDWARE

SENIOR DESIGN PROJECT

Bora Kıran
040210069
Cenker ÇETİNKAYA
040210106

ELECTRONICS AND COMMUNICATION ENGINEERING DEPARTMENT

Project Advisor: Prof. Dr. Sıdıka Berna Örs Yalçın

AUGUST 2025

İSTANBUL TEKNİK ÜNİVERSİTESİ
ELEKTRİK-ELEKTRONİK FAKÜLTESİ

FALCON POST KUANTUM DİJİTAL İMZA ALGORİTMASININ DONANIMSAL OLARAK HIZLANDIRILMASI

LİSANS BİTİRME TASARIM PROJESİ

Bora Kıran
040210069
Cenker ÇETİNKAYA
040210106

Proje Danışmanı: Prof. Dr. Sıdıka Berna Örs Yalçın

ELEKTRONİK VE HABERLEŞME MÜHENDİSLİĞİ BÖLÜMÜ

AĞUSTOS, 2025

We are submitting the Senior Design Project Report entitled as “ACCELERATION OF THE FALCON POST QUANTUM DIGITAL SIGNATURE ALGORITHM IN HARDWARE”. 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 .

BORA KIRAN ....
040210069

CENKER ÇETİNKAYA ....
040210106

FOREWORD

Our journey at this inspiring, educative, and joyful institution is coming to an end. Istanbul Technical University holds a special place in our lives. The friends we have met here and the professors who have supported our development each day have contributed immensely to our personal and academic growth.

Above all, we are deeply grateful to our advisor and greatest supporter, Prof. Dr. Sıddıka Berna Örs Yalçın. Without her invaluable guidance and encouragement, this project would not have been possible. It has always been an honor to work under her, and we have learnt things from her that will help us in school and in life for a long time.

We would also like to express our heartfelt thanks to our families and friends. Whenever we stumbled, they were there to lend a hand and help us rise again.

Leaving this chapter of our lives with cherished memories, valuable experiences, and a deep sense of gratitude, ready to take the next steps in our journey.

Aug 2025

Bora KIRAN
Cenker ÇETİNKAYA

TABLE OF CONTENTS

Page

FOREWORD .... **v**
TABLE OF CONTENTS .... **vii**
ABBREVIATIONS .... **viii**
LIST OF FIGURES .... **ix**
SUMMARY .... **xi**
ÖZET .... **xii**
1. INTRODUCTION – **1**
  1.1 Purpose of Project 1
  1.2 About the Project Report 1
2. GENERAL INFORMATION **2**
  2.1 Post-Quantum Crypthography and Falcon Algorithm 2
  2.2 Floating Point Representation 3
  2.3 RISC-V Architecture 4
  2.4 Nexys 4 DDR 6
  2.5 Vivado and Vitis 8
  2.6 MicroblazeV 9
  2.7 AXI4/AXI-Stream Bus 10
  2.8 Literature Review 12
3. REALIZATION OF FALCON ON COMPUTER **14**
  3.1 Structure of the Falcon C Code 14
  3.2 Execution of the Falcon C Code 17
  3.3 Signature Generation on Computer 19
4. IMPLEMENTATION OF THE FALCON ON MICROBLAZE-V **27**
  4.1 Preparing the Block Diagram 27
  4.2 Executing the Falcon Software on Microblaze V 31
  4.3 Design of the Accelerator 35
5. TEST RESULTS **54**
  5.1 Acceleration of the Determined Functions 54
6. REALISTIC CONSTRAINTS AND CONCLUSIONS **58**
  6.1 Practical Application of this Project 58
  6.2 Realistic Constraints 58
    6.2.1 Social, environmental and economic impact 58
    6.2.2 Cost analysis 58
    6.2.3 Standards 59
    6.2.4 Health and safety concerns 59
  6.3 Future Work and Recommendations 59
REFERENCES **60**
CURRICULUM VITAE **63**

ABBREVIATIONS

ALU :Arithmetic Logic Unit
CSR :Control and Status Registers
DDR : Double Data Rate
DSP : Digital Signal Processor
FFT : Fast Fourier Transform
FPGA **:** Field Programmable Gate Array
IEEE : Institute of Electrical and Electronics Engineers
IP : Intellectual Property core
IoT **:** Internet of Things
LUT : Look-Up Table
MIG : Memory Interface Generator
NIST : National Institute of Standards and Technology
NTRU : Nth-degree Truncated Polynomial Ring Units
RTL : Register-Transfer Level
SoC : System on Chip
UART : Universal Asynchronous Receiver-Transmitter
USB : Universal Serial Bus
ZIP : ZIP archive format

LIST OF FIGURES

Page

Figure 2.1 : Nexys 4 DDR FPGA. 7
Figure 3.1 : Refence Code of the Falcon Implementation..14
Figure 3.2 : Makefile for the Build System 16
Figure 3.3 : Functional Tests. 16
Figure 3.4 : Naming Macros. 17
Figure 3.5 : Configurations. 17
Figure 3.6 : Modification for Printing the Configuration. 18
Figure 3.7 : Output of the Functional Tests. 18
Figure 3.8 : Code of the test_sign() 19
Figure 3.9 : Verification of the Signatures Inside the Test..22
Figure 3.10 : The Extracted Message, Signature and the Public Key … 22
Figure 3.11 : Changes to the Test Code …...26
Figure 4.1 : Configuration of the Microblaze V … 27
Figure 4.2 : AXI Configuration of the Microblaze V …..28
Figure 4.3 : Changes to the Test Code …..28
Figure 4.4 : Addition of the DDR2 SDRAM …..29
Figure 4.5 : Block Diagram of the MicroblazeV System … 30
Figure 4.6 : Address Map of The MicroblazeV system. … 31
Figure 4.7 : Validation of the Block Diagram … 31
Figure 4.8 : Stack and Heap Sizes … 32
Figure 4.9 : Inline Assembly Function For Measuring Clock Cycles … 32
Figure 4.10 : Code for measuring the Performance of the Hardware System …......33
Figure 4.11 : Test Sign …..33
Figure 4.12 : Validation Code of the Extracted Signature in the Hardware …..........34
Figure 4.13 : Validation of the Extracted Signature on FPGA …..............................34
Figure 4.14 : Function of poly_add()…...35
Figure 4.15 : Function of poly_sub() …..35
Figure 4.16 : Code for the poly_mul_fft Function … 36
Figure 4.17 : Code for the poly_muladj_fft Function … 37
Figure 4.18 : Code of the poly_mulselfadj_fft Function … 37
Figure 4.19 : Code of the poly_mulconst Function … 38
Figure 4.20 : Xilinx Floating Point IP … 38
Figure 4.21 : Flow Control Option … 39
Figure 4.22 : TLAST Signal Configuration …..39
Figure 4.23 : AXI4 Peripheral IP Generation …..40
Figure 4.24 : New IP with the Interfaces …..41
Figure 4.25 : Source Files of The VVD2…...41
Figure 4.26 : Memory of the Accelerator … 42
Figure 4.27 : Address Routing Logic and Write Enable Logic … 43

Figure 4.28 : Control Logic for the Registers … 44
Figure 4.29 : Master Stream Pointer Logic …...45
Figure 4.30 : Logic for the Stream Slave … 45
Figure 4.31 : Block Diagram With The Accelerator …..46
Figure 4.32 : Definitions for the Accelerator …..47
Figure 4.33 : Modification of the poly_mul Function … 48
Figure 4.34 : Modification of the poly_sub Function … 49
Figure 4.35 : The Modification of the poly_mul Function … 52
Figure 4.36 : Modification of the poly_mulselfadj Function …................................53
Figure 5.1 : Test function for poly_add … 54
Figure 5.2 : Test Results of the Accelerator … 55
Figure 5.3 : Utilization and Power Consumption of the Accelerated System … 56
Figure 5.4 : Utilization and Power Consumption of the Base System … 56
Figure 5.5 : Timing Summary of the Base System …...57
Figure 5.6 : Timing Summary of the Accelerated System … 57

ACCELERATION OF THE FALCON POST QUANTUM DIGITAL SIGNATURE ALGORITHM IN HARDWARE

SUMMARY

Classical public-key encryption schemes are not secure against quantum attacks. The rapid maturation of quantum computers in hardware and algorithms necessitates studies on quantum-resistant cryptography (PQC). The Falcon digital signature scheme, selected by NIST during the standardization process, is a prominent candidate in this context. In this study, a hardware accelerator integrated into the system via the AXI bus was designed and evaluated to accelerate Falcon's intensive floating-point vector operations on embedded platforms with limited processing power. While the design is presented to the software as a single memory-mapped device, the module contains two serial feeder blocks and two pipeline cores (an adder and a multiplier). Each feeder sequentially reads data from the memory to which it is connected, performs the necessary formatting, and transmits it to the relevant core in a pipelined manner. The cores operate in a time-sharing manner, producing one output per cycle after the pipeline is full. Control/status registers and input/output buffers are memory-mapped and accessed via AXI; burst access is supported in data transfer.

In the application flow, the official Falcon C reference was first validated on a computer, then the same codebase was run on an FPGA to compare the hardware-accelerated and software-accelerated paths. Vector functions that significantly contributed to Falcon's overall time were selected as speedup targets: poly_add and multiplication-based routines in the transformation domain (poly_mul_fft, poly_muladj_fft, poly_mulselfadj_fft, poly_mulconst). Measurements were taken on a cycle-by-cycle basis, with mcycle readpoints placed around software and hardware executions invoked consecutively with the same inputs. The results indicate a speedup of approximately 78 percent on average, with a range of 70–83 percent for these vector functions. The accelerator's impact on area and power was limited, maintaining a clock frequency of 100 MHz. Under favorable data path conditions, significant reductions in power consumption per job were also observed.

However, the overall gain in end-to-end signing time is limited, as the total latency is concentrated in the transformation steps, particularly the FFT and NTT. This finding demonstrates that to extend the high throughput achieved at the hardware core level to the entire Falcon pipeline, data movement and transformation layers must also be addressed. Consequently, this accelerator, comprised of adder and multiplier pipeline cores and two independent serial feeders under a single module and programmed via AXI, effectively accelerates Falcon's vector operations in embedded systems, prioritizing low cost and portability. Future work aims to further optimize data path utilization and, in particular, directly accelerate FFT/NTT in hardware. This is expected to yield further gains in both end-to-end latency and energy cost.

FALCON POST KUANTUM DİJİTAL İMZA ALGORİTMASININ DONANIMSAL OLARAK HIZLANDIRILMASI

ÖZET

Klasik açık anahtarlı şifreleme şemaları kuantum saldırılarına karşı güvenli değildir. Kuantum bilgisayarlarının donanım ve algoritma cephesinde hızla olgunlaşması, kuantuma dayanıklı kriptografi (PQC) çalışmalarını zorunlu kılmaktadır. NIST’in standartlaşma sürecinde seçtiği Falcon sayısal imza şeması, bu bağlamda öne çıkan adaylardan biridir. Bu çalışmada, sınırlı işlem gücüne sahip gömülü platformlarda Falcon’un yoğun kayan noktalı vektör işlemlerini hızlandırmak amacıyla AXI veri yolu üzerinden sisteme entegre edilen bir donanım hızlandırıcı tasarlanmış ve değerlendirilmiştir. Tasarım, yazılıma tek bir bellek-eşlemeli aygıt olarak sunulmakla birlikte, modül içinde iki seri besleyici blok ile iki boruhattı çekirdeği (toplayıcı ve çarpıcı) barındırır. Her besleyici, bağlı olduğu bellekteki verileri sıralı olarak okur, gerekli biçimlendirmeyi yapar ve ilgili çekirdeğe boruhattı düzeninde iletir; çekirdekler zaman paylaşımlı çalışır ve boruhattı dolumundan sonra çevrim başına bir çıktı üretir. Denetim/durum kayıtları ve giriş/çıkış tamponları bellek eşlemeli olup AXI üzerinden erişilir; veri aktarımında burst erişimler desteklenmektedir.

Uygulama akışında önce resmi Falcon C referansı bilgisayarda doğrulanmış, ardından aynı kod tabanı FPGA üzerinde çalıştırılarak donanım hızlandırmalı yol ile yazılım yolu karşılaştırılmıştır. Hızlandırma hedefi olarak Falcon’un toplam süresine anlamlı katkı veren vektör fonksiyonları seçilmiştir: poly_add ve dönüşüm alanındaki çarpım temelli rutinler (poly_mul_fft, poly_muladj_fft, poly_mulselfadj_fft, poly_mulconst). Ölçümler, aynı girişlerle arka arkaya çağrılan yazılım ve donanım yürütmeleri etrafına yerleştirilen mcycle okuma noktalarıyla çevrim bazında alınmıştır. Elde edilen sonuçlar, söz konusu vektör fonksiyonlarında yaklaşık yüzde 70–83 aralığında, ortalama yaklaşık yüzde 78 hızlanmaya işaret etmektedir. Hızlandırıcının alan ve güç üzerindeki etkisi sınırlı kalmış; kullanılan 100 MHz saat frekansı korunmuştur. Uygun veri yolu koşullarında, iş başına enerji tüketiminde de belirgin azalmalar gözlenmiştir.

Bununla birlikte uçtan uca imzalama süresindeki toplam kazanç sınırlı kalmaktadır; zira toplam gecikme, dönüşüm adımlarında özellikle FFT ve NTT'de yoğunlaşmaktadır. Bu bulgu, donanım çekirdeği düzeyinde sağlanan yüksek verimin Falcon’un tüm akışına taşınması için veri hareketinin ve dönüşüm katmanlarının da ele alınması gerektiğini göstermektedir. Sonuç olarak, tek modül altında toplayıcı ve çarpıcı boruhattı çekirdekleri ile iki bağımsız seri besleyiciden oluşan ve AXI üzerinden programlanan bu hızlandırıcı, düşük maliyet ve taşınabilirlik öncelikleriyle Falcon’un vektör işlemlerini gömülü sınıf sistemlerde etkin biçimde hızlandırmaktadır. Gelecek çalışmalarda, veri yolu kullanımının daha da verimli hâle getirilmesi ve özellikle FFT/NTT’nin doğrudan donanımsal hızlandırılması hedeflenmektedir. Bu sayede hem uçtan uca gecikme hem de enerji maliyetinde daha yüksek ölçekli kazanımların elde edilmesi beklenmektedir.

1. INTRODUCTION

1.1 Purpose of Project

As quantum computers grows more capable, current cryptographic methods risk being broken, for this reason new cryptographic algorithms are being designed for quantum secure cryptography. As part of the NIST standartization process for post quantum cryptography, post quantum digital signature algorithm Falcon is chosen for standardization. However, Falcon demands a high processing power which can be a bottleneck for systems with limited processing power esspecially in double persicion floating point support. By designing a loosely coupled hardware accelerator for the Falcon algorithm on a RISC-V-based System-on-Chip (SoC), this project aims to improve the algorithm's performance while maintaining functionality and modularity. The proposed solution seeks to speed up the selected vector operations to facilitate the adoption of the Falcon Algorithm into resource constrainted systems. Through this effort, the project contributes to the development of efficient, secure, and future-proof cryptographic systems.

1.2 About The Project Report

The report is organized as follows. Sections 1–2 state the purpose of the project and summarize the technologies and standards used. Sections 3–4 trace the implementation path from the official Falcon C reference on a computer to the FPGA system: Section 3 builds and validates the reference code, and Section 4 implements the MicroBlaze-V–based SoC, and measures/validates execution with and without the accelerator. Section 5 reports the experimental results. Section 6 discusses realistic constraints, standards and safety considerations, and concludes with findings and future work.

2. GENERAL INFORMATION

2.1 Post Quantum Crypthography and Falcon Algorithm

The field of Cryptography is the study and practice of securely encrypting information using mathematical techniques. Modern cryptography started in the mid-20th century with the founding of computers and their need for secure communications.

There are two main branches of cryptography: symmetric-key cryptography and asymmetric-key cryptography. In symmetric-key cryptography one key is used for encryption and decryption. On the other hand, in asymmetric-key cryptography there is one public key which is known by everyone and a private key which is kept secret. Messages that are encrypted by public key can only be decrypted by the private key. Messages that are encrypted by the private key can be decrypted by the public key resulting in confirmation of the sender, since private key is kept secret. This process of verifying the sender's identity is called a digital signature. This happens because if message can be decrypted by the public key, it must have been encrypted by the private key.

Asymmetric-key cryptography is used extensively on the internet. In both types of cryptography algorithms are designed so that any attack that tries to break them takes so much time that secret information becomes obsolete. However, algorithms that were designed in the past assumed attackers would only possess a classical computer, not a quantum computer.

With advancements in quantum computers, attackers using a quantum computer is becaming a possible scenario. This scenario is threat for many algorithms. Especially for public-key algorithms because all standard contemporary public-key algorithms currently in use are vulnerable to attacks from quantum computers, as noted in [1].

To address this emerging threat, many alghoritms are proposed for standartisation to NIST (National Institute of Standarts and Technology [2]. One such algorithm is Falcon [3], which is chosen for standartisation by NIST. Falcon, while offering security against quantum attacks, has significantly higher computational demands compared to traditional algorithms.

Falcon is a lattice-based algorithm that uses the NTRU mathematical structure, known for its small signature size and fast verification. Its security comes from both the short vector problem and polynomials that represent the complex state of the lattice.

For key generation, first a large lattice is designed. The private key is a set that contains short vectors inside this lattice. The public key is a polynomial that represents the complex form of the lattice and is used to describe it. This polynomial is obtained using the Fast Fourier Transform (FFT).

To sign a message, first its hash is taken with SHAKE-256. The resulting hash is then transformed in a random but controlled way using Gaussian Sampling. The short vector obtained in the end becomes the signature of the message. The security of the private key is ensured thanks to Gaussian sampling.

For verification, first it is checked whether the public key is in the correct place on the lattice. If the key is correct and the hashes match, the verification is successfully completed. This is how the algorithm works, and it is expected to stay secure in the post-quantum era.

2.2 Floating Point Representation

Computers were originally designed to work with integers. But over time, there was also a need to work with decimal numbers. To solve this, floating-point representation was developed. This method allows computers to understand decimal numbers and do calculations with them.

Floating-point representation [4] has a sign bit, an exponent bit, and a mantissa (fraction) part. The Falcon algorithm uses the double-precision floating-point format from the IEEE 754 standard. In this format, the sign bit is 1 bit, the exponent is 11 bits, and the mantissa is 52 bits. In binary form, these bits represent a number, and to convert it to decimal following formula is used:

$$Decimal\ Number = (-1)^{\text{sign}}\ x\ \left(1 + \frac{M}{2^{52}}\right) x\ 2^{(E-bias)}$$

E : Exponent
M : Mantissa

The bias value depends on the precision of the floating-point representation. According to IEEE standards, this value is 127 for single-precision and 1023 for double-precision. Except for subnormal numbers, there's an "invisible" 1 bit on the left of the mantissa. But if the exponent value is all zeros (which means the number is subnormal), this hidden bit becomes 0 instead of 1. This way, the problem of representing very small positive numbers is solved, and the gap between normal numbers is filled.

The Falcon algorithm makes some changes to the floating-point representation. First, instead of using a bias of 1023, it uses 1076. Because of mantissa scaling, the effective value is actually 1077. When Falcon encounters values that would be subnormal in IEEE, instead of setting the hidden bit in the mantissa to 0, it just treats the value as 0 directly, because it's too small. Also, when rounding results, Falcon rounds to the nearest value. If the result is exactly halfway between two numbers, it picks the even one. It checks this by looking at the last 3 bits of the mantissa.

If Falcon wants to represent the whole thing with a single value, it uses the `uint_64t` type. For this, the exponent is given directly as the mathematical exponent, without applying the bias. The mantissa should be between 2^{54} and 2^{55}.

2.3 RISC-V Architecture

Processors are designed to follow an instruction set that draws a path for them to run certain commands. RISC-V [5] is based on Reduced Instruction Set Computing (RISC) [10], which focuses on simplicity and efficiency by using small and effective instruction sets. Unlike other instruction sets, RISC-V is open-source and non-copyrighted, meaning everyone can use and implement it without paying fees. Thanks to this openness, it can be used without restrictions in a wide range of areas, starting from embedded systems, and it’s highly accessible. Many companies and educational institutions around the world design new systems and educate people because of this. But RISC-V doesn’t mean just a single instruction set. It’s actually a structure that brings together many instruction sets depending on capabilities.

The most basic instruction set is called RV32I. Here, the “I” stands for Integer, and “32” shows how many bits it uses for addressing. This set is only capable to do some certain operations on integers. The processor can access memory through load and store commands. Mathematical operations are mostly stored in memory blocks called registers, and then transferred to memory using load and store commands. Some of the commands this set can do are addition, subtraction, shifting, conditional branching, and jumping.

Another set is RV32IMC. This set includes all RV32I commands and adds two more instruction sets. The “M” set contains multiplication and division instructions, both signed and unsigned, as well as commands like finding the remainder. In RISC-V, multiplication and division operations take longer and affect the processor’s design. In single-cycle designs, these operations would determine the processor’s speed, which wouldn’t be efficient. That’s why multi-cycle designs are more common in processors with the M set.

To use 32-bit addressing more efficiently with compact commands, the “C” set can be used. This set uses 16-bit commands, not of 32-bit commands. With this change code size decreases. By encoding the commands differently, the same tasks take up less space in memory. It’s one of the most important sets for embedded systems and devices designed for low power consumption. Besides the sets mentioned here, there are also others like the “F” set for floating-point operations and the “A” set for atomic memory operations in 32-bit designs.

RISC-V also has 64-bit and even 128-bit instruction sets. RV64 includes 64-bit commands and memory. The 32 registers store, process, and save 64-bit data. The Program Counter (PC) is 64 bits long and, just like in RV32, increases by 4 each time. Since most modern operating systems are based on 64-bit, this set is more compatible. It also allows a better environment for designing more complex extensions like vector operations and AI accelerators.

Apart from the instructions explained so far, RISC-V also have special registers called CSRs which is control and status registers. These are used for things like fixing errors in pipelined designs, triggering interrupts when needed depending on the program flow, and handling counters. Control and status registers can have three access levels: user level, supervisor level, and machine level. User level is for user applications, supervisor level is for operating systems, and machine level is for firmware. These 12-bit commands monitor the state of programs or hardware while they're running to help prevent possible errors.

One of the registers accessible at the machine level is called "mcycle." It's used for things like system monitoring, performance measurement, and counters. Outside of machine mode, it can only be read, and the system increases it by one with every clock signal, storing how long the processor has been running. On 32-bit systems, it's stored as "mcycle" and "mcycleh" (high 32 bits), while on 64-bit systems, it's stored as a single 64-bit register. Thanks to this feature, it will be used later in the project as a source for speed measurement.

2.4 Nexys 4 DDR

FPGA, or Field Programmable Gate Array, is a chip that can be reprogrammed and doesn't have a permanent layout. As the name suggests, it has a lot of logic gates, and besides these gates, it also includes components like DSP blocks specialized for digital signal processing, LUT units that store results for different inputs instead of using too many gates, and DDR memory. These elements are brought together and designed according to the application to create a system. The gates can be arranged in parallel if needed, so more than one separate application can run in parallel at the same time. Even though it's slower compared to processors, thanks to this parallel processing ability, it works differently from the sequential logic of processors and can be more advantageous in certain applications.

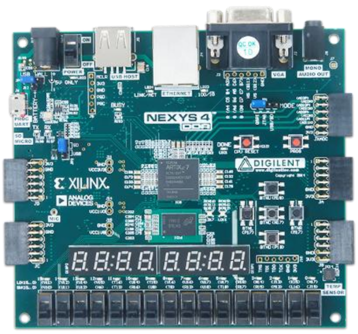


Figure 2.1: Nexys 4 DDR FPGA

Specifications of Nexys 4 DDR [6] model is listed below.

- Artix-7 100T (XC7A100TCSG324-1)
- 15,850 logic slices, each with four 6-input LUTs and 8 flip-flops
- 4,860 Kbits of fast block RAM
- Six clock management tiles, each with phase-locked loop (PLL)
- 240 DSP slices
- Internal clock speeds exceeding 450MHz
- On-chip analog-to-digital converter (XADC)
- 16 user switches
- USB-UART Bridge
- 12-bit VGA output
- 3-axis accelerometer
- 128MiB DDR2
- Pmod for XADC signals
- 16 user LEDs
- Two tri-color LEDs
- PWM audio output
- Temperature sensor
- Serial Flash
- Digilent USB-JTAG port for FPGA programming and communication
- Two 4-digit 7-segment displays
- Micro SD card connector
- PDM microphone
- 10/100 Ethernet PHY
- Four Pmod ports
- USB HID Host for mice, keyboards and memory sticks

2.5 Vivado and Vitis

Vivado Design Suite [7] is a software used for making hardware designs for AMD’s FPGA boards. Using VHDL, Verilog, or SystemVerilog languages, you can write RTL code that will be turned into hardware. VHDL has a strict syntax and is a language where many designs are described with words. On the other hand, Verilog is similar to the C language, but it’s important to remember that it’s used for hardware design. SystemVerilog is basically an improved version of Verilog, and with its Object-Oriented Programming capability, it also includes testing and software-like features.

When creating a project, you first name it and decide whether it’s going to be an RTL project or a post-synthesis project. For RTL projects, you can add the files you’ll use and choose the target language. Here you’ll only see VHDL and Verilog, but Verilog also covers SystemVerilog. Plus, these languages can work together in a compatible way. You can also add constraint files, which include information about the pins to be used on the target board. Finally, you select the target FPGA and start the project.

This design package comes with ready-to-use IP cores. These IP cores could be things like clock dividers, DDR memory cores, block RAMs, or critical parts like the AXI communication protocol. After adding them to the design, they can be defined in the chosen language and connected to other designs. You can also package your own RTL code and combine it with other cores to create a more complete block design.

The designs go through synthesis to get the logic circuits that will implement them. During synthesis, Vivado checks if the design is suitable for synthesis. Timeouts or code written only for testing that can’t be synthesized are reported as errors.

In the implementation stage, the logic circuits are arranged according to the capabilities of the target FPGA. DSP blocks, LUTs, and block RAMs inside the FPGA are used as needed. After this stage, you get results like timing analysis, resource usage reports, and power analysis.

Sometimes the FPGA’s LUT count can be over the number of LUTs in the FPGA, so the design may need to be changed.

Finally, the Bit Generation stage starts. The design adapted for the board is turned into a .bit file, which will be loaded onto the FPGA to make the FPGA as designed hardware. Vivado also includes logic analyzers that let you observe inside the FPGA while it’s running.

While Vivado helps with the hardware design, Vitis [8] continues the process by being the environment the software that will run on the designed hardware and loading it onto the hardware which is in the FPGA. The design completed in Vivado is first sent to Vitis as an .xsa (Xilinx Support Archive) file. Vitis uses this file to create the Board Support Package (BSP). The BSP is a package that acts as a bridge between hardware and software. It includes drivers, header files, and settings for the hardware defined in Vivado.

Then, the software is designed to run on the board is written in C or C++. Vitis loads this software onto the FPGA and starts its execution. To get output from the software, the Vivado design needs to have a protocol like UART connected to the output. With this extension, you can understand if the program works well.

2.6 Microblaze V

The Microblaze V [9], based on the 32-bit RISC-V architecture, is developed by AMD with high configurability in its own ecosystem, allowing developers to design systems adaptable to many different fields. A separate 64-bit design option can also be used if needed.

Among the configuration options, apart from the basic integer set, there’s the M extension that adds multiply and divide instructions, the A extension that allows atomic operations on memory hardware, the F extension for working with floating-point numbers and making complex operations easier, the C extension that reduces code size with 16-bit instructions for more efficiency, and bit manipulation extensions. Users can choose combinations of these sets according to their needs, and they can also select a microcontroller configuration.

This RISC-V core offers performance, area, frequency, and efficiency options depending on the project’s goals. For high performance, ready-to-use instruction and data caches, shifter structures, hardware multipliers, a mid-level pipeline, and peripherals are added. This boosts performance, but the area or efficiency might not be at the desired level. If the speed-up rate achieved with the hardware is higher than the area loss, this can be considered an acceptable trade-off. For applications where saving area is a priority, a short pipeline and minimal peripherals are used, which lowers performance. For frequency and efficiency, the goal is to make the most out of the available features to get the best possible results.

These ready-made designs only have the capacity for a single floating-point unit internally, so they’re not suitable for processing longer data.

Edge devices are systems that handle processing locally, instead of sending decision-making to cloud computing centers. They get necessary data via sensors or connections, process it, make decisions, and control other devices in the system if needed. Microblaze V is a suitable core for being an edge device, and it will be used for this purpose in the project.

For this project, the RISC-V core will be set to RV32IMC and run at a base frequency of 100 MHz. Depending on the desired balance between performance, area, frequency, and efficiency, it can have a 3, 4, 5, or 8-stage pipeline. With these features, it’s a good candidate for being an edge device. Its capacity is enough to handle the intended local application but not suitable for much bigger ones. It will process incoming data locally when needed, sometimes relying on software, sometimes on external hardware, and will make the final decision itself—making it a solid example of an edge device.

2.7 AXI4/AXI-Stream Bus

Advanced eXtensible Interface (AXI) [10] is a protocol that provides communication between internal hardware blocks in systems like FPGA, SoC, and ASIC. It offers high bandwidth, low latency, and flexible data transfer, which increases system performance during data transfer and creates a highly efficient working environment.

It controls data transfer through a handshake mechanism using VALID and READY signals. VALID signal is sent when the data is ready for sending to receiver. While READY signal is reflecting that receiver is ready for getting the data. When both VALID and READY are 1, the data transfer starts. For the transfer to begin, it’s enough for both the Master and Slave to have their ready signals at 1. There’s no fixed external delay.

The protocol also supports a burst mechanism. It takes the address once and transfers the data sequentially in up to 256 blocks. It can do this in three ways: continuously increasing the address (INCR), reading from the same address like in a FIFO (FIXED), or wrapping the address in certain block sizes (WRAP). AXI4 supports burst mechanism, but the lite version is not capable as AXI4.

In the AXI protocol, there are 5 separate channels for communication: Writing address channel, writing data channel, writing status channel, reading address channel and read data channel. Thanks to these separate channels, reading and writing can happen in parallel. The write address, write data, and read address channels transfer data from the Master to the Slave, while the write response and read data channels transfer data from the Slave to the Master.

AXI can work in memory-mapped or streaming mode. Memory-mapped mode works like how processors access memory elements: you send a write address and data to write, or a read address to read the data stored there. It uses the 5 separate channels mentioned above. Random access is possible, meaning it behaves like memory, but because it uses many signals, it’s not the most efficient for continuous data transfer.

Streaming mode doesn’t depend on an address. Data transfer continues as a flow without checking addresses, like a river carrying data to its destination. Without an address, it only sends data using the TDATA signal. TVALID and TREADY handle the handshake. TLAST indicates the end of a packet and is important for correct data transfer. Byte mask is sent by TKEEP signal, while custom user information is sent through TUSER.

AXI has 3 types of transmission. The first is AXI4, which fully uses the memory-mapped system and includes all features. AXI4-Lite supports only single data transfers and is used for carrying simple control and status signals. The last type is AXI-Stream [11], which transfers data in a continuous flow and is designed for systems that need continuity and speed, like video, network packets, audio, or high-speed FIFO-like data paths. In addition to the stream signals mentioned earlier, other signals can be used: TSTRB (similar to TKEEP), TID, and TDEST. TSTRB is mostly used in AXI4 to show which bits are valid but can sometimes be used in Stream (its direct equivalent here is TKEEP). TID prevents confusion between multiple streams and tags them, while TDEST (optional) is used to route the stream to multiple destinations.

For video transfer, when sending data frame by frame or based on line ends, AXI4-Stream can be used. At the end of the stream, TLAST shows the end of the line, and TUSER can indicate the start of a line. In Ethernet packet transfer, TKEEP can show which bytes are valid in the last beat of each packet. TUSER can also be used for things like marking frame starts or checking for bit errors in the system.

In the Stream protocol, controlling data with TVALID and TREADY creates backpressure. This prevents fixed delay. If the receiver is too slow or the FIFO is full, TREADY goes to 0, stopping the transfer. In this case, TVALID stays at 1, preventing the system to go into IDLE mode. When the receiver gets ready again, the system ignores the data from when the receiver wasn’t ready and starts sending from the next available data.

2.8 Literature Review

Although there are few hardware implementations of Falcon, only one implementation accelerates both verification and signature generation [12]. This design however uses Scalar-Vector Architecture of the RISCV and modifies the ALU of the core. This design is very fast and powerful far due to use of vector extension. Thus, a strong core is needed for this implementation. Also, this solution is currently limited to one specific processor whose ALU was modified. To use this design in another processor ALU of the desired processor must be modified accordingly to architecture in [12].

Design in [13] adds two accelerators on a very basic RISCV core. One of the accelerators is a loosely coupled accelerator thus it can be used in any RISCV core, but design also modifies the ALU of the core with a tightly coupled accelerator which limits the usage of the accelerator to the core modified. Thus, only the loosely coupled accelerator can be used flexibly. Also, in [13] only signature verification is accelerated not the signature generation which takes more time but is also less frequent.

Although signature generation is accelerated in [14], the design was made for ARM architecture which is a proprietary architecture unlike this RISCV in this project.

There is one full hardware implementation [15] in which High Level Synthesis was used. This kind of realization is not hardware software co-design since it doesn't accelerate the software but realizes the algorithm in hardware which is not suitable for many applications. In addition, only hardware implementation is might be wasteful since many applications readily use a processor thus adding a full hardware implementation to system means that processor (and software) is not utilized to its best.

There is one design that runs the algorithm on GPU [16] which shows the trend for more parallelism in computing but unfortunately this design requires considerable hardware. Thus, it is not feasible to realize in systems smaller in size.

Overall considering the literature and limitations of the previous designs we decided on designing the Falcon accelerator as a loosely coupled accelerator that connects to the RISCV core using AXI Bus. This choice is a tradeoff between speed and usage flexibility. Using RISCV provides flexibility to system since RISCV is open source meaning that every designer can use it without royalties. Attaching the accelerator memory bus provides flexibility since any core with basic AXI bus will be able to use the accelerator but many clock cycles will be sacrificed for memory access which limits the speed.

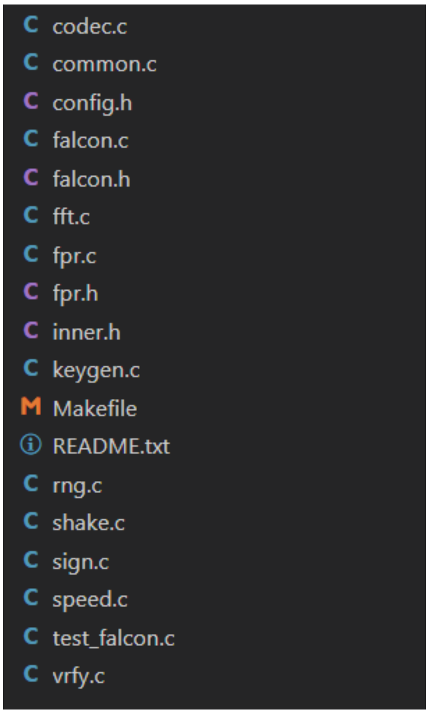
3. REALIZATION OF FALCON ON COMPUTER

3.1 Structure of the Falcon C code

Before the hardware accelerator development and software realization on MicroblazeV, the official Falcon reference implementation [3] was compiled and executed on a standard computer. This step was taken to verify the reference software and the algorithm's operation, so that outputs could be replicated on the MicroBlazeV with and without the designed accelerator.

The compilation and execution were performed in an Ubuntu 24.04.2 LTS environment running under Windows Subsystem for Linux on a Windows 10 host. GCC 13.3.0, GNU Make 4.3 and Visual Studio Code were used as the toolchain.

The source code was obtained from the official Falcon project repository[8] as a ZIP file and after extracting the contents of the downloaded ZIP, source code is analyzed. The source code consists of the files in the Figure 3.1.



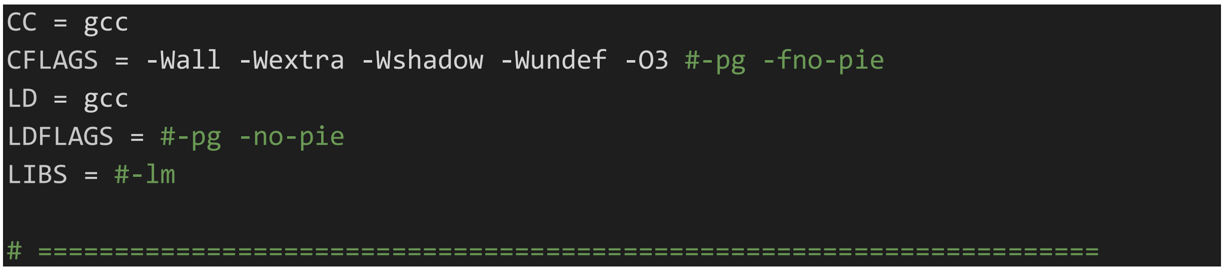
codec.c
common.c
config.h
falcon.c
falcon.h
fft.c
fpr.c
fpr.h
inner.h
keygen.c
Makefile
README.txt
rng.c
shake.c
sign.c
speed.c
test_falcon.c
vrfy.c

Figure 3.1: Refence Code of the Falcon Implementation

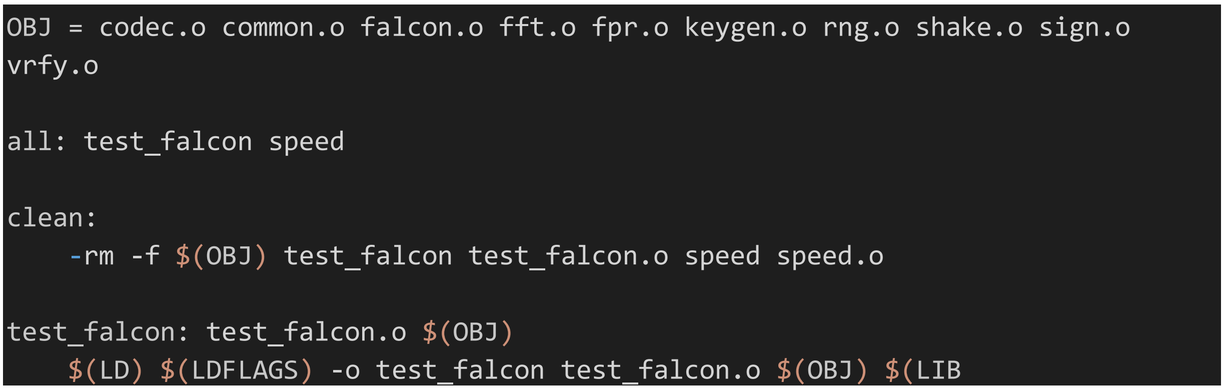
Contents of the most important files for the project are listed as follows :

- config.h: The configuration header holds compile-time macros that control the use of native or emulated floating-point arithmetic, as well as optional AVX2, FMA, or ARM-specific optimizations; if these are not defined manually, defaults are used.
- falcon.h: Header file for the Application Programming Interface. Acts as a wrapper to the implementation. Intended as the only header required by external C code. This API is not used in our project, instead the implementation is used directly.
- falcon.c: Implementation of the API, wrapper around the internal primitive functions.
- inner.h: The internal API, collects all the function declarations, constants, types, and naming rules used between the source files, so that the different parts of the code can work together without showing these details in the public API.
- fpr.h and fpr.c: Floating-point math routines for Falcon, using either native double precision if supported by the hardware or a constant-time software emulation when FPU is not available.
- fft.c: FFT and inverse FFT implementation and polynomial operations in the Number Theoric Transform and FFT domain.
- sign.c: Functions for generating a signature.
- vrfy.c: Functions for verifying a signature.
- test_falcon.c: The test

The Makefile(Figure 3.2) in the reference code is configured to produce two main executables: self checking test suite *test_falcon()*, and commented out *speed()* for benchmarking. The speed test was not used. The used functional tests that are in the reference implementation can be seen in the Figure 3.3.

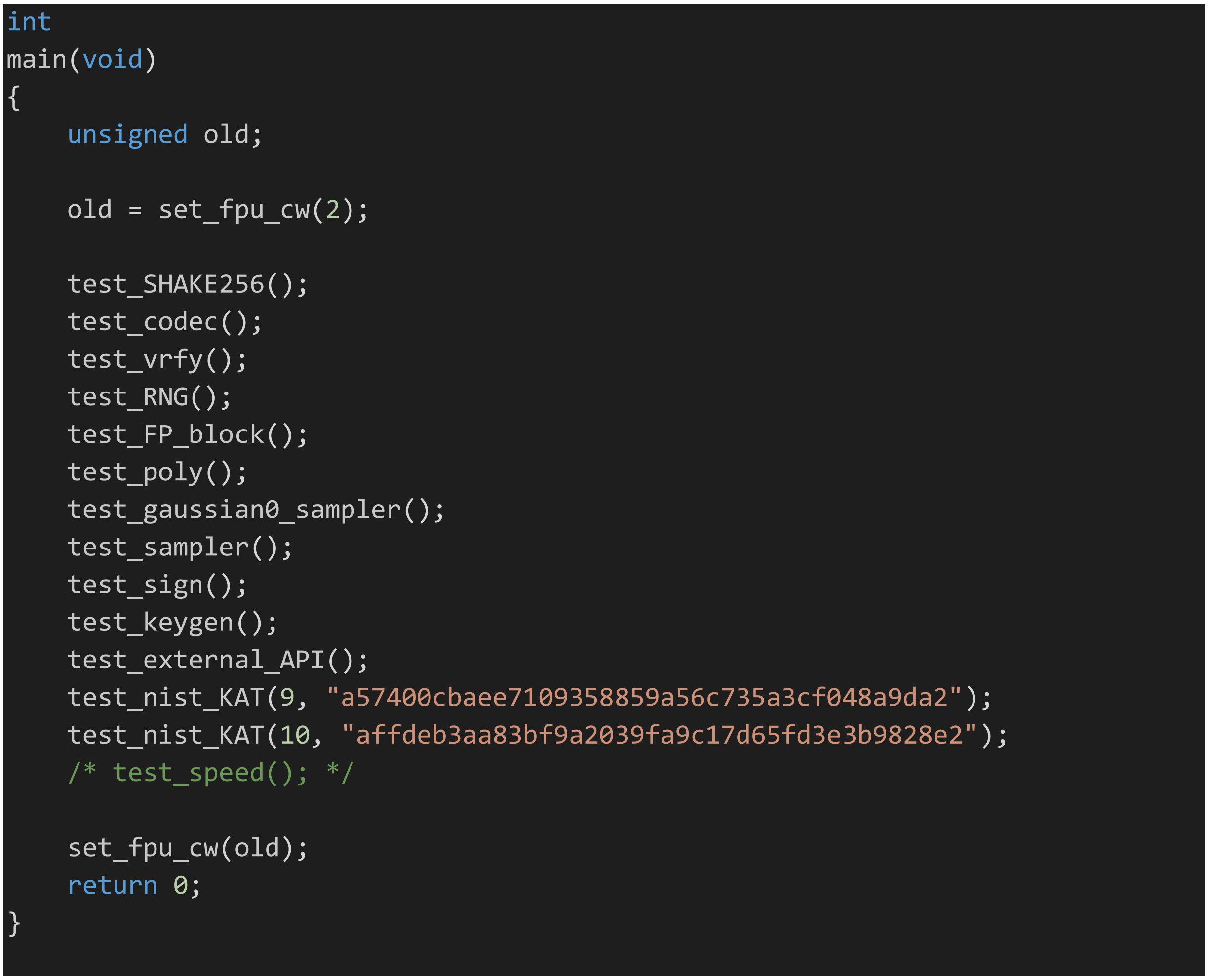


```
CC = gcc
CFLAGS = -Wall -Wextra -Wshadow -Wundef -O3 #-pg -fno-pie
LD = gcc
LDFLAGS = #-pg -no-pie
LIBS = #-lm

# =====================================================================
```



```
OBJ = codec.o common.o falcon.o fft.o fpr.o keygen.o rng.o shake.o sign.o
vrfy.o

all: test_falcon speed

clean:
    -rm -f $(OBJ) test_falcon test_falcon.o speed speed.o

test_falcon: test_falcon.o $(OBJ)
    $(LD) $(LDFLAGS) -o test_falcon test_falcon.o $(OBJ) $(LIB
```

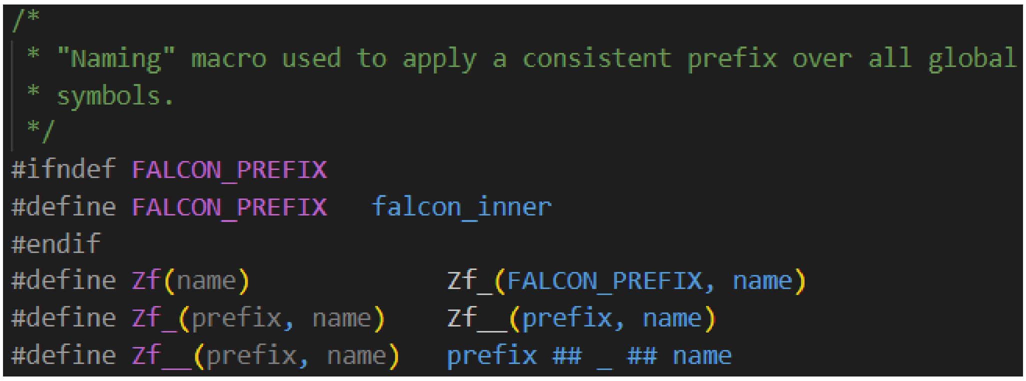
Figure 3.2 : Makefile For The Build System



```
int
main(void)
{
    unsigned old;

    old = set_fpu_cw(2);

    test_SHAKE256();
    test_codec();
    test_vrfy();
    test_RNG();
    test_FP_block();
    test_poly();
    test_gaussian0_sampler();
    test_sampler();
    test_sign();
    test_keygen();
    test_external_API();
    test_nist_KAT(9, "a57400cbaee7109358859a56c735a3cf048a9da2");
    test_nist_KAT(10, "affdeb3aa83bf9a2039fa9c17d65fd3e3b9828e2");
    /* test_speed(); */

    set_fpu_cw(old);
    return 0;
}
```

Figure 3.3 : Functional Tests

Naming macros are also used for namespace separation(Figure 3.4), as the source code contains many small functions with generic names that could clash with functions from other cryptographic libraries, standard library functions in the same project. By wrapping each internal function name in Zf(), these names are transformed into long function names such as falcon_inner_fft to create a private namespace behaviour in C, which has no built-in namespace support unlike C++. This method prevents naming conflicts.

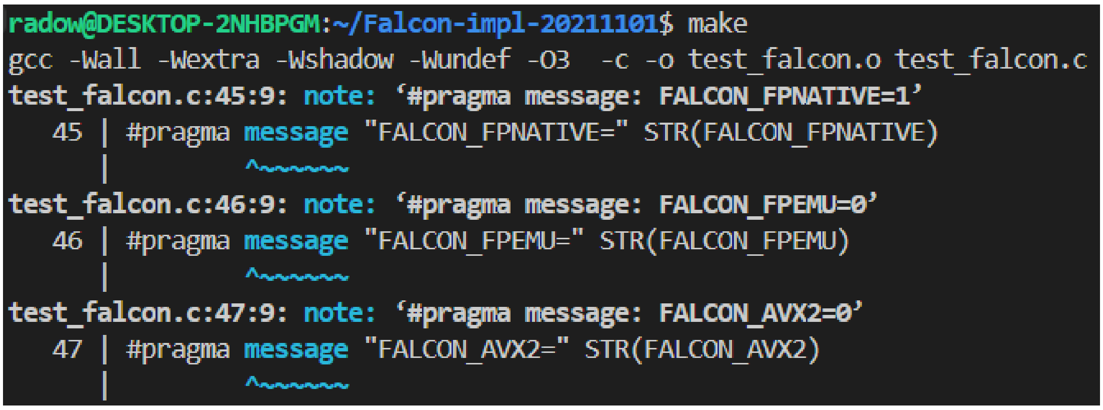


```
/*
 * "Naming" macro used to apply a consistent prefix over all global
 * symbols.
 */
#ifndef FALCON_PREFIX
#define FALCON_PREFIX   falcon_inner
#endif
#define Zf(name)              Zf_(FALCON_PREFIX, name)
#define Zf_(prefix, name)     Zf__(prefix, name)
#define Zf__(prefix, name)    prefix ## _ ## name
```

Figure 3.4 : Naming Macros

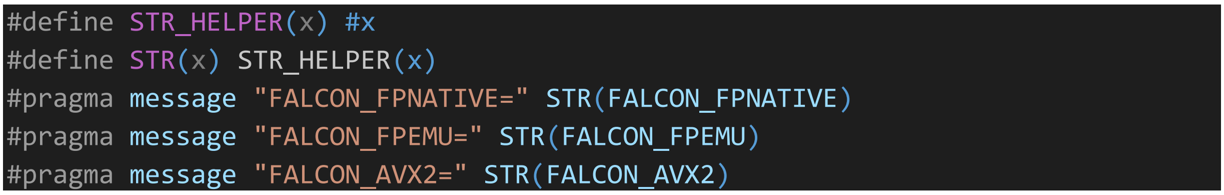
3.2 Execution of the Falcon C code

Source code was compiled with the default configuration, which enabled native double-precision floating-point operations without AVX2-specific optimizations and instructions. This configuration was automatically selected by the build system as seen in the Figure 3.5. Values of the Macros can be checked at the compile time with a slight modification of the C code (Figure 3.6). Also -O3 flag is used in compilation for optimization by default.



```
radow@DESKTOP-2NHBPGM:~/Falcon-impl-20211101$ make
gcc -Wall -Wextra -Wshadow -Wundef -O3  -c -o test_falcon.o test_falcon.c
test_falcon.c:45:9: note: ‘#pragma message: FALCON_FPNATIVE=1’
   45 | #pragma message "FALCON_FPNATIVE=" STR(FALCON_FPNATIVE)
      |         ^~~~~~~
test_falcon.c:46:9: note: ‘#pragma message: FALCON_FPEMU=0’
   46 | #pragma message "FALCON_FPEMU=" STR(FALCON_FPEMU)
      |         ^~~~~~~
test_falcon.c:47:9: note: ‘#pragma message: FALCON_AVX2=0’
   47 | #pragma message "FALCON_AVX2=" STR(FALCON_AVX2)
      |         ^~~~~~~
```

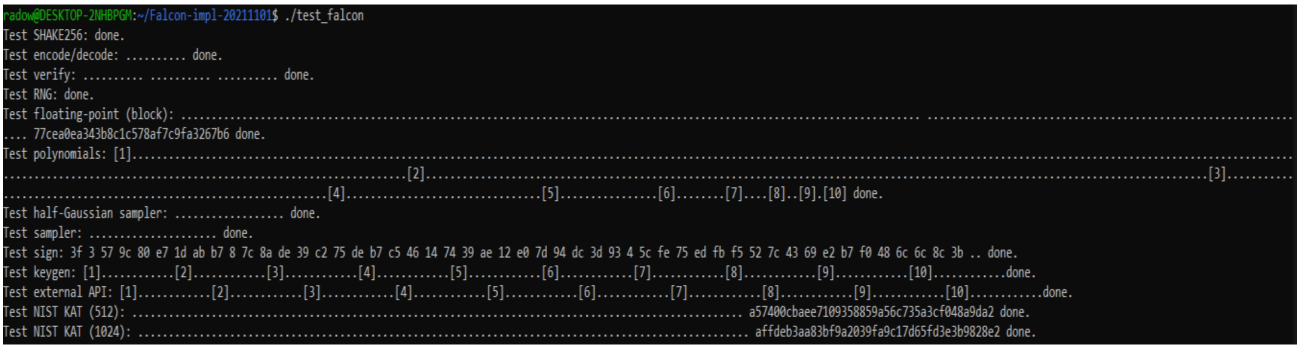
Figure 3.5 : Configurations



```
#define STR_HELPER(x) #x
#define STR(x) STR_HELPER(x)
#pragma message "FALCON_FPNATIVE=" STR(FALCON_FPNATIVE)
#pragma message "FALCON_FPEMU=" STR(FALCON_FPEMU)
#pragma message "FALCON_AVX2=" STR(FALCON_AVX2)
```

Figure 3.6 : Modification For Printing The Configuration

After the build and compilation process was complete initiated by the *make* command, *./test_falcon* command was run to initiate the desired tests in. The outputs of the tests are evaluated using the command line outputs seen in the Figure 3.7.



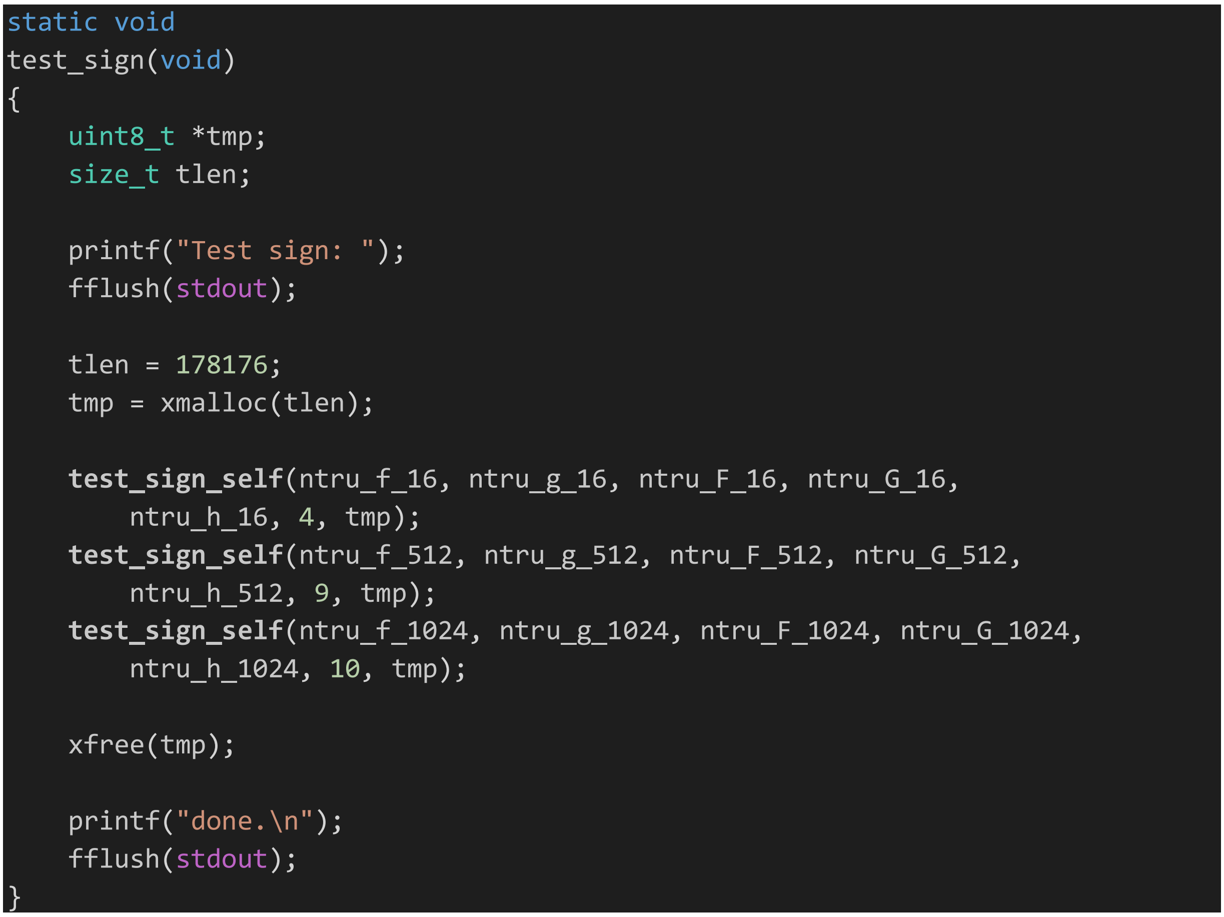
```
radow@DESKTOP-2NHBPGM:~/Falcon-impl-20211101$ ./test_falcon
Test SHAKE256: done.
Test encode/decode: .......... done.
Test verify: .......... .......... .......... done.
Test RNG: done.
Test floating-point (block): ..................................................................................................................... ...........................................................
.... 77cea0ea343b8c1c578af7c9fa3267b6 done.
Test polynomials: [1]..................................................................................................................................................................................
.................................................................................................[2]............................................................................................................................[3]..........
.................................................................................[4]..............................[5]...............[6]........[7]....[8]..[9].[10] done.
Test half-Gaussian sampler: ................. done.
Test sampler: .................... done.
Test sign: 3f 3 57 9c 80 e7 1d ab b7 8 7c 8a de 39 c2 75 de b7 c5 46 14 74 39 ae 12 e0 7d 94 dc 3d 93 4 5c fe 75 ed fb f5 52 7c 43 69 e2 b7 f0 48 6c 6c 8c 3b .. done.
Test keygen: [1]............[2]............[3]............[4]............[5]............[6]............[7]............[8]............[9]............[10]............done.
Test external API: [1]............[2]............[3]............[4]............[5]............[6]............[7]............[8]............[9]............[10]............done.
Test NIST KAT (512): ........................................................................................................... a57400cbaee7109358859a56c735a3cf048a9da2 done.
Test NIST KAT (1024): .......................................................................................................... affdeb3aa83bf9a2039fa9c17d65fd3e3b9828e2 done.
```

Figure 3.7 : Output of the Functional Tests

This program evaluated all major components of the Falcon signature scheme, including SHAKE256 hashing, encoding and decoding functions, signature generation and verification, random number generation, floating-point computations, polynomial multiplication and transforms, Gaussian sampling, key generation, external application interface, and NIST Known Answer Tests (KATs) for both the n=512 and n=1024 parameter lenghts. The output confirmed that all tests were passed successfully, validating the reference implementation.

3.3 Signature Generation On Computer

Among the functional tests of the Falcon, test_sign() includes the tests for the scope of this project. Code of the test can be seen in the Figure 3.8. It tests the signing procedure with 3 different parameters. Inside test_sign function, the test_sign_self function performs the actual test. It takes the four integer polynomials that make up the Falcon private key and the public key polynomial to generate and verify signatures for deterministically generated pseudo-random test messages. It also takes a temporary buffer and a parameter length as highlighted in the Figure 3.6.

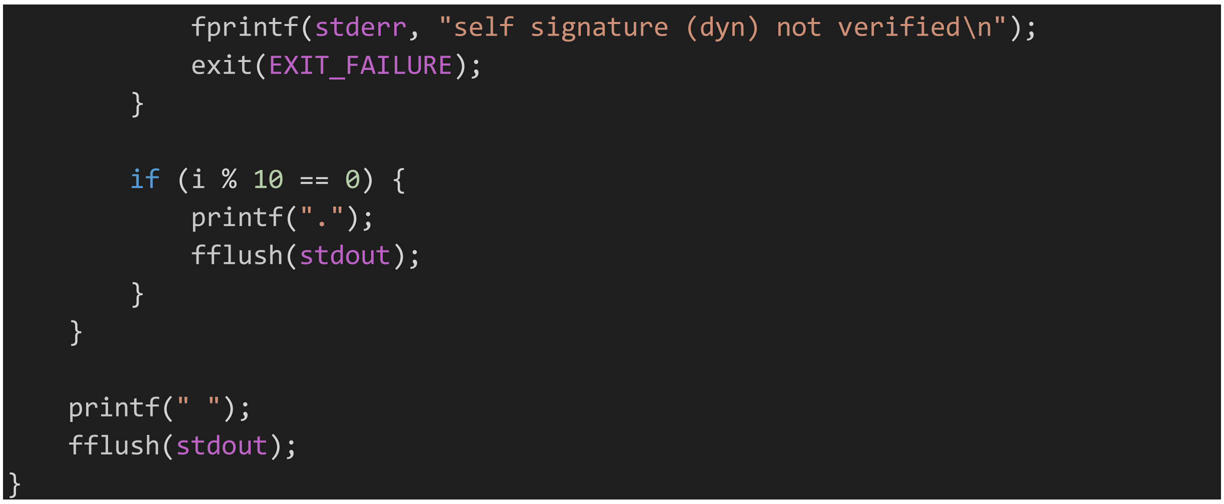


```
static void
test_sign(void)
{
    uint8_t *tmp;
    size_t tlen;

    printf("Test sign: ");
    fflush(stdout);

    tlen = 178176;
    tmp = xmalloc(tlen);

    test_sign_self(ntru_f_16, ntru_g_16, ntru_F_16, ntru_G_16,
        ntru_h_16, 4, tmp);
    test_sign_self(ntru_f_512, ntru_g_512, ntru_F_512, ntru_G_512,
        ntru_h_512, 9, tmp);
    test_sign_self(ntru_f_1024, ntru_g_1024, ntru_F_1024, ntru_G_1024,
        ntru_h_1024, 10, tmp);

    xfree(tmp);

    printf("done.\n");
    fflush(stdout);
}
```

Figure 3.8 : Code of the test_sign()

As can be seen from the code in the Figure 3.9, the test_sign_self function function runs two consecutive loops of 100 iterations each: in the first loop, signatures are produced using the dynamic signing method, which computes all necessary structures on the fly; in the second loop, signatures are produced using the tree-based signing method, which relies on a precomputed and expanded private key for faster execution. In both cases, each signature is immediately verified for correctness and consistency between the two approaches.

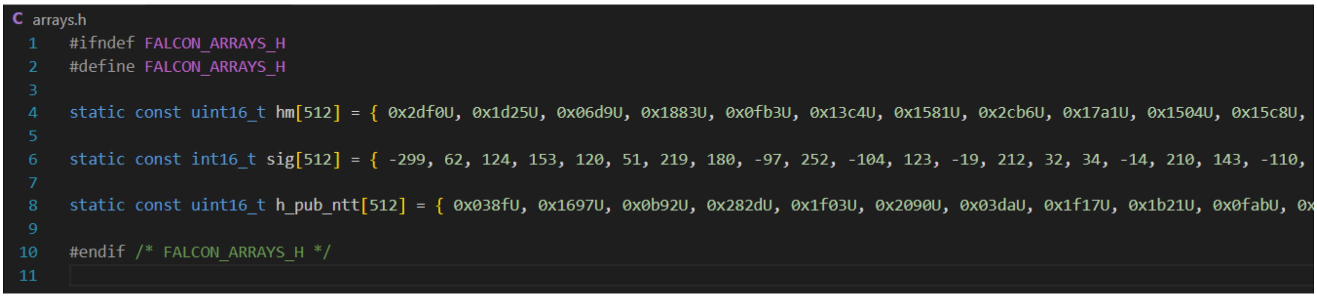
```c
static void
test_sign_self(const int8_t *f, const int8_t *g,
    const int8_t *F, const int8_t *G, const uint16_t *h_src,
    unsigned logn, uint8_t *tmp)
{
    int i;
    size_t n;
    inner_shake256_context rng;
    char buf[20];
    uint16_t *h, *hm, *hm2;
    int16_t *sig;
    uint8_t *tt;
    fpr *expanded_key;

    n = (size_t)1 << logn;
    h = (uint16_t *)tmp;
    hm = h + n;
    sig = (int16_t *)(hm + n);
    hm2 = (uint16_t *)sig;
    tt = (uint8_t *)(sig + n);
    if (logn == 1) {
        tt += 4;
    }

    memcpy(h, h_src, n * sizeof *h);
    Zf(to_ntt_monty)(h, logn);

    /* sprintf(buf, "sign %u", logn); */
    memcpy(buf, "sign 0", 7);
    buf[5] = '0' + logn;

    inner_shake256_init(&rng);
    inner_shake256_inject(&rng, (uint8_t *)buf, strlen(buf));
    inner_shake256_flip(&rng);
    for (i = 0; i < 100; i ++) {
        uint8_t msg[50];  /* nonce + plain */
        inner_shake256_context sc, sc2;
        size_t u;

        inner_shake256_extract(&rng, msg, sizeof msg);

        inner_shake256_init(&sc);
        inner_shake256_inject(&sc, msg, sizeof msg);
        inner_shake256_flip(&sc);
        sc2 = sc;
        Zf(hash_to_point_vartime)(&sc, hm, logn);
        Zf(hash_to_point_ct)(&sc2, hm2, logn, tt);
        for (u = 0; u < n; u ++) {
            if (hm2[u] != hm[u]) {
                fprintf(stderr, "hash_to_point() mismatch\n");
                exit(EXIT_FAILURE);
            }
        }
        Zf(sign_dyn)(sig, &rng, f, g, F, G, hm, logn, tt);
        if (!Zf(verify_raw)(hm, sig, h, logn, tt)) {
            fprintf(stderr, "self signature (dyn) not verified\n");
            exit(EXIT_FAILURE);
        }

        if (i % 10 == 0) {
            printf(".");
            fflush(stdout);
        }
    }

    expanded_key = (fpr *)tt;
    tt = (uint8_t *)expanded_key + (8 * logn + 40) * n;
    Zf(expand_privkey)(expanded_key, f, g, F, G, logn, tt);

    for (i = 0; i < 100; i ++) {
        uint8_t msg[50];  /* nonce + plain */
        inner_shake256_context sc;

        inner_shake256_extract(&rng, msg, sizeof msg);

        inner_shake256_init(&sc);
        inner_shake256_inject(&sc, msg, sizeof msg);
        inner_shake256_flip(&sc);
        Zf(hash_to_point_vartime)(&sc, hm, logn);
        Zf(sign_tree)(sig, &rng, expanded_key, hm, logn, tt);

        if (!Zf(verify_raw)(hm, sig, h, logn, tt)) {
```



```
            fprintf(stderr, "self signature (dyn) not verified\n");
            exit(EXIT_FAILURE);
        }

        if (i % 10 == 0) {
            printf(".");
            fflush(stdout);
        }
    }

    printf(" ");
    fflush(stdout);
}
```

Figure 3.9 : Verification of the Signatures Inside the Tests

For the purpose of testing the MicroblazeV with and without the hardware accelerator, a signature along with the associated message and the public key is extracted from the code as seen in Figure 3.10. This data will be used to verify the signature on the FPGA implementation of the MicroblazeV, to ensure that the verification process functions correctly in the MicroblazeV hardware environment.

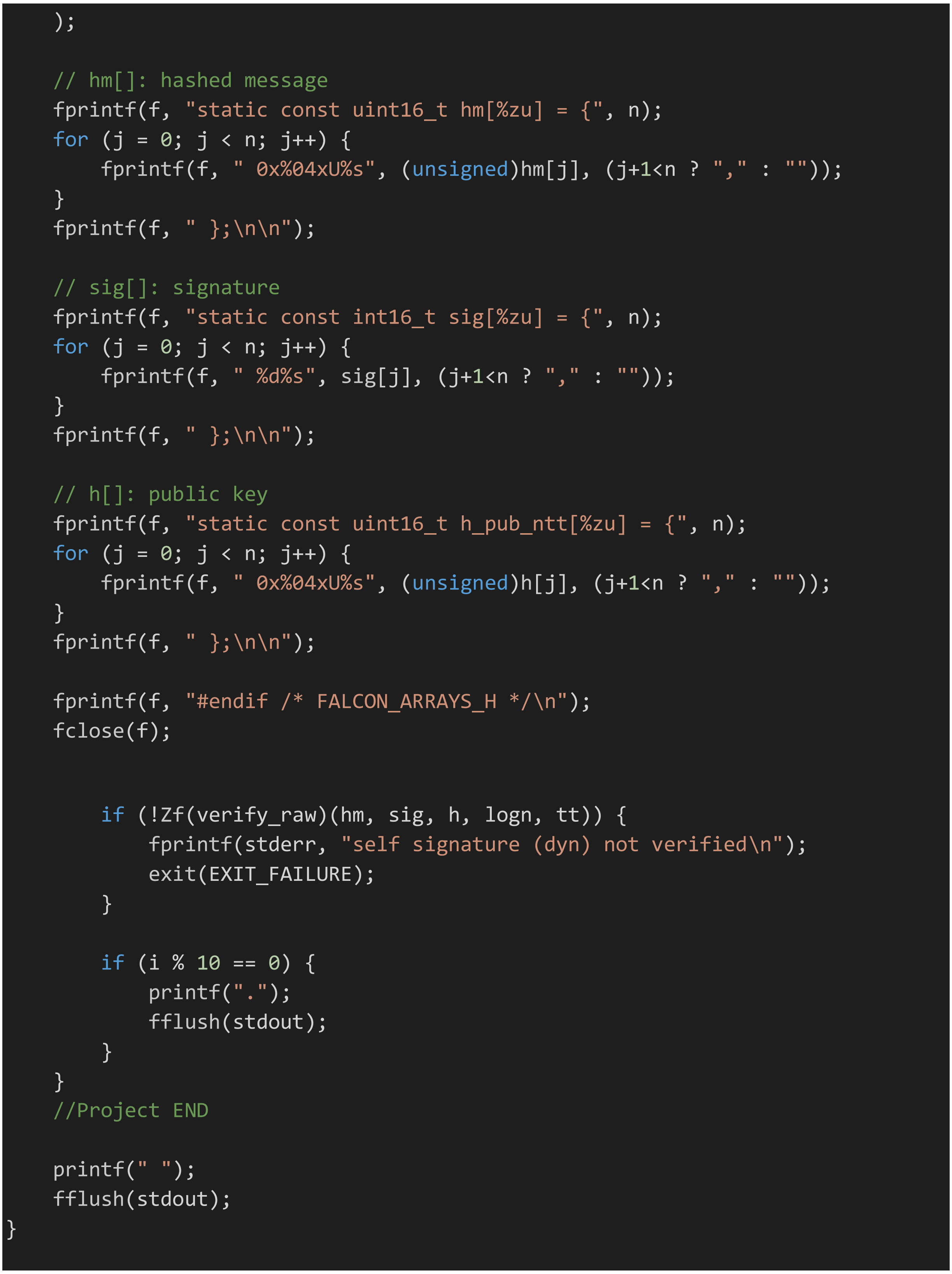


```
#ifndef FALCON_ARRAYS_H
#define FALCON_ARRAYS_H

static const uint16_t hm[512] = { 0x2df0U, 0x1d25U, 0x06d9U, 0x1883U, 0x0fb3U, 0x13c4U, 0x1581U, 0x2cb6U, 0x17a1U, 0x1504U, 0x15c8U,

static const int16_t sig[512] = { -299, 62, 124, 153, 120, 51, 219, 180, -97, 252, -104, 123, -19, 212, 32, 34, -14, 210, 143, -110,

static const uint16_t h_pub_ntt[512] = { 0x038fU, 0x1697U, 0x0b92U, 0x282dU, 0x1f03U, 0x2090U, 0x03daU, 0x1f17U, 0x1b21U, 0x0fabU, 0x

#endif /* FALCON_ARRAYS_H */
```

Figure 3.10: The Extracted Message, Signature and The Public Key

To extract the necessary data, changes seen in the Figure 3.11 has made. Specifically, the iteration count is reduced to one and a block of code that prints out the data is placed between the signing procedure and verification procedure. Thus a header file is printed with the desired data. After the creation of the arrays, the verify_raw function was run on the computer with the arrays to verify that the extracted data from the signing procedure is indeed the desired data. Copy pasting this arrays into the C code of MicroblazeV and using the verify_raw function, signature that is generated on the computer can be verified by our MicroblazeV based environment.

```
static void test_sign_self(const int8_t *f, const int8_t *g,
    const int8_t *F, const int8_t *G, const uint16_t *h_src,
    unsigned logn, uint8_t *tmp)
{
    int i;
    size_t n;
    inner_shake256_context rng;
    char buf[20];
    uint16_t *h, *hm, *hm2;
    int16_t *sig;
    uint8_t *tt;
    fpr *expanded_key;

    n = (size_t)1 << logn;
    h = (uint16_t *)tmp;
    hm = h + n;
    sig = (int16_t *)(hm + n);
    hm2 = (uint16_t *)sig;
    tt = (uint8_t *)(sig + n);
    if (logn == 1) {
        tt += 4;
    }
    memcpy(h, h_src, n * sizeof *h);
    Zf(to_ntt_monty)(h, logn);

    /* sprintf(buf, "sign %u", logn); */
    memcpy(buf, "sign 0", 7);
    buf[5] = '0' + logn;

    inner_shake256_init(&rng);
    inner_shake256_inject(&rng, (uint8_t *)buf, strlen(buf));
    inner_shake256_flip(&rng);
    for (i = 0; i < 1/*00*/; i ++) {
        uint8_t msg[50];  /* nonce + plain */
        inner_shake256_context sc, sc2;
        size_t u;

        inner_shake256_extract(&rng, msg, sizeof msg);

        inner_shake256_init(&sc);
        inner_shake256_inject(&sc, msg, sizeof msg);
        inner_shake256_flip(&sc);
        sc2 = sc;
        Zf(hash_to_point_vartime)(&sc, hm, logn);
        Zf(hash_to_point_ct)(&sc2, hm2, logn, tt);
    for (u = 0; u < n; u ++) {
        if (hm2[u] != hm[u]) {
            fprintf(stderr, "hash_to_point() mismatch\n");
            exit(EXIT_FAILURE);
        }
    }
    Zf(sign_dyn)(sig, &rng, f, g, F, G, hm, logn, tt);
    if (!Zf(verify_raw)(hm, sig, h, logn, tt)) {
        fprintf(stderr, "self signature (dyn) not verified\n");
        exit(EXIT_FAILURE);
    }

    if (i % 10 == 0) {
        printf(".");
        fflush(stdout);
    }
}

expanded_key = (fpr *)tt;
tt = (uint8_t *)expanded_key + (8 * logn + 40) * n;
Zf(expand_privkey)(expanded_key, f, g, F, G, logn, tt);

for (i = 0; i < 1/*00*/; i ++) {
    uint8_t msg[50];  /* nonce + plain */
    inner_shake256_context sc;

    inner_shake256_extract(&rng, msg, sizeof msg);

    inner_shake256_init(&sc);
    inner_shake256_inject(&sc, msg, sizeof msg);
    inner_shake256_flip(&sc);
    Zf(hash_to_point_vartime)(&sc, hm, logn);
    Zf(sign_tree)(sig, &rng, expanded_key, hm, logn, tt);
//Project
size_t n = (size_t)1 << logn;
size_t j;
FILE *f = fopen("arrays.h", "w");
if (!f) {
    perror("fopen");
    exit(1);
}

//  Guard to avoid double-include
fprintf(f,
    "#ifndef FALCON_ARRAYS_H\n"
    "#define FALCON_ARRAYS_H\n\n"
```



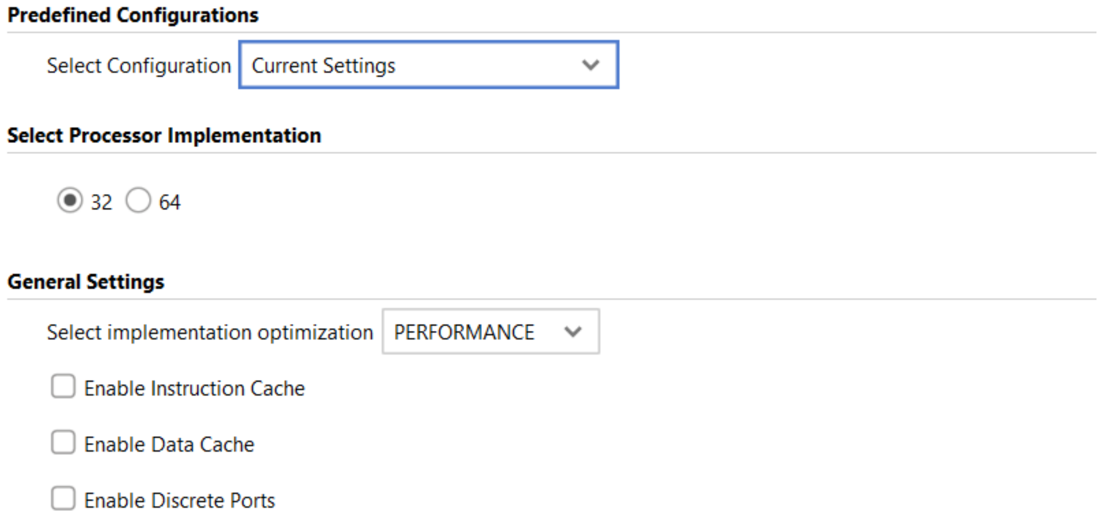
```
    );

    // hm[]: hashed message
    fprintf(f, "static const uint16_t hm[%zu] = {", n);
    for (j = 0; j < n; j++) {
        fprintf(f, " 0x%04xU%s", (unsigned)hm[j], (j+1<n ? "," : ""));
    }
    fprintf(f, " };\n\n");

    // sig[]: signature
    fprintf(f, "static const int16_t sig[%zu] = {", n);
    for (j = 0; j < n; j++) {
        fprintf(f, " %d%s", sig[j], (j+1<n ? "," : ""));
    }
    fprintf(f, " };\n\n");

    // h[]: public key
    fprintf(f, "static const uint16_t h_pub_ntt[%zu] = {", n);
    for (j = 0; j < n; j++) {
        fprintf(f, " 0x%04xU%s", (unsigned)h[j], (j+1<n ? "," : ""));
    }
    fprintf(f, " };\n\n");

    fprintf(f, "#endif /* FALCON_ARRAYS_H */\n");
    fclose(f);


        if (!Zf(verify_raw)(hm, sig, h, logn, tt)) {
            fprintf(stderr, "self signature (dyn) not verified\n");
            exit(EXIT_FAILURE);
        }

        if (i % 10 == 0) {
            printf(".");
            fflush(stdout);
        }
    }
    //Project END

    printf(" ");
    fflush(stdout);
}
```

Figure 3.11 : Changes to the Test Code

4. IMPLEMENTATION OF THE FALCON ON MICROBLAZE-V

4.1 Preaparing the Block Diagram

MicroblazeV was chosen as the processor for the creation of the system on the FPGA. It is chosen as it is a configurable and modular processor. It also is integrated to the Vitis and Vivado ecosystem. Thus it is a suitable choice for embedded applications including our own project. We have configured it to be 32 bits performance mode as seen in the Figure 4.1



Predefined Configurations

Select Configuration: Current Settings

Select Processor Implementation

● 32 ○ 64

General Settings

Select implementation optimization: PERFORMANCE

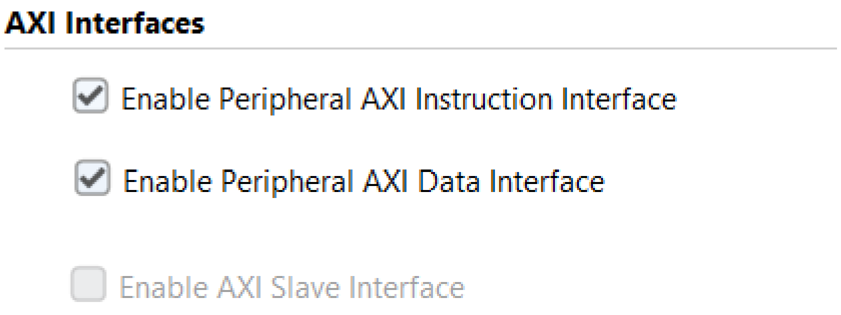
☐ Enable Instruction Cache

☐ Enable Data Cache

☐ Enable Discrete Ports

Figure 4.1 : Configuration of the MicroblazeV

It’s instruction set is configured to have Code Compression and Multiplication units along with the Zbb bit manipulation extention. Base counters and timers are also enabled for mcycle csr register to measure performance. AXI bus for both instruction and data is enabled to use on board DDR2 memory as in the Figure 4.2. Instruction set configuration can be seen in the Figure 4.3



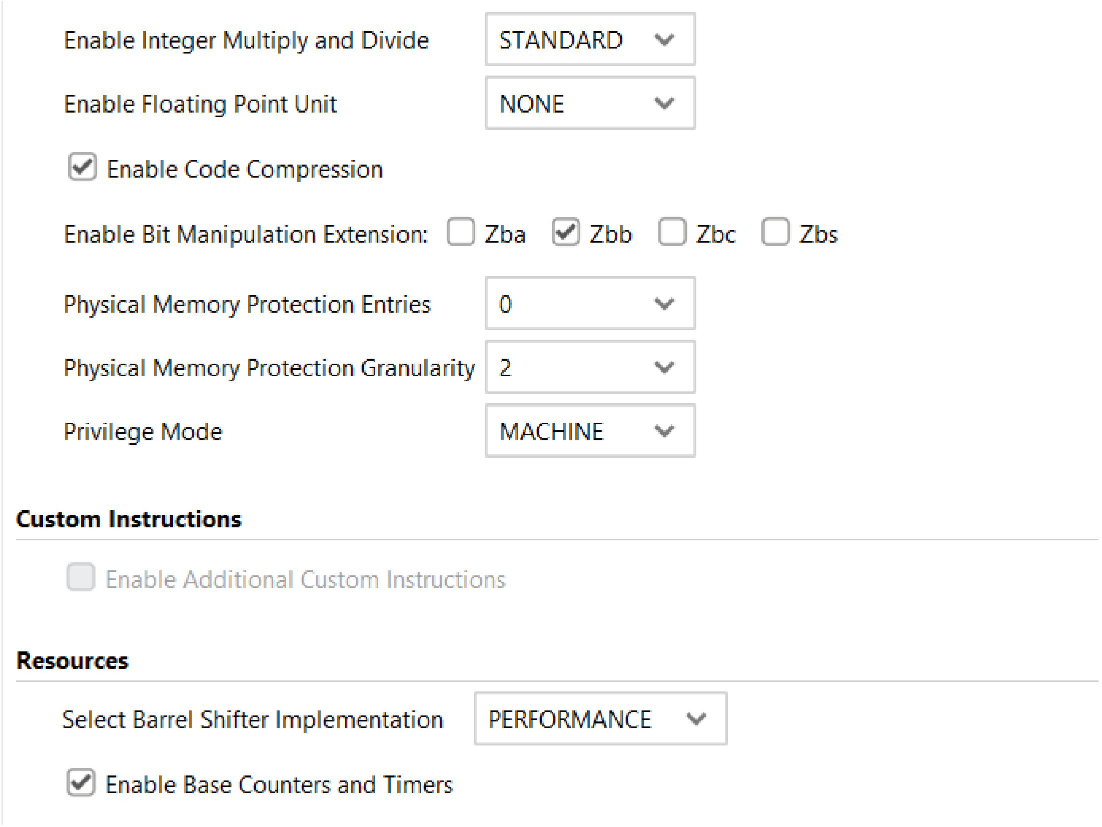
AXI Interfaces

☑ Enable Peripheral AXI Instruction Interface

☑ Enable Peripheral AXI Data Interface

☐ Enable AXI Slave Interface

Figure 4.2 : AXI configuration of the MicroblazeV



Enable Integer Multiply and Divide: STANDARD

Enable Floating Point Unit: NONE

☑ Enable Code Compression

Enable Bit Manipulation Extension: ☐ Zba ☑ Zbb ☐ Zbc ☐ Zbs

Physical Memory Protection Entries: 0

Physical Memory Protection Granularity: 2

Privilege Mode: MACHINE

Custom Instructions

☐ Enable Additional Custom Instructions

Resources

Select Barrel Shifter Implementation: PERFORMANCE

☑ Enable Base Counters and Timers

Figure 4.3 : Configuration of the Instruction Set

For using the on board DDR2 in the NEXYS4 DDR FPGA board, Memory Generation IP was added to the block diagram with the default configurations that comes with the board from the drag and drop menu on the block diagram creation as seen in the Figure 4.4.

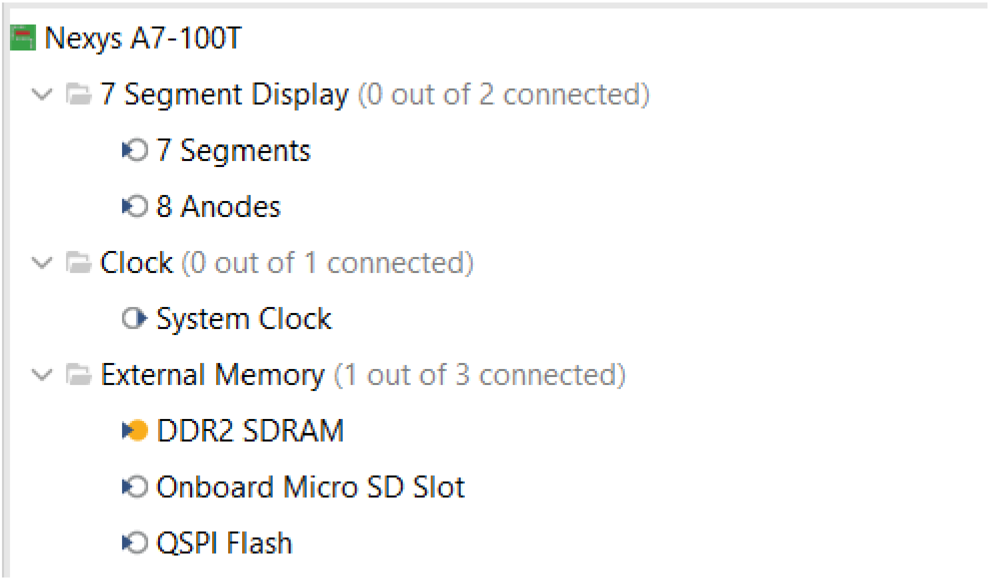

Nexys A7-100T
- 7 Segment Display (0 out of 2 connected)
  - 7 Segments
  - 8 Anodes
- Clock (0 out of 1 connected)
  - System Clock
- External Memory (1 out of 3 connected)
  - DDR2 SDRAM
  - Onboard Micro SD Slot
  - QSPI Flash

Figure 4.4 : Addition of the DDR2 SDRAM

USB to UART IP was also placed in the same menu to communicate with the MicroblazeV. This communications will be used to verify and validate the Falcon running on the FPGA. A clock wizard and a AXI interconnect is also placed. Both the UART IP and the MIG will communicate with the processor via the interconnect. Main system frequency is chosen as 100Mhz. MIG generates its own clock of 81Mhz from the input system clock. Resulting system diagram is the Figure 4.5.

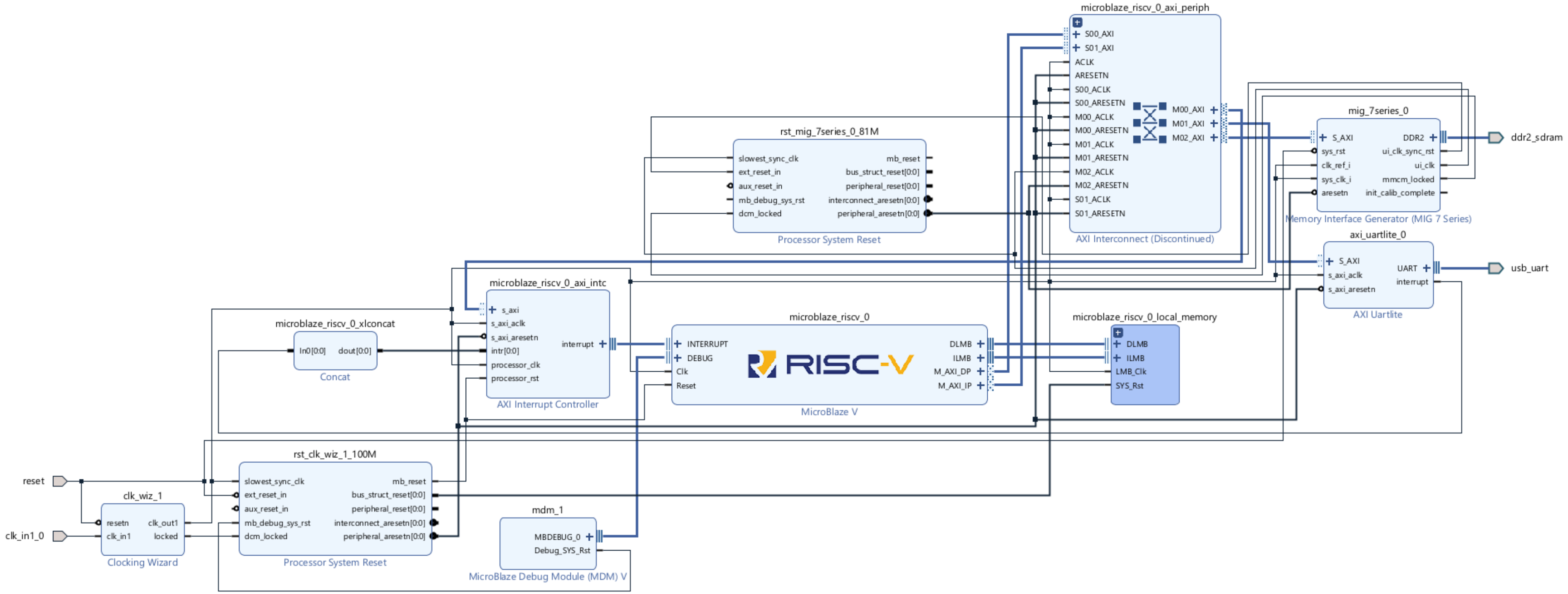
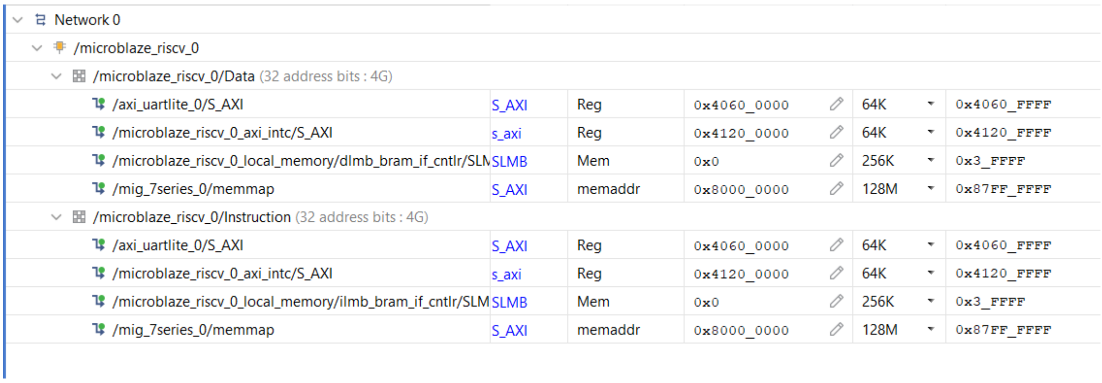


Figure 4.5 : Block Diagram of The MicroblazeV System

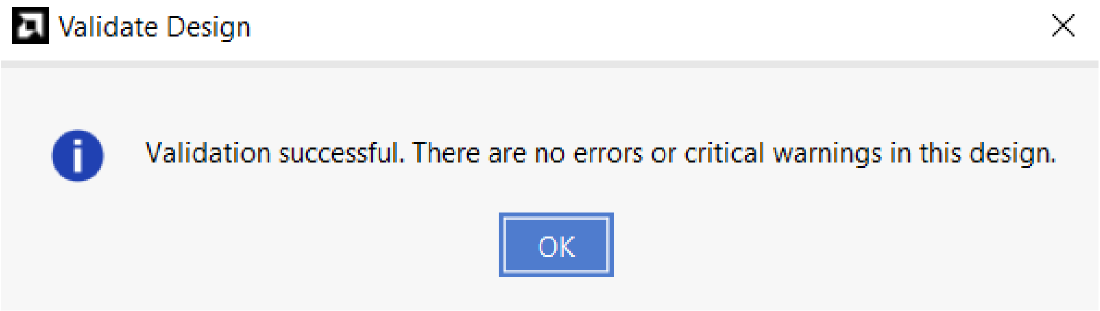
The resulting memory map is as seen in the Figure 4.6.



Network 0					
/microblaze_riscv_0					
/microblaze_riscv_0/Data (32 address bits : 4G)					
/axi_uartlite_0/S_AXI	S_AXI	Reg	0x4060_0000	64K	0x4060_FFFF
/microblaze_riscv_0_axi_intc/S_AXI	s_axi	Reg	0x4120_0000	64K	0x4120_FFFF
/microblaze_riscv_0_local_memory/dlmb_bram_if_cntlr/SLN	SLMB	Mem	0x0	256K	0x3_FFFF
/mig_7series_0/memmap	S_AXI	memaddr	0x8000_0000	128M	0x87FF_FFFF
/microblaze_riscv_0/Instruction (32 address bits : 4G)					
/axi_uartlite_0/S_AXI	S_AXI	Reg	0x4060_0000	64K	0x4060_FFFF
/microblaze_riscv_0_axi_intc/S_AXI	s_axi	Reg	0x4120_0000	64K	0x4120_FFFF
/microblaze_riscv_0_local_memory/ilmb_bram_if_cntlr/SLM	SLMB	Mem	0x0	256K	0x3_FFFF
/mig_7series_0/memmap	S_AXI	memaddr	0x8000_0000	128M	0x87FF_FFFF

Figure 4.6 : Address Map of The MicroblazeV system.

The system is succesfull validated by the block design generator as seen in the Figure 4.7.



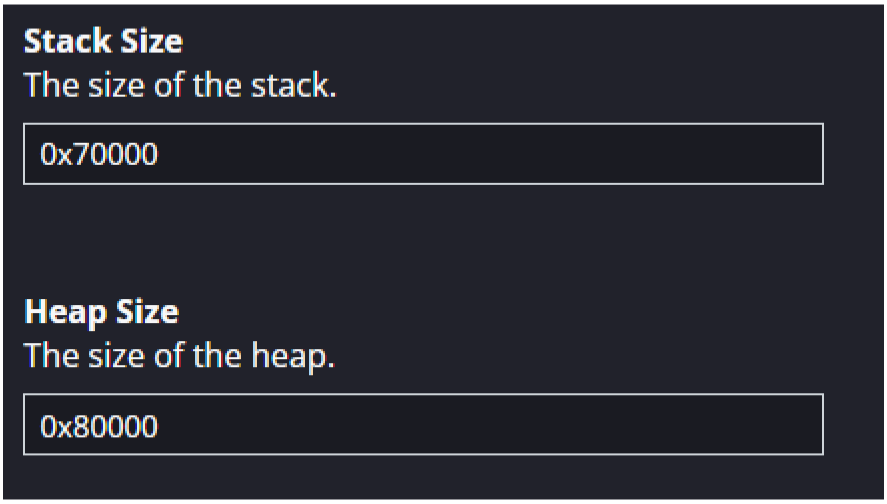
Validate Design

Validation successful. There are no errors or critical warnings in this design.

Figure 4.7 : Validation of the Block Diagram

4.2 Executing the Falcon Software on MicroblazeV

After the design of the MicroblazeV block diagram, the hardware is synthesized for bitstream generation. The generated bitstream is then exported as Hardware for platform generation in Vitis. Falcon refence code that was used in the computer is used in with the necessary configurations that allow floating point emulation in Vitis. FALCON_FPEMU macro is defined as true and FALCON_FPNATIVE macro is not defined. Linker in the Vitis is configured to use DDR2 for all sections of the code including the heap and the stack since Local Memory from the BRAMS in the block diagram is not enough for the entire Falcon. Stack and Heap sizes are configured as in the Figure 4.8.

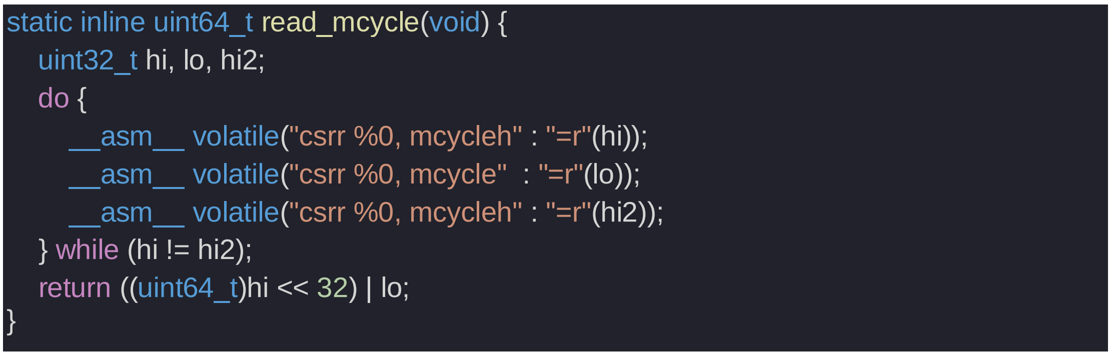


Stack Size
The size of the stack.
0x70000

Heap Size
The size of the heap.
0x80000

Figure 4.8 : Stack and Heap Sizes

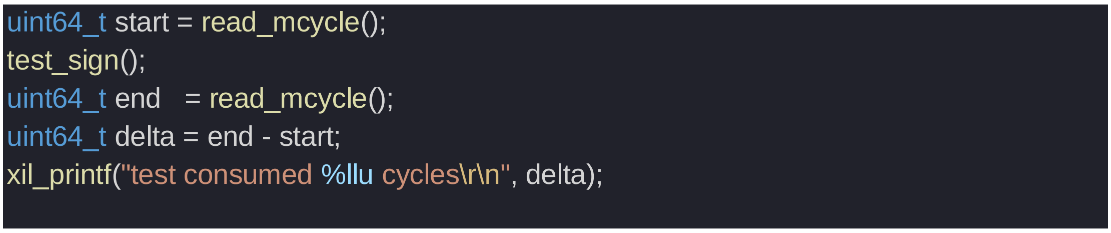
To count the clock cycles and measure the performance mcycle csr register that counts the clock cycles is read, before and after the tested functions. This reading is done by inline assembly function as seen in the Figure 4.9.



```
static inline uint64_t read_mcycle(void) {
    uint32_t hi, lo, hi2;
    do {
        __asm__ volatile("csrr %0, mcycleh" : "=r"(hi));
        __asm__ volatile("csrr %0, mcycle"  : "=r"(lo));
        __asm__ volatile("csrr %0, mcycleh" : "=r"(hi2));
    } while (hi != hi2);
    return ((uint64_t)hi << 32) | lo;
}
```

Figure 4.9 : Inline Assembly Function For Measuring Clock Cycles

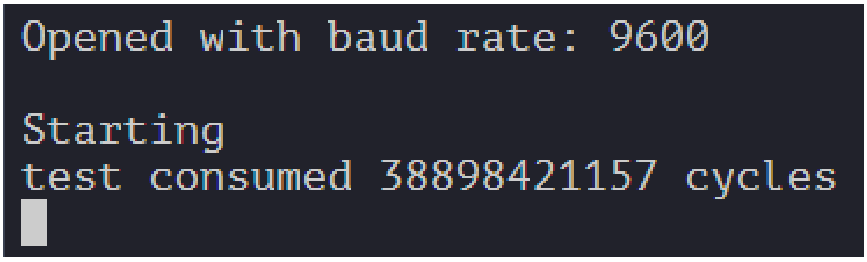
This functions performs reading of the mcycle control status register which is a register that has a lower and higher part. First the high register is read and then the lower register. Finally the higher register is read again to check that the lower register is not overflowed to the higher register since it is first read. If the value last read is different than the first read of the high register this means that when the high register is first read lower register has overflown and thus the value read from the lower register is incorrect. The main is modfied with the following in the Figure 4.10. In test_sign function print statements are commented out for accurate measurement of the speed.



```
uint64_t start = read_mcycle();
test_sign();
uint64_t end   = read_mcycle();
uint64_t delta = end - start;
xil_printf("test consumed %llu cycles\r\n", delta);
```

Figure 4.10 : Code for measuring the Performance of the Hardware System

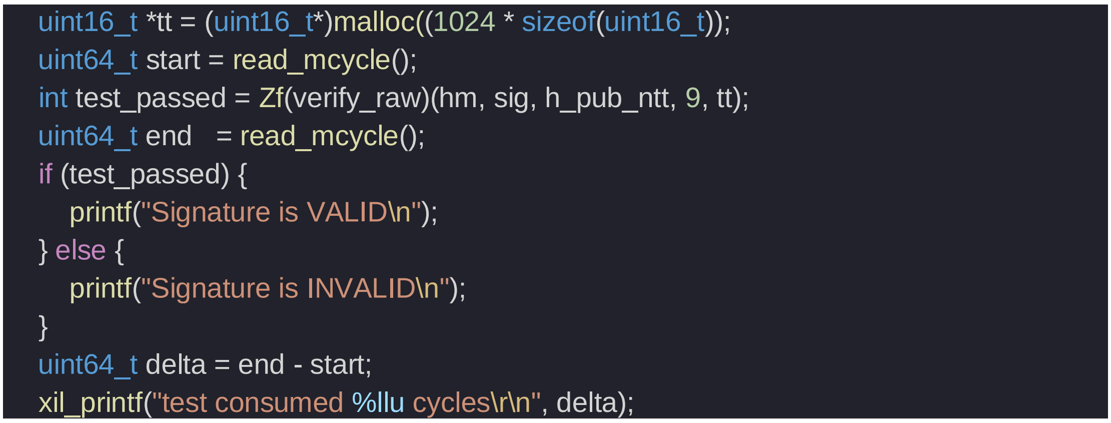
The result of this test can be seen in the Figure 4.11. The results are printed using UART and the Vitis serial monitor.



```
Opened with baud rate: 9600

Starting
test consumed 38898421157 cycles
```

Figure 4.11 : Test Sign

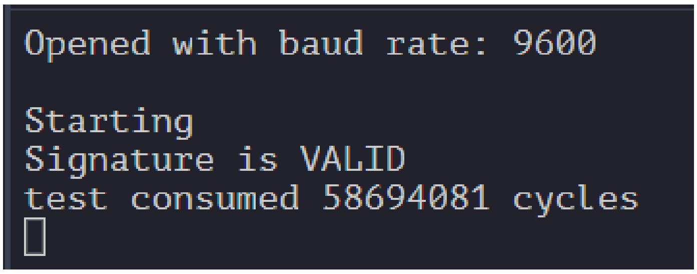
Using the extracted signature, public key and the message from the computer, signature verification process was also run on the MicroblazeV system for validation of the systems capabilities. The Main function was modified for measuring the performance and verification of the signature as seen in the Figure 4.12.



```
uint16_t *tt = (uint16_t*)malloc((1024 * sizeof(uint16_t));
uint64_t start = read_mcycle();
int test_passed = Zf(verify_raw)(hm, sig, h_pub_ntt, 9, tt);
uint64_t end   = read_mcycle();
if (test_passed) {
    printf("Signature is VALID\n");
} else {
    printf("Signature is INVALID\n");
}
uint64_t delta = end - start;
xil_printf("test consumed %llu cycles\r\n", delta);
```

Figure 4.12 : Validation Code of the Extracted Signature in the Hardware

The extracted signature is acurately verified, also validating the created MicroBlazeV environment as seen in the Figure 4.13.



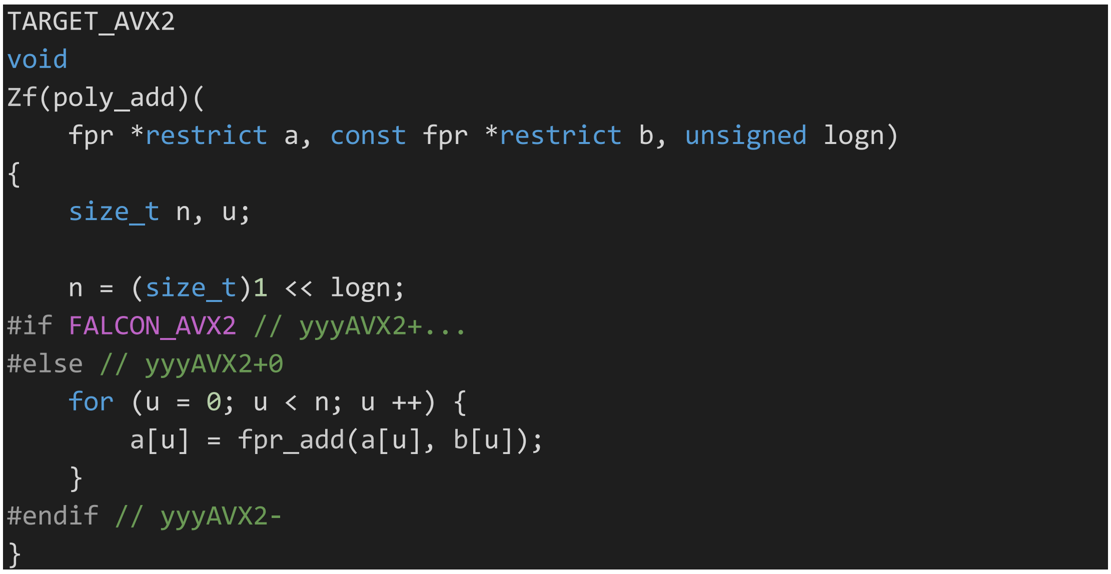
```
Opened with baud rate: 9600

Starting
Signature is VALID
test consumed 58694081 cycles
```

Figure 4.13 : Validation of the Extracted Signature on FPGA

4.3 Design of the Accelerator

Functions that are decided to be accelerated is as follows:

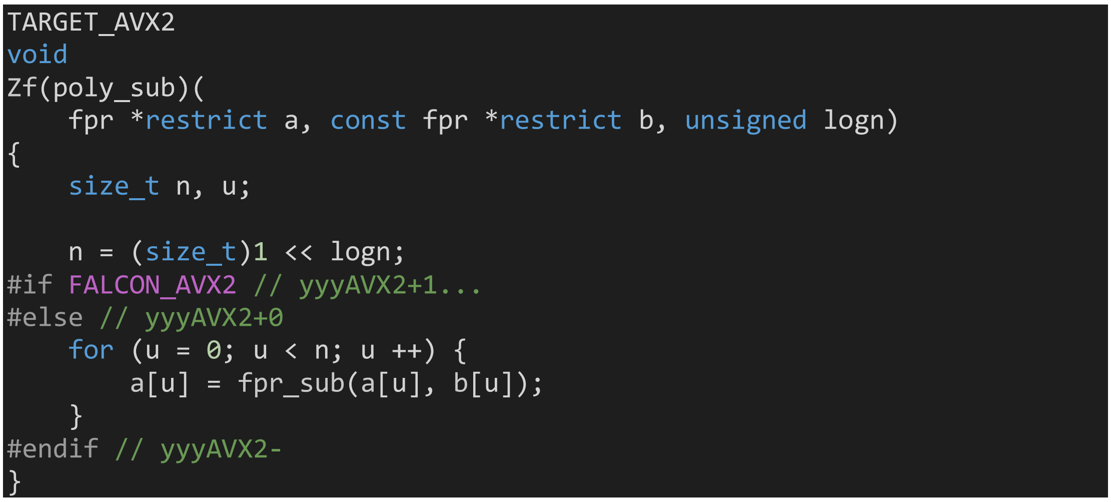
- First function is the poly_add function that performs addition of two arrays as seen in the Figure 4.14.



```
TARGET_AVX2
void
Zf(poly_add)(
    fpr *restrict a, const fpr *restrict b, unsigned logn)
{
    size_t n, u;

    n = (size_t)1 << logn;
#if FALCON_AVX2 // yyyAVX2+...
#else // yyyAVX2+0
    for (u = 0; u < n; u ++) {
        a[u] = fpr_add(a[u], b[u]);
    }
#endif // yyyAVX2-
}
```

Figure 4.14 : Function of poly_add()

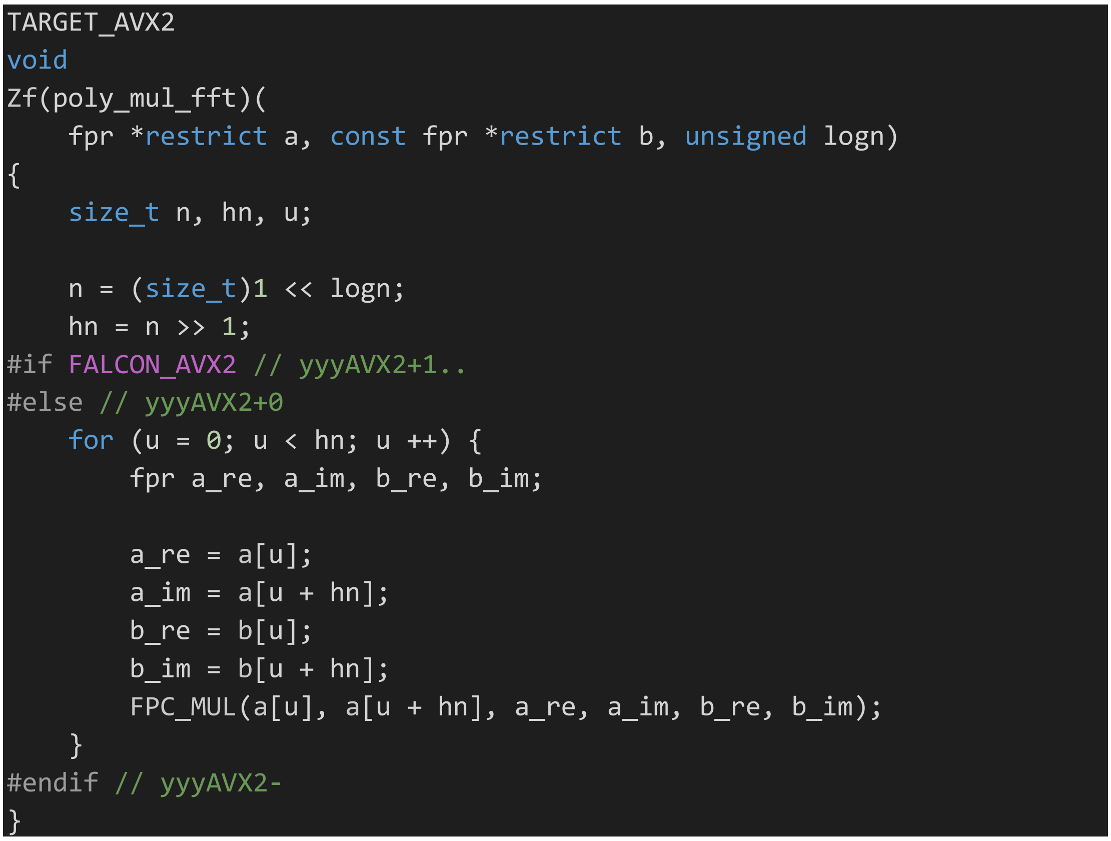
- poly_sub function that performs substraction between two arrays. Code for the function can be seen in the Figure 4.15.



```
TARGET_AVX2
void
Zf(poly_sub)(
    fpr *restrict a, const fpr *restrict b, unsigned logn)
{
    size_t n, u;

    n = (size_t)1 << logn;
#if FALCON_AVX2 // yyyAVX2+1...
#else // yyyAVX2+0
    for (u = 0; u < n; u ++) {
        a[u] = fpr_sub(a[u], b[u]);
    }
#endif // yyyAVX2-
}
```

Figure 4.15 : Function of poly_sub

- The poly_mul_fft function that performs complex multiplication of two arrays where the first half of the array stores the real values while the second half stores the imaginary values as seen in the Figure 4.16. AVX2 optimization parts are not given in the figures better readability.



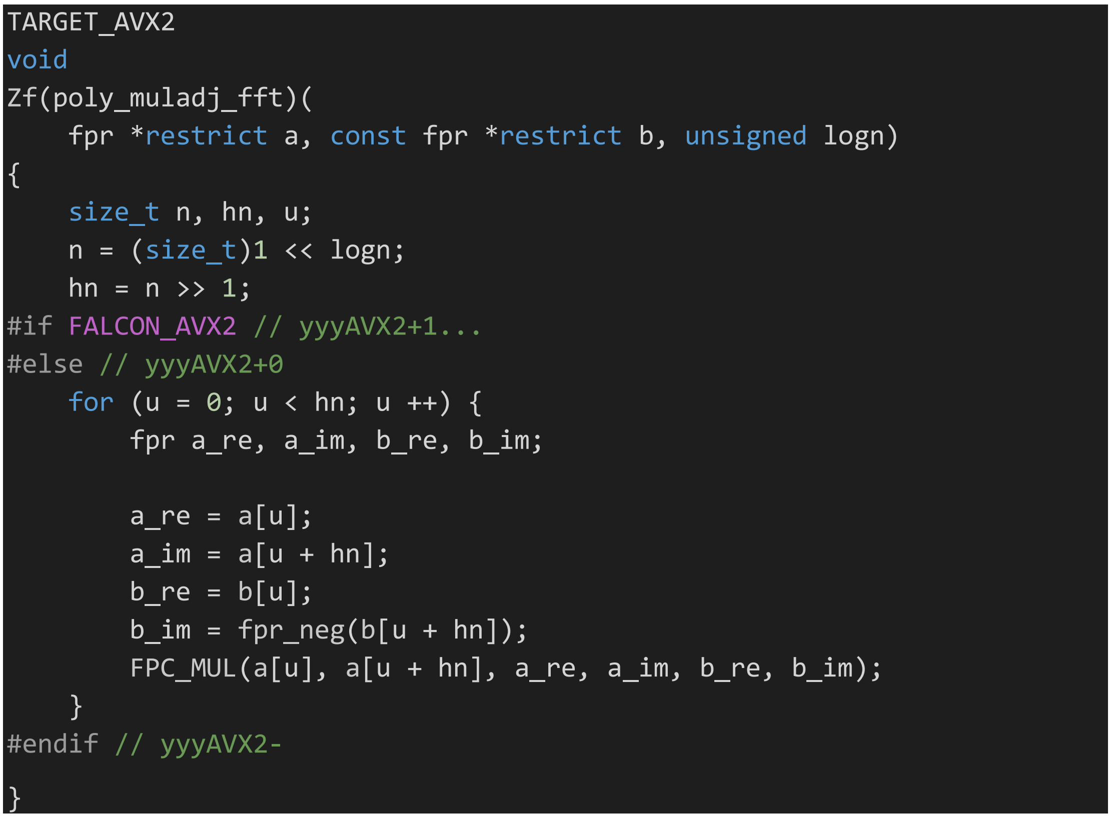
```
TARGET_AVX2
void
Zf(poly_mul_fft)(
    fpr *restrict a, const fpr *restrict b, unsigned logn)
{
    size_t n, hn, u;

    n = (size_t)1 << logn;
    hn = n >> 1;
#if FALCON_AVX2 // yyyAVX2+1..
#else // yyyAVX2+0
    for (u = 0; u < hn; u ++) {
        fpr a_re, a_im, b_re, b_im;

        a_re = a[u];
        a_im = a[u + hn];
        b_re = b[u];
        b_im = b[u + hn];
        FPC_MUL(a[u], a[u + hn], a_re, a_im, b_re, b_im);
    }
#endif // yyyAVX2-
}
```

Figure 4.16 : Code for the poly_mul_fft Function

- The poly_muladj_fft function that performs complex multiplication of one array with the complex conjugate of the other. Order of real and imaginary parts are same as the above function. Code of the function is in Figure 4.17.

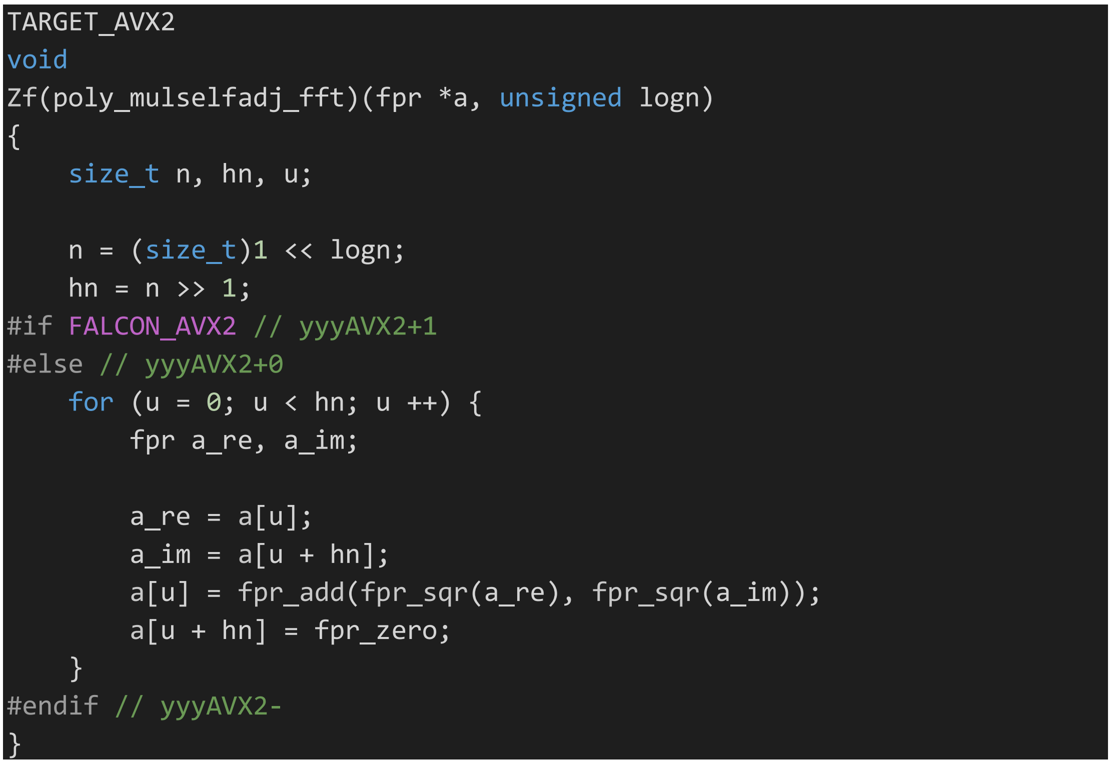
- The poly_mulselfadj_fft function which computes the magnitude of the given complex array as shown in the Figure 4.18.

- The poly_mulconst_fft function that scales a vector by a constant, as displayed in Figure 4.19.



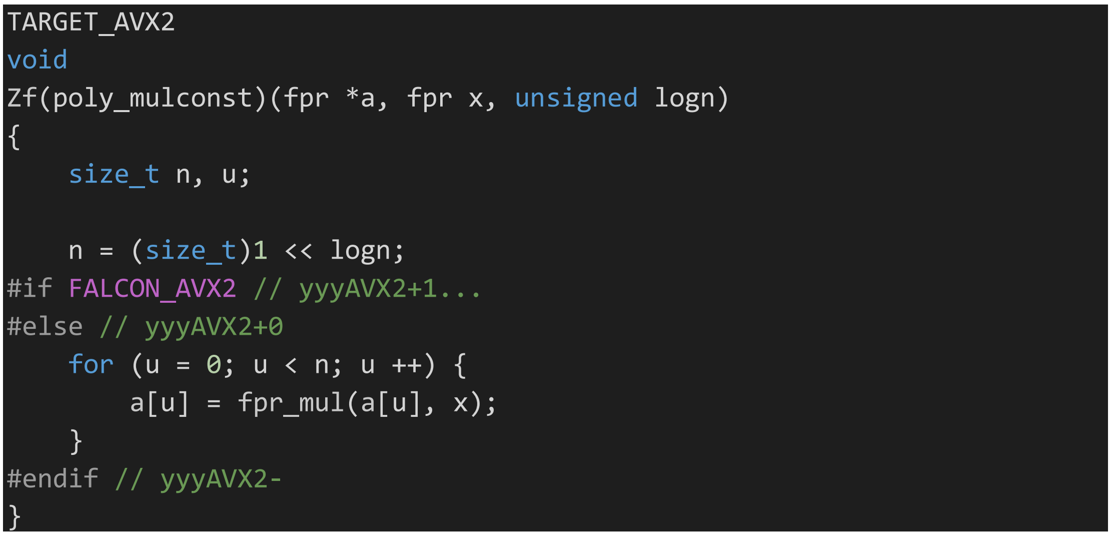
```
TARGET_AVX2
void
Zf(poly_muladj_fft)(
    fpr *restrict a, const fpr *restrict b, unsigned logn)
{
    size_t n, hn, u;
    n = (size_t)1 << logn;
    hn = n >> 1;
#if FALCON_AVX2 // yyyAVX2+1...
#else // yyyAVX2+0
    for (u = 0; u < hn; u ++) {
        fpr a_re, a_im, b_re, b_im;

        a_re = a[u];
        a_im = a[u + hn];
        b_re = b[u];
        b_im = fpr_neg(b[u + hn]);
        FPC_MUL(a[u], a[u + hn], a_re, a_im, b_re, b_im);
    }
#endif // yyyAVX2-
}
```

Figure 4.17 : Code for the poly_muladj_fft Function



```
TARGET_AVX2
void
Zf(poly_mulselfadj_fft)(fpr *a, unsigned logn)
{
    size_t n, hn, u;

    n = (size_t)1 << logn;
    hn = n >> 1;
#if FALCON_AVX2 // yyyAVX2+1
#else // yyyAVX2+0
    for (u = 0; u < hn; u ++) {
        fpr a_re, a_im;

        a_re = a[u];
        a_im = a[u + hn];
        a[u] = fpr_add(fpr_sqr(a_re), fpr_sqr(a_im));
        a[u + hn] = fpr_zero;
    }
#endif // yyyAVX2-
}
```

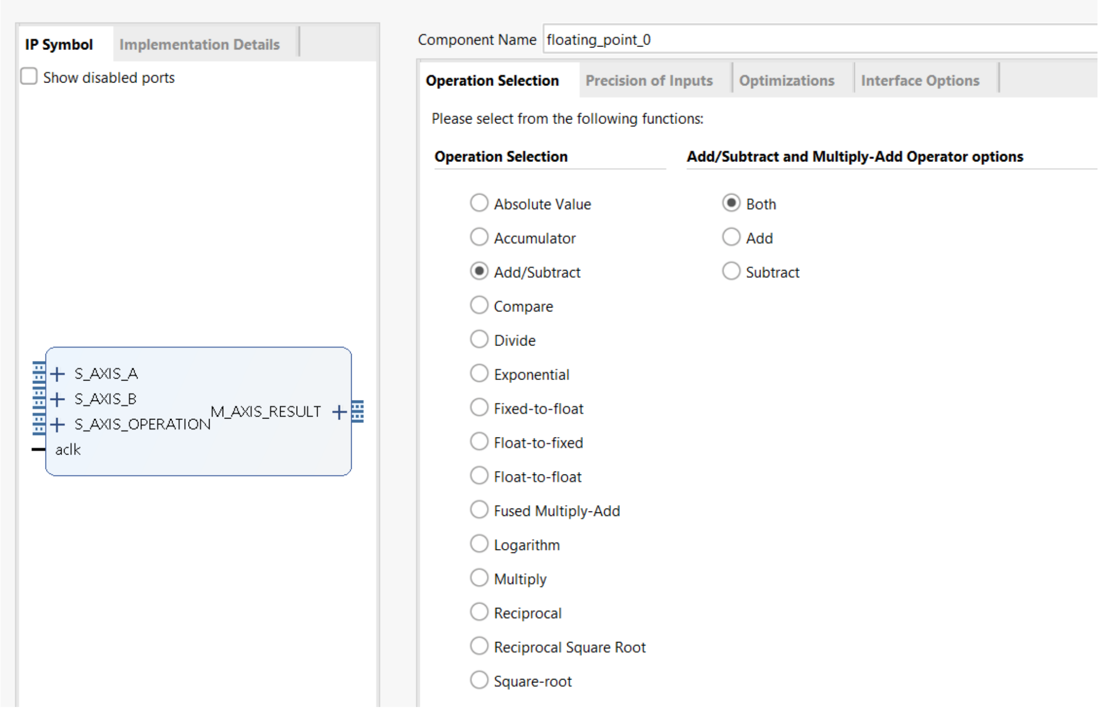
Figure 4.18 : Code of the poly_mulselfadj_fft Function



```
TARGET_AVX2
void
Zf(poly_mulconst)(fpr *a, fpr x, unsigned logn)
{
    size_t n, u;

    n = (size_t)1 << logn;
#if FALCON_AVX2 // yyyAVX2+1...
#else // yyyAVX2+0
    for (u = 0; u < n; u ++) {
        a[u] = fpr_mul(a[u], x);
    }
#endif // yyyAVX2-
}
```

Figure 4.19 : Code of the poly_mulconst Function

To accelerate these functions, designing one vector adder and one vector multiplier was decided. The vector units will be connected to the MicroblazeV via the AXI bus. Vivado Floating Point IP is decided to be used as the functional units. The floating point IP has uses AXI-Stream for inputs and outputs and can be configured for variety of functions and precisions as shown in Figure 4.20.



Component Name floating_point_0

Please select from the following functions:

Operation Selection

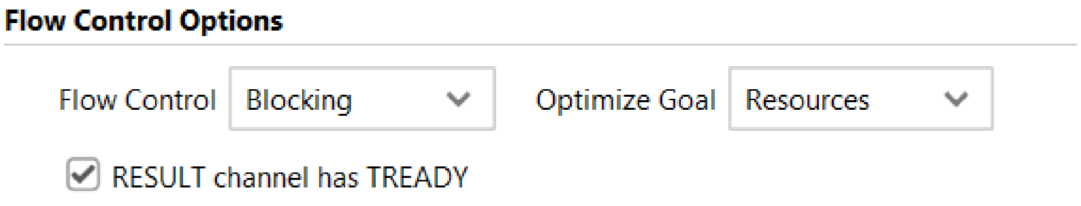
○ Absolute Value
○ Accumulator
● Add/Subtract
○ Compare
○ Divide
○ Exponential
○ Fixed-to-float
○ Float-to-fixed
○ Float-to-float
○ Fused Multiply-Add
○ Logarithm
○ Multiply
○ Reciprocal
○ Reciprocal Square Root
○ Square-root

Add/Subtract and Multiply-Add Operator options

● Both
○ Add
○ Subtract

Figure 4.20 : Xilinx Floating Point IP

Two instences of this IP is used in the accelerator and they are configured either as multiplier or adder for double percision. Optimization configuration is selected as high speed and full DSP usage for maximum frequency. In addition AXI-Stream configuration on the allows blocking usage and TLAST signal. Thus inputs will only be accepted when both inputs are valid and TLAST signal will be propagated alongside the computation. Interface configurations can be seen in the Figures 4.21 and 4.22.



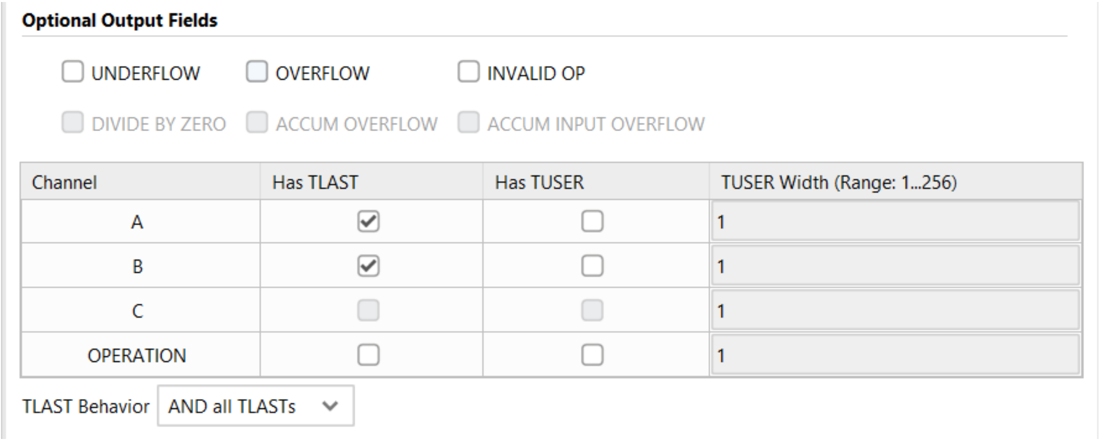
Flow Control Options

Flow Control: Blocking

Optimize Goal: Resources

☑ RESULT channel has TREADY

Figure 4.21 : Flow Control Option



Optional Output Fields

☐ UNDERFLOW ☐ OVERFLOW ☐ INVALID OP

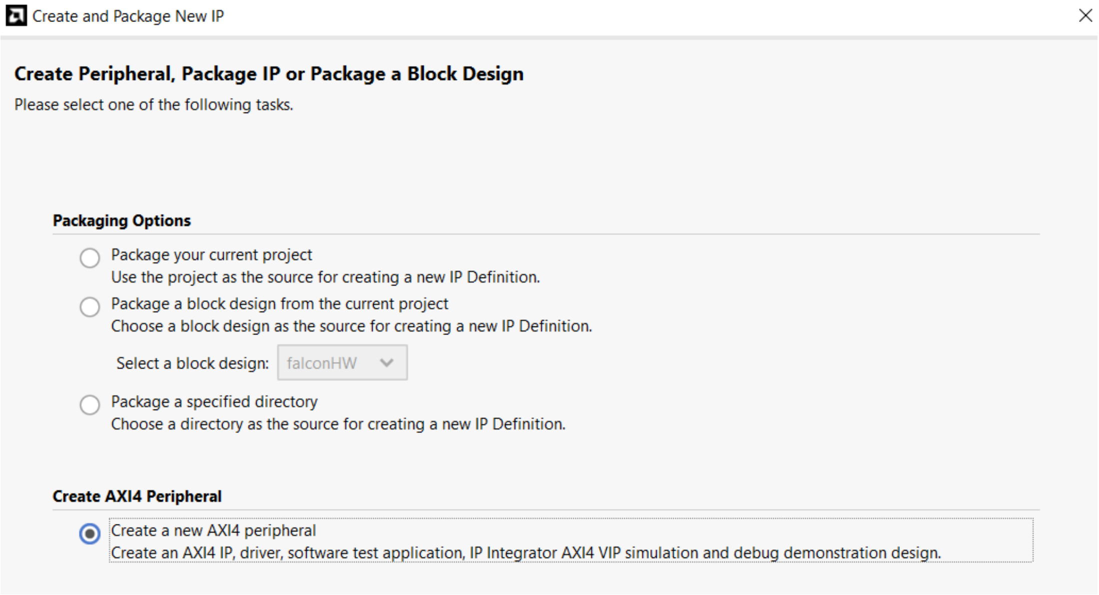
☐ DIVIDE BY ZERO ☐ ACCUM OVERFLOW ☐ ACCUM INPUT OVERFLOW

Channel	Has TLAST	Has TUSER	TUSER Width (Range: 1...256)
A	☑	☐	1
B	☑	☐	1
C	☐	☐	1
OPERATION	☐	☐	1

TLAST Behavior: AND all TLASTs

Figure 4.22 : TLAST Signal Configuration

To initiate the process of receiving and storing data a module with one AXI4 slave, two AXI-Stream master and one AXI-Stream slave is designed. The module recieves data from the MicroblazeV with the AXI4 port, stores the data and then sends the data to floating point IP using the master stream ports. When the computation is finished, the result from the slave stream port is written to the memory for MicroblazeV to read using the AXI port. The data that needs to be stored is two arrays of doubles that has a maximum length of 1024. Thus for each array 8192 bytes are needed. Thus 16384 bytes of memory is needed for storing the arrays and one more 8192 byte memory for storing the result of the computation. Three additional registers are also needed to operate and monitoring, Status, Control and Size registers. This module is called VVD2 in our design. To design this module a new AXI4 peripheral IP is created in the Vivado with the necessary interfaces as seen in the Figure 4.23 and 4.24.


Create and Package New IP

Create Peripheral, Package IP or Package a Block Design

Please select one of the following tasks.

Packaging Options

○ Package your current project
Use the project as the source for creating a new IP Definition.

○ Package a block design from the current project
Choose a block design as the source for creating a new IP Definition.

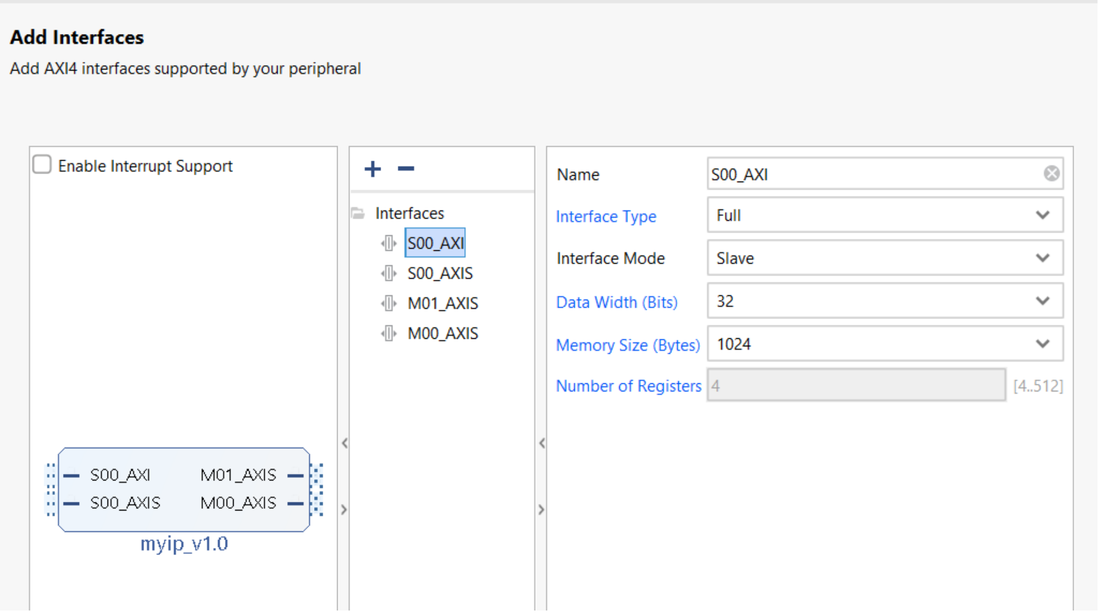
Select a block design: falconHW

○ Package a specified directory
Choose a directory as the source for creating a new IP Definition.

Create AXI4 Peripheral

● Create a new AXI4 peripheral
Create an AXI4 IP, driver, software test application, IP Integrator AXI4 VIP simulation and debug demonstration design.

Figure 4.23 : AXI4 Peripheral IP Generation



Add Interfaces

Add AXI4 interfaces supported by your peripheral

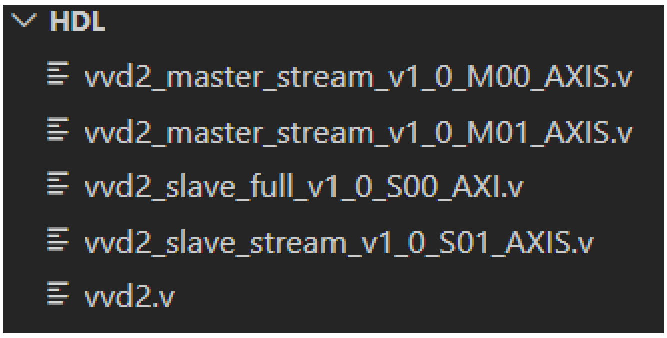
☐ Enable Interrupt Support

Interfaces
- S00_AXI
- S00_AXIS
- M01_AXIS
- M00_AXIS

Name: S00_AXI
Interface Type: Full
Interface Mode: Slave
Data Width (Bits): 32
Memory Size (Bytes): 1024
Number of Registers: 4 [4..512]

Figure 4.24 : New IP with the Interfaces

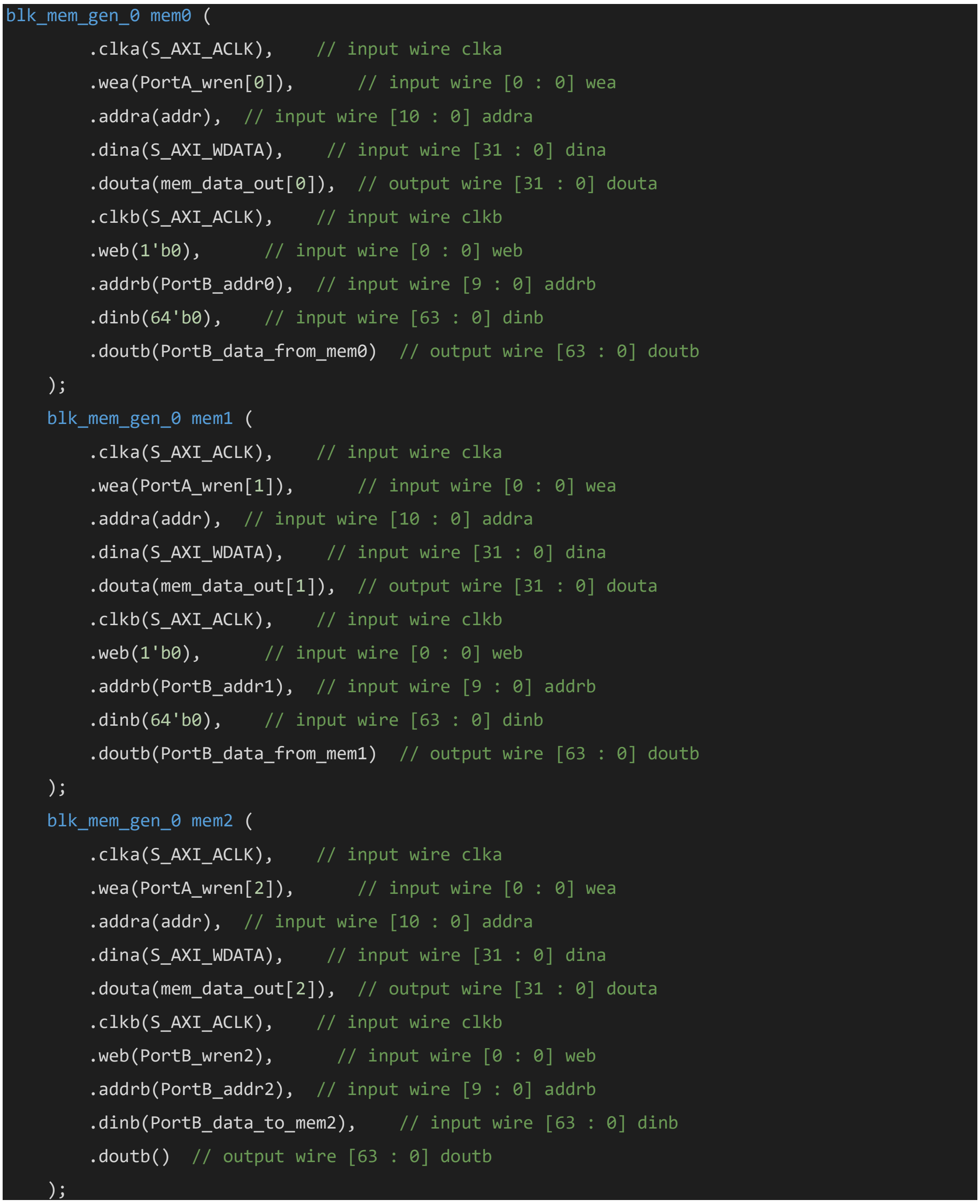
This new IP comes with example codes for interfaces. After creating the new IP these example codes are used as a baseline and modifed according to the projects needs. The designed module VVD2 has 5 verilog files as seen in the Figure 4.25.



HDL
- vvd2_master_stream_v1_0_M00_AXIS.v
- vvd2_master_stream_v1_0_M01_AXIS.v
- vvd2_slave_full_v1_0_S00_AXI.v
- vvd2_slave_stream_v1_0_S01_AXIS.v
- vvd2.v

Figure 4.25 : Source Files of The VVD2

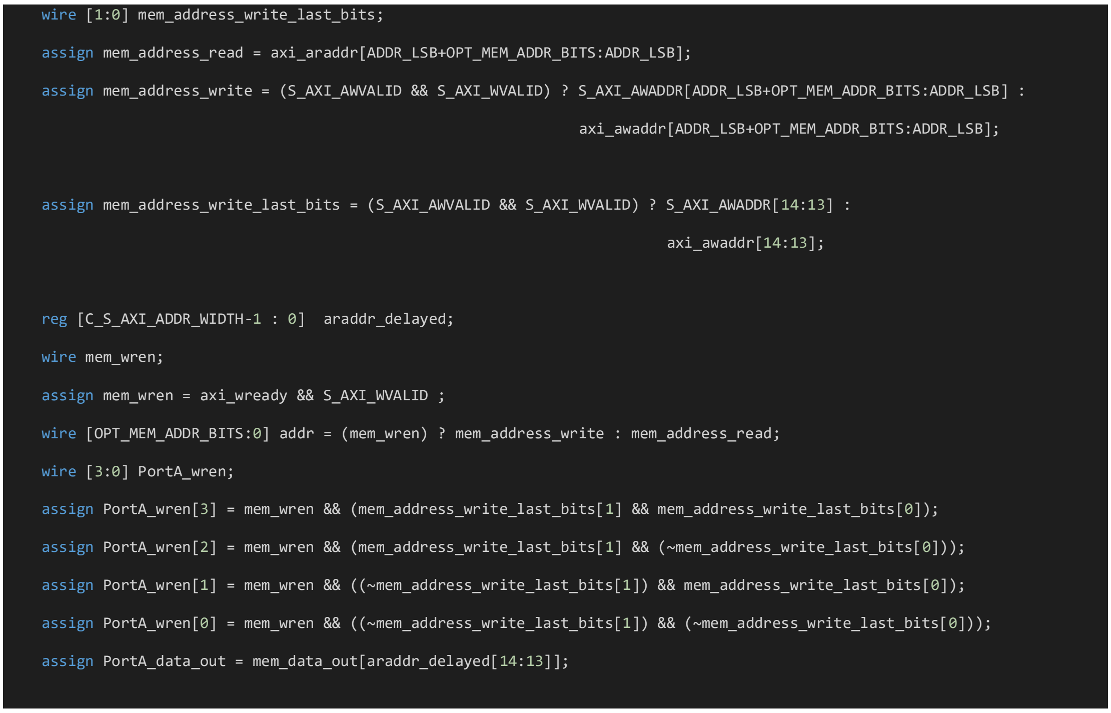
AXI4 slave module in Figure 4.25 includes three BRAMs as shown in the Figure 4.26. BRAMs are dual port with one 32 bit one 64 bit ports. The port with 32 bit data length is used by MicroblazeV via AXI slave. The 64 bit ports are accessed by AXI Stream interfaces to send or receive data. Thus the double percision floating point numbers are written to the memory as two 32 bit parts, the IP automatically combines them to create the correct values. Addresses lenghts are configured accordingly. BRAMs are read and written in 1 clock cycle after the data and address. Thus one more clock cycle delay is added to the read transaction. On the other hand write transaction is not affected by it since actual writing to BRAMs can happen at the same time with the handshake.



```
blk_mem_gen_0 mem0 (
        .clka(S_AXI_ACLK),    // input wire clka
        .wea(PortA_wren[0]),      // input wire [0 : 0] wea
        .addra(addr),  // input wire [10 : 0] addra
        .dina(S_AXI_WDATA),    // input wire [31 : 0] dina
        .douta(mem_data_out[0]),  // output wire [31 : 0] douta
        .clkb(S_AXI_ACLK),    // input wire clkb
        .web(1'b0),      // input wire [0 : 0] web
        .addrb(PortB_addr0),  // input wire [9 : 0] addrb
        .dinb(64'b0),    // input wire [63 : 0] dinb
        .doutb(PortB_data_from_mem0)  // output wire [63 : 0] doutb
    );
    blk_mem_gen_0 mem1 (
        .clka(S_AXI_ACLK),    // input wire clka
        .wea(PortA_wren[1]),      // input wire [0 : 0] wea
        .addra(addr),  // input wire [10 : 0] addra
        .dina(S_AXI_WDATA),    // input wire [31 : 0] dina
        .douta(mem_data_out[1]),  // output wire [31 : 0] douta
        .clkb(S_AXI_ACLK),    // input wire clkb
        .web(1'b0),      // input wire [0 : 0] web
        .addrb(PortB_addr1),  // input wire [9 : 0] addrb
        .dinb(64'b0),    // input wire [63 : 0] dinb
        .doutb(PortB_data_from_mem1)  // output wire [63 : 0] doutb
    );
    blk_mem_gen_0 mem2 (
        .clka(S_AXI_ACLK),    // input wire clka
        .wea(PortA_wren[2]),      // input wire [0 : 0] wea
        .addra(addr),  // input wire [10 : 0] addra
        .dina(S_AXI_WDATA),    // input wire [31 : 0] dina
        .douta(mem_data_out[2]),  // output wire [31 : 0] douta
        .clkb(S_AXI_ACLK),    // input wire clkb
        .web(PortB_wren2),      // input wire [0 : 0] web
        .addrb(PortB_addr2),  // input wire [9 : 0] addrb
        .dinb(PortB_data_to_mem2),    // input wire [63 : 0] dinb
        .doutb()  // output wire [63 : 0] doutb
    );
```

Figure 4.26 : Memory of the Accelerator

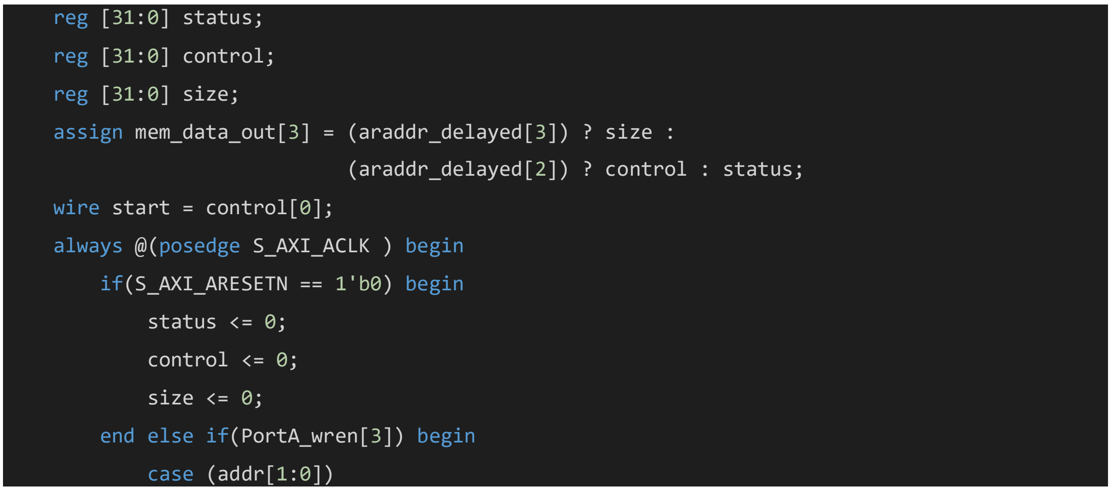
Decoding the write or read address of the AXI revals the BRAM to be written or read as seen in the Figure 4.27. If the two extra bits of the incoming address is both set to one then instead of BRAMs the registers are read or written.



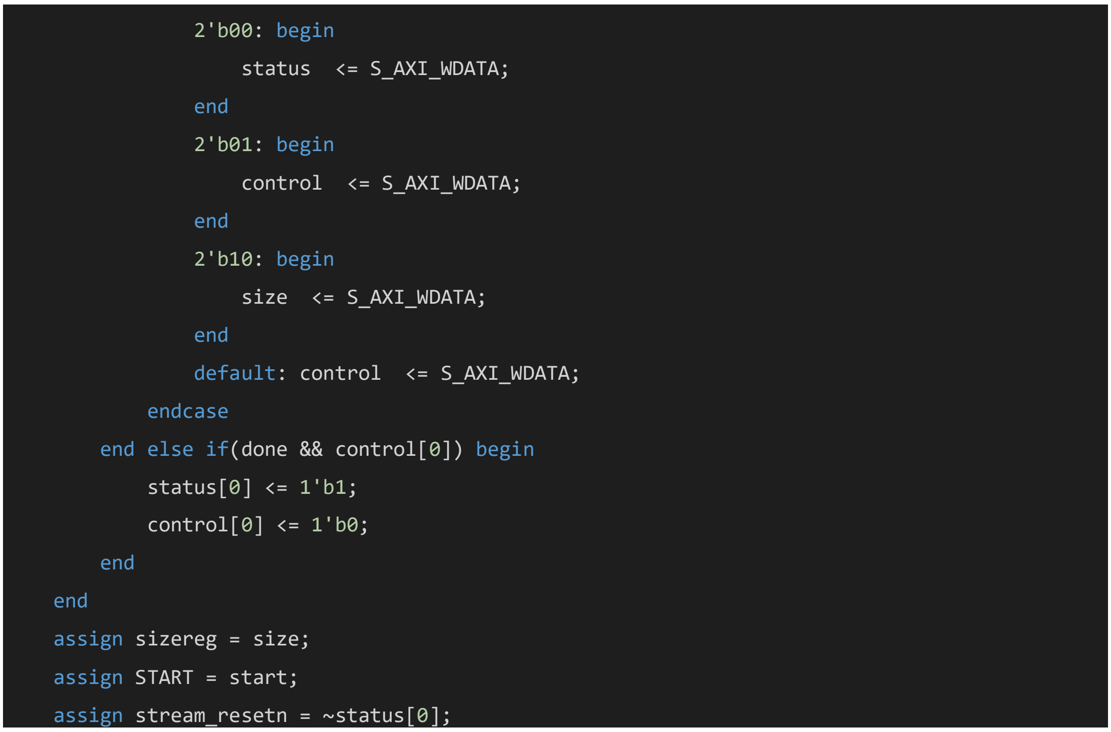
```
wire [1:0] mem_address_write_last_bits;
assign mem_address_read = axi_araddr[ADDR_LSB+OPT_MEM_ADDR_BITS:ADDR_LSB];
assign mem_address_write = (S_AXI_AWVALID && S_AXI_WVALID) ? S_AXI_AWADDR[ADDR_LSB+OPT_MEM_ADDR_BITS:ADDR_LSB] :
                                                             axi_awaddr[ADDR_LSB+OPT_MEM_ADDR_BITS:ADDR_LSB];

assign mem_address_write_last_bits = (S_AXI_AWVALID && S_AXI_WVALID) ? S_AXI_AWADDR[14:13] :
                                                                       axi_awaddr[14:13];

reg [C_S_AXI_ADDR_WIDTH-1 : 0]  araddr_delayed;
wire mem_wren;
assign mem_wren = axi_wready && S_AXI_WVALID ;
wire [OPT_MEM_ADDR_BITS:0] addr = (mem_wren) ? mem_address_write : mem_address_read;
wire [3:0] PortA_wren;
assign PortA_wren[3] = mem_wren && (mem_address_write_last_bits[1] && mem_address_write_last_bits[0]);
assign PortA_wren[2] = mem_wren && (mem_address_write_last_bits[1] && (~mem_address_write_last_bits[0]));
assign PortA_wren[1] = mem_wren && ((~mem_address_write_last_bits[1]) && mem_address_write_last_bits[0]);
assign PortA_wren[0] = mem_wren && ((~mem_address_write_last_bits[1]) && (~mem_address_write_last_bits[0]));
assign PortA_data_out = mem_data_out[araddr_delayed[14:13]];
```

Figure 4.27 : Address Routing Logic and Write Enable Logic

The necessary control logic for registers are implemented as in the Figure 4.28. The done signal, comes from the AXI Stream slave to determine, via the propagation of the TLAST signal, the end of the computation. The control register holds start bit as its first bit. This bits starts the calculation and is set to zero when the calculation is complete. The first bit of the status register is set to indicate the calculation is complete. Thus, AXI Streams interfaces remain in reset until the user clears the status register. To start another calculation status register must be cleared and then the control register's first bit must be set.



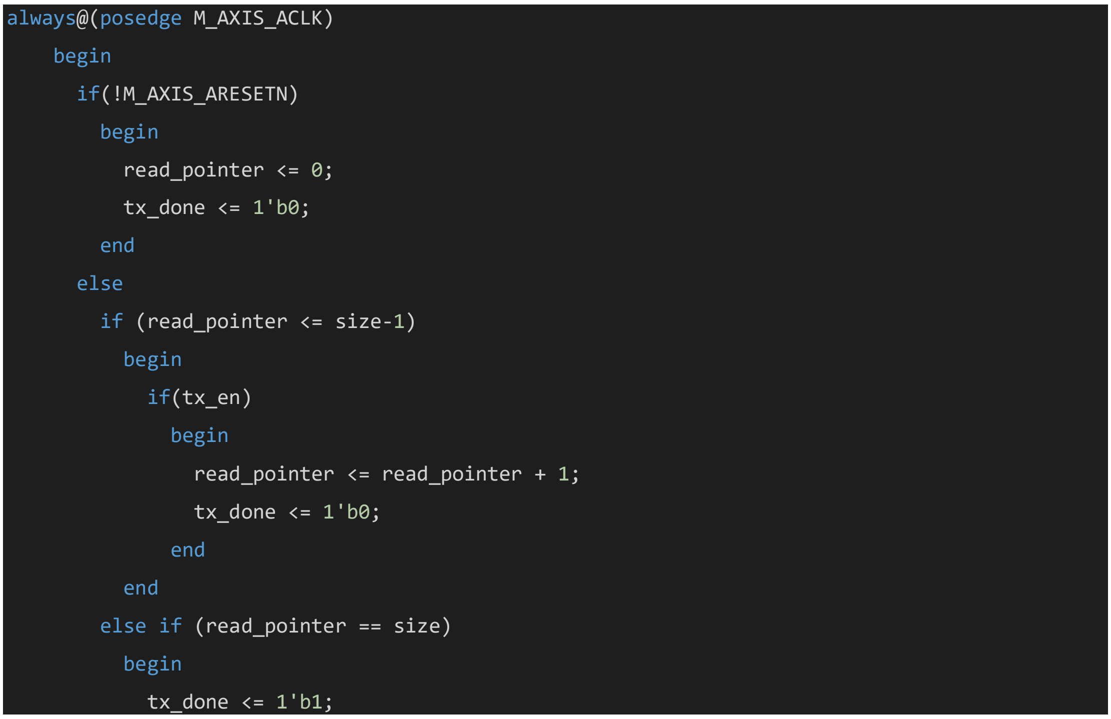
```
reg [31:0] status;
reg [31:0] control;
reg [31:0] size;
assign mem_data_out[3] = (araddr_delayed[3]) ? size :
                         (araddr_delayed[2]) ? control : status;
wire start = control[0];
always @(posedge S_AXI_ACLK ) begin
    if(S_AXI_ARESETN == 1'b0) begin
        status <= 0;
        control <= 0;
        size <= 0;
    end else if(PortA_wren[3]) begin
        case (addr[1:0])
```



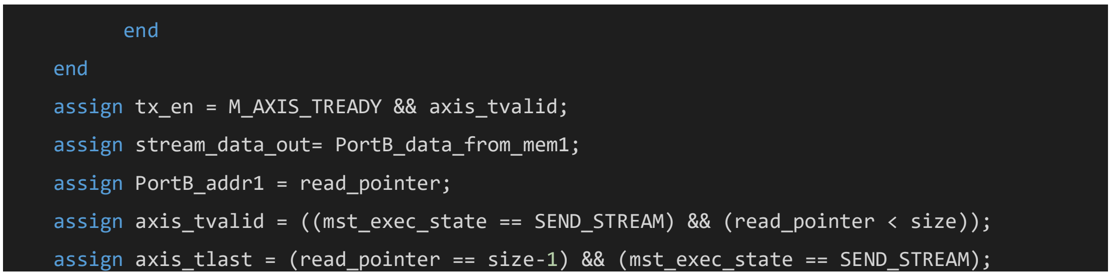
```
                2'b00: begin
                    status  <= S_AXI_WDATA;
                end
                2'b01: begin
                    control  <= S_AXI_WDATA;
                end
                2'b10: begin
                    size  <= S_AXI_WDATA;
                end
                default: control  <= S_AXI_WDATA;
            endcase
        end else if(done && control[0]) begin
            status[0] <= 1'b1;
            control[0] <= 1'b0;
        end
    end
    assign sizereg = size;
    assign START = start;
    assign stream_resetn = ~status[0];
```

Figure 4.28 : Control Logic for the Registers

Master Stream interfaces have internal pointers hat are connected to the second address port of the BRAMs. This pointer is incremented in every handshake of TREADY and TVALID by one as seen in the Figure 4.29. When the read pointer reaches the maximum amount determined by the size register the TLAST signal is sent.



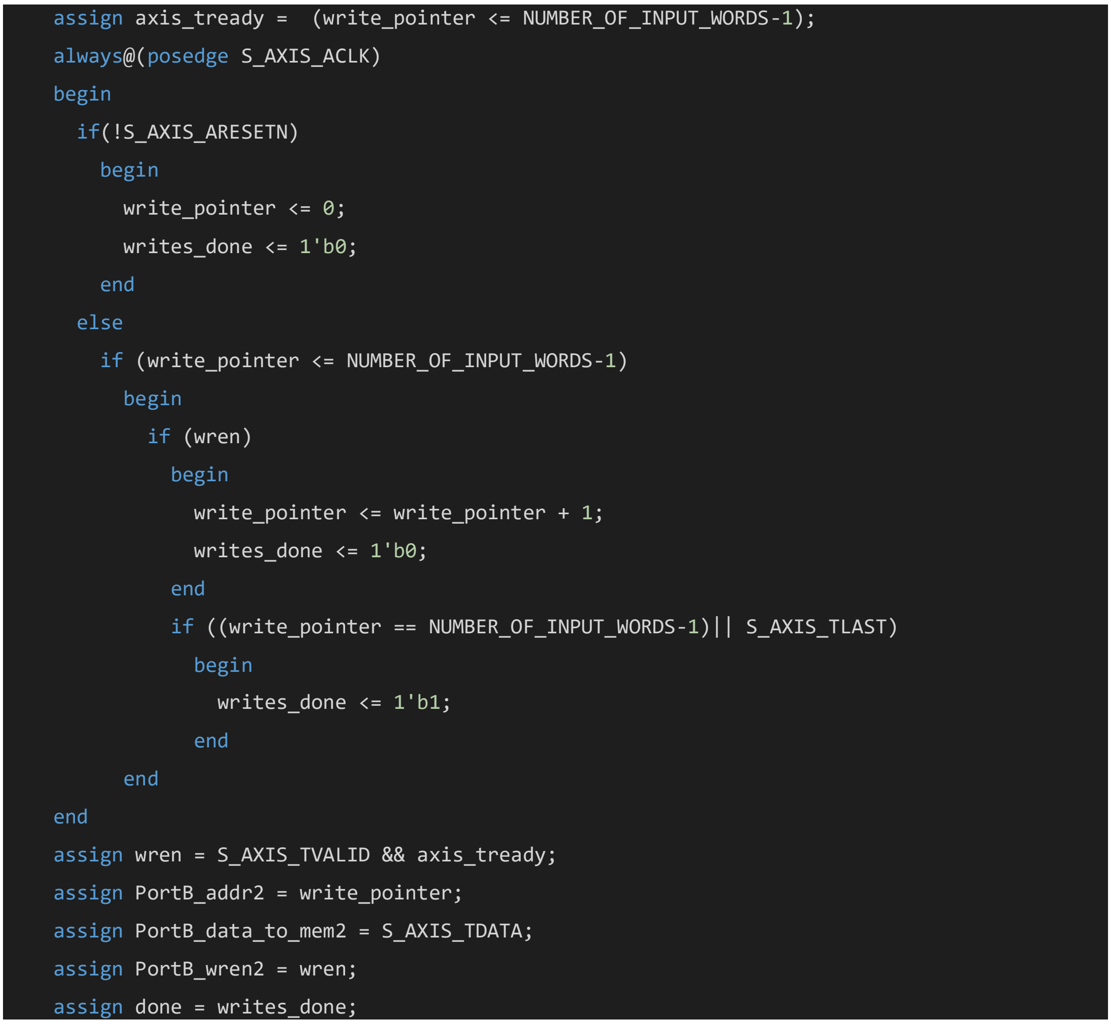
```
always@(posedge M_AXIS_ACLK)
    begin
      if(!M_AXIS_ARESETN)
        begin
          read_pointer <= 0;
          tx_done <= 1'b0;
        end
      else
        if (read_pointer <= size-1)
          begin
            if(tx_en)
              begin
                read_pointer <= read_pointer + 1;
                tx_done <= 1'b0;
              end
          end
        else if (read_pointer == size)
          begin
            tx_done <= 1'b1;
```



```
      end
end
assign tx_en = M_AXIS_TREADY && axis_tvalid;
assign stream_data_out= PortB_data_from_mem1;
assign PortB_addr1 = read_pointer;
assign axis_tvalid = ((mst_exec_state == SEND_STREAM) && (read_pointer < size));
assign axis_tlast = (read_pointer == size-1) && (mst_exec_state == SEND_STREAM);
```

Figure 4.29 : Master Stream Pointer Logic

Slave Stream interface waits in ready state and updates its write pointer in every handshake. When TLAST is detected the done signal is set as seen in the Figure 4.30.



```
assign axis_tready =  (write_pointer <= NUMBER_OF_INPUT_WORDS-1);
always@(posedge S_AXIS_ACLK)
begin
  if(!S_AXIS_ARESETN)
    begin
      write_pointer <= 0;
      writes_done <= 1'b0;
    end
  else
    if (write_pointer <= NUMBER_OF_INPUT_WORDS-1)
      begin
        if (wren)
          begin
            write_pointer <= write_pointer + 1;
            writes_done <= 1'b0;
          end
          if ((write_pointer == NUMBER_OF_INPUT_WORDS-1)|| S_AXIS_TLAST)
            begin
              writes_done <= 1'b1;
            end
      end
end
assign wren = S_AXIS_TVALID && axis_tready;
assign PortB_addr2 = write_pointer;
assign PortB_data_to_mem2 = S_AXIS_TDATA;
assign PortB_wren2 = wren;
assign done = writes_done;
```

Figure 4.30 : Logic for the Stream Slave

With the addition of the new IP, designed block diagram is shown in the Figure 4.31.

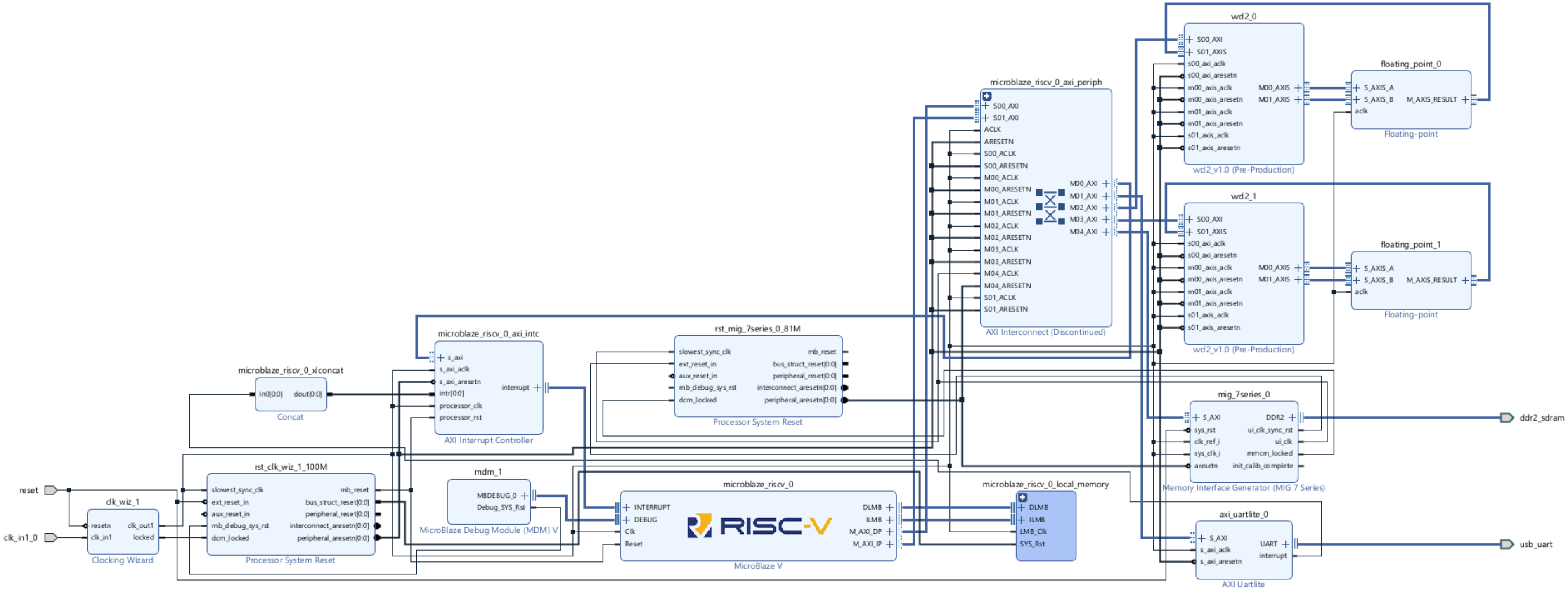


Figure 4.31 : Block Diagram With The Accelerator

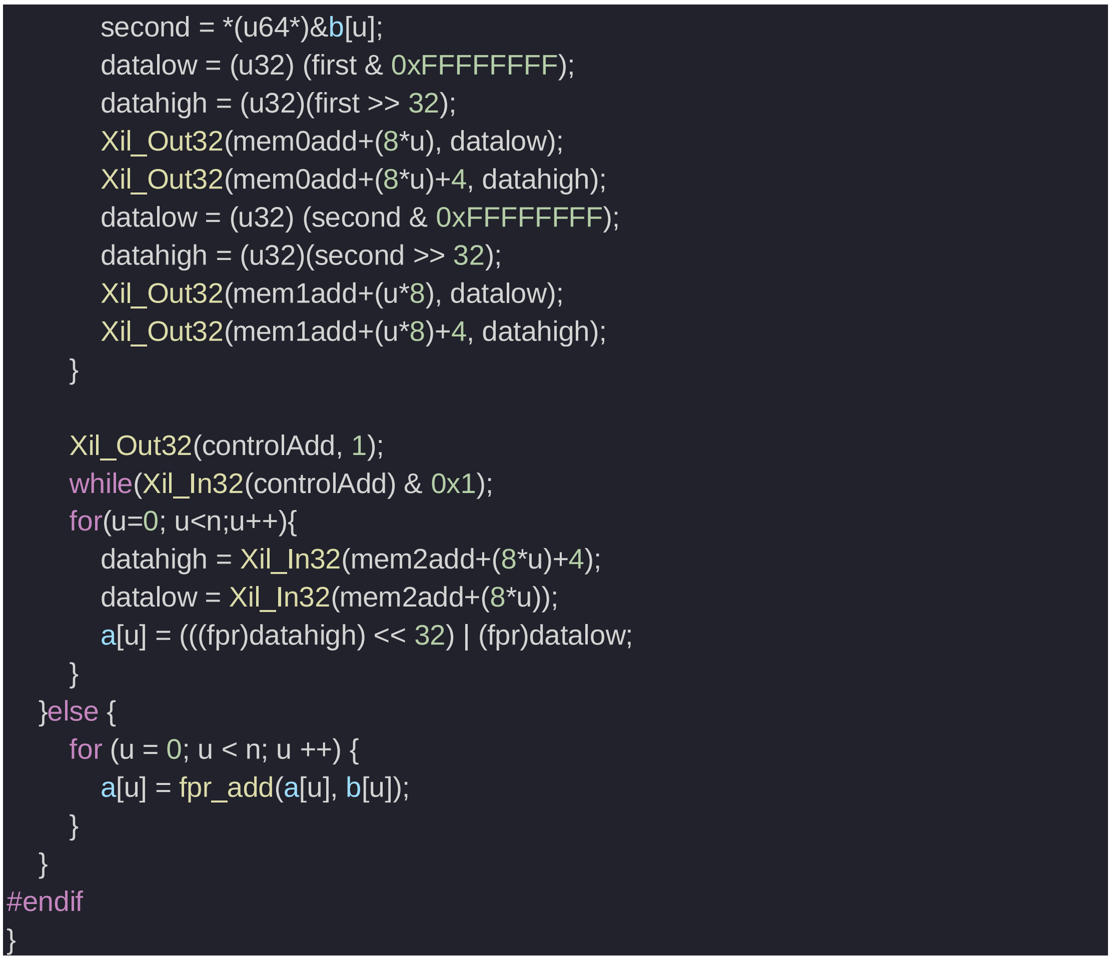
After exporting the newly designed hardware selected C functions are modified to use the accelerator. For easier control the macros seen in the Figure 4.32 are defined. The poly_add function is modified as in Figure 4.33. The first for loop sends the data to the accelerator. The second for loops recieves the data from the accelerator. The control register is monitored to detect the end of the computation.

```
#define mem0add XPAR_VVD2_0_BASEADDR
#define mem1add XPAR_VVD2_0_BASEADDR   +0x2000
#define mem2add XPAR_VVD2_0_BASEADDR   +0x4000
#define statusAdd  XPAR_VVD2_0_BASEADDR   +0x6000
#define controlAdd XPAR_VVD2_0_BASEADDR   +0x6004
#define sizeAdd    XPAR_VVD2_0_BASEADDR   +0x6008

#define mem0mul XPAR_VVD2_1_BASEADDR
#define mem1mul XPAR_VVD2_1_BASEADDR   +0x2000
#define mem2mul XPAR_VVD2_1_BASEADDR   +0x4000
#define statusMul  XPAR_VVD2_1_BASEADDR   +0x6000
#define controlMul XPAR_VVD2_1_BASEADDR   +0x6004
#define sizeMul    XPAR_VVD2_1_BASEADDR   +0x6008
```

Figure 4.32 : Definitions for the Accelerator

The functions only use the accelerator if the size of the arrays are sufficiently large. Otherwise the time lost in memory accesse is not amortized by the pipelined execution in the floating point IPs. The sufficient size is found heuristicly.

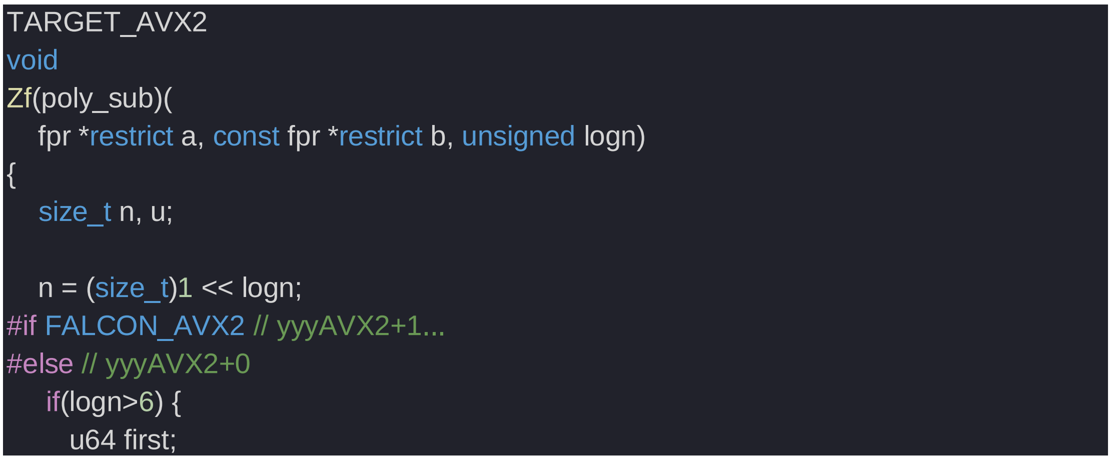
```
TARGET_AVX2
void
Zf(poly_add)(
    fpr *restrict a, const fpr *restrict b, unsigned logn)
{
    size_t n, u;
    n = (size_t)1 << logn;
#if FALCON_AVX2 // yyyAVX2+1..
#else // yyyAVX2+0
    if(logn>6) {
        u64 first;
        u64 second;
        u32 datalow;
        u32 datahigh;
        Xil_Out32(sizeAdd,n);
        Xil_Out32(statusAdd,0);
        for(u=0; u<n; u++){
            first = *(u64*)&a[u];
```



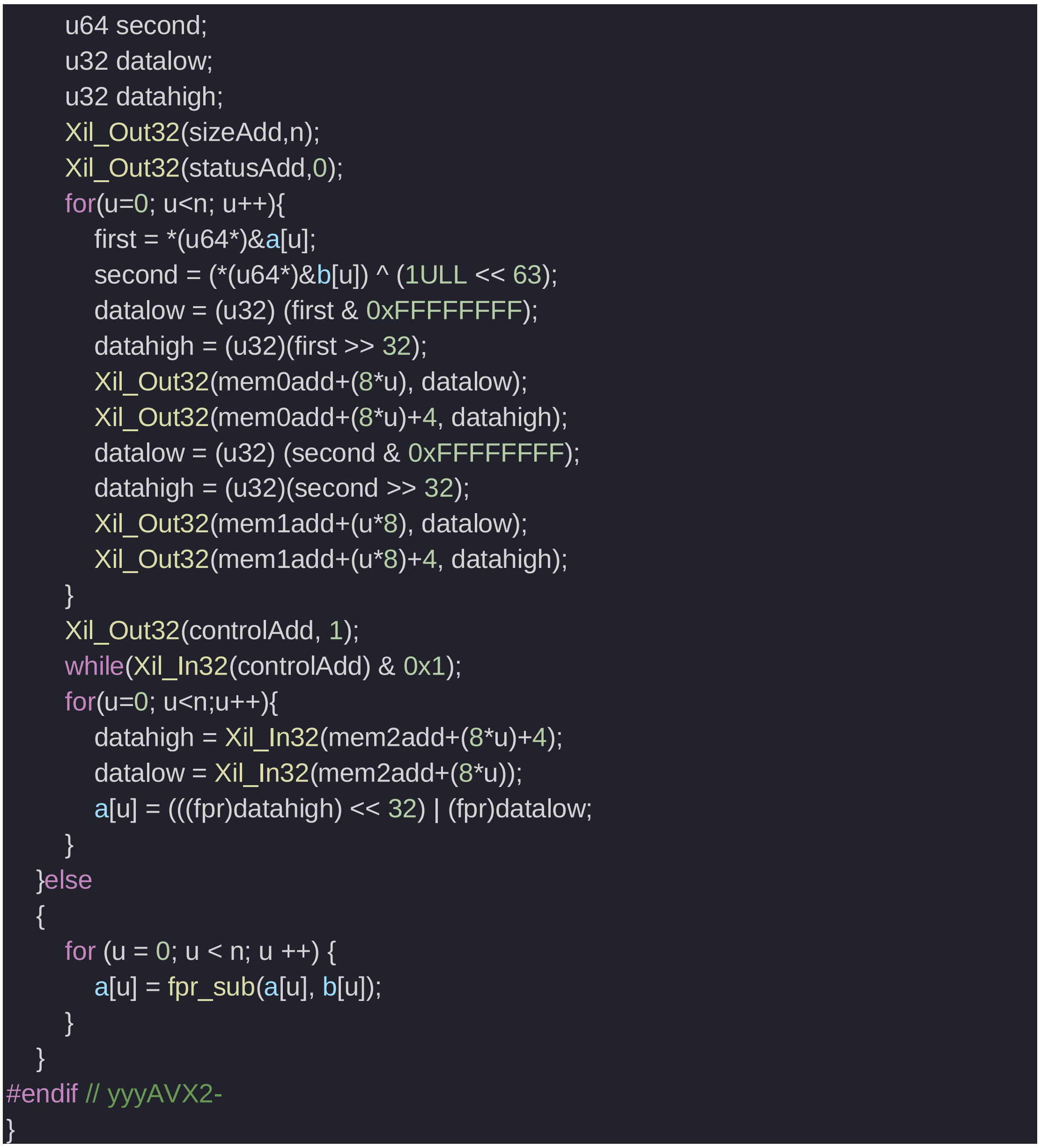
```
            second = *(u64*)&b[u];
            datalow = (u32) (first & 0xFFFFFFFF);
            datahigh = (u32)(first >> 32);
            Xil_Out32(mem0add+(8*u), datalow);
            Xil_Out32(mem0add+(8*u)+4, datahigh);
            datalow = (u32) (second & 0xFFFFFFFF);
            datahigh = (u32)(second >> 32);
            Xil_Out32(mem1add+(u*8), datalow);
            Xil_Out32(mem1add+(u*8)+4, datahigh);
        }

        Xil_Out32(controlAdd, 1);
        while(Xil_In32(controlAdd) & 0x1);
        for(u=0; u<n;u++){
            datahigh = Xil_In32(mem2add+(8*u)+4);
            datalow = Xil_In32(mem2add+(8*u));
            a[u] = (((fpr)datahigh) << 32) | (fpr)datalow;
        }
    }else {
        for (u = 0; u < n; u ++) {
            a[u] = fpr_add(a[u], b[u]);
        }
    }
#endif
}
```

Figure 4.33 : Modification of the poly_mul Function

The poly sub functions is modified likewise as seen in the Figure 4.34. Only difference is that sign bit is flipped when sending the second operand to the accelerator.



```
TARGET_AVX2
void
Zf(poly_sub)(
    fpr *restrict a, const fpr *restrict b, unsigned logn)
{
    size_t n, u;

    n = (size_t)1 << logn;
#if FALCON_AVX2 // yyyAVX2+1...
#else // yyyAVX2+0
     if(logn>6) {
        u64 first;
```



```
        u64 second;
        u32 datalow;
        u32 datahigh;
        Xil_Out32(sizeAdd,n);
        Xil_Out32(statusAdd,0);
        for(u=0; u<n; u++){
            first = *(u64*)&a[u];
            second = (*(u64*)&b[u]) ^ (1ULL << 63);
            datalow = (u32) (first & 0xFFFFFFFF);
            datahigh = (u32)(first >> 32);
            Xil_Out32(mem0add+(8*u), datalow);
            Xil_Out32(mem0add+(8*u)+4, datahigh);
            datalow = (u32) (second & 0xFFFFFFFF);
            datahigh = (u32)(second >> 32);
            Xil_Out32(mem1add+(u*8), datalow);
            Xil_Out32(mem1add+(u*8)+4, datahigh);
        }
        Xil_Out32(controlAdd, 1);
        while(Xil_In32(controlAdd) & 0x1);
        for(u=0; u<n;u++){
            datahigh = Xil_In32(mem2add+(8*u)+4);
            datalow = Xil_In32(mem2add+(8*u));
            a[u] = (((fpr)datahigh) << 32) | (fpr)datalow;
        }
    }else
    {
        for (u = 0; u < n; u ++) {
            a[u] = fpr_sub(a[u], b[u]);
        }
    }
#endif // yyyAVX2-
}
```

Figure 4.34 : Modification of the poly_sub Function

The poly_mul_fft function is likewise modified as seen in the Figure 4.35. First half of the code computes the real part of the complex multiplication while the second half computes the imaginary part. The poly_muladj_fft function has the same changes except that sign bit of the imaginary part is flipped for second array before sending to the accelerator.

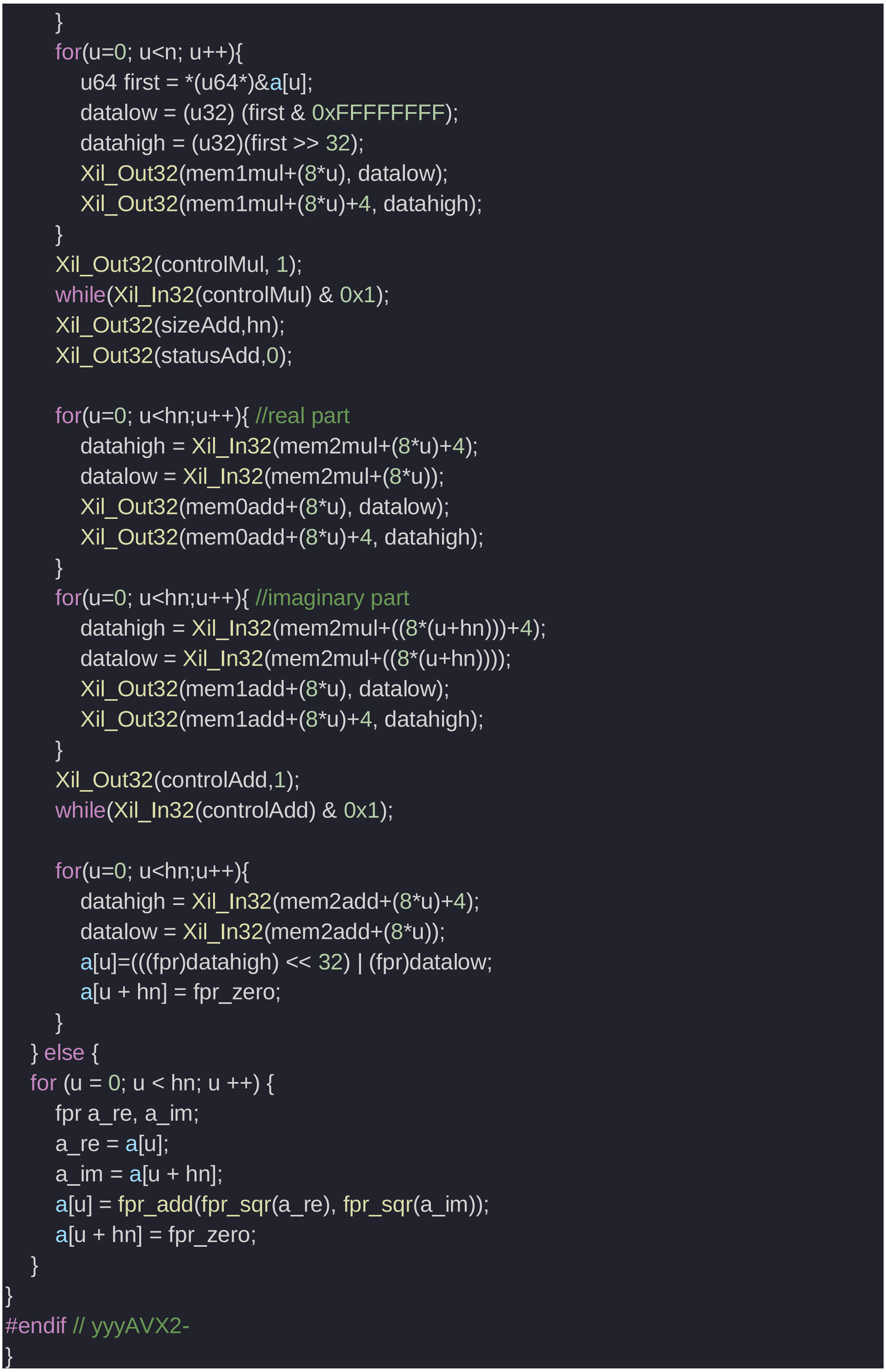
```
TARGET_AVX2
void
Zf(poly_mul_fft)(
    fpr *restrict a, const fpr *restrict b, unsigned logn)
{
    size_t n, hn, u;

    n = (size_t)1 << logn;
    hn = n >> 1;
#if FALCON_AVX2 // yyyAVX2+1..
#else // yyyAVX2+0
    if(logn>7){
        //(ar*br - ai*bi)+j(ar*bi+br*ai)
        u32 datalow;
        u32 datahigh;
        Xil_Out32(sizeMul,n);
        Xil_Out32(statusMul,0);
        for(u=0; u<n; u++){ // multiplications of (ar*br - ai*bi)
            u64 first = *(u64*)&a[u];
            u64 second = *(u64*)&b[u];

            datalow = (u32) (first & 0xFFFFFFFF);
            datahigh = (u32)(first >> 32);
            Xil_Out32(mem0mul+(8*u), datalow);
            Xil_Out32(mem0mul+(8*u)+4, datahigh);


            datalow = (u32) (second & 0xFFFFFFFF);
            datahigh = (u32)(second >> 32);
            Xil_Out32(mem1mul+(8*u), datalow);
            Xil_Out32(mem1mul+(8*u)+4, datahigh);
        }

        Xil_Out32(controlMul, 1);
        while(Xil_In32(controlMul) & 0x1);

        Xil_Out32(sizeAdd,hn);
        Xil_Out32(statusAdd,0);

        for(u=0; u<hn;u++){ // writing the ar*br of (ar*br - ai*bi)
            datahigh = Xil_In32(mem2mul+(8*u)+4);
            datalow = Xil_In32(mem2mul+(8*u));
            Xil_Out32(mem0add+(8*u), datalow);
            Xil_Out32(mem0add+(8*u)+4, datahigh);
        }

        for(u=0; u<hn;u++){ // writing the ai*bi of (ar*br - ai*bi)
            datahigh = Xil_In32(mem2mul+((8*(u+hn)))+4);
    datalow = Xil_In32(mem2mul+((8*(u+hn))));
    datahigh ^= 0x80000000; //fliping the sign bit
    Xil_Out32(mem1add+(8*u), datalow);
    Xil_Out32(mem1add+(8*u)+4, datahigh);
}
Xil_Out32(controlAdd,1);
while(Xil_In32(controlAdd) & 0x1); //if real part of the result is ready
//the real part of the result is ready (ar*br - ai*bi)



        //Begining of the imaginary part j(ar*bi+br*ai)
for(u=0; u<hn; u++){
    //mem0mul is already filled with a[u] in needed format
    u64 second = *(u64*)&b[u+hn];
    datalow = (u32) (second & 0xFFFFFFFF);
    datahigh = (u32)(second >> 32);
    Xil_Out32(mem1mul+(8*u), datalow);
    Xil_Out32(mem1mul+(8*u)+4, datahigh);
}
for(u=hn; u<n; u++){
    //mem0mul is already filled with a[u] in needed format
    u64 second = *(u64*)&b[u-hn];
    datalow = (u32) (second & 0xFFFFFFFF);
    datahigh = (u32)(second >> 32);
    Xil_Out32(mem1mul+(8*u), datalow);
    Xil_Out32(mem1mul+(8*u)+4, datahigh);
}
Xil_Out32(statusMul,0);
Xil_Out32(controlMul, 1);
while(Xil_In32(controlMul) & 0x1);
for(u=0; u<hn;u++){ //writing the (ar*br - ai*bi) to a[u]
    datahigh = Xil_In32(mem2add+(8*u)+4);
    datalow = Xil_In32(mem2add+(8*u));
    a[u]=(((fpr)datahigh) << 32) | (fpr)datalow;
}
//adding of the j(ar*bi+br*ai)
for(u=0; u<hn;u++){
    datahigh = Xil_In32(mem2mul+(8*u)+4);
    datalow = Xil_In32(mem2mul+(8*u));
    Xil_Out32(mem0add+(8*u), datalow);
    Xil_Out32(mem0add+(8*u)+4, datahigh);
}
for(u=0; u<hn;u++){
    datahigh = Xil_In32(mem2mul+((8*(u+hn)))+4);
    datalow = Xil_In32(mem2mul+((8*(u+hn))));
    Xil_Out32(mem1add+(8*u), datalow);
    Xil_Out32(mem1add+(8*u)+4, datahigh);
}
```

```
        Xil_Out32(statusAdd,0);
        Xil_Out32(controlAdd,1);
        while(Xil_In32(controlAdd) & 0x1);
        for(u=0; u<hn;u++){ //writing the j(ar*bi+br*ai) part
            datahigh = Xil_In32(mem2add+(8*u)+4);
            datalow = Xil_In32(mem2add+(8*u));
            a[u+hn]=(((fpr)datahigh) << 32) | (fpr)datalow;
        }
    }else{
        for (u = 0; u < hn; u ++) {
            fpr a_re, a_im, b_re, b_im;

            a_re = a[u];
            a_im = a[u + hn];
            b_re = b[u];
            b_im = b[u + hn];
            FPC_MUL(a[u], a[u + hn], a_re, a_im, b_re, b_im);
        }
    }
#endif // yyyAVX2-
}
```

Figure 4.35 : The Modification of the poly_mul Function

The mul_selfadj function is also accelerated as in the Figure 4.36.

```
TARGET_AVX2
void
Zf(poly_mulselfadj_fft)(fpr *a, unsigned logn)
{
    size_t n, hn, u;

    n = (size_t)1 << logn;
    hn = n >> 1;
#if FALCON_AVX2 // yyyAVX2+1
#else // yyyAVX2+0
    if(logn>7){
        u32 datalow;
        u32 datahigh;
        Xil_Out32(sizeMul,n);
        Xil_Out32(statusMul,0);
        for(u=0; u<n; u++){
            u64 first = *(u64*)&a[u];
            datalow = (u32) (first & 0xFFFFFFFF);
            datahigh = (u32)(first >> 32);
            Xil_Out32(mem0mul+(8*u), datalow);
            Xil_Out32(mem0mul+(8*u)+4, datahigh);
```



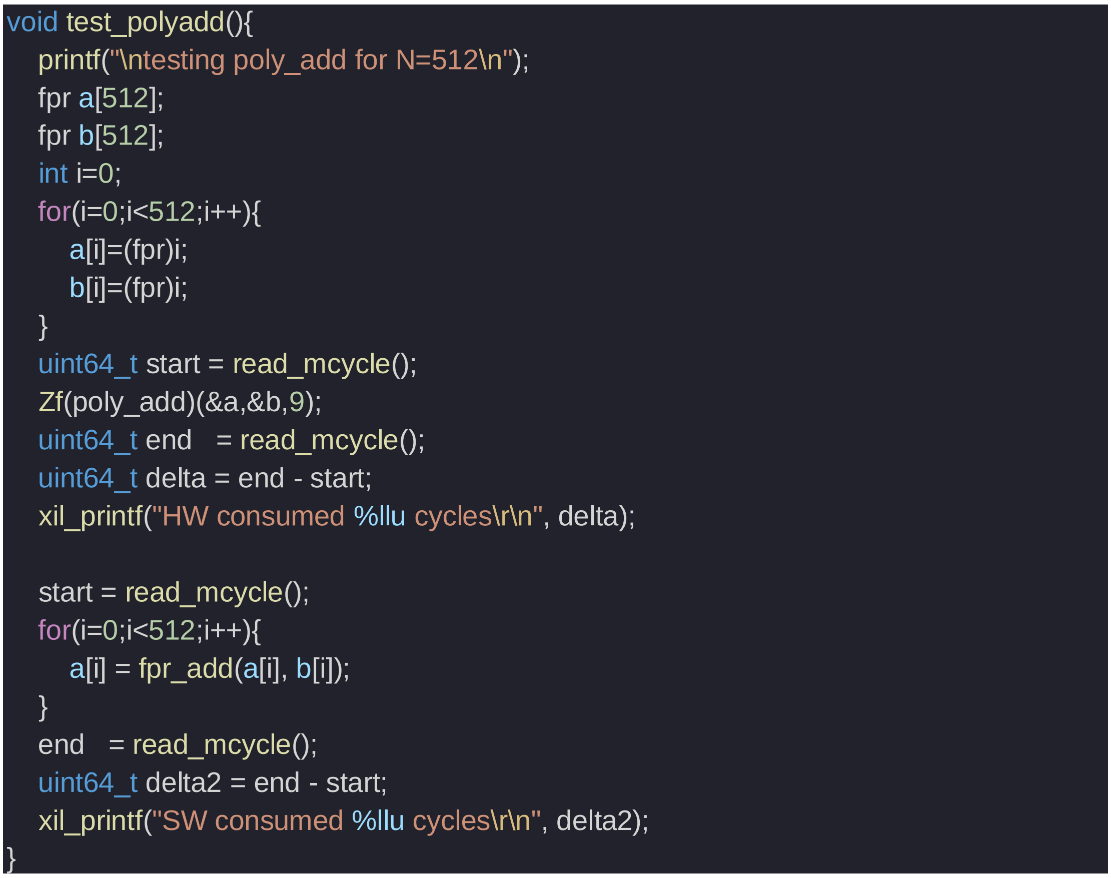
```
        }
        for(u=0; u<n; u++){
            u64 first = *(u64*)&a[u];
            datalow = (u32) (first & 0xFFFFFFFF);
            datahigh = (u32)(first >> 32);
            Xil_Out32(mem1mul+(8*u), datalow);
            Xil_Out32(mem1mul+(8*u)+4, datahigh);
        }
        Xil_Out32(controlMul, 1);
        while(Xil_In32(controlMul) & 0x1);
        Xil_Out32(sizeAdd,hn);
        Xil_Out32(statusAdd,0);

        for(u=0; u<hn;u++){ //real part
            datahigh = Xil_In32(mem2mul+(8*u)+4);
            datalow = Xil_In32(mem2mul+(8*u));
            Xil_Out32(mem0add+(8*u), datalow);
            Xil_Out32(mem0add+(8*u)+4, datahigh);
        }
        for(u=0; u<hn;u++){ //imaginary part
            datahigh = Xil_In32(mem2mul+((8*(u+hn)))+4);
            datalow = Xil_In32(mem2mul+((8*(u+hn))));
            Xil_Out32(mem1add+(8*u), datalow);
            Xil_Out32(mem1add+(8*u)+4, datahigh);
        }
        Xil_Out32(controlAdd,1);
        while(Xil_In32(controlAdd) & 0x1);

        for(u=0; u<hn;u++){
            datahigh = Xil_In32(mem2add+(8*u)+4);
            datalow = Xil_In32(mem2add+(8*u));
            a[u]=(((fpr)datahigh) << 32) | (fpr)datalow;
            a[u + hn] = fpr_zero;
        }
    } else {
    for (u = 0; u < hn; u ++) {
        fpr a_re, a_im;
        a_re = a[u];
        a_im = a[u + hn];
        a[u] = fpr_add(fpr_sqr(a_re), fpr_sqr(a_im));
        a[u + hn] = fpr_zero;
    }
}
#endif // yyyAVX2-
}
```

Figure 4.36 : Modification of the poly_mulselfadj Function

5. TEST RESULTS

5.1 Acceleration of the Determined Functions

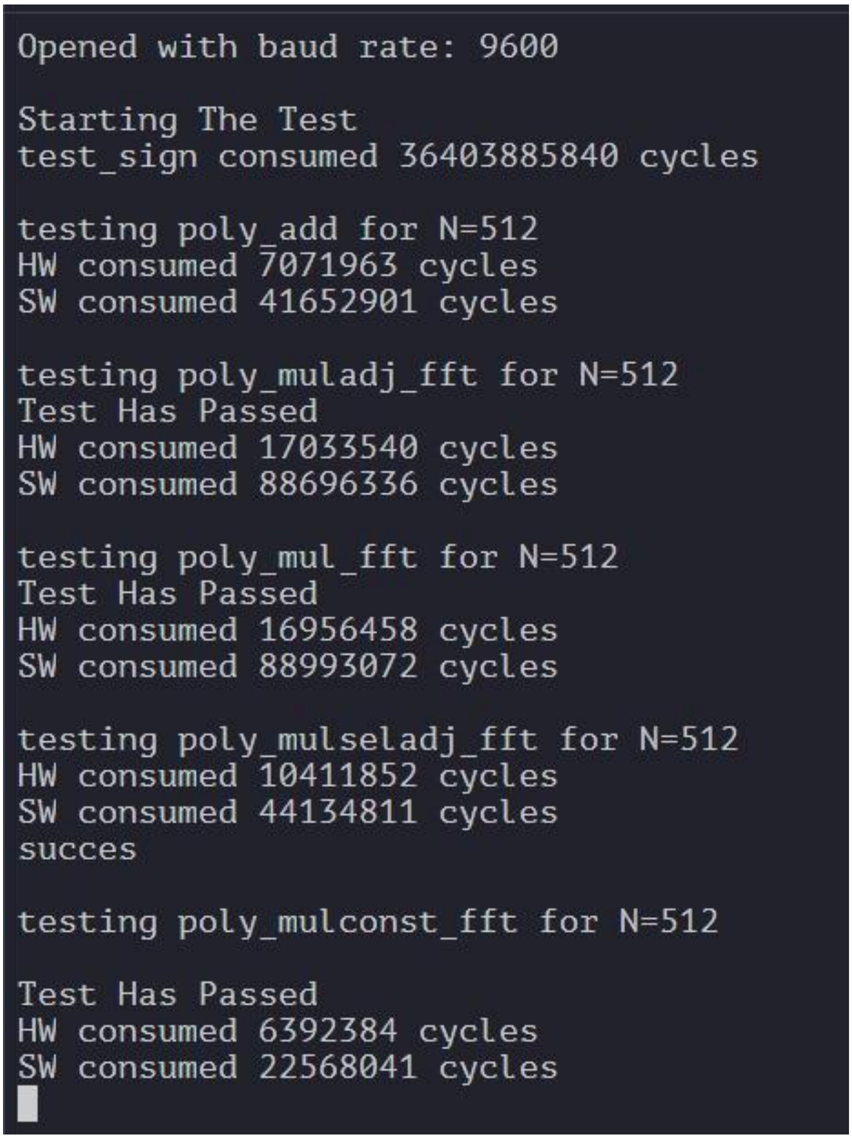
Special test functions are developed for measuring the latency of the designed hardware. These functions compare the floating-point emulation and accelerator implementation as seen in Figure 5.1.



```
void test_polyadd(){
    printf("\ntesting poly_add for N=512\n");
    fpr a[512];
    fpr b[512];
    int i=0;
    for(i=0;i<512;i++){
        a[i]=(fpr)i;
        b[i]=(fpr)i;
    }
    uint64_t start = read_mcycle();
    Zf(poly_add)(&a,&b,9);
    uint64_t end   = read_mcycle();
    uint64_t delta = end - start;
    xil_printf("HW consumed %llu cycles\r\n", delta);

    start = read_mcycle();
    for(i=0;i<512;i++){
        a[i] = fpr_add(a[i], b[i]);
    }
    end   = read_mcycle();
    uint64_t delta2 = end - start;
    xil_printf("SW consumed %llu cycles\r\n", delta2);
}
```

Figure 5.1: Test function for poly_add

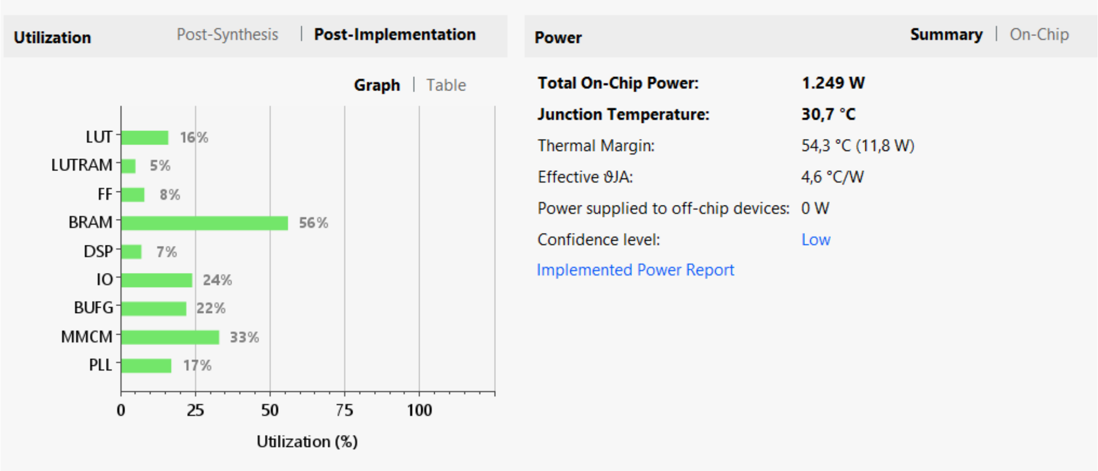
With these functions and the test_sign function the performance of the designed accelerator is measured as shown in Figure 5.2. Since poly_sub and poly_add functions are nearly identical except one sign bit flipping, poly_sub is not explicitly shown.



```
Opened with baud rate: 9600

Starting The Test
test_sign consumed 36403885840 cycles

testing poly_add for N=512
HW consumed 7071963 cycles
SW consumed 41652901 cycles

testing poly_muladj_fft for N=512
Test Has Passed
HW consumed 17033540 cycles
SW consumed 88696336 cycles

testing poly_mul_fft for N=512
Test Has Passed
HW consumed 16956458 cycles
SW consumed 88993072 cycles

testing poly_mulseladj_fft for N=512
HW consumed 10411852 cycles
SW consumed 44134811 cycles
succes

testing poly_mulconst_fft for N=512

Test Has Passed
HW consumed 6392384 cycles
SW consumed 22568041 cycles
```

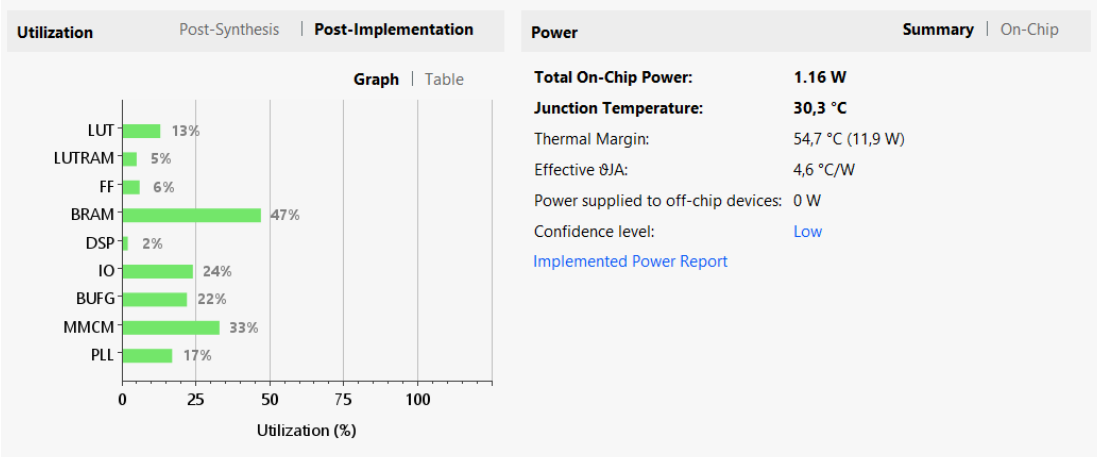
Figure 5.2: Test Results of the Accelerator

As seen in the Figure 5.2 Accelerator accelerated poly_add function by 83%, poly_muladj_fft by 80.7%, poly_mul_fft by 80.9%, poly_mulselfadj_fft by 76.4%, and poly_mulconst_fft by 71.67%. While the LUT count is increased by about 23% as seen in Figures 5.3 and 5.4. Power consumption has only increased by 7.6%.



Total On-Chip Power: **1.249 W**
Junction Temperature: **30,7 °C**
Thermal Margin: 54,3 °C (11,8 W)
Effective ϑJA: 4,6 °C/W
Power supplied to off-chip devices: 0 W
Confidence level: Low

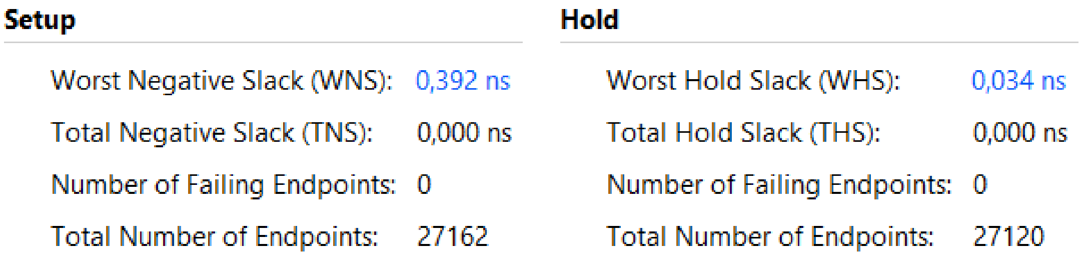
Figure 5.3: Utilization and Power Consumption of the Accelerated System



Total On-Chip Power: **1.16 W**
Junction Temperature: **30,3 °C**
Thermal Margin: 54,7 °C (11,9 W)
Effective ϑJA: 4,6 °C/W
Power supplied to off-chip devices: 0 W
Confidence level: Low

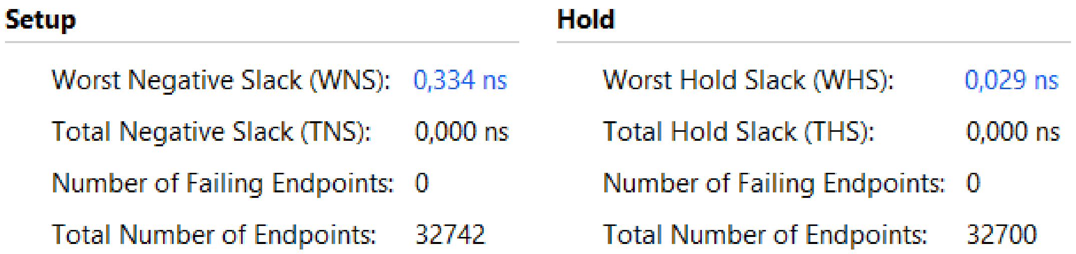
Figure 5.4: Utilization and Power Consumption of the Base System

Considering both designs are clocked at 100Mhz, the effect of the custom accelerator on maximum frequency is only 0.6% as can be seen in Figure 5.5 and 5.6. On the other hand, test_sign function only saw 6.4% improvement as seen by comparing Figure 4.11 and 5.2. This is because test_sign function also includes verification of the signature which does not use floating point and there are other heavier tasks such as FFT that takes the most time in signature generation.



Setup		Hold	
Worst Negative Slack (WNS):	0,392 ns	Worst Hold Slack (WHS):	0,034 ns
Total Negative Slack (TNS):	0,000 ns	Total Hold Slack (THS):	0,000 ns
Number of Failing Endpoints:	0	Number of Failing Endpoints:	0
Total Number of Endpoints:	27162	Total Number of Endpoints:	27120

Figure 5.5: Timing Summary of the Base System



Setup		Hold	
Worst Negative Slack (WNS):	0,334 ns	Worst Hold Slack (WHS):	0,029 ns
Total Negative Slack (TNS):	0,000 ns	Total Hold Slack (THS):	0,000 ns
Number of Failing Endpoints:	0	Number of Failing Endpoints:	0
Total Number of Endpoints:	32742	Total Number of Endpoints:	32700

Figure 5.6: Timing Summary of the Accelerated System

The total energy consumption as calculated by muliplying the power with the clock period and the number of clock cycles for the completion of the functions changed as follows:

- poly_add: 18.2% of what it was before the in the base system.
- poly_muladj_fft: 20.7% of what it was before.
- poly_mul_fft: 20.5% of what it was before.
- poly_mulselfadj_fft: 25.3% of what it was before.
- poly_mulconst: 30.4% of what it was before.
- test_sign: increased by 0.7% with the accelerator.

6. REALISTIC CONSTRAINTS AND CONCLUSIONS

6.1 Practical Application of this Project

The hardware accelerator developed in this project can be integrated with any 32-bit processor that supports an AXI bus interface. By offloading the selected functions in Figure 5.2 to custom hardware, the accelerator reduces the execution time. This makes it suitable for embedded systems, IoT devices, and other platforms where the processor has limited computing power but may still require secure post quantum digital signatures. The AXI interface establishes compatibility with a range of different SoC designs and processor architectures, allowing the accelerator to be used in both custom and commercially available FPGA-based or ASIC-based systems.

6.2 Realistic Constraints

The most important constraint in the usage of the custom accelerator in the project is that the bus speed of the processor affects the performance. Accelerator supports burst transactions to minimize the latency of data transfers before the computation. However, if the processor's AXI bus does not have burst transaction capabilities the performance degrades.

6.2.1 Social, environmental and economic impact

By contributing to hardware accelerator design for Falcon, this project helps keep systems secure against future quantum computer attacks, protecting against financial losses, security breaches, and other risks.

6.2.2 Cost analysis

Assuming 7 hours of workload per week for 28 weeks as in the course catalogs of the project and a salary of 10$ an hour, total salary cost for two members of the project is 3920$. Equipment cost is 300$ for the NEXYS 4 DDR. Thus the total cost is 4220$.

6.2.3 Standards

In the project IEEE-754 Floating Point Standart[4] is followed, and the resulting hardware is compatible with the standart.

6.2.4 Health and safety concerns

The project does not pose any health and safety risks.

6.3 Future Work and Recommendations

Future improvements should focus on optimizing the time-consuming parts of Falcon’s signing process. In particular, accelerating the Fast Fourier Transform and Number Theoretic Transform operations would provide the largest performance gains.

REFERENCES

[1] **V. Mavroeidis, K. Vishi, M. D. Zych, and A. Jøsang**, “*The impact of quantum computing on present cryptography*”, arXiv preprint, arXiv:1804.00200, 2018. [Online]. Available: https://arxiv.org/pdf/1804.00200.)

[2] **National Institute of Standards and Technology,** "*Post-Quantum Cryptography: Selected Algorithms 2022*," 2022. [Online]. Available: https://csrc.nist.gov/Projects/post-quantum-cryptography/selected-algorithms-2022. [Accessed: Nov. 24, 2024].

[3] **Falcon Project**, "Falcon: Fast-Fourier Lattice-based Compact Signatures over NTRU," [Online]. Available: https://falcon-sign.info/. [Accessed: Nov. 25, 2024].

[4] ***IEEE Standard for Floating-Point Arithmetic***, IEEE Std 754-2019 (Revision of IEEE Std 754-2008), pp. 1–84, 22 July 2019, doi: 10.1109/IEEESTD.2019.8766229 .

[5] **RISC-V International,** *The RISC-V Instruction Set Manual, Volume I: Unprivileged Architecture*, Version 20250508, May 2025. Available: https://riscv.org/technical/specifications/

[6] **Digilent Inc.,** ***Nexys 4 DDR (Legacy) Board*** *— Reference*, Digilent Reference, Rev. A–C. [Online]. Available: https://digilent.com/reference/programmable-logic/nexys-4-ddr/start. [Accessed: Aug. 13, 2025].

[7] **AMD**, “What is the Vivado Design Suite?”, *Vivado Design Suite User Guide: Getting Started (UG910)*, Version 2025.1 EN, Document ID UG910, May 29, 2025. [Online]. Available: https://docs.amd.com/r/en-US/ug910-vivado-getting-started/What-is-the-Vivado-Design-Suite. [Accessed: Aug. 13, 2025].

[8] **AMD**, “What is Vitis?”, *Vitis Tutorials: Getting Started (XD098)*, Version 2025.1 EN, Document ID XD098, July 31, 2025. [Online]. Available: https://docs.amd.com/r/en-US/Vitis-Tutorials-Getting-Started/What-is-Vitis. [Accessed: Aug. 13, 2025].

[9] AMD, *MicroBlaze V Processor Reference Guide (UG1629)*, Document ID UG1629, Version 2025.1 English, Release Date: July 9, 2025. [Online]. Available: https://docs.amd.com/r/en-US/ug1629-microblaze-v-user-guide. [Accessed: Aug. 13, 2025].

[10] ARM, "AMBA® AXI™ and ACE™ Protocol Specification," ARM IHI 0022E, 2015. [Online]. Available: http://www.gstitt.ece.ufl.edu/courses/fall15/eel4720_5721/labs/refs/AXI4_specification.pdf

[11] Arm Limited, AMBA AXI-Stream Protocol Specification, IHI 0051B, 2.2 ed., Apr. 2021.

[12] X. Yu, Y. Sun, Y. Zhao, H. Kuang and J. Han, "*RVCE-FAL: A RISC-V Scalar-Vector Custom Extension for Faster FALCON Digital Signature,*" 2024 Design, Automation & Test in Europe Conference & Exhibition (DATE), Valencia, Spain, 2024, pp. 1-6, doi: 10.23919/DATE58400.2024.10546713.

[13] P. Karl, J. Schupp, T. Fritzmann, and G. Sigl, "*Post-Quantum Signatures on RISC-V with Hardware Acceleration***,"** Cryptology ePrint Archive, Paper 2022/538, 2022. [Online]. Available: https://eprint.iacr.org/2022/538

[14] Y. Lee et al., "An Efficient Hardware/Software Co-Design for FALCON on Low-End Embedded Systems," in IEEE Access, vol. 12, pp. 57947-57958, 2024, doi: 10.1109/ACCESS.2024.3387489

[15] M. Schmid, D. Amiet, J. Wendler, P. Zbinden, and T. Wei, *"Falcon Takes Off - A Hardware Implementation of the Falcon Signature Scheme,"* Cryptology ePrint Archive, Paper 2023/1885, 2023. [Online]. Available: https://eprint.iacr.org/2023/1885

[16] S.C. Seo, S.W. An, and D. Choi, "*Accelerating Falcon Post-Quantum Digital Signature Algorithm on Graphic Processing Units,*" *Comput. Mater. Contin.*, vol. 75, no. 1, pp. 1963-1980, 2023. https://doi.org/10.32604/cmc.2023.033910

CURRICULUM VITAE

Name Surname	**: Bora Kıran**
Place and Date of Birth	**: Denizli, 15/09/2002**
E-Mail	**: kiran21@itu.edu.tr**

CURRICULUM VITAE

Name Surname : Cenker Çetinkaya

Place and Date of Birth : Istanbul, 24/02/2002

E-Mail : cetinkayac21@itu.edu.tr