Encryption and Authentication on a RISC-V Architecture

João Cassiano Vicente Fernandes

Dissertação para obtenção do Grau de Mestre em
Engenharia Eletrotécnica e de Computadores
2º ciclo de estudos

Orientador: Prof. Doutor António Eduardo Vitória do Espírito Santo

novembro de 2023

Declaração de Integridade

Eu, João Cassiano Vicente Fernandes, que abaixo assino, estudante com o número de inscrição M10928 de Engenharia Eletrotécnica e de Computadores da Faculdade Engenharia, declaro ter desenvolvido o presente trabalho e elaborado o presente texto em total consonância com o **Código de Integridades da Universidade da Beira Interior**.

Mais concretamente afirmo não ter incorrido em qualquer das variedades de Fraude Académica, e que aqui declaro conhecer, que em particular atendi à exigida referenciação de frases, extratos, imagens e outras formas de trabalho intelectual, e assumindo assim na íntegra as responsabilidades da autoria.

Universidade da Beira Interior, Covilhã 02/11 /2023

Assinado por: **João Cassiano Vicente Fernandes**
Num. de Identificação: 15666668
Data: 2023.11.02 17:34:51+00'00'

(assinatura conforme Cartão de Cidadão ou preferencialmente
assinatura digital no documento original se naquele mesmo formato)

Agradecimentos

Tenho de agradecer aos meus pais, pela minha formação pessoal e académica, por nunca me abandonarem e sempre me incentivarem a lutar e a continuar, por me ensinarem a acreditar em mim e nos meus sonhos, por toda a paciência e apoio nos meus momentos mais difíceis, por sempre estarem presentes nas minhas vitórias e especialmente nas minhas derrotas com palavras de força e de carinho. Obrigado, mãe, obrigado pai. Também não posso deixar de agradecer à minha irmã e ao meu cunhado que sempre me apoiaram e acreditaram que era capaz.

Um agradecimento muito especial, Cristiana Mota, à qual agradeço cada momento que simplesmente me ouviu, cada palavra de incentivo e força, cada sermão, cada palavra de amor.

Devo também um agradecimento a todos os meus amigos de curso, que estiveram ao meu lado durante todo este percurso, em especial ao João Moura e ao José Rebelo, pela ajuda e amizade incondicional que sempre me deram, tanto a nível universitário como a nível pessoal.

A todos os meus amigos mais próximos, aos meus colegas de laboratório Helbert, David, Renato e António. Ao Professor Doutor António Espírito-Santo, meu orientador, ao qual estou muito grato pela confiança e por ter sempre acreditado nas minhas capacidades.

Resumo

Desenvolvido em 2010 na Universidade da Califórnia, em Berkley, o novo ISA, denominado de RISC-V, foi criado tendo em mente a capacidade de extensibilidade do código aberto. O RISC-V ISA foi projetado para ser modular, permitindo a implementação de instruções personalizadas para aplicações específicas. A sua flexibilidade e facilidade de uso deram à comunidade acadêmica e comercial a escolha ideal para o desenvolvimento de sistemas embutidos, permitindo a criação de processadores otimizados para os requisitos específicos de cada sistema.

Grandes empresas, como Google, NVIDIA, Western Digital, Samsung e Qualcomm, são membros da fundação RISC-V, que é responsável por garantir a evolução do RISC-V, bem como promover a sua adoção e desenvolvimento de novos ISAs. O apoio da comunidade de código aberto (open-source) ajudou a aumentar a popularidade do RISC-V. As atualizações diárias de diferentes implementações, bem como as adaptações e desenvolvimento de ferramentas de programação, proporcionaram a qualquer utilizador, experiente ou não, uma forma rápida e fácil de aprender e implementar um ISA personalizado e adequado às suas necessidades.

Garantir a segurança dos dados e a privacidade do usuário é uma grande preocupação em todos os dispositivos, especialmente aqueles com recursos computacionais limitados. Construir um sistema com bom desempenho e segurança é um desafio. O esforço computacional necessário para encriptar e autenticar mensagens deve ser viável com a capacidade de processamento disponível. Uma implementação em hardware de algoritmos de criptografia e autenticação pode libertar alguma carga de trabalho de um microcontrolador e, permitir que arquiteturas mais simples e eficientes sejam desenvolvidas.

Esta dissertação apresenta um estudo em torno da arquitetura RISC-V e dos algoritmos de encriptação ChaCha20 e autenticação Poly1305, aproveitando a capacidade de personalização do RISC-V para implementar um módulo de segurança na arquitetura, capaz de lidar com a encriptação e autenticação de dados sem o uso dos recursos do SoC.

Palavras-chave

RISC-V, ChaCha20, Poly1305, Open-source, Soc.

Resumo Alargado

O IoT está presente vários setores, como a saúde, o transporte e serviços públicos, através de dispositivos que recolhe, processam e transmitem grandes quantidades de dados. Os ataques e a sofisticação dos mesmos evoluem ao mesmo ritmo que as novas tecnologias. Segundo um relatório de segurança informática da Security Boulevard [5], o crime informático causou aproximadamente 1,5 biliões de dólares em perdas em 2018. Os ataques impulsionados pela Inteligência Artificial (AI) [6] colocam ainda mais pressão nos sistemas de segurança. Inúmeros ataques informáticos expuseram vulnerabilidades de hardware em processadores modernos [7].

Ao enviar ou receber dados, é necessário algum nível de confiança entre os dispositivos. A encriptação de dados é necessária para garantir que apenas o remetente e o destinatário possam o possam aceder e interpretar. Uma solução para o problema é implementar mecanismos de segurança que permitam que uma mensagem seja encriptada e autenticada pelos dispositivos. No entanto, as tecnologias de encriptação são conhecidas pela exigência substancial dos recursos computacionais necessários para a encriptar/autenticar uma mensagem. Os algoritmos de encriptação devem ser compatíveis com a limitada capacidade de processamento dos microcontroladores usados em dispositivos IoT.

A implementação de algoritmos de encriptação e autenticação em software pode não ser compatível com os recursos computacionais disponíveis no MCU (Microcontrolador). Uma implementação de hardware é uma alternativa atraente para fornecer segurança e autenticação em pequenos dispositivos. Os algoritmos de encriptação ChaCha20 e de autenticação Poly1305 são uma possibilidade a ser explorada [8].

O ambiente tecnologicamente avançado de hoje tem pressionado as pequenas empresas, e até mesmo a nível acadêmico, a acompanhar a procura por desenvolvimento de hardware [9]. O mercado de opções de CPU inclui as arquiteturas ARM, x86 e RISC-V. No entanto, ARM e x86 são conhecidos pela sua complexidade e elevadas taxas de utilização. Uma arquitetura *open-source*, de uso gratuito, como o RISC-V, pode ser uma alternativa de mercado [8].

Um processador *softcore* é uma implementação de uma arquitetura computacional dentro de um FPGA. A versatilidade desta abordagem está na possibilidade de reconfigurar o hardware para executar uma determinada tarefa conforme a vontade do utilizador.

Este trabalho utiliza um FPGA Xilinx XC7A100T-1CSG324C, que alimenta a placa de desenvolvimento Nexys A7 Artix-7 da Digilent.

O trabalho utiliza uma implementação da arquitetura RISC-V RVfpgaNexys, desenvolvida para fins educacionais e fornecida pela Imagination, com um módulo adicional de funções de encriptação e autenticação. Esse módulo é uma implementação de hardware dos algoritmos de encriptação Chacha20 e autenticação poly1305.

Abstract

Developed in 2010 at the University of California, Berkley, the new ISA, called RISC-V, was created with the open-source extensibility ability in mind. The RISC-V ISA is designed to be modular, allowing for the implementation of custom instruction sets for specific applications. Its flexibility and ease of use gave the academic and commercial community an ideal choice for embedded systems development, it allows the creation of optimized processors for specific system requirements.

Big companies, such as Google, NVIDIA, Western Digital, Samsung, and Qualcomm, are members of the RISC-V foundation, which is responsible for ensuring the RISC-V evolution as well as promoting its adoption and development of new ISAs. The support from the open-source community helped grow the RISC-V's popularity. The daily updates of different implementations, as well as adaptations and development of programming tools, gave any user, experienced or not, a fast and easy way to learn and implement a custom ISA tailored to his needs.

Ensuring data security and user privacy is a big concern for all devices, especially those with limited computational resources. Building a system with good performance and security is a challenge. The computational effort required for encrypting and authenticating messages must be feasible with the available processing capacity. A hardware implementation of encryption and authentication algorithms can release some workload from the smaller microcontroller and, thus, allow for simpler and more efficient architectures to be diploid.

This dissertation presents a survey around the RISC-V ISA and the ChaCha20 encryption and Poly1305 authentication algorithms to take advantage of the customization ability of the RISC-V ISA to implement a security module in the architecture, able to handle data encryption and authentication without the use of the SoC resources.

Keywords

RISC-V, ChaCha20, Poly1305, Open-source, Soc.

Index

Figures List

Capítulo 1
Introduction

1.1 Motivation

IoT is transforming various industries, including healthcare, transportation, and public services, through interconnected devices that collect, process, and transmit vast amounts of data. The industry widely recognizes big data for its value. Mining large quantities of data can give enterprises competitive advantages and provide a basis for the decision-making of social departments. IDC predicts 41.6 billion IoT devices will be connected by 2025, and the collected data will exceed 79ZB [1].

In the age of big data, security and privacy issues brought new challenges [2]–[4]. The sophistication of attacks is developing at the same rate as the technology evolves. Cyber-attacks have become more complex, illusory, and targeted than ever. According to a cyber security report by Security Boulevard [5], cybercrime caused approximately $1.5 trillion in losses in 2018. AI-driven attacks [6] worsen the situation and make security defences increasingly challenging. Countless cyber-attacks have exposed hardware vulnerabilities in modern processors[7].

When sending or receiving data, some level of trust between devices is required. Data encryption is needed to ensure that only the sender and the receiver can access and interpret it. One solution to the problem is to deploy security mechanisms that allow for a message to be encrypted and authenticated by the devices. Although those cryptographic technologies are known for the substantial computational effort required for the encryption/authentication of messages, they must be compatible with the constrained processing capacity of the microcontroller used in IoT devices.

Implementing encryption and authentication algorithms in software may not be compatible with the computational resources available at the MCU. A hardware implementation is an attractive alternative to provide security and authentication to small devices. ChaCha20 encryption and Poly1305 authentication algorithms are a possibility to explore [8].

Today's technologically advanced environment has pressured small companies, and even at the academic level, to keep up with the demand for hardware development [9]. The market for CPU options for a developer to work on includes the ARM, x86, and RISC-V architectures. However, ARM and x86 are based on Instruction Set Architecture (ISA) and are known for their complexity and substantial licensing fees. Open-source hardware and free-to-use ISA, like the RISC-V, described as the Linux of hardware, can help with this struggle[8].

A softcore processor is an implementation of a computational architecture within an FPGA, and the versatility of this approach is the possibility to reconfigure the hardware to perform a given task at the developer's will.

The present work uses a Xilinx XC7A100T-1CSG324C FPGA, which powers the Nexys A7 Artix-7 FPGA Trainer Board from Digilent.

The work uses an implementation of the RISC-V architecture RVfpgaNexys, developed for educational purposes and provided by Imagination, with an added encryption and authentication functions module. That module is a hardware implementation of the Chacha20 encryption and poly1305 authentication algorithms.

1.2 Objectives

The primary aim of this task is to create a RISC-V softcore processor that includes a message encryption and authentication module. To summarize, the key goals are as follows:

- Hardware implementation of the ChaCha20 and Poly1305 algorithm;
- Implementation of RISC-V ISA core on an FPGA;
- Connect the RISC-V to the ChaCha20 and Poly1305 modules thru the wishbone bus;
- Evaluate the performance of the implementations using simulation platforms such as Verilator and GTKwave;
- Use the UART to establish an encrypted message exchange platform between the RISC-V core and a computer.

1.3 Contributions

This thesis was developed to explore and implement the ChaCha20 encryption and Poly1305 message authentication algorithms as a practical solution for transducer security.

The open-source of a RISC-V core architecture, the excellent documentation, and the possibility to implement it on an FPGA provide a good platform to add encryption and authentication peripherals to a RISC-V softcore.

The work uses an implementation of the RISC-V architecture RVfpgaNexys, which was developed by Imagination for educational purposes. Although many other core implementations are available to exploit and use, the RVfpgaNexys was selected for its ease of use and well-suited documentation for beginner users.

1.4 Structure

This thesis is divided into six chapters that are organized as follows:

- Chapter 2 presents the RISC-V ISA and its extensions, then compares various RISC-V softcore processors. The chapter also explores the RVfpgaNexys structure.
- Chapter 3 covers the essential development tools for interacting with the processor, including the FPGA board, HDL design software for synthesis and analysis, and simulation interfaces.
- In Chapter 4, an in-depth understanding of the ChaCha20 message encryption and Poly1305 authentication algorithms is exploited.
- Chapter 5 evaluates the software and hardware implementation of the encryption/authentication algorithms. Additionally, it establishes a simple encrypted message exchange between the RISC-V transducer and a computer using a UART interface.
- Chapter 6 presents the concluding remarks and recommendations for further work.

Capítulo 2
RISC-V ISA

The instruction set architecture (ISA) outlines how machine code should behave, regardless of the specific hardware being used. This allows programs to be compatible across different implementations of the same architecture. The ISA defines the interface between hardware and software, the behaviors allowed for processor implementation, and establishes the assumptions for software development and verification [10].

2.1 Creation of RISC-V

In the summer of 2010, at the University of Berkley, in California, Prof. Krste Asanović , Prof. David Patterson, and graduate students Yunsup Lee and Andrew Waterman started a three-month project to develop their clean slate ISA [11].

The RISC-V got its name because it represents the fifth major ISA coming out of Berkeley.

Name	Date
RISC-I	1981
RISC-II	1983
RISC-III (SOAR)	1984
RISC-IV (SPUR)	1988
RISC-V (RAVEN-1)	2011

Table 1: RISC ISAs develop in Berkley University.

The original motivation for RISC-V was to make architecture education and research more real. It is possible to teach students how to improve systems using the system's people use in their industry instead of having unrealistic simple systems.

The three-month project grew, some silicon implementations and articles about the new RISC-V ISA were published, and people outside Berkeley started to use and implement the new ISA. In

2014, four years after beginning a simple summer project, the first RISC-V base user specification was frozen and released [7], [11].

In May 2014, the RISC-V foundation was created to ensure the evolution of the RISC-V and promote its adoption and development of the new ISA. The foundation is a membership organization where companies, individuals, or other groups can join. There are 13 University members, 29 research and consulting organizations that joined to do software and development tools, and companies that design and manufacture silicon wafer chips. The individual membership allows anyone to join the foundation for free. Big companies, such as Google, NVIDIA, Western Digital, Samsung, and Qualcomm, are members of the RISC-V foundation[8].

There are other open ISAs:

- Openrisc: The major goal of the project is to create a free and open processor for embedded systems. A free and Openrisc instruction set architecture with DSP features[12]:
    - open-source implementations of the architecture
    - software development tools, libraries, operating systems, and applications.
    - system-on-chip and system simulators.
- lattice LM 32 core: is a 32-bit Harvard, RISC architecture available for free with an open IP core licensing agreement. It includes software development tools (LatticeMico™ System) and evaluation boards to test the designs in hardware.

2.2 Differences between X86, ARM, and RISC-V

Intel, AMD and ARM are proprietary ISAs. That means closed intellectual property, where each company sells a product, a CPU, or they sell a license to use the instruction set architecture. You're paying to use their intellectual property [13].

RISC-V, on the other hand, is an open-source specification. It is not necessarily an open-source processor. The ISA is free to use and implement, although it's still necessary to purchase the CPU cores from the manufactures.

The differences between X86, ARM, and RISC-V are subtle. They're related to how memory is being addressed, were branches executed, and how exceptions are handled.

RISC Computing is reduced instruction set and CISC is complex instruction set Computing. The differences between RISC and CISC are the trade-offs between the power consumption, computing performance, security, and how virtualization is implemented.

ARM and RISC-V are both based on RISC reduced instruction set Computing Concepts, while X86 is based on CISC. The reduced number of instructions is what it makes advantageous to choose a RISC instead of a CISC machine [13].

For example, to execute a multiplication on old RISC ISAs, four instructions were needed:

1. load data into one register.
2. load into a second register.
3. execute a multiply where it multiplied the number in the first register against the number in the second.
4. store the result into a third register.

Break down a task to multiply into multiple instructions that complete the operation of multiply two numbers together.

CISC uses complex instructions. There are instructions in the CISC that multiply two registers together and store it in a third. It takes fewer instructions to write a CISC application but when it executes the multiply A B command takes more clock cycles to complete.

On RISC, each one of the simplified instructions takes usually only one CPU cycle to complete that function. RISC will overall consume less power than a CISC instruction does.

The CISC construction takes more clock cycles to complete than a RISC one does. RISC use simpler instructions which execute in a single clock cycle but with a cost that it takes more instructions to be able to execute the same task. CISC takes less instructions but consumes more power because there are more cycles of the CPU involved to accomplish it.

There are only two ways to accomplish performance improvement as shown by the following equation. Either minimize the number of instructions on a program or reduce the number of cycles per instruction to execute it faster.

$$\frac{time}{program} = \frac{time}{cycle} \times \frac{cycle}{instruction} \times \frac{instruction}{program}$$

The RISC approach to reduce the number of cycles per instruction is the more successful way of reducing power consumption. RISC instruction takes less power consumption at the expense of performance.

ARM, combat that problem by adding more complex instructions at the expense of power consumption to gain performance.

Intel has gone the other way. It has started to break down some of its op codes into “micro-operands” or Micro Ops. Those are more RISC-like instructions. They reduce the power consumption at the expense of performance.

2.3 RISC-V base integer instruction sets

A RISC-V ISA is defined as a base integer ISA where optional extensions could be added to meet the need of the user application. The base ISA is restricted to a minimal set of instructions to provide a reasonable target for compilers, assemblers, and operating systems development. This gives the developers a convenient ISA, properly documented, easy to learn, and with access to all the tools needed for customization.

The RISC-V is usually treated as a single ISA, although it represents a family of five related ISAs characterized by the width of each integer registers and consequently, the corresponding size of the address space. There are two primary base integer variants, RV32I and RV64I which provide 32-bit or 64-bit address respectively. RV32E and, most recently, RV64E are subset variants of the RV32I and RV64I base instruction sets respectively, which use only 16 integer registers (half of the RV32I instruction set) to support small microcontrollers. A RV128I variant of the base integer instruction was also developed with the data center and security uses with future applications in mind[11].

One easy way to know how an ISA works is by analyzing its architectural state and instructions. In the nest section the RV32I base instruction set will be analyze, follow by a briefly description of the other base ISAs and an enumeration of the most popular RISC-V extensions.

2.4 RV32I base integer instruction set

The RV32i core state is composed by 31 general-purpose registers, x1 through x31, x0 is hardwired to 0 (zero), and a program counter (PC). The registers are 32bit wide, making a total of 1024bits of architectural state on the RV32I. The term XLEN is used to refer to the width of an integer register in bits[11].

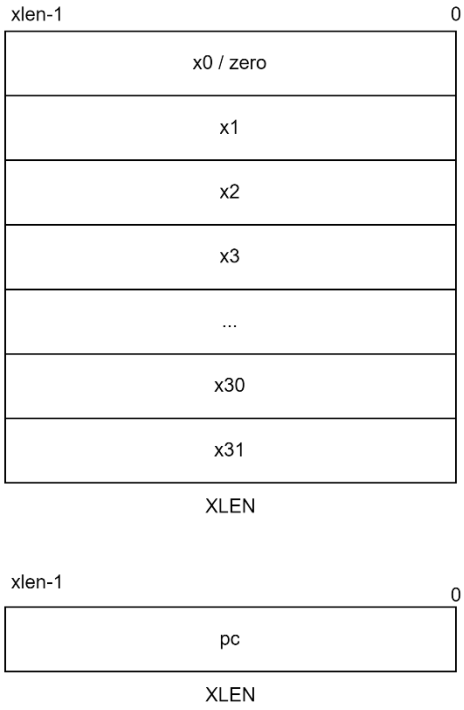


Figure 1: RISC-V base unprivileged integer register state from [11].

All the instructions have a length of 32bit and are represented in 6 different formats: 4 base formats (R/I/S/U) and 2 immediate-encoding variants (B/J) witch only differ how the immediate is encoded. As shown in the figure, colored in red, green, and blue are the register specifiers. Rd, in red, is the destination register, where it's written the result of an instruction to a register. In green and blue are respectively representing the first and the second source registers that are used as operands in an instruction. If they are in the instruction encoding, they're always in the same position in al formats to simplify decoding, allowing to get the values of rs1 and rs2 before the instruction be decoded. Although they are marginal, the gains can assume a big difference when applied to a superscalar or out of order implementation.

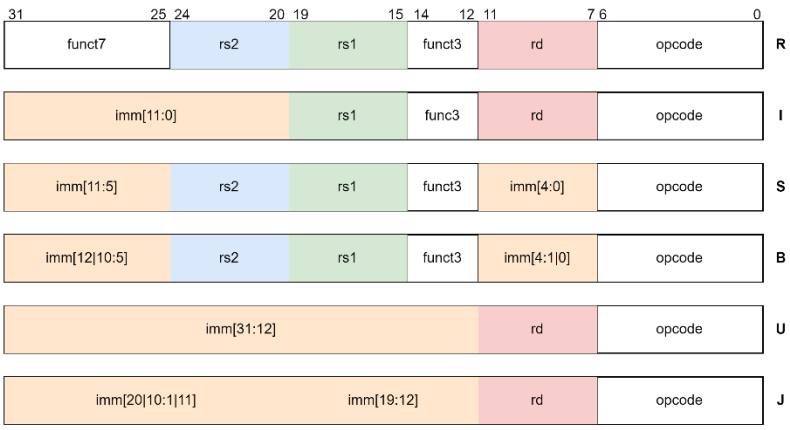


Figure 2: RISC-V base instruction formats based on [11].

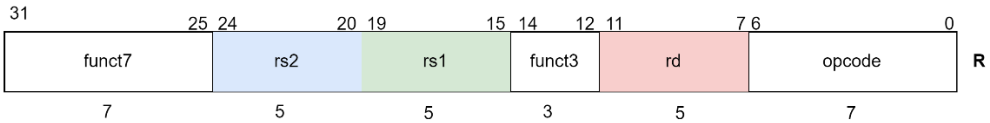
2.4.1 Integer computational instructions

Integer register-register operations

RV32i defines 10 arithmetic operations: addition, subtraction, logical operations, three shifts variants (left, right, and arithmetic), and two instructions where a Boolean value is set on a register based upon the result of a comparison, treating the registers as sign two's complement numbers (SLT) or as unsigned integers (SLTU).

- Arithmetic operations:
- Arithmetic: ADD, SUB.
- Bitwise: AND, OR, XOR.
- Shifts: SLL, SRL, SRA.
- Comparisons: SLT (rd=rs1<rs2? 1:0), SLTU (similar, but unsigned).

These arithmetic operations use the R-type instruction format. All operations read rs1 and rs2 registers, compute the values on those registers, and write the result into register rd. The type of operation is selected by *func7* and *func3*.



7	5	5	3	5	7
0000000	src2	src1	ADD/SLT/SLTU	dest	OP
0000000	src2	src1	AND/OR/XOR	dest	OP
0000000	src2	src1	SLL/SRL	dest	OP
0100000	src2	src1	SUB/SRA	dest	OP

Figure 3: Register-Register Operations instruction structure[11].

Integer Register-Immediate instructions

A large portion of arithmetic operations use a constant as an operand. Taking that in consideration, an immediate variant of all the arithmetic operations was added, excluding the subtraction instruction because an equivalent result can be obtained with the add immediate instruction. There are 9 register-immediate operations, and they use I-type format instructions.

- Arithmetic Immediate operations:
- Arithmetic: ADDI.
- Bitwise: ANDI, ORI, XORI.
- Shifts: SLLI, SRLI, SRAI.
- Comparisons: SLTI, SLTUI.

All operations read register rs1, execute the pretended operation between the source register and the immediate, and store the value to rd register. To simplify the implementation, the immediate SRA is always sign-extended even when the operation is logically doing something that's unsigned.

The I type encoding uses a 12bits wide immediate, which is smaller than the 16bits wide immediate used on the MIPS ISA. This downsizing release a large amount of encoding space.

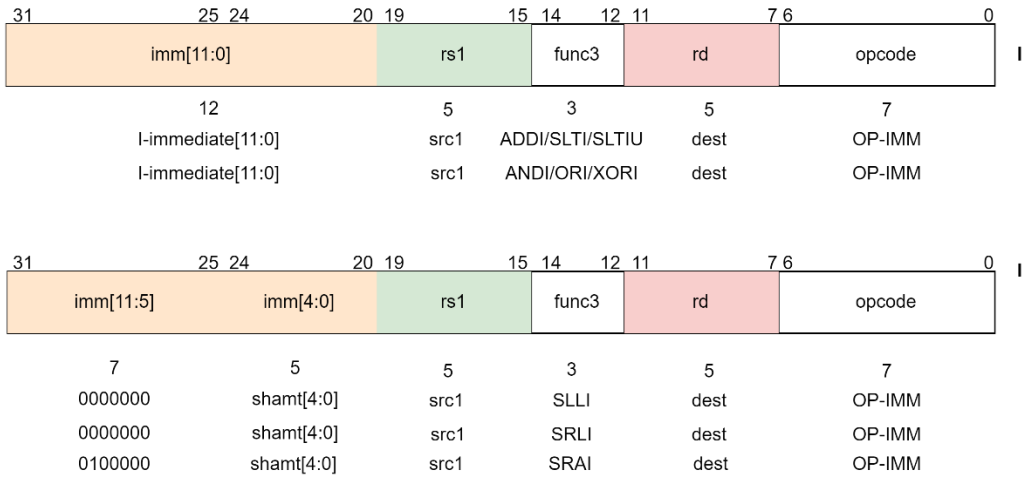


Figure 4: Register-Immediate Operations instruction structure [11].

LUI (load upper immediate) and AUIPC (add upper immediate to pc) are two U-type instructions that provide a global variable addressing for some other operations. Both instructions place the 20 most significant bits of the U-immediate in the top of the destination register rd, filling the lowest 12 bits with zeros. What differ between them is that the LUI instruction is used to build 32-bit constants and the AUIPC is used to build pc-relative addresses. LUI instruction can be used in conjunction with one of the I-type instructions to form a complete 32-bit address.

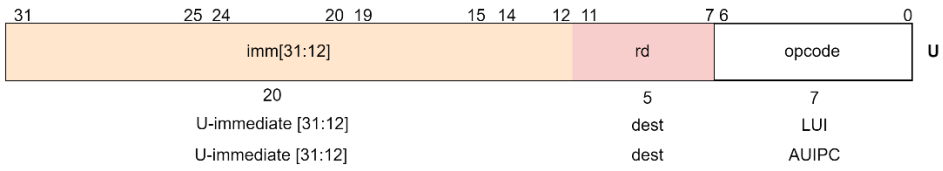


Figure 5: LUI (load upper immediate) and AUIPC (add upper immediate to pc) instructions [11].

2.4.2 Load and Store instructions

RV32I defined five load instructions, three for each of the data type sizes between 8 bits and 32 bits, being load byte, load half word, and load word. Since it's impossible to make a partial register update, a full register write is needed every time a load instruction is perform. An unsigned variant instruction was also added into the byte and half word loads.

The LW instruction loads a 32-bit value from memory into rd. LH loads a 16-bit value from memory, sign-extends it to 32-bits and store in rd. LHU loads a 16-bit value from memory, zero extends to 32-bits before storing in rd. LB and LBU are defined analogously for 8-bit values.

The load operation uses the I-type format just like the arithmetic instructions. The register rs1 is added to the immediate to produce an effective address, the memory is referenced at that address and the sign, or zero extension is carried if necessary, writing the value in rd to finish the operation.

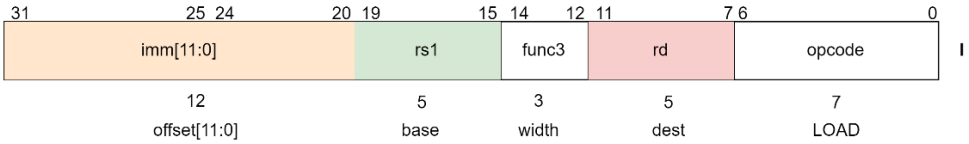


Figure 6: Load instruction [11].

The RV32I implements 3 Store instructions storing word (SW), halfword (SH), and byte (SB) size values from the low bits of register rs2 to memory. In case the of 16bit and 8bit values, they are sign-extended to 32bits as in the Load instruction. The Store instruction is implemented in the S-type format.

The register rs1 is added to the immediate to generate an effective address taking the least significant bits of the register and storing them to memory. In this case, some immediate shuffling is needed. The Load instruction has the bit 4 down to 0 of the immediate were in bits 24 through 20, although the Store uses bits 11 through 7 to the low-order bits of the immediate. The reason for this is to maintain the rs register specifier placed in the instruction, simplifying the data path design with the cost of the need to shuffle the immediate.

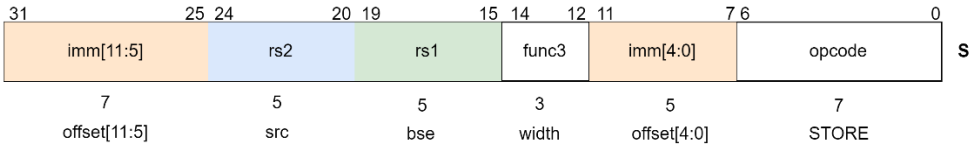


Figure 7: Store instruction [11].

2.4.3 Control transfer instructions

Conditional branches

All branch instructions use the B-type instruction format. The 12-bit immediate encodes signed offsets in multiples of 2 bytes. The offset is sign-extended and added to the address of the branch instruction to give the target address. The instructions in the base isa are always four bytes wide. There is no need to encode the zero bit since odd-number addresses don't align with the instructions. In this case, the branches have twice the displacement, allowing to an address range of ±4 KiB. Risc-v isa extensions don't have the constraint that all instructions are 32 bits, so if one instruction is 16-bit wide this ability to branch to arbitrary 16-bit offsets is needed.

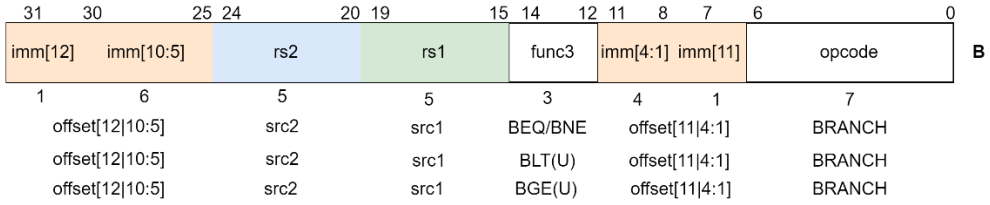


Figure 8: Conditional Branches instruction [11].

These branch instructions read registers rs1, rs2 and perform a comparison on then. If that comparison is true, it increments the PC by the immediate otherwise just falls through to the next instruction. The comparison options available are equality (BEQ), inequality (BNE), and magnitude comparisons like less than (BLT) and greater than (BGT) which can both use signed and unsigned comparisons (BLTU and BGEU, respectively).

Unconditional branches

The jump and link (JAL) instruction uses the J-type format. It encodes a signed 20bit offset immediate. The offset is sign-extended and added to the address of the jump instruction to form the jump target address allowing to jump within a megabyte of the current PC in either direction. The register rd is set with the value of the instruction after the JAL. The standard software calling convention uses x1 as the return address register and x5 as an alternate link register. If rd is set to x0, which is hardwired to 0, the JAL instruction turns into a simple direct jump instruction.

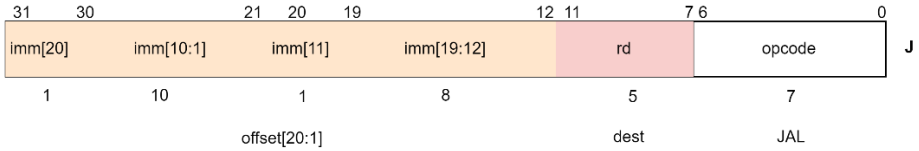


Figure 9: Unconditional Branches instruction – JAL (jump and link) [11].

The indirect jump instruction JALR (jump and link register) is used for returning from functions and uses the I-type encoding. The target address is obtained by adding the sign-extended 12bit immediate to the register rs1, then setting the least-significant bit of the result to zero. The address of the following instruction is written to register rd.

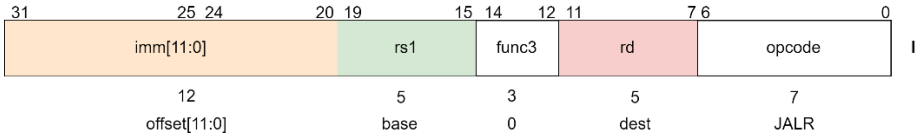


Figure 10: Unconditional Branches instruction – JALR (jump and link register)[11].

2.4.4 Memory ordering instructions

A FENCE instruction provides I/O and memory accesses ordering. The combination of memory reads (R), device input (I), device output (O), and memory writes (W) should be ordered with respect to anu combination of the same.

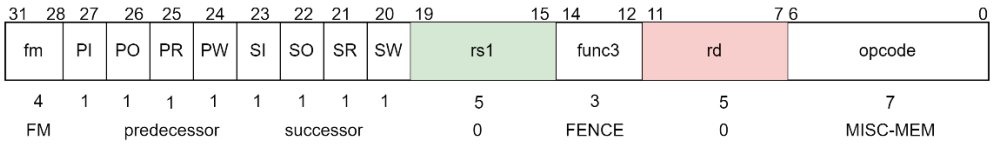


Figure 11: Memory Ordering instruction – FENCE [11].

FENCE.TSO is an optional instruction encoded as a FENCE instruction that orders all load and store operations in its predecessor set before all memory and store, respectively, operations in its successor set.

The FENCE.I synchronizes the data and the instruction stream. By default, in risc-v if a store is done to some address and then jump to address it's not guaranteed that the processor lacked will fetch the stored data.

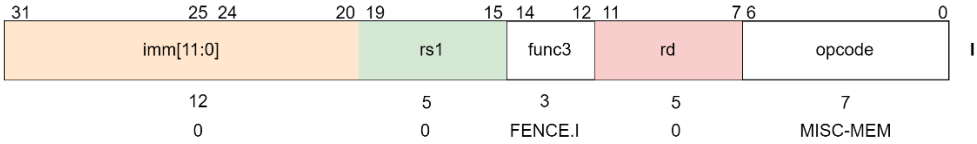


Figure 12: Memory Ordering instruction – FENCE.I [11].

2.4.5 Environment Call and Breakpoints

Encoded with the I-type instructions format, SYSTEM instructions are used to access system functionality that might require privileged access. The ECALL and EBRAKE instructions interact with the OS and the debugger. ECALL is used to make a service request to the execution environment and the EBRAKE instruction is used to return control to a debugging environment.

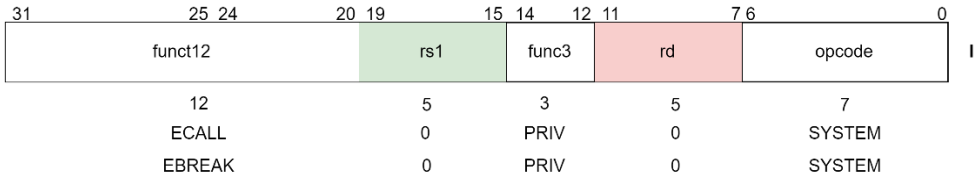


Figure 13: 2.4.5Environment Call and Breakpoints instruction [11].

2.5 RV64I base integer instruction set

The RV64I expands the integer registers to 64 bits wide supporting 64-bit addressing. This variant is targeted at systems that need more than a few gigabytes of RAM or use a large I/O space.

Aside from expanding the integer registers it's very similar to RV32I. Since most integer computational instructions operate on XLEN-bit values all the arithmetic instructions shown before now operate 64bit register length. Additional instruction variants were added to manipulate 32-bit values in RV64I, indicated by a 'W' suffix. All the "*W" instructions ignore the upper 32 bits of their inputs to produce a 32-bit signed value [11].

2.5.1Integer computational instructions

[11]Additional integer computational instructions exclusive to the RV64I were created. Besides the ADDIW instruction that adds the sign-extended 12-bit immediate to register rs1 and produces the proper sign-extension of a 32-bit result in rd, the other eight new instructions only operate on 32bit values and produce 32bit results:

- ADDW and SUBW
- SLLIW, SRLIW, and SRAIW.
- SLLW, SRLW, and SRAW.

2.5.2Load and Store instructions

In this ISA the LW instruction loads a 32-bit value from memory and sign-extends this to 64 bits before storing it in register rd. The LWU instruction, zero-extends the 32-bit value from memory.

A new STORE instruction, SD, was created to store 64bit values from the bits of register rs2 to memory [11].

2.6 RV128I base integer instruction set

RV128I was created to support larger address spaces although the authors from [11] specify that is not yet clear when an address space larger than 64 bits will be required. Data centers and some security applications could be able to take advantage of the extra address bits in the future.

The RV128I is built upon RV64I, with integer registers extended to 128 bits.

Integer computational instructions:

Most integer computational instructions are unchanged.

The "*W" integer instructions operate on 32-bit values in the low bits of a register are sign extend to 128 bits. The same ways as the new set of "*D" integer instructions operate on 64-bit values held in the low bits of the 128-bit integer registers and sign extend to 128-bit[11]

Shifts by an immediate (SLLI/SRLI/SRAI) are now encoded using the low 7 bits of the I-immediate, and variable shifts (SLL/SRL/SRA) use the low 7 bits of the shift amount source register.

Load and Store instructions

The LDU (load double unsigned) instruction was added along with new LQ and SQ instructions to load and store quadword values.

2.7 RV32E and RV64E base integer instruction set

The RV32E and RV64E (released in Jan. 2023) base integer instruction set is a reduced version of RV32I and RV64I, respectively, designed for embedded systems. The big difference in this variant is the reduced integer register count to 16 general-purpose registers, (x0–x15), where x0

is a dedicated zero register. As mentioned in [11], this reduction allows for a 25% downsizing on the core area corresponding to a core power reduction making it ideal for embedded microcontrollers. There are also non-low-end implementations where this ISA could thrill, for example, in very deeply multi-threaded machines where the register file is dominant [11].

2.8 RISC-V extensions

The RISC-V ISA compilers, by standard, separate the architecture into two parts. One to accommodate the RISC-V-specific extensions and the other to custom software libraries.

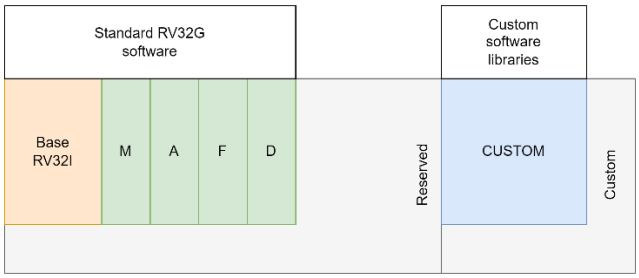


Figure 14: RISC-V specific and custom extensions organization.

RISC-V ISA is divided into non-privileged ISA and privileged ISA. The non-privileged ISA includes the basic integer instruction set and several non-privileged ISA extensions. The privilege ISA was created to run operating systems and support hardware virtualization. In this section, the non-privileged ISA extensions, and privileged ISA extensions [11].

M-EXTENSION

The M-extension is the integer multiplication and division instruction set extension. It offers a range of five multiplication and eight division instructions, enabling the user to efficiently multiply or divide values stored in two integer registers. It can operate with signed and unsigned integers[11], [14].

A-EXTENSION

The A-extension set extension, also known as atomic instruction, has two types of atomic instruction: atomic fetch-and-op memory and load-reserved/store conditional instructions, that read, modify, and write memory atomically to synchronize several RISC-V harts in the same memory space [11], [14].

F-EXTENSION

The F-extension implements instructions for single-precision floating-point operations that are a direct implementation of the IEEE 754-2008 standard [15]. It includes a floating-point control

and status register and an extra thirty-two floating-point registers, each with a width of thirty-two bits [11], [14].

D-EXTENSION

The D-extension supports double-precision floating-point operations on 64bit registers. The D-extension depends on the F-extension, using similar types of instructions [11], [14].

G-ABBREVIATION

The first four released standard extensions (M, A, F, and D) are also the most frequently used. Since the naming of any RISC-V implementation variant is made by the size of the Xlen plus the letter corresponding to the included extensions set, for example, "RV32IMAFD" witch represent a 32bit base ISA with the multiplication, atomic, and two floating point extension. For this specific case, an abbreviation of a single letter "G" was created to represent the standard general-purpose ISA [11], [14].

Q-EXTENSION

The Q-extension is the quad-precision binary floating-point ISA extension, meaning that the floating-point register is extended to 128-bit. This includes a new quad-precision load, store, convert, move instructions, and quad-precision floating-point classify, compare, and computational instructions. The D-extension is a prerequisite for using Q-extension [11], [14].

C-EXTENSION

The C-extension is the compressed instruction-set extension. Provides a shorter 16-bit versions of common 32-bit RISC-V, making it possible to freely intermixed 16-bit instructions with 32-bit instructions. It can reduce the size of static and dynamic code [11], [14].

COUNTERS-EXTENSION

The Counters-extension provides a set of up to 32x64bit counters, which can be accessed through specific read-only CSR registers with unprivileged XLEN. The first three counters: CYCLE, TIME, and INSTRET, are used for cycle counting, real-time clock, and instruction retirement. The remaining counters can be used for programmable event counting [11], [14].

B-EXTENSION

The B-extension is a bit manipulation instruction-set extension. It consists of a set of instructions (zba, zbb, zbc, zbs) that can support bit manipulation, including bit counts, shifts, rotations, insertions, extractions, and permutations [11], [14].

J-EXTENSION

The J-extension is designed to support dynamically translated languages like Java and JavaScript. These languages are typically implemented via dynamic translation. The J-extension aims to provide additional instructions to support dynamic checks, garbage collection, and JIT acceleration [11], [14].

P-EXTENSION

The P-extension is the packed single instruction multiple data (Packed-SIMD) instruction set extension. Supports efficient parallel data computing for audio, voice, and image processing applications. P-extension is mainly designed for embedded or DSP devices [11], [14].

V-EXTENSION

The V-extension is the vector ISA extension that adds a set of vector registers and vector operation instructions to the basic scalar RISC-V ISA. Supports Vector Length Agnostic SIMD operations that decouple the vector length from the compiled code. The same executable can run on processors with different implementations of vector length. Currently, V-extension is a research hotspot in both industry and academia [11], [14].

ZAM-EXTENSION

The Zam-extension a misaligned atomic instruction set extension. It adds standardized support for misaligned atomic memory operations (AMOs) to the A-extension. By utilizing Zam-extension, misaligned AMOs need only execute atomically regarding other accesses to the same address and size [11], [14].

ZTSO-EXTENSION

The Ztso-extension is the total store ordering instruction set extension. A basic memory consistency model is defined in the official RISC-V specification called RISC-V weak memory ordering (RVWMO), providing more freedom for hardware implementation and performance optimization. The Ztso-extension provides a RISC-V total store ordering (RVTSO) memory model [11], [14].

2.9 RISC-V cores

Over time, the number of processors based on the RISC-V ISA has grown. It is possible to find RISC-V ISA in microcontrollers, coprocessors, and multicore SoC capable of running an OS like Linux. The following RISC-V cores will be analyzed:

- PicoRV32.
- Rocket Chip.
- BOOM.
- Ri5cy.
- SweRV.

2.9.1 PicoRV32

PicoRV32 [16] is a 32-bit RISC-V core written in Verilog and suitable for FPGAs implementation. It supports the following ISA extensions:

- Integer Reduced Instructions (RV32E);
- Integer (RV32I);
- Integer Compressed (RV32IC);
- Integer Multiplication and Division (RV32IM).

The core exists in three variants, with different interfaces to connect peripherals:

- Picorv32, provides a simple native memory interface to be used in simple applications.
- Picorv32 axi, features an Advanced Xtensible Interface (AXI) Lite Master interface, useful for applications that use AXI to connect peripherals.
- Picorv32 wb, provides a Wishbone master interface.

A maximum core frequency of 714 MHz can be achieved in the Xilinx Virtex UltraScale+ board with the xcvu3p-ffvc1517-3-e FPGA.

2.9.2 Rocket chip

The Rocket Chip [17] is an open-source SoC generator created at the University of California, Berkeley. The Chisel hardware construction language embedded in Scala is used to generate object-oriented cores, caches, and interconnections for a complete SoC. The Chisel generates fast cycle-accurate C++ simulators and low-level Verilog Register Transfer Level (RTL) for FPGA or ASIC .

The Rocket Chip is a library of generators that can be configured and connected in different ways, originating different SoC designs. It can generate an in-order or an out-of-order core, Rocket, and BOOM, respectively, which can be attached to coprocessors using an interface named RoCC [8].

The Rocket [17] is a 5-stage in-order core supporting the RV32I and RV64I ISAs. It is compatible with the M, A, F, and D extensions. This core allows configuring the number of floating points, pipeline stages, caches, and TLB sizes. It has a configurable branch prediction with support for:

- branch target buffer (BTB);
- branch history table (BHT);
- return address stack (RAS).

When used with the RoCC interface, the Rocket can control external coprocessors and accelerators such as:

- vector processors;
- crypto units;
- image processing units.

It supports FPGAs implementations but is primarily used for tests and simulations.Currently, Rocket is being used by SiFive, a company created by several RISC-V authors, which produces RISC-V processors based on Rocket.

2.9.3 BOOM

BOOM [18] is an out-of-order core that implements the RV64G ISA built to use the Rocket Chip generator infrastructure. Some modules created for Rocket are also compatible with BOOM, including the caches, the functional units, and the TLBs.

BOOM is a 10-stage processor, but some of them are combined:

- Fetch;
- Decode;
- Register Rename;
- Dispatch;
- Issue;
- Register Read;
- Execute;
- Memory;
- Write-Back;
- Commit phases.

BOOM also supports branch speculation: each instruction in the pipeline receives a tag that marks which branch is responsible for the instruction being executed. If the branch was mispredicted, all the instructions that depend on that branch are removed from the pipeline.

BOOM is primarily optimized for ASIC [18] but also usable on FPGAs. It supports the FireSim flow, an open-source FPGA-accelerated cycle-accurate hardware simulator that runs on FPGAs on AWS EC2 F1 instances, running at 90+ MHz.

2.9.4 RI5CY

RI5CY [19] is a 4-stage in-order 32-bit RISC-V processor core extended to support multiple additional instructions, including hardware loops, post-increment load, store instructions, and additional ALU instructions not part of the standard RISC-V ISA. Figure 1 shows a block diagram of the core.

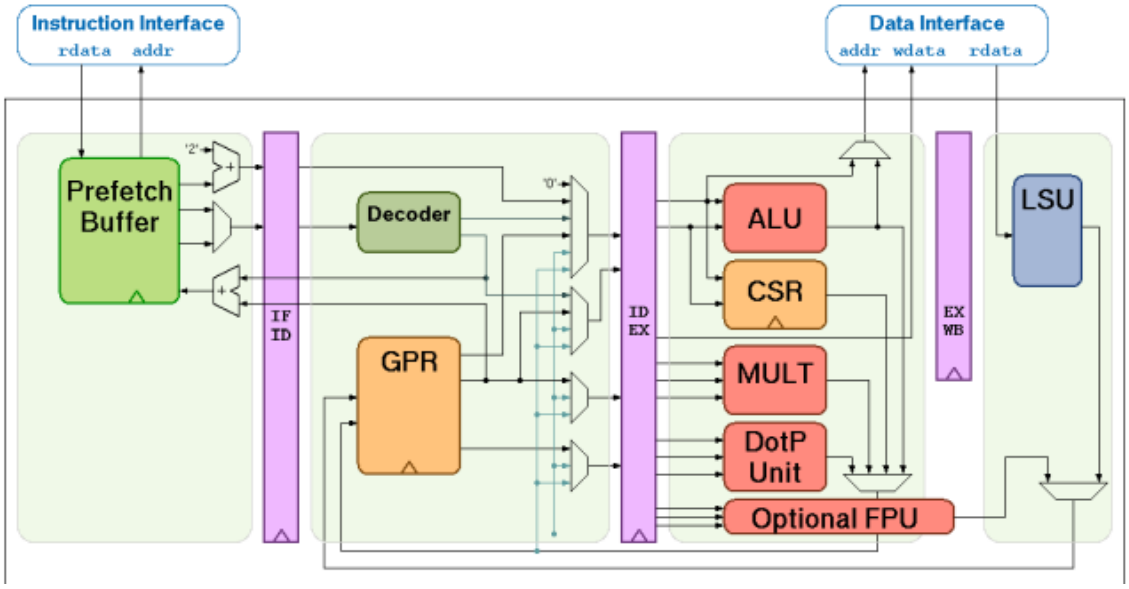


Figure 15: RI5CY block diagram from [20].

RI5CY supports the following instructions:

- Full support for RV32I Base Integer Instruction Set.
- Full support for RV32C Standard Extension for Compressed Instructions.
- Full support for RV32M Integer Multiplication and Division Instruction Set Extension.
- Optional full support for RV32F Single Precision Floating Point Extensions.
- PULP specific extensions
    - Post-Incrementing load and stores.
    - Multiply-Accumulate extensions.
    - ALU extensions.
    - Hardware Loops.

Floating-point support can be enabled by setting the parameter FPU of the top level file "riscv_core" to one. It will instantiate the FPU in the execution stage, and also extend the register file to host floating-point operands and extend the ALU to support the floating-point comparisons and classifications. ASIC and FPGA synthesis are supported for RI5CY.

2.9.5 SweRV

Western Digital developed three RISC-V cores [21]:

- SweRV EH1, preferred for its high performance per clock cycle;
- SweRV EH2, an expansion of the EH1 with a dual-threaded capability added to improve performance;
- SweRV EL2, which is a smaller and less powerful core.

The System on Chip (SoC) used in the referred RVfpga package, called SweRVolfX, is built on top of the SweRV EH1 Core Complex developed by Western Digital [23].

The SwerRV EH1 is a 32-bit RISC-V CPU core with integer, compressed instructions, and integer multiplication and division extensions implemented. All the I/O peripherals are connected to the CPU Core thru an AXI to Wishbone Bridge.

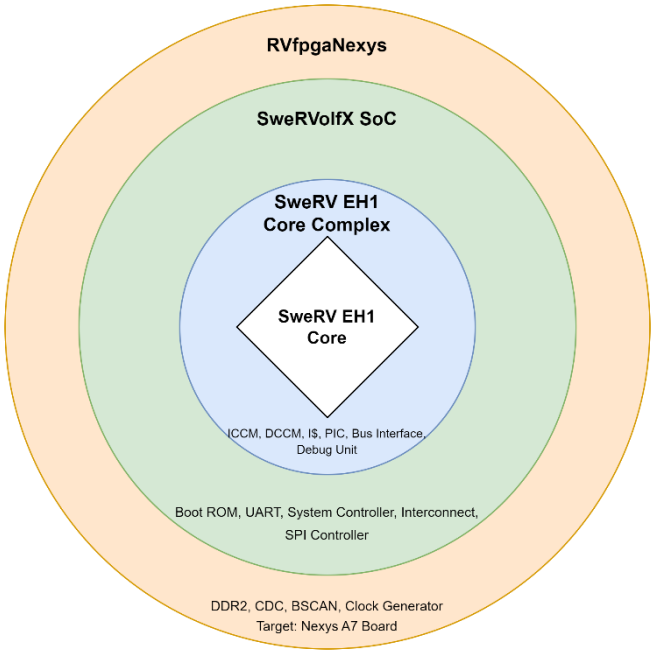


Figure 16. RVfpga System Hierarchy adapted from [21].

Western Digital provides an extension to the EH1 core, called the SwerRV EH1 Core Complex [23], which adds elements like:

- two tightly coupled to the core memories for low-latency access;
- one for data (DCCM) and the other for instructions (ICCM);
- 4-way instruction set cache with EEC or parity protection;
- Programmable Interrupt Controller (PIC);
- Core Debug Unit and a four-system bus interface.

The SweRVolf SoC includes a Boot ROM, a UART, a System Controller and an SPI controller. Besides that, the RVfpga implementation adds functionalities such as a second SPI controller, a GPIO controller, a PWM, Timer, a Counter module, and a controller for interfacing with 8-digit 7- Segment Displays .

Capítulo 3
Platforms and development tools

As stated at the beginning of this dissertation, one of the objectives of this work is to implement a functional RISC-V softcore on an FPGA with added capabilities.

This chapter will describe the software tools and hardware necessary to implement, simulate, and verify the HDL code from a RISC-V ISA base core. After that, an I2C temperature sensor will be used as an example of the steps needed to create a new function block in Verilog and how to verify it. At the end of the chapter, the SweRVolfX, which is a RISC-V SoC targeted to the Digilent Nexys A7 FPGA board, will be implemented on its two variants: on the FPGA (RVfpganexys), and with a simulation software (RVfpgaSim).

3.1 Software

VIVADO

Vivado was introduced in April 2012 as an integrated design environment (IDE) with system-to-IC level tools. It's built on a shared base data model with a common debug environment [22]. Vivado includes:

- electronic system level (ESL) design tools for synthesizing and verifying C-based algorithmic IP;
- packaging for algorithmic and RTL IP reuse;
- IP stitching and systems integration of all types of system building blocks;
- the verification of blocks and systems.

VSCode and PlatformIO

PlatformIO is an integrated development environment (IDE) built as an extension of Microsoft's Visual Studio Code (VSCode). PlatformIO is cross-platform and includes a built-in debugger.

Verilator

Verilator is an open and free HDL simulator that accepts synthesizable Verilog or SystemVerilog. It is used in industry and academia and provides out-of-the-box support from ARM and RISC-V vendor Ips [21].

Icarus Verilog

Icarus Verilog is a Verilog simulator and synthesis tool. It operates as a compiler, compiling source code written in Verilog into some target format. The compiler can generate an intermediate form called vvp assembly for batch simulation [23].

GTKwave

Gtkwave is a waveform analyzer and is the primary tool used for visualization. It provides a method for viewing analog and digital data simulation results, allowing various search operations and temporal manipulations. It can save partial results from a full simulation dump and generate PostScript and FrameMaker output for hard copy[24].

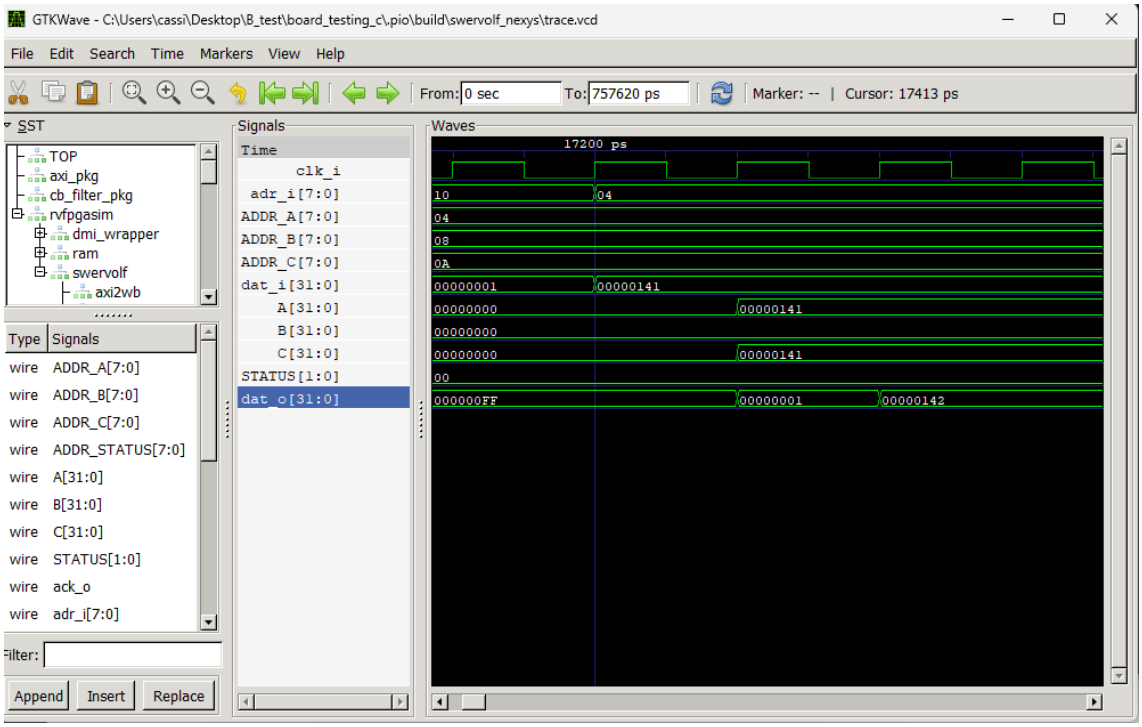


Figure 17: GTKwave.

3.2 Hardware

3.2.1 FPGA

With its origins in 1983, Field-Programmable Gate Arrays (FPGAs) have increased their popularity in recent years, being used as a platform for embedded systems development, given their flexibility and easy reconfigurability. One of the advantages of using FPGAs is their ability to implement custom processors, known as softcore processors, which can be tailored to meet specific system requirements[25].

The desired hardware architecture and its behavior can be described on a hardware description language (HDL), like VHDL, Verilog or System Verilog (an extension of Verilog), which is analyzed and synthesized with high-level synthesis (HLS) tools, such as Vivado HLS, Intel High-Level Synthesis Compiler or HDL Coder. Those tools allow the design to be described in a higher-level programming language and automatically generate the described hardware's register-transfer level (RTL), assuming that the architecture was properly designed and constrained for the FPGAs in use[26].

This approach allows any user to easily implement and modify a softcore processor like a RISC-V on an FPGA, giving the high level of abstraction and the ability to design and test the processor without the need for the time-consuming and complex task of designing manually the RTL implementation.

Nexys A7

The Nexys A7 board is a complete, ready-to-use digital circuit development platform based on the Artix-7™ FPGA from Xilinx with several built-in peripherals, including an accelerometer, temperature sensor, MEMs digital microphone, a speaker amplifier, and several I/O devices[21]

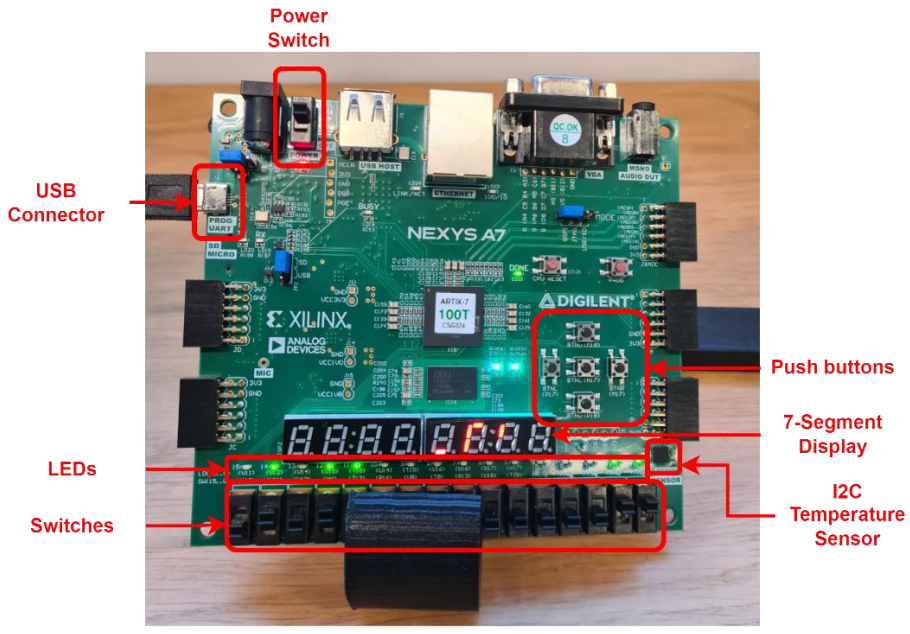


Figure 18: Digilent's Nexys A7 FPGA board's I/O interfaces.

Purchasing options

The Nexys A7 can be purchased with either a XC7A100T or XC7A50T FPGA. Both Artix-7 FPGAs have the same capabilities, but the XC7100T has a two times larger internal FPGA than the XC750T[27]. The differences between the two variants are summarized below:

Table 2 Nexys A7-100T and Nexys A7-50T table of content, from the Nexys A7 user Manual [27].

Product Variant	Nexys A7-100t	Nexys A7-50t
Look-up Tables (LUTs)	63400	32600
Flip-Flops	126800	65200
Block RAM	1188 Kb	600 Kb
DSP Slices	240	120
Clock Managment Tiles	6	6

Features

The Nexys A7-100T FPGA board includes the following interfaces and devices[21]:

- 128 MiB DDR RAM
- 128 Mibit SPI Flash Memory
- 8-digit 7-Segment Displays
- 16 Switches
- 16 LEDs
- Sensors and connectors, including a microphone, audio jack, VGA 25 port, USB host port, RGB-LEDs, I2C temperature sensor, SPI accelerometer, among other.
- Xilinx Artix-7 FPGA, which has the following features:
    - 15.850 Logic slices of four 6-input LUTs and 8 flip-flops.
    - 4.860 Kbits of total block RAM
    - 6 clock management tiles (CMTs)
    - 170 I/O pins
    - 450 MHz internal clock frequency

FPGA configuration

After power-on, the Artix-7 FPGA must be programmed before performing any functions. The FPGA can be programmed in one of the following ways[21], [27]:

1. A PC can use the Digilent USB-JTAG circuitry to program the FPGA whenever the power is on.
2. A file stored in the nonvolatile serial (SPI) flash device can be transferred to the FPGA using the SPI port.
3. A micro SD card can transfer a programming file to the FPGA.

4. A programming file can be transferred from a USB memory stick attached to the USB HID port.

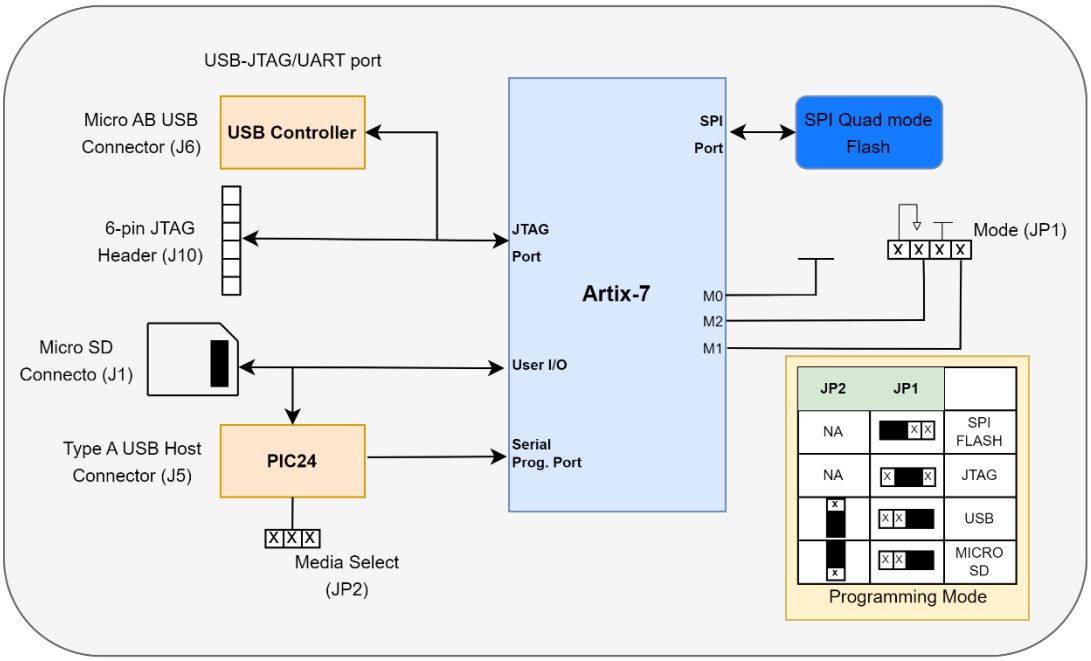


Figure 19: Artix-7 FPGA configuration, inspired on the NEXYS A7 user manual [27].

The FPGA configuration data is stored in files called bitstreams that have the .bit file extension. The ISE or Vivado software from Xilinx can create bitstreams from VHDL, Verilog, or schematic-based source files[21], [27].

Bitstreams are stored in SRAM-based memory cells within the FPGA. This data defines the FPGA's logic functions and circuit connections, and it remains valid until it is erased by removing board power, pressing the reset button attached to the PROG input, or writing a new configuration file using the JTAG port[21], [27].

3.3 Temperature sensor

The Nexys A7 contains a built-in temperature sensor, the ADT7420. This sensor uses an I2C interface to communicate with the system to which it is connected, in this case, to the FPGA. Despite being integrated into the board, the FPGA on the Nexys A7 doesn't have any I2C interface implemented. It is necessary to implement a module with the protocol in question to communicate not only the sensor temperature but with any other device connected to this data bus [27].

In the Nexys A7 reference manual [27], the communication interface between the Artix 7 and the temperature sensor is described. There are SCL and DAS signals, referred to in the I2C protocol, and two control signals. To establish communication with the sensor (slave), the Artix 7, which will be the master, must use address 0x4b, as specified in the reference manual.

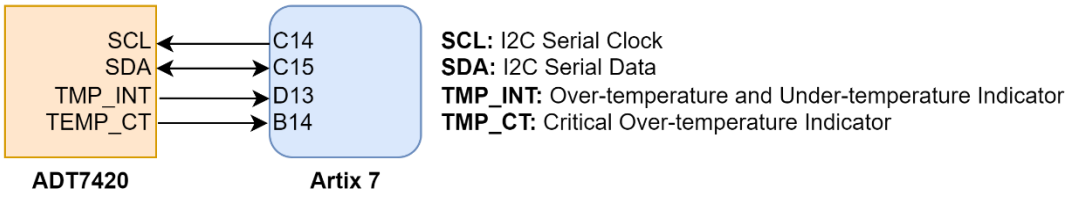


Figure 20**:** ADT7420 temperature sensor to ARTIX7 communication interface from NEXYS A7 user manual [27].

The moment it starts, the ADT7420 automatically starts reading temperature, its standard mode. By default, the device address points pair to the temperature MSB register. It is possible to read the temperature obtained from the sensor with just one reading of two bytes. The first byte contains the most significant value (MSB) of the data temperature and the second byte contains the least significant value (LSB). Both combined to form number with the temperature value.

In this scenario, the most significant bit of the MSB can be eliminated, as this is the sign bit in a two's complement value. The three least significant bits of the LSB can be equally ignored since they are status bits, as described in the ADT7420 sensor manual[28]. To obtain a reading in degrees Celsius (ºC) it's necessary to multiply the data received by 0.0625, which is equivalent to dividing by 16 or performing a shift right of 4bit, as shown in the following figure:

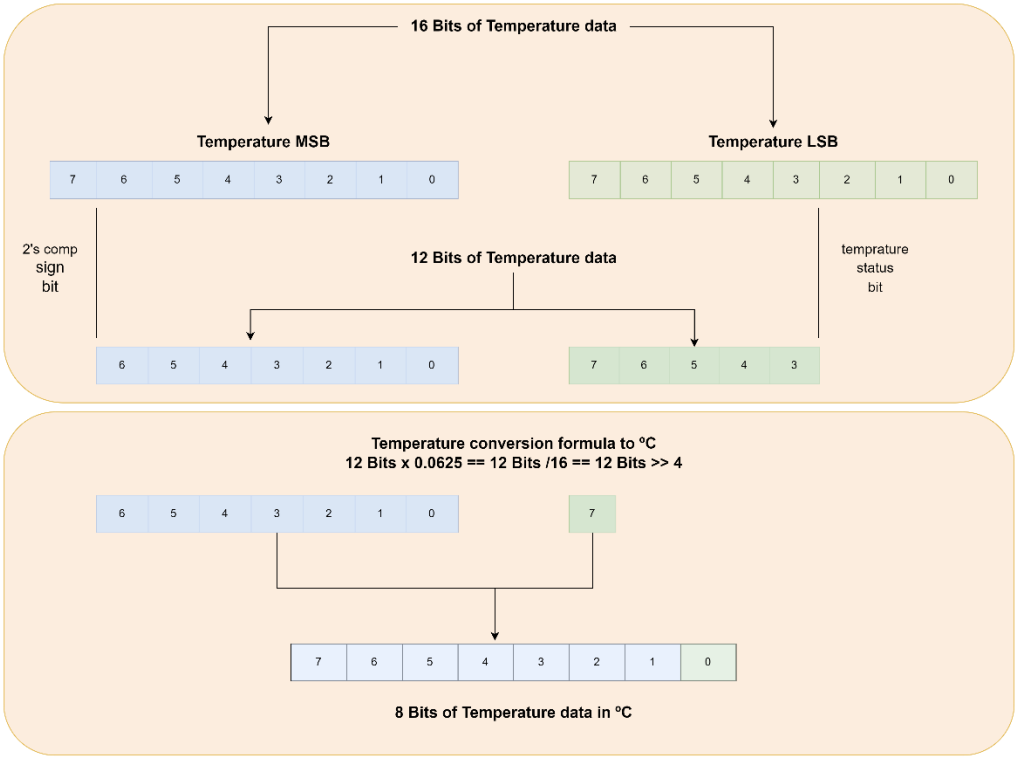


Figure 21: Manipulation of read data from the ADT7420 sensor. Convert 16bit into 8bit temperature data in °C.

As referred to in [28], the sensor takes 6ms to perform its first temperature reading after starting. The maximum frequency allowed on the clock line (SCL) is 400KHz.

To properly implement the I2C protocol, the following steps suggested in the sensor manual need to be complied with:

1. The START condition at the beginning is defined by a transition from high to low on the SDA line while SCL remains high.
2. The STOP condition at the end of the communication is signaled by the transition of low to high on the SDA line while SCL remains high.
3. The Master must generate the two acknowledgment (ACK) signals at the end of the reading, stating that no additional data is required.
4. MSB and LSB temperature log data are always separated by an ACK bit to zero.
5. The R/W bit is set to 1 to indicate that a reading will be taken.
6. The data placed on the SDA line must be updated before each signal clock and must remain intact during the period in which the signal SCL is high.

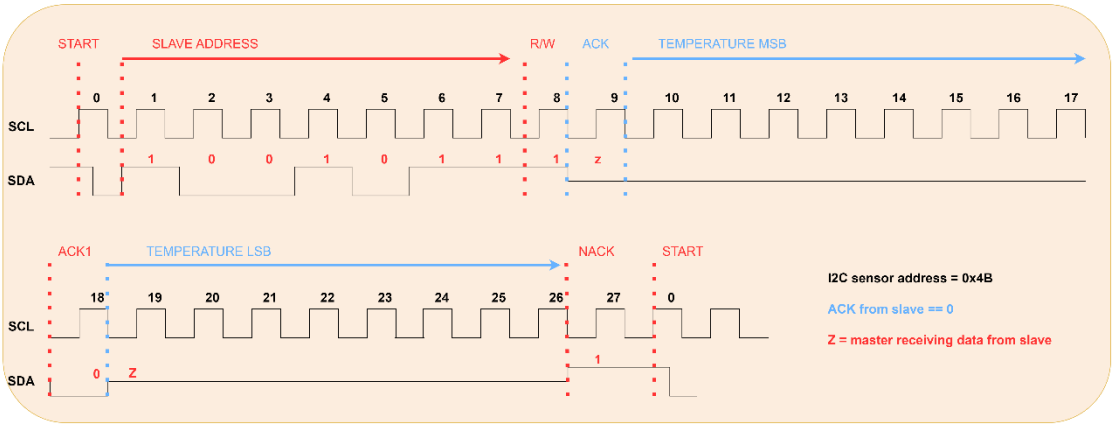


Figure 22 I2C protocol.

A reading is initiated by the Master when it transitions the SDA signal from high to low while the SCL line is high. Then the Master sends the address of the device (0x4B) with which it wants to communicate followed by a bit of reading/writing (R/W), which in this case will be 1 as it is intended to read data. The Master releases the SDA line so that the Slave can send a signal ACK, followed by the 8 bits of MSB. Then the Master takes control of the line SDA, placing it at the bottom to signal receipt of the message (ACK). The remaining 8 bits of the LSB of the temperature data are sent by the Slave. The transmission ends when the Master sends the NACK signal, placing the SDA line high.

3.3.1 I2C Verilog module implementation

Step 1. Write the RTL code

The I2C module RTL code will be written in Verilog code in the Visual Studio Code.

The communication with the temperature sensor will be done through an I2C block. The NEXYS A7 100MHz clock signal is input in the block that generates a 200KHz signal.

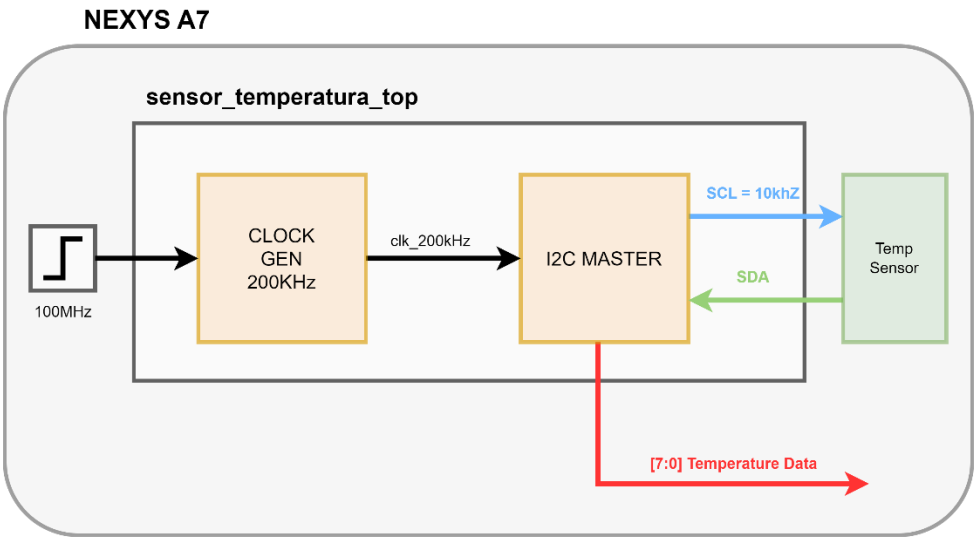


Figure 23: interaction between modules to integrate a functional I2C temperature sensor in the Nexys A7.

The CLOCKGEN_200KHZ module has as input the 100MHz clock signal from the NEXYS A7 and as output the 200KHz clock signal that is intended to be generated. An 8-bit register is created to be used as a counter, as well as a clk_reg signal as a buffer. Whenever the signal clk_100MHz input transitions from low to high, the counter is incremented. When the counter reaches the value of 249, it is set back to zero and the signal present in the clk_reg buffer is inverted. The clk_reg signal is connected to the clk_200Khz output.

```
1   `timescale 1ns/1ps
2
3   module clkgen_200KHz (
4           input clk_100MHz,
5           output clk_200KHz
6           );
7
8           reg [7:0]   counter = 8'b0;
9           reg clk_reg = 1'b1;
10
11          always @(posedge clk_100MHz) begin
12              if(counter ==249)begin
13                  counter <= 8'h00;
14                  clk_reg<=~clk_reg;
15              end
16              else
17                  counter<=counter+1;
18          end
19
20          assign clk_200KHz = clk_reg;
21
22      endmodule
```

Figure 24: Verilog code of a 100MHz to 200Khz clock signal generator.

The i2c_master module has as input the clock signal clk_200MHZ, generated in the module above described, and a reset signal. As output, there is the I2C SCL signal, an SDA_dir signal that works as indicated in the direction the data is being transmitted (from Master to Slave or the

opposite), and an 8-bit register with the temperature in °C. There is an inout signal, the SDA which It has these characteristics because it is both a data transmission and reception line.

The 200MHZ clock signal is used to time the data placement on the SDA line, since it is an asynchronous data line compared to the SCL line, as mentioned above and exemplified. A 10MHz clock signal is also generated, with a similar feature to the module above described and whose function is to be the SCL signal.

```
1   `timescale 1ns/1ps
2
3   module i2c_master(
4           input   clk_200KHz,
5           input   reset,
6           inout   SDA,
7           output [7:0]    temp_data,
8           output  SDA_dir,
9           output SCL
10          );
11
12          reg [3:0] couter = 4'b0;
13          reg clk_reg = 1'b1;
14
15
16  //*****************************************************************
17  // criar o clk de 10KHz
18
19          always @(posedge clk_200KHz or posedge reset)
20              if (reset) begin
21                  couter = 4'b0;
22                  clk_reg = 1'b0;
23              end
24              else
25                  if(couter == 9) begin
26                      couter <= 4'b0;
27                      clk_reg <= ~clk_reg;
28                  end
29                  else
30                      couter <= couter+1;
31
32
33
34          assign SCL = clk_reg;
35
```

Figure 25: Verilog code for the I2C module.

A simplified way to implement the i2c protocol is using a state machine where each state will correspond to a high transition of the SCL signal.

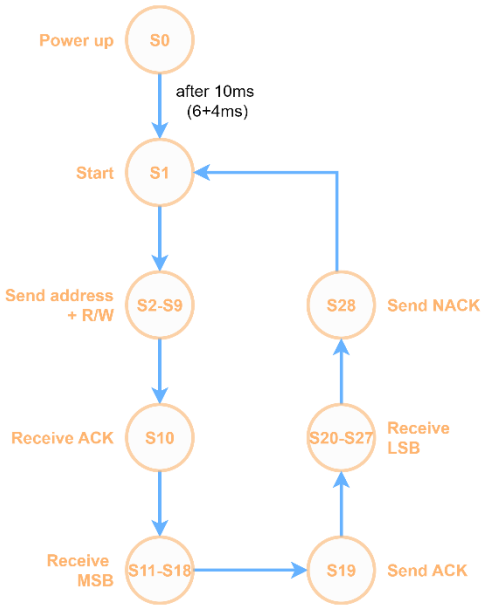
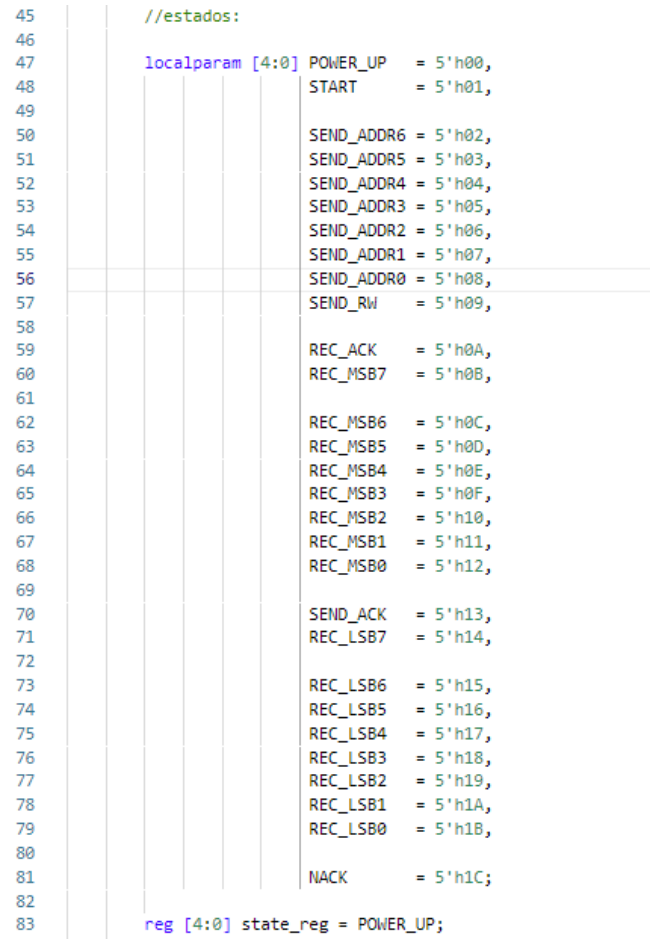


Figure 26: I2C state machine.

The sensor address as well as the R/W bit are parameterized in an 8-bit memory. Three 8-bit registers are created to store the MSB, LSB, and temperature in °C. In addition, a 12-bit register is also created that will be used as a counter of clock signals and which will serve as a time reference for the execution of each state. Local parameters with sequential values and with the name of each state are created. A status register is created, and the current state is placed in there.



```
45      //estados:
46
47      localparam [4:0] POWER_UP   = 5'h00,
48                       START      = 5'h01,
49
50                       SEND_ADDR6 = 5'h02,
51                       SEND_ADDR5 = 5'h03,
52                       SEND_ADDR4 = 5'h04,
53                       SEND_ADDR3 = 5'h05,
54                       SEND_ADDR2 = 5'h06,
55                       SEND_ADDR1 = 5'h07,
56                       SEND_ADDR0 = 5'h08,
57                       SEND_RW    = 5'h09,
58
59                       REC_ACK    = 5'h0A,
60                       REC_MSB7   = 5'h0B,
61
62                       REC_MSB6   = 5'h0C,
63                       REC_MSB5   = 5'h0D,
64                       REC_MSB4   = 5'h0E,
65                       REC_MSB3   = 5'h0F,
66                       REC_MSB2   = 5'h10,
67                       REC_MSB1   = 5'h11,
68                       REC_MSB0   = 5'h12,
69
70                       SEND_ACK   = 5'h13,
71                       REC_LSB7   = 5'h14,
72
73                       REC_LSB6   = 5'h15,
74                       REC_LSB5   = 5'h16,
75                       REC_LSB4   = 5'h17,
76                       REC_LSB3   = 5'h18,
77                       REC_LSB2   = 5'h19,
78                       REC_LSB1   = 5'h1A,
79                       REC_LSB0   = 5'h1B,
80
81                       NACK       = 5'h1C;
82
83      reg [4:0] state_reg = POWER_UP;
```

Figure 27: Verilog coding of a state machine parameterization.

At each transition from low to high of the reset signal, the state machine and its counter are set to zero. With each positive transition of the clock signal, the state machine counter is increased. Each machine state is executed based on the value placed in state_reg. The full code could be found on the Appendix A.

```
85          always @(posedge clk_200KHz or posedge reset) begin
86              if(reset) begin
87                  state_reg <= START;
88                  count <= 12'd2000;
89              end
90              else begin
91                  count <= count +1;
92
93                  case (state_reg)
94                      POWER_UP    :begin
95                                      if (count == 12'd1999)
96                                          state_reg <= START;
97                      end
98                      START       :begin
99                                      if (count == 12'd2004)
100                                         o_bit <= 1'b0;
101                                     if (count == 12'd2013)
102                                         state_reg <= SEND_ADDR6;
103                     end
104
```

Figure 28: Verilog code of a state machine.

Step 2. Create a test bench

To verify that the module is working correctly, a test bench is created. A test bench is a series of known input and output conditions that are set into the module under test. If the outputs from the module correspond to does expected on the test bench, the module is work as expected.

The I2C module test bench only uses de clock and reset signal to drive the i2c_master module as shown in the next figure.

C: > Users > cassi > Desktop > UBI > PORG_FPGA > I2C > i2c_master_TB.v

```
1   `timescale 1ns/1ps
2
3   module i2c_master_TB;
4       reg clk;
5       reg reset;
6       wire SDA;
7       wire SDA_dir;
8       wire SCL;
9
10
11      // "instancializar" o modulo i2c_master
12      i2c_master master(
13          .clk_200KHz(clk),
14          .reset(reset),
15          .SDA(SDA),
16          .SDA_dir(SDA_dir),
17          .SCL(SCL)
18          );
19
20      always #1 clk = !clk;
21
22      //criação do ficheiro para guardar as variáveis
23      initial begin
24          $dumpfile ("i2c_master.vcd");
25          $dumpvars(0,i2c_master_TB);
26      end
27
```

Figure 29: test bench for the i2c module.

Step 3. Simulation

Icarus Verilog is a Verilog simulator and synthesis tool that generates a dump file with all the values obtained from the test bench. The generated file is then used in GTKwave to reproduce all de signals from the module under test.

Terminal commands (Figure 30):

- iverilog command creates a simulation file base on the module and the test bench code files.
- vvp runs the simulation and generate the dump files in the .vcd format.
- gtkwave open the GTKwave tool and load the .vcd file.

```
PROBLEMS   OUTPUT   DEBUG CONSOLE   TERMINAL

*** These modules were missing:
        i2c_master referenced 1 times.
***
PS C:\Users\cassi\Desktop\UBI\PORG_FPGA\I2C> iverilog -o i2c_master i2c_master.v i2c_master_TB.v
PS C:\Users\cassi\Desktop\UBI\PORG_FPGA\I2C> vvp i2c_master
VCD info: dumpfile i2c_master.vcd opened for output.
i2c_master_TB.v:33: $finish called at 30000000 (1ps)
PS C:\Users\cassi\Desktop\UBI\PORG_FPGA\I2C> gtkwave i2c_master.vcd

GTKWave Analyzer v3.3.100 (w)1999-2019 BSI
```

Figure 30: Iverilog and GTKwave terminal commands.

Step 4. GTKwave data analyze

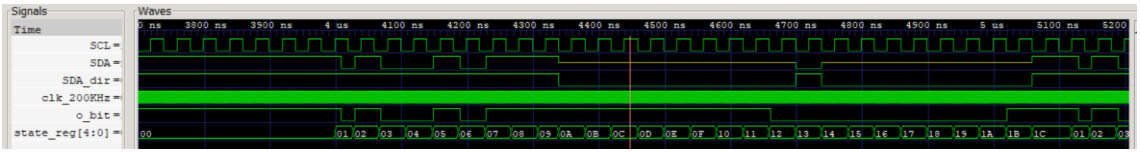


Figure 31: Signal analyze of the i2c module.

It's possible to observe the 10ms waiting time at state zero, in the state_reg, correspond of the first 1999 clk ticks. At the 2000th tick the state machi step up to the first state.

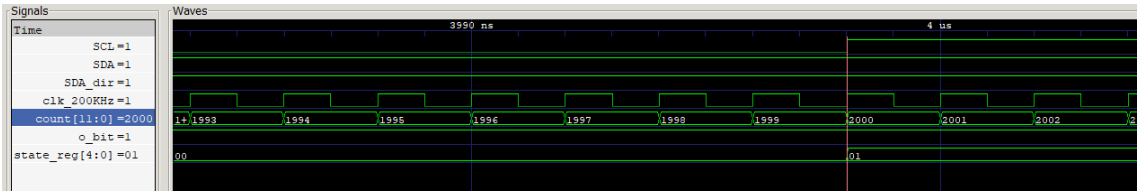


Figure 32: 10ms waiting time to start the temperature sensor.

The SCL clock at high and in the middle of the signal the master assumes the control of the SLA line, represented by the transition to low of the o_bit and the SDA lines. This is the start condition.

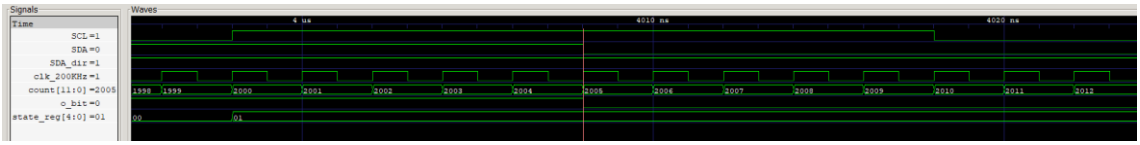


Figure 33: address bit setting on a middle of a SCL signal for data stability.

After the start condition state, the slave address is sent. The address bit needs to be for the entire high period of the SCL. For that reason, the bits are set in the center of the low points of SCL.

Figure 34: SDA line, controled by the Master, sending the slave address.

After sending the final address bit, the master gives up the SDA line, entering in a tri-state z, represented by the yellow color. The tri-states allows the slave to transmit its information.

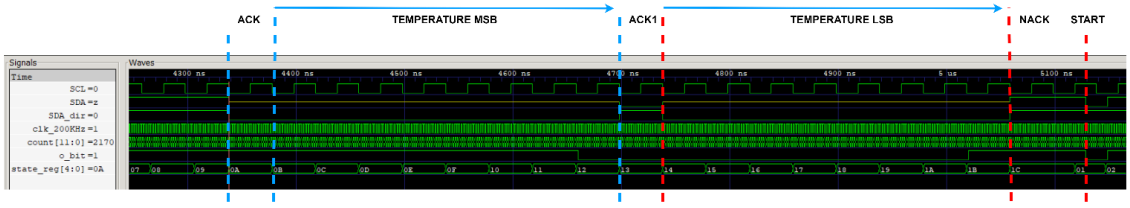


Figure 35: Slave controlling the SDA line.

After that the master send the no acknowledge which is sending SDA high. The state transitions back to start.

3.4 SweRVolfX SoC implementations

The follow-up section uses an implementation of the RISC-V architecture RVfpgaNexys,[21] which can be obtained from https://university.imgtec.com. The architecture was developed by Imagination for educational purposes.

The SweRVolfX SoC can run on the Nexys A7 FPGA board, whose configuration is referred to as RVfpgaNexys, or, in simulation, referred to as RVfpgaSim.

RVfpgaNexys is the same as SweRVolf Nexys except that the latter is based on SweRVolf. The main elements used by RVfpgaNexys are:

- Hardware programmed onto the FPGA:
    - SweRVolfX SoC
    - Lite DRAM controller
    - Clock Generator: the Nexys A7 board includes a single 100 MHz crystal oscillator used by the Lite DRAM controller. The frequency of this clock is scaled down to 50 MHz to use in the SweRVolfX SoC.
    - Clock Domain Crossing module
    - BSCAN logic for the JTAG

- Memory/Peripherals used in RVfpgaNexys from the Nexys A7 (or Nexys4 DDR) FPGA board:
    - DDR2 memory (accessed through the Lite DRAM controller mentioned above)
    - USB connection
    - SPI Flash memory
    - SPI Accelerometer
    - 16 LEDs and 16 Switches
    - 8-digit 7-Segment Displays

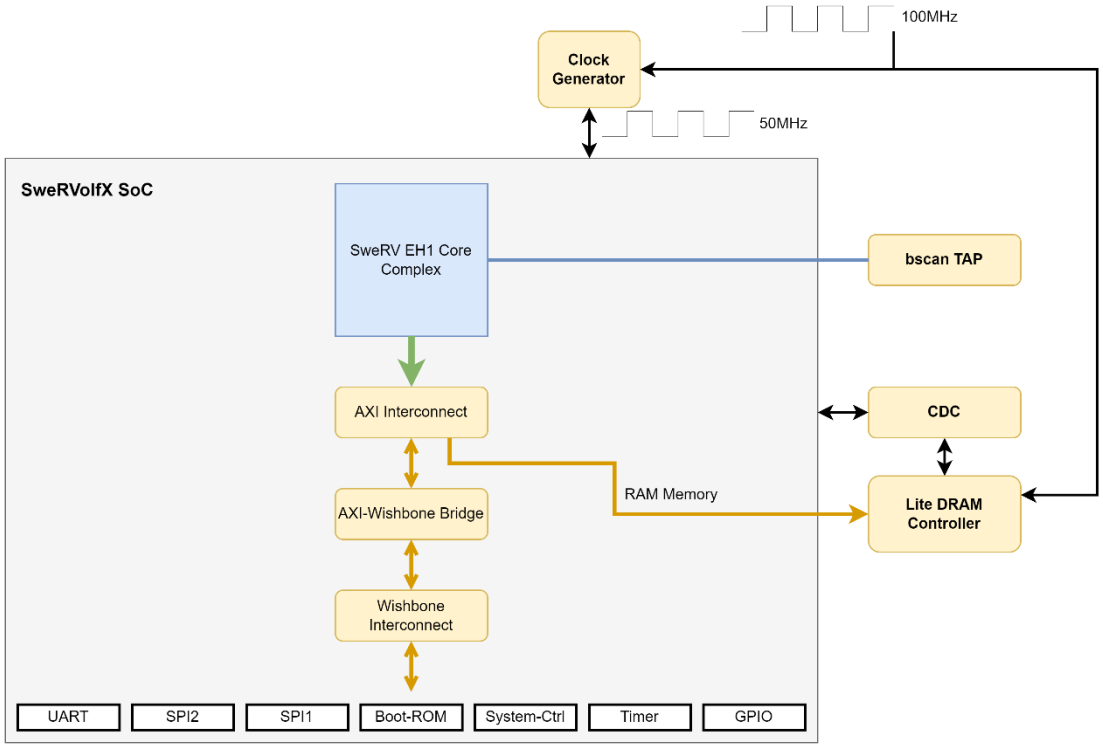


Figure 36: RVfpgaNEXYS.

3.4.1 C program for RVfpga

This section will show how to create a C project in PlatformIO that can run on the RVfpga System [21].

STEP 1. Create an Rvfpga project

After opening VSCode and the PlatformIO by clicking the icon, create a new project, as shown in the figure.

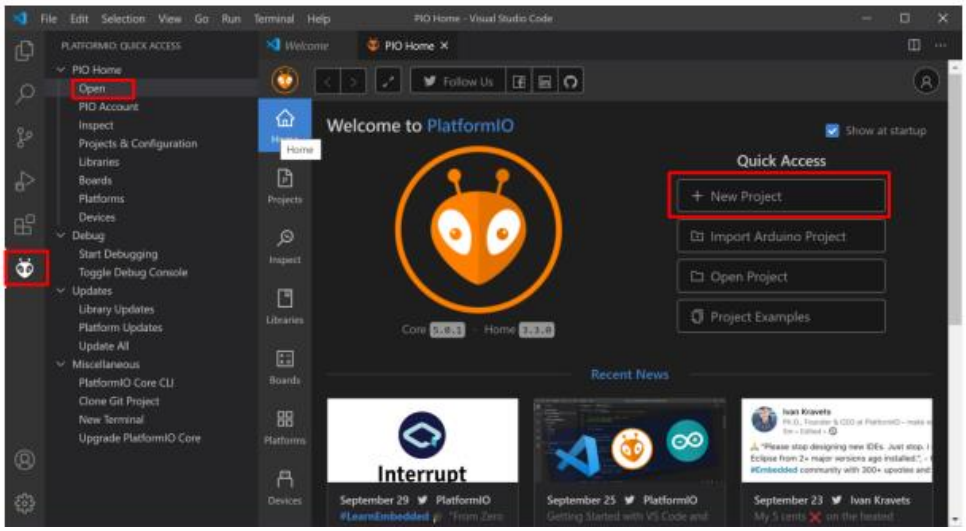


Figure 37: PlatformIO.

Name the project and choose the Board as RVfpga: Digilent Nexys A7. It is possible to choose a location for the new project by unclicking the User default location and inserting the desired one (Figure 38)

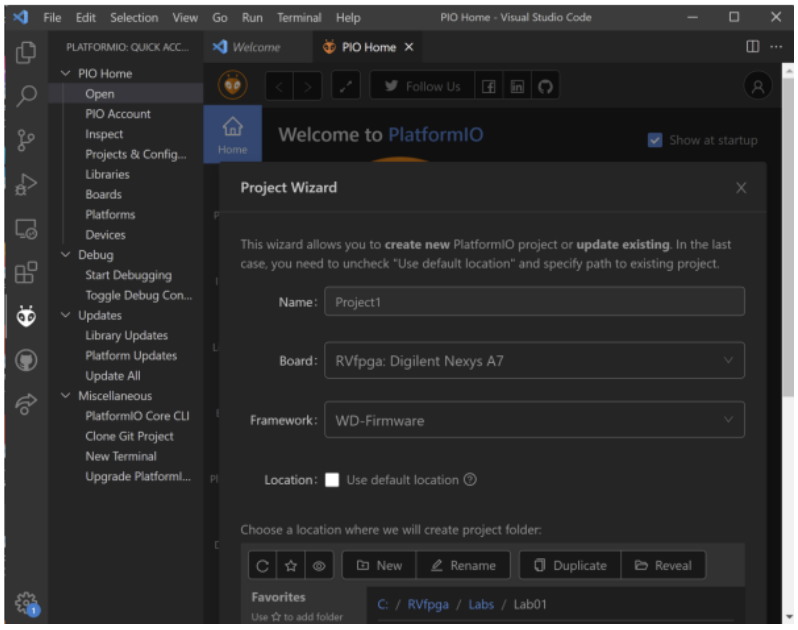


Figure 38: create a new project on PlatformIO.

To enable PlatformIO to find the bitstream file for the FPGA, add the following line to the platformio.ini file, as shown in the figure, and save the file.

board_build.bitstream_file = [RVfpgaPath]/RVfpga/src/rvfpganexys.bit

The Getting Started Guide provides the RVfpgaNexys bitstream.

STEP 2. Write a C program

Create a new file in the project's src folder, write the desired C code, and then save it.

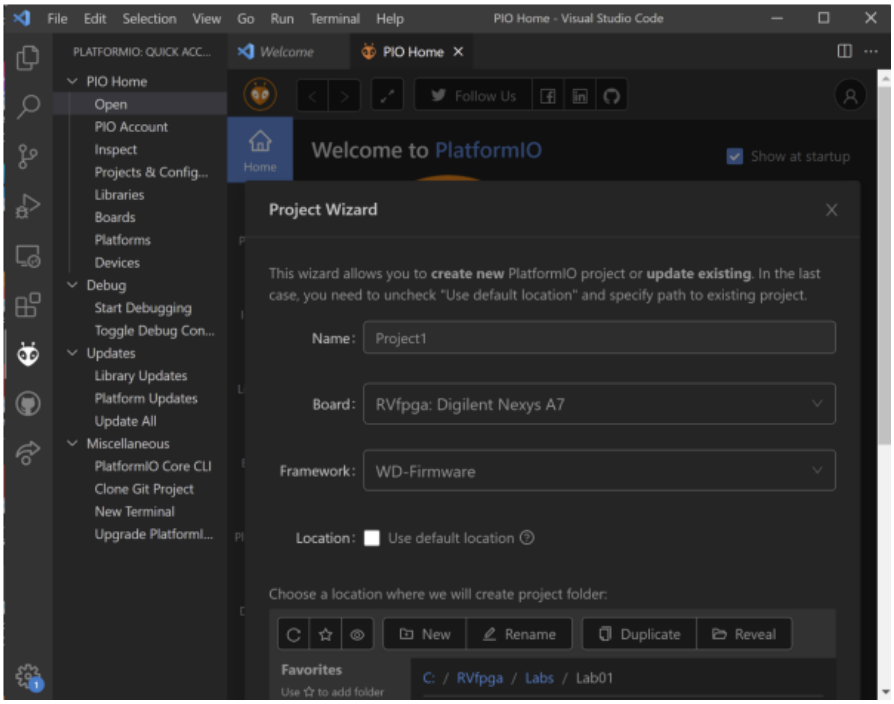


Figure 39: Select the destination forder for the new project on PlatformIO.

STEP 3. Upload Rvfpganexys onto Nexys A7 Fpga board

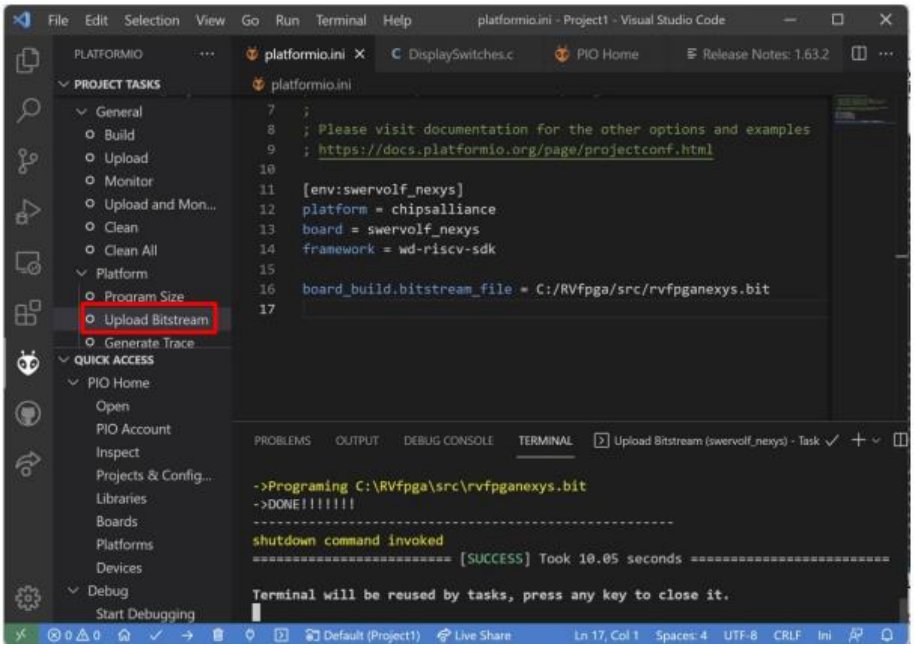


Figure 40: Upload Bitstream.

STEP 4. Compile, download, and run C program

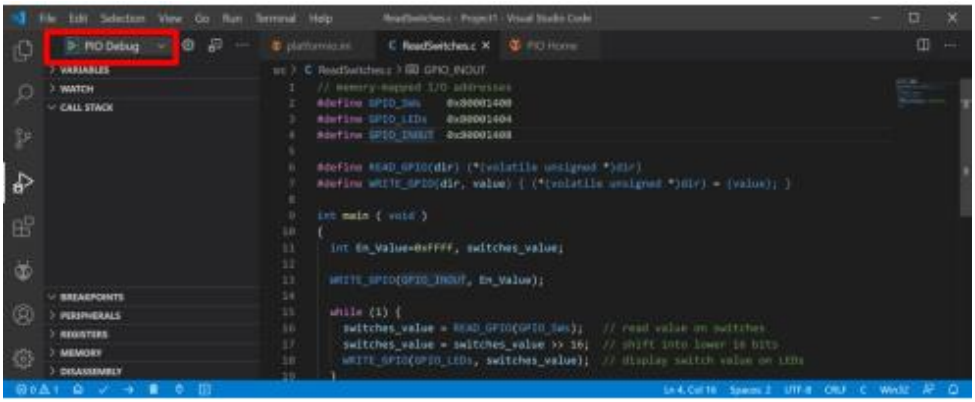


Figure 41: Run and Debug.

After uploading the bitstream, the RVfpgaNexys board is operational. Compile and upload the program, then run or debug it.

3.4.2RVfpganexys

In order to work with and modify the RVfpga System, it is necessary to compile a project that includes all of the Verilog, SystemVerilog, header, configuration, and text files that define the system.

This section will show how to create a Vivado project that targets the SweRVolfX SoC to Digilent's Nexys A7 FPGA board. Vivado will be used to build the RVfpgaNexys system using the RTL and the Verilog files that define the system.

By following these steps, build the RVfpgaNexys system and target it to a Nexys A7 FPGA board.

STEP 1. Open Vivado

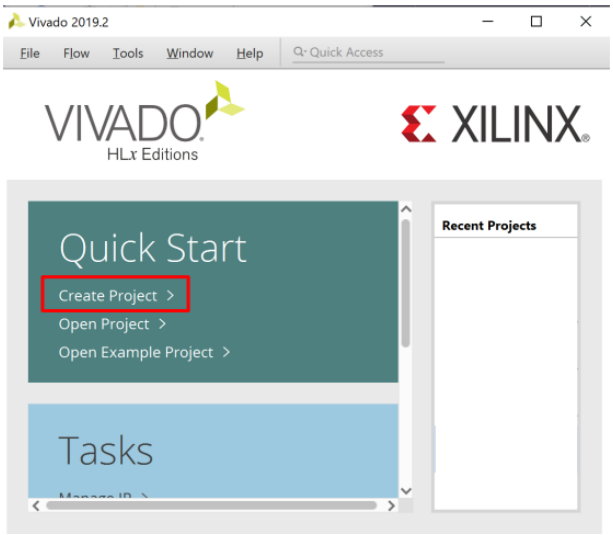


Figure 42: Vivado inicial page.

STEP 2. Create a new Rtl project

The Create a New Vivado Project Wizard will now open.

STEP 3. Add the Rtl source files and the constraint files

- Add RTL include files into the project
- Add sources from subdirectories

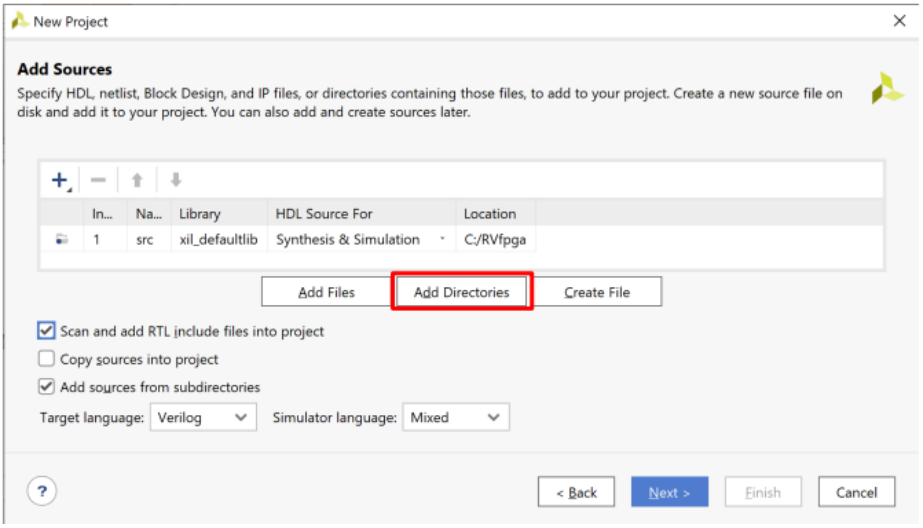


Figure 43: adding source files to a Vivado Project.

STEP 4. Select Nexys A7 as the target board

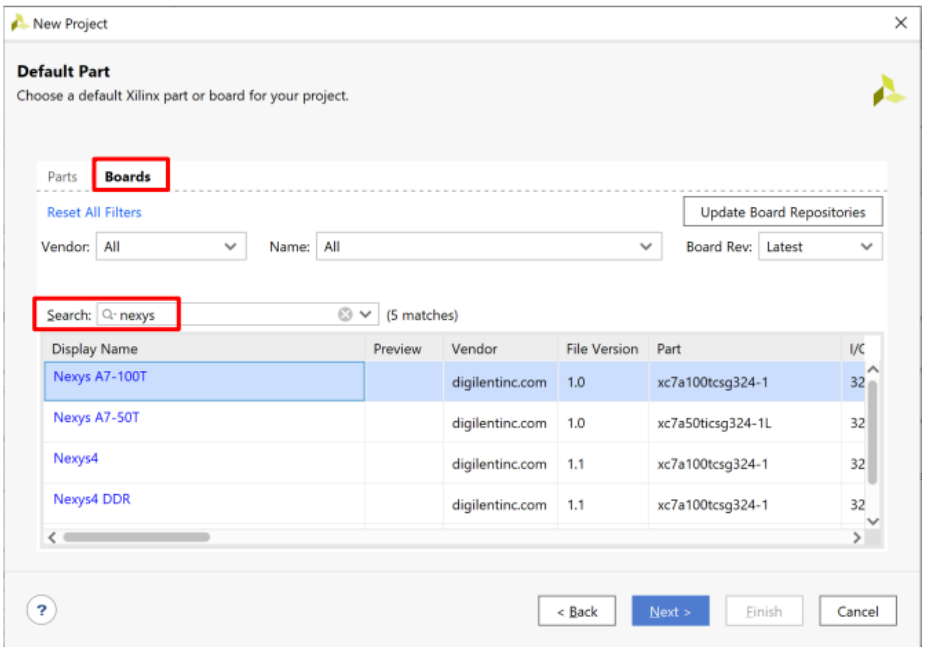


Figure 44: Select the target board.

STEP 5. Project configuration

Set rvfpganexys as Top Module:

In the Sources pane, under Design Sources, right-click on the rvfpganexys module and select Set as Top (see Figure 45). This will set rvfpganexysas as the highest-level module in the hierarchy and the target to be synthesized and implemented into the FPGA. After setting rvfpganexys as the top module, the hierarchy will update.

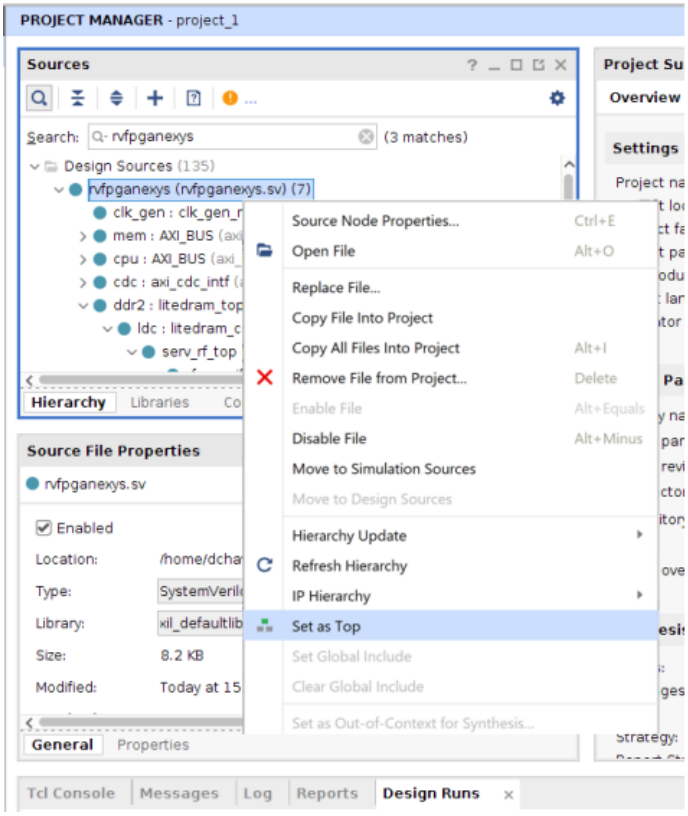


Figure 45: set rcfpganexys.sv as top.

Set file common_defines.vh as Global Include:

Expand the Non-modules file group and click on common_defines.vh. Click on Global Include to tick that box (see Figure46). The hierarchy will update and include that file in Design Sources/Global Include.

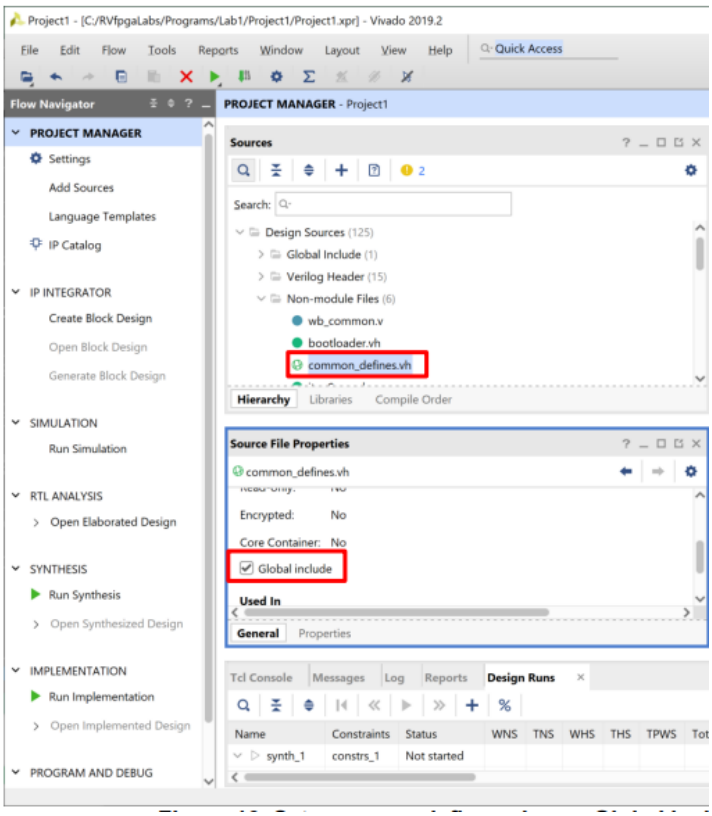


Figure 46: Set the common_defines.vh as global include.

Add boot_main.mem to the project:

Click on Add Sources, and click on Add Files (see Figure 47). Navigate to [RVfpgaPath]/RVfpga/src/SweRVolfSoC/BootROM/sw and select boot_main.mem (as shown in Figure 11). The hierarchy will update and include that file in the Design Sources/Memory File.

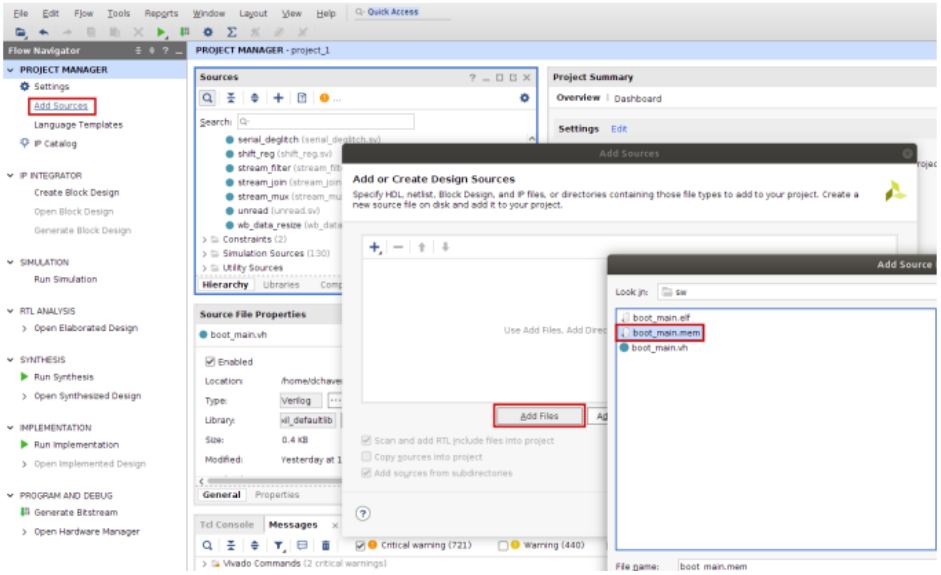


Figure 47: Add boot_main.mem to the project.

Include folders:

Include two folders for the Pulp Platform (see Figure 48). Click on Settings, General, and then on Verilog options. In the new window, add the two following include directories by clicking on and browsing to the directories:

[RVfpgaPath]/RVfpga/src/SweRVolfSoC/Interconnect/AxiInterconnect/pp-platform.org__axi_0.25.0/include
[RVfpgaPath]/RVfpga/src/OtherSources/pulpplatform.org__common_cells_1.20.0/include

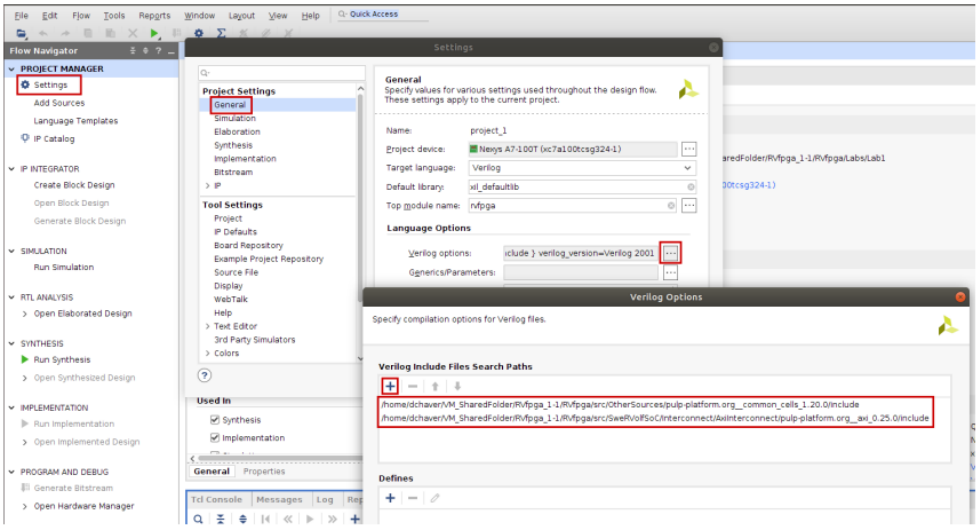


Figure 48 :Include two folders for the Pulp Platform.

STEP 6. Generate Bitstream

Click on Generate Bitstream to synthesize and map RVfpgaNexys into the FPGA, creating a bitstream. Processing time varies depending on computer speed.

It is recommended to use PlatformIO instead of Vivado for uploading RVfpgaNexys to the Nexys A7 board.

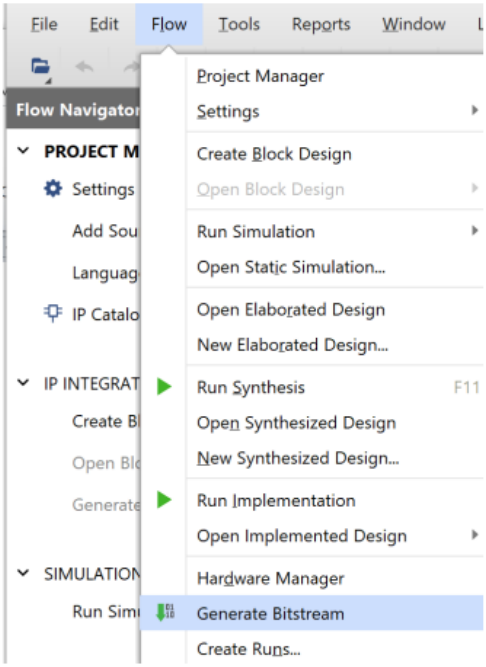


Figure 49:GENERATE BITSTREAM.

3.4.3 RVfpgaSim

In this section, the program used in the "C program for RVfpga" section will be run on RVfpgaSim using Verilator. Verilator is a hardware description language (HDL) Simulator that simulates the Verilog that defines the SoC. This way of running the SoC allows the analysis of the internal signals of the system.

PlatformIO will be used to generate the simulation trace, add the clock, instructions for both ways of the super-scalar processor, and signals to the simulation waveform. GTKWave will be used to see the instruction and register signals change as the program executes.

Generate the simulation binary, Vrvfpgasim:

The directory [RVfpgaPath]/RVfpga/verilatorSIM includes the Makefile and swervolf_0.7.vc script for creating the RVfpgaSim binary. The script has details for Verilator to locate the SoC sources [21].

STEP 1. In a terminal window, the RVfpgaSim simulation binary Rvfpgasim can be generated by executing the following commands:

cd [RVfpgaPath]/RVfpga/verilatorSIM

make clean

make

Once the Vrvfpgasim has been generated, it will be used inside PlatformIO for generating the simulation trace.vcd of the program.

STEP 2. Open VSCode and then PlatformIO and open the directory from the program.

STEP 3. Open file platformio.ini.

Establish the path to the RVfpgaSim simulation binary generated by editing the following line (Figure 50):

board_debug.verilator.binary=[RVfpgaPath]\RVfpga\verilatorSIM\Vrvfpgasim

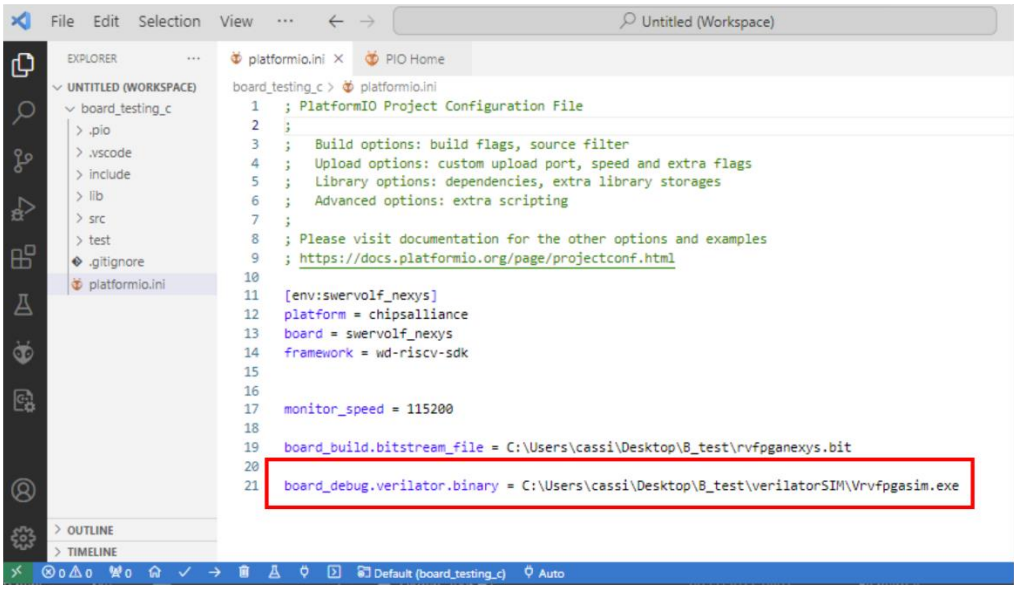


Figure 50: Open file platformio.ini.

STEP 4. Run the simulation by clicking on Generate Trace, as shown in Figure 51 .

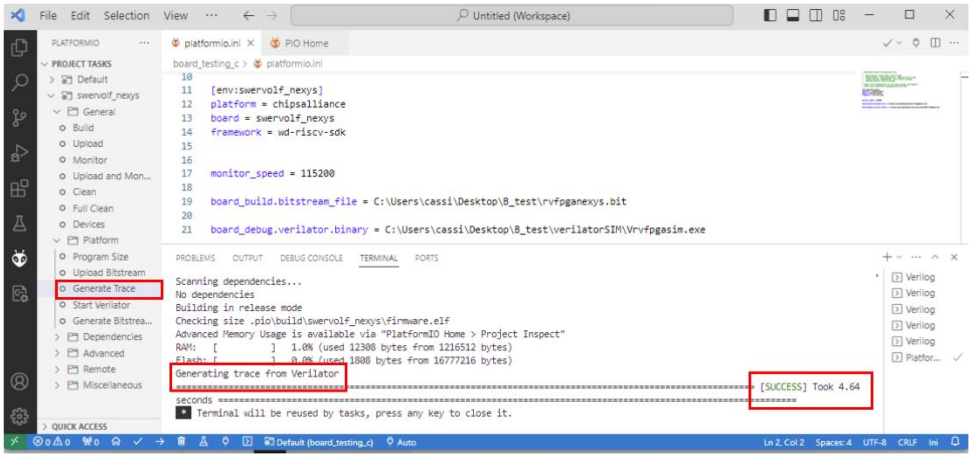


Figure 51: Run the simulation.

A few seconds after the previous step, file trace.vcd should have been generated inside [RVfpgaPath]/RVfpga/examples/AL_Operations/.pio/build/swervolf_nexys, and it can be open with GTKWave.

3.5 Wishbone Bus

The WISHBONE System-on-Chip (SoC) Interconnection Architecture for Portable IP Cores is a versatile design method for semiconductor IP cores. It provides a standardized interface between IP cores and can be applied to soft or hard-core IPs. This methodology does not require the use of particular development tools or target hardware and is fully compatible with most logic synthesis tools[29].

The WISHBONE bus protocol works as follows on a single read\write cycle (figure 52):

- CLOCK EDGE
  - MASTER presents a valid address on [ADR_O()] and [TGA_O()].
  - MASTER negates [WE_O] to indicate a READ cycle.
  - MASTER presents bank select [SEL_O()] to indicate where it expects data.
  - MASTER asserts [CYC_O] and [TGC_O()] to indicate the start of the cycle.
  - MASTER asserts [STB_O] to indicate the start of the phase.
- CLOCK EDGE 1
  - SLAVE decodes inputs, and responding SLAVE asserts [ACK_I].
  - SLAVE presents valid data on [DAT_I()] and [TGD_I()].
  - SLAVE asserts [ACK_I] in response to [STB_O] to indicate valid data.

    - MASTER monitors [ACK_I], and prepares to latch data on [DAT_I()] and [TGD_I()].
- CLOCK EDGE 2
    - MASTER latches data on [DAT_I()] and [TGD_I()].
    - MASTER negates [STB_O] and [CYC_O] to indicate the end of the cycle.
    - SLAVE negates [ACK_I] in response to negated [STB_O].

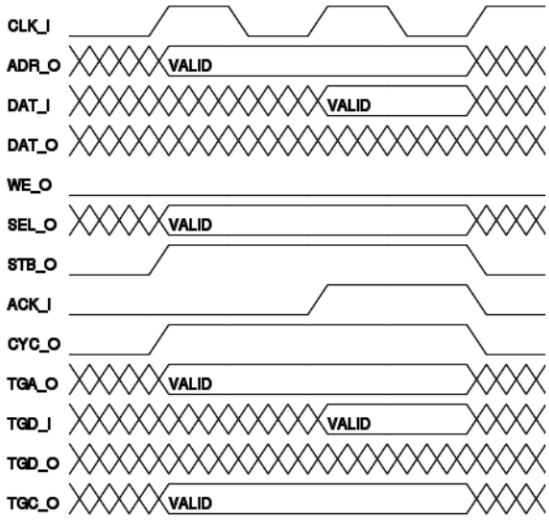


Figure 52: Wishbone signals from [].

3.5.1 Wishbone implementation

This section will perform a step-by-step practical implementation and analysis of the Wishbone bus using a simple Verilog module that performs a sum between two memories.

Step 1: Verilog Code

The Verilog Base module performs addition operations between variables A and B and stores the result in C. The module reads two operands A and B and uses them to perform the addition. A third variable, STATUS, acts as a controller for the output variable C. If STATUS is set to 1, C receives the result of A+B. Otherwise, it defaults to 0xfe.

Figure 53: Wishbone based Verilog module.

The read and write operations are implemented as referred by the Wishbone standard manual [29], as all input and output signals.

Step 2: Integration into the SoC

The base module is instantiated in the system on a chip (SOC), as shown in the figure 54.

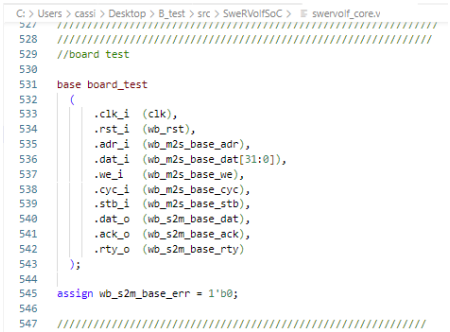


Figure 54: base module integration into the SoC.

Step 3: Connection to SweRV EH1 core

To connect the device controllers with the SweRV EH1 Core, a multiplexer and a bridge are used. The multiplexer selects one peripheral among N, depending on the address generated by the CPU. The bridge is responsible for translating the Wishbone signals used by the device controllers to

the AXI4 signals used by the SweRV Core, and vice versa. The figure below illustrates the multiplexer's function in the system.

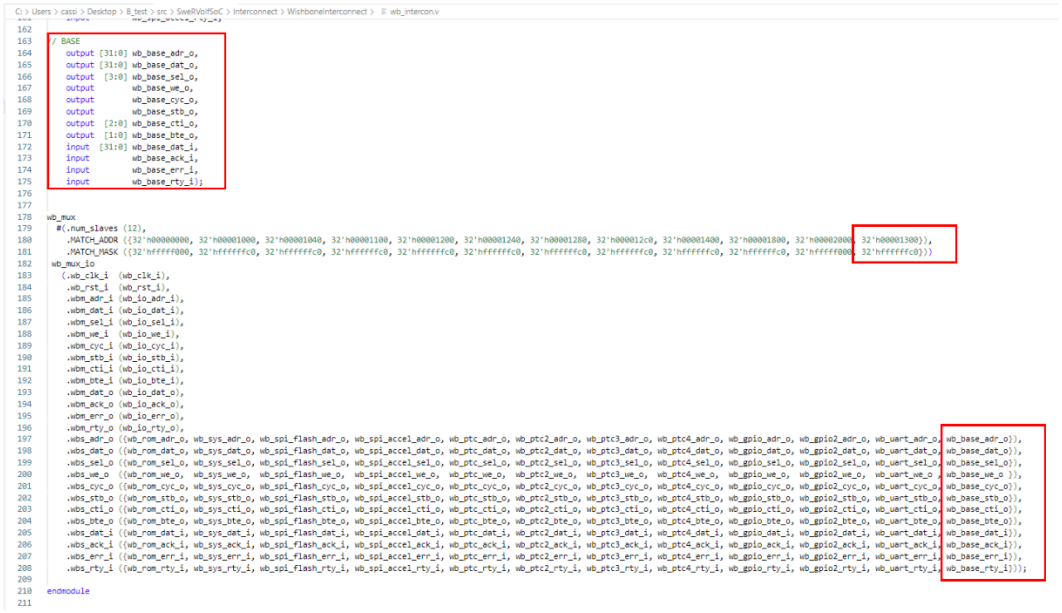


Figure 55: Base module connection to SweRV EH1 core.

Step 4: Generate the bitstream

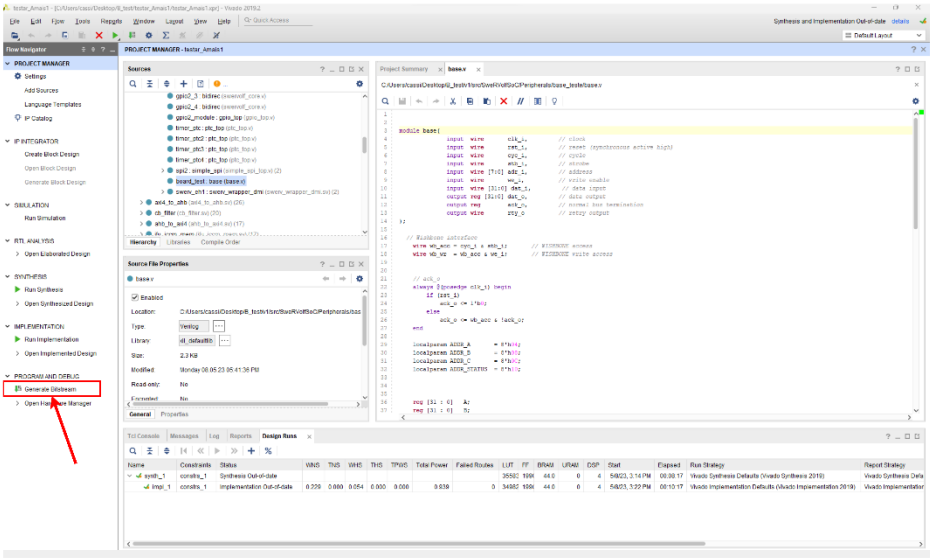


Figure 56: Bitstream generation.

Step 5: Generate the simulation and trace

Before generating RVfpgaSim, it's important to add directories for the base module's source code to the SWERVOLF_0.7.VC script (Figure 57). Then execute the code shown in Figure 58 to generate Vrvfpgasim.

```
C: > Users > cassi > Desktop > B_test > verilatorSIM > ≡ swervolf_0.7.vc
1   --Mdir .
2   --cc
3
4
5   +incdir+../src/SweRVolfSoC/SweRVEh1CoreComplex/include
6   -CFLAGS -I../src/SweRVolfSoC/SweRVEh1CoreComplex/include
7   +incdir+../src/OtherSources/jtag_vpi_0-r5/
8   -CFLAGS -I../src/OtherSources/jtag_vpi_0-r5/
9   +incdir+../src/SweRVolfSoC/Interconnect/AxiInterconnect/pulp-platform.org__axi_0.25.0/include
10  -CFLAGS -I../src/SweRVolfSoC/Interconnect/AxiInterconnect/pulp-platform.org__axi_0.25.0/include
11  +incdir+../src/OtherSources/pulp-platform.org__common_cells_1.20.0/include
12  -CFLAGS -I../src/OtherSources/pulp-platform.org__common_cells_1.20.0/include
13  +incdir+../src/SweRVolfSoC/Peripherals/uart
14  -CFLAGS -I../src/SweRVolfSoC/Peripherals/uart
15  +incdir+../src/SweRVolfSoC/Peripherals/spi
16  -CFLAGS -I../src/SweRVolfSoC/Peripherals/spi
17  +incdir+../src/SweRVolfSoC/Peripherals/ptc
18  -CFLAGS -I../src/SweRVolfSoC/Peripherals/ptc
19  +incdir+../src/SweRVolfSoC/Peripherals/gpio
20  -CFLAGS -I../src/SweRVolfSoC/Peripherals/gpio
21  +incdir+../src/SweRVolfSoC/Peripherals/base_teste
22  -CFLAGS -I../src/SweRVolfSoC/Peripherals/base_teste
23  +incdir+../src/SweRVolfSoC/Interconnect/WishboneInterconnect/wb_common_1.0.3
24  -CFLAGS -I../src/SweRVolfSoC/Interconnect/WishboneInterconnect/wb_common_1.0.3
25  +incdir+../src/SweRVolfSoC/Interconnect/WishboneInterconnect
26  -CFLAGS -I../src/SweRVolfSoC/Interconnect/WishboneInterconnect
27  +incdir+../src/SweRVolfSoC/Interconnect/AxiInterconnect
28  -CFLAGS -I../src/SweRVolfSoC/Interconnect/AxiInterconnect

208 ../src/SweRVolfSoC/Interconnect/AxiInterconnect/axi_intercon.sv
209 ../src/SweRVolfSoC/Interconnect/WishboneInterconnect/wb_intercon.v
210
211 ../src/SweRVolfSoC/Peripherals/gpio/gpio_top.v
212
213 ../src/SweRVolfSoC/Peripherals/base_teste/base.v
214
215 ../src/SweRVolfSoC/Peripherals/ptc/ptc_top.v
216
217 --top-module rvfpgasim
218 --exe
219
220 ../src/OtherSources/jtag_vpi_0-r5/jtag_common.c
221 ../src/OtherSources/jtag_vpi_0-r5/jtagServer.cpp
222 ./tb.cpp
```

Figure 57: SWERVOLF_0.7.vc file edit to include the base module verilog code.

```
/cygdrive/c/Users/cassi/Desktop/B_test/verilatorSIM

cassi@JoaoCassiano ~
$ cd /cygdrive/c/Users/cassi/Desktop/B_test/verilatorSIM

cassi@JoaoCassiano /cygdrive/c/Users/cassi/Desktop/B_test/verilatorSIM
$ make clean
rm Vrvfpgasim*
rm *.o
rm *.d

cassi@JoaoCassiano /cygdrive/c/Users/cassi/Desktop/B_test/verilatorSIM
$ make
```

Figure 58: Commands to generate the Rvfpgasim.

The INI Platformio file is updated with the generated file directory to then trace SoC signals (figure 59).

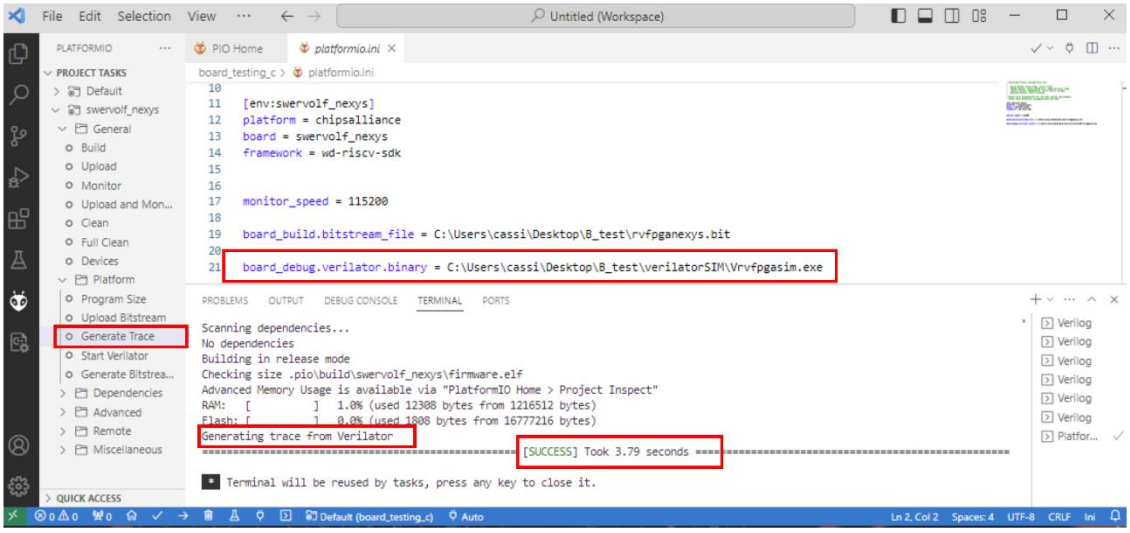


Figure 59: Generate trace.

Step 6: GTKwave

The VCD generated file was opened with GTKwave, allowing analysis of Wishbone bus signals (figure 60-61). The signals are similar to the standard manual ones in Figure 52.

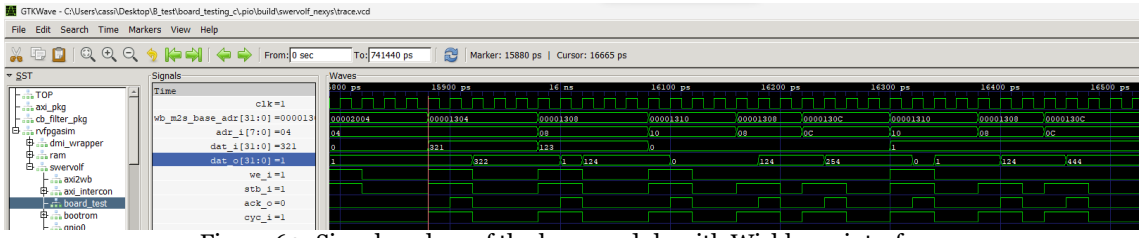


Figure 60: Signal analyze of the base module with Wishbone interface.

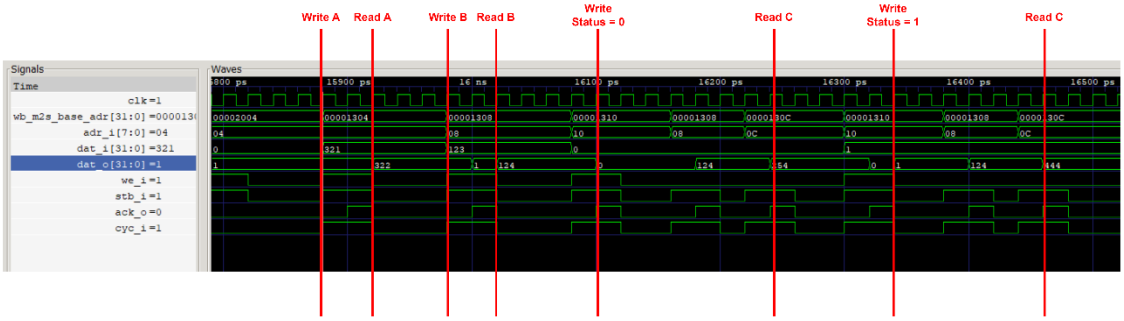


Figure 61: Wishbone Read and Write signals.

Capítulo 4
Data encryption and authentication

The market demands both product performance and security. Security mechanisms for the Internet of Things, artificial intelligence, and wireless sensor network platforms are proposed for the application layer. IoT systems usually run on small cores of embedded systems, known for their low computing power and limited resources. Implementing complex security policies in such constrained environments takes much work. Hardware-based security mechanisms are a good solution due to their high efficiency of execution. They provide a balance between security and system performance[7].

4.1 Cryptographic Instruction Set

Performing cryptographic operations as securely and efficiently as possible is a requirement for computing platforms. One of the solutions is through dedicated instructions set extensions (ISE), more specifically, cryptographic instructions[7]. Those have two advantages over the software-based implementations:

- Maximize the performance of cryptographic operations.
- Hide implementation details, reducing the attack surface.

4.2 Hardware security

The semiconductor industry uses various verification techniques to ensure systems-on-chip (SoC) security. However, attacks are becoming more sophisticated, especially in embedded devices such as IoT ones. Side channel or fault injection attacks are a significant hazard that could be minimized by prioritizing security over pure performance in architecture designs.

The security architecture must balance the application's high performance and low power consumption. The open-source and capability of the new RISC-V instruction set extensions provide significant opportunities for innovation to grow in the development of new chip security solutions.

4.3 Cryptographic Algorithms

Cryptographic standards and guidelines exist for many cryptographic functions, including block cipher techniques, digital signatures, hash functions, and key management. Cryptographic algorithms are used in security mechanisms but are heavy to compute.

Integrating cryptographic algorithms as functions of an ISA extension and implementing them in hardware makes it possible to meet the performance requirements or the lack of it in a system.

Block or stream cipher algorithms are used for online data encryption, where computer power is not concerned.

Because of its speed, a stream cipher is helpful for tasks like streaming video or mobile communications. It also allows for jumping straight to the middle of a data block. Instead of encrypting a message with a key, it uses the key to generate a long pseudo-random key stream for as long as required and then uses XOR operations to perform the encryption. The encryption occurs one bit at a time. If a bit is changed in the ciphertext, it also flips a bit on the plaintext. Any third party can manipulate the message without knowing its content.

A block cipher encrypts the block of data as one. Any bit change in the ciphertext will completely wreck its output when tried to decrypt.

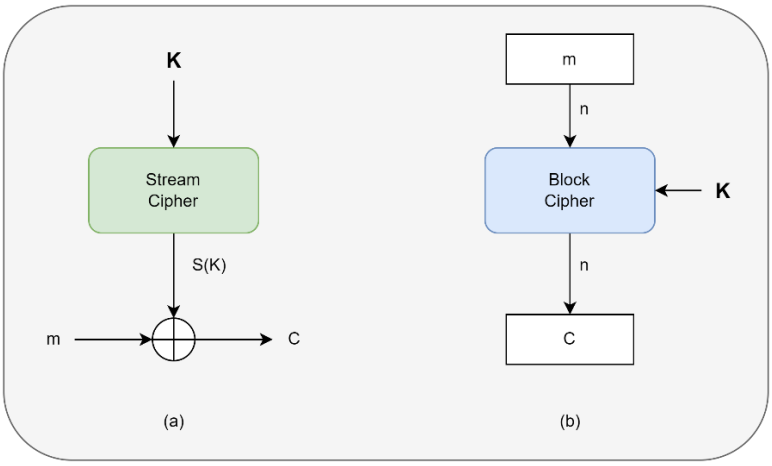


Figure 62: a) stream cipher. b) block cipher.

Both bank transactions and logins are sent using the same format. A bank does not need to change the data structure when executing an operation. The same bit will often correspond to the transaction account numbers, values, and client data. Message authentication codes (MACs) become crucial to ensure the authenticity of data exchange.

MACs are attached to most of the messages sent on the internet. They can be built into the cipher or appended to every message, even if it is not encrypted. MACs can guarantee that the message has stood intact.

A checksum operation is a simple example of a MAC. In this case, a message can be hashed by a checksum, and then that hash append to the end of the message itself. The hash is a short, fixed-length string that summarizes the message. It is impossible to get the original data from the hash alone. Its only purpose is to act as a fingerprint of the message, ensuring its integrity.

The problem with using a simple checksum as a MAC is that a third party could intercept the message, stop the transmission, and change it to whatever it wants. It could just recompute a new hash, attach it, and the device verifying this falsified message on the other end would hash it and validate it as if nothing ever happened. This verification is acceptable and even used at the internal data management level. Beyond that, a more secure and able-to-avoid hash collision implementation is necessary.

If both ends, deviceA and deviceB, have shared a key privately, only they know this information. DeviceA can produce a hash of the shared key and append that to a message or even a hash from a combination of a message and the key.

If both devices agree with the steps to generate the hash by having a shared key and a well-established protocol for using it, and if no one else knows the key, a third party cannot interfere with the message. If a single bit gets altered, neither of the devices can recompute the correct hash; therefore, the received data is treated as faulty.

4.4 ChaCha20 encryption

The Advanced Encryption Standard (AES) specifies a cryptographic algorithm approved by the National Institute of Standards and Technology (NIST) that can be used to protect electronic data. Most modern machines have a library that supports AES and is also in browsers and web servers[7].

Having just one algorithm is not a compelling idea. An attack on AES is possible, although, and up to now, it is only theoretical. A mathematical weakness or another weakness could be found in AES that is out of public knowledge, and an unstoppable attack could be released. If that happens, a backup algorithm is needed, and currently, one of the alternatives to AES is ChaCha[30].

In 2008, Daniel J. Bernstein proposed a new family of ciphers flow called ChaCha, based on the Salsa family of ciphers. The construction of this cipher was standardized in RFC 8439, where the size of the nonce and counter were modified to 32 and 96 bits, respectively, in contrast to Bernstein's original article[31].

According to the author[31], Salsa20 can be an alternative to AES in applications where the seed is a bigger priority than the integrity of the cipher's security. ChaCha20 has better characteristics than Salsa. It provides higher integrity while consistently faster than AES on machines without hardware accelerators. These properties allowed ChaCha20 to earn recognition in the cryptographic community

The ChaCha is a stream cipher that can be described as a 4 by 4 32-bit block (see Figure 63) organized in little-endian form. The first 4 blocks are constants, followed by 8 blocks of key (256-bit in total), a single 32-bit block counter, and a 96-bit nonce split into 3 blocks. If needed, it is possible to implement a 64-bit counter, reducing the nonce to 64-bit too. The nonce ensures that a different key stream can be generated with the same key[30].

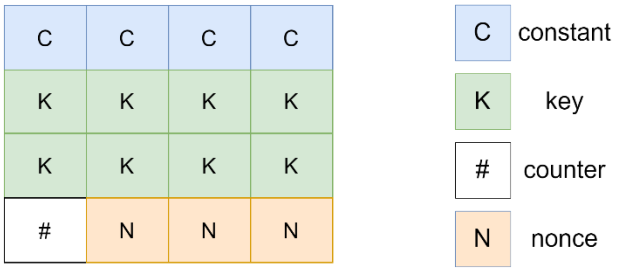


Figure 63: ChaCha cypher block.

The ChaCha algorithm is based on an ARX cipher, meaning it can be implemented with only three operations: add, rotation, and XOR. By only using these three basic operations, the time to generate a new key stream will always be the same and smaller compared to the complex operations used in AES . In addition, doing this over the AES is that the time cipher is always the same, preventing time-based attacks. The addition is made in mod 32, adding two integers together without any carry bits. Next, the rotation (see Figure 64) happens on an integer level, shifting every bit and wrapping it back around. Finally, the XOR operation is performed between two integers [30].

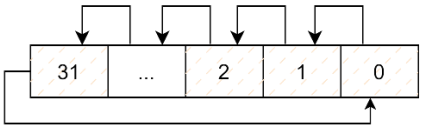


Figure 64: rotation operation on an integer level.

A quarter-round is a set of operations composed of four 32-bit unsigned integers, denoted a, b, c, and d. This set is performed on a column and a diagonal level to get a greater diffusion of the data, making the cypher harder to break. As shown in Fig. from b is added to a, a is XOR to d, and d is rotated 16 bits to the left; d is added to c, c is XOR to b, and b is rotated 12 bits to the left; b is

added to a, a is XOR to d and d rotated 8 bits to the left. Finally, d is added to c, c is XOR to b, and then b is rotated 7 bits to the left [30].

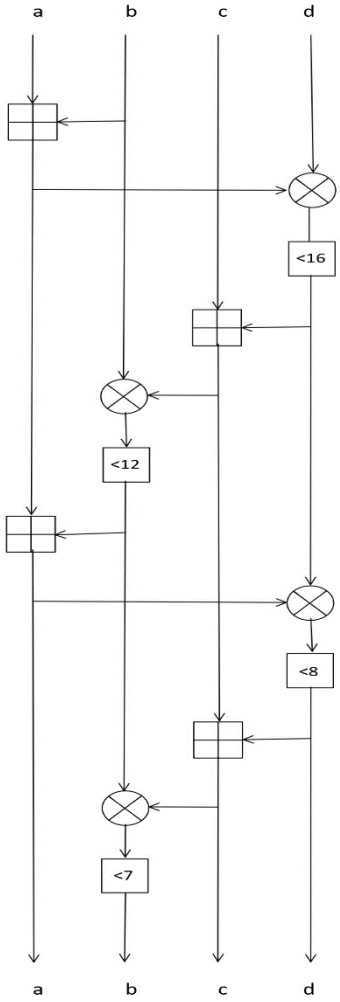


Figure 65: ChaCha quarter-round set of operations.

ChaCha20 designation comes from the number of rounds, alternating between "column rounds" and "diagonal rounds". Each round consists of four quarter-rounds, as shown in Figure 66.

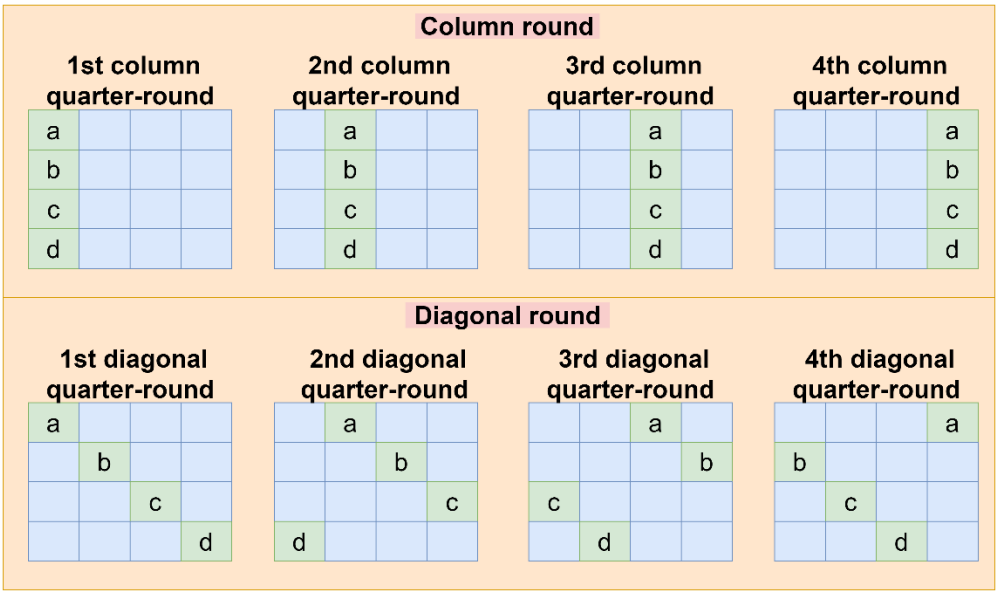


Figure 66: ChaCha20 column and diagonal round.

Once all rounds are completed, the original state block is added to the generated ChaCha block creating the keystream. Then, the desired message can be encrypted by XORing his plaintext with the now-generated keystream. The encrypted message will always be the same size as the keystream, assuming the plaintext does not exceed 512-bit. If that happens, the value on the counter is increased, and another key stream is generated to encrypt the remaining text [30].

Chacha20 pseudocode

Chacha20 pseudocode from [32]:

```
chacha20_encrypt (key, counter, nonce, plaintext) :
    for j = 0 upto floor (len(plaintext) / 64) -1
        key_stream = chacha20_block (key, counter+j, nonce)
        block = plaintext [ (j*64) .. (j*64+63) ]
        encrypted_message += block ^ key_stream
        end
    if ((len (plaintext) % 64) != 0)
        j = floor (len (plaintext) / 64)
        key_stream = chacha20__block (key, counter+j, nonce)
        block = plaintext [ (j*64) .. len (plaintext) - 1]
        encrypted_message += (block^key stream) [0..len(plaintext) % 64]
        end
    return encrypted_message
    end
```

Figure 67: Chacha20 pseudocode from [32].

Chacha20 implementation

Joachim Strömbergson has committed on his Github page a Verilog implementation of the ChaCha stream cipher. The link to download it is the following: [GitHub - secworks/chacha: Verilog 2001 implementation of the ChaCha stream cipher.]

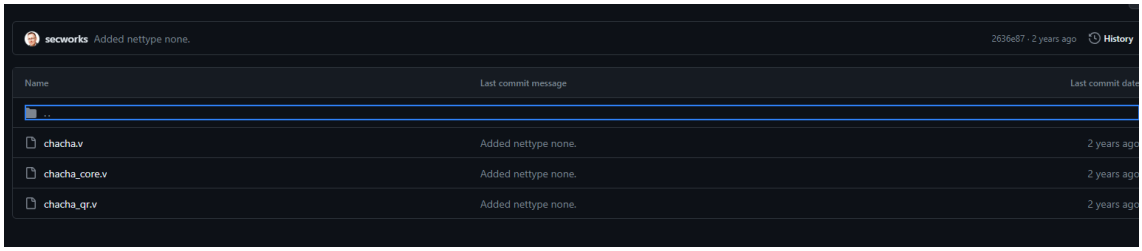


Figure 68: Joachim Strömbergson chacha20 GitHub folder [33].

It´s a complete and well-commented implementation. The code is separated into three modules:

- Chacha.v, which represents the top module of the RTL code. It uses an address-based data input and output, useful for integration with other modules or processors.
- Chacha_core.v, has the ChaCha algorithm functions described in HDL.
- Chcha_qr.v, it´s instantiated in the Chacha_core.v and describes the round function of the ChaCha algorithm.

Strömbergson also provided test bench to evaluate the performance of the Chacha.v module. The following images represent the execution of a base test from [32] through the test bench, where the all the initial parameters are set to zero. The output data corresponds to the expected and described on the Nir and Langley informational document [32].

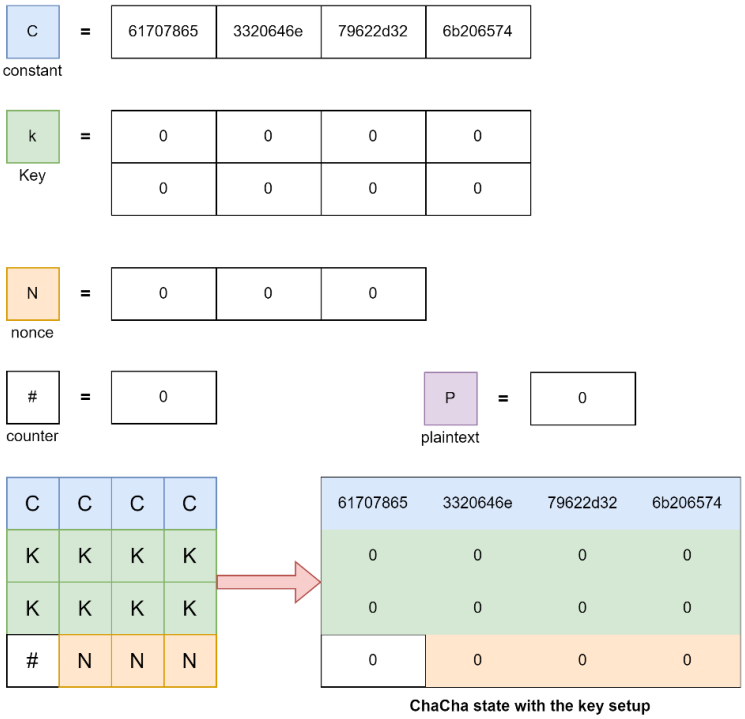


Figure 69. Nir and Langley ChaCha20 test vector [32]

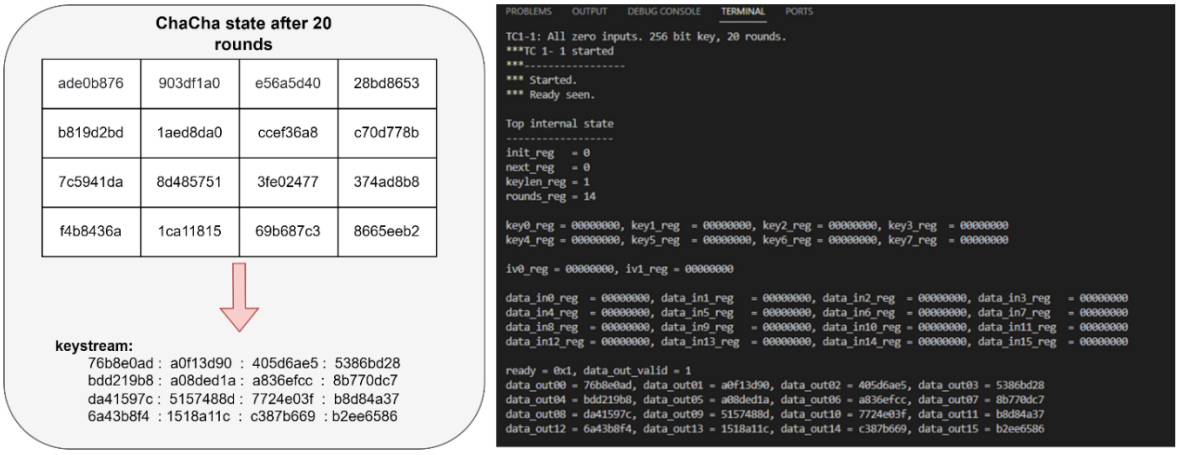


Figure 70: Nir and Langley ChaCha20 test vector [32] on the left. Result of the ChaCha.v module test.

4.5 Poly1305 authentication

In 2005 D. J. Bernstein published an article entitled "The Poly1305-AES message-authentication code", where he presented his one-time authenticator algorithm called Poly1305. It was initially designed to work alongside the AES, although the author brings, for several times, the possibility of combining the message authentication capabilities of the Poly1305 with other encryption functions. Poly1305 produces a 16-byte message authentication tag with a 32-byte key and a variable-length message [34].

Poly1305 was designed to offer consistent high speed, regardless of the CPU where it was running or the size of the message, mainly because of its easy-to-implement architecture and well-thought

choice of a straightforward polynomial evaluation modulo 2^130 – 5. The ability to be implemented with a different cipher shows the authors' concerns about an unlikely scenario of an AES being broken down. Bernstein ensures the user that even if something wrong happens to AES, a non-AES Poly1305 implementation could sustain its integrity and reliability, providing the same security guarantee relative to the security of the non-AES implementation[30].

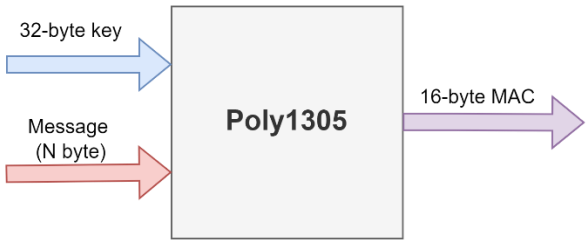


Figure 71: Poly1305 diagram.

Keys

Assuming that the 32-byte key is unpredictable, kept in secret, and shared between the sender and the receiver safely, the Poly1305 splits its key into two parts: a 16-byte key named s and a second 16-byte string r [0], r[1],...,r[15].

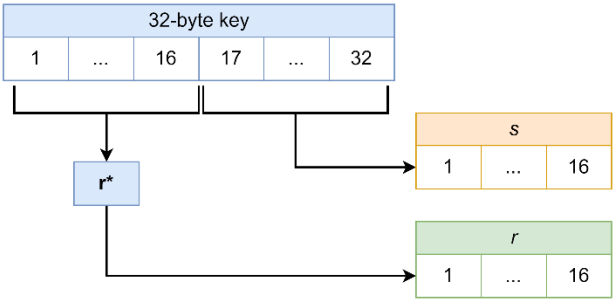


Figure 72: Number r and s generation from a 32-byte key.

The second key, called "r key," is a 128-bit unsigned integer in a little-endian form that receive some modifications in order to be usable for the Poly1305 implementation. The top four bits of r[3], r[7], r[11], and r[15] are required to be 0 (zero) as the bottom two bits from r[4], r[8], and r[12]. In other words, it is the same as making follow AND operation:

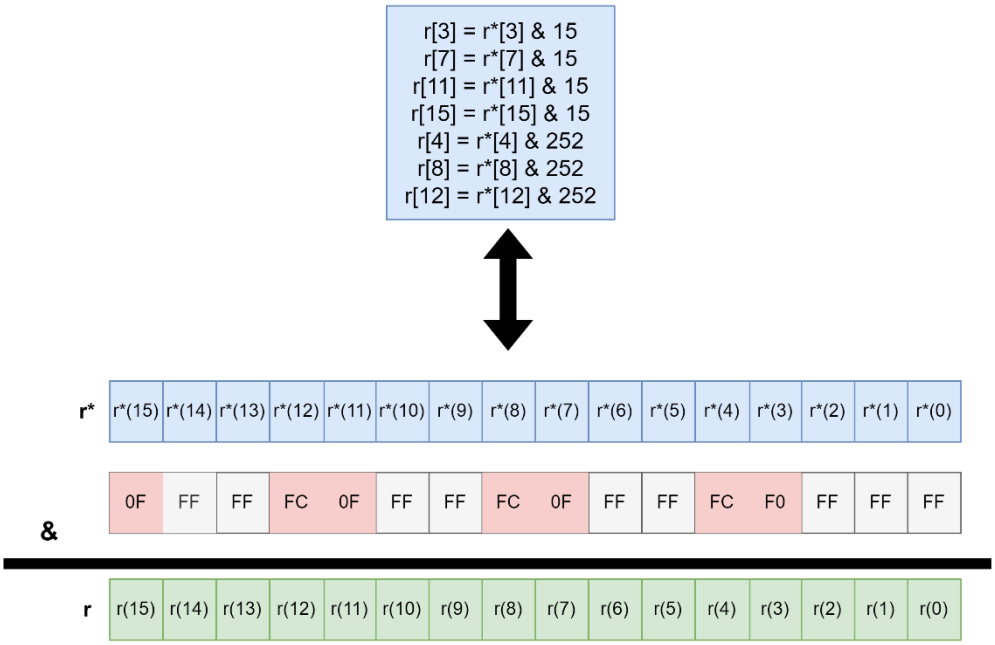


Figure 73: "r" key generation.

Messages

As mentioned above, the Poly1035 bock accepts different length messages and spreads them into 16-byte chunks of data (Figure 74). Each 16-byte of data is padded to a 17th byte by appending a 1. If the final chunk length is between 1 and 15 bytes, 1 is appended to the chunk and then zero-pad the chunk to 17 bytes (Figure 74). That way, every message, regardless of its length, is treated as a 17-byte block of data, that is converted in a little-endian integer.

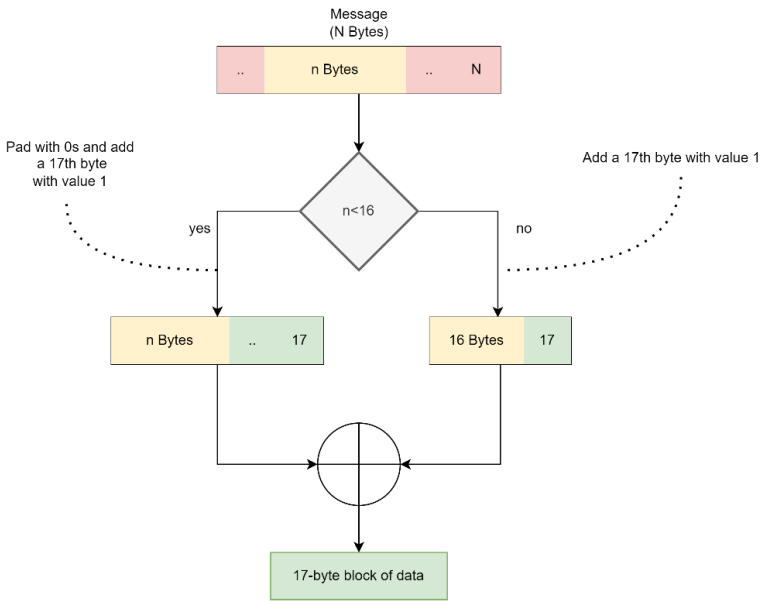


Figure 74: Poly1305 spliting messages in 16-byte chunks.

The author made the little-endian instead of big-endian choice to improve overall performance. Besides making no difference on most CPUs, the little-endian format saved some running time on the most popular CPUs of the era, like the Pentium and the Athlon [30].

Authenticators

The Poly1305 algorithm is only capable of authenticating 16-byte messages at a time, so it uses an accumulator to iterate each part of the message. After setting a constant prime "P" with the value 2^130-5, or 0x3fffffffffffffffffffffffffffffffb, the 16-byte of the message is added to the accumulator and then is multiplied by the "r" key, and the result is modeled with P. To summarize:

$$ACC = \big((ACC + block) + r\big)\%P$$

The iteration continues until there is no more data to authenticate. Finally, the key "s" is added to the accumulator, and the generated 128 bits are serialized in the little-endian form to generate the MAC [30], [34].

Poly1305 pseudocode

Poly1305 pseudocode from [32]:

```
clamp (r): r &= 0x0ffffffc0ffffffc0ffffffc0fffffff

poly1305_mac(msg,  key):
        r = le_bytes_to_num (key [0..15])
        clmap (r) s = le_bytes_to_num (key[16..31])
        a  = 0 /* a is a accumulator */
        P = (1<<130)-5

        for i = 1 up to ceil (msg lenght in bytes /16)
            n = le_bytes_to_num (msg [((i-1)+16)..(i*16)] | [0x01])
            a += n
            a = (r*a)% P
            end

        a += s
        return num_to_16_le_bytes (a)

        end
```

Figure 75: Poly1305 pseudocode.

Poly1305 implementation

Besides the ChaCha implementation, Strömbergson also commit on is Github page a Verilog implementation of the Poly1305 authentication algorithm [35]. The link to download it is the follow: *GitHub – secworks/poly1305: Hardware implementation of the poly1305 message authentication function*.

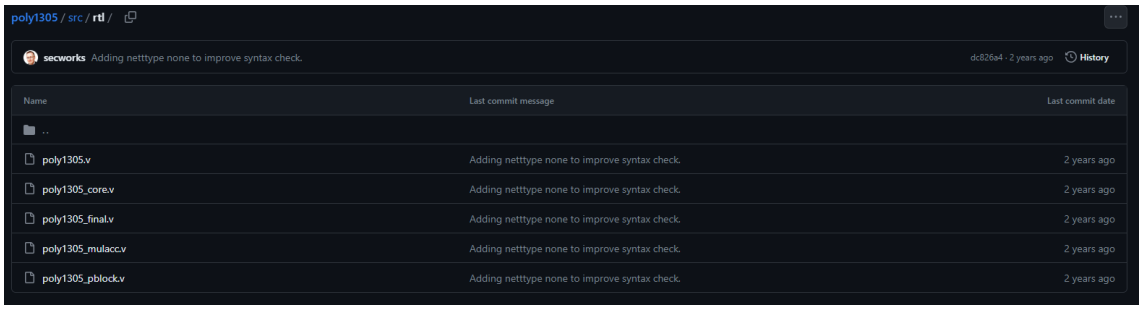


Figure 76: Joachim Strömbergson Poly1305 GitHub folder.

To begin processing a message, first set the key and activate the 'init' port for one cycle. The message is divided into 16-byte blocks and each block is processed by activating the 'next' port and setting the block length. All blocks except the final one, are expected to have a length of 16 bytes. The message processing is completed by activating the 'finish' port for one cycle. Strömbergson [35] also provided test bench to evaluate the performance of the Poly1305 module.

The following images show a test vector for the Poly1305 based on the one presented in [32]. In Figure 77 the S and R keys are created, in Figure 78 the message to be authenticated is split into blocks of 16-bytes a, in the Figures 79, 80, and 81 the authentication of each chunk of data is authenticated and added to the accumulator to produce the final authentication TAG, as shown in Figure 82.

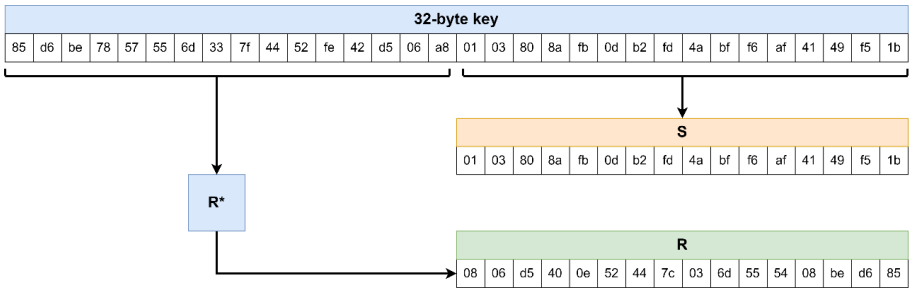


Figure 77: R and S Keys generation from a 32-byte key.

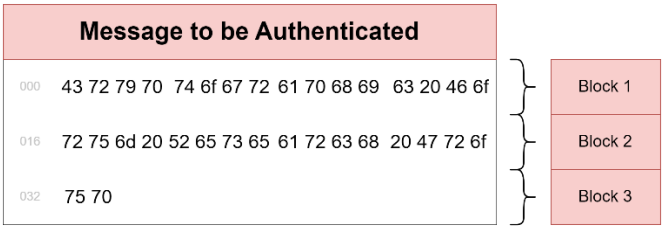


Figure 78: Message to be authenticated being split in blocks of 16-byte.

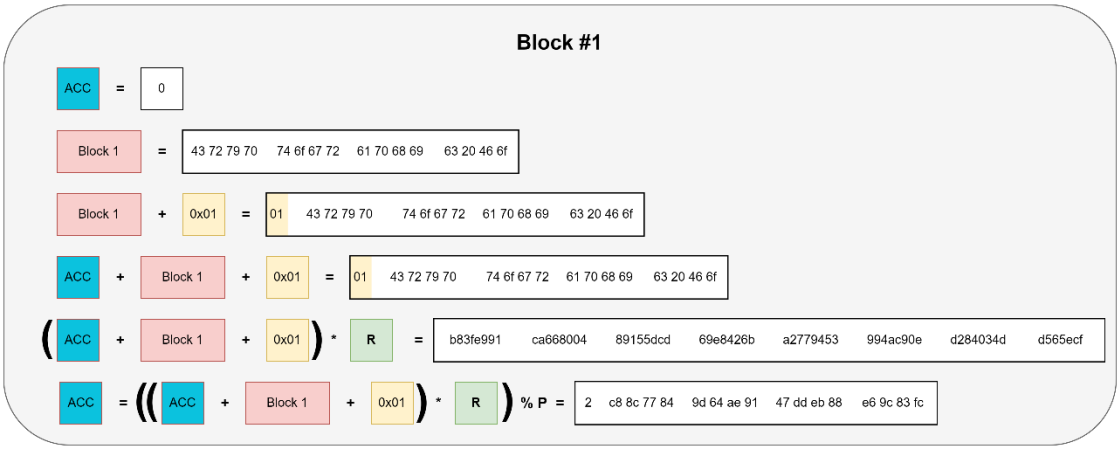


Figure 79: Poly1305 first 16-byte message calculation.

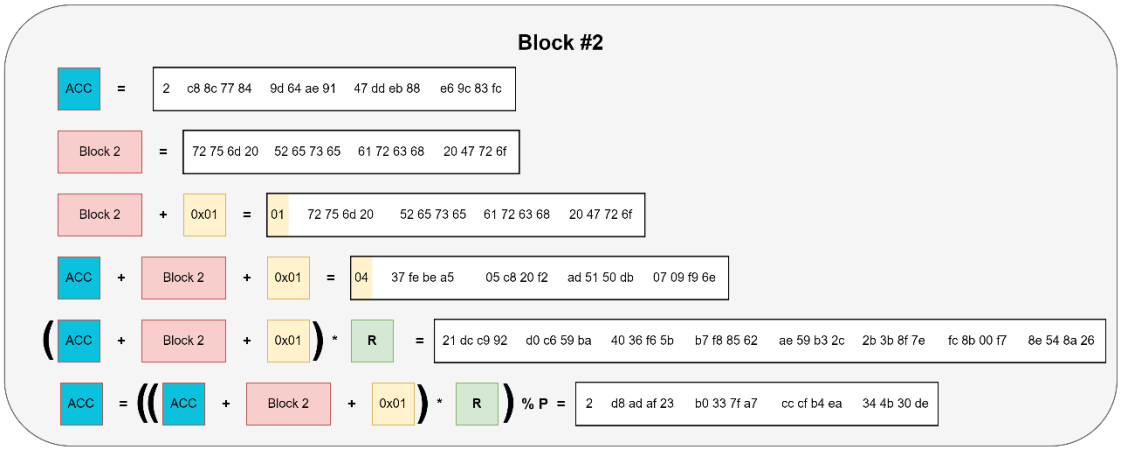


Figure 80: Poly1305 second 16-byte message calculation.

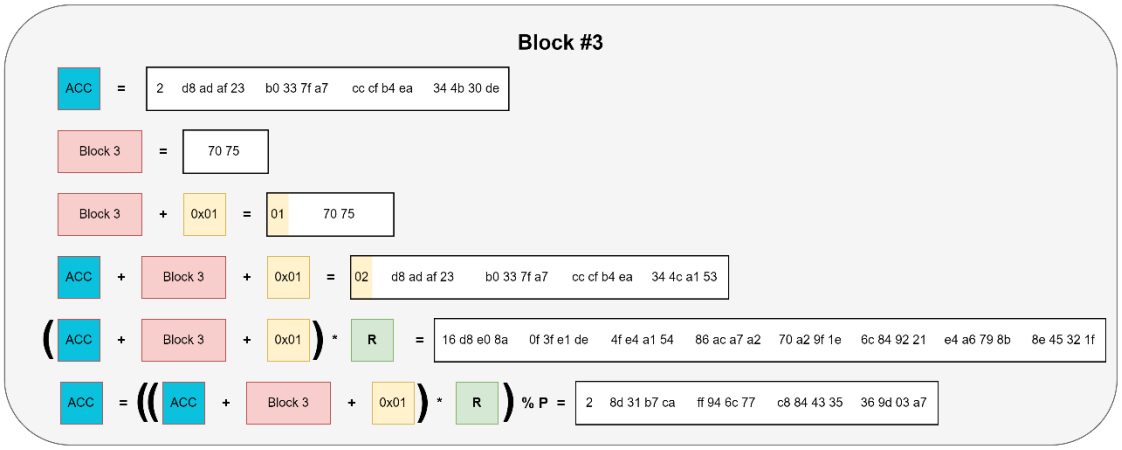


Figure 81: Poly1305 last bytes of message calculation.

Figure 82: Poly1305 TAG generation.

4.6 AEAD ChaCha20-Poly1305 Construction

ChaCha20-Poly1305 is an Authenticated Encryption with Additional Data (AEAD) algorithm that combines the data encryption capabilities from ChaCha20 with the message authentication tag generated from Poly1305. Only taking a 256-bit key, a 96-bit nonce, and an arbitrary length of additional authenticated data (AAD) makes it possible to encrypt and authenticate a plaintext [30], [32], [36].

The Poly1305 pseudo-randomly key is obtained using the ChaCha20 key, with the counter block set to zero, executing the ChaCha20 block function and taking the first 256 bits of the 512-bit state. The ChaCha20 function encrypts the plaintext, using the same key and nonce but increasing the count after each iteration. Finally, the message is constructed as a concatenation of a cipher text, with the same length as the plaintext and a 128-bit authentication tag generated by the Poly1305 function [32].

If, for example, the desired output payload contains a header at the beginning with information like the packet's length, as shown and described in [37], the header should be encrypted by the ChaCha20 in the first instance, with the counter block set to zero. In this case, the Poly1305 key would be generated in the second instance (counter block =1), and the encryption of the plain text should follow it as described above.

In 2022 Ronaldo et al. analyzed the performance of integrating the AEAD into a RISC-V environment, where a 15x performance increase was reported on a ChaCha20-Poly1305 implemented in the system compared to software alternative [36].

Poly1305 key generator from ChaCha20 pseudocode:

Poly1305 pseudocode from a ChaCha20 cypher block:

```
poly1305_key_gen (key,nonce):
    counter = 0
    block = chacha20_block (key, counter, nonce)
    return block [0..31]
    end
```

Figure 83: Poly1305 pseudocode from a ChaCha20 cypher block[32]

Poly1305 key generator from ChaCha20 test vector:

The figures below illustrate how to generate a one-time key for Poly1305 from a ChaCha20 cipher block, using a test vector from [32] as reference. Figure 84 shows the initial conditions for the ChaCha block.

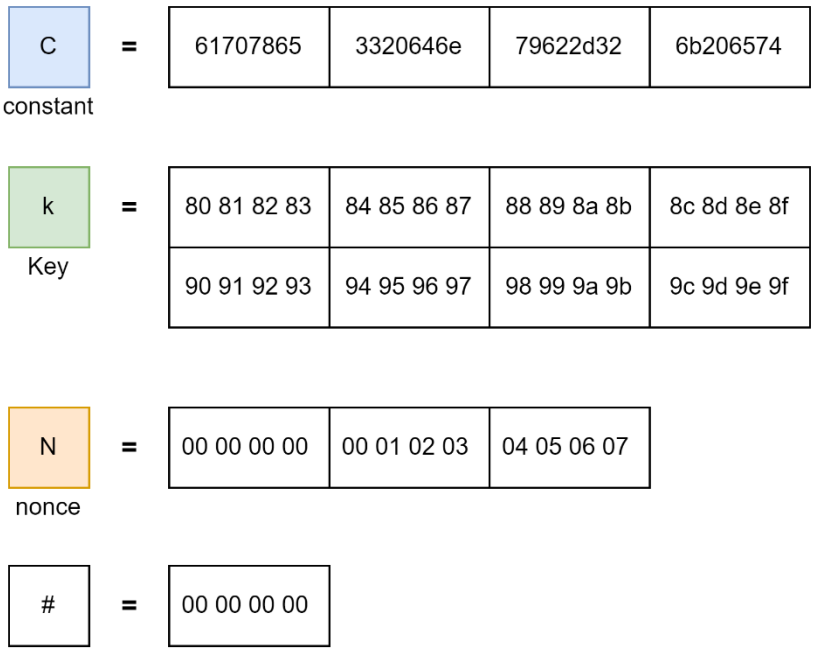


Figure 84: ChaCha20 initial conditions for the test vector from [32]

Figure 85 shows the ChaCha20 block construction and its final state after 20 rounds. Figure 86 presents the one-time Poly1305 key generated from the ChaCha20 Cipher block.

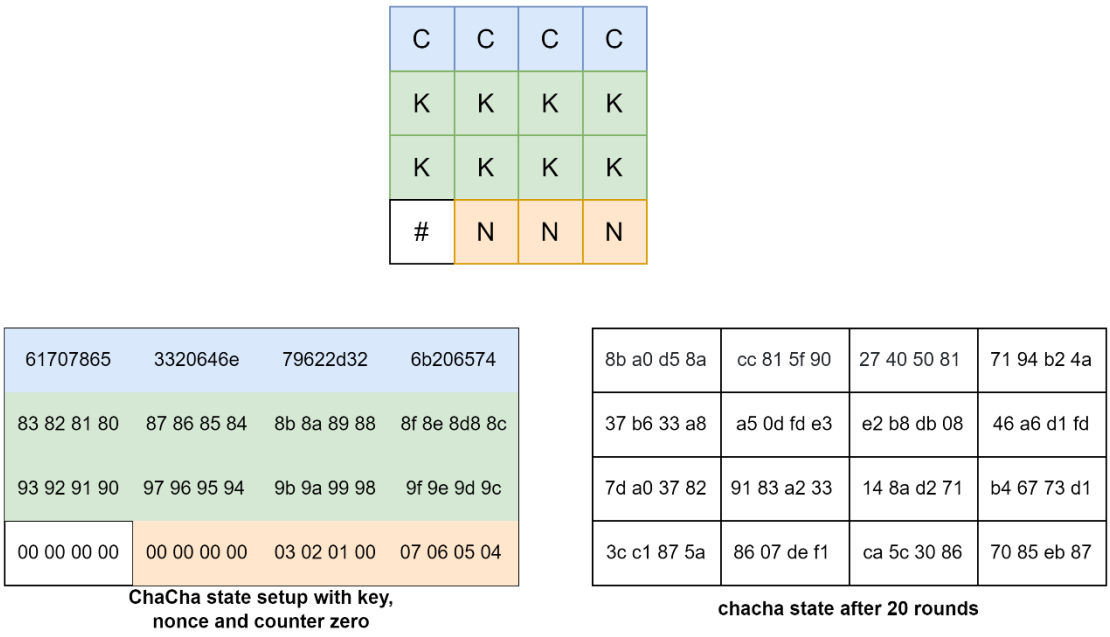


Figure 85: ChaCha20 block construction and it's final state after 20 rounds.

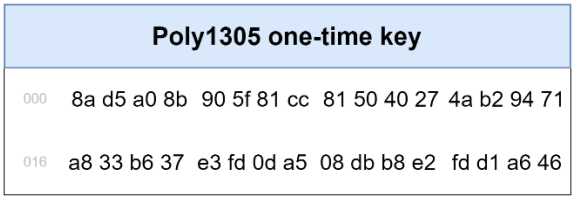


Figure 86: Poly1305 one-time Key, generated from the ChaCha20 Cipher block.

4.7 RISC-V security extensions

Marshall et al. [38] investigated AES's latest industrial and academic ISEs. They evaluated five different ISEs, concluding that 32 and 64-bit architecture ISEs can achieve 4x and 10x the performance compared to a software T-table-based implementation of AES-128 block, respectively[7].

Secure and efficient implementation of AES is an essential requirement on most computing platforms. The RISC-V is working on extending the instruction set to support an efficient AES execution. Work is also done to understand how to use a standard bit manipulation of the RISC-V ISA to implement AES-GCM [39]. This work is part of the ongoing RISC-V cryptography extension standardization process[7].

The RISC-V Cryptographic Extensions Task Group is developing proposals for cryptographic extensions, including bit manipulation, scalar AES, SHA, SM3 and SM4 acceleration, and TRNG entropy source interface [39].

Capítulo 5
Experimental setup

In this section, the RVfpga's capabilities will be extended with three modules that enable the RISC-V to communicate with a temperature sensor through an I2C bus and encrypt and authenticate data.

The first module to be presented, the I2C, has been presented and developed through chapter three and will now be used once again as an example setup to integrate new modules to the RVfpga architecture.

The encryption and the authentication module are based on the ChaCha20 and Poly1305 implementations shown in the previous chapter. The Joachim Strömbergson Verilog implementations will be adapted to the interface of the SoC in order to build two functional blocks compatible with the RVfpga architecture.

The final section of this chapter is the use case combination of the three new modules. The RVfpga will read temperature values from the I2C sensor integrated on the Nexys A7 FPGA, and the collected data will be encrypted and authenticated before being sent to a computer through UART.

Figure provides an overview of the main units and their roles:

- The CPU acts as the controller for any I/O transactions.
- The Device Controller waits for read/write requests from the controller before performing any necessary actions. It consists of a series of registers accessible from the CPU, and the values of these registers instruct the peripheral on what action to take.
- The interconnect, such as Wishbone, establishes a clear path between the controller and the peripherals.

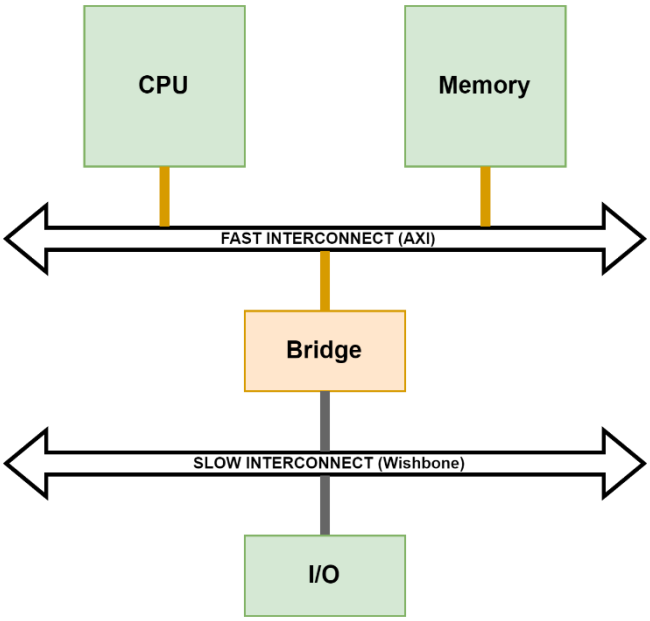


Figure 87: Generic Computing System.

The RVfpga's I/O system includes the seven peripherals:

- LEDs and Switches
- 7-segment displays, connected to the System Controller module
- Flash Memory, connected to the SPI1 module
- Accelerometer, connected to the SPI2 module
- Timer
- UART
- Boot ROM

A multiplexer selects one peripheral among the seven possibilities and connects it with the CPU.

The original SweRVolf includes some peripherals: the Boot ROM, System Controller, SPI Flash Memory and UART. The SweRVolfX SoC extends the original SoC with new peripherals: an SPI Accelerometer, a Timer, a GPIO module, and a 7-segment display controller. Each peripheral receives values from the processor and sends values back to the processor. Memory addresses are reserved for I/O values and are called registers.

5.1 I2C module implementation into a RISC-V architecture

The I2C module implementation can be divided into three steps:

Step 1. Connection and integration of the I2C with the SoC:

The project's constraints file specifies the connections between input/output SoC signals and board devices, with each device linked to a specific FPGA pin. Since the I2C module needs to use connect the SDA and SCL lines to an external pin, the following modification of the contraints file was done:

```
##Accelerometer
set_property -dict { PACKAGE_PIN E15   IOSTANDARD LVCMOS33 } [get_ports { i_accel_miso }]; #IO_L11P_T1_SRCC_15 Sch=acl_miso
set_property -dict { PACKAGE_PIN F14   IOSTANDARD LVCMOS33 } [get_ports { o_accel_mosi }]; #IO_L5N_T0_AD9N_15 Sch=acl_mosi
set_property -dict { PACKAGE_PIN F15   IOSTANDARD LVCMOS33 } [get_ports { accel_sclk }]; #IO_L14P_T2_SRCC_15 Sch=acl_sclk
set_property -dict { PACKAGE_PIN D15   IOSTANDARD LVCMOS33 } [get_ports { o_accel_cs_n }];

##Buttons
set_property -dict { PACKAGE_PIN N17   IOSTANDARD LVCMOS33 } [get_ports { i_pb[0] }]; #IO_L9P_T1_DQS_14 Sch=btnc
set_property -dict { PACKAGE_PIN M18   IOSTANDARD LVCMOS33 } [get_ports { i_pb[1] }]; #IO_L4N_T0_D05_14 Sch=btnu
set_property -dict { PACKAGE_PIN P17   IOSTANDARD LVCMOS33 } [get_ports { i_pb[2] }]; #IO_L12P_T1_MRCC_14 Sch=btnl
set_property -dict { PACKAGE_PIN M17   IOSTANDARD LVCMOS33 } [get_ports { i_pb[3] }]; #IO_L10N_T1_D15_14 Sch=btnr
set_property -dict { PACKAGE_PIN P18   IOSTANDARD LVCMOS33 } [get_ports { i_pb[4] }]; #IO_L9N_T1_DQS_D13_14 Sch=btnd

##I2C
set_property -dict { PACKAGE_PIN C14   IOSTANDARD LVCMOS33 } [get_ports { SCL }]
set_property -dict { PACKAGE_PIN C15   IOSTANDARD LVCMOS33 } [get_ports { SDA }]
```

Figure 88: Add SDA and SCL lines to the constraints file.

Sep 2. Integration of the I2C module in the SoC:

It’s necessary to establish a bus to allow the SweRV EH1 Core to communicate with the I2C. For that case, it will be a WISHBONE bus, since the core already uses it to communicate with other peripherals.

A Verilog top module with all the Wishbone bus interface signals and an instantiation of the I2C is made.

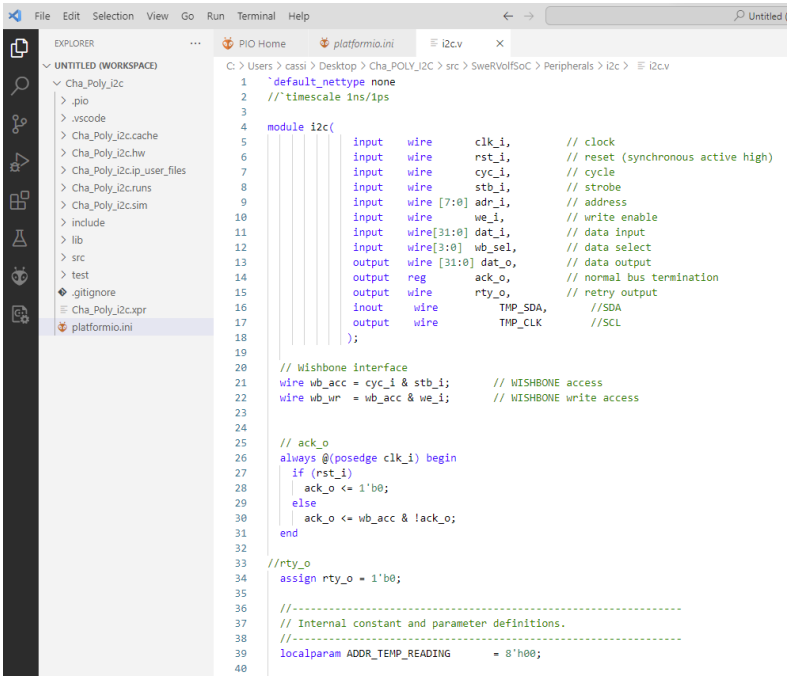


Figure 89: Verilog Wishbone bus interface signals.

The new module, i2c_master.v is then instantiated into the SOC (Figure 90).

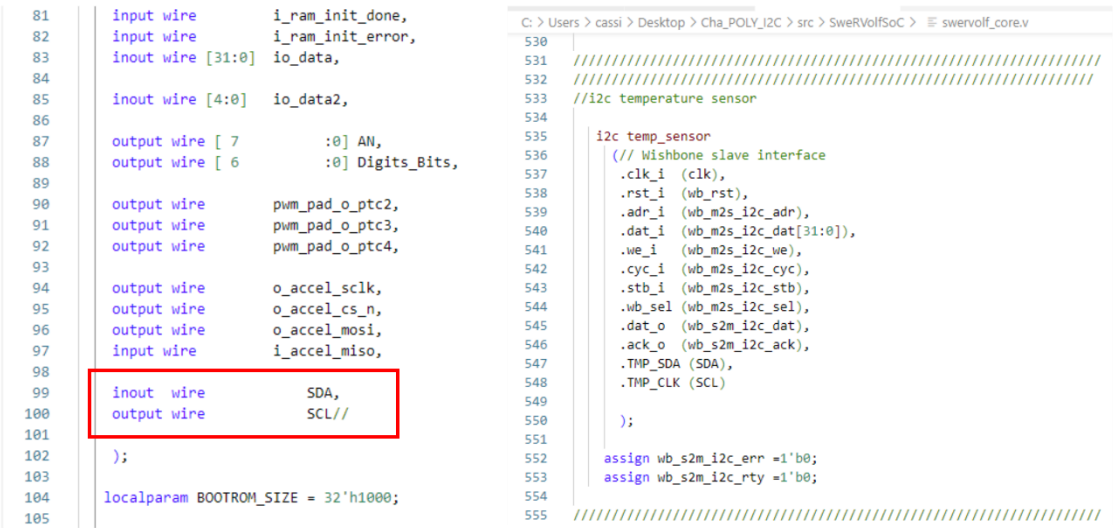


Figure 90: i2c_master.v instantiation into the SOC.

Step 3. Connection between the I2C and the SweRV EH1 Core

The device controllers are connected to the SweRV EH1 Core through a multiplexer and a bridge. The multiplexer selects one among the N possible peripherals, depending on the address generated by the CPU (Figure 91). The bridge translates the Wishbone signals used by the device controllers to the AXI4 signals used by the SweRV Core and vice versa.

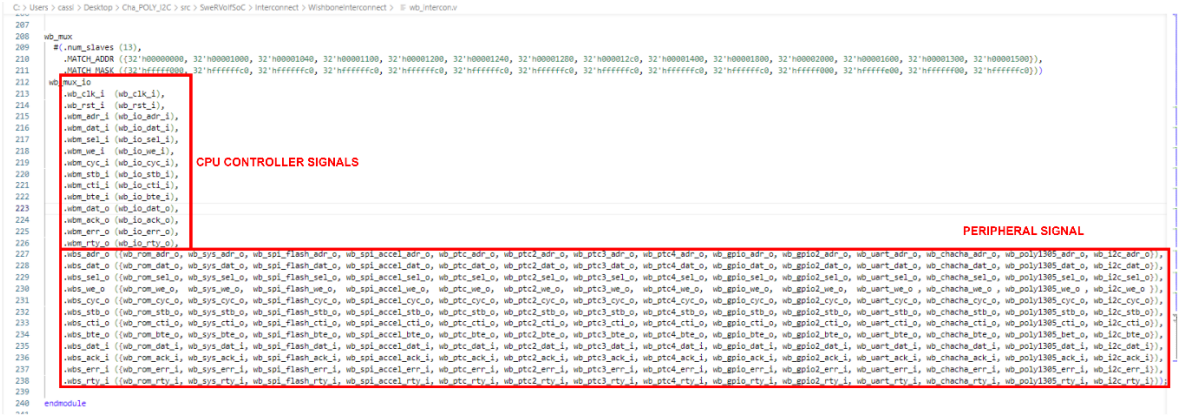


Figure 91: multiplexer selects the peripheral to connect to the CPU (wb_intercon.v).

5.1.1 I2C implementation on RVfpgaNexys

After a new module to the architecture, the process of compiling the new modified RvFpga System is the same as shown in chapter three. After generating the bitstream on Vivado, a new project in PlatformIO is created. In the .ini file, the directory to the new bitstream is added.

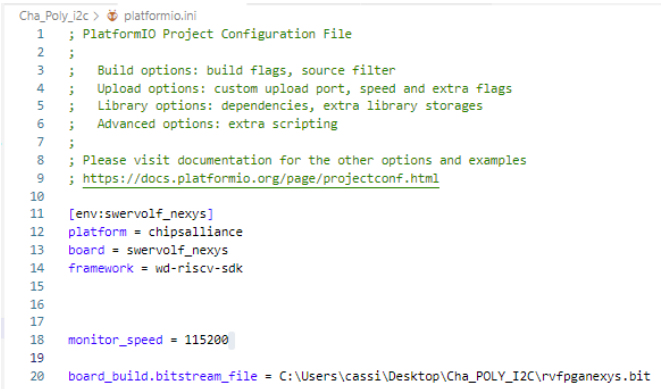


Figure 92: Platformio initialization file: platformio.ini.

It´s possible to see the temperature values registered by the I2C temperature sensor by writing a C code (Figure 93) that outputs the reading of a specific memory address where the I2C peripheral is connected (Figure 94).

C: > Users > cassi > Desktop > Cha_POLY_I2C > I2c_c_code > src > C i2c_main.c > main(void)

```c
// memory-mapped I/O addresses
#define TEMP_I2C    0x80001500

#define READ_TEMP(dir) (*(volatile unsigned *)dir)

#if defined(D_NEXYS_A7)
 #include <bsp_printf.h>
 #include <bsp_mem_map.h>
 #include <bsp_version.h>
#else
 PRE_COMPILED_MSG("no platform was defined")
#endif

#include <psp_api.h>
#define DELAY 10000000


int main(void)
{

int read_temp, i =0;

 // Initialize UART
 uartInit();

 while (1) {

   read_temp = READ_TEMP(TEMP_I2C);
   printfNexys("Temperature reading: %d\n", read_temp);

   for (i=0; i < DELAY; i++) ; // delay between printf's

 }
 return(0);

}
```

Figure 93: I2C C program.

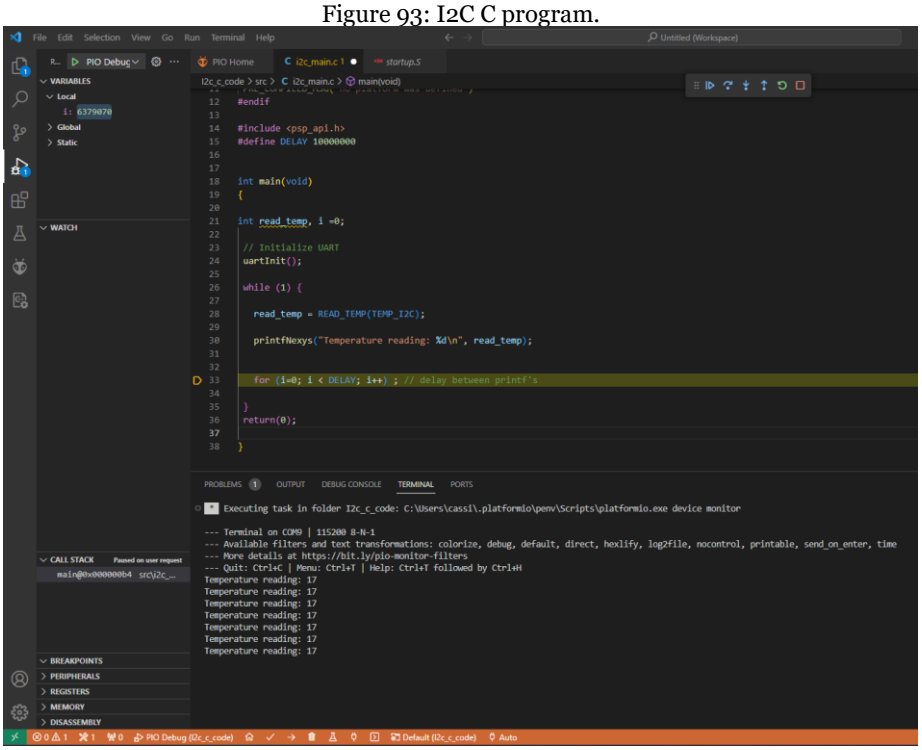


Figure 94: I2C running on RVfpgaNEXYS.

5.2 Chacha20 and Poly1305 module implementation into a RISC-V architecture

The Chacha20 encryption and the Poly1304 authentication modules were added to the RVfpga system as peripherals following the same steps as the I2C module.

5.2.1ChaCha20

The ChaCha20 C code can be found on Appendix C. It was implemented in RVfpgaNexys using a test vector from Chapter 4. The results are shown in the figure below.

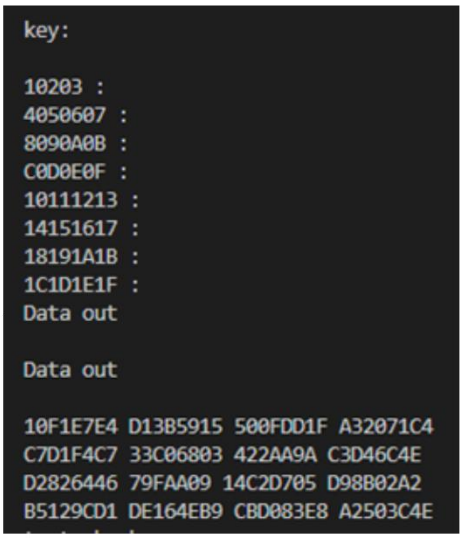


Figure 95: Chacha20 test vector on the left and RVfpgaNexys performing the same test vector at the right.

The SoC was simulated, and its wave forms were analyzed in GTKWave. It´s possible to notice that the ChaCha20 module takes 23 clock cycles to encrypt a message.

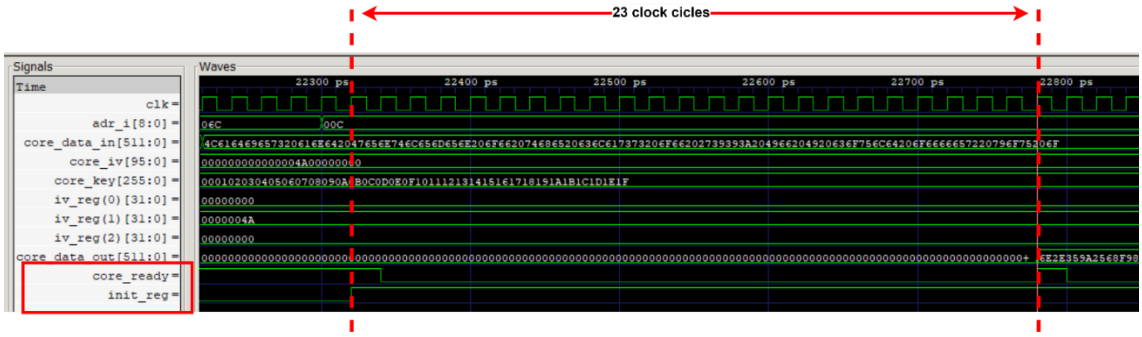


Figure 96: Chacha20 using 23clock cycles to encrypt a message.

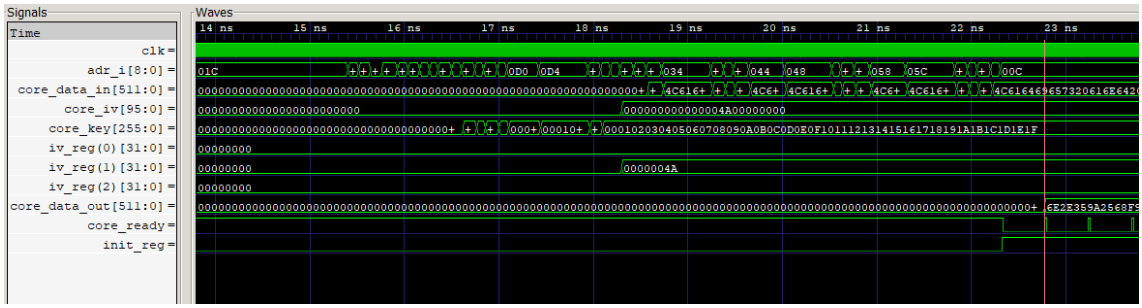


Figure 97: Chacha20 running on RVfpgaSim.

5.2.2Poly1305

The code for Poly1305 can be found in Appendix D. It was implemented in RVfpgaNexys using a test vector from Chapter 4. The results are shown in the figure below.

```
test POLY1305
DATA SENT:

43727970 746F6772  61706869  6320466F :
72756D20 52657365  61726368  38A0 :
2047726F 75700000  0  38A0 :


key:
85D6BE78 57556D33  7F4452FE  42D506A8 :
103808A FB0DB2FD  4ABFF6AF  38A0 :
4149F51B :


Generated MAC:
A8061DC1 305136C6 C22B8BAF C0127A9 :
```

Figure 98: Poly1305 running a 34-byte message test vector on a RVfpgaNexys.

The SoC was simulated, and its wave forms were analyzed in GTKWave (Figures 100, and 101).

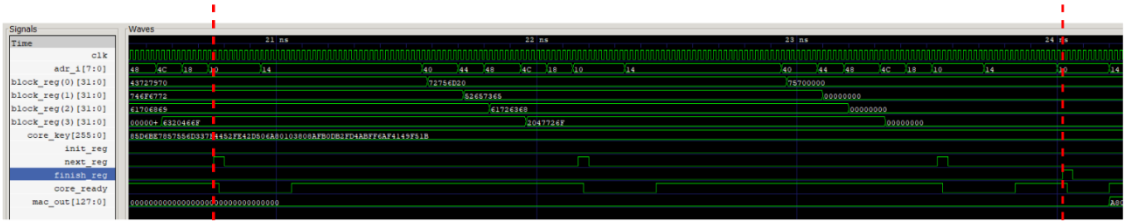


Figure 99: Poly1305 timing from the start to the end of authentication of the 34-byte test vector.

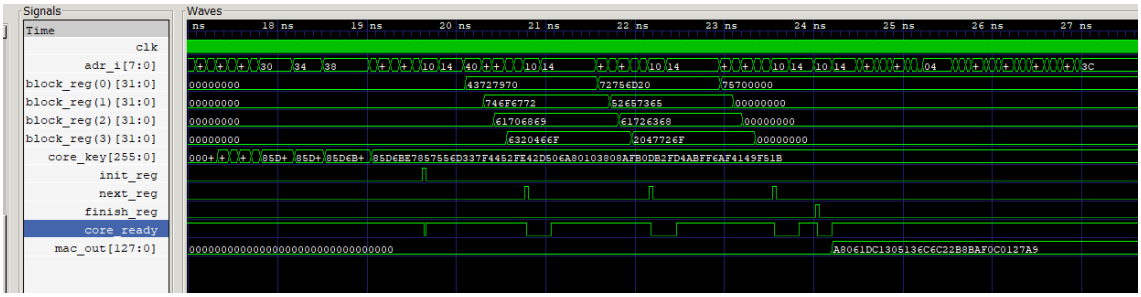


Figure 100: Poly1305 running a 34-byte message test vector on a RVfpgaSim.

5.3 Chacha20-Poly1305: send an encrypted and authenticated message through UART

The RVfpga will read temperature values from the I2C sensor integrated on the Nexys A7 FPGA, and the collected data will be encrypted and authenticated before being sent to a computer through UART. The C code can be found on Appendix E.

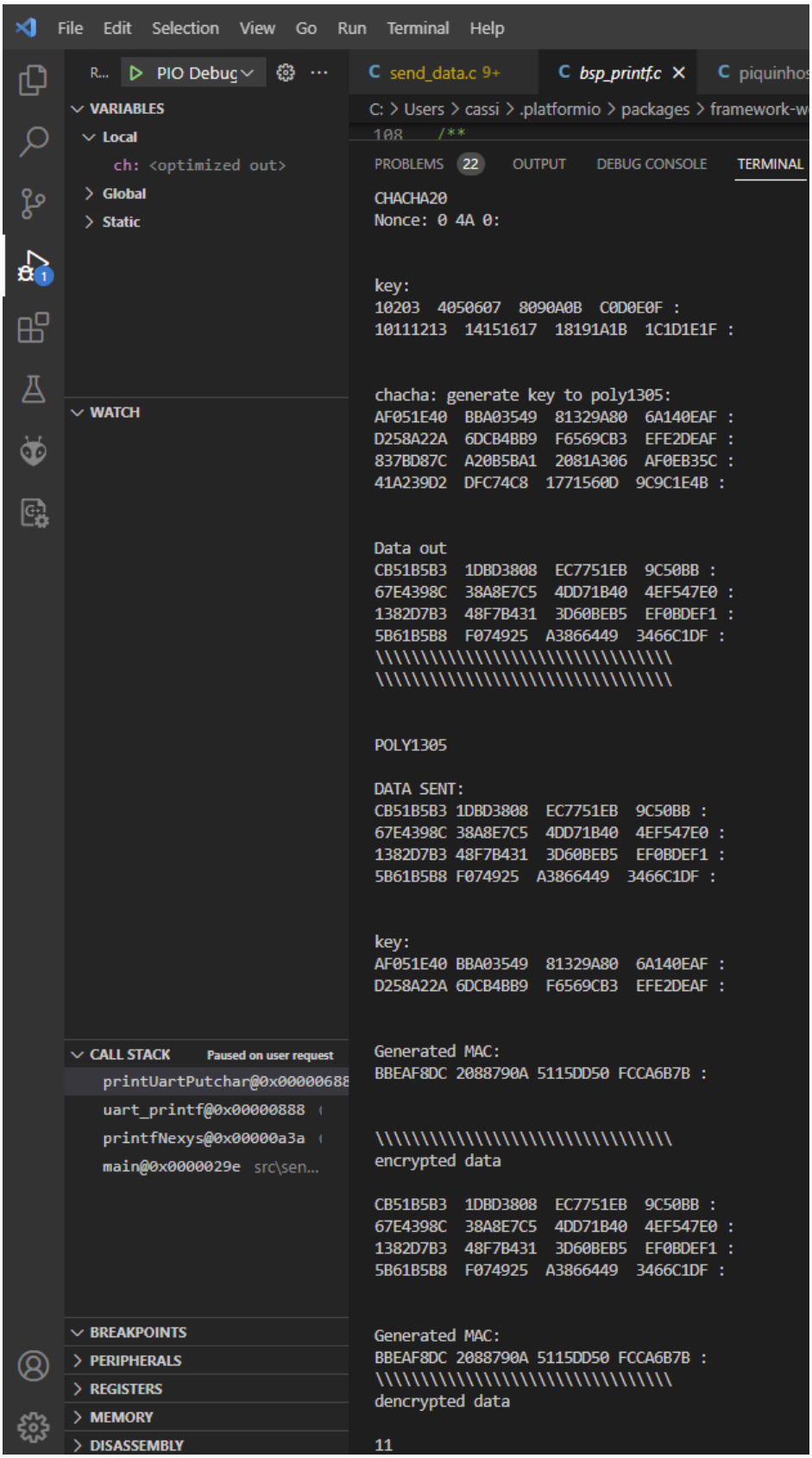


Figure 101: Sending an encrypted and authenticated temperature data from an I2C temperature sensor through UART on a RVfpgaNexys Core.

Conclusion and Future Work

In this dissertation, a survey of the RISC-V ISA architecture was performed. An emphasis was placed on the 32-bit base integer ISA, the RV32I. It is impressive to learn how a summer project grows up to be one of the most popular open-source ISAs. The RISC-V foundation has the objective of becoming the standard commercial ISA.

The students are the world's future; they are the ones responsible for keeping what is being done alive, but they also have the power to do new things. If they start to learn in the university, or even sooner, an ISA that is being used by the industry, the learning curve that usually appears when there's a transition from the academic to "real" life can be mitigated.

The more investment companies or foundations, like the RISC-V foundation, make in the newest generations, the bigger the revenue they can get in the future.

In the second part of this dissertation, to be more precise, chapter 4, an abstract survey on data encryption and authentication was done. It is essential to emphasize how exposed and, at the same time, protected society is. We are surrounded by technology. Technology also means data exchange. For systems with unlimited access to resources, it is easy to deal with all the data flow and the requirement to protect it from unauthorized systems to see, steal, or damage the information. The same does not happen with power-constraint devices. Smaller systems like smart sensor nodes have limited hardware resources and computing power in order to be able to sustain as much time as possible with the same power source, usually a battery. In order to save some computing power, the manufacturers cut some corners in the security since the algorithms require a long computational time or memory space to be implemented.

As an alternative to AES, ChaCha20-Poly1305 encryption and authentication algorithms were implemented in this dissertation. In theory, ChaCha20-Poly1305 is a good alternative, although the 512-bit cipher from ChaCha can be an unnecessary extensive use of bits for small device applications like smart sensor nodes.

For future work, I suggest implementing the ChaCha20 and the Poly1305, not as a peripheral but as part of the ISA. In other words, create a custom extension with instructions for those algorithms.

References

[1] "IoT Signals report: IoT's promise will be unlocked by addressing skills shortage, complexity and security - The Official Microsoft Blog." Accessed: Oct. 09, 2023. [Online]. Available: https://blogs.microsoft.com/blog/2019/07/30/iot-signals-report-iots-promise-will-be-unlocked-by-addressing-skills-shortage-complexity-and-security/

[2] C. Tankard, "Big data security," *Network Security*, vol. 2012, no. 7, pp. 5–8, Jul. 2012, doi: 10.1016/S1353-4858(12)70063-6.

[3] R. Kashyap and A. D. Piersson, "Impact of Big Data on Security," *https://services.igi-global.com/resolvedoi/resolve.aspx?doi=10.4018/978-1-5225-4100-4.ch015*, pp. 283–299, Jan. 1AD, doi: 10.4018/978-1-5225-4100-4.CH015.

[4] S. P. R. Asaithambi, S. Venkatraman, and R. Venkatraman, "Proposed big data architecture for facial recognition using machine learning.," *AIMS Electronics and Electrical Engineering*, vol. 5, no. 1, pp. 68–93, Mar. 2021, Accessed: Oct. 09, 2023. [Online]. Available: https://go.gale.com/ps/i.do?p=AONE&sw=w&issn=25781588&v=2.1&it=r&id=GALE%7CA684661634&sid=googleScholar&linkaccess=fulltext

[5] "33 Alarming Cybercrime Statistics You Should Know in 2019 - Security Boulevard." Accessed: Oct. 09, 2023. [Online]. Available: https://securityboulevard.com/2019/11/33-alarming-cybercrime-statistics-you-should-know-in-2019/

[6] N. Kaloudi and L. I. Jingyue, "The AI-Based Cyber Threat Landscape," *ACM Computing Surveys (CSUR)*, vol. 53, no. 1, Feb. 2020, doi: 10.1145/3372823.

[7] T. Lu, "A Survey on RISC-V Security: Hardware and Architecture," Jul. 2021, Accessed: Oct. 09, 2023. [Online]. Available: https://arxiv.org/abs/2107.04175v1

[8] J. C. V. Fernandes, H. Da Rocha, and A. Espirito-Santo, "Encryption and Authentication in Smart Transducers Implemented in RISC-V Softcore Processors," *Proceedings of the IEEE International Conference on Industrial Technology*, vol. 2023-April, 2023, doi: 10.1109/ICIT58465.2023.10143043.

[9] H. da Rocha, R. Abrishambaf, J. Pereira, and A. E. Santo, "Integrating the IEEE 1451 and IEC 61499 Standards with the Industrial Internet Reference Architecture," *Sensors*, vol. 22, no. 4, Feb. 2022, doi: 10.3390/S22041495.

[10] A. Armstrong *et al.*, "ISA semantics for ARMv8-a, RISC-v, and CHERI-MIPS," *Proceedings of the ACM on Programming Languages*, vol. 3, no. POPL, p. 31, Jan. 2019, doi: 10.1145/3290384.

[11] A. Waterman and K. A. Asanovi´c, "The RISC-V Instruction Set Manual," 2019.

[12] "OpenRISC - OpenRISC." Accessed: Oct. 09, 2023. [Online]. Available: https://openrisc.io/

[13] E. Blem, J. Menon, and K. Sankaralingam, "Power struggles: Revisiting the RISC vs. CISC debate on contemporary ARM and x86 architectures," *Proceedings - International Symposium on High-Performance Computer Architecture*, pp. 1–12, 2013, doi: 10.1109/HPCA.2013.6522302.

[14] E. Cui, T. Li, and Q. Wei, "RISC-V Instruction Set Architecture Extensions: A Survey," *IEEE Access*, vol. 11, pp. 24696–24711, 2023, doi: 10.1109/ACCESS.2023.3246491.

[15] H. Chen and H. Tang, “IEEE Standard for Technology Supervision Code for Wind Turbine Rotor Systems,” no. January, 2020, doi: 10.1109/IEEESTD.2008.4610935.
[16] “GitHub - YosysHQ/picorv32: PicoRV32 - A Size-Optimized RISC-V CPU.” Accessed: Oct. 09, 2023. [Online]. Available: https://github.com/YosysHQ/picorv32
[17] “GitHub - chipsalliance/rocket-chip: Rocket Chip Generator.” Accessed: Oct. 09, 2023. [Online]. Available: https://github.com/chipsalliance/rocket-chip
[18] “RISC-V BOOM - RISC-V BOOM.” Accessed: Oct. 09, 2023. [Online]. Available: https://boom-core.org/
[19] M. Gautschi and P. D. Schiavone, “RI5CY: User Manual,” 2019. [Online]. Available: http://solderpad.org/licenses/SHL-0.51.
[20] “A dive into RI5CY core internals – Embecosm.” Accessed: Oct. 30, 2023. [Online]. Available: https://www.embecosm.com/2019/08/13/a-dive-into-ri5cy-core-internals/
[21] “THE IMAGINATION UNIVERSITY PROGRAMME RVfpga Getting Started Guide RVfpga Getting Started Guide.” [Online]. Available: https://www.linkedin.com/in/valeros/
[22] “Vivado Overview.” Accessed: Oct. 30, 2023. [Online]. Available: https://www.xilinx.com/products/design-tools/vivado.html
[23] “Icarus Verilog for Windows.” Accessed: Oct. 30, 2023. [Online]. Available: https://bleyer.org/icarus/
[24] “GTKWave.” Accessed: Oct. 30, 2023. [Online]. Available: https://gtkwave.sourceforge.net/
[25] R. Nane *et al.*, “A Survey and Evaluation of FPGA High-Level Synthesis Tools,” *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 35, no. 10, pp. 1591–1604, 2016, doi: 10.1109/TCAD.2015.2513673.
[26] R. Nane *et al.*, “A Survey and Evaluation of FPGA High-Level Synthesis Tools,” *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 35, no. 10, pp. 1591–1604, 2016, doi: 10.1109/TCAD.2015.2513673.
[27] “Nexys A7 Reference Manual - Digilent Reference.” Accessed: Oct. 30, 2023. [Online]. Available: https://digilent.com/reference/programmable-logic/nexys-a7/reference-manual
[28] A. Devices, “±0.25°C Accurate, 16-Bit Digital I 2 C Temperature Sensor”, Accessed: Oct. 30, 2023. [Online]. Available: www.analog.com
[29] “Wishbone B4 WISHBONE System-on-Chip (SoC)Interconnection Architecturefor Portable IP Cores Brought to You By OpenCores,” 2010, Accessed: Oct. 30, 2023. [Online]. Available: www.opencores.org
[30] J. C. V. Fernandes, H. Da Rocha, and A. Espirito-Santo, “Encryption and Authentication in Smart Transducers Implemented in RISC-V Softcore Processors,” *Proceedings of the IEEE International Conference on Industrial Technology*, vol. 2023-April, 2023, doi: 10.1109/ICIT58465.2023.10143043.
[31] D. J. Bernstein, “ChaCha, a variant of Salsa20”.
[32] Y. Nir and A. Langley, “ChaCha20 and Poly1305 for IETF Protocols,” Jun. 2018, doi: 10.17487/RFC8439.
[33] “GitHub - secworks/chacha: Verilog 2001 implementation of the ChaCha stream cipher.” Accessed: Oct. 30, 2023. [Online]. Available: https://github.com/secworks/chacha

[34] D. J. Bernstein, “The Poly1305-AES message-authentication code”, Accessed: Feb. 15, 2023. [Online]. Available: http://cr.yp.to/talks.html#2002.06.15,
[35] “GitHub - secworks/poly1305: Hardware implementation of the poly1305 message authentication function.” Accessed: Oct. 30, 2023. [Online]. Available: https://github.com/secworks/poly1305
[36] R. Serrano, C. Duran, M. Sarmiento, C. K. Pham, and T. T. Hoang, “ChaCha20–Poly1305 Authenticated Encryption with Additional Data for Transport Layer Security 1.3,” *Cryptography*, vol. 6, no. 2, Jun. 2022, doi: 10.3390/CRYPTOGRAPHY6020030.
[37] L. Ada, “ChaCha 20 – Poly 1305.” [Online]. Available: https://www.adalabs.com/adalabs-chacha20poly1305.pdf
[38] B. Marshall, G. R. Newell, D. Page, M.-J. O. Saarinen, and C. Wolf, “The design of scalar AES Instruction Set Extensions for RISC-V”, Accessed: Oct. 30, 2023. [Online]. Available: https://github.com/mjosaarinen/sm4ni
[39] “RISC-V Cryptography Extensions Task Group Announces Public Review of the Scalar Cryptography Extensions – RISC-V International.” Accessed: Oct. 30, 2023. [Online]. Available: https://riscv.org/blog/2021/09/risc-v-cryptography-extensions-task-group-announces-public-review-of-the-scalar-cryptography-extensions/

Appendix A

```
`default_nettype none

module i2c_master(
        input   wire clk_200KHz,
        input   wire reset,
        inout   wire SDA,
        output  wire [7:0]  temp_data,
        output  wire  SDA_dir,
        output  wire SCL
        );

        reg [3:0] couter = 4'b0;
        reg clk_reg = 1'b1;


//*****************************************************************

// criar o clk de 10KHz

        always @(posedge clk_200KHz or posedge reset)
            if (reset) begin
                couter = 4'b0;
                clk_reg = 1'b0;
            end
            else
                if(couter == 9) begin
                    couter <= 4'b0;
                    clk_reg <= ~clk_reg;
                end
                else
                    couter <= couter+1;



        assign SCL = clk_reg;

// ***************************************************************

        parameter [7:0] sensor_address_plus_read = 8'b1001_0111;
        reg [7:0] tMSB = 8'b0;
        reg [7:0] tLSB = 8'b0;
        reg o_bit = 1'b1;
        wire i_bit;
        reg [11:0] count = 12'b0;
        reg [7:0] temp_data_reg;
```

```
//estados:

localparam [4:0] POWER_UP   = 5'h00,
                 START      = 5'h01,

                 SEND_ADDR6 = 5'h02,
                 SEND_ADDR5 = 5'h03,
                 SEND_ADDR4 = 5'h04,
                 SEND_ADDR3 = 5'h05,
                 SEND_ADDR2 = 5'h06,
                 SEND_ADDR1 = 5'h07,
                 SEND_ADDR0 = 5'h08,
                 SEND_RW    = 5'h09,

                 REC_ACK    = 5'h0A,
                 REC_MSB7   = 5'h0B,

                 REC_MSB6   = 5'h0C,
                 REC_MSB5   = 5'h0D,
                 REC_MSB4   = 5'h0E,
                 REC_MSB3   = 5'h0F,
                 REC_MSB2   = 5'h10,
                 REC_MSB1   = 5'h11,
                 REC_MSB0   = 5'h12,

                 SEND_ACK   = 5'h13,
                 REC_LSB7   = 5'h14,

                 REC_LSB6   = 5'h15,
                 REC_LSB5   = 5'h16,
                 REC_LSB4   = 5'h17,
                 REC_LSB3   = 5'h18,
                 REC_LSB2   = 5'h19,
                 REC_LSB1   = 5'h1A,
                 REC_LSB0   = 5'h1B,

                 NACK       = 5'h1C;

reg [4:0] state_reg = POWER_UP;

always @(posedge clk_200KHz or posedge reset) begin
    if(reset) begin
        state_reg <= START;
        count <= 12'd2000;
    end
    else begin
        count <= count +1;
```

```
                case (state_reg)
                    POWER_UP    :begin
                                    if (count == 12'd1999)
                                        state_reg <= START;
                    end
                    START       :begin
                                    if (count == 12'd2004)
                                        o_bit <= 1'b0;
                                    if (count == 12'd2013)
                                        state_reg <= SEND_ADDR6;
                    end

//------------------------------------------------------------------------
----
                    SEND_ADDR6  :begin
                                    o_bit <= sensor_address_plus_read[7];
                                    if (count == 12'd2033)
                                        state_reg <= SEND_ADDR5;
                    end
                    SEND_ADDR5  :begin
                                    o_bit <= sensor_address_plus_read[6];
                                    if (count == 12'd2053)
                                        state_reg <= SEND_ADDR4;
                    end
                    SEND_ADDR4  :begin
                                    o_bit <= sensor_address_plus_read[5];
                                    if (count == 12'd2073)
                                        state_reg <= SEND_ADDR3;
                    end
                    SEND_ADDR3  :begin
                                    o_bit <= sensor_address_plus_read[4];
                                    if (count == 12'd2093)
                                        state_reg <= SEND_ADDR2;
                    end
                    SEND_ADDR2  :begin
                                    o_bit <= sensor_address_plus_read[3];
                                    if (count == 12'd2113)
                                        state_reg <= SEND_ADDR1;
                    end
                    SEND_ADDR1  :begin
                                    o_bit <= sensor_address_plus_read[2];
                                    if (count == 12'd2133)
                                        state_reg <= SEND_ADDR0;
                    end
                    SEND_ADDR0  :begin
                                    o_bit <= sensor_address_plus_read[1];
                                    if (count == 12'd2153)
                                        state_reg <= SEND_RW;
                    end
```

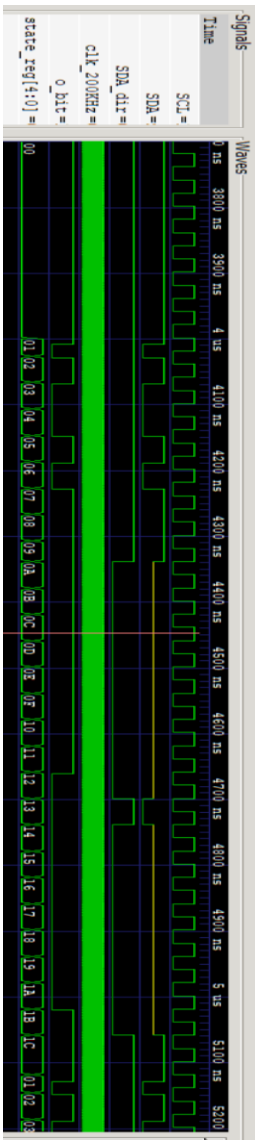
```
//-----------------------------------------------------------------------
--

                    SEND_RW     :begin
                                    o_bit <= sensor_address_plus_read[0];
                                    if (count == 12'd2169)
                                        state_reg <= REC_ACK;
                    end
                    REC_ACK     :begin
                                    if (count == 12'd2189)
                                        state_reg   <= REC_MSB7;
                    end

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

                    REC_MSB7    :begin
                                    tMSB[7] <= i_bit;
                                    if(count == 12'd2209)
                                        state_reg <=
REC_MSB6;
                    end
                    REC_MSB6        :begin
                                        tMSB[6] <= i_bit;
                                        if(count == 12'd2229)
                                            state_reg <= REC_MSB5;
                    end
                    REC_MSB5        :begin
                                        tMSB[5] <= i_bit;
                                        if(count == 12'd2249)
                                            state_reg <= REC_MSB4;

                    end
                    REC_MSB4        :begin
                                        tMSB[4] <= i_bit;
                                        if(count == 12'd2269)
                                            state_reg <= REC_MSB3;
                    end
                    REC_MSB3        :begin
                                        tMSB[3] <= i_bit;
                                        if(count == 12'd2289)
                                            state_reg <= REC_MSB2;
                    end
                    REC_MSB2        :begin
                                        tMSB[2] <= i_bit;
                                        if(count == 12'd2309)
                                            state_reg <= REC_MSB1;
                    end
                    REC_MSB1        :begin
```

```
                                tMSB[1] <= i_bit;
                                if(count == 12'd2329)
                                    state_reg <= REC_MSB0;

                    end
                    REC_MSB0        :begin
                                tMSB[0] <= i_bit;
                                if(count == 12'd2349)
                                    state_reg <= SEND_ACK;
                    end
//------------------------------------------------------------------------
------

                        SEND_ACK    :begin
                                if(count == 12'd2369)
                                    state_reg <= REC_LSB7;
                        end

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

                    REC_LSB7        :begin
                                tLSB[7] <= i_bit;
                                if(count == 12'd2389)
                                    state_reg <= REC_LSB6;
                    end
                    REC_LSB6        :begin
                                tLSB[6] <= i_bit;
                                if(count == 12'd2409)
                                    state_reg <= REC_LSB5;
                    end
                    REC_LSB5        :begin
                                tLSB[5] <= i_bit;
                                if(count == 12'd2429)
                                    state_reg <= REC_LSB4;
                    end
                    REC_LSB4        :begin
                                tLSB[4] <= i_bit;
                                if(count == 12'd2449)
                                    state_reg <= REC_LSB3;
                    end
                    REC_LSB3        :begin
                                tLSB[3] <= i_bit;
                                if(count == 12'd2469)
                                    state_reg <= REC_LSB2;
                    end
                    REC_LSB2        :begin
                                tLSB[2] <= i_bit;
```

```
                                    if(count == 12'd2489)
                                        state_reg <= REC_LSB1;
                    end
                    REC_LSB1        :begin
                                    tLSB[1] <= i_bit;
                                    if(count == 12'd2509)
                                        state_reg <= REC_LSB0;
                    end
                    REC_LSB0        :begin
                                    o_bit <= 1'b1;
                                    tLSB[0] <= i_bit;
                                    if(count == 12'd2529)
                                        state_reg <= NACK;
                    end

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

                    NACK            :begin
                                    if (count == 12'd2559) begin
                                        count <=12'd2000;
                                        state_reg <= START;
                    end
                end
                endcase
            end
        end

// Buffer da temperatura

        always @(posedge clk_200KHz)
            if(state_reg == NACK)
                temp_data_reg <= { tMSB[6:0],tLSB[7]};


// controlo da direção do SDA

        assign SDA_dir = (state_reg == POWER_UP || state_reg == START ||
state_reg == SEND_ADDR6 || state_reg == SEND_ADDR5 ||
                        state_reg == SEND_ADDR4 || state_reg == SEND_ADDR3
|| state_reg == SEND_ADDR2 || state_reg == SEND_ADDR1 ||
                        state_reg == SEND_ADDR0 || state_reg == SEND_RW ||
state_reg == SEND_ACK || state_reg == NACK) ? 1 : 0;

    // defenir o SDA como output - do master para o sensor
    assign SDA = SDA_dir ? o_bit : 1'bz;


    // definir o SDA como input - do sensor para o master
```

```
    assign i_bit = SDA;

    // output do valor de temperatura
    assign temp_data = temp_data_reg;

endmodule
```

Appendix B


Signals
Time
SCL
SDA
SDA_dir
clk 200MHz
o_bit
state_reg[4:0]
Waves

Appendix C

```
#if defined(D_NEXYS_A7)
#include <bsp_printf.h>
#include <bsp_mem_map.h>
#include <bsp_version.h>
#else
PRE_COMPILED_MSG("no platform was defined")
#endif
#include <psp_api.h>

#define uart_data (*(volatile unsigned int *)0x80002000)
#define uart_lsr (*(volatile unsigned int *)0x80002014)

#define uart_lsr_r_mask 0x01
#define uart_lsr_w_mask 0x20


////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////
////////                            chacha20
////////
////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////

// memory-mapped I/O addresses
#define GPIO_SWs        0x80001400
#define GPIO_LEDs       0x80001404
#define GPIO_INOUT      0x80001408

#define NAME0           0x80001600
#define NAME1           0x80001604
#define VERSION         0x80001608

#define CTRL            0x8000160c
#define STATUS          0x80001610
#define KEYLEN          0x80001614
#define ROUNDS          0x80001618
#define COUNTER         0x8000161c

#define KEY0            0x800016c0
#define KEY1            0x800016c4
#define KEY2            0x800016c8
#define KEY3            0x800016cc
#define KEY4            0x800016d0
#define KEY5            0x800016d4
#define KEY6            0x800016d8
#define KEY7            0x800016dc
```

```
#define IV0             0x80001624
#define IV1             0x80001628
#define IV2             0x8000162c

#define DATA_IN0        0x80001630
#define DATA_IN1        0x80001634
#define DATA_IN2        0x80001638
#define DATA_IN3        0x8000163c
#define DATA_IN4        0x80001640
#define DATA_IN5        0x80001644
#define DATA_IN6        0x80001648
#define DATA_IN7        0x8000164c
#define DATA_IN8        0x80001650
#define DATA_IN9        0x80001654
#define DATA_IN10       0x80001658
#define DATA_IN11       0x8000165c
#define DATA_IN12       0x80001660
#define DATA_IN13       0x80001664
#define DATA_IN14       0x80001668
#define DATA_IN15       0x8000166c

#define DATA_OUT0       0x80001680
#define DATA_OUT1       0x80001684
#define DATA_OUT2       0x80001688
#define DATA_OUT3       0x8000168c
#define DATA_OUT4       0x80001690
#define DATA_OUT5       0x80001694
#define DATA_OUT6       0x80001698
#define DATA_OUT7       0x8000169c
#define DATA_OUT8       0x800016a0
#define DATA_OUT9       0x800016a4
#define DATA_OUT10      0x800016a8
#define DATA_OUT11      0x800016ac
#define DATA_OUT12      0x800016b0
#define DATA_OUT13      0x800016b4
#define DATA_OUT14      0x800016b8
#define DATA_OUT15      0x800016bc


#define DATA_CHIFRA0        0x80001700
#define DATA_CHIFRA1        0x80001704
#define DATA_CHIFRA2        0x80001708
#define DATA_CHIFRA3        0x8000170c
#define DATA_CHIFRA4        0x80001710
#define DATA_CHIFRA5        0x80001714
#define DATA_CHIFRA6        0x80001718
#define DATA_CHIFRA7        0x8000171c
#define DATA_CHIFRA8        0x80001720
```

```
#define DATA_CHIFRA9       0x80001724
#define DATA_CHIFRA10      0x80001728
#define DATA_CHIFRA11      0x8000172c
#define DATA_CHIFRA12      0x80001730
#define DATA_CHIFRA13      0x80001734
#define DATA_CHIFRA14      0x80001738
#define DATA_CHIFRA15      0x8000173c


#define READ_CHACHA(dir) (*(volatile unsigned *)dir)
#define WRITE_CHACHA(dir, value) { (*(volatile unsigned *)dir) = (value);
}



void    chacha20_gen (  int key_len,
                    int  numb_rounds,
                    int count_init_val,
                    int k0,
                    int k1,
                    int k2,
                    int k3,
                    int k4,
                    int k5,
                    int k6,
                    int k7,
                    int N0,
                    int N1,
                    int N2,
                    int data0,
                    int data1,
                    int data2,
                    int data3,
                    int data4,
                    int data5,
                    int data6,
                    int data7,
                    int data8,
                    int data9,
                    int data10,
                    int data11,
                    int data12,
                    int data13,
                    int data14,
                    int data15,
                    int controlo,
                    int dout[]
                    )
{
```

```
    int i=1,s;
////////////////////////////////////////////////////////////////////
 //variaveis do chacha

//keylen to 256bit
    WRITE_CHACHA(KEYLEN,key_len);

// number of rounds to 20
    WRITE_CHACHA(ROUNDS,numb_rounds);

// set up the counter
    WRITE_CHACHA(COUNTER, count_init_val);

// set the key
    WRITE_CHACHA(KEY0,k0);
    WRITE_CHACHA(KEY1,k1);
    WRITE_CHACHA(KEY2,k2);
    WRITE_CHACHA(KEY3,k3);
    WRITE_CHACHA(KEY4,k4);
    WRITE_CHACHA(KEY5,k5);
    WRITE_CHACHA(KEY6,k6);
    WRITE_CHACHA(KEY7,k7);

// set the nonce
    WRITE_CHACHA(IV0,N0);
    WRITE_CHACHA(IV1,N1);
    WRITE_CHACHA(IV2,N2);


// data to chacha
    WRITE_CHACHA(DATA_IN0, data0 );
    WRITE_CHACHA(DATA_IN1, data1 );
    WRITE_CHACHA(DATA_IN2, data2 );
    WRITE_CHACHA(DATA_IN3, data3 );
    WRITE_CHACHA(DATA_IN4, data4 );
    WRITE_CHACHA(DATA_IN5, data5 );
    WRITE_CHACHA(DATA_IN6, data6 );
    WRITE_CHACHA(DATA_IN7, data7 );
    WRITE_CHACHA(DATA_IN8, data8 );
    WRITE_CHACHA(DATA_IN9, data9 );
    WRITE_CHACHA(DATA_IN10,data10);
    WRITE_CHACHA(DATA_IN11,data11);
    WRITE_CHACHA(DATA_IN12,data12);
    WRITE_CHACHA(DATA_IN13,data13);
    WRITE_CHACHA(DATA_IN14,data14);
    WRITE_CHACHA(DATA_IN15,data15);


// data back from chacha
```

```
int d0,
    d1,
    d2,
    d3,
    d4,
    d5,
    d6,
    d7,
    d8,
    d9,
    d10,
    d11,
    d12,
    d13,
    d14,
    d15;


//control variables
WRITE_CHACHA(CTRL, controlo );

int wait=1;
int estado=0x00;

while (wait==1) {
    estado=READ_CHACHA(STATUS);
    if (estado!=0x00){
        wait=0;
    }
}

d0  = READ_CHACHA(DATA_OUT0);
d1  = READ_CHACHA(DATA_OUT1);
d2  = READ_CHACHA(DATA_OUT2);
d3  = READ_CHACHA(DATA_OUT3);
d4  = READ_CHACHA(DATA_OUT4);
d5  = READ_CHACHA(DATA_OUT5);
d6  = READ_CHACHA(DATA_OUT6);
d7  = READ_CHACHA(DATA_OUT7);
d8  = READ_CHACHA(DATA_OUT8);
d9  = READ_CHACHA(DATA_OUT9);
d10 = READ_CHACHA(DATA_OUT10);
d11 = READ_CHACHA(DATA_OUT11);
d12 = READ_CHACHA(DATA_OUT12);
d13 = READ_CHACHA(DATA_OUT13);
d14 = READ_CHACHA(DATA_OUT14);
d15 = READ_CHACHA(DATA_OUT15);

dout[0]=d0;
```

```
    dout[1]=d1;
    dout[2]=d2;
    dout[3]=d3;
    dout[4]=d4;
    dout[5]=d5;
    dout[6]=d6;
    dout[7]=d7;
    dout[8]=d8;
    dout[9]=d9;
    dout[10]=d10;
    dout[11]=d11;
    dout[12]=d12;
    dout[13]=d13;
    dout[14]=d14;
    dout[15]=d15;




    return(dout[0],dout[1],dout[2],dout[3],dout[4],dout[5],dout[6],dout[7
],
                                                                    dout[
8],dout[9],dout[10],dout[11],dout[12],dout[13],dout[14],dout[15]);

}


////////////////////////////////////////////////////////////////////////
//////////////////////////////////////////////
/////////                          UART
/////////
////////////////////////////////////////////////////////////////////////
//////////////////////////////////////////////


int uart_getchar(void)
{
    int c;
    while((uart_lsr_w_mask & uart_lsr) == 0);
    while((uart_lsr_r_mask & uart_lsr) == 0);
    c = uart_data;

    int i=0, j;

    i = (int)c;

    if (i>47 && i<58) j=i-48;
    else if (i>96 && i<103) j=i-87;
    else if (i>64 && i<71) j=i-55;
```

```
    else j=0;

    return j;
}


int uart_putchar(int c)
{

    uart_data = c;

    return 0;
}

int read_32bit(void)
{
    int b=0,y=0;

    unsigned int x=0;
    unsigned char a[4];

    b = uart_getchar();
    y = uart_getchar()<<4;
    a[0] =  (b|y) & 0xFF;
    b = uart_getchar();
    y = uart_getchar()<<4;
    a[1] =  (b|y) & 0xFF;
    b = uart_getchar();
    y = uart_getchar()<<4;
    a[2] =  (b|y) & 0xFF;
    b = uart_getchar();
    y = uart_getchar()<<4;
    a[3] =  (b|y) & 0xFF;

    x = *(int *)a;

    return (x);
}




////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////
/////////                          MAIN
/////////
////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////
```

```
int main ( void )
{
    uartInit();
    //int i;
    int key_len,
        numb_rounds,
        count_init_val,
        k0,
        k1,
        k2,
        k3,
        k4,
        k5,
        k6,
        k7,
        N0,
        N1,
        N2,
        data0,
        data1,
        data2,
        data3,
        data4,
        data5,
        data6,
        data7,
        data8,
        data9,
        data10,
        data11,
        data12,
        data13,
        data14,
        data15;

    printfNexys("ready to receive data:\n");

    printfNexys("key[0]:\n");
    k0 = read_32bit();
    printfNexys("%X\n",k0);
    printfNexys("key[1]:\n");
    k1 = read_32bit();
    printfNexys("%X\n",k1);
    printfNexys("key[2]:\n");
    k2 = read_32bit();
    printfNexys("%X\n",k2);
    printfNexys("key[3]:\n");
    k3 = read_32bit();
    printfNexys("%X\n",k3);
```

```
printfNexys("key[4]:\n");
k4 = read_32bit();
printfNexys("%X\n",k4);
printfNexys("key[5]:\n");
k5 = read_32bit();
printfNexys("%X\n",k5);
printfNexys("key[6]:\n");
k6 = read_32bit();
printfNexys("%X\n",k6);
printfNexys("key[7]:\n");
k7 = read_32bit();
printfNexys("%X\n",k7);

printfNexys("Key: %X_%X_%X_%X_%X_%X_%X_%X",k0,k1,k2,k3,k4,k5,k6,k7);

printfNexys("key_len:");
key_len = read_32bit();
printfNexys("%X\n",key_len);


printfNexys("numb_rounds:");
numb_rounds = read_32bit();
printfNexys("%X\n",numb_rounds);

printfNexys("count_init_val:");
count_init_val = read_32bit();
printfNexys("%X\n",count_init_val);

printfNexys("N0:");
N0 = read_32bit();
printfNexys("%X\n",N0);
printfNexys("N1:");
N1 = read_32bit();
printfNexys("%X\n",N1);
printfNexys("N2:");
N2 = read_32bit();
printfNexys("%X\n",N2);

printfNexys("data to enchript:");
printfNexys("D0:");
data0 = read_32bit();
printfNexys("D1:");
data1 = read_32bit();
printfNexys("D2:");
data2 = read_32bit();
printfNexys("D3:");
data3 = read_32bit();
printfNexys("D4:");
data4 = read_32bit();
```

```
    printfNexys("D5:");
    data5 = read_32bit();
    printfNexys("D6:");
    data6 = read_32bit();
    printfNexys("D7:");
    data7 = read_32bit();
    printfNexys("D8:");
    data8 = read_32bit();
    printfNexys("D9:");
    data9 = read_32bit();
    printfNexys("D10:");
    data10 = read_32bit();
    printfNexys("D11:");
    data11 = read_32bit();
    printfNexys("D12:");
    data12 = read_32bit();
    printfNexys("D13:");
    data13 = read_32bit();
    printfNexys("D14:");
    data14 = read_32bit();
    printfNexys("D15:");
    data15 = read_32bit();

    printfNexys("data in:\n");
    printfNexys("%X:%X:%X:%X:%X:%X:%X:%X:%X:%X:%X:%X:%X:%X:%X:%X",
data0,data1,data2,data3,data4,data5,data6,data7,data8,
                                                                data9
,data10,data11,data12,data13,data14,data15);


    int dout[16];

    //chacha20_gen(key_len,numb_rounds,count_init_val,k0,k1,k2,k3,k4,k5,k
6,k7,
    chacha20_gen(0x01,0x14,0x01,k0,k1,k2,k3,k4,k5,k6,k7,
                                                                N0,N1
,N2,data0,data1,data2,data3,data4,data5,data6,data7,data8,data9,
                                                                data1
0,data11,data12,data13,data14,data15,0x01,dout);


    printfNexys("Data out:\n");
    printfNexys("%X:%X:%X:%X:%X:%X:%X:%X:%X:%X:%X:%X:%X:%X:%X:%X",dout[0]
,dout[1],dout[2],dout[3],dout[4],dout[5],dout[6],dout[7],
                                                                dout[
8],dout[9],dout[10],dout[11],dout[12],dout[13],dout[14],dout[15]);

    return(0);
}
```

Appendix D

```
// memory-mapped I/O addresses
#define GPIO_SWs        0x80001400
#define GPIO_LEDs       0x80001404
#define GPIO_INOUT      0x80001408

//poly1305
#define NAME_POLY_0     0x80001600
#define NAME_POLY_1     0x80001604
#define VERSION_POLY    0x80001608

#define CTRL_POLY       0x80001610
#define STATUS_POLY     0x80001614

#define BLOCKLEN_POLY   0x80001618

#define KEY_POLY_0      0x80001620
#define KEY_POLY_1      0x80001624
#define KEY_POLY_2      0x80001628
#define KEY_POLY_3      0x8000162c
#define KEY_POLY_4      0x80001630
#define KEY_POLY_5      0x80001634
#define KEY_POLY_6      0x80001638
#define KEY_POLY_7      0x8000163c

#define BLOCK_POLY_0    0x80001640
#define BLOCK_POLY_1    0x80001644
#define BLOCK_POLY_2    0x80001648
#define BLOCK_POLY_3    0x8000164c

#define MAC_POLY_0      0x80001650
#define MAC_POLY_1      0x80001654
#define MAC_POLY_2      0x80001658
#define MAC_POLY_3      0x8000165c

////////////////////////////////////

#define READ_GPIO(dir) (*(volatile unsigned *)dir)
#define WRITE_GPIO(dir, value) { (*(volatile unsigned *)dir) = (value); }

#define READ_POLY1305(dir) (*(volatile unsigned *)dir)
#define WRITE_POLY1305(dir, value) { (*(volatile unsigned *)dir) =
(value); }

#if defined(D_NEXYS_A7)
 #include <bsp_printf.h>
```

```
 #include <bsp_mem_map.h>
 #include <bsp_version.h>
#else
 PRE_COMPILED_MSG("no platform was defined")
#endif

#include <psp_api.h>

#define DELAY 10000000
#define DLEAY_SHORT 50


int main ( void )
{
    int i;
 // Initialize UART
    uartInit();

 //definir o sentido dos pinos I/O
    int En_Value=0xFFFF, switches_value;
    WRITE_GPIO(GPIO_INOUT, En_Value);


////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////////
// INIT POLY1305

//POLY1305 key
int key_p0 = 0x85d6be78,
    key_p1 = 0x57556d33,
    key_p2 = 0x7f4452fe,
    key_p3 = 0x42d506a8,
    key_p4 = 0x0103808a,
    key_p5 = 0xfb0db2fd,
    key_p6 = 0x4abff6af,
    key_p7 = 0x4149f51b;

int k0 = 0,
    k1 = 0,
    k2 = 0,
    k3 = 0,
    k4 = 0,
    k5 = 0,
    k6 = 0,
    k7 = 0;

//POLY1305 message block

int block_len = 0x10;
```

```
int block_p0 = 0x00,
    block_p1 = 0x00,
    block_p2 = 0x00,
    block_p3 = 0x00;
int d0 =0x43727970,
    d1 =0x746f6772,
    d2 =0x61706869,
    d3 =0x6320466f,

    d4 =0x72756d20,
    d5 =0x52657365,
    d6 =0x61726368,
    d7 =0x2047726f,

    d8 =0x75700000,
    d9 =0x00;
//POLY1305 MAC
int mac_p0 =0x00,
    mac_p1 = 0x00,
    mac_p2 =0x00,
    mac_p3 =0x00;

//POLY1305 CONTROL SIGNALS

int INIT_POLY = 0x01,
    NEXT_POLY = 0x02,
    FINISH_POLY = 0x04;

//POLY1305 STATUS COMPARISON SIGNALS

int READY_POLY = 0x01;

////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////
//////////////////////////////////////////////////////////
//////////////////////////////////////////////////////////
////////////////// POLY1305 START ////////////////////////
//while (1) {
//write the key
WRITE_POLY1305(KEY_POLY_0,key_p0);
WRITE_POLY1305(KEY_POLY_1,key_p1);
WRITE_POLY1305(KEY_POLY_2,key_p2);
WRITE_POLY1305(KEY_POLY_3,key_p3);
WRITE_POLY1305(KEY_POLY_4,key_p4);
WRITE_POLY1305(KEY_POLY_5,key_p5);
WRITE_POLY1305(KEY_POLY_6,key_p6);
WRITE_POLY1305(KEY_POLY_7,key_p7);
```

```
//send block clear
WRITE_POLY1305(BLOCK_POLY_0,0x00);
WRITE_POLY1305(BLOCK_POLY_1,0x00);
WRITE_POLY1305(BLOCK_POLY_2,0x00);
WRITE_POLY1305(BLOCK_POLY_3,0x00);

//send an INIT value to CTRL
WRITE_POLY1305(CTRL_POLY,INIT_POLY);


//wait for ready status
int s = 0x01;
while (s == READY_POLY)
    {
        s = READ_POLY1305(STATUS_POLY);
        //printfNexys("START:\n");
        //printfNexys("%X:\n",s);
    }

////////////////////////////////////////////////////////
////////////// POLY1305 SEND DATA ////////////////////////

//send the 128bit of data
WRITE_POLY1305(BLOCK_POLY_0,d0);
WRITE_POLY1305(BLOCK_POLY_1,d1);
WRITE_POLY1305(BLOCK_POLY_2,d2);
WRITE_POLY1305(BLOCK_POLY_3,d3);

//send the block len information
WRITE_POLY1305(BLOCKLEN_POLY,block_len);

//send a NEXT value to CTRL
WRITE_POLY1305(CTRL_POLY,NEXT_POLY);

//wait for ready status
int status = 0x01;
while (status == READY_POLY)
    {
        status = READ_POLY1305(STATUS_POLY);
        //printfNexys("NEXT:\n");
        //printfNexys("%X:\n",s);
    }


////////////////////////////////////////////////////////
////////////// POLY1305 SEND DATA ////////////////////////

//send the 128bit of data
```

```
WRITE_POLY1305(BLOCK_POLY_0,d4);
WRITE_POLY1305(BLOCK_POLY_1,d5);
WRITE_POLY1305(BLOCK_POLY_2,d6);
WRITE_POLY1305(BLOCK_POLY_3,d7);

//send the block len information
WRITE_POLY1305(BLOCKLEN_POLY,block_len);

//send a NEXT value to CTRL
WRITE_POLY1305(CTRL_POLY,NEXT_POLY);

//wait for ready status
 status = 0x01;
while (status == READY_POLY)
    {
        status = READ_POLY1305(STATUS_POLY);
        //printfNexys("NEXT 1:\n");
        //printfNexys("%X:\n",s);
    }


////////////////////////////////////////////////////////
////////////// POLY1305 SEND DATA ////////////////////////

//send the 128bit of data
WRITE_POLY1305(BLOCK_POLY_0,d8);
WRITE_POLY1305(BLOCK_POLY_1,d9);
WRITE_POLY1305(BLOCK_POLY_2,d9);
WRITE_POLY1305(BLOCK_POLY_3,d9);

//send the block len information
WRITE_POLY1305(BLOCKLEN_POLY,0x2);

//send a NEXT value to CTRL
WRITE_POLY1305(CTRL_POLY,NEXT_POLY);

//wait for ready status
 status = 0x01;
while (status == READY_POLY)
    {
        status = READ_POLY1305(STATUS_POLY);
        //printfNexys("NEXT 2:\n");
        //printfNexys("%X:\n",s);
    }

////////////////////////////////////////////////////////
///////// POLY1305 AUTHENTICATE THE DATA /////////////////

//send a FINISH value to CTRL
```

```
WRITE_POLY1305(CTRL_POLY,FINISH_POLY);

//wait for ready status
 status = 0x01;
while (status == READY_POLY)
    {
        status = READ_POLY1305(STATUS_POLY);
        //printfNexys("FINISH:\n");
        //printfNexys("%X:\n",s);
    }

//read the generated MAC
mac_p0 = READ_POLY1305(MAC_POLY_0);
mac_p1 = READ_POLY1305(MAC_POLY_1);
mac_p2 = READ_POLY1305(MAC_POLY_2);
mac_p3 = READ_POLY1305(MAC_POLY_3);


//////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////
//////////////////////////////////////////////////////////


    while (1) {
        switches_value = READ_GPIO(GPIO_SWs); // read value on switches
        switches_value = switches_value >> 16; // shift into lower 16
bits

        WRITE_GPIO(GPIO_LEDs, switches_value); // display switch value on
LEDs

        status = READ_POLY1305(STATUS_POLY);
       // printfNexys("status: %X", status);


        //read the generated MAC
        /*mac_p0 = READ_POLY1305(MAC_POLY_0);
        mac_p1 = READ_POLY1305(MAC_POLY_1);
        mac_p2 = READ_POLY1305(MAC_POLY_2);
        mac_p3 = READ_POLY1305(MAC_POLY_3);*/

        // read the sended Key
        k0 = READ_POLY1305(KEY_POLY_0);
        k1 = READ_POLY1305(KEY_POLY_1);
        k2 = READ_POLY1305(KEY_POLY_2);
        k3 = READ_POLY1305(KEY_POLY_3);
        k4 = READ_POLY1305(KEY_POLY_4);
        k5 = READ_POLY1305(KEY_POLY_5);
```

```
        k6 = READ_POLY1305(KEY_POLY_6);
        k7 = READ_POLY1305(KEY_POLY_7);
        printfNexys("\n");

        printfNexys("test POLY1305");
        printfNexys("DATA SENT:\n");
        printfNexys("%X %X  %X  %X : ",d0,d1,d2,d3);
        printfNexys("%X %X  %X  %X : ",d4,d5,d6);
        printfNexys("%X %X  %X  %X : ",d7,d8,d9);


        printfNexys("\n");
        printfNexys("key:");
        printfNexys("%X %X  %X  %X : ",k0,k1,k2,k3);
        printfNexys("%X %X  %X  %X : ",k4,k5,k6);
        printfNexys("%X : ",k7);
        printfNexys("\n");

        /*printfNexys("block of data:\n");
        printfNexys("%X : ",block_p0);
        printfNexys("%X : ",block_p1);
        printfNexys("%X : ",block_p2);
        printfNexys("%X : ",block_p3);*/

        /*printfNexys("key_core_out:\n");
        printfNexys("%X : ",k0);
        printfNexys("%X : ",k1);
        printfNexys("%X : ",k2);
        printfNexys("%X : ",k3);
        printfNexys("%X : ",k4);
        printfNexys("%X : ",k5);
        printfNexys("%X : ",k6);
        printfNexys("%X : ",k7);*/

        printfNexys("Generated MAC:");
        printfNexys("%X %X %X %X : ",mac_p0,mac_p1,mac_p2,mac_p3);

        for (i=0; i < DELAY; i++) ;
    }

    return(0);
}
```

Appendix E

```
// memory-mapped I/O addresses
#define NAME0           0x80001600
#define NAME1           0x80001604
#define VERSION         0x80001608

#define CTRL            0x8000160c
#define STATUS          0x80001610
#define KEYLEN          0x80001614
#define ROUNDS          0x80001618
#define COUNTER         0x8000161c

#define KEY0            0x800016c0
#define KEY1            0x800016c4
#define KEY2            0x800016c8
#define KEY3            0x800016cc
#define KEY4            0x800016d0
#define KEY5            0x800016d4
#define KEY6            0x800016d8
#define KEY7            0x800016dc

#define IV0             0x80001624
#define IV1             0x80001628
#define IV2             0x8000162c

#define DATA_IN0        0x80001630
#define DATA_IN1        0x80001634
#define DATA_IN2        0x80001638
#define DATA_IN3        0x8000163c
#define DATA_IN4        0x80001640
#define DATA_IN5        0x80001644
#define DATA_IN6        0x80001648
#define DATA_IN7        0x8000164c
#define DATA_IN8        0x80001650
#define DATA_IN9        0x80001654
#define DATA_IN10       0x80001658
#define DATA_IN11       0x8000165c
#define DATA_IN12       0x80001660
#define DATA_IN13       0x80001664
#define DATA_IN14       0x80001668
#define DATA_IN15       0x8000166c

#define DATA_OUT0       0x80001680
#define DATA_OUT1       0x80001684
#define DATA_OUT2       0x80001688
#define DATA_OUT3       0x8000168c
```

```
#define DATA_OUT4       0x80001690
#define DATA_OUT5       0x80001694
#define DATA_OUT6       0x80001698
#define DATA_OUT7       0x8000169c
#define DATA_OUT8       0x800016a0
#define DATA_OUT9       0x800016a4
#define DATA_OUT10      0x800016a8
#define DATA_OUT11      0x800016ac
#define DATA_OUT12      0x800016b0
#define DATA_OUT13      0x800016b4
#define DATA_OUT14      0x800016b8
#define DATA_OUT15      0x800016bc


#define DATA_CHIFRA0       0x80001700
#define DATA_CHIFRA1       0x80001704
#define DATA_CHIFRA2       0x80001708
#define DATA_CHIFRA3       0x8000170c
#define DATA_CHIFRA4       0x80001710
#define DATA_CHIFRA5       0x80001714
#define DATA_CHIFRA6       0x80001718
#define DATA_CHIFRA7       0x8000171c
#define DATA_CHIFRA8       0x80001720
#define DATA_CHIFRA9       0x80001724
#define DATA_CHIFRA10      0x80001728
#define DATA_CHIFRA11      0x8000172c
#define DATA_CHIFRA12      0x80001730
#define DATA_CHIFRA13      0x80001734
#define DATA_CHIFRA14      0x80001738
#define DATA_CHIFRA15      0x8000173c

//poly1305
#define NAME_POLY_0     0x80001300
#define NAME_POLY_1     0x80001304
#define VERSION_POLY    0x80001308

#define CTRL_POLY       0x80001310
#define STATUS_POLY     0x80001314

#define BLOCKLEN_POLY   0x80001318

#define KEY_POLY_0      0x80001320
#define KEY_POLY_1      0x80001324
#define KEY_POLY_2      0x80001328
#define KEY_POLY_3      0x8000132c
#define KEY_POLY_4      0x80001330
#define KEY_POLY_5      0x80001334
#define KEY_POLY_6      0x80001338
#define KEY_POLY_7      0x8000133c
```

```
#define BLOCK_POLY_0    0x80001340
#define BLOCK_POLY_1    0x80001344
#define BLOCK_POLY_2    0x80001348
#define BLOCK_POLY_3    0x8000134c

#define MAC_POLY_0      0x80001350
#define MAC_POLY_1      0x80001354
#define MAC_POLY_2      0x80001358
#define MAC_POLY_3      0x8000135c

////////////////////////////////////////////////////////////////////
#define TEMP_I2C        0x80001500

#define READ_TEMP(dir) (*(volatile unsigned *)dir)


////////////////////////////////////////////////////////////////////

#define READ_GPIO(dir) (*(volatile unsigned *)dir)
#define WRITE_GPIO(dir, value) { (*(volatile unsigned *)dir) = (value); }

#define READ_CHACHA(dir) (*(volatile unsigned *)dir)
#define WRITE_CHACHA(dir, value) { (*(volatile unsigned *)dir) = (value);
}

#define READ_POLY1305(dir) (*(volatile unsigned *)dir)
#define WRITE_POLY1305(dir, value) { (*(volatile unsigned *)dir) =
(value); }


#if defined(D_NEXYS_A7)
 #include <bsp_printf.h>
 #include <bsp_mem_map.h>
 #include <bsp_version.h>
#else
 PRE_COMPILED_MSG("no platform was defined")
#endif

#include <psp_api.h>

#define DELAY 10000000
#define DLEAY_SHORT 50

////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////////
int main ( void )
{
```

```
 // Initialize UART
    uartInit();

////////////////////////////////////////////////////////////////////
 //variaveis do chacha
while(1){
//keylen to 256bit
    int key_len=0x01;
    WRITE_CHACHA(KEYLEN,key_len);

// number of rounds to 20
    int  numb_rounds=0x14;
    WRITE_CHACHA(ROUNDS,numb_rounds);

// set up the counter
    int count_init_val = 0x00;
    WRITE_CHACHA(COUNTER, count_init_val);

// set the key
    int K_temp0,K_temp1,K_temp2,K_temp3,K_temp4,K_temp5,K_temp6,K_temp7;

    int k0=0x00010203,
        k1=0x04050607,
        k2=0x08090a0b,
        k3=0x0c0d0e0f,
        k4=0x10111213,
        k5=0x14151617,
        k6=0x18191a1b,
        k7=0x1c1d1e1f;
    WRITE_CHACHA(KEY0,k0);
    WRITE_CHACHA(KEY1,k1);
    WRITE_CHACHA(KEY2,k2);
    WRITE_CHACHA(KEY3,k3);
    WRITE_CHACHA(KEY4,k4);
    WRITE_CHACHA(KEY5,k5);
    WRITE_CHACHA(KEY6,k6);
    WRITE_CHACHA(KEY7,k7);

// set the nonce
    int N_temp0,N_temp1,N_temp2;
    //int N0=0X00000009,
    //    N1=0X0000004a;

    int N0=0X00000000,
        N1=0X0000004a,
        N2=0x00000000;

    WRITE_CHACHA(IV0,N0);
    WRITE_CHACHA(IV1,N1);
```

```
    WRITE_CHACHA(IV2,N2);


// data to chacha
    int
d_temp0,d_temp1,d_temp2,d_temp3,d_temp4,d_temp5,d_temp6,d_temp7,d_temp8,d
_temp9,d_temp10,d_temp11,d_temp12,d_temp13,d_temp14,d_temp15;
    int
d_out_temp0,d_out_temp1,d_out_temp2,d_out_temp3,d_out_temp4,d_out_temp5,d
_out_temp6,d_out_temp7,d_out_temp8,d_out_temp9,d_out_temp10,
        d_out_temp11,d_out_temp12,d_out_temp13,d_out_temp14,d_out_temp15;

    int d_chifra0,
        d_chifra1,
        d_chifra2,
        d_chifra3,
        d_chifra4,
        d_chifra5,
        d_chifra6,
        d_chifra7,
        d_chifra8,
        d_chifra9,
        d_chifra10,
        d_chifra11,
        d_chifra12,
        d_chifra13,
        d_chifra14,
        d_chifra15;

    int read_temp = READ_TEMP(TEMP_I2C);

    int data0= read_temp,
        data1= 0x00,
        data2= 0x00,
        data3= 0x00,
        data4= 0x00,
        data5= 0x00,
        data6= 0x00,
        data7= 0x00,
        data8= 0x00,
        data9= 0x00,
        data10=0x00,
        data11=0x00,
        data12=0x00,
        data13=0x00,
        data14=0x00,
        data15=0x00;
```

```
    WRITE_CHACHA(DATA_IN0, data0 );
    WRITE_CHACHA(DATA_IN1, data1 );
    WRITE_CHACHA(DATA_IN2, data2 );
    WRITE_CHACHA(DATA_IN3, data3 );
    WRITE_CHACHA(DATA_IN4, data4 );
    WRITE_CHACHA(DATA_IN5, data5 );
    WRITE_CHACHA(DATA_IN6, data6 );
    WRITE_CHACHA(DATA_IN7, data7 );
    WRITE_CHACHA(DATA_IN8, data8 );
    WRITE_CHACHA(DATA_IN9, data9 );
    WRITE_CHACHA(DATA_IN10,data10);
    WRITE_CHACHA(DATA_IN11,data11);
    WRITE_CHACHA(DATA_IN12,data12);
    WRITE_CHACHA(DATA_IN13,data13);
    WRITE_CHACHA(DATA_IN14,data14);
    WRITE_CHACHA(DATA_IN15,data15);


    //control variables
    int init=0x01,
        next=0x02;


    WRITE_CHACHA(CTRL, 0x01 );
    int wait=1;
    int estado=0x00;


////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////////////


        printfNexys("CHACHA20");

        N_temp0=READ_CHACHA(IV0);
        N_temp1=READ_CHACHA(IV1);
        N_temp2=READ_CHACHA(IV2);
        printfNexys("Nonce: %X %X %X: ",N_temp0,N_temp1,N_temp2);



        estado=0x0;
        wait=1;

        while (wait==1) {
        estado=READ_CHACHA(STATUS);
        if (estado!=0x00){
            wait=0;
        }
```

```
        }

        printfNexys("\n");

        printfNexys("key:");
        K_temp0= READ_CHACHA(KEY0 );
        K_temp1= READ_CHACHA(KEY1 );
        K_temp2= READ_CHACHA(KEY2 );
        K_temp3= READ_CHACHA(KEY3 );
        K_temp4= READ_CHACHA(KEY4 );
        K_temp5= READ_CHACHA(KEY5 );
        K_temp6= READ_CHACHA(KEY6 );
        K_temp7= READ_CHACHA(KEY7 );
        printfNexys("%X  %X  %X  %X : ",K_temp0 ,K_temp1,K_temp2
,K_temp3);
        printfNexys("%X  %X  %X  %X : ",K_temp4,K_temp5,K_temp6,K_temp7);

         printfNexys("\n");


        d_chifra0= READ_CHACHA(DATA_CHIFRA0 );
        d_chifra1= READ_CHACHA(DATA_CHIFRA1 );
        d_chifra2= READ_CHACHA(DATA_CHIFRA2 );
        d_chifra3= READ_CHACHA(DATA_CHIFRA3 );
        d_chifra4= READ_CHACHA(DATA_CHIFRA4 );
        d_chifra5= READ_CHACHA(DATA_CHIFRA5 );
        d_chifra6= READ_CHACHA(DATA_CHIFRA6 );
        d_chifra7= READ_CHACHA(DATA_CHIFRA7 );
        d_chifra8= READ_CHACHA(DATA_CHIFRA8 );
        d_chifra9= READ_CHACHA(DATA_CHIFRA9 );
        d_chifra10=READ_CHACHA(DATA_CHIFRA10);
        d_chifra11=READ_CHACHA(DATA_CHIFRA11);
        d_chifra12=READ_CHACHA(DATA_CHIFRA12);
        d_chifra13=READ_CHACHA(DATA_CHIFRA13);
        d_chifra14=READ_CHACHA(DATA_CHIFRA14);
        d_chifra15=READ_CHACHA(DATA_CHIFRA15);
        printfNexys("chacha: generate key to poly1305:");
        printfNexys("%X  %X  %X  %X : ",d_chifra0,d_chifra1
,d_chifra2,d_chifra3 );
        printfNexys("%X  %X  %X  %X :
",d_chifra4,d_chifra5,d_chifra6,d_chifra7);
        printfNexys("%X  %X  %X  %X :
",d_chifra8,d_chifra9,d_chifra10,d_chifra11);
        printfNexys("%X  %X  %X  %X :
",d_chifra12,d_chifra13,d_chifra14,d_chifra15);

////////////////////////////////////////////////////////////////////////
//////////////
```

```
////////////////////////////////////////////////////////////////////////
//////////////

    WRITE_CHACHA(CTRL, next );
     wait=1;
     estado=0x00;

        while (wait==1) {
        estado=READ_CHACHA(STATUS);
        if (estado!=0x00){
            wait=0;
        }
        }

////////////////////////////////////////////////////////////////////////
///////////////
////////////////////////////////////////////////////////////////////////
///////////////

        d_out_temp0= READ_CHACHA(DATA_OUT0 );
        d_out_temp1= READ_CHACHA(DATA_OUT1 );
        d_out_temp2= READ_CHACHA(DATA_OUT2 );
        d_out_temp3= READ_CHACHA(DATA_OUT3 );
        d_out_temp4= READ_CHACHA(DATA_OUT4 );
        d_out_temp5= READ_CHACHA(DATA_OUT5 );
        d_out_temp6= READ_CHACHA(DATA_OUT6 );
        d_out_temp7= READ_CHACHA(DATA_OUT7 );
        d_out_temp8= READ_CHACHA(DATA_OUT8 );
        d_out_temp9= READ_CHACHA(DATA_OUT9 );
        d_out_temp10=READ_CHACHA(DATA_OUT10);
        d_out_temp11=READ_CHACHA(DATA_OUT11);
        d_out_temp12=READ_CHACHA(DATA_OUT12);
        d_out_temp13=READ_CHACHA(DATA_OUT13);
        d_out_temp14=READ_CHACHA(DATA_OUT14);
        d_out_temp15=READ_CHACHA(DATA_OUT15);
        printfNexys("\n");
        printfNexys("Data out");
        printfNexys("%X  %X  %X  %X : ",d_out_temp0
,d_out_temp1,d_out_temp2 ,d_out_temp3 );
        printfNexys("%X  %X  %X  %X : ",d_out_temp4
,d_out_temp5,d_out_temp6 ,d_out_temp7 );
        printfNexys("%X  %X  %X  %X : ",d_out_temp8
,d_out_temp9,d_out_temp10 ,d_out_temp11 );
        printfNexys("%X  %X  %X  %X : ",d_out_temp12
,d_out_temp13,d_out_temp14 ,d_out_temp15 );
         printfNexys("\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
\\\\\\\\\\\\\\\");
          printfNexys("\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
\\\\\\\\\\\\\\\\");
```

```
////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////////////
// INIT POLY1305

//POLY1305 key
int key_p0 = d_chifra0,
    key_p1 = d_chifra1,
    key_p2 = d_chifra2,
    key_p3 = d_chifra3,
    key_p4 = d_chifra4,
    key_p5 = d_chifra5,
    key_p6 = d_chifra6,
    key_p7 = d_chifra7;


//POLY1305 message block

int block_len = 0x10;

int block_p0 = 0x00,
    block_p1 = 0x00,
    block_p2 = 0x00,
    block_p3 = 0x00;
int d0 =d_out_temp0,
    d1 =d_out_temp1,
    d2 =d_out_temp2,
    d3 =d_out_temp3,

    d4 =d_out_temp4,
    d5 =d_out_temp5,
    d6 =d_out_temp6,
    d7 =d_out_temp7,

    d8 =d_out_temp8,
    d9 =d_out_temp9,
    d10 =d_out_temp10,
    d11 =d_out_temp11,

    d12 =d_out_temp12,
    d13 =d_out_temp13,
    d14 =d_out_temp14,
    d15 =d_out_temp15;
//POLY1305 MAC
int mac_p0 =0x00,
    mac_p1 = 0x00,
    mac_p2 =0x00,
    mac_p3 =0x00;
```

```
//POLY1305 CONTROL SIGNALS

int INIT_POLY = 0x01,
    NEXT_POLY = 0x02,
    FINISH_POLY = 0x04;

//POLY1305 STATUS COMPARISON SIGNALS

int READY_POLY = 0x01;

////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////
////////////////////////////////////////////////////////
////////////////////////////////////////////////////////
////////////////// POLY1305 START ////////////////////////
//while (1) {
//write the key
WRITE_POLY1305(KEY_POLY_0,key_p0);
WRITE_POLY1305(KEY_POLY_1,key_p1);
WRITE_POLY1305(KEY_POLY_2,key_p2);
WRITE_POLY1305(KEY_POLY_3,key_p3);
WRITE_POLY1305(KEY_POLY_4,key_p4);
WRITE_POLY1305(KEY_POLY_5,key_p5);
WRITE_POLY1305(KEY_POLY_6,key_p6);
WRITE_POLY1305(KEY_POLY_7,key_p7);

//send block clear
WRITE_POLY1305(BLOCK_POLY_0,0x00);
WRITE_POLY1305(BLOCK_POLY_1,0x00);
WRITE_POLY1305(BLOCK_POLY_2,0x00);
WRITE_POLY1305(BLOCK_POLY_3,0x00);

//send an INIT value to CTRL
WRITE_POLY1305(CTRL_POLY,INIT_POLY);


//wait for ready status
int s = 0x01;
while (s == READY_POLY)
    {
        s = READ_POLY1305(STATUS_POLY);
        //printfNexys("START:\n");
        //printfNexys("%X:\n",s);
    }

////////////////////////////////////////////////////////
////////////// POLY1305 SEND DATA ////////////////////////
```

```
//send the 128bit of data
WRITE_POLY1305(BLOCK_POLY_0,d0);
WRITE_POLY1305(BLOCK_POLY_1,d1);
WRITE_POLY1305(BLOCK_POLY_2,d2);
WRITE_POLY1305(BLOCK_POLY_3,d3);

//send the block len information
WRITE_POLY1305(BLOCKLEN_POLY,block_len);

//send a NEXT value to CTRL
WRITE_POLY1305(CTRL_POLY,NEXT_POLY);

//wait for ready status
int status = 0x01;
while (status == READY_POLY)
    {
        status = READ_POLY1305(STATUS_POLY);
        //printfNexys("NEXT:\n");
        //printfNexys("%X:\n",s);
    }


////////////////////////////////////////////////////////
////////////// POLY1305 SEND DATA ////////////////////////

//send the 128bit of data
WRITE_POLY1305(BLOCK_POLY_0,d4);
WRITE_POLY1305(BLOCK_POLY_1,d5);
WRITE_POLY1305(BLOCK_POLY_2,d6);
WRITE_POLY1305(BLOCK_POLY_3,d7);

//send the block len information
WRITE_POLY1305(BLOCKLEN_POLY,block_len);

//send a NEXT value to CTRL
WRITE_POLY1305(CTRL_POLY,NEXT_POLY);

//wait for ready status
 status = 0x01;
while (status == READY_POLY)
    {
        status = READ_POLY1305(STATUS_POLY);
        //printfNexys("NEXT 1:\n");
        //printfNexys("%X:\n",s);
    }


////////////////////////////////////////////////////////
```

```
////////////// POLY1305 SEND DATA ////////////////////////

//send the 128bit of data
WRITE_POLY1305(BLOCK_POLY_0,d8);
WRITE_POLY1305(BLOCK_POLY_1,d9);
WRITE_POLY1305(BLOCK_POLY_2,d10);
WRITE_POLY1305(BLOCK_POLY_3,d11);

//send the block len information
WRITE_POLY1305(BLOCKLEN_POLY,block_len);

//send a NEXT value to CTRL
WRITE_POLY1305(CTRL_POLY,NEXT_POLY);

//wait for ready status
 status = 0x01;
while (status == READY_POLY)
    {
        status = READ_POLY1305(STATUS_POLY);
        //printfNexys("NEXT:\n");
        //printfNexys("%X:\n",s);
    }


////////////////////////////////////////////////////////
////////////// POLY1305 SEND DATA ////////////////////////

//send the 128bit of data
WRITE_POLY1305(BLOCK_POLY_0,d12);
WRITE_POLY1305(BLOCK_POLY_1,d13);
WRITE_POLY1305(BLOCK_POLY_2,d14);
WRITE_POLY1305(BLOCK_POLY_3,d15);

//send the block len information
WRITE_POLY1305(BLOCKLEN_POLY,block_len);

//send a NEXT value to CTRL
WRITE_POLY1305(CTRL_POLY,NEXT_POLY);

//wait for ready status
 status = 0x01;
while (status == READY_POLY)
    {
        status = READ_POLY1305(STATUS_POLY);
        //printfNexys("NEXT 1:\n");
        //printfNexys("%X:\n",s);
    }
```

```
////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////
///////// POLY1305 AUTHENTICATE THE DATA //////////////////

//send a FINISH value to CTRL
WRITE_POLY1305(CTRL_POLY,FINISH_POLY);

//wait for ready status
 status = 0x01;
while (status == READY_POLY)
    {
        status = READ_POLY1305(STATUS_POLY);
        //printfNexys("FINISH:\n");
        //printfNexys("%X:\n",s);
    }

//read the generated MAC
mac_p0 = READ_POLY1305(MAC_POLY_0);
mac_p1 = READ_POLY1305(MAC_POLY_1);
mac_p2 = READ_POLY1305(MAC_POLY_2);
mac_p3 = READ_POLY1305(MAC_POLY_3);


////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////
////////////////////////////////////////////////////////////
                // read the sended Key
        k0 = READ_POLY1305(KEY_POLY_0);
        k1 = READ_POLY1305(KEY_POLY_1);
        k2 = READ_POLY1305(KEY_POLY_2);
        k3 = READ_POLY1305(KEY_POLY_3);
        k4 = READ_POLY1305(KEY_POLY_4);
        k5 = READ_POLY1305(KEY_POLY_5);
        k6 = READ_POLY1305(KEY_POLY_6);
        k7 = READ_POLY1305(KEY_POLY_7);

        printfNexys("\n");

        printfNexys("POLY1305\n");
        printfNexys("DATA SENT:");
        printfNexys("%X %X  %X  %X : ",d0,d1,d2,d3);
        printfNexys("%X %X  %X  %X : ",d4,d5,d6, d7);
        printfNexys("%X %X  %X  %X : ",d8,d9,d10,d11);
        printfNexys("%X %X  %X  %X : ",d12,d13,d14,d15);


        printfNexys("\n");
        printfNexys("key:");
```

```
        printfNexys("%X %X  %X  %X : ",k0,k1,k2,k3);
        printfNexys("%X %X  %X  %X : ",k4,k5,k6,k7);
        printfNexys("\n");

        printfNexys("Generated MAC:");
        printfNexys("%X %X %X %X : ",mac_p0,mac_p1,mac_p2,mac_p3);

        printfNexys("\n");
         printfNexys("\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
\\\\\\\\\\\\\\\\");


        printfNexys("encrypted data\n");
        printfNexys("%X  %X  %X  %X : ",d_out_temp0
,d_out_temp1,d_out_temp2 ,d_out_temp3 );
        printfNexys("%X  %X  %X  %X : ",d_out_temp4
,d_out_temp5,d_out_temp6 ,d_out_temp7 );
        printfNexys("%X  %X  %X  %X : ",d_out_temp8
,d_out_temp9,d_out_temp10 ,d_out_temp11 );
        printfNexys("%X  %X  %X  %X : ",d_out_temp12
,d_out_temp13,d_out_temp14 ,d_out_temp15 );
        printfNexys("\n");

        printfNexys("Generated MAC:");
        printfNexys("%X %X %X %X : ",mac_p0,mac_p1,mac_p2,mac_p3);
}
        return(0);
        }
```