Low Power Bluetooth Baseband with a RISC-V Core for Autonomous Sensors using Chisel

Ignacio Garcia Ezquerro

Master of Science Thesis

Delft University of Technology

Microelectronics Department
Electronic Instrumentation Laboratory

Low Power Bluetooth Baseband with a RISC-V Core for Autonomous Sensors using Chisel

Master of Science Thesis

For the degree of Master of Science in Electrical Engineering at Delft University of Technology

By
Ignacio Garcia Ezquerro

Supervisor:
Dr. Ir. Sijun Du

October 6, 2022

Student Number:
5366844

Faculty of Electrical Engineering, Mathematics and Computer Science (EWI) · Delft University of Technology

Delft
University of
Technology

Microelectronics Department

Abstract

The number of Internet of Things (IoT) devices and their applications is projected to continue increasing in the future, creating a demand for low-power wireless communications that allow IoT devices to either be connected directly to the internet or to use other devices as a back gate to it. Bluetooth Low Energy (BLE) is already one of the most used wireless communication protocols for this kind of devices, and its use is expected to continue increasing in the near future. Therefore the need for very low-power Bluetooth baseband designs arises. This thesis work presents a study on previous existing BLE baseband architectures. It proposes a new BLE baseband architecture where every non-bit-intensive function has been moved into software to fully make use of the efficient processor core in the System on Chip (SoC), and new functionalities have been added to the design in order to eliminate unnecessary communication between the baseband and processor. Furthermore, a novel non table based Cyclic Redundancy Check (CRC) error correction algorithm has been implemented in hardware and included in the bitstream processing chain in order to improve the reliability of the connection and therefore further reduce power consumption. In order to test the functionality of the proposed architecture, the bitstream processing blocks have been implemented and connected to a RISC-V processor core on the designed SoC. That was possible thanks to Chisel, a novel Hardware Description Language (HDL) based on Scala, and the Rocket-Chip generator that facilitates the task of integrating parameterizable cores and accelerators/coprocessors into an SoC design. The presented design has been simulated and implemented on a Field Programmable Gate Array (FPGA) to test it and show its feasibility.

Table of Contents

List of Figures

List of Tables

Acknowledgements

I would like to thanks first my supervisor Dr. Sijun Du for introducing me to this interesting topic and his support and availability through all the project duration.

I also want to thank the Electronic Instrumentation Lab for the support received through adversities during the year and the chance to work and learn there.

I would like to thank also my fellow microelectronics's students, and of them David, Emre, Filippo, Tariq, Alex, Nick, Victor and Ming Zhe for their constant support during this two years journey. At this point I have to confess my most relevant finding, the best tool of an engineer is more engineers.

I want to thank also TU Delft for the opportunity that allowed me to meet such an amazing mix of people and cultures in addition to spend the last two years surrounded by brilliant minds and professionals. On the same note, I want to thank also my previous university and previous fellow students, for giving me the tools, motivation and support to come here.

I want to dedicate my last paragraph to thank my family for their constant support and care, and all the friends I made along the way here.

Delft, University of Technology
October 6, 2022

Ignacio Garcia Ezquerro

Chapter 1

Introduction

Digital circuits play a significant role in most electronic devices, and optimizing them to suit the specific requirements of their applications has become an important task. While some of these circuits are required to achieve the highest computation capabilities or the lowest latencies, others are required to achieve a balance on those two parameters while achieving the lowest possible power consumption. That is the case for the Internet of Things (IoT) domain, which englobes sensors, biomedical implants, wearable devices, beacons for localization services, etc [10].

New wireless communication protocols and their high integration have made possible to get rid of the previously used wired communications for many applications and devices, leading to the rise of the IoT era and the exponential increase in the number of IoT devices [10].

Some statistics become necessary to understand the reasons behind the phrase "the rise of the IoT era" and the importance of Bluetooth Low Energy (BLE). First, it is estimated that around 18.8 billion devices are interconnected today. Out of that number, over 30 % of them use BLE as the enabling communication protocol. In addition to this, since 2018, all shipped smartphones and tablets already included BLE [10].

Among the newly emerged wireless technologies applied to IoT, BLE is one of the most used ones. BLE is a variation of the previously existing Bluetooth technology simplified to achieve a very low power consumption. The BLE technology was developed targeting some IoT use cases, such as applications in the healthcare field: fitness, beacons, home entertainment, security, and sensors [10]. Although BLE appeared as a variant of the previously existing Bluetooth protocol, they are not compatible but coexistent.

Many of these IoT applications require computation capabilities in the end node, the one with the highest requirement for low power consumption. It is then apparent that a combined design of its hardware is necessary to achieve ultra-low power consumption in a device where wireless communications and a processor are present.

In order to overcome that issue, the Rocket-Chip generator is used. The Rocket-Chip generator allows the complete design of a System on Chip (SoC) that consists of the desired number of processor cores (Rocket Core), of which several types are available and are customizable, and coprocessor accelerators.

All the modules are programmed using Chisel, although adding and integrating Verilog blocks is possible. Chisel is a Berkeley Hardware Description Language (HDL) language based on Scala that allows creating highly parameterizable hardware generators and obtaining synthesizable Verilog out of them.

A new architecture for a BLE baseband will be proposed, and the improved parts will be designed and implemented using the SoC generator and Chisel.

1-1 BLE as a Wireless Communication Protocol for IoT

Both technologies, BLE and Bluetooth, are developed and maintained by the same group: the Bluetooth Special Interest Group (SIG). However, there are substantial differences among them, allowing each technology to target different use cases. Therefore, it is necessary to introduce Bluetooth before going into detail about its Low Energy variant and introducing its application in the IoT domain.

1-1-1 Internet of Things

The IoT can be defined as:

"An open and comprehensive network of intelligent objects that have the capacity to auto-organize, share information, data and resources, reacting and acting in face of situations and changes in the environment." [11]

It can be considered a global wide network that makes communication possible between all things by providing them with a unique identity, where basically anything can communicate and be connected intelligently [12]. Ever since the rise of the IoT, the number of connected IoT devices has exponentially increased, the forecast for the end of 2022 is 18 billion. Therefore, the variety of IoT devices and their application is very wide, and so is the variety of communication protocols they use. Looking specifically at small battery-powered IoT devices, we have several wireless communication protocols: ZigBee, Radio Frequency Identification (RFID), BLE, Near Field Communication (NFC), IEE 802.15.4, and WiFi, among others.

Although BLE is not the most used Wireless communication protocol in IoT, it is among the most used ones. To quantify this statement about Bluetooth's importance, the Bluetooth SIG estimated that around 35% of all connected IoT devices currently rely on Bluetooth technology.

1-1-2 Bluetooth

Given that BLE has its origin in the previously existing Bluetooth protocol, it becomes necessary to introduce the latter Bluetooth protocol.

The first consumer device using Bluetooth (a hands-free mobile headset) was released in 1999. The protocol was developed as a short-link radio communication to replace wires in headsets by Ericsson Mobile in Sweden. A year before that first product's release, the Bluetooth SIG was created with five members: Intel, Toshiba, Ericsson Mobile, IBM, and Nokia. That same organization, the Bluetooth SIG, oversees the development and licensing of the Bluetooth standard.

Bluetooth protocol operates at frequencies between 2.402 and 2.480 GHz or 2.400 and 2.4835 GHz in the globally unlicensed Industrial, Scientific and Medical (ISM) 2.4 GHz short-range radio frequency band. The data is divided into packets, and each one of the packets is transmitted on one of the 79 channels the Bluetooth protocol defines. Each channel has a 1 MHz width, making use of the radio technology called frequency-hopping spread spectrum and performing up to 1600 hops per second when the Adaptive Frequency-Hoping (AFH) is enabled.

1-1-3 Bluetooth Low Energy

Bluetooth Low Energy (BLE), previously named Bluetooth Smart, was initially developed by Nokia (previously named Wibree [13]) and later introduced in the Bluetooth 4.0 Core Specification [14] in 2009. It is not compatible with classic Bluetooth (either Basic Rate (BR) or Enhanced Data Rate (EDR)) but can coexist with it, which means that often both protocols are supported by the same device.

BLE continues to be designed and developed by the Bluetooth SIG, aiming at novel applications in the healthcare, fitness, beacons, security, and home entertainment industries. Its objective is to provide considerably reduced power consumption and cost while maintaining a similar communication range.

BLE uses the same 2.4 GHz frequency as the classic Bluetooth but with a more straightforward modulation system, effectively sharing the same antenna in dual mode devices (Classic Bluetooth and BLE) but requiring an additional modulation system.

In contrast to Classic Bluetooth, the same spectrum range of 2.4 to 2.4835 GHz is now divided into 40 channels of 2 MHz width instead of the 79 channels of 1 MHz width of BR/EDR. The channel distribution through the spectrum can be seen in Figure 1-1. The data is transmitted on each channel using Gaussian Frequency Shift Key (GFSK) modulation, and the bit rates, which depend on the mode (Low Energy Coded (LEC)/Low Energy 1 Msym/s (LE1M)/Low Energy 2 Msym/s (LE2M)), are: 125 Kbits/s, 500 Kbits/s, 1 Mbits/s, and 2 Mbits/s respectively. However, the most important difference between classic Bluetooth modes (BR/EDR) and BLE is that in BLE, the module remains in sleep mode unless a connection is initiated and data is transmitted.

It is important to introduce how the BLE connection methods work to allow the system to go into sleep mode. BLE supports two possible connection methods: Unicast (peer-peer, where data is exchanged between two devices) and Broadcast (where data is sent from one device and received by another without any exchange). Both possible connection modes use an advertising process to establish the initial connection or broadcast the data, where the devices initially take two roles: advertiser and scanner. Three channels are used for advertising/scanning out of the 40 available channels, which can be seen in Figure 1-1.

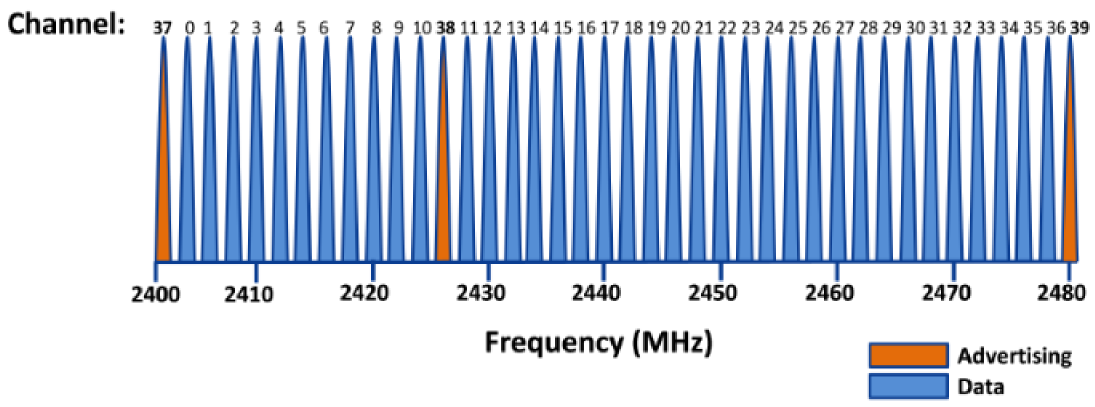


Figure 1-1: BLE channels [1].

The advertiser (device requesting a connection to be initiated or broadcasting data) sends advertisement packets in one of the 3 channels at a repetition period called advertising interval.

The scanner (device performing listening for advertisers) listens periodically on one of the advertisement channels at a scanning period and for the time defined by the scan window.

Therefore the discovery latency can be modeled as a probabilistic process that depends on three parameters: the advertising interval, the scanning period, and the scan window. Whenever a scanner correctly receives the advertisement from the advertiser, a connection request is sent back (in the case of a peer-to-peer connection), and the roles of the two devices change. Alternatively, the broadcasted data is received, and no reply or acknowledgment is sent back from the scanner.

Once the devices are discovered, both will stay in sleep mode unless the time to transmit/receive arrives, reducing power consumption. As can be expected, while a device performs, the discovery consumes a higher power than when the devices are already discovered and the transmission/listening times are established. That is why fixing the advertising interval, scanning period, and scan window are relevant and depend on the specific application (a study about the impact of these parameters on power consumption can be found in [10]).

By applying the previously described changes in addition to reduced adjustable transmission/reception power and reduced complexity in the algorithms applied in the bit processing over conventional Bluetooth modes (BR/EDR), BLE achieves the searched reduction in power consumption.

1-1-4 Use cases of BLE in IoT

BLE was specially conceived for specific application scenarios and is therefore well suited for sensors, actuators, and any other device with very low power consumption and size requirements. Among other features, BLE works well with a high number of communication nodes with reduced latency requirements, very low power consumption, and short wake-up and connection time while being as robust as Classic Bluetooth [15].

One of the key points of using BLE as an IoT wireless communication standard is that BLE is present in most hardware platforms such as laptops and phones. In addition, these same phones are usually used as a gateway for Internet access by many other IoT devices, allowing them to connect to the internet.

Considering that the BLE protocol was developed to operate with coin cell batteries that may last from months to several years, depending on the application, it is necessary to understand the communication requirements of each application in order to obtain an optimal implementation.

That is the reason why the two possible communication methods that BLE makes possible are relevant, and these two methods are peer-to-peer connection and broadcasting. In some applications, such as sensors, broadcasting data for any other device listening to it can be enough. In contrast, direct communication between the IoT device and a so-called central device is required in some applications. In addition, some other considerations are relevant, such as security or privacy of the transmitted data, latency, and time for discovery. Many of these considerations must be considered when setting up the parameters of a BLE connection and will highly impact its performance and total power consumption.

1-2 BLE Baseband Controller

In order to introduce in the following sections the proposed design, it is necessary to explain what a baseband controller is and what its function is inside the BLE protocol. The baseband controller is the hardware responsible for performing the necessary real-time operations and packet/frame processing required to enable communications compliant with the standard. That means processing received raw binary data into BLE protocol-compliant packets ready to be transmitted.

The BLE protocol stack can be seen in Figure 1-2, defining the BLE device and its functions. The protocol stack architecture consists of two parts, the controller and the host, and both are interfaced using Host Controller Interface (HCI). The Host and Application layers are implemented on software, while the Link Layer (LL) and the Physical Layer (PHY) are typically implemented in hardware.

The mentioned hardware layers are (LL and PHY) and their function:

- Link Layer: it is the part that directly interfaces with the PHY, being responsible for advertising, scanning, and creating or maintaining connections.
- Physical Layer: it contains the analog communications circuitry responsible for translating digital symbols over the air. It is the lower layer of the BLE protocol stack.

Although the Controller layers are usually implemented in hardware, some designs implement them as a combination of software and hardware. Which tasks and functions are implemented in hardware or software is part of the designer's work to decide.

It is possible to see how a processed BLE packet looks like at Figure 1-3, where only the Protocol Data Unit (PDU) corresponds to actual data to be transmitted, and the rest of the fields are part of the BLE protocol packet framing.

It is possible to see what a processed BLE packet looks like in Figure 1-3, where only the PDU corresponds to the actual data to be transmitted, and the rest of the fields are part of the BLE protocol packet framing.

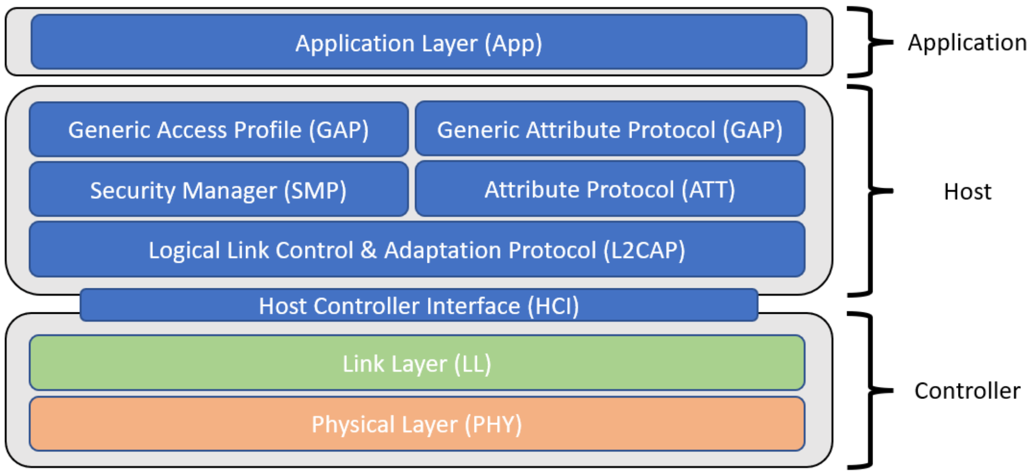

Application Layer (App)
Generic Access Profile (GAP)
Generic Attribute Protocol (GAP)
Security Manager (SMP)
Attribute Protocol (ATT)
Logical Link Control & Adaptation Protocol (L2CAP)
Host Controller Interface (HCI)
Link Layer (LL)
Physical Layer (PHY)
Application
Host
Controller

Figure 1-2: Bluetooth Low Energy Protocol Stack

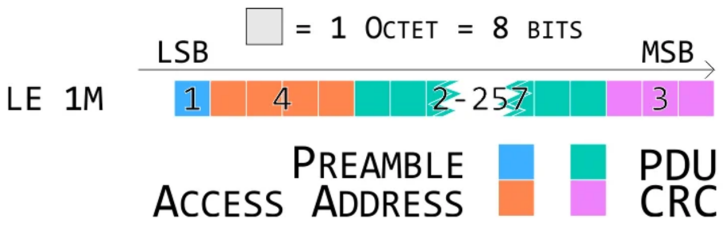


Figure 1-3: Packet format for the LE1M PHY [2]

1-3 Objective of this project

This thesis project aims to propose a new architecture design for a BLE Baseband Controller with low-power IoT applications specifically in mind to achieve a very low power consumption in specific use cases.

As mentioned before, the hardware part of a Baseband Controller is typically formed by the LL and the PHY. The PHY will be regarded as an external block, given that the RF and analog blocks are not the focus of this project.

All the improvements will then be focused on the Digital Logic part of the Baseband Controller, the LL. Out of all the parts that form the LL, the improvements will be focused on the bit-intensive functions, which are the bit processing and packet framing functions, and only those will be implemented and tested.

The expected outcomes of the project can be summarized as follows:

- A new architecture proposal of an SoC BLE module that fully uses its resources, combining the processor with accelerators to perform the tasks of a traditional BLE Baseband Controller design for IoT applications.
- The design, improvement, simulation, and testing of the bit-intensive tasks of the LL controller and its interfaces with the rest of the architecture.
- The conclusions about performing such a design that makes use of the SoC generator Rocket-Chip and the Constructing Hardware in a Scala Embedded Language (Chisel) language.

1-4 Outline of the project

The chapters of this thesis project are organized as follows:

- Chapter 2 introduces the required BLE protocol knowledge for the design presented in this thesis project.
- Chapter 3 summarises the findings in the literature and commercial designs for BLE baseband architectures, possible improvements, and the proposed architecture for the project.
- Chapter 4 explains the blocks performing the design's bit processing and packet framing.
- Chapter 5 explains in detail the rest of the blocks that compose the implemented part of the architecture and are in charge of the proposed extra functions, the error correction, and the bitstream flow through the design.
- Chapter 6 presents the result of the simulations and the Field Programmable Gate Array (FPGA) implementation.
- Chapter 7 presents the conclusion and future work for the project.

- Appendix A introduce some other topics such as Chipyard or the Rocket-Chip generator.
- Appendix B presents the introduced changes to simulate the SoC and the topics required to connect the designed baseband with the processor.
- Appendix C presents the used FPGA setup.
- Appendix D presents various simulations not included at the Simulations chapter.

Chapter 2

Bluetooth Low Energy Protocol

The Bluetooth Low Energy (BLE) protocol is detailed in the Bluetooth Core Specification [3]. It is an extensive document that defines all the layers of the Bluetooth protocol for both Classic Bluetooth: Basic Rate (BR) and Enhanced Data Rate (EDR); and BLE.

All the information in this section has been extracted from it [3] and summarized to only keep the strictly necessary parts related to the bit processing and packet framing functions that a baseband controller has to perform.

2-1 Bluetooth Low Energy Stack

The whole BLE protocol functions and tasks are divided over the different layers that compose the protocol stack. The BLE protocol stack can be seen in the previously presented Figure 1-2.

It is divided in:

- Application: implemented in software, it is the part that interacts directly with the user. It contains the user interface, application logic, and general application architecture.
- Host: implemented in software and composed of several layers that describe how the BLE devices communicate, their parameters, the security of the connection, and some other tasks.
- Controller: the physical part of the BLE architecture controls all the tasks related to managing the connection, processing the BLE packets, and sending/receiving them through the antennas.

2-2 Controller

A baseband controller is composed of both hardware and software. The hardware consists of two layers: the Link Layer (LL) and the Physical Layer (PHY). The PHY will not be

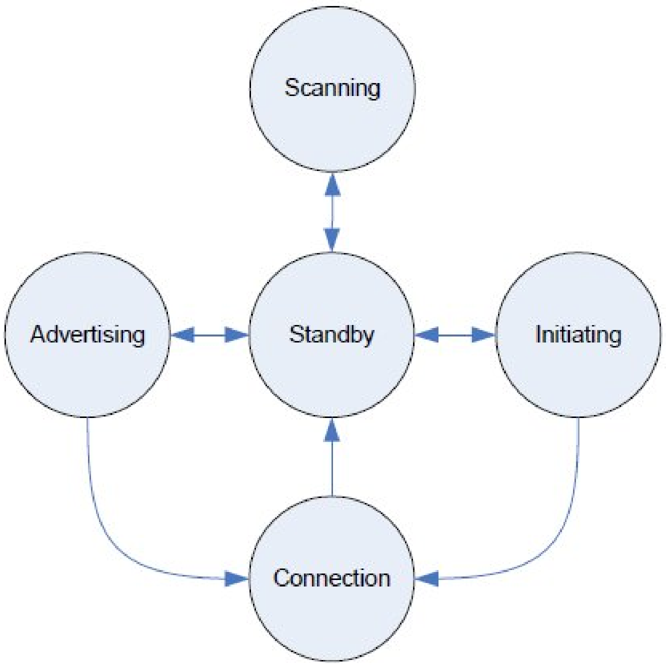


Figure 2-1: Link layer states in BLE [3].

explained or designed as Radio Frequency (RF) and analog circuits are not the focus of this project.

It becomes necessary then to explain what the LL is and its functions. The LL is the part of the controller that directly interfaces with the PHY. It is responsible for creating, controlling, and maintaining the BLE connections.

The LL controls how the connections are established using a state machine with five main states: Standby, Scanning, Advertising, Initiating, and Connection. The states and their transitions can be seen in Figure 2-1.

The role of a BLE device is defined by those states and the transitions that led to that state:

- Advertiser/Scanner(Initiator): these are the initial roles in establishing a direct connection. The advertiser will send packets in the advertisement channels looking for a scanner to initiate a connection, that will become the initiator.

- Slave/Master: these roles are achieved once a connection is established between a device advertising and a device scanning. The advertiser will become the slave, and the scanner will become the master.

- Broadcaster/Observer: the broadcaster is in the advertising state but is non-connectable; the observer is in the scanning state but not looking for a connection.

2-3 PHY modes

BLE supports different PHY modes. Those are the uncoded Low Energy 1 Msym/s (LE1M) and Low Energy 2 Msym/s (LE2M), and the coded Low Energy Coded (LEC). These different PHY modes also imply a series of differences in the LL and the PHY: such as different symbol rate speeds, additional steps in the bit processing (convolutional encoding and mapping), and different packet structures.

Concerning the symbol rate speed: both LE1M and LEC PHY modes use a 1 $Msym/s$ rate, while the LE2M PHY uses a 2 $Msym/s$ rate. Concerning the differences in the bit processing: both LE1M and LE2M PHY modes share the same exact processing with some minor differences in the packet structure (different Preamble), while the LEC PHY mode adds a convolutional encoder and a patter mapper steps to the bit processing in addition to a longer preamble.

This convolutional encoder, in addition to the pattern mapper, creates the difference in bit rate between the LE1M PHY (1 Mb/s) and the LE1M PHY (125 Kb/s or 500 Kb/s). That is because to improve the communications range and reliability of the connection, extra symbols are used to represent each data bit over the air. The number of symbols used for the coded PHY depends on the Coding Scheme (S) and can be either 2 or 8.

2-4 Packet Format

There are several differences in the packet structure of the different PHY modes, especially between the coded and uncoded ones. The packet structure for the different PHY modes is detailed below.

2-4-1 Uncoded PHYs modes: LE1M and LE2M

The packet structure of the uncoded PHY modes can be seen in Figure 2-2. This packet format is used for both advertising and data channels. Each field of that packet structure is explained below:

Preamble:

The Preamble is used at the receiver side to perform some adjustments such as frequency synchronization, symbol timing estimation, and Automatic Gain Control (AGC) training. It consists of a sequence of alternating ones and zeros of length 8 bits for LE1M and 16 bits for LE2M. The first bit of the transmitted Preamble (first to be transmitted) will be the same as the Least Significant Bit (LSB) (first to be transmitted) of the Access Address (AA).

Access Address (AA):

The AA is a four octet (32 bits) field that has to change with every Protocol Data Unit (PDU) and must follow a series of requirements, they are all summarized on page 2564 in the Core Specification [3]. Except for the AA for advertisement channel packets, which is 0x8E89BED6.

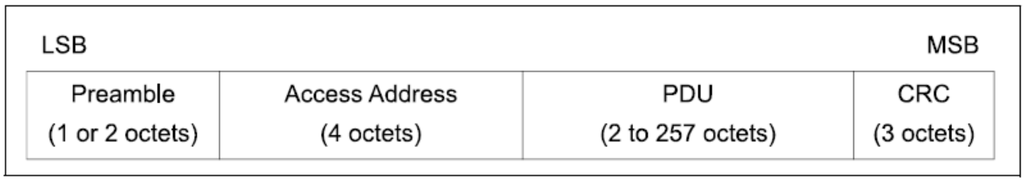


Figure 2-2: Link layer packet format for the LE uncoded PHYs [3].

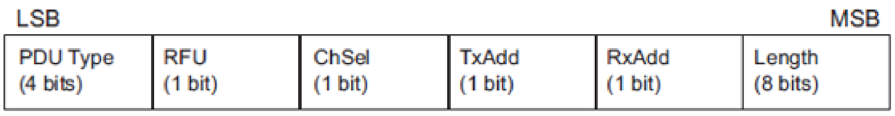


Figure 2-3: Advertising PDU Header structure [3].

Protocol Data Unit (PDU):

The PDU is formed by a Header of fixed length two octets, and the Payload, 0-37 octets when Payload Length Extension is disabled or 0-255 octets when Payload Length Extension is enabled. The PDU packet structure is defined for data and advertising channels [3].

The PDU Header contains some info divided into fields that can be seen in Figure 2-3. The field named Length is the most relevant for the baseband controller design, which indicates the PDU Payload's length, as it is not fixed and can vary from packet to packet, and corresponds with the last transmitted 8 bits of the PDU Header.

The PDU Payload is the field that contains the data to be transmitted with the packet, which can be either part of the BLE connection establishment determined packets or application/sensor data that one device is transmitting to the other device.

Cyclic Redundancy Check (CRC):

The Cyclic Redundancy Check (CRC) field has a length of 3 octets (24 bits), and it is calculated over the whole length of the PDU and later transmitted after it. It is used to detect any possible transmission bit error at the receiver side.

2-4-2 Coded PHY mode: LEC

The packet structure of the coded PHY (LEC) can be seen at Figure 2-4. As in the uncoded PHYs, this packet format is used for both advertising and data channels. Each packet consists of Preamble, Forward Error Correction (FEC) block 1, and FEC block 2. Each part is explained below.

The entirety of the packet is transmitted at $1Msym/s$, then every symbol takes 1 μs. Depending on the encoding scheme (S=2 or S=8), 2 or 8 symbols will represent every data bit, except when the section is uncoded where every symbol represents a single data bit.

Given the different coding schemes (S=2 or S=8) and the variable length of the PDU field, the table in Figure 2-5 has been added to indicate the possible lengths and durations of each section of the LEC packet.

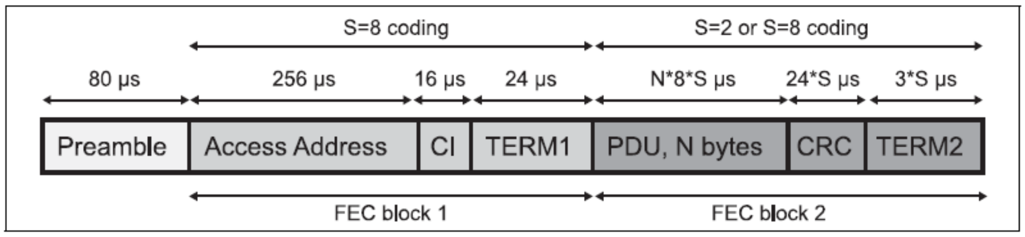


Figure 2-4: Link layer packet format for the LE coded PHY [3].

Preamble:

It has a length of 80 symbols and consists of the sequence "00111100" repeated ten times. This section must not be encoded.

FEC block 1:

It consists of three sections: the AA, the Coding Indicator (CI), and the TERM1. The entirety of the block is coded using the scheme S=8.

The AA field is the same as in the uncoded PHYs but coded.

The CI indicates which coding scheme is used for the FEC block 2.

The TERM1 corresponds with the termination sequence of the convolutional FEC encoder for the FEC block 1, and it has a length of 3 bits.

FEC block 2:

It consists of three sections: the PDU, the CRC, and the TERM2. The entirety of the block is coded using the same scheme, either S=8 or S=2. The election of one coding scheme or the other is indicated in the CI field of the previous block (FEC Block 1).

The PDU field is the same as in the uncoded PHYs but coded following the specified scheme in the CI field.

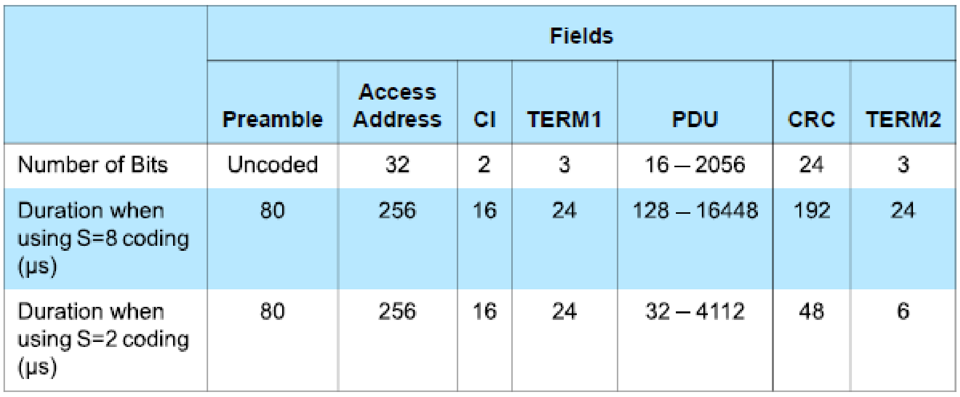
The CRC field is the same as in the uncoded PHYs but coded following the specified scheme in the CI field.

The TERM2 corresponds with the termination sequence of the convolutional FEC encoder for the FEC block 2, and it has a length of 3 bits.

2-5 Bit Stream Processing

As previously mentioned, the bit processing to be performed on the packets depends on the PHY mode. Then differences can be found between the uncoded PHY modes (LE1M, LE2M) and the coded PHY mode (LEC). Those differences can be seen in Figures 2-6 and 2-7..

It is also important to explain that each field of the BLE packet does not undergo the same processing. Each packet field (shown in Figures 2-2 and 2-4) and the processing that must



	Fields						
	Preamble	Access Address	CI	TERM1	PDU	CRC	TERM2
Number of Bits	Uncoded	32	2	3	16 – 2056	24	3
Duration when using S=8 coding (µs)	80	256	16	24	128 – 16448	192	24
Duration when using S=2 coding (µs)	80	256	16	24	32 – 4112	48	6

Figure 2-5: LE Coded PHY packet field sizes and durations [3].

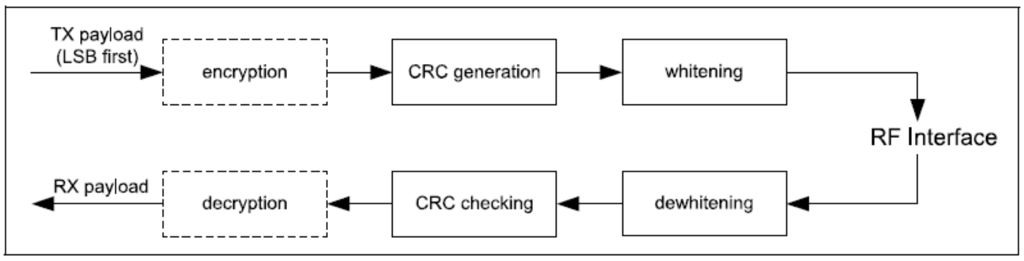


Figure 2-6: Payload bit stream processing for uncoded PHYs [3].

be done to them: Preamble, none; Access Address, whitening; PDU, whitening and CRC calculation; CRC, whitening; CI, whitening.

In addition, all the coded packets' fields except the Preamble must be encoded and pattern-mapped following the correspondent encoding scheme. Below, each processing step and its function are explained.

2-5-1 Cyclic Redundancy Check (CRC)

It is an error detection technique based on attaching some redundant bits at the end of each packet on the transmitter side. The attached bits have been calculated by performing a module 2 division of the data bits by a predetermined binary number or polynomial. Then the same operation is performed on the receiver side, and if the calculated bits do not match the ones attached at the end of the data block, an error was introduced during transmission

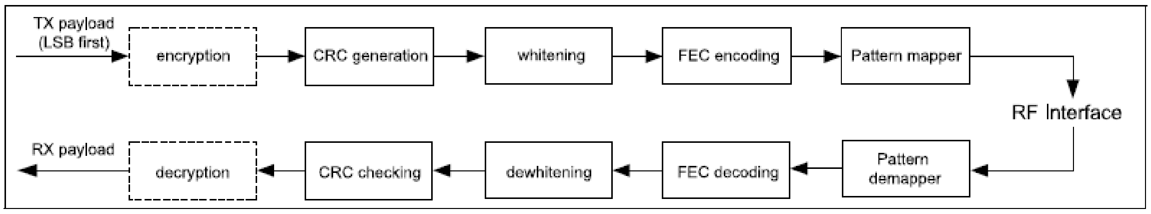


Figure 2-7: Bit stream processing for the coded PHY [3].

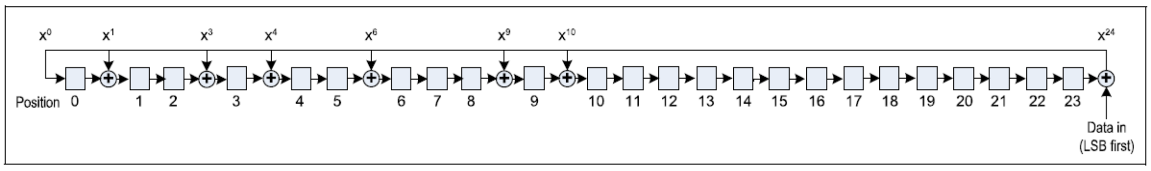

x0
x1
x3
x4
x6
x9
x10
x24
Position 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23
Data in (LSB first)

Figure 2-8: LFSR circuit for the CRC generation [3].

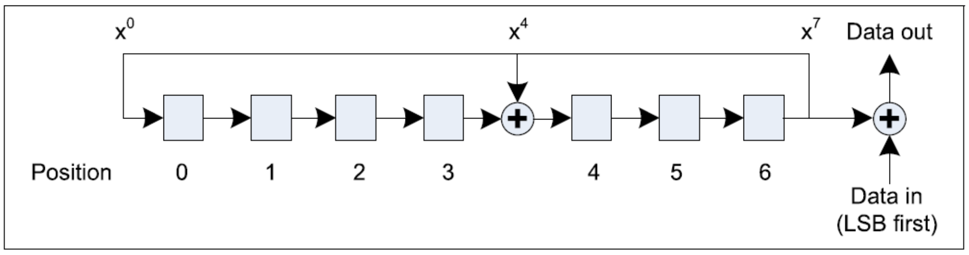


Figure 2-9: LFSR circuit for the data whitening [3].

and detected during the reception.

The polynomial used to calculate the CRC in BLE is: $x^{24} + x^{10} + x^9 + x^6 + x^4 + x^3 + x + 1$. The module 2 division by the previous polynomial is implemented using a Linear Feedback Shift Register (LFSR) that can be seen in Figure 2-8. An LFSR is a shift register in which its input is a linear function of its previous state and data input, and the initial state is often called seed (in this document is called initial value or state).

The initial state of the registers before starting the division is set to a specific value. This value will differ for the packets sent in the data and advertisement channels. For most of the advertisement packets, the initial value of the CRC registers is set to $0x555555$. Alternatively, for the rest of the advertisement and all the data PDUs, the value is sent to the specific one set/received during the exchange of packets [3].

First, the AA is checked to be correct; if it is not, the packet is discarded. After that, if the CRC is incorrect, the packet is rejected. If a packet is not rejected after the AA check and the CRC, it is considered valid.

2-5-2 Whitening

In order to avoid long consecutive sequences of 0s or 1s in the transmitted packet (as they can introduce a DC offset in the transmitted signal [16]), whitening is performed.

In order to achieve this, the combination of the unwhitened bit sequence and an LFSR are added. The LFSR polynomial is $x^7 + x^4 + 1$, and the whitening circuit can be seen in Figure 2-9.

The initial state of the used LFSR is determined in the following way: position 0 is set to "1", and positions 1 to 6 are set to the channel index of the used channel when receiving/transmitting [3]. For example, the initial state when transmitting/receiving at channel 23 (0x17) would be "1010111" (ordered from left to right from position 0 to 6).

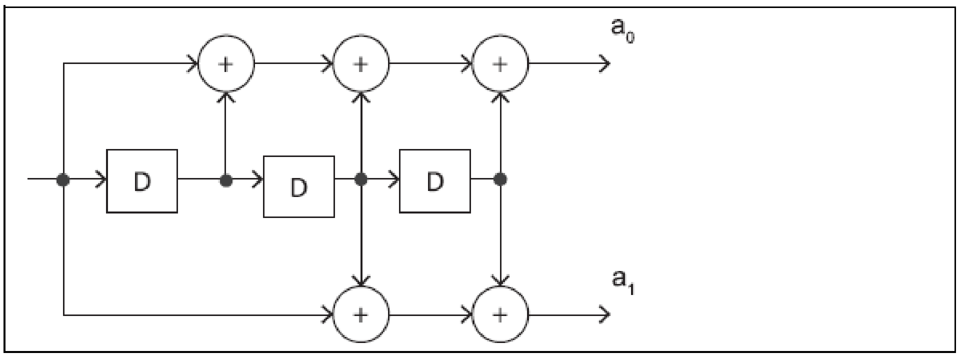
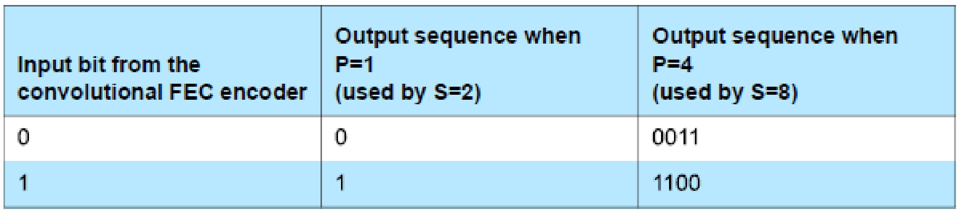


Figure 2-10: Convolutional Forward Error Correction (FEC) Encoder [3].



Input bit from the convolutional FEC encoder	Output sequence when P=1 (used by S=2)	Output sequence when P=4 (used by S=8)
0	0	0011
1	1	1100

Figure 2-11: Patter Mapper Inputs/Outputs [3].

2-5-3 Forward Error Correction (FEC) encoder

This bit processing part only applies to the coded PHY (LEC) [3]. In order to be able to correct errors on the receiver side, two additional steps are included in the bit processing on the transmitter side: a coding step using a convolutional FEC encoder and a spread of the coded packet using a pattern mapper [3].

The convolutional FEC encoder uses a non-systematic, non-recursive rate 1/2 code with a Constraint Length (K) of 4, and the following two generator polynomials: $G_0(x) = 1 + x + x^2 + x^3$, $G_1(x) = 1 + x^2 + x^3$. The circuit implementation of the FEC encoder can be seen in Figure 2-10.

The initial state of the registers in the encoder is always set to 0s. A sequence of K $-$ 1 input 0s will always bring the registers back to the initial null state, and this sequence is called the termination sequence [3].

In order to decode the bit sequence on the receiver side, any convolutional decoder can be used, taking into account the generator polynomials. The most referenced one in literature, in this case, is the Viterbi Decoder [17] [18] [19].

2-5-4 Pattern Mapper

The pattern mapper converts each output bit from the convolutional FEC encoder into P symbols. The value P depends on the coding scheme that we are using (S) [3]. The conversion depending on P can be seen in Figure 2-11.

2-6 Bit Ordering

It is also relevant to mention how the bits are ordered inside the packet and sent through the air.

The bit ordering when defining fields of the packet or PDU follows the Little Endian format, where the LSB is the first to be sent through the air, corresponds to position 0 (LSB), and is always represented on the left side.

Chapter 3

Baseband Design

This chapter presents existing literature designs and architectures for Bluetooth Low Energy (BLE) baseband controllers and related devices, in addition to existing commercial modules. From the literature review, insights that will be useful when coming up with a new architecture and possible improvements over the existing state of the art are extracted. After that, a new BLE baseband architecture is proposed and explained.

3-1 Existing Literature Designs

A literature review was performed in order to find relevant designs or features as a background to start the design presented in this project. The most relevant publication related to the design and architecture of a baseband controller of a BLE module is: "Architecture and Design of a Bluetooth Low Energy Controller" [4]. However, other publications that expose some relevant ideas were found [5] [6].

3-1-1 Literature: Architecture and Design of a Bluetooth Low Energy Controller

In [4], the full implementation of the Controller part of the BLE stack is presented. The BLE stack can be seen in Figure 1-2.

In the paper's implementation, the BLE controller is part of a System on Chip (SoC) that has a Central Processing Unit (CPU). That adds some flexibility to the design process, as part of the Link Layer (LL) functions can be implemented on software. However, it is relevant to add that the BLE protocol [3] does not strictly define how the layers that compose the controller (LL and Physical Layer (PHY))) are divided between hardware and software.

In the paper's design [4], the implementation of the LL is divided between hardware and software, while they do not go into detail about how to implement the PHY as they do not focus on the Radio Frequency (RF) part.

The function division between software and hardware chosen in the paper's design [4]:

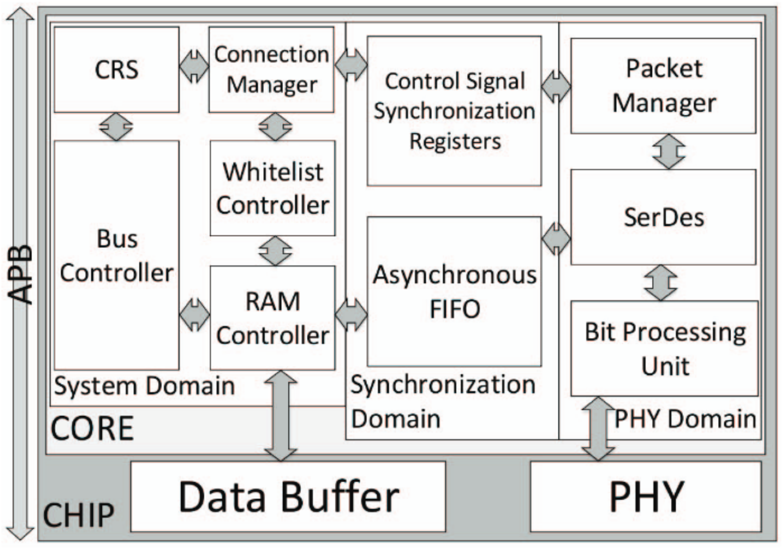


Figure 3-1: Link Layer block diagram [4].

- Software: translating Host Controller Interface (HCI) arguments to the controller of the hardware part, argument validation, reporting controller events through the HCI, and managing the controller's internal timers.
- Hardware: bitstream processing, Protocol Data Unit (PDU) framing and analysis, managing BLE LL states, controlling the protocol timing, executing Host commands and reporting results, calculating frequency hopping parameters, and detecting Bluetooth devices.

The LL block diagram of their design can be seen in Figure 3-1. In the same block diagram, it is also possible to see that they have divided the design into two different time domains, one fixed by the CPU and the other shared with the PHY layer and fixed by its timing requirements. The communication between the two domains is conducted via an asynchronous First In First Out (FIFO) (synchronization Domain in the block diagram).

3-1-2 Other relevant publications

As previously mentioned, some other published papers also present some relevant ideas that can be applied to the design of a BLE controller [5] [6].

Literature: "Fully synthesizable Bluetooth baseband module for a SoC"

The published paper [5], although from 2003 and being a Classic Bluetooth baseband design, presents some architectural design ideas and decisions that are also applicable and interesting for a present BLE baseband design.

The main architecture presented in the paper [5] can be seen in Figure 3-2. From it, an explanation of their architectural design decisions can start. Following the ideas presented in

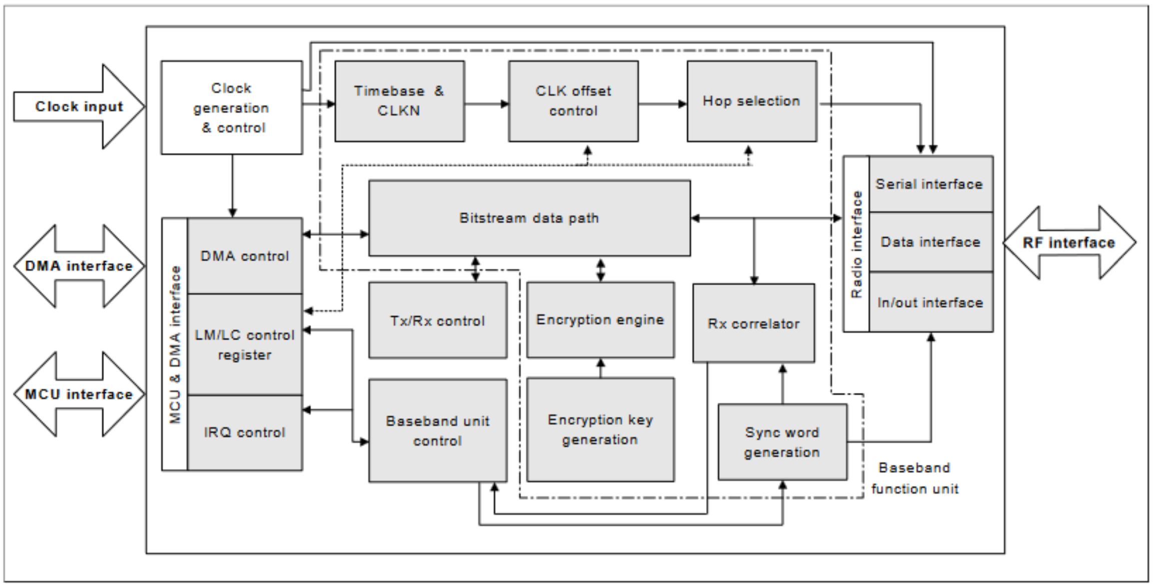


Figure 3-2: Top level architecture block diagram from [5].

the paper [4], part of the LL functions were moved into software as the design is supposed to be a soft Intelectual Property (IP) to be added in an SoC implementation which will count with a processor or a microcontroller.

The LL functions that have been implemented as software are the LL management tasks, and these tasks are presented below.

The first software implemented function is translating commands and data between the LL and the Host part of the Bluetooth stack (upper layers). The equivalent to this function in the more modern BLE protocol would be the HCI, which can be seen in the protocol stack in Figure 1-2. Second, controlling the link establishment, destruction, and configuration of connections between BLE devices.

The rest of the LL functions are implemented on hardware and consist of maintaining a link once established, the bit intensive, and the time-critical tasks.

Going into details about the functions they have implemented in hardware, the designed blocks in Figure 3-2 can be grouped into: channel selection and timer interrupts (the top 3 blocks inside the baseband function unit), bitstream processing (bitstream data path, rx correlator, sync word generation), encryption (encryption engine and encryption key generation) and some control blocks outside the function unit.

If the same approach as the one presented in the paper [5] were to be followed for BLE, the resultant design would be composed only of the timer interrupts, the equivalent of the Hop selection block, and the bit stream processing. The reason is that encryption is not standard in BLE, as it is only used in some cases when it is activated on a connection between two devices (most BLE packets are not encrypted).

It is also important to look at their implementation of the bit processing, which can be seen in Figure 3-3. Given that no comments are made in the paper about any improvement or

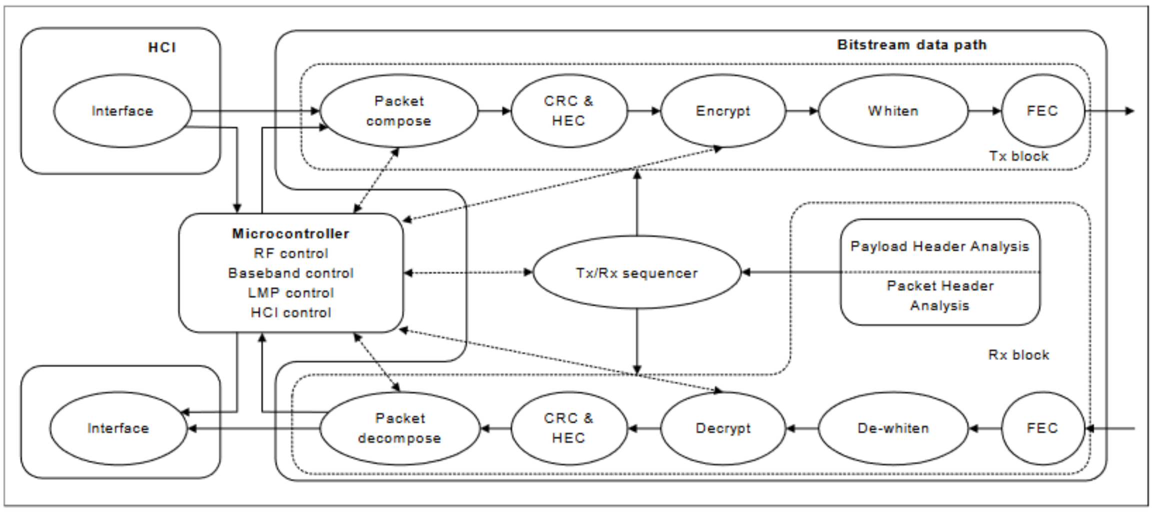


Figure 3-3: Bit stream flow in the bitstream data path at [5].

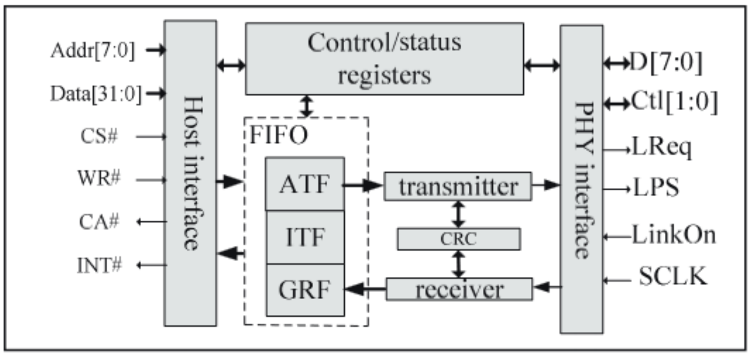


Figure 3-4: Link Layer controller architecture from [6].

relevant design decision in this part, a hardware block per function in the bit stream processing diagram depicted in Figure 3-3 is assumed.

Literature: "Design of link layer controller for high speed serial bus"

The publication [6] presents the design of a LL controller for a high-speed serial bus, but still, some ideas can be extracted and applied to the LL controller for the BLE baseband.

The designed architecture in [6] can be seen in Figure 3-4. On it, it is possible to see some interesting design decisions such as the use of FIFOs for the exchange of data between the device and the controller, the use of registers capable of triggering interruptions in the device for the control/status of the controller, the share of resource blocks (the Cyclic Redundancy Check (CRC) calculation block), and the use of an interface block for the connection between the controller and the PHY layer under it.

3-1-3 Conclusions extracted from literature designs

Some useful ideas can be extracted then from the analysis of the above-presented publications [4] [5] [6]:

- Use of asynchronous FIFOs for the data exchange between the LL controller and the microcontroller/processor: This allows an easy interface to send/receive data to/from the LL controller. It also allows using two different clock domains, one for the processor and another for the controller, which can be synchronized with the PHY layer (RF part).
- Use interrupt triggering registers to communicate the control and status data of the LL controller.
- Use of an interface adaptor between the LL controller and the PHY layer. This allows us to change the interface to work with different PHY implementations easily.
- Use the SoC capabilities, the microcontroller/processor, as the non-intensive operations do not require specific hardware and can be moved into software.
- Some other hardware blocks can be interesting, such as an AES-128 encryption block, in case it was interesting for the application.
- Specific timer implementations can help with the timing accuracy of the protocol.

3-2 Existing Commercial Designs

In order to have a reference point for architecture designs of BLE modules targeted to a similar application as the one in this project, the available commercial modules from Texas Instruments (TI), Nordic Semiconductors (NS), and Microchip (M) were checked. All three manufacturers had several available modules ranging from full implementations of the BLE module with big memories and diverse functionalities to bare minimum implementations for small Internet of Things (IoT) applications such as beacons.

Given the chosen application, autonomous sensors for IoT, those small modules were the ones that would suit the best. Table 3-1 summarizing their characteristics are presented below.

Table 3-1: Summary of commercial BLE modules for small applications

Manufacturer	Model	BLE PHYs	BR/EDR	Processor	RAM/Cache
TI [20]	CC2564C	5.1: LE1M LEC	No/No	ARM Cortex M3 (32-bit)	28K/8K
NS [21]	nRF52805	5.0: LE1M LE2M	Yes/Yes	ARM Cortex M4 (32-bit)	24K/8K
M [22]	IS1870	5.0: LE1M	No	MCU	-/-

Given the low power and area requirement of the application and the lack of need for a higher bandwidth or range BLE PHY, the designed system should keep the bare necessary minimum for making possible BLE connections. That means supporting only the basic data rate Low Energy 1 Msym/s (LE1M) and having a small embedded applications core (32-bit) was a common choice among them. It is also possible to see how the commercial implementations have also maintained reduced RAM and Cache memories in the SoC. Therefore the implementation of this project will count with a reduced size Cache memory.

3-3 Possible Improvements/Literature Research

Thanks to the previous literature and commercial product research about BLE Baseband Controllers and modules, an initial architecture design can be developed, but it is still necessary to look for more improvements.

In this section, all possible improvements at the block level will be explored, then changes and improvements at the architecture level will be considered.

3-3-1 LE1M bitstream processing improvements

The first part to be checked for improvements is the bitstream processing chain, the bitstream flow diagram of it can be seen in Figure 2-6. This part of the LL is the most relevant in terms of power consumption if the PHY is not taken into consideration, as it performs bit-intensive tasks during both transmission and reception of packets. All the blocks performing bit-intensive calculation tasks in the bitstream processing are already as simple and small as possible; those blocks are the CRC and the Whitening block.

Given that the BLE Baseband will never transmit and receive simultaneously, sharing common blocks between both transmitter and receiver sides is possible. Following this, the two common blocks between transmitter and receiver logic in the bitstream processing chain will be shared. The two common blocks between sides are the ones used for performing the CRC and Whitening/Dewhitening.

The proposed bitstream processing block will be introduced later and can be seen in Figure 4-1.

3-3-2 Improving the reliability of the connection

A way of further reducing power consumption when connections are already established is improving their reliability and reducing the number of retransmitted packets due to transmission errors.

The reason behind this is the way BLE connections work (when a direct connection is established, not the case for broadcasting); when a packet is correctly received, the receiver transmits an acknowledgment packet. Conversely, when an erroneous packet is received, and no acknowledgment packet is sent, the transmitter will send the same packet again.

However, in order to improve reliability, BLE already counts with the Low Energy Coded (LEC) PHY mode. The problem with that PHY mode is that it introduces additional bit processing steps and duplicates or even quadruplicates the transmission time for the same raw data length. In addition, it is worth noting that the time period when the antenna is transmitting or receiving is when more instantaneous power is consumed [23]. This mode is used only when an improved communications range is required, but not for reduced power consumption.

CRC error correction

In order to improve the connection reliability and avoid possible retransmissions, CRC error correction is presented.

It lacks some of the presented problems that the Forward Error Correction (FEC) has in the LEC. It does not add any transmission/reception antenna time, uses the already calculated and sent CRC, and can correct multiple erroneous bits on each packet. Although it requires an additional processing step but only on the receiver side

Several approaches presented in published papers were considered, using look-up tables for error patterns and table-free [24] [7].

Connection parameters

Parameters mentioned before such as the advertising time and period, the channel used for the transmission, the transmission power, and the antenna's gain have a significant impact on power consumption. Although the selection of such parameters will not be covered in this project, an interesting block to consider adding in the future would be a whitelisting algorithm block.

Channel whitelisting is another method to improve connection reliability and avoid possible retransmissions, and it consists of avoiding the use of channels where interference has been previously found. Several implementable whitelisting algorithms can be found in the literature [25] [26].

3-3-3 Reducing interactions between Processor and Baseband

Given the presented design where the BLE LL is implemented as a combination of software and hardware on an SoC, a way of reducing its power consumption consists of reducing the time when both parts of the design are simultaneously active. That means sending the processor to sleep mode as soon as possible when the designed baseband is processing a data packet.

The data connection between the processor and the hardware part of the LL controller design takes place via FIFOs, which means that the processor will send the whole data of the packet and the configuration required for it in a series of consecutive writes. Therefore, power consumption can be reduced by reducing the number of FIFO writes required for sending a packet.

In the chosen application case for the project, the same data is likely to be repeated in several transmitted packets, for example, a sensor sending a measurement to several receivers or broadcasting it. If the repeated data of the packet is stored at convenience in the hardware part of the LL controller, it would be possible to send it with very few commands/writes (entries in the FIFO) from the processor. That would effectively reduce the active time of the processor and the total memory reads and writes.

Therefore a series of functions will be added to the hardware part to avoid repetitive data communication between the processor and the LL controller hardware.

3-4 CRC error correction algorithms

Several techniques to perform CRC error correction are present in the literature. The existing techniques can be classified into two groups: table-based approaches and iterative error correction algorithms.

The look-up table technique working principle is to store all the possible error vectors and their resultant CRC syndrome. Then, when an error is detected, the CRC error syndrome is used to find the error vector that generated it in the table [27]. Unfortunately, this technique is not feasible for BLE, given that the packet's length is not fixed. In addition, the table size increases exponentially with the number of errors taken into consideration, which is why they are usually implemented for single bit error correction.

The second group, iterative algorithms, are generally more flexible in terms of the length of the corrected packet and the number of bit errors to be corrected [24]. However, they also present a counterpoint, their complexity to be implemented on hardware and latency.

The chosen iterative algorithm [7] lacks one of the previously mentioned negative points, which is the high complexity of the algorithm. The CRC error correction algorithm is based on performing the opposite operations used to calculate the CRC syndrome, which can be reduced to XOR operations and shifts, as opposed to the rest of the found iterative algorithms.

3-4-1 Selected CRC error correction algorithm

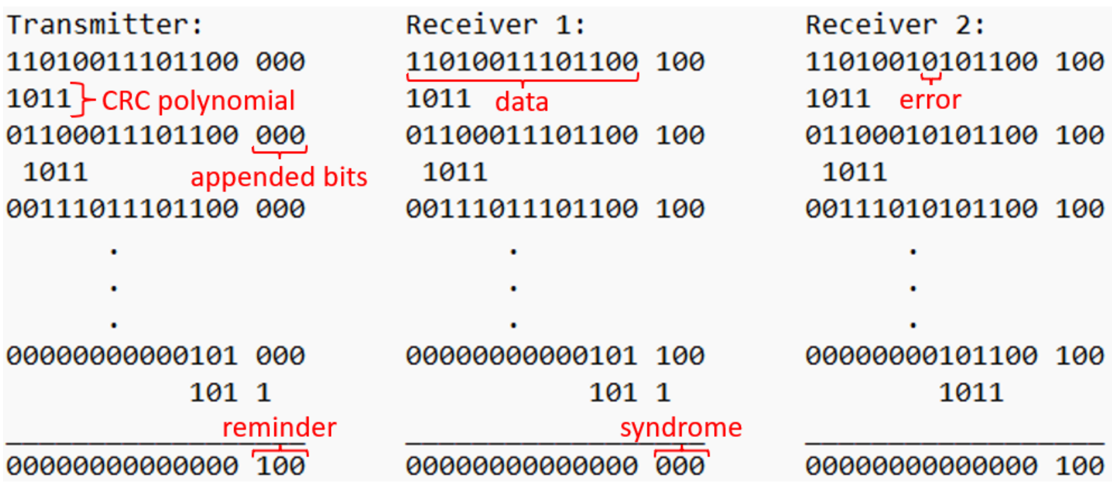
The selected algorithm is based on performing the opposite process to the CRC syndrome calculation to find candidate error patterns that result in that syndrome.

It is first necessary to explain the CRC syndrome: it is the reminder of the module 2 division calculation of the CRC at the receiver side. The syndrome value will equal zero when no error presence is detected and will differ from it whenever an error is present.

With this algorithm, more than one error pattern may generate the same syndrome, which is why it can not always correct directly all the packets. However, it is possible to directly correct all packets with a single error pattern candidate, which is the approach that will be exploited in the designed block for this project.

3-4-2 CRC calculation as a module 2 division

The working principle of the CRC syndrome calculation can be seen in Figure 3-5, it consists of the module 2 division of the data to be protected with a selected polynomial and a second calculation on the receiver side with the remainder of the transmitter side appended.



```
Transmitter:              Receiver 1:               Receiver 2:
11010011101100 000        11010011101100 100        11010010101100 100
1011                      1011                      1011
01100011101100 000        01100011101100 100        01100010101100 100
 1011                      1011                      1011
00111011101100 000        00111011101100 100        00111010101100 100
      .                         .                         .
      .                         .                         .
      .                         .                         .
00000000000101 000        00000000000101 100        00000000101100 100
           101 1                     101 1                 1011
__________________        __________________        __________________
00000000000000 100        00000000000000 000        00000000000000 100
```

Figure 3-5: CRC calculation and check, transmitter and receiver side with/without errors respectively.

The module 2 division consists of XORing the selected polynomial with the data aligning the polynomial with the first bit different from zero starting from the left. After XORing, the polynomial is again aligned with the first MSB non-zero bit, and it's XORed again. The previous process is repeated until the first non-zero bit from the left is already in a position that does not allow to perform the XOR operation with the polynomial.

In order to understand the working principle of the correction algorithm, it is necessary to explain a few concepts. The remainder is the leftover after performing the module 2 division over the whole packet plus the appended bits. The syndrome is the reminder of the calculated module 2 division performed at the receiver. Both the appended bits and the syndrome will be of length $K - 1$, where K is the length of the polynomial, which corresponds with its degree plus one.

In the case of the BLE, the syndrome and appended bits have a length of 24 bits, and the CRC polynomial of 25 bits, being its value 0b1000000000000011001011011.

3-4-3 Error pattern search algorithm

As mentioned before, the process consists of the inverse operation used for the CRC calculation, using the module 2 division to find candidate error vectors [7]. The initial state for the error correction algorithm when an error is detected is the same as the result of the receiver's module 2 division.

The error pattern search algorithm for single errors can be seen in Figure 3-6 and works as follows:

1. The error vector is built (t_0). It is formed by a sequence of zeros and the appended non-zero syndrome at the end. The sequence of zeros length is the same as the packet.
2. The CRC polynomial is aligned with the last non-zero bit of the constructed error vector, and then they are XORed (t_0).

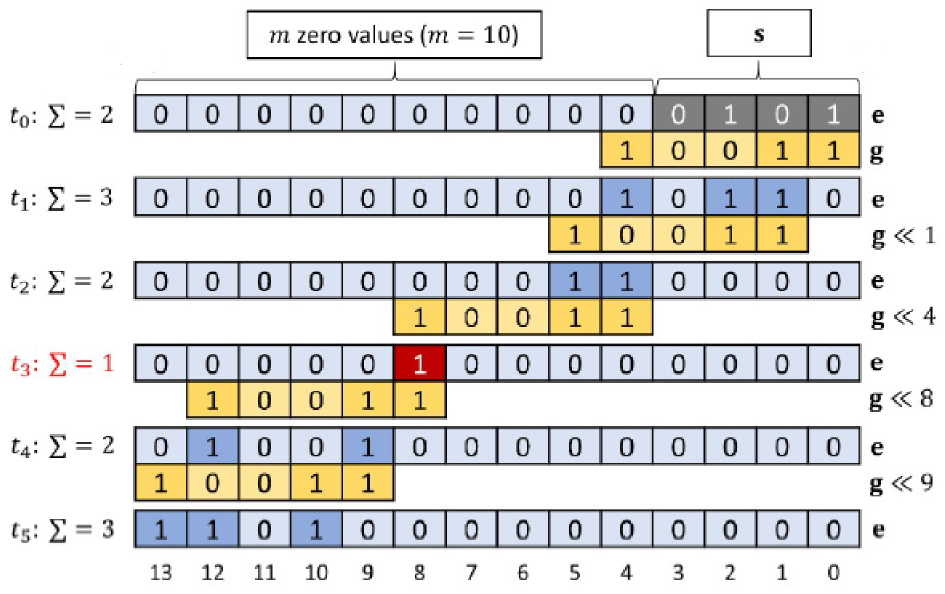


Figure 3-6: Single error pattern search [7].

3. If the number of non-zero bits equals 0, then the algorithm stops. If the number of non-zero bits equals 1, the error vector is saved as a candidate (t_3). Finally, if the number of non-zero bits is bigger or equal than 1 then the previous step is repeated, given that the end of the error vector has not been reached (t_1, t_2, t_4).

4. The list of saved error vectors is evaluated when the algorithm stops or reaches the final (t_5). If only one candidate is in the saved list of error vectors, the data can be directly corrected by flipping the data position different from one in the error vector.

   In the example in Figure 3-6, the 8^{th} position would be the erroneous one to be flipped in the data packet.

In order to extend the algorithm to find error pattern candidates for more than a single error, some changes have to be introduced. The changes can be seen in Figure 3-7 and consist on:

1. The Least Significant Bit (LSB) with value one is fixed, and the polynomial is shifted to be aligned with the second LSB with value one (t_0). Then the single bit error pattern search is performed. In this case, when the amount of non-zero bits equals 2 (considering the fixed position), the error vector is saved in the list (t_1).

2. Once the previous step finishes, the state of the error vector when a position was fixed is restored (t_2). Then, the polynomial is shifted to that position and XORed.

3. Then the first and second steps are repeated (between t_3 and t_{17}) until the end of the error vector is reached (t_{18})

4. The packet can be directly corrected if there is a single candidate error pattern. However, the algorithm fails to correct the error if there is no candidate for the error pattern.

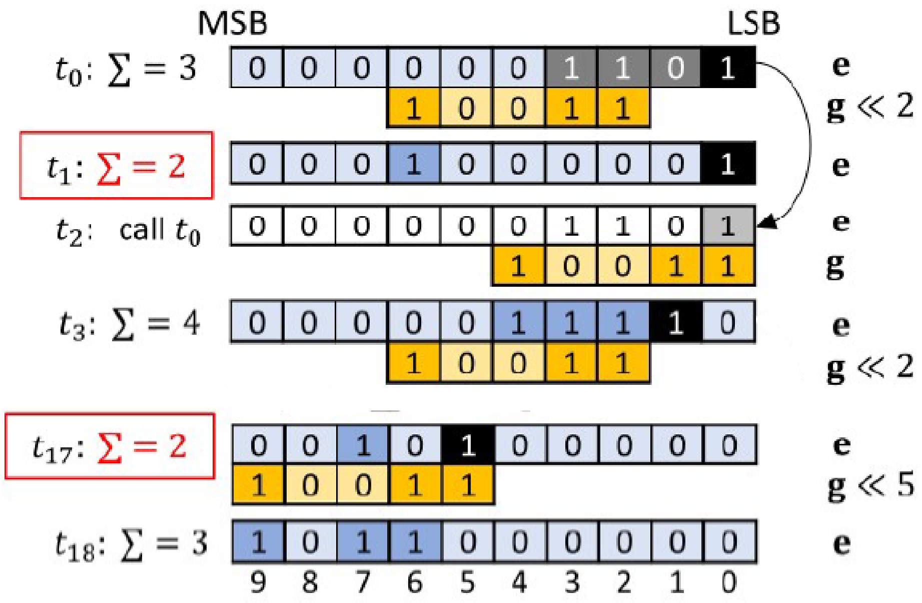


Figure 3-7: Multiple error pattern search [7].

The introduced changes for double error search could be explained as supposing that a position contains an error and running the algorithm for single errors all over the rest of the data. That can also be extended for more than double errors by fixing more than one position and covering the rest of the data with the single error search algorithm.

3-5 Proposed Architecture Design

The proposed architecture for the combined hardware-software BLE LL design can be seen in Figure 3-8.

3-5-1 Explanation

The proposed architecture combines the ideas extracted from the literature and the improvements mentioned before.

In order to take full advantage of the processor, the non-intensive LL tasks have been moved to software. As a result, the distribution of LL between hardware and software is as follows:

- Software: connection state and setup, data framing, device address generation, and HCI interface.
- Hardware: bitstream processing, PHY parameters calculation, protocol timing, and interfacing with the PHY.

In order to implement this hardware-software design, the communications between the hardware part and the processor need to be carefully considered. In order to exchange long

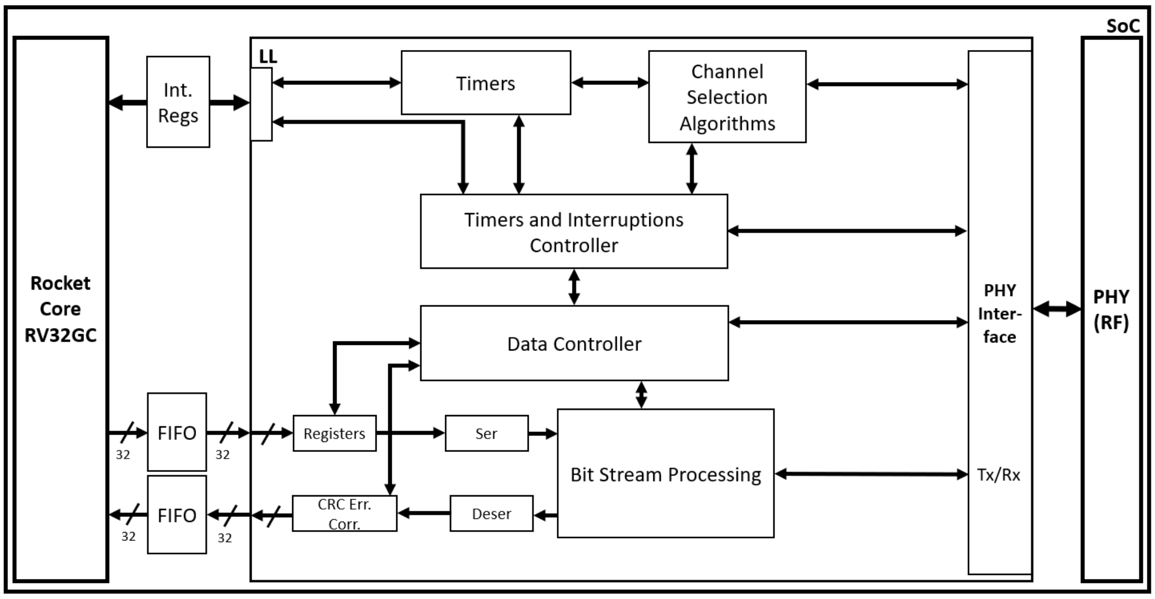


Figure 3-8: Proposed LL architecture design.

sequences of data for sending and receiving packets, a couple of asynchronous FIFOs will be used. In order to control the hardware block and wake up the processor when a packet is received, or a timer is triggered, a series of interruption triggering and control registers will be used.

In order to isolate the design of the LL from the analog and RF domain, a so-called "PHY interface" will be used, this is only a block to adapt the format of the signal to the vendor-specific or the later designed PHY.

The design in this thesis is focused only on the hardware and architecture part of this LL design and, more precisely, on improving the architecture and the bit stream processing part of it.

That is why given that the design can then be split into two parts, the bit processing and packet framing, and the one in charge of the timing, control signals, and control of the PHY. Only the first one will be designed in this project.

3-5-2 Connections in the SoC

As mentioned before, the connections between the designed hardware block and the processor must be carefully considered to obtain the performance and meet the requirements fixed by the Bluetooth specification.

- Interruption registers: in order to wake up the processor when a packet is received or when a timer is triggered.
- Control and flag registers: used to control the LL hardware block and report the state of the baseband to the processor
- Data FIFOs: used to send and receive data from the LL hardware block.

3-5-3 Timers

Timers are required to ensure the baseband controller's timing accuracy and the BLE protocol's timing correctness. The timers should be able to be configured to trigger interruptions, in addition, to using them for periodically broadcasting saved data packets.

3-5-4 PHY Parameters calculation

It is part of the LL set of tasks: channel selection, whitelisting/blacklisting channels, and transmission power selection.

For the selection of the channel Frequency Hoping Algorithm (FHA) are used, two of these algorithms exist and are used (#1 and #2), one introduced in BLE 4.0 (#1) and the other introduced in BLE 5.0 (#2). Both must be available if a BLE 5.0 capable device connects with a BLE 4.0 device.

Also, a list of the available channels to be used must be present and modifiable either by the processor or by a channel whitelisting algorithm. The FHA uses this list to select among the whitelisted channels (available).

3-5-5 Timers and Interruptions Controller

This design block will interface between the bit stream processing controller and the control and interruption registers, in addition to managing the timers and PHY parameters calculation blocks.

This block will be in charge of translating the signals received through the control register into commands for the bitstream controller, reporting any necessary flag to the processor such as a discarded packet, triggering timing and reception interruptions, and informing about the current state of the baseband.

3-5-6 Data path Controller

This design block controls the bit flow between the data FIFOs and the rest of the blocks, in addition to controlling all the bit stream processing steps of the data packets. This block is controlled by a series of control bits received from the "Timers and interruptions Controller".

3-5-7 PHY Interface

This block acts as the interface between the LL and the PHY layer. It translates the signals coming from and to the LL into the specific format required by the PHY layer below. That includes all the signals required to set the channel, the Received Signal Strength Indicator (RSSI), to activate the PHY, and both data in and out of it.

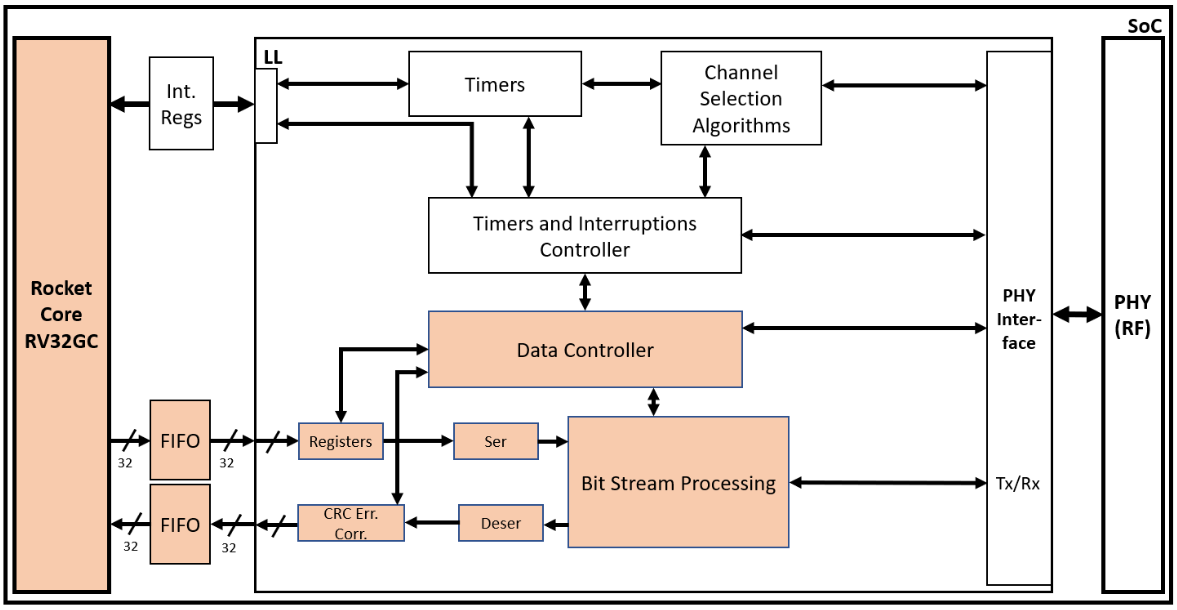


Figure 3-9: Implemented part of the proposed baseband design.

3-6 Design narrowing in the project

Given that the improvement ideas are focused on the bit stream processing path, the size of the design of the LL, and the high number of specifications/requirements it has to meet, only the bit stream processing part of the LL will be designed. The designed parts belonging to the data path can be seen in Figure 3-9 in orange color.

Chapter 4

Bitstream Processing Block

This chapter explains in detail the design of the block in charge of performing the processing and packet framing for transmission and reception of Bluetooth Low Energy (BLE) packets. The block and its place inside the proposed architecture can be seen in Figure 3-8

The proposed architecture for the bitstream processing block sharing the Cyclic Redundancy Check (CRC) and Whitening blocks can be seen in Figure 4-1. The bit processing flow of the BLE Low Energy 1 Msym/s (LE1M) mode can be seen in Figure 2-6.

4-1 Shared Blocks

The blocks under this section are shared and used in both transmission and reception of BLE packets. Therefore, given that transmission and reception do not coincide in time, it is possible to share them. An explanation of the designed blocks can be found below.

4-1-1 CRC

The basic operation of the CRC block is defined in the BLE protocol specification [3] and can be seen in Figure 2-8.

The designed block followed that specification and can be seen in Figure 4-2. The "init" signal acts as an enable; if it is kept at a high level, the CRC block will not change its state and will be fixed to the "init_state" or initialization value signal. The blue wires correspond to Decoupled interfaces, which means that the input/output consists of 3 signals: ready, valid, and data.

4-1-2 Whitening

The basic operation of the Whitening block is defined in the BLE protocol specification [3] and can be seen in Figure 2-9.

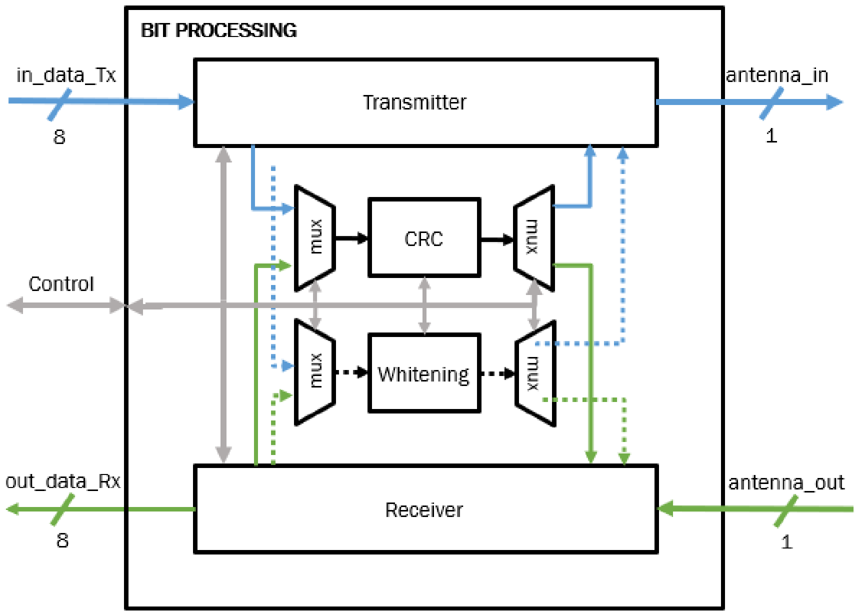


Figure 4-1: Bit processing block architecture.

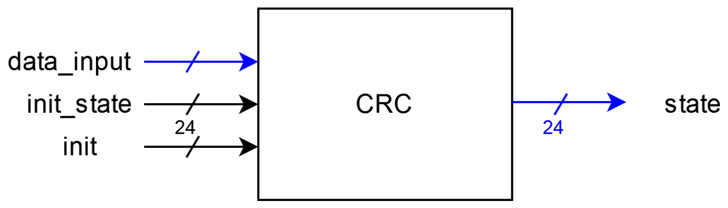


Figure 4-2: Designed CRC block.

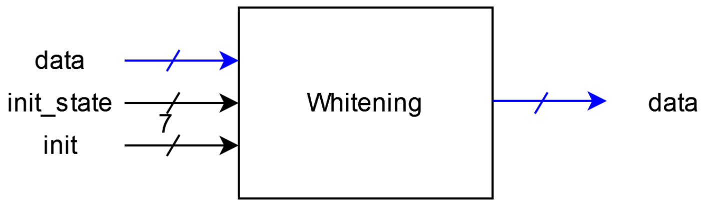


Figure 4-3: Designed Whitening block.

The designed block followed that specification and can be seen in Figure 4-3. As in the designed CRC block, the "init" signal works as an enable, keeping the state of the registers fixed to the value in the "init_state" signal. Both "data" signals are Decoupled interfaces with ready and valid signals, marked in blue.

4-2 Transmitter bitstream processing block

In order to explain in detail the designed block, it is necessary to explain its specific function step by step, the specific inputs and outputs, and then any other relevant design-specific decision.

4-2-1 Functionality

The transmission of a LE1M packet consists of the following steps, which can also be seen in Figure 4-4:

1. The preamble sequence is transmitted without further processing. The transmitted preamble depends on the specific Access Address (AA) used for transmitting the packet, as explained in section 2-4-1.

   During this first step, the Whitening module has to be initialized to a specific state that depends on the channel selected for the packet transmission.

2. The selected AA is whitened and transmitted in Least Significant Bit (LSB) to Most Significant Bit (MSB) order. This AA is either fixed during communication or equals 0x8E89BED6, as explained in section 2-4-1.

   Until this point, the CRC module had to be disabled and initialized to a given value.

3. The CRC module is enabled to calculate the CRC value over the whole Protocol Data Unit (PDU). The PDU starts to be serialized and is used for the CRC calculation simultaneously as it is whitened.

4. The PDU transmission is finished, and it is time to perform the whitening of the calculated CRC before transmitting it from LSB to MSB order.

Several data fields are required in addition to the PDU for the block to perform all of these steps. The required fields are the initial CRC and Whitening states and the chosen AA. How those values are decided does not depend on the bitstream processing part of the Link Layer (LL) and can be regarded as inputs to the specific blocks.

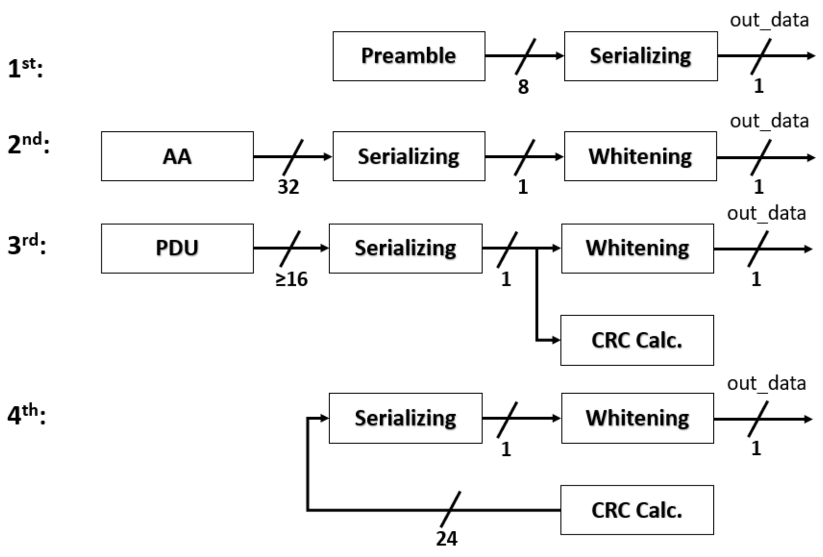


Figure 4-4: Processing steps in the transmission flow of the bitstream.

4-2-2 Design Decisions

Some design decisions that have been made taking some considerations into account are listed below:

- The control of the functions performed on each part of the packet is managed with a Finite States Machine (FSM) with the following states: idle, preamble, Access Address, Header, Payload, and CRC. The changes between states are triggered via counters of transmitted octets.
- The bit width of the data input: all the data fields on a BLE packet are organized in octets. Therefore the best way to receive/transmit the raw data is using an 8-bit width.
- The transmission of the preamble will only start when the enable signal is on high level, and the first byte of the PDU is received. That allows for initially configuring the block, setting up the multiplexers for shared resources, setting the value of the AA, and starting the transmission after all the described things have been configured.
- The "start" and "done" flags have been added as control signals for the higher level controller, and in case they are required by the interface between the Physical Layer (PHY) and the LL.

4-2-3 Block Details

The high-level block diagram of the transmitter part of the bitstream processing chain can be seen in Figure 4-5.

On it (4-5), it is possible to see all the inputs and outputs, and they can be classified into three groups:

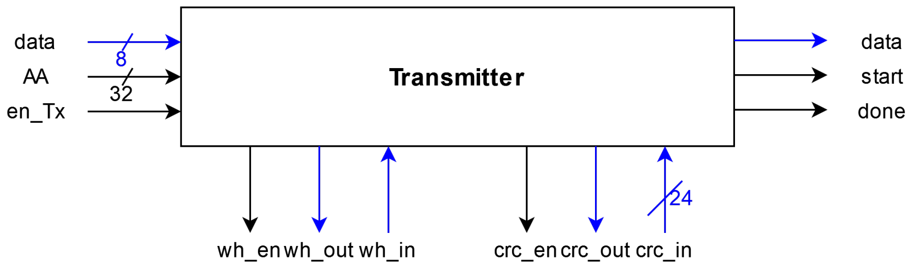


Figure 4-5: Transmitter block of the bitstream processing chain

- Inputs for the transmitter, including the data input from the device and the expected AA, and enable signal coming from the control of the bitstream processing. It is located on the left in the block diagram (4-5).
- Connections with the shared blocks, which include their enable and data input/output signals. It is located at the bottom of the block diagram (4-5).
- Outputs of the transmitter include the data output to the PHY interface and the "start" and "done" flags used for control, located on the right of the block diagram (4-5).

As in previous diagrams, the blue signals correspond with Decoupled interfaces, meaning each has a ready and valid signal.

4-3 Receiver bitstream processing block

In order to explain the designed block in detail, it is necessary first to explain its specific function step by step, the specific inputs and outputs, and then any other relevant design-specific decision.

4-3-1 Functionality

The reception and processing of a LE1M packet consist of the following steps that can also be seen in Figure 4-6:

1. Preamble: the first received bits are used to identify the start of a packet, the preamble.

   During this first step, the Whitening module is initialized, and the AA needs to be received to perform the check with the one received through the air.

2. Access Address: The received 32 bits after the preamble are stored after being unwhitened as it corresponds with the AA, and it is necessary to check it with the expected value. If the AA check is not successful, the packet is then discarded.

   All the received bits after the preamble detection have to be unwhitened. Also, during this step, the CRC is initialized and kept inactive.

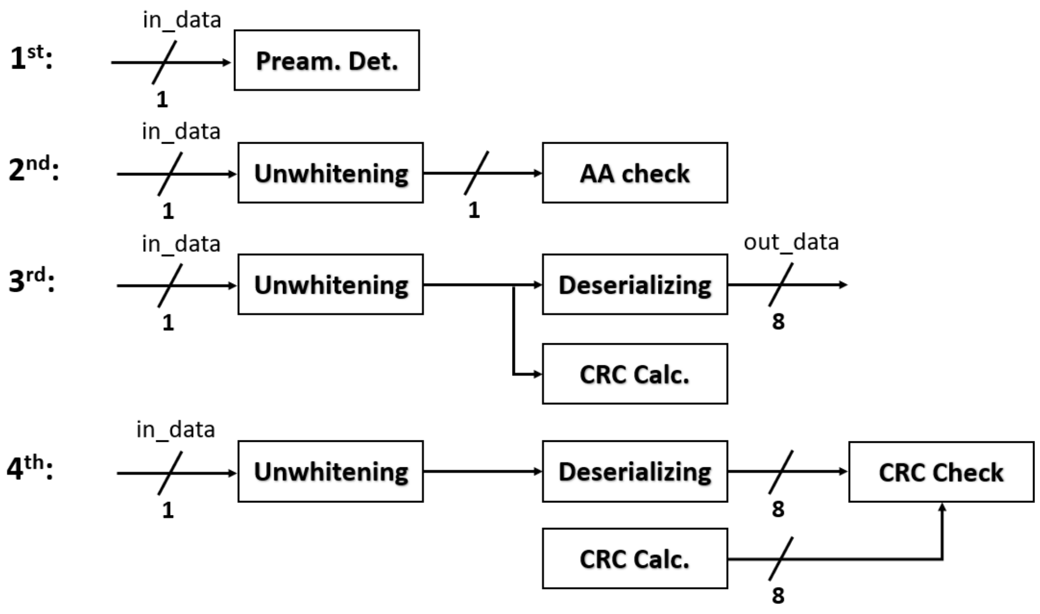


Figure 4-6: Processing steps in the reception flow of the bitstream.

3. PDU: As soon as the whole AA is received in the previous step, the CRC block has to start calculating the CRC value over all the following bits. All the bits need to be unwhitened before being used for the calculation.

   The first 16 bits of this step correspond with the Header of the PDU. Where 5 of them correspond with the length of the payload of the received packet in bytes/octets, this length field is read and used to configure the expected number of bits to be received by the bit processing block.

   This step finishes when the whole PDU is received, including the Header and Payload.

4. CRC: Once the PDU is received, the CRC calculation has to be stopped. The following 24 bits are unwhitened and correspond with the calculated CRC at the transmitter side. This transmitter CRC is compared then with the calculated CRC field at the receiver.

   Any erroneous packet (which didn't pass the CRC check) will then be discarded.

As in the transmitter block of the bitstream processing, some data fields are required to perform all the functions required for the block (AA check and the initial states of the Whitening and CRC blocks). However, as those data fields do not depend on the bitstream processing part of the LL, they can be regarded as inputs to each specific block where they are required.

4-3-2 Design Decisions

In order to achieve the desired functionality, the following design decisions have been taken:

- The control of the different actions performed on each packet part is managed with an FSM with the following states: idle, Access Address, Header, Payload, and CRC. The changes between states are triggered via counters of received bits.

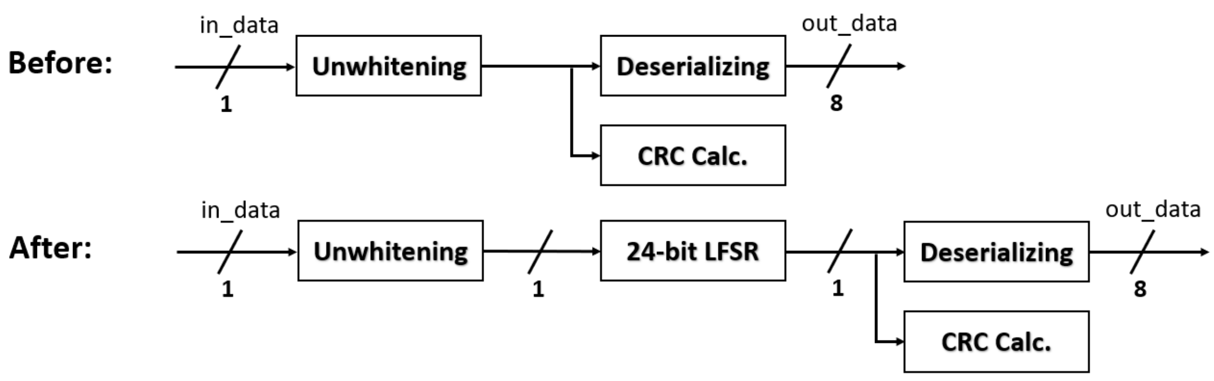


Figure 4-7: Change in the bitstream processing of the PDU Payload.

- The bit width of the data output has been fixed to 8-bits. The reason is the same as in the transmitter; all the BLE packet sections are organized in octets, and using a bigger data width simplifies data transmission.

- Enable signal for reception. As in the transmitter side, an enable signal has been added as part of the logic used to share blocks.

- Several control flags such as start, done, corr_crc, corr_aa have been added, as they are required for control logic.

- Logic to avoid getting stuck in any state due to induced bit errors. If an error is introduced in the "length" field of the PDU Header, two things can happen: the received "length" is lower or higher than it should be. In the first case, this would cause the receiver to stop the processing earlier and therefore get an incomplete and erroneous packet, but the receiver would not get stuck in any state. In the second case, when the received "length" is higher, the receiver would process the whole packet and still be waiting for more bits and, therefore, would get stuck in any of the last two states (payload or CRC).

  In order to avoid this, a 24-bit Linear Feedback Shift Register (LFSR) is placed and used at the output of the Whitening module as soon as the PDU Payload processing starts. By doing this, we avoid calculating the receiver side CRC over the transmitted CRC field, which allows us to get a complete well-processed packet even when the errors are introduced in that critical field of the PDU Header. Another advantage of this change is that it does not increase the number of used registers, as they were already required to store the received CRC value. The change can be seen in Figure 4-7.

- CRC remainder calculation. The remainder is needed as part of the chosen CRC error correction method, and its calculation has been added to this block.

- Payload length output. The following block logic requires the length of the payload. Therefore, an output for the length value is added.

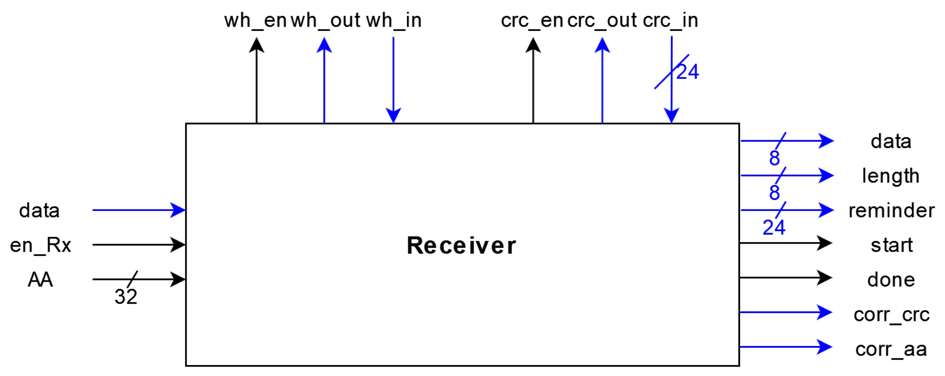


Figure 4-8: Receiver block of the bitstream processing chain.

4-3-3 Block Details

The high-level block diagram of the receiver part of the bitstream processing chain can be seen in Figure 4-8.

In it (4-8), it is possible to see all the inputs and outputs and can be classified into three groups:

- Inputs for the receiver in the bitstream processing: include the data input from the PHY and the expected AA and enable signal coming from the control of the bitstream processing. The inputs can be seen on the left in the block diagram (4-8).
- Connections with the shared blocks: this includes their enable and data input/output signals. It is located on the top side of the block diagram (4-8).
- Outputs of the receiver in the bitstream processing include the data output and several logic control signals and flags. This last group includes a reminder signal used for the CRC error correction, the length of the received payload, a start and done flags used to control the transmission, and the "corr_crc" and "corr_aa" signals used for discarding packets and control purposes. The outputs are located on the right of the block diagram (4-8).

Blue-colored signals in the block diagram shown in Figure 4-8 correspond with Decoupled interface inputs/outputs, which means that the connection also counts with a ready and valid signal in addition to the data signal.

4-4 Bitstream processing block

The high-level block diagram of the bitstream processing block that includes the transmitter, the receiver, and shared resources was already presented in 4-1.

Given that the blocks inside it are now explained, it is possible to explain its function in more detail. It acts mainly as a wrapper to enclose the several blocks that form it and to add the little logic required to share some of its blocks.

The first crucial additional logic is added to make sure that under no circumstance, both enable signals for the receiver/transmitter side of the processing are enabled.

The rest of the logic is added to multiplex signals between the transmitter/receiver and the shared blocks (CRC and Whitening).

The inputs and outputs of the block are the same as in the receiver and transmitter blocks, excluding the ones used to connect the shared blocks or any of their initializations.

Chapter 5

Data Path Design

The blocks that are part of the data path in the proposed architecture for the Link Layer (LL) can be seen in Figure 3-9 colored in orange. More details about the blocks and their connections can be seen in Figure 5-1.

It is composed of all the blocks used to control the data input and output and bit processing between processor and LL, and LL and Physical Layer (PHY) interface. The data path chain is the LL section performing all the LL bit intensive tasks.

5-1 Serializer and Deserializer

Although the data input and output of the Bit Stream Processing block is already an 8-bit width signal, this can be grouped to form bigger packets, such as 32-bits in this case. By doing this, with only one write and one read of the First In First Out (FIFO), the processor can process more data on each clock cycle, avoiding overhead. In this project case, with a 32-bit core, the width of the FIFO and its connections are chosen to be 32-bit.

Therefore the function of these blocks is to pack octets of output data on the receiver side to form 32-bit blocks to be written on the data output FIFO and to perform the opposite operation on the transmitter side. These blocks are also required to manage the necessary ready/valid signals used for all the data connections between blocks and handle the data flow of the design.

It is also worth noting that in the output of the Deserializer, although it is 32-bit wide, only 24 bits correspond to packet data. The other 8 are flags used to identify the section of the data and some other control flags such as the correctness and the start/end of the packet.

Both block diagrams can be seen in Figures 5-2 and 5-3.

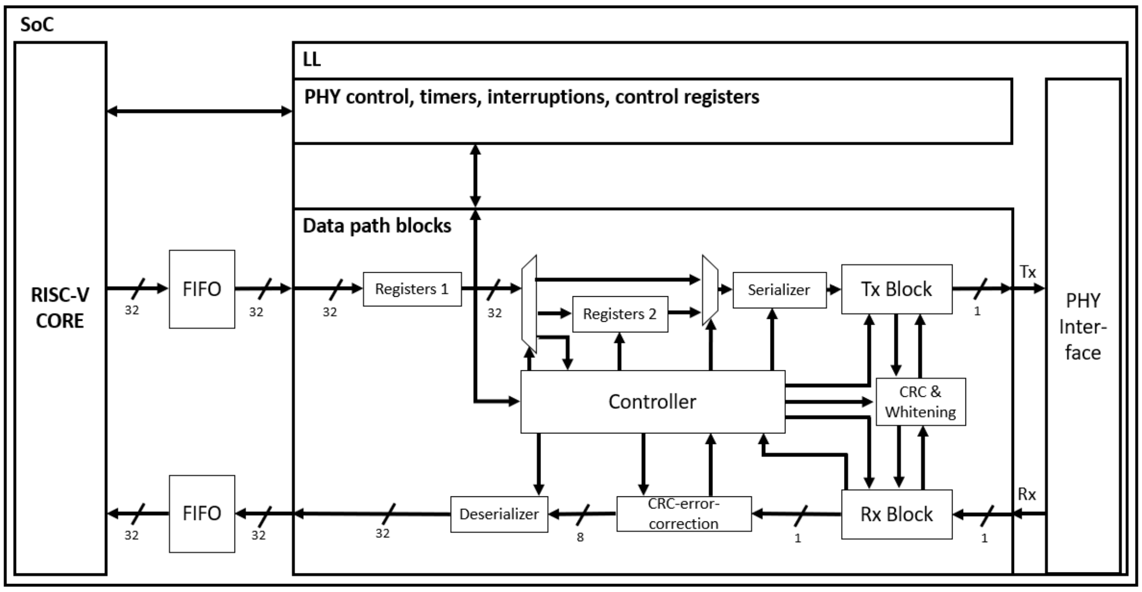


Figure 5-1: In detail data path architecture and connections.

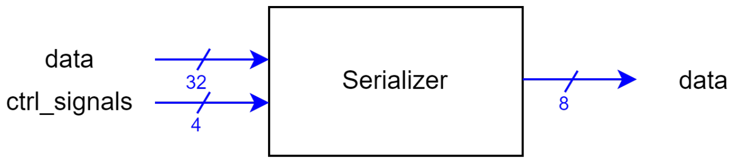


Figure 5-2: Designed Serializer block.

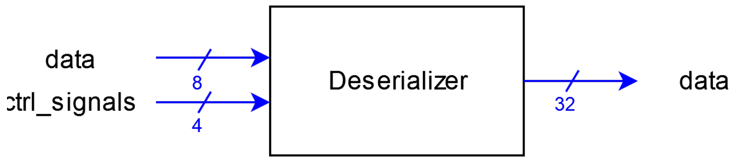


Figure 5-3: Designed Deserializer block.

5-2 CRC Error Correction Block

The algorithm on which this block is based can be found in [7] and has also been described in Section 3-4-3.

Although the chosen algorithm presented fewer issues than other Cyclic Redundancy Check (CRC) error correction techniques, it still presents some challenges. Given the algorithm it uses to find error patterns, the delay in finding a suitable error pattern is linearly related to the packet's length and exponentially with the number of errors considered (single bit error, double bit error ...). If implemented directly, as presented in the paper, the area required for the implementation would also be directly proportional to the length of the considered packet.

5-2-1 Algorithm optimization

In order to reduce the delay between starting the error search and finding the solution of the block, it is necessary to reduce the number of errors considered and the length of the packets to which it will be applied.

The work presented in [7] studied the distribution of error appearances depending on the length. The results of this study are shown in Table 5-1. Also, the algorithm's performance when searching for error patterns under the same conditions is shown, see Table 5-2. The metric they use is the Single Candidate Error Ratio, which is only a percentage of the cases in which a single candidate error pattern is found. That means it is just a percentage of the cases in which the packet can be corrected directly.

From these two previous tables and to reduce latency when correcting errors, applying the error correction algorithm only when the packet length is under 29 Bytes and for single and double errors has been decided. The reasons behind this decision are:

- Correcting only single and double bit errors per packet, the algorithm can correct around 43% of the total erroneous packets of up to 39 Bytes in length.
- When correcting only single and double errors, the algorithm will always find a single candidate error pattern for packets of up to 39 Bytes in length.
- Although Bluetooth Low Energy (BLE) packet lengths can go up to 257 Bytes, this is only with the Data Length Extension enabled; otherwise, the maximum packet length is 29B. For Internet of Things (IoT) applications, the packet length is usually short as they do not require high throughput and therefore do not usually make use of the Data Length Extension.

It is also possible to optimize the algorithm to use even fewer hardware resources, in this case, registers, by applying the following:

- As it is possible to see in Figure 3-7, the only values that will be different from 0 will always be contained in a segment of the error vector of length k, which is then XORed with the polynomial. The previous is true when excluding the positions fixed to 1 as part of the algorithm. k corresponds with the length of the polynomial used for the CRC.

Table 5-1: Error distribution in real environment for BLE and 2 different packet lengths [7]

Number of bit errors	21B	39B
1	18%	16%
2	28%	27%
3	12%	11%
>3	42%	46%

Table 5-2: Single Candidate Error Ratio for a given number of errors and different packet lengths [7]

Number of bit errors	21B	39B
1	100%	100%
2	100%	100%
3	90%	50%
>3	-	-

- The CRC polynomial length will always be shorter than the error vector, whose length is $k - 1 + packetLength$.
- It is possible to implement the error vector that previously required $k + packetLength$ registers as a vector of registers of length k and some additional registers to keep the value of the current displacement from the initial position (shift).

Figure 5-4 shows an example with a polynomial length of 5 and a packet length of 6. In order to represent the error vector, only five registers will be used (darker blue), which contain the actual values that will be used to do the calculations in the next step. In addition, two integer values (its bit width depends on the length of the error vector) will be used to represent the shift and the fixed position to 1.

In the previous case, with the simplification, more registers are required than without, but for higher packet lengths, that changes. For example, in the case of a BLE packet of length 29 Bytes without using the simplification presented above, 255 registers would be required to represent the error vector. However, if the simplification were to be used, only 24 registers would be required for the error vector, in addition to 8 registers for the shift value, which would make a total of 32 registers.

This simplification adds some logic to the design, but it is a small area overhead compared to the amount of extra used registers.

5-2-2 Architecture

The designed block and its connections can be seen in Figure 5-5.

In order to apply the error correction only in some cases, a state machine and additional logic were required to avoid interrupting the data flow through the bitstream processing chain for too long and to store the whole received erroneous packet.

The additional logic consists of the following:

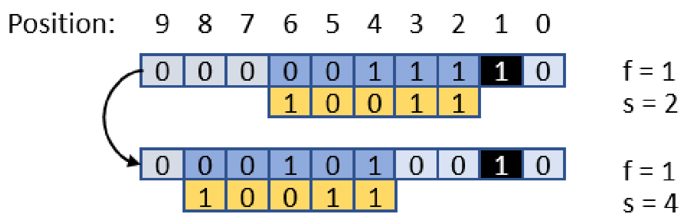


Figure 5-4: Hardware implementation CRC error correction algorithm.

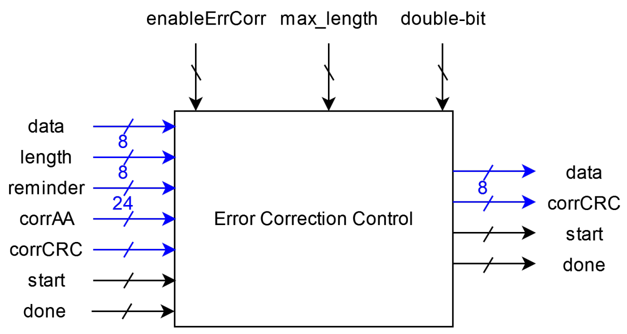


Figure 5-5: Block diagram of the error correction control block

- Enough registers to store the whole packet of any length under the fixed threshold.
- The logic required to either store or send the received octets of the packet.
- The logic to correct a stored erroneous packet when an error is detected.
- The logic used to trigger the error correction block with the algorithm when required.

5-2-3 Packet management and control block

The state machine designed to control the additional logic and the correction block can be seen in Figure 5-6. Its function is as follows:

- As soon as the start signal is received, the block leaves the idle state, either saving the received packet (header1) or directly transmitting it (passthrough), depending on whether the error correction is enabled.
- Once the packet length is received, either the block saves the packet (payload) or directly sends the packet without further processing (send_header1). That depends on whether the packet length is lower than the threshold fixed for error correction or not.
- In the cases where the packet length is below the fixed threshold, and the error correction is enabled, the whole payload is stored (payload). Then, when the corrCRC flag is triggered, the packet is either corrected and transmitted or directly transmitted.
- In the cases where the packet's length is over the fixed threshold, the already saved octets of the packet are sent (send_header), and the following received octets are directly transmitted (passthrough).

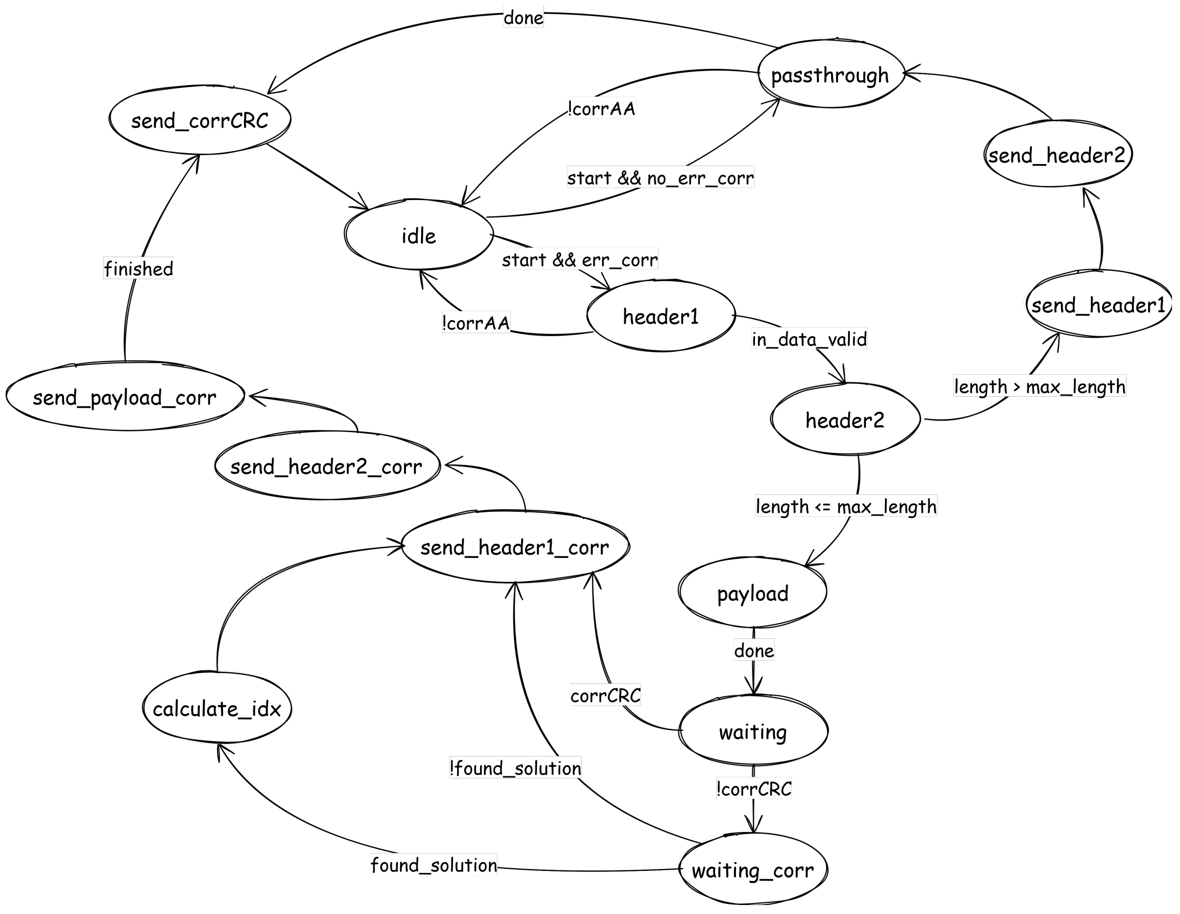


Figure 5-6: State machine modeling the logic required for the application of the error correction algorithm.

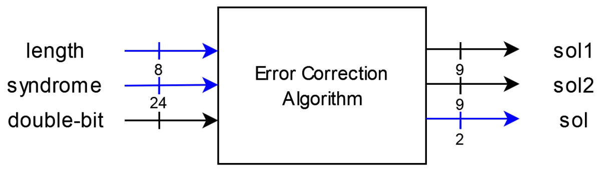


Figure 5-7: CRC error correction algorithm block.

5-2-4 Correction Block

The correction block applying the algorithm and its Input/Output (I/O) can be seen in Figure 5-7.

The inputs of the block are just the length of the packet being corrected and the calculated CRC syndrome. As soon as these two inputs are asserted, the correction starts. The outputs are the (up to) two erroneous bit positions found (sol1 and sol2) and the flag (sol), which indicates which one has to be used if any. When a single bit error solution is found, it will always be asserted in the output sol1.

The block consists of the following:

- A 25 registers Linear Feedback Shift Register (LFSR) and three 9-bit registers are used to model the error vector. They correspond to the section of the error vector being used, the current shift, the maximum shift, and the current fixed to one position.
- The logic used to XOr the selected section of the error vector with the CRC polynomial, the logic used to find the position of the Least Significant Bit (LSB) and second LSB ones, and the required logic to update the fixed position and current shift.
- The state machine that models the algorithm and the logic to set the length limit of the correction algorithm.

5-3 Advanced Control Functionality

Given the specific application of the design, IoT sensors, where it is common to send the same data packet several times, simplifying how this is done to avoid the unnecessary active time of the processor and memory reads becomes relevant. This packet retransmission occurs because the device usually broadcasts it or sends it to several devices connected to the sensor node. Because of this, a bank of registers capable of storing up to 29 Byte packets is implemented in the baseband.

The implemented use of these registers is to broadcast the data packet stored in them with just a single command from the processor or to send them to a given Access Address (AA). To further reduce interactions, the AA used for several categories of broadcasted messages is also stored inside the baseband, allowing broadcasts of a saved packet with just one control command.

It is possible to enable/disable the CRC error correction at will, allowing to disable it even when the packet length is below the established threshold to avoid the impact of the delay

Table 5-3: Implemented commands for the bitstream controller.

	Name	Description
#1	rxOne	Baseband enters Rx mode until a packet is received
#2	rxContinuous	Baseband enters Rx mode until rxStop is received
#3	rxStop	Stop Rx mode
#4	sendStatus	Baseband sends various parameters to the FIFO
#5	txSavedBroad	Baseband broadcasts the saved packet
#6	modifyAA	Modify the Rx AA to a new value
#7	setTrPower	Sets the Transmission Power of the antenna to a new value
#8	savePacket	Saves the following packet in the registers
#9	txSavedAA	Baseband transmits the saved packet to the following AA
#10	tx	Baseband transmits the following AA + Data Packet
#11	actEC	Activates the CRC error correction
#12	deactEC	Desactivates the CRC error correction
#13	paramsEC	Sets several parameters of the CRC error correction
#14	soft_reset	Soft reset that sets all the baseband blocks to the idle state

when necessary. Following this idea, the maximum length of the data packet to be corrected must also be editable to reduce possible delays further, allowing for applying error correction even in more time-constrained situations.

5-4 Bit stream Controller

This block supervises and controls all the advanced functionalities and data flow through the bit stream processing chain and manages the shared resources between transmitter and receiver processing blocks.

The bit stream controller supervises and controls all the functionalities mentioned above, the error correction, and the data flow through the bit stream processing chain.

It is designed to use an 8-bit commands input to configure the data flow through the designed baseband and either direct it to the processing blocks or store the received data through the FIFO in order to use it to configure any of the parameters (such as transmission power or maximum data length for the error correction).

A complete list of the enabled commands in the final version of this project and their description can be seen in Table 5-3.

It is essential to mention how the structure of the data sent through the input and output acFIFOs is fixed to be easily read by the processor.

The data structure of the 32-bit entries of the FIFOs connecting the processor to the baseband:

- When saving a data packet (command #8): First FIFO entry: number of octets in the last read [17:15]; payload length [14:7]; the number of packet data entries[6:0]. The second and following entries already correspond with packet data.

- When transmitting a packet directly (#10): First FIFO entry: number of octets in the last read [17:15]; payload length [14:7]; the number of packet data entries[6:0]. The second entry corresponds with the AA. Third and following entries correspond with the packet data.
- When setting the error correction parameters (#13): unused [31:9]; single or double error correction [8]; maximum packet length between 256 and 1 bytes [7:0].
- When manually setting the transmission power (#7): unused [31:8]; transmission power [7:0].
- When changing the AA of the baseband (#6), no structure is needed, it is 32-bits long.

The data structure of the 32-bit entries of the FIFOs connecting baseband to the processor:

- When receiving a packet (#1 and #2), all the FIFO entries have some appended flags in the 8 Most Significant Bit (MSB). The structure is: start [31]; done [30]; number of data octets in the entry [29:28]; correct packet [27]; unused [26:24]; 3 octets of packet data [23:0].
- When reading the status (#4), the structure is as follows: ones sequence [31:24]; current used channel [23:16]; measured Received Signal Strength Indicator (RSSI) [15:8]; transmission power [7:0].

Chapter 6

Simulation and Results

Several simulations have been performed during the design of the project. This chapter shows the simulations performed to measure the performance of the Cyclic Redundancy Check (CRC) error correction block, and the obtained results during the Field Programmable Gate Array (FPGA) emulation step. More in detail simulations of every function can be found in the simulations shown in Appendix D.

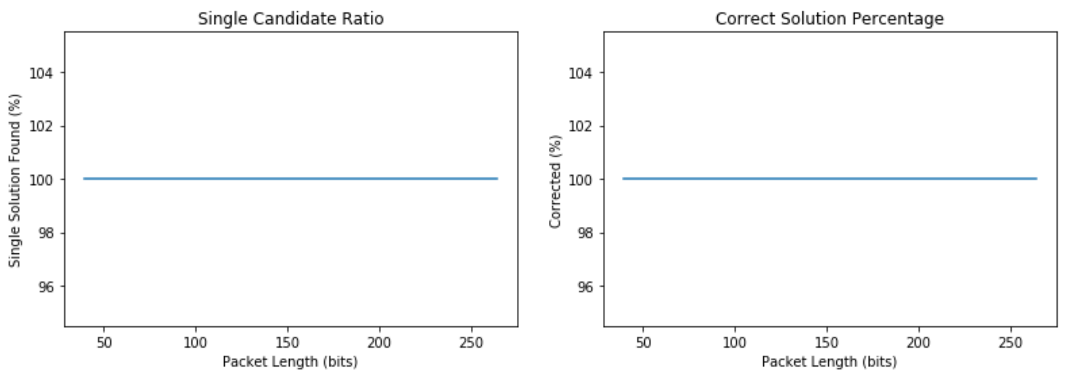
6-1 Python Simulations

This section is only about the simulation results of the python model of the designed CRC error correction block. The reason behind simulating this block in python rather than simulating the Register Transfer Level (RTL) is reduced simulation time and flexibility.

This section has been added as most of the initial simulations of the CRC error correction algorithm have been performed on python. A model of the hardware block implementing the error correction algorithm has been created, and several study cases have been tested on it.

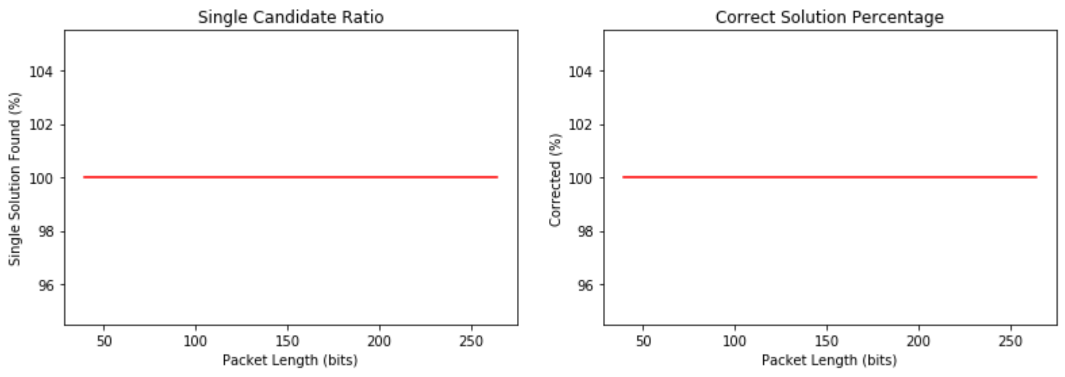
The results of these simulations can be seen in Figures 6-1 and 6-2. In subfigures 6-1a and 6-2a, it is possible to see the obtained Single Candidate Ratio (SCR) percentage for single and double bit error correction under different packet lengths; in all the cases, the SCR is 100 %. The SCR is a metric that indicates the cases in which a single solution candidate for a given erroneous packet is found. It is also possible to see the performance of this algorithm implementation in the subfigures 6-1b and 6-2b.

No relation between the length of the corrected packet and the performance can be seen for small packet lengths. However, in literature [28], the packet length effect can be seen taking relevance for longer packets than the ones used in this project.



(a) Single candidate ratio percentage. **(b)** Percentage of corrected packets.

Figure 6-1: Single bit error correction simulations in python



(a) Single candidate ratio percentage. **(b)** Percentage of corrected packets.

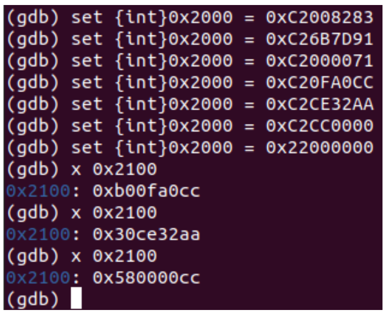
Figure 6-2: Double bit error correction simulations in python

6-2 FPGA Emulation

As part of the Rocket-Chip generator, everything needed to implement and test a designed System on Chip (SoC) is provided for a series of FPGA models. In addition, the tools provided with the generator will create a wrapper to properly connect all the signals coming in and out of the SoC and set up and connect the JTAG interface to one of the pin headers in the board.

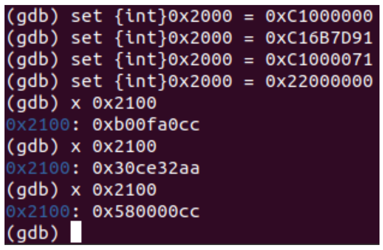
The environment also contains all the necessary constraints and clocks used to work correctly in the specific FPGA, making it fairly fast and simple to implement in one.

A more in detail explanation about how to set-up the FPGA environment and simulate the SoC is shown in Appendix C. An explanation about the SoC changes introduced to simulate the SoC or emulate it is shown in Appendix B.



```
(gdb) set {int}0x2000 = 0xC2008283
(gdb) set {int}0x2000 = 0xC26B7D91
(gdb) set {int}0x2000 = 0xC2000071
(gdb) set {int}0x2000 = 0xC20FA0CC
(gdb) set {int}0x2000 = 0xC2CE32AA
(gdb) set {int}0x2000 = 0xC2CC0000
(gdb) set {int}0x2000 = 0x22000000
(gdb) x 0x2100
0x2100: 0xb00fa0cc
(gdb) x 0x2100
0x2100: 0x30ce32aa
(gdb) x 0x2100
0x2100: 0x580000cc
(gdb) 
```

Figure 6-3: Transmission and reception of a packet, FPGA emulation.



```
(gdb) set {int}0x2000 = 0xC1000000
(gdb) set {int}0x2000 = 0xC16B7D91
(gdb) set {int}0x2000 = 0xC1000071
(gdb) set {int}0x2000 = 0x22000000
(gdb) x 0x2100
0x2100: 0xb00fa0cc
(gdb) x 0x2100
0x2100: 0x30ce32aa
(gdb) x 0x2100
0x2100: 0x580000cc
(gdb) 
```

Figure 6-4: Transmission and reception of a saved packet, FPGA emulation.

6-2-1 Results

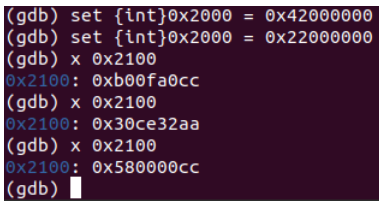
Transmission and Reception

In order to test everything the easiest way possible, first the desired packet to be sent has been stored in the baseband registers. Then the transmit command has been written in the correspondent memory address. After that, as the transmitted packet is in the First In First Out (FIFO) connecting output and input from the antenna (this is described in Appendix B), it is possible to write the simple rx command and proceed to its read from the correspondent memory address.

The above described can be seen in Figure 6-3. Where the consecutive writings starting with 0xC2 correspond with the transmission's configuration, the transmission, the Access Address (AA) of the transmission is split into two entries, and the transmission's data packet is split into three entries. The last writing corresponds with the baseband's simple reception mode (0x22000000). After that three consecutive reads are executed to check the output of the receiver in the baseband. It is possible to see that the packet has been correctly transmitted and received.

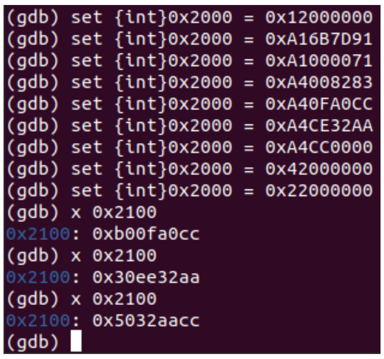
Transmission of a saved packet

This mode can be seen in Figure 6-4. The three first written instructions correspond with the configuration of the txSavedAA (commands starting with 0xC1) mode and the two sections of the AA. The fourth one is again the simple rx mode (0x22000000).



```
(gdb) set {int}0x2000 = 0x42000000
(gdb) set {int}0x2000 = 0x22000000
(gdb) x 0x2100
0x2100: 0xb00fa0cc
(gdb) x 0x2100
0x2100: 0x30ce32aa
(gdb) x 0x2100
0x2100: 0x580000cc
(gdb)
```

Figure 6-5: Broadcast and reception of a saved packet, FPGA emulation.



```
(gdb) set {int}0x2000 = 0x12000000
(gdb) set {int}0x2000 = 0xA16B7D91
(gdb) set {int}0x2000 = 0xA1000071
(gdb) set {int}0x2000 = 0xA4008283
(gdb) set {int}0x2000 = 0xA40FA0CC
(gdb) set {int}0x2000 = 0xA4CE32AA
(gdb) set {int}0x2000 = 0xA4CC0000
(gdb) set {int}0x2000 = 0x42000000
(gdb) set {int}0x2000 = 0x22000000
(gdb) x 0x2100
0x2100: 0xb00fa0cc
(gdb) x 0x2100
0x2100: 0x30ee32aa
(gdb) x 0x2100
0x2100: 0x5032aacc
(gdb)
```

Figure 6-6: Packet with introduced bit error FPGA emulation

Broadcast of a saved packet

This mode can be seen in Figure 6-5 where the first written command (0x42000000) corresponds with the txBroadcast mode and the second with the simple rx mode (0x22000000). It is possible to see that the saved packet 0x0FA0CC CE32AACC is sent and received to the previously set-up broadcast AA.

Error correction block

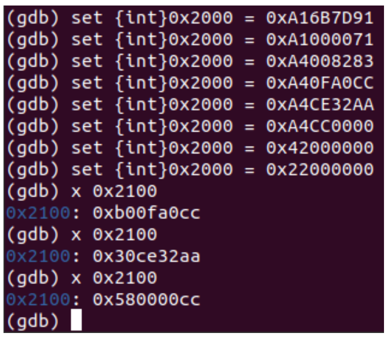
As in the whole SoC simulation using Verilator, it becomes necessary to introduce errors in the data stored in the FIFO connecting output and input to the antenna to test the error correction block.

With this last change, and to test the mode, first the command actEC needs to be sent to the baseband, and then it is possible to transmit and receive a packet. Then, if the transmitted and received packet match, we can assume it works as expected.

As it is possible to see if the error correction is not enabled or it has been disabled with the command 0x12000000, an erroneous packet is received, as seen in Figure 6-6. Nevertheless, if error correction is enabled, which is by default, the packet received is corrected. The reception of the corrected packet can be seen in 6-7.

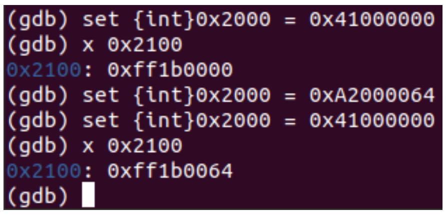
Reading the state of the baseband

Figure 6-8 shows how to read the baseband's state and update its internal registers' value. It is possible to see how to read the status using the command 0x41000000 and how to change



```
(gdb) set {int}0x2000 = 0xA16B7D91
(gdb) set {int}0x2000 = 0xA1000071
(gdb) set {int}0x2000 = 0xA4008283
(gdb) set {int}0x2000 = 0xA40FA0CC
(gdb) set {int}0x2000 = 0xA4CE32AA
(gdb) set {int}0x2000 = 0xA4CC0000
(gdb) set {int}0x2000 = 0x42000000
(gdb) set {int}0x2000 = 0x22000000
(gdb) x 0x2100
0x2100: 0xb00fa0cc
(gdb) x 0x2100
0x2100: 0x30ce32aa
(gdb) x 0x2100
0x2100: 0x580000cc
(gdb)
```

Figure 6-7: Single bit error correction FPGA emulation



```
(gdb) set {int}0x2000 = 0x41000000
(gdb) x 0x2100
0x2100: 0xff1b0000
(gdb) set {int}0x2000 = 0xA2000064
(gdb) set {int}0x2000 = 0x41000000
(gdb) x 0x2100
0x2100: 0xff1b0064
(gdb)
```

Figure 6-8: Reading and updating status registers FPGA emulation

the value of some internal registers, such as the one used for the transmission power using the command 0xA2000064.

6-2-2 FPGA Reports

The area used by the design on the FPGA can be seen in Table 6-1, a more in detail area report can be seen in Appendix C.

The power consumed by the design on the FPGA was a total of 0.023 W for the whole SoC, and 0.004 W for the designed baseband block and its connections. A more in detail power report can be seen in Appendix C.

Table 6-1: Area utilization report FPGA

Resource	Utilization #	Available #	Utilization %
LUT	10480	20800	50.38
LUTRAM	782	9600	8.15
FF	7257	41600	17.44
BRAM	5.5	50	11.00
DSP	2	90	2.22

Chapter 7

Conclusion and Future Work

7-1 Conclusion

This thesis project provides a novel Bluetooth Low Energy (BLE) baseband architecture design for very low power and area for Internet of Things (IoT) autonomous applications. Its advantage is the reduction in the area specifically dedicated to the BLE protocol implementation, the reduction of unnecessary communication between that hardware and the processor, and the inclusion of a double-bit error correction hardware block that makes use of the Cyclic Redundancy Check (CRC) to reduce the power consumption due to retransmissions while improving the reliability of the connection. All the bit and packet processing tasks that form the bitstream processing chain, which includes the error correction and the extra functionalities, have been designed and tested as a proof of concept both in simulation and a Field Programmable Gate Array (FPGA) implementation.

7-2 Future Work

Given the size of the project, only the bit-intensive tasks of the baseband have been implemented, as can be seen in Figure 3-9 in color.

Therefore more work is required to finish designing the unimplemented remaining parts of the baseband. These parts are the timers, the control algorithm used to select the channel, a whitelisting algorithm implementation in hardware, the controller for the interruption and the control registers, and the Physical Layer (PHY) interface block.

7-2-1 Finish Implementation of the baseband

Once the proposed architecture is finished, it would also be interesting to follow the Application-Specific Integrated Circuit (ASIC) implementation flow with the finished architecture to check the area, power, and timing results of the architecture.

7-2-2 Chipyard

In the last months, several other 32-bit cores have been added as Intelectual Property (IP) to Chipyard. Therefore implementing the same System on Chip (SoC) with a different processor core should be studied as it could lead to better area/power results. A publication in which open source core IPs to be used for low power and low computation requirements are already compared. That publication also considered the small version of the Rocket-Core used in this project, but the authors ended up selecting the PICORV32 over it [29].

It is also important to mention that although the tools provided within the environment are extremely useful, they are also not very well documented, leading to a big investment of time in learning their use.

7-2-3 New CRC error correction algorithms

Although the selected CRC error correction algorithm seems like a promising implementation, the same authors of the publication in which is based [7] have recently published (2022) an improved CRC error correction algorithm that seems to outperform their previous publication [28].

Therefore it would be interesting to study and implement the new algorithm and compare it with the implemented in this project.

7-2-4 Error correction delay study and model

The proposed algorithm has a reduced total delay thanks to the decisions taking to limit the maximum packet length and the number of errors, but still its maximum delay (around a 1000 clock cycles, or 1 ms) could be too much in some cases. In addition to that, the delay can not be fully determined, as for different error syndromes and under same packet and error correction characteristics different correction delays are obtained.

Therefore in order to apply the error correction on the more time constrained scenarios, two actions could be taken. First, implement a counter and a maximum to that counter to limit the correction time. And second, study and model the delay of the algorithm to characterize it and apply the error correction algorithm in time constrained cases.

7-2-5 Implementation of the software layers

Once the hardware design is completed, all the software layers of the BLE stack will need to be implemented to obtain a functional baseband design. However, many manufacturers of BLE modules offer their software stack online; therefore, only the specific changes introduced as part of the proposed architecture would be required.

Appendix A

Appendix: Other Introduction Topics

A-1 Chisel

Chisel is the Hardware Description Language (HDL) that will be used for the design presented in this project, an introduction to it can be found below.

Constructing Hardware in a Scala Embedded Language (Chisel) [30] [31] is an open-source HDL used to describe digital electronics and circuits at a Register Transfer Level (RTL) level. It facilitates the generation of advanced circuits and the design reuse for both hard (Application-Specific Integrated Circuit (ASIC))) and soft (Field Programmable Gate Array (FPGA)) digital logic design implementations [30].

Traditionally used HDLs, Verilog and VHSIC Hardware Description Language (VHDL), were developed as hardware simulation languages and were later adopted as the base for hardware synthesis. Due to this, their semantics are not based on synthesis, and synthesizable designs must be inferred from the language, complicating the tool development [30]. These languages also lack the powerful abstraction facilities that are common in modern software languages, leading to low productivity and difficult component reusability [30]. Some of these traditional limitations have been recently improved with the addition of extensions such as SystemVerilog, but it still lacks some modern programming language features.

Chisel is based on the Scala programming language, to which it adds hardware construction primitives [32], providing a modern programming language capable of writing complex and parameterizable hardware generators. That allows the creation of reusable components and libraries, raising the abstraction level in the design while still keeping a low-level control. Furthermore, it is powered by Flexible Intermediate Representation for RTL (FIRRTL), a hardware compiler framework that performs optimizations of Chisel-generated circuits and supports custom user-defined circuit transformations [30]. That means that between the Chisel code and the generated Verilog, there is an intermediate transformation into FIRRTL.

From the originalChisel paper [30]: "Chisel is intended to be a simple platform that provides modern programming language features for accurately specifying low-level hardware blocks, but which can be readily extended to capture many use full high-level hardware design patterns. By using a flexible platform, each module in a project can employ whichever design pattern best fits that design and designers can freely combine multiple modules regardless of their programming model".

In order to illustrate how Chisel can be used to create flexible and parameterizable hardware generators, an Adder code example and the changes to make it generic for different widths are introduced below. Also, in the same code snippet, a few hardware blocks that are part of the language have been added.

```
// 8-bit Width Adder Module
class Add extends Module {
  val io = IO(new Bundle {
    val a = Input(UInt(8.W))
    val b = Input(UInt(8.W))
    val y = Output(UInt(8.W))
  })

  io.y := io.a + io.b
}

//Parameterizable Width Adder Module
class Add(val bitWidth: Int) extends Module {
  val io = IO(new Bundle {
    val a = Input(UInt(bitWidth.W))
    val b = Input(UInt(bitWidth.W))
    val y = Output(UInt(bitWidth.W))
  })

  io.y := io.a + io.b
}

//A 32-bit register with a reset value of 0:
val reg = RegInit(0.U(32.W))

//A multiplexer (part of the Chisel library) with selection signal (sel):
val result = Mux(sel, a, b)
```

A-2 RISC-V

Given the nature of the cores available to be used on the final System on Chip (SoC) of the design, it becomes necessary to explain what RISC-V is. RISC-V is a recent Instruction Set Architecture (ISA) that was initially developed by the University of California Berkeley (UCB) and that was later managed by the RISC-V Foundation. As its name indicate, RISC-V is an ISA of the Reduced Instruction Set Computer (RISC) type. The first base user spec was released in 2014.

It is also necessary to define what an ISA is. An ISA is a part of the abstract model of a computer that defines how the Central Processing Unit (CPU) is controlled by the software [33]. It acts as an interface between the hardware and the software, specifying what the processor can do and how it gets it done. It defines the supported data types, the registers, how the hardware manages the main memory, key features (such as virtual memory), which instructions a microprocessor can execute, and the input/output model of multiple ISA implementations.

ISAs can be extended by adding instructions, other capabilities, or support for larger addresses or data types. All of this is governed by the above-mentioned RISC-V International organization.

The base implementation of RISC-V includes only integer operations for either 32-bit (RV32I) or 64-bit (RV64I). The simplest implementation (RV32I) includes only 40 unique instructions and 32 32-bit width registers. Many other ratified by RISC-V International extensions exist:

- M: Integer Multiplication and Division
- A: Atomic Instructions
- F: Single-Precision Floating-Point
- D: Double-Precision Floating-Point
- Q: Quad-Precision Floating Point
- C: Standard Extension for Compressed Instructions
- G: Is the short version for the extensions M, A, F and D all together

A-3 Rocket-Chip generator

Rocket-Chip is an open-source SoC generator developed by the Berkeley Architecture Research (BAR) group at the UCB [8]. BAR's parameterizable chip generator serves as the basis for all the RISC-V implementations produced by the group. It can be used to generate synthesizable RTL Verilog to push the designs through Very Large Scale Integration (VLSI) Computer Aided Design (CAD) tools or to generate cycle-accurate C++ models for simulation. This tool is devised to enable designers to build and customize their SoC around RISC-V cores. The whole generator IPs are written in Chisel.

The Rocket-Chip generator allows to generate custom SoC in a modular approach, thanks to Chisel's versatility as a parameterized hardware generator and the standardization of the interface between blocks. It offers the possibility to customize many parameters of the design, such as customizing the cores to support different ISA extensions to generate SoCs that satisfy the requirements of each required design. It also allows the addition and interconnection of Verilog/Chisel designed modules to the SoC. In this aspect, the generator can be regarded as a collection of configurable Intelectual Property (IP) and the tools to thoroughly combine them with your own designed blocks easily and quickly.

The design flow using Rocket-Chip is as follows: first new hardware blocks are designed in Chisel (Adding Verilog source files is also possible), then new configurations are created

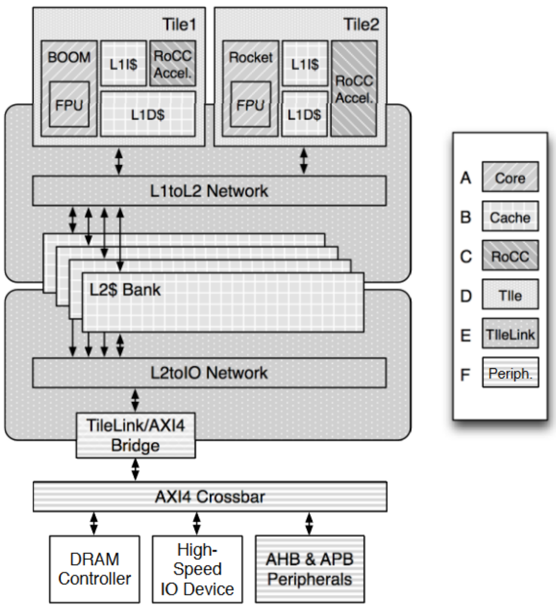


Figure A-1: Example of a generated SoC [8].

to include the new blocks in the SoC, then the design is compiled to obtain C++ cycle-accurate models for simulation, and in the last step the synthesizable Verilog is generated for conventional VLSI CAD tools.

An example of an SoC generated using the Rocket core libraries, and Rocket-Chip can be seen at A-1. It is possible to see there are two tiles with independent L1 cache connected to the shared four banks of L2 cache. The shared L2 cache is connected to the external I/O and memory systems via an AXI4 connection [8]. Each of the tiles is composed of different cores/configurations and coprocessors.

A-3-1 The Rocket Core

The Rocket Core library contains the files to parameterize and create several processor core designs. All the designs are based on the same five-stage in-order scalar pipeline processor core (can be seen at A-2), and as a base, implement either 32-bit or 64-bit integer RISC-V ISA. As part of the generator configuration options [8], it is possible to:

- Add several ISA standard extensions such as M, A, F, or D.
- Modify the cache and Translation Lookaside Buffer (TLB) sizes
- Modify the number of floating-point pipeline stages

The Rocket-Core has a Memory Management Unit (MMU) that supports page-based virtual memory, a non-blocking data cache, and a front-end with branch prediction. The included branch prediction is configurable and provided by: a Branch Target Buffer (BTB), a Branch

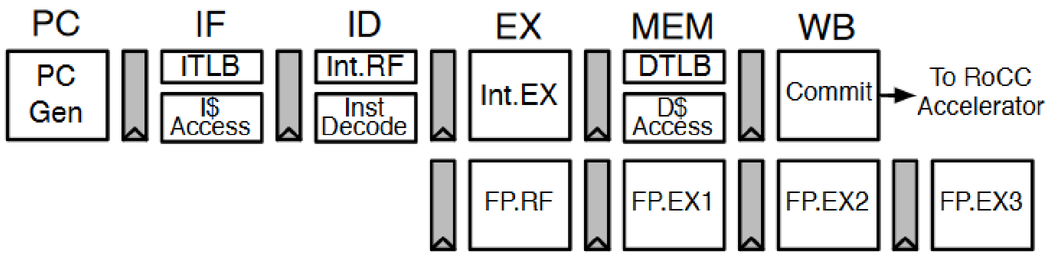


Figure A-2: Rocket core pipeline [8].

History Table (BHT) and a Return Address Stack (RAS) [8]. It uses Berkeley's Chisel implementations of the floating-point units and supports all RISC-V machine, supervisor, and user privilege levels [8].

It also provides the Rocket Custom Coprocessor Interface (RoCC) that facilitates the decoupled communication between the Rocket Core and attached coprocessors. The interface also accepts coprocessor commands generated by specific instructions (custom instructions/commands) executed by the Rocket Core, making it possible to write in a register in response [8]. The interface also allows the attached coprocessor to share the Rocket core's data cache and provides a facility for the coprocessor to interrupt the core [8]. Also, RoCC coprocessors can connect to the outer memory system via the TileLink interconnection [8].

A-3-2 Chipyard

Chipyard is only a bigger and improved SoC generation tool compared to the Rocket-Chip generator. Since the creation of the Rocket-Chip generation, many other engineers saw the tool's potential if more IPs were to be added instead of just relying on always using Rocket-Cores on the SoC. Chipyard was created then, and a quote from the original Chipyard publication can help answer why this is necessary if Rocket-Chip already existed.

"Chipyard provides a framework to bring together a collection of independently developed open source tools and RTL generators, allowing the development of heterogeneous SoCs through integrated design, simulation, and implementation environments. Chipyard helps alleviate many of the challenges that exist when using independent and uncoordinated open-source tools and designs, as often experienced in concurrent and nonuniform feature design iterations, typical in the agile design process." [34].

Chipyard can be considered a tool/framework in which many different parameterizable generators, simulation tools, and implementation environments are combined. That allows us to use IP blocks from several sources that have been added to the environment and our own Chisel/Verilog designs and combine them into an SoC. Another reason for using Chipyard instead of directly the Rocket-Chip generator is the fact that in case of needing a different Core (instead of a variant of the Rocket Core), the change is very fast and straightforward (Chipyard includes several open-source core designs coming both from companies and universities). Another relevant point is that the documentation is more recent and up to date, therefore easier to follow when compared to the one of the Rocket-Chip generator, while having added more tools such as simulators, support for more FPGAs, compilers, and others.

Appendix B

Appendix: System on Chip Design with Rocket-Chip

In order to test the whole System on Chip (SoC), given that the Timers and Interruptions Controller and the Interruption and Control registers have not been implemented, it becomes necessary to attach the 8-bit control signal of the Bit stream Controller in a different way to the processor core.

In order to achieve that, the input data First In First Out (FIFO) (the one connecting the processor with baseband) signals have been modified. Instead of 32-bit data, now it will be 8-bits of control signals for the controller and 24-bits of data.

This change allows testing the functionality of the designed part of the baseband when connected to the processor, making it possible to use processor instructions to write and read data from the baseband to test that it is working correctly.

Also, a second change needs to be introduced to test the whole SoC this way. The data output and input of the baseband (the 1-bit signals that would connect with the Physical Layer (PHY) Interface) are connected via an additional FIFO. This way, it is possible to transmit a packet that will get stored in the FIFO, and we can receive a packet after from that same FIFO.

A block diagram showing the changes applied for testing the whole SoC can be seen in Figure B-1, marked with the color orange.

B-1 Connecting the design with the processor

In order to send and receive data from the baseband, a series of connections have to be established with the processor. The chosen connection for exchanging data is a memory mapped connection with FIFOs.

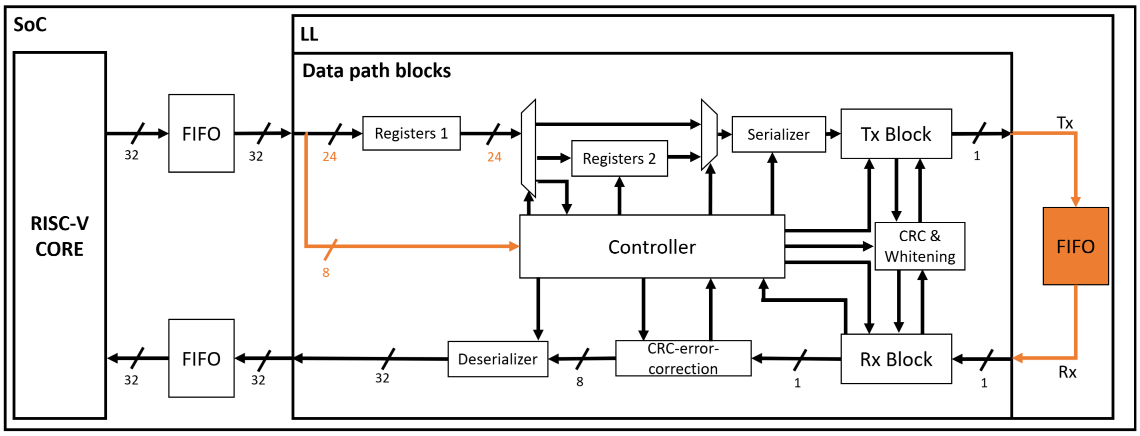


Figure B-1: Changes introduced for testing the SoC (in orange).

B-1-1 MMIO peripheral

A memory mapped connection is usually used to connect accelerators or peripherals to a processor. The connection is given an address in the address map, the same as RAMs or other memories would have, and then the same instructions used to write in memory can be used to write and read data from those addresses.

In this case, in rocket-chip, a Memory Mapped Input/Output (MMIO) device is connected as follows:

```
class TLReadQueue
(
  depth: Int = 4,//4 for 32bit, 8 for 64 bit.
  csrAddress: AddressSet = AddressSet(0x2100, 0xff),
  beatBytes: Int = 4//4 for 32bit, 8 for 64 bit.
)(implicit p: Parameters) extends ReadQueue(depth) with TLHasCSR {
  val devname = "tlQueueOut"
  val devcompat = Seq("ucb-art", "dsptools")
  val device = new SimpleDevice(devname, devcompat) {
    override def describe(resources: ResourceBindings): Description = {
      val Description(name, mapping) = super.describe(resources)
      Description(name, mapping)
    }
  }
  // make diplomatic TL node for regmap
  override val mem = Some(TLRegisterNode(address = Seq(csrAddress), device =
      device, beatBytes = beatBytes))
}
```

Where the *depth* and *beatBytes* variables indicate the bit width of the connection, and the *csrAddress* indicates the memory address to which it is attached and the reserved address space. The extended *ReadQueue*(*depth*) class is the created FIFO class. It is also worth pointing out the *overridevalmem* which adds the created memory connection to the memory map of the SoC.

B-1-2 FIFO connections

It is a common practice to split the entire length of some data into smaller pieces, as processors can only handle determined lengths (i.e. 32-bits, 64-bits, and even 512-bits), and the data to be treated is usually longer than those determined lengths.

Then, to move data between two devices (i.e. processor and coprocessor), it is sent split into pieces in a specific order. These pieces can be stored into FIFOs to be read in the same order when the device reading them is ready. This use of FIFOs also presents an advantage, as by using asynchronous FIFOs, we can have two different clock domains (one on each device) and not have a problem with the clock domain crossing.

This presented case is the same as in the project, and therefore 32-bit wide FIFOs have been used as an interconnection for packet data between the designed baseband and the processor. The used FIFOs in the design have been implemented as follows:

```
abstract class ReadQueue
(
  val depth: Int = 8,
  val streamParameters: AXI4StreamSlaveParameters = AXI4StreamSlaveParameters()
)(implicit p: Parameters)extends LazyModule with HasCSR {
  val streamNode = AXI4StreamSlaveNode(streamParameters)

  lazy val module = new LazyModuleImp(this) {
    require(streamNode.in.length == 1)

    // get the input bundle associated with the AXI4Stream node
    val in = streamNode.in(0)._1
    // width (in bits) of the input interface
    val width = 32
    // instantiate a queue
    val queue = Module(new Queue(UInt(width.W), depth))
    // connect queue output to streaming output

    queue.io.enq.valid := in.valid
    queue.io.enq.bits := in.bits.data
    in.ready := queue.io.enq.ready
    // don't use last
    //in.bits.last := false.B

    regmap(
      // each read cuts an entry from the queue
      0x0 -> Seq(RegField.r(width, queue.io.deq)),
      // read the number of entries in the queue
      (width+7)/8 -> Seq(RegField.r(width, queue.io.count)),
    )
  }
}
```

Where the variables *width* and *depth* indicate each entry's bit-width and the FIFO's maximum number of entries. Also, it is worth noting the *ready* and *valid* signals used as part of the

Decoupled Interface and the *regmap* variable that indicates how the signals of the FIFO will be mapped to the given absolute memory address.

Once the memory mapped connection and the FIFO class are created, this is added to the design as follows:

```
class BasebandThing
(
  val depthWriteDataFIFO: Int = 100,
  val depthReadDataFIFO: Int = 90
)(implicit p: Parameters) extends LazyModule {
  // instantiate lazy modules
  val writeQueue = LazyModule(new TLWriteQueue(depthWriteDataFIFO))
  val basebandblock = LazyModule(new TLBasebandBlock())
  val readQueue = LazyModule(new TLReadQueue(depthReadDataFIFO))

  // connect streamNodes of queues and designed block
  readQueue.streamNode := basebandblock.streamNode := writeQueue.streamNode

  lazy val module = new LazyModuleImp(this)
}
```

The variables $depthWriteDataFIFO$ and $depthReadDataFIFO$ indicate the maximum number of entries of the used FIFOs.

B-2 Clock Domains

The idea presented for the design is to use two different clock domains in the SoC, one being the main clock of the processor and the other (used in the baseband) being the one given by the specific PHY. This idea would eliminate any possible clock domain crossing, as the baseband would synchronously process bits at the specific clock domain of the PHY.

Multi clock domain has support in Chipyard/Rocket-chip, but given that the PHY has not been implemented and there were no plans for connecting one for testing the design, support for a second clock domain was not provided. Therefore the multi-clock domain has not been implemented in the designed baseband.

B-3 Custom Extra Pins Top Level

In Rocket-chip extra top-level pins can be easily added to the designed SoC and attached to the designed block, how to do it is specified in 6.14.1. IOBinders in [35].

Given that the specific PHY interface hasn't been implemented, extra top level I/O was not implemented.

B-4 Configuring the SoC

After checking the configurations of the existing commercial Bluetooth Low Energy (BLE) modules, it is clear that using a single 32-bit core and a small cache is necessary.

The selected configuration will have only a single 32-bit RV32M Rocket-Core and 1 KB of L1 Cache. In the following code, it can be seen how to achieve this configuration with the Rocket-Chip generator.

```
class myModifiedTinyRocketConfig extends Config(
  new chipyard.config.WithTLSerialLocation(
    freechips.rocketchip.subsystem.FBUS,
    freechips.rocketchip.subsystem.PBUS) ++                // attach TL serial
        adapter to f/p busses
  new chipyard.WithMulticlockIncoherentBusTopology ++       // use incoherent bus
      topology
  new freechips.rocketchip.subsystem.WithNBanks(0) ++       // remove L2$
  new freechips.rocketchip.subsystem.WithNoMemPort ++       // remove backing memory
  new freechips.rocketchip.subsystem.WithMyTinyCore ++      // single modified tiny
      rocket-core (1Mb DCache)
  new chipyard.config.AbstractConfig)
```

Appendix C

Appendix: FPGA Setup

The setup, configuration, and software used to connect via the JTAG interface to the implemented System on Chip (SoC) in the Field Programmable Gate Array (FPGA) comes out of one of the SiFive developers's kits, the SiFive Freedom E310 Arty FPGA Dev Kit [9]. This developer's kit uses of the Xilinx Artix-7 35T Arty FPGA Evaluation Kit.

In order to debug the SoC using the JTAG connection at the FPGA PMOD Header, a JTAG debugger is required. The JTAG debugger used is the Olimex ARM-USB-TINY-H. This debugger will be connected using jumper cables to the FPGA, and a USB cable to the computer. The jumper cable connections between the adaptor and the FPGA can be seen in Table C-1.

The FPGA setup for the testing can be seen in Figure C-1.

Launching vivado to program the FPGA with the bitstream generated using the tools provided in the Rocket-Chip generator is also necessary.

C-1 Software Setup

In order to connect to the SoC implemented on the FPGA several software tools are required.

RISCV32 Compiler

In order to compile a complete C test and run it in the SoC as in the previous simulation section, a proper RISCV32 compiler and some setup files are required to properly build the address map and the initialization sequence of the SoC.

Unfortunately, this part of the Rocket-Chip generator environment does not work as intended yet. Although the posted issues on the tools' discussion platforms have helped fix several bugs, no solution has been found yet to this problem. Therefore, compiling RV32 code to run on the FPGA emulation hasn't been possible.

Table C-1: Debugger connections between the debugger adaptor and the FPGA board [9].

Signal Name	**ARM-USB-TINY-H Pin Number**	**Suggested Jumper Color**	**Freedom E310 Arty Dev Kit JD Pin Number**
VREF	1	red	12
VREF	2	brown	6 ("VCC")
nTRST	3	orange	2
TDI	5	yellow	7
TMS	7	green	8
TCK	9	blue	3
TDO	13	purple	1
GND	14	black	5 ("GND")
nRST	15	grey	9
GND	16	white	11

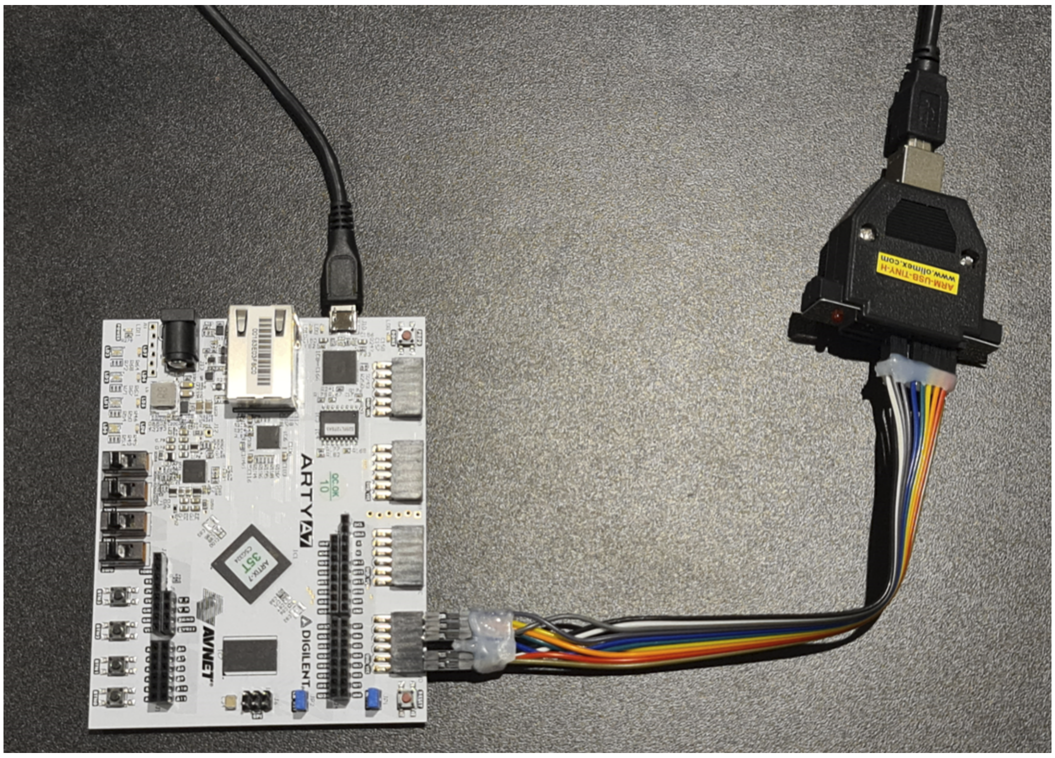


Figure C-1: FPGA emulation setup.

Instead of testing the FPGA implementation by running an executable, Open On-Chip Debugger (OpenOCD) and a debugger (GDB) have been used to manually write and read the memory addresses that are mapped to the designed baseband. The previously described step can be done as follows:

```
//set the memory address 0x2000 with the following integer value in hex 0x2000064:
(gdb) set {int}0x2000 = 0xA2000064

//read the memory address 0x2100:
(gdb) x 0x2100
//the console will reply with something similar to this:
0x2100: 0xb00fa0cc
//where 0xb00fa0cc is the data read in the memory address 0x2100
```

C-1-1 OpenOCD

The OpenOCD is an open-source software development tool that allows on-chip debugging and programming applications via JTAG/SWD hardware interface. OpenOCD runs on a host computer along with a debugger which will communicate with OpenOCD over RSP protocol, similar to debugging an application running on hardware.

In order to configure OpenOCD to work for the specific implementation in the FPGA, configuration files have been created/edited to establish the connection.

C-1-2 GDB

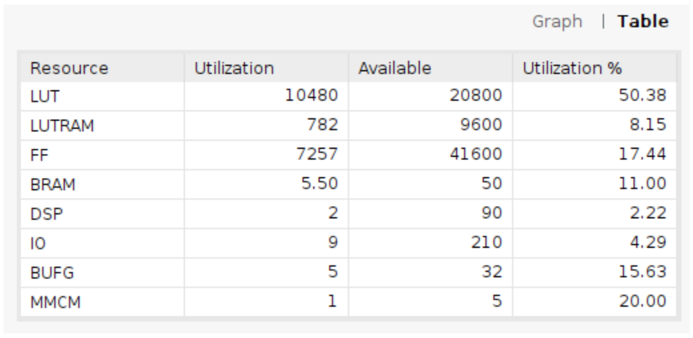
GDB is an open-source debugger that allows debugging any software at run time. In this case, it will be used to execute commands manually in the processor core.

As with the OpenOCD, a specific debugger used for RISCV executables was required and used. Several connection parameters must be configured upon connection, and an executable must be loaded to the processor. These steps can be seen below.

```
(gdb) set remotetimeout unlimited
(gdb) target extended-remote localhost:3333
(gdb) file test.riscv
(gdb) load
```

C-2 Synthesis and implementation

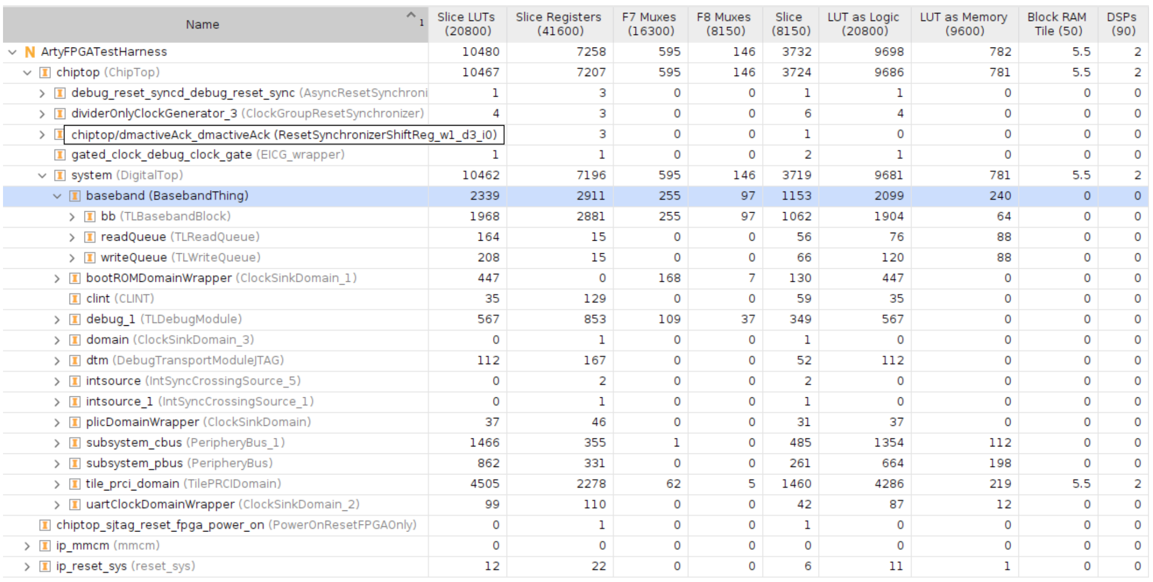
As mentioned, the Rocket-Chip generator manages all the aspects of testing the design on the FPGA. In addition, it generates intermediate points in the project implementation in case the users want to change some of the constraints or implement any debug tool like an Integrated Logic Analyzer (ILA). It also generates reports of every step of the synthesis and implementation process.



Graph | **Table**

Resource	Utilization	Available	Utilization %
LUT	10480	20800	50.38
LUTRAM	782	9600	8.15
FF	7257	41600	17.44
BRAM	5.50	50	11.00
DSP	2	90	2.22
IO	9	210	4.29
BUFG	5	32	15.63
MMCM	1	5	20.00

Figure C-2: FPGA resources utilization report.



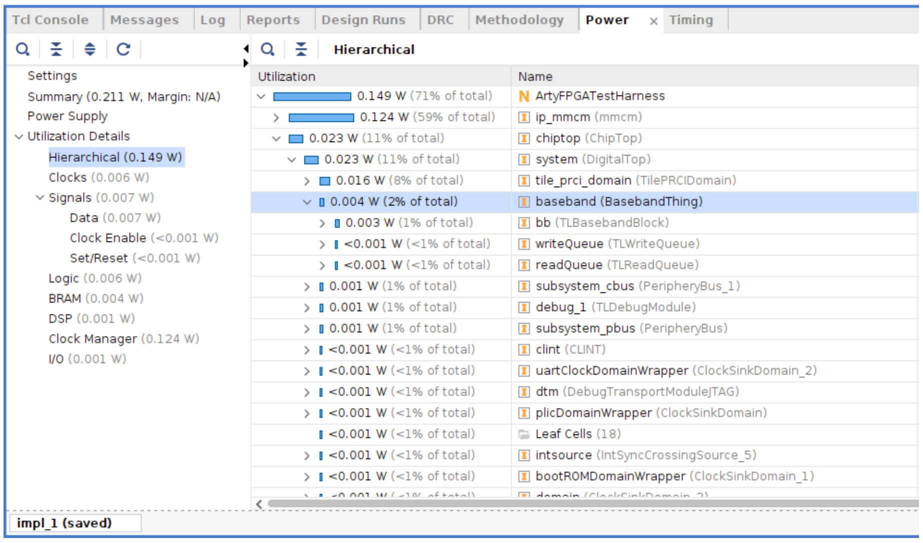
Name	Slice LUTs (20800)	Slice Registers (41600)	F7 Muxes (16300)	F8 Muxes (8150)	Slice (8150)	LUT as Logic (20800)	LUT as Memory (9600)	Block RAM Tile (50)	DSPs (90)
ArtyFPGATestHarness	10480	7258	595	146	3732	9698	782	5.5	2
chiptop (ChipTop)	10467	7207	595	146	3724	9686	781	5.5	2
debug_reset_syncd_debug_reset_sync (AsyncResetSynchroni	1	3	0	0	1	1	0	0	0
dividerOnlyClockGenerator_3 (ClockGroupResetSynchronizer)	4	3	0	0	6	4	0	0	0
chiptop/dmactiveAck_dmactiveAck (ResetSynchronizerShiftReg_w1_d3_i0)		3	0	0	1	0	0	0	0
gated_clock_debug_clock_gate (EICG_wrapper)	1	1	0	0	2	1	0	0	0
system (DigitalTop)	10462	7196	595	146	3719	9681	781	5.5	2
baseband (BasebandThing)	2339	2911	255	97	1153	2099	240	0	0
bb (TLBasebandBlock)	1968	2881	255	97	1062	1904	64	0	0
readQueue (TLReadQueue)	164	15	0	0	56	76	88	0	0
writeQueue (TLWriteQueue)	208	15	0	0	66	120	88	0	0
bootROMDomainWrapper (ClockSinkDomain_1)	447	0	168	7	130	447	0	0	0
clint (CLINT)	35	129	0	0	59	35	0	0	0
debug_1 (TLDebugModule)	567	853	109	37	349	567	0	0	0
domain (ClockSinkDomain_3)	0	1	0	0	1	0	0	0	0
dtm (DebugTransportModuleJTAG)	112	167	0	0	52	112	0	0	0
intsource (IntSyncCrossingSource_5)	0	2	0	0	2	0	0	0	0
intsource_1 (IntSyncCrossingSource_1)	0	1	0	0	1	0	0	0	0
plicDomainWrapper (ClockSinkDomain)	37	46	0	0	31	37	0	0	0
subsystem_cbus (PeripheryBus_1)	1466	355	1	0	485	1354	112	0	0
subsystem_pbus (PeripheryBus)	862	331	0	0	261	664	198	0	0
tile_prci_domain (TilePRCIDomain)	4505	2278	62	5	1460	4286	219	5.5	2
uartClockDomainWrapper (ClockSinkDomain_2)	99	110	0	0	42	87	12	0	0
chiptop_sjtag_reset_fpga_power_on (PowerOnResetFPGAOnly)	0	1	0	0	1	0	0	0	0
ip_mmcm (mmcm)	0	0	0	0	0	0	0	0	0
ip_reset_sys (reset_sys)	12	22	0	0	6	11	1	0	0

Figure C-3: FPGA resources utilization hierarchy report.

The total utilization report of the FPGA can be seen in Figure C-2. Going into the hierarchy, we can check the resource utilization of the designed baseband only (not the whole SoC) in Figure C-3.

It is also possible to check the timing report, where no violations or relevant warnings appeared. The clock frequency used for the implementation is 100 MHz, although internally in the SoC several other clock divisions are obtained. In this case, the logic and the processor core are running at 10 MHz. This frequency is much higher than required for the baseband design, which was aimed at 1 MHz. The core is capable of running at 1 GHz in Application-Specific Integrated Circuit (ASIC) implementations.

Also, a power report is generated, but the constraints were defined in the Rocket-Chip generator and are not specific to the design. The power report is shown in Figure C-4. The reported total power consumption of the design on the SoC in the FPGA is estimated to be 0.023 W. Out of that, only 0.004 W correspond to the designed baseband and First In First Out (FIFO) connections, while the rest are consumed by the SoC structure and the Rocket-Core.



Tcl Console | Messages | Log | Reports | Design Runs | DRC | Methodology | Power | Timing

Hierarchical

- Settings
- Summary (0.211 W, Margin: N/A)
- Power Supply
- Utilization Details
  - Hierarchical (0.149 W)
  - Clocks (0.006 W)
  - Signals (0.007 W)
    - Data (0.007 W)
    - Clock Enable (<0.001 W)
    - Set/Reset (<0.001 W)
  - Logic (0.006 W)
  - BRAM (0.004 W)
  - DSP (0.001 W)
  - Clock Manager (0.124 W)
  - I/O (0.001 W)

Utilization	Name
0.149 W (71% of total)	ArtyFPGATestHarness
0.124 W (59% of total)	ip_mmcm (mmcm)
0.023 W (11% of total)	chiptop (ChipTop)
0.023 W (11% of total)	system (DigitalTop)
0.016 W (8% of total)	tile_prci_domain (TilePRCIDomain)
0.004 W (2% of total)	baseband (BasebandThing)
0.003 W (1% of total)	bb (TLBasebandBlock)
<0.001 W (<1% of total)	writeQueue (TLWriteQueue)
<0.001 W (<1% of total)	readQueue (TLReadQueue)
0.001 W (1% of total)	subsystem_cbus (PeripheryBus_1)
0.001 W (1% of total)	debug_1 (TLDebugModule)
0.001 W (1% of total)	subsystem_pbus (PeripheryBus)
<0.001 W (<1% of total)	clint (CLINT)
<0.001 W (<1% of total)	uartClockDomainWrapper (ClockSinkDomain_2)
<0.001 W (<1% of total)	dtm (DebugTransportModuleJTAG)
<0.001 W (<1% of total)	plicDomainWrapper (ClockSinkDomain)
<0.001 W (<1% of total)	Leaf Cells (18)
<0.001 W (<1% of total)	intsource (IntSyncCrossingSource_5)
<0.001 W (<1% of total)	bootROMDomainWrapper (ClockSinkDomain_1)
<0.001 W (<1% of total)	domain (ClockSinkDomain_2)

impl_1 (saved)

Figure C-4: FPGA power utilization report.

Appendix D

Appendix: Simulations

This chapter contains three sections with various simulations. The first section contains the functional simulations performed to the designed baseband using chisel tests. The second section contains the simulations of the whole System on Chip (SoC) running custom C programs. Using those C programs, waveforms can be generated to check the correct functionality of every part. The last section contains the emulation of the System on Chip (SoC) of a Field Programmable Gate Array (FPGA).

All the simulations in this chapter will be performed with the same data packet and Access Address (AA) in order to facilitate checking the results. A proper test coverage has been performed with random packets of random lengths.

D-1 Chisel Tests

Chisel tests are the first ones to be performed on any design using Chisel and Rocket-Chip. These tests are performed using a specific package for Chisel. The designed baseband module is tested without the processor. Stimulus are directly applied to the inputs of the block, and outputs can be read at will using a designed testbench. It is also possible to generate waveform files of the performed simulation.

The following data packet will be used for all the tests in this section (hexadecimal format): 0x0FA0CCCE32AACC. It consists of a standard 2 Bytes Header (0x0FA0) and a 5 Bytes long Payload (0xCCCE32AACC). The processed output packet is 120 bits long and corresponds to the following, 8 bit preamble (0b01010101), 32-bit AA (0b01101011011111011001000101110001), 16-bit Header (0b1110110000010001), 40-bit Payload (0b10000111001001001011011100010110 00101001) and 24-bit Cyclic Redundancy Check (CRC) (0b01110111101000 1010100010).

D-1-1 Transmitter

First, the transmission side of the bitstream processing chain will be tested which consists of Three different transmission modes. The three implemented modes are normal Tx (tx),

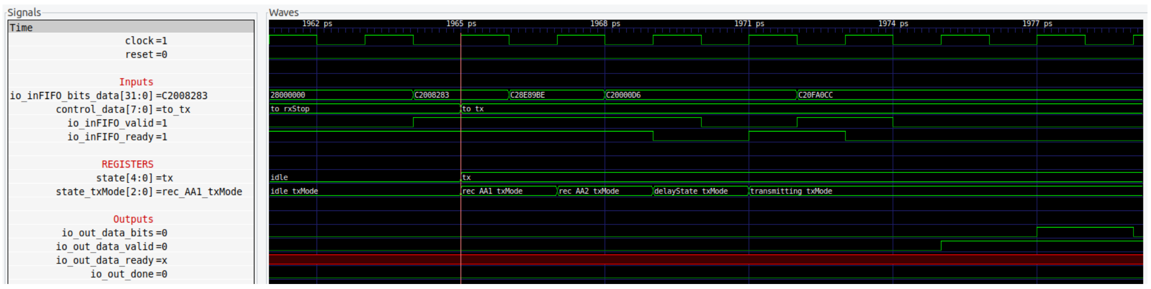


Figure D-1: Normal Tx mode of the baseband.

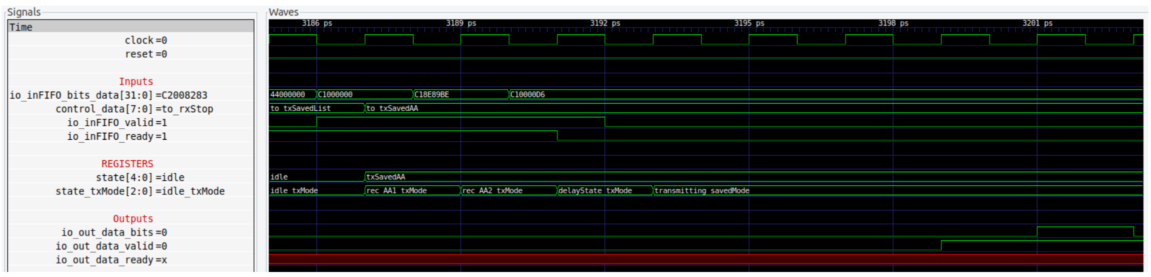


Figure D-2: Transmission of a saved packet.

transmitting a saved packet to a specific AA (txSavedAA), and broadcasting a saved packet (txSavedbroadcast).

Simple transmission

The first mode (tx) corresponds with the basic operation of the baseband, where a complete data packet is sent from the processor to the baseband and then transmitted to the Physical Layer (PHY). This mode can be seen in Figure D-1.

On it (D-1), it is possible to see how the state of the baseband changes from *idle* to *tx* when the command *to_tx* is received, and the valid signal is asserted. Immediately after that, the baseband starts reading entries from the First In First Out (FIFO) and transmitting the packet. The output data signal is *io_out_data_bits*, and the *ready* and *valid* signals that form the interface with it, there is also an *io_out_done* signal whose level becomes high when the packet is fully transmitted to indicate its end.

Saved packet transmission

The second implemented tx mode of the baseband (txSavedAA) corresponds with one of the advanced functions implemented in the design. This mode can be seen in Figure D-2.

On this mode (D-2), when the command $to_t xSavedAA$ is received, the baseband will read the desired AA from the input FIFO and transmit the stored in registers packet with that AA. It is possible to see how the packet transmission starts with only three reads from the input FIFO (*io_inFIFO_bits_data*).

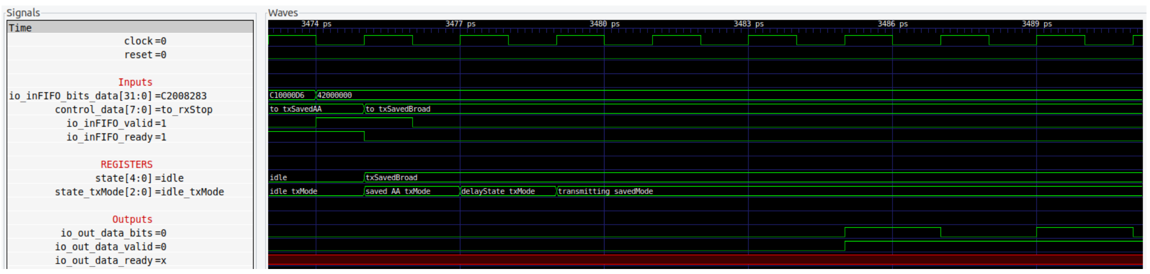


Figure D-3: Broadcast of a saved packet.

Saved packet broadcast

The third implemented tx mode of the baseband (txSavedBroadcast) corresponds with another one of the advanced functions implemented in the design. This mode can be seen in Figure D-3.

On this mode (D-3), when the command $to\_txSavedBroad$ is received, the baseband will proceed to transmit the stored in registers packet with the broadcasting AA. It is possible to see how it only needs 1 read from the input FIFO ($io\_inFIFO\_bits\_data$) to start the transmission of the packet.

D-1-2 Receiver

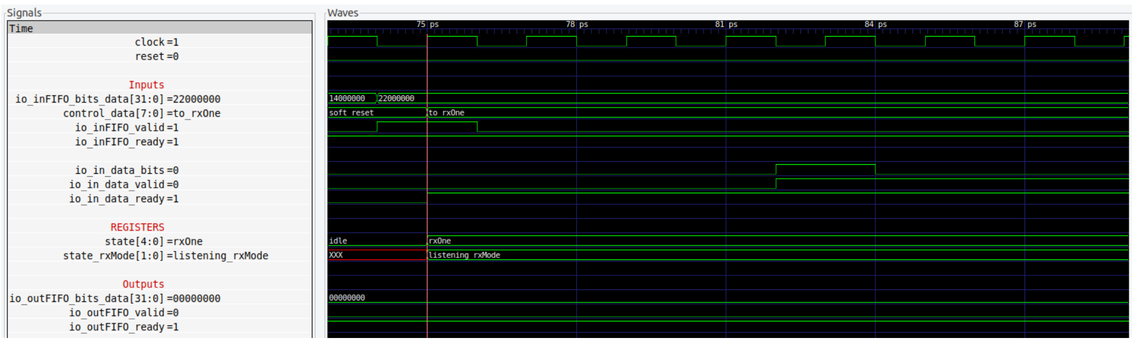
The receiver side of the bitstream processing chain will be tested in the section; two reception modes are implemented in the baseband's design. Those two modes are simple Rx (rxOne), where the baseband will be active until a packet is received, and continuous reception (rxContinuous), where the baseband will remain active until a stop command (rxStop) is received.

Simple reception

The simple reception mode (rx), corresponds with the basic rx operation of a Bluetooth Low Energy (BLE) baseband, where the receiver side becomes active until a packet is received and then becomes inactive. This mode can be seen in Figure D-4 and Figure D-5. The two figures correspond with the configuration into Rx mode for the reception of the packet, and the end of the reception where the packet is sent to the data FIFO.

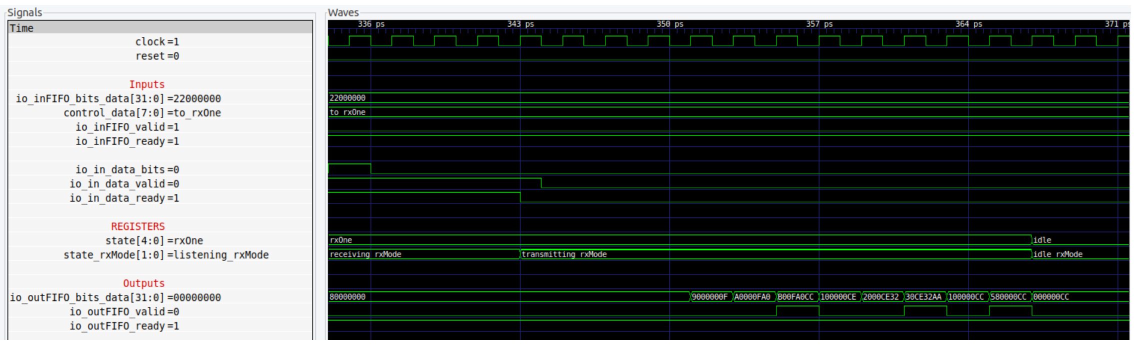
Figure D-4 shows how the state changes from $idle$ to rx when the command to_tx is received, and the valid signal is asserted. A few clock cycles after that happened, a packet starts to be received through $io\_in\_data\_bits$ as its valid signal is asserted $io\_in\_data\_valid$. A few internal signals are generated and used to control the data flow internally but aren't shown in the screenshot.

Figure D-5 shows how the processed data packet is sent into the data FIFO that connects the baseband to the processor ($io\_outFIFO\_bits\_data$). This signal is only stored in the output FIFO when the valid signal ($io\_outFIFO\_valid$) is also asserted.



```
Signals
Time
                    clock=1
                    reset=0

                   Inputs
 io_inFIFO_bits_data[31:0]=22000000
        control_data[7:0]=to_rxOne
           io_inFIFO_valid=1
           io_inFIFO_ready=1

           io_in_data_bits=0
          io_in_data_valid=0
          io_in_data_ready=1

                 REGISTERS
              state[4:0]=rxOne
   state_rxMode[1:0]=listening_rxMode

                  Outputs
io_outFIFO_bits_data[31:0]=00000000
          io_outFIFO_valid=0
          io_outFIFO_ready=1
```

Figure D-4: Start of the reception of a packet in simple Rx mode.



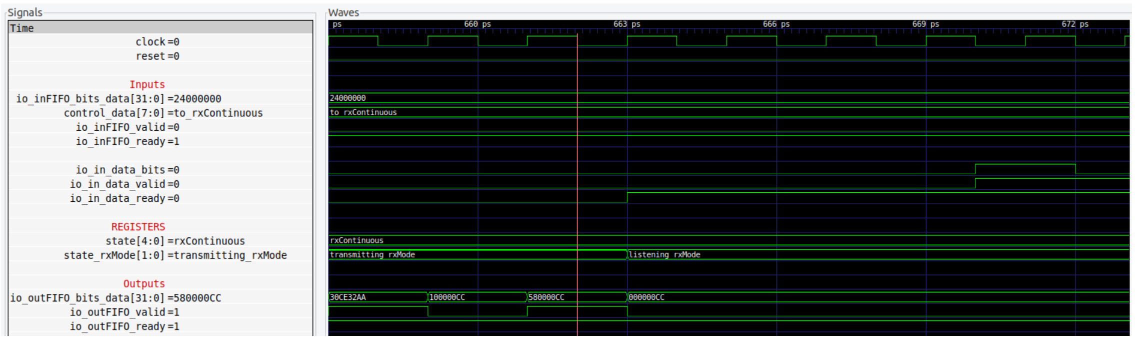
```
Signals
Time
                    clock=1
                    reset=0

                   Inputs
 io_inFIFO_bits_data[31:0]=22000000
        control_data[7:0]=to_rxOne
           io_inFIFO_valid=1
           io_inFIFO_ready=1

           io_in_data_bits=0
          io_in_data_valid=0
          io_in_data_ready=1

                 REGISTERS
              state[4:0]=rxOne
   state_rxMode[1:0]=listening_rxMode

                  Outputs
io_outFIFO_bits_data[31:0]=00000000
          io_outFIFO_valid=0
          io_outFIFO_ready=1
```

Figure D-5: End of the reception of a packet in simple Rx mode.

Continuous reception

The second implemented rx mode of the baseband (rxContinuous) corresponds with a functionality that can be useful in some scenarios, such as when performing the discovery of other BLE devices. During this mode, packets will be received and processed normally, but once a packet is received, the baseband will not exit the rx mode, and will stay on it until a stop command is received (rxStop). This functionality can be seen in Figure D-6 and Figure D-7.

In Figure D-6 it is possible to see how the *state* of the baseband controller does not go back to idle after receiving a packet. The baseband does not go out of the *rxContinuous* until a stop command is received, which can be seen in Figure D-7.



```
Signals
Time
                    clock=0
                    reset=0

                   Inputs
 io_inFIFO_bits_data[31:0]=24000000
        control_data[7:0]=to_rxContinuous
           io_inFIFO_valid=0
           io_inFIFO_ready=1

           io_in_data_bits=0
          io_in_data_valid=0
          io_in_data_ready=0

                 REGISTERS
              state[4:0]=rxContinuous
   state_rxMode[1:0]=transmitting_rxMode

                  Outputs
io_outFIFO_bits_data[31:0]=580000CC
          io_outFIFO_valid=1
          io_outFIFO_ready=1
```

Figure D-6: End of reception in rxContinuous mode.

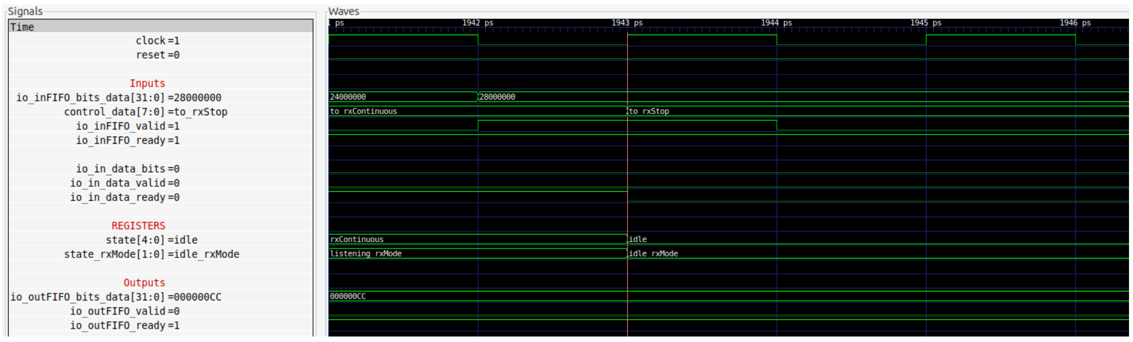


Figure D-7: Exiting rxContinuous mode.

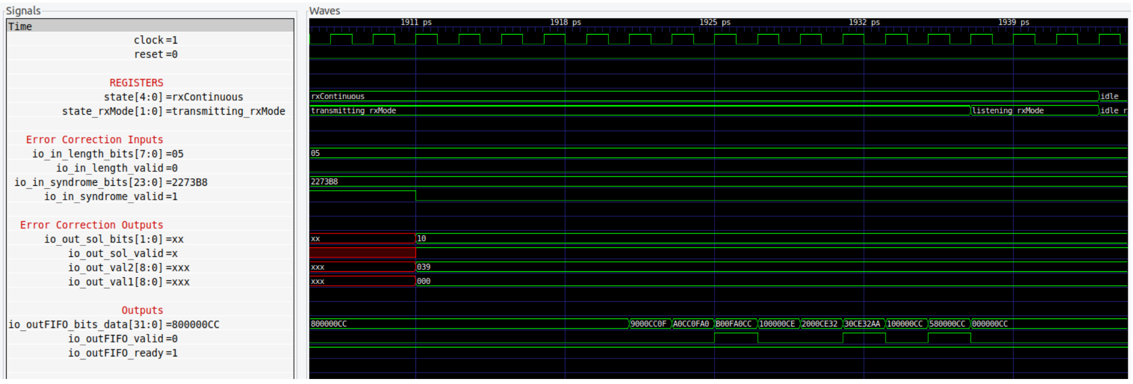


Figure D-8: Single bit error correction SoC.

D-1-3 Error Correction

In the simulations shown in this section, the error correction block is configured to correct packets of a maximum of 29 Bytes in length and to correct both single and double bit errors.

Single bit error correction

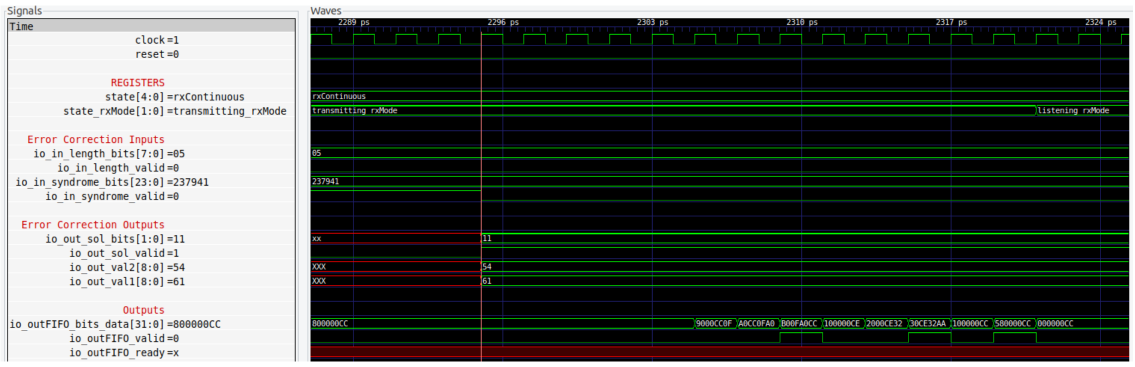
First, the single bit error correction capabilities are tested; an erroneous packet is sent on purpose to the receiver, and the packet contains only a single bit error.

After some delay required for the correction, which can be seen in Figure D-8, the signal *io_out_sol_valid* becomes 1, indicating that the error correction block has finished trying to correct the error. Finally, the output *io_out_sol_bits* value is 0b10 indicating that the packet only had an error and that its position is in the io_out_val2 signal.

A few cycles after the correction finishes, the baseband block outputs the corrected packet to the data FIFOs connecting the baseband with the processor.

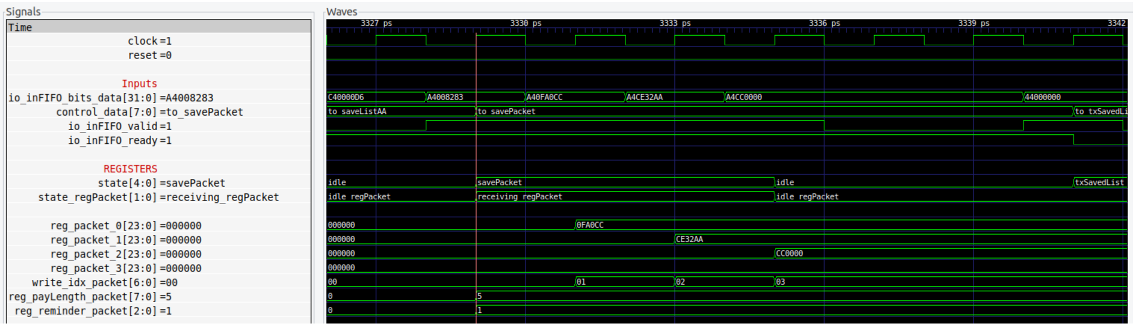
Double bit error correction

In this section, the double bit error correction capabilities are tested; an erroneous packet containing two single bit errors is sent on purpose to the receiver. The simulation of this case can be seen in Figure D-9.



```
Signals
Time
                    clock=1
                    reset=0

                 REGISTERS
            state[4:0]=rxContinuous
   state_rxMode[1:0]=transmitting_rxMode

       Error Correction Inputs
     io_in_length_bits[7:0]=05
        io_in_length_valid=0
  io_in_syndrome_bits[23:0]=237941
       io_in_syndrome_valid=0

      Error Correction Outputs
       io_out_sol_bits[1:0]=11
          io_out_sol_valid=1
          io_out_val2[8:0]=54
          io_out_val1[8:0]=61

                  Outputs
io_outFIFO_bits_data[31:0]=800000CC
          io_outFIFO_valid=0
          io_outFIFO_ready=x
```

Figure D-9: Double bit error correction.



```
Signals
Time
                    clock=1
                    reset=0

                   Inputs
io_inFIFO_bits_data[31:0]=A4008283
      control_data[7:0]=to_savePacket
          io_inFIFO_valid=1
          io_inFIFO_ready=1

                 REGISTERS
            state[4:0]=savePacket
  state_regPacket[1:0]=receiving_regPacket

     reg_packet_0[23:0]=000000
     reg_packet_1[23:0]=000000
     reg_packet_2[23:0]=000000
     reg_packet_3[23:0]=000000
   write_idx_packet[6:0]=00
reg_payLength_packet[7:0]=5
 reg_reminder_packet[2:0]=1
```

Figure D-10: Saving data packet in the baseband registers.

As for the single bit error correction, when the signal *io_out_sol_valid* becomes 1 it is possible to check if the block was able to correct the packet. In this case, the output *io_out_sol_bits* value is 0b11, indicating that the two values *io_out_val*2 and *io_out_val*1 have to be applied to correct the packet.

A few cycles after the correction finishes, the baseband block outputs the corrected packet to the data FIFOs connecting the baseband with the processor.

D-1-4 Other Functionalities

Saving a data packet

One of the implemented functions shown before, broadcasting or transmitting a saved packet, requires first saving it on the baseband. This process can be seen in Figure D-10.

It is possible to see how the baseband is configured (to_savePacket) to receive the packet, and then it's saved during the following clock cycles. The stored packet can be seen on signals *ref_packet*_0, *ref_packet*_1, and *ref_packet*_2.

Reading the state of the baseband

It is possible to read the value of several parameters and registers from the baseband using the command to_sendStatus The output of this command is the channel being used now,

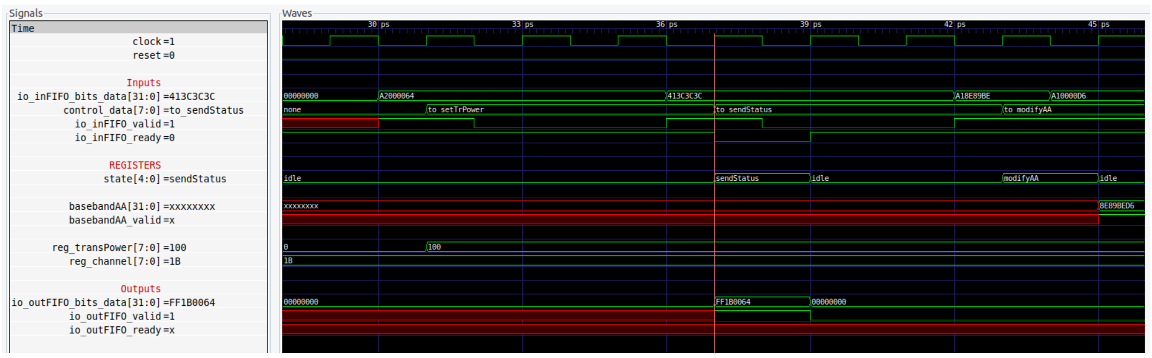


Figure D-11: Several baseband commands.

the last measured Received Signal Strength Indicator (RSSI), and the selected transmission power.

The command and its output through the output data FIFO can be seen in Figure D-11.

Modifying the AA of the baseband

The command to change the value of the AA used to receive transmitted packets can be seen in Figure D-11.

It is possible to see how after the command to_modifyAA the register *basebandAA* changes its value to the received one.

D-2 Simulation of the SoC with Verilator

Verilator is a cycle accurate open source simulator implemented as part of the tools in Chipyard and the Rocket-Chip generator. The Chipyard environment is used to run C programs compiled to binary executables on the RISC-V core. It can be used with a console to print data or in a debug mode to generate waveforms. This section will use the second approach to show that the SoC implementation works.

As mentioned, executing a test bench with a series of stimuli for the baseband inputs and outputs is now not possible. However, a binary executable is run on the processor, and the generated waveforms are inspected. A piece of the C code used to generate the executable can be seen below. On it, a packet is transmitted, and then a packet is received and read.

```
//_______________
//STATE: tx
//---------------
reg_write32(BASEBAND_FIFO_WRITE, (command12 << 24)|txLengths);
reg_write32(BASEBAND_FIFO_WRITE, (command12 << 24)|accessAddress1);
reg_write32(BASEBAND_FIFO_WRITE, (command12 << 24)|accessAddress2);
reg_write32(BASEBAND_FIFO_WRITE, (command12 << 24)|inPacket1);
reg_write32(BASEBAND_FIFO_WRITE, (command12 << 24)|inPacket2);
reg_write32(BASEBAND_FIFO_WRITE, (command12 << 24)|inPacket3);
```

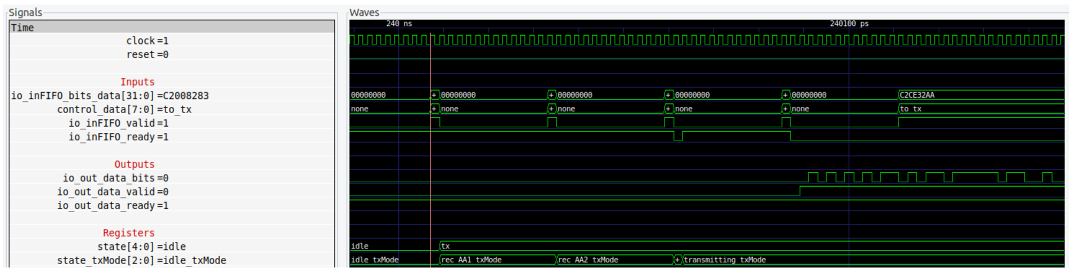

```
Signals
Time
                clock=1
                reset=0

               Inputs
io_inFIFO_bits_data[31:0]=C2008283
        control_data[7:0]=to_tx
           io_inFIFO_valid=1
           io_inFIFO_ready=1

              Outputs
         io_out_data_bits=0
        io_out_data_valid=0
        io_out_data_ready=1

             Registers
              state[4:0]=idle
      state_txMode[2:0]=idle_txMode
```

Figure D-12: Tx mode SoC simulation.

```
//_______________
//STATE: rxOne
//---------------
reg_write32(BASEBAND_FIFO_WRITE, (command01 << 24));
FIFO_OUT = reg_read32(BASEBAND_FIFO_READ);
//printf("received data: %x\n", FIFO_OUT & 0x00FFFFFFU);
FIFO_OUT = reg_read32(BASEBAND_FIFO_READ);
//printf("received data: %x\n", FIFO_OUT & 0x00FFFFFFU);
FIFO_OUT = reg_read32(BASEBAND_FIFO_READ);
//printf("received data: %x\n", FIFO_OUT & 0x00FFFFFFU);
```

In the following simulations the implemented modes will be shown again. The main difference between the simulations shown in this chapter and the previous one resides in the fact that now the data is sent and received directly by the processor. Therefore it is necessary to check if the handshake used to read and write data is well implemented and if any other issue is present.

D-2-1 Transmission

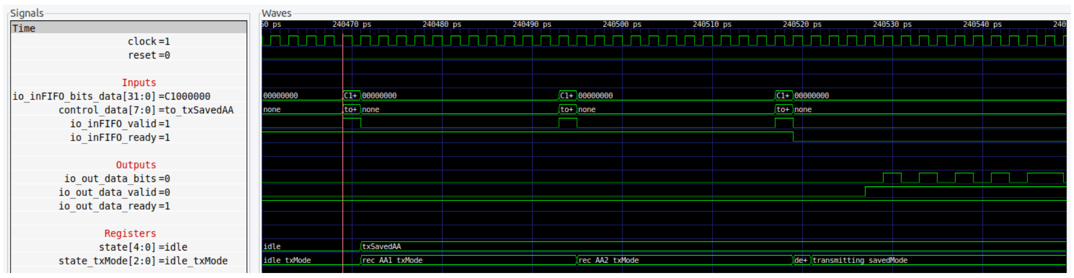
Simple transmission

In Figure D-12 it is possible to see how the normal transmission mode (tx). On it its possible to see how the signal *io_inFIFO_ready* makes backpressure to the data FIFO. The *io_inFIFO_ready* signal value is only 1 when the baseband is ready to process the following piece of data. This can be seen at the end of the simulation where the FIFO is waiting, as the signal *io_inFIFO_valid* is 1 and the data input *io_inFIFO_data* is constant but can not proceed until some clock cycles after when the baseband will assert the signal *io_inFIFO_ready*.

It is also possible to see in the same figure the start of the transmission of the baseband, corresponding with signals *io_out_data_bits* and *io_out_data_valid*.

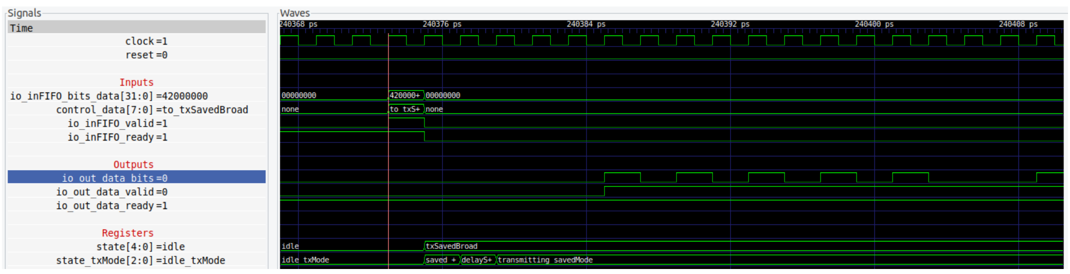
Saved packet transmission

Figure D-13 shows the txSavedAA transmission mode, where the saved packet in the baseband is transmitted to a given AA.



```
Signals
Time
                 clock=1
                 reset=0

                 Inputs
io_inFIFO_bits_data[31:0]=C1000000
       control_data[7:0]=to_txSavedAA
         io_inFIFO_valid=1
         io_inFIFO_ready=1

                 Outputs
         io_out_data_bits=0
        io_out_data_valid=0
        io_out_data_ready=1

                 Registers
             state[4:0]=idle
      state_txMode[2:0]=idle_txMode
```

Figure D-13: Transmission of a saved packet SoC simulation.



```
Signals
Time
                 clock=1
                 reset=0

                 Inputs
io_inFIFO_bits_data[31:0]=42000000
       control_data[7:0]=to_txSavedBroad
         io_inFIFO_valid=1
         io_inFIFO_ready=1

                 Outputs
         io_out_data_bits=0
        io_out_data_valid=0
        io_out_data_ready=1

                 Registers
             state[4:0]=idle
      state_txMode[2:0]=idle_txMode
```

Figure D-14: Broadcast of a saved packet SoC simulation.

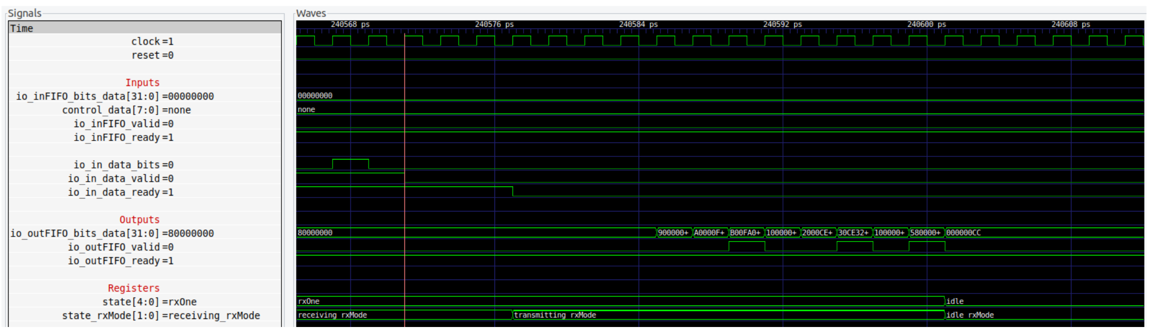
Saved packet broadcast

Figure D-13 shows the txSavedBroadcast transmission mode, where the saved packet in the baseband is broadcasted.

D-2-2 Reception

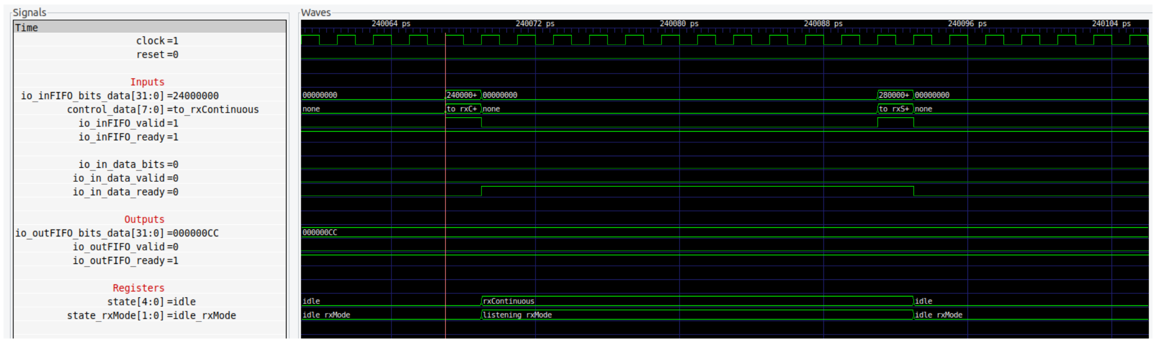
Simple reception

Figure D-15 shows how the reception mode (rx) interfaces with the data FIFO to send the received packet to the processor.



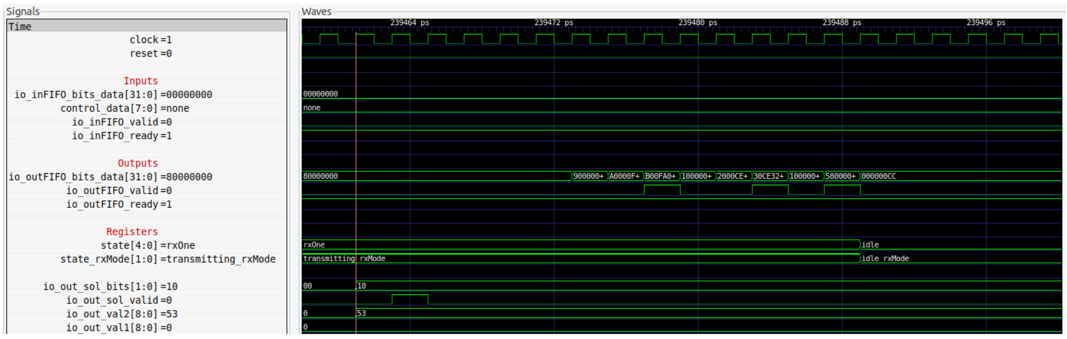
```
Signals
Time
                 clock=1
                 reset=0

                 Inputs
io_inFIFO_bits_data[31:0]=00000000
       control_data[7:0]=none
         io_inFIFO_valid=0
         io_inFIFO_ready=1

          io_in_data_bits=0
         io_in_data_valid=0
         io_in_data_ready=1

                 Outputs
io_outFIFO_bits_data[31:0]=80000000
         io_outFIFO_valid=0
         io_outFIFO_ready=1

                 Registers
             state[4:0]=rxOne
    state_rxMode[1:0]=receiving_rxMode
```

Figure D-15: Rx mode SoC simulation.



```
Signals
Time
                      clock=1
                      reset=0

                      Inputs
io_inFIFO_bits_data[31:0]=24000000
    control_data[7:0]=to_rxContinuous
         io_inFIFO_valid=1
         io_inFIFO_ready=1

         io_in_data_bits=0
        io_in_data_valid=0
        io_in_data_ready=0

                     Outputs
io_outFIFO_bits_data[31:0]=000000CC
        io_outFIFO_valid=0
        io_outFIFO_ready=1

                    Registers
              state[4:0]=idle
    state_rxMode[1:0]=idle_rxMode
```

Figure D-16: Rx continuous mode and rxStop command SoC simulation.



```
Signals
Time
                      clock=1
                      reset=0

                      Inputs
io_inFIFO_bits_data[31:0]=00000000
      control_data[7:0]=none
         io_inFIFO_valid=0
         io_inFIFO_ready=1

                     Outputs
io_outFIFO_bits_data[31:0]=80000000
        io_outFIFO_valid=0
        io_outFIFO_ready=1

                    Registers
              state[4:0]=rxOne
   state_rxMode[1:0]=transmitting_rxMode

     io_out_sol_bits[1:0]=10
        io_out_sol_valid=0
       io_out_val2[8:0]=53
       io_out_val1[8:0]=0
```

Figure D-17: Single bit error correction SoC simulation.

Continuous reception

Figure D-16 shows how the baseband is configured in the rxContinuous mode and then back to idle with the rxStop command.

D-2-3 Error Correction

In order to test the error correction capabilities, it is necessary to modify the FIFO to connect the output to the antenna and the input from the antenna. The modified block will introduce errors in specified positions of the packet.

In Figure D-17 it is possible to see how the error correction block can correct a packet and the baseband sends it to the output data FIFO connecting with the processor.

D-2-4 Other Functionalities

Saving a data packet

Figure D-18 shows how the processor sends a packet to be stored on the baseband registers and how they change its value.

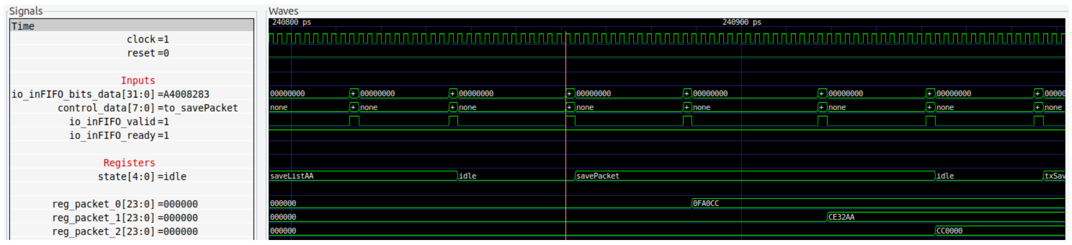


Figure D-18: Saving a data packet SoC simulation.

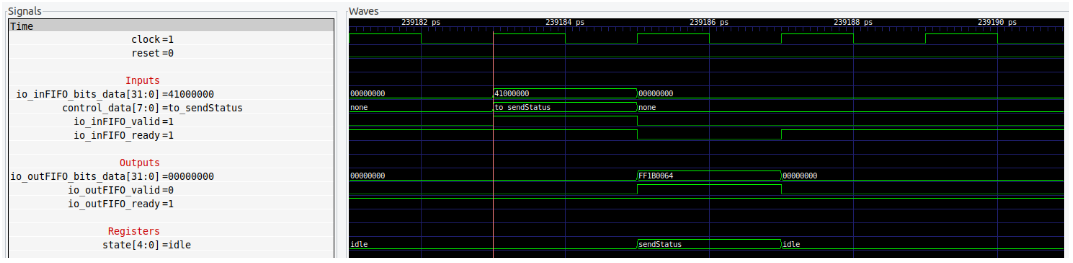


Figure D-19: Reading status SoC simulation.

Reading the state of the baseband

In Figure D-19 it is possible to see how the status of the baseband is requested and sent to the processor.

Modifying the AA of the baseband

In Figure D-20 it is possible to see how the AA used to perform the check when receiving packets is modified.

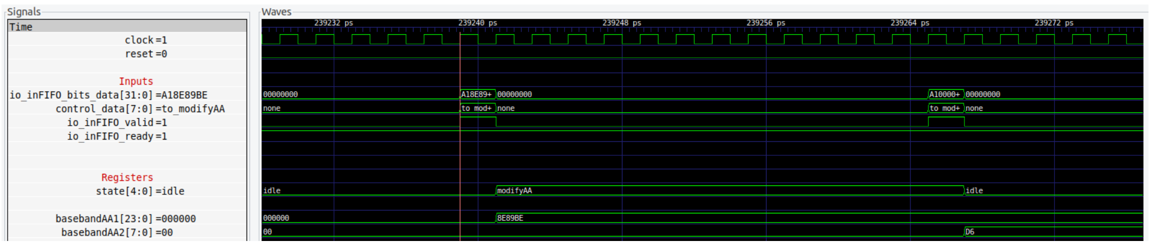


Figure D-20: Modifying the AA of the baseband SoC simulation.

Bibliography

[1] J. Tosi, F. Taffoni, M. Santacatterina, R. Sannino, and D. Formica, "Performance evaluation of bluetooth low energy: A systematic review," *Sensors*, vol. 17, no. 12, p. 2898, 2017.

[2] M. Hughes, "Bit paths behind the new ble standard." `https://www.allaboutcircuits.com/technical-articles/long-distance-bluetooth-low-energy-bit-data-paths/`. Accessed: 2022-07-31.

[3] S. I. G. Bluetooth, "Bluetooth core specification 5.0," *Specification of the Bluetooth System*, 2017.

[4] P. Wiecha, M. Cieplucha, P. Kloczko, and W. A. Pleskacz, "Architecture and design of a bluetooth low energy controller," in *2016 MIXDES-23rd International Conference Mixed Design of Integrated Circuits and Systems*, pp. 164–167, IEEE, 2016.

[5] I.-J. Chun, B.-G. Kim, and I.-C. Park, "A fully synthesizable bluetooth baseband module for a system-on-a-chip," *ETRI journal*, vol. 25, no. 5, pp. 328–336, 2003.

[6] Y. Yun, W. Danghui, Y. Ke, and F. Zhihua, "Design of link layer controller for high speed serial bus," in *The 2nd International Conference on Information Science and Engineering*, pp. 1997–2000, IEEE, 2010.

[7] V. Boussard, S. Coulombe, F.-X. Coudoux, and P. Corlay, "Table-free multiple bit-error correction using the crc syndrome," *IEEE Access*, vol. 8, pp. 102357–102372, 2020.

[8] K. Asanović, R. Avizienis, J. Bachrach, S. Beamer, D. Biancolin, C. Celio, H. Cook, D. Dabbelt, J. Hauser, A. Izraelevitz, S. Karandikar, B. Keller, D. Kim, J. Koenig, Y. Lee, E. Love, M. Maas, A. Magyar, H. Mao, M. Moreto, A. Ou, D. A. Patterson, B. Richards, C. Schmidt, S. Twigg, H. Vo, and A. Waterman, "The rocket chip generator," Tech. Rep. UCB/EECS-2016-17, EECS Department, University of California, Berkeley, Apr 2016.

[9] S. Inc, "Sifive freedom e310 arty fpga dev kit getting started guide." `https://static.dev.sifive.com/SiFive-E310-arty-gettingstarted-v1.0.6.pdf`. Accessed: 2022-09-10.

[10] E. Garcia-Espinosa, O. Longoria-Gandara, I. Pegueros-Lepe, and A. Veloz-Guerrero, “Power consumption analysis of bluetooth low energy commercial products and their implications for iot applications,” *Electronics*, vol. 7, no. 12, p. 386, 2018.

[11] S. Madakam, V. Lake, V. Lake, V. Lake, *et al.*, “Internet of things (iot): A literature review,” *Journal of Computer and Communications*, vol. 3, no. 05, p. 164, 2015.

[12] R. Aggarwal and M. L. Das, “Rfid security in the context of" internet of things",” in *Proceedings of the First International Conference on Security of Internet of Things*, pp. 51–56, 2012.

[13] T. P. F. e Fizardo, “Wibree: wireless communication technology,” in *Fourth International Conference on Machine Vision (ICMV 2011): Computer Vision and Image Analysis; Pattern Recognition and Basic Technologies*, vol. 8350, pp. 507–511, SPIE, 2012.

[14] S. I. G. Bluetooth, “Bluetooth core specification 4.0,” *Specification of the Bluetooth System*, 2010.

[15] M. Andersson, “Use case possibilities with bluetooth low energy in iot applications,” *White Paper*, vol. 2, 2014.

[16] “Implementing data whitening and crc verification in software in kinetis kw01 microcontrollers.” https://www.nxp.com/docs/en/application-note/AN5070.pdf/. Accessed: 2022-08-03.

[17] Y. S. Han and P.-N. Chen, “Sequential decoding of convolutional codes,” 2002.

[18] B. Chandavarkar, A. Byju, and E. Thomas, “An improved and reliable sequential decoding of convolution codes,” in *2020 11th International Conference on Computing, Communication and Networking Technologies (ICCCNT)*, pp. 1–7, IEEE, 2020.

[19] K. Gupta, P. Ghosh, R. Piplia, and A. Dey, “A comparative study of viterbi and fano decoding algorithm for convolution codes,” in *AIP Conference Proceedings*, vol. 1324, pp. 34–38, American Institute of Physics, 2010.

[20] T. Instruments, “Cc2640r2l product details.” https://www.ti.com/product/CC2640R2L. Accessed: 2022-08-07.

[21] N. Semiconductors, “nrf52805 product specification.” https://www.nordicsemi.com/Products/nRF52805. Accessed: 2022-08-07.

[22] Microchip, “Is1870 product details.” https://www.microchip.com/en-us/product/IS1870#. Accessed: 2022-08-07.

[23] K. E. Jeon, J. She, P. Soonsawad, and P. C. Ng, “Ble beacons for internet of things applications: Survey, challenges, and opportunities,” *IEEE Internet of Things Journal*, vol. 5, no. 2, pp. 811–828, 2018.

[24] E. Tsimbalo, X. Fafoutis, and R. Piechocki, “Fix it, don’t bin it!-crc error correction in bluetooth low energy,” in *2015 IEEE 2nd World Forum on Internet of Things (WF-IoT)*, pp. 286–290, IEEE, 2015.

[25] M. Spörk, J. Classen, C. A. Boano, M. Hollick, and K. Römer, "Improving the reliability of bluetooth low energy connections.," in *EWSN*, pp. 144–155, 2020.

[26] M. Spörk, C. A. Boano, and K. Römer, "Improving the timeliness of bluetooth low energy in dynamic rf environments," *ACM Transactions on Internet of Things*, vol. 1, no. 2, pp. 1–32, 2020.

[27] S. Shukla and N. W. Bergmann, "Single bit error correction implementation in crc-16 on fpga," in *Proceedings. 2004 IEEE International Conference on Field-Programmable Technology (IEEE Cat. No. 04EX921)*, pp. 319–322, IEEE, 2004.

[28] V. Boussard, S. Coulombe, F.-X. Coudoux, and P. Corlay, "Crc-based correction of multiple errors using an optimized lookup table," *IEEE Access*, vol. 10, pp. 23931–23947, 2022.

[29] D. A. N. Gookyi and K. Ryoo, "Selecting a synthesizable risc-v processor core for low-cost hardware devices," *Journal of Information Processing Systems*, vol. 15, no. 6, pp. 1406–1421, 2019.

[30] J. Bachrach, H. Vo, B. Richards, Y. Lee, A. Waterman, R. Avižienis, J. Wawrzynek, and K. Asanović, "Chisel: Constructing hardware in a scala embedded language," in *DAC Design Automation Conference 2012*, pp. 1212–1221, 2012.

[31] "Chisel/firrtl hardware compiler framework." `https://www.chisel-lang.org/`. Accessed: 2022-07-26.

[32] M. e. a. Odersky, "The scala programming language." `http://www.scala-lang.org/`. Accessed: 2022-07-26.

[33] "Instruction set architecture (isa)." `https://www.arm.com/glossary/isa/`. Accessed: 2022-07-27.

[34] A. Amid, D. Biancolin, A. Gonzalez, D. Grubb, S. Karandikar, H. Liew, A. Magyar, H. Mao, A. Ou, N. Pemberton, P. Rigge, C. Schmidt, J. Wright, J. Zhao, Y. S. Shao, K. Asanović, and B. Nikolić, "Chipyard: Integrated design, simulation, and implementation framework for custom socs," *IEEE Micro*, vol. 40, no. 4, pp. 10–21, 2020.

[35] "Chipyard documentation." `https://chipyard.readthedocs.io/en/stable/`. Accessed: 2022-09-07.

Glossary

List of Acronyms

AA	Access Address
AFH	Adaptive Frequency-Hoping
AGC	Automatic Gain Control
ASIC	Application-Specific Integrated Circuit
BAR	Berkeley Architecture Research
BHT	Branch History Table
BLE	Bluetooth Low Energy
BR	Basic Rate
BTB	Branch Target Buffer
CAD	Computer Aided Design
Chisel	Constructing Hardware in a Scala Embedded Language
CI	Coding Indicator
CPU	Central Processing Unit
CRC	Cyclic Redundancy Check
EDR	Enhanced Data Rate
FEC	Forward Error Correction
FHA	Frequency Hoping Algorithm
FIFO	First In First Out
FPGA	Field Programmable Gate Array
FSM	Finite States Machine
GFSK	Gaussian Frequency Shift Key
HCI	Host Controller Interface
HDL	Hardware Description Language
FIRRTL	Flexible Intermediate Representation for RTL

I/O	Input/Output
ILA	Integrated Logic Analyzer
IoT	Internet of Things
IP	Intelectual Property
ISA	Instruction Set Architecture
ISM	Industrial, Scientific and Medical
K	Constraint Length
LE1M	Low Energy 1 Msym/s
LE2M	Low Energy 2 Msym/s
LEC	Low Energy Coded
LFSR	Linear Feedback Shift Register
LL	Link Layer
LSB	Least Significant Bit
M	Microchip
MMIO	Memory Mapped Input/Output
MMU	Memory Management Unit
MSB	Most Significant Bit
NFC	Near Field Communication
NS	Nordic Semiconductors
OpenOCD	Open On-Chip Debugger
PDU	Protocol Data Unit
PHY	Physical Layer
RAS	Return Address Stack
RF	Radio Frequency
RFID	Radio Frequency Identification
RISC	Reduced Instruction Set Computer
RoCC	Rocket Custom Coprocessor Interface
RSSI	Received Signal Strength Indicator
RTL	Register Transfer Level
S	Coding Scheme
SCR	Single Candidate Ratio
SIG	Special Interest Group
SoC	System on Chip
TI	Texas Instruments
TLB	Translation Lookaside Buffer
UCB	University of California Berkeley

VHDL	VHSIC Hardware Description Language
VLSI	Very Large Scale Integration