EKKO: an open-source RISC-V soft-core microcontroller

Diogo Dias

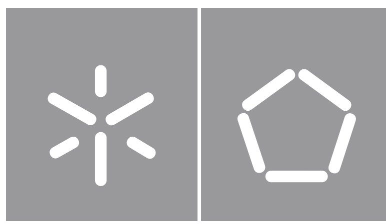
UMinho | 2022

Universidade do Minho
Escola de Engenharia

Diogo Henrique Rosado de Oliveira da Costa Dias

EKKO: an open-source RISC-V soft-core microcontroller

março de 2022



Universidade do Minho
Escola de Engenharia

Diogo Henrique Rosado de Oliveira da Costa Dias

EKKO: an open-source RISC-V soft-core microcontroller

Dissertação de Mestrado
Engenharia Eletrónica Industrial e Computadores
Sistemas Embebidos e Computadores

Trabalho efetuado sob a orientação do
Professor Doutor Jorge Cabral
Professor Doutor Rui Machado

março de 2022

DIREITOS DE AUTOR E CONDIÇÕES DE UTILIZAÇÃO DO TRABALHO POR TERCEIROS

Este é um trabalho académico que pode ser utilizado por terceiros desde que respeitadas as regras e boas práticas internacionalmente aceites, no que concerne aos direitos de autor e direitos conexos.

Assim, o presente trabalho pode ser utilizado nos termos previstos na licença abaixo indicada.

Caso o utilizador necessite de permissão para poder fazer um uso do trabalho em condições não previstas no licenciamento indicado, deverá contactar o autor, através do RepositóriUM da Universidade do Minho.

Licença concedida aos utilizadores deste trabalho

Atribuição-NãoComercial
CC BY-NC

https://creativecommons.org/licenses/by-nc/4.0

Acknowledgements

First, I would like to thank Professor Jorge Cabral and especially Professor Rui Machado for their guidance, help and transmission of knowledge during this entire year.

Thanks to my family and friends for always supporting me. To my girlfriend, Joana, for all her patience, friendship, and affection in the most challenging moments.

Lastly, thanks to my colleagues, specifically those who suffered with me in the 4th year ESRG classes.

STATEMENT OF INTEGRITY

I hereby declare having conducted this academic work with integrity. I confirm that I have not used plagiarism or any form of undue use of information or falsification of results along the process leading to its elaboration.

I further declare that I have fully acknowledged the Code of Ethical Conduct of the University of Minho.

Resumo

Com o surgimento da Internet das Coisas (IoT em inglês) nos últimos anos, o número de "coisas" conectadas está a crescer a um ritmo bastante rápido. Estes dispositivos tornaram-se rapidamente parte do nosso dia a dia e já podem ser encontrados nos mais diversos domínios de aplicação, tais como, telecomunicações, saúde, agricultura, e automação industrial. Devido a este crescimento exponencial, a demanda por sistemas embebidos é cada vez maior, trazendo assim diversos desafios no seu desenvolvimento. De todos os desafios, o *time-to-market* e os custos de desenvolvimento são de inegável importância, logo, a escolha de uma plataforma de desenvolvimento adequada é essencial no desenho destes sistemas.

Devido a este novo paradigma, o grupo de investigação da Universidade do Minho onde esta dissertação se insere tem desenvolvido aplicações neste domínio. No entanto, as atuais plataformas de desenvolvimento utilizadas são complexas, têm custos associados e são de código fechado. Por estas razões, o grupo de investigação tem interesse em ter a sua própria plataforma de desenvolvimento.

De modo a solucionar os problemas enumerados acima, esta dissertação tem como objetivo desenvolver uma plataforma de desenvolvimento tanto para *hardware* como para *software*. A plataforma deve ser simples de utilizar e *open-source*, reduzindo assim os custos e a tornando a gestão de licenças mais simples. Para além disto, o facto de o sistema ser de código aberto faz também com que este possa ser facilmente estendido e customizado de acordo com os requisitos da aplicação.

Neste sentido, esta dissertação apresenta um *soft-core microcontroller*, o qual contem um processador RISC-V, uma *RAM*, uma unidade de depuração, um temporizador, um periférico *I2C* e um barramento *AXI*. Em adição, este contem também um *kit* de desenvolvimento de *software* (SDK em inglês), o qual inclui um depurador, a opção de utilizar o sistema operativo *Azure RTOS ThreadX*, e outras ferramentas importantes, tornando o ciclo de desenvolvimento mais fácil, rápido e seguro.

keywords: *IoT, FPGA, RISC-V, ThreadX*

Abstract

With the advent of the Internet of Things (IoT) in most recent years, the number of connected "things" is increasing quickly. These devices rapidly became part of our daily lives and can be found in the most different applications domains, such as telecommunications, health care, agriculture and industrial automation. With this exponential growth, the demand for embedded devices is increasing, bringing several challenges to the development of these systems. From these challenges, the time-to-market and development costs are undeniable extremely important. Thus, choosing a suitable development platform is essential when designing an embedded system.

Due to this new paradigm, the University of Minho research group where this dissertation fits has been developing applications in this domain. However, the current development platforms are complex, have associated costs and are closed-source. For these reasons, the research group has interesting in having its development platform.

To solve these problems, this dissertation aims to build a development platform for both hardware and software. The platform must be simple and open-source, reducing development costs and simplifying license management. Besides, due to its open nature, it will also be easier to extend and modify the system according to the application's needs.

In this context, this dissertation presents EKKO, an open-source soft-core microcontroller that contains a RISC-V core, an on-chip RAM, a debug unit, a timer and an I2C peripheral, and an AXI bus. In addition, it also contains a Software Development Kit (SDK), which includes a debugger, the option to use Azure RTOS ThreadX, and other crucial tools, turning the development cycle more accessible, faster and safer.

keywords: IoT, FPGA, RISC-V, ThreadX

Contents

Resumo v

Abstract vi

1 Introduction 1

1.1 Contextualisation 1

1.2 Motivation 3

1.3 Objectives 4

1.4 Dissertation structure 5

2 State of the art 7

2.1 Embedded systems processors 7

2.1.1 Types of processors 7

2.1.2 Instruction Set Architectures 10

2.2 Operating Systems 18

2.2.1 Real-time Operating Systems 18

2.3 Development platforms 25

2.3.1 Software 25

2.3.2 Hardware 26

2.3.3 Xilinx Arty A7-100 27

3 RISC-V core 30

3.1 RISC-V ISA 30

3.1.1 RV32I Base Integer Instruction Set 30

3.1.2 Extensions 31

3.2 RISC-V implementations 33

4 Soft-core microcontroller 36

4.1 Architecture overview 36

4.2 Ibex 37

4.3 Debug unit 39
4.4 Peripherals 40
4.4.1 I2C 40
4.4.2 Timer 44
4.5 Advanced eXtensible Interface 45
4.6 Memory map 46

5 Software Development Kit 47
5.1 Support files 47
5.1.1 Makefile 47
5.1.2 Linker 49
5.1.3 OpenOCD 50
5.2 Board Support Package 51
5.2.1 C Runtime Initialization 51
5.2.2 Device drivers 53

6 Azure RTOS ThreadX 58
6.1 ThreadX overview 58
6.1.1 Execution overview 58
6.1.2 Initialization 59
6.1.3 Thread execution 59
6.2 ThreadX port 61

7 Tests and results 65
7.1 Unit tests 65
7.1.1 Ibex 65
7.1.2 AXI 67
7.1.3 Timer 67
7.1.4 I2C 68
7.1.5 ThreadX port 69
7.2 System test 70

8 Conclusion 73
8.1 Future work 73

A Support files **81**

A.1 Makefile 81

A.2 Linker 82

A.3 VSCode debug script 84

B Board Support Package **85**

B.1 C Runtime Initialization 87

B.2 Device drivers 88

B.2.1 I2C 88

B.2.2 Timer 90

C ThreadX **93**

C.1 Port files 93

D Tests **98**

D.1 Ibex tests 98

D.2 System test 106

List of Figures

1.1 Estimation of the IoT market. . . . 1
1.2 IoT's general architecture. . . . 3

2.1 Flexibility versus performance and consumption efficiency in the different types of processors. . . . 8
2.2 Overview of a single-purpose processor architecture. . . . 8
2.3 Overview of an application-specific processor architecture. . . . 9
2.4 Overview of a general-purpose processor architecture. . . . 10
2.5 Register addressing mode. . . . 12
2.6 Immediate addressing mode. . . . 12
2.7 PC-relative addressing mode. . . . 12
2.8 Pseudo-direct addressing mode. . . . 12
2.9 Base addressing mode. . . . 13
2.10 ARM addressing modes. . . . 16
2.11 VxWorks' architecture. . . . 19
2.12 QNX microkernel's architecture. . . . 20
2.13 FreeRTOS' architecture. . . . 21
2.14 LynxOS' architecture. . . . 22
2.15 Azure RTOS ThreadX's core architecture. . . . 22
2.16 RT-Thread's architecture. . . . 23
2.17 RIOT's architecture. . . . 24
2.18 Xilinx Arty A7-100 development board. . . . 28
2.19 CLB arrangement and constitution. . . . 29

3.1 RISC-V base instruction formats. . . . 31

4.1 EKKO architecture overview. . . . 36
4.2 Ibex pipeline architecture. . . . 37
4.3 Debug flow overview. . . . 39

4.4 Debug unit block diagram. 40
4.5 I2C connection diagram. 41
4.6 Representation of the I2C protocol. 41
4.7 I2C module architecture. 42
4.8 I2C control unit finite state machine. 43
4.9 Timer module architecture. 44
4.10 Timer control unit finite state machine. 45
4.11 EKKO memory map. 46

5.1 Compilation process. 48
5.2 OpenOCD block diagram. 50

6.1 ThreadX execution overview. 58
6.2 ThreadX initialization. 59
6.3 ThreadX thread state transition. 60
6.4 ThreadX directory layout. 61

7.1 ALU test simulation. 66
7.2 AXI test simulation. 67
7.3 Timer test in the oscilloscope. 68
7.4 Tests setup. 68
7.5 I2C protocol sniffed by the oscilloscope. 69
7.6 TraceX sequential view. 70
7.7 System test debugging. 72

List of Tables

2.1 Summary of several ISAs’ information. 18
2.2 Summary of several RTOS’ information. 24
3.1 RISC-V standard extensions. 32
3.2 Summary of several RISC-V soft-cores’ features. 35
4.1 I2C configuration registers description. 43
4.2 Timer configuration registers description. 45

Code Snippets

4.1 Ibex instantiation code. 38
5.1 Makefile. 48
5.2 Makefile. 48
5.3 Makefile. 49
5.4 Linker script "MEMORY" and "SECTIONS" commands. 49
5.5 OpenOCD Ibex configuration file. 51
5.6 crt0 file. 52
5.7 crt0 file. 52
5.8 crt0 file. 53
5.9 Timer structure 53
5.10 Timer driver header file. 54
5.11 Timer driver source file. 54
5.12 Timer driver source file. 55
5.13 I2C structure. 55
5.14 I2C Initialize function. 56
5.15 I2C Transmit function. 56
5.16 I2C Receive function. 57
6.1 Assembly code example for the RISC-V IAR compiler. 62
6.2 Assembly code example for the RISC-V GNU compiler. 62
6.3 tx_port.h file. 63
6.4 tx_initialize_low_level.S file. 63
7.1 ALU assembly test file. 65
7.2 AXI test file. 67
7.3 I2C test file. 69
7.4 System test thread0. 70
7.5 System test thread1. 71
A.1 makefile. 81
A.2 linker.ld. 82

A.3 launch.json 84
B.1 bsp.h 85
B.2 bsp.c 86
B.3 crt0.S 87
B.4 i2c.h 88
B.5 i2c.c 89
B.6 timer.h 90
B.7 timer.c 91
C.1 tx_port.h 93
C.2 tx_initialize_low_level.S 96
D.1 test_alu.S 98
D.2 test_lui_auipc.S 100
D.3 test_jal.S 101
D.4 test_jmps.S 102
D.5 test_load_store 104
D.6 main.c 106

List of Abbreviations

ABI	Application Binary Interface.
ACK	acknowledgement.
AI	Artificial intelligence.
ALU	Arithmetic Logic Unit.
AMBA	Advanced Microcontroller Bus Architecture.
AMS	Analog Mixed Signal.
ASIC	Application-Specific Integrated Circuit.
ASIP	Application Specific Instruction-set Processor.
AXI	Advanced eXtensible Interface.
BSP	Board Support Package.
CIoT	Cellular IoT.
CISC	Complex Instruction Set Computer.
CLB	Configurable Logic Block.
CPU	Central Processing Unit.
DSP	Digital Signal Processor.
EA	Effective Address.
EIoT	Enterprise IoT.
EPIC	Explicitly Parallel Instruction Computing.
ESL	Electronic System Level.
FPGA	Field-Programmable Gate Array.
GPP	General-Purpose Processor.
GUI	Graphical User Interface.
HDL	Hardware Description Language.
HPS	Hard Processor System.

I2C	Inter-Integrated Circuit.
IDE	Integrated Development Environment.
IIoT	Industrial IoT.
IoE	Internet of Everything.
IoMT	Internet of Medical Things.
IoT	Internet of Things.
IP	Intellectual Property.
IPC	Instructions Per Cycle.
IPC	Inter Process Communication.
ISA	Instruction Set Architecture.
ISR	Interrupt Service Routine.
LE	Logic Element.
LUT	Look-up Table.
MIPS	Microprocessor without Interlocked Pipelined Stages.
MISC	Minimal Instruction Set Computer.
MMU	Memory Management Unit.
NRE	Non-Recurring Engineering.
OpenOCD	Open On-Chip Debugger.
OS	Operating System.
PC	Personal Computer.
PC	Program Counter.
PMP	Physical Memory Protection.
RAM	Random Access Memory.
RISC	Reduced Instruction Set Computer.
RTL	Register Transfer Level.
RTOS	Real-time Operating System.
SDK	Software Development Kit.
SoC	System on Chip.
SPARC	Scalable Processor Architecture.

TAP	Test Access Port.
UART	Universal Asynchronous Receiver-Transmitter.
VLIW	Very Long Instruction Word.

Chapter 1: Introduction

This chapter presents the contextualisation, motivation, objectives, and structure of this dissertation. The first two explain the reasons behind the elaboration of this work and some contextualisation. The objectives describe the goals that should be met and how to accomplish them. Lastly, the dissertation structure aims to guide the reader throughout the whole document.

1.1 Contextualisation

The Internet of Things (IoT) is the extension of the internet connectivity into physical devices and everyday objects which enables these “things” to connect and exchange data [1]. This concept was first introduced in 1999 by Kevin Ashton [2], and, with its evolution, there will be much better synergy between many different fields of knowledge, such as telecommunications, e-health, informatics, electronics and social science [3, 4]. According to Xia et al. [5], in the next couple of years, the IoT “will increase the ubiquity of the Internet by integrating every object for interaction via embedded systems, which leads to a highly distributed network of devices communicating with human beings as well as other devices”. This will undoubtedly benefit the everyday life of its users in both working and domestic fields. Furthermore, as shown in Figure 1.1, due to the exponential growth from the previous years, it is expected that 75 billion smart devices will be deployed by 2025 [6].

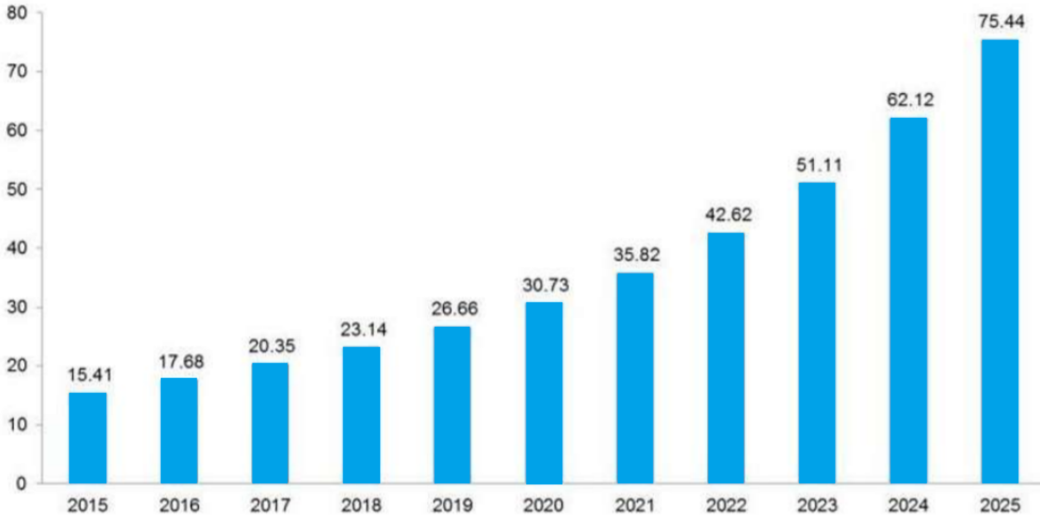


Figure 1.1: Estimation of the IoT market (adapted from [6]).

Another important topic when talking about the IoT is its fundamental characteristics. As mentioned by Patel et al. [7], these characteristics can be divided as follows:

- Interconnectivity: Concerning the IoT, anything can be interconnected, with the global information and communication infrastructure.
- Heterogeneity: Devices in IoT are based on different hardware platforms and networks and can interact with other devices or service platforms through different networks.
- Dynamic nature: The state of devices change dynamically (between connected and/or disconnected) as well as its context: speed, temperature and location. In addition to the state of the device, the number of devices also changes dynamically with a person, place and time.
- Things-related services: The IoT is capable of providing things-related service within constrains (e.g. privacy protection and consistency between physical things and their associated virtual things).
- Connectivity: This enables network accessibility and compatibility. With this connectivity, new market opportunities for the Internet of Things can be created by the networking of smart things and applications.
- Enormous scale: The number of devices that need to be handled and that communicate with each other will be at least an order of magnitude larger than the devices connected to the current Internet. Thus, the management of all data generated and their interpretation for applications purposes becomes crucial.
- Security: IoT devices are naturally vulnerable to security threats. Therefore, it is important to secure the endpoints, the networks, and the data moving across all of it means creating a security paradigm that will scale.

Nowadays, IoT has already achieved such a dimension that is now included in a larger concept, the Internet of Everything (IoE). The IoT's applications not only include everyday objects but also many different areas. As a result, it has started being associated with specific fields, such as Internet of Medical Things (IoMT), Cellular IoT (CIoT), Industrial IoT (IIoT), Enterprise IoT (EIoT), among others.

The IoT doesn't have a standard consensus architecture which is universally defined. However, some few ideas have been proposed based on the selected technology, business needs and technical requirements [8]. For example, Jia et al. [9] and Domingo [10] suggested a three-layer architecture: perception

layer, network layer and service layer, whereas, [11] proposed one with five different layers: sensing, accessing, networking, middleware, and application layers. Nonetheless, the key building blocks on which the IoT is built can be broadly classified into a four-layer architecture: (i) Sensing Layer, (ii) Connection Layer, (iii) Middleware Layer and (iv) Application Layer (see Figure 1.2).

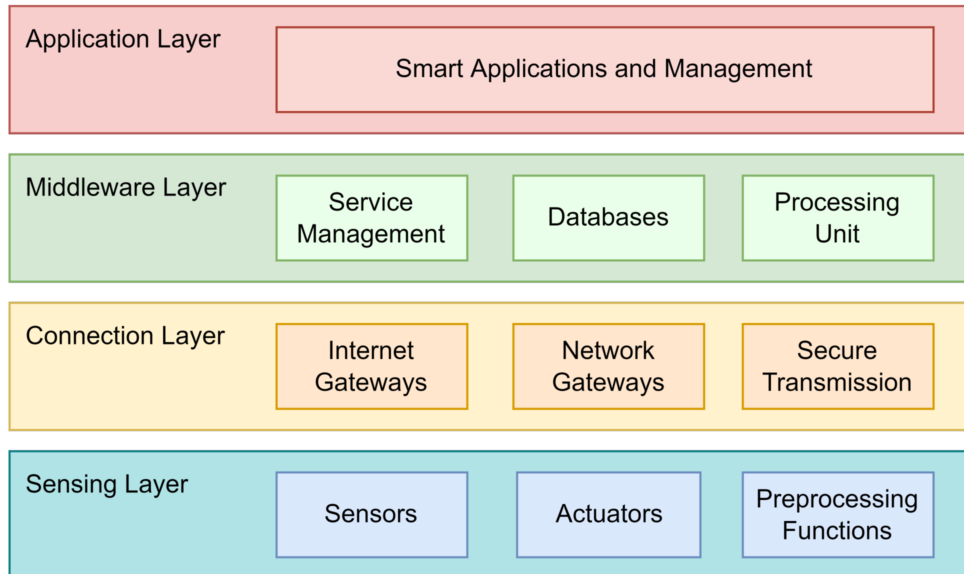


Figure 1.2: IoT's general architecture.

The first layer, where physical objects are integrated with sensors and actuators, is responsible for the data acquisition of its surroundings. These objects can be connected via wire or wireless and they allow the communication between the physical and the digital worlds. Depending on the type of sensors, they may be able to take measurements such as temperature, vibration, humidity, pressure, air quality, etc. Afterwards, the collected data is passed to the Connection Layer, whose purpose is the secure transfer of this gathered information to the Middleware layer. The transmission technology can be 5G, Ethernet, Wi-fi, Bluetooth, ZigBee, depending on the sensor devices. The Middleware layer is responsible for the service management and for storing the data acquired from each physical object into databases. Besides, within the processing unit, some information is processed and decisions are made in accordance. Ultimately, the last layer provides an interface between the user and the applications, which cover diverse sectors such as smart cities, healthcare, tourism and agriculture.

1.2 Motivation

Nowadays, modern society is very dependent on electronic devices for its daily life. From the most diverse application domains, such as the automotive industry, medicine and automation, to basic daily

tasks, the necessity for technological solutions is higher than ever. At the heart of such systems is software and hardware components working seamlessly together to perform a narrow range of functions with the physical world, denominated embedded systems. Due to this high demand, reinforced with the Internet of Things (IoT) urge in most recent years, embedded systems development has become complex, competitive and challenging.

There are several design metrics to consider when developing an embedded system. These include performance, timing, power and cost. To meet these requirements, along with the current time-to-market pressure, it is crucial to choose the right development tools (e.g. compiler, editor, debugger) for building an embedded product. Furthermore, according to the system's complexity, it may also be beneficial to include an operating system. Despite the deleterious effect on the systems' performance, when compared to bare-metal, the insertion of an operating system facilitates the development process in high complexity applications. Therefore, to reduce these performance costs, Real-time Operating System (RTOS) are commonly used in embedded applications.

As a result of this current reality, the University of Minho research group where this dissertation fits is interested in developing a platform for hardware and software components in the embedded systems field. Thus, this dissertation aims to develop a soft-core microcontroller and have a robust and reliable Software Development Kit (SDK) for it. The microcontroller will be implemented in Field-Programmable Gate Array (FPGA), and it needs to include a debugger, a standard system bus for easier peripheral integration, and an open-source Instruction Set Architecture (ISA) for more customisation and cost reduction. Moreover, the SDK should contain all the software development tools, a debugger (preferably with a graphical interface) and the option to include an RTOS.

1.3 Objectives

The objective of the present dissertation is to develop an open-source soft-core microcontroller and have a solid development environment for it. To achieve this goal, the following objectives must be met:

- **Study and choice of the microcontroller's instruction set**

  The first goal is the decision regarding the ISA of the microcontroller's processor. To do so, a study and a comparison between several different architectures will be made. The final choice will be based on the ISA's open nature so that users can change the instruction set for specific applications and do not pay royalties for its use.

- **Study and choice of the ISA's microarchitecture**

Since the processor's ISA must be open-source, a broad spectrum of microarchitecture implementations will be analysed. According to the documentation and FPGA and debug support, an adequate soft-core will be picked from the options.

- **Study and choice of the RTOS**

  Real-time operating systems make the development process simple, fast and secure. Therefore, the option to use an RTOS should be available. Similarly to the first objective, from the different RTOS compared, its decision should focus on its features and unrestricted license.

- **Microcontroller development**

  After the design decision are made, the next goal will be to develop the microcontroller on the FPGA. The system should include the processor core, a debug unit, a memory, and a standard system bus along with some peripherals.

- **SDK development**

  A software development kit will be created to program and debug the developed microcontroller. This kit should include the Board Support Package (BSP), loading and debugging tools and the optional operating system.

- **RTOS port**

  At last, since RTOS are very architecture-specific, the port and validation of the chosen real-time operating system will have to be done to the microcontroller's core ISA.

1.4 Dissertation structure

After this introductory chapter, the state of the art of the most relevant topics of this dissertation are presented. These include embedded systems processors, operating systems, development platforms and some decisions concerning these themes.

The third chapter gives an overview of the RISC-V architecture, mainly the RV32I Base Integer Instruction Set and the Integer Multiplication and Division extensions. Besides, a decision between different RISC-V soft-core implementations is also given.

The following two chapters describe the soft-core microcontroller and its SDK, respectively. The former contains all the system's hardware details, from the core to the peripherals. The latter explains needed software tools and how the BSP was implemented.

Chapter six concerns the chosen RTOS. It starts by describing how it works, followed by a detailed explanation of how it was ported to the system's ISA and how it was validated.

Ultimately, the last two chapters present the tests and results of the developed system, the conclusions and some future work of this dissertation.

Chapter 2: State of the art

This chapter presents a study and a comparison between different instruction set architectures, real-time operating systems and developing platforms. Besides, some major decisions concerning these themes and how they are relevant to this work will also be discussed.

2.1 Embedded systems processors

A processor, also known as Central Processing Unit (CPU), is the electronic circuitry that executes and processes the basic instructions that drive a computer. The CPU is responsible for interpreting most of the computer commands as well as basic arithmetic, logic, controlling and input/output (I/O) operations.

Embedded systems are hardware and software components working together to perform a narrow range of functions with no or minimal user intervention. These are used in everyday life and can be found in the most diverse areas such as medical applications, automotive, smartphones and video game consoles. Therefore, due to the high demand, embedded systems must be produced in large quantities, which undeniably makes the reduction of its production costs a significant concern. Besides, embedded systems often have significant energy, area and performance constraints as well [12, 13]. Unlike desktop PCs (Personal Computers) or server applications, embedded systems processors are quite broad and most architectures come in a large number of different variants and shapes. For all these reasons, in the next subsections, one will be looking into the different types of processors as well as its instruction set architectures.

2.1.1 Types of processors

Processors can be divided into three different categories: (i) General-purpose, (ii) Application-specific and (iii) Single-purpose processors. All of these have specific advantages and disadvantages, which will be presented in the following pages. Single-purpose processors are not intended for general-purpose use due to their lack of flexibility. However, when compared to the other processors of the spectrum, these are the ones with the best performance and consumption efficiency. Similarly, despite being more flexible, application-specific processors are also not suitable for general use. By contrast, general-purpose processors, such as the ones in desktop computers, are designed to be flexible and to cover a lot of end-

user needs. Nonetheless, the main drawback of these computers is their inefficient power consumption and their performance [14]. Figure 2.1 illustrates a comparison between the different types of processors concerning their flexibility and their performance and consumption efficiency.

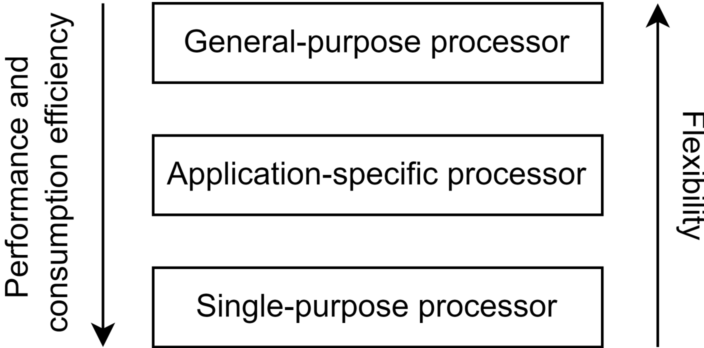


Figure 2.1: Flexibility versus performance and consumption efficiency in the different types of processors.

Single-purpose processor

A single-purpose processor, also called coprocessor, accelerator or peripheral, such as a timer or a counter, is a digital circuit designed to execute exactly one task. There are several metrics advantages and drawbacks when using this type of processors. Contrarily to the general-purpose CPUs, these have a good performance and are small and are very efficient when it comes to power consumption. However, due to the lack of flexibility, design time and Non-Recurring Engineering (NRE) costs are much higher [15]. Since these processors are very inflexible, they only contain the components needed to execute a single program. Thus, as shown in Figure 2.2, their architecture does not require a program memory, and their datapath is designed according to the application it will execute.

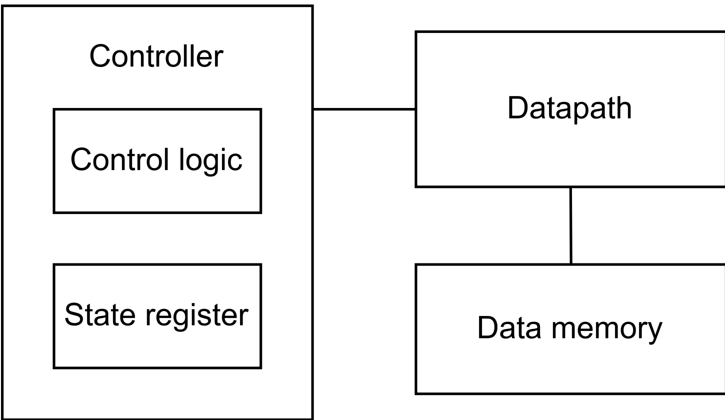


Figure 2.2: Overview of a single-purpose processor architecture.

Application-specific processor

An Application Specific Instruction-set Processor (ASIP) is a programmable processor optimised for a particular class of applications that exhibit common characteristics, such as digital signal processing, telecommunications and embedded control. These kind of CPUs are a compromise between the other two types of processors (GPPs and Single-purpose processors) since they offer a trade-off between the flexibility of a general-purpose CPU and the performance and power consumption of a single-purpose one [16]. Moreover, concerning the general architecture, ASIPs usually come with a program memory alongside an optimised datapath and a custom arithmetic logic unit. An overview of this type of architecture is represented in Figure 2.3.

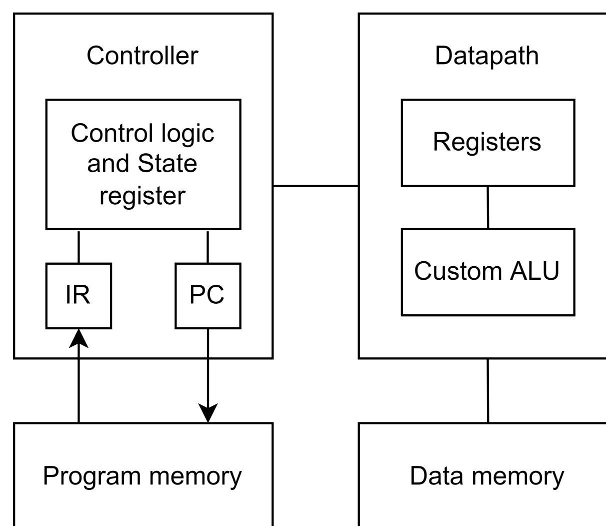


Figure 2.3: Overview of an application-specific processor architecture.

Lastly, due to its tremendous importance, it is also worth mentioning the Digital Signal Processor (DSP), which is a type of specialised processor within application-specific processors. This CPU has an architecture optimised for the operational needs of digital signal processing. Therefore, it commonly contains parallel hardware units, specialised instruction sets and high data throughput. DSPs serve in an enormous variety of applications, including image processing, sonar, radar and speech processing.

General-purpose processor

General-Purpose Processor (GPP) is a programmable device used in numerous applications. These are the kind of processors that first come to everyone's mind since they power desktop computers and are at the centre of the computer revolution that began in the 1970s [17].

As previously mentioned, these processors are well known for their high flexibility. This peculiarity, combined with the fact that GPPs are usually coupled with an operating system, makes these processors

simpler for compilers and easier for programmers to develop with. Furthermore, general-purpose processors also deliver powerful performance as a result of their highly optimised circuit and technology as well as their common usage of parallelism (e.g. superscalar and super-pipelining). However, despite all of these different qualities, GPPs aren't the most suitable processors for embedded systems due to their unpredictable execution times but also because of their high power consumption [18]. Lastly, as can be seen in Figure 2.4, these processors contain a general datapath with a large register file and general ALU (Arithmetic Logic Unit) along with a program memory.

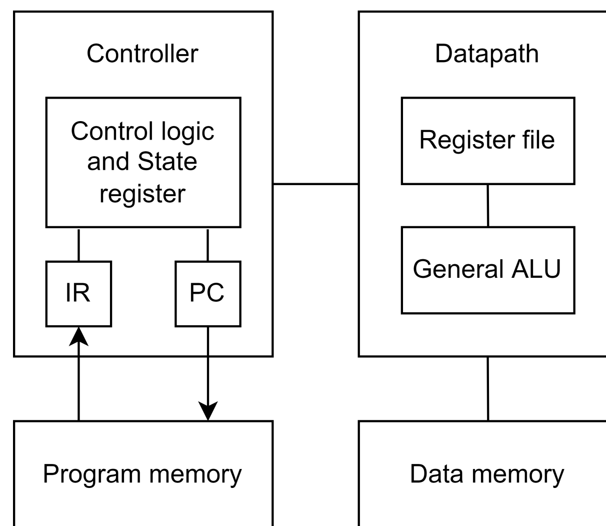


Figure 2.4: Overview of a general-purpose processor architecture.

2.1.2 Instruction Set Architectures

An Instruction Set Architecture (ISA), also referred to as computer architecture or architecture, defines the design of a computer in terms of the basic operations it must support. It serves as the interface between software and hardware since it specifies supported data types, instructions, registers, external I/O, addressing modes and memory architecture [19].

There are several types of instruction set architectures that can be classified in many different ways. The most common classification relies on the architecture complexity of the instruction set architecture.

A Complex Instruction Set Computer (CISC) contains many particular and specialised instructions. With this ISA, it is possible not only to execute several low-level and multi-step operations but also to address modes within single instructions. A Reduced Instruction Set Computer (RISC), contrarily to the previous one, only implements the instructions that are frequently used in programs. This means that it only contains small but highly optimised instructions rather than the more specialised set found in CISC [20].

Other types include Minimal Instruction Set Computer (MISC), which have a minimal small number of basic instruction operations and corresponding opcodes. Furthermore, Very Long Instruction Word (VLIW) and Explicitly Parallel Instruction Computing (EPIC) are architectures designed to exploit instruction-level parallelism with less hardware.

There are several different instructions set architectures, each with its own advantages and drawbacks. In the following pages, one will be looking into some of the features of the most used ISAs, particularly in embedded systems. Furthermore, to choose the most suitable architecture for this dissertation, a comparison between all of them will also be presented.

MIPS

Microprocessor without Interlocked Pipelined Stages (MIPS) is a Reduced Instruction Set Computer (RISC) instruction set architecture, originally developed at Stanford University in the 1980s [21]. Like most Reduced Instruction Set Computer (RISC) processors, it is a load-store architecture with general-purpose registers, which means the only instructions that can deal with the memory are the load/store and that arithmetic only operates on the registers. This design reduces the complexity of both the instruction set and the hardware, facilitating inexpensive pipelined implementations while relying on improved compiler technology [22]. Multiple versions of MIPS have emerged throughout the years, starting from MIPS I all the way to MIPS V. There are also other two types of MIPS architectures developed, MIPS-32 and MIPS-64 for 32 and 64-bit implementations, respectively [23]. Listed below are some of the most important characteristics concerning this architecture:

- MIPS provides low overhead and power efficiency since it uses fewer instructions when achieving a specific task, making it a less power consuming processor.
- MIPS I has thirty-two 32-bit general-purpose registers. The register R0 is hardwired to zero and R31 is the link register. Furthermore, there are two special registers for integer multiplication and division instructions, HI and LO. At last, the Program Counter (PC) has 32 bits and the two low-order bits are always zero.
- MIPS only has two operating modes: the kernel (supervisor) mode and the user mode.
- MIPS supports five addressing modes:
  - Register Addressing: the simplest addressing mode. As shown in Figure 2.5, the value in the register is an operand instead of a memory address to an operand.

Figure 2.5: Register addressing mode.

- Immediate Addressing: this mode implies that one operand is a constant within the instruction itself. Therefore, there is no need to have extra memory access to fetch the operand, making it relatively faster than the other modes (see Figure 2.6).

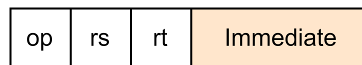


Figure 2.6: Immediate addressing mode.

- PC-Relative Addressing: this mode is used for conditional branches and is illustrated in the image below. The address is the sum of the address value with the PC.

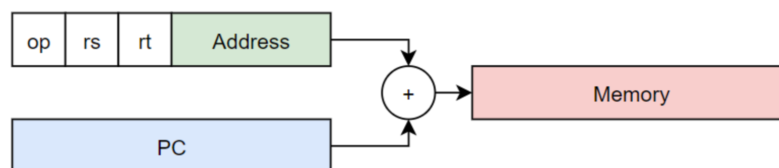


Figure 2.7: PC-relative addressing mode.

- Pseudo-direct Addressing: this mode is used in the jump instruction where the value of the offset is 6-bits and the target of the instruction jumped to is 26-bits. The upper four bits of the PC and the least two significant bits, which are 00, are all concatenated with the 26-bit immediate, resulting in a 32-bit instruction (Figure 2.8).

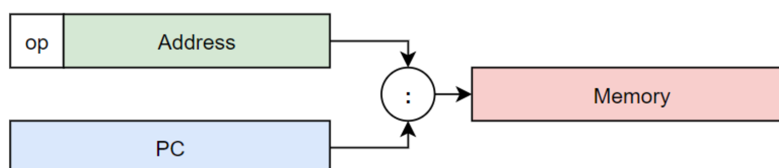


Figure 2.8: Pseudo-direct addressing mode.

- Base Addressing: this mode is used in the store/load instructions. The content of a base register (rs) is added to the address part of the instruction to obtain the effective address. This process is easier to understand in Figure 2.9.

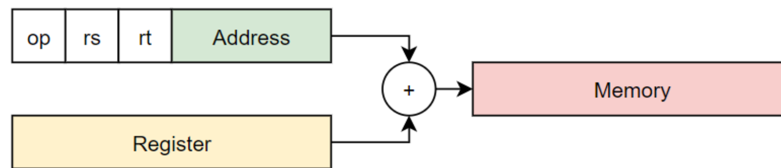


Figure 2.9: Base addressing mode.

Despite MIPS being a proprietary instruction set, it has many open-source implementations, making it suitable for beginners and very popular for educational purposes.

SPARC

Another worth mentioning and famous instruction set architecture is SPARC, which stands for Scalable Processor Architecture. Developed by Sun Microsystems in 1987 and acquired by Oracle Corporation in 2010, SPARC is a reduced instruction set computing ISA suitable for high-performance applications [24]. It has become a widely-used architecture for hardware used with UNIX-based operating systems, so it has over 8000 compatible software application programs, many compilers tools and many other features [23, 25]. Throughout the years, the architecture has gone through several revisions, which included the following features:

- SPARC's processors usually contain as many as 160 general-purpose registers. However, only 32 of them are immediately visible to software. Eight are a set of global registers and the other twenty-four are from the stack registers. These latter form a register window, a feature to improve the performance of procedure calls.
- Like MIPS, SPARC executes the instructions in only two modes: the supervisor and user modes.
- SPARC supports two addressing modes:
  - The Register indirect with index: the effective address is determined as the sum of the base register with the contents of an index register.
  - The Register indirect with immediate: this mode computes the effective address by sign extending the 13-bit immediate to 64 bits and then adding the base register's contents to it. The effective address could be made equal to the base register by making the constant equal to zero.

Unlike MIPS, due to SPARC International, SPARC is fully open, non-proprietary and royalty-free.

x86

The x86 instruction set family contains all processors derived from the Intel 8068 and its 8088 variant. Designed in 1976 by Intel Corporation, it has become the most famous architecture in the laptop, desktop, and server markets. According to Waterman [22], the reasons for this popularity are the architecture's serendipitous availability at the inception of the IBM PC; Intel's laser focus on binary compatibility; their aggressive microarchitectural implementations; and their leading-edge fabrication technology. Over the last 45 years, the ISA has undergone continuous revisions and updates, while still maintaining backwards compatibility. As a result, the x86 is extremely complex, with more than 1300 instructions, several addressing modes and dozens of special-purpose registers. The instruction set has kept instructions and modes that are no longer used today [26]. Despite this labyrinth of new and ancient technologies, the x86 contains several good features:

- The x86 ISA has a variable-length instruction format, which improves code density and consequently static code size. Despite this feature, [22] showed that IA-32 (x86 32-bit version) is only marginally denser than the fixed-width 32-bit ARMv7 encoding and that the x86-64 (x86 64-bit version) is just a bit less dense than ARMv8.
- The x86 instruction set has eight General-Purpose Registers (GPR), six Segment Registers, one Flag Register and an Instruction Pointer. All registers can be accessed in 16-bit and 32-bit modes. Furthermore, the x86-64 has additional registers and can also access them in 64-bit modes.
- The x86 architecture has several CPU operation modes. The most used are the real mode, which is supported by all x86 models, the protected mode and, lastly, the long mode, which is only supported in 64-bits machines.
- x86 supports several addressing modes, some of which were already explained:
  - Register Addressing
  - Immediate Addressing
  - Direct Addressing: this mode specifies a constant memory address where the true operand is located.
  - Indirect Addressing: this mode specifies a register or memory location whose value will be used as the effective address in memory where the true operand is located.
  - Base Addressing

  - Scaled Index Addressing: this mode uses the result of a register multiplied by a scaling factor and then added with another register as the effective address of the actual operand in memory.
  - Scaled Index with Displacement Addressing: equal to the previous mode except that an displacement is added to calculate the effective address of the actual operand in memory.

Finally, the x86 is a proprietary instruction set, making it illegal to attempt to implement a x86 microprocessor competitive with Intel's offerings.

ARM

ARM is a RISC-inspired processor architecture currently being developed by the ARM corporation. With over 130 billion processors produced in 2019, it is by far the most widely implemented ISA in the world, especially in cost-sensitive embedded systems [23, 27]. According to [28], this supremacy relies on a unique combination of features, such as its simplicity compared to most general-purpose processors, high performance and low power consumption, which is a critical requirement in embedded systems applications. Since the 1980s, when the first ARM processor was developed, several updates have been made to the architecture. Versions ARMv3 to ARMv7 support 32-bits whereas the most recent version, ARMv8, is capable of 64-bit implementations. Ultimately, despite ARM being a closed standard, which makes it illegal to subset the ISA or extend it with new instructions, it supports several features:

- Due to their low cost, high performance, low implementation size and low power consumption, ARM processors are suitable for light, portable and battery-powered devices.
- ARM processors, except for ARMv6-M and ARMv7-M based processors, have a total of 37 registers. The first 16 registers are accessible in user-level mode, whereas the additional registers are only available in other privileged modes. These 16 registers can be divided into general purpose and special purpose registers.
- ARM has 7 operating modes. Firstly, there are the user and the privileged mode. The user mode is where most applications tasks run. The privileged one is divided into two: the system mode, which uses the same registers as the user mode, and the exception mode, which is also divided into 5 different modes: Supervisor, Abort, Undefined, Interrupt and Fast Interrupt.
- ARM supports multiple addressing modes. The four main ones are listed below and for each of them, one will assign, in the respective order, the value of the Effective Address (EA) and R5 (see

Figure 2.10).

- Pre-indexed Addressing: in this mode, the source/destination address is stored in a register offset by another value. In Figure 2.10, for the instruction LDR R3, [R5,R6]:

$$EA = 1000 + 200 = 1200 \qquad R5 = 1000$$

- Pre-indexed Addressing with Write Back: this mode is similar to the previous one with the difference of writing the new address in a register. With the instruction LDR R3, [R5,R6]!:

$$EA = 1000 + 200 = 1200 \qquad R5 = 1200$$

- Post-Indexed Addressing: this mode is similar to Pre-Indexed Addressing with Write Back. However, the address is only modified and saved after the load/store operation. For the instruction LDR R3, [R5], R6, the values are:

$$EA = 1000 \qquad R5 = 1000 + 200 = 1200$$

- Program Counter Relative Addressing: this mode was already explained.

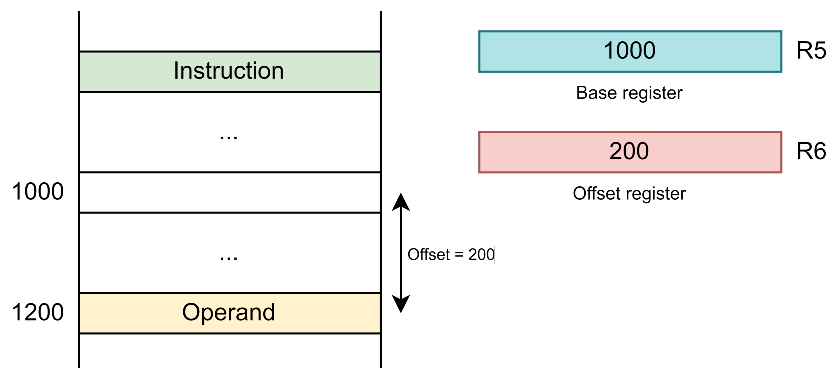


Figure 2.10: ARM addressing modes.

RISC-V

RISC-V is an entirely free and open architecture based on RISC. Its development started in 2010 at the University of California, Berkeley, and over the last years, it has already seen several implementations, as well as a big increase in its popularity [29]. With more than 30 years of hindsight, RISC-V fixes most of the failures shown by the older architectures, resulting in a clean, simple, and modular instruction set.

It includes a load-store architecture, bit patterns, IEEE 754 floating-point, and three word widths, 32, 64, and 128 bits, as well as a variety of subsets [30]. According to these widths, there are 4 base ISAs, also called Base Integer Sets: RV32I, RV64I, RV128I for 32, 64, and 128 bits, respectively, and the RV32E, a smaller instruction set for embedded systems.

Besides these features, listed below are other important ones:

- RISC-V supports variable-length instruction set extensions, both for improved code density and for expanding the space of possible custom ISA extensions.

- RISC-V base ISA has 32 general-purpose registers. The register x0 is hardwired to the ground. There is an additional user-visible program counter PC register which holds the address of the current instruction. The width of those registers are defined by the RISC-V base variant used. Therefore, for RV32, RV64 and RV128, the registers are 32, 64 and 128 bits wide, respectively. Lastly, for the special variant E, there are only 16 registers.

- RISC-V only supports three software privilege levels, the machine mode (M-mode), the supervisor mode (S-mode), and the user-mode (U-mode). The first one is the most privileged level and has access to the machine implementation. The user-mode is where most applications task run and the S-mode is intended for operating system usage [30].

- RISC-V supports four addressing modes, all of them already explained:

  - Register Addressing
  - Immediate Addressing
  - PC-Relative Addressing
  - Base Addressing

Summary

Table 2.1 summarises some of the presented instruction set characteristics. These include the machine word, the architecture kind, the design, the endianess, the instruction encoding, and the license type. As mentioned in the dissertation's goals, choosing a free and open ISA is very important to reduce costs. Given this limitation, only two option were left, SPARC and RISC-V. Ultimately, the latter was the chosen architecture since SPARC has several ISA design decisions that make it less attractive, such as the register window, delay slots, and no support for unaligned memory accesses. Furthermore, since RISC-V

is a recent instruction set, it has fixed many mistakes from the older architectures, turning it very promising for the next couple of years.

Table 2.1: Summary of several ISAs' information.

ISA	Bits	Type	Design	Endianess	Free and Open	Instruction Encoding
MIPS	32, 64	Load-Store	RISC	Big	✗	Fixed (32-bit)
ARM	32, 64	Load-Store	RISC	Bi	✗	Fixed (32-bit)
SPARC	32, 64	Load-Store	RISC	Bi	✓	Fixed (32-bit)
x86	16, 32, 64	Register Memory	CISC	Little	✗	Variable
RISC-V	32, 64, 128	Load-Store	RISC	Little	✓	Variable

2.2 Operating Systems

An Operating System (OS) is a program that manages the computer's hardware. It provides a basis for application programs and acts as an interface between the user and the machine's hardware [31]. The main function of an operating system is resource sharing, which means that several tasks will compete for the computer's resources, such as processor time, storage space, and peripheral devices. Thus, an OS must have a policy to decide the order and the time in which the resources will be shared among the different requests [32]. Furthermore, other important functions include security, system monitoring, error handling and memory, processor, device and file management.

There are several types of operating systems (e.g. distributed, templated, embedded, real-time), some more convenient, others more efficient, and others a combination of both. The type that is relevant to this work is the real-time operating systems, so, a description of how they work, as well as a comparison between some of them will be presented in the following pages.

2.2.1 Real-time Operating Systems

Real-time Operating System (RTOS) is an operating system that manages computer resources and guarantees that all timing constraints are satisfied. It is intended to serve real time applications that process data as it comes in, typically without buffer delays [33]. Hambarde et al. [34] explain that contrarily to a normal operating system, a RTOS is a system in which its correctness not only depends on the logical results of the computation but also on the time when they are produced. Therefore, it is essential that all the critical tasks have priority over all the others and that all the processing is done within the defined time constraints, otherwise, the system will fail.

According to the amount of time that real-time operating systems take to accept and complete an application's task, they can be broadly classified into two types, hard and soft. A hard RTOS is a system purely deterministic, in which delays can't be tolerated. On the other hand, soft real-time operating systems are the ones that, despite being highly undesirable, missing a deadline by a small amount of time is acceptable.

There are several different RTOS and each one has its trade-offs. In the following pages, one will look into some of the features of the most relevant RTOS. Furthermore, to choose the most suitable OS for this dissertation, a comparison between all of them will also be presented.

VxWorks

VxWorks is a real-time operating system developed by Wind River Systems. It is a proprietary software and is one of the most used RTOS in the embedded systems industry due to its deterministic performance, safety and security certifications. Examples of these industries are aerospace (the International Space Station), robotics (two of the famous NASA robots that explored Mars: Spirit and Opportunity) and automotive [34]. According to [35], VxWorks is very effective in highly complex and high-performance applications, whereas Aroca et al. [36] defend that it is an RTOS between the ones with the smallest response time and jitter, which makes it a very predictable system. Concerning hardware architectures, VxWorks supports ARM, AMD/Intel, POWER and RISC-V. Lastly, this RTOS comes with the kernel, middleware, proprietary development suite and board support packages (BSP). Figure 2.11 displays this operating system's architecture.

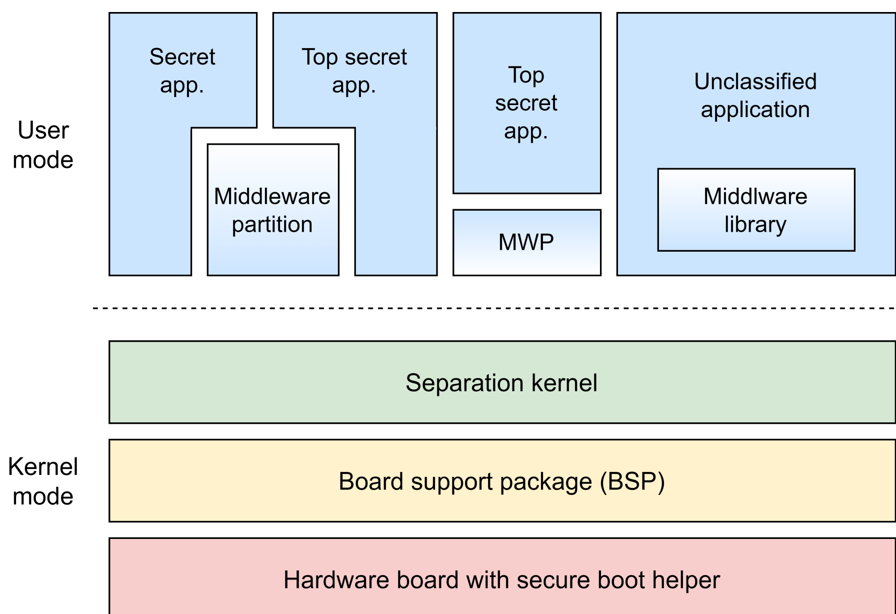


Figure 2.11: VxWorks' architecture (adapted from Mahani et al. [37]).

QNX

QNX is one of the most traditional commercial RTOS available in the market [36]. Developed in the early 1980s by Quantum Software Systems, later renamed QNX Software Systems, QNX was one of the first commercially successful microkernel-based operating systems. As a microkernel-based RTOS, the kernel implements only four basic services, IPC (Inter Process Communication), interrupt handler, task scheduler and network interface (see Figure 2.12), whereas all the remaining parts of the operating system are implemented in the user-space. [38]. According to [36], one of the advantages of this type of kernel is that, when a critical error occurs, all the remaining parts of the OS are not affected. As of today, QNX is used in plentiful embedded systems devices such as cars and mobile phones. Furthermore, it almost supports any CPU family used in the embedded market, such as x86, MIPS, PowerPC, SH-4, StrongARM, XScale and ARM.

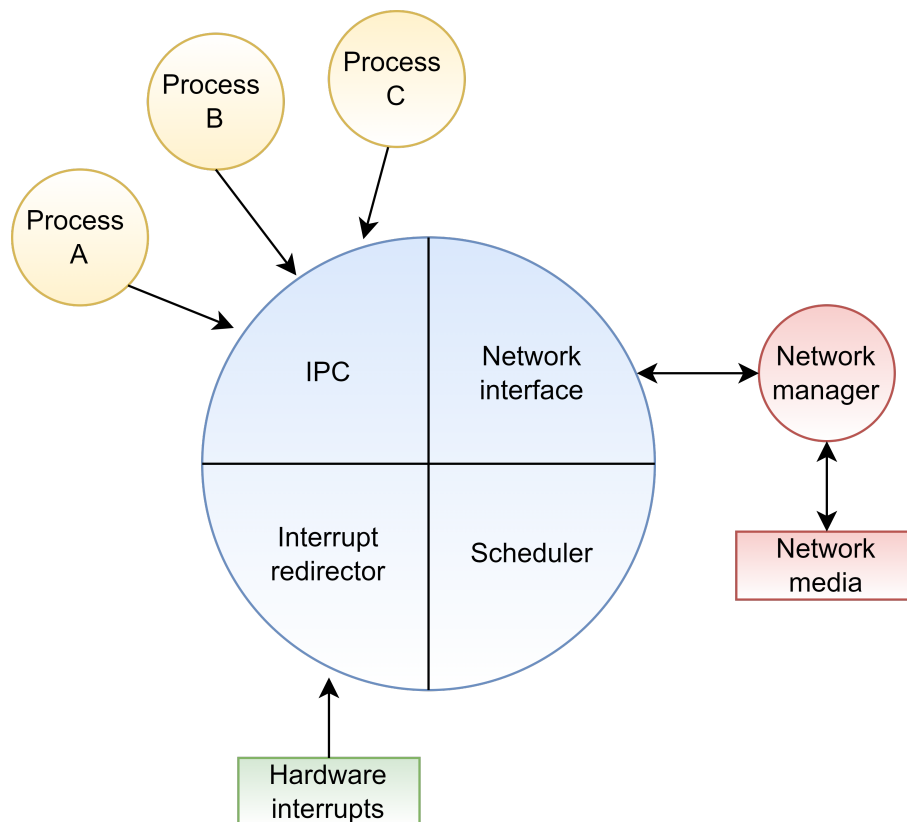


Figure 2.12: QNX microkernel's architecture. (adapted from [38]).

FreeRTOS

FreeRTOS is a market-leading real-time operating system developed by Real Time Engineers Ltd. Downloaded every 170 seconds [39]. With support for 40+ architectures, it is designed to be small and simple. The kernel consists of only three or four (depending on the usage of co-routines) C files with a few assembler

functions, with a binary image between 4 to 9KB [40]. FreeRTOS is open-source (licensed under a modified GPL) and its main advantages are portability, simplicity and scalability. It supports several scheduling policies (pre-emptive, cooperative, round-robin and hybrid), multithreading, mutexes, semaphores and software timers, with small memory footprint, low overhead, and fast execution. Moreover, it is also widely used in IoT applications, and it is even considered by [41] as the most real-time IoT OS. Another advantage is that any system that uses FreeRTOS can be rapidly migrated to SafeRTOS, which offers more safety features and turns the software certification easier. Finally, Figure 2.13 displays the architecture of this OS.

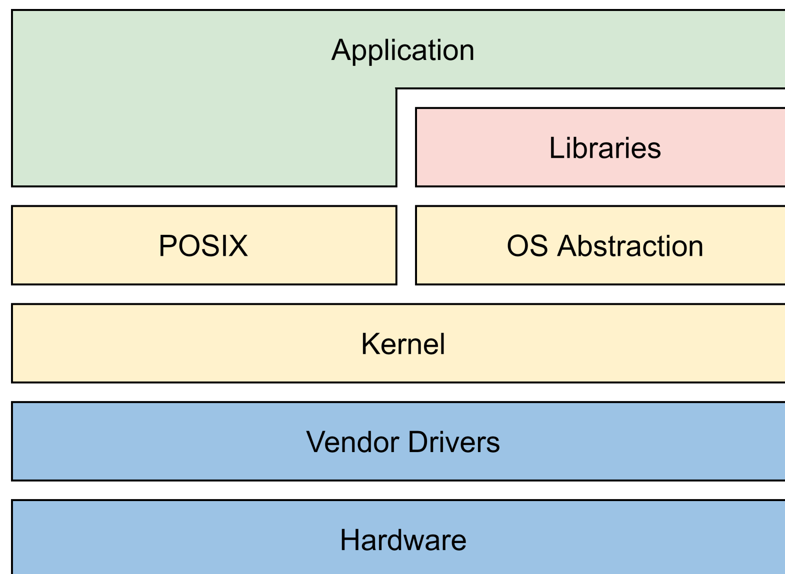


Figure 2.13: FreeRTOS' architecture (adapted from [39]).

LynxOS

Developed by Lynx Software Technologies (former LynuxWorks), LynxOS is a proprietary, Unix-like real-time operating system. It is a POSIX-compatible hard RTOS, mainly used in real-time embedded systems, from large and complex switching systems to small, highly embedded products. LynxOS runs on a microkernel architecture that combines performance, reliability and security for real-time event handling. An interesting feature of this OS is that it requires the use of a memory management unit (MMU). Kensing et al. [42] state that this is good for the system's safety but not a good characteristic for flexibility (e.g. small projects or others where security isn't a significant concern). Concerning architecture's support, it runs on PowerPC, Freescale QorIQ, x86 and even ARM. At last, an overview of the LynxOS architecture can be seen in Figure 2.14.

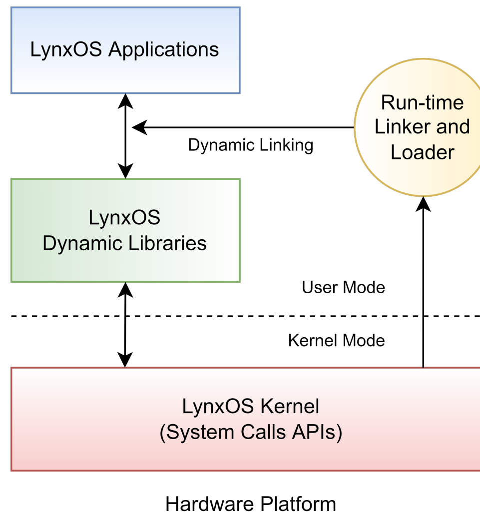


Figure 2.14: LynxOS' architecture (adapted from Mahani et al. [37]).

Azure RTOS ThreadX

Azure RTOS ThreadX is a high deterministic real-time operating system designed for deeply embedded applications. It was originally developed by Express Logic and was acquired by Microsoft in 2019. The source code is available on GitHub and its licenses are royalty-free. ThreadX is multithreading, has a preemptive scheduler and contains several RTOS features, as shown in Figure 2.15. In addition, it also has some major distinguishing technologies, such as preemption threshold, fast software timers and event-chaining. Additionally, according to Kamboh et al. [43] and Borges et al. [44], it has a small footprint (about 2KB on an ARM architecture) and can execute as fast as 1.7us per context switch in a 40MHz CPU.

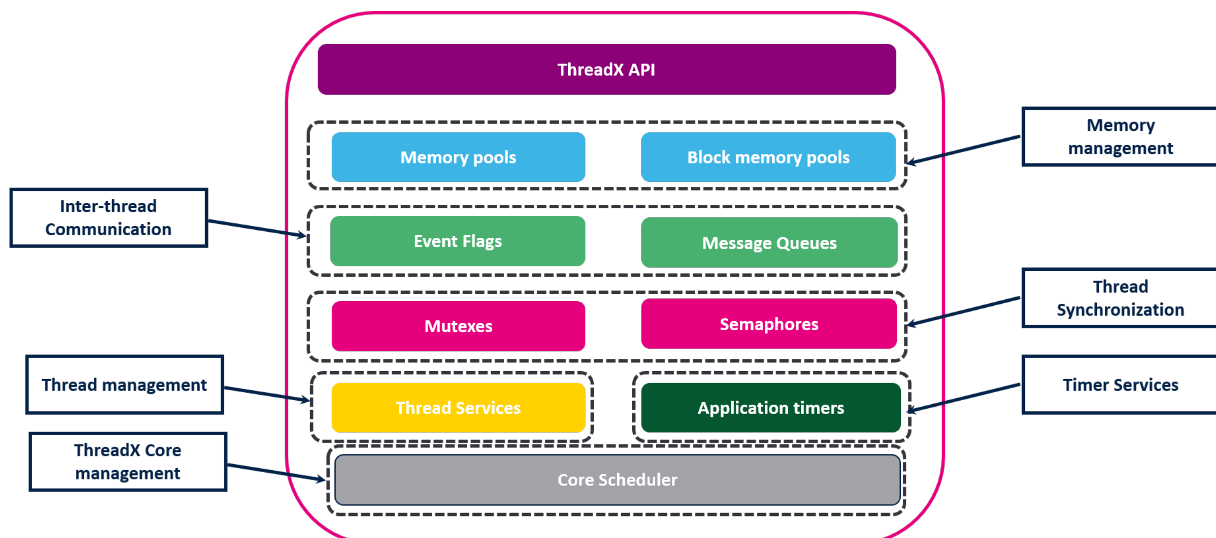


Figure 2.15: Azure RTOS ThreadX's core architecture (from [45]).

RT-Thread

RT-Thread is an open-source, neutral, and community-based RTOS developed by Bernard Xiong and the RT-Thread Development Team [46]. The kernel includes a real-time multi-tasking scheduler, semaphores, message queues, memory and interrupt management, among others. It is mainly written in the C language, turning it easier to port to several microcontrollers as well as easier to understand [47]. Besides, it has a high-quality, scalable software architecture (Figure 2.16), composed of the kernel layer, the core part of the OS; the components layer, an upper-level software on top of the kernel such as virtual file system and network interfaces; and the software layer, which includes several components for different application areas. RT-Thread supports two versions, the Standard and the Nano. The former consists of the kernel, the components, and the software layer, whereas the latter only includes the kernel and requires 3KB of Flash and 1.2KB of RAM [46]. Furthermore, as shown in Figure 2.16, RT-Thread supports several computer architectures such as ARM, MIPS, ARC, and even RISC-V.

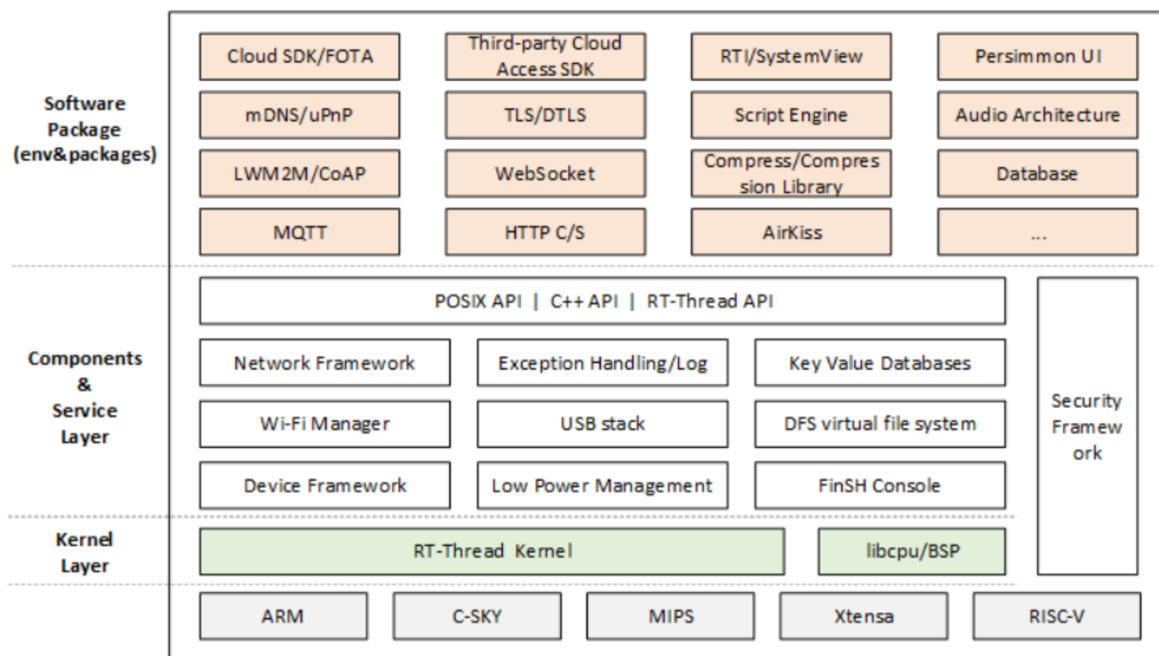


Figure 2.16: RT-Thread's architecture (adapted from [46]).

RIOT

RIOT is a small, free and open-source RTOS for low-end embedded devices maintained by a worldwide community of developers. Described as "The friendly Operating System for the Internet of Things" [48], it is designed to facilitate the development across a wide range of IoT devices [47]. RIOT is based on a microkernel's architecture and supports most of the microcontrollers ISAs (e.g. ARM, MIPS, RISC-V).

In addition, it allows application programming with C and C++ and provides multi-threading as well as real-time capabilities [49]. Besides these features, RIOT also offers several libraries such as Wiselib as well as a full IPV6 stack. The following figure illustrates an overview of the architecture.

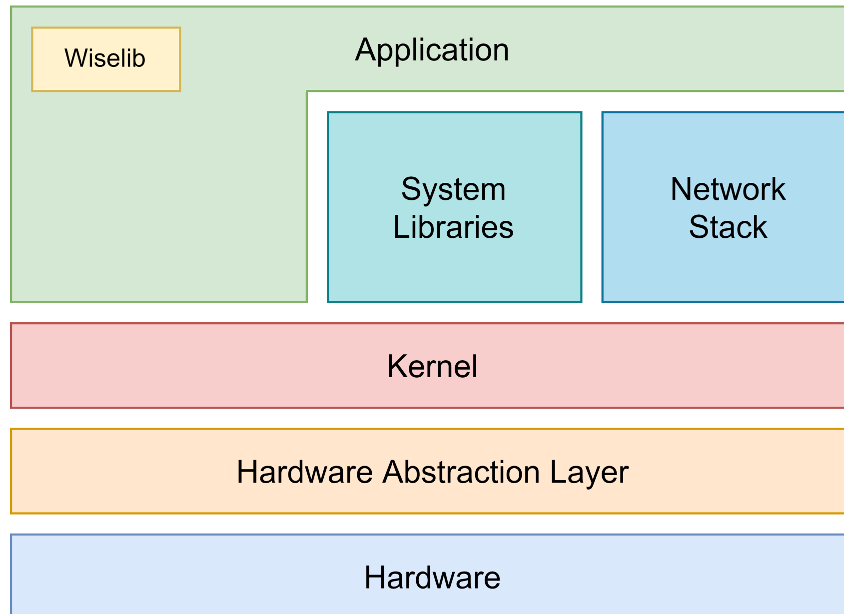


Figure 2.17: RIOT's architecture (adapted from [48]).

Summary

Table 2.2 summarises some of the important information about the previously mentioned real-time operating systems. Similarly to the ISA choice, the RTOS shouldn't have royalties, therefore, there were only four options left, RT-Thread, FreeRTOS, ThreadX and RIOT. The first one was easily discarded due to the documentation, which is mainly written in the Chinese language. From the options left, ThreadX was the final choice due to its small footprint, RISC-V support and interest from the research group.

Table 2.2: Summary of several RTOS' information.

RTOS	Kernel type	Programming model	Scheduling model	RISC-V support	Language support	Open Source
VxWorks	Monolithic	Multithreading	Cooperative and preemptive	✓	C, C++	✗
QNX	Microkernel	Event-driven	Preemptive	✗	C, C++, Java	✗
LynxOS	Microkernel	Multithreading	Preemptive	✗	C. C++, Ada	✗
ThreadX	Microkernel	Multithreading	Multiple	✓	C	✗
FreeRTOS	Microkernel	Multithreading	Multiple	✓	C	✓
RT-Thread	Microkernel	Multithreading	Preemptive	✓	C, C++	✓
RIOT	Microkernel	Multithreading	Preemptive	✓	C, C++	✓

2.3 Development platforms

The complexity of developing a soft-core microcontroller and porting an operating system to a new instruction set requires powerful tools that provide an efficient and fast development cycle. Currently, there are many development platforms by different manufacturers, each one with a large community of developers and robust support. As a result, the following subsections describe some of the relevant software and hardware tools provided by the industry and the chosen solutions to this work.

2.3.1 Software

Nowadays, every FPGA vendor offers different features and capabilities in their respective hardware design tools. Accordingly, one will be looking into the software offers from Intel, Lattice, Xilinx, and Microchip.

Quartus Prime is a programmable logic device design software produced by Intel, which offers design entry, synthesis to optimisation, verification, and simulation. It enables synthesis and analysis of Hardware Description Language (HDL) projects, which allows the developer to compile its designs, examine Register Transfer Level (RTL) diagrams, realise timing and power analysis and configure the target device. Quartus Prime's main features include SOCEDS, platform designer, external memory interface, and DSP builder. Concerning the editions, it comes with three: (i) Lite, (ii) Standard, and (iii) Pro. The first one is free and has a limit of FPGAs support, whereas the others require a paid license and have no limitations.

Lattice also offers a software design environment for FPGAs, Diamond. This tool provides a comprehensive Graphical User Interface (GUI) for controlling the FPGAs implementation, design, and verification process, as well as for timing and power analysis. Moreover, it comes with HDL code checking, several synthesis options, and a rich combination of features for a more efficient design flow. Similarly to the previous software, it has two versions, a free one and a paid license, in which the main difference is the number of supported devices.

From Xilinx, there is Vivado Design Suite for the synthesis and analysis of HDL designs. This software offers capabilities for each design stage, alongside optimisation tools for several design metrics such as timing, congestion, total wire length, and power. Vivado provides numerous features, like standards-based packaging of both algorithmic and RTL Intellectual Property (IP) for reuse, blocks and systems verification, C-based Intellectual Property (IP) generation, and Electronic System Level (ESL). Regarding the licenses, it has a WebPACK edition, with no costs associated for some devices, whereas the rest of the editions are commercial.

One more important software from Xilinx is Vitis. This tool, besides being free, unifies every aspect of Xilinx software development into a single platform. It supports embedded software and application acceleration development flow for Software Development Kit (SDK) users as well as software developers looking to accelerate the most performance-intensive parts in an FPGA. The key components of Vitis are the Artificial intelligence (AI) development environment, the accelerated libraries, the Xilinx run-time library, and the Vitis target platforms.

At last, Microchip offers the Libero Design Suite for the PolarFire, IGLOO2, SmartFusion2, RTG4, SmartFusion, IGLOO, ProASIC3, and Fusion FPGA families. This software supports programming and debugging tools capabilities, multiple constraint scenarios to optimise timing using SmartTime, Mentor Graphics ModelSim simulation, and constant transceiver eye monitoring with SmartDebug. It has four different licenses: Evaluation, Silver, Gold, and Platinum. These licenses are all paid, however, the first two can be used for free for a limited time.

2.3.2 Hardware

Currently, the FPGA market is growing at a rapid pace [50]. This market offers a different variety of solutions, with distinct specifications, for several applications. Thus, this subsection compares some of the FPGAs' families from the previously mentioned vendors.

The first FPGA family considered was the Intel Agile F-Series [51]. These field-programmable gate arrays are optimised for several Networking, Data Center, and Edge applications. Their unprecedented levels of customisation and flexibility provide acceleration and connectivity required by power-sensitive applications. The main features and benefits from these FPGAs are their high-performance processor interface speeds, architecture optimisations for artificial intelligence, the high-speed serial transceiver, and the 2nd generation of Intel HyperFlex which can improve up to 40% the performance and power consumption when compared with Intel Stratix 10 FPGAs [52].

Another interesting FPGA family from Intel is the Cyclone V [53]. Known as the industry's lowest-system cost and power FPGAs, alongside their high-performance and low time-to-market, Cyclone V is ideal for high-volume and cost-sensitive applications. Enhanced with better logic integration and differentiation capabilities, higher bandwidth, a Hard Processor System (HPS), and hard memory controllers, this solution was revealed to be a good option for its price [54].

From Lattice, the primary FPGA family analysed was the LatticeXP2 [55]. This one is optimised for high-volume and low-cost applications by combining up to 40k Look-up Tables (LUTs) with non-volatile Flash cells. Its main features are the flexiFLASH architecture, sysDSP blocks, distributed and embedded

memory, and flexible I/O buffers.

An alternative also from the previous manufacturer is the Certus-NX [56]. Lattice says that this low-power general-purpose FPGA family targets applications "from data processing in automated industrial equipment to system management in communications infrastructure". Furthermore, Certus-NX is the leading-market FPGA family when it comes to I/O density, offering up to twice as much I/O density per mm^2 when compared to related competitors. It also offers high-speed interfaces, small size, strong authentication and encryption, instantaneous start-up features, and high reliability.

A different possibility is the Artix-7 from Xilinx [57]. This FPGA family leads in system performance-per-watt for cost-sensitive applications, including software-defined radio, machine vision cameras, and low-end wireless backhaul [57]. Besides their high-performance and low-cost, these low-end devices offer DSP processing, transceiver line rates, small footprint and packaging, and Analog Mixed Signal (AMS) integration. Aside from being part of the All Programmable Cost-Optimised Portfolio, the Artix-7 FPGAs also belongs to the Zynq-7000 All Programmable SoCs, which integrates the software programmability of an ARM-based processor with the hardware programmability of an FPGA [58].

Virtex Ultrascale+ is the flagship family developed by Xilinx [59]. It has several power options for an ideal mix between the required system performance and the smallest power envelope. Additionally, these devices provide the highest transceiver bandwidth, highest DSP count, and highest on-chip and in-package memory, making them ideal for compute-intensive applications. Such as the Artix-7, Virtex FPGAs are also part of an SoC portfolio, the Zynq Ultrascale+ SoCs.

Ultimately, there are the PolarFire FPGAs from Microchip [60]. This family provides the industry's lowest power at mid-range densities and is ideal for a diverse landscape of applications (e.g. IoT, cellular infrastructures, industrial automation). With exceptional security and reliability, these devices feature up to 500K Logic Elements (LEs), 1.6Gbps I/Os, 12.7G transceivers, and hardened security IPs.

2.3.3 Xilinx Arty A7-100

From all the previous FPGAs landscape analysed, the most suitable option was one from the Artix-7 family. The reason for this choice was the fact that, since the objectives of this dissertation were already ambitious, choosing a platform in which the author was already familiarised was important. Therefore, adopting an FPGA from Xilinx was mandatory because it would reduce the learning curve and the development cycle time. Thus, only two options remained, the Artix-7 and the Virtex Ultrascale+. The former prevailed due to the high-end specs of the Virtex, which would be overabundant for this work and also more expensive and energy-consuming.

The Arty A7-100 is a development platform designed around the Artix-7 FPGA family [61]. This board includes the FPGA part XC7A100TCSG324-1, is very flexible, and is supported under the free license of Vivado, turning this solution much cheaper. This platform can be seen in Figure 2.18 and it includes 15850 logic slices, 126.8K flip-flops, 4860Kbits of block RAM (Random Access Memory), 240 DSP slices and 6 clock management tiles. It can connect to the computer through the 10/100 Mbps Ethernet port or the USB-UART bridge. Other system features are 256MB DDR3L with a 16-bit bus @ 667MHz, 16MB Quad-SPI Flash, and USB-JTAG programming circuitry. The board capabilities can be extended through the Arduino/chipKIT Shield connector or by the 4 Pmod connectors. At last, concerning the interaction and sensory devices, it has one reset button, four switches, four buttons, and four RGB and normal LEDs.

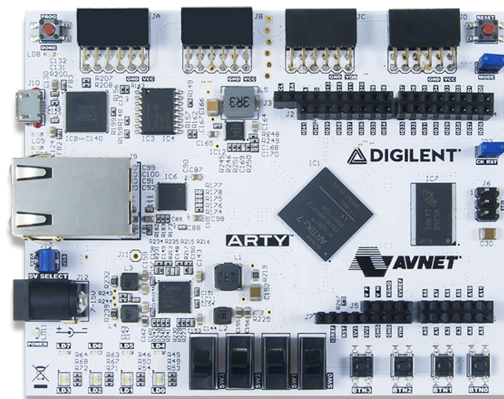


Figure 2.18: Xilinx Arty A7-100 development board.

Artix-7 XC7A100T FPGA

Nowadays, the software tools for hardware design are so advanced that the allocation of CLBs (Configurable Logic Blocks) resources don't require human intervention anymore. However, CLBs are the main logic resources for implementing sequential as well as combinational circuits, so it can still be useful for the designer to know some CLBs details. Thus, one will be looking into some features, the resources, the arrangement and the constitution of the CLBs of Arty's FPGA.

Firstly, all the Artix-7 family and even the 7 series FPGAs configurable logic blocks provide the following features [62]:

- 6-input LUT technology
- Dual LUT5 option

- Distributed Memory and Shift Register Logic capability
- Dedicated high-speed carry logic
- Several multiplexers

Concerning the CLBs arrangement, as shown in Figure 2.19, they are disposed in columns and are based on the approach provided by the ASMBL architecture (for further information, consult [63]). Lastly, a CLB has 2 slices, and each slice is composed of 4 LUTs, 8 flip-flops, and 1 carry logic.

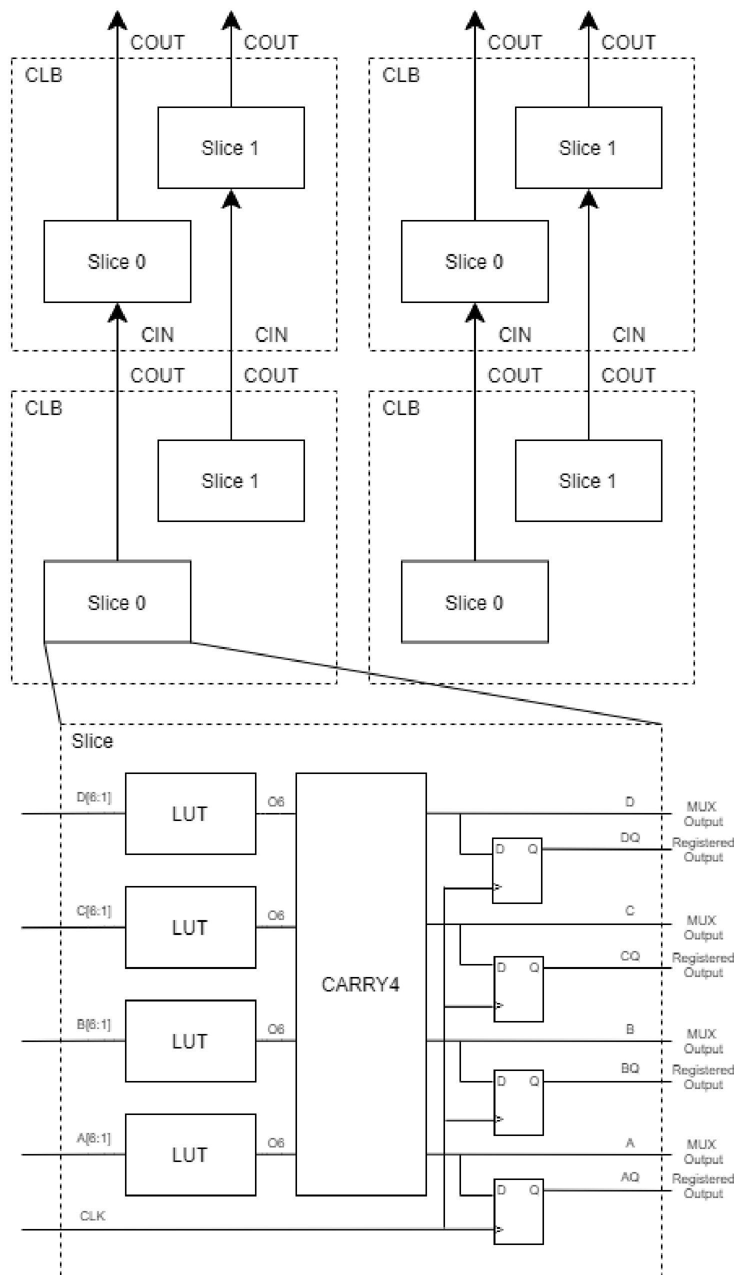


Figure 2.19: CLB arrangement and constitution (adapted from [62]).

Chapter 3: RISC-V core

This chapter describes an overview of the RISC-V ISA, focusing on the Base Integer Instruction Set as well as the standard extensions. Furthermore, due to the huge number of RISC-V implementations available, a comparison between several soft-cores is made to see which one is more suitable for this dissertation.

3.1 RISC-V ISA

RISC-V has 4 ISA bases, also called Base Integer Sets, RV32I, RV64I, and RV128I for 32, 64, and 128 bits, respectively, and the RV32E, a smaller instruction set for embedded systems. Just from these bases, it is possible to understand the flexibility of RISC-V. It supports the entire processor spectrum, going from small embedded devices to high-performance 128 bits machines. As a result, to keep up with this flexibility, optional extensions can be added to the base ISA, defining the exact features of the processor to a specific application.

The following pages focus on the RV32I base ISA, which is the relevant one to this work. Moreover, a quick overview of the RISC-V extensions, more precisely, the Integer Multiplication and Division extension, will also be discussed.

3.1.1 RV32I Base Integer Instruction Set

According to The RISC-V Instruction Set Manual [30], the RV32I was designed to reduce the hardware required while still adequately supporting modern operating system environments. It contains 32 registers (x0-x31), each one 32 bits wide, with x0 hardwired to the ground. Instructions are 32 bit long and must be stored naturally aligned in memory, in little-endian byte order. At the moment, in version 2.1, there are 40 instructions, which can be divided as follow:

- Computational instructions
- Memory Access instructions
- Control Flow instructions

- Ordering instruction
- System instructions

Regarding the instructions formats, there are six of them, R, I, S, B, U, and J, as shown in Figure 3.1. As can be seen, the RISC-V ISA keeps the source (rs1 and rs2), as well as the destination (rd) registers, at the same position so it can simplify decoding logic. In addition, immediates are always sign-extended and are packed towards the leftmost available bits in the instruction [30].

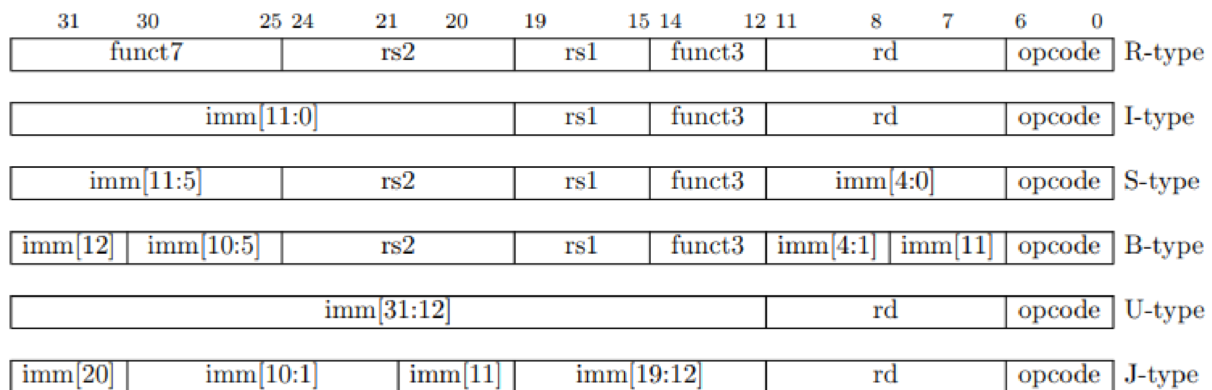


Figure 3.1: RISC-V base instruction formats (from [30]).

3.1.2 Extensions

As mentioned earlier, RISC-V has been designed to support extensive customisation and specialisation. Although the base instruction set can implement a simple general-purpose processor by itself, it lacks several functions and features that might be needed for some applications. Thus, once the base is selected, it is also possible to add extensions to define the exact features of the processor. This mechanism permits adding and removing capabilities to the base ISA incrementally and suitably, optimising the trade-offs between chip size, performance, and system features [64]. Furthermore, these extensions can be divided into standard, generally useful for several applications, and non-standard, usually highly specialised. Table 3.1 features the list of all the RISC-V standard extensions. As can be seen, some of them already suffered rectifications, such as the Atomic Instruction (A) and the Integer Multiplication and Division (M), whereas the ones marked as Open are still in early stages and under development.

Table 3.1: RISC-V standard extensions.

Name	Extensions	Status
M	Standard Extension for Integer Multiplication and Division	Ratified
A	Standard Extension for Atomic Instructions	Ratified
F	Standard Extension for Single-Precision Floating-Point	Ratified
D	Standard Extension for Double-Precision Floating-Point	Ratified
Q	Standard Extension for Quad-Precision Floating-Point	Ratified
L	Standard Extension for Decimal Floating-Point	Open
C	Standard Extension for Compressed Instructions	Ratified
B	Standard Extension for Bit Manipulation	Open
J	Standard Extension for Dynamically Translated Languages	Open
T	Standard Extensions for Transactional Memory	Open
P	Standard Extension for Packed-SIMD Instructions	Open
V	Standard Extension for Vectors Operations	Open
N	Standard Extension for User-Level Interrupts	Open
H	Standard Extension for Hypervisor	Open
S	Standard Extension for Supervisor-level Instructions	Open

Standard Extension for Integer Multiplication and Division

From all the RISC-V extensions, only the Integer Multiplication and Division is used, so a brief overview of it is worth giving. This extension is currently in version 2.0 and adds the functionality of multiplying or dividing values between two operands stored in two registers. As a result, some new instructions are added to the ISA, which can be seen below.

- **mul**: Multiplication between two operands and returns the lower 32 bits
- **mulh**: Multiplication between two signed operands and returns the upper 32 bits
- **mulhu**: Multiplication between two unsigned operands and returns the upper 32 bits
- **mulhsu**: Multiplication between signed and unsigned operands and returns the upper 32 bits
- **div**: Division between two signed operands
- **divu**: Division between two unsigned operands
- **rem**: Returns the remainder of div
- **remu**: Returns the remainder of divu

3.2 RISC-V implementations

Due to the RISC-V open nature and increased popularity in recent years, there is a huge number of processor implementations based on its ISA. These implementations differ a lot from each other, going from simple commercial CPU cores to advanced open-source SoCs capable of running an operating system. Therefore, based on the features that are important to this work (documentation, extensions, and FPGA and debug support), some implementations from the list of the official RISC-V website [65] are going to be compared.

VexRiscv

VexRiscv [66] is a RISC-V implementation written in SpinalHDL, a high-level hardware description language that generates VHDL/Verilog files. The core supports the RV32IMCA instruction set and it can have between 2 and 5 pipeline stages (Fetch, Decode, Execute, Memory and Write-Back). It offers debug support and even an eclipse extension which permits debugging the core with GDB inside the IDE, turning the process much easier.

Regarding the FPGA resources and clock frequency, it can run at 216Mhz, in the Artix 7, occupying 1418 LUTs and 949 FFs. These values concern a very similar configuration to what is needed in this dissertation, which includes a debug module and uses the RV32IM ISA. When it comes to documentation, is it at best acceptable and the core architecture is not described.

PicoRV32

PicoRV32 [67] is a 32-bit RISC-V soft-core that implements the RV32IMC ISA. It is written in Verilog and it comes with three variations, picorv32, picorv32_axi, and picorv32_wb. The first one comes with a simple memory interface, whereas picorv32_axi and picorv32_wb provide AXI-4 Lite and Wishbone master interfaces, respectively. These variants are useful since it makes easier to integrate the core with existing systems that already use these interfaces.

Despite not having a debugger, PicoRV32 is suitable for FPGA and even supports three configurations: small, regular, and large. The first configuration doesn't include counter instructions, two-stage shifts, and catching of misaligned memory accesses and illegal instructions, using only 761 LUTs on the Xilinx 7-Series FPGAs. Then, the regular architecture, which works with only 917 LUTs, is nothing more than the core in its default configuration. The last configuration supports the M and C extension, barrel shifter, and the Pico Co-Processor Interface, requiring 2019 LUTs. Documentation concerning the architecture is

non-existent. Besides, the fact that the core is written in just one file without any comments, turns the process of understanding the core microarchitecture even harder.

PULPino

Developed by ETH Zurich, PULPino [68] is a 32-bit RISC-V System on Chip (SoC) that includes several peripherals such as Inter-Integrated Circuit (I2C), debug unit, timers, and Universal Asynchronous Receiver-Transmitter (UART). This SoC can be configured to use either the RISCY (now called CV32E40P and maintained by the OpenHW Group) or the zero-riscy (now denominated Ibex and maintained by lowRISC) core, which were two interesting options to analyse.

CV32E40P [69] is an in-order, single-issue RISC-V core with 4 pipeline stages. The ISA includes the standard M, F and C extensions (see table 3.1). It has also been extended with several non-standard instructions such as hardware loops, post-increment load and store instructions and additional ALU instructions.

Ibex [70] is RISC-V core written in SystemVerilog that supports the Integer (I) or Embedded (E) bases as well as the Integer Multiplication/Division (M), Compressed (C), and B (Bit Manipulation) extensions. It has a 2-stage pipeline and it has been designed for efficiency and area optimisation to target embedded applications.

Both cores have good documentation and are suitable for either FPGA or Application-Specific Integrated Circuit (ASIC) implementations. Debug support is also available for the two CPUs, and the fact that PULPino already integrates them with a debug module can be very useful.

Potato

At last, Potato [71] is a simple RISC-V soft-core, written in VHDL, with support for the base integer ISA (RV32I). It includes a 5-stage pipeline, optional instruction cache, 8 IRQs, and a Wishbone interface for peripheral connection. It doesn't have a debugger, and on its GitHub page, an example of an SoC using potato is available for the Arty FPGA, running at a clock frequency of 50MHz. This SoC includes a 32-bit timer with compare interrupt, a GPIO module, a block RAM memory, and a UART with RX/TX interruptions. Like some other cores already analysed, Potato also doesn't have any documentation concerning the microarchitecture.

Summary

Table 3.2 summarises the extracted information from the soft-cores previously analysed. As mentioned at the beginning of the sub-chapter, the CPU features which are important when choosing the RISC-V core for this dissertation are the documentation, the extensions, and the support for FPGA and debugger.

Table 3.2: Summary of several RISC-V soft-cores' features.

RISC-V core	Extensions	HLD	FPGA support	Documentation	Debug support
VexRiscv	I, M, C, A	SpinalHDL	✓	✓	✓
PicoRV32	I, E, M, C	Verilog	✓	✗	✗
CV32E40P	I, M, F, C	SystemVerilog	✓	✓	✓
Ibex	I, E, M, C, B, F	SystemVerilog	✓	✓	✓
Potato	I	VHDL	✓	✗	✗

Since they don't have debug support and practically non-existent documentation, both PicoRV32 and Potato were discarded. From the three options left, despite checking all the boxes, VexRiscv was rejected due to its disadvantages when compared to the PULPino cores. First, the documentation isn't so complete, then, the fact that it is written in SpinalHDL, a language not known by the author and currently with a small community. Therefore, both CV32E40P or Ibex could be chosen, however, the latter prevailed since it was designed for embedded systems.

Chapter 4: Soft-core microcontroller

This chapter presents the developed soft-core microcontroller. It starts with an architecture overview of the entire system, followed by a more detailed explanation of its main components. These include the Ibex core, the debug unit, the peripherals, and the AXI interface. Lastly, the memory map of the soft-core microcontroller is also presented.

4.1 Architecture overview

Figure 4.1 illustrates the EKKO architecture overview. It contains the Ibex CPU, the chosen RISC-V core, that connects to the system bus, which is responsible for connecting every major component and carrying data, instructions and control signals between them. Besides Ibex, this bus is also shared with a debug unit, which purpose is to debug and load programs into the core. In addition, there is also a RAM, which serves as an instruction and data memory, and an AXI master, for easy peripheral integration. Ultimately, there are three peripherals, each one with an AXI slave interface, making it possible for the CPU to communicate with them through the AXI interconnect.

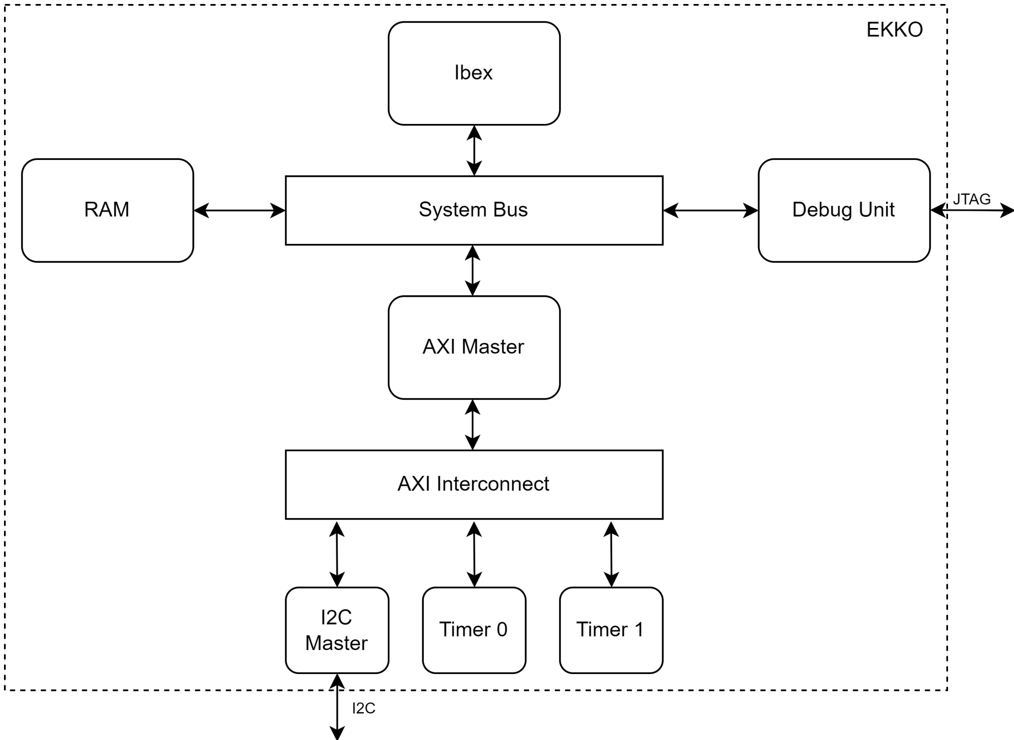


Figure 4.1: EKKO architecture overview.

4.2 Ibex

Ibex is an open-source standards-compliant 32-bit RISC-V processor currently maintained by lowRisc. It is highly parametrizable, as seen in Code 4.1, and is suitable for embedded systems. Ibex supports the RV32I or the RV32E Base Integer Instruction Set, in addition to the C, Zicsr and Zifencei extensions which are always enabled as well as the C and M extensions which are optional. Moreover, as can be seen in Figure 4.2, this core has a 2-stage pipeline: Instruction Fetch (IF) and Instruction Decode and Execute (ID/EX).

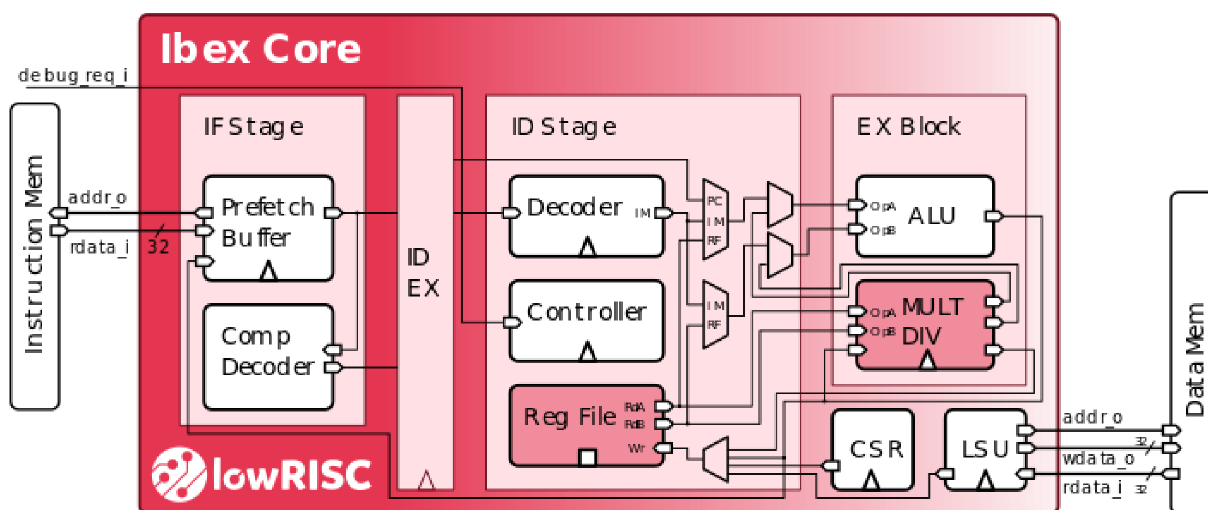


Figure 4.2: Ibex pipeline architecture (from [70]).

The first stage is responsible for fetching instructions from the memory and supplying them to the ID stage. This process can be done in one cycle (if the instruction memory is capable), and it goes throw a prefetch buffer for better performance and timing constraints. The last stage decodes the received instruction and immediately executes it. All of this can also be done within one clock cycle, including the register read and write, unless it is a multi-cycle instruction, which will stall in this stage until it completes. As a result, the maximum Instructions Per Cycle (IPC) that can be achieved in this core is 1 when multi-cycle instructions are not used.

Concerning the configurations, Ibex has several parameters to be suitable for different types of applications. Code 4.1 presents the instantiation of the Ibex core alongside the chosen parameters, which are almost equal to the default ones. The Physical Memory Protection (PMP) is disabled, the base instruction set is the RV32IMC, the register file is selected to FPGA implementation instead of the ASIC, and lastly, the debugger trigger is enabled.

Code 4.1: Ibex instantiation code.

```
ibex_core #(
    .PMPEnable          ( 0                              ),
    .PMPGranularity     ( 0                              ),
    .PMPNumRegions      ( 4                              ),
    .MHPMCounterNum     ( 0                              ),
    .MHPMCounterWidth   ( 40                             ),
    .RV32E              ( 0                              ),
    .RV32M              ( ibex_pkg::RV32MFast            ),
    .RV32B              ( ibex_pkg::RV32BNone            ),
    .RegFile            ( ibex_pkg::RegFileFPGA          ),
    .BranchTargetALU    ( 0                              ),
    .WritebackStage     ( 0                              ),
    .ICache             ( 0                              ),
    .ICacheECC          ( 0                              ),
    .BranchPrediction   ( 0                              ),
    .DbgTriggerEn       ( 1                              ),
    .DbgHwBreakNum      ( 2                              ),
    .SecureIbex         ( 0                              ),
    .DmHaltAddr         ( DEBUG_START + dm::HaltAddress      ),
    .DmExceptionAddr    ( DEBUG_START + dm::ExceptionAddress )
)

u_core (
    // Clock and reset
    .clk_i                   (clk_sys),
    .rst_ni                  (rst_core_n),
    .test_en_i               (0),

    // Configuration
    .hart_id_i               (0),
    .boot_addr_i             (0),

    // Instruction memory interface
    .instr_req_o             (instr_req),
    .instr_gnt_i             (instr_gnt),
    .instr_rvalid_i          (instr_rvalid),
    .instr_addr_o            (instr_addr),
    .instr_rdata_i           (ssb_rdata),
    .instr_err_i             (0),

    // Data memory interface
    .data_req_o              (data_req),
    .data_gnt_i              (data_gnt),
    .data_rvalid_i           (data_rvalid),
    .data_we_o               (data_we),
    .data_be_o               (data_be),
    .data_addr_o             (data_addr),
    .data_wdata_o            (data_wdata),
    .data_rdata_i            (ssb_rdata),
    .data_err_i              (0),

    // Interrupt inputs
    .irq_software_i          (0),
    .irq_timer_i             (timer0_irq),
    .irq_external_i          (0),
    .irq_fast_i              (0),
    .irq_nm_i                (0),

    // Debug interface
    .debug_req_i             (dm_debug_req),

    // Special control signals
    .fetch_enable_i          (1),
    .alert_minor_o           (0),
    .alert_major_o           (0),
    .core_sleep_o            (0)
);
```

4.3 Debug unit

Due to the complexity increase in designing and developing embedded systems in recent years, some major development cycle challenges are testing and debugging. As a result, a debug unit is integrated into the microcontroller, which permits the execution of the target program under controlled conditions, making it easier to monitor the system's behaviour.

Figure 4.3 illustrates the different components which usually are used in an end-to-end debug system. Starting with the host device, it typically contains an IDE, which is optional, a debugger and a debug translator. An IDE can be any program with a graphical interface and built-in debug capabilities, such as Eclipse. The debugger, usually gdb, is responsible for loading code, monitoring and modifying values of internal variables, halting the CPU, managing breakpoints, etc. At last, the debug translator (OpenOCD) is the software tool responsible for receiving high-level debug commands from gdb, translating them into JTAG signals, and sending them to the JTAG bridge through a debug dongle (JLink in this case). On the target side, the JTAG bridge feeds the JTAG signals into a Test Access Port (TAP), which will transfer the received commands to the Debug Module and consequently to the debug logic of the CPU.

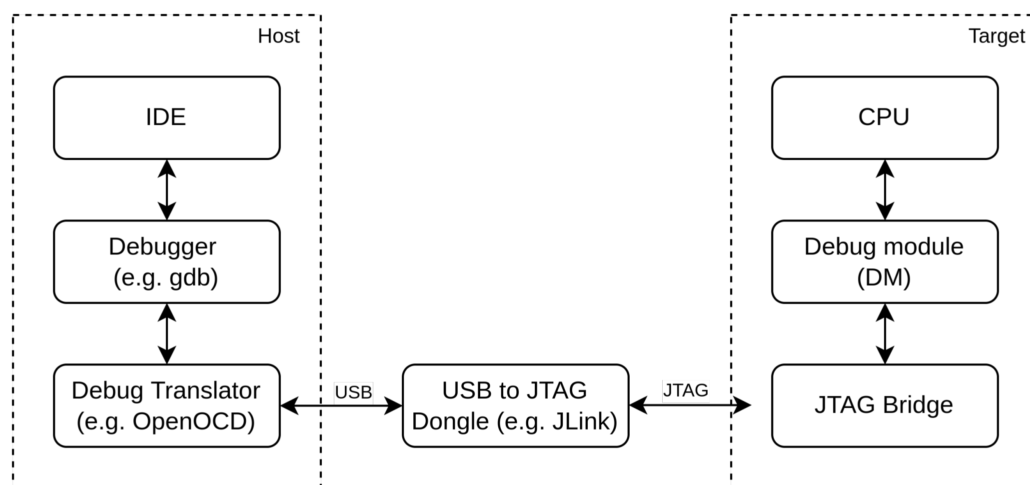


Figure 4.3: Debug flow overview (adapted from [72]).

The debug unit is composed of the JTAG bridge and the debug module, and the one used [73], such as PULPino, was developed by the pulp platform. The reasons for this choice are its run-control debug and bus access functionality according to the RISC-V Debug Specification 0.13.2 [74], which makes it compatible with the riscv-gdb from the RISC-V GNU toolchain, and the fact that its bus host interface is compatible with the instruction and data interfaces of Ibex, making it easier to integrate into the system.

According to the RISC-V Debug Specification, the debug unit consists of a JTAG Debug Transport Module (DTM) and a Debug Module (DM), as shown in Figure 4.4. The first one contains a typical JTAG

TAP, which permits access to the JTAG registers. The Debug Module implements a translation interface between abstract debug operations and their specific implementation [73].

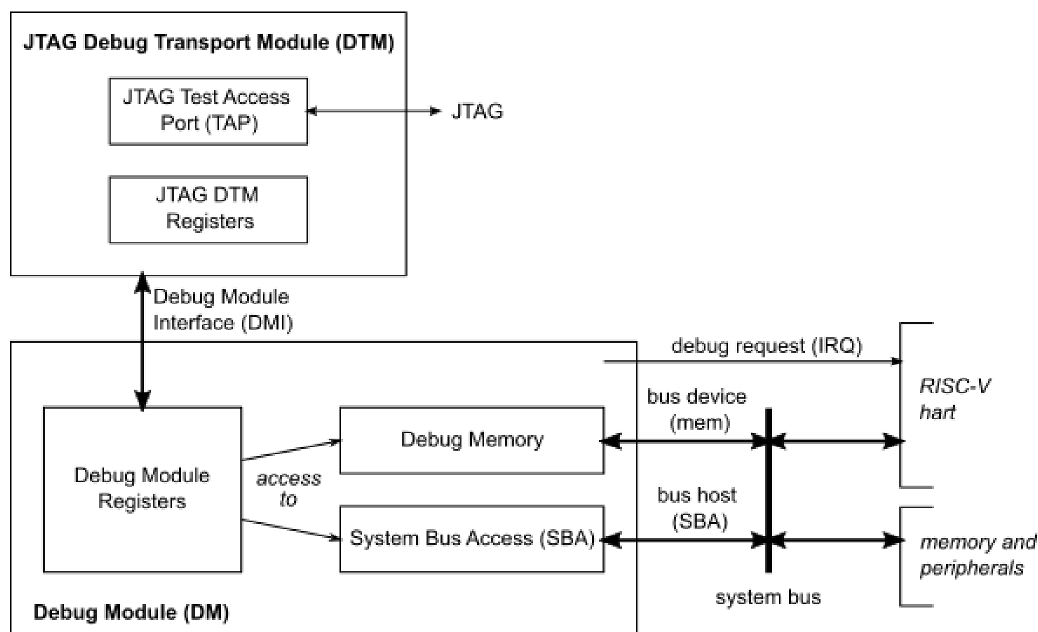


Figure 4.4: Debug unit block diagram. (from [73]).

4.4 Peripherals

Peripherals are external devices that permit the processor to interact with the "outside world". Thus, this section presents two peripherals developed to enlarge the core capabilities. The first one is an I2C module, which can connect up to 127 sensors to the core. Then, two timers are also available, which can have many applications such as counting time and working as a system tick for an RTOS.

4.4.1 I2C

Inter-Integrated Circuit (I2C) is a synchronous, serial bus protocol intended for low speed and short-distance communications. It was designed by Philips Semiconductor [75], and due to its nature, it is widely used to integrate slower devices, such as peripherals, into processors and microcontrollers.

As represented in Figure 4.5, I2C only requires two wires, the serial data (SDA), a bidirectional line for data transmission, and the serial clock (SCL), which is responsible for the protocol synchronisation between the master and the slave. This protocol supports up to 127 slaves devices, which are identified by the most significant 7 bits of the device address. Furthermore, I2C contains 3 different operation modes: standard mode (up to 100 kbit/s), fast mode (up to 400 kbit/s), and high-speed mode (up to 3.4Mbit/s).

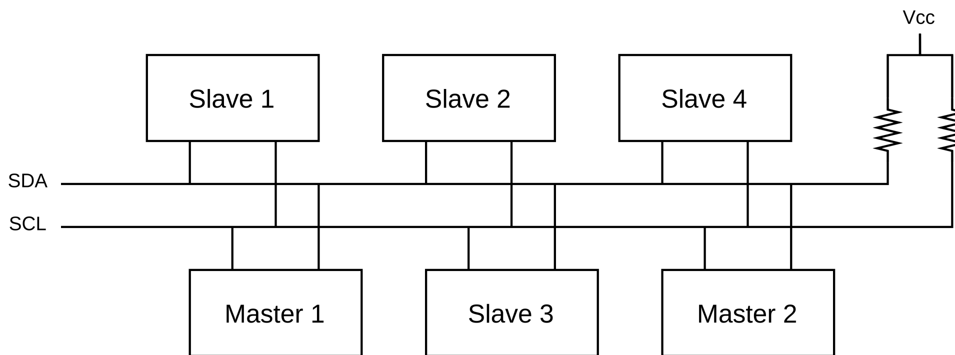


Figure 4.5: I2C connection diagram (from [76]).

Concerning the data transmission, it begins with a start from the master, causing all slaves devices to "wake up". Then, the master will send the device address of the slave with which it wants to communicate with, alongside the read/write bit. Afterwards, all the slaves will compare the address received with their internal one, sending an acknowledgement (ACK) in case of a match. Once the master receives this acknowledgement, according to the already sent RW bit, it will either send or receive the data byte, followed by the respective ACK. Lastly, a stop signal is emitted by the master, finishing the transmission. This entire process can be seen in Figure 4.6, which also clarifies the states of the SDA and SCL signals during the communication.

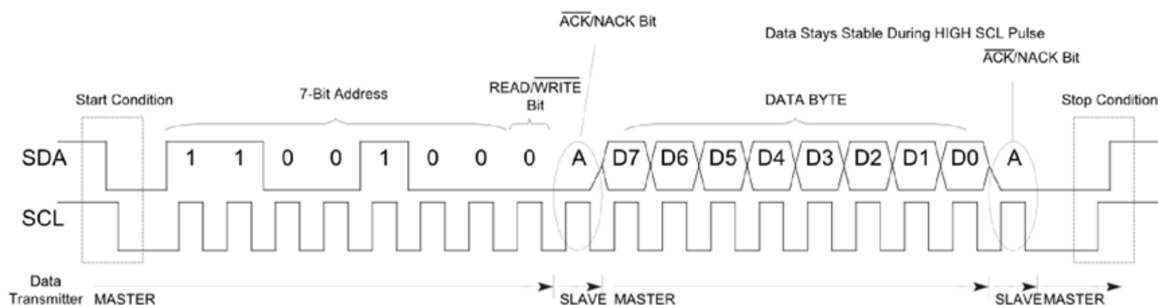


Figure 4.6: Representation of the I2C protocol.

Figure 4.7 represents the architecture of the developed I2C module, which is composed of a control unit and a datapath. The former generates all the control signals accordingly to the finite state machine presented in Figure 4.8, whereas the latter controls the SDA and SCL signals which are connected to the FPGA pins.

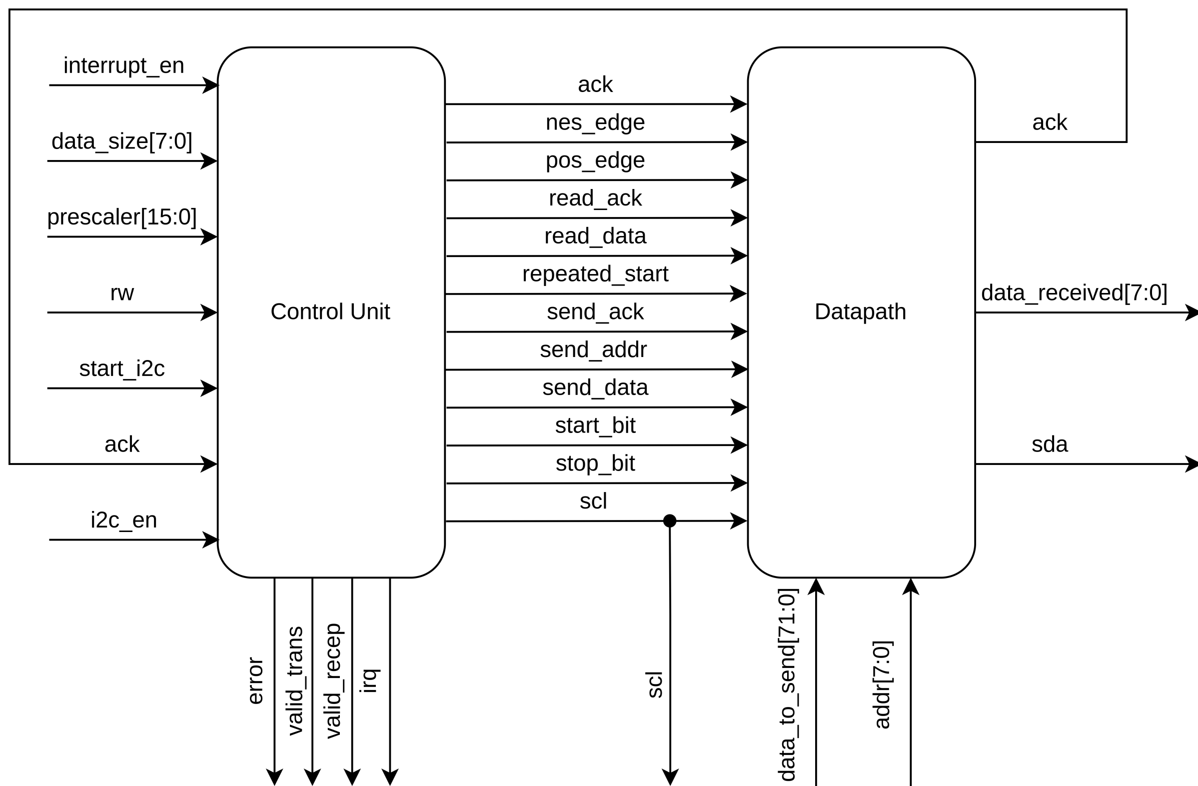


Figure 4.7: I2C module architecture.

The state machine starts when the **start_i2c** signal is received, consequently producing the start bit of the transmission. Afterwards, the device address will be sent in the **ADDR** state, followed by the reading of the ack bit in the **ADDR ACK**. From here, it can go to **ERROR** (in case of an error in the transmission) or to the **READ** or **WRITE** states, depending on the **addr_sent** flag. If it is set to 0, it means we are in a writing operation, so it will change to its respective state. On the other hand, if the **addr_sent** is 1, it means we have previously sent the register address, meaning it is in a read operation. This flag is set to 1 in the **REPEATED START** state because before a read, a write operation needs to be done to specify from which register of the slave the master wants to read. Therefore, after one entire byte is sent in the **WRITE** state, the **WRITE ACK** state will check the **wr** bit to decide if it should go to the **REPEATED START** or the **STOP**. In case of a write operation (wr=0), before going to the **STOP** state, it will also check if there are no more bytes to send (**data_sent**=1) since it is possible to send 9 bytes for each transmission. Finally, the **STOP** state generates the stop bit of the protocol.

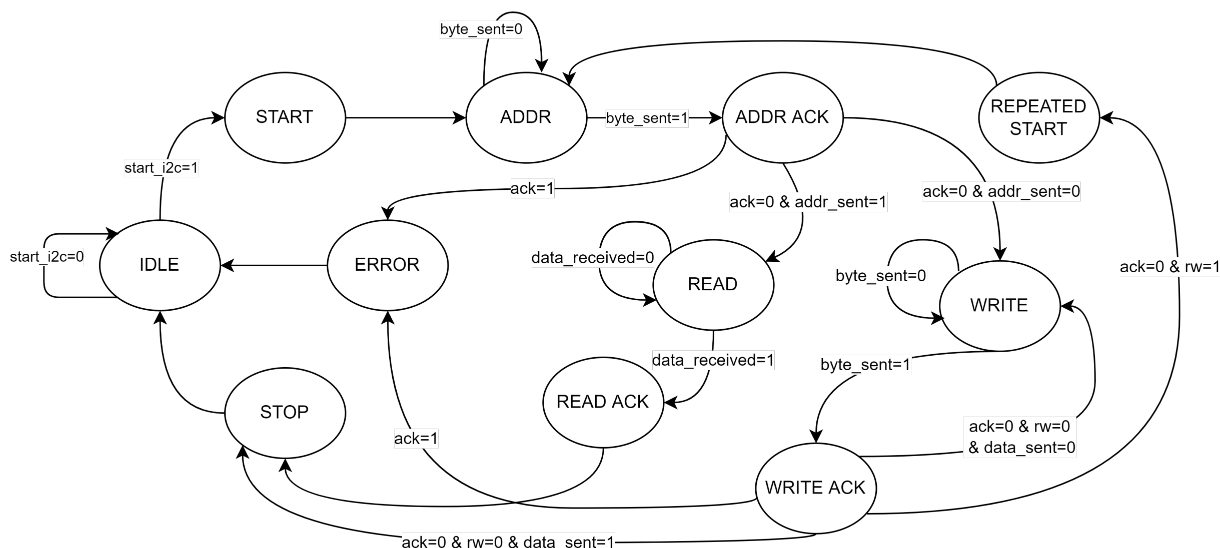


Figure 4.8: I2C control unit finite state machine.

To conclude, to configure and control this module, the control unit and the datapath have input signals which are controlled by software, through the use of the registers presented in Table 4.1. Therefore, to simplify the use of this peripheral, a device driver was developed with the most common functions such as I2C_Initialize(), I2C_Transmit() and I2C_Receive() (see Section 5.2.2).

Table 4.1: I2C configuration registers description.

Register	Bits	Access	Description
i2c_conf0	[31:16]	RW	Prescaler: Prescales the SCL clock line.
	[15:14]	-	Unused
	13	R	INT: This bit enables the I2C interrupt. If set, an interrupt will occur for each valid reception or transmission.
	12	RW	EN: This bit indicates that the core is enabled.
	11	R	Valid reception: This bit indicates that the data was successfully received and can be read from the core. This bit is cleared every time a start condition is emitted.
	10	R	Valid transmission: This bit indicates that the data was successfully sent. This bit is cleared every time a start condition is emitted.
	9	R	Error: This bit indicates an error in the communication.
	8	W	Start: This bit starts the state machine.
	[7:1]	W	Addr: Address of the slave.
	0	W	RW: This bit indicates if it is a read or write operation.
i2c_conf1	[31:0]	W	Data to send: Data which will be sent through the I2C bus.
i2c_conf2	[31:0]	W	Data to send: Data which will be sent through the I2C bus.
i2c_conf3	[31:24]	W	Data to send: Data which will be sent through the I2C bus.
	[23:16]	RW	Data size: Indicates the number of data bytes to be sent in a write operation (max of 9).
	[15:8]	R	Data received: Contains the data received from the I2C bus (only 1 byte can be read from the slave).
	[7:0]	-	Unused

4.4.2 Timer

A timer is a specialized clock used to measure time intervals, which can have several purposes, such as a stopwatch or a CPU's performance meter. The timer's internal clock frequency is directly related to its resolution and range. For example, a higher frequency will cause a greater timer resolution, however, the total time that can be measured will be smaller. Another factor for the range is the width of the counter register. If the width is big enough, the clock frequency can be high so that the timer can offer both high resolution, as well as a good range. As an example, a timer with a 64-bit counter register and a 150MHz clock can count up to 3950 years.

The module architecture is illustrated in Figure 4.9, composed of two units. To start, the control unit will emit a signal to the datapath, then a variable will be incremented until it matches the compare value. For a better understanding, the state machine of the control unit and the register table description are explained below.

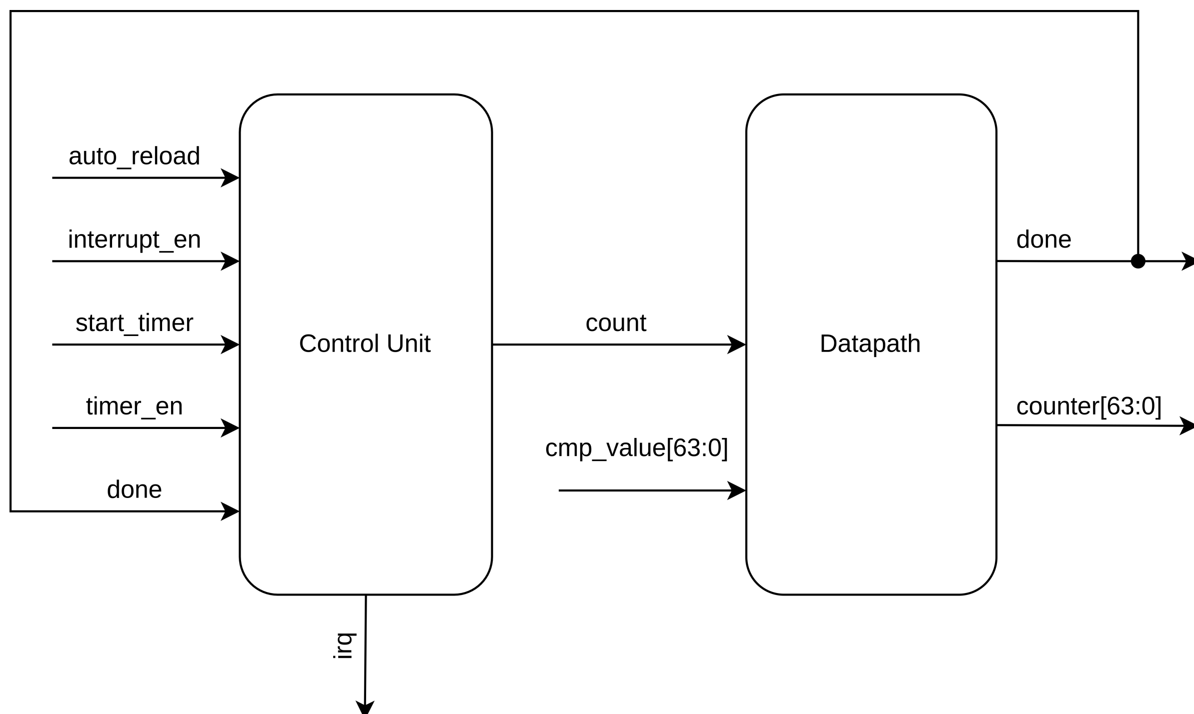


Figure 4.9: Timer module architecture.

Once in the **IDLE** state, a **start_timer**=1 will start the counting process in its respective state. Then, when the counter variable matches the desired value, the machine changes to its last state. Here, if the **auto_reload** value is 1, the counter variable is cleared and it goes back to counting. On the other hand, if **auto_reload**=0, it goes to idle and waits for a new start.

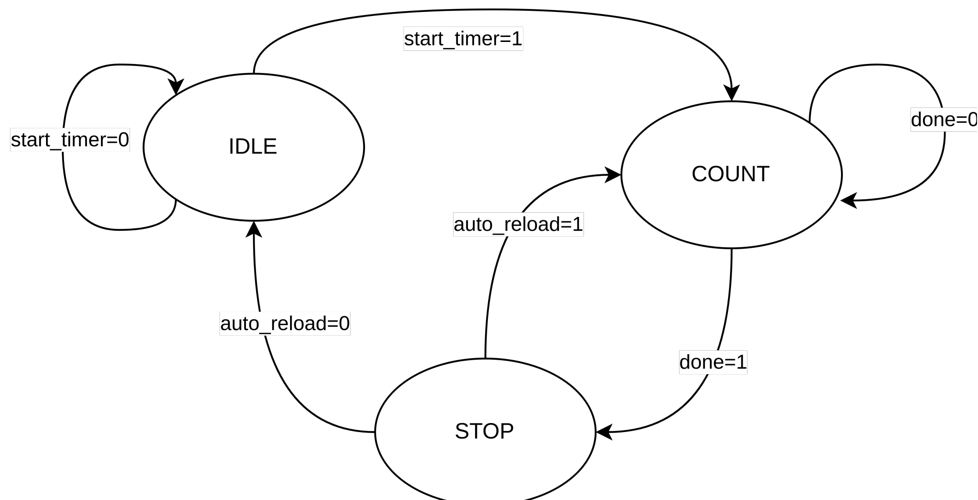


Figure 4.10: Timer control unit finite state machine.

At last, Table 4.1 presents the registers to control this timer peripheral. It includes a configuration register, basically composed of control and status bits, and four other registers for data transfer and reception to the device.

Table 4.2: Timer configuration registers description.

Register	Bits	Access	Description
timer_conf	31	W	Start: This bit starts the timer.
	30	W	EN: This bit enables the timer unit.
	29	W	INT: This bit enables the timer interrupt. If set, an interrupt will occur for each overflow.
	28	W	Auto Reload: This bit enables the auto-reload mode. If set, the timer will start again after each overflow.
	27	R	Overflow: This bit indicates that the timer reached the desired value. It is cleared by hardware when INT is enabled.
	[26:0]	-	Unused
timer_value_high	[31:0]	R	Timer value: First 32 bits of the value of the current counter (which is incremented by 1 each clock cycle).
timer_value_low	[31:0]	R	Timer value: Less significant 32 bits of the value of the current counter,
timer_cmp_high	[31:0]	W	Compare value: First 32 bits of the desired value to an overflow to occur.
timer_cmp_low	[31:0]	W	Compare value: Less significant bits of the desired value to an overflow to occur.

4.5 Advanced eXtensible Interface

Nowadays, one of the most important factors in SoC design is the on-chip interconnection between processors and their peripherals. Furthermore, due to the time to market and development costs, on-chip

bus architecture standards are often used, simplifying integration and reuse. There are several different bus architectures, such as Advanced Microcontroller Bus Architecture (AMBA), Avalon, Wishbone, and many others, each one developed from distinct companies and with its advantages and disadvantages. Of all the different offers, the chosen one was the Advanced eXtensible Interface (AXI) protocol, from AMBA, because of its hegemony among connectivity standards [77].

The Advanced eXtensible Interface (AXI) is a high-performance, high-frequency and robust bus interface developed by ARM. It is part of the AMBA specification, and it was introduced in 2003 with AXI3. Some years later, in 2011, an enhanced version of AMBA came out, AMBA4, which defined the AXI4, AXI4-Lite and AXI4-Stream protocols. AXI4-Lite is a reduced version of AXI4, designed for simple peripherals, whereas AXI4-Stream is intended for high-speed burst applications.

The generic architecture of the AXI protocol is composed of an AXI Master, Slave and Interconnect, as shown in Figure 4.1. The Master is the one that starts the transmission on the bus, whereas the Slave responds to the Master's requests. At last, the AXI Interconnect is responsible for connecting one or more AXI memory-mapped Master devices to one or more memory-mapped Slave devices [77].

4.6 Memory map

Figure 4.11 shows the memory map of EKKO. It contains 128KB of RAM, divided into 120KB of data and instruction memory, and 8KB reserved for the stack. Besides the memory, it also contains three peripherals, each one with a size of 4KB.

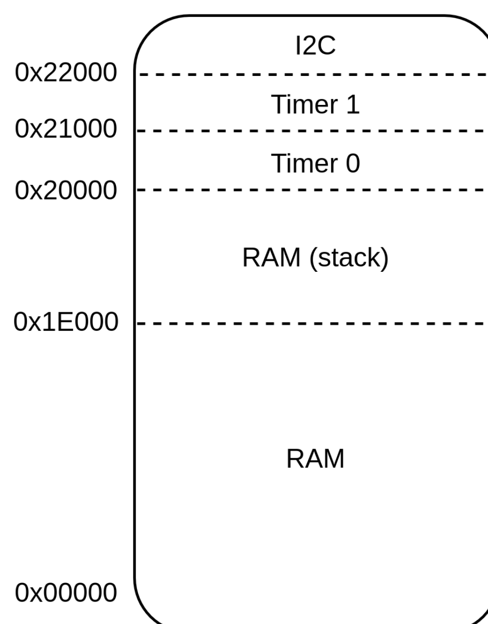


Figure 4.11: EKKO memory map.

Chapter 5: Software Development Kit

This chapter presents the microcontroller's Software Development Kit (SDK), which contains all the software tools needed to develop programs on the host machine (PC) and send them to the target (FPGA). Section 5.1 contains a description of some essential files, such as the makefile and the linker, whereas Section 5.2 focuses on the Board Support Package (BSP).

5.1 Support files

To program the target machine, some necessary files need to be used. These are responsible for compiling, linking, loading and debugging the executable file, and, to use them, the following software programs are required:

- **RISC-V GNU toolchain**: contains all the tools for RISC-V development (e.g. compiler, assembler, gdb, objcopy).
- **OpenOCD**: used to debug and load programs to the core
- **Visual Studio Code**: permits debugging through a GUI (optional)

The link to these tools, how to install them, and the SDK are all available in the author's GitHub page [78].

5.1.1 Makefile

Makefiles are special files that help build executable programs and manage projects automatically through the compilation process illustrated in Figure 5.1.

Initially, the c source files are compiled and converted into assembly code. Afterwards, these new files, together with other assembly source files, will be converted into object files, through the assembler. Ultimately, all the object files are linked together, generating the executable file, which can then be loaded into the microcontroller's core, through the use of OpenOCD.

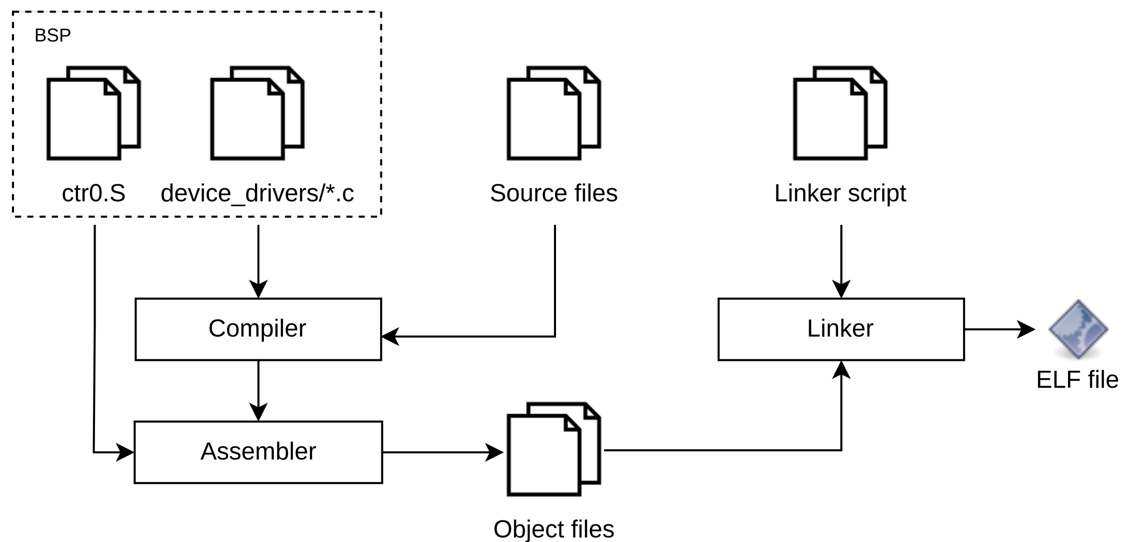


Figure 5.1: Compilation process.

Concerning the makefile content, the following code snippets present its most relevant parts. Starting by Code 5.1, ThreadX will only be compiled if the variable THREADX_EN is defined. In addition, the COMMON_SCRS, the PROGRAM_C and the EXTRA_SCRS, will always be compiled, which correspond to the BSP, main and application source files respectively.

Code 5.1: Makefile.

```
ifdef THREADX_EN
SRCS = $(COMMON_SRCS) $(PROGRAM_C) $(EXTRA_SRCS) $(TX_SRCS)
INCS = -I$(COMMONS_INCS) $(EXTRA_INCS) $(TX_INCS)
else
SRCS = $(COMMON_SRCS) $(PROGRAM_C) $(EXTRA_SRCS)
INCS = -I$(COMMONS_INCS) $(EXTRA_INCS)
endif

C_SRCS   = $(filter %.c, $(SRCS))
ASM_SRCS = $(filter %.S, $(SRCS))
```

Code 5.2 presents some important variables, such as the linker script, the C runtime initialization file (crt0), the compiler and its respective flags. Since the RISC-V instruction set is very flexible, the compiler must know which base ISA and extensions are being used, so the flags -march and -mabi must be called alongside it. The former is rv32im because the Integer Multiplication and Division is used. The latter defines the integer Application Binary Interface (ABI) in use, which is 32-bit for the ”int”, ”long” and pointer types, 8-bit for ”char”, 16-bit ”short” and lastly 64-bit ”long long”.

Code 5.2: Makefile.

```
ARCH = rv32im
CC = riscv32-unknown-elf-gcc
LINKER_SCRIPT ?= $(COMMON_DIR)/linker.ld
CRT ?= $(COMMON_DIR)/crt0.S
CFLAGS ?= -march=$(ARCH) -mabi=ilp32 -mcmodel=medany -Wall -g -O0\
        -fvisibility=hidden -nostdlib -nostartfiles $(PROGRAM_CFLAGS)
```

```
OBJS := ${C_SRCS:.c=.o} ${ASM_SRCS:.S=.o} ${CRT:.S=.o}
DEPS  = $(OBJS:%.o=%.d)
```

Lastly, Code 5.3 contains all the makefile rules. These serve to define when and how to remake certain files, which are called the targets, along with the target's prerequisites and the recipe for it. As an example, the rule for the $(PROGRAM).elf executable (the target), depends on all the object files and the linker script (the target's prerequisites), and its recipe is the command in the line below. This means that each time the object files or the linker script are updated, the compiler will be called with its respective flags (the recipe), creating a new elf file (see Figure 5.1).

Code 5.3: Makefile.

```
OUTFILES := $(PROGRAM).elf

all: $(OUTFILES)

$(PROGRAM).elf: $(OBJS) $(LINKER_SCRIPT)
	$(CC) $(CFLAGS) -T $(LINKER_SCRIPT) $(OBJS) -o $@ $(LIBS)

%.dis: %.elf
	$(OBJDUMP) -SD $^ > $@

%.o: %.c
	$(CC) $(CFLAGS) -MMD -c $(INCS) -o $@ $<

%.o: %.S
	$(CC) $(CFLAGS) -MMD -c $(INCS) -o $@ $<

clean:
	$(RM) -f $(OBJS) $(DEPS) $(PROGRAM).elf $(PROGRAM).dis
```

5.1.2 Linker

The linker is a program that gathers all the objects files from the assembler and generates an executable (see Figure 5.1). To work, it requires a linker script to inform how the memory map should be organised.

Code 5.4 has part of the linker script used for a simple overview, whereas Appendix A.2 contains the entire file. As can be seen, the first part describes the starting address and the size of both the ROM and the stack, which match the addresses of the microcontroller's memory map (see Figure 4.11). The SECTIONS command contains the memory layout of the output file. For example, the code and constants (.text), the uninitialised data (.bss) and the initialised data (.data) all go to the ROM (because the core has a von Neumann architecture).

Code 5.4: Linker script "MEMORY" and "SECTIONS" commands.

```
MEMORY {
    rom         : ORIGIN = 0x000000000, LENGTH = 0x1E000 /* 120 kB */
    stack       : ORIGIN = 0x00001E000, LENGTH = 0x02000 /*   8 kB */
}
```

```
SECTIONS {
    .vectors : {
    } > rom

    .text  : {
    }  > rom

    .rodata : {
    } > rom

    .shbss : {
    } > rom

    .bss : {
    } > rom

    .sdata : {
    } > rom

    .data : {
    } > rom

    .stack (NOLOAD): {
    } > stack
```

5.1.3 OpenOCD

According to the official website, Open On-Chip Debugger (OpenOCD) aims to provide debugging, in-system programming and boundary-scan testing for embedded target devices [79]. Nonetheless, as explained in Section 4.3, other tools are also required to program and debug the RISC-V core.

In a nutshell, OpenOCD is responsible for receiving high-level debug commands from gdb/telnet, translating them into CPU specific JTAG transactions, and sending these transactions through a JTAG dongle/debug adapter to the target device. Figure 5.2 helps in understanding how it works. First, it needs two configuration files, one for the debug adapter (JLink) and another for the target device core (Ibex). Afterwards, a connection between the host and the target is established, and OpenOCD will wait for a gdb or telnet connection to process the received debug commands.

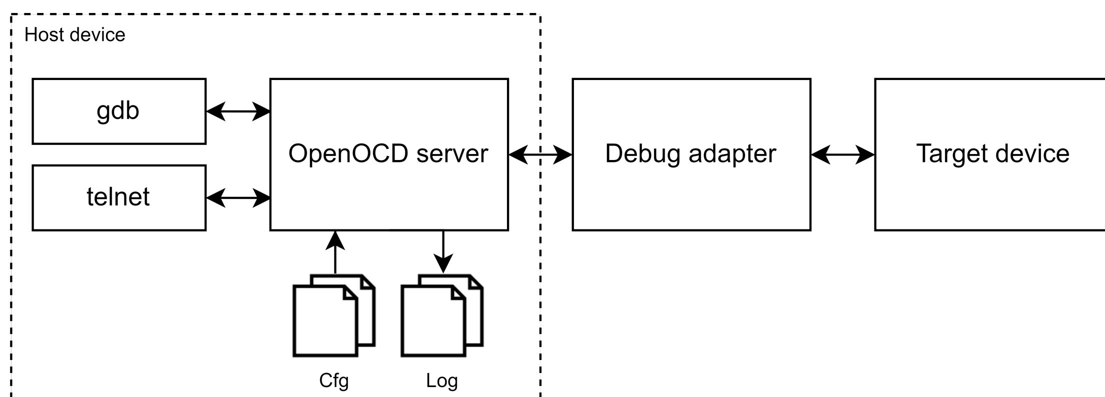


Figure 5.2: OpenOCD block diagram.

Regarding the Ibex core, despite OpenOCD having several configuration files for different adapters and processors, including the JLink debug probe, a .cfg file for the target device didn't exist. Thus, Code 5.5 contains the Ibex configuration file used to run OpenOCD.

Code 5.5: OpenOCD Ibex configuration file.

```
reset_config none
adapter speed 3000
transport select jtag

set _CHIPNAME riscv

jtag newtap $_CHIPNAME cpu -irlen 5 -expected-id 0x0494284d

set _TARGETNAME $_CHIPNAME.cpu
target create $_TARGETNAME riscv -chain-position
              $_TARGETNAME -rtos hwthread

gdb_report_data_abort enable
gdb_report_register_access_error enable

riscv set_reset_timeout_sec 2000
riscv set_command_timeout_sec 2000
riscv set_prefer_sba on

scan_chain
init
halt
```

With the two configuration files, OpenOCD can be started with the following command:

```
$ openocd -f openocd/jlink.cfg -f openocd/ibex.cfg
```

Then, riscv32-unknown-elf-gdb can be used to program and debug the Ibex core. Furthermore, Appendix A.3 contains a launch.json file, developed during this dissertation, that enables the use of a graphical debugging mechanism from Visual Studio Code, turning it easier to program and debug the core through the usage of breakpoints and variable watchers.

5.2 Board Support Package

The Board Support Package (BSP) is a software layer used to boot and run an embedded system. It contains all the essential hardware-specific routines needed to work with the microcontroller, such as the start-up code and the peripherals device drivers.

5.2.1 C Runtime Initialization

C Runtime Initialization (also known as crt0), is a group of start-up functions that initialize the processor before calling the program's main routine.

The following code snippets separate the content of the crt0.S file in its main tasks. Code 5.6 specifies three different handlers: the systick_timer_handler, the default_exc_handler and the reset_handler. The first one jumps into a ThreadX kernel function (see Code 6.4) and is called when there is an interrupt in timer 0 (which works as a systick timer when the RTOS is available). The following one is an infinite loop and it's called for most exceptions (see Code 5.8). Lastly, when a reset occurs, all the registers, as well as the mepc, mstatus and mie CSRs will be set to 0, along with the stack pointer initialization.

Code 5.6: crt0 file.

```
.section .text

systick_timer_handler:
    j _tx_timer_interrupt_handler

default_exc_handler:
    jal x0, default_exc_handler

reset_handler:
    mv   x1, x0
    mv   x2, x1
    ...
    mv x31, x1

    csrwi mepc,    0
    csrwi mstatus, 0
    csrwi mie,     0

    la   x2, _stack_start
```

The following code snippet is responsible for clearing the .bss section (because it contains all the uninitialised data) and jumping to the program's main function. Firstly, both _bss_start and _bss_end addresses, which are defined in the linker script (see Appendix A.2), are loaded into the register x26 and x27, respectively. Then, in case there is uninitialised data in the .bss section, all of it will be set to 0 in the zero_loop function. Finally, the program jumps to the main function with x10 = x11 = 0 since these are used as function parameters accordingly to the RISC-V ABI, representing the argc and argv parameters.

Code 5.7: crt0 file.

```
_start:
.global _start
    la x26, _bss_start
    la x27, _bss_end
    bge x26, x27, zero_loop_end

zero_loop:
    sw x0, 0(x26)
    addi x26, x26, 4
    ble x26, x27, zero_loop
    zero_loop_end:

main_entry:
    addi x10, x0, 0
    addi x11, x0, 0
    jal x1, main
```

At last, Code 5.8 presents the processor's vector table, containing all the handlers for the CPU interruptions and exceptions, such as resets, illegal instructions and more.

Code 5.8: crt0 file.

```
.section .vectors, "ax"
.option norvc;

.org 0x00
.rept 7
jal x0, default_exc_handler
.endr

/* Systick handler */
jal x0, systick_timer_handler

.rept 23
jal x0, default_exc_handler
.endr

/* Reset vector */
.org 0x80
jal x0, reset_handler

/* Illegal instruction exception */
.org 0x84
jal x0, default_exc_handler

/* Ecall handler */
.org 0x88
jal x0, default_exc_handler
```

5.2.2 Device drivers

A device driver is a software module that controls an external I/O device. It provides a software interface to the peripheral, which permits a computer program to control it without knowing any specific details about its hardware. In this context, the following pages contain some relevant code snippets concerning the device drivers for the I2C and timer peripherals.

Timer

First, as can be seen in the code below, a structure for managing the timer registers was created (see Table 4.2). This way, controlling the peripheral registers becomes a lot easier and more modular since there is no need to work with pointers constantly.

Code 5.9: Timer structure

```
typedef struct {
    union {
        struct {
            uint32_t reserved    : 27;
            uint8_t  overflow    : 1;
            uint8_t  auto_reload : 1;
            uint8_t  interrupt   : 1;
            uint8_t  enable      : 1;
            uint8_t  start       : 1;
        } bit;
        uint32_t config_register;
    } CONF_REG;             /* (TIMER_BASE_ADDR + 0x00) */
    uint32_t DATA_REG_HIGH; /* (TIMER_BASE_ADDR + 0x04) */
    uint32_t DATA_REG_LOW;  /* (TIMER_BASE_ADDR + 0x08) */
    uint32_t CMP_REG_HIGH;  /* (TIMER_BASE_ADDR + 0x0C) */
```

```
    uint32_t CMP_REG_LOW;    /* (TIMER_BASE_ADDR + 0x10) */
} TIMER_Type;
```

Afterwards, a structure for initialising the timer was also created in the header file. In addition, beyond the function prototypes for handling the driver, which are explained in Code 5.11, two extern variables for both timers were declared. These will be passed to the device drivers' functions, so the respective registers are manipulated accordingly.

Code 5.10: Timer driver header file.

```
typedef uint32_t TIMER_ValueType;

typedef struct{
    uint64_t compare_value;
    uint8_t  interrupt_enable;
    uint8_t  auto_reload;
} TIMER_InitType;

TIMER_ValueType TIMER_Initialize(TIMER_Type *timer,
                                 TIMER_InitType init_values);
TIMER_ValueType TIMER_Start(TIMER_Type *timer);
uint64_t        TIMER_Get_Counter(TIMER_Type *timer);
TIMER_ValueType TIMER_Stop(TIMER_Type *timer);
TIMER_ValueType TIMER_Wait(TIMER_Type *timer);

extern TIMER_Type *timer0;
extern TIMER_Type *timer1;
```

The following code contains the first part of the driver source file. Initially, the previously declared timer variables are initialised with their respective addresses (see the memory map in Figure 4.11). Then, the implementation of the TIMER_Initialize is presented, where all the registers are cleared, followed by their respective initialisation.

Code 5.11: Timer driver source file.

```
#include "timer.h"

TIMER_Type *timer0 =  ((TIMER_Type *) TIMER0_BASE); /* 0x20000 */
TIMER_Type *timer1 =  ((TIMER_Type *) TIMER1_BASE); /* 0x21000 */

static inline void TIMER_Clear_Regs(TIMER_Type *timer) {
  timer->CMP_REG_HIGH              = 0;
  timer->CMP_REG_LOW               = 0;
  timer->CONF_REG.config_register = 0;
  timer->DATA_REG_HIGH             = 0;
  timer->DATA_REG_LOW              = 0;
}

TIMER_ValueType TIMER_Initialize(TIMER_Type *timer,
                                 TIMER_InitType init_values) {
  TIMER_Clear_Regs(timer);

  timer->CMP_REG_HIGH = (uint32_t) (init_values.compare_value >> 32);
  timer->CMP_REG_LOW = (uint32_t) init_values.compare_value;
  timer->CONF_REG.bit.auto_reload = init_values.auto_reload;
  timer->CONF_REG.bit.interrupt = init_values.interrupt_enable;
  timer->CONF_REG.bit.enable = 1;

  return TIMER_VALUE_OK;
}
```

At last, Code 5.12 contains the functions' implementation for controlling this peripheral. These include starting and stopping the timer, waiting for it to reach its desired value and reading how many cycles passed since it started counting.

Code 5.12: Timer driver source file.

```
TIMER_ValueType TIMER_Start(TIMER_Type *timer) {
  timer->CONF_REG.bit.start = 1;
  return TIMER_VALUE_OK;
}

uint64_t TIMER_Get_Counter(TIMER_Type *timer) {
  return (((uint64_t) timer->DATA_REG_HIGH) << 32) |
          ((uint64_t) timer->DATA_REG_LOW);
}

TIMER_ValueType TIMER_Stop(TIMER_Type *timer) {
  timer->CONF_REG.bit.enable = 0;
  return TIMER_VALUE_OK;
}

TIMER_ValueType TIMER_Wait(TIMER_Type *timer) {
  while(timer->CONF_REG.bit.overflow == 0);
  timer->CONF_REG.bit.overflow = 0;
  return TIMER_VALUE_OK;
}
```

I2C

The device driver for the I2C peripherals follows the same logic as the timer. Therefore, only the register and init structures and the implementations of the functions, which are a little more complex than the previous ones, will be presented.

Code 5.13 includes the I2C_Type, for managing registers and the I2C_InitType, composed of an operation mode and an interrupt_enable variable, which will be used for the I2C initialisation.

Code 5.13: I2C structure.

```
typedef struct {
    union {
        struct {
            uint8_t  addr;
            uint8_t  start              : 1;
            uint8_t  error              : 1;
            uint8_t  valid_transmission : 1;
            uint8_t  valid_reception    : 1;
            uint8_t  enable             : 1;
            uint8_t  interrupt          : 1;
            uint8_t  reserved           : 2;
            uint16_t prescaler;
        } bit;
        uint32_t config_register0;
    } CONF0_REG;           /* (I2C_BASE_ADDR + 0x00) */
    uint32_t CONF1_REG;    /* (I2C_BASE_ADDR + 0x04) */
    uint32_t CONF2_REG;    /* (I2C_BASE_ADDR + 0x08) */
    union {
        struct {
            uint8_t  reserved;
            uint8_t  data_received;
            uint8_t  data_size;
            uint8_t  data_to_send;
```

```
        } bit;
        uint32_t config_register3;
    } CONF3_REG;          /* (I2C_BASE_ADDR + 0x0C) */
} I2C_Type;
```

To set up the peripheral, the following function just needs to receive the respective device and its initialisation values. Then, according to the operating mode, the correct prescaler value to accomplish the desired SCL frequency will be calculated based on the clock speed.

Code 5.14: I2C Initialize function.

```
I2C_ValueType I2C_Initialize(I2C_Type *i2c, I2C_InitType init_values) {
  I2C_Clear_Regs(i2c);

  i2c->CONF0_REG.bit.prescaler = SYS_CLK_FREQ /
                                  (2*init_values.OperationMode*1000)-1;
  i2c->CONF0_REG.bit.enable    = 1;
  i2c->CONF0_REG.bit.interrupt = init_values.interrupt_enable;

  return I2C_VALUE_OK;
}
```

Moreover, to transmit data through the I2C protocol, the following function needs to be used. First, it clears the registers that are going to be used. Afterwards, it checks if both the device address and the data size are valid and, in case they are, fills the respective registers with the data to send. Ultimately, the communication is started, and it waits for the valid transmission bit to be set to one.

Code 5.15: I2C Transmit function.

```
I2C_ValueType I2C_Transmit(I2C_Type *i2c, uint8_t addr, uint8_t *data,
                           uint8_t data_size) {
  i2c->CONF0_REG.bit.addr = 0;
  i2c->CONF1_REG = 0;
  i2c->CONF2_REG = 0;
  i2c->CONF3_REG.bit.data_size = 0;

  if(addr & 0x1) {
    return I2C_VALUE_INVALID_ADDR;
  }

  if(data_size > 9) {
    return I2C_VALUE_INVALID_SIZE;
  }

  i2c->CONF1_REG = (data[0] << 24) | (data[1] << 16) |
                   (data[2] << 8) | data[3];
  i2c->CONF2_REG = (data[4] << 24) | (data[5] << 16) |
                   (data[6] << 8) | data[7];
  i2c->CONF3_REG.bit.data_to_send = data[8];
  i2c->CONF3_REG.bit.data_size    = data_size;
  i2c->CONF0_REG.bit.addr         = addr;

  i2c->CONF0_REG.bit.start        = 1;

  while(i2c->CONF0_REG.bit.valid_transmission == 0){}
  return I2C_VALUE_OK;
}
```

Code 5.16 contains the last function of the driver, responsible for receiving data. As usual, the used registers are set to zero. Then, if the address is ok, both the device and register addresses are filled

in. Finally, the start bit is sent, and the CPU will wait for the reception bit, which signalises that the communication has ended and so the data can be read from the respective register.

Code 5.16: I2C Receive function.

```
I2C_ValueType I2C_Receive(I2C_Type *i2c, uint8_t addr, uint8_t *data) {
  i2c->CONF0_REG.bit.addr = 0;
  i2c->CONF1_REG = 0;

  if(!(addr & 0x1)) {
    return I2C_VALUE_INVALID_ADDR;
  }

  i2c->CONF1_REG   = (data[0] << 24);
  i2c->CONF0_REG.bit.addr   = addr;
  i2c->CONF0_REG.bit.start = 1;

  while(i2c->CONF0_REG.bit.valid_reception == 0){}

  data[1] = i2c->CONF3_REG.bit.data_received;
  return I2C_VALUE_OK;
}
```

Chapter 6: Azure RTOS ThreadX

This chapter concerns the chosen OS for this system, Azure RTOS ThreadX. The first section will present an overview of how it works. Then, the remaining sections will focus on how ThreadX was ported to the Ibex core and its validation.

6.1 ThreadX overview

With more than 8.2 billion deployments, ThreadX is Microsoft's real-time operating system designed for embedded and IoT applications. This OS has a small footprint, is fast, contains multicore support and also several advanced features such as its picokernel architecture, preemption-threshold scheduling, and system event tracing [80]. The following pages give an overview of the functional components of ThreadX's kernel, which are relevant to this work. These include execution overview, ThreadX initialisation and thread execution.

6.1.1 Execution overview

As shown in Figure 6.1, ThreadX supports four different types of program execution: Initialisation, Thread Execution, Interrupt Service Routine (ISR), and Application Timers. ISRs and applications timers are simple. The only difference between them is that in the latter, the hardware implementation is hidden from the application. The kernel initialisation and thread execution are explained further ahead with more detail.

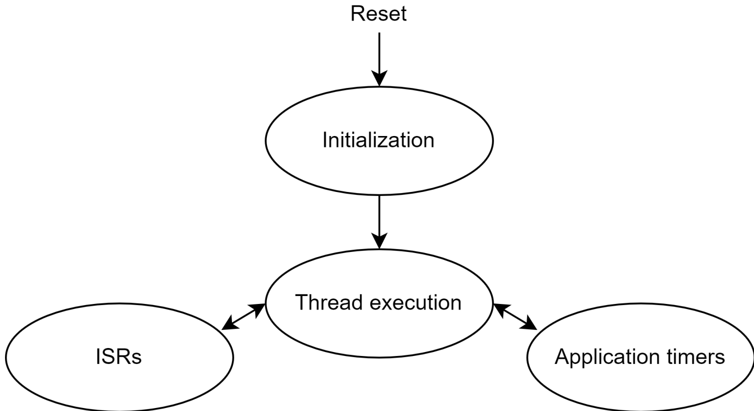


Figure 6.1: ThreadX execution overview (adapted from [80]).

6.1.2 Initialization

Understanding the initialisation of an RTOS is crucial, especially when there is a need to port it to a different architecture. Figure 6.2 illustrates the kernel's initialisation process, which is composed of four steps. First, when a reset occurs, the processor jumps to the address of the system's entry point, which is usually written in assembly and architecture-specific. Then, after the low-level setup, control transfers to the development tool initialisation, which normally contains global and static C variables. Next, the program jumps to the main function, where the tx_kernel_enter() will be called. This is the entry function into ThreadX, where several internal data structures are initialised and the application's definition function tx_application_define is invoked. tx_application_define, as the name implies, defines all the initial ThreadX objects, such as threads, queues, mutexes and semaphores. Lastly, when this function returns its execution to the tx_kernel_enter, control will be transferred to the thread scheduling loop.

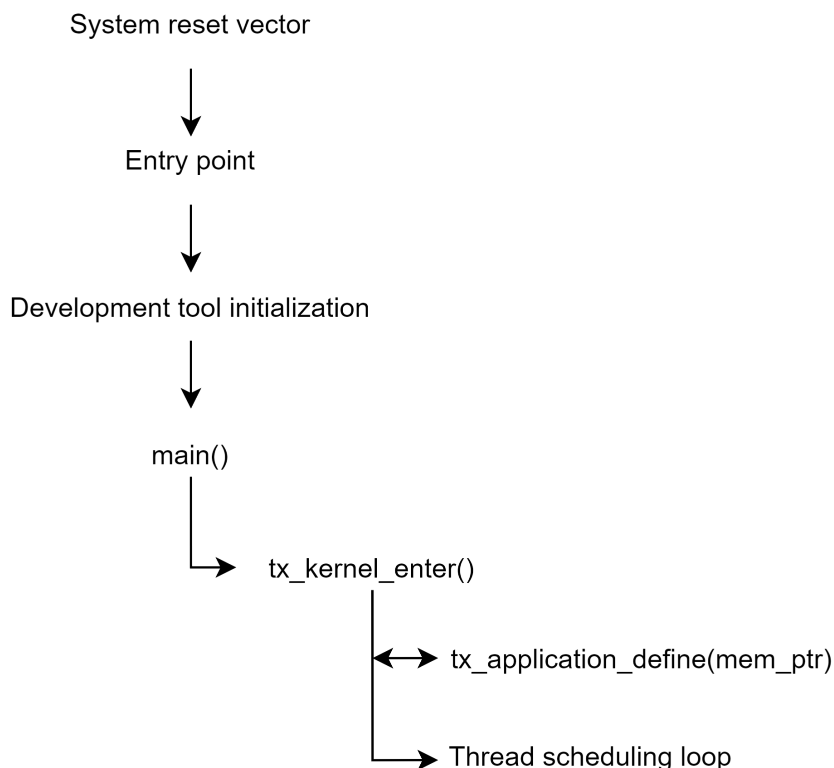


Figure 6.2: ThreadX initialization (adapted from [80]).

6.1.3 Thread execution

In every RTOS, some of the most important activities are thread execution and scheduling policy. For the ThreadX case, after a thread is running, it can only be stopped by an ISR or an application timer (see Figure 6.1). Furthermore, threads support five different states: ready, suspended, executing, terminated

and completed. Figure 6.3 illustrates how threads transition between these distinctive states.

When a thread is created, it can be configured to be in a ready or suspended state. For the first case, it means the thread is ready for execution, however, it will only run when it is the highest priority thread in this state. When this happens, its state changes to executing, and it will keep running until a higher priority thread becomes ready. Furthermore, when in the suspended state, threads are not eligible for execution. This can happen for a lot of different reasons, such as messages queues, mutexes, event flags, and thread suspension. Once in this state, a thread can only be ready to execute if the cause of suspension is removed. Concerning the last two stages, a thread is completed when its execution flow returns from its entry function, whereas a thread only terminates if the tx_thread_terminate service is called. Threads in both completed and terminated states cannot execute again.

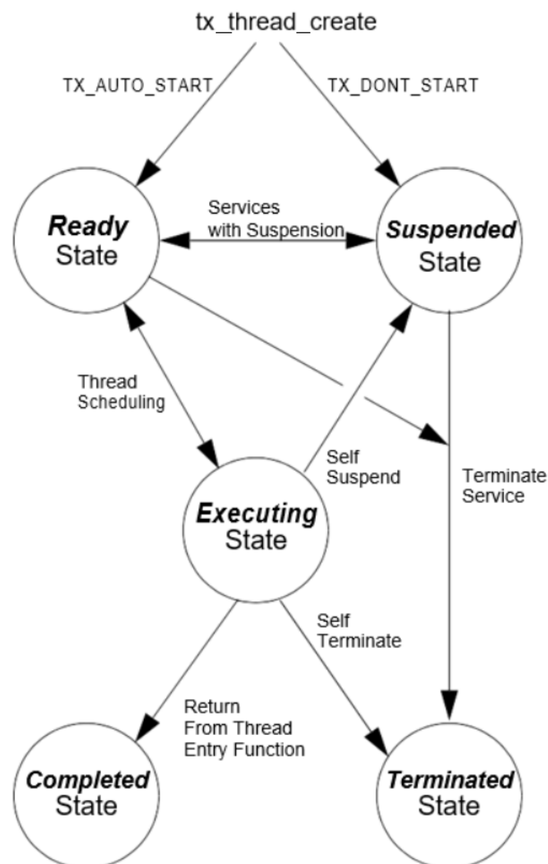


Figure 6.3: ThreadX thread state transition (from [80]).

Another important topic when talking about thread execution is its scheduling policy. ThreadX schedules its threads based on their priority. There are 32 priority levels by default, where 0 is the highest priority and 31 the lowest. Regarding the scheduling models, ThreadX supports two round-robin algorithms in the case of multiple threads in the ready state with the same priority. The first relies on the cooperative call of tx_thread_relinquish, which will give the processor to all other ready threads with the same priority before

the caller executes again. The other form is time-slicing, which specifies the number of timer ticks that a thread should run before giving up its control. Ultimately, when the threads in the ready state have different priorities, ThreadX uses preemption. This policy consists of interrupting an executing thread to run a high priority one until it's finished, which will transfer the control back to the place where preemption occurred.

6.2 ThreadX port

Before explaining how the port was done, it is important to clarify the directory layout of ThreadX. As can be seen in Figure 6.4, there are three main directories. The first one, the **common** folder, contains the implementation of all ThreadX APIs as well as some utility features. The **ports** directory consists of the kernel for several different architectures and compilers and it is exclusively written in assembly, except for one header file, tx_port.h, which will be explained further ahead. Lastly, the **samples** folder contains a demo file to validate if the OS is working properly.

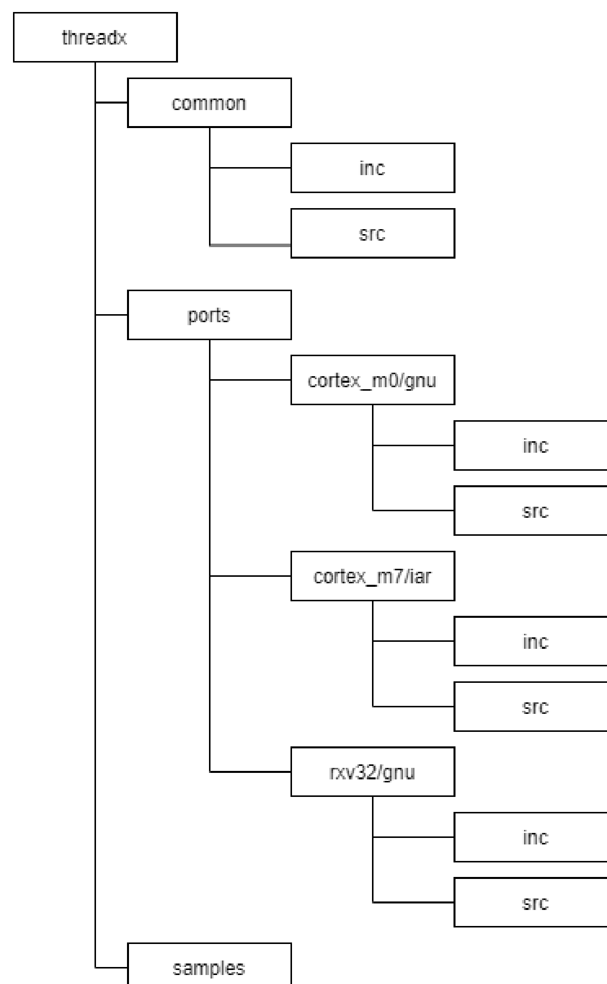


Figure 6.4: ThreadX directory layout.

The way that ThreadX is designed turns its support for multiple architectures much easier than in other similar RTOS. In ThreadX, the API is completely independent of the hardware below. Thus, to port it to a different architecture, one must change only the kernel files and the tx_port header inside the ports folder.

Currently, ThreadX distribution already includes a port to the RISC-V architecture, which was very helpful to this work. However, it is designed to execute under the IAR Windows-based RISC-V simulator, and, as a result, it uses the IAR compiler. Therefore, the first step of this port was to adapt the assembly code to the GNU compiler, which basically consisted of adapting some macros. Refer to Code 6.1 and 6.2 to understand some differences between the assembly code for IAR and the one for GNU.

Code 6.1: Assembly code example for the RISC-V IAR compiler.

```
    RETURN_MASK      DEFINE          0x0000000F
    SET_SR_MASK      DEFINE          0xFFFFFFF0

    SECTION `.text`:CODE:REORDER:NOROOT(2)
    CODE

    PUBLIC   _tx_thread_interrupt_control
_tx_thread_interrupt_control:
    csrr     t0, mstatus
    mv       t1, t0                  ; Save original mstatus for return
    li       t2, SET_SR_MASK         ; Build set SR mask
    and      t0, t0, t2              ; Isolate interrupt lockout bits
    or       t0, t0, a0              ; Put new lockout bits in
    csrw     mstatus, t0
    andi     a0, t1, RETURN_MASK     ; Return original mstatus.
    ret
    END
```

Code 6.2: Assembly code example for the RISC-V GNU compiler.

```
    RETURN_MASK = 0x0000000F
    SET_SR_MASK = 0xFFFFFFF0

    .section .text

    .global   _tx_thread_interrupt_control
_tx_thread_interrupt_control:
    csrr     t0, mstatus
    mv       t1, t0                  ; Save original mstatus for return
    li       t2, SET_SR_MASK         ; Build set SR mask
    and      t0, t0, t2              ; Isolate interrupt lockout bits
    or       t0, t0, a0              ; Put new lockout bits in
    csrw     mstatus, t0
    andi     a0, t1, RETURN_MASK     ; Return original mstatus.
    ret
```

After porting the code to the GNU compiler, the next stage was to edit the port.h header. This file contains data type definitions and macros that make ThreadX kernel function identically on a variety of different instruction sets, so it is architecture-specific. As an example, Code 6.3 presents some code parts of this file to the RISC-V architecture, such as the data types size and the tasks maximum priority (refer to Appendix C for the entire file).

Code 6.3: tx_port.h file.

```
#ifndef TX_PORT_H
#define TX_PORT_H

/* Include prototypes for memset */
#include "bsp.h"

/* Define ThreadX basic types for this port */
#define VOID                void
typedef char                CHAR;
typedef unsigned char       UCHAR;
typedef int                 INT;
typedef unsigned int        UINT;
typedef long                LONG;
typedef unsigned long       ULONG;
typedef short               SHORT;
typedef unsigned short      USHORT;

/* Define the priority levels for ThreadX */
#ifndef TX_MAX_PRIORITIES
#define TX_MAX_PRIORITIES   32
#endif

/* Define the minimum stack for a ThreadX thread on this processor */
#ifndef TX_MINIMUM_STACK
#define TX_MINIMUM_STACK   512
#endif

/* Define various constants for the ThreadX RISC-V port.  */
#define TX_INT_DISABLE 0x00000000
#define TX_INT_ENABLE  0x00000008

#define TX_INLINE_INITIALIZATION

unsigned int _tx_thread_interrupt_control(unsigned int new_posture);

#define TX_INTERRUPT_SAVE_AREA register INT interrupt_save;

#define TX_DISABLE interrupt_save = _tx_thread_interrupt_control(
    TX_INT_DISABLE);
#define TX_RESTORE _tx_thread_interrupt_control(interrupt_save);

#endif
```

The final step of this port was to set up the tx_initialize_low_level(), which is called during the initialisation of the ThreadX's kernel. This function is inside one of the kernel architecture-specific assembly files and it can generally be separated into two parts, _tx_initialize_low_level and _tx_timer_interrupt_handler, as can be seen in Code 6.4. The first one finds the first available RAM address, saves the system stack pointer for later ISR processing and sets up the systick timer. The other part, as the name indicates, is the handler for the periodic timer and it will be called from the corresponding interrupt in the system's vector table (see Code 5.8).

Code 6.4: tx_initialize_low_level.S file.

```
    .section .data
    .global  __tx_free_memory_start
__tx_free_memory_start:

    .section .text
    .global  _tx_initialize_low_level
_tx_initialize_low_level:
    sw     sp, _tx_thread_system_stack_ptr, t0  # Save system stack pointer
    la     t0, __tx_free_memory_start           # Pickup first free address
```

```
    sw     t0, _tx_initialize_unused_memory, t1 # Save unused memory address
    nop
    li     t0, 0x80
    csrw   mie, t0                              # Enable systick interrupt
    csrwi  mstatus, 0x8                         # Enable global interrupt
    ret

    .global _tx_timer_interrupt_handler
_tx_timer_interrupt_handler:
    addi   sp, sp, -128                # Allocate space for all registers
    sw     x1, 0x70(sp)                # Store RA
    call   _tx_thread_context_save     # Call ThreadX context save
    call   _tx_timer_interrupt         # Call timer interrupt handler
    j      _tx_thread_context_restore  # Jump to ThreadX context restore
```

Chapter 7: Tests and results

Throughout the microcontroller and the SDK development, some tests were elaborated to validate certain implementation stages. Therefore, this section presents an overview of the most relevant unit tests as well as a final system test to verify that everything works seamlessly together.

7.1 Unit tests

The primary goal of unit testing is to take the smallest piece of a system, isolate it from the rest, and determine whether it behaves as expected or not. This is a crucial step to avoid problems when integrating all the modules because it minimises the scope of those errors.

7.1.1 Ibex

Since the first implementation step of this dissertation was to port the Ibex core to the Arty FPGA, it was also the first unit to be tested. To do so, a few assembly files were created to validate the different parts of the CPU. All of these can be seen in Appendix D.1, however, to maintain this chapter more concise, only some parts of the test concerning the ALU is going to be analysed.

Code 7.1 contains a piece of the test_alu.s file, which after compiled was loaded into the memory connected to Ibex.

Code 7.1: ALU assembly test file.

```
main:
    /* Test ADDI */
    addi x1 , x0,    1000  /* x1  = 1000 0x3E8 */
    addi x2 , x1,    2000  /* x2  = 3000 0xBB8 */
    addi x3 , x2,   -1000  /* x3  = 2000 0x7D0 */
    addi x4 , x3,   -2000  /* x4  = 0    0x000 */
    addi x5 , x4,    1000  /* x5  = 1000 0x3E8 */
    addi x6 , x5,    2000  /* x6  = 3000 0xBB8 */
    addi x7 , x6,   -1000  /* x7  = 2000 0x7D0 */
    addi x8 , x7,   -2000  /* x8  = 0    0x000 */
    addi x9 , x8,    1000  /* x9  = 1000 0x3E8 */
    addi x10, x9,    2000  /* x10 = 3000 0xBB8 */
    addi x11, x10,  -1000  /* x11 = 2000 0x7D0 */
    addi x12, x11,  -2000  /* x12 = 0    0x000 */
    addi x13, x12,   1000  /* x13 = 1000 0x3E8 */
    addi x14, x13,   2000  /* x14 = 3000 0xBB8 */
    addi x15, x14,  -1000  /* x15 = 2000 0x7D0 */

    /* Test Positive numbers */
    slti x15, x1,   1200  /* x15 = 1 */
    slti x14, x2,   1200  /* x14 = 0 */
```

```
/* Test Negative numbers */
addi x3,  x0, -1000
slti x13, x3, -1200   /* x13 = 0 */
slti x12, x3, -900    /* x12 = 1 */

/* Test Unsigned */
sltiu x11, x1,  1200   /* x11 = 1 */
sltiu x10, x2,  1200   /* x10 = 0 */

/* Test XOR */
addi x1, x0, 0x7FF
addi x2, x0, 0x70F
addi x3, x0, 0x0F0

xori x15, x1, -1      /*  x15 = 0xFFFFF800 */
xori x14, x2, -1      /*  x14 = 0xFFFFF8F0 */
xori x13, x3, -1      /*  x13 = 0xFFFFFF0F */

/* Test OR */
addi x1, x0, 0x7FF
addi x2, x0, 0x70F
addi x3, x0, 0x0F0

ori x15, x1, -1       /*  x15 = 0xFFFFFFFF */
ori x14, x2, -1       /*  x14 = 0xFFFFFFFF */
ori x13, x3, -1       /*  x13 = 0xFFFFFFFF */
ori x12, x1,  0       /*  x12 = 0x000007FF */
ori x11, x2,  0       /*  x11 = 0x0000070F */
ori x10, x3,  0       /*  x10 = 0x000000F0 */
```

Furthermore, for an easier analysis, in front of each assembly line is a comment containing the expected register value after the respective ALU operation.

Figure 7.1 illustrates the simulation results concerning the code snippet above. Despite not being possible to see the content of some variables in the image, by examining the registers' values, one can see that they match what was expected from the ALU.

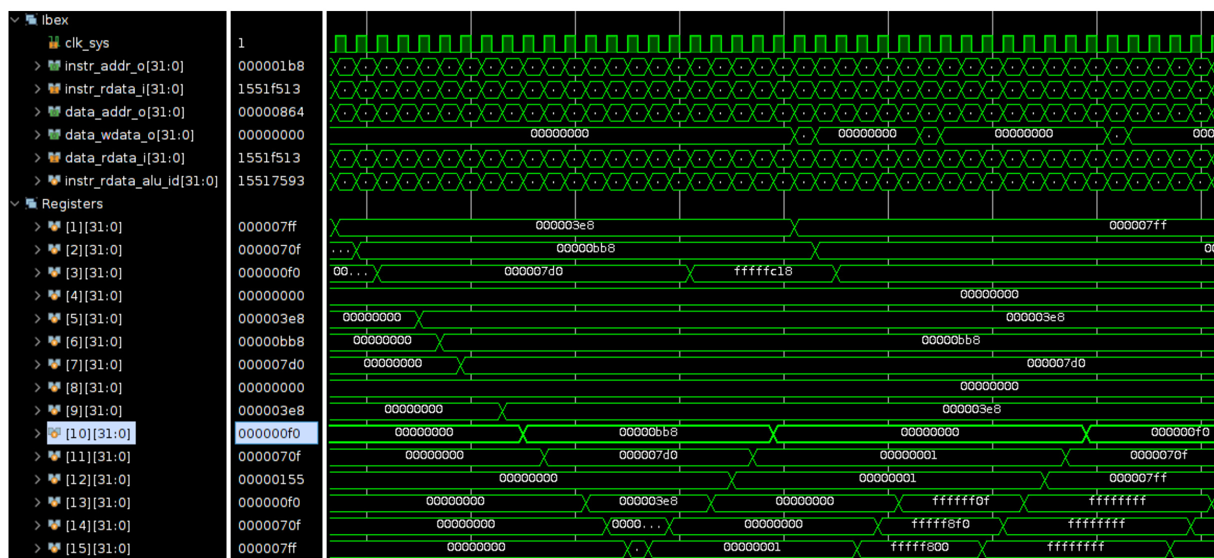


Figure 7.1: ALU test simulation.

7.1.2 AXI

To test the AXI bus properly, all of the following components need to be validated: AXI Master, AXI Slave and AXI Interconnect. Thus, a simple C program (Code 7.2) was created, responsible for writing and reading from the AXI memory region.

The code is very straightforward, however, it is enough to verify that the bus architecture is working. First, the AXI Master should understand that the address belongs to the AXI memory region and not the RAM. Then, according to the data address, the Interconnect should identify to which device the core wants to access. Lastly, the Slave is also validated since it should respond appropriately to the Master's requests.

Code 7.2: AXI test file.

```c
#include <stdint.h>

int main(int argc, char **argv) {
    uint32_t *addr_axi = (uint32_t*) 0x20000;
    *addr_axi = 8;
    uint32_t result = *addr_axi;

    while(1);

    return 0;
}
```

From the figure below, it is possible to understand that the AXI is working properly. Starting in the vertical yellow line, one can see that the system bus is signalising that the CPU wants to write (ssb_we=1) to the correct address (ssb_addr=0x20000) with the expected data (ssb_wdata=8). Furthermore, in this moment, the system bus is requesting data to the AXI (ssb_req_axi=1) instead of the RAM (ssb_req_sram=0), as it should. Finally, some cycles later, when there is an AXI request for the second time, the core is trying to read (ssb_we=0) from the same address (ssb_addr=0x20000) and the response is correct (ssb_rdata=8).

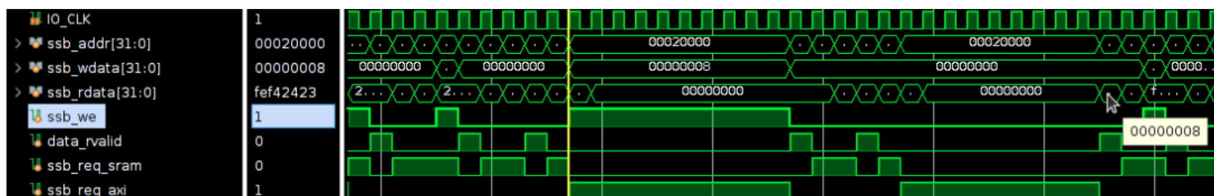


Figure 7.2: AXI test simulation.

7.1.3 Timer

In order to test the timer, one of the FPGA pins was connected to the oscilloscope. This pin was toggled every time an overflow occurred, and the timer was configured to overflow every 10ms, producing a 50Hz square wave. Figure 7.3 depicts the obtained result.

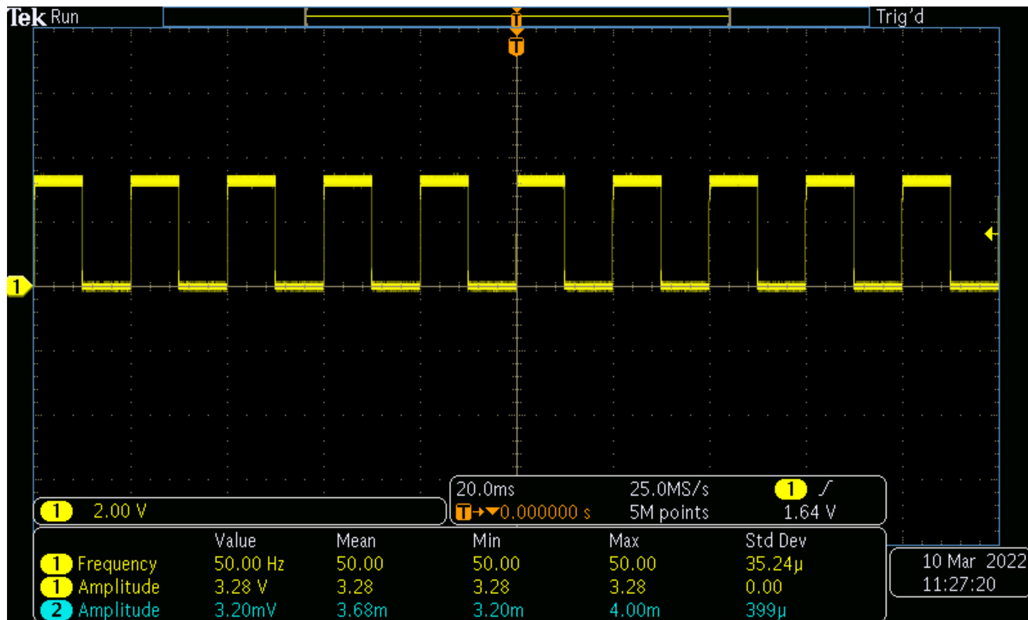


Figure 7.3: Timer test in the oscilloscope.

7.1.4 I2C

After developing the I2C master peripheral, due to the protocol constraints, it had to be connected to a slave for testing. Therefore, an RTC module was attached to the I2C lines, alongside an oscilloscope for sniffing the communication (see Figure 7.4).

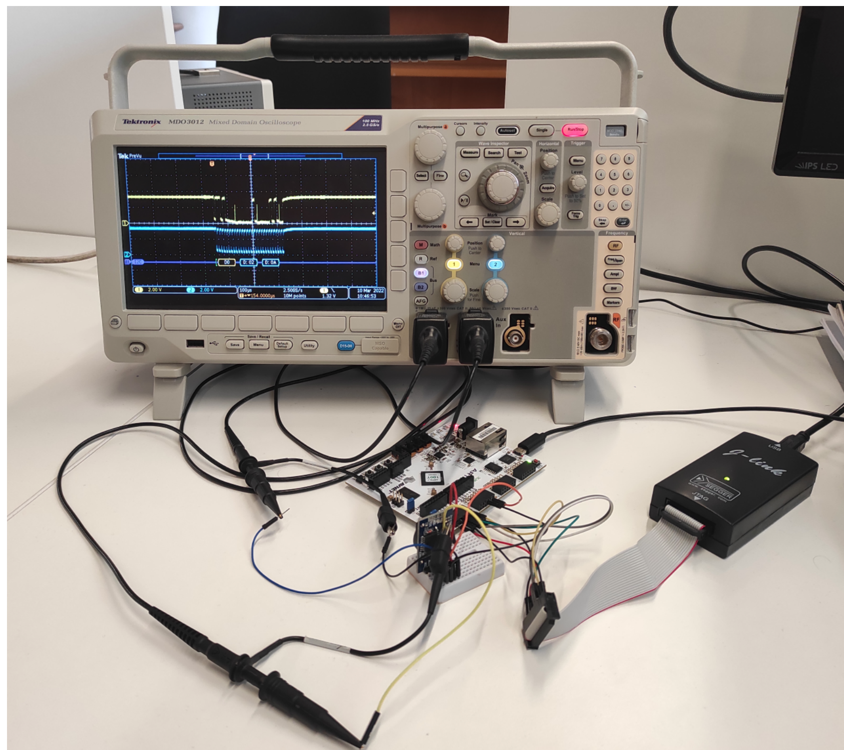


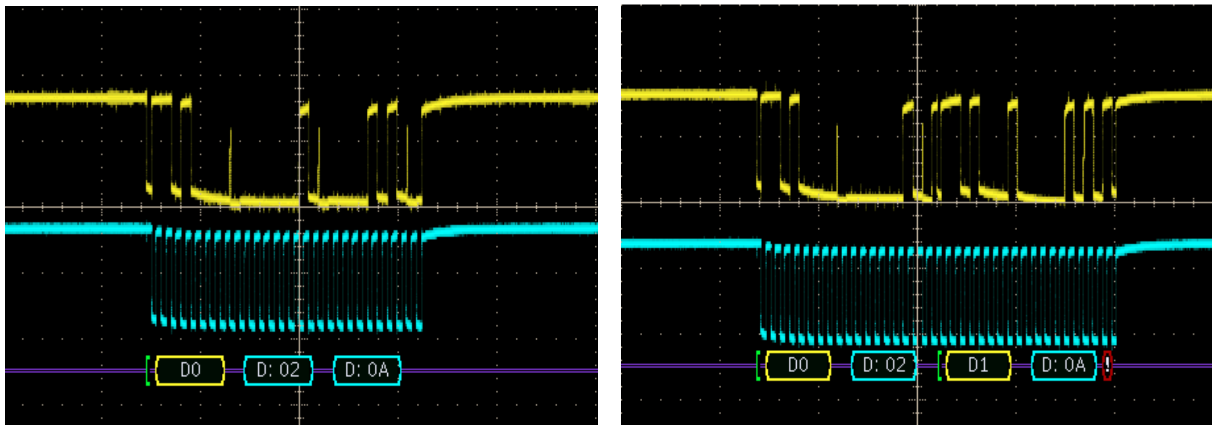
Figure 7.4: Tests setup.

Code 7.3 contains the C program that was used to validate this peripheral. First, it configures the I2C in standard mode and with interrupts disabled. Then, it writes to register number 2 the value 0xA and reads from that same register.

Code 7.3: I2C test file.

```
int main(int argc, char **argv) {
    uint8_t data[2];
    I2C_InitType i2c_values;

    /* Configure I2C */
    i2c_values.interrupt_enable = 0;
    i2c_values.OperationMode = Standard;
    I2C_Initialize(i2c, i2c_values);

    data[0] = 0x02; /* Register addr */
    data[1] = 0x0A; /* Data */
    I2C_Transmit(i2c, 0xD0, data, 2);

    data[0] = 0x02; /* Register addr */
    I2C_Receive(i2c, 0xD1, data);

    return 0;
}
```

By sniffing the communication with the oscilloscope and using its protocol analyser tool to decode the I2C protocol, one can see in Figure 7.5 that the sent and received data goes accordingly to the above test program.



(a) I2C transmission.

(b) I2C reception.

Figure 7.5: I2C protocol sniffed by the oscilloscope.

7.1.5 ThreadX port

To validate that the OS port was successful, a demo application was examined with Azure RTOS TraceX. This Microsoft analysis tool helps to visualise and understand system events through a graphical interface, so its use is recommended after any port.

At first, the source file inside the samples folder was tested (see Figure 6.4), with some minor changes.

However, due to the large quantity of information required by TraceX, the system didn't have enough memory to store that amount of data. Therefore, due to those memory constraints, the best example which could be analysed only had two threads.

The code of these threads was pretty basic and similar. Both threads had the same priority, and they only incremented a counter variable followed by a thread sleep. The difference between them was that an event flag was used to just start thread 1 after thread 0 executed five times. This behaviour can easily be seen in Figure 7.6. Firstly, thread 0 executed for the first time, followed by thread 1, which did an event flag get (EG orange block). Afterwards, thread 0 ran four more times and then set the event flag (ES orange block) so that thread 1 could finally run. Finally, it is also possible to notice that, after every time thread 0 executed, the other thread ran two times because its sleep time was half of the other.

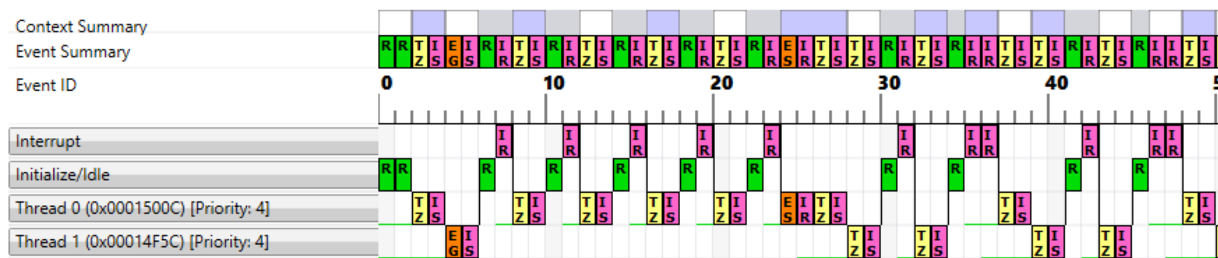


Figure 7.6: TraceX sequential view.

7.2 System test

To validate the final system, with all of its components integrated, a demo application (see Appendix D.2) with two threads was made.

Code 7.4 presents the content of the first thread. Initially, both the RTC module (which was previously configured through I2C) and timer 1 (configured to overflow after 5 minutes) are started at the same moment. Then, the number of seconds elapsed is read from the RTC and inserted into queue 0. Lastly, a sleep of 500ms is made to release the CPU, so that thread 1 can execute.

Code 7.4: System test thread0.

```
void thread_0_entry(ULONG thread_input) {
    UINT status = 0;
    UINT rtc_sec_elapsed = 0;

    RTC_Set_Time(0, 0, 0); /* Starts the RTC */
    TIMER_Start(timer1);

    while(1) {
        rtc_sec_elapsed = RTC_Get_Seconds();

        /* Send data to queue 0 */
        status =  tx_queue_send(&queue_0, &rtc_sec_elapsed,
                                TX_WAIT_FOREVER);
```

```
        /* Check completion status */
        if (status != TX_SUCCESS)
            break;

        /* Sleep 500ms */
        tx_thread_sleep(50);
    }
}
```

Concerning the other thread, as can be seen in Code 7.5, it waits infinitely for data to be inserted into queue 0. When this happens (when the RTC seconds are inserted by thread 0), it will retrieve that data and store it. Afterwards, the thread will read how many seconds have passed since timer 1 started counting. Finally, the RTC seconds and the timer 1 seconds will be compared. If they match, the thread will continue, otherwise, the thread will break from the loop and consequently kill itself.

Code 7.5: System test thread1.

```
void thread_1_entry(ULONG thread_input) {
    UINT status = 0;
    UINT tim_sec_elapsed = 0;;
    UINT rtc_sec_elapsed = 0;

    while(1) {
        /* Retrieve data from the queue */
        status = tx_queue_receive(&queue_0, &rtc_sec_elapsed,
                                  TX_WAIT_FOREVER);

        /* Read timer 1 seconds */
        tim_sec_elapsed = (TIMER_Get_Counter(timer1) / SYS_CLK_FREQ);

        /* Check completion status */
        if (status != TX_SUCCESS)
            break;

        /* Compare both values */
        if(tim_sec_elapsed != rtc_sec_elapsed) {
            break; /* Insert break point here */
        };
    }
}
```

Since the RTC and timer 1 start at the same time, it is expected that the comparison between the seconds of both matches. However, due to how the RTC works, after 1 minute, its internal second variable will overflow, changing to 0, making, tim_sec_elapsed != rtc_sec_elapsed, and making thread 1 exit the loop.

To see this behaviour, a breakpoint was inserted into the break inside the comparison if statement. Then, after running the application for 1 minute, Figure 7.7 shows that the program stopped exactly when it was supposed to (tim_sec_elapsed = 60 and rtc_sec_elapsed = 0).

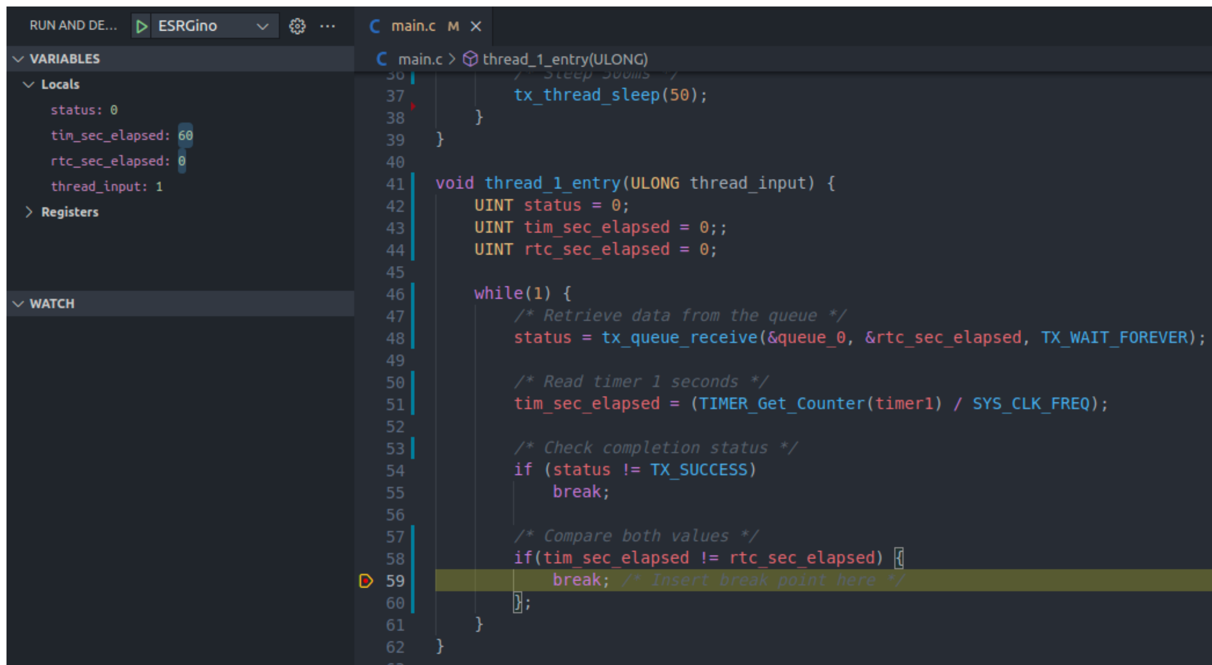


Figure 7.7: System test debugging.

Chapter 8: Conclusion

Embedded systems are growing rapidly in our everyday life, representing a huge part of our key infrastructures. With this current trend, the development cycle of these systems has become more complex. Thus, the choice of an adequate development platform has become quite important since it represents a big part of this development process. For this reason, the purpose of this dissertation was to develop an entirely free soft-core microcontroller, alongside its software kit, so that the University of Minho research group could use it for future projects.

To accomplish this goal, a study and consequently decision between three different topics were made. Firstly, after comparing different ISAs, the RISC-V architecture was picked for its advantages alongside its open nature. Then, due to its great documentation, free use, and features, ThreadX was the selected real-time operating system. Ultimately, to reduce the costs and the development cycle time, the Xilinx Arty A7-100 was the chosen development platform.

With the ISA decided, the next step was a comparison based on some crucial features between different RISC-V implementations for FPGA. Based on this evaluation, it was possible to conclude that Ibex was the most viable option due to its excellent documentation, debug support, HDL, and suitability for embedded systems applications.

After all these design decisions, the development of the microcontroller was made. This implementation started with the integration of the Ibex core with the debug unit and the RAM. Then, the I2C and the timer peripherals were created and connected to the CPU through an AXI interface.

In parallel with the hardware development, the SDK was also created. It includes all the necessary support files (e.g. linker, OpenOCD files, makefile) to program and debug the microcontroller and also its BSP.

Lastly, after finishing the soft-core microcontroller and the SDK, the port of the Azure ThreadX RTOS was made to this system.

8.1 Future work

Despite all the work developed during this dissertation, there are still some limitations when using the EKKO microcontroller. For this reason, some topics for further improvement are presented below,

alongside possible solutions.

- **User customisation**

  For a better user experience, some tools can be used for easier customisation of the microcontroller's hardware and software. Examples of these tools are FuseSoC, which includes a set of build tools for HDL code and CMake, which would allow cross-platform development.

- **Peripherals**

  Besides the I2C and the timers, there are still tons of peripherals that can be integrated into the soft-core microcontroller. These should have an AXI-Lite Slave interface, and its respective device drivers have to be inserted into the EKKO SDK.

- **ASIC**

  After developing more peripherals for the microcontroller and turning it more complete, it would be beneficial to do an ASIC. This would increase performance and reduce power consumption and production cost in mass production scenarios.

- **Hardware accelerators**

  Another interesting feature would be the development of hardware accelerators. These could be used in the ThreadX scheduling algorithm and cryptography, which is common in IoT applications.

- **Power consumption**

  Power consumption isn't examined in this dissertation. Therefore, the integration of consumption reduction techniques such as clock gating and multiple frequency clock domains could be appealing for future work.

References

[1] J. M. P. C. Miranda, "Sensing technologies in pervasive healthcare: Evaluation and design for senior citizens and continuous care," 2019.

[2] K. Ashton, "June 22. that "internet of things" thing," *RFID Journal*, 1999.

[3] K. Ashton *et al.*, "That 'internet of things' thing," *RFID journal*, vol. 22, no. 7, pp. 97–114, 2009.

[4] L. Atzori, A. Iera, and G. Morabito, "The internet of things: A survey," *Computer networks*, vol. 54, no. 15, pp. 2787–2805, 2010.

[5] F. Xia, L. T. Yang, L. Wang, and A. Vinel, "Internet of things," *International journal of communication systems*, vol. 25, no. 9, p. 1101, 2012.

[6] Insight., "Is Tech The New Currency? Why You Need Modern IT," 2017. Accessed on 2020-12-28.

[7] K. K. Patel, S. M. Patel, *et al.*, "Internet of things-iot: definition, characteristics, architecture, enabling technologies, application & future challenges," *International journal of engineering science and computing*, vol. 6, no. 5, 2016.

[8] L. Da Xu, W. He, and S. Li, "Internet of things in industries: A survey," *IEEE Transactions on industrial informatics*, vol. 10, no. 4, pp. 2233–2243, 2014.

[9] X. Jia, Q. Feng, T. Fan, and Q. Lei, "Rfid technology and its applications in internet of things (iot)," in *2012 2nd international conference on consumer electronics, communications and networks (CECNet)*, pp. 1282–1285, IEEE, 2012.

[10] M. C. Domingo, "An overview of the internet of things for people with disabilities," *Journal of Network and Computer Applications*, vol. 35, no. 2, pp. 584–596, 2012.

[11] I. T. Union, *Internet of Things: IoT Day Special*, vol. 7. LexInnova Technologies, LLC, 2005.

[12] J. G. Tong, I. D. L. Anderson, and M. A. S. Khalid, "Soft-core processors for embedded systems," in *2006 International Conference on Microelectronics*, pp. 170–173, 2006.

[13] A. Cohen and E. Rohou, "Processor virtualization and split compilation for heterogeneous multicore embedded systems," in *Design Automation Conference*, pp. 102–107, IEEE, 2010.

[14] C. Tanougast, A. Dandache, M. S. Azzaz, and S. Said, *Hardware Design of Embedded Systems for Security Applications*. 03 2012.

[15] F. Vahid and T. Givargis, "Embedded system design: A unified hardware/software approach," *Department of Computer Science and Engineering University of California*, 1999.

[16] D. Liu, "Asip (application specific instruction-set processors) design," in *2009 IEEE 8th International Conference on ASIC*, pp. 16–16, 2009.

[17] P. Robert and B. A. Fette, "The software defined radio as a platform for cognitive radio," in *Cognitive radio technology*, pp. 73–118, Elsevier, 2006.

[18] He Zhiqiang, Sun Jianxing, Duan Ran, and Yue Chen, "Analysis for singal processing development with general purpose processor," in *7th International Conference on Communications and Networking in China*, pp. 792–796, 2012.

[19] T. Shanley, *x86 Instruction Set Architecture*. MindShare press, 2010.

[20] F. Masood, "Risc and cisc," *arXiv preprint arXiv:1101.5364*, 2011.

[21] J. Hennessy, N. Jouppi, S. Przybylski, C. Rowen, T. Gross, F. Baskett, and J. Gill, "Mips: A microprocessor architecture," *SIGMICRO Newsl.*, vol. 13, p. 17–22, Oct. 1982.

[22] A. S. Waterman, *Design of the RISC-V instruction set architecture*. PhD thesis, UC Berkeley, 2016.

[23] S. El Kady, M. Khater, and M. Alhafnawi, "Mips , arm and sparc-an architecture comparison," in *Proceedings of the World Congress on Engineering*, vol. 1, 2014.

[24] B. J. Catanzaro, *The SPARC technical papers*. Springer Science & Business Media, 2012.

[25] S. I. Inc and D. L. Weaver, *The SPARC architecture manual*. Prentice-Hall, 1994.

[26] C. Domas, "Breaking the x86 isa," *Black Hat*, 2017.

[27] The Silicon Review, "Architecting a smart world and powering Artificial Intelligence: ARM," 2019. Accessed on 2020-11-28.

[28] L. Ryzhyk, "The arm architecture," *Chicago University, Illinois, EUA*, 2006.

[29] R. Höller, D. Haselberger, D. Ballek, P. Rössler, M. Krapfenbauer, and M. Linauer, "Open-source risc-v processor ip cores for fpgas—overview and evaluation," in *2019 8th Mediterranean Conference on Embedded Computing (MECO)*, pp. 1–6, IEEE, 2019.

[30] A. Waterman and K. Asanovic, "The risc-v instruction set manual, volume i: Unprivileged isa document," *RISC-V Foundation, Tech. Rep*, 2019.

[31] A. Silberschatz, P. B. Galvin, and G. Gagne, *Operating system principles*. John Wiley & Sons, 2006.

[32] P. B. Hansen, *Operating system principles*. Prentice-Hall, Inc., 1973.

[33] J. W. Valvano, *Embedded Systems: Real-time Operating Systems for ARM® Cortex"! M Microcomputers*. Self-published Jonathan W. Valvano, 2017.

[34] P. Hambarde, R. Varma, and S. Jha, "The survey of real time operating system: Rtos," in *2014 International Conference on Electronic Systems, Signal Processing and Computing Technologies*, pp. 34–39, IEEE, 2014.

[35] A. Barbalace, A. Luchetta, G. Manduchi, M. Moro, A. Soppelsa, and C. Taliercio, "Performance comparison of vxworks, linux, rtai, and xenomai in a hard real-time application," *IEEE Transactions on Nuclear Science*, vol. 55, no. 1, pp. 435–439, 2008.

[36] R. V. Aroca, G. Caurin, and S. Carlos-SP-Brasil, "A real time operating systems (rtos) comparison," in *WSO-Workshop de Sistemas Operacionais*, vol. 12, 2009.

[37] R. T. Mahani and N. Mahani, "Vxworks vs. lynxos real-time operating systems for embedded systems," 2015.

[38] E. Fernández and T. Sorgente, "A pattern language for secure operating system architectures," 01 2021.

[39] AWS, "FreeRTOS - Market leading RTOS (Real Time Operating System)." `https://www.freertos.org/`. Accessed on 2020-12-04.

[40] R. Inam, J. Mäki-Turja, M. Sjödin, S. M. Ashjaei, and S. Afshar, "Support for hierarchical scheduling in freertos," in *ETFA2011*, pp. 1–10, IEEE, 2011.

[41] C. Sabri, L. Kriaa, and S. L. Azzouz, "Comparison of iot constrained devices operating systems: A survey," in *2017 IEEE/ACS 14th International Conference on Computer Systems and Applications (AICCSA)*, pp. 369–375, IEEE, 2017.

[42] H. Kensing and A. Vrålstad, "A comparison between vxworks and lynxos regarding memory management," 2006.

[43] A. M. Kamboh, A. H. Krishnamurthy, and J. K. K. Vallabhaneni, "Demonstration of multitasking using threadx rtos on microblaze and powerpc," *Tech. Rep.*, vol. 1, pp. 1–12, 2006.

[44] M. Borges, S. Paiva, A. Santos, B. Gaspar, and J. Cabral, "Azure rtos threadx design for low-end nb-iot device," in *2020 2nd International Conference on Societal Automation (SA)*, pp. 1–8, 2021.

[45] S. Wiki, "ThreadX Overview." `https://wiki.st.com/stm32mcu/wiki/THREADX_overview#References`. Accessed on 2020-24-12.

[46] "RT-Thread." `https://github.com/RT-Thread/rt-thread`. Accessed on 2020-12-24.

[47] A. Milinković, S. Milinković, and L. Lazic, "Choosing the right rtos for iot platform," 03 2015.

[48] "RIOT: The friendly Operating System for the Internet of Things." `https://www.riot-os.org/`. Accessed on 2020-12-24.

[49] E. Baccelli, O. Hahm, M. Günes, M. Wählisch, and T. C. Schmidt, "Riot os: Towards an os for the internet of things," in *2013 IEEE conference on computer communications workshops (INFOCOM WKSHPS)*, pp. 79–80, IEEE, 2013.

[50] G. W. Morris, D. B. Thomas, and W. Luk, "Fpga accelerated low-latency market data feed processing," in *2009 17th IEEE Symposium on High Performance Interconnects*, pp. 83–89, 2009.

[51] Intel, "Intel® Agilex™ F-Series FPGAs & SoCs." `https://www.intel.com/content/www/us/en/products/programmable/fpga/agilex/f-series.html`. Accessed on 2021-10-01.

[52] Intel, "Intel® Agilex™ FPGAs Deliver a Game-Changing Combination of Flexibility and Agility for the Data-Centric World." Accessed on 2021-10-01.

[53] Intel, "Cyclone® V FPGAs." `https://www.intel.com/content/www/us/en/products/programmable/fpga/cyclone-v.html`. Accessed on 2021-10-01.

[54] Intel, "Cyclone V Device Overview." `https://www.intel.com/content/www/us/en/programmable/documentation/sam1403480548153.html`. Accessed on 2021-10-01.

[55] Lattice, "LatticeXP2: Get flexible, get flexiFLASH." `https://www.latticesemi.com/en/Products/FPGAandCPLD/LatticeXP2`. Accessed on 2021-13-01.

[56] Lattice, "Certus-NX: Low-Power General Purpose FPGA." https://www.latticesemi.com/en/Products/FPGAandCPLD/Certus-NX. Accessed on 2021-13-01.

[57] Xilinx, "Artix-7." https://www.xilinx.com/products/silicon-devices/fpga/artix-7.html. Accessed on 2021-14-01.

[58] Xilinx, "SoCs with Hardware and Software Programmability." https://www.xilinx.com/products/silicon-devices/soc/zynq-7000.html. Accessed on 2021-14-01.

[59] Xilinx, "Virtex Ultrascale+ FPGAs." https://www.xilinx.com/products/silicon-devices/fpga/virtex-ultrascale-plus.html. Accessed on 2021-14-01.

[60] Microchip, "PolarFire® FPGAs." https://www.microchip.com/en-us/products/fpgas-and-plds/fpgas/polarfire-fpgas. Accessed on 2021-15-01.

[61] Xilinx, "Arty A7-100: Artix-7 FPGA Development Board for Makers and Hobbyists." https://www.xilinx.com/products/boards-and-kits/1-w51quh.html. Accessed on 2021-15-01.

[62] Xilinx, "7 Series FPGAs Configurable Logic Block User Guide." https://www.xilinx.com/support/documentation/user_guides/ug474_7Series_CLB.pdf. Accessed on 2021-18-01.

[63] Xilinx, "Revolutionary Architecture for the Next Generation Platform FPGAs." https://m.eet.com/media/1081699/XIL.PDF. Accessed on 2021-18-01.

[64] DZone, "The RISC-V Architecture." https://dzone.com/articles/introduction-to-the-risc-v-architecture. Accessed on 2021-11-04.

[65] RISC-V, "RISC-V Exchange: Cores & SoCs." https://riscv.org/exchange/cores-socs/. Accessed on 2021-11-04.

[66] "VexRiscv." https://github.com/SpinalHDL/VexRiscv. Accessed on 2021-11-04.

[67] "PicoRV32." https://github.com/cliffordwolf/picorv32. Accessed on 2021-11-04.

[68] "PULPino." https://github.com/pulp-platform/pulpino. Accessed on 2021-13-04.

[69] "CV32E40P." https://github.com/openhwgroup/cv32e40p. Accessed on 2021-13-04.

[70] "Ibex." https://github.com/lowRISC/ibex. Accessed on 2021-13-04.

[71] "Potato." https://github.com/skordal/potato. Accessed on 2021-13-04.

[72] T. Verbeure, "VexRiscv, OpenOCD, and Traps." https://tomverbeure.github.io/2021/07/18/VexRiscv-OpenOCD-and-Traps.html. Accessed on 2021-05-11.

[73] "RISC-V DBG." https://github.com/pulp-platform/riscv-dbg.

[74] T. Newsome and M. Wachs, "Risc-v debug specification 0.13.2," 2019.

[75] P. Semiconductor, "The i2c-bus specification: Version 2.1, january 2000."

[76] D. Kaith, J. B. Patel, and N. Gupta, "An introduction to functional verification of i2c protocol using uvm," *International Journal of Computer Applications*, vol. 121, no. 13, 2015.

[77] F. Key, "On-Chip Buses." https://www.fpgakey.com/tutorial/section526. Accessed on 2021-29-09.

[78] "Diogo Dias GitHub." https://github.com/mrdiogodias?tab=repositories.

[79] "OpenOCD." https://openocd.org/doc/html/About.html. Accessed on 2021-28-09.

[80] Microsoft, "Azure RTOS ThreadX documentation." https://docs.microsoft.com/en-us/azure/rtos/threadx/. Accessed on 2021-21-06.

Appendix A: Support files

A.1 Makefile

Code A.1: makefile.

```
COMMON_DIR      := $(shell dirname $(realpath $(lastword $(MAKEFILE_LIST))))
THREADX_PORT_DIR    := ${PROGRAM_DIR}/threadx/ports/risc-v32/gnu/src
THREADX_COMMON_DIR := ${PROGRAM_DIR}/threadx/common/src

TX_ASM_SRCS       := $(wildcard ${THREADX_PORT_DIR}/*.S)
TX_THREAD_SRCS      := $(wildcard ${THREADX_COMMON_DIR}/tx_thread_*.c)
TX_TIMER_SRCS       := $(wildcard ${THREADX_COMMON_DIR}/tx_time*.c)
TX_INIT_SRCS        := $(wildcard ${THREADX_COMMON_DIR}/tx_initialize_*.c)
TX_EF_SRCS          := $(wildcard ${THREADX_COMMON_DIR}/tx_event_flags_*.c)
TX_QUEUE_SRCS       := $(wildcard ${THREADX_COMMON_DIR}/tx_queue_*.c)
TX_SEMAPHORE_SRCS   := $(wildcard ${THREADX_COMMON_DIR}/tx_semaphore_*.c)
TX_MUTEX_SRCS       := $(wildcard ${THREADX_COMMON_DIR}/tx_mutex_*.c)

COMMON_SRCS      = $(wildcard $(COMMON_DIR)/src/*.c)
COMMONS_INCS       = $(COMMON_DIR)/inc
TX_SRCS         := ${TX_ASM_SRCS} ${TX_THREAD_SRCS} ${TX_TIMER_SRCS} ${
    TX_INIT_SRCS} \
                ${TX_EF_SRCS}  ${TX_QUEUE_SRCS}  ${TX_MUTEX_SRCS} ${
                    TX_SEMAPHORE_SRCS}
TX_INCS           := -I${PROGRAM_DIR}/threadx/common/inc -I${PROGRAM_DIR}/
    threadx/ports/risc-v32/gnu/inc

ARCH = rv32im

ifdef PROGRAM
PROGRAM_C := $(PROGRAM).c
endif

ifdef THREADX_EN
SRCS = $(COMMON_SRCS) $(PROGRAM_C) $(EXTRA_SRCS) $(TX_SRCS)
INCS = -I$(COMMONS_INCS) $(EXTRA_INCS) $(TX_INCS)
else
SRCS = $(COMMON_SRCS) $(PROGRAM_C) $(EXTRA_SRCS)
INCS = -I$(COMMONS_INCS) $(EXTRA_INCS)
endif

C_SRCS   = $(filter %.c, $(SRCS))
ASM_SRCS = $(filter %.S, $(SRCS))

CC = riscv32-unknown-elf-gcc

OBJCOPY ?= $(subst gcc,objcopy,$(wordlist 1,1,$(CC)))
OBJDUMP ?= $(subst gcc,objdump,$(wordlist 1,1,$(CC)))

LINKER_SCRIPT ?= $(COMMON_DIR)/linker.ld
CRT ?= $(COMMON_DIR)/src/crt0.S
CFLAGS ?= -march=$(ARCH) -mabi=ilp32 -mcmodel=medany -Wall -g -O0\
  -fvisibility=hidden -nostdlib -nostartfiles $(PROGRAM_CFLAGS)

OBJS := ${C_SRCS:.c=.o} ${ASM_SRCS:.S=.o} ${CRT:.S=.o}
DEPS = $(OBJS:%.o=%.d)

ifdef PROGRAM
OUTFILES := $(PROGRAM).elf
else
```

```
OUTFILES := $(OBJS)
endif

all: $(OUTFILES)

ifdef PROGRAM
$(PROGRAM).elf: $(OBJS) $(LINKER_SCRIPT)
	$(CC) $(CFLAGS) -T $(LINKER_SCRIPT) $(OBJS) -o $@ $(LIBS)

.PHONY: disassemble
disassemble: $(PROGRAM).dis
endif

%.dis: %.elf
	$(OBJDUMP) -SD $^ > $@

%.o: %.c
	$(CC) $(CFLAGS) -MMD -c $(INCS) -o $@ $<

%.o: %.S
	$(CC) $(CFLAGS) -MMD -c $(INCS) -o $@ $<

clean:
	$(RM) -f $(OBJS) $(DEPS) $(PROGRAM).elf $(PROGRAM).dis
```

A.2 Linker

Code A.2: linker.ld.

```
OUTPUT_ARCH(riscv)

/* required to correctly link newlib */
GROUP( -lc -lgloss -lgcc -lsupc++ )

SEARCH_DIR(.)
__DYNAMIC  =  0;

MEMORY
{
    rom          : ORIGIN = 0x000000000, LENGTH = 0x1E000 /* 120 kB */
    stack        : ORIGIN = 0x00001E000, LENGTH = 0x02000 /*   8 kB */
}

/* Stack information variables */
_min_stack      = 0x1000;   /* 4K - minimum stack space to reserve */
_stack_len      = LENGTH(stack);
_stack_start    = ORIGIN(stack) + LENGTH(stack);

SECTIONS
{
    .vectors :
    {
        . = ALIGN(4);
        KEEP(*(.vectors))
    } > rom

    .text : {
        . = ALIGN(4);
        _stext = .;
        *(.text)
        *(.text.*)
        _etext  =  .;
        __CTOR_LIST__ = .;
        LONG((__CTOR_END__ - __CTOR_LIST__) / 4 - 2)
        *(.ctors)
        LONG(0)
```

```
        __CTOR_END__ = .;
        __DTOR_LIST__ = .;
        LONG((__DTOR_END__ - __DTOR_LIST__) / 4 - 2)
        *(.dtors)
        LONG(0)
        __DTOR_END__ = .;
        *(.lit)
        *(.shdata)
        . = ALIGN(4);
        _endtext = .;
    }  > rom

    .rodata : {
        . = ALIGN(4);
        *(.rodata);
        *(.rodata.*)
    } > rom

    .shbss :
    {
        . = ALIGN(4);
        *(.shbss)
    } > rom

    .bss :
    {
        . = ALIGN(4);
        _bss_start = .;
        *(.bss)
        *(.bss.*)
        *(.sbss)
        *(.sbss.*)
        *(COMMON)
        _bss_end = .;
    } > rom

    .sdata : {
        . = ALIGN(4);
        *(.rodata);
        *(.rodata.*)
    } > rom

    .data : {
        . = ALIGN(4);
        sdata  =  .;
        _sdata  =  .;
        *(.data);
        *(.data.*)
        edata  =  .;
        _edata  =  .;
    } > rom

    /* ensure there is enough room for stack */
    .stack (NOLOAD): {
        . = ALIGN(4);
        . = . + _min_stack ;
        . = ALIGN(4);
        stack = . ;
        _stack = . ;
    } > stack

    .stab  0 (NOLOAD) :
    {
        [ .stab ]
    }

    .stabstr  0 (NOLOAD) :
    {
        [ .stabstr ]
    }
}
```

A.3 VSCode debug script

Code A.3: launch.json.

```
{
    "version": "0.2.0",
    "configurations": [
        {
            "name": "EKKO",
            "type": "cppdbg",
            "targetArchitecture":"ARM",
            "request": "launch",
            "program": "${workspaceFolder}/main.elf",
            "cwd": "${workspaceFolder}",
            "args": [],
            "stopAtEntry": true,
            "environment": [],
            "externalConsole": false,
            "MIMode": "gdb",
            "miDebuggerServerAddress": "localhost:3333",
            "customLaunchSetupCommands": [
                { "text": "file ${workspaceFolder}/main.elf", "description
                    ": "Loading ELF File", "ignoreFailures": false },
                { "text": "target remote:3333", "description": "Connecting
                    to the target", "ignoreFailures": false },
                { "text": "load", "description": "Load ELF file to the
                    target", "ignoreFailures": false },
                { "text": "b main", "description": "Insert breakpoint in
                    main", "ignoreFailures": false },
                { "text": "jump reset_handler", "description": "Jump to the
                     beggining of the program", "ignoreFailures": true },
            ],
            "launchCompleteCommand": "None",
            "logging": { "engineLogging": true },
            "linux": {
                "MIMode": "gdb",
                "miDebuggerPath": "riscv32-unknown-elf-gdb"
            }
        }
    ]
}
```

Appendix B: Board Support Package

Code B.1: bsp.h.

```
#ifndef __BSP__ /* prevent circular inclusions */
#define __BSP__ /* by using protection macros */


/*****  Include Files *****/

#include <stdint.h>


/***** Macros *****/

#define TIMER0_BASE  0x20000
#define TIMER1_BASE  0x21000
#define I2C_BASE     0x22000

#define SYS_CLK_FREQ 40000000 /* 40 MHz */


/*****  Device Peripheral Access Layer *****/

typedef struct {
    union {
        struct {
            uint32_t reserved    : 27;
            uint8_t  overflow    : 1;
            uint8_t  auto_reload : 1;
            uint8_t  interrupt   : 1;
            uint8_t  enable      : 1;
            uint8_t  start       : 1;
        } bit;
        uint32_t config_register;
    } CONF_REG;                /* (TIMER_BASE_ADDR + 0x00) */
    uint32_t DATA_REG_HIGH; /* (TIMER_BASE_ADDR + 0x04) */
    uint32_t DATA_REG_LOW;  /* (TIMER_BASE_ADDR + 0x08) */
    uint32_t CMP_REG_HIGH;  /* (TIMER_BASE_ADDR + 0x0C) */
    uint32_t CMP_REG_LOW;   /* (TIMER_BASE_ADDR + 0x10) */
} TIMER_Type;


typedef struct {
    union {
        struct {
            uint8_t  addr;
            uint8_t  start              : 1;
            uint8_t  error              : 1;
            uint8_t  valid_transmission : 1;
            uint8_t  valid_reception    : 1;
            uint8_t  enable             : 1;
            uint8_t  interrupt          : 1;
            uint8_t  reserved           : 2;
            uint16_t prescaler;
        } bit;
        uint32_t config_register0;
    } CONF0_REG;            /* (I2C_BASE_ADDR + 0x00) */
    uint32_t CONF1_REG;  /* (I2C_BASE_ADDR + 0x04) */
    uint32_t CONF2_REG;  /* (I2C_BASE_ADDR + 0x08) */
    union {
        struct {
            uint8_t  reserved;
            uint8_t  data_received;
```

```
            uint8_t  data_size;
            uint8_t  data_to_send;
        } bit;
        uint32_t config_register3;
    } CONF3_REG;          /* (I2C_BASE_ADDR + 0x0C) */
} I2C_Type;


/***** Function prototypes *****/

/* Software delays */
void sleep_us(unsigned long us);
void sleep_ms(unsigned long ms);

/* Memset implementation */
void *my_memset(void *s, int c, unsigned int len);

#endif
```

Code B.2: bsp.c.

```
#include <bsp.h>

static void delay_loop_ibex(unsigned long loops) {
    int out; /* only to notify compiler of modifications to |loops| */
    asm volatile(
        "1: nop             \n" // 1 cycle
        "   nop             \n" // 1 cycle
        "   nop             \n" // 1 cycle
        "   nop             \n" // 1 cycle
        "   addi %1, %1, -1 \n" // 1 cycle
        "   bnez %1, 1b     \n" // 3 cycles
        : "=&r"(out)
        : "0"(loops));
}

static void usleep_ibex(unsigned long usec) {
    unsigned long freq = SYS_CLK_FREQ / 1000000;
    unsigned long usec_cycles = freq * usec / 8;

    delay_loop_ibex(usec_cycles);
}

void sleep_us(unsigned long us) {
    usleep_ibex(us);
}

void sleep_ms(unsigned long ms) {
    usleep_ibex(ms * 1000);
}

void *my_memset(void *s, int c, unsigned int len) {
    unsigned char *p = s;

    while (len--) {
        *p++ = (unsigned char)c;
    }

    return s;
}
```

B.1 C Runtime Initialization

Code B.3: crt0.S.

```
.section .text

systick_timer_handler:
    /* Jump to threadx timer handler */
    j _tx_timer_interrupt_handler

default_exc_handler:
    jal x0, default_exc_handler

reset_handler:
    /* Set all registers to zero */
    mv  x1, x0
    mv  x2, x1
    mv  x3, x1
    mv  x4, x1
    mv  x5, x1
    mv  x6, x1
    mv  x7, x1
    mv  x8, x1
    mv  x9, x1
    mv x10, x1
    mv x11, x1
    mv x12, x1
    mv x13, x1
    mv x14, x1
    mv x15, x1
    mv x16, x1
    mv x17, x1
    mv x18, x1
    mv x19, x1
    mv x20, x1
    mv x21, x1
    mv x22, x1
    mv x23, x1
    mv x24, x1
    mv x25, x1
    mv x26, x1
    mv x27, x1
    mv x28, x1
    mv x29, x1
    mv x30, x1
    mv x31, x1

    /* Clear CSRs */
    csrwi mepc,    0
    csrwi mstatus, 0
    csrwi mie,     0

    /* Stack initilization */
    la   x2, _stack_start

_start:
.global _start
    /* Clear BSS */
    la x26, _bss_start
    la x27, _bss_end

    bge x26, x27, zero_loop_end

zero_loop:
    sw x0, 0(x26)
    addi x26, x26, 4
    ble x26, x27, zero_loop
    zero_loop_end:
```

```
main_entry:
    /* Jump to main program entry point (argc = argv = 0) */
    addi x10, x0, 0
    addi x11, x0, 0
    jal x1, main

sleep_loop:
    j sleep_loop

/* =============== Exceptions =============== */

.section .vectors, "ax"
.option norvc;

.org 0x00
.rept 7
jal x0, default_exc_handler
.endr

/* Systick handler */
jal x0, systick_timer_handler

.rept 23
jal x0, default_exc_handler
.endr

/* Reset vector */
.org 0x80
jal x0, reset_handler

/* Illegal instruction exception */
.org 0x84
jal x0, default_exc_handler

/* Ecall handler */
.org 0x88
jal x0, default_exc_handler
```

B.2 Device drivers

B.2.1 I2C

Code B.4: i2c.h.

```
#ifndef __I2C__ /* prevent circular inclusions */
#define __I2C__ /* by using protection macros */

/***** Include Files *****/

#include "bsp.h"

/***** Macros (Inline Functions) Definitions *****/

#define I2C_VALUE_OK           0U
#define I2C_VALUE_INVALID_SIZE 1U
#define I2C_VALUE_INVALID_ADDR 2U
#define I2C_VALUE_ERROR        3U

typedef uint32_t I2C_ValueType;


/***** Type Definitions *****/

typedef enum {
```

```
    Standard     = 100,  /* 100khz */
    Fast         = 400,  /* 400khz */
    High_Speed   = 3400, /* 3.4Mhz */
} I2C_OperationModeType;

typedef struct{
    I2C_OperationModeType OperationMode;
    uint8_t  interrupt_enable;
} I2C_InitType;


/***** Type Definitions *****/

I2C_ValueType I2C_Initialize(I2C_Type *i2c, I2C_InitType init_values);
I2C_ValueType I2C_Transmit(I2C_Type *i2c, uint8_t addr, uint8_t *data,
   uint8_t data_size);
I2C_ValueType I2C_Receive(I2C_Type *i2c, uint8_t addr, uint8_t *data);


/***** External Variables *****/

extern I2C_Type *i2c;

#endif
```

Code B.5: i2c.c.

```
#include "i2c.h"

I2C_Type *i2c =  ((I2C_Type *) I2C_BASE);

static inline void I2C_Clear_Regs(I2C_Type *i2c) {
  i2c->CONF0_REG.config_register0 = 0;
  i2c->CONF1_REG                  = 0;
  i2c->CONF2_REG                  = 0;
  i2c->CONF3_REG.config_register3 = 0;
}

I2C_ValueType I2C_Initialize(I2C_Type *i2c, I2C_InitType init_values) {
  I2C_Clear_Regs(i2c);

  i2c->CONF0_REG.bit.prescaler = SYS_CLK_FREQ / (2 * init_values.
   OperationMode * 1000) - 1;
  i2c->CONF0_REG.bit.enable    = 1;
  i2c->CONF0_REG.bit.interrupt = init_values.interrupt_enable;

  return I2C_VALUE_OK;
}

I2C_ValueType I2C_Transmit(I2C_Type *i2c, uint8_t addr, uint8_t *data,
   uint8_t data_size) {
  /* Clear device addr */
  i2c->CONF0_REG.bit.addr = 0;

  /* Clear data and data size */
  i2c->CONF1_REG = 0;
  i2c->CONF2_REG = 0;
  i2c->CONF3_REG.bit.data_size = 0;

  /* In a transmission RW needs to be 0 */
  if(addr & 0x1) {
    return I2C_VALUE_INVALID_ADDR;
  }

  /* 9 bytes is the maximum that can be transfered for each transmission */
  if(data_size > 9) {
    return I2C_VALUE_INVALID_SIZE;
  }

  /* Fill the register with the data to send */
```

```
  i2c->CONF1_REG = (data[0] << 24) | (data[1] << 16) | (data[2] << 8) |
   data[3];
  i2c->CONF2_REG = (data[4] << 24) | (data[5] << 16) | (data[6] << 8) |
   data[7];
  i2c->CONF3_REG.bit.data_to_send = data[8];
  i2c->CONF3_REG.bit.data_size    = data_size;

  /* Start communication */
  i2c->CONF0_REG.bit.addr  = addr;
    i2c->CONF0_REG.bit.start = 1;

    /* Wait for the valid transmission bit */
  while(i2c->CONF0_REG.bit.valid_transmission == 0){}

    return I2C_VALUE_OK;
}

I2C_ValueType I2C_Receive(I2C_Type *i2c, uint8_t addr, uint8_t *data) {
  /* Clear device addr */
  i2c->CONF0_REG.bit.addr = 0;

  /* Clear reg1 because the 1st byte is the register addr */
  i2c->CONF1_REG = 0;

  /* In a reception RW needs to be 1 */
  if(!(addr & 0x1)) {
    return I2C_VALUE_INVALID_ADDR;
  }

  /* Insert the register addr in the 1st MSByte of reg1 */
  i2c->CONF1_REG  = (data[0] << 24);

  /* Start communication */
  i2c->CONF0_REG.bit.addr  = addr;
    i2c->CONF0_REG.bit.start = 1;

  /* Wait for the valid reception bit */
    while(i2c->CONF0_REG.bit.valid_reception == 0){}

  data[1] = i2c->CONF3_REG.bit.data_received;

  return I2C_VALUE_OK;
}
```

B.2.2 Timer

Code B.6: timer.h.

```
#ifndef __TIMER__ /* prevent circular inclusions */
#define __TIMER__ /* by using protection macros */


/*****  Include Files *****/

#include "bsp.h"

/***** Macros (Inline Functions) Definitions *****/

#define TIMER_VALUE_OK       0U
#define TIMER_VALUE_ERROR    1U

typedef uint32_t TIMER_ValueType;


/***** Type Definitions *****/

typedef struct{
```

```
    uint64_t compare_value;
    uint8_t  interrupt_enable;
    uint8_t  auto_reload;
} TIMER_InitType;


/***** Type Definitions *****/

TIMER_ValueType TIMER_Initialize(TIMER_Type *timer, TIMER_InitType
    init_values);
TIMER_ValueType TIMER_Start(TIMER_Type *timer);
uint64_t        TIMER_Get_Counter(TIMER_Type *timer);
TIMER_ValueType TIMER_Stop(TIMER_Type *timer);
TIMER_ValueType TIMER_Wait(TIMER_Type *timer);


/***** External Variables *****/

extern TIMER_Type *timer0;
extern TIMER_Type *timer1;

#endif
```

Code B.7: timer.c.

```
#include "timer.h"

TIMER_Type *timer0 =  ((TIMER_Type *) TIMER0_BASE);
TIMER_Type *timer1 =  ((TIMER_Type *) TIMER1_BASE);

static inline void TIMER_Clear_Regs(TIMER_Type *timer) {
  timer->CMP_REG_HIGH               = 0;
  timer->CMP_REG_LOW                = 0;
  timer->CONF_REG.config_register = 0;
  timer->DATA_REG_HIGH        = 0;
  timer->DATA_REG_LOW         = 0;
}

TIMER_ValueType TIMER_Initialize(TIMER_Type *timer, TIMER_InitType
    init_values) {
  TIMER_Clear_Regs(timer);

  timer->CMP_REG_HIGH        = (uint32_t) (init_values.compare_value >> 32);
  timer->CMP_REG_LOW           = (uint32_t) init_values.compare_value;
  timer->CONF_REG.bit.auto_reload = init_values.auto_reload;
  timer->CONF_REG.bit.interrupt    = init_values.interrupt_enable;
  timer->CONF_REG.bit.enable       = 1;

  return TIMER_VALUE_OK;
}

TIMER_ValueType TIMER_Start(TIMER_Type *timer) {
  timer->CONF_REG.bit.start = 1;

  return TIMER_VALUE_OK;
}

uint64_t TIMER_Get_Counter(TIMER_Type *timer) {
  return (((uint64_t) timer->DATA_REG_HIGH) << 32) | ((uint64_t) timer->
    DATA_REG_LOW);
}

TIMER_ValueType TIMER_Stop(TIMER_Type *timer) {
  timer->CONF_REG.bit.enable = 0;

  return TIMER_VALUE_OK;
}

TIMER_ValueType TIMER_Wait(TIMER_Type *timer) {
  while(timer->CONF_REG.bit.overflow == 0);
```

```
  timer->CONF_REG.bit.overflow = 0;

  return TIMER_VALUE_OK;
}
```

Appendix C: ThreadX

C.1 Port files

Code C.1: tx_port.h.

```
#ifndef TX_PORT_H
#define TX_PORT_H


/* Include prototypes for memset.  */
#include "bsp.h"


/* Determine if the optional ThreadX user define file should be used.  */
#define TX_INCLUDE_USER_DEFINE_FILE
#ifdef TX_INCLUDE_USER_DEFINE_FILE


/* Yes, include the user defines in tx_user.h. The defines in this file may
   alternately be defined on the command line.  */

#include "tx_user.h"
#endif


/* Define compiler library include files.  */


/* Define ThreadX basic types for this port.  */

#define VOID                                    void
typedef char                                    CHAR;
typedef unsigned char                           UCHAR;
typedef int                                     INT;
typedef unsigned int                            UINT;
typedef long                                    LONG;
typedef unsigned long                           ULONG;
typedef short                                   SHORT;
typedef unsigned short                          USHORT;


/* Define the priority levels for ThreadX.  Legal values range
   from 32 to 1024 and MUST be evenly divisible by 32.  */

#ifndef TX_MAX_PRIORITIES
#define TX_MAX_PRIORITIES                       32
#endif


/* Define the minimum stack for a ThreadX thread on this processor. If the
   size supplied during thread creation is less than this value, the
   thread create call will return an error.  */

#ifndef TX_MINIMUM_STACK
#define TX_MINIMUM_STACK 512    /* Minimum stack size for this port  */
#endif


/* Define the system timer thread's default stack size and priority.
   These are only applicable if TX_TIMER_PROCESS_IN_ISR is not defined.  */
```

```
#ifndef TX_TIMER_THREAD_STACK_SIZE
#define TX_TIMER_THREAD_STACK_SIZE 1024 /* Default timer thread stack size
    */
#endif

#ifndef TX_TIMER_THREAD_PRIORITY
#define TX_TIMER_THREAD_PRIORITY 0 /* Default timer thread priority    */
#endif


/* Define various constants for the ThreadX RISC-V port.  */

#define TX_INT_DISABLE  0x00000000  /* Disable interrupts value */
#define TX_INT_ENABLE   0x00000008  /* Enable interrupt value   */


/* Define the clock source for trace event entry time stamp. The following
   two item are port specific. For example, if the time source is at the
   address 0x0a800024 and is 16-bits in size, the clock source constants
    would be:

#define TX_TRACE_TIME_SOURCE                           *((ULONG *) 0x0a800024)
#define TX_TRACE_TIME_MASK                             0x0000FFFFUL

*/

#ifndef TX_TRACE_TIME_SOURCE
#define TX_TRACE_TIME_SOURCE                           ++_tx_trace_simulated_time
#endif
#ifndef TX_TRACE_TIME_MASK
#define TX_TRACE_TIME_MASK                             0xFFFFFFFFUL
#endif


/* Define the port specific options for the _tx_build_options variable.
   This variable indicates how the ThreadX library was built.  */

#define TX_PORT_SPECIFIC_BUILD_OPTIONS                 0


/* Define the in-line initialization constant so that modules with in-line
   initialization capabilities can prevent their initialization from being
   a function call.  */

#define TX_INLINE_INITIALIZATION


/* Determine whether or not stack checking is enabled. By default, ThreadX
   stack checking is disabled. When the following is defined, ThreadX
   thread stack checking is enabled. If stack checking is enabled
   (TX_ENABLE_STACK_CHECKING is defined), the TX_DISABLE_STACK_FILLING
   define is negated, thereby forcing the stack fill which is necessary
   for the stack checking logic.  */

#ifdef TX_ENABLE_STACK_CHECKING
#undef TX_DISABLE_STACK_FILLING
#endif


/* Define the TX_THREAD control block extensions for this port. The main
   reason for the multiple macros is so that backward compatibility can
   be maintained with existing ThreadX kernel awareness modules.  */

#define TX_THREAD_EXTENSION_0
#define TX_THREAD_EXTENSION_1
#define TX_THREAD_EXTENSION_2
#define TX_THREAD_EXTENSION_3


/* Define the port extensions of the remaining ThreadX objects.  */

#define TX_BLOCK_POOL_EXTENSION
```

```
#define TX_BYTE_POOL_EXTENSION
#define TX_EVENT_FLAGS_GROUP_EXTENSION
#define TX_MUTEX_EXTENSION
#define TX_QUEUE_EXTENSION
#define TX_SEMAPHORE_EXTENSION
#define TX_TIMER_EXTENSION


/* Define the user extension field of the thread control block.  Nothing
   additional is needed for this port so it is defined as white space.  */

#ifndef TX_THREAD_USER_EXTENSION
#define TX_THREAD_USER_EXTENSION
#endif


/* Define the macros for processing extensions in tx_thread_create,
   tx_thread_delete, tx_thread_shell_entry, and tx_thread_terminate.  */

#define TX_THREAD_CREATE_EXTENSION(thread_ptr)
#define TX_THREAD_DELETE_EXTENSION(thread_ptr)
#define TX_THREAD_COMPLETED_EXTENSION(thread_ptr)
#define TX_THREAD_TERMINATED_EXTENSION(thread_ptr)


/* Define ThreadX object creation extensions for the remaining objects.  */

#define TX_BLOCK_POOL_CREATE_EXTENSION(pool_ptr)
#define TX_BYTE_POOL_CREATE_EXTENSION(pool_ptr)
#define TX_EVENT_FLAGS_GROUP_CREATE_EXTENSION(group_ptr)
#define TX_MUTEX_CREATE_EXTENSION(mutex_ptr)
#define TX_QUEUE_CREATE_EXTENSION(queue_ptr)
#define TX_SEMAPHORE_CREATE_EXTENSION(semaphore_ptr)
#define TX_TIMER_CREATE_EXTENSION(timer_ptr)


/* Define ThreadX object deletion extensions for the remaining objects.  */

#define TX_BLOCK_POOL_DELETE_EXTENSION(pool_ptr)
#define TX_BYTE_POOL_DELETE_EXTENSION(pool_ptr)
#define TX_EVENT_FLAGS_GROUP_DELETE_EXTENSION(group_ptr)
#define TX_MUTEX_DELETE_EXTENSION(mutex_ptr)
#define TX_QUEUE_DELETE_EXTENSION(queue_ptr)
#define TX_SEMAPHORE_DELETE_EXTENSION(semaphore_ptr)
#define TX_TIMER_DELETE_EXTENSION(timer_ptr)


/* Define ThreadX interrupt lockout and restore macros for protection on
   access of critical kernel information.  The restore interrupt macro must
   restore the interrupt posture of the running thread prior to the value
   present prior to the disable macro.  In most cases, the save area macro
   is used to define a local function save area for the disable and restore
   macros.  */
#define TX_DISABLE_INLINE
#ifdef TX_DISABLE_INLINE

unsigned int _tx_thread_interrupt_control(unsigned int new_posture);

#define TX_INTERRUPT_SAVE_AREA  register INT interrupt_save;

#define TX_DISABLE  interrupt_save =  _tx_thread_interrupt_control(
    TX_INT_DISABLE);
#define TX_RESTORE  _tx_thread_interrupt_control(interrupt_save);

#else

#define TX_INTERRUPT_SAVE_AREA __istate_t interrupt_save;
#define TX_DISABLE {interrupt_save = __get_interrupt_state();
    __disable_interrupt();};
#define TX_RESTORE {__set_interrupt_state(interrupt_save);};

#endif
```

```
/* Define the interrupt lockout macros for each ThreadX object.  */

#define TX_BLOCK_POOL_DISABLE                    TX_DISABLE
#define TX_BYTE_POOL_DISABLE                     TX_DISABLE
#define TX_EVENT_FLAGS_GROUP_DISABLE             TX_DISABLE
#define TX_MUTEX_DISABLE                         TX_DISABLE
#define TX_QUEUE_DISABLE                         TX_DISABLE
#define TX_SEMAPHORE_DISABLE                     TX_DISABLE


/* Define the version ID of ThreadX.This may be utilized by the application
   .  */

#ifdef TX_THREAD_INIT
CHAR _tx_version_id[] =
                        "Copyright (c) Microsoft Corporation. All rights
   reserved.  *  ThreadX RISC-V32/IAR Version G6.1.6 *";
#else
extern  CHAR                      _tx_version_id[];
#endif

#endif
```

Code C.2: tx_initialize_low_level.S.

```
/* #define TX_SOURCE_CODE  */

/* Include necessary system files.  */

/*  #include "tx_api.h"
    #include "tx_initialize.h"
    #include "tx_thread.h"
    #include "tx_timer.h"  */

    .section .data
    .global  __tx_free_memory_start
__tx_free_memory_start:

    .section .text

/* VOID   _tx_initialize_low_level(VOID)
{  */
    .global  _tx_initialize_low_level
_tx_initialize_low_level:
    sw  sp, _tx_thread_system_stack_ptr, t0  # Save system stack pointer

    la  t0, __tx_free_memory_start           # Pickup first free address
    sw  t0, _tx_initialize_unused_memory, t1 # Save unused memory address

    nop
    li    t0, 0x80
    csrw  mie, t0                             # Enable systick interrupt
    csrwi mstatus, 0x8                        # Enable global interrupt

    ret


    /* Define the actual timer interrupt/exception handler.  */

    .global _tx_timer_interrupt_handler
_tx_timer_interrupt_handler:

    /* Before calling _tx_thread_context_save, we have to allocate an
    interrupt stack frame and save the current value of x1 (ra).  */
.if __riscv_base_isa == rv32f
    addi     sp, sp, -260 # Allocate space for all registers - with floating
         point enabled
.else
    addi     sp, sp, -128 # Allocate space for all registers - without
```

```
        floating point enabled
.endif
    sw      x1, 0x70(sp)  # Store RA
    call    _tx_thread_context_save # Call ThreadX context save

    /* Call the ThreadX timer routine.  */
    call    _tx_timer_interrupt  # Call timer interrupt handler

    /* Timer interrupt processing is done, jump to ThreadX context restore.
         */
    j _tx_thread_context_restore # Jump to ThreadX context restore
        function. Note: this does not return!
```

Appendix D: Tests

D.1 Ibex tests

Code D.1: test_alu.S.

```
/* Vector table */
.org 0x00
.rept 31
    nop
.endr
    jal x0, boot

/* Reset */
.org 0x80
    jal x0, boot

/* Init Registers */
boot:
    mv   x1, x0
    mv   x2, x1
    mv   x3, x1
    mv   x4, x1
    mv   x5, x1
    mv   x6, x1
    mv   x7, x1
    mv   x8, x1
    mv   x9, x1
    mv  x10, x1
    mv  x11, x1
    mv  x12, x1
    mv  x13, x1
    mv  x14, x1
    mv  x15, x1
    mv  x16, x1
    mv  x17, x1
    mv  x18, x1
    mv  x19, x1
    mv  x20, x1
    mv  x21, x1
    mv  x22, x1
    mv  x23, x1
    mv  x24, x1
    mv  x25, x1
    mv  x26, x1
    mv  x27, x1
    mv  x28, x1
    mv  x29, x1
    mv  x30, x1
    mv  x31, x1

/* Test ADDI */
main:
    addi x1 , x0,   1000  /* x1  = 1000 0x3E8 */
    addi x2 , x1,   2000  /* x2  = 3000 0xBB8 */
    addi x3 , x2,  -1000  /* x3  = 2000 0x7D0 */
    addi x4 , x3,  -2000  /* x4  = 0    0x000 */
    addi x5 , x4,   1000  /* x5  = 1000 0x3E8 */
    addi x6 , x5,   2000  /* x6  = 3000 0xBB8 */
    addi x7 , x6,  -1000  /* x7  = 2000 0x7D0 */
    addi x8 , x7,  -2000  /* x8  = 0    0x000 */
    addi x9 , x8,   1000  /* x9  = 1000 0x3E8 */
```

```
addi x10, x9,   2000  /* x10 = 3000 0xBB8 */
addi x11, x10, -1000  /* x11 = 2000 0x7D0 */
addi x12, x11, -2000  /* x12 = 0    0x000 */
addi x13, x12,  1000  /* x13 = 1000 0x3E8 */
addi x14, x13,  2000  /* x14 = 3000 0xBB8 */
addi x15, x14, -1000  /* x15 = 2000 0x7D0 */

/* Test Positive numbers */
slti x15, x1,  1200   /* x15 = 1 */
slti x14, x2,  1200   /* x14 = 0 */

/* Test Negative numbers */
addi x3,  x0, -1000
slti x13, x3, -1200   /* x13 = 0 */
slti x12, x3, -900    /* x12 = 1 */

/* Test Unsigned */
sltiu x11, x1,  1200   /* x11 = 1 */
sltiu x10, x2,  1200   /* x10 = 0 */

/* Test XOR */
addi x1, x0, 0x7FF
addi x2, x0, 0x70F
addi x3, x0, 0x0F0

xori x15, x1, -1      /*  x15 = 0xFFFFF800 */
xori x14, x2, -1      /*  x14 = 0xFFFFF8F0 */
xori x13, x3, -1      /*  x13 = 0xFFFFFF0F */

/* Test OR */
addi x1, x0, 0x7FF
addi x2, x0, 0x70F
addi x3, x0, 0x0F0

ori x15, x1, -1      /*  x15 = 0xFFFFFFFF */
ori x14, x2, -1      /*  x14 = 0xFFFFFFFF */
ori x13, x3, -1      /*  x13 = 0xFFFFFFFF */
ori x12, x1,  0      /*  x12 = 0x000007FF */
ori x11, x2,  0      /*  x11 = 0x0000070F */
ori x10, x3,  0      /*  x10 = 0x000000F0 */

/* Test AND */
addi x1, x0, 0x7FF
addi x2, x0, 0x70F
addi x3, x0, 0x0F0

andi x15, x1, -1      /*  x15 = 0x000007FF */
andi x14, x2, -1      /*  x14 = 0x0000070F */
andi x13, x3, -1      /*  x13 = 0x000000F0 */
andi x12, x1, 341     /*  x12 = 0x00000155 */
andi x11, x2, 341     /*  x11 = 0x00000105 */
andi x10, x3, 341     /*  x10 = 0x00000050 */

/* Test Shift Left */
addi x1, x0, 0x7FF
slli x15, x1, 0       /* x15 = 0x000007FF */
slli x14, x1, 1       /* x14 = 0x00000FFE */
slli x13, x1, 2       /* x13 = 0x00001FFC */
slli x12, x1, 3       /* x12 = 0x00003FF8 */
slli x11, x1, 8       /* x11 = 0x0007FF00 */
slli x10, x1, 16      /* x10 = 0x07FF0000 */
slli x9, x1,  24      /* x9  = 0xFF000000 */

/* Test Shift Right */
addi x6,  x9, 0       /* Backup X9 with 0xFF000000 */
addi x1,  x9, 0
srli x15, x1, 0       /* x15 = 0xFF000000 */
srli x14, x1, 1       /* x14 = 0x7F800000 */
srli x13, x1, 2       /* x13 = 0x3FC00000 */
srli x12, x1, 3       /* x12 = 0x1FE00000 */
srli x11, x1, 8       /* x11 = 0x00FF0000 */
```

```
    srli x10, x1, 16       /* x10 = 0x0000FF00 */
    srli x9,  x1, 24       /* x9  = 0x000000FF */

    /* Test Shift Right */
    addi x1,  x6, 0
    srai x15, x1, 0        /* x15 = 0xFF000000 */
    srai x14, x1, 1        /* x14 = 0xFF800000 */
    srai x13, x1, 2        /* x13 = 0xFFC00000 */
    srai x12, x1, 3        /* x12 = 0xFFE00000 */
    srai x11, x1, 8        /* x11 = 0xFFFF0000 */
    srai x10, x1, 16       /* x10 = 0xFFFFFF00 */
    srai x9,  x1, 24       /* x9  = 0xFFFFFFFF */

    /* Test ADD */
    addi x1, x6, 0         /* x1  = 0xFF000000 */
    add  x15, x1, x1       /* x15 = 0xFE000000 */
    add  x14, x1, x9       /* x14 = 0xFEFFFFFF */
    add  x13, x0, x1       /* x13 = 0xFF000000 */

    /* Test SUB */
    addi x1, x6, 0         /* x1  = 0xFF000000 */
    sub  x15, x1, x1       /* x15 = 0x00000000 */
    sub  x14, x1, x9       /* x14 = 0xFF000001 */
    sub  x13, x0, x1       /* x13 = 0x01000000 */

    /* NOPs */
    nop
    nop
    nop
```

Code D.2: test_lui_auipc.S.

```
/* Vector table */
.org 0x00
.rept 31
    nop
.endr
    jal x0, boot

/* Reset */
.org 0x80
    jal x0, boot

/* Init Registers */
boot:
    mv  x1, x0
    mv  x2, x1
    mv  x3, x1
    mv  x4, x1
    mv  x5, x1
    mv  x6, x1
    mv  x7, x1
    mv  x8, x1
    mv  x9, x1
    mv x10, x1
    mv x11, x1
    mv x12, x1
    mv x13, x1
    mv x14, x1
    mv x15, x1
    mv x16, x1
    mv x17, x1
    mv x18, x1
    mv x19, x1
    mv x20, x1
    mv x21, x1
    mv x22, x1
    mv x23, x1
    mv x24, x1
    mv x25, x1
    mv x26, x1
```

```
    mv x27, x1
    mv x28, x1
    mv x29, x1
    mv x30, x1
    mv x31, x1

main:
    lui x1, 0xFFFFF    /*  x1 = 0xFFFFF000 */
    nop
    nop
    nop
    nop
    nop
    auipc x2, 0xFFFFF /*  x2 = 0xFFFFF018 */
    nop
```

Code D.3: test_jal.S.

```
/* Vector table */
.org 0x00
.rept 31
    nop
.endr
    jal x0, boot

/* Reset */
.org 0x80
    jal x0, boot

/* Init Registers */
boot:
    mv  x1, x0
    mv  x2, x1
    mv  x3, x1
    mv  x4, x1
    mv  x5, x1
    mv  x6, x1
    mv  x7, x1
    mv  x8, x1
    mv  x9, x1
    mv x10, x1
    mv x11, x1
    mv x12, x1
    mv x13, x1
    mv x14, x1
    mv x15, x1
    mv x16, x1
    mv x17, x1
    mv x18, x1
    mv x19, x1
    mv x20, x1
    mv x21, x1
    mv x22, x1
    mv x23, x1
    mv x24, x1
    mv x25, x1
    mv x26, x1
    mv x27, x1
    mv x28, x1
    mv x29, x1
    mv x30, x1
    mv x31, x1

main:
    nop
    nop
    nop
    nop
    nop
    jal x1, branch1
```

```
branch2:
    beq x0, x0, branch2 /* Unreachable */

branch3:
    addi x3, x0, 1
    addi x3, x0, 2

branch1:
    addi x3, x0, 3
    jalr x2, x1, 4
    addi x3, x0, 4
    addi x3, x0, 5
    addi x3, x0, 6

    nop
```

Code D.4: test_jmps.S.

```
/* Vector table */
.org 0x00
.rept 31
    nop
.endr
    jal x0, boot

/* Reset */
.org 0x80
    jal x0, boot

/* Init Registers */
boot:
    mv  x1, x0
    mv  x2, x1
    mv  x3, x1
    mv  x4, x1
    mv  x5, x1
    mv  x6, x1
    mv  x7, x1
    mv  x8, x1
    mv  x9, x1
    mv x10, x1
    mv x11, x1
    mv x12, x1
    mv x13, x1
    mv x14, x1
    mv x15, x1
    mv x16, x1
    mv x17, x1
    mv x18, x1
    mv x19, x1
    mv x20, x1
    mv x21, x1
    mv x22, x1
    mv x23, x1
    mv x24, x1
    mv x25, x1
    mv x26, x1
    mv x27, x1
    mv x28, x1
    mv x29, x1
    mv x30, x1
    mv x31, x1

main:
    nop
    nop
    nop
    nop

    addi x15, x0, 1000    /* X15 = 1000 */
```

```
    /* Test BEQ */
    beq x0, x0, branch1
    nop

lbl0:
    beq x0, x15, should_not_branch1
    nop
    beq x0, x0, lbl1

branch1:
    beq x0, x0, lbl0
    nop

should_not_branch1:
    nop

    /* Test BNE */
lbl1:
    bne x0, x0, should_not_branch1
    nop
    bne x0, x15, lbl2
    nop
    beq x0, x0, lbl1

should_not_branch2:
    nop

lbl2:
    /* Test BLT Positive */
    addi x1, x0, 1200
    addi x2, x0, 1400
    blt  x2, x1, should_not_branch2
    blt  x1, x2, lbl3
    beq  x0, x0, lbl2

should_not_branch3:
    nop

lbl3:
    /* Test BLT Negative */
    addi x1, x0, -1400
    addi x2, x0, -1200
    blt  x2, x1, should_not_branch3
    blt  x1, x2, lbl4
    beq  x0, x0, lbl3

should_not_branch4:
    nop

lbl4:
    /* Test BGE Positive */
    addi x1, x0, 1200
    addi x2, x0, 1400
    bge  x1, x2, should_not_branch4
    bge  x2, x1, lbl5
    beq  x0, x0, lbl4

lbl5:
    /* Test BGE Equal */
    addi x1, x0, 1200
    addi x2, x0, 1200
    bge  x2, x1, lbl6
    beq  x0, x0, lbl5

lbl6:
    /* Test BGE Negative */
    addi x1, x0, -1400
    addi x2, x0, -1200
    bge  x1, x2, should_not_branch4
    bge  x2, x1, lbl7
    beq  x0, x0, lbl6
```

```
should_not_branch5:
    nop

lbl7:
    /* Test BLTU */
    addi x1, x0, 1200
    addi x2, x0, 1400
    bltu x2, x1, should_not_branch5
    bltu x1, x2, lbl8
    beq  x0, x0, lbl7


should_not_branch6:
    nop

lbl8:
    /* Test BGEU */
    addi x1, x0, 1200
    addi x2, x0, 1400
    bgeu x1, x2, should_not_branch6
    bgeu x2, x1, lbl9
    beq  x0, x0, lbl8

lbl9:
    nop
    nop
```

Code D.5: test_load_store.

```
/* Vector table */
.org 0x00
.rept 31
    nop
.endr
    jal x0, boot

/* Reset */
.org 0x80
    jal x0, boot

/* Init Registers */
boot:
    mv   x1, x0
    mv   x2, x1
    mv   x3, x1
    mv   x4, x1
    mv   x5, x1
    mv   x6, x1
    mv   x7, x1
    mv   x8, x1
    mv   x9, x1
    mv  x10, x1
    mv  x11, x1
    mv  x12, x1
    mv  x13, x1
    mv  x14, x1
    mv  x15, x1
    mv  x16, x1
    mv  x17, x1
    mv  x18, x1
    mv  x19, x1
    mv  x20, x1
    mv  x21, x1
    mv  x22, x1
    mv  x23, x1
    mv  x24, x1
    mv  x25, x1
    mv  x26, x1
    mv  x27, x1
    mv  x28, x1
    mv  x29, x1
```

```
    mv x30, x1
    mv x31, x1

main:
    lui x1, %hi(data0)
    addi x1, x1, %lo(data0)
    lw x15, 0(x1)              /* x15 = 0xDEADBEEF */
    nop

testhalf:
    lhu x14, 0(x1)              /* x14 = 0x0000BEEF */
    lhu x13, 1(x1)              /* x13 = 0x0000ADBE */
    lhu x12, 2(x1)              /* x12 = 0x0000DEAD */
    nop

testbyte:
    lbu x11, 0(x1)              /* x11 = 0x000000EF */
    lbu x10, 1(x1)              /* x10 = 0x000000BE */
    lbu x9, 2(x1)               /* x9  = 0x000000AD */
    lbu x8, 3(x1)               /* x8  = 0x000000DE */
    nop

    lui x1, %hi(data1)
    addi x1, x1, %lo(data1)

testhalfsign:
    lh x15, 0(x1)               /* x15 = 0xFFFF8281 */
    lh x14, 1(x1)               /* x14 = 0xFFFF8382 */
    lh x13, 2(x1)               /* x13 = 0xFFFF8483 */
    nop

testbytesign:
    lb x12, 0(x1)               /* x11 = 0xFFFFFF81 */
    lb x11, 1(x1)               /* x10 = 0xFFFFFF82 */
    lb x10, 2(x1)               /* x9  = 0xFFFFFF83 */
    lb x9,  3(x1)               /* x8  = 0xFFFFFF84 */
    nop

teststorebyte:
    lui x1, %hi(data1)
    addi x1, x1, %lo(data1)
    lw x12, 0(x1)               /* x11      = 0x84838281 */

    lui x1, %hi(data2)
    addi x1, x1, %lo(data2)

    sb x12, 0(x1)               /* data2[0] = 0x00000081 */
    sh x12, 4(x1)               /* data2[1] = 0x00008281 */
    sw x12, 8(x1)               /* data2[2] = 0x84838281 */
    nop

testunalignedstore:
    sw x0, 0(x1)
    sw x0, 4(x1)
    sw x0, 8(x1)
    sw x0, 16(x1)

    sb x12, 0(x1)                /* data2[0] = 0x00000081 */
    sb x12, 5(x1)                /* data2[1] = 0x00008100 */
    sb x12, 10(x1)               /* data2[2] = 0x00810000 */
    sb x12, 15(x1)               /* data2[3] = 0x81000000 */

    sw x0, 16(x1)
    sw x0, 20(x1)
    sw x0, 24(x1)

    sh x12, 16(x1)               /* data2[4] = 0x00008281 */
    sh x12, 21(x1)               /* data2[5] = 0x00828100 */
    sh x12, 26(x1)               /* data2[6] = 0x82810000 */
    nop
    nop
    nop
```

```
    nop

.align 4
data0:
    .word 0xDEADBEEF

data1:
    .word 0x84838281

.section .data
data2:
    .word 0x00000000 /* data2[0] => data2 + 0 */
    .word 0x00000000 /* data2[1] => data2 + 4 */
    .word 0x00000000 /* data2[2] => data2 + 8 */
    .word 0x00000000 /* data2[3] => data2 + 12 */

    .word 0x00000000 /* data2[4] => data2 + 16 */
    .word 0x00000000 /* data2[5] => data2 + 20 */
    .word 0x00000000 /* data2[6] => data2 + 24 */
    .word 0x00000000 /* data2[7] => data2 + 28 */
```

D.2 System test

Code D.6: main.c.

```
#include "bsp/inc/bsp.h"
#include "bsp/inc/timer.h"
#include "bsp/inc/i2c.h"
#include "threadx/common/inc/tx_api.h"

#define DEMO_STACK_SIZE 64
#define DEMO_QUEUE_SIZE 8

UCHAR stack_thread_0[DEMO_STACK_SIZE];
TX_THREAD thread_0;

UCHAR stack_thread_1[DEMO_STACK_SIZE];
TX_THREAD thread_1;

TX_QUEUE queue_0;

void thread_0_entry(ULONG thread_input) {
    UINT status = 0;
    UINT rtc_sec_elapsed = 0;

    RTC_Set_Time(0, 0, 0); /* Starts the RTC */
    TIMER_Start(timer1);

    while(1) {
        rtc_sec_elapsed = RTC_Get_Seconds();

        /* Send data to queue 0 */
        status =  tx_queue_send(&queue_0, &rtc_sec_elapsed, TX_WAIT_FOREVER
   );

        /* Check completion status */
        if (status != TX_SUCCESS)
            break;

        /* Sleep 500ms */
        tx_thread_sleep(50);
    }
}

void thread_1_entry(ULONG thread_input) {
```

```
    UINT tim_sec_elapsed = 0;;
    UINT status = 0;
    UINT rtc_sec_elapsed = 0;

    while(1) {
        /* Retrieve a message from the queue */
        status = tx_queue_receive(&queue_0, &rtc_sec_elapsed,
    TX_WAIT_FOREVER);

        /* Convert to seconds */
        tim_sec_elapsed = (TIMER_Get_Counter(timer1) / SYS_CLK_FREQ);

        /* Check completion status */
        if (status != TX_SUCCESS)
            break;

        /* Compare both values */
        if(tim_sec_elapsed != rtc_sec_elapsed) {
            break;
        };
    }
}

void tx_application_define(void *first_unused_memory) {
    ULONG queue[DEMO_QUEUE_SIZE];

    tx_thread_create(&thread_0, "thread 0", thread_0_entry, 0,
                     stack_thread_0, DEMO_STACK_SIZE,
                     4, 4, TX_NO_TIME_SLICE, TX_AUTO_START);

    tx_thread_create(&thread_1, "thread 1", thread_1_entry, 1,
                     stack_thread_1, DEMO_STACK_SIZE,
                     4, 4, TX_NO_TIME_SLICE, TX_AUTO_START);

    tx_queue_create(&queue_0, "queue 0", TX_1_ULONG, &queue,
    DEMO_QUEUE_SIZE * sizeof(ULONG));
}

int main(int argc, char **argv) {
    /* Configure timer 0 as a 10ms systick */
    TIMER_InitType timer0_values;
    timer0_values.compare_value = SYS_CLK_FREQ / 100;
    timer0_values.auto_reload = 1;
    timer0_values.interrupt_enable = 1;
    TIMER_Initialize(timer0, timer0_values);
    TIMER_Start(timer0);

    /* Configure I2C */
    I2C_InitType i2c_values;
    i2c_values.interrupt_enable = 0;
    i2c_values.OperationMode = Standard;
    I2C_Initialize(i2c, i2c_values);

    /* Configure timer 1 */
    TIMER_InitType timer1_values;
    timer1_values.compare_value = SYS_CLK_FREQ * 60 * 5;
    timer1_values.auto_reload = 0;
    timer1_values.interrupt_enable = 0;
    TIMER_Initialize(timer1, timer1_values);

    /* ThreadX scheduler */
    tx_kernel_enter();

    /* Unreachable code */
    while (1) {}

    return 0;
}
```