Politecnico di Milano

Dipartimento di Elettronica, Informazione e Bioingegneria
Master of Science in Electronics Engineering

Novel GNSS Sensor Processing Architecture for a Space SoC based on Arm Cortex-R52

Author:
Nuno Bernardes Barcellos

University Supervisors:
Prof. Dr. William Fornaciari
Dr. Federico Terraneo

Company Supervisor:
Engr. Isaac Tejerina

In collaboration with Airbus Defence & Space, Spacecraft Equipment.
Hardware & Logic Engineering Department.

AIRBUS

July, 2020

Abstract

The lack of microprocessor resources is currently preventing GNSS receivers targeted to space applications from performing high-level tasks such as Precise Orbit Determination post-processing on-board, thus limiting the achievable accuracy in real-time to about 1 m Position, Velocity, and Time. Whereas this is sufficient for most missions, future applications will require better precision. New receivers also have to fulfill the needs of increased PRN code length of the modernized GNSS signals and more satellites in line-of-sight due to upcoming GNSS constellations like Galileo. Current state-of-the-art receivers, such as the AGGA-4, are overloaded by the processing of the channels as part of code and carrier loops, in order to keep the incoming signal locked. In this thesis, I present a GNSS sensor processing architecture for the novel DAHLIA, a very high-performance microprocessor System-on-Chip with four Arm Cortex-R52 cores and an embedded FPGA, as a feasibility study of using this system for performing GNSS sensor processing. I exploit the Cortex-R52 Low-latency Peripheral Port for interfacing the GNSS hardware signal processing module as a way of isolating the high-frequency channel processing task from the main system bus, thus not affecting nor being affected by the shareability implications of the main interconnect. I propose a memory layout to allow run-time critical software to be placed on fast memories, the Cortex-R52 Tight Coupled Memories, in order to increase algorithm determinism and performance. An implementation on an FPGA-based prototype is performed and the results extended to the ASIC point out to the NG-Ultra being able to track 72 channels using less than 10% of CPU processing power, leaving room for higher-level tasks such as Navigation and Precise Orbit Determination to be handled by a single CPU core.

Keywords: GNSS Receiver, Space SoC, Arm Cortex-R52, DAHLIA, Sensor Processing.

Sommario

La scarsità di risorse computazionali è attualmente un fattore limitante nei ricevitori GNSS destinati ad applicazioni spaziali, specie per attività quali il post-processing e il Precise Orbit Determination direttamente a bordo dello sistema, limitando così l'accuratezza raggiungibile (Posizione, Velocità e Tempo) in tempo reale a circa 1 m. Nonostante questo sia sufficiente per la maggior parte delle missioni, le applicazioni future richiederanno più precisione. I nuovi ricevitori devono anche soddisfare le esigenze dei PRN più lunghi dei segnali GNSS modernizzati e supportare più satelliti in line-of-sight resi disponibili dalle nuove costellazioni GNSS come Galileo. I ricevitori all'avanguardia, come l'AGGA-4, sono sovraccaricati dal processamento dei canali come parte dei code e carrier loop, al fine di tenere in traccia del segnale in ingresso. In questo lavoro, presento un'architettura di processamento di segnali GNSS per DAHLIA, un System-on-Chip ad alte prestazioni con quattro core Arm Cortex-R52 e un embedded FPGA, come studio di fattibilità sull'uso di questo sistema per eseguire processamento di segnali GNSS. Utillizo la Cortex-R52 Low-latency Peripheral Port per interfacciare il modulo hardware di elaborazione del segnale GNSS come un modo per isolare l'intensa attività di processamento dei canali dal bus del sistema principale, quindi non influenzando né essendo influenzato dal nondeterminismo temporale della main interconnect. Propongo un layout di memoria che permette software time critical di essere allocati su memorie veloci, le Cortex-R52 Tight Coupled Memories, al fine di aumentare il determinismo e le prestazioni degli algoritmi. Viene eseguita un'implementazione su un prototipo basato su FPGA e i risultati estesi all'ASIC indicano che NG-Ultra è in grado di tracciare 72 canali utilizzando meno del 10% della potenza della CPU, lasciando spazio per altre attività come Navigation e Precise Orbit Determination ad essere gestite da un singolo CPU core.

Acknowledgments

I would like to express my sincerest gratitude to Prof. Dr. William Fornaciari and Dr. Federico Terraneo for their words of advice through each stage of this project, encouraging me to explore new ideas.

I would also like to extend my thanks to my company supervisor Isaac Tejerina and Max Ghiglione for providing guidance, support, and feedback during my stay at Airbus Defence & Space. I must also thank Yanitsa Stoyanova and Antoine Lasserre for their a great amount of assistance, and Tim Helfers for giving me the opportunity to carry out my research project in the Hardware & Logic Engineering Department.

My sincere thanks to Airbus Defence & Space for funding this work and to Politecnico di Milano and University of São Paulo whose partnership made this possible.

Finally, I would like to thank my family and friends who have always stood by me in the course of my studies.

Contents

List of Figures

List of Tables

List of Acronyms

AGGA-4	Advanced GPS/Galileo ASIC version 4
AHB	Advanced High-performance Bus
AMBA	Advanced Microcontroller Bus Architecture
APB	Advanced Peripheral Bus
ASIC	Application-Specific Integrated Circuit
ASSP	Application-Specific Standard Product
AXI	Advanced eXtensible Interface
CPU	Central Process Unit
DAHLIA	Deep sub-micron microprocessor for spAce rad-Hard appLIcation Asic
DLL	Delay-Locked Loop
DMA	Direct Memory Access
DSU	Debug Support Unit
eFPGA	embedded Field-Programmable Gate Array
eRAM	embedded Random-Access Memory
FDSOI	Fully Depleted Silicon On Insulator
FPGA	Field-Programmable Gate Array
GCC	GNU Compiler Collection
GIC	Generic Interrupt Controller - GNSS Interrupt Controller
GLONASS	GLObal'naya NAvigatsionnaya Sputnikovaya Sistema
GNSS	Global Navigation Satellite System
GPS	Global Positioning System
HDL	Hardware Description Language
IP	Intellectual Property
JTAG	Joint Test Action Group
LLPP	Low-Latency Peripheral Port
LUT	Lookup Table
NCO	Numeric-Controlled-Oscillators

PLL	Phase-Locked Loop
PMU	Performance Monitor Unit
POD	Precise Orbit Determination
PRN	Pseudorandom Noise
PVT	Position, Velocity, and Time
RAM	Random-Access Memory
RTL	Register-Transfer Level
SIMD	Single Instruction Multiple Data
SoC	System-on-Chip
SPI	Serial Peripheral Interface
TCM	Tight Coupled Memory
UART	Universal Asynchronous Receiver-Transmitter

1 Introduction

As space is becoming more affordable and accessible, new space missions and applications have been emerging with increasingly stringent requirements being imposed upon GNSS receivers. These devices are widely used for real-time spacecraft navigation, timing, Precise Orbit Determination (POD), and scientific observations. For the time being, GNSS receivers for space applications achieve about 1 m Position, Velocity, and Time (PVT) accuracy in real-time. Whereas this is sufficient for most missions, some future applications require even better precision, thus requiring e.g. on-board POD post-processing. The current state of the art method is to perform post-processing on-ground using precise ephemeris to reach a level of accuracy on the order of 10 cm. The lack of microprocessor resources is the major factor preventing current GNSS receivers from performing on-board POD post-processing in near real-time. Other reasons include the lack of real-time precise ephemeris and clock correction.

The current GNSS space receiver designed by Airbus Defence & Space and the Europen Space Agency (ESA), AGGA-4 [1], includes a single-core LEON-2 FT processor that is significantly loaded by the processing of the GNSS channels so that other high-level tasks such as on-board POD cannot be performed since they would have limited microprocessor resources allocated. Each channel encapsulates blocks for signal acquisition and tracking that, in order to keep locked the phase-locked loop (PLL) and delay-locked loop (DLL) for carrier and code corrections, requires a high-frequency task to update its numeric-controlled-oscillators (NCOs). These tracking loops are necessary to overcome frequency deviations arising (e.g. from Doppler Effects) during signal transmission from the GNSS satellites to the receivers.

The signal acquisition and tracking in the channel task typically do not require much RAM but a high microprocessor processing power due to the short millisecond interaction time with the channels. The low communication bandwidth of the Advanced High-performance Bus (AHB) interconnect, which is shared between modules, is the major performance bottleneck in AGGA-4. Additionally, the next generation of receivers, AGGA-5, is planned to have more than 72 channels, which would be impossible to process with this

current system architecture. Thus, a faster system is necessary to serve these needs and be able to cope with the increased code length of the modernized GNSS signals.

Even though advanced Systems-on-Chip (SoCs) are being designed with newer generations of LEON processors, such as the LEON-3-FT and the LEON-4-FT, which offer superior performance over the LEON-2-FT used in AGGA-4, these systems are still based on AHB interfaces, posing doubts regarding their suitability for solving AGGA-4's performance limitations. An exception is the GR740 SoC [2], which includes two separate AHB buses that could potentially dilute the traffic in the interconnect. However, it comes with the downside of a very high cost of implementation. A more suitable system is the NG-Ultra chip [3] which implements the DAHLIA SoC [4] that, besides a much cheaper implementation, offers several other important advantages over LEON-based SoCs. This novel system includes four Arm Cortex-R52 cores and uses the Advanced Extensible Interface (AXI) protocol for the internal interconnect buses instead of AHB. The performance is expected to be 20 to 40 times the performance of the existing SoC for space and more than 2 times the performance of the future quad-core LEON4 chip, GR740 SoC. Additionally, considering the Cortex-R52 performance and the frequency achievable on the targeted STM 28nm Fully Depleted Silicon On Insulator (FDSOI) technology, the overall DAHLIA SoC performance is estimated beyond 4000 DMIPS. Figure 1.1 presents a performance comparison between the afore-mentioned SoCs.

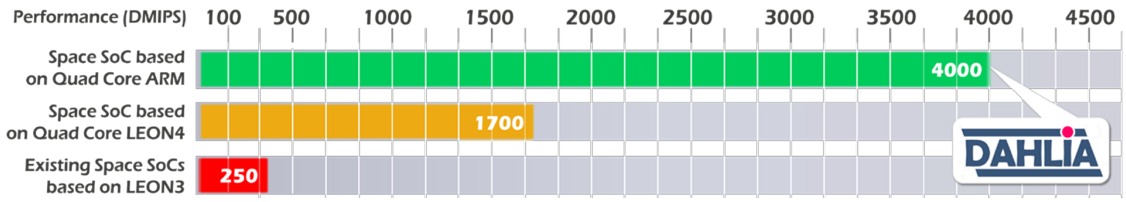

Figure 1.1: Performance comparison between SoCs targeted to space applications [4].

The Cortex-R52 represents Arm's most advanced processor for safety-critical applications. It allows the integration of complex software to be simplified through the strong separation of mixed-criticality without impacting real-time performance. It offers improved signal processing with efficient parallel calculations using the NEON Single Instruction Multiple Data (SIMD) and floating-point extension that is particularly interesting as it has been shown that efficient floating-point handling provides great performance improvement in the processing of the GNSS channels (see [5]).

The NG-Ultra presents also greater flexibility compared to other SoCs as it includes on-chip an embedded FPGA. The GNSS hardware core and interfaces can be implemented

in reprogrammable logic, making it possible to present several designs with custom characteristics differing in the number of channels, interfaces, and input modules for different RF front-ends. The implementation in FPGA also ensures potential issues to be fixed in a very late stage of the design phase, avoiding costly metal-fixes. On the other hand, LEON processors have been the baseline for European chips in many projects and missions (see [6]) and there is a lot of heritage in the use of such processors. The use of a novel technology is challenging and has a major risk for being still under development. Thus, besides the benefits that this technology could bring to GNSS sensor processing, its immaturity also justifies research before a major implementation.

In this thesis, I propose a GNSS sensor processing architecture for the DAHLIA SoC. Taking as the baseline an FPGA design of a GNSS receiver based on AGGA-4, I go over the necessary software and hardware modifications of the existing design for allowing it to be integrated into the novel DAHLIA SoC. I present a memory layout for allowing run-time critical software to be placed on fast memories, i.e. the Cortex-R52 Tight Coupled Memories (TCMs), while non-time-critical software is allocated to standard RAM. I exploit the Cortex-R52 Low-latency Peripheral Port (LLPP) for isolating the channel processing high-frequency task from the main SoC AXI bus, thus not affecting nor being affected by the shareability implications of this interconnect. I then perform an implementation of this proposed architecture in an FPGA-based prototype and evaluate its performance. Lastly, I estimate the design performance as if it were implemented as an ASIC, based on the results obtained with the prototype and compare with the characteristics of AGGA-4. I pay greater attention to GPS as its satellite signals are relatively easier to replicate when compared to other GNSSs and therefore more suitable to create a deterministic test environment. Nevertheless, the modifications to the sensor processing algorithm and the challenges encountered during implementation are extendible to other GNSSs.

This work is organized as follows. In Chapter 2, I provide a theoretical background on Global Navigation Satellite System (GNSS), give an overview of Arm Advanced Microcontroller Bus Architecture (AMBA) protocols, and present the laboratory tools used during development and testing. In Chapter 3, I introduce the novel DAHLIA SoC and the Arm Cortex-R52 it implements. In Chapter 4, I present AGGA-4 and hardware-related modifications necessary to bring its GNSS sensor processing capabilities into the new SoC. In Chapter 5, I go over the software-related adaptions to port the existing software to new system architecture. In Chapter 6, I present results concerning different test cases and estimations. Finally, in Chapter 7, I discuss the results and limitations of this work.

2 Background

In this chapter, I review the basis of GNSS with a special look at GPS signals. I then introduce the Arm Advanced Microcontroller Bus Architecture (AMBA) protocols for an easier understanding of the devices presented in the following chapters. Lastly, I go over the laboratory tools and software used during development.

2.1 Global Navigation Satellite System

Global Navigation Satellite System (GNSS) is the standard generic term for a constellation of satellites providing signals from space that transmit positioning and timing data to GNSS receivers, with global coverage [7]. The receivers then use this data to determine their geo-location. Examples of GNSS include United States' NAVSTAR Global Positioning System (GPS) [8], Russian's Global'naya Navigatsionnaya Sputnikovaya Sistema (GLONASS) [9], China's BeiDou Navigation Satellite System [10] and Europe's Galileo [11].

Although the first systems were originally intended for military usage, location awareness has soon proven invaluable for many civilian applications. Examples of applications that profit from GNSS include:

- Civil: pedestrian and outdoor navigation, games, carpooling, social networking.
- Aviation: autonomous flying, attitude determination, air traffic control, landing, and take-off.
- Maritime: en-route navigation, dredging, rescuing.
- Roads: tolling, emergency services, traffic control, fleet management.
- Industry: precision agriculture, mining, heavy machinery.
- Surveying: land surveying, mapping.
- Space: precise orbit determination, satellite real-time navigation, satellite formation flying.

Three main segments make up a GNSS. The space segment, which comprises the satellites themselves; the control segment, that is responsible for providing command and maintenance services to the satellites; the user segment, represented by the receivers. The space segment consists of a constellation of satellites transmitting radio signals to users. A GNSS satellite constellation has to ensure that there are at least four satellites in view from virtually any point on Earth. Each satellite broadcasts a signal that identifies it and provides its time, orbit, and status. These transmissions are controlled by highly stable atomic clocks on-board the satellites. The control segment consists of ground facilities that monitor the satellites' transmissions, perform analyses, and send commands and data to the constellation. These facilities are responsible for the proper system operation, performing services such as satellite maneuvers and navigation data updates for all the satellites. The user segment consists of the GNSS receivers and the user community. The receivers process the incoming satellite signals and solve the navigation equations to obtain their coordinates and provide very accurate time reference.

2.1.1 Fundamentals

The time required for a signal to travel from the satellite to the receiver is the basic observable in a GNSS system. This traveling time, multiplied by the speed of light, provides a measure of the apparent distance (pseudorange) between them [7]. The term pseudorange comes from the fact that this value does not represent the real distance between transmitter and receiver as it may be affected by various sources of errors.

The positioning principle of GNSS is based on solving a geometric equation, involving the distances from the satellites to the receiver, using a technique commonly known as triangulation but more precisely called trilateration [12]. Assuming initially as a simplifying hypothesis that the receiver clock is perfectly synchronized with the satellite, processing the signals coming from one satellite indicates that the receiver is somewhere on the sphere that surrounds the satellite and is of a radius equal to the signal's travel distance. With information from a second satellite, the location can be narrowed down to the circle formed by the intersection of the two spheres. Repeating to a third one, the receiver's location is limited to two possible points. On a more theoretical approach, one of these points can be discarded, once its altitude will be illogical (e.g. far deep in space or inside Earth) and three satellites would be enough for most situations. However, due to the inaccuracy of the receivers' inexpensive clocks, a fourth satellite is added to improve location accuracy. Therefore, with four or more satellites in view, the receiver can determine the user's lat-

itude, longitude, altitude, and clock deviation from satellite time. Figure 2.1 illustrates this scenario.

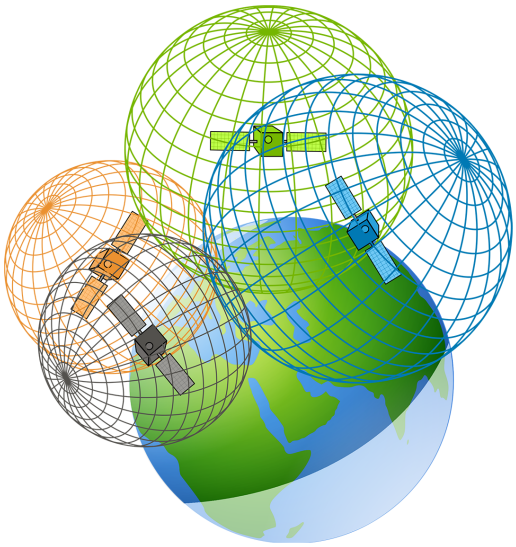

Figure 2.1: GNSS trilateration [13].

2.1.2 Signal Processing

GNSS satellites continually broadcast navigation signals with their orbit status to allow the receiver to compute its location. The navigation data are a data stream that is first modulated onto a Pseudorandom Noise (PRN) signal, a carefully engineered code specific to each satellite. PRN codes are designed in such a way they have a low cross-correlation with other PRN codes and a good autocorrelation peak value. This allows the receiver to identify from which satellite the navigation data are coming from during the demodulation of the received signal, and the satellites to transmit on the same signal frequency without suffering significant interference. After being mixed with the PRN code, the signal is modulated onto a carrier sinusoidal wave at a higher frequency in order to be transmitted. On the receiver side, local replicas of the carrier and code are mixed with the incoming signal in order to extract the navigation message. Figure 2.2 illustrates the GNSS signal transmission.

In GPS, the navigation data are sent at a 50-bits/s rate and then modulated by the Coarse/Acquisition (C/A) code, also called civilian code, or the Precision (P) code which is encrypted and for military usage. The carrier wave is at the so-called L1 band, centered at 1575.42 MHz. The C/A code sequence is made up of 1023 chips with a duration of 1 us each.

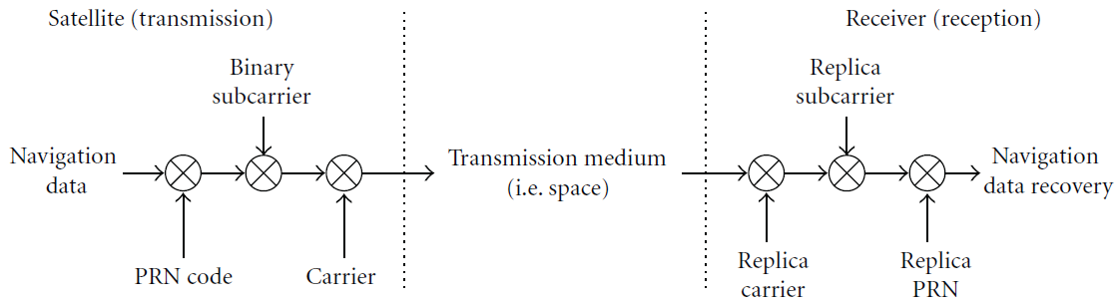

Figure 2.2: GNSS signal transmission [14].

In order to extract the navigation message, the receiver must be synchronized with the incoming signal. This is done by correlating the input signal with local replicas of the code and carrier, and then sweeping these signals in time and frequency, respectively, until a correlation peak is found. The frequency compensation is needed to eliminate the residual carrier due to noise sources while the code matching synchronizes the receiver and satellite time, an observable to calculate the pseudorange. When the correlation peak is found, the receiver is able to lock the signal and from there on keep track of it. Figure 2.3 illustrates this scenario.

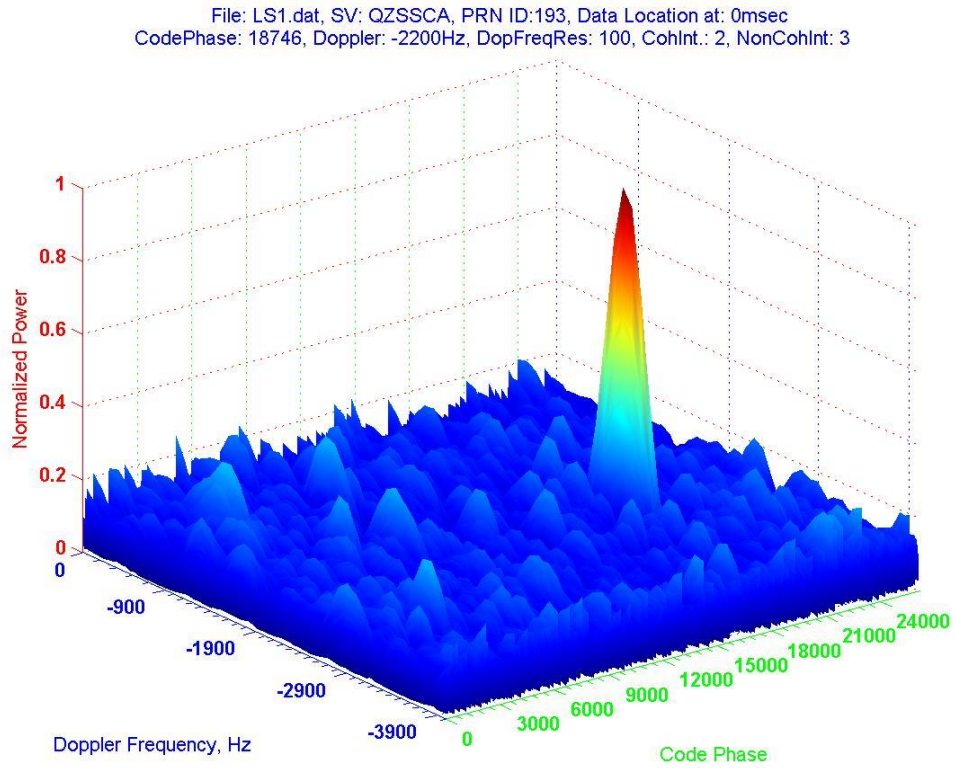

Figure 2.3: Acquisition of GPS L1 C/A Signal [15].

2.1.3 Error Sources

Measurement errors arise from many different sources, such as the signal attenuation by the atmosphere (ionosphere and troposphere), relativistic effects, instrumental limited accuracy, multipath, and receiver noise.

Doppler effects

Because the GNSS signals are originated from moving satellites, they are prone to frequency shifting introduced by Doppler effects. This effect is reflected in a continuous movement of the phase of the signal coming into the receiver. The frequency shifting in the incoming signal can be greater in space applications when compared to receivers on Earth because the receivers might be moving at a very high speed.

Multipath

Multipath is the phenomenon that results in signals reaching the receiver's antenna by two or more paths. Like any other wireless communication, GNNS is multipath prone, which is accentuated in urban and indoor environments. It affects the phase measurements, as well as the code measurements. In the case of the code, it can reach a theoretical value of 1.5 times the chip's wavelength. This means, for instance, that multipath in the GPS C/A code can reach up to 450 m, although higher values than 15 m are difficult to observe. Typically, it is less than 2 or 3 m [7].

2.1.4 Receiver architecture

Figure 2.4 presents a block diagram of a generic GNSS receiver. The GNSS signal is captured by the antenna and first treated by an analog radio front-end responsible for down-converting the signal to an intermediate frequency. This stage involves amplifying and filtering the signal to recover a good signal-to-noise ratio. The signal goes through an A/D converter with resolution varying from 1-bit (low-cost receivers) to 4 bits (high-end receivers). At an intermediate frequency, the residual carrier frequency, mainly due to Doppler effects, is then wiped off in the carrier's loop, resulting in a signal in baseband frequency. Further along the processing chain, the GNSS signal is correlated with the local replica of the code, as part of the code loop, and the navigation data is extracted. The tracking loops, a PLL and a DLL, are used to continuously lock the incoming signal.

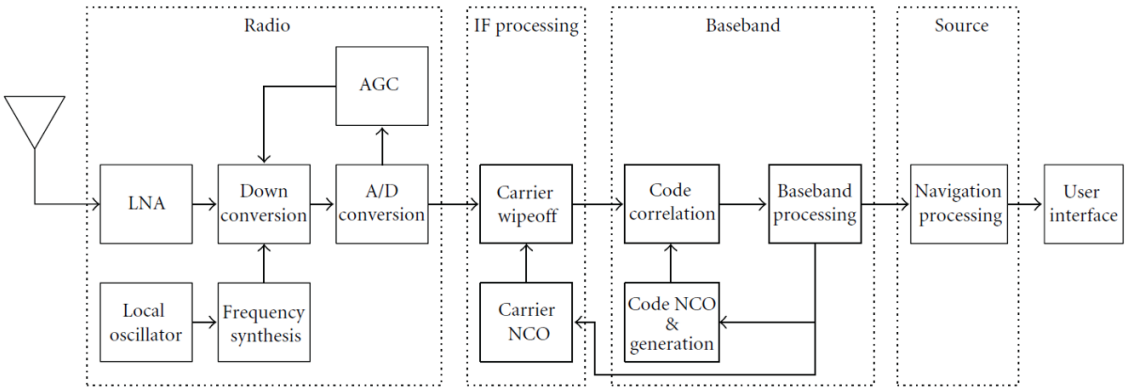

Figure 2.4: Generic GNSS receiver block diagram [14].

2.2 Advanced Microcontroller Bus Architecture

The Advanced Microcontroller Bus Architecture (AMBA) [16] is an open-standard from Arm widely used for the connection of functional blocks in a SoC. It defines multiple protocols targeted to different requirements.

2.2.1 APB

The Advanced Peripheral Bus (APB) is a simple non-pipelined protocol designed for low bandwidth control accesses, presenting a low complexity signal list, not allowing burst transactions. It is generally used for interfacing general-purpose peripherals such as timers, interrupt controllers, UARTs, and I/O ports.

2.2.2 AHB

The Advanced High-performance Bus (AHB) is designed for high bandwidth interconnect having a single-channel shared bus. Similar to APB, it is a shared bus protocol for multiple masters and slaves, but higher bandwidth is possible through burst data transfers. The AHB supports pipelined operation, with the address and data phases occurring during different clock periods.

2.2.3 AHB-Lite

The AHB-Lite protocol is a simplified version of AHB. The simplification comes with support for only a single master design that removes the need for any arbitration, retry, and split transactions.

2.2.4 AXI

The Advanced Extensible Interface (AXI) is targeted at high performance for low latency and high clock frequency systems design. It overcomes the limitations of a shared bus protocol about the number of agents that can be connected.

2.3 Laboratory Tools

In this section, I briefly introduce the laboratory tools and software used during project development.

Arm Development Studio

Arm Development Studio (Arm DS) is an embedded C/C++ development toolchain designed specifically for Arm-based SoCs [17]. I extensively used this tool to run and debug the GNSS software in the FPGA evaluation board.

Arm DSTREAM

Arm DSTREAM is a high-performance debug and trace that enables software debugging and optimization of any Arm processor-based hardware target, using a hardware interface such as JTAG.

GNU Compiler Collection

The GNU Compiler Collection (GCC) [18] is a compiler system produced by the GNU Project supporting various programming languages. It is free software distributed under the GNU General Public License (GNU GPL). GCC was the compiler used for the GNSS software and I extensively exploited this toolset for allowing different code and data placement in the SoC.

Vivado Design Suite

Vivado is a software suite produced by Xilinx [19] for the synthesis and analysis of HDL designs. I used Vivado for the generation of the AXI4-to-AHB-Lite bridge through its IP Integrator tool for synthesizing the DAHLIA SoC with the new GNSS capabilities, and for loading the FPGA bitstream into the evaluation board.

HDL Designer

HDL Designer is a tool by Mentor Graphics that allows the user to graphically connect different components described in VHDL/Verilog and automatically generate the necessary code for that. I used HDL Designer in the first part of the hardware design process while creating and evaluating the GPS C/A code generator and during the AXI4-to-AHB-Lite bridge interfacing with the GNSS Module. The final integration into the DAHLIA SoC was done in plain Verilog language.

Mentor QuestaSim

QuestaSim is a multi-language HDL simulation environment by Mentor Graphics for simulation of HDL designs. I used this tool to compile and simulate RTL designs during the whole development flow. It offers a graphical interface with signal timing wave plotting that was extremely useful.

MATLAB

MATLAB is a multi-paradigm numerical computing environment and proprietary programming language developed by MathWorks [20]. I used MATLAB in this project for a generating a sampled GPS C/A code dataset to be used in a look-up table used by the GPS code generator, and also for signal tracking performance estimation.

Evaluation Board

I carried out development and testing on a Xilinx Virtex UltraScale+ FPGA VCU118 evaluation board together with a Xilinx FMC XM105 Debug Card to provide JTAG connection to Arm DSTREAM.

3 Deep Sub-micron Microprocessor for Space Rad-hard Application ASIC

In this chapter, I introduce the novel DAHLIA SoC that will be implemented on the NG-Ultra chip along with its system architecture and specification. Then, I give an overview of the Arm Cortex-R52, highlighting the main features relevant to this study.

3.1 Overview

The deep sub-micron microprocessor for space rad-hard application ASIC (DAHLIA) is a quad-core Arm-based System-on-Chip dedicated to both platform and payload applications. This chip is designed to boost competitiveness and ensure the strategic non-dependence of future European space equipment [4], by a consortium of European companies with the supervision of European entities like ESA, EU, and CNES [21].

The project targets the STM 28nm Fully Depleted Silicon On Insulator (FDSOI) technology, providing a very good tolerance to radiation while its performance is expected to be 20 to 40 times the performance of the existing SoC for space and more than 2 times the performance of the future quad-core LEON4 chip.

The DAHLIA SoC is composed of two major groups:

- Hard Macroblocks: the NG-ULTRA SoC, a physical application-specific standard product (ASSP) that comprises two main functional units:
  - an ASIC digital processor subsystem, with four Arm Cortex-R52 cores.
  - a large embedded FPGA (eFPGA).
- Soft Macrblocks: a collection of different soft IPs that implement peripheral I/O functions typical of satellite computers, mapped onto the eFPGA fabric.

Figure 3.1 presents a block diagram of DAHLIA SoC.

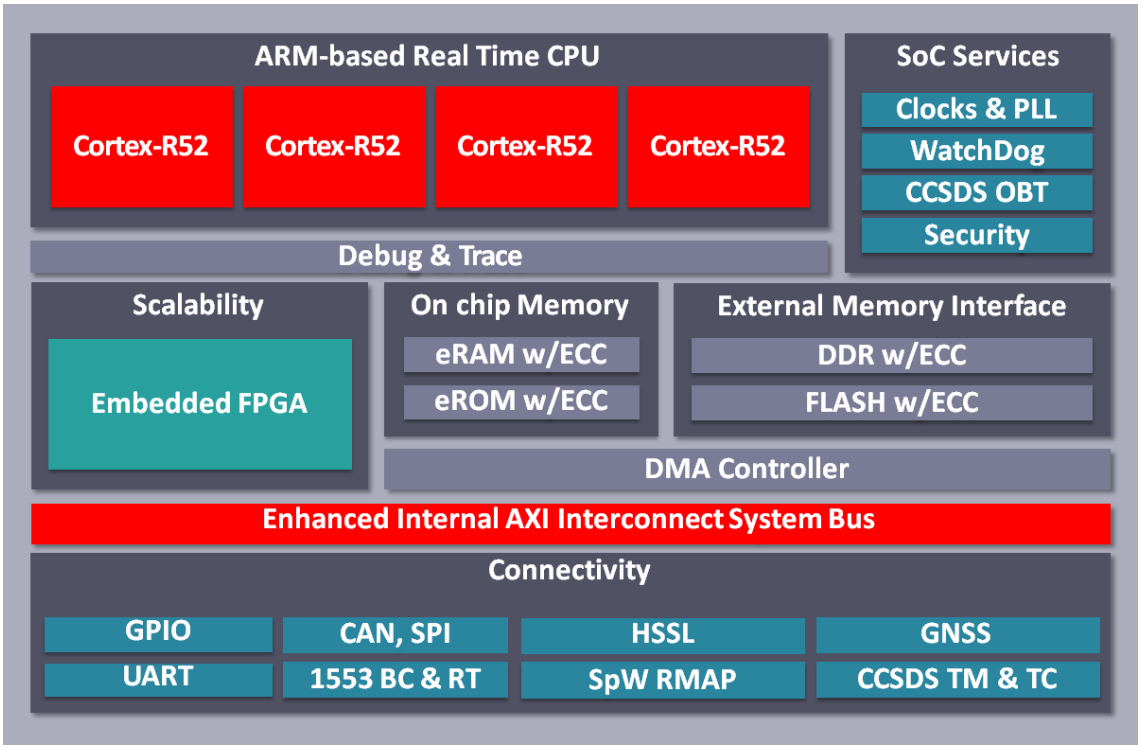

Figure 3.1: DAHLIA SoC block diagram [4].

3.2 Hard Macroblocks

The hard macroblocks are part of the NG-ULTRA SoC developed by NanoXplore, a high-end radiation hardening-by-design (RHBD) FPGA device, stated as the most advanced FPGA in the world to this time [3]. The available resources are the following:

- Logic: 536928 LUT4 + 505344 DFFs + 126336 CY chains.
- Memories: 672 BRAM blocks of 48kb (= 32256kb).
- DSP: 1344 DSP blocks which can be cascaded.

3.2.1 Processor Subsystem

The processing unit is constituted by two double-core Arm Cortex-R52 clusters and periphery to allow application software to be executed. A more thorough analysis of the Cortex-R52 and its Armv8-R architecture is carried out in section 3.6.

SoC Serivces

The services module is a collection of IPs that guarantees the functional SoC's behavior. It includes PLLs and a clock generation subsystem, a JTAG connection, a Debug and Trace

module, power and error management IPs, security services, and others.

On-Chip Memory

DAHLIA includes on-chip a 128 KB eROM and a 4 MB eRAM. Flash and DDR memories may be connected through an external memory interface.

3.2.2 Embedded FPGA

The embedded FPGA is the key aspect for providing flexibility to the applications since it allows multiple design variants, changing requirements, and better performance by eliminating chip-to-chip delays. The eFPGA is a family of SRAM based programmable logic blocks implemented with 4 inputs LUT and DFF fabric [3]. The high-end FPGA fabric is meant to hold the soft macroblocks and the second layer of the interconnect and to provide reprogrammable logic embedded in the DAHLIA chip to the application.

3.3 Soft Macroblocks

The soft macros provide complementary I/O functions and the IPs can be instantiated with different configurations by the applications. It includes peripherals such as a Space Wire controllers, a UART, CAN/SPI/I2C controllers, and others.

3.4 Network Interconnect

Even though the network interconnect is part of the macroblocks, I introduce it in this separate section for a better understanding of the SoC architecture. The purpose of the network interconnect is the connection and management of functional blocks inside the SoC so that the different masters, such as the four CPU cores, can have access to different interfaces and memory resources. Having the NIC-400 associated with QoS-400 features as the interconnect baseline, DAHLIA implements an AMBA bus topology.

The interconnect has a multi-layer architecture that manages clock domain crossings, data width, and arbitration between APB, AHB, and AXI protocols. The first layer is fixed and responsible to handle the functions implemented in the ASIC part while a second layer is implemented on the FPGA fabric, being adaptable to features implemented by the application. A simplified view of the interconnect ASIC layer is given in Table 3.1. Table 3.2 identifies which master-slave connections are enabled. The module *R52 Core-0* refers

to the first CPU core inside the first cluster, while the other three cores are not shown as they were not used in this project, as it will be better clarified in the next chapters.

Module	Protocol	Type	Width	Clock	CPU Interface
R52 Core-0	AXI4	Master	128 bits	600 MHz	AXIM
R52 Core-0	AXI4	Slave	128 bits	600 MHz	AXIS
R52 Core-0	AXI4	Master	32 bits	600 MHz	LLPP
eFPGA	AXI4	Master	128 bits	- (*)	-
eFPGA	AXI4	Slave	128 bits	- (*)	-
eFPGA	AXI4	Slave	32 bits	- (*)	-
eFPGA	APB4	Slave	32 bits	- (*)	-
eRAM	AXI4	Slave	32 bits	400 MHz	-
eROM	AXI4	Slave	32 bits	200 MHz	-
UART	APB2	Slave	32 bits	200 MHz	-

(*) Clock is provided by the eFPGA

Table 3.1: ASIC layer of the network interconnect.

Masters \ Slaves	R52 Core-0 AXIS AXI4 (128 bits)	eFPGA AXI4 (128 bits)	eFPGA AXI4 (32 bits)	eFPGA APB4 (32 bits)	eRAM AXI (32 bits)	eROM AXI4 (32 bits)	UART APB2 (32 bits)
R52 Core-0 AXIM AXI4 (128 bits)	X	X		X	X	X	X
R52 Core-0 LLPP AXI4 (32 bits)			X				
eFPGA AXI4 (128 bits)	X	X		X	X		X

Table 3.2: ASIC layer interconnect matrix.

As it can be seen, the AXI is the main interconnect in DAHLIA SoC, interfacing the CPU cores, the eFPGA, the eRAM, and the eROM while the APB is used for interfacing low bandwidth system peripherals such as the UART. The 32-bit AXI4 bus is instead a separate bus, which is a direct connection from the LLPP interface of the R52 Core to the eFPGA. It only goes through the interconnect for allowing clock domain crossing so that clock synchronization does not have to be performed separately. This interface is better explained in section 3.6.3.

3.5 FPGA-based Prototype

Although DAHLIA SoC is intended to be an ASIC, a silicon chip is not yet available. I carried out the development of the project on an FPGA-based design implemented in a Xilinx Virtex UltraScale+ FPGA. Due to the limited logic gate space and lower frequency, the prototype has several limitations concerning the ASIC design, including:

- Only the first CPU cluster is implemented, with its two cores.
- The clock frequencies are downscaled by a factor of 12 (roughly).
- The eFPGA is not included and its interfaces are mapped to the prototype's FPGA fabric.
- The eROM memory is a RAM and, therefore, reprogrammable.
- No flash memory is available.

Table 3.3 presents the ASIC interconnect layer that is implemented in the FPGA-based prototype, highlighting the downscaled clock frequencies.

Module	Protocol	Type	Width	Clock	CPU Interface
R52 Core-0	AXI4	Master	128 bits	50 MHz	AXIM
R52 Core-0	AXI4	Slave	128 bits	50 MHz	AXIS
R52 Core-0	AXI4	Master	32 bits	50 MHz	LLPP
FPGA fabric	AXI4	Master	128 bits	- (*)	-
FPGA fabric	AXI4	Slave	128 bits	- (*)	-
FPGA fabric	AXI4	Slave	32 bits	- (*)	-
FPGA fabric	APB4	Slave	32 bits	- (*)	-
eRAM	AXI4	Slave	32 bits	33 MHz	-
eROM	AXI4	Slave	32 bits	16.5 MHz	-
UART	APB2	Slave	32 bits	16.5 MHz	-

(*) Clock is provided by the eFPGA

Table 3.3: ASIC layer of the network interconnect implemented in the FPGA-based prototype.

Since the eFPGA is not present, its interfaces were connected to AXI Traffic Generator IPs implemented in the FPGA fabric. These IPs provided by Xilinx can generate AXI4 transactions so that register access and data transfers are possible without getting the bus into an error or timeout state.

3.6 Arm Cortex-R52

As DAHLIA SoC implements a quad-core Arm Cortex-R52, it is crucial to understand its characteristics. In this section, I introduce the main components of this processor core. The figures and the majority of the information are taken from its technical reference manual [22].

3.6.1 Overview

The Cortex-R52 is currently the most advanced processor in the Arm Cortex-R family, a series of 32-bit RISC processor cores designed for real-time applications, offering high standards of functional safety. It is 35 % faster than the previous generation Cortex-R5, which is already deployed in a range of safety applications. This is the first processor implementing the Armv8-R architecture, offering several error-resiliency features to ensure accuracy. The multi-core lock step allows two cores to execute the same task in parallel for redundancy. Hardware-enforced separation of software tasks ensures that safety-critical code is fully isolated and virtualization allows for multiple tasks to be executed on the R52 without interfering with each other. This enables fewer processors within the device since these tasks can be consolidated onto a smaller number of processors. By ensuring robust separation of software, the Cortex-R52 also decreases the amount of code that must be safety-certified, speeding up development as software integration, maintenance, and validation becomes easier. Figure 3.2 presents a processor example with two cores.

3.6.2 Exception Model

The exception model for the Cortex-R52 processor is mostly defined by the architecture it implements, the Armv8-R AArch32 profile. The Armv8-R exception model defines three exception levels EL0, EL1, and EL2 (lowest to highest priority) where:

- EL0 is called unprivileged execution, commonly used for applications.
- EL1 is described as privileged execution, commonly for OS kernels.
- EL2 is called hypervisor level and provides support for processor virtualization.

Due to the immaturity of the framework available for development, which still does not handle access prioritization between these levels, I only considered the EL2. At this level, the software has full access to all Cortex-R52 capabilities.

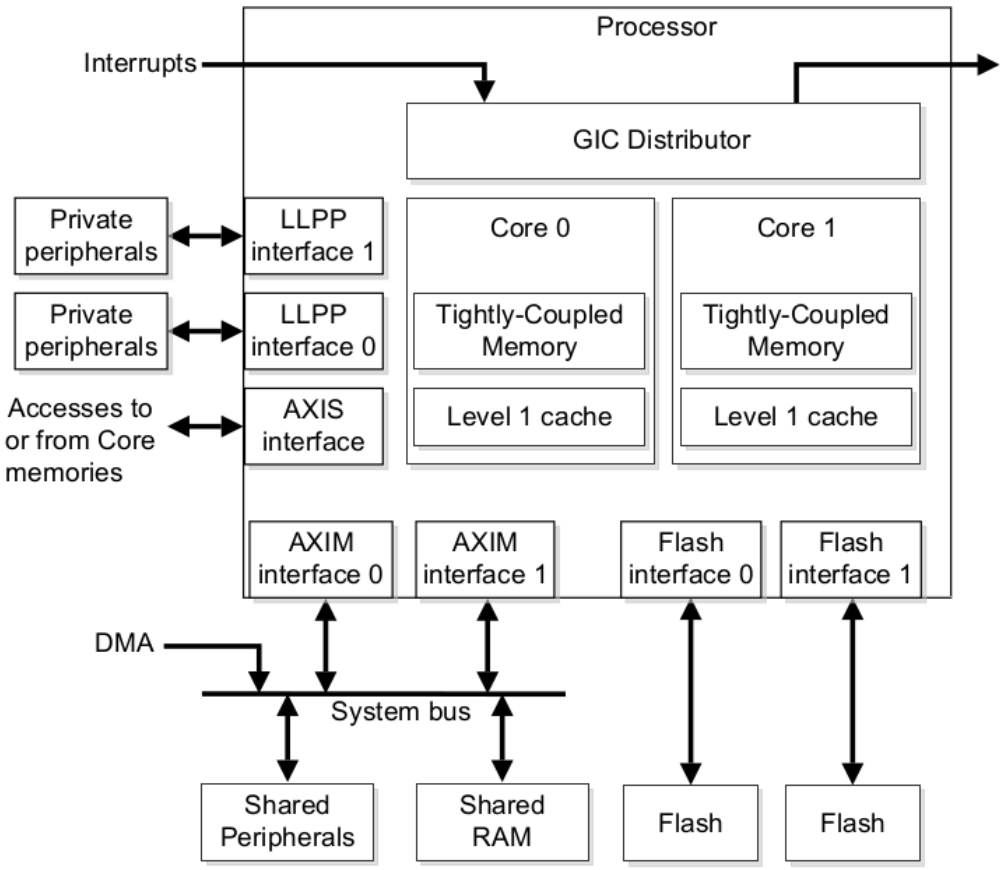

Figure 3.2: Example of a Cortex-R52 implementation with two CPU cores (modified from [22]).

3.6.3 Memory System

The memory system controls access to internal RAM, cache, external memory, and the peripheral port. Its block diagram is presented in Figure 3.3.

In DAHLIA SoC, the implemented memory system consists of:

- 3 x 128 KB TCMs.
- 4 KB L1 instruction cache.
- 4 KB L1 data cache.
- LLPP master interface that conforms to AXI4 (32-bit).
- Flash interface that conforms to AXI4 (128-bit).
- AXIM interface that conforms to AXI4 (128-bit).
- AXIS interface that conforms to AXI4 (128-bit).

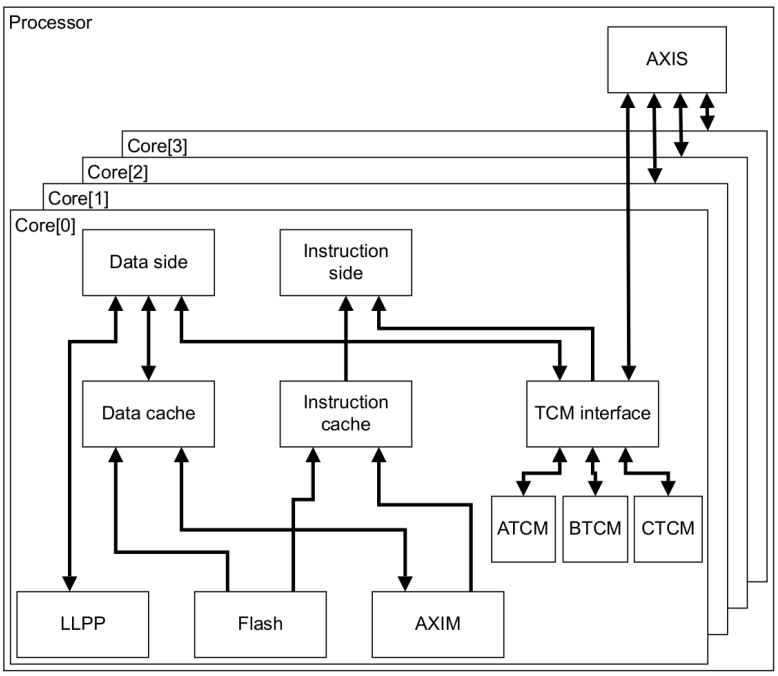

Figure 3.3: Cortex R52 memory system block diagram [22].

The Cortex-R52 has a Harvard memory architecture, meaning that the core has independent buses to access instructions and data. The instruction side fetches instructions while the data side reads and writes data. Concurrent accesses to both sides can be done.

Tightly-Coupled Memory

Tightly-Coupled Memories (TCMs) are for highly deterministic applications, providing low-latency memory that the processor can use without the unpredictability of caches. This memory can be accessed by the CPU in a single cycle and it is unified, meaning that it can hold both instructions and data, as shown in Figure 3.3. TCMs are suited for holding service routines and data that require intense processing which cannot wait for cache misses, such as interrupt handling routines, interrupt stacks and other critical data structures, and data types whose locality is not suited for caching.

Level-1 Caches

The Cortex-R52 implemented in DAHLIA SoC includes both instruction and data 4 KB caches. Instruction fetches from flash or AXIM interfaces can be cached in the instruction cache, and when the instruction flow is sequential, the cache can automatically prefetch the next line from memory. Data accessing from flash or AXIM interfaces can be cached in the data cache depending on the configuration of the Memory Protection Unit (MPU).

AXIM

The AXI-Master interface conforms to the AXI4 specification and is the main interface to memories external to the core and device systems. In DAHLIA SoC, it is connected to the main AXI interconnect.

AXIS

The AXI-Slave interface is shared between all cores in the processor. It provides external access to the TCMs so they may be used by DMA controllers and other processor cores, for example. This interface, in DAHLIA SoC, is also connected to the main AXI interconnect.

Flash Interface

The Flash interface provides access to an external read-only memory controller, but it is not analyzed in this project once there was no flash memory in the FPGA-based prototype used for development.

Low-Latency Peripheral Port

The Low-latency Peripheral Port (LLPP) provides direct access to external devices or small specialized memory systems. It is intended to be used for private peripherals requiring low-latency access and can run at the processor's clock frequency consequently providing real-time behavior. The LLPP conforms to a subset of the AXI4 specification.

Memory Protection Unit

The memory management in the Cortex-R52 is configured by the Memory Protection Unit (MPU). It allows setting the attributes for each memory location such as permissions, type, and cacheability. The MPU defines the permissions for each exception level and can be controlled from EL1 and EL2.

3.6.4 Generic Interrupt Controller

The Generic Interrupt Controller (GIC) is a resource for supporting and managing interrupts in a CPU cluster. It supports interrupt prioritization, interrupts routing to a core or export port, interrupts preemption, and interrupts virtualization. A block diagram of the GIC is given in Figure 3.4.

The GIC architecture is made of three main components: a Distributor, a Redistributor, and a CPU interface. The GIC Distributor receives three types of interrupts: wired inter-

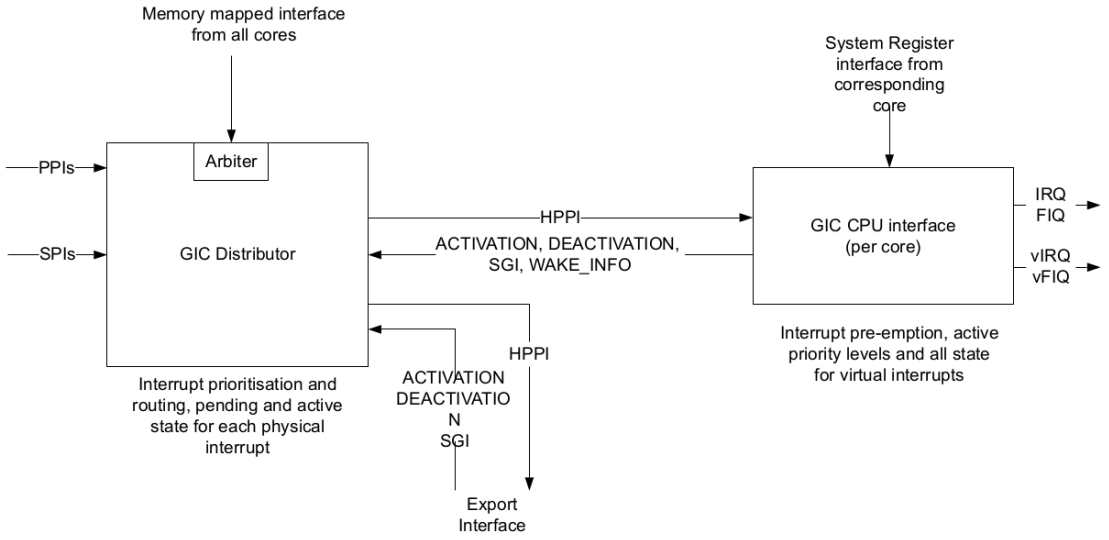

Figure 3.4: Cortex-R52 GIC block diagram [22].

rupts from peripherals, so-called Shared Peripheral Interrupts (SPIs), Private Peripheral Interrupts (PPIs), and Software Generated Interrupts (SGIs). The latter two are private to each CPU core. The Distributor contains the prioritization logic which defines the highest priority pending interrupt for each core. The Redistributor contains the registers supporting PPIs and SGIs. The CPU interface tracks the current running priority and virtual interrupts, then determines whether the core is interrupted. The Cortex-R52 processor implements one internal GIC CPU interface and one GIC Distributor per core.

In DAHLIA SoC there are implemented, per core, 16 PPIs and 16 SGIs. Table 3.4 gathers some of the interrupts that can be signaled as PPI. Additionally, more than 100 SPIs can be set to signals in the eFPGA. An interrupt is configured as either a Group 0 interrupt or a Group 1 interrupt. Group 0 interrupts are signaled with fast interrupt request (FIQ) while Group 1 interrupts are signaled with interrupt request (IRQ). FIQs are for fast, low-latency interrupt handling, while interrupt request (IRQ) for general interrupts. An FIQ always takes priority over an IRQ.

PPI
Non-secure physical timer interrupt
Virtual timer interrupt
Hypervisor timer interrupt
Virtual CPU Interface Maintenance interrupt
Cross Trigger Interface interrupt
Performance Monitor Counter Overflow interrupt
Debug Communications Channel interrupt

Table 3.4: Private Peripheral Interrupts assignments.

3.6.5 Advanced SIMD and Floating-Point

The Advanced Single Instruction Multiple Data (SIMD) and floating-point module uses NEON technology, providing improved signal processing with efficient parallel calculations. It offers:

- Instructions for single and double-precision floating-point operations.
- Hardware support for conversion, addition, subtraction, multiplication, division, and square-root operations.
- Hardware support for rounding modes.

NEON is a very advanced technology developed at first for audio and video encoding and decoding, and graphics rendering and implemented in the Arm Cortex-A processor series. It was then extended to the Cortex-R family and has shown great advantages for intense signal processing applications other than multimedia.

3.6.6 Performance Monitor Unit

The Performance Monitor Unit (PMU) is a Cortex-R52 peripheral that enables gathering various statistics on the operation of the core and its memory system during runtime, providing useful information about the behavior of the processor. The PMU provides four counters that can count system events, such as error events, memory request events, cache predictions, memory access requests, and many others. Figure 3.5 shows a block diagram of the PMU.

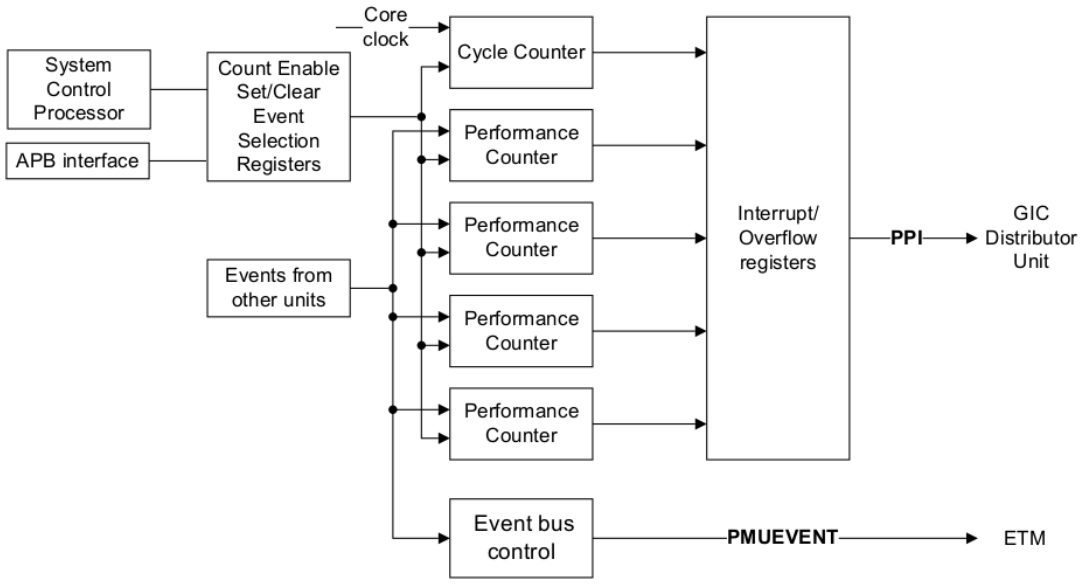

Figure 3.5: Performance Monitor Unit block diagram [22].

4 Hardware Design

In this chapter, I describe the design process from a hardware point of view. I first introduce the GNSS receiver taken as starting point, describing its relevant modules and interfaces, and then go over the adaptions needed to integrate it into DAHLIA SoC.

4.1 Advanced GPS/Galileo ASIC

The Advanced GPS/Galileo ASIC version 4 (AGGA-4) is a radiation-tolerant GNSS space receiver designed by Airbus and ESA and fabricated by Atmel [1]. This receiver includes on-chip the LEON-2 FT (Fault Tolerant) processor based on the SPARC V8 standard. The processor and periphery consist of a cache subsystem, a memory controller, a GNSS Interrupt Controller (GIC), a Communication Interrupt Controller (CIC), a Primary Interrupt Controller (PCI), four 32-bit timers, a watchdog, watchpoint registers, and a 32-bit I/O-port. It includes also a Cobham Gaisler Floating-Point Unit (GRFPU). AGGA-4's block diagram is depicted in Figure 4.1.

The GNSS receiver taken as the baseline for this project was an FPGA design based on AGGA-4. Although this receiver differs from AGGA-4 in a few aspects, the components are here mentioned as from AGGA-4, referring to the functionalities derived from it. When necessary, I point out the differences.

4.1.1 GNSS Core

The GNSS Core is the hardware component that processes GNSS signals. Together with the AHB interface, DMA and synchronization controllers, it is referred to as GNSS Module. In AGGA-4, this module is connected to an AHB interconnect with two interfaces: an AHB master and an AHB slave interface. The AHB master interface is required to implement DMA capabilities to the GNSS Module while the AHB slave interface is used to provide access to the internal registers via read and write operations. These interfaces are shown in greater detail in Figure 4.2.

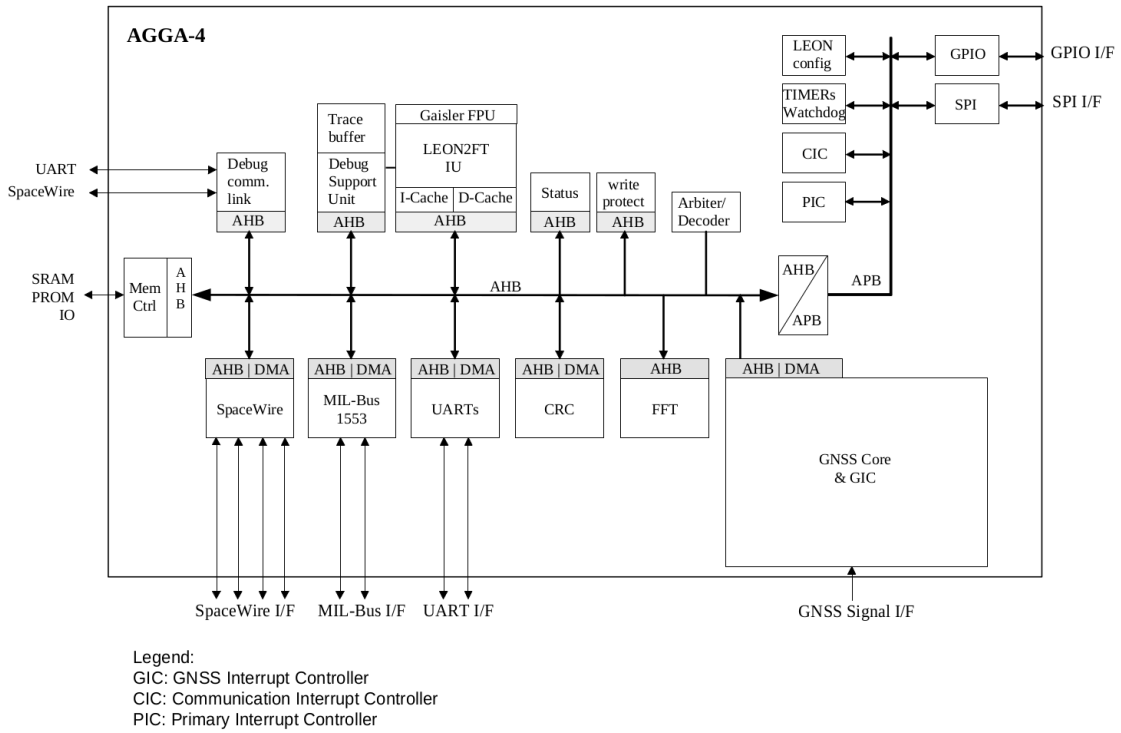

Figure 4.1: AGGA-4 block diagram [23].

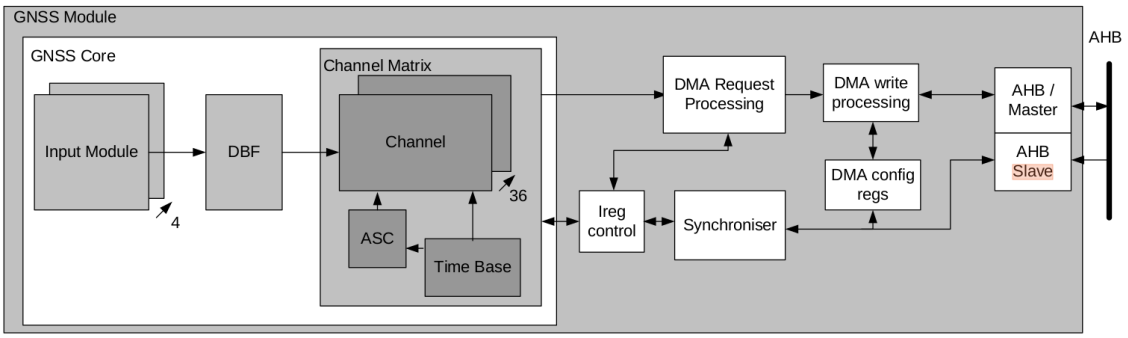

Figure 4.2: GNSS Module block diagram [23].

Besides the AHB interfaces and the modules that handle DMA, there is a register controller that arbitrates accesses to the GNSS Core between the AHB slave and the DMA controller, and a synchronization module that guarantees that requests via AHB are synchronized with the GNSS Core since they are placed in different clock domains. The AHB address decoding is also done within this module. The peripheral that performs processing of incoming GNSS signals is the GNSS Core. Figure 4.3 presents its architecture and the components that make up this module. The Digital Beam Forming module and the Aiding Unit inside each channel are not included in the FPGA design.

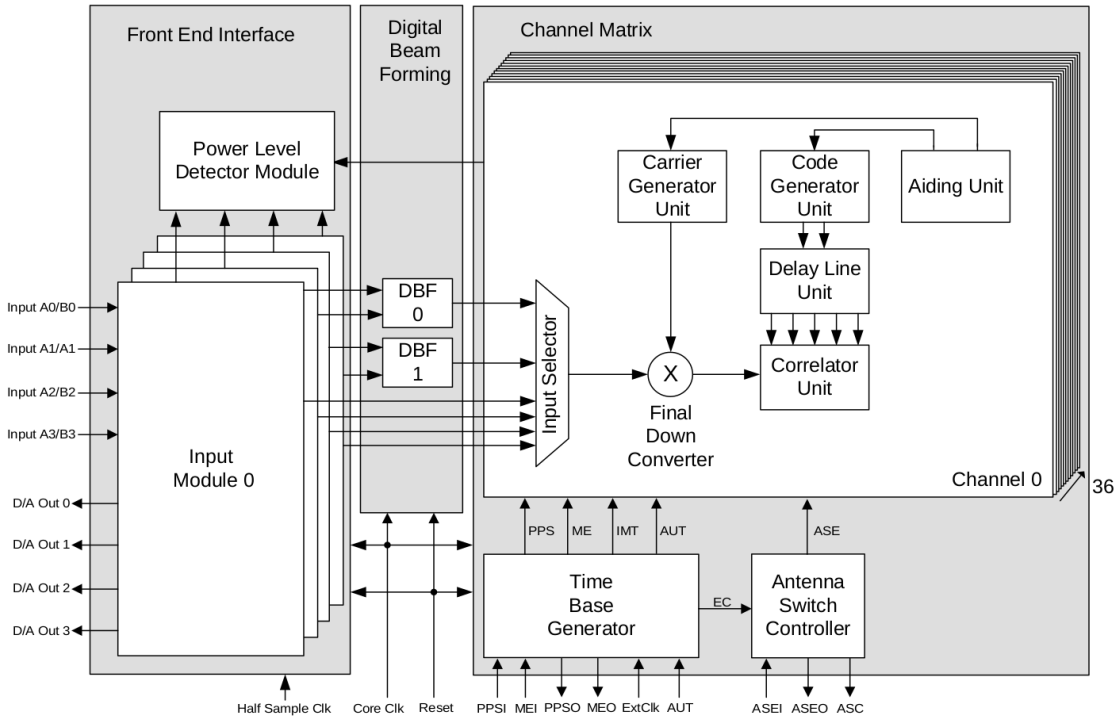

Figure 4.3: GNSS Core architecture [23].

Input Module

The GNSS Core includes input modules that interface with the ADCs of the RF front-ends. They support multiple input formats in baseband and intermediate frequency that perform data sampling and formatting of the data before being introduced into the channels.

Power Level Detector Module

The Power Level Detector is used to monitor the input levels of the incoming signal for allowing the Automatic Gain Control (AGC) of the RF Front Ends. It generates sample statistics that are used by the software to calculate an appropriate AGC value for the RF front-end.

Input Selector

The Input Selector selects between the I/Q signals coming from the Input Module or from the previous channel if channel slaving is to be used. Each I/Q component is represented in a 3-bit format.

Final Down Converter

The Final Down Converter is responsible for removing the residual carrier and strip off the Doppler frequency from the input signal by mixing it with the locally generated carrier.

Carrier Generator Unit

The Carrier Generator provides the carrier replica to convert the navigation signal to baseband frequency. The carrier frequency is set via software, as part of the carrier loop.

Code Generator Unit

The Code Generator Unit is responsible for generating the channel's local replica of the PRN code. The code frequency is set via software, as part of the code loop.

Delay Line Unit

The Delay Line Unit controls the delay to be applied to the code correlators, generating Early Early, Early, Punctual, Late and Late Late (i.e. EE, E, P, L, LL) samples of the selected code sequence within the channel.

Correlator Unit

The Correlator Unit performs the cross-correlation between the input signal and the local copy of the PRN code. The results of the integration are made available as code observables.

4.1.2 Other Peripherals

Debug Support Unit

The Debug Support Unit (DSU) offers hardware debug support to the LEON processor, in order to aid software debugging on the target hardware. The DSU can put the processor in debug mode, allowing read and write access to all processor registers and cache memories.

Primary Interrupt Controller

The interrupts generated by on-chip units are all forwarded to the Primary Interrupt Controller (PIC). The controller core then propagates the interrupt with the highest priority to the LEON processor. This module combines several sources of interrupts, including the other interrupt controllers, onto a single output bus.

GNSS Interrupt Controller

The GNSS Interrupt Controller (GIC) is located inside the GNSS clock domain and handles signals coming from the GNSS Core before forwarding them to the PIC. Events such as the status of integration epochs, DMA transactions, and others can be configured to trigger an interrupt.

Communication Interrupt Controller

The Communication Interrupt Controller is similar to the GIC but manages signals related to communication peripherals before forwarding them to the PIC.

Serial Peripheral Interface

The Serial Peripheral Interface (SPI) module works as an SPI master and is used to communicate with the RF front-end chipset. During the software initialization, the RF front-end is configured through this interface.

UART

The GNSS Receiver Software needs to communicate to an Application Software that runs externally and this is done via UART. It is also used for debugging of the LEON processor as it allows access to all internal registers in the design.

4.2 Onto DAHLIA SoC

In this thesis, I propose an architectural solution for implementing an AGGA-4's based GNSS receiver on DAHLIA SoC. Given DAHLIA's architecture, the hardware peripherals necessary for performing GNSS signal processing are intended to be loaded in the reprogrammable part of the SoC, the eFPGA. Even though the FPGA-based prototype available did not include an eFPGA, these hardware components were restricted to only make use of the eFPGA interfaces and consider their limitations. The following sections clarify all modules from AGGA-4 necessary to provide GNSS capabilities to the new SoC along with modifications to comply with the new system architecture. I also consider adaptions for allowing subsequent testing and system verification.

Although DAHLIA SoC has on-chip four Arm Cortex-R52 cores, only one was used for running the GNSS sensor processing algorithm. This was a design choice based on the assumption that a single core has sufficient processing power budget to accommodate the

software needs, taking into consideration the much greater expected performance of the Cortex-R52 over the LEON processor (see Figure 1.1). As it is shown in the following chapters, this assumption is verified. A second reason for making this choice was to spare the other CPU cores for future applications. Lastly, it reduces the complexity and power consumption of the overall system.

4.3 Clock Generation

The first design step was to decide the frequency in which the GNSS Core should operate. A constraint concerned the generation of specif clock frequencies arose, once it would require setting the PLLs in the evaluation board to the desired frequency, implying in big design modifications as the clocks signals are generated by an IP produced externally which I could not modify in order to not violate project requirements. Limited to use clock signals already in the design or frequency factors of them, I selected a clock frequency that, considering the ASIC to FPGA prototype frequency downscale factor of 12, would lead to the best approximation of the GNSS Core running at 60 MHz, an actual target frequency inside Airbus Defence and Space. Taking a clock signal at 39 MHz, I implemented a simple frequency 8th divider, resulting in a signal at 4.875 MHz. In the ASIC, this would be translated to the GNSS Core running at 58.5 MHz, a reasonable approximation to the target frequency. The clock signal at 4.875 MHz is from here on referred to as GNSS Core clock.

4.4 Signal Aquisition

Having a clock source created, I moved to the signal acquisition of GPS signals, analyzing different possibilities and their feasibility, and then performing an implementation of the selected solution.

Dealing with real GPS signals poses many challenges to the overall system. First, it requires setting up an antenna which, considering that a receiver expects line-of-sight visibility of at least four satellites to establish a reliable location, would be a hard limiting for a prototype running at an indoor laboratory. Second, there is a need for an RF front-end to bring down the incoming signal to an intermediate or baseband frequency, that besides the extra device and its peripheral circuitry, demands also a careful noise power level calibration. Additionally, interfacing this front-end with the FPGA prototype requires setting up its I/O ports.

Software-generated GNSS signals could be used instead of real signals, from simulators such as the JC Air GPS emulator. The problem with this solution is that GPS signals generated by this tool are not repeatable and, consequently, a replicable test case cannot be created, affecting comparisons between different designs. An alternative software could be the Spirent GNSS emulator, which allows simulating the entire GPS satellite constellation in static and dynamic scenarios, granting repeatable signals and, therefore, more accurate performance comparisons between test cases. These solutions, however, still do not tackle the need for an RF front-end and interfacing with the FPGA prototype.

To overcome these challenges encountered during signal acquisition, I opted to create a GPS signal generator as close to ideal as possible, with signals already in baseband frequency and integrate it directly into the GNSS Core, to be synthesized together with the whole design. This solution solves the problems presented so far and, although acquiring real GPS signals is critical to demonstrate the GNSS capabilities of this new device, this has to happen in a later stage when the framework is fully functioning with signals close to ideal.

By making this choice, there was no need anymore for the Front-End Interface in Figure 4.3, with its Input Modules and Power Detector Module, as these components are responsible for acquiring the GNSS signals being output by the ADC in the RF front-end. I then removed the connections from the Input Modules to the Input Selector, to later on let the synthesis tool optimize out these modules. In addition to the front-end modifications, given the limited gate space in the FPGA prototype, I also opted to reduce the number of channels in the Channel Matrix to four.

Regarding the signal generator, the target signal was the GPS C/A in the L1 band, where the navigation data broadcast by a satellite is first modulated by a 1.023 MHz C/A code and then by a 1.57542 GHz carrier wave. As the generator was meant to be placed inside the GNSS Core, the signal frequency was required to be already in baseband. For the sake of simplicity, I set the signal's residual carrier frequency to be zero, considering a case with no error sources affecting the signal carrier frequency. The constraint on clock generation formerly presented posed an issue on the C/A code generation, once 1.023 MHz is not an exact division of the GNSS Core clock frequency. For this reason, I opted for creating a sample counter looping over a look-up table (LUT) with signal sampled-data, in the way that a new sample is put out on every GNSS Core clock cycle and when the counter reaches the last sample in the look-up table, it loops back again to the first one. The outputs with the in-phase and quadrature samples were then linked to the channels' Ipunt

Selector with a 3 bits representation of the range [-7,-5,-3,-1,+1,+3,+5,+7], complying to AGGA-4's specification. The input selection was done via software.

The sample-data were generated in MATLAB. Setting the carrier wave frequency to be zero implied in identical I/Q samples at each point in time, with only a C/A code sampled. By distributing a whole C/A sequence, i.e. 1023 code chips, over an integration epoch with 1 ms period, the code frequency is $f_{code} = \frac{n_{chips}}{\Delta_{IE}} = \frac{1023\,[chips]}{1\,[ms]} = 1.023\,MHz$, as desired. In order to avoid phase shifting between integration epochs, the LUT should have to be looped over on every new integration epoch and this was accomplished by setting the number of samples to be:

$$n_{samples} = f_{CoreClock} \cdot \Delta_{IE} = 4.875\,[MHz] \cdot 1\,[ms] = 4875 \tag{4.1}$$

Therefore, the look-up table was generated by distributing 1023 chips over 4875 samples. The modified GNSS Core architecture with the GPS C/A code generator included is depicted in Figure 4.4.

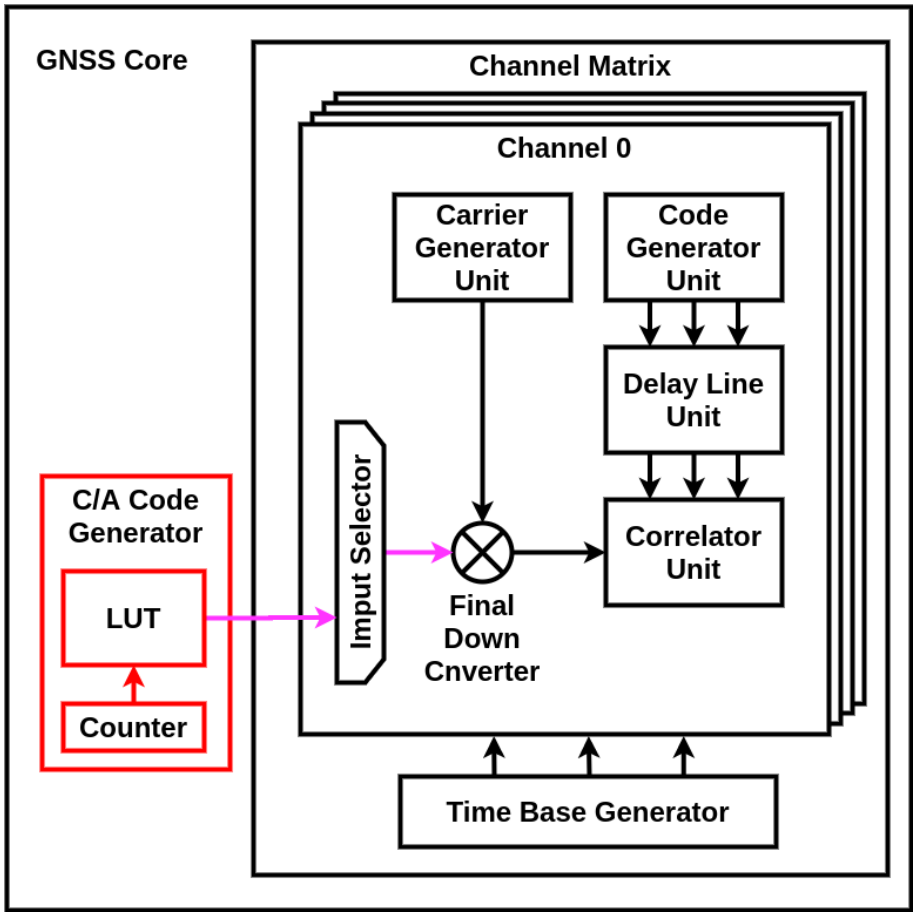

Figure 4.4: GPS C/A code generator

4.5 Bus Interface

The eFPGA can be accessed by the CPU core through a 128-bit AXI4, a 32-bit APB, or a 32-bit AXI4 interface, as previously shown in the interconnect matrix (see Table 3.2). The first two happen through the Cortex-R52 AXIM interface, on the CPU side, while the latter is done via LLPP.

As formerly presented in Figure 4.2, the GNSS Module is connected to AGGA-4's AHB with both a master and a slave interface. The AHB master is used to provide DMA capabilities but it is not used by the GNSS software, as it will be shown in the next chapter. The AHB slave is responsible for read and write transactions and it has to be accessible by the R52 core. The DAHLIA SoC, however, does not include an AHB interface on the eFPGA. Therefore, I had to implement a sort of protocol translation. The APB was discarded straight away, as the communication on this bus has a rather slow bandwidth. The choice between using the 128-bit AXI4 and 32-bit AXI4 had to be analyzed, and this was done from a CPU perspective.

4.5.1 LLPP over AXIM

The R52 core has two fast interfaces that provide access to the eFPGA, the AXIM and the LLPP. The AXIM is connected to the main AXI interconnect, shared between external memories and system devices. The LLPP is instead a direct link between the R52 core and the eFPGA. This bus is not shared with other devices and only passes through the interconnect for allowing clock domain crossing and synchronization.

Given the shared bus, transactions via AXIM are prone to latencies introduced by wait-states when the bus is busy being used by other SoC components. This could become an issue when the interconnect is under high processing from other modules, for instance, the other 3 CPU cores. Additionally, when a request arrives on the 128-bit AXI bus on the eFPGA side it has to still pass through the second layer of the interconnect for the management between multiple devices that can be loaded in the reprogrammable fabric.

Another issue concerns the bus width of the 128-bit AXI4 interface. The AHB interface in the GNSS Module has a 32-bit data width bus, so if it had to be connected to the AXI, only a quarter of the transaction width would be used. Additionally, an extra decoder would have to be designed to make the addresses in the 128-bit range which are not used by the GNSS Module available for other systems loaded in the eFPGA, or even the GNSS Module would have to be redesigned to a 128-bit architecture.

Choosing to use the LLPP for interfacing the GNSS Module solves all issues discussed

above. It is a CPU port designed for real-time peripherals with same bus width as the GNSS Module, eliminating the need for an architectural redesign of this module. In addition, the LLPP is separate from the main interconnect. Such characteristic isolates the main interconnect from the overhead introduced by the high bus usage of the channel processing task writing data to the GNSS Core. The same applies to the other way around, the channel processing task is not affected by other applications using the main AXI interconnect.

Even though the LLPP is a better choice over the AXIM, it still complies with the AXI4 protocol, unlike the AHB in the GNSS Module. To make this connection possible, I added an AXI4-to-AHB bridge between the LLPP and the AHB interface. The bridge used was from the Xilinx IP libraries and is presented in the following section. The complete interface is illustrated in Figure 4.5.

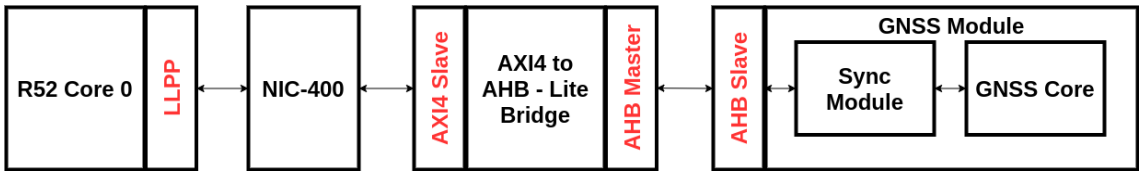

Figure 4.5: Hardware connection scheme between the Cortex-R52 Core-0 and the GNSS Module.

The interfaces of the NIC-400 interconnect were omitted for simplicity while the Sync Module was highlighted inside the GNSS Module because it had to be modified to comply with the bridge interface and the LLPP address range. I explain this better in the following sections.

4.5.2 AXI4-to-AHB-Lite Bridge

The AXI4-to-AHB-Lite Bridge translates AXI4 transactions into AHB-Lite transactions. The bridge functions as a slave on the AXI4 interface and as a master on the AHB-Lite interface. This IP was taken from the Xilinx LogiCORE IP and was generated and customized with the Vivado IP integrator tool. The AXI4-to-AHB-Lite Bridge diagram is shown in Figure 4.6.

The AXI4 address and data bus width were configured to have 32 bits and, by design, the AHB-Lite interface has the same data width that the AXI4 interface. The Timeout Module was set to generate a timeout when the AHB-Lite slave does not respond for 256 clock cycles. The bridge was configured to respond to the 4 MB address range correspondent to the LLPP and the address that is presented on the AHB-Lite is exactly as received on the AXI4. The bridge is a synchronous design and uses the same clock signal at both

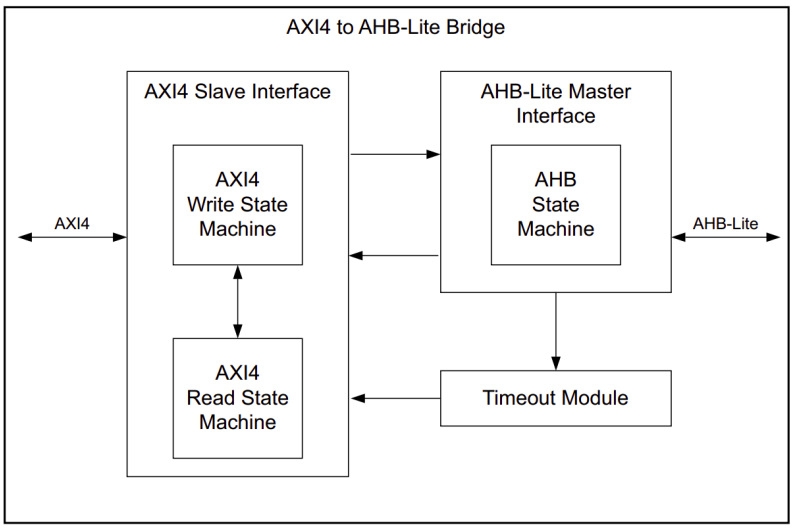

Figure 4.6: AXI4-to-AHB-Lite bridge block diagram [24].

interfaces.

4.5.3 Bridge Connection

The above-described bridge includes an AHB-Lite interface. This protocol is a simplified version of the full AHB specification, which in turn is used in the GNSS Module. For this reason, I had to make some adaptions.

The AHB-Lite protocol is used with a central read data multiplexor interconnection scheme. The master drives out the address and control signals to all the slaves with an address decoder selecting the appropriate slave. Any response data from the selected slave passes through the read data multiplexor to the master. Figure 4.7 shows the multiplexor interconnection structure for an example implementation with three slaves.

Given that the GNSS Module was the only slave to be connected to the master interface, I simply connected the HRDATA bus and HRESP directly to the bridge, without implementing a multiplexor. I then connected the HREADYOUT coming from the GNSS Module to the HREADY signal of the bridge and also fed this signal back into the HREADY input of the GNSS Module. For better visualization, a complete signal diagram of an AHB-Lite slave is shown in Figure 4.8. Regarding the input address and control, and data signal groups, I simply connected them directly from the bridge to the GNSS Module. I then attached the HSEL to a constantly HIGH signal, leaving the GNSS Module constantly selected, thus not having to implement any decoder. Finally, I connected the clock and reset signals to the Clock Generator formerly presented in section 4.3.

Besides the signals shown in Figure 4.8, the GNSS Modules has also an input signal

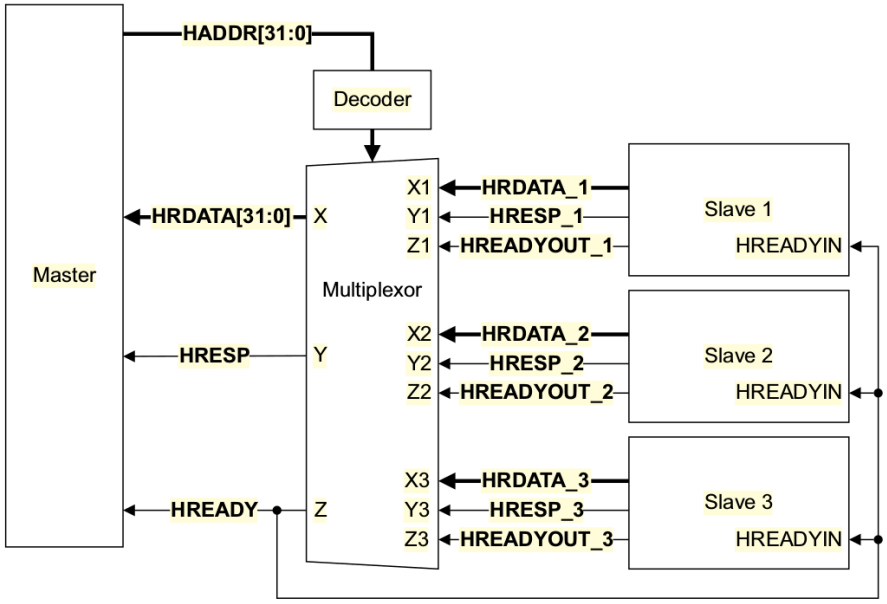

Figure 4.7: AHB-Lite block diagram for one master and three slaves [25].

HMASTER that is used to provide information about which master is currently accessing it. The AHB-Lite protocol supports only a single master, thus the bridge does not have this signal as output. For this reason, I simply attached the HMASTER signal to a constant LOW signal, making the GNSS Module identify every transaction as coming from the same master.

A problem arose with this implementation because the GNSS Module turned out to be non-compliant to the standard AHB. The specification states that the HREADYOUT signal when set to HIGH indicates that a transfer has finished on the bus and it may be driven LOW to extend a transfer. The GNSS Module, however, outputs a double-cycle pulse when a transaction has finished. This signal is actually generated by the Sync Module and indicates when the synchronization is done after a transaction has completed. This is because, in AGGA-4, there is an arbiter at a higher level of the design that converts this non-standard HREADYOUT synchronization signal before feeding it to a multiplexor structure similar to the one illustrated in Figure 4.7. To overcome this incompatibility, I modified the Sync Module to output a custom signal instead of the synchronization one. I implemented a simple logic based on internal signals to drive this custom signal LOW when the Sync Module is accessed and drive it back HIGH once the synchronization pulse has been generated. It is important to notice that both the bridge and the GNSS Module have the same clock source, therefore all signals in these modules are always synchronized to each other.

Regarding the connection between the bride's AXI4 interface and the NIC-400, all signals

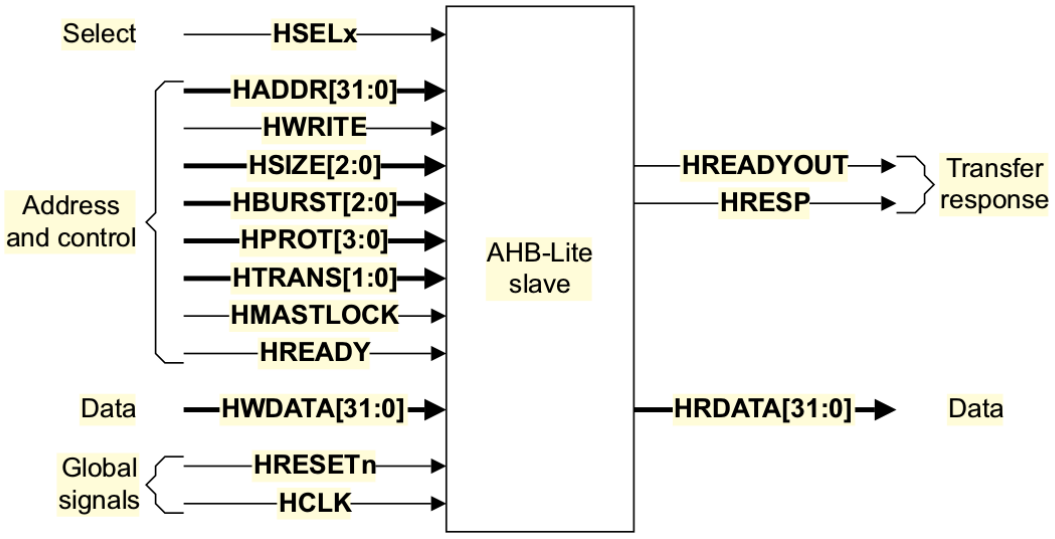

Figure 4.8: AHB-Lite slave interface [25].

were compatible and no adaption was needed.

4.5.4 Address Map

The LLPP is constrained to a 4 MB address range. However, the GNSS Module address map in AGGA-4 has a very sparse structure that exceeds this LLPP address limit. To make all peripherals accessible, I defined a new address table to be used by the GNSS Receiver Software that could fit in the 4 MB range. Opting for a less intrusive solution, I added an address decoder inside the Sync Module to translate this new address map back to the original AGGA-4's sparse structure.

4.6 Interrupt Signals

In AGGA-4, many signals can trigger an interrupt of the CPU core. They are used for handling communication, watchdog functions, timer overflows, and to signal the GNSS Core status and readiness. These signals are handled by the interrupt controllers PIC, CIC, and GIG. When integrating the GNSS Module into DAHLIA SoC, I had to make a design choice whether to include these controllers in the new system or to solely use the Cortex-R52 GIC. The advantage of keeping the AGGA-4 interrupt controller is that less code would have to be changed in order to handle the interrupts used by the GNSS Receiver Software, as the configuration of these peripherals is already done, such as setting up the timers, masking and unmasking the interrupt sources, setting the interrupt priorities.

These modules, however, would be loaded into the eFPGA which is in a slower clock domain than the CPU, introducing an unnecessary latency that is critical in interrupt handling. Furthermore, this choice would also implicate greater logic gate usage in the FPGA fabric. For these reasons, I opted for only use the Cortex-R52 GIC. As I better explain in the next chapter, the interrupts coming from hardware timers in AGGA-4 were replaced by resources available in the Cortex-R52. Additionally, DAHLIA SoC includes more than 100 eFPGA signals connected to the GIC, more than enough to accommodate all interrupt signals in the GNSS Module.

4.7 Peripherals

In this section, I discuss theoretical solutions for some system peripherals in AGGA-4 which were not implemented in this project but are necessary to be integrated into the DAHLIA SoC for the full functioning of the given GNSS Receiver Software.

Only two peripherals from AGGA-4 other than the GNSS Module are necessary for implementing all the original features of the GNSS Receiver Software, a UART and an SPI module. In reality, as these modules implement widely used communication protocols, other IPs could be used for providing their functionalities, although it might require modifications in the software drivers.

The SPI module is fully dedicated to the communication with the RF Front End chipset, which is programmed and initialized through this interface. In AGGA-4, this peripheral is connected to the interconnect via a 32-bit APB. Fortunately, DAHLIA SoC includes a 32-bit APB interface on the eFPGA, as already shown in Table 3.2, meaning that this module simply would have to be connected to this interface.

The UART is for communication with the Navigation Module, which will be presented in the next chapter. In AGGA-4, this module is interconnected via a 32-bit AHB interface. To allow the R52 core to access such a peripheral, a bridging solution similar to the one adopted for interfacing of the GNSS Module would have to be performed. This solution is not optimal, as it requires a greater logic gate usage in the eFPGA and the performance is affected by the latency introduced by the bridge. The advantage is that no changes in the software driver would be necessary. If performance is not critical whatsoever, this module could even be connected to the 32-bit APB interface of the eFPGA or replaced by the UART already present in the DAHLIA SoC. If performance does matter, a UART with a 32-bit AXI4 could be the best solution. As one can see, that are many options for integrating this module to the new system but more in-depth analysis and following

implementation are objects for future research.

4.8 FPGA Design Synthesis

The logic synthesis of the RTL design proved to be very challenging. Given the complexity of the SoC design, the synthesis tool would take up to five days to finish generating an FPGA bitstream, running on a machine with an 8th generation Intel i7 processor and 16 GB of RAM. I faced several problems when including external IPs and could not find synthesis and place-and-route strategies that would not lead to timing violations in the final routed design. These timing violations would then make the debug interface to not work, precluding software tests to be performed. For the evaluation, I then considered tests of the GNSS software without the GNSS Module integrated into the DAHLIA SoC together with results obtained during simulation of the RTL design.

5 Software Design

In this chapter, I discuss the GNSS Receiver Software. I first present this multi-module application while retaining greater attention to the components responsible for the processing of the GNSS channels. Then, I describe the porting process of GNSS software to the new SoC, explaining the design choices I made along the way. Lastly, I propose a new memory layout aiming to exploit new features introduced by the Arm Cortex-R52.

5.1 GNSS Receiver Software

The GNSS software taken as the baseline for this project is composed of two modules, depicted in yellow in Figure 5.1. The Application Software calculates the receiver position, velocity, and precise time (PVT) based on attitude and acceleration information from the on-board computer (OBC) and signal measurements coming from the Receiver module such as pseudorange and range rate, and navigation data retrieved from GPS satellites like almanac and ephemeris. It delivers channel predictions and clock corrections towards the Receiver Software. The Application Software runs on an Arm Cortex R5 processor and is referred to as Navigation Module. The analysis of this module is out of the scope of this thesis. The Receiver Software, instead, is where hard real-time processing is performed, running on a LEON processor synthesized in a Spartan-6 FPGA. It has the Receiver Framework that manages the processor low-level configuration and the Sensor Module, which contains the code and carrier loops for searching and tracking the signals broadcast by GPS.

5.1.1 Sensor Processing Module

The Sensor Module contains all the functions needed to acquire, track, and decode GPS L1 C/A signals. A functional overview is depicted in Figure 5.2, with software components being represented in yellow and hardware in grey.

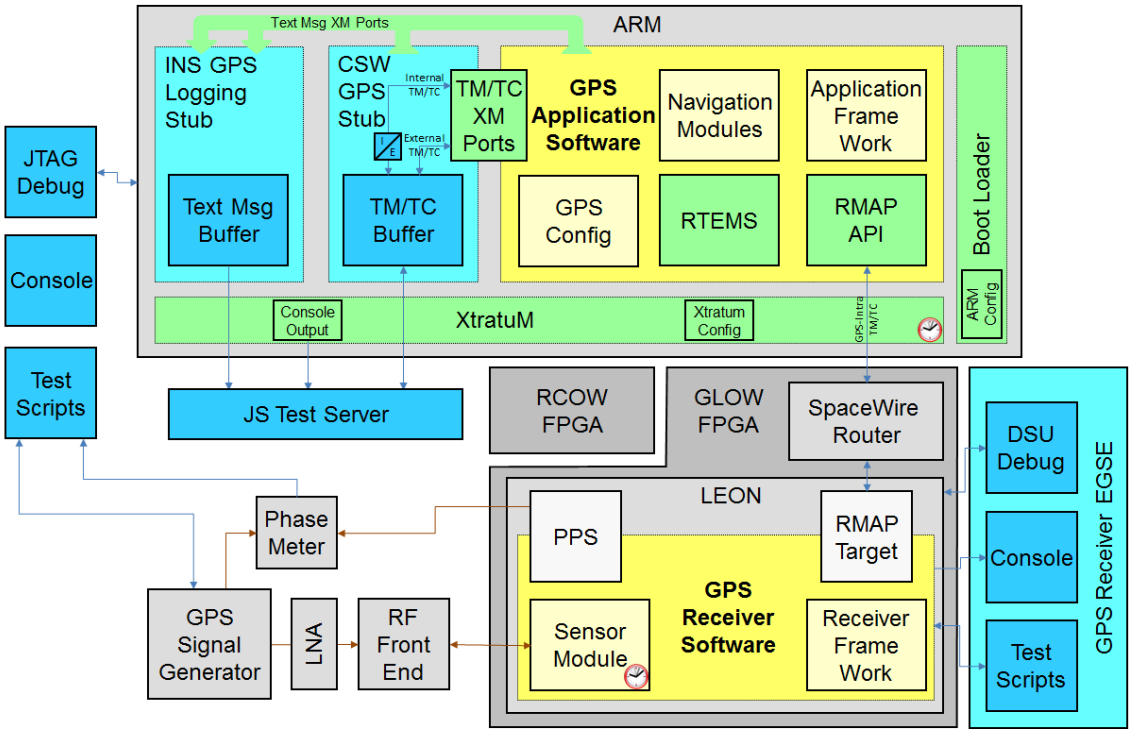

Figure 5.1: GNSS baseline software block diagram [26].

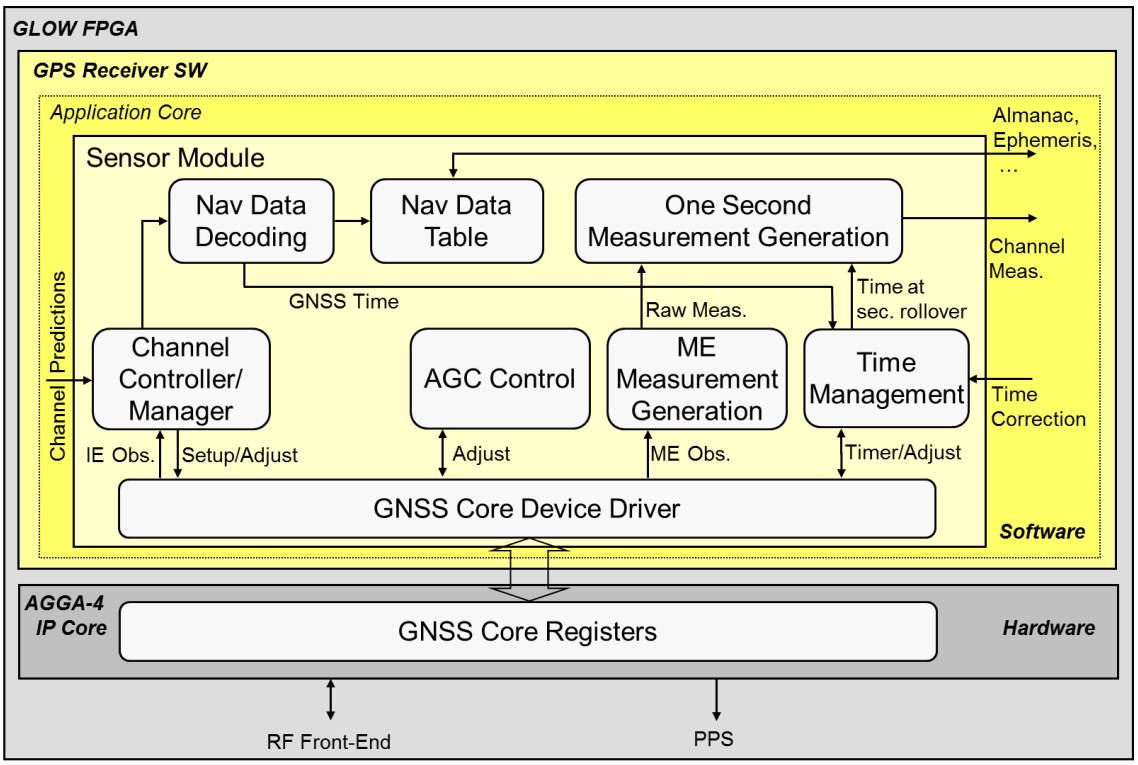

Figure 5.2: Sensor Module block diagram [26].

This module receives as input the GPS L1 RF signal coming from the RF front-end and channel predictions from the Navigation Module. It then outputs measurements on the individual GPS satellite, such as pseudorange and range rate, and navigation data like the almanac, ephemeris, and iono. As the Receiver Software is running bare metal, after a first initialization is done, the processor is kept in an infinite loop, waiting for an interrupt to happen. Three possible interrupts may occur, and each block from Figure 5.2 is assigned to one of these interrupts as shown in Table 5.1.

Interrupt	Functional Blocks
Channel Controller Interrupt	Channel Controller
Measurement Epoch Interrupt	ACG Control ME Measurement Generation Time Management
One Second Interrupt	Nav Data Decoding and Tables One Second Measurement Generation

Table 5.1: Interrupt assignment of the functional blocks in the Sensor Module .

The Channel Controller Interrupt has the highest priority and is triggered by a hardware timer overflow signal every 500 us (2 kHz). The Measurement Epoch Interrupt is invoked whenever a Measurement Epoch (ME) event occurs, which happens roughly every 20 ms (50 Hz). The ME event is acknowledged by the Channel Controller that, from software, forces the Measurement Epoch Interrupt to be triggered. Within the Measurement Epoch Interrupt the receiver time is maintained. Once the time management detects a second rollover, it triggers the One Second Interrupt, which then executes all the procedures related to this module. Table 5.2 summarizes the characteristics of the Sensor Module's interrupts, specifying the priority scheme, the invoking frequency, and how they are triggered.

Interrupt	Priority	Frequency	Source
Channel Controller Interrupt	High	2000 Hz	Timer
Measurement Epoch Interrupt	Medium	50 Hz	Software generated
One Second Interrupt	Low	1 Hz	Software generated

Table 5.2: Characteristics of the Sensor Module's interrupts.

At system startup, all channels are idle until the Channel Controller receives predictions for the individual channels containing e.g. the PRN which shall be processed within the requested channel number. The software keeps checking for new predictions in the endless loop. When a prediction is received, the Channel Controller then runs the search initialization task for configuring the carrier and code NCO values and the initial integration time. Once the signal is found, the Channel Controller moves to a tracking state.

GNSS Core Device Driver

The GNSS Core Device Driver implements a hardware abstraction layer for the application software for accessing all AGGA-4 hardware registers. A function in the device driver is implemented for every hardware feature accessed via software. It controls access to the PIC, the SPI interface, and the GNSS Core.

Channel Controller/Manager

The Channel Manager configures all channel parameters. After each Integration Epoch (IE) the channel generates new IE observables. They are used by the Channel Controller to compute new correction values for the code and carrier NCOs. They are re-programmed at the rate of the integration epoch to make sure that the channel keeps the tracking with the carrier and code phase of the incoming GNSS signal, as part of the code and carrier tracking loops. After the acquisition, the estimation of the code phase and carrier frequency is used to check whether it is possible to lock the loops or not. This is done at 1 ms integration time. If its is possible to lock the signal, then it is tracked from there on and the loops are switched to a 4 ms integration time. Otherwise, the Channel Controller goes back to the search procedure. The tracking state machine is shown in Figure 5.3.

The Channel Controller processes new IE observables with a rate of up to 1000 IE observables per second per channel, during acquisition and tracking. Given this high processing rate, the Channel Controller is responsible for most of the CPU usage.

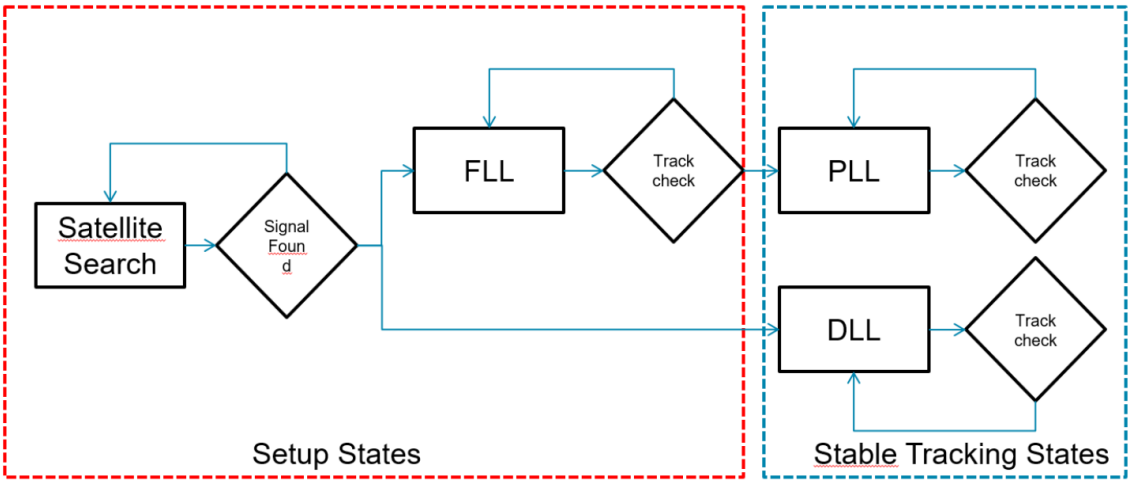

Figure 5.3: Signal tracking sequence inside the Channel Controller [5].

Nav Data Decoding and Table

The Navigation Data Decoding block decodes the navigation data out of the signal and fills the Navigation Data Tables with data such as the ephemeris and almanac. These data are fetched by the Application Software to be used in navigation planning.

AGC Control

The Automatic Gain Control (AGC) is used to keep the noise floor of the RF front-end at an optimum level such that the ADC operates at its optimal range. The AGC can compensate for noise power fluctuations caused by e.g. temperature or aging effects.

ME Measurement Generation

The ME Measurement Generation block processes the ME observables from AGGA-4. While the IE observables contain the correlation results and are used for acquisition and tracking of the individual GNSS signals, the ME observables contain the raw measurement results of the individual GNSS signals and are used to produce raw measurements such as the pseudorange, and phase and range rate measurements.

One Second Measurement Generation

The One Second Measurement Generation module is executed once per second and gathers the raw measurements which the ME Measurement Generation produced. From these data, the final one-second measurements are calculated.

Time Management

The Time Management block is responsible for the time management in the receiver. It keeps the receiver time based on the hardware internal time and the time retrieved from the navigation data.

5.1.2 CPU Usage

The CPU usage of the GNSS Receiver Software is currently increasing linearly with the number of used channels either in acquisition or tracking state. With every added channel more time is spent in the Channel Controller, ME Measurement Generation, and One Second Measurement Generation. The Channel Controller is using by far most of the processing time since it is executed 2000 times a second. While this rate poses no issues with only one channel, it becomes an issue when multiple channels are being processed in

parallel, so optimizing the Channel Controller for performance gives the best improvements in timing. The CPU load in the ME Measurement Generation is increasing with every channel which is not in the idle state, as more measurement observables generated by the hardware must be filled in the measurement data queues. Nevertheless, since it runs every 20 ms, it does by far not consume as much CPU load as the Channel Controller. The same applies to the One Second Measurement Generation, which runs at an even lower rate.

5.2 Onto DAHLIA SoC

The porting process of GNSS software to the new SoC can be summarized in three main sequent conceptual steps. The first refers to the modifications needed in the software to compile for the new processor architecture with the new compiler. The second step was to adapt the software to the new processor architecture, including fixing the functionalities that were lost in the first phase, in order to recover the GNSS signal processing. The third step relates to the software adaptions to take advantage of the new processor's features for optimizing the algorithm performance.

5.3 SPARC Assembly Code

Being architecture-dependent and specific to the LEON processor, all the code written in SPARC assembly language had to be removed. I did it using C compiler directives, replacing the inline assembly code segments and functions by empty stubs. I then removed the assembly files from the compilation list. This strategy adopted was meant to be less intrusive than removing all function calls in the code and I fixed the affected functionalities later with the Cortex-R52 features. The modules affected relevant for the tracking of signals were the Debug Support Unit, in which assembly was used to implement timing routines such as delay functions; the Serial General Purpose Output (SGPO) driver, the peripheral port used to output debug text messages; and the interrupt and exception handlers.

5.4 Dependences on SPARC Compiler Definitions

Due to a combined framework of the Application Software and the Receiver Software, some pieces of code rely on data structures and definitions coming from the SPARC compiler and the Real-Time Executive for Multiprocessor Systems (RTEMS). As they are simply defined in plain C language, I added the files containing the used definitions to the compilation list, causing no impact on functionality.

5.5 Floating-Point Support

After every integration epoch, the Channel Controller reads out all correlator results from the AGGA-4 hardware and transforms them on floating-point values for further processing. These values are used either by the acquisition or tracking, which includes several filters that contain additions, multiplications, and other floating-point operations. The GNSS software requires double-precision floating-point calculations to process the integration epoch observables and produce the measurement epoch data, such as the pseudorange, phase, and range rate. At first, to get the code compiled, I set the compiler directives to use software emulated floating-points, as it can handle all kinds of numbers representations and precisions. Later, however, considering that double-precision software emulated libraries require many more accesses and have a performance up to 50 times slower than a hardware Floating-Point Unit (see [27]), I switched to using the Cortex-R52 Advanced SIMD and floating-point unit.

5.6 Serial Debug Messages

The purpose of the SGPO peripheral is to output debug text messages and even though it could have been also integrated into the SoC, I replaced it for the UART available in DAHLIA. The main reason being the extra work needed to integrate it into the design and have a connection from the evaluation board to the computer. As the SGPO uses a FIFO scheme, this design choice is likely to introduce some extra latency during the software execution but it is not critical once text messages are output only during the initialization phase and once a second during the tracking phase. Furthermore, it is included only in debug mode.

5.7 Time Management

One of the observables in a GNSS application is the receiver time, hence time functions are constantly invoked during the execution of the receiver software. In AGGA-4, the DSU is responsible for providing timing information and management to the LEON processor. For DAHLIA, I implemented a custom library based on the CPU counter-timer available on the Cortex-R52 and replaced the calls to the DSU with this library. The CPU counter-timer is placed in a clock domain slower than the one that the CPU cores are at. In the FPGA prototype, this counter runs at 6.35 MHz. This means that the time management library is limited to a precision of $t = \frac{1}{6.35 \cdot 10^{-6}} \approx 157ns$. This precision is enough for the

delay functions as they are used during the software initialization only, to make the CPU wait for the reset and initialization of the GNSS Core, which is not timing critical. On the other hand, the tracking loops highly rely on timestamps to adjust the carrier and code NCOs during signal tracking and this limited precision is likely to introduce phase errors and mismatches, thus limiting the accuracy that can be achieved with the FPGA prototype. Nevertheless, the CPU counter-timer in the ASIC design is in a clock domain at 80 MHz, leading to the same time precision that the current system with the LEON processor offers, known to be enough for the GNSS application.

5.8 Interrupt Handling

The Cortex-R52 defines three types of interrupts: Shared Peripheral Interrupts (SPIs), Private Peripheral Interrupts (PPIs), and Software Generated Interrupts (SGIs). In AGGA-4, the Channel Controller Interrupt is triggered by a hardware timer. To replace this interrupt's source, I set the Hypervisor Timer to trigger a PPI when an overflow occurs and wrote an interrupt routine that reloads the timer every time it is executed, with the value corresponding to a 500 us period. This routine calls the Channel Controller Interrupt handler. As the Measurement Epoch Interrupt and the One Second Interrupt are software-generated, I attached each one to an SGI. The former to SGI0 and the latter to SGI1. I then wrote simple functions to force the activation of these interrupts and replaced them in the Sensor Module code where they were previously forced in AGGA-4. To not interfere in the algorithm's logic, I replaced the interrupts' masking and enabling at the same places where they were before. Since I did not wire any hardware peripheral from the eFPGA to the GIC, no SPI was used.

Even though fast interrupts (FIQs) requests were available, I opted for signaling all three interrupts as normal interrupt requests (IRQs), thus setting them to the GIC's Group 1 of interrupts. Given that a signal processing algorithm is executed inside the interrupt handlers, the interrupt invoking latency is negligible when compared to the handler's execution time. FIQs were left for higher priority sources, as watchdog timers, semaphores, and others. Finally, to allow interrupt preemption, I configured the interrupts' priorities in the same way as described in Table 5.2. At this point, the full execution of the Sensor Module was recovered.

5.9 Address Map

The Receiver Software makes use of three main structures. One related to the LEON configuration registers, one to the DSU registers, and one to the GNSS Module. They are combined structs to append in one variable all registers with the right offset. As presented in the last chapter, I had to shrink the addresses in the GNSS Module so they would fit the 4 MB LLPP range. In addition, I reduced the number of channels down to 4. Therefore, I had to restructure the GNSS struct by setting new memory offsets and set the base address of the GNSS struct to point to the base address of the LLPP. As the DSU and the LEON registers were not included in the new design and their functionalities were accordingly replaced with features of the DAHLIA SoC, I removed all references to them from the code using compiler directives.

5.10 Configuration Parameters

Some software definitions were fixed for the application type and some depended on the building tools used. To comply with the new system, I had to redefine several of these parameters. The number of channels and input modules present in the GNSS Core, so the loops and minimum and maximum checks would properly work; the clock frequency used to calculate timespans to the frequency of the CPU-counter-timer; several compiler directives.

5.11 Navigation Dependence

At system start, all channels are in an idle state and after channel predictions have been received, the Channel Controller enters a searching state and starts sweeping the locally generated codes and carrier trying to find the incoming signal from the predicted satellite. In order to bypass these predictions coming from the not present Navigation Module, I created a function to locally force these predictions into the Channel Controller. This method is called after the initialization phase has finished and before entering the endless loop, and forces all channels into searching state.

5.12 Instructions and Data Placement

Regarding instructions and data placement, at first, I allocated code and data to the eRAM and setup the stack on TCMC. Later, aiming to exploit the full capabilities of the

Cortex-R52, I analyzed a new memory layout. The goal was to take advantage of the TCMs for allowing run-time critical software to be placed on these fast memories while non-time-critical software to be kept in the eRAM. The single-cycle access TCMs provide a much faster code execution as instructions and data do not have to undergo the shared interconnect to access the memory and it is in the same clock domain as the CPU.

The first step was to define which components in the Sensor Module should be considered as run-time critical. Among the interrupts in Table 5.1, the Channel Controller Interrupt is the most recurrently invoked and it demands intensive calculations in the tracking loops, to search and lock signals. It is, therefore, reasonable to consider the Channel Controller/-Manager that runs inside this interrupt handler as run-time critical. Another time-sensitive block is the GNSS Core Device Driver, which contains driver functions to control the registers in the GNSS Module. On every Channel Controller Interrupt, new values are written to the carrier and code NCOs inside the GNSS Core, so one should expect performance improvement by optimizing this task. Therefore, the GNSS Core Device Driver was also considered as run-time critical. Besides the critical parts in the GNSS software, I defined as run-time critical also the reset vector table, the interrupt service routines, and the C libraries such as *libc*, *memcpy*, and *math*.

In order to redefine the target memory layout of the output binary file, I needed to interfere in the linking phase of the code build process. The first step was to split all object files generated after the compilation and assembling of every source file into two groups: **criticalsw_objects**, the group containing all the software blocks defined as run-time critical; and **noncriticalsw_objects**, the group containing the remaining files, categorized as non-run-time critical. Each group is then first linked to a single object: **libcritsw.o** and **libnoncritsw.o**. These two objects are next passed to the *GNU linker* and a custom linker script is used to drive it and generate the output file. The instructions (.text section) of the object **libcritsw.o** are mapped to the TCMA while the data sections (.bss and .data) put on TCMB. All the sections of the object **libnoncritsw.o** are allocated to the eRAM. Finally, the stack is placed at the end of TCMC. The software does not perform any dynamic allocations, so there is no need for allocating a heap. The final memory layout is shown in Table 5.3.

It will be shown in the next chapter that, in theory, the whole Sensor Module could fit inside the 3 TCM's. Even so, I did not consider this approach because the software for the next generation of receivers aims to process many more GNSS signals such as Galileo and Glonass, rather than only GPS C/A in L1. Thus, it is reasonable to expect greater

Memory Unit	Memory Content
TCMA	Reset vector table Interrupt handlers System libraries (math, libc, memcpy) Channel Controller (text) GNSS Core Device Driver (text)
TCMB	Channel Controller (data and bss) GNSS Core Device Driver (data and bss)
TCMC	Stack
eRAM	Non-critical code (text, data and bss)

Table 5.3: Memory layout.

memory usage.

Even though accesses to the TCMs are unified, meaning that both instruction and data can be placed on the same memory, separating instruction code and data code into separate TCMs can lead to performance improvement. The reason is that all 3 TCMs can be accessed simultaneously. Therefore, splitting code and data into different TCMs avoids waiting states being introduced on concurrent accesses from the data and instruction buses, once each TCM can only be accessed by one source at a time. Furthermore, according to the Cortex-R52 Technical Reference Manual, TCMA is optimized for the reset vector table and exception handler code and, when using bus ECC protection, the TCMA is also optimized for instruction fetching while TCMB and TCMC are optimized for data fetching. This explains the design choices I made even if ECC protection is optional and not present in DAHLIA.

In a case where an external device accesses the TCMs via the AXIS interface, such as another CPU core or a DMA controller, a better memory layout can be designed. For instance, data accessed by different masters could be placed into different memories making simultaneous accessess possible.

5.13 Profiling

To allow performance profiling and dynamic code analysis, I wrote a HAL library for the Cortex-R52 Performance Monitor Unit (PMU) and configured it to count events such as LLPP read and write requests, instruction and data cache misses, and CPU cycles between consecutive integration epochs.

6 Evaluation

In this chapter, I first present code correlation tests, where I assure that the PRN code generator is producing a signal compatible with the local replica in the channels and meaningful observables are produced. Then, I evaluate the latency of accesses to the eRAM which are done through the AXIM interface and to the GNSS Module through the LLPP. Finally, I present a code profiling of the GNSS Receiver Software, evaluating its memory usage and estimating the channel tracking performance for both the FPGA-based prototype and the ASIC.

6.1 Correlation Tests

I performed the code cross-correlation tests using the software Mentor QuestaSim, where I set a testbench to input control signals to the GNSS Core modified with the PRN code generator, as previously depicted in Figure 4.4. By sending inputs via a TCL script, I first configured *Channel 0* to be active and preloaded its primary RAM with the chosen PRN code. The code stored in this memory is forwarded by the Code Unit Generator towards the Correlator Unit. I selected the following channel configuration:

- $f_{clock} = 4.875$ MHz
- $f_{carrier} = 0$ Hz (input signal without carrier)
- $f_{code} = 1.023$ MHz
- Inegration epoch = 1023 chips

These settings lead to an integration epoch's period of $\Delta_{IE} = \frac{1023\,[chips]}{1.023\,[MHz]} = 1\ ms.$ I then set a delay in the PRN code generator to have its signal being output at the same clock cycle as the code from the Code Generator Unit, right at the beginning of a new integration epoch.

Figure 6.1 presents signal timing waves from a watchpoint placed at the input of the Correlator Unit, where it is possible to see the input code, at the bottom, matching exactly

the code generated in the channel, on top. Throughout an integration epoch, which covers 1023 PRN chips, only a few samples from the input generator differed from the code locally generated in the channel as highlighted in red in Figure 6.2. We can see that these mismatches last only one clock cycle, so we may expect good cross-correlation between both signals. Indeed, that is the case. In Figure 6.3, if we first look at the accumulator signal, we see random values when the input generator has not started yet and then the increasingly accumulated value result of the codes' correlation as the integration happens. At the end of this integration epoch, identifiable by a sharp spike in its signal wave, we see the accumulated value being flushed for the next epoch and the integration observables changing from very low values to very high values.

To have a quantitative comparison, a precise code match would lead to an integration value of:

$$IntValue_{max} = f_{clock} \cdot \Delta_{IE} = 4.875\,[MHz] \cdot 1\,[ms] = 4875 \tag{6.1}$$

Whereas, from the timing waves, we have the following integration observables:

- $IntValue\_EarlyEalry = 2715 \approx 55.7\%$
- $IntValue\_Early = 3739 \approx 76.7\%$
- $IntValue\_Prompt = 4763 \approx 97.7\%$
- $IntValue\_Late = 3963 \approx 81.3\%$
- $IntValue\_LateLate = 2939 \approx 60.3\%$

We verify a very high correlation value in the prompt signal that perpetuates over the following integrations epochs, once the PRN generator loops over other 1023 code chips as new integration epoch begins. This means that once the signal is tracked, the tracking loops do not need to modify the code frequency in the Code Generator Unit. This is a good result because we can proceed to test cases with a replicable input scenario, removing the uncertainty usually present during the acquisition of GNSS signals. We also verify that the correlation observables follow the trend in Figure 2.3, that is, once the signal is tracked we have a correlation peak in the prompt signal that diminishes as we move towards early and late samples. Lastly, the period of an integration epoch seen in the timing waves is $\Delta_{IE} = 999999\,ns \approx 1\,ms$, which in line with our expectations.

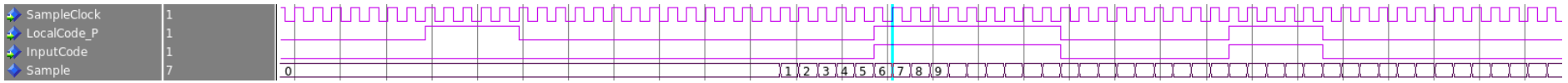

Figure 6.1: Input PRN code matching exatcly the code generated inside the channel.

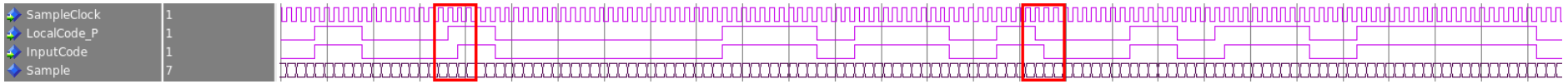

Figure 6.2: Mismatches between input and channel PRN codes.

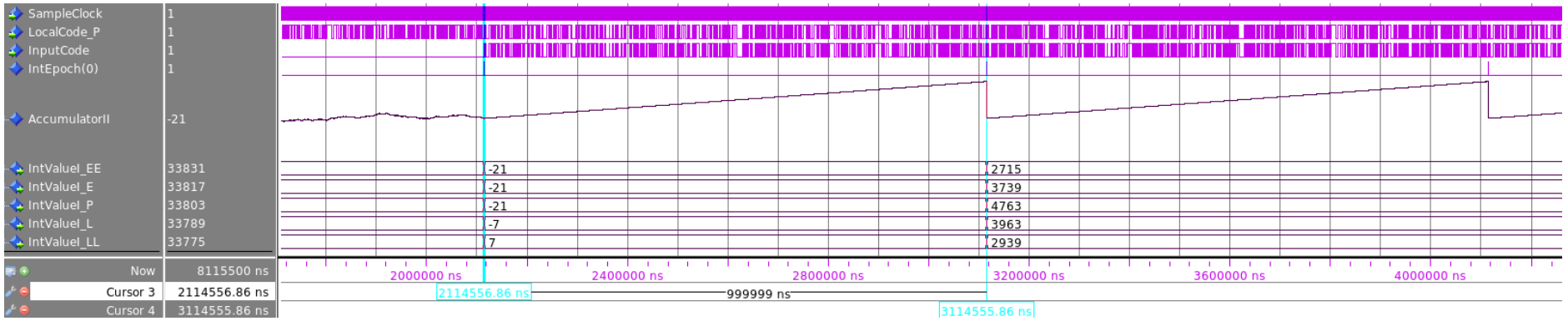

Figure 6.3: Result signals of cross-correlation between input and channel PRN codes.

6.2 Memory System Latency Analysis

To perform latency analysis I used the Mentor QuestaSim software with the DAHLIA SoC design using the Arm Design Simulation Model (DSM) for the Cortex-R52. The DSM offers full device functionality, meaning that the simulation model fully matches the architecture and functionality of the RTL model. Additionally, it is cycle-accurate, exhibiting the same intra-cycle timing as the RTL model.

I set up a testbench with a command sequence to initialize all modules in the SoC and configure the *R52 Core-0* to incrementally fetch instructions from the eROM and, at the beginning of the simulation, load the eROM content with an external HEX file. In this way, I could externally compile a small test case in Assembly language, and generate a HEX file with the instructions to be executed by the CPU core.

6.2.1 eRAM Transactions

When data are loaded into cache memory, access times for subsequent loads and stores are reduced, due to the one-cycle access, resulting in overall performance benefits. An access transaction to information already in a cache is known as a cache hit while other accesses are called cache misses. When a cache miss occurs, the processor has to go fetch the instruction or data in the device or external memory, requiring many more CPU cycles to complete the task.

The tests presented in this section aimed to analyze the latency of accesses of data stored in the eRAM, representing the case where a data cache miss occurs, to then compare with the case as if these data were located in a TCM. Four test cases were considered: a read of a single word, 32 bits; a burst read of two single-words, 64 bits; a write of a single word, 32 bits; a write to four words, 128 bits. As the R52 core accesses both eRAM and eROM via the AXIM interface, I opted for small-size burst transactions so there would be no instruction fetching on the eROM between reads and writes to the eRAM. The signals analyzed were from the CPU side, i.e. from the AXIM interface.

Read

The timing waves of a read transaction of 32 bits are shown in Figure 6.4. It is possible to identify a reading request being issued by the R52 core when the ARRVALID signal, which is steadily LOW, becomes HIGH and the ARADDR signal is set to a memory address in the eRAM. The read data are valid and in the read bus, RDATA, 22 CPU cycles later, identifiable by a data change in the read bus and by the pulse in the RVALID signal.

Therefore, a read to a single word takes:

$$\Delta_{read} = \frac{n_{cycles}}{f_{CPU}} = \frac{22}{50\,[MHz]} = 0.44\,us \tag{6.2}$$

Burst Read

A burst read of two words is presented in Figure 6.5. By following the same analysis procedure done in the last case, we see that the CPU reads the first word 22 cycles after issuing the request and the second word 2 cycles later, thus summing 24 cycles to complete the transaction, that in time is $0.48\,us$. It is important to notice that a burst read transaction size larger than two words has to be interrupted by a new CPU instruction fetch, leading to a case similar to a new transaction.

Write

Considering now a case of a write transaction of a single word, the test results are shown in Figure 6.6. We can spot the moment when the write request is issued by looking at the AWADDR signal when it changes to an address inside the eRAM and the data become available on the WDATA bus. The confirmation that these data have been successfully written in the memory is identifiable by the BVALID signal switching from LOW to HIGH. This takes 23 CPU cycles, that in time is $0.46\,us$.

Burst Write

A burst write of 4 words is shown in Figure 6.7. Again, following the same procedure, we find that this transaction is concluded in 31 CPU cycles. In time, it is $0.62\,us$.

Analysis

For performance comparison between the TCM and the eRAM, considering the case of a single read operation to a TCM, the R52 takes one cycle to finish it, since these memories have a zero wait-state. For the eRAM, a read operation also takes one cycle in the case of a cache hit. If we have instead a cache miss, the same operation takes at least 22 cycles to complete. It is important to notice that these test cases consider the best possible scenario where the AXI bus is not being used by other SoC peripherals, so there is no waiting time from the moment that a request is sent by the R52 core until it is actually propagated in the bus. In a more realistic scenario, the actual latency of each transaction is likely to be

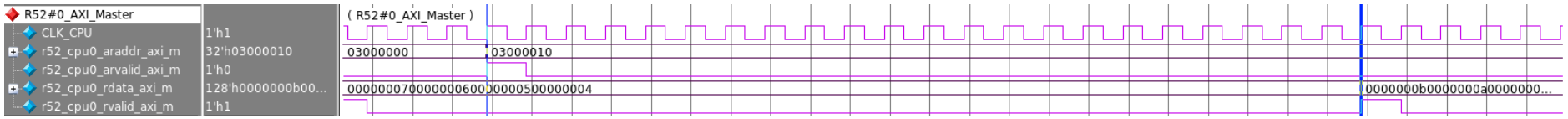

Figure 6.4: Read operation of a single word located in the eRAM.

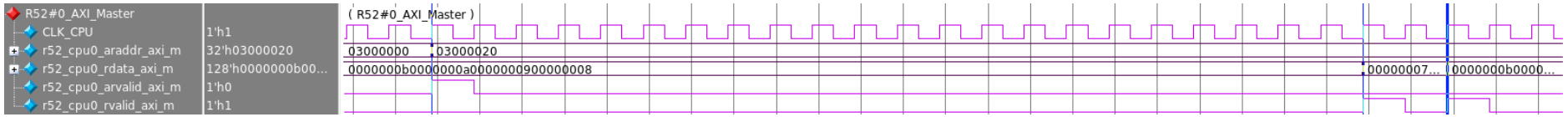

Figure 6.5: Burst-read operation of two words located in the eRAM.

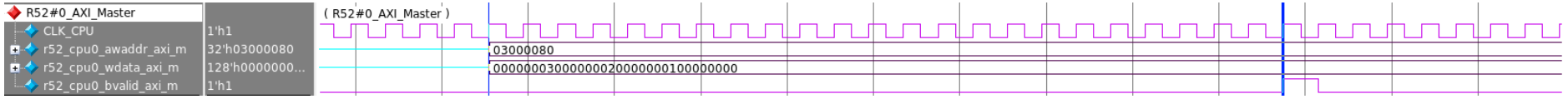

Figure 6.6: Write operation of a single word located in the eRAM.

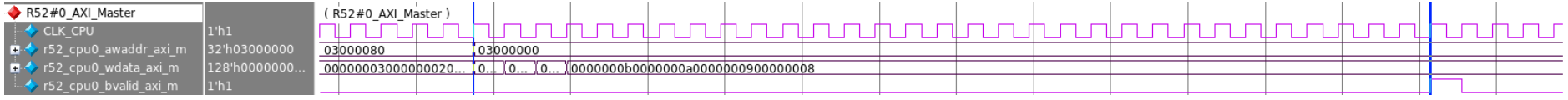

Figure 6.7: Burst-write operation of four words located in the eRAM.

greater and will highly depend on the system and how much the AXI interconnect is being stressed.

Altogether, it is verified that a cache miss of data placed in the eRAM will imply in an access time at least 22 times slower than if these data were placed in a TCM but, even though this performance difference is notorious, one should expect a great impact on the overall software performance only if cache misses frequently happen and the CPU core has to then fetch the data from the external memory. Thus, it is important to estimate how often this situation is encountered. I perform such an evaluation in section 6.3.2.

6.2.2 LLPP Transactions

Moving onto a latency analysis of LLPP transactions, the scope of the following tests was to estimate how long the R52 core takes to access registers in the GNSS Core. As the GNSS Module is not able to perform burst transactions, only single read and write operations were considered. The signals of the intra-components (see Figure 4.5) that perform each operation were analyzed to break down the overall latency and expose possible modules that could be optimized.

In Figure 6.8 and Figure 6.9, the *LLPP0_Master* group of signals refers to the LLPP interface on the R52 core, where we can spot when a read or write operation request is sent by the CPU core. The *LLPP0_Slave* represents the AXI interface in the AXI4-to-AHB-Lite bridge, i.e. the AXI signals after they have passed through the interconnect. The group labeled as *Sync_module* is the AHB slave interface of the GNSS Module. Finally, the GNSS Core group refers to the signals at the front interface on the GNSS Core such as chip select signals, address and data buses.

Read

A read operation of a 32-bit register inside GNSS Cores's *Channel 0* is presented in Figure 6.8. The read request is sent by the R52 core when the signal ARVALID in *LLPP_Master* switches from LOW to HIGH. It then takes 21 CPU cycles for the request to reach the AXI4-to-AHB-Lite bridge. This latency is due to the interconnect synchronization between the different clock domains. Now in the GNSS Core clock domain, the bridge spends 2 GNSS Core cycles on the protocol translation until the AHB request is available for the *Sync_module*. This moment is identifiable by the address change in the *Sync_module*'s HADDR bus. This component performs an address decoding, then the register address is sent out to the GNSS Core 2 clock cycles later. The GNSS Core takes 2 more cycles

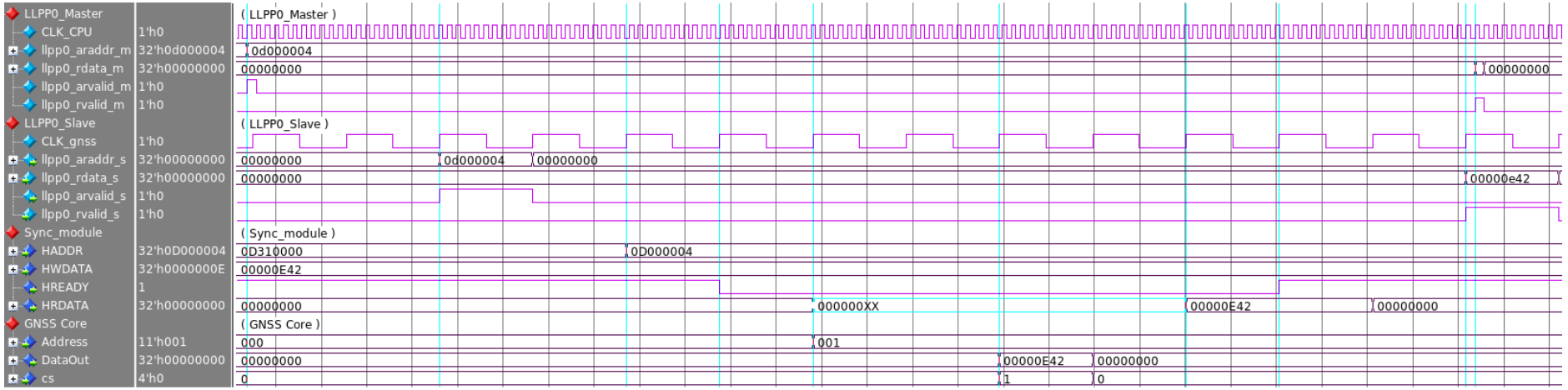

Figure 6.8: Read operation of a register inside the GNSS Core through the LLPP.

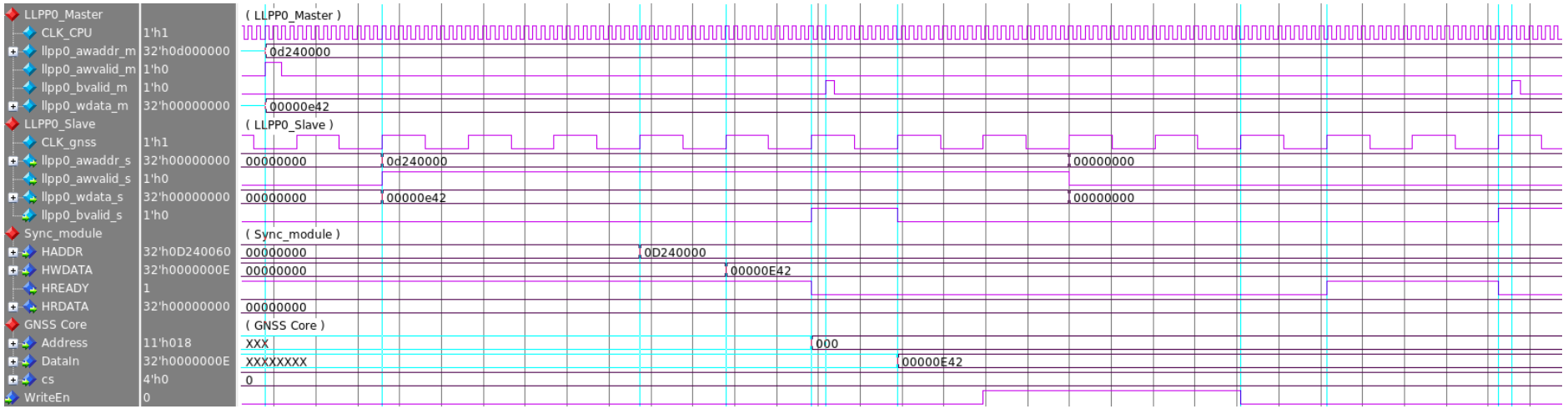

Figure 6.9: Write operation of a register inside the GNSS Core through the LLPP.

to fetch and output the data on the *DataOut* bus. These data are available back in the *Sync_module* 2 cycles later. Another 3 cycles until finally reach the bridge's AXI interface. The read operation finishes 1 CPU cycle later after the synchronization has finished and the data are finally available in the CPU core. Table 6.1 summarizes the latency introduced on each stage of the transaction.

Interface	Latency	Latency
R52 core -> AXI4-to-AHB Bridge	21 CPU cycles	0.420 us
AXI4-to-AHB Bridge -> Sync Module	2 GNSS Core cycles	0.410 us
Sync Module -> GNSS Core	2 GNSS Core cycles	0.410 us
GNSS Core -> GNSS Core	2 GNSS Core cycles	0.410 us
GNSS Core -> Sync Module	2 GNSS Core cycles	0.410 us
Sync Module -> AXI4-to-AHB Bridge	3 GNSS Core cycles	0.615 us
AXI4-to-AHB Bridge ->R52 core	1 CPU cycle	0.020 us

Table 6.1: Latency introduced by each component in a read operation through the LLPP.

It is worth it reminding that the Sync Module is not performing any sort of synchronization in this new system since the AHB interface and GNSS Module are placed in the same clock domain. This task is performed by the interconnect. The synchronization time is variable and depends on the relative phase mismatch between the clock edges of the CPU clock and the GNSS Core clock at the time when a request is sent. Nevertheless, the average value can be estimated. For the crossing from the CPU clock to the GNSS Core clock, we have:

$$1 \cdot T_{CPU} + 1.5 \cdot T_{GNSS} < \Delta_t < 1 \cdot T_{CPU} + 2.5 \cdot T_{GNSS} \tag{6.3}$$

$$0.338\,us < \Delta_t < 0.533\,us \tag{6.4}$$

That on average will result in $\Delta_t \approx 0.435\,us$. On the other hand, for the crossing from the GNSS Core clock to the CPU clock, we have:

$$1 \cdot T_{CPU} < \Delta_t < 2 \cdot T_{CPU} \tag{6.5}$$

$$0.020\,us < \Delta_t < 0.04\,us \tag{6.6}$$

That on average will result in $\Delta_t \approx 0.030\,us$. Summing the latency introduced by each stage and considering average synchronization times we get the overall read transaction latency of $\Delta_{read} \approx 2.720\,us$.

We notice that the bridge is responsible for a big part of the overall latency as it requires 5 GNSS Core clock cycles during the whole transaction. A more adequate solution would be to remove this component and substitute the Sync Module by an AXI4 slave interface, sparing these 5 extra clock cycles and resulting in a transaction latency of $\Delta_{read} \approx 1.695us$, more than 37 % faster than the current approach.

A GNSS Core clock frequency of 4.875 MHz was considered for these calculations, which translates to the GNSS Module running at 58.5 MHz in the ASIC. If this frequency could be pushed to 80 MHz or even 100 MHz, which for the FPGA-based prototype would be approximately 6.667 MHz and 8.333 MHz respectively, we would have $\Delta_{read@6.667MHz} \approx 2.000\,us$ and $\Delta_{read@8.333MHz} \approx 1.610\,us$, by following the same procedure done above and considering the bridge included. If we instead consider an AXI4 slave interface in the place of the bridge and Sync Module, we get $\Delta_{read@6.667MHz} \approx 1.250\,us$ and $\Delta_{read@8.333MHz} \approx 1.010\,us$. Table 6.2 summarizes the afore-mentioned results.

	AXI4-to-AHB bridge	AXI4 slave interface
GNSS Module @4.875 MHz	2.720 us	1.695 us
GNSS Module @6.667 MHz	2.000 us	1.250 us
GNSS Module @8.333 MHz	1.610 us	1.010 us

Table 6.2: Latency summary of a read operation through the LLPP of a register located in the GNSS Core.

Write

Figure 6.9 shows a write operation of a 32-bit register inside *Channel 0* of the GNSS Core. By doing the same analysis of the signal timing waves, we get the results presented in Table 6.3 and Table 6.4. This case differs from a read operation because as soon as the Sync Module sends out the write request to the GNSS Core, the bridge signals back to the R52 core that the transaction has finished even if the register has not been actually written yet. Therefore, the latencies reported in Table 6.4 do not account for the overall latency that turns out to be the sum of the stages reported in Table 6.3. Naturally, a following-up write request will last longer than accounting only the components in Table 6.3 as the Sync Module would have to wait until the GNSS Core has finished writing the data from the first operation. However, this situation is not encountered during the execution of the timing-critical routines of the GNSS Receiver Software. The ChannelController, while in

tracking state, updates the values of the carrier and code NCOs with some data processing in between these two write transactions, thus giving room for the GNSS Core to properly finish handling a write request before another one is made. This means that the effective CPU halt time is the sum of the latencies reported in Table 6.3, that is $\Delta_{write} \approx 1.490\,us$. As for the case of a read operation, I considered the average synchronization times.

Interface	Latency	Latency
R52 core -> AXI4-to-AHB Bridge	14 CPU cycles	0.280 us
AXI4-to-AHB Bridge -> Sync Module	3 GNSS Core cycles	0.615 us
Sync Module -> AXI4-to-AHB Bridge	2 GNSS Core cycles	0.410 us
AXI4-to-AHB Bridge ->R52 core	1.5 CPU cycles	0.035 us

Table 6.3: Latency introduced by the first components group in a write operation through the LLPP.

Interface	Latency	Latency
Sync Module -> GNSS Core	2 GNSS Core cycles	0.410 us
GNSS Core -> GNSS Core	4 GNSS Core cycles	0.820 us
GNSS Core -> Sync Module	2 GNSS Core cycles	0.205 us

Table 6.4: Latency introduced by the second components group in a write operation through the LLPP.

We verify also in this case that the bridge is responsible for the majority of the transaction time. If it were replaced by an AXI4 slave interface, instead of 5 GNSS Core clock cycles spent by the bridge, only 3 cycles would be necessary to receive the request, the data, and signal back to the core that the transaction has finished. The overall write latency would then be $\Delta_{write} \approx 1.080\,us$, an improvement of more than 27%.

Estimating this latency for the case where the GNSS Core is running at a higher frequency, we get $\Delta_{write@6.667MHz} \approx 1.100\,us$ and $\Delta_{write@8.333MHz} \approx 0.890\,us$ if we consider the bridge, and $\Delta_{write@6.667MHz} \approx 0.800\,us$ and $\Delta_{write@8.333MHz} \approx 0.650\,us$ if we consider an AXI4 slave interface. Table 6.5 summarizes the results.

	AXI4-to-AHB bridge	AXI4 slave interface
GNSS Module @4.875 MHz	1.490 us	1.080 us
GNSS Module @6.667 MHz	1.100 us	0.800 us
GNSS Module @8.333 MHz	0.890 us	0.650 us

Table 6.5: Latency summary of a write operation through the LLPP of a register located in the GNSS Core.

Analysis

By extending the analysis procedure followed above to a more general case, I obtained in MATLAB the plots in Figure 6.10 and Figure 6.11. They present curves of how the latency of read and write transactions vary as a function of the GNSS Core clock frequency. We verify that removing the bridge considerably decreases the latency in these accesses. In both cases, the access time exponentially decreases as the GNSS Core clock frequency is increased and to evaluate how this affects the overall GNSS Receiver Software performance, I needed to evaluate the overall CPU load for different GNSS Core clock frequencies. I perform such an evaluation in section 6.3.2.

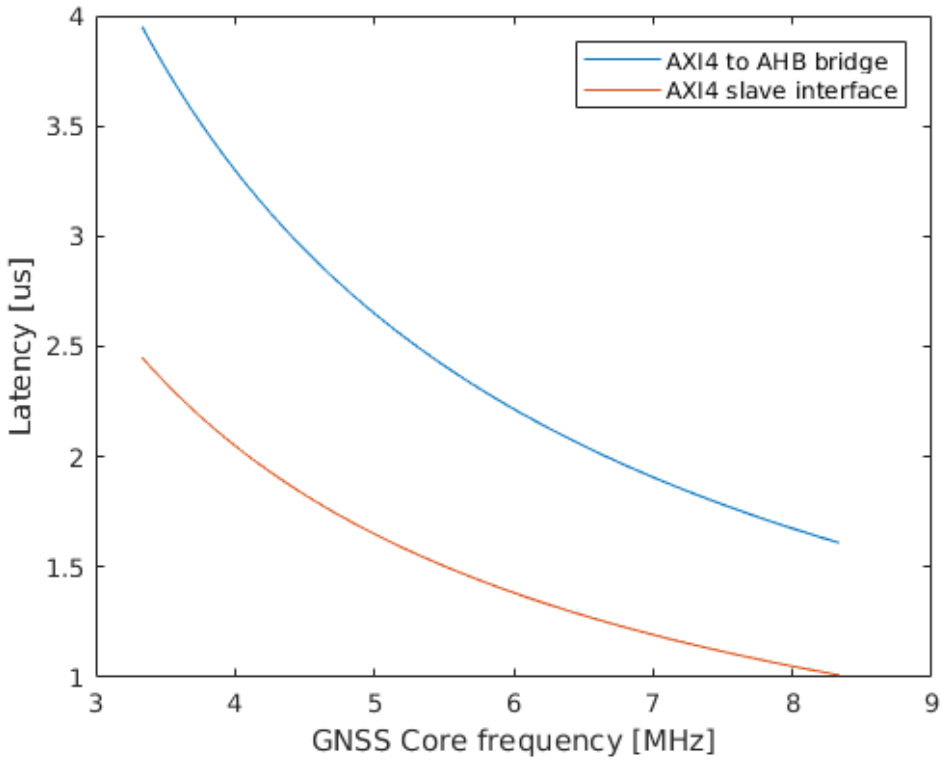

Figure 6.10: Latency of a read transaction through the LLPP over the GNSS Core clock frequency.

6.3 Code Profiling

I obtained a code profiling by instrumenting the binary executable using the GCC toolset and through the statistics produced by the event-based Cortex-R52 PMU. Even though these events were exported to the Embedded Trace Macrocell (ETM), they could not be accessed due to the unavailability of a trace connection in the evaluation board. Nevertheless, the CPU core can access the event counters in run-time so I set the counter values to

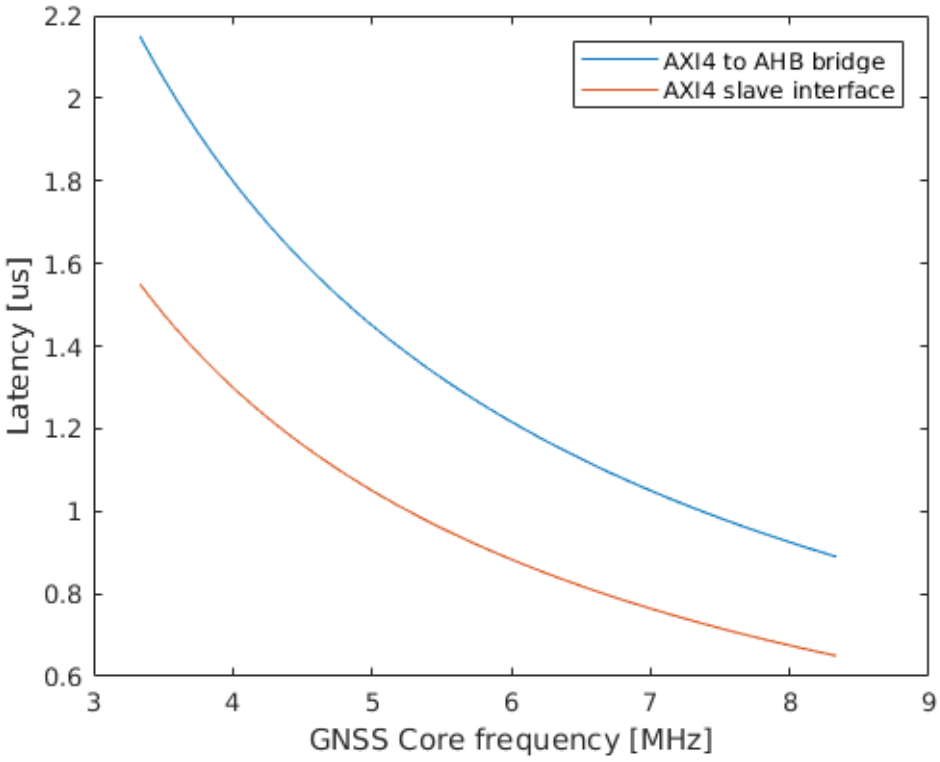

Figure 6.11: Latency of a write transaction through the LLPP over the GNSS Core clock frequency.

be stored in an unused memory region of the eRAM which I could then export via Arm DS debug interface. All tests described in this section were performed with the GCC -O2 optimization level, which provides maximum optimization resulting in a smaller and faster executable.

6.3.1 Memory Budget

Table 6.6 presents the size of each segment of the output binary. The text section refers to executable instructions; the data segment contains all initialized global variables and static variables; the bss accounts for all global variables and static variables that are initialized to zero or do not have explicit initialization in the source code without including the stack, which is reported separately.

Text	Data	Bss (no stack)	Stack
88488 bytes	9324 bytes	100696 bytes	65536 bytes

Table 6.6: Segments' size of the code output binary.

The memory usage in the case where all software is located in the eRAM is reported in Table 6.7, where we see that the whole software needs less than 5% of the memory available

in the eRAM. The stack is placed in TCMC.

Memory Unit	Available	Used (bytes)	Used (%)
TCMC	128 KB	65536	50
eRAM	4 MB	198508	4.7

Table 6.7: Memory budget with all code loaded in the eRAM.

Table 6.8 presents how much each memory unit is occupied when considering the splitting of run-time critical and non-critical code into different memories. The eRAM usage is reduced to 2.5% while all the TCMs are far from being full.

Memory Unit	Available	Used (bytes)	Used (%)
TCMA	128 KB	60520	46.2
TCMB	128 KB	32064	24.5
TCMC	128 KB	65536	50
eRAM	4 MB	105924	2.5

Table 6.8: Memory budget with splitting of run-time critical and non-critical code into different memories.

Analysis

We notice that the use of TCMs is advantageous for reducing eRAM usage that is a shared resource between the other CPU cores. In fact, we see that the TCMs are sufficiently large to hold the entire GNSS software. Nevertheless, I did not consider this scenario because the code size could potentially increase in the case where the software processes signals other than GPS C/A in L1.

6.3.2 Signal Tracking Performance

Given the problems I encountered during the FPGA bitstream generation, I could not run tests on the GNSS Receiver Software with the GNSS Module integrated into the SoC design. For the software tests, there was an AXI Traffic Generator connected to the LLPP interface that guaranteed that every transaction could complete without errors even if with meaningless data. To still evaluate the signal tracking performance in the DAHLIA SoC, I moved onto estimating the CPU usage considering the access timing to registers inside the GNSS Core through the LLPP that were obtained during the simulations with the Cortex-R52 Arm DSM presented in the previous section. The DSM exhibits the same intra-cycle timing as the RTL model which ensures accuracy in these estimations.

As one may expect, the software flow highly depends on data coming from the GNSS

Core and is likely to run into error states if this component is not present, due to the lack of coherent data. To overcome this issue, I manually set all the conditional expressions to make the CPU core follow the expected program flow as if it were in a situation with coherent data. However, the scenario with channels in searching state proved difficult to ensure correctness in the software execution flow due to the constantly changing conditional trees. Thus, I considered all channels in tracking state.

As discussed before, the tracking state executes the PLL and DLL as part of the carrier and code loops which are responsible for the majority of the CPU load in AGGA-4. I did not consider the Measurement Epoch Interrupt and the One Second Interrupt because, out of the three interrupts in the GNSS Software, the Channel Controller Interrupt is the one that consumes by far more CPU resources since it runs 2000 times per second, which makes it the one worth analyzing.

I set the PMU to count the number of CPU cycles spent during 1000 Channel Controller Interrupt elapses together with the number of instruction and data cache miss occurrences. I computed the timespan of a single interrupt occurrence based on the CPU clock frequency. In every test that I performed with either all software allocated in the eRAM or split between TCMs and eRAM, no cache misses occurred whatsoever. Since a cache hit takes one CPU cycle to complete, both cases presented the same results. Therefore, all estimations in this section apply for both memory layouts. Table 6.9 gathers the results considering cases with different numbers of channels active at the same time, with the Advanced SIMD and floating-point unit enabled.

Channels used	Timespan (us)
0	9.575
1	20.297
2	31.685
3	42.591
4	53.396

Table 6.9: Measured Channel Controller Interrupt timespan for different number of active channels in tracking state.

Resorting to the least-squares method, I generated the plot in Figure 6.12. As we can see, all measured samples fit well the least-square curve meaning that the processing time of each channel is very deterministic and follows the relation described in equation 6.7.

$$\Delta_{t_{total}} = \Delta_{t_{offset}} + \Delta_{t_{chann}} \cdot n_{channels} = 9.5216 + 10.9936 \cdot n_{channels} \tag{6.7}$$

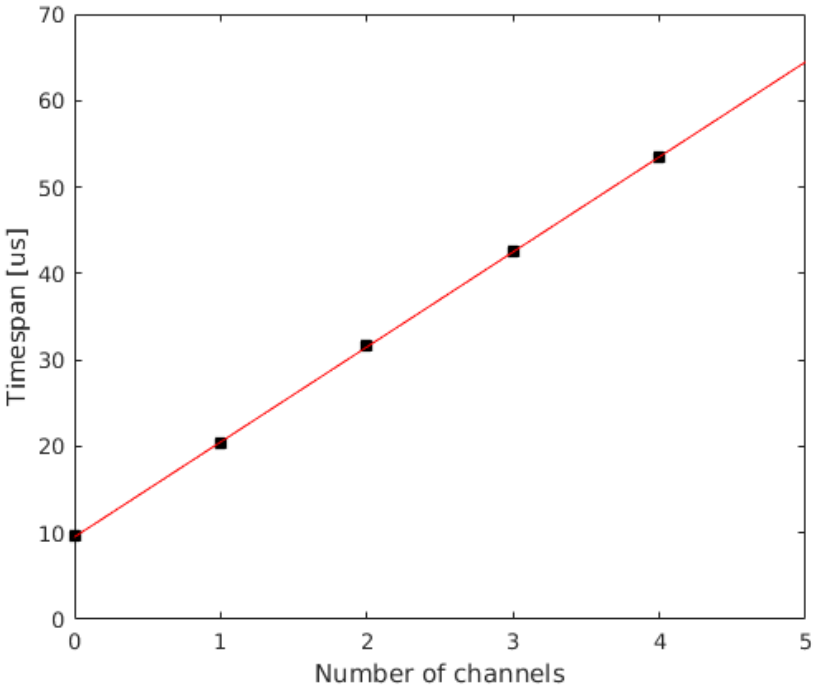

Figure 6.12: Least-square curve of the Channel Controller Interrupt timespan as a function of the number of active channels in tracking state, with the Advanced SIMD and floating-point unit enabled.

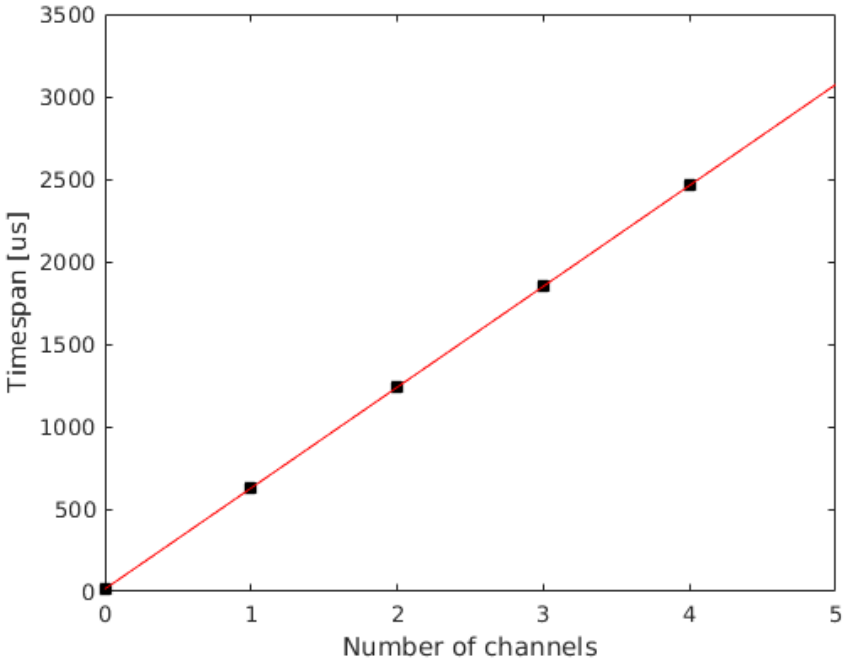

Figure 6.13: Least-square curve of the Channel Controller Interrupt timespan as a function of the number of active channels in tracking state, with software emulation floating-point support.

Doing the same analysis but considering software support for floating-points, Figure 6.13 was obtained. Overall, it presents a performance ≈ 55 times slower when compared to hardware floating-points handling.

Back to the Advanced SIMD and floating-point unit enabled, we see that the R52 core spends $\Delta_{t_{chann}} = 10.9936$ us processing each active channel in tracking state plus an overhead of $\Delta_{t_{offset}} = 9.5216$ us that is present even if there is no channel being processed. By inverting equation 6.7 and extrapolating to a timespan value up to 800 us that, considering that the integration epoch period was set to 1 ms, represents a CPU usage of 80%, we get the maximum number of channels that can be in tracking state at the same time, $n_{channels_{max}} = 69$. Figure 6.14 shows this projection. In this analysis, I assumed no cache misses occurring and accounted for 2 Channel Controller Interrupts happening during an Integration Epoch, with interrupt latency of a single interrupt being 11.11 us, which was measured in a separate test.

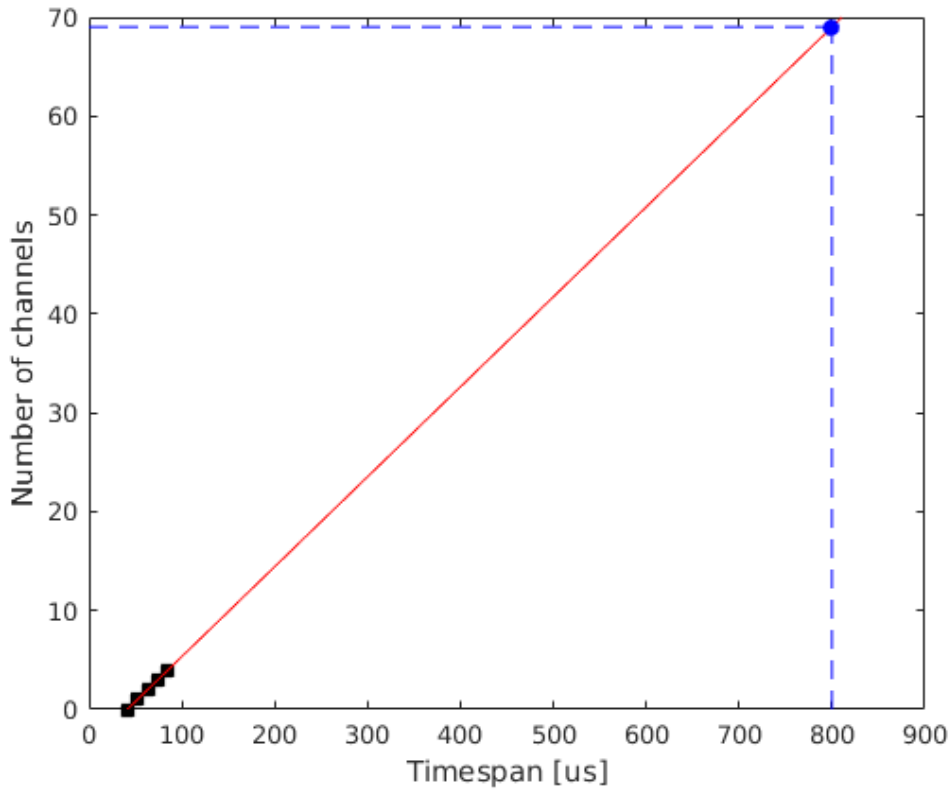

Figure 6.14: Number of tracked channels as a function of an interrupt timespan extrapolated up to a CPU usage of 80%.

These results come with the use of an AXI Traffic Generator connected to the LLPP which does not represent the actual influence of the GNSS Module latency in the software execution. With a setup similar to the previous test, I measured the CPU processing

time spent in single read and write transactions to the LLPP considering this AXI Traffic Generator, getting the results shown in Table 6.10. In this case, the results came from averaging 100000 samples and are presented in microseconds for easier comparison with the latencies obtained in simulation with the Cortex-R52 Arm DSM, summarized in Table 6.2 and Table 6.5.

LLPP read	LLPP write
0.35702 us	0.08000 us

Table 6.10: Latency of read and write operations with an AXI Traffic Generator connected to the LLPP.

On each integration epoch, the Channel Controller performs two write operations to registers in the GNSS Core for each channel in tracking state, which are new computed values for the carrier and code NCOs. Recalculating $\Delta_{t_{chann}}$ by substituting the time contribution of these transactions with the AXI Traffic Generator by the latencies of the each case reported in Table 6.5, we get the $\Delta_{t_{chann}}$ values shown in Table 6.11. Naturally, $\Delta_{t_{offset}}$ is not affected since it does not depend on the LLPP transactions inside the channels.

	AXI4-to-AHB bridge	AXI4 slave interface
GNSS Module @4.875 MHz	13.8136 us	12.9936 us
GNSS Module @6.667 MHz	13.0336 us	12.4336 us
GNSS Module @8.333 MHz	12.6136 us	12.1336 us

Table 6.11: Interrupt timespan recalculated with the write through the LLPP latency values obtained in simulation with the Arm DSM.

To then find out how many channels $n_{channels_{max}}$ that could be processed under 80% of CPU usage, I used the same extrapolation technique of inverting equation 6.7 but with the recalculated $\Delta_{t_{chann}}$ values. The results, rounded to the nearest smaller integer, are presented in Table 6.12.

	AXI4-to-AHB bridge	AXI4 slave interface
GNSS Module @4.875 MHz	55	58
GNSS Module @6.667 MHz	58	61
GNSS Module @8.333 MHz	60	62

Table 6.12: Maximum number of channels that could be tracked at the same time considering different GNSS Module configurations.

Performing the same analysis procedure for different GNSS Core clock frequencies, via MATLAB, I obtained Figure 6.15. It presents a general curve of the maximum number of channels that could be tracked in parallel for a given GNSS Core clock frequency.

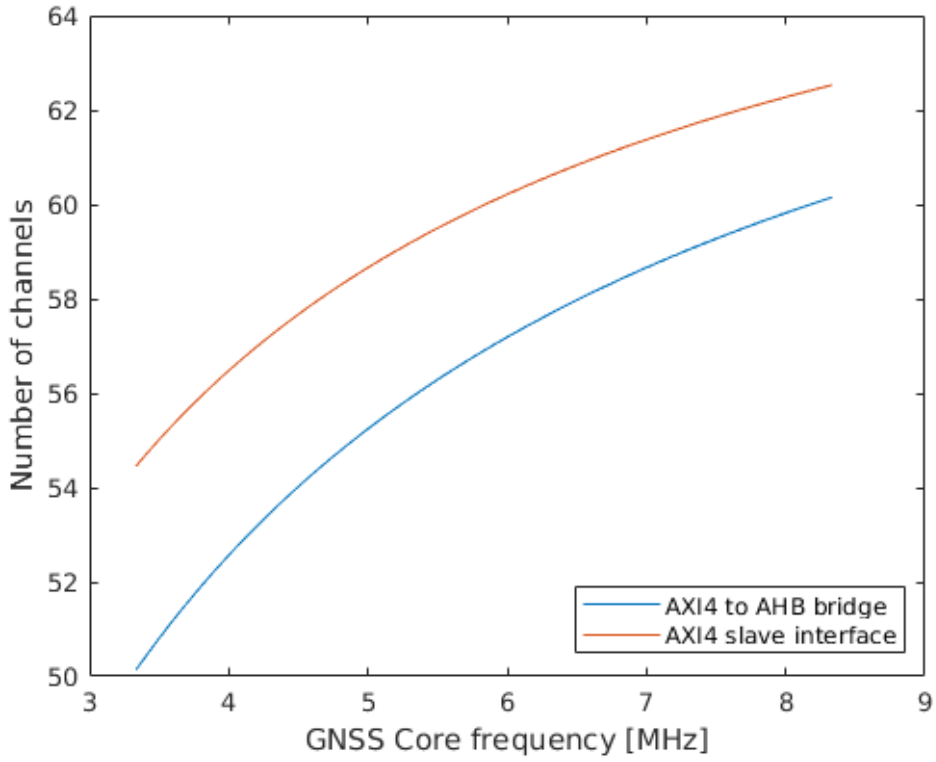

Figure 6.15: Maximum number of channels that could be tracked at the same time as a function of the GNSS Core clock frequency.

Extending these results to the ASIC scenario, where the clock signals are 12 times faster, and doing a similar analysis for this case, I could obtain the maximum numbers of channels that could be ideally tracked as a function of the GNSS Core clock frequency. The resulting plot is shown in Figure 6.16 while Table 6.13 highlights the values for the GNSS Core running at 58.5 MHz, 80 MHz, and 100 MHz. They point out to a processing capacity of more than 600 channels for every GNSS Module configuration considered so far. Naturally, this scenario is not realistic because the number of GNSS satellites is way below that.

	AXI4-to-AHB bridge	AXI4 slave interface
GNSS Module @58.5 MHz	660	700
GNSS Module @80 MHz	698	732
GNSS Module @100 MHz	721	750

Table 6.13: Maximum number of channels that could be tracked at the same time in the ASIC considering different GNSS Module configurations.

For a more realistic scenario, I then considered 72 channels being tracked in parallel and calculated the CPU usage for this case. This is the actual target for the next generation of GNSS receivers, AGGA-5. Figure 6.17 shows how the CPU usage varies with the GNSS

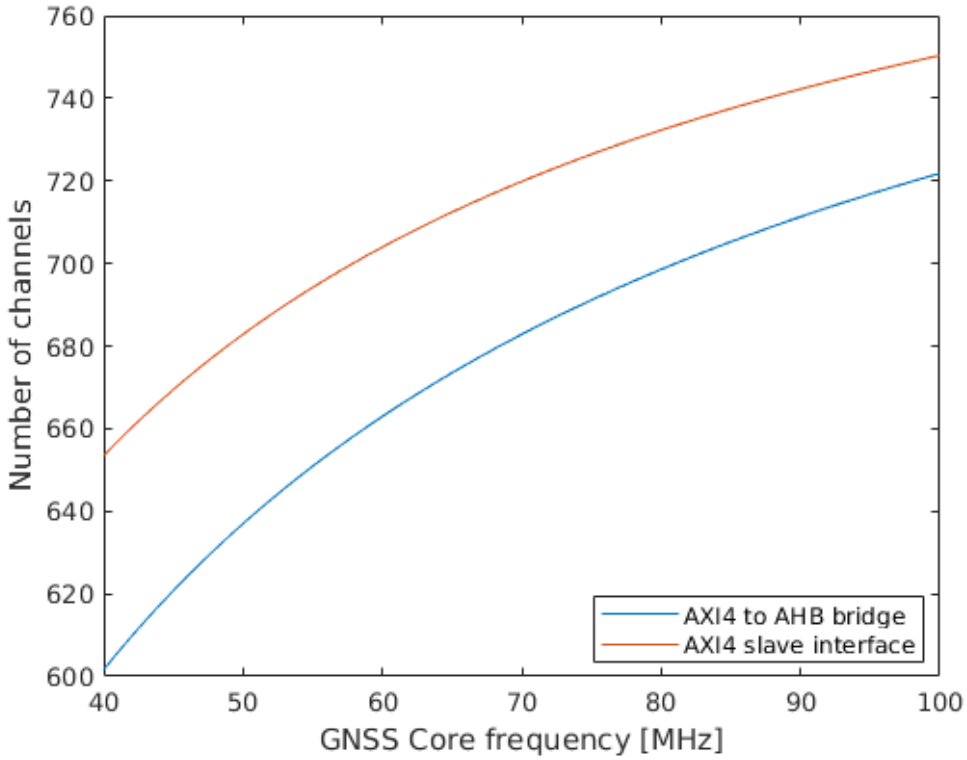

Figure 6.16: Maximum number of channels that could be tracked at the same time in the ASIC as a function of the GNSS Core clock frequency.

Core clock frequency for the case of the GNSS Module having an AHB interface while Figure 6.18 considers an AXI4 interface. The figures specify the amount of the total usage that is due to the interrupt latency, the channel processing task timespan without considering accesses to the GNSS Module, and the latency of these accesses themselves. Table 6.14 highlights the CPU usage for the GNSS Core running at 58.5 MHz, 80 MHz, and 100 MHz.

	AXI4-to-AHB bridge	AXI4 slave interface
GNSS Module @58.5 MHz	8.62%	8.13%
GNSS Module @80 MHz	8.16%	7.80%
GNSS Module @100 MHz	7.91%	7.62%

Table 6.14: CPU usage ins the ASIC with 72 channels in tracking state considering different GNSS Module configurations.

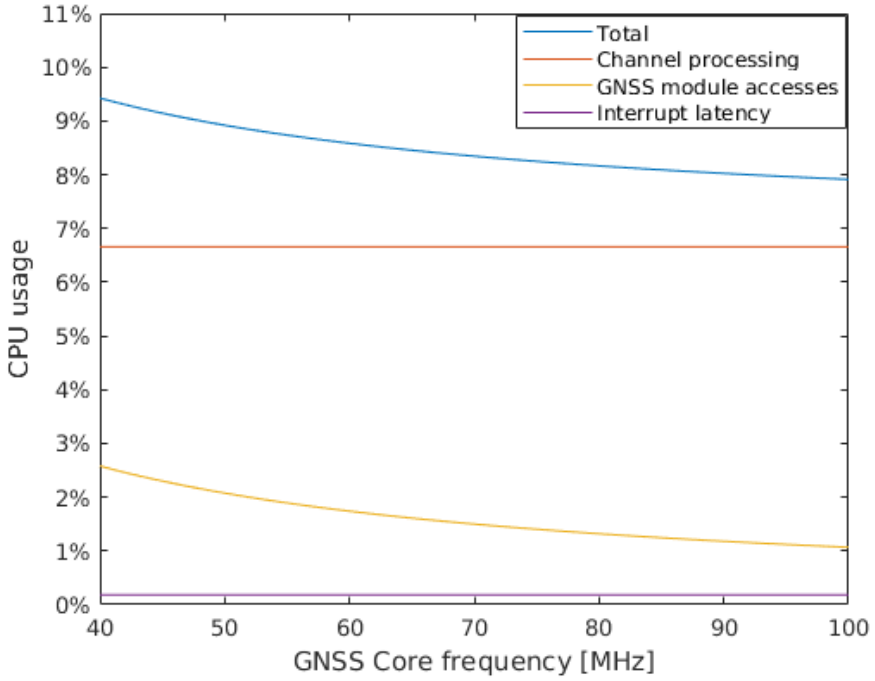

Figure 6.17: CPU usage ins the ASIC with 72 channels in tracking state as a function of the GNSS Core clock frequency, considering the GNSS Module with an AHB interface.

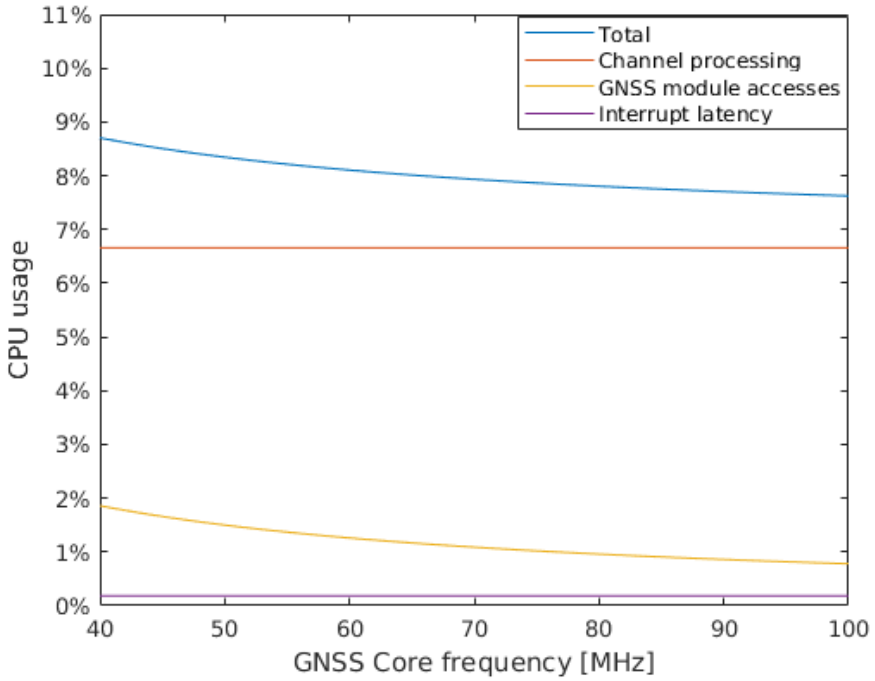

Figure 6.18: CPU usage ins the ASIC with 72 channels in tracking state as a function of the GNSS Core clock frequency, considering the GNSS Module with an AXI4 interface.

Analysis

The statistics for these estimations were collected for a maximum number of active channels varying from 0 to 4. Hence, those considering a number of channels greater than 4 assume no cache misses occurring whatsoever. Although this might be the case for several channels in tracking state, this assumption is not likely to hold for a very high number of channels such as the results found for a CPU usage of 80% in the ASIC. In this case, the performance is expected to drop since a cache miss is at least 22 times slower than a cache hit. Thus, it is reasonable to expect a lower maximum number of channels in the ASIC than what the estimations indicate. Nevertheless, these results are still valid if the allocation of run-time critical code into the TCMs is considered.

The software processing burden added by each channel proved to be very deterministic leading to a processing capacity in the FPGA-based prototype of 55 channels if we consider a CPU usage of 80% in the implemented solution that has the GNSS Module running at 4.875 MHz and interfaced with an AHB-to-AXI4 bridge.

Moreover, increasing the GNSS Core clock frequency from 58.5 MHz to 100 MHz indicates that the ASIC would be able to track about 50 channels more. Nevertheless, this represents less than 1% decrease of the overall CPU load and raise questions if worthwhile, given the greater power consumption that comes with increasing the design frequency.

6.4 FPGA Logic Gate Count

Table 6.15 reports a rough post-placement gate count of the GNSS Module in the FPGA prototype. The values are relative to the total FPGA resources. *GNSS Module* accounts for the AXI4-to-AHB bridge and does not include any channel in the Channel Matrix. *Channel* represents the gate count of a single channel, thus increasing with every added channel in the GNSS Core.

	LUTs	FFs	BRAMs
GNSS Module	0,26%	0,16%	0.15%
Channel	0,20%	0,13%	0.02%

Table 6.15: GNSS Module gate count in the FPGA prototype.

7 Conclusion

In this work, I sought to provide a GNSS sensor processing architecture for the novel DAHLIA SoC. Exploiting the Cortex-R52 LLPP has allowed accesses to the GNSS Module by the high-frequent channel processing task to be isolated from the main AXI interconnect, precluding the latter from the overhead of the writing transactions to the carrier and code NCOs and preventing the real-time channel task from being affected by traffic on the main AXI bus from other SoC components. The LLPP is a direct connection from the CPU core to the eFPGA, meaning that these transactions do not undergo the long shared multi-layer interconnect. Additionally, this choice tackles problems that would come along with the bus width of the 128-bit AXI interconnect, e.g. eliminating the need for redesigning the 32-bit GNSS Module to a 128-bit architecture.

When considering a maximum CPU usage of 80%, the implementation performed in the FPGA prototype exhibited an expected capacity for processing 55 channels in tracking state. Extending the results to the ASIC showed that DAHLIA SoC should be able to track 72 channels using less than 10% of CPU processing power for every case analyzed. In addition, increasing the GNSS Core clock frequency from 40 MHz to 100 MHz led to an expected decrease in the overall CPU load of around 1.5%, raising questions if worthwhile, given that a higher switching activity implies greater dynamic power consumption. Moreover, this study also showed that if the AHB interface of the GNSS Module along with the AXI4-to-AHB bridge were replaced by an AXI4 interface, read and write operations would be performed more than 37% and 27% faster, respectively.

The Cortex-R52 TCMs proved to be large enough to hold run-time critical routines including the main parts of the GNSS Receiver Software. This memory placement increases determinism in the channel processing and overcomes the performance drop expected when the data limit that the cache memories can process is reached, avoiding accesses to the eRAM that take at least 22 longer to complete. This memory layout also reduces eRAM usage, sparing it for other applications running in the SoC.

Altogether, the performance exhibited by DAHLIA SoC opens new possibilities for future GNSS sensor processing. The low CPU usage gives room for higher-level tasks such as

Precise Orbit Determination to be performed on-board the GNSS receiver using a single Cortex-R52 core. Also, new searching techniques can be exploited like multiple channels being set to search for the same satellite signal, potentially reducing the Time-To-First-Fix (TTFF) e.g. after a satellite reboot. Beyond sensor processing, the navigation software can be thought of being integrated into the same CPU core, given that it currently runs on an Arm Cortex-R5 processor.

7.1 Future Work

The implementation presented in this thesis has some limitations that are meant to be overcome in future works, which include:

- Complete FPGA synthesis and validate the results and estimations in the FPGA prototype.
- Validate the results and estimations in the ASIC.
- Determine data limit of the cache memories and validate the advantages of the memory layout that allocates run-time critical software into the TCMs.
- Evaluate performance with channels in searching state.
- Validate design with real GPS signals, requiring integration of an SPI module and front-end circuitry.
- Replace AHB interface of the GNSS Module for an AXI4 interface and validate results.
- Implement capabilities for allowing communication with the Navigation Module.

Bibliography

[1] J Roselló, P Silvestrin, G Lopez Risueño, R Weigand, JV Perelló, Jens Heim, and Isaac Tejerina. Agga-4: Core device for gnss space receivers of this decade. In *2010 5th ESA Workshop on Satellite Navigation Technologies and European Workshop on GNSS Signals and Signal Processing (NAVITEC)*, pages 1–8. IEEE, 2010.

[2] Magnus Hijorth, Martin Aberg, Nils-Johan Wessman, Jan Andersson, Remy Chevallier, Russel Forsyth, Rolad Weigand, and Luca Fossati. Gr740: Rad-hard quad-core leon4ft system-on-chip. In *DASIA 2015-DAta Systems in Aerospace*, volume 732, 2015.

[3] NanoXplore. Ng-ultra, rad-hard fpga & socs, 2018. [Online; accessed March 21, 2020, https://www.nanoxplore.com/].

[4] JL Poupat, T Helfers, P Basset, A Grau Llovera, M Mattavelli, C Papadas, and O Lepape. Dahlia-very high performance microprocessor for space applications. *ADCSS 2017*, 2017.

[5] Max Ghiglione. Novel sensor processing architecture for next generation gnss receivers, 2018.

[6] Jan Andersson, Magnus Hjorth, Fredrik Johansson, and Sandi Habinc. Leon processor devices for space missions: First 20 years of leon in space. In *2017 6th International Conference on Space Mission Challenges for Information Technology (SMC-IT)*, pages 136–141. IEEE, 2017.

[7] J Sanz Subirana, JM Juan Zornoza, and M Hernández-Pajares. Gnss data processing, volume i: Fundamentals and algorithms. *ESA Communications, ESTEC, Noordwijk, Netherlands*, pages 145–161, 2013.

[8] ICD GPS. Navstar gps space segment/navigation user interfaces, interface specification. Technical report, IS-GPS-200E, 2010.

[9] ICD GLONASS. Global navigation satellite system glonass interface control document, 2002.

[10] ICD BeiDou. Beidou navigation satellite system signal in space interface control document open service signal b1i (version 1.0), 2012.

[11] ICD Galileo. Galileo open service, signal in space interface control document (os sis icd), 2008.

[12] Rashmi Bajaj, Samantha Lalinda Ranaweera, and Dharma P Agrawal. Gps: location-tracking technology. *Computer*, 35(4):92–94, 2002.

[13] GISGeography. Gps accuracy: Hdop, pdop, gdop, multipath & the atmosphere, 2019. [Online; accessed March 21, 2020, https://gisgeography.com/gps-accuracy-hdop-pdop-gdop-multipath].

[14] Heikki Hurskainen, Jussi Raasakka, Tapani Ahonen, and Jari Nurmi. Multicore software-defined radio architecture for gnss receiver signal processing. *EURASIP Journal on Embedded Systems*, 2009(1):543720, 2009.

[15] Dinesh Manandhar. Introduction to global navigation satellite system (gnss) software gps receiver. Technical report, Center for Spatial Information Science, The University of Tokyo, 2018.

[16] AMBA Specification. Rev. 2.0. *Arm, http://www. arm. com*, 1999.

[17] Arm Limited. Arm development studio, 2018.

[18] Brian J Gough and Richard Stallman. *An Introduction to GCC*. Citeseer, 2004.

[19] Tom Feist. Vivado design suite. *White Paper*, 5:30, 2012.

[20] Desmond J Higham and Nicholas J Higham. *MATLAB guide*. SIAM, 2016.

[21] DAHLIA Consortium, 2017. [Online; accessed March 21, 2020, https://dahlia-h2020.eu/consortium/].

[22] Arm Limited. Arm cortex-r52 processor technical reference manual, 2016.

[23] Jens Heim. Agga4 datasheet issue 1.0. Technical report, Airbus Defence & Space, 2017.

[24] XILINX. Axi4 to ahb-lite bridge v3.0, logicore ip product guide, 2015.

[25] Arm Limited. Amba 3 ahb-lite protocol specification v1.0. Technical report, 2001.

[26] Jens Heim. Icam gps receiver software design description issue 3.0. Technical report, Airbus Defence & Space, 2017.

[27] Jaume Joven, Per Strict, David Castells-Rufas, Akash Bagdia, Giovanni De Micheli, and Jordi Carrabina. Hw-sw implementation of a decoupled fpu for arm-based cortex-m1 socs in fpgas. In *2011 6th IEEE International Symposium on Industrial and Embedded Systems*, pages 1–8. IEEE, 2011.