Integration of an ADC on a RISC-V Softcore Implemented in an FPGA

Renato José dos Santos Farinha

Dissertação para obtenção do Grau de Mestre em

Engenharia Electrotécnica e de Computadores

(2º ciclo de estudos)

Orientador: Prof. Doutor António Eduardo Vitória do Espírito Santo

Março de 2024

Declaração de Integridade

Eu, Renato José dos Santos Farinha, que abaixo assino, estudante com o número de inscrição m11045 de Engenharia Electrotécnica 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 25 / 03 / 2024

Agradecimentos

Agradeço aos meus pais, Daniel Farinha António e Maria do Carmo Gaspar dos Santos. Pela minha educação a nível pessoal e académico.
Agradeço ao meu irmão, Helder André dos Santos Farinha, pela paciência e ajuda ao longo da vida.
Agradeço à minha namorada, Mónica Susana Silva. Por todo o apoio.
Agradeço ao meu orientador António Espítito Santo. Por me dar a conhecer as FPGAs e acompanhar este trabalho.
Agrandeço a todos os meus amigos que se cruzaram comigo ao longo do meu percurso e acrescentaram valor à minha pessoa individual.

Resumo

Com a evolução da indústria 4.0, a presença dos sensores tornou-se essencial para monitorizar e optimizar muitos dos sistemas actuais. Com o aumento do número de sensores há uma necessidade maior para este se tornarem mais eficientes energeticamente, reduzir o seu tamanho e melhorar capacidade de processamento, desta forma procura-se abordagens diferentes para ir de encontro dos objectivos anteriores. Actualmente os sensores são projectados para um determinado efeito e após a sua produção não é possível adicionar ou alterar as suas características.

Nesta dissertação há uma abordagem diferente ao desenvolvimento de um sensor utilizado uma FPGA, permitindo a sua reconfiguração e expansão para novas funcionalidades, para a execução de código é utilizado o RISC-V um ISA que permite a execução de código C, o RISC-V é uma arquitetura de código aberto que torna possível qualquer pessoa aceder e modificar a mesma, disponibiliza um conjunto de instruções base e algumas extensões dando aos seus utilizadores flexibilidade no desenvolvimento dos seus projectos. Para a medição é utilizado o XADC nativo da FPGA, este tem um conjunto de funcionalidades que permite diversos modos de funcionamento.

Palavras-chave

FPGA;RISC-V;XADC;Open-source

Abstract

With the evolution of Industry 4.0, sensors have become essential to monitor and optimize many of today's systems. With the increase in the number of sensors, they need to become more energy efficient, reduce their size, and improve processing capacity. Thus, different approaches are sought to meet the previous objectives. Currently, sensors are designed for a specific purpose, and after production, it is impossible to add or change their characteristics.

This dissertation adopts a different approach to the designing a sensor using an FPGA, enabling reconfiguration and expansion for new functionalities. For code execution, it utilizes RISC-V, an ISA capable of running C code. RISC-V's open source architecture facilitates accessibility and modification by anyone. It provides a set of base instructions with additional extensions, giving users flexibility in developing their projects. The FPGA's native XADC is used for measurement, with features that allow different operating modes.

Keywords

Fpga;RISC-V;XADC;Open-source

Índice

Chapter 1 1
Introduction 1
1.1 Field Of Work 1
1.2 Importance, relevance, and impact 1
1.3 General and specific objectives of the work 2
1.4 Documentation structure and organization 2
Chapter 2 3
Architecture RISC-V 3
2.1 RISC-V 3
2.2 Wishbone B4 10
2.3 XADC 24
Chapter 3 35
Integration of XADC on RISC-V 35
3.1 Software Tools 35
3.2 Hardware Tools 36
3.3 XADC Verilog Development 37
Chapter 4 51
Experimental Validation 51
4.1 Experimental Setup 51
4.2 Operational Validations 55
4.3 Results Analysis 61
Chapter 5 63
Conclusion and Future Works 63
References 65
Appendix A 67
Appendix B 69
Appendix C 73
Appendix D 77

Lista de Figuras

Figure 1 - Program Counter from [6] 4
Figure 2 – RISC-V Instruction Format from [6] 5
Figure 3 - RVfpga hierarchy from [11] 7
Figure 4 - WD RISC-V Cores from [11] 8
Figure 5 - SweRV EH1 Pipeline from [11] 9
Figure 6 - WishBone Signals from [13] 12
Figure 7 - Point-to-point Interconnection from [12] 13
Figure 8 - Data Flow Interconnection from [12] 13
Figure 9 - Bus Shared Interconnection from [12] 14
Figure 10 - Crossbar Interconnection from [12] 15
Figure 11 - Single Read from [12] 16
Figure 12 - Single Write from [12] 16
Figure 13 - Block Read from [12] 17
Figure 14 - Block Write from [12] 17
Figure 15 - RWM Cycle from [12] 18
Figure 16 - Input/Output Module, Wishbone Interface 19
Figure 17 - Mnemonics, registers, and assigns 19
Figure 18 - Wishbone read and write operation 20
Figure 19 - Terminal commands to generate simulation 21
Figure 20 - Mnemonics and C code 21
Figure 21 - How to generate trace 22
Figure 22 - Run the waveform 22
Figure 23 - Write through Wishbone on register A 22
Figure 24 - Write through Wishbone on register B 23
Figure 25 - Read through Wishbone on register C 23
Figure 26 - Instantiation Diagram from [14] 24
Figure 27 - Status Register Structure from [14] 25
Figure 28 - Unipolar Mode from [14] 26
Figure 29 - Bipolar Mode from [14] 27
Figure 30 - Common mode noise rejection from [14] 28
Figure 31 - XADC Registers from [14] 30
Figure 32 - External Analog Multiplexer from [14] 31
Figure 33 - Continuous Sampling from [14] 32
Figure 34 - Event-Driven Sampling from [14] 33
Figure 35 - DRP Timing from [14] 33
Figure 36 – XADC module, input/output, and instantiation 38
Figure 37 - Testbench input/output and variables 39
Figure 38 - Find XADC wizard 39
Figure 39 - XADC wizard configurations 40
Figure 40 - Add testbench file 41
Figure 41 - Create a testbench file 41
Figure 42 - Add emulation file 42
Figure 43 - Emulation file struture 42
Figure 44 - XADC testbench 43
Figure 45 - Reading after conversion code 43
Figure 46 - Instantiation XADC module 44
Figure 47 - XADC simulation on Vivado 44
Figure 48 - XADC input/output with Wishbone 46
Figure 49 - Used mnemonics 46
Figure 50 - XADC wizard instantiation 47
Figure 51 - Wishbone interface 47

Figure 52 - Module logic to use DRP 49
Figure 53 - Constraints for XADC external signals 51
Figure 54 - Inputs in RVfpgaNEXYS...52
Figure 55 - Instantiation the signals in RVfpgaNEXYS to SweRV 52
Figure 56 - Inputs in SweRV 52
Figure 57 - Instantiating the XADC module on SweRV...53
Figure 58 - Wishbone XADC signals...53
Figure 59 - Add XADC peripheral in Wishbone 54
Figure 60 - Find "wb_intercon.vh" file 54
Figure 61 - "wb_intercon.vh" changes 55
Figure 62 - Open PlatformIO in VSC 56
Figure 63 - Create new project 56
Figure 64 - "platformio.ini" file changes...57
Figure 65 - C code mnemonics and libraries 57
Figure 66 - UART initialization, initialization variables and test READ 58
Figure 67 - C code to write and read on XADC module 58
Figure 68 - Reconfiguration of DRP 59
Figure 69 - Circuit to be read 60
Figure 70 - Upload bitstream button location 60
Figure 71 - Run the program 61
Figure 72 - Serial monitor button location 61
Figure 73 - Results from the project 61
Figure 74 - Multimeter read...62

Lista de Tabelas

Table 1 - RISC-V Extensions 6
Table 2 - XADC signals..25

Lista de Acrónimos

ADC	Analog to Digital Converter
UART	Universal Asynchronous Receiver/Transmitter
FPGA	Field-Programmable Gate Array
XADC	Xilinx Analog to Digital Converter
RISC	Reduced Instruction Set Computer
ISA	Instruction Set Architecture
WD	Western Digital
SoC	System-on-Chip
IP	Intellectual Property
ASIC	Application-Specific Integrated Circuit
VHDL	VHSIC Hardware Description Language
VHSIC	Very High-Speed Integrated Circuits
PCI	Peripheral Component Interconnect
CPCI	Compact Peripheral Component Interconnect
VMEbus	Versa Module Eurocard bus
VSC	Visual Studio Code
MSPS	Megasamples per second
DRP	Dynamic Reconfiguration Port
MSB	Most Significant Bit
LSB	Least Significant Bit
IDE	Integrated Development Environments
RAM	Random Access Memory
RMW	Read Modify Write
PMOD	Peripheral Module Interface

Chapter 1

Introduction

Nowadays, sensors are everywhere, and it is possible to find them in every aspect of our lives, such as wealth, cars, smartphones, etc. Throughout history, human beings have needed to monitor the environment around them. For temperature, thermometers were created; for distance, the tape measure was one of the tools used; for time, the clock has allowed a better knowledge and control of the environment and human beings. These tools have been perfected throughout technological development, improving their precision and usability. With the advent of digital electronics, the data obtained began to be processed, and the possibility of controlling systems without the human touch. The last few decades have seen the dawn of the internet era, and all devices are connected to the cloud [1]. This allows not only control but also real-time access to the values measured by the sensors. As a result, there is a need to find the best way to monitor a system. This dissertation presents a sensor that allows voltage readings [2].

1.1 Field Of Work

The work carried out in this dissertation is in the field of instrumentation and measurement, as it uses an analog-to-digital converter (ADC) [3], which allows an analog signal, for example, the voltage between two points, to be read and converted into a digital signal with a binary representation. Another area of this work is computer architecture, where the entire architecture will be dissected, from the central processing unit, in this case, RISC-V, to the communication between the central processing unit and the modules (Wishbone), responsible for collecting signals and also for communication (Uart) [4], with communication being the last area of knowledge.

1.2 Importance, relevance, and impact

Most conventional systems, like microcontrollers, lack adaptability as their functionalities are fixed upon design. For instance, once a microcontroller is manufactured, its capabilities remain static, limiting its versatility to run various applications. However, FPGAs revolutionize this paradigm by offering reconfigurable hardware. This means users can tailor applications to their needs and incorporate new features without requiring hardware changes [5]. In practical terms, consider a scenario where an 8-bit microcontroller is initially selected for an application. As the system

evolves, requiring the capabilities of a 16-bit microcontroller, conventional hardware would necessitate replacement. Conversely, with FPGAs, transitioning from 8-bit to 16-bit architecture is feasible without altering the hardware. Thus, FPGAs empower users with highly adaptable systems capable of seamlessly accommodating new functionalities.

1.3 General and specific objectives of the work

The general objectives of this work are to create a sensor that will read a voltage using the XADC (analog-to-digital converter incorporated in the FPGA Artix-7 100T csg324), RISC-V will be used to control the XADC using a C program.
The development of this work contributed to the learning of Verilog. This hardware language allows the reconfiguration of the FPGA, an in-depth knowledge of the XADC, exploring its modes of operation and architecture, exploring the operation of the UART, as well as having knowledge of the bus that allows and facilitates communication between the RISC-V and the modules called Wishbone.

1.4 Documentation structure and organization

This dissertation is divided into five chapters that are organized as follows:

- Chapter 2 introduces RISC-V ISA and extensions, explains the RVfpgaNEXYS, presents the Wishbone B4 interface, and introduces the XADC.
- Chapter 3 shows the creation of the XADC module, tools, and hardware used.
- Chapter 4 explains the experimental setup.
- Chapter 5 presents the conclusions drawn and presents the future work.

Chapter 2

Architecture RISC-V

2.1 RISC-V

Risc-V is an open standard instruction set architecture (ISA) based on Reduced Instruction Set Computer (RISC), developed at the University of California, Berkeley, in 2010. The main goal of this ISA is to be simple, modular, and extensible to support a wide range of applications, going from embedded systems to supercomputers. RISC-V was set to be a small base ISA but supports several extensions. The base ISA has an instruction length of 32-bit but also supports 64-bit and 128-bit [6],[7].

2.1.1 Benefits and Challenges of RISC-V

The open-source RISC-V ISA is free and supported by a community of development. It is a straightforward and elegant base ISA used for research and education, and its simplicity is more than enough for low-power embedded devices. While this approach keeps things simple and avoids introducing unnecessary complications, it also makes the instruction set architecture (ISA) modular bt design. This modularity allows for potential expansion by adding new functionalities through extensions. The scalable and adaptable architecture supports different space addresses, word sizes, and variable-length instructions. This way, the user can adapt all architecture to the development [8].
There are three main challenges that RISC-V has to face: compatibility and standardization. Although RISC-V allows the user to add extensions, there is a risk of fragmentation and incompatibility in different implementations to guarantee that interoperability and portability of software and hardware are necessary to create precise specifications and compliance tests. Another challenge facing RISC-V is establishing a robust ecosystem and adequate support structures. When compared to other ISAs, RISC-V is relatively new; there is still a lack of tools and libraries for development and debugging, and more education and courses are also needed for developers to be able to learn. The last main challenge is security and reliability; as it is an open source and customizable architecture, it allows for security flaws, whether intentional to break systems or bad implementations [9].

2.1.2 Base Integer Instruction Set

The RV32I has 32 registers. Each register has 32-bits. The register x0, stores zeros, and the other registers hold values of instructions. An additional register holds the current instruction address (program counter) [6], as shown in Figure 1.

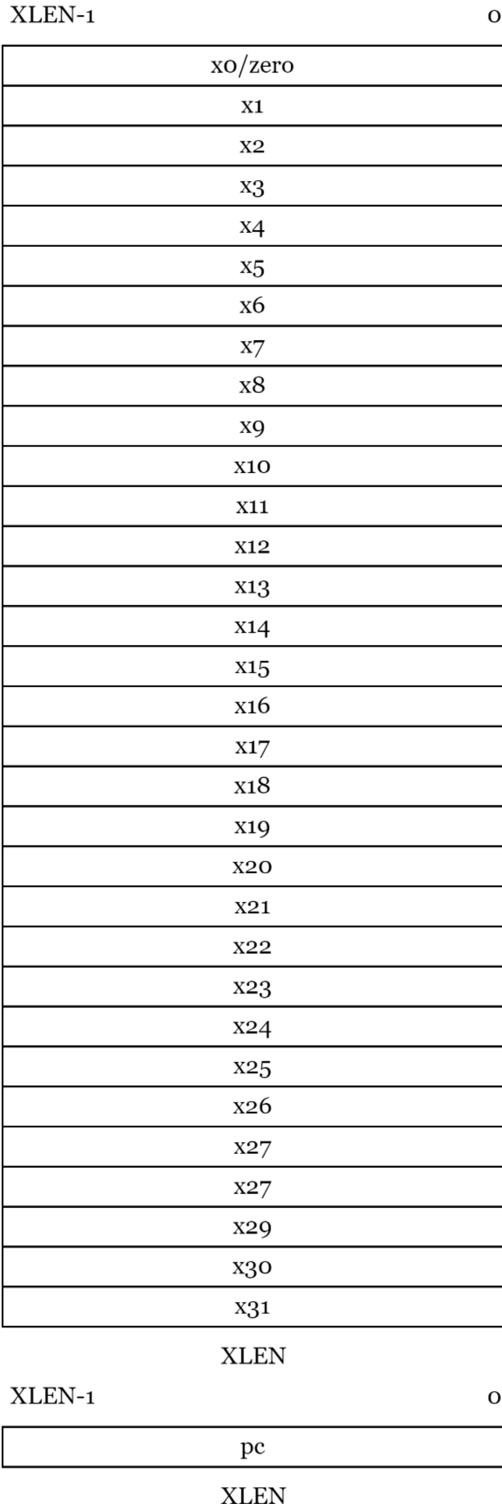

Figure 1 - Program Counter from [6]

In RV32I ISA, there are six instruction formats R, I, S, B, U, and J, these formats have 32-bit lengths, the source registers (rs1, rs2) and destination registers (rd) are in the same position to simplify decoding, on U and J formats there are no source registers, Figure 2 shows the RISC-V instruction formats [6].

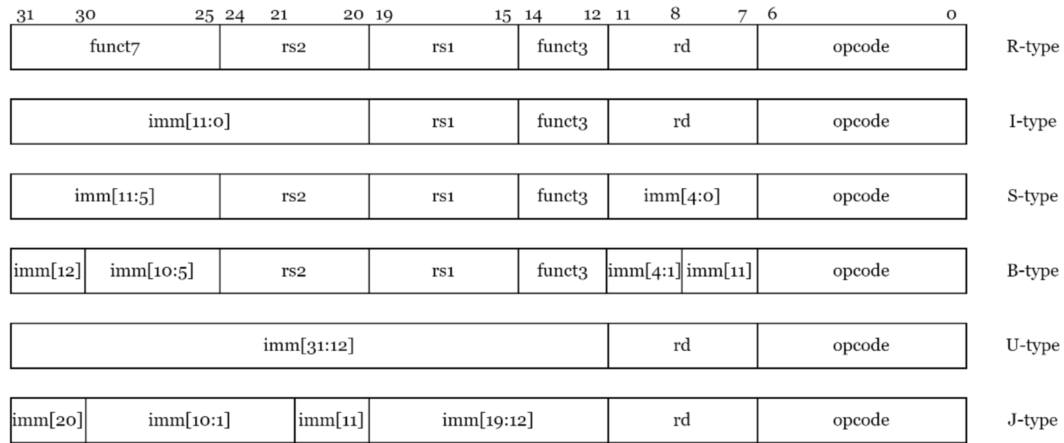

Figure 2 – RISC-V Instruction Format from [6]

In version 2.1 of RV32i, there are 40 instructions. These instructions were designed with hardware reduction in mind for small implementations, yet this ISA is recognized as a compilation target and is supported by several operating systems. These instructions are divided into five groups [6]:

- 21 Integer Computational Instructions:

Arithmetic (addition, subtraction, and bitwise shifts), logical (bitwise Boolean operations), and comparison (arithmetic magnitude comparisons) instructions.

- 8 Load and Store Instructions

Load byte and Load half-word: These two work for signed and unsigned, Load Word Store byte, Store half-word, and Store word.

- 1 Ordering Instruction
- 8 Control Transfer Instructions
- 2 Environment Call and Breakpoints instructions

2.1.3 Extensions

One of the main features of RISC-V is extensions, which allow RISC-V to be adapted to the implementation needs; for example, base ISA does not include floating point, multiplication, or double precision; in the table below are all extensions [10].

Extension	Version	Status
M	2.0	Ratified
A	2.1	Ratified
F	2.2	Ratified
D	2.2	Ratified
Q	2.2	Ratified
C	2.0	Ratified
Counters	2.0	Draft
L	0.0	Draft
B	0.0	Draft
J	0.0	Draft
T	0.0	Draft
P	0.2	Draft
V	0.7	Draft
N	1.1	Draft
Zicsr	2.0	Ratified
Zifencei	2.0	Ratified
Zam	0.1	Draft
Ztso	0.1	Frozen

Table 1 - RISC-V Extensions

The standard extensions are the following:

M: The M extension adds instructions for integer multiplication and division.

A: The A extension adds instructions for atomic memory operations. This is used for synchronization between multiple RISC-V harts running in the same memory.

F: The F extension adds instructions for single-precision floating-point arithmetic.

D, Q: The D and Q extensions add double and quad-precision floating-point arithmetic instructions.

C: The C extension adds compressed instructions to reduce code size and improve performance.

2.1.4 RVfpga

The RVfpga system allows RISC-V to be integrated into a chip/board. In this case, the core used is the SweRV EH1. Around this core, an SoC is implemented to incorporate RAM and add peripherals.

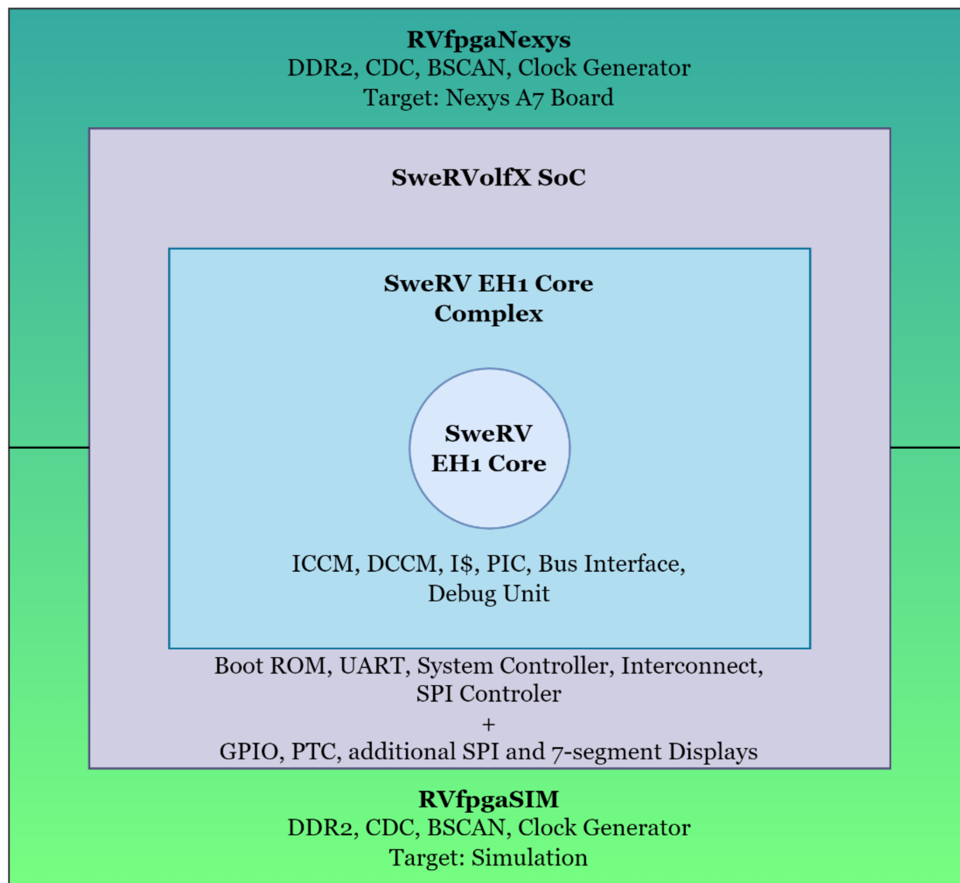

Figure 3 - RVfpga hierarchy from [11]

Figure 3 shows the organization of the system, from SweRV EH1 Core and SweRV EH1 Core Complex to SweRVolfX SoC and RVfpgaNexys and RvfpgaSim, which part has a functionality that will be explained next.

SweRV EH1 Core and SweRV EH1 Core Complex

The SweRV EH1 was developed by the Western Digital (WD) company; this company also has two more versions of cores based on RISC-V, the SweRV EH2 and SweRV EL2. Figure 4 illustrates a comparison between these three cores from WD.

The SweRV EH1 Core is a 32-bit, 2-way superscalar, 9-stage pipeline core. The significant change in the SweRV Eh2 is a dual thread, and the SweRV El2 is a smaller core [11].

Core Name	RISC-V Type	Pipeline Stages	Threads	Size @ TSMC	CoreMarks/MHz
SweRV Core EH1	RV32IMC	9-dual issue	Single	.11mm @ 28nm	4.9
SweRV Core EH2	RV32IMC	9-dual issue	Dual	.067mm @ 16nm	6.3
SweRV Core EL2	RV32IMC	4-single issue	Single	.023mm @ 16nm	3.6

Figure 4 - WD RISC-V Cores from [11]

The SweRV EH1 is built on the standard RISC-V RV32IMC architecture and operates solely in machine mode (M-mode). This mode grants unrestricted access to memory and peripherals, providing comprehensive control over both hardware and software aspects. This core supports RISC-V integer (I), compressed instructions (C), and integer multiplication and division (M) extensions. The SweRV EH1 has a 9-stage pipeline. This is a mechanism that divides the processing of an instruction, allowing different instructions to be executed simultaneously. The first stage is "fetch" in this stage, the processor searches for the next instruction to be executed. The third stage is "align", where the instruction is pushed from fetch. The fourth stage is "decode", where the processor decodes the instruction into smaller operations; in the fifth, sixth, and seventh stages, the processor executes the instruction. The eighth stage is "commit," where two instructions are confirmed, and the last stage is "write-back" in this stage, the processor updates the registers with the operation executed. The first three stages are executed sequentially; the others can be executed in parallel [11]. The pipeline is represented in Figure 5.

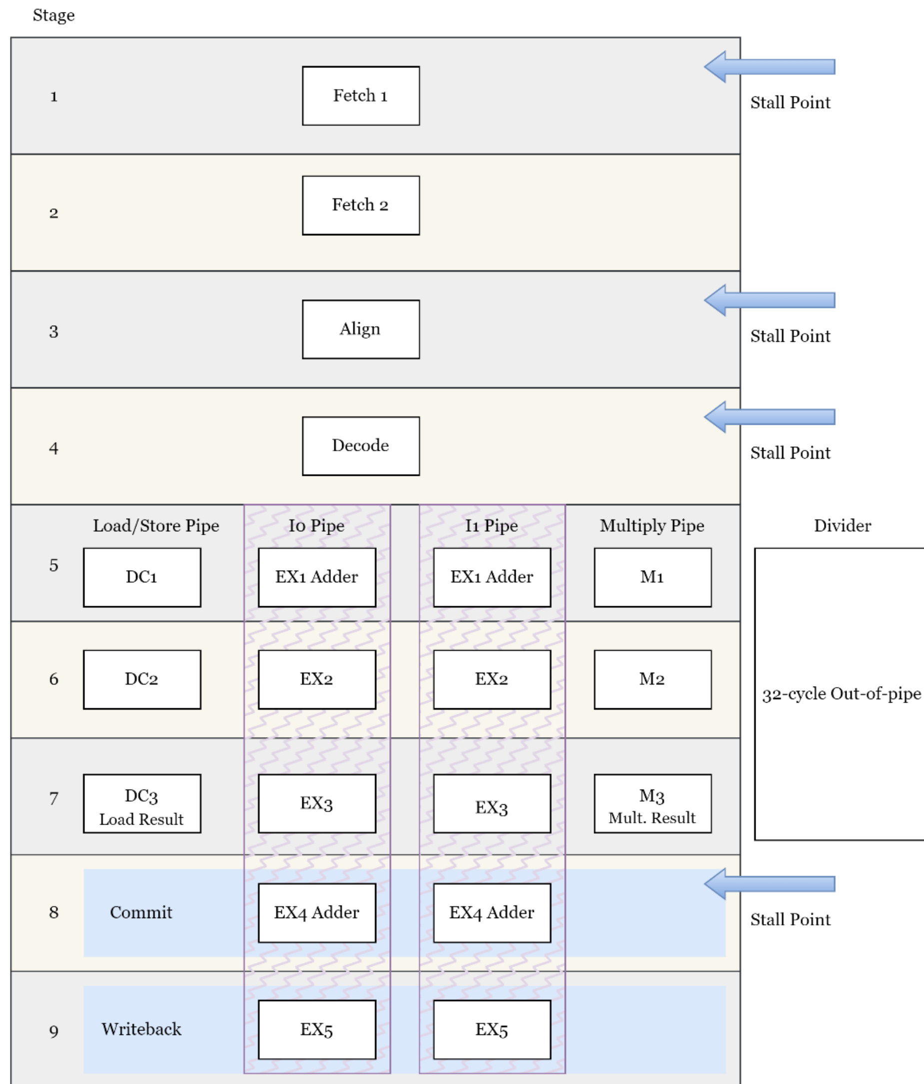

Figure 5 - SweRV EH1 Pipeline from [11]

SweRVolfX SoC

The SweRvolfX SoC allows the addition of peripherals to the SweRV internally, the SweRV uses an AXI interconnection, and in the SoC, there is a bridge between AXI and Wishbone lately the peripherals are coupled in the Wishbone interconnection. Some peripherals included in this SoC are Boot ROM, UART, System Controller, and SPI.

RVfpgaNEXYS and RVfpgaSIM

The RVfpgaNEXYS is the adaptation for the Nexys A7 board from Diligent; this way, every peripheral can be integrated into the SoC. The RVfpgaSIM is a testbench created to see all signals on the SweRVolfX SoC, enabling the system to debug.

2.2 Wishbone B4

The Wishbone System-on-Chip (SoC) Interconnection is a developed method to connect IP cores to form integrated circuits. The Wishbone bus interface is used to help developers reuse and avoid integration problems. This is possible by standardizing this bus interface using a much easier bus.
Key objectives of Wishbone are flexibility in interconnecting IP cores to form a System-on-Chip, creating a robust standard that enforces compatibility between IP cores, being a standard that is easy to use and understand for developers and end-users, and embracing the reusability, creating a portable interface that supports different semiconductor technology, for example, Wishbone interconnections support FPGA and ASIC devices, the interface is independent of logic signal levels, be a flexible interconnection scheme that is independent of the IP cores delivery method (soft core, firm core, hard core), it is also impartial of hardware description language used (VHDL, Verilog or another language) [12].

2.2.1 Wishbone Basics

Wishbone uses a MASTER/SLAVE architecture. In this type of architecture, the MASTER interface begins the data transfer to the SLAVE interface. This communication is enabled by an interconnection interface called INTERCON. The INTERCON contains all the circuits needed to allow the communication between MASTER/SLAVE; this INTERCON is variable because it's possible to change the number of SLAVES and change the type of MASTER and SLAVE communication, for example, point-to-point or data flow. Further, every communication type will be explained. The INTERCON is very different from other traditional buses like PCI, CPCI, VMEbus, and ISA bus; these buses are hardwired; therefore, it's impossible to change after production, which limits how microcomputers communicate with each other. This way, Wishbone is a game changer, allowing the system interconnection paths to be adjusted with hardware description language.
The Wishbone interconnection is a synchronous circuit designed to logically operate over a nearly infinite frequency range. Still, every circuit has physical characteristics that limit the maximum frequency, so theoretically, Wishbone can operate at any frequency [12].
The Wishbone is developed in VHDL, so it is just a file that allows people to share their applications and develop on their own.

2.2.2 Wishbone Interface Specification

This topic will describe the signaling between the MASTER interface, SLAVE interface, and SYSCON module; this is important to create the documentation for the IP core.

Wishbone Required Documentation for IP cores

Every IP core compatible with Wishbone has to include documentation to help the end user understand the operation of the core, and how to connect with other cores. Including this documentation in the technical reference manual or the source code of the IP core is possible. The information included in the Wishbone Datasheet must be the following [12]:

1.The revision level of Wishbone specification to which it was designed

2.The type of interface if it is a MASTER or a SLAVE

3.If a signal name is different than that defined in this specification, then must be cross-referenced to the corresponding signal name which is used in this specification.

4.If the MASTER or SLAVE have the optional signal error or retry, they must describe how they respond to the signal.

5.All interfaces that support tag signals have to describe the name, for example, TAG TYPE: TGA_O() indicates an address tag.

6.The Wishbone Datasheet must indicate the port size, port granularity, and the maximum operand size, which are available in the next lengths: 8-bit, 16-bit, 32-bit or 64-bit. If is unknown the maximum operand size, then the maximum operand size should be equal to the granularity.

7.The Wishbone Datasheet must indicate the data transfer ordering and should be indicated as BIG ENDIAN or LITTLE ENDIAN.

8.The Wishbone Datasheet must indicate if there are any constraints on the clock signal and what constraints and what are these constraints.

Wishbone Interface Signal and Signal Description

Wishbone MASTER and SLAVE interfaces have to follow the following requirements to be flexible and reusable [12]:

1. The signals allow MASTER and SLAVE interfaces to support different interconnections.
2. The signals allow three basic types of bus cycles.
3. The handshaking mechanism can adjust the data transfer rate during a bus cycle, so Wishbone cycles should run at the speed of the slowest interface.

4. Using the handshake mechanism, the SLAVE can accept or reject a data transfer. When rejected, it's possible to send back to MASTER an ERROR or a RETRY bus cycle, the error and retry signal are optional, but acknowledgment is mandatory.
5. All signals on MASTER and SLAVE are unidirectional because FPGA and ASIC devices don't support bi-directional signals, but is possible to use bi-directional signals in the interconnection logic if the device supports it.
6. The address and data bus can be altered to fit the application.

Signal	Name	Optional	Description
Acknowledged Out	ACK_O	No	Acknowledged signal from the Master to Slave
AddressOut/In	ADR_O/I()	No	Address Array
Clock In	CLK_I	No	System Clock for Wishbone Interface
Error In/Out	ERR_I/O	Yes	Indicates an abnormal cycle termination occurred
Data In/Out	DAT_I/O	No	Data input/output array, used to send/recive data
Write Enable In	WE_I	No	Read or Write signals. If asserted it is Write signal otherwise Read signal
Strobe In	STB_I	No	Indicates that the slave is selected, the slave asserts either ACK_O, ERR_O
Strobe Out	STB_O	No	Handshanking signal
Address tag Out	TGA_O()	Yes	Contains information about the address array
Cycle tag type Out	TGC_O()	Yes	Contains information about the transfer cycle
Write Enable Out	WE_O	No	Shows if the transfer cycle is a Read or Write cycle
Reset	RST_I	No	Reset signal

Figure 6 - WishBone Signals from [13]

2.2.3 Types of Wishbone Interconnection

The Wishbone has four types of interconnections; another one is off-chip implementation but usually fits one of the other four defined. These interconnections are not certain by Wishbone specifications because interconnection is an IP core called INTERCON, so the system integrator can use or modify a default INTERCON or create their own. Then, the following sections will explain the four types of interconnections.

Point-to-point Interconnection

The simplest way to connect two IP cores is the interconnection point-to-point; this type of interconnection allows a single MASTER to communicate with a single SLAVE; for

example, a microprocessor IP core could be a MASTER interface, and a serial port could be a SLAVE [12]. As shown in Figure 7.

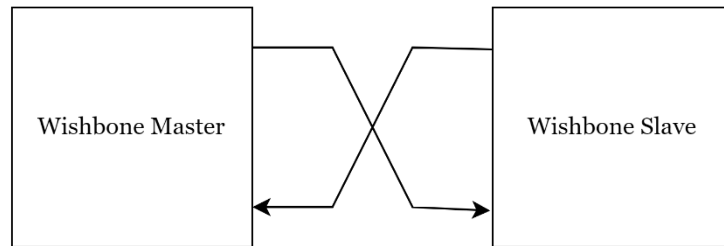

Figure 7 - Point-to-point Interconnection from [12]

Data Flow Interconnection

In data flow interconnection, each IP core has a MASTER and SLAVE interface; they are connected sequentially, the data flows core to core as shown in Figure 8; this process is also called pipelining.

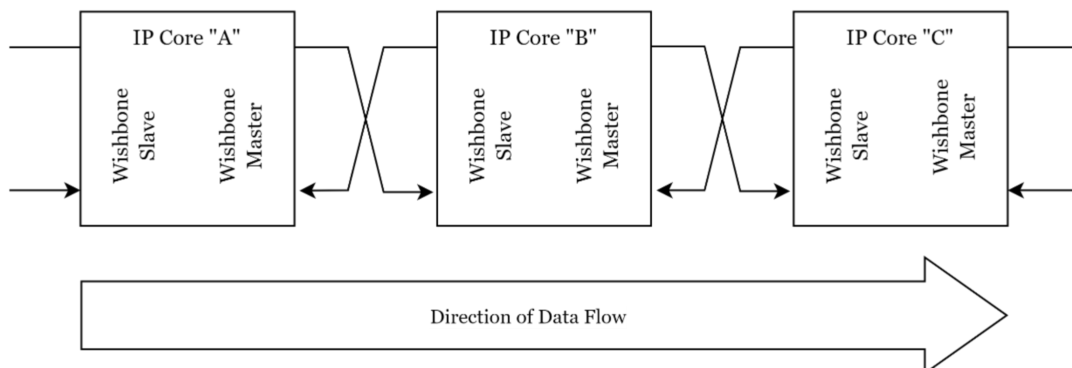

Figure 8 - Data Flow Interconnection from [12]

The data flow architecture explores parallelism, which increases the speed of execution. For example, suppose each IP core represented in Figure 8 is a floating-point processor. In this case, the system has three times the power of processing of a single unit; in this example, it is assumed that each core has the same capability to solve the problem and take the same time, and the problem can be solved sequentially. In a real application, this interconnection may not be the best option. It depends on the problem being solved [12].

Shared Bus Interconnection

The shared bus interconnection is a helpful connection type for connecting one or more MASTERs interfaces to communicate with one or more SLAVE interfaces. As shown in

Figure 9, the MASTER begins a data transfer to a selected SLAVE, and then the chosen SLAVE, in one or more bus cycles, communicates with the MASTER. An arbiter controls when the MASTER can access the shared bus [12]. The arbiter isn't shown in Figure 9, but allows the MASTER to grant access to the shared bus and tells when and to whom the MASTER has access.

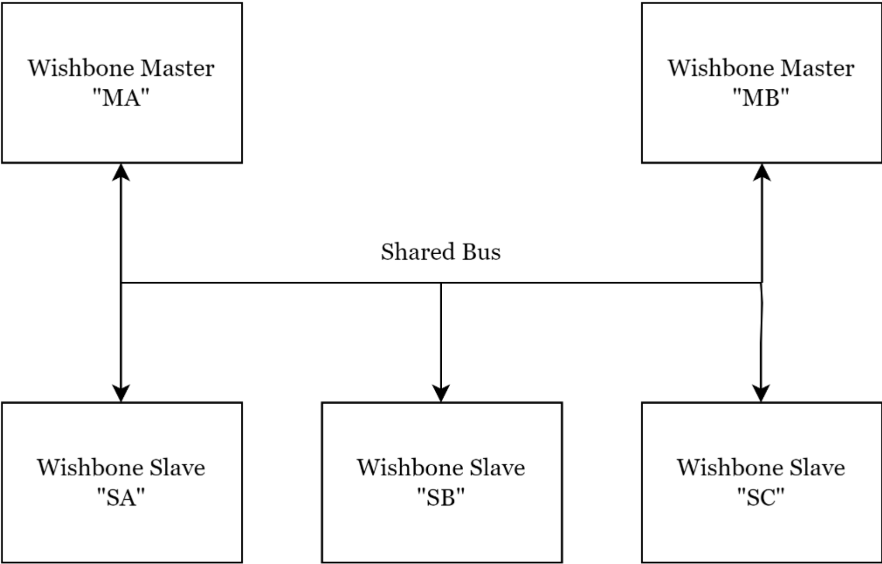

Figure 9 - Bus Shared Interconnection from [12]

Crossbar Switch Interconnection

The crossbar switch interconnection is very similar to the shared bus interconnection, but the MASTERs can communicate at the same time with different SLAVEs, for example, MASTER A can communicate with SLAVE B and MASTER B can communicate with SLAVE A. If a channel can operate with a data rate of 100 mb/s, then two channels can operate at a rate of 200 mb/s in parallel. One disadvantage of using this type of interconnection is that have more logic and routing resources [12]. Figure 10 is a schematic of the crossbar switch interconnection.

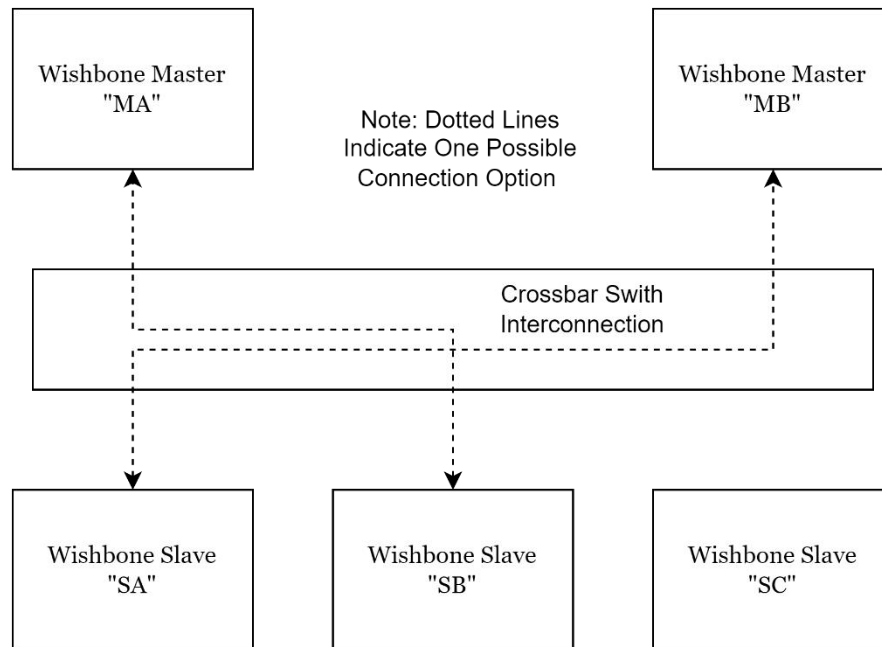

Figure 10 - Crossbar Interconnection from [12]

2.2.4 The Wishbone Bus Cycles

Wishbone has three types of bus cycles:
SINGLE READ/WRITE
BLOCK READ/WRITE
READ MODIFY WRITE(RMW)

SINGLE READ/WRITE
The SINGLE READ/WRITE is the most basic bus cycle. At clock edge 0, the MASTER presents a valid address (ADR_O), negates the write enable (WE_O), indicating a READ cycle, the select (SEL_O) is blank, indicating where it expects data and asserts cycle (CYC_O) and strobe (STB_O) to start the cycle, at clock edge 1 the SLAVE decodes inputs and assert acknowledgment (ACK_I) in response to the strobe (STB_O) signal to indicate valid data (DAT_I), the SLAVE presents a valid data, the MASTER monitors ACK_I and prepares to receive the data on DAT_I, at clock edge 2, MASTER receive the data on DAT_I and negates STB_O and CYC_O to indicate the end of the cycle finally the SLAVE negates ACK_I in response to the negated STB_O of the MASTER. The write cycle is very similar, but instead of MASTER receiving data, it will send it to the SLAVE; the differences are that WE_O will be asserted to represent a WRITE cycle and will present valid data on DAT_O [12].

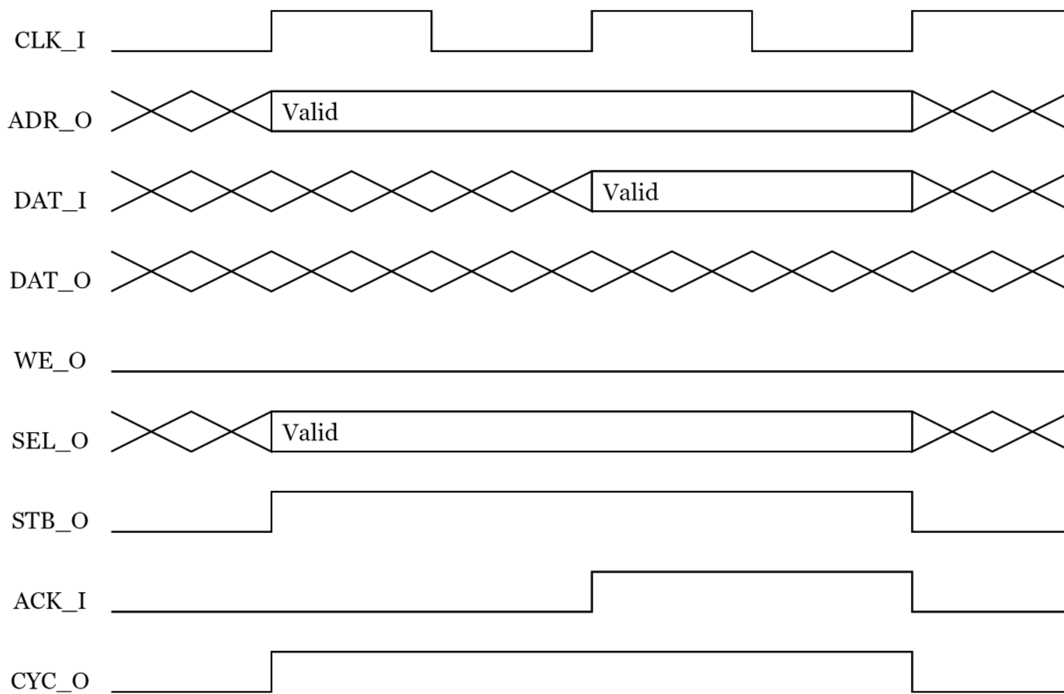

Figure 11 - Single Read from [12]

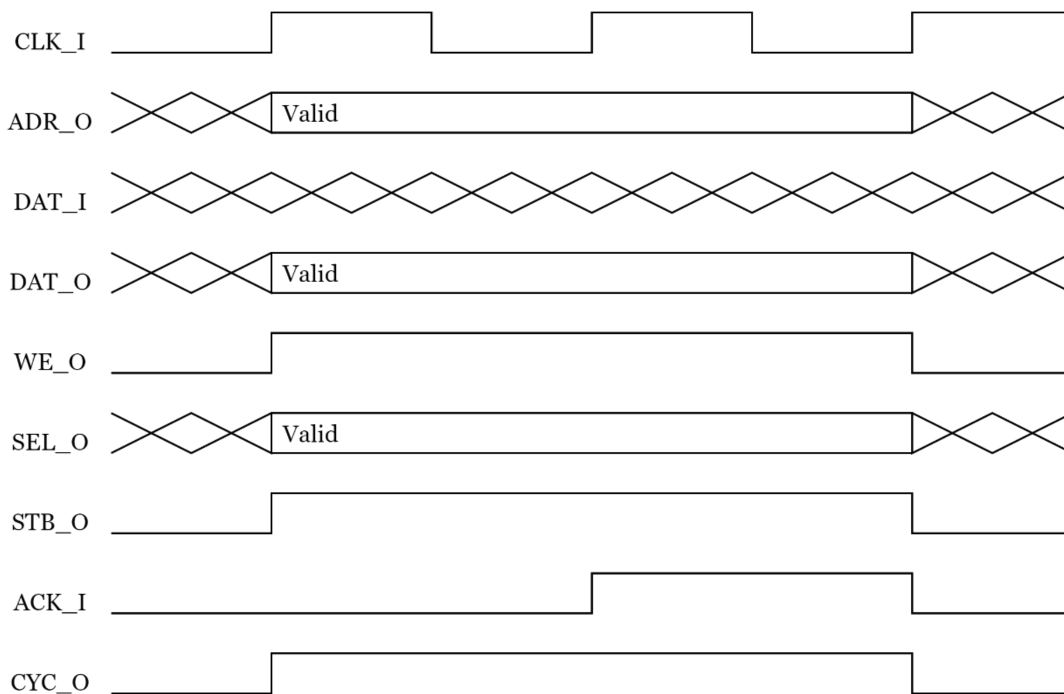

Figure 12 - Single Write from [12]

BLOCK READ/WRITE Cycles

The Block transfer cycles are very similar to the single READ/WRITE cycle, but in this cycle, they perform multiple single cycles; these individual cycles are called phases, but they are combined to form a single BLOCK cycle. This function is used when a SLAVE has a shared memory; there is an arbiter that determines when the MASTER finishes the

transfer of data and gives access to another MASTER to communicate with the same SLAVE.
On a BLOCK READ cycle, the MASTER presents a valid address (ADR_O) and bank select (SEL_O), negates write enable (WE_O) to indicate a READ cycle, asserts strobe (STB_O) and cycle (CYC_O), the CYC_O is asserted until the end of BLOCK cycle when is negated mean that [3]the block was transferred, the STB is asserted in every phase and is negated when acknowledgment (ACK_I) is negated, the ACK_I just during one clock cycle because represent the end of the phase, when every phase is over CYC_O is negated [12]. The process is shown in figure 13.

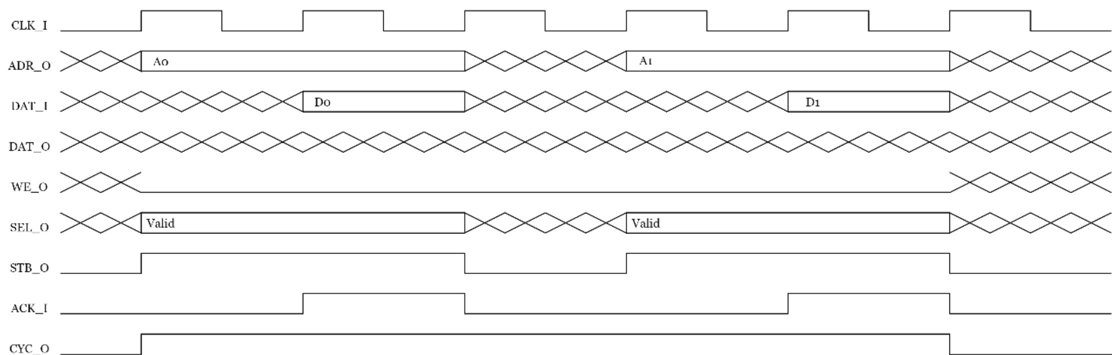

Figure 13 - Block Read from [12]

The BLOCK WRITE is very similar to the READ cycle. The difference is that WE_O is asserted, indicating that it is the WRITE cycle and has to have valid data on DAT_O.

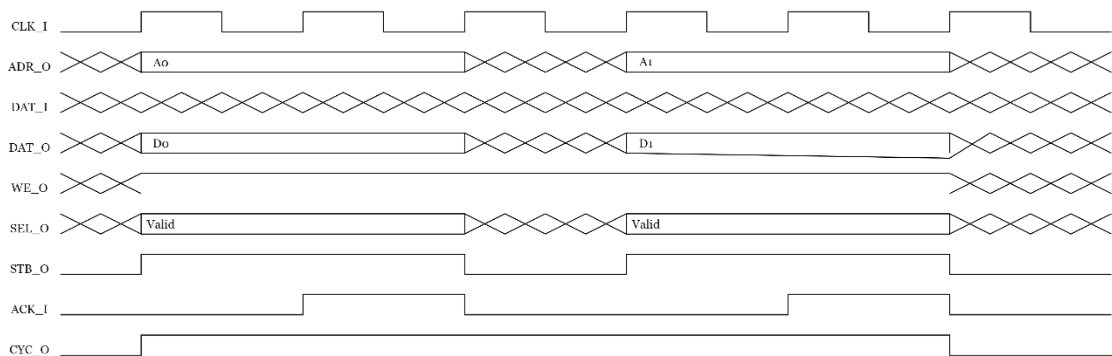

Figure 14 - Block Write from [12]

RMW Cycle

The RMW (read-modify-write) is used in multiprocessors and multitasking systems. This cycle allows multiprocessors to use the same resources as disk controllers, serial ports, and memory, also known as an indivisible cycle. This cycle has two phases: a reading phase and a writing phase. In RMW, there is an arbiter that drives the data flow, ensuring there is no missing information. If the arbiter grants access to a SLAVE, have to be sure

that no other MASTER accesses the same SLAVE during the cycle, for this is used as a semaphore that monitors the CYC_O signal because, during the cycle, this signal stays asserted, so if MASTER wants to have access to SLAVE have to check this semaphore. This semaphore is crucial to the system's excellent working. For example, a system has two microprocessors and one memory. When the first microprocessor accesses the memory, it takes some data, reads from it then, modifies it, and finally writes back to the memory. If the second microprocessor tries to access the memory during the cycle, it will take incorrect data because the first microprocessor has not yet finished the cycle; this may lead to crashes in the system. Is also important to guarantee that the interfaces support RMW cycles [12]. The diagram signal of this cycle is very similar to the diagram of the BLOCK cycle, but instead of reading or writing different blocks in this, perform a read and write in the same cycle, as shown in Figure 15.

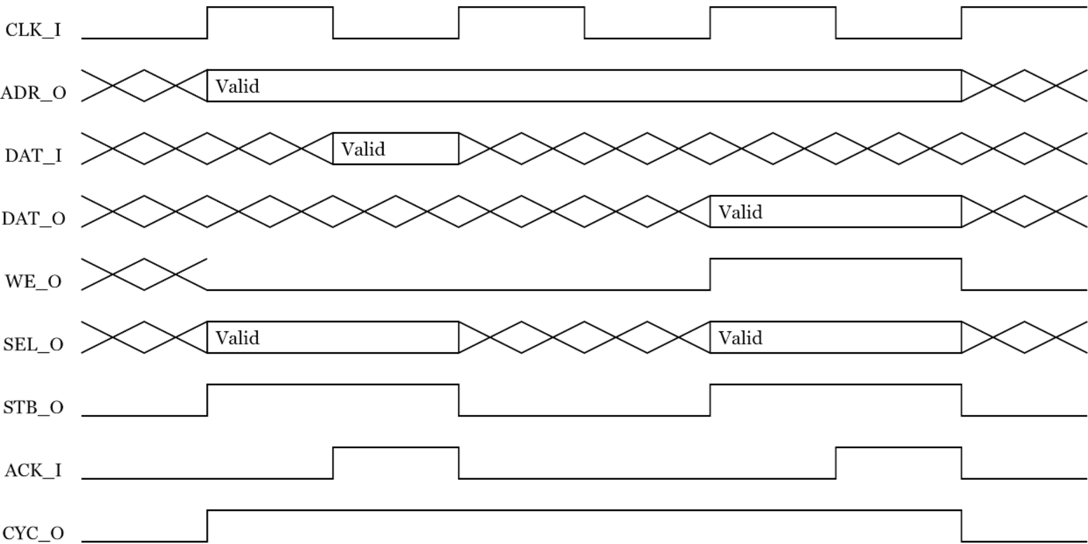

Figure 15 - RWM Cycle from [12]

2.2.5 Wishbone Signals on Simulation

To demonstrate Wishbone signals, a module was elaborated that adds two registers; these registers have different addresses, and the result is saved in a third one. Using this simple module, it is possible to see the two most basic operations in Wishbone, a single read and write.

Chapter 4 fully describes how to connect a module to RISC-V using the Wishbone interconnection; for now, the focus is analyzing the results obtained from this experiment.

The module created is represented in Figures 16, 17, and 18.

```
1   `timescale 1ns/1ps
2
3   module base(
4                   input  wire        clk_i,          // clock
5                   input  wire        rst_i,          // reset (synchronous active high)
6                   input  wire        cyc_i,          // cycle
7                   input  wire        stb_i,          // strobe
8                   input  wire [7:0]  adr_i,          // address
9                   input  wire        we_i,           // write enable
10                  input  wire [31:0] dat_i,           // data input
11                  output wire [31:0] dat_o,           // data output
12                  output reg         ack_o,          // normal bus termination
13                  output wire        rty_o           // retry output
14  );
15
16      // Wishbone interface
17      wire wb_acc = cyc_i & stb_i;        // WISHBONE access
18      wire wb_wr  = wb_acc & we_i;        // WISHBONE write access
19
20      // ack_o
21      always @(posedge clk_i) begin
22          if (rst_i)
23              ack_o <= 1'b0;
24          else
25              ack_o <= wb_acc & !ack_o;
26      end
27
28      // rty_o
29      assign rty_o  = 1'b0;
```

Figure 16 - Input/Output Module, Wishbone Interface

```
31      localparam ADDR_A       = 8'h00;
32      localparam ADDR_B       = 8'h04;
33      localparam ADDR_C       = 8'h08;
34
35      reg  [31 : 0]   A;
36      reg  [31 : 0]   B;
37      reg  [31 : 0]   C = 32'b0;
38      wire  [31 : 0]   sum;
39      reg [31:0] temp_data;
40
41      assign dat_o = temp_data;
42      assign sum = A + B;
43
```

Figure 17 - Mnemonics, registers, and assigns

```
45  always @(posedge clk_i)
46      if (rst_i)
47        begin
48            A <= 32'h0;  // set master bit
49            B <= 32'h0;
50        end
51      else if (wb_wr)
52        begin
53          //C <= sum;
54          if (adr_i == ADDR_A)
55            A <= dat_i; // always set master bit
56
57          if (adr_i == ADDR_B)
58            B <= dat_i;
59
60          if (adr_i == ADDR_C)
61            C <= dat_i;
62
63        end
64
65    // dat_o
66    always @(posedge clk_i)
67      case(adr_i) // synopsys full_case parallel_case
68         ADDR_A:      temp_data <= A+B;
69         ADDR_B:      temp_data <= sum;
70         ADDR_C:      temp_data <= C+sum;
71        default:      temp_data <= 8'hff;
72      endcase
73
74  endmodule
```

Figure 18 - Wishbone read and write operation

In this module, the sum is done in three different ways to test the behavior of Verilog design: the first way is to add the two registers at data out, the second way is to assign to "sum"," A" plus "B" and the last one is to add "C" plus "sum" at the data out, every operation used worked. In the module, the addresses that Wishbone uses to write and read on registers and the Wishbone interface.

To see the waveform, it is necessary to create a project in PlatformIO; this step is detailed in Chapter 4.2, it is also necessary to develop a simulation in Verilator, following the steps represented in Figure 19.

```
/cygdrive/c/Users/Renato/Documents/riscvadder/verilatorSIM

Renato@DESKTOP-AOMGNMP /cygdrive/c/Users/Renato/Documents/riscvadder/verilatorSIM
$ cd ..

Renato@DESKTOP-AOMGNMP /cygdrive/c/Users/Renato/Documents/riscvadder
$ cd verilatorSIM/

Renato@DESKTOP-AOMGNMP /cygdrive/c/Users/Renato/Documents/riscvadder/verilatorSIM
$ make clean
rm Vrvfpgasim*
rm: cannot remove 'Vrvfpgasim*': No such file or directory
make: [Makefile:22: clean] Error 1 (ignored)
rm *.o
rm: cannot remove '*.o': No such file or directory
make: [Makefile:23: clean] Error 1 (ignored)
rm *.d
rm: cannot remove '*.d': No such file or directory
make: [Makefile:24: clean] Error 1 (ignored)

Renato@DESKTOP-AOMGNMP /cygdrive/c/Users/Renato/Documents/riscvadder/verilatorSIM
$ make
```

Figure 19 - Terminal commands to generate simulation

After creating the simulation, it is necessary to create a C code to test the architecture; in this case, the code used is represented in Figure 20.

```
1   // memory-mapped I/O addresses
2   #define A       0x80001600
3   #define B       0x80001604
4   #define C       0x80001608
5
6   #define READ_BASE(dir) (*(volatile unsigned *)dir)
7   #define WRITE_BASE(dir, value) { (*(volatile unsigned *)dir) = (value); }
8
9
10
11  int main ( void )
12  {
13
14      int v_a=50;
15
16      int v_b=60;
17
18      int c;
19
20      WRITE_BASE(A, v_a);
21      WRITE_BASE(B, v_b);
22
23
24
25      c=READ_BASE(C);
26
27  }
```

Figure 20 - Mnemonics and C code

This is a simple code to write and read data from the module through Wishbone.

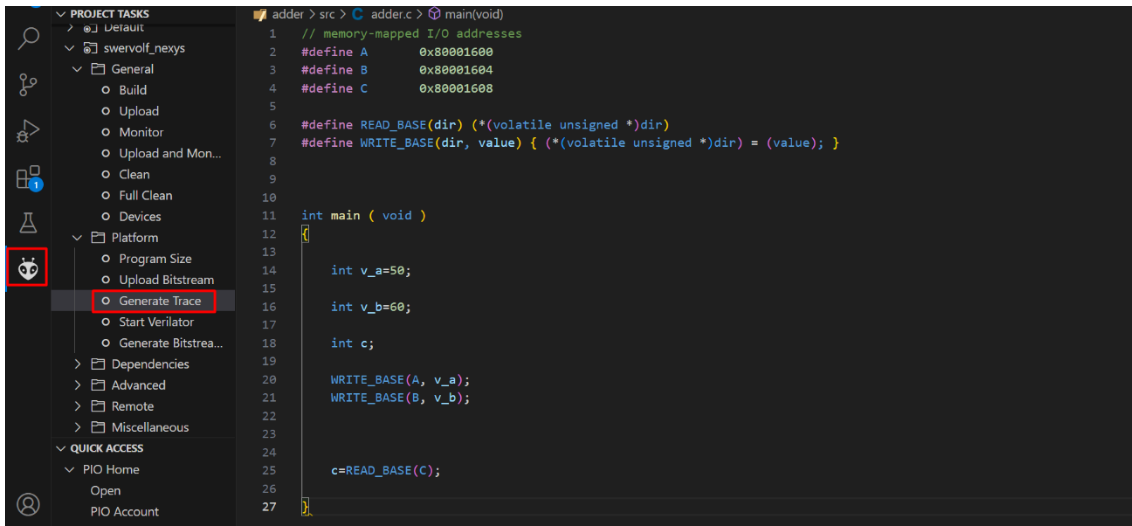

Figure 21 - How to generate trace

To obtain the waveform from the simulation, it is necessary to generate a trace; this option will run the C code in the simulation. This step is represented in Figure 21. The last step is to run the command in the VSC terminal. This step is represented in Figure 22.

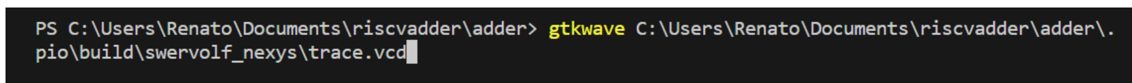

Figure 22 - Run the waveform

The simulation results are the following in Figures 23, 24, and 25.

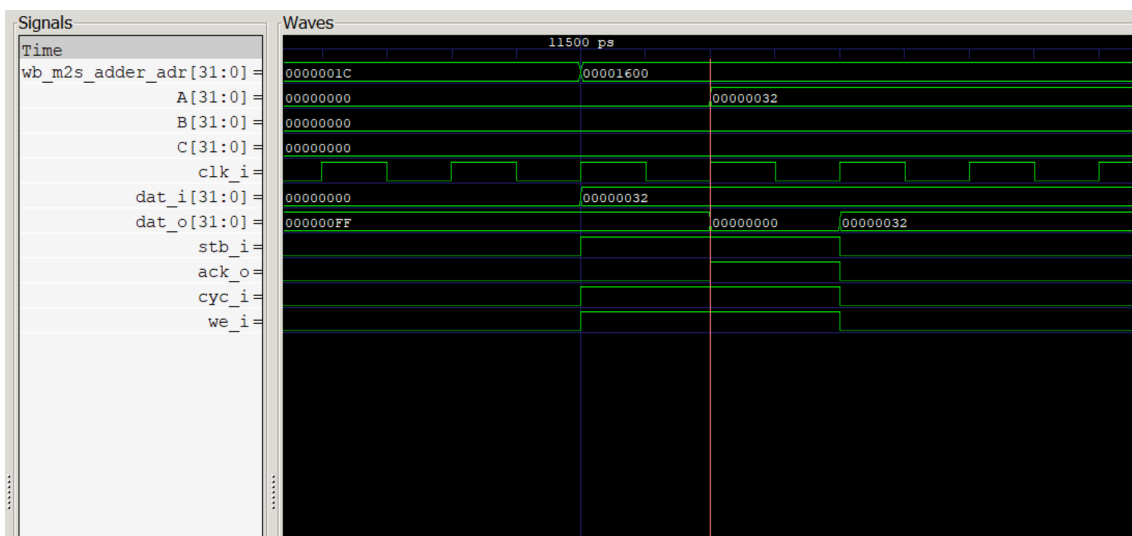

Figure 23 - Write through Wishbone on register A

Figure 23 is a single write operation on address 1600 where writes the hexadecimal value of 32 at A register, the signals of "stb", "ack", "cyc" are the same as the Wishbone specifications when performed a writing operation the signal write enable (we) is active.

The same happens in Figure 24, a single write operation on address 1604. In this operation, the data out is also the result because the module's construction performs three different ways to execute the sum.

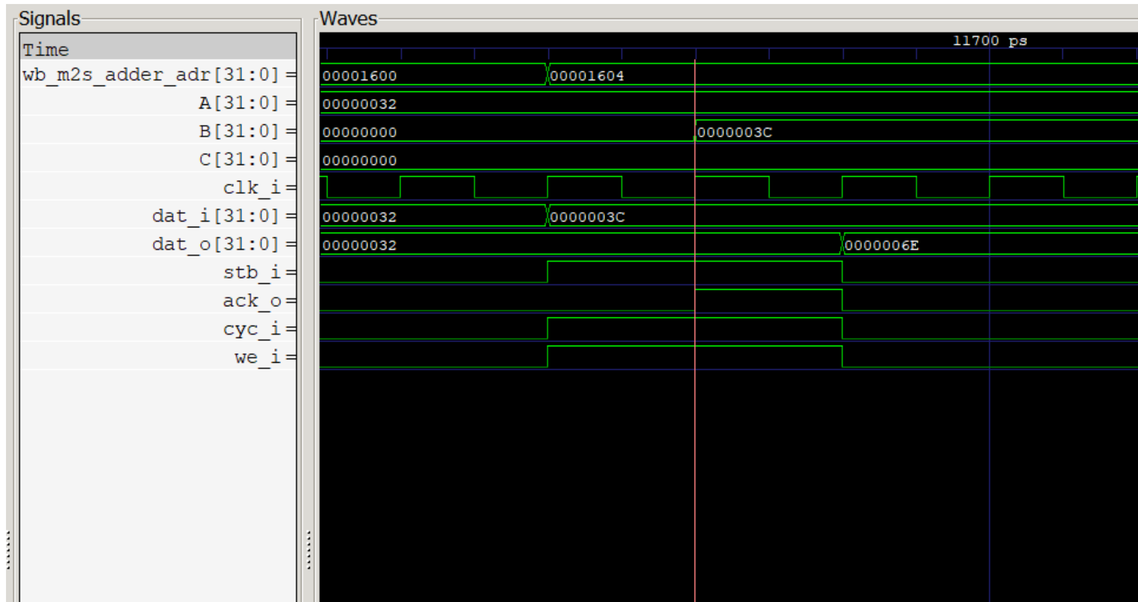

Figure 24 - Write through Wishbone on register B

Figure 25 represents the waveform of a single read operation on address 1608, in this case, the signal of "stb", "ack" and "cyc" are the same, but the write enable is not activated to perform a reading.

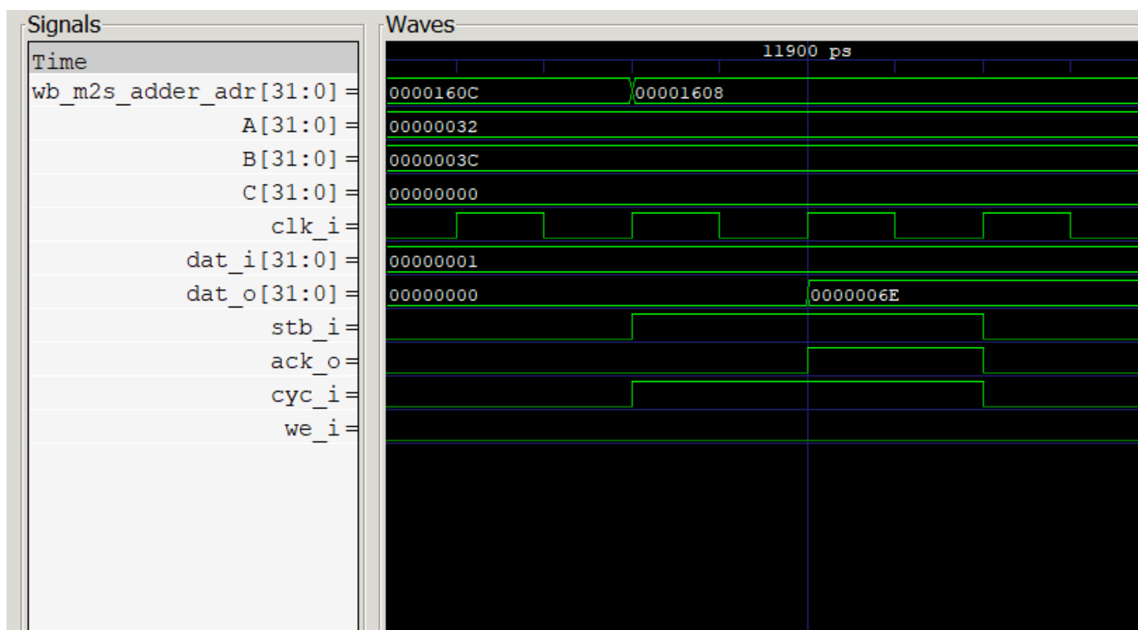

Figure 25 - Read through Wishbone on register C

2.3 XADC

2.3.1 XADC Overview

The XADC includes a dual 12-bit, 1 Mega sample per second (MSPS) ADC, and on-ship sensors; the ADC provides a high-precision analog interface for various applications. The ADCs can access 17 external analog input channels. The XADC includes some on-chip sensors for measuring power supply, voltage, and temperature on-chip. Have dedicated registers to store data conversion and registers to keep and modify the configuration of the XADC. These registers are available through the dynamic reconfiguration port (DRP) [14].

2.3.2 XADC Instantiating

Instatiating the XADC to access the on-chip monitoring capability is unnecessary, to have access to the measurement results, it is necessary to instantiate the XADC; Figure 26 shows the primitive ports of XADC, and Table 2 is a brief description of the main ports.

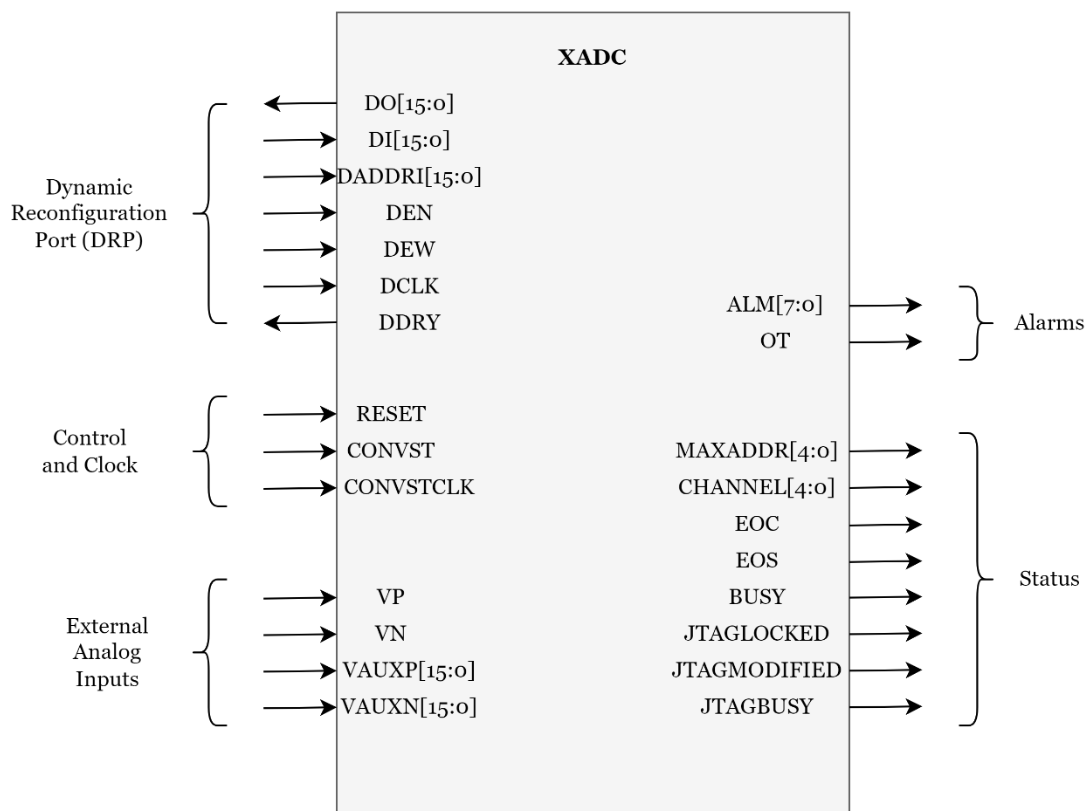

Figure 26 - Instantiation Diagram from [14]

Port	I/O	Description
DI	Input	Input data for the DRP
DO	Output	Output data for the DRP
DADDR	Input	Address for the DRP
DEN	Input	Enable for the DRP
DWE	Input	Write enable for the DRP
DRDY	Output	Data ready for the DRP
RESET	Input	Reset signal for the XADC
DCLK	Input	Clock input for the DRP
CONVST	Input	Convert start input
VAUX	Input	Auxiliary analog input
EOC	Output	End of conversion signal
EOS	Output	End of sequence signal
BUSY	Output	ADC busy signal

Table 2 - XADC signals

2.3.3 ADC Transfer Function

The ADC has transfer functions for unipolar and bipolar operating modes. All on-chip sensors use unipolar mode, but users can choose between these operating modes for external analog inputs. Figure 27 represents how data is saved into the registers. The 12 MSBs (Most Significant Bit) are the result of the conversion, and the 4 LSBs (Least Significant Bit) can be used to filter, minimize quantification, or improve resolution through averaging [14].

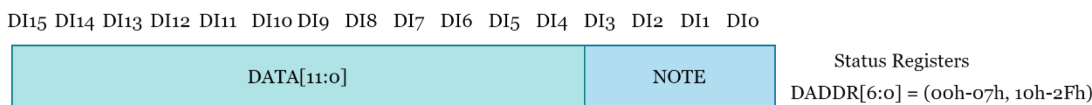

Figure 27 - Status Register Structure from [14]

Unipolar Mode

Figure 28 illustrates the 12-bit unipolar function transfer; this function has a range from 0V up to 1V; when the input of the ADC is 0V, the ADC will represent these values with 0x000h (hexadecimal), and when the voltage is 1V, the ADC will represent with 0xFFFh. In unipolar mode, it is straight binary because there are no negative numbers. As shown in Figure 28, the minimum voltage is 0,244V value, equivalent to 001h; this value comes from 1V/4096=244uV. The differential nature of these values requires measuring the positive (Vp) and negative (Vn) inputs [14].

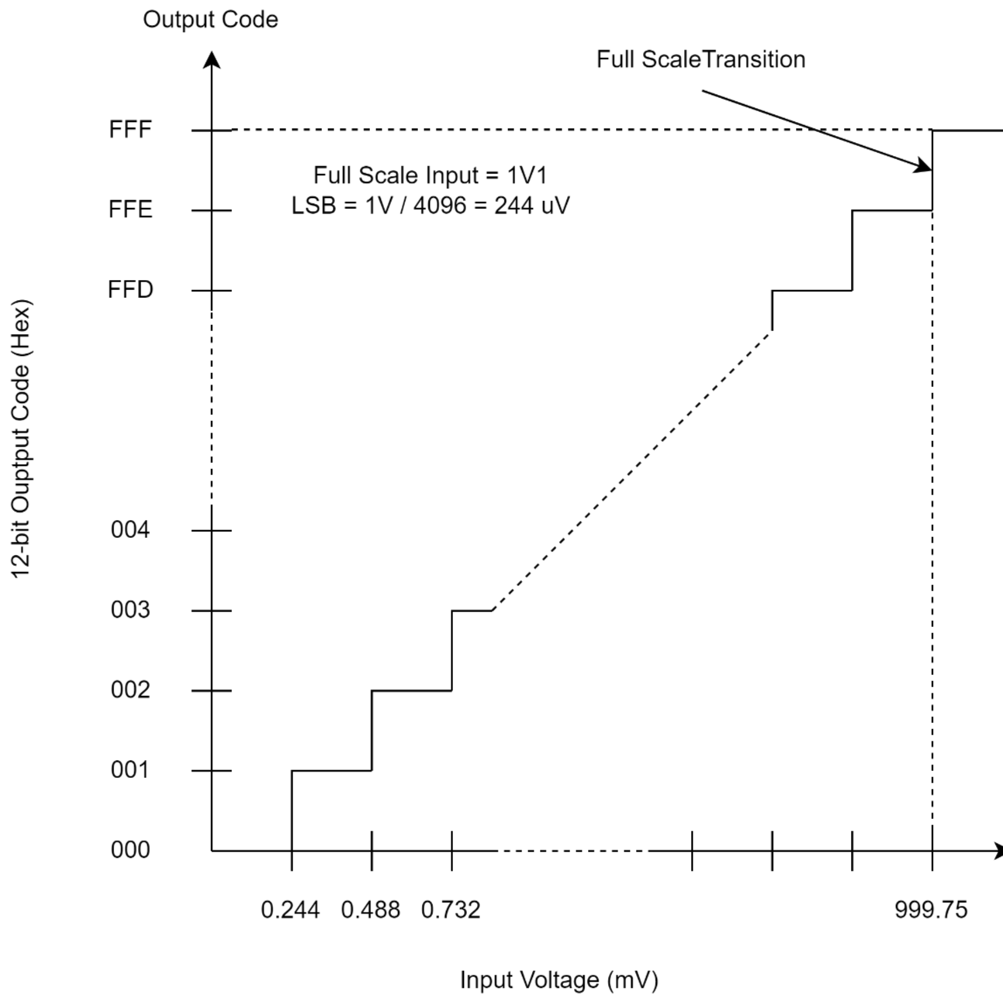

Figure 28 - Unipolar Mode from [14]

Bipolar Mode

When the external input channels are configured as bipolar, they can provide true differential and bipolar signals. While working with bipolar signals, having information regarding their signal and magnitude is crucial. In Figure 29, it is possible to see how the ADC gives a value in hexadecimal according to the input value.

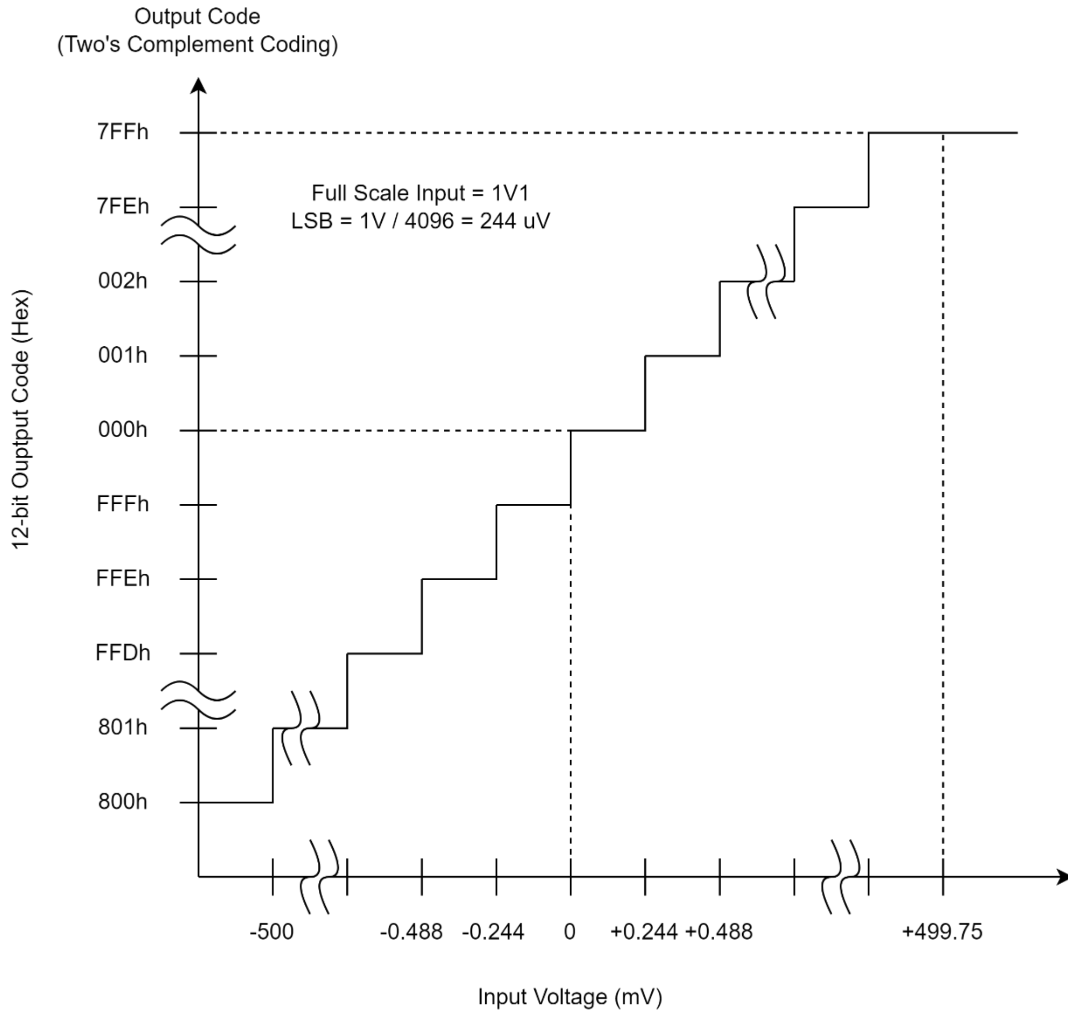

Figure 29 - Bipolar Mode from [14]

2.3.4 Analog Inputs

A differential sampling scheme is used at the analog inputs of the ADC to reduce the effects of common-mode noise signal. In noisy digital environments, the common-mode rejection improves the ADC performance. Figure 30 shows the difference of the signal with and without differential sampling. Common ground impedances (Rg) connect noise voltages (switching digital currents) to other parts of the system, and these noise signals can reach up to 100mV or more. When converting from analog to digital, these noisy signals can mean hundreds of LSBs, representing a significant measurement error. The differential sampling scheme samples the signal and common mode noise voltages at analog inputs (Vp and Vn). The Track-and-Hold amplifier captures the difference between Vp and Vn. To take advantage of the common mode rejection, the user has to connect Vp and Vn in a differential configuration [14].

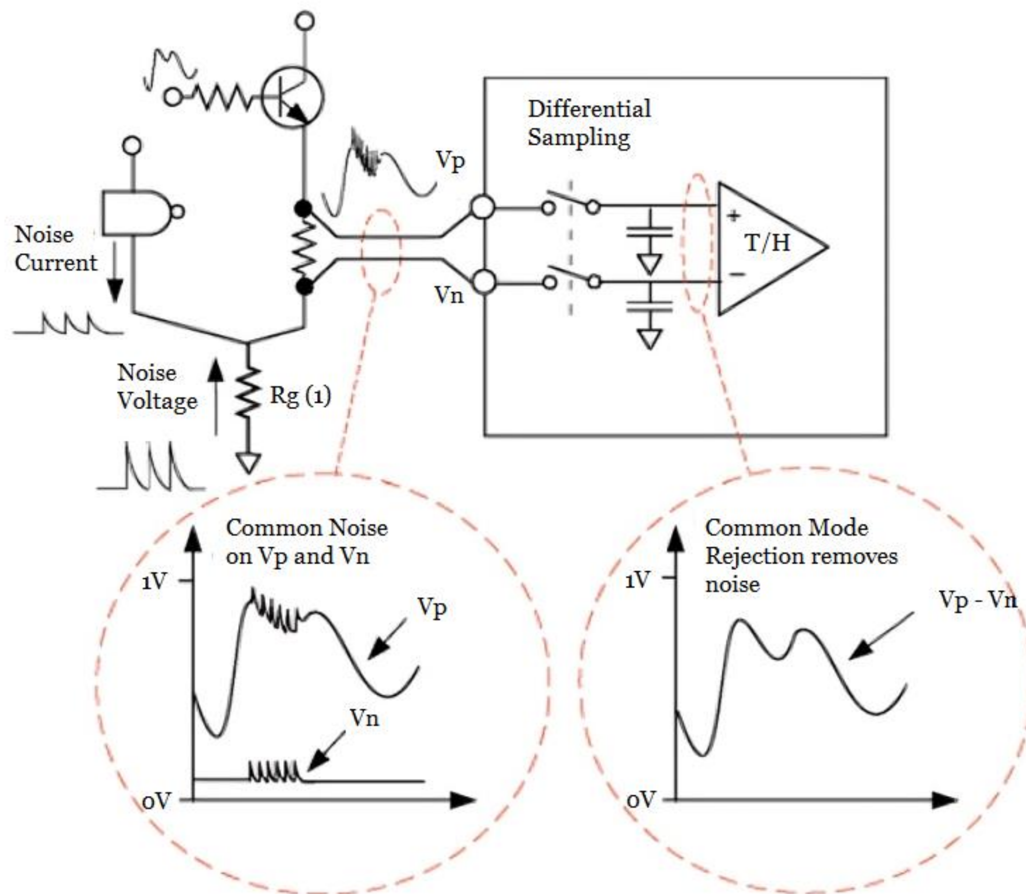

Figure 30 - Common mode noise rejection from [14]

Auxiliary Analog Inputs

The auxiliary analog inputs, described as VAUXP and VAUXN, are automatically enabled when the XADC is instantiated in a design without requiring user constraints or pin locations.

2.3.5 XADC Registers Interface

The XADC register interface has two categories: status registers and control registers. The status registers are read-only and contain measurement data, and the control registers are readable or writable and save the configuration of the ADC. These registers are accessible through the dynamic reconfiguration port (DRP).

Status registers

The status registers are the first 64 address locations (DADDR[6:0] = 00h to 3Fh); they contain all results from ADC conversion, such as the on-chip sensors and external analog channels. Every sensor and external channel has a unique address.

For example, the result of the on-chip temperature sensor is stored at address 00h, the external inputs are from 10h to 1Fh, and the XADC also records the maximum and minimum of the sensors on-chip [14].

Control registers

There are 32 control registers located at addresses 40h to 5Fh. These registers control all functionality of the XADC. When the XADC is instantiated in a design, the control registers are initialized using the XADC attributes. There are four types of control registers: configuration, test, sequence, and alarm, as shown in Figure 31.
The configuration registers are three (40h, 41h, and 42h). They are used, for example, to select the input channel on single mode or external mode, configure if the ADC is operating in continuous mode or event-driven sampling, and for analog inputs, if the ADC is on unipolar mode or bipolar mode.
The test registers are intended for factory test purposes. These shouldn't be written. Have a default status of zero. They go from 0x43h to 0x47h.
The channel sequencer registers are used to program the channel sequencer functionality.
The alarm registers are used to program automatic alarms in sensors on-chip. It is possible to select the values of thresholds, and when some of the sensors register a value upper or lower of the predefined in these registers, it generates an alarm [14].

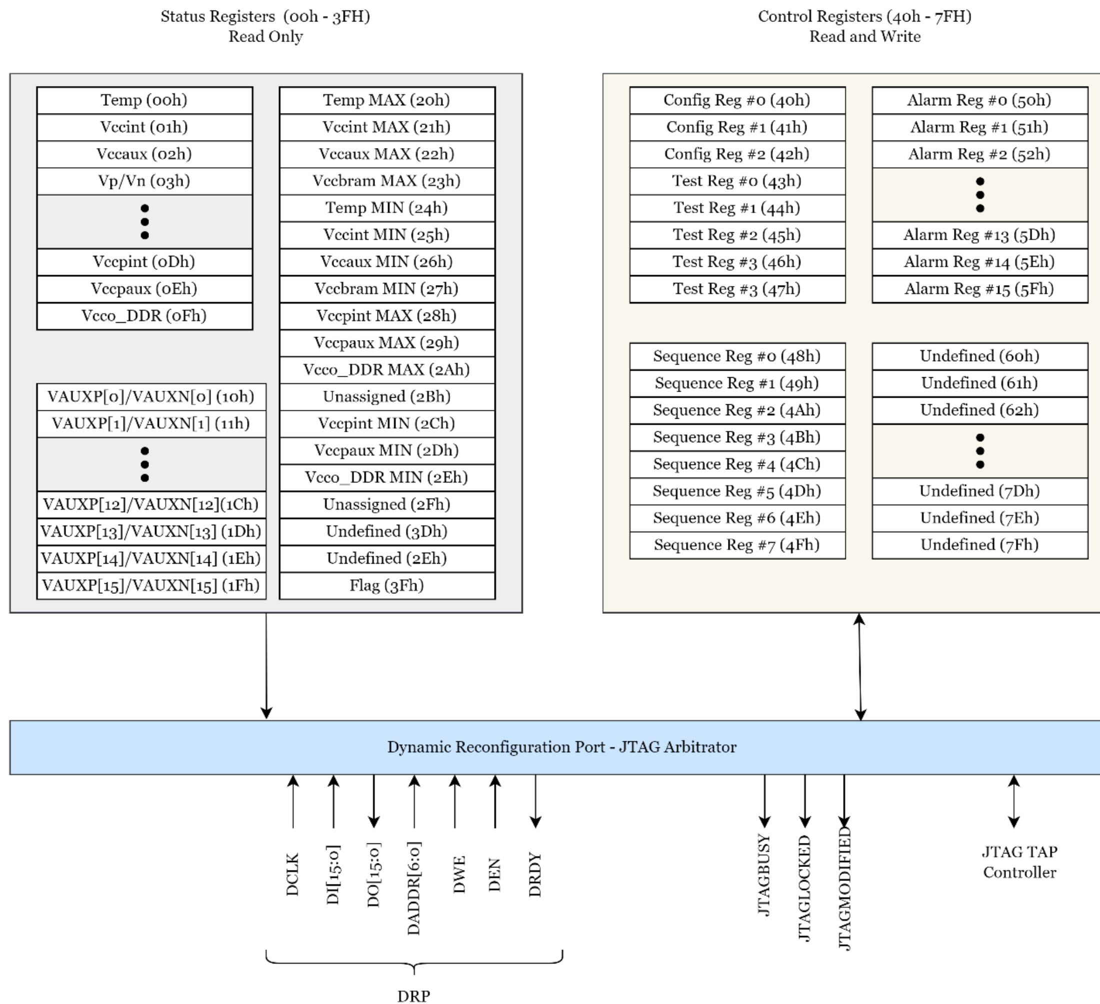

Figure 31 - XADC Registers from [14]

2.3.6 XADC Operating Modes

The XADC offers several operating modes to adapt to different applications. Unless otherwise noted, these modes can operate in continuous mode and event-driven sampling modes.

The ADC performs sampling in single-channel mode and saves the data at the selected address.

In the automatic channel sequencer, the sequencer selects the next conversion channel, sets the averaging, configures the analog input channels, sets the required settling time for acquisition, and stores the result in the status registers based on a once-off setting. The selection registers are 0x48h and 0x49h. The 0x48h address selects the on-chip sensors, and 0x49h selects the external analog inputs. The channel averaging selects the channels with averaging and can use 16, 64, or 256 samples. The amount of samples is configured on the control register 40h on bit AVG1 and AVG2, and on registers 4Ah and 4Bh, choosing what channels have enabled averaging is possible. The channel analog-

input mode selects if the ADC runs on unipolar or bipolar modes and uses the addresses 0x4Ch and 0x4Dh. The channel settling time by default is four ADCCLK cycles, but on addresses 0x4Eh and 0x4Fh, it is possible to set channels to have ten ADCCLK cycles to settle.

There are several sequencer modes. The default mode automatically monitors all the on-chip sensors and stores them in status registers. In the single pass mode, the sequencer sees the channels enabled and converts every selected channel. Then, another conversion is needed to change the operating mode and change it to single-pass mode again. The continuous sequence mode is like the single pass mode but operates in a loop. The simultaneous sampling mode makes it possible to sample simultaneously from different channels; this is useful in applications that preserve the phase between two signals. In the independent ADC mode, the ADC A is responsible for the alarms of the on-chip sensors, and the ADC B is assigned to auxiliary analog inputs. The external multiplexer mode allows a multiplexer to expand the number of channels from one channel up to sixteen channels, or it is possible to use two multiplexers and do simultaneous sampling [14]; these schematics are shown in Figure 32. It is also possible to get the maximum and minimum of the on-chip sensors and set alarms for a range of values defined by the developer.

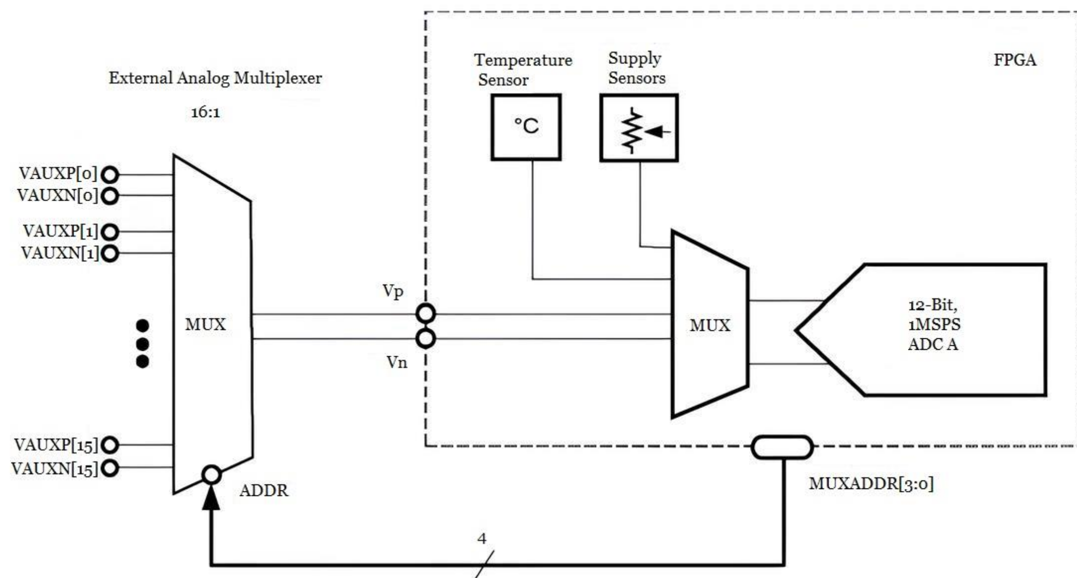

Figure 32 - External Analog Multiplexer from [14]

2.3.7 XADC Timing

The XADC clock is synchronized with the DRP clock (DCLK), DCLK generates the ADCCLK, and it is possible to change the period of ADCCLK in configuration registers;

the ADCCLK is only available internally on XADC. There are two possible timing modes: continuous and event-driven mode.

Continuous Sampling

The analog to digital conversion has two parts: the acquisition and conversion phases. The acquisition phase involves charging a capacitor in the ADC from the selected channel; this phase lasts four ADCCLK by default. The conversion phase starts at the next rising DCLK, and for the next 4 or 10 ADCCLK, according to settling time, the busy goes high, indicating that a conversion is taking place. This phase takes 22 ADCCLK cycles, then busy goes low, past sixteen DCLK of busy goes low, the end of conversion goes high for one DCLK, and the data goes to output registers [14]; this process is represented in Figure 33.

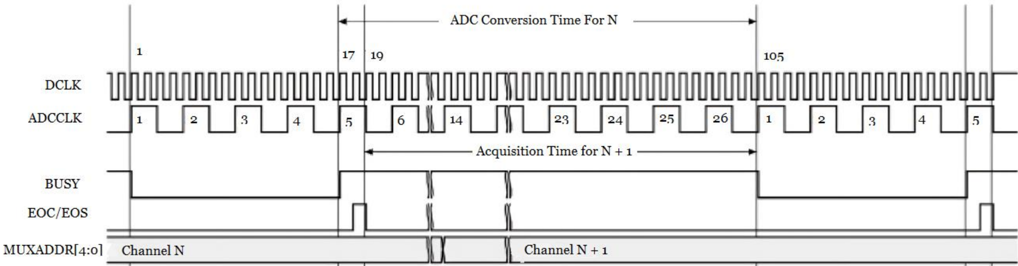

Figure 33 - Continuous Sampling from [14]

Event-Driven Sampling

In event-driven sampling, the sampling instant and conversion process are initiated by a trigger signal called convert start (CONVST); this is useful to have precise control over the sampling instant; this mode requires DCLK if this signal isn't present the XADC reverts to continuous mode. When CONVST goes from low to high, it defines the exact sampling moment; on the next DCLK, the busy signal goes high; the CONVST can be an asynchronous signal, so the XADC automatically resynchronizes the conversion process to the ADCCLK [14]. The process is similar to the continuous mode, as is possible to see in Figure 34. The major change is the trigger signal.

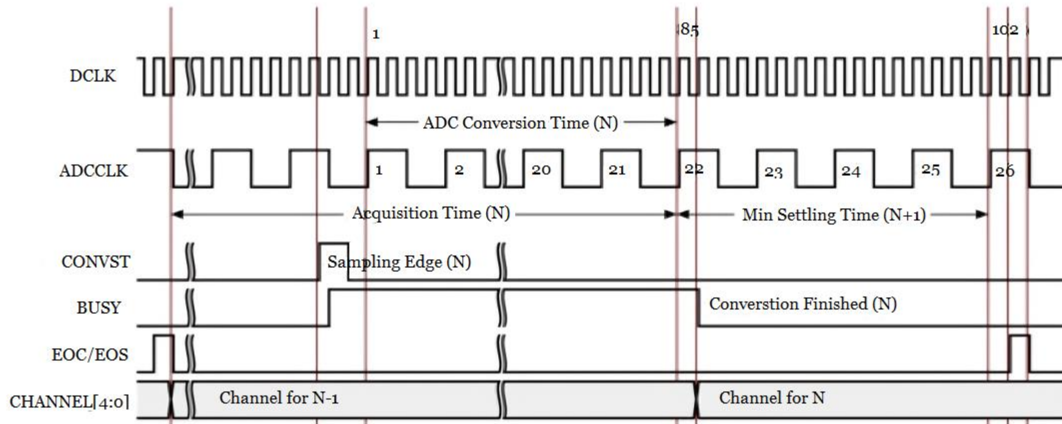

Figure 34 - Event-Driven Sampling from [14]

Dynamic Reconfiguration Port (DRP) Timing

The DRP allows to read and write on the XADC. When performing a read operation, the DEN signal goes high for a one-clock cycle, the DRP captures the write enable (DWE) and address (DADDR). If DWE is low is performed a reading operation on the address selected, and the data is available on data out (DO) when the ready (DRDY) signal goes high. If DWE is high, it captures the data in (DI) and the DADDR; the operation is also over when the DRDY goes high [14]. These timing operations are represented in Figure 35.

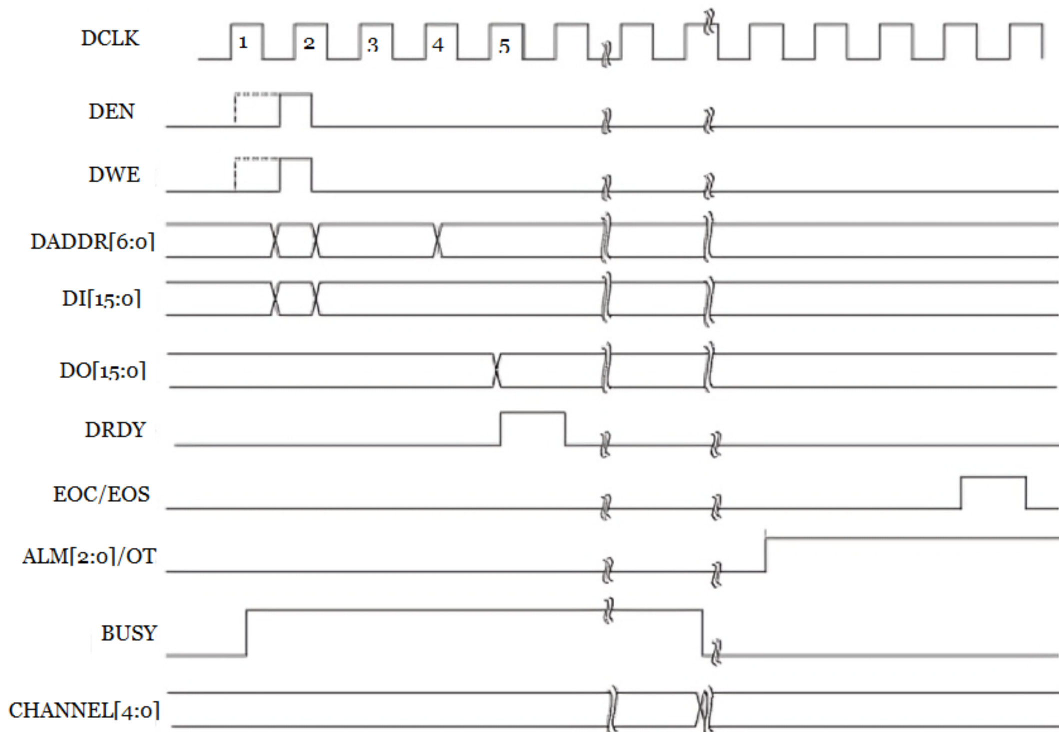

Figure 35 - DRP Timing from [14]

Chapter 3

Integration of XADC on RISC-V

3.1 Software Tools

Vivado

Vivado is a software design suite for AMD adaptive SoCs and FPGAs. Provides an extensive set of tools for designing, implementing, and verifying digital circuits. Vivado allows the user to design entries in traditional hardware description languages (HDLs) like VHDL and Verilog, and has a graphical user interface called IP Integrator to integrate pre-designed IP blocks into the design, Vivado synthesizes the HDL code into a netlist of logic gates and other components, Vivado places and routes the logic gates and other components in the netlist onto the FPGA fabric, Vivado provides a variety of tools for verifying the functionality and timing correctness of the design, including simulation, emulation, and static timing analysis [15].

Visual Studio Code

Visual Studio Code is a code editor that enables users to edit, debug, and automate tasks. It is a cross-platform editor that runs on Windows, macOS, and Linux. VS Code is extensible through a rich ecosystem of extensions. Extensions exist for many embedded systems development tasks, such as debugging, code completion, and hardware abstraction layers. VS Code is highly customizable. Users can change their appearance and behavior to suit their user preferences. VS Code also has a debugger to execute the code step by step and see the values held by the variables. Finally, VS Code has an embedded terminal that allows users to execute command lines in the editor [16].

PlatformIO

PlatformIO is a specialized IDE for embedded systems supporting more than 1000 boards. This dissertation uses the VSC extension, allowing you to use the VS Code and PlatformIO functionalities. PlatformIO enables loading bitstreams as well as loading C programs and debugging. PlatformIO also allows using the serial port for communication, if necessary, with a computer [17].

Icarus Verilog

Icarus is a simulator of hardware open-source that can be used to simulate and debug Verilog designs, allows users to test the behavior of projects digitally, avoid unnecessary costs to the projects, also allows users to create a VCD file to see a waveform of every signal of the design [18].

GTKWave

GTKWave is a waveform viewer that is capable of opening VCD files. This software allows to see the waveform of hardware simulations. This software is essential in the creation of hardware because it will enable to see the behavior of the hardware design graphically. The main features are allowing to see the signals over time and to see the module signals, for example, when working on a particular module, mainly to see the signals that enter and leave that module and sometimes specific signals from the same module. These features help in development because It is possible to focus on what is essential. In addition to these features, there is a marker that return to specific points in the simulation [19].

3.2 Hardware Tools

Nexys A7

The Nexys A7 board includes a Field-Programmable Gate Array (FPGA). An FPGA is an integrated circuit that can be programmed after manufacturing. This makes FPGA a very powerful tool able to be used to implement a wide range of digital circuits. Besides, this also allows to fix bugs and add new features. To configure the FPGA is used the hardware description language (HDL), like Verilog or VHDL, these languages describes the behavior of the FPGA, the codes generate configuration files to load into the FPGA, and the configuration file tells the FPGA how to connect the logic blocks to create the desired circuit. These logic blocks contain a set of combinatorial blocks and flip-flops. An FPGA includes memory that works in conjunction with the logic, and the manufacturer chooses the amount of memory in the chip, which is called RAM (Random Access Memory). Generally, FPGA supports two forms of clock, Delay Locked Loops (DLLs) and Phase Locked Loops (PLLs). Finally, the FPGA has communication pins, which can be inputs, outputs, or both, in this way, communication between the chip and other components[20],[21],[7].
The Nexys A7 is a development board created by Digilent. This board is equipped with Artix-7 XC7A100T-1CSG324C FPGA from Xilinx with the following characteristics [22]:

- 15850 Programmable logic slices, each with four 6-input LUTs and 8 flip-flops
- 4860 Kbits of fast block RAM
- Six clock management tiles, each with a phase-locked loop (PLL)
- 240 DSP slices
- Internal clock speeds exceeding 450 MHz
- Dual-channel, 1 MSPS internal analog-to-digital converter (XADC)

This board has several peripherals:

- Memory, 128MB DDR2, Serial Flash, MicroSD card slot.
- USB and Ethernet, 10/100 Ethernet PHY, USB-JTAG programming circuitry, USB UART bridge, USB HID Host for mice, keyboards, and memory sticks.
- Simple User Input/Output, 16 switches, 16 LEDs, 2 RGB LEDs, 2 4-digit 7-segment displays.
- Audio and Video, 12-bit VGA output, PWM audio output, PDM microphone.
- Sensors, 3-axis accelerometer, temperature sensor.
- Expansion Connectors, Pmod connector for XADC signals, 4 Pmod connectors providing 32 total FPGA I/O.

3.3 XADC Verilog Development

Before building the module, it's important to simulate how the XADC works. Chapter 2.3 introduces the XADC, how to instantiate the wizard and the signal for conversion and reconfiguration by DRP. Following the specification in Chapter 2.3, the following module, represented in Figure 36, was created.

```
1   `timescale 1ns / 1ps
2
3   module XADC_test(
4       input  wire [15 : 0] di_in,
5       input  wire [6 : 0] daddr_in,
6       input  wire den_in,
7       input  wire dwe_in,
8       output wire drdy_out,
9       output wire [15 : 0] do_out,
10      input  wire dclk_in,
11      input  wire reset_in,
12      input wire convst_in,
13      input  wire vp_in,
14      input  wire vn_in,
15      output wire user_temp_alarm_out,
16      output wire vccint_alarm_out,
17      output wire vccaux_alarm_out,
18      output wire ot_out,
19      output wire [4 : 0] channel_out,
20      output wire eoc_out,
21      output wire vbram_alarm_out,
22      output wire alarm_out,
23      output wire eos_out,
24      output wire busy_out
25      );
26
27  xadc_wiz_0 xiladc (
28    .di_in(di_in),                              // input wire [15 : 0] di_in
29    .daddr_in(daddr_in),                        // input wire [6 : 0] daddr_in
30    .den_in(den_in),                            // input wire den_in
31    .dwe_in(dwe_in),                            // input wire dwe_in
32    .drdy_out(drdy_out),                        // output wire drdy_out
33    .do_out(do_out),                            // output wire [15 : 0] do_out
34    .dclk_in(dclk_in),                          // input wire dclk_in
35    .reset_in(reset_in),                        // input wire reset_in
36    .convst_in(convst_in),                      // input wire convst_in
37    .vp_in(vp_in),                              // input wire vp_in
38    .vn_in(vn_in),                              // input wire vn_in
39    .user_temp_alarm_out(user_temp_alarm_out),  // output wire user_temp_alarm_out
40    .vccint_alarm_out(vccint_alarm_out),        // output wire vccint_alarm_out
41    .vccaux_alarm_out(vccaux_alarm_out),        // output wire vccaux_alarm_out
42    .ot_out(ot_out),                            // output wire ot_out
43    .channel_out(channel_out),                  // output wire [4 : 0] channel_out
44    .eoc_out(eoc_out),                          // output wire eoc_out
45    .vbram_alarm_out(vbram_alarm_out),          // output wire vbram_alarm_out
46    .alarm_out(alarm_out),                      // output wire alarm_out
47    .eos_out(eos_out),                          // output wire eos_out
48    .busy_out(busy_out)                         // output wire busy_out
49  );
```

Figure 36 – XADC module, input/output, and instantiation

The module, represented in Figure 36, is an introductory module only used to test the interface of the instantiation of the wizard and is composed of two parts: the input/output and the instantiation of the wizard. In the input/output part, every signal used to read and control the XADC is declared. The instantiation part is where the variables of the module are associated with the wizard; in this case, are used the inputs and outputs; can be used because they are set as "wire". Another way is necessary to do what is represented in Figure 37.

```
1   `timescale 1ns / 1ps
2
3   module XADC_test(
4       input  di_in,
5       input  daddr_in,
6       input  den_in,
7       input  dwe_in,
8       output drdy_out,
9       output do_out,
10      input  dclk_in,
11      input  reset_in,
12      input convst_in,
13      input  vp_in,
14      input  vn_in,
15      output user_temp_alarm_out,
16      output vccint_alarm_out,
17      output vccaux_alarm_out,
18      output ot_out,
19      output channel_out,
20      output eoc_out,
21      output vbram_alarm_out,
22      output alarm_out,
23      output eos_out,
24      output busy_out
25      );
26
27      wire [15 : 0] di_in;
28      wire [6 : 0] daddr_in;
29      wire den_in;
30      wire dwe_in;
31      wire drdy_out;
32      wire [15 : 0] do_out,
```

Figure 37 - Testbench input/output and variables

The first method saves time when writing code and produces the same result.
For this module to be usable, it is necessary to configure the wizard that allows the use of XADC; this wizard can be found in the IP Catalog on Vivado. Figure 38 represents all the steps to find the wizard; clicking on "IP Catalog" will appear in a new tab where it is possible to search on the search box by the "XADC".

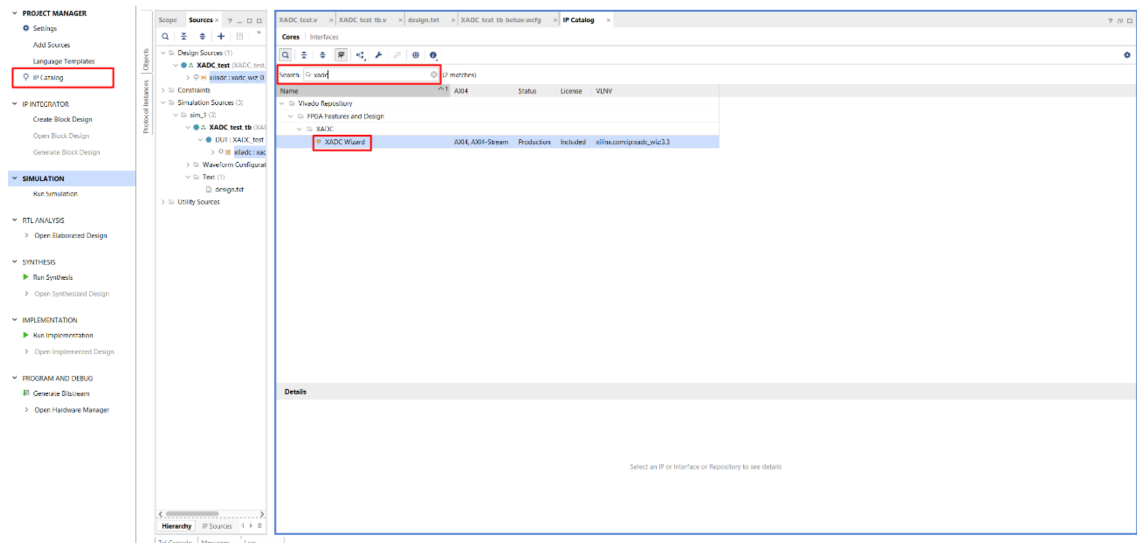

Figure 38 - Find XADC wizard

By double-click, the "XADC Wizard" will appear in a new window, in this window, it is possible to configure the XADC, from interface options, timing mode, startup channel selection, DRP timing mode, control/status ports, calibration, alarms, there are more configurations, but these are the most relevant. For this work, it is necessary to configure the XADC with the definitions set in Figure 39, the interface has to be DRP, to be able to reconfigure the XADC while running, the DRP allows to write on configuration registers, the timing mode has to be event mode this way is possible to generate events and only when this happens the XADC will convert a reading, the startup channel selection has to be channel sequencer to read from different channels, on event mode trigger is used the “convst in”, on the analog sim file options has to be the same represented in Figure 39. In the other tabs, the options are more arbitrary.

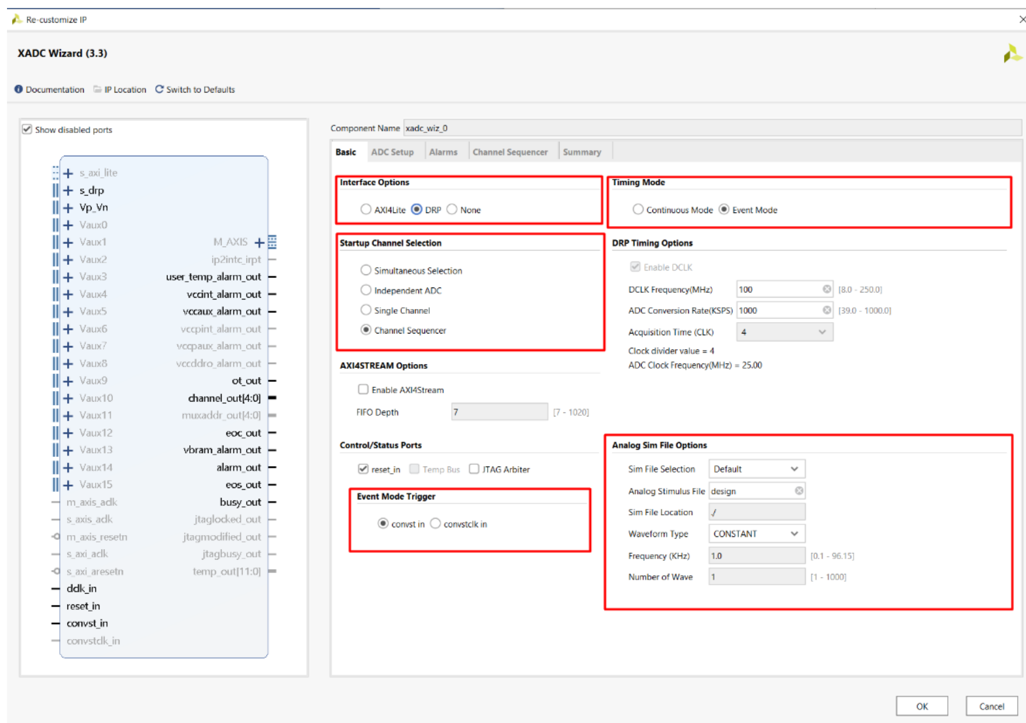

Figure 39 - XADC wizard configurations

The main module is fully completed after these steps are completed, and the whole code is available in Appendix A. To see the behavior of the main module is necessary to create a testbench, following the steps in Figure 40, click on "Add sources", "Add or create simulation sources", click "Next". In the next window represented in Figure 41, clicking on "Create file" will pop up a new window to name the file, select the file type and the file location. The file type is set as "Verilog", the location is set to default, the file name is arbitrary.

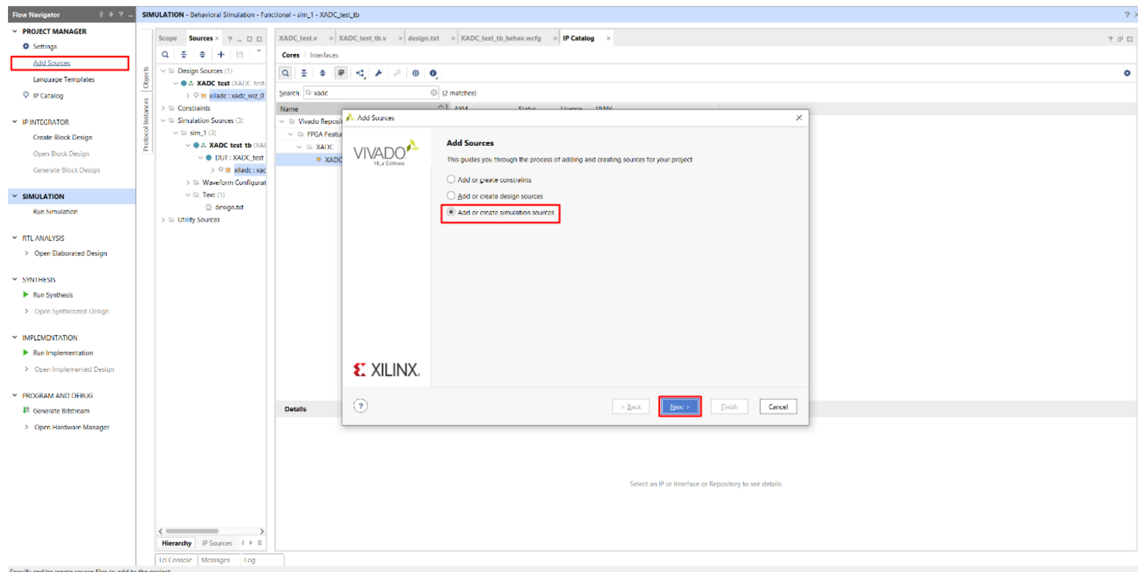

Figure 40 - Add testbench file

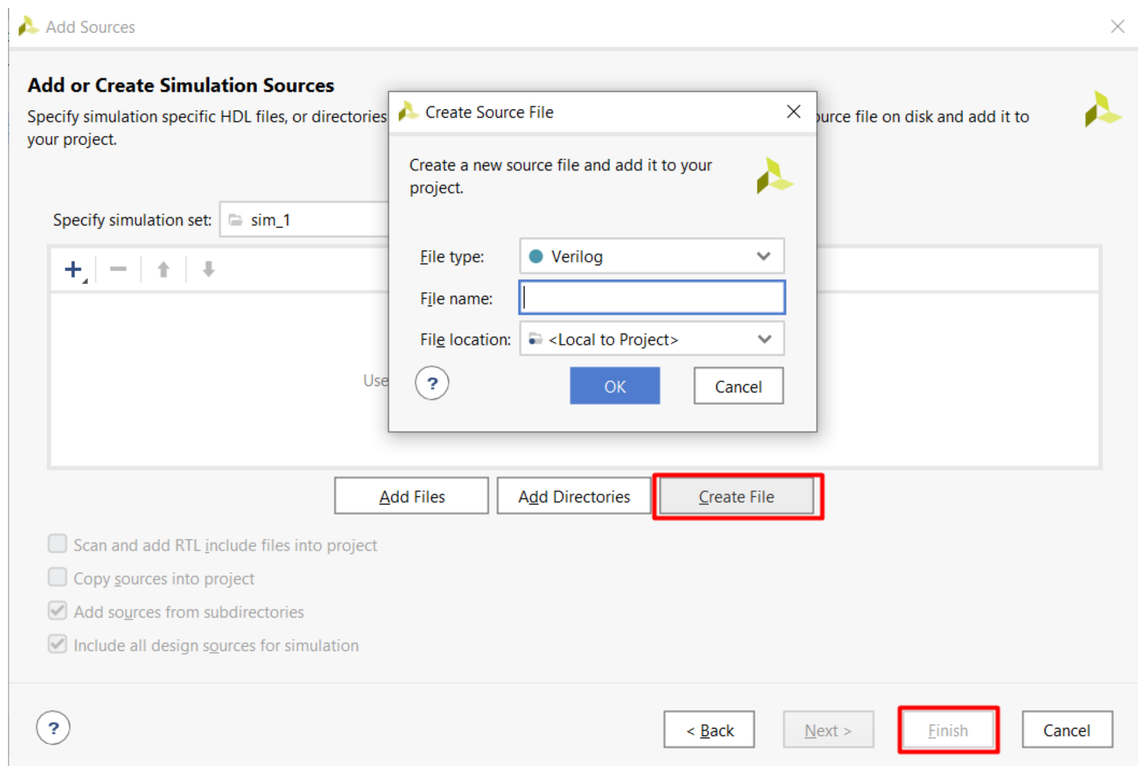

Figure 41 - Create a testbench file

Before starting to work on the testbench, adding another file to the simulation using the same process is necessary. In this case, instead of creating a new file to add a file to the simulation, figure 42 is represented the file needed, "design.txt", to archive the same result is necessary to set the "Files of type: " to "All files", another way is not possible to find this file, the file directory look like [Project name]/[Project name].ip_user_files/mem_init_files.

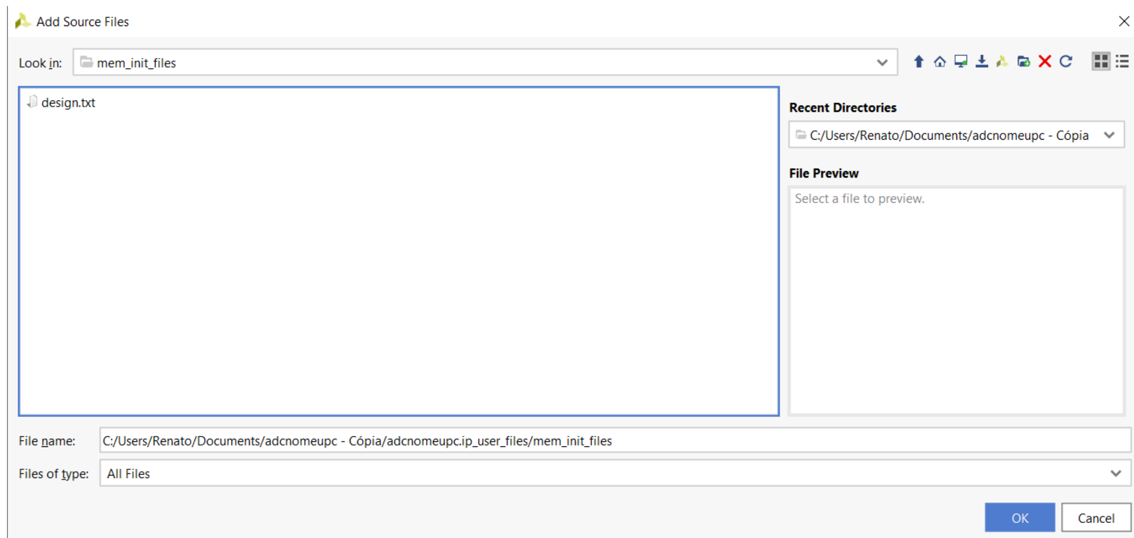

Figure 42 - Add emulation file

This "design.txt" file emulates the conditions of the test. Once are not reading data from the real world, this file sets the value read by the XADC; for example, this file defines the temperature from instant 0 to instant 1250 nanoseconds as 63 degrees Celsius, as represented in Figure 43.

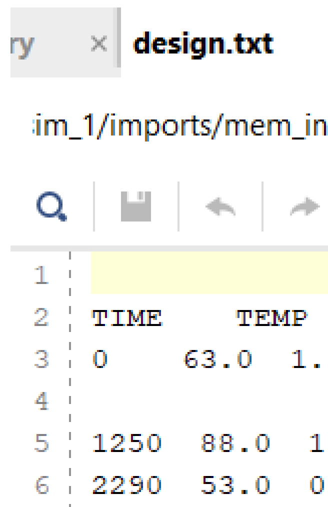

Figure 43 - Emulation file struture

With the testbench file created and added the "design.txt" file, is possible to start working on the testbench. In the testbench, the inputs of the main module turn into registers because it wants to have control of the inputs to see the module's behavior; the outputs on the testbench turn into wires to see the module's outcome; this is represented in Figure 44.

```
1   `timescale 1ns / 1ps
2
3   module XADC_test_tb(
4
5       );
6
7       reg [15 : 0] di_in;
8       reg [6 : 0] daddr_in;
9       reg den_in;
10      reg dwe_in;
11      wire drdy_out;
12      wire [15 : 0] do_out;
13      reg dclk_in;
14      reg reset_in;
15      reg convst_in;
16      wire vp_in;
17      wire vn_in;
18      wire user_temp_alarm_out;
19      wire vccint_alarm_out;
20      wire vccaux_alarm_out;
21      wire ot_out;
22      wire [4 : 0] channel_out;
23      wire eoc_out;
24      wire vbram_alarm_out;
25      wire alarm_out;
26      wire eos_out;
27      wire busy_out;
28
29      initial
30      begin
31          dclk_in = 1'b0;
32          forever #5 dclk_in = ~dclk_in;
33      end
34
35      initial
36      begin
37          convst_in = 0;
38          di_in = 4'h0000;
39          den_in = 0;
40          dwe_in = 0;
41          daddr_in = 2'h00;
42          reset_in = 1'b1;
43          #200 reset_in = 1'b0;
44          #400
45          daddr_in = 6'b0000000;
46          convst_in = 1;
47          #10
48          convst_in = 0;
49          #1160
```

Figure 44 - XADC testbench

Also, Figure 44 represents how the clock signal is generated; it starts at 0, and every 5 ns is logically negated, alternating between 0 and 1. After the block of clock generation is the block of test, where the value of the registers such as an address, data read, data write, and "convst_in" are set. These registers are manipulated to do three events: start a conversion, read data, and write data; the whole code is in Appendix B. The block represented in Figure 45 generates automatic readings after a conversion.

```
always@(posedge dclk_in)begin
    if(eoc_out == 1)begin
        den_in <= 1;
        #10
        den_in <= 0;
    end
end
```

Figure 45 - Reading after conversion code

The last part of the test bench is to instantiate the main module for the testbench to know where the test will be performed, as represented in Figure 46.

```
100      XADC_test DUT(
101          .di_in(di_in),
102          .daddr_in(daddr_in),
103          .den_in(den_in),
104          .dwe_in(dwe_in),
105          .drdy_out(drdy_out),
106          .do_out(do_out),
107          .dclk_in(dclk_in),
108          .reset_in(reset_in),
109          .convst_in(convst_in),
110          .vp_in(1'b0),
111          .vn_in(1'b0),
112          .user_temp_alarm_out(user_temp_alarm_out),
113          .vccint_alarm_out(vccint_alarm_out),
114          .vccaux_alarm_out(vccaux_alarm_out),
115          .ot_out(ot_out),
116          .channel_out(channel_out),
117          .eoc_out(eoc_out),
118          .vbram_alarm_out(vbram_alarm_out),
119          .alarm_out(alarm_out),
120          .eos_out(eos_out),
121          .busy_out(busy_out)
122      );
123
124
125  endmodule
```

Figure 46 - Instantiation XADC module

With the main module and the testbench created, it is possible to simulate on Vivado. After simulating, if do not appear the whole simulation, one possible resolution is to run the command "run [time of simulation in ns] ", this way, if everything else is correct, it will solve the problem.

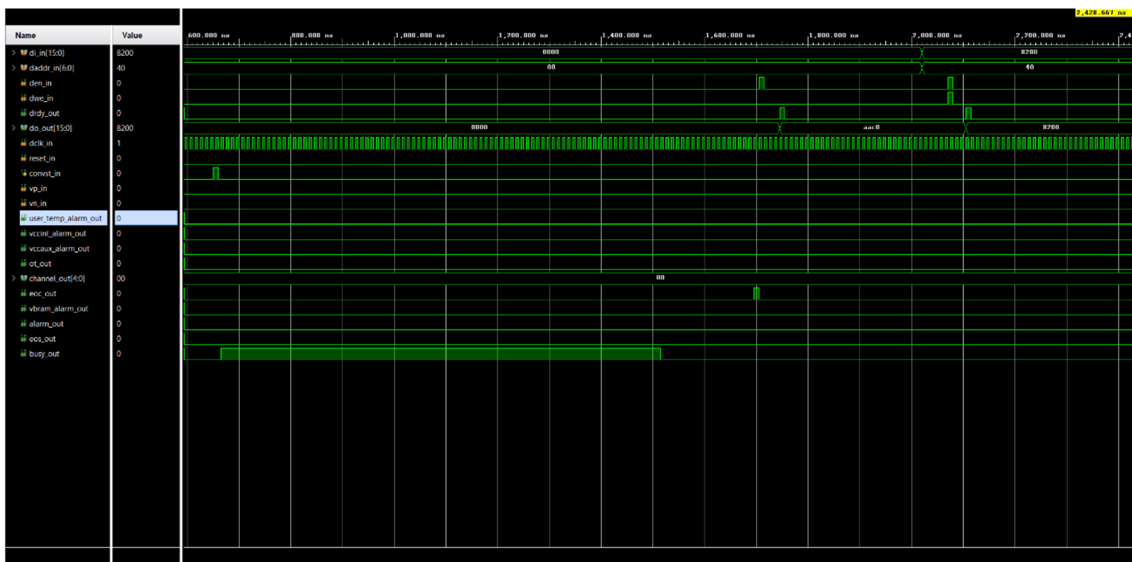

Figure 47 - XADC simulation on Vivado

Figure 47 represents the result of the simulation. Analyzing this waveform is possible to confer the specifications of manufacture when "convst_in" goes high for one clock signal, in the next rising clock the signal "busy_out" goes high, when "busy_out" goes low after

16 clock signals, the signal end of conversion (eoc_out) goes high, in the testbench was created a branch of code to every time a conversion is done reading is executed, so the signal "den_in" goes high. After a few clock cycles, the signal "drdy_out" goes high. The value is ready to be read in data out (do_out), in the data out is possible to see the value "AAC0", at this value the three most significative is used to convert this into temperature in Celsius, by converting AAC from hexadecimal to decimal is given the value of 2732, this is the ADC code used in the equation 3.1.

$$Temperature(°C) = \frac{ADC\ code \times 503.975}{4096} - 273.15 \qquad (3.1)$$

Replacing the ADC code for 2732, the result is approximately 63, this value is the same on the "design.txt" file. By seeing this, two out of the three operations have already been verified: creating an event to start a conversion and reading data from the XADC. For the last operation, is possible to see that the value of data in (di_in), and the value of address in (daddr_in) change, the value of “di_in” is what is wanted to write on XADC, and on "daddr_in" is the value where is wanted to write, in this example, is wanted to write "0x8200" on address "0x40". Analyzing the signals it is possible to see when is enabled the read and write enable (den_in and dwe_in) at the same time, after a few clock signals the ready signal (drdy_out) goes high and on data out (do_out) appears the value written on the XADC confirming the operation. After working on this simulation, was obtained knowledge of how XADC works, this way, it is possible to start working on the XADC module that will be integrated into RISC-V architecture.

Star building the XADC module to be integrated into the RISC-V architecture is necessary to create the interface that allows the module to communicate, this module will communicate through the Wishbone interface, so the inputs and outputs are the Wishbone signals, with the addition of the externals inputs from the “PMOD” connector to perform an external reading of the chip. The signals of the Wishbone will perform a complete control on the module, such as clock, reset, the address where is to read or write, data read, data write, and control if the operation is completed. This is represented in Figure 48.

```
 1  `timescale 1ns/1ps
 2
 3  module CHIPXADC(
 4      input   wire        clk_i,      //check
 5      input   wire        rst_i,      //check
 6      input   wire        cyc_i,      //check
 7      input   wire [7:0]  adr_i,      //endereÃ§o de entrada
 8      input   wire [31:0] dat_i,      //dados de entrada
 9      input   wire [3:0]  sel_i,      //check
10      input   wire        we_i,       //check
11      input   wire        stb_i,      //check
12      output  wire [31:0] dat_o,      //dados de saida
13      output  reg         ack_o,      //check
14
15
16      //external
17      input vauxn3,
18      input vauxp3,
19      input vauxn10,
20      input vauxp10,
21      input vauxn2,
22      input vauxp2,
23      input vauxn11,
24      input vauxp11,
25      input vp_in,
26      input vn_in);
27
```

Figure 48 - XADC input/output with Wishbone

After the interface, are defined some mnemonics to the addresses used by the module, these mnemonics are used to help the development and avoid errors, these mnemonics allow to attribute letters to numbers, for example, if the address used to save data is "8'h04", every time that is needed to use instead of write these number, is possible to write just "ADDR_DATA" and don't be confused by other address. The mnemonics used are represented in Figure 49.

```
28      localparam ADDR_ADR        = 8'h00;
29      localparam ADDR_DATA       = 8'h04;
30      localparam ADDR_STATUS     = 8'h08;
31      localparam ADDR_CRTL       = 8'h0c;
32      localparam ADDR_TESTE      = 8'h10;
33      localparam ADDR_RW         = 8'h14;
```

Figure 49 - Used mnemonics

It is also necessary to create registers and wires to operate the module but these are in the complete code in Appendix C.

As seen in the simulation is needed to instantiate the wizard that allows the module to communicate to the XADC, this instantiation is very similar to the simulation, the main difference is in the signal that allows to read and write on the XADC, in this module is used the register "den[0]" to enable the reading and the register "dwe[0]" to enable the writing. Figure 50 represents the instantiation of the XADC on the module.

```
71      xadc_wiz_0 XLXI_7 (
72            .convst_in(conv_in),                // Convert Start Input
73            .daddr_in(address_in),  //DRP       // Address bus for the dynamic reconfiguration port
74            .dclk_in(clk_i),        //DRP       // Clock input for the dynamic reconfiguration port
75            .den_in(den[0]),        //DRP       // Enable Signal for the dynamic reconfiguration port
76            .di_in(data_in),        //DRP      // Input data bus for the dynamic reconfiguration port
77            .dwe_in(dwe[0]),   //DRP     // Write Enable for the dynamic reconfiguration port
78            .reset_in(0),                       // Reset signal for the System Monitor control logic
79            .vauxp2(vauxp2),                    // Auxiliary channel 2
80            .vauxn2(vauxn2),
81            .vauxp3(vauxp3),                    // Auxiliary channel 3
82            .vauxn3(vauxn3),
83            .vauxp10(vauxp10),                  // Auxiliary channel 10
84            .vauxn10(vauxn10),
85            .vauxp11(vauxp11),                  // Auxiliary channel 11
86            .vauxn11(vauxn11),
87            .busy_out(busy),                        // ADC Busy signal
88            .channel_out(),                     // Channel Selection Outputs
89            .do_out(datakeep),      //DRP       // Output data bus for dynamic reconfiguration port
90            .drdy_out(ready),       //DRP       // Data ready signal for the dynamic reconfiguration port
91            .eoc_out(enable),                   // End of Conversion Signal
92            .eos_out(),                         // End of Sequence Signal
93            .vccaux_alarm_out(),                // VCCAUX-sensor alarm output
94            .vccint_alarm_out(),                //  VCCINT-sensor alarm output
95            .alarm_out(),                       // OR'ed output of all the Alarms
96            .vp_in(vp_in),                      // Dedicated Analog Input Pair
97            .vn_in(vn_in));
```

Figure 50 - XADC wizard instantiation

After instantiating the XADC, it is necessary to perform some operations to read and write data in the module through Wishbone. Figure 51 shows these operations.

```
103     // Wishbone interface
104     wire wb_acc = cyc_i & stb_i;        // WISHBONE access
105     wire wb_wr  = wb_acc & we_i;        // WISHBONE write access
106
107     // ack_o
108     always @(posedge clk_i) begin
109         if (rst_i)
110             ack_o <= 1'b0;
111         else
112             ack_o <= wb_acc & !ack_o;
113     end
114
115
116     //dat_in
117     always @(posedge clk_i)begin:ola
118     data_out = 32'h0;
119       if (rst_i)
120         begin
121           address_in <= 32'h0;  // set master bit
122           //data = 32'h0;
123           STATUS = 32'h0;
124         end
125       else if (wb_wr)
126         begin
127           if (adr_i == ADDR_ADR)
128             address_in <= dat_i; // always set master bit
129
130           if (adr_i == ADDR_DATA)
131             data_in <= dat_i;
132
133           if (adr_i == ADDR_CRTL)
134             CRTL <= dat_i;
135
136           if (adr_i == ADDR_STATUS)
137             STATUS <= dat_i;
138
139           if (adr_i == ADDR_TESTE)
140             teste <= dat_i;
141
142           if (adr_i == ADDR_RW)
143             rw = dat_i;
144         end
145
146     // dat_o
147       case(adr_i) // synopsys full_case parallel_case
148          ADDR_ADR:    data_out = address_in;
149          ADDR_DATA:   data_out = data;
150          ADDR_STATUS: data_out = stat;
151          ADDR_CRTL:   data_out = CORTL;
```

Figure 51 - Wishbone interface

The last part of the code is where XADC is controlled. This part is constituted by three always blocks, the first block is to create a driven-event to start a conversion on the

XADC, and for this to happen, an always block is checking in every ascendant clock cycle if the register "CRTL" is equal to one, this register "CRTL" is written by Wishbone, until this register is zero nothing happen, when is equal to one, the register “conv” is load with one to start the conversion, then start checking the busy signal, when is different of zero, means that the conversion it began so the "conv" register goes again to zero because the driven-event should go only high for one clock signal, as specified in the user guide.
The second always block checks if the data is ready to be used; when the data is ready, the data from XADC goes to the register "data", which will be read by the Wishbone.
The last always block is responsible for controlling the read and write on the XADC registers, from register 00h to 3Fh are only read registers where XADC writes the conversions, from register 40h to 7Fh are read and write registers where is possible to configure anything on the XADC, configurations, sequences and alarms [14], to control these two operations is necessary a register in this case “rw” to define the operations on the XADC, when "rw" is zero the XADC do not perform any operation when the "rw" is one the XADC perform a reading operation on the XADC, the condition to perform any operation is the busy signal be zero, if busy signal equal to zero, the register "den" is load the value 2, this happens to create a signal that dure only one clock signal, according to the user manual. This way the register "den" is loaded with 10 in the next clock signal is done a right swift register so the next value is 01 when is performed the reading, in the next clock signal the value is 00 this way the XADC only captures the “den” register high for one clock signal, when performed a reading the XADC capture the address value, the value on this register is where is performed the reading. The same happens in the write operation, when is "rw" equal to two, is performed a write on the XADC, but this time are used two registers, "den" and "dwe" to perform writing, when performed writing, the XADC capture the address and the data in, this way knowing where and what to write.
Figure 52 represents these three blocks.

```
160     always @(posedge clk_i) begin
161         if(CRTL == 1'h1)begin
162             conv = 1'h1;
163             if(enable != 0)begin
164                 conv = 1'h0;
165                 CORTL = 1'h0;
166                 stat = 1'h1;
167                 end
168         end
169     end
170     always @(posedge clk_i) begin
171         if(ready == 1) begin
172             data <= datakeep;
173         end
174    end
175     always@(posedge clk_i) begin
176         case(rw)
177             2'b00: begin
178                 den[0]  <= 0;
179                 dwe[0]  <= 0;
180             end
181             2'b01: begin
182                 if(busy == 0) begin
183                     den <= 2'h2;
184                 end
185
186                 else begin
187                     den <= { 1'b0, den[1] };
188                 end
189             end
190             2'b10: begin
191                 if(busy == 0) begin
192                     den  <= 2'h2;
193                     dwe  <= 2'h2;
194                 end
195                 else begin
196                     den <= {1'b0, den[1]};
197                     dwe <= {1'b0, dwe[1]};
198                 end
199             end
200          endcase
201     end
202
203
204 endmodule
```

Figure 52 - Module logic to use DRP

Chapter 4

Experimental Validation

4.1 Experimental Setup

Integrating the XADC module requires a comprehensive understanding of RISC-V architecture, Wishbone, and hardware, in this case, the NEXYS A7, because these technologies have to work together to not miss anything.

Let's divide the process of coupling the XADC module to RISC-V into 5 steps:

1st Step

When working with hardware is important to read the datasheet from the manufacturer, in this case, is necessary to know the pinout of the FPGA chip. For example, the clock on Xilinx Artix-7 is on pin E3, so connecting to other pins is impossible; if this happens, the hardware won't work, and the same occurs to the XADC. AMD, when creating the FPGA, reserves 8 pins for external reading on XADC; this specification of pins happens in a file called constraints; this file is a bridge between the hardware and the code, and this file allows to give a variable to the pins.

```
#Pmod Header JXADC

set_property -dict { PACKAGE_PIN A14   IOSTANDARD LVCMOS33     } [get_ports { vauxn3  }]; #IO_L9N_T1_DQS_AD3N_15 Sch=xa_n[1]
set_property -dict { PACKAGE_PIN A13   IOSTANDARD LVCMOS33     } [get_ports { vauxp3  }]; #IO_L9P_T1_DQS_AD3P_15 Sch=xa_p[1]
set_property -dict { PACKAGE_PIN A16   IOSTANDARD LVCMOS33     } [get_ports { vauxn10 }]; #IO_L8N_T1_AD10N_15 Sch=xa_n[2]
set_property -dict { PACKAGE_PIN A15   IOSTANDARD LVCMOS33     } [get_ports { vauxp10 }]; #IO_L8P_T1_AD10P_15 Sch=xa_p[2]
set_property -dict { PACKAGE_PIN B17   IOSTANDARD LVCMOS33     } [get_ports { vauxn2  }]; #IO_L7N_T1_AD2N_15 Sch=xa_n[3]
set_property -dict { PACKAGE_PIN B16   IOSTANDARD LVCMOS33     } [get_ports { vauxp2  }]; #IO_L7P_T1_AD2P_15 Sch=xa_p[3]
set_property -dict { PACKAGE_PIN A18   IOSTANDARD LVCMOS33     } [get_ports { vauxn11 }]; #IO_L10N_T1_AD11N_15 Sch=xa_n[4]
set_property -dict { PACKAGE_PIN B18   IOSTANDARD LVCMOS33     } [get_ports { vauxp11 }]; #IO_L10P_T1_AD11P_15 Sch=xa_p[4]
```

Figure 53 - Constraints for XADC external signals

In Figure 53 is possible to see on the left rectangle the pins from the hardware and on the right rectangle the name given to use in the code.

2nd step

Is define on RVfpgaNEXYS if these variables are inputs or outputs, in this case, are inputs. This allows the use of the signal on the architecture.

```
// ADC
input wire vauxn3,
input wire vauxp3,
input wire vauxn10,
input wire vauxp10,
input wire vauxn2,
input wire vauxp2,
input wire vauxn11,
input wire vauxp11);
```

Figure 54 - Inputs in RVfpgaNEXYS

The notation of "wire" is used to the variable be available through all the architecture. The SweRV core is instantiated in this file, so it is also necessary to instantiate these signals in this file, which is shown in Figure 55.

```
.vauxn3 (vauxn3),
.vauxp3 (vauxp3),
.vauxn10 (vauxn10),
.vauxp10 (vauxp10),
.vauxn2 (vauxn2),
.vauxp2 (vauxp2),
.vauxn11 (vauxn11),
.vauxp11 (vauxp11));
```

Figure 55 - Instantiation the signals in RVfpgaNEXYS to SweRV

3rd step

The same way is needed to define on RVfpgaNEXYS, is also needed to do the same on SweRV core. As shown in Figure 56.

```
// ADC
input wire vauxn3,
input wire vauxp3,
input wire vauxn10,
input wire vauxp10,
input wire vauxn2,
input wire vauxp2,
input wire vauxn11,
input wire vauxp11);
```

Figure 56 - Inputs in SweRV

On core is also needed to instantiate the module development of the XADC.

```
CHIPXADC adc1
   (//Wishbone slave interface
    .clk_i      (clk),
    .rst_i      (wb_rst),
    .cyc_i      (wb_m2s_adc_cyc),
    .adr_i      (wb_m2s_adc_adr),
    .dat_i      (wb_m2s_adc_dat),
    .sel_i      (wb_m2s_adc_sel),
    .we_i       (wb_m2s_adc_we),
    .stb_i      (wb_m2s_adc_stb),
    .dat_o      (wb_s2m_adc_dat),
    .ack_o      (wb_s2m_adc_ack),

    //ADC interface
    .vauxn3  (vauxn3),
    .vauxp3  (vauxp3),
    .vauxn10  (vauxn10),
    .vauxp10  (vauxp10),
    .vauxn2  (vauxn2),
    .vauxp2  (vauxp2),
    .vauxn11  (vauxn11),
    .vauxp11  (vauxp11));

 assign wb_s2m_adc_rty = 1'b0;
```

Figure 57 - Instantiating the XADC module on SweRV

Figure 57 shows the instantiation of the XADC module; there are two interfaces: the Wishbone interface that allows the module to communicate with RISC-V architecture, and the external signals to connect to the XADC interface.

4th step

In the Wishbone module, it is necessary to add the Wishbone signal of the XADC module, as shown in Figure 58.

```
// ADC
    output [31:0] wb_adc_adr_o,
    output [31:0] wb_adc_dat_o,
    output  [3:0] wb_adc_sel_o,
    output        wb_adc_we_o,
    output        wb_adc_cyc_o,
    output        wb_adc_stb_o,
    output  [2:0] wb_adc_cti_o,
    output  [1:0] wb_adc_bte_o,
    input  [31:0] wb_adc_dat_i,
    input         wb_adc_ack_i,
    input         wb_adc_err_i,
    input         wb_adc_rty_i);
```

Figure 58 - Wishbone XADC signals

Then, it is necessary to edit the Wishbone multiplexer; this multiplexer allows the peripherals to communicate with the core, and every peripheral from memory to the XADC has an address space to operate.

```
wb_mux
  #(.num_slaves (12),
    .MATCH_ADDR ({32'h00000000, 32'h00001000, 32'h00001040, 32'h00001100, 32'h00001200, 32'h00001240, 32'h00001280, 32'h000012c0, 32'h00001400, 32'h00001800, 32'h00002000, 32'h00001600}),
    .MATCH_MASK ({32'hfffff000, 32'hffffffc0, 32'hffffffc0, 32'hffffffc0, 32'hffffffc0, 32'hffffffc0, 32'hffffffc0, 32'hffffffc0, 32'hffffffc0, 32'hffffffc0, 32'hfffff000, 32'hffffffc0}))
 wb_mux_io
   (.wb_clk_i  (wb_clk_i),
    .wb_rst_i  (wb_rst_i),
    .wbm_adr_i (wb_io_adr_i),
    .wbm_dat_i (wb_io_dat_i),
    .wbm_sel_i (wb_io_sel_i),
    .wbm_we_i  (wb_io_we_i),
    .wbm_cyc_i (wb_io_cyc_i),
    .wbm_stb_i (wb_io_stb_i),
    .wbm_cti_i (wb_io_cti_i),
    .wbm_bte_i (wb_io_bte_i),
    .wbm_dat_o (wb_io_dat_o),
    .wbm_ack_o (wb_io_ack_o),
    .wbm_err_o (wb_io_err_o),
    .wbm_rty_o (wb_io_rty_o),
    .wbs_adr_o ({wb_rom_adr_o, wb_sys_adr_o, wb_spi_flash_adr_o, wb_spi_accel_adr_o, wb_ptc_adr_o, wb_ptc2_adr_o, wb_ptc3_adr_o, wb_ptc4_adr_o, wb_gpio_adr_o, wb_gpio2_adr_o, wb_uart_adr_o, wb_adc_adr_o}),
    .wbs_dat_o ({wb_rom_dat_o, wb_sys_dat_o, wb_spi_flash_dat_o, wb_spi_accel_dat_o, wb_ptc_dat_o, wb_ptc2_dat_o, wb_ptc3_dat_o, wb_ptc4_dat_o, wb_gpio_dat_o, wb_gpio2_dat_o, wb_uart_dat_o, wb_adc_dat_o}),
    .wbs_sel_o ({wb_rom_sel_o, wb_sys_sel_o, wb_spi_flash_sel_o, wb_spi_accel_sel_o, wb_ptc_sel_o, wb_ptc2_sel_o, wb_ptc3_sel_o, wb_ptc4_sel_o, wb_gpio_sel_o, wb_gpio2_sel_o, wb_uart_sel_o, wb_adc_sel_o}),
    .wbs_we_o  ({wb_rom_we_o,  wb_sys_we_o,  wb_spi_flash_we_o,  wb_spi_accel_we_o,  wb_ptc_we_o,  wb_ptc2_we_o,  wb_ptc3_we_o,  wb_ptc4_we_o,  wb_gpio_we_o,  wb_gpio2_we_o,  wb_uart_we_o, wb_adc_we_o }),
    .wbs_cyc_o ({wb_rom_cyc_o, wb_sys_cyc_o, wb_spi_flash_cyc_o, wb_spi_accel_cyc_o, wb_ptc_cyc_o, wb_ptc2_cyc_o, wb_ptc3_cyc_o, wb_ptc4_cyc_o, wb_gpio_cyc_o, wb_gpio2_cyc_o, wb_uart_cyc_o, wb_adc_cyc_o}),
    .wbs_stb_o ({wb_rom_stb_o, wb_sys_stb_o, wb_spi_flash_stb_o, wb_spi_accel_stb_o, wb_ptc_stb_o, wb_ptc2_stb_o, wb_ptc3_stb_o, wb_ptc4_stb_o, wb_gpio_stb_o, wb_gpio2_stb_o, wb_uart_stb_o, wb_adc_stb_o}),
    .wbs_cti_o ({wb_rom_cti_o, wb_sys_cti_o, wb_spi_flash_cti_o, wb_spi_accel_cti_o, wb_ptc_cti_o, wb_ptc2_cti_o, wb_ptc3_cti_o, wb_ptc4_cti_o, wb_gpio_cti_o, wb_gpio2_cti_o, wb_uart_cti_o, wb_adc_cti_o}),
    .wbs_bte_o ({wb_rom_bte_o, wb_sys_bte_o, wb_spi_flash_bte_o, wb_spi_accel_bte_o, wb_ptc_bte_o, wb_ptc2_bte_o, wb_ptc3_bte_o, wb_ptc4_bte_o, wb_gpio_bte_o, wb_gpio2_bte_o, wb_uart_bte_o, wb_adc_bte_o}),
    .wbs_dat_i ({wb_rom_dat_i, wb_sys_dat_i, wb_spi_flash_dat_i, wb_spi_accel_dat_i, wb_ptc_dat_i, wb_ptc2_dat_i, wb_ptc3_dat_i, wb_ptc4_dat_i, wb_gpio_dat_i, wb_gpio2_dat_i, wb_uart_dat_i, wb_adc_dat_i}),
    .wbs_ack_i ({wb_rom_ack_i, wb_sys_ack_i, wb_spi_flash_ack_i, wb_spi_accel_ack_i, wb_ptc_ack_i, wb_ptc2_ack_i, wb_ptc3_ack_i, wb_ptc4_ack_i, wb_gpio_ack_i, wb_gpio2_ack_i, wb_uart_ack_i, wb_adc_ack_i}),
    .wbs_err_i ({wb_rom_err_i, wb_sys_err_i, wb_spi_flash_err_i, wb_spi_accel_err_i, wb_ptc_err_i, wb_ptc2_err_i, wb_ptc3_err_i, wb_ptc4_err_i, wb_gpio_err_i, wb_gpio2_err_i, wb_uart_err_i, wb_adc_err_i}),
    .wbs_rty_i ({wb_rom_rty_i, wb_sys_rty_i, wb_spi_flash_rty_i, wb_spi_accel_rty_i, wb_ptc_rty_i, wb_ptc2_rty_i, wb_ptc3_rty_i, wb_ptc4_rty_i, wb_gpio_rty_i, wb_gpio2_rty_i, wb_uart_rty_i, wb_adc_rty_i}))
```

Figure 59 - Add XADC peripheral in Wishbone

In Figure 59, it is possible to observe that there are 12 peripherals; one was added to the XADC module and has the memory position "32'h00001600", and also the XADC signal; these changes are in the red rectangles.

5th step

The fifth and last step to fully configure the Wishbone bus is to add the file shown in Figure 60. The changes are shown in Figure 61.

Figure 60 - Find "wb_intercon.vh" file

```verilog
// ADC
wire [31:0] wb_m2s_adc_adr;
wire [31:0] wb_m2s_adc_dat;
wire  [3:0] wb_m2s_adc_sel;
wire        wb_m2s_adc_we;
wire        wb_m2s_adc_cyc;
wire        wb_m2s_adc_stb;
wire  [2:0] wb_m2s_adc_cti;
wire  [1:0] wb_m2s_adc_bte;
wire [31:0] wb_s2m_adc_dat;
wire        wb_s2m_adc_ack;
wire        wb_s2m_adc_err;
wire        wb_s2m_adc_rty;
```

```verilog
// ADC
 .wb_adc_adr_o      (wb_m2s_adc_adr),
 .wb_adc_dat_o      (wb_m2s_adc_dat),
 .wb_adc_sel_o      (wb_m2s_adc_sel),
 .wb_adc_we_o       (wb_m2s_adc_we),
 .wb_adc_cyc_o      (wb_m2s_adc_cyc),
 .wb_adc_stb_o      (wb_m2s_adc_stb),
 .wb_adc_cti_o      (wb_m2s_adc_cti),
 .wb_adc_bte_o      (wb_m2s_adc_bte),
 .wb_adc_dat_i      (wb_s2m_adc_dat),
 .wb_adc_ack_i      (wb_s2m_adc_ack),
 .wb_adc_err_i      (wb_s2m_adc_err),
 .wb_adc_rty_i      (wb_s2m_adc_rty));
```

Figure 61 - "wb_intercon.vh" changes

This way, the implementation of the XADC module into RISC-V architecture is completed.

4.2 Operational Validations

To validate this work, it is necessary to create a C program to verify if the Wishbone connection and the module are working properly, it is also necessary to develop a circuit to measure signal from outside of the FPGA.
Before start develop the program, let’s create a project in PlatformIO. To use PlatformIO, it is necessary to install Visual Studio Code and add the PlatformIO extension; after doing these steps successfully, it is possible to create a project.

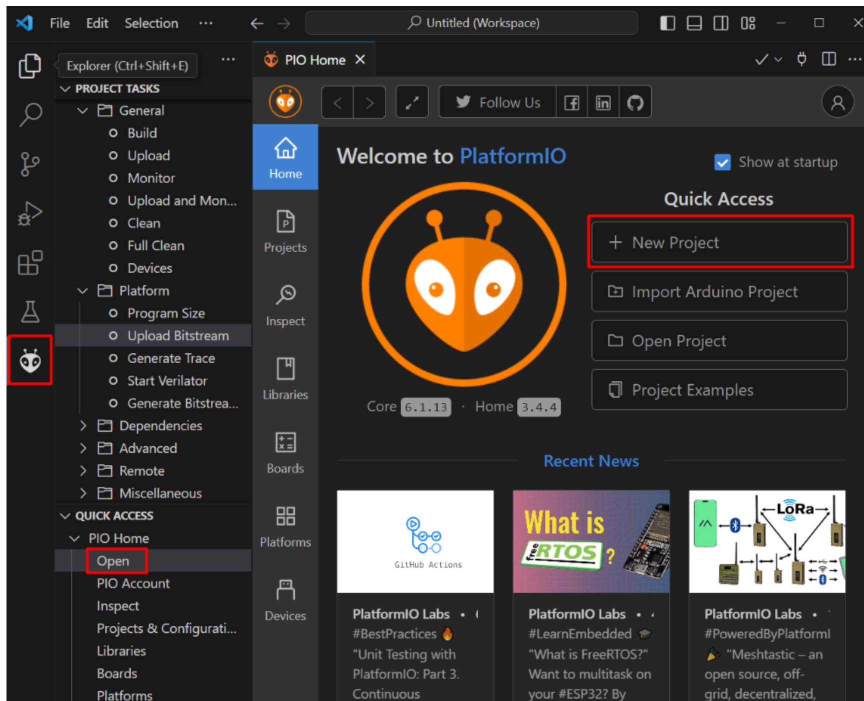

Figure 62 - Open PlatformIO in VSC

After installing the PlatformIO on VSC, a new icon on the left bar will open the PlatformIO by clicking this icon. Figure 62 is the open page of PlatformIO. If this page does not appear, try to click on "Open", which is also represented in the figure. To create a project, click "New Project". After clicking, a new window will appear. In this window, represented in Figure 63, is possible to choose the name of the project, and select the board. In this parameter, it is possible to search for "NEXYS A7" it is essential to guarantee that "RVfpga: Diligent Nexys A7" from CHIPS Alliance is selected; in the framework is selected the "WD-Firmware" from Western Digital, the location is arbitrary.

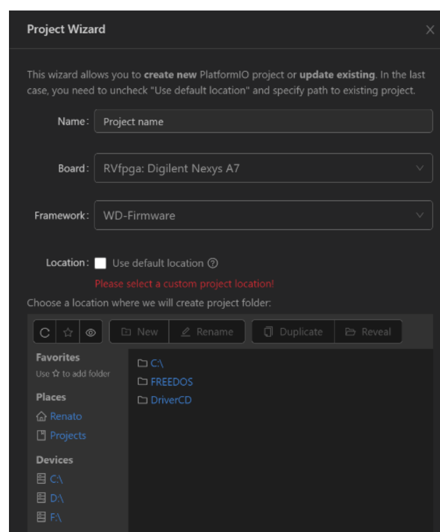

Figure 63 - Create new project

With the project created is necessary to add two lines to the "platformio.ini" file. One is to use the UART, defining the speed of the UART, in this case, was used "monitor_speed = 115200". The other line is the directory of the bitstream generated by Vivado, "board_build.bitstream_file = [directory] ", these two steps are represented in Figure 64.

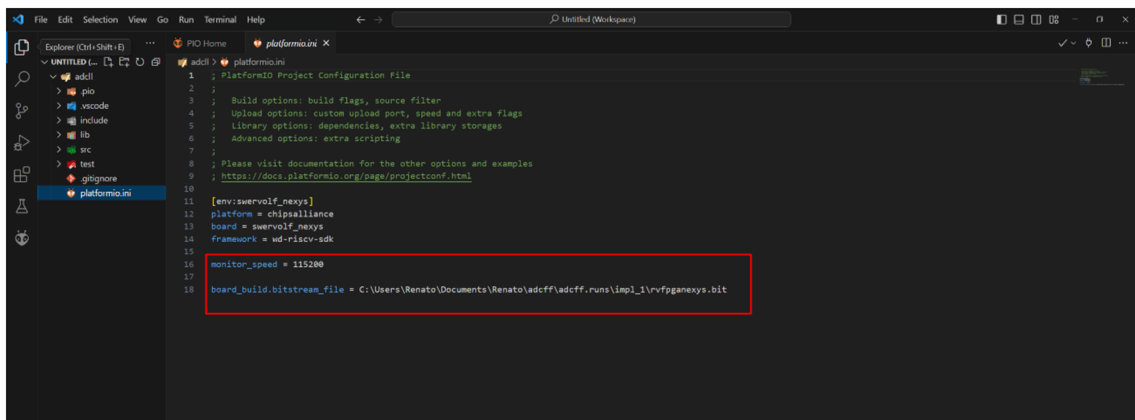

Figure 64 - "platformio.ini" file changes

Now the project is fully configured to run on the board.

It is necessary to start building the C program to test the integration of the module into RISC-V architecture and the module itself.

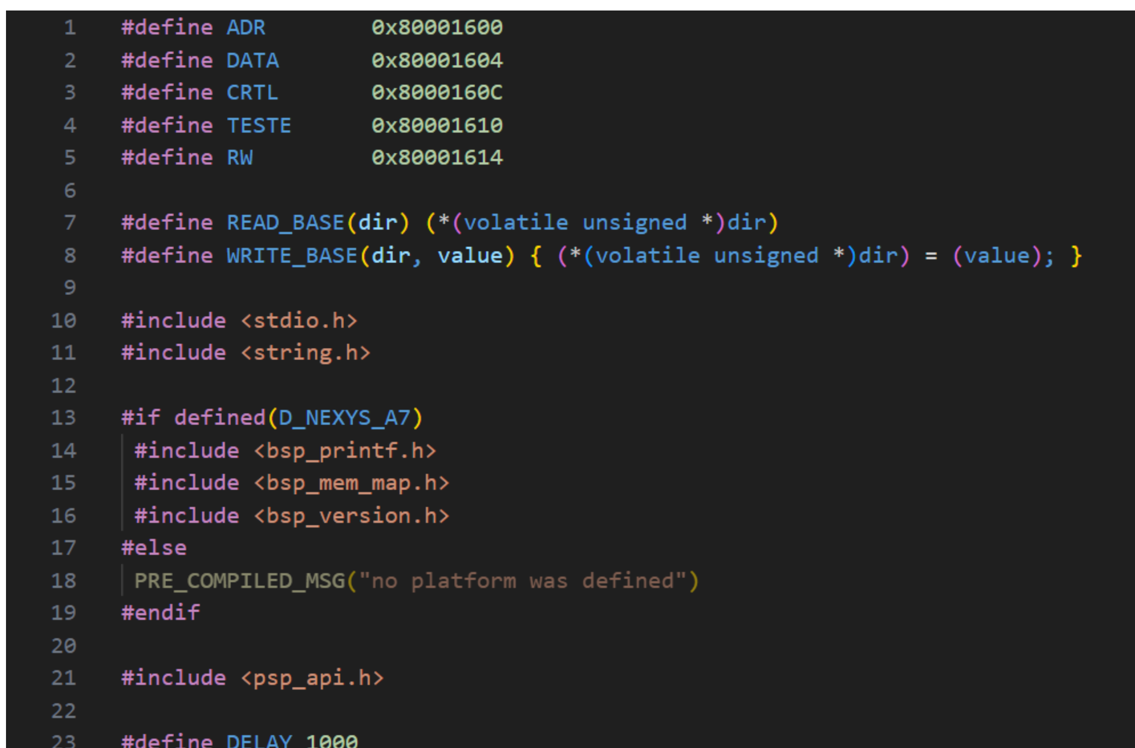
```
1  #define ADR         0x80001600
2  #define DATA        0x80001604
3  #define CRTL        0x8000160C
4  #define TESTE       0x80001610
5  #define RW          0x80001614
6
7  #define READ_BASE(dir) (*(volatile unsigned *)dir)
8  #define WRITE_BASE(dir, value) { (*(volatile unsigned *)dir) = (value); }
9
10 #include <stdio.h>
11 #include <string.h>
12
13 #if defined(D_NEXYS_A7)
14  #include <bsp_printf.h>
15  #include <bsp_mem_map.h>
16  #include <bsp_version.h>
17 #else
18  PRE_COMPILED_MSG("no platform was defined")
19 #endif
20
21 #include <psp_api.h>
22
23 #define DELAY 1000
```
Figure 65 - C code mnemonics and libraries

Figure 65 is the beginning of the C code, where the addresses are defined, functions, and libraries. The addresses have to be same used in the module to be consistent with the development of the module. The functions to read and write on the module are "READ_BASE" and "WRITE_BASE". When wanted to perform a reading, it is only needed the address, and when wanted to perform writing is needed the address, and the value wanted to write; this will be shown through the program. There are also some libraries for the operation of the code and UART; without them, the code won't work.

```
25  int main ( void )
26  {
27  // Initialize UART
28      uartInit();
29      //variables
30      int adr, crtl, adrc, data, adrmem, test, i;
31
32      //Test wishbone reading on module
33      test = READ_BASE(TESTE);
34      printfNexys("Test value: %d ", test);
35
```

Figure 66 - UART initialization, initialization variables and test READ

Figure 66 represents the initialization of the UART at line 28, at line 30 is the initialization of the variables used to operate the code and a test to ensure that Wishbone is operational by reading the address "TESTE" and then sending via UART to guarantee a fixed value.

```
36      //Execute a reading on XADC Event-Driven on ADR 13
37      adr = 0x13;
38      crtl = 0x01;
39
40      WRITE_BASE(ADR,adr);
41      WRITE_BASE(CRTL,crtl);
42
43      adrc = READ_BASE(ADR);
44      printfNexys("Value reading address: %X",adrc);
45
46      adrc = READ_BASE(ADR);
47      WRITE_BASE(RW,0x1);
48      for(i=0;i<DELAY;i++){
49          i=i+1;
50      }
51      data =((READ_BASE(DATA) >> 4)*1000)/4096;
52
53      printfNexys("Results");
54      printfNexys("Value read: %d mV",data);
55      printfNexys("Address of read: %X",adrc);
```

Figure 67 - C code to write and read on XADC module

In Figure 67, the code represented performs a conversion on address 0x13 and a driven-event; RISC-V native doesn't support floating point, so it is necessary to convert the value read on XADC to millivolts and display the reading through UART.

```
57    //Reconfigure the DRP of the XADC
58    WRITE_BASE(ADR,0x40);
59    WRITE_BASE(DATA,0x9000);
60    WRITE_BASE(RW,0x2);
61
62    for(i=0;i<DELAY;i++){
63        i=i+1;
64    }
65
66    WRITE_BASE(RW,0x1)
67
68    for(i=0;i<DELAY;i++){
69        i=i+1;
70    }
71
72    adrmem = READ_BASE(DATA);
73    printfNexys("Value address 40: %X",adrmem);
74
```

Figure 68 - Reconfiguration of DRP

Figure 68 represents a write operation on register 0x40 of the XADC, first is loaded the address to write, then the value to write in this case is 0x9000, and the command to write, sometimes is necessary to give a delay to Wishbone to complete the operations, the Wishbone is a more straightforward interface to use but also is slower compared to the AXI interface. The complete code is in Appendix D.

The last step is to test the code, for this is necessary to mount a circuit to XADC to perform the conversion, this is necessary because the XADC only can perform external readings from 0 to 1 Volt in unipolar mode and from -500 to 500 Millivolts e bipolar mode, the board Nexys A7 have outputs of 3.3 volts, with this in mind is necessary to create a circuit to read different values with this specification. The resulting circuit is represented in Figure 69.

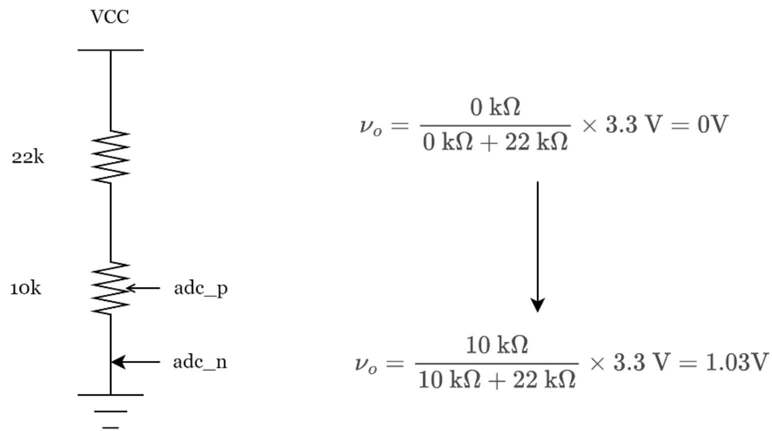

Figure 69 - Circuit to be read

The final step is to upload the bitstream generated by Vivado into the FPGA and run the program. This step is represented in Figure 70 and Figure 71.

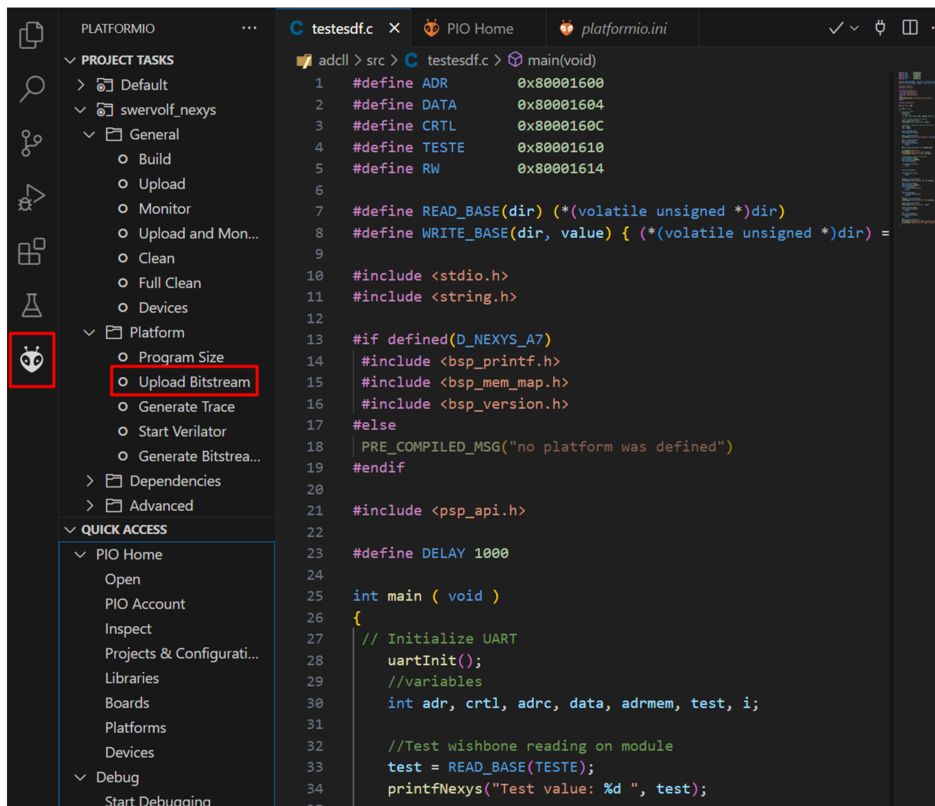

Figure 70 - Upload bitstream button location

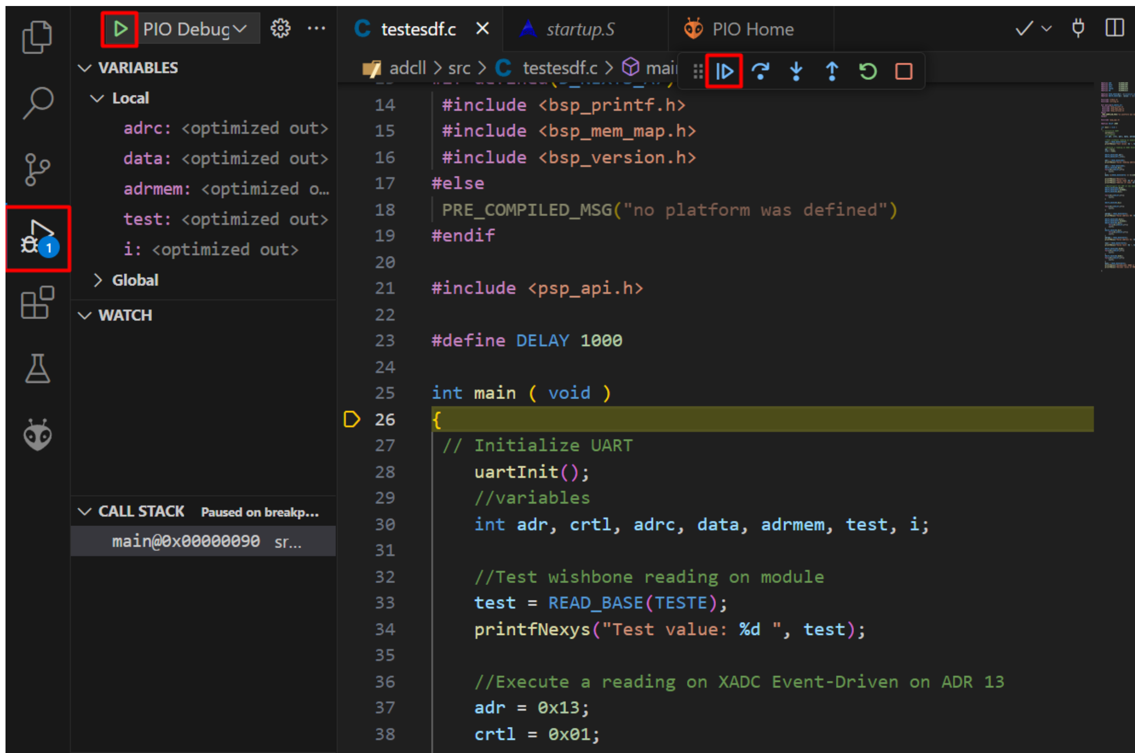

Figure 71 - Run the program

4.3 Results Analysis

With everything together, it is possible to run the program to obtain the results, with a multimeter measuring the tension on the variable resistance to have an outside measurement, to see the results from the whole architecture developed in this work is necessary to open the serial monitor to see the communication through UART, this feature is available by clicking the icon represented in Figure 72. This bar is located at the bottom of VSC.

Figure 72 - Serial monitor button location

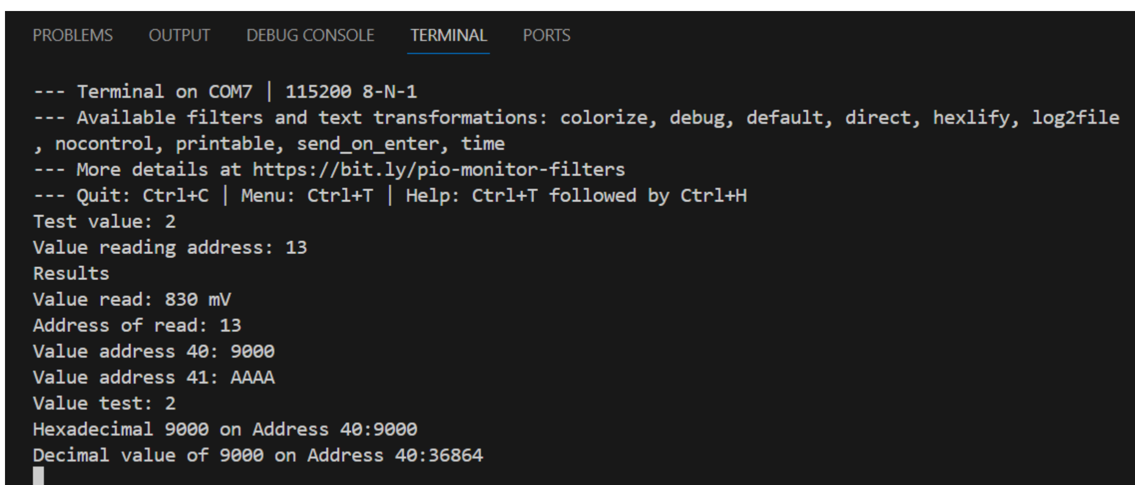

Figure 73 - Results from the project

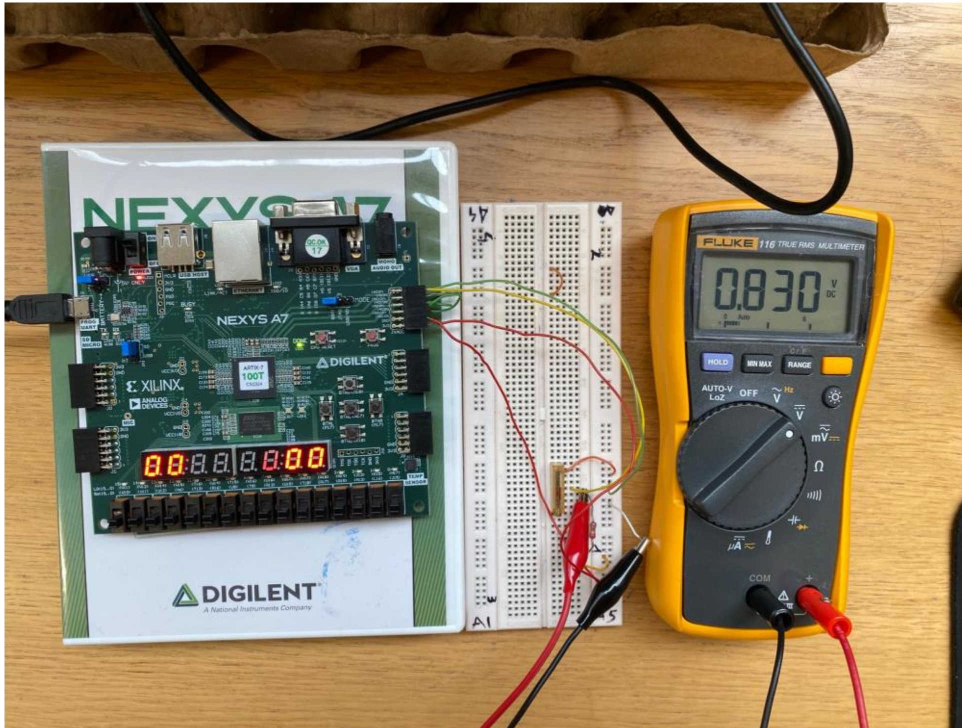

Figure 74 - Multimeter read

Chapter 5

Conclusion and Future Works

At the beginning of this project, several objectives were established. To achieve them, it was necessary to go through a learning process. It started by developing a module that performed the addition of two registers and stored the result in a third. This step was fundamental to understanding the Verilog language and allowed the first integration into the RISC-V architecture through the Wishbone bus. This integration provided a detailed understanding of the operation of the bus and all its signals. Furthermore, the UART peripheral was used, already integrated into the architecture. The ability to use this peripheral is crucial to obtaining feedback on the successful execution of operations. During this phase, we also conducted the first simulations of the architecture. These simulations are essential to observe the system's functioning, identify possible failures, and correct them to ensure successful development.
Completed the project's first objective, which was to integrate a peripheral into the RISC-V architecture. After integrating the first peripheral into the architecture, we focused on an in-depth study of the XADC, exploring its operating modes, status and control registers, timing, and reconfiguration via DRP. This process allowed the creation of an XADC module and a testbench to see its entire operation and not just what is presented in the user manual. From this module, was developed what will be integrated into the RISC-V architecture through the Wishbone bus. This new module makes it possible, through a C program, to generate events to initiate conversions in XADC, read any XADC status or control registers, and even dynamically reconfigure XADC via DRP. This ability to reconfigure at run time, without the need to access the XADC wizard or create a new bitstream with each change, makes XADC an extremely versatile and dynamic tool. This adaptability throughout the program expands its application possibilities. Finally, the integration of XADC into the RISC-V architecture was achieved, providing a complete and flexible solution for monitoring and control needs.
Future work on this project, would be interesting to add a WI-FI module to the Nexys A7 board to communicate with a gateway, thus making it a wireless sensor. Another improvement to this work would be to add the F extension to the RISC-V architecture, this way it would natively have a floating point, expanding its capabilities.

References

[1] N. Sharma, M. Shamkuwar, and I. Singh, “The History, Present and Future with IoT,” in *Internet of Things and Big Data Analytics for Smart Generation*, V. E. Balas, V. K. Solanki, R. Kumar, and M. Khari, Eds., Cham: Springer International Publishing, 2019, pp. 27–51. doi: 10.1007/978-3-030-04203-5_3.

[2] G. C. M. Meijer, G. Wang, and F. Fruett, “Temperature Sensors and Voltage References Implemented in CMOS Technology,” 2001.

[3] S. Rapuano *et al.*, “IEEE Instrumentation & Measurement Magazine Part 6 in a series of tutorials in instrumentation and measurement,” 2005. [Online]. Available: http://grouper.ieee.org/groups/.

[4] Chunzhi. Wang, Zhiwei. Ye, and Hua zhong li gong da xue., *2011 3rd International Workshop on Intelligent Systems and Applications : proceedings : ISA 2011 : 28-29 May 2011, Wuhan, China*. IEEE, 2011.

[5] A. Amara, F. Amiel, and T. Ea, “FPGA vs. ASIC for low power applications,” 2006, doi: 10.1016/j.mejo.2005.11.003.

[6] A. A. Waterman Yunsup Lee David Patterson Krste Asanovic, A. Waterman, Y. Lee, D. Patterson, and K. Asanovi, “The RISC-V Instruction Set Manual, Volume I: Base User-Level ISA The RISC-V Instruction Set Manual Volume I: Base User-Level ISA Version 1.0,” 2011. [Online]. Available: http://www.eecs.berkeley.edu/Pubs/TechRpts/2011/EECS-2011-62.html

[7] S. Harris and D. Harris, “RISC-V Instruction Set Summary,” in *Digital Design and Computer Architecture*, Elsevier, 2022, pp. IBC1–IBC2. doi: 10.1016/b978-0-12-820064-3.00025-8.

[8] S. Greengard, “Will RISC-V revolutionize computing?,” *Commun ACM*, vol. 63, no. 5, pp. 30–32, Apr. 2020, doi: 10.1145/3386377.

[9] G. Gomez-Sanchez *et al.*, “Challenges and Opportunities for RISC-V Architectures towards Genomics-based Workloads,” Jun. 2023, [Online]. Available: http://arxiv.org/abs/2306.15562

[10] A. Pedro Charana Silva, J. João Henriques Teixeira de Sousa, and F. André Corrêa Alegria Supervisor, “Development Environment for a RISC-V Processor Electrical and Computer Engineering Examination Committee,” 2020.

[11] "THE IMAGINATION UNIVERSITY PROGRAMME RVfpga Getting Started Guide RVfpga Getting Started Guide." [Online]. Available: https://www.linkedin.com/in/valeros/

[12] "Wishbone B4 WISHBONE System-on-Chip (SoC)Interconnection Architecturefor Portable IP Cores Brought to You By OpenCores," 2010. [Online]. Available: www.opencores.org

[13] M. Arjun and M. M. Tech, "A Literature Review on Wishbone Bus Technique for Network on Chip Architecture," *International Journal of Innovative Research in Engineering & Management (IJIREM)*, no. 2, p. 5, 2015.

[14] "7 Series FPGAs and Zynq-7000 SoC XADC Dual 12-Bit 1 MSPS Analog-to-Digital Converter User Guide (UG480) • 7 Series FPGAs and Zynq-7000 SoC XADC Dual 12-Bit 1 MSPS Analog-to-Digital Converter User Guide (UG480) • Reader • AMD Adaptive Computing Documentation Portal." Accessed: Feb. 16, 2024. [Online]. Available: https://docs.xilinx.com/r/en-US/ug480_7Series_XADC/7-Series-FPGAs-and-Zynq-7000-SoC-XADC-Dual-12-Bit-1-MSPS-Analog-to-Digital-Converter-User-Guide-UG480

[15] "Vivado Overview." Accessed: Feb. 19, 2024. [Online]. Available: https://www.xilinx.com/products/design-tools/vivado.html

[16] "Documentation for Visual Studio Code." Accessed: Nov. 22, 2023. [Online]. Available: https://code.visualstudio.com/docs

[17] "PlatformIO IDE — PlatformIO latest documentation." Accessed: Feb. 19, 2024. [Online]. Available: https://docs.platformio.org/en/latest/integration/ide/pioide.html

[18] "Icarus Verilog for Windows." Accessed: Nov. 22, 2023. [Online]. Available: https://bleyer.org/icarus/

[19] "GTKWave." Accessed: Nov. 22, 2023. [Online]. Available: https://gtkwave.sourceforge.net/

[20] J. Serrano, "Introduction to FPGA design".

[21] F. Bruno and an O. M. Company. Safari, *FPGA Programming for Beginners*.

[22] "Nexys A7 Reference Manual - Digilent Reference." Accessed: Feb. 19, 2024. [Online]. Available: https://digilent.com/reference/programmable-logic/nexys-a7/reference-manual

Appendix A

```
`timescale 1ns / 1ps

module XADC_test(
   input  wire [15 : 0] di_in,
   input  wire [6 : 0] daddr_in,
   input  wire den_in,
   input  wire dwe_in,
   output wire drdy_out,
   output wire [15 : 0] do_out,
   input  wire dclk_in,
   input  wire reset_in,
   input wire convst_in,
   input  wire vp_in,
   input  wire vn_in,
   output wire user_temp_alarm_out,
   output wire vccint_alarm_out,
   output wire vccaux_alarm_out,
   output wire ot_out,
   output wire [4 : 0] channel_out,
   output wire eoc_out,
   output wire vbram_alarm_out,
   output wire alarm_out,
   output wire eos_out,
   output wire busy_out
   );

xadc_wiz_0 xiladc (
  .di_in(di_in),                        // input wire [15 : 0] di_in
  .daddr_in(daddr_in),                  // input wire [6 : 0] daddr_in
  .den_in(den_in),                      // input wire den_in
  .dwe_in(dwe_in),                      // input wire dwe_in
  .drdy_out(drdy_out),                  // output wire drdy_out
  .do_out(do_out),                      // output wire [15 : 0] do_out
  .dclk_in(dclk_in),                    // input wire dclk_in
  .reset_in(reset_in),                  // input wire reset_in
  .convst_in(convst_in),                // input wire convst_in
  .vp_in(vp_in),                        // input wire vp_in
  .vn_in(vn_in),                        // input wire vn_in
  .user_temp_alarm_out(user_temp_alarm_out),  // output wire
user_temp_alarm_out
  .vccint_alarm_out(vccint_alarm_out),        // output wire vccint_alarm_out
  .vccaux_alarm_out(vccaux_alarm_out),        // output wire vccaux_alarm_out
  .ot_out(ot_out),                      // output wire ot_out
  .channel_out(channel_out),            // output wire [4 : 0] channel_out
  .eoc_out(eoc_out),                    // output wire eoc_out
  .vbram_alarm_out(vbram_alarm_out),    // output wire vbram_alarm_out
  .alarm_out(alarm_out),                // output wire alarm_out
  .eos_out(eos_out),                    // output wire eos_out
  .busy_out(busy_out)                   // output wire busy_out
);
Endmodule
```

Appendix B

```
`timescale 1ns / 1ps

module XADC_test_tb(

    );

    reg [15 : 0] di_in;
    reg [6 : 0] daddr_in;
    reg den_in;
    reg dwe_in;
    wire drdy_out;
    wire [15 : 0] do_out;
    reg dclk_in;
    reg reset_in;
    reg convst_in;
    wire vp_in;
    wire vn_in;
    wire user_temp_alarm_out;
    wire vccint_alarm_out;
    wire vccaux_alarm_out;
    wire ot_out;
    wire [4 : 0] channel_out;
    wire eoc_out;
    wire vbram_alarm_out;
    wire alarm_out;
    wire eos_out;
    wire busy_out;

    initial
    begin
        dclk_in = 1'b0;
        forever #5 dclk_in = ~dclk_in;
    end

    initial
    begin
        convst_in = 0;
        di_in = 4'h0000;
        den_in = 0;
        dwe_in = 0;
        daddr_in = 2'h00;
        reset_in = 1'b1;
        #200 reset_in = 1'b0;
        #450
        daddr_in = 6'b0000000;
        convst_in = 1;
        #10
        convst_in = 0;
        #1360
        daddr_in = 7'b1000000;
        di_in = 16'h8200;
        #50
```

```
    den_in = 1;
    dwe_in = 1;
    #10
    den_in = 0;
    dwe_in = 0;
    #900
    di_in = 16'h0000;
    daddr_in = 7'b0000000;
    convst_in = 1;
    #10
    convst_in = 0;
    #1400
    daddr_in = 7'b1000000;
    #100
    daddr_in = 7'b0000000;
    convst_in = 1;
    #10
    convst_in = 0;
    #2000
    daddr_in = 7'b0000001;
    #10
    convst_in = 1;
    #10
    convst_in = 0;
    #2000
    daddr_in = 7'b0000000;
    convst_in = 1;
    #10
    convst_in = 0;
    #2000
    daddr_in = 7'b0000000;
    convst_in = 1;
    #10
    convst_in = 0;



end

always@(posedge dclk_in)begin
    if(eoc_out == 1)begin
        den_in <= 1;
        #10
        den_in <= 0;
    end
end

XADC_test DUT(
    .di_in(di_in),
    .daddr_in(daddr_in),
    .den_in(den_in),
    .dwe_in(dwe_in),
    .drdy_out(drdy_out),
    .do_out(do_out),
    .dclk_in(dclk_in),
    .reset_in(reset_in),
```

```
    .convst_in(convst_in),
    .vp_in(1'b0),
    .vn_in(1'b0),
    .user_temp_alarm_out(user_temp_alarm_out),
    .vccint_alarm_out(vccint_alarm_out),
    .vccaux_alarm_out(vccaux_alarm_out),
    .ot_out(ot_out),
    .channel_out(channel_out),
    .eoc_out(eoc_out),
    .vbram_alarm_out(vbram_alarm_out),
    .alarm_out(alarm_out),
    .eos_out(eos_out),
    .busy_out(busy_out)
  );

Endmodule
```

Appendix C

```
`timescale 1ns/1ps

module CHIPXADC(
   input   wire       clk_i,     //check
   input   wire       rst_i,     //check
   input   wire       cyc_i,     //check
   input   wire [7:0]  adr_i,     //endereÃƒÂ§o de entrada
   input   wire [31:0] dat_i,     //dados de entrada
   input   wire [3:0]  sel_i,     //check
   input   wire       we_i,      //check
   input   wire       stb_i,     //check
   output  wire [31:0] dat_o,     //dados de saida
   output  reg        ack_o,     //check


   //external
   input vauxn3,
   input vauxp3,
   input vauxn10,
   input vauxp10,
   input vauxn2,
   input vauxp2,
   input vauxn11,
   input vauxp11,
   input vp_in,
   input vn_in);

   localparam ADDR_ADR       = 8'h00;
   localparam ADDR_DATA      = 8'h04;
   localparam ADDR_STATUS    = 8'h08;
   localparam ADDR_CRTL      = 8'h0c;
   localparam ADDR_TESTE     = 8'h10;
   localparam ADDR_RW        = 8'h14;

   reg teste;
   reg [3:0] test = 4'b0010;

   reg [31:0] address_in;

   reg [15:0] data;
   wire [15:0] datakeep;

   reg [15:0] data_in;

   wire enable;
   wire ready;

   reg [7:0] STATUS;
   reg [31:0] stat;
```

```
   reg [31:0] CRTL;
   reg [31:0] CORTL;

   reg conv = 0;
   wire conv_in;
   assign conv_in = conv;

   wire stat_out;
   assign stat_out = stat;

   reg [31:0] data_out;
   assign dat_o = data_out;

   reg [1:0]rw;
   reg [1:0] den;
   reg [1:0] dwe;

   wire busy;


   xadc_wiz_0 XLXI_7 (
       .convst_in(conv_in),          // Convert Start Input
       .daddr_in(address_in),  //DRP    // Address bus for the dynamic reconfiguration
port
       .dclk_in(clk_i),       //DRP    // Clock input for the dynamic reconfiguration port
       .den_in(den[0]),       //DRP    // Enable Signal for the dynamic reconfiguration
port
       .di_in(data_in),       //DRP    // Input data bus for the dynamic reconfiguration
port
       .dwe_in(dwe[0]),  //DRP    // Write Enable for the dynamic reconfiguration port
       .reset_in(0),                  // Reset signal for the System Monitor control logic
       .vauxp2(vauxp2),               // Auxiliary channel 2
       .vauxn2(vauxn2),
       .vauxp3(vauxp3),               // Auxiliary channel 3
       .vauxn3(vauxn3),
       .vauxp10(vauxp10),             // Auxiliary channel 10
       .vauxn10(vauxn10),
       .vauxp11(vauxp11),             // Auxiliary channel 11
       .vauxn11(vauxn11),
       .busy_out(busy),               // ADC Busy signal
       .channel_out(),                // Channel Selection Outputs
       .do_out(datakeep),      //DRP    // Output data bus for dynamic reconfiguration
port
       .drdy_out(ready),       //DRP    // Data ready signal for the dynamic
reconfiguration port
       .eoc_out(enable),              // End of Conversion Signal
       .eos_out(),                    // End of Sequence Signal
       .vccaux_alarm_out(),           // VCCAUX-sensor alarm output
       .vccint_alarm_out(),           //  VCCINT-sensor alarm output
       .alarm_out(),                  // OR'ed output of all the Alarms
       .vp_in(vp_in),                 // Dedicated Analog Input Pair
       .vn_in(vn_in));

   //!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
   //implementaion!!!!!!!!!!!!!!!!!!
   //!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
```

```
// Wishbone interface
        wire wb_acc = cyc_i & stb_i;       // WISHBONE access
        wire wb_wr  = wb_acc & we_i;       // WISHBONE write access

        // ack_o
        always @(posedge clk_i) begin
                if (rst_i)
                        ack_o <= 1'b0;
                else
                        ack_o <= wb_acc & !ack_o;
        end


//dat_in
always @(posedge clk_i)begin:ola
data_out = 32'h0;
  if (rst_i)
    begin
      address_in <= 32'h0;  // set master bit
      //data = 32'h0;
      STATUS = 32'h0;
    end
  else if (wb_wr)
    begin
      if (adr_i == ADDR_ADR)
        address_in <= dat_i; // always set master bit

      if (adr_i == ADDR_DATA)
        data_in <= dat_i;

      if (adr_i == ADDR_CRTL)
        CRTL <= dat_i;

                  if (adr_i == ADDR_STATUS)
        STATUS <= dat_i;

      if (adr_i == ADDR_TESTE)
        teste <= dat_i;

      if (adr_i == ADDR_RW)
        rw = dat_i;
    end

// dat_o
  case(adr_i) // synopsys full_case parallel_case
     ADDR_ADR:    data_out = address_in;
     ADDR_DATA:   data_out = data;
     ADDR_STATUS: data_out = stat;
     ADDR_CRTL:   data_out = CORTL;
     ADDR_TESTE:  data_out = test;
    default:      data_out = 1234;
endcase
end

//!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
```

```
//!!!!!Logic!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
//!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
always @(posedge clk_i) begin
    if(CRTL == 1'h1)begin
        conv = 1'h1;
        if(enable != 0)begin
            conv = 1'h0;
            CORTL = 1'h0;
            stat = 1'h1;
            end
    end
end
always @(posedge clk_i) begin
    if(ready == 1) begin
        data <= datakeep;
    end
end
always@(posedge clk_i) begin
    case(rw)
        2'b00: begin
            den[0]  <= 0;
            dwe[0]  <= 0;
        end
        2'b01: begin
            if(busy == 0) begin
                den <= 2'h2;
            end

            else begin
                den <= { 1'b0, den[1] };
            end
        end
        2'b10: begin
            if(busy == 0) begin
                den  <= 2'h2;
                dwe  <= 2'h2;
            end
            else begin
                den <= {1'b0, den[1]};
                dwe <= {1'b0, dwe[1]};
            end
        end
      endcase
end


endmodule
```

Appendix D

```
#define ADR       0x80001600
#define DATA      0x80001604
#define CRTL      0x8000160C
#define TESTE     0x80001610
#define RW        0x80001614

#define READ_BASE(dir) (*(volatile unsigned *)dir)
#define WRITE_BASE(dir, value) { (*(volatile unsigned *)dir) = (value); }

#include <stdio.h>
#include <string.h>

#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 1000

int main ( void )
{
 // Initialize UART
   uartInit();
   //variables
   int adr, crtl, adrc, data, adrmem, test, i;

   //Test wishbone reading on module
   test = READ_BASE(TESTE);
   printfNexys("Test value: %d ", test);

   //Execute a reading on XADC Event-Driven on ADR 13
   adr = 0x13;
   crtl = 0x01;

   WRITE_BASE(ADR,adr);
   WRITE_BASE(CRTL,crtl);

   adrc = READ_BASE(ADR);
   printfNexys("Value reading address: %X",adrc);

   adrc = READ_BASE(ADR);
   WRITE_BASE(RW,0x1);
   for(i=0;i<DELAY;i++){
      i=i+1;
   }
   data =((READ_BASE(DATA) >> 4)*1000)/4096;
```

```
    printfNexys("Results");
    printfNexys("Value read: %d mV",data);
    printfNexys("Address of read: %X",adrc);

    //Reconfigure the DRP of the XADC
    WRITE_BASE(ADR,0x40);
    WRITE_BASE(DATA,0x9000);
    WRITE_BASE(RW,0x2);

    for(i=0;i<DELAY;i++){
      i=i+1;
    }

    WRITE_BASE(RW,0x1)

    for(i=0;i<DELAY;i++){
      i=i+1;
    }

    adrmem = READ_BASE(DATA);
    printfNexys("Value address 40: %X",adrmem);

    WRITE_BASE(ADR,0x41);
    WRITE_BASE(DATA,0xAAAA);
    WRITE_BASE(RW,0x2);
      for(i=0;i<DELAY;i++){
      i=i+1;
    }
    WRITE_BASE(RW,0x1)
      for(i=0;i<DELAY;i++){
      i=i+1;
    }
    adrmem = READ_BASE(DATA);
    printfNexys("Value address 41: %X",adrmem);

    test = READ_BASE(TESTE);
    printfNexys("Value test: %d ", test);

    WRITE_BASE(ADR,0x40);
    for(i=0;i<DELAY;i++){
      i=i+1;
    }
    WRITE_BASE(RW,0x01);
    for(i=0;i<DELAY;i++){
      i=i+1;
    }
    data = READ_BASE(DATA);
    printfNexys("Hexadecimal 9000 on Address 40:%X", data);
    printfNexys("Decimal value of 9000 on Address 40:%d", data);

}
```