An Assessment of Available Software Defined Radio Platforms Utilizing Iterative Algorithms

by

Nathan C. Ferreira

A Thesis
Submitted to the Faculty
of the
WORCESTER POLYTECHNIC INSTITUTE
in partial fulfillment of the requirements for the
Degree of Master of Science
in
Electrical and Computer Engineering
by

May 2015

APPROVED:

Professor Alexander Wyglinski, Major Advisor

Professor Lifeng Lai

Professor Shamsnaz Virani

Abstract

As the demands of communication systems have become more complex and varied, software defined radios (SDR) have become increasingly popular. With behavior that can be modified in software, SDR's provide a highly flexible and configurable development environment. Despite its programmable behavior, the maximum performance of an SDR is still rooted in its hardware. This limitation and the desire for the use of SDRs in different applications have led to the rise of various pieces of hardware to serve as SDR platforms. These platforms vary in aspects such as their performance limitations, implementation details, and cost. In this way the choice of SDR platform is not solely based on the cost of the hardware and should be closely examined before making a final decision.

This thesis examines the various SDR platform families available on the market today and compares the advantages and disadvantages present for each during development. As many different types of hardware can be considered an option to successfully implement an SDR, this thesis specifically focuses on general purpose processors, system on chip, and field-programmable gate array implementations. When examining these SDR families, the Freescale BSC9131 is chosen to represent the system on chip implementation, while the Nutaq PicoSDR 2x2 Embedded with Virtex6 SX315 is used for the remaining two options. In order to test each of these platforms, a Viterbi algorithm is implemented on each and the performance measured. This performance measurement considers both how quickly the platform is able to perform the decoding, as well as its bit error rate performance in order to ascertain the implementations' accuracy. Other factors considered when comparing each platform are its flexibility and the amount of options available for development. After testing, the details of each implementation are discussed and guidelines for choosing a platform are suggested.

Acknowledgements

I would like to acknowledge my advisor Professor Alexander Wyglinski for his guidance and encouragement. Along with Charles Swannack, Ryan Frazho, and David Li of the MITRE Corporation for their continued support. I would also like to thank Shamsnaz Virani and Lifeng Lai for serving on my committee. In addition I want to acknowledge both the financial and academic support of The MITRE Corporation and Peter Sherlock. Finally I would like to thank my parents, brothers, sisters, and my friend Rodrigo for their continuous support. Without them I would not be where I am today and they are what keeps me going.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

As communication systems have become more complex and their demands more varied, software defined radios (SDR), have become increasingly popular. An SDR is a radio whose behavior can be modified in software, with minimal to no changes in its hardware. This ability gives the SDR extra flexibility and a highly configurable nature, allowing it to adapt to a multitude of situations. Their ability to adapt makes SDRs an attractive choice for many applications and has led to SDRs seeing use in education, commercial, and government systems.

Despite its programmable behavior, the performance of an SDR is still rooted in its hardware. This limitation and the desire for the use of SDRs in so many different environments has led to the rise of various platforms that serve as solutions for implementing the functions of an SDR. These vary in aspects such as their performance limitations, ease of implementation, and cost, in both the financial and execution sense. SDR platforms, in addition to having varying performance and cost, also use different software and application program interfaces (API). Therefore the choice of SDR platform not only has a direct cost in terms of money spent on the system, but also a development cost which varies depending on how powerful the software is as a tool.

The differences in each platform allow the SDR to be built and implemented on the

platform that makes the most sense for the specific user and his needs. An example is that for education a user may look for a relatively low cost SDR that is easy to implement and the exact performance of the radio may not be a huge issue. For a military application, however, there may be more time for development and prototyping. The SDR would also have to adhere to strict performance metrics. As SDRs have become more popular over time, the possible platforms to implement them on has also increased. With such a large pool to choose from, the choice of which platform to use is not always clear. There is also a lack of documentation comparing the various aspects of the possible platforms and their abilities in order to help a user make the proper choice, despite the fact that the decision can help save both money and development time.

1.2 State of the Art

As mentioned before, the number of possible SDR devices has vastly increased over time. Currently it is possible to get an SDR dongle for less than ten dollars, perfect for someone just starting out in the field, as well as an SDR worth a few thousand dollars, aimed at companies and businesses that need something more complex with better performance. While there are a vast amount of options, there are three major families of SDRs that exist for use in most systems. These platforms are general purpose processors (GPP), field-programmable gate array (FPGA) and GPP hybrids, and system on chip (SOC). Other SDR types are either close relatives of these families or fill a unique niche, such as the dongle SDRs, and wouldn't be considered over the other options presented in general applications.

When it comes to comparing SDR platforms, the two major trends are to either compare hardware for a specific interest, such as in reference [12] or in mobile applications, or look at a single platform and implementation to determine what the hardware does well and what it lacks [4, 13]. This creates an issue when a user is trying to decide on an SDR platform for an application that does not have its own study. The user either needs to spend more money to guarantee that the platform can perform well above the expected performance, or use trial and error to determine if a project is able to be completed on a certain piece of hardware. Either of these solutions have the potential to increase costs and time to

implement for the user.

The goal of this thesis is to implement iterative algorithms on a representative of each of the families of SDRs, then compare the performance of each SDR in addition to the time to implement, cost, and development environment. Using these factors, this thesis aims to provide guidelines and suggestions in platform choice for users aiming to conduct projects utilizing SDRs. This will allow them to make more informed decisions on which device to use for an application to save development time and costs.

With the large amount of SDR platforms and interfaces available, it isn't possible to perform an analysis which covers all possibilities. Different algorithms and applications will vary in difficulty. In addition each program will need to be implemented differently on each platform to suite the platform's specific requirements, interface, and coding language. In order to accurately represent the possible programs that could be put on a piece of hardware, iterative algorithms, specifically Viterbi and Turbo Decoders, were chosen. This is due to these algorithms being computationally intensive and therefore being an accurate representation of the hardware working at close to full capacity. In addition, many software packages supply a partially completed Viterbi or Turbo decoder example, reducing the time to obtain performance results.

1.3 Thesis Contributions

This thesis will contribute the following to the SDR and Digital Signal Processing (DSP) community:

- An analysis on the various SDR platform families available on the market today. Comparing their abilities as well as their advantages and disadvantages during development.
- Practical implementations of iterative algorithms on three different types of SDRs. These implementations will discuss the difficulties of development on each SDR as well as its performance in relation to the other options.
- Overall guidelines for choosing SDR platforms based on their performance, overall

cost, and ease of implementation. These guidelines aim to reduce prototyping and development costs for the user.

1.4 Thesis Organization

This rest of this thesis is organized into five remaining chapters: Chapter 2 discusses the necessary background to understand Viterbi and Turbo Decoders. It also covers the basics of SDRs and goes into detail on SDR platforms and interfaces. Chapter 3 will discuss the proposed test setup used for each implementation. Chapter 4 contains specifications and abilities of the hardware used in this thesis as well as implementation details. Chapter 5 then covers the results as well as an analysis of each platform's performance. Finally Chapter 6 concludes the thesis and suggests a direction for future work.

Chapter 2

SDR Fundamentals

This chapter will cover the background information necessary for the remainder of this thesis. It begins by explaining the basics of software defined radios, and after covering this base knowledge, looks at the platform types and possible interfaces. Exact specifications and implementation details for the platforms used will be covered in Chapter 3. Afterwards the basics of iterative algorithms and channel encoding are covered before specifically going into detail on Viterbi and turbo decoders.

2.1 Software Defined Radio

As alluded to in the introduction of this thesis, as well as by its very name, an SDR is a radio that can be reconfigured via software to have different physical layer characteristics. This means that it is capable of performing various functions on the same piece of hardware and that its baseband functionality is defined in software, not hardware. Actions directly involving the RF front-end should also adjustable through software, such as defining a carrier frequency [14]. This aspect of SDRs, in that they are defined by their software and not their hardware, is what gives an SDR its flexibility and makes it appealing to a large amount of applications.

A diagram mapping out the basic components of a communications system can be seen in Figure 2.1. In this figure, orange represents the hardware portion of the communications system, while blue is for those dictated by software. These software operations include

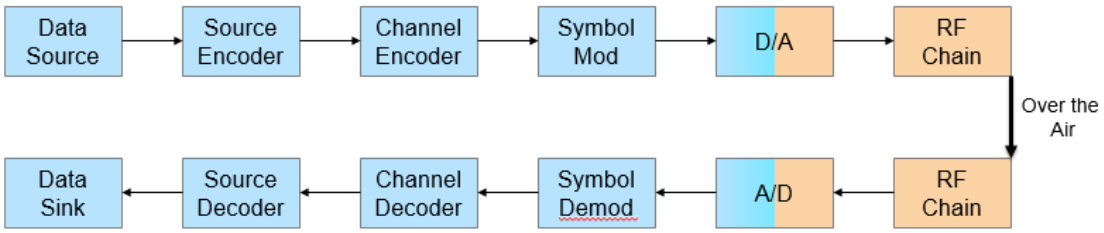

Figure 2.1: Flowchart illustrating the basic components of a communications system. Blocks in blue are in the software domain while those in orange are handled strictly by hardware.

functions such as encoding, decoding, and modulation. If the top row is a transmitter and the bottom is a receiver, each row can also be seen as its own SDR. In this case, a user would aim to use software to define all of the radio operations in blue. This would allow him to take advantage of the SDR's unique abilities, for goals such as rapid prototyping, as the hardware wouldn't need to be modified [14, 15].

2.1.1 Platforms

Due to having flexible software, SDRs also gain flexibility in their hardware. There are multiple types of platforms that can act as a solution to implementing an SDR, and each has their own unique advantages and disadvantages. The options often used are: FPGAs, GPPs, Digital Signal Processors (DSP), Application Specific Integrated Circuits (ASIC), and SOC [12, 16]. Due to both time and costs, it is not possible to test every single type of SDR, therefore three different types will be selected and analyzed in depth: GPPs, FPGA/GPP hybrids, and SOC. To restate what is said in the introduction, these platforms were chosen based on availability for testing and the belief that they adequately represent the three major families of possible SDRs.

General Purpose Processor

The first option this thesis will explore is the use of a general purpose processor as an SDR. GPPs provide substantial benefits when it comes to prototyping and development. These include the ability to program software radio components in a high level language

such as C/C++ due to their use of an operating system (OS). With the availability of a higher level language, comes the advantage of a designer being able to use object-orientated programming tools and a familiar debugging environment [1, 17]. This freedom can lead to increased productivity, reducing development time and cost both when building and debugging the system.

With this better development environment comes a few drawbacks. The first of which is that GPPs are not optimized for certain mathematical calculations like DSPs, nor are they particularly fast like FPGAs. GPPs are meant to be able to do everything generally well, they are not specialized. This limits their performance and means they cannot perform tasks that require speed. An example is when dealing with high carrier frequencies. A GPP is not fast enough to properly sample a signal at Radio Frequency (RF) or most Intermediate Frequencies (IF). This limitation usually leads to an FPGA performing the sampling and then down converting the signal to baseband as shown in Figure 2.2 [1]. Another option is to have dedicated hardware perform the down conversion [18].

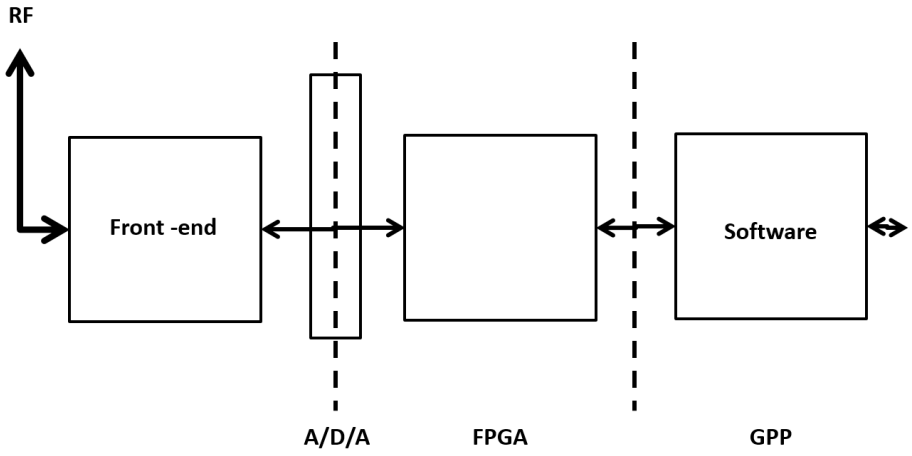

Figure 2.2: GPP with an FPGA performing the sampling frequency selection and down conversion [1]

Both of the mentioned options have a flaw when it comes to SDRs though, the GPP is not performing the down conversion itself. For the FPGA example, the FPGA will usually need to be programmed in a low level language such as VHDL or Verilog. Therefore, at

least for this step, the entire benefit of the GPP's easily programmable environment is lost. In the case of having dedicated hardware performing the down conversion, a different issue comes up. While the user avoids a more difficult programming language, the SDR loses flexibility. This is because if the user ever wants to change a parameter such as the SDR's carrier frequency, he will need to modify the hardware.

There is another drawback to using GPPs, which is also due to their use of an OS and capability to be programmed in a high level language. As the user only interacts with the SDR through that OS, it adds ambiguity to the system when observing execution time and resource availability [18]. The uncertainty can become an issue when dealing with complex applications and assuring they run in real time. It also does not allow a user to take advantage of extra space on the GPP to have a process run more quickly, or sacrifice speed to allow a design to take up less space and fit on the processor.

FPGA/GPP Hybrid

The next SDR platform this thesis looks at is a hybrid consisting of an FPGA and GPP. This is similar to the FPGA and GPP hybrid discussed in the previous section, though the roles of the FPGA and GPP can be significantly different. Before getting into details on the hybrid, an FPGA itself should be analyzed. As can be seen in Figure 2.3, an FPGA is made up of a finite number of resources. These resources include configurable logic blocks, mainly consisting of flip-flops and lookup tables (LUTs), as well as fixed logic blocks, examples including a multiplier or DSP slice [2]. The final resource of interest is the memory or RAM available on the FPGA. These resources are all connected via multiple programmable interconnections. The interconnections are either opened or closed through software, effectively allowing the user to create various digital circuits on a single piece of hardware.

The programmable nature of the interconnections make the FPGA extremely flexible and allows the user to reconfigure the designed digital circuit as well as its I/O connections. If a design does not take up much room on the FPGA, it can be made to run faster by utilizing the extra resources available on the FPGA through processes such as parallelization. The opposite is also true, if a particular design is quite large and there is not enough resources,

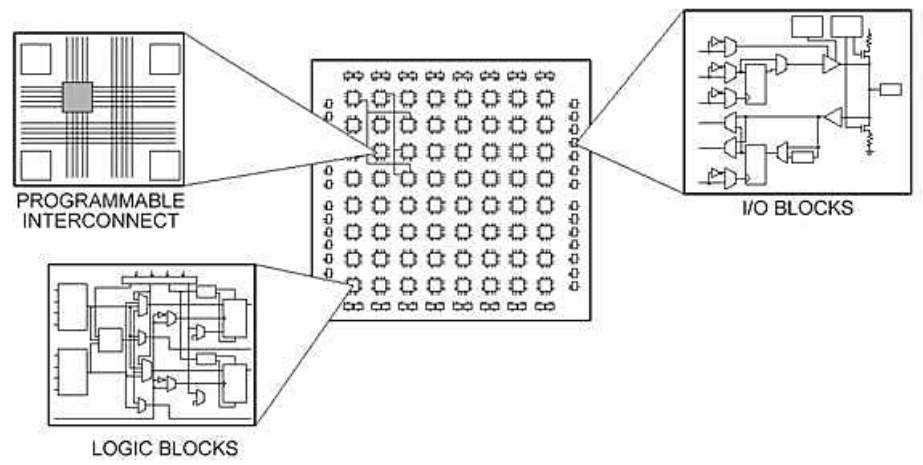

Figure 2.3: Basic structure of an FPGA [2]

it can usually be made smaller by allowing it to run slower. This gives the FPGA the advantage of being able to trade off between speed or resource efficiency [19].

These advantages and the FPGA's ability for strict customization is due to one of the FPGA's biggest drawbacks, however, and that is that it needs to be programmed in a hardware description language (HDL) such as VHDL or Verilog. This restriction can lead to slower design time and difficulty in debugging, leading to an increase in development costs. Recent advancements in high level tools such as Simulink have aimed to mitigate this, however, and have seen positive results through automatic HDL code generation for specific algorithms [19].

A hybrid between an FPGA and GPP attempts to utilize the advantages of both platforms. In this approach, complex signal processing can be completed in the FPGA and less computationally intensive control operations are handled by the GPP [19]. This takes advantage of the FPGA's faster speed for processing and allows the GPP to take care of slower operations. The use of a hybrid structure also encourages the use of a graphical environment such as Simulink along with Xilinx System Generator. With the available flowchart blocks already designed to model DSP cores, these tools allows a user to focus on the design at a system level instead of the programming language [20]. Many implementations today utilize a combination of VHDL with these graphical environments [21], this allows the user

to still program in VHDL when more precise control is needed of the FPGA's speed and resources. Together these advancements have helped to make FPGAs more accessible.

System on Chip

The final family of SDRs to look at is the system on chip. SOCs are very fast and consume very little power, making them a good option for mobile applications [12]. The exact definition for a SOC can vary, but the main requirement is that a SOC must be an entire system contained on a single chip. This system usually consists of a microprocessor, memory, and various peripherals [22, 23]. A SOC also has the advantage of being contained in a single chip, making it smaller and portable, which compliments its low power consumption in making it attractive for mobile applications. While initially designing a chip for an SOC, the amount of control provided rivals an FPGA design [12] and it can support many levels of parallelism [16].

Despite its strong advantages, the SOC route does have its disadvantages. While an FPGA has gates that can be reprogrammed to form a different digital circuit, an SOC is frozen in silicon and as such lacks this flexibility. SOCs also have very long design cycles and require a logic synthesis tool to translate high level descriptions of the desired behavior into boolean equations that can be put onto the logic elements of the chip [12]. The lack of flexibility as well as the long design cycles can cause high development costs for a SOC.

2.1.2 Interfaces

As hinted at in previous sections, each of these SDR platforms has its own interface. For GPPs, there are a multitude of choices but this thesis will focus on the Eclipse Integrated Development Environment (IDE) for C/C++ development. The FPGA/GPP hybrid will utilize the Xilinx System Generator tools available for Simulink, and the SOC will be developed using the CodeWarrior IDE available from Freescale. These choices were based on what was available for testing as well as what could be completed in the allotted time.

Eclipse IDE

The Eclipse IDE for C/C++ is a software application that, through the use of plug-ins, provides a designer with the necessary tools to develop various types of software. When working with an SDR on a GPP platform, eclipse allows the user to define the SDR's behavior through the C++ language, which is then run and interpreted through the GPP's OS. Eclipse also provides its own debugger and compiler for aid in development of code.

Eclipse has the ability to use external libraries. In order to simulate and code many of the functions and operations used in everyday communication systems, the IT++ library will be used. IT++ is a free C++ library containing mathematical, signal processing, and communication classes and functions [24]. These features make it useful for both research and performing simulations involving communication systems.

Simulink and Xilinx System Generator

Simulink is a block diagram environment made for simulation and model based design. It can directly interface with MATLAB and supports simulation, automatic MATLAB code generation, and verification of embedded systems [25]. Figure 2.4 shows a basic Simulink model and how it allows a user to connect function blocks in order to perform a simulation. Simulink can also run different subsystems at once, allowing the user to compare their outputs and verify if their behavior is equivalent. This is also shown in Figure 2.4 through the use of the Subtract blocks, whose output represents the difference between the outputs of the subsystem and the original model at the top. This difference is then sent to a scope for viewing.

Xilinx System Generator is a tool that works with Simulink to provide the user a way to design and program Xilinx FPGAs. As shown in Figure 2.5, Xilinx System Generator provides Xilinx specific blocks that can be used directly in the Simulink environment. The specific model shown is a subsystem of the model shown in Figure 2.4. In this model, the Gateway In and Gateway Out blocks represent the beginning and end of the FPGA design, respectively. In between these blocks are the provided Xilinx specific blocks that allow Simulink to simulate the FPGA behavior. Outside of the Gateway blocks the user is

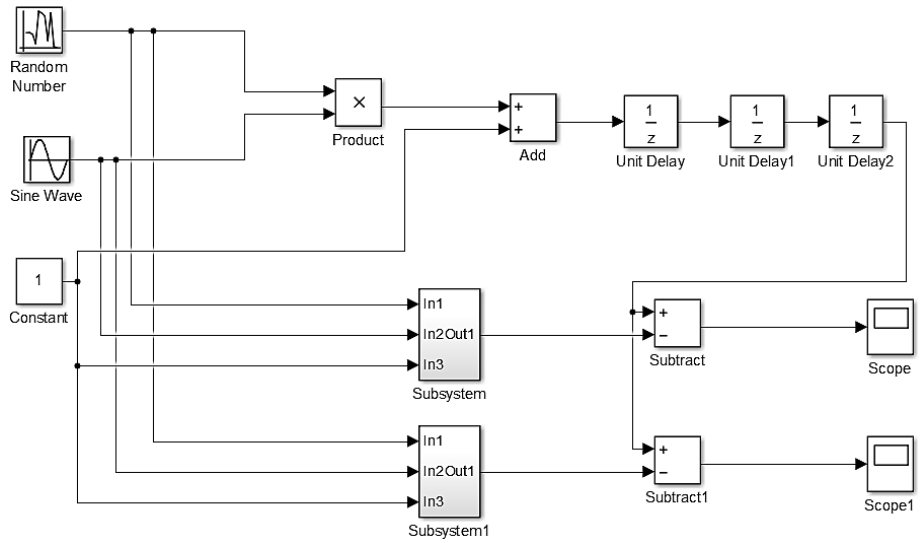

Figure 2.4: Basic Simulink example model

allowed to use any native MATLAB and Simulink functions. This allows the user to test MATLAB defined inputs and perform post-processing in the MATLAB environment.

In addition to providing a means of simulation, Xilinx System Generator also has the ability to automatically generate code for Xilinx FPGAs using MATLAB's HDL Coder [26]. This feature is only available for the provided Xilinx specific blocks located in between the Gateway In and Out blocks. This functionality allows a user to create a model in Simulink of a specific FPGA design, and after simulating and testing it, run that same design on the FPGA without needing to write his own VHDL. If a user cannot implement the design using the given blocks, however, handwritten HDL code will still need to be incorporated.

CodeWarrior IDE

The final interface used in this thesis is the CodeWarrior IDE. This IDE is based off the previously mentioned Eclipse, providing plugins and tools for Freescale products [27]. CodeWarrior's framework focuses on embedded applications, however, as opposed to Eclipse which is aimed to be used more for general C/C++ code development. This causes some of the tools and features to be different when switching between the two IDEs as CodeWarrior comes prepackaged with most necessary plug-ins and features.

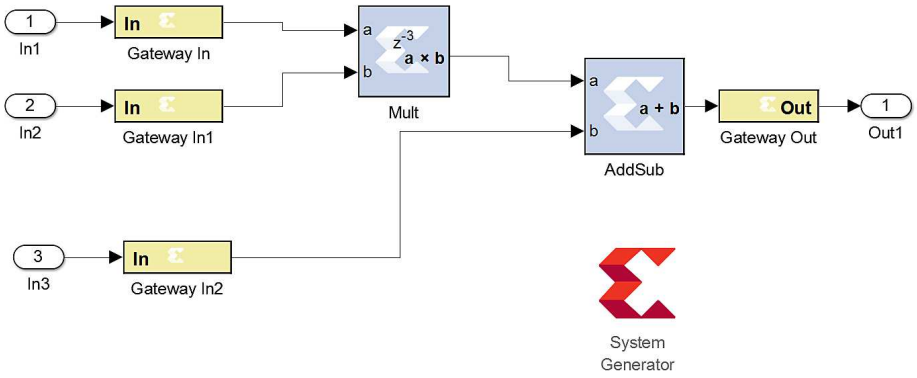

Figure 2.5: Simulink example model using Xilinx System Generator

As the CodeWarrior IDE is specifically tailored towards Freescale devices, it provides a few demos at installation for testing. These demos use a combination of the StarCore high-performance DSP cores and the multi acceleration platform engine (MAPLE), both of which make up the physical layer on the aforementioned Freescale devices [28]. MAPLE accelerators provide the user with hardware implementations of processes and functions normally used in communications such as Fourier transform processing, turbo decoding, and Viterbi decoding. These demos exist for both single and multi-core implementations.

2.2 Channel Encoding

Now that the various SDR platform and interface options used in this thesis have been explored, the following sections will go over the theories and algorithms. The two algorithms used to test each SDR type are the Viterbi and turbo decoders, which will be explored in detail later in this chapter. Each are meant to be used in a receiver to reverse the channel encoding done in a transmitter. Channel encoding itself is used to improve the capacity of a channel through the process of adding redundancies to the data [29]. The extra redundancy helps to protect the integrity of the data from possible errors during transmission. This protection stems from the extra information included in the transmission providing the receiver with enough data to correctly decode the message even if a bit was received incorrectly [30].

A very simple example is if the transmitter wanted to send '010' and added redundancy by repeating each bit three times. This would mean the transmitter would actually send '000111000'. The extra bits, while reducing the throughput of the transmitter, helps the receiver detect an error if it was to occur. Lets assume that an error occurs due to noise, resulting in the receiver reading the code as '000111010'. As the receiver knows the encoding scheme, it would look at each received codeword: '000', '111', '010' and check to see if it matches one of the possible words that could be sent: '000' and '111'. Once it detects an error, '010', the receiver then measures the Hamming distance, a measure of the number of bits that are different, between the received codeword and the possible transmissions. Finally it chooses the codeword that has the least distance from what was received and assumes that is what was actually sent, in this case that would be '000'. This is not a very strong type of channel encoding, as if two bit errors occur in a single codeword the receiver would decode the message incorrectly, but illustrates the basic concept.

When the transmitter performs channel encoding, including redundant information so the receiver can properly decode the signal, this is a form of forward error correction. A potential issue with this type of correction is when the transmission is subjected to bursts of noise for any extended period of time. This results in a time period in which no data is properly received and even the redundant information is corrupt. In order to combat this, an interleaver is used to randomize the order of the bits to be sent before transmission occurs. This process ensures that errors appear random and avoids long error bursts destroying an entire section of the transmission. This is due to the data being likely to turn up again at the receiver, at a different time period, due to interleaver [30]. The process of randomizing the order of the bits before transmission is reversible by mirroring the process at the receiver.

2.2.1 Convolutional Encoding

There are two major types of channel encoding used in current day communication systems, convolutional and block encoding. [29]. Block encoding operates by taking in a group of bits and creating a codeword from that specific block. Convolutional encoding on the other hand, operates on serial data, taking in a stream of data and mapping the bits into an output stream [29]. The data rate of a convolutional code is measured by

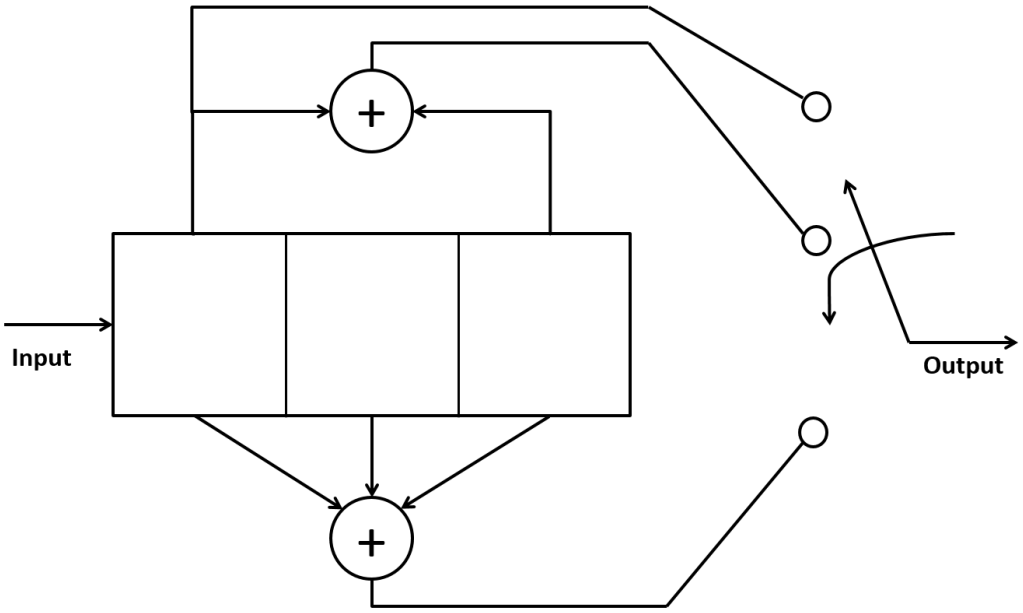

Figure 2.6: m=3, k=1, n=3 Convolutional encoder shown as a finite-state shift register [3]

the ratio of bits that go into the encoder, k, compared to the number of bits that leave the encoder, n, meaning the rate, R, is equal to $R = k/n$ [29, 3]. When implementing a convolutional encoder, there must also be memory that remembers a finite amount of the previous bits. This memory is used to help encode future bits, and the number of bits the encoder remembers is known as the constraint length, m [3, 31].

In order to better understand how a convolutional encoder actually works, as well as to show how it can be implemented in hardware, one can look at it as a linear finite-state shift register, as shown in Figure 2.6. In this figure, each block is a shift register that shifts its contents to the right at every clock cycle. They represent the earlier mentioned "memory" in the system. The encoder also outputs all three bits whenever the data is shifted. If k=2 in this example, n=6 instead, as each delay would hold two bits. In order for the receiver to decode the message properly, the state registers should be initialized to zero [3].

There is one other way to look at a convolutional encoder that will help later on when looking at a Viterbi decoder, and that is as a trellis diagram. The trellis diagram for the encoder shown in Figure 2.6 is shown in Figure 2.7. It can be seen that there are four

different states, *a,b,c,* and *d*, each of which represent a possible state of the shift registers [3]. It can be seen that there are two lines going to the right of each node, these are the possible state transitions based on the input. The solid line represents the transition if the input is zero, and the dotted line is the transition if the state is equal to one. Any lines coming from the left of a node are previous state transitions that could lead to that state occurring at that particular time.

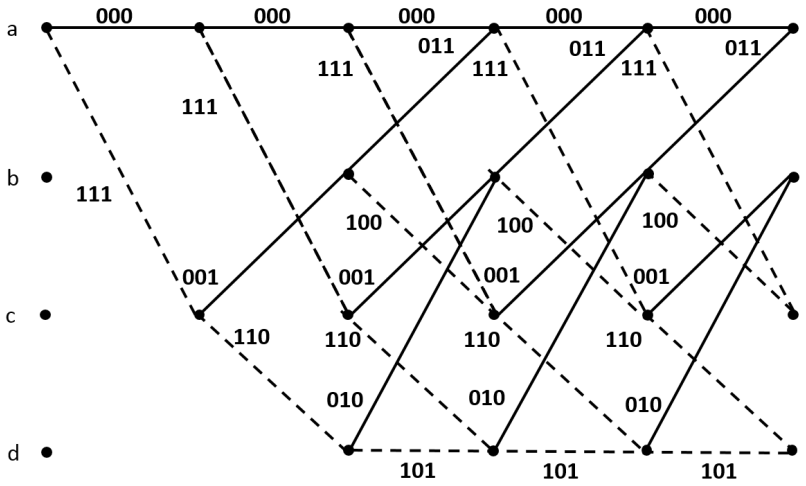

Figure 2.7: m=3, k=1, n=3 Convolutional encoder shown as a trellis diagram. Dotted line is when input=1, solid when input=0 [3]

2.3 Iterative Algorithms

Iterative algorithms are processes that generate a sequence of improving approximations. In other words, the longer the algorithm is allowed to iterate, the closer its estimation becomes to the true value. In terms of iterative decoding, this estimation is conducted in the receiver, and attempts to estimate the original symbols sent from the transmitter. Traditional iterative encoding and decoding involves of a concatenation of at least two codes, which can be done in either serial or parallel. When concatenated in parallel and using recursive convolution codes, the algorithm is referred to as Turbo Code [32]. An example of a turbo encoder is shown in Figure 2.8 and the accompanying decoder will be covered in more detail in a later section [4].

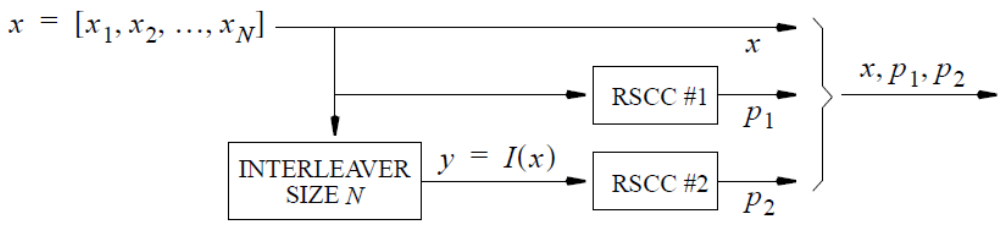

Figure 2.8: Basic implementation of a Turbo encoder with parallel concatenation [4]

2.3.1 Viterbi Algorithm

For the past thirty years the Viterbi algorithm has been constantly used in digital communications, especially in cell phones and digital video broadcasting [29, 33]. It has also found use in other forward error correction systems such as W-CDMA, IEEE802.11, satellite communication system, and high speed hard drives [5]. The reason for its high use is that it is a decoding algorithm for the convolutional encoder mentioned in Subsection 2.2.1 and when the restrained length of the code is short the Viterbi algorithm is faster, more efficient, and simpler than other possible decoding algorithms [5]. Work has also been done using the Viterbi algorithm to perform iterative decoding with positive results [33].

The Viterbi algorithm acts as a maximum likelihood sequence estimator and involves searching through the trellis, such as the one in Figure 2.7, to find the most probable sequence. Depending on if the detector following the demodulator in the receiver performs hard or soft decisions, the metric used while working with the trellis can either be a Hamming metric or euclidean metric, respectively [3]. While explaining the algorithm the Hamming metric will be used as the overall concept is the same.

The way a Viterbi decoder operates is by calculating path metrics which help it decide which path through the trellis is most likely. As each individual path relates to a unique sequence of states, it also relates to a unique sequence of received bits. The calculation for each path metrics go as follow:

- For every branch (state-transition) in each possible path, a branch metric,μ, is calculated. This metric is equal to the logarithm of the joint probability of the decoder

output sequence conditioned on the transmitted sequence. This can also be seen as

$$\mu = \log P(\mathbf{Y}|\mathbf{C}) \tag{2.1}$$

In which $\mathbf{Y}$ is the decoder output for the particular branch, and $\mathbf{C}$ is the transmitted sequence. $\mathbf{C}$ in the case of Figure 2.7 can be either a 0 or a 1, and as both events are independent, μ will be the summation of both of the conditional probabilities for the two transmissions. Once again μ has to be calculated for every single branch in every possible path between two states.

- For each path, its corresponding branch metrics are summed together to create a path metric, PM.
- When choosing which path is the correct path, the criterion is to pick the one having the largest path metric PM.

These calculations maximize the probability of a correct decision. For Hamming metrics, Hamming distance between what was received and the possible branches from each state can also be used as the branch and path metrics instead of calculating the logarithmic joint probabilities. In that case the path with the lowest total Hamming distance would be chosen [3].

The above instructions are how the Viterbi algorithm works theoretically, but as the Viterbi decoder needs to work on a constant stream of incoming data, it doesn't wait for all of the bits to be received and then calculate every μ for every branch in all possible paths. What the Viterbi algorithm actually does is it calculates PM for a path between two states, a starting state and each of the possible ending states. Then for each of the end states, it picks the path with the larger path metric as the survivor and discards the others. This results in $2^{k(m-1)}$ surviving paths remaining, one leading to each end state and one for each bit. The algorithm also keeps track of each path's metric to use in future calculations.

The algorithm is able to discard the other paths because any new data that stems from the end state wont affect the old data and would need to pass through that state transition of the trellis regardless, so the same path for that end state will still be chosen. The algorithm then repeats the process at each stage of the trellis as new information is received [3]. At

the end the Viterbi decoder determines the most likely path by picking the end state that has the highest corresponding path metric. It then determines the sent sequence by tracing the path backwards through the trellis, recording the states visited, and then going forward again following the path and determining what bits must have been sent to result in that path [29]. How far back the decoder traces backwards is known as the traceback length.

A diagram showing the basic structure of a Viterbi decoder can be seen in Figure 2.9. The figure shows that a Viterbi decoder is composed of three parts: the Branch Metric Unit (BMU), the Add-Compare-Select Unit (ACSU), and the survivor-path memory unit (SPMU) [34]. The BMU compares the received signal with the expectations calculated from the trellis and computes each branch metric. These metrics are then sent to the ACSU and adds to the corresponding path metrics. The ACSU then compares the resulting path metrics, chooses the best path through the trellis, and saves it in the path metric for the SPMU. The SPMU saves the past $2^{k(m-1)}$ survivor paths and uses them to calculate the decoder output as previously described [5].

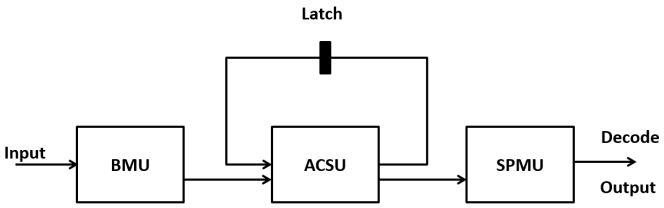

Figure 2.9: Basic structure of a Viterbi decoder [5]

2.3.2 Turbo Code

Turbo codes have already been mentioned as a way to encode and decode messages. It is worth mentioning, however, that they have also found a place in other applications as well. These areas cover processes such as detection and synchronization [35], as well as equalization [30], and that a single receiver can implement turbo code in more than just one component at a time. For all of these processes, whenever the turbo code deals with soft information or probabilities, it is assumed that the information is independent for each

bit [30]. This assumption allows simple algorithms to be used. It is also important to make sure only new information is being passed between turbo decoder and equalizer, otherwise one ends up telling the other what it already knows and the bit information will not be independent. This is done through the use of an interleaver and making the two work on distant parts of the received signal [30].

As shown previously in Figure 2.8, a turbo encoder concatenates its outputs in parallel. The turbo encoder also utilizes the convolution encoders gone over in the previous Subsection 2.2.1, except these encoders use recursive systematic convolutional codes (RSCC). Recursive encoders are necessary for turbo codes to have high performance [36]. The difference between the two types of convolutional code is that an RSC encoder feeds back one of the outputs and adds it to the next new input. An example if this can be seen in Figure 2.10. In RSC encoders the input bits also appear unaltered in the output. A final note is that the feedback allows the RSC encoder to be viewed as an infite impulse response filter, meaning that some inputs will lead better outputs than others [6].

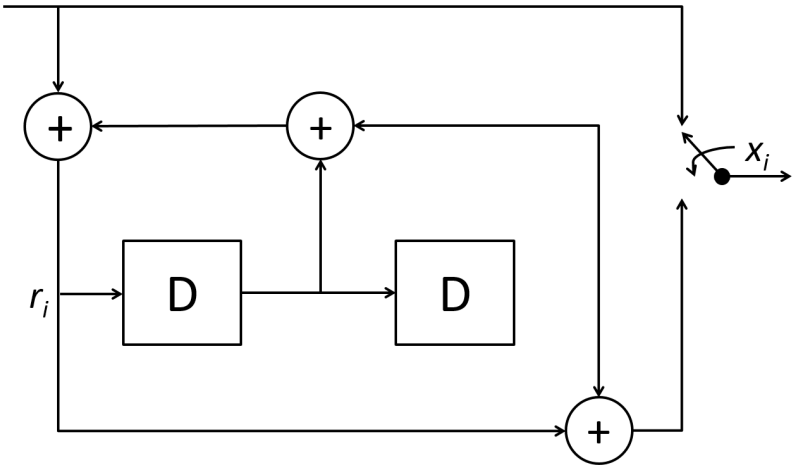

Figure 2.10: RSC encoder created by feeding back one of the outputs [6]

The accompanying turbo decoder is shown in Figure 2.11. It consists of two Single Input Single Output (SISO) decoders with an interleaver between them. The input, x, is split into two streams and fed into each SISO decoder. Each decoder receives its individual stream as well as the a systematic input x_s. For each iteration, data from the de-interleaver is sent

through the first SISO decoder, through the interleaver, and into the second SISO decoder. This result is then fed back into the de-interleaver and this process is repeated until the stopping criteria has been met and the output taken from the de-interleaver [7].

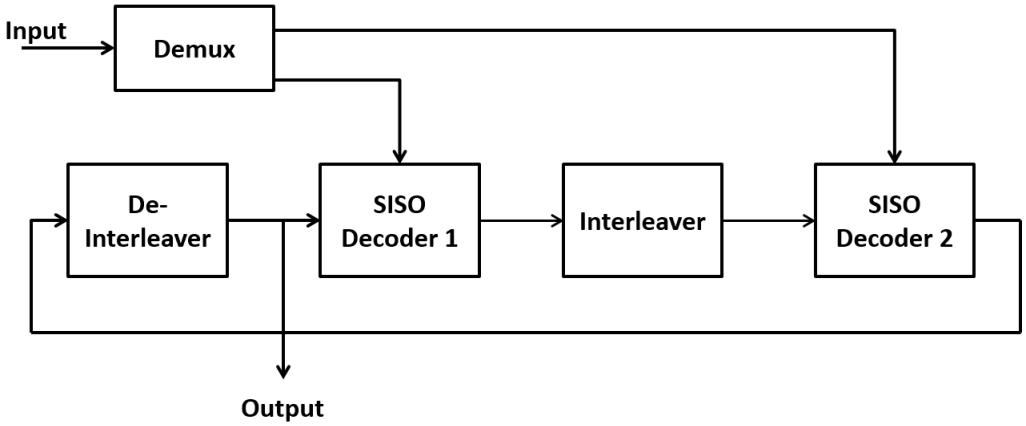

Figure 2.11: Diagram of a turbo decoder [7].

When passing data between the two decoders, it is vital to only pass extrinsic data. This is to prevent one decoder from making a measurement based of the other decoder's measurements and allows each bit to be assumed independent. If the decoders themselves do not estimate and automatically output the extrinsic data, then the sum of the systematic input and the extrinsic output from the one decoder must be subtracted from the output of the other decoder to ensure only extrinsic information is exchanged [4]. This can more clearly be seen in Figure 2.12.

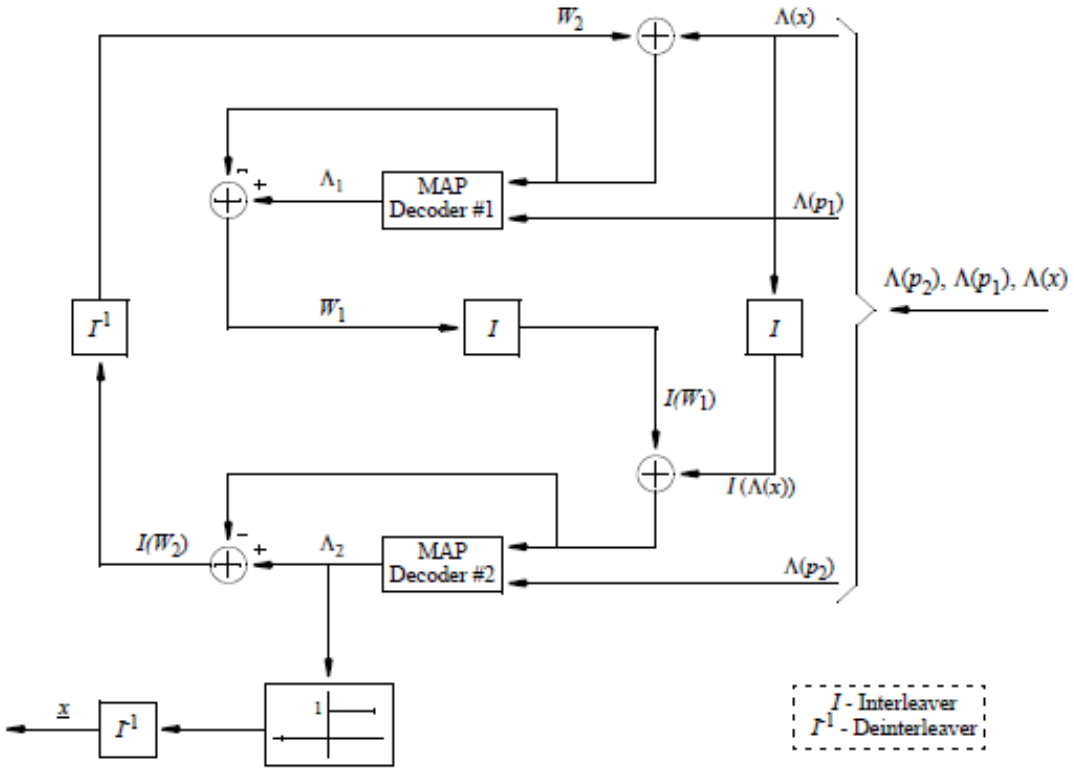

Figure 2.12: Diagram of a turbo decoder showing the steps needed to ensure the exchange of only extrinsic data [4].

2.4 Summary

This chapter went over the software Defined Radio platforms and their respective interfaces to be used in this thesis. In addition it covered the basics of channel encoding, specifically delving into convolutional encoding. Finally this chapter discussed the Viterbi and turbo algorithms and analyzed how they function. The next chapter will cover the proposed test setup for each implementation and explain how each implementation will be analyzed afterwards.

Chapter 3

Proposed Test Configuration

3.1 Overview

This chapter covers the proposed test configuration used throughout each implementation. First the test setup is shown, including what measurements are taken and the general approach for each platform. Next the chapters covers the choice of parameters for the Viterbi decoder and how they factor in to each test. Finally the analysis portion is discussed, explaining how the measurements are taken and how they are used in calculations.

3.2 Test Setup

Tests were created to define the performance of each SDR platform. A platform's performance consists of its accuracy and its speed. When dealing with a Viterbi decoder these qualities are the decoder's ability to properly decode a transmission at various Signal to Noise Ratios (SNR), as well as how quickly the device is able to complete this process. In order to ascertain the decoder's performance at various SNR levels, Bit Error Rate (BER) curves are generated. The details of which will be touched upon at the end of this chapter. To measure the speed, clock cycles will be used. Once again this will be discussed at the end of this chapter in the analysis section.

The overall test setup can be seen in Figure 3.1, which shows a separation between the code used for BER calculations and the code for the encoder and decoder. With this test

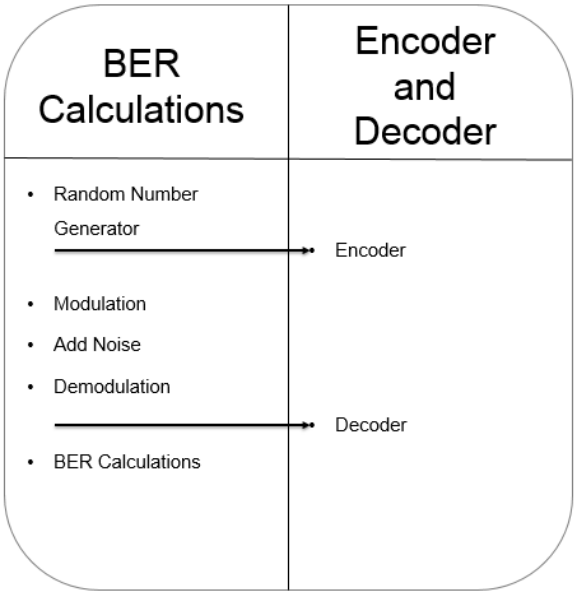

Figure 3.1: Diagram illustrating the overall setup of the testing procedure. The BER curve calculations are separated from the encoder and decoder to allow for the three platforms to use the BER code as a base.

configuration, the generation of the BER curves and other performance measurements can be done in post processing for each implementation. Additionally the output of the encoder can be recorded, allowing the input to each decoder to be predefined and saved. This frees the test to only encompass the actual decoder itself instead of needing to include additional steps such as the addition of the noisy channel and modulations. The encoder is shown separated from the general BER curve calculations as the exact implementation varies with each platform and will be discussed in the next chapter.

3.2.1 Choosing Parameters

As discussed in Subsection 2.3.1, the Viterbi algorithm has many parameters which need to be set properly in order for the decoding to be successful. Adjustment of these parameters not only change the behavior of the decoder, but also the time it takes for the decoder to run. As such in order to compare the performance of the decoders directly, it is important that each implementation operates under the same settings. These settings include the constraint length, generator polynomials used, and traceback length, all of which can affect

the decoder's behavior [11].

While the traceback length and constraint length parameters are both discussed in Chapter 2, the generator polynomials have not been explicitly explained. The generator polynomials define the structure of both the convolutional encoder and Viterbi decoder. For instance, the generator polynomial for the convolution encoder seen in Figure 2.6 is [100 101 111]. This is because for the first output bit, only the first shift register affects the output. Then for the second output bit, only the first and last registers affect the output. Finally for the last output bit, every register attributes to the value of the output bit.

3.3 Analysis Setup

This section will cover the basics of the analysis used to verify the behavior of the Viterbi decoders. There were two main aspects recorded for each decoder's performance, the first is its BER curve across various SNR levels. These curves are used to make sure the decoders are working properly. The second is how fast each SDR platform was able to complete the decoding operation. This is to provide a metric to compare the platforms with one another and see if one outperforms the others.

3.3.1 BER Curves

As previously mentioned, BER curves are calculated for each implementation in order to assess its accuracy. These BER curves involve calculating the probability of error at various SNR levels. These calculations involve running the decoder at a particular SNR level until a certain number of output bits are collected. The number of errors are then counted, and the number of errors divided by the number of total bits gives the probability of error at that SNR level. This is then repeated for each SNR value. Calculations for the BER curve were done in post processing via MATLAB for both the FPGA/GPP and SOC implementations, while the GPP implementation calculated the BER after each decoder function call.

Another approach to calculating the BER curve involves fixing the number of errors and running the decoder until this number of errors is reached. The fixed error number is then

divided by the total number of bits needed to reach it. This approach has the advantage of generating BER values for higher SNRs, in which the low probability of an error occurring can cause the first approach to fail. This approach can require more time, however, and due to limitations on the implementations, the first technique was considered preferable. The details of this will be discussed in the next chapter.

3.3.2 Timing Calculations

The second major part of the analysis is the timing of the Viterbi decoder on each platform. This varied from one implementation to another as each platform provided a different means of measuring clock cycles during a process. For the GPP implementation, clock cycles were counted using the standard library clock function. The Viterbi algorithm was implemented multiple times per SNR, with the clock function counting the total number of clock cycles that had passed after all iterations. The average is then found by dividing the total number of cycles by the number of iterations.

The FPGA/GPP implementation has a similar setup, in that a free running counter runs alongside the Viterbi decoder and is used to count the number of cycles passed. Finally the SOC implementation varies slightly. Instead of counting clock cycles, the hardware clock present on the chip is used to get the start time and end time before and after all iterations. Then the average is found for a single decoder iteration. It is possible to use the CodeWarrior IDE's profiling tool in order to count clock cycles as well.

One concern for taking these measurements is verifying that the measurement represents the proper information. Depending on where these clock cycles are counted, the test runs the disadvantage of timing the data channels as well. Meaning the measurement may not be just the device but also how long it takes to talk to the host computer. This is a larger concern for some implementations compared to others, but the way the measurements discussed in this section are setup should minimize any risk.

3.4 Summary

This chapter covered the approach used for the implementations explored in this thesis, first covering the basic test setup and the choice of decoder parameters. In addition it went over the analysis setup and how the BER curves and timing calculations are done. The next chapter will cover the platform specifications, implementation and simulation details, and the hurdles associated with each platform. Afterwards, the results of each implementation will be studied and conclusions drawn from them.

Chapter 4

Platform Implementation Details

4.1 Overview

This chapter will cover the specifications of the hardware chosen to represent each SDR family. It will also go over the details of how each hardware platform is tested and the algorithm implemented. First this chapter will cover the specifications of the various pieces of hardware and explore how the decoder is implemented on each platform. The final sections will detail the initial simulations used to test each implementation, as well as the transition from simulation to implementation.

4.2 Nutaq PicoSDR 2x2 Embedded with Virtex-6 SX315

The first platform used for testing is the Nutaq PicoSDR 2x2 Embedded with Virtex-6 SX315, the structure of which can be seen in Figure 4.1. As shown in the diagram, the Nutaq PicoSDR consists of a Virtex-6 FPGA and an i7 quad core processor, along with the various peripherals connected to each. These peripherals include different types of memory, as well as a Nutaq Radio 420x FMC module equipped with two transceivers. Between the FPGA and processor it can be seen that there exists up to two Peripheral Component Interconnect Express (PCIe) 4x links [8]. PCIe involves the use of high-speed parallel buses to connect the two components and is able to stream data between them [37]. The Nutaq PicoSDR is usually configured through the use of a remote computer, with the GigE interface and

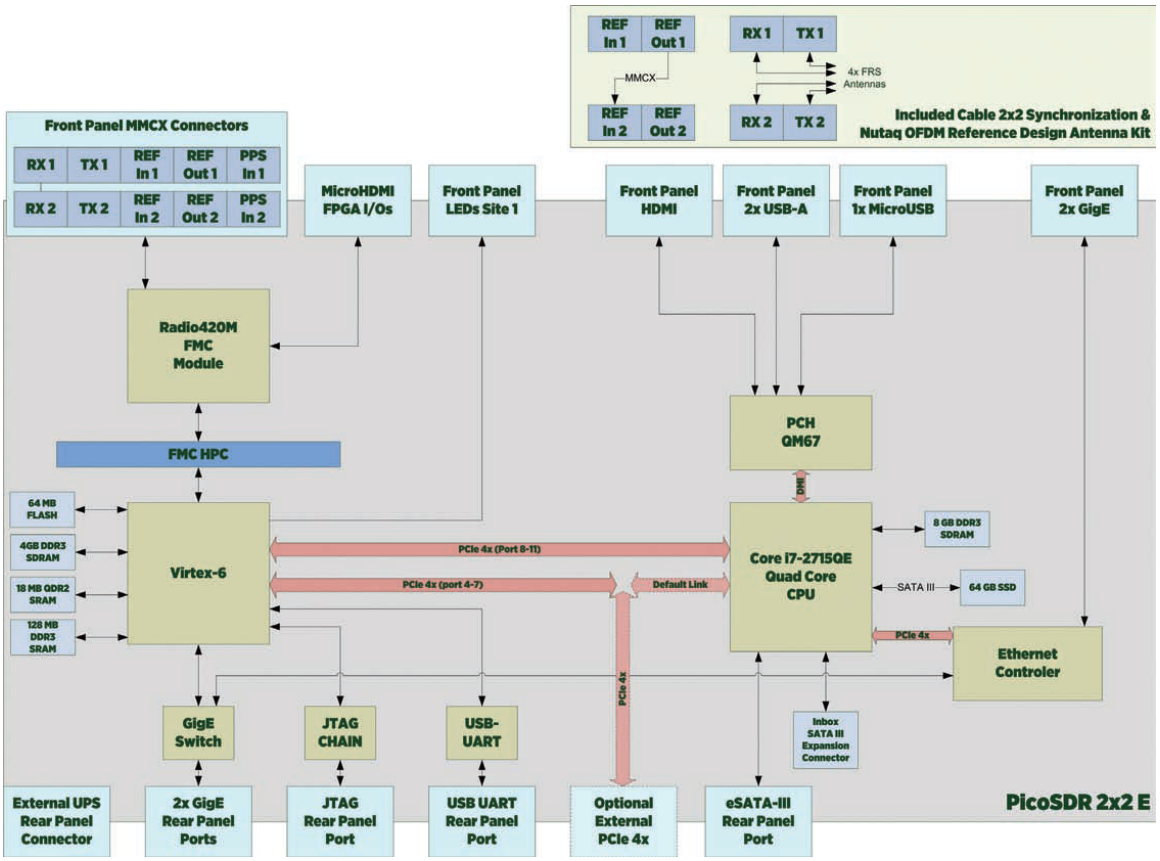

Figure 4.1: Diagram showing the structure of the Nutaq PicoSDR 2x2 Embedded CPU SDR [8].

optional external (PCIe) 4x link providing a way to send and receive data streams between the computer and SDR.

The Nutaq PicoSDR includes an i7 processor as well as a Virtex-6 FPGA so it will be used for both the GPP and FPGA/GPP hybrid implementations. The details of how this will be done is covered in the following subsections. The major concern with this type of approach is making sure that each implementation is isolated to the proper pieces of hardware. This concern is mitigated, however, by the very different interfaces and languages necessary to interact with each part of the PicoSDR. Each interface can only communicate with specific portions of the SDR. Also as different programming languages are necessary for the FPGA and GPP, a user is prevented from accidentally using an unintended portion of the hardware.

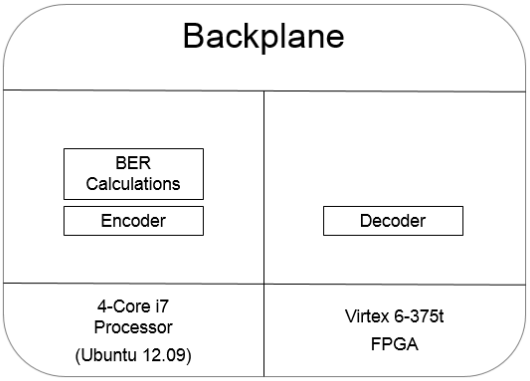

Figure 4.2: Diagram illustrating the setup of the testing procedure for the FPGA and GPP hybrid implementation. As well as the separation of the BER curve calculations from the encoder and decoder.

4.2.1 FPGA and GPP Hybrid Implementation

The first implementation to be covered is the FPGA/GPP hybrid. The overall setup of this implementation can be seen in Figure 4.2. As shown in the figure, the BER calculations are handled by the i7 processor, acting as the GPP. The processor also handles all of the overhead necessary in the test. This includes tasks such as setting SNR values to be tested and adding channel noise between the encoder and decoder. For this implementation the GPP also handles the encoder, so as to make the decoder the only process running on the FPGA.

The actual decoder is implemented within the Virtex-6 FPGA and communication between the FPGA and GPP is handled by the previously mentioned PCIe connection. The FPGA itself is programmed through the JTAG connection. When programming the FPGA, it was decided to use a model-based design utilizing a Viterbi decoder originally designed by Xilinx. This choice minimizes any bias due to the necessity to use a low-level programming language such as VHDL to program the FPGA. The model-based design is done using the graphical environment Simulink, along with Xilinx's System Generator and Platform Studio. Within Simulink, System Generator provides functional Simulink blocks that can be used to generate the HDL code needed to program the FPGA. This code includes not only the HDL for the design itself, but a clock wrapper that encloses the design and a HDL

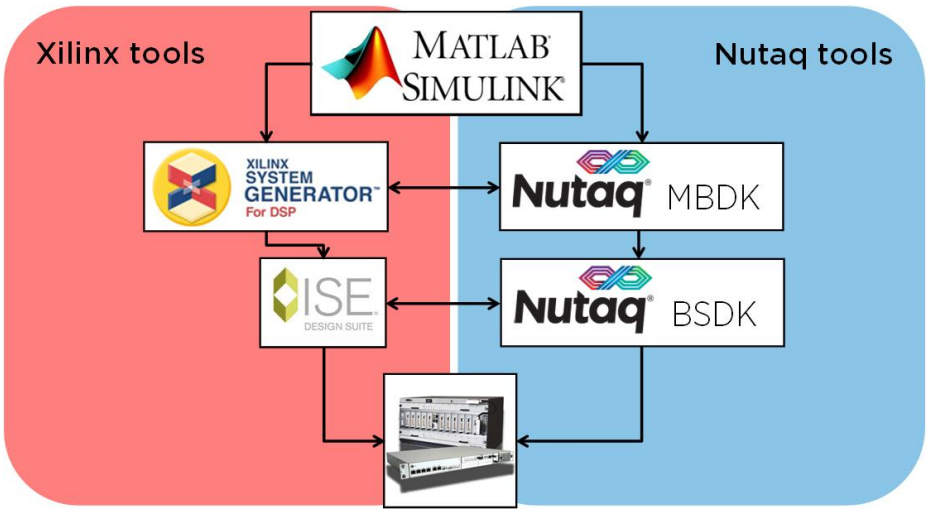

Figure 4.3: Flowchart showing connections between Simulink, System Generator, Nutaq MBDK, and Nutaq BSDK [9].

testbench enclosing the clock wrapper. System Generator also creates the files necessary for other tools such as XST and Project Navigator to work with the HDL [38].

A few more programs are needed in order to allow for a model based design. These are the Nutaq Model Based Design Kit (MBDK) and the Nutaq Board Software Development Kit (BSDK). These allow the auto-generated HDL code to properly interface with the Nutaq FPGAs, mapping the inputs and outputs to actual pins on the board. Without these tools, the Viterbi decoder can not actually be put on the board itself. A flowchart showing the connections between these programs and the expected design flow can be seen in Figure 4.3.

An example of the Xilinx Viterbi block chosen, the Viterbi Decoder 7.0, can be seen in Figure 4.4. The Viterbi decoder operates on the notion of soft-decision decoding, in which the inputs to this block are log-likelihood ratios (LLRs). An example of a possible LLR structure is if the Viterbi decoder is set to accept an input three bits long. These bits can represent values from zero to seven, with a seven representing a "most confident one" and a zero representing a "most confident zero". The confidence of each choice decreases as one moves away from the extremes, for example, a six would be the "second most confident one" [39]. These inputs are generated using a simulink model, which will be discussed in detail in subsection 4.4.

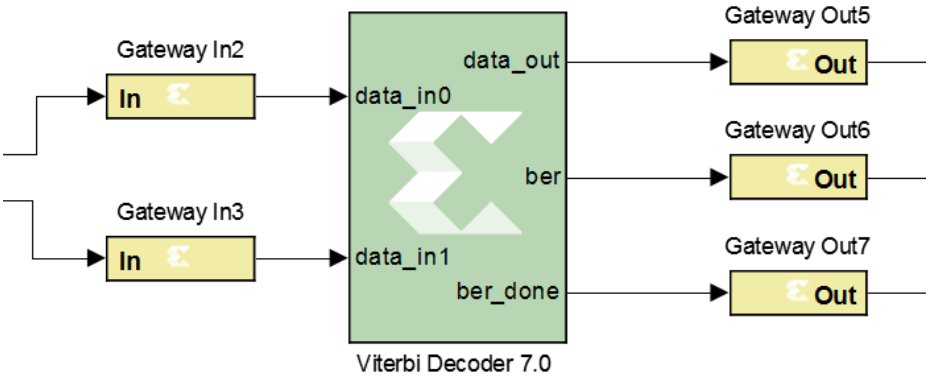

Figure 4.4: Example of Xilinx Viterbi decoder used in Simulink.

The use of the simulink model also opens up the possibility of saving these inputs and sending them straight to the FPGA for testing, allowing one to skip the use of the GPP all together in order to test the FPGA alone. This implementation and others like it can make use of the Real Time Data Exchange (RTDEx) connection available for the Nutaq PicoSDR. Figure 4.5 illustrates the concept of the RTDEx connection. The GPP in this setup takes care of generating data and other overhead processes such as can be seen on the left side of Figure 3.1. Then when the GPP reaches a process which is dependent on speed and necessitating the use of the FPGA, it pushes data through the RTDEx connection. The GPP then waits for a response from the RTDEx connection while the FPGA processes the data. Once the FPGA is finished it pushes data back through the RTDEx connection to the waiting GPP. The RTDEx also allows data to be sent to and received from the FPGA at the same time if set to full-duplex mode [40].

4.2.2 GPP Implementation

The second implementation is of the General Purpose Processor, or GPP. The setup of this implementation can be seen in Figure 4.6. In Figure 4.6, it can be seen that for the GPP implementation, everything is handled by the same device, the 4-core i7 processor.

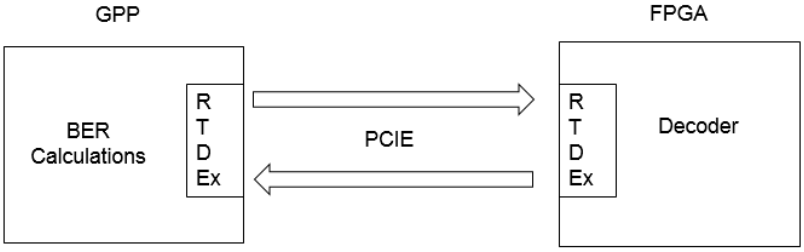

Figure 4.5: Diagram illustrating basic design for RTDEx connection.

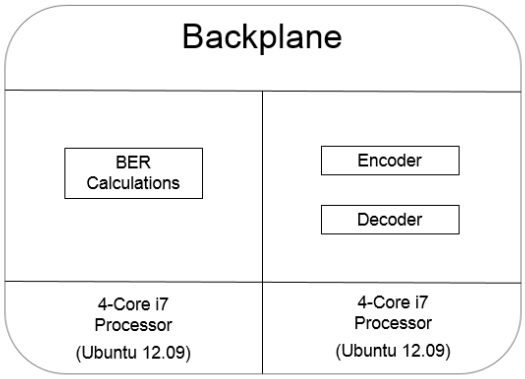

Figure 4.6: Diagram illustrating the setup of the testing procedure for the GPP implementation. As well as the separation of the BER curve calculations from the encoder and decoder.

This includes both the BER calculations as well as the encoder and decoder. The processor is programmed using C++ and the standard libraries are used to generate BER curves. The encoder and decoder, on the other hand, make use of IT++ library functions for both their initialization and implementation. As previously described, the IT++ library contains mathematical, signal processing, and communication classes and functions. This specific implementation makes use of the library's random bit generator, convolutional encoder, modulation, channel, demodulation, and Viterbi decoder functions.

The hardware itself, the i7 processor, is identical in behavior to the type of processor used in an average computer. For the GPP implementation, the processor is running its own OS, Ubuntu 12.04. This allows a user to log into the processor, and using linux commands,

build and run the Viterbi program on it. This process is very similar to how one would run the program on any other type of computer. This allows the decoder to be easily implemented and its behavior measured.

4.3 Freescale BSC9131 RDB

The final platform and implementation to be tested is of the Freescale BSC9131 RDB system on chip. A diagram showing the major functional units of the BSC9131 can be seen in Figure 4.7.This particular implementation focuses on the StarCore DSP Core, the Multi Accelerator Platform Engine for Femto BaseStation Baseband Processing (MAPLE), and the memory controller. Another major part of the BSC9131's hardware, although not specifically used in this implementation, is the Power Architecture subsystem including an e500 processor [41]. MAPLE is an algorithm accelerator for operations such as channel encoding, decoding, and Fourier transforms. As such, it is the portion of the system on chip that actually performs the Viterbi decoding [42].

The overall test setup can be seen in Figure 4.8. As mentioned before, the Viterbi decoder is actually handled by the MAPLE driver. The Starcore DSP on the other hand, handles all of the overhead such as reading the inputs from memory, pushing data to the MAPLE, and recording the outputs to memory. In order to simplify the implementation, the input to the decoder is generated using MATLAB and then recorded in the BSC9131's shared memory. From here, the Starcore DSP reads in a section of the input and pushes it to MAPLE where the actual Viterbi function is performed. Once the MAPLE returns the decoder's output, the Starcore then records it and sends the MAPLE the next section of the input. Once the process is complete for the entire input, the output is then read from the shared memory and BER calculations are handled in post processing. In reference to the choice of BER calculation discussed in subsection 3.3.1, the decision is due to the need to predefine each input sequence to the BSC9131. This means that the input can not just run until a certain number of errors are reached as the input will no longer exist if it is run for too long, or if it is looped, stop being unique.

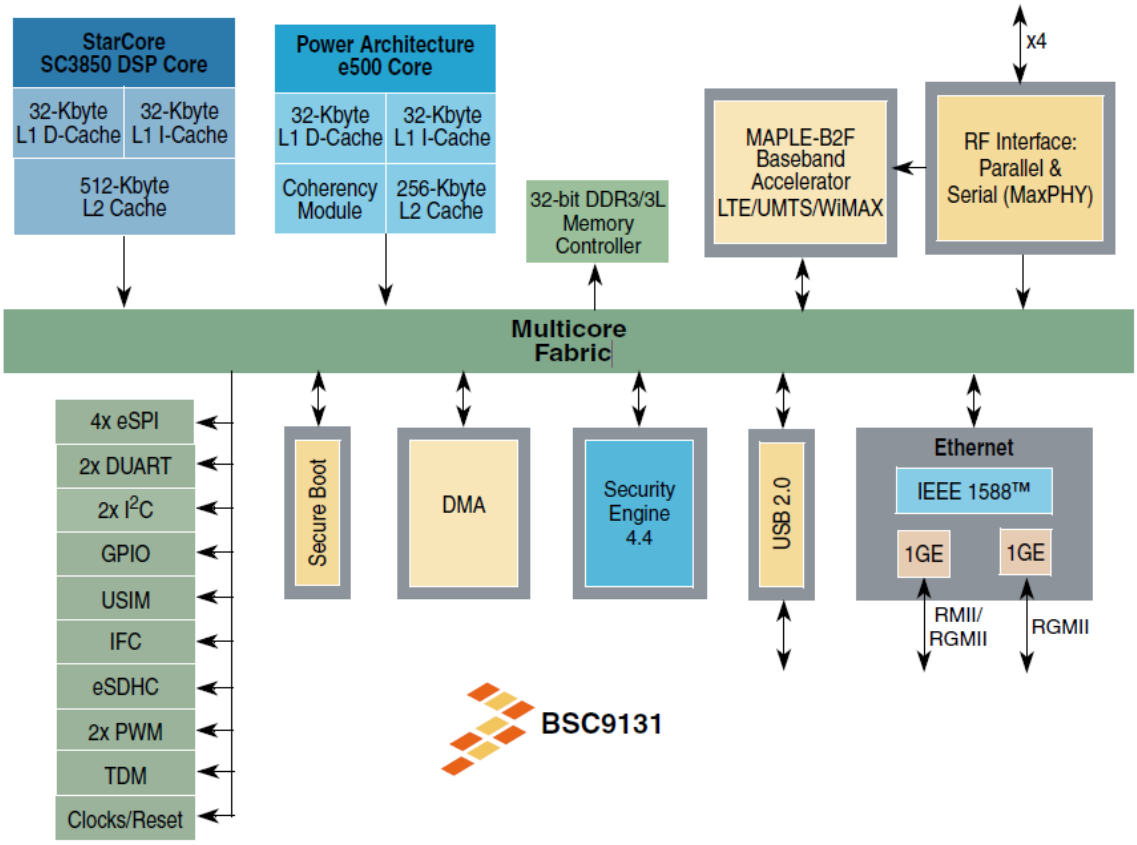

Figure 4.7: Diagram showing the major functional units of the Freescale BSC9131 [10].

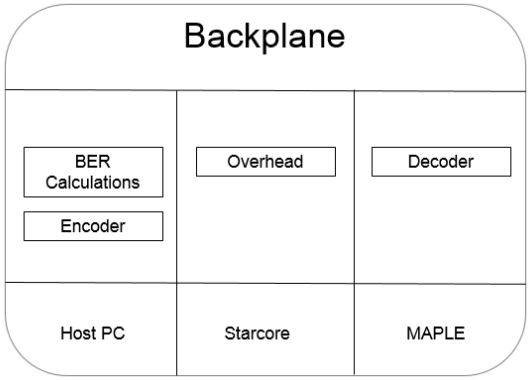

Figure 4.8: Diagram illustrating the setup of the testing procedure for the SoC implementation. As well as the separation of the BER curve calculations from the encoder and decoder.

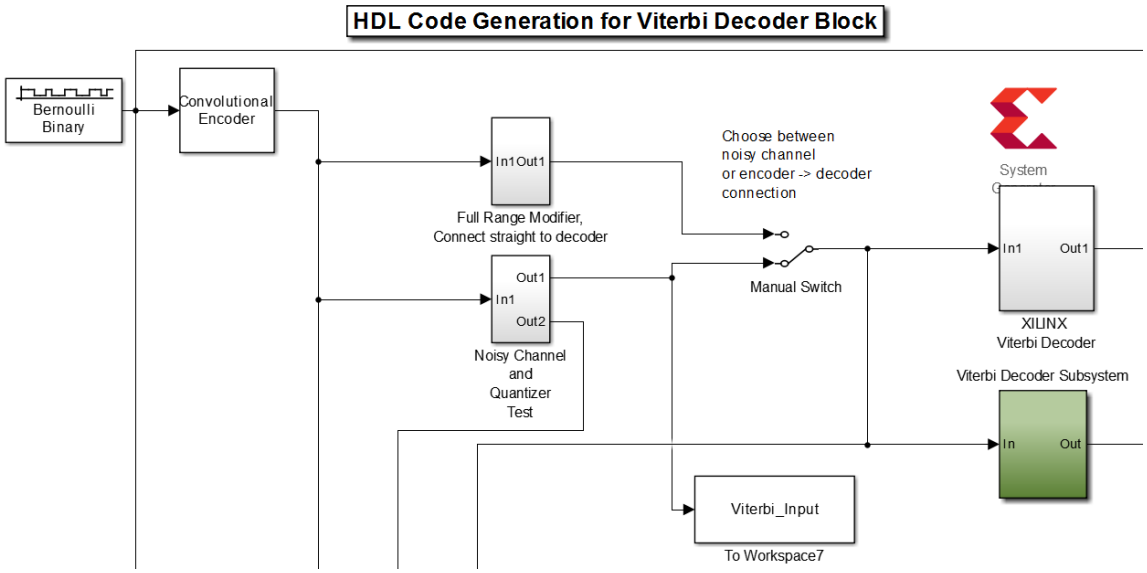

Figure 4.9: Simulink model used to test the Xilinx and Simulink Viterbi decoders. The switch allows for easy selection between a channel with no noise or a noisy channel. The Xilinx Viterbi decoder is identical to the one seen in Figure 4.4. The portions of the model not shown are scopes and data sinks used to record data and verification.

4.4 Initial Simulations

Before each implementation was completed in hardware, simulations were done to set a baseline for the behavior of the Viterbi decoders and to verify the implementations' accuracy. For the FPGA/GPP implementation, this involves running Xilinx's Viterbi Decoder along with Simulink's built in Viterbi decoder. The Simulink model created to do so is shown in Figure 4.9. The XILINX Viterbi Decoder subsystem is identical to the system shown in Figure 4.4. The switch within the model provides a way to quickly switch between a noisy channel or directly connecting the encoder to the decoder.

For this simulation the two decoders were compared to verify that the Xilinx decoder was operating properly, as well as to align the two decoder outputs for comparison. An example of the two aligned outputs can be seen in Figure 4.10. This figure also shows a caveat of the Xilinx decoder, which is that during initialization and before performing any actual decoding, the decoder will output a few incorrect bits, in this case a bit set to one. This initialization error occurs across multiple SNR levels and settings. The main issue

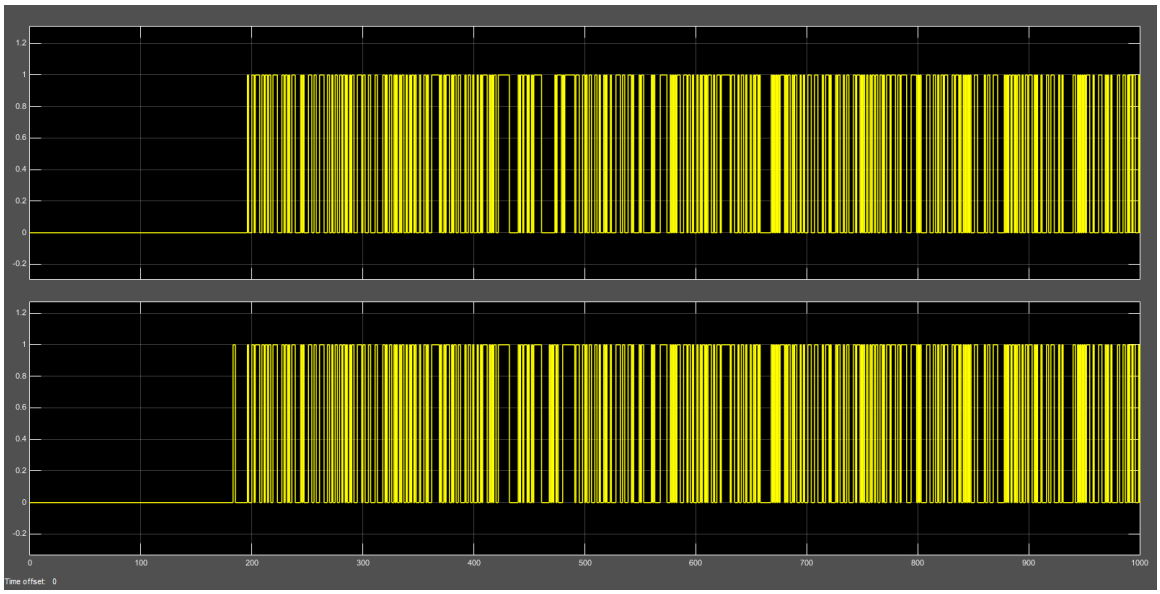

Figure 4.10: Aligned outputs of the Simulink (top) and Xilinx System Generator (bottom) Viterbi decoders. It can be seen that the Xilinx Viterbi decoder has an initial error of setting a bit to one that occurs during initialization.

this raises is difficulty in aligning the output signals or determining the delay introduced by the Xilinx Viterbi decoder. An example BER curve for this simulation can also be seen in Figure 4.11. While this is just a preliminary test, it suggests that the Viterbi decoder is implementing the algorithm somewhat correctly as the BER curve has a waterfall shape and decreases with increased SNR values. The BER Simulink Viterbi decoder, however, can be seen in Figure 4.12. This suggests that the initial simulations of the Viterbi decoder are incorrect as the Simulink Viterbi decoder far out performs the Xilinx version.

For the GPP implementation, the initial simulations involved running the C++ code involving the IT++ libraries on a computer before transferring the code over to the PicoSDR's GPP. The BER curve generated for this simulation can be seen in Figure 4.13. The difference between the GPP and FPGA/GPP implementation can be attributed to two major factors. The first is that for this particular GPP simulation, less SNR points were tested meaning less resolution. The second is that the parameters of the Viterbi decoders, previously discussed in Subsection 2.3.1, varied between the two simulations. This variance changes the SNR values that the Viterbi decoder is effective at, among other details which

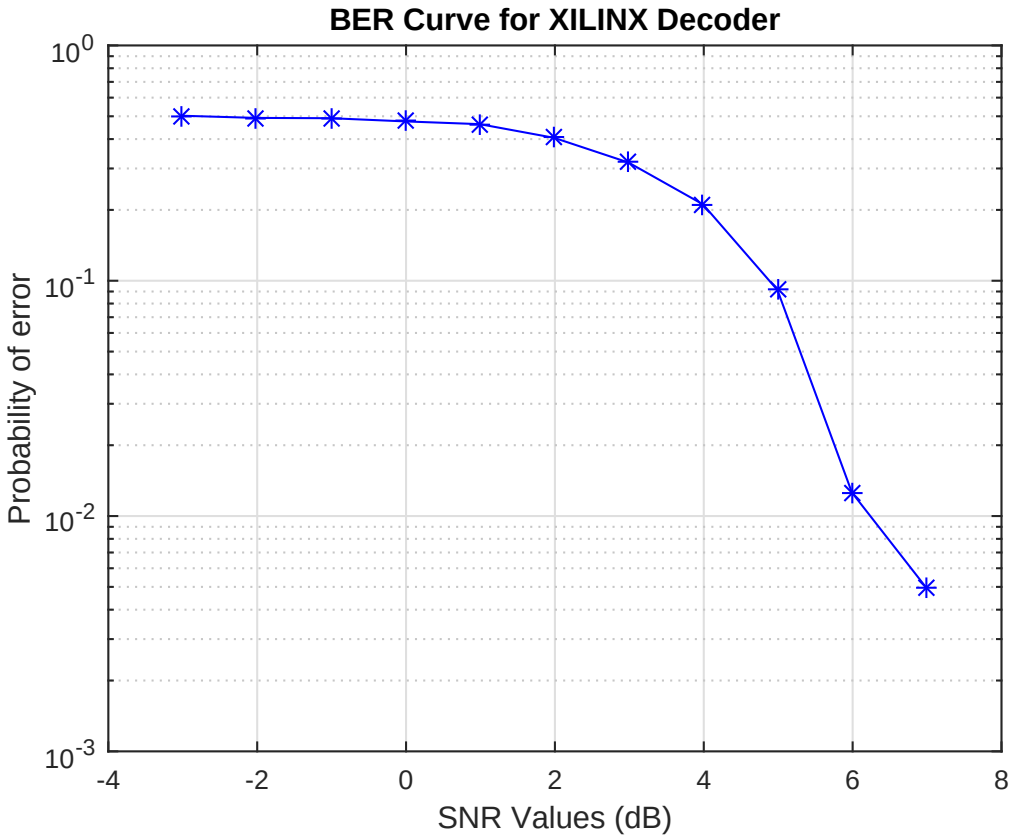

Figure 4.11: BER curve for initial simulation of Xilinx Viterbi decoder.

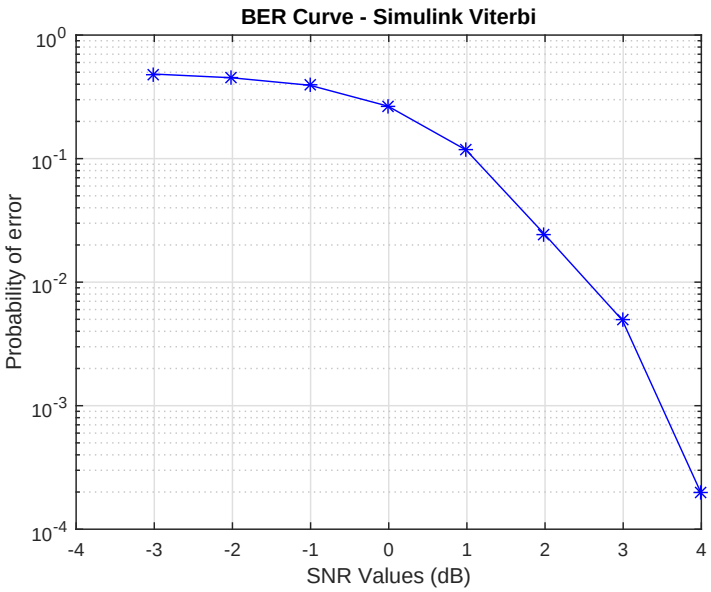

Figure 4.12: BER curve for initial simulation of Simulink Viterbi decoder.

will be discussed in the next section.

For the SOC implementation, there is no a way to simulate running the Viterbi decoder on MAPLE on the BSC9131. As such a demo provided by Freescale is used as a baseline for the implementation. This demo works by having a predefined input and expected output. The demo runs the input through the MAPLE, and compares its output with what is expected. Then the demo outputs the message "Core Passed" if the decoding was successful. While this demo did not provide any sort of BER calculations, it did serve as a baseline for the actual implementation. For the test implementation, the demo was adjusted in two major ways. The first change was that instead of a small predefined input, a much larger one was created via MATLAB and put into the shared memory of the SOC. This input is then read into the SOC in blocks. This change allowed the input to simulate a continuous input such as would be seen in an actual application, as well as provide enough data to have accurate BER calculations. The second change is that instead of saving the output and instantly checking it against what was expected, the output was instead stored in a much larger array and read out from the shared memory. This allowed the BER calculations to be handled in post processing.

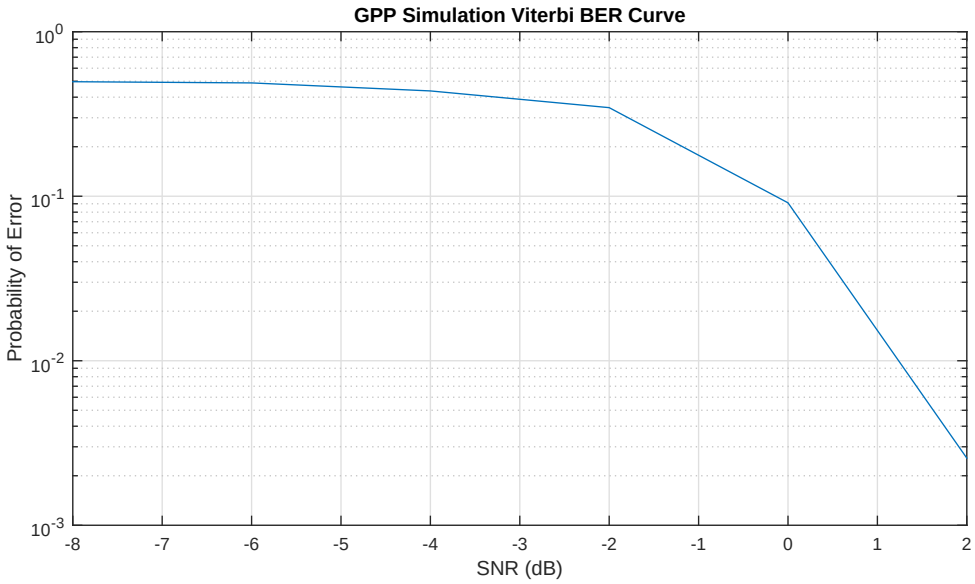

Figure 4.13: BER curve for initial simulation of IT++ Viterbi decoder.

4.5 Issues with Implementations

The goal of this section is to list particular issues that came up during implementation, as well as any solutions that were decided upon. For the GPP implementation, because of the ability to log into the processor and run the code that way, there were very little issues aside from linking the IT++ library to the code. The major hurdle was to increase the resolution of the clock function used to count clock cycles that had passed. This problem originated when the clock function was trying to count the cycles for every single decoder, and was patched by taking the total cycles over many iterations and then taking the average.

The FPGA/GPP implementation did run into a few issues involving the myriad of programs needed to make the model-based design successful. These issues mainly focused on making sure the right version of one program was properly interacting with a specific version of another, an example being that the license for the Viterbi decoder can only be recognized if one is using certain paired versions of System Generator and Simulink. Another example of this is that the RTDEx demos could only work with older versions of Platform Studio, making verifying connections between the FPGA and GPP troublesome.

This issue in particular caused the FPGA/GPP implementation to mainly focus on the FPGA portion of the design and ignore the connection between the GPP and FPGA.

The SOC implementation had little issues as it had the Viterbi demo to act as a foundation. The major concern with this implementation is where to store the inputs and outputs to memory. Depending on how the inputs were stored, it was possible that they would be left in the cache, meaning that the decoder would be able to run much faster. This increased speed, however, would not be accurate compared to how the device actually operates when decoding constantly changing inputs. By invalidating the cache after each set of iterations and guaranteeing that the input is long enough to remain unique during this time, this problem was avoided. This solution had the drawback of needing to load the entire input sequence ahead of time onto the board's shared memory.

4.6 Summary

This chapter covers the platform specifications, implementation and simulation details, and some hurdles associated with each platform. Specifically diving into details about how the platforms send and receive their outputs and inputs. In the next chapter the results of each implementation will be studied and conclusions drawn from them.

Chapter 5

Simulation and Hardware Results

5.1 Overview

This chapter outlines the outcome of each implementation and evaluates the success or failure of each by examining the decoder's BER curve and timing behavior. Each implementation posed its own challenges, and each challenge caused a change in how the decoder was enabled for that platform. Because of this, some implementations were more successful than other, such as having a more accessible and user-friendly interface or being able to operate on larger sets of data. A viable option for future work would be to create a single interface in which all three platforms could be tested.

5.2 Performance

As previously mentioned, an implementation's performance is judged on its BER curve accuracy and its speed when implemented. For each of the following subsections, a platform is discussed and its BER curve shown. In addition, the latency and potential throughput of each is discussed. When possible the implementation will be compared with the corresponding simulation, and at the end all implementations will be compared. For all of these implementations the following parameters were chosen: the generator polynomial in octal formal is [561 753], the rate of the encoder is 1/2, and the constraint length is 9.

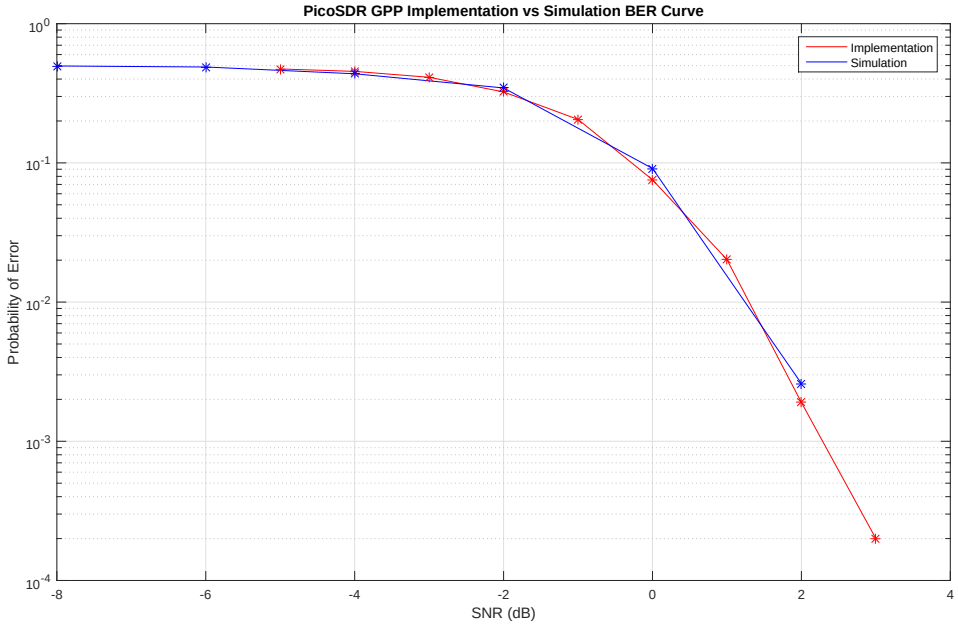

Figure 5.1: PicoSDR GPP Implementation BER curve. The red line is the actual implementation while the blue is the initial simulation.

5.2.1 GPP: Nutaq PicoSDR 2x2 Embedded with Virtex-6 SX315

The first platform to be examined is the Nutaq PicoSDR, specifically its GPP implementation. As can be seen in Figure 5.1, the implementation of the decoder on the PicoSDR closely follows the results from the initial simulations. This gives credit to the GPP implementation and shows that the Viterbi decoder is decoding as expected.

The timing aspect of the GPP implementation can be seen in Figure 5.2. The clock cycles per second is set to be 1 MHz, and initially it can be seen that the average number of cycles per Viterbi decoder seems to jump between two values. This is due to the BER calculations breaking at some SNRs upon reaching a certain number of error, cutting the amount of iterations short. It can also be seen that as the SNR increases, the average time smooths out. In the case of the Viterbi decoder being tested, this occurs around 367840 clock cycles per decoder, meaning a single Viterbi decoder takes 0.36784 seconds to run. As each Viterbi decoder is operating on 100000 bits, this puts the throughput of the decoder at about 0.2718 Mbps.

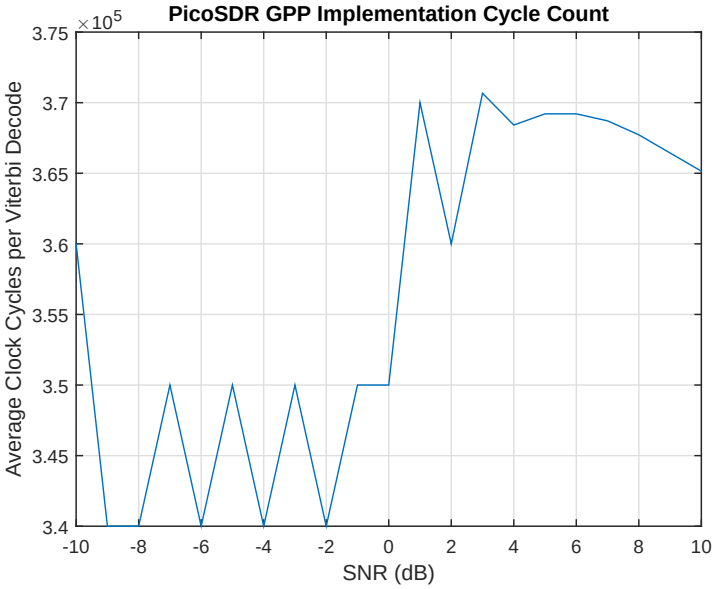

Figure 5.2: Plot of average clock cycles per Viterbi decoder for the PicoSDR GPP implementation.

5.2.2 FPGA/GPP: Nutaq PicoSDR 2x2 Embedded with Virtex-6 SX315

This section covers the FPGA/GPP implementation on the PicoSDR. This implementation had many issues, including the license for the Viterbi decoder expiring at certain points. The Nutaq MBDK product also expired, and at some points had to be reinstalled. These issues led to the decoder being unable to be successfully implemented on the actual FPGA. As such for these results, the Xilinx System Generator Viterbi decoder will be compared with the Simulink decoder.

The Xilinx Viterbi decoder is advertised as being bit and cycle accurate, so while it is not ideal that the decoder is not implemented on the hardware, it should have provided an accurate representation. Unfortunately as can be seen in Figure 5.3, the Xilinx Viterbi decoder actually underperformed the equivalent Simulink decoder. The discrepancy is large enough to believe that there is some underlying error, but it was unable to be found. This notion is strengthened by the BER curves shown in the Viterbi Decoder 7.0 Data Sheet, as seen in Figure 5.4 [11]. While the parameters in the data sheet are not identical to the ones tested in this thesis, the two should still be significantly closer than they currently are.

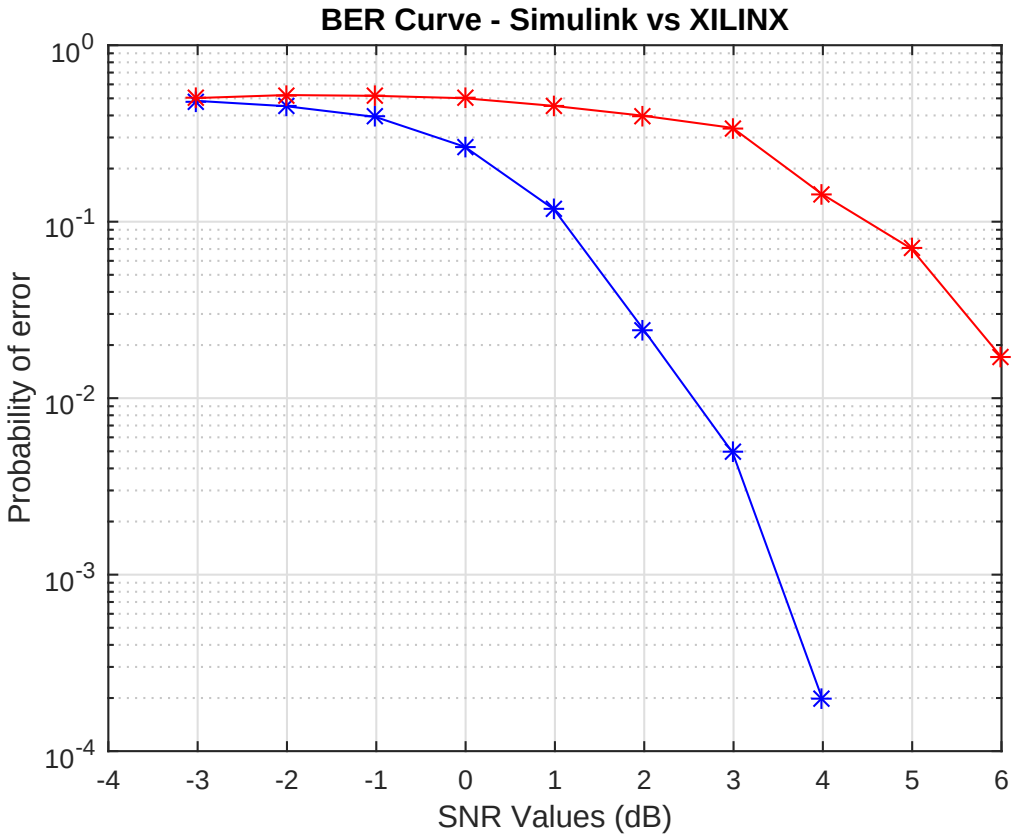

Figure 5.3: PicoSDR FPGA/GPP BER curve. The red line is the Xilinx System Generator decoder while the blue is Simulink's decoder. The large discrepancy proves that there is an error in the decoder's implementation.

The final aspect to inspect is the timing of the decoder. As previously seen in Figure 4.10, the decoder has an initial delay or latency, and then consistently has an output every clock cycle. This latency can be seen in Table 5.1, along with the corresponding traceback and constraint lengths. As this test is unable to measure the throughput of the device, it instead used the System Generator core to inspect the effects of these parameters on the latency.

5.2.3 SOC: Freescale BSC9131 RDB

The final implementation to cover is the system on chip utilizing the BSC9131 RDB. While this implementation did not have any initial simulations, the corresponding BER curve can be seen in Figure 5.5. In the next section this BER curve will be compared with

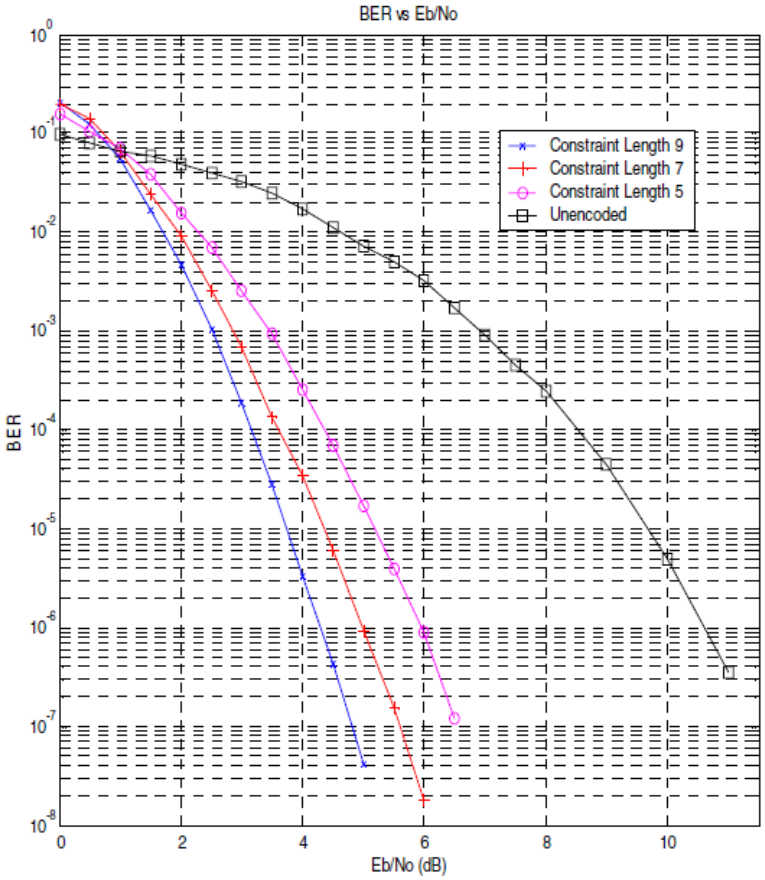

Figure 5.4: Xilinx Viterbi Decoder behavior according to the Viterbi Decoder 7.0 Data sheet [11].

Xilinx System Generator, Viterbi Implementation		
Traceback Length	**Constraint Length**	**Delay in Clock Cycles**
42	7	196
42	9	200
32	7	162
32	9	186

Table 5.1: Table showing the change in latency caused by adjusting decoder parameters.

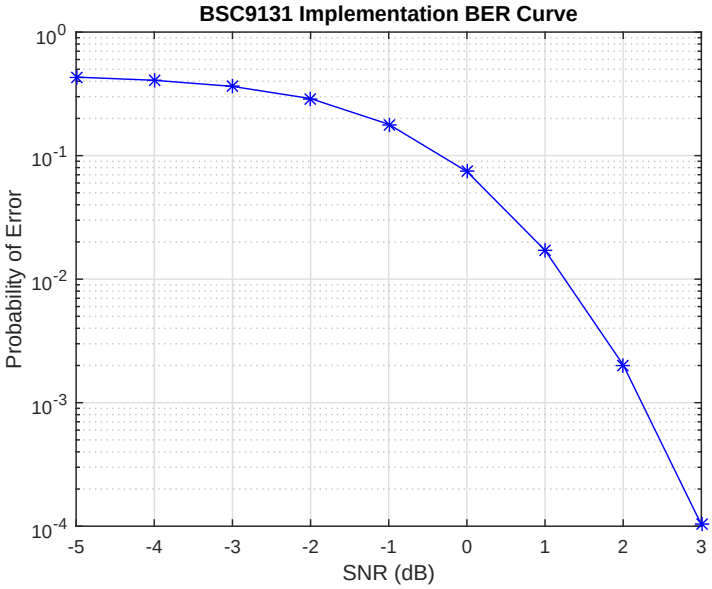

Figure 5.5: Freescale BSC9131 BER curve.

the other implementations' results. The next aspect to look at it the speed of the decoder on the Freescale device. The average time to run a single decoder can be seen in Table 5.2.

The reason for a table as opposed to a single value is to show a quality of the SOC. As can be seen in the table, increasing the number of input bits by nine-hundred percent only leads to an increase of a little over 1 microsecond in the average time it takes a Viterbi decoder to run. This is due to two reasons, the first is that the MAPLE has an initial latency as it initializes, similar to what was seen in the FPGA/GPP implementation. After this initial latency, however, it is able to perform Viterbi decoding very quickly. This means that there is always a set delay but the longer it is able to run and decode, the better an option it is. The second reason is that there is a fixed amount of overhead that comes with moving memory from the Starcore DSP to the MAPLE, and increasing the input block size makes this more efficient. A greater percentage of the input block is actual data and the BSC9131 can spend less time reading and writing smaller packets to memory. The throughput of the Viterbi decoder is 12.63 Mbps for the smaller packet size and 90.87 Mbps for the larger packet size.

System on Chip, Viterbi Implementation		
Number of Output Bits	**Number of Input Bits**	**Average Time to Run (Microseconds)**
40	96	7.60
400	816	8.98

Table 5.2: Table showing the average time to run for a single Viterbi decoder on the Freescale BSC9131, along with the corresponding input and output block sizes for the decoder.

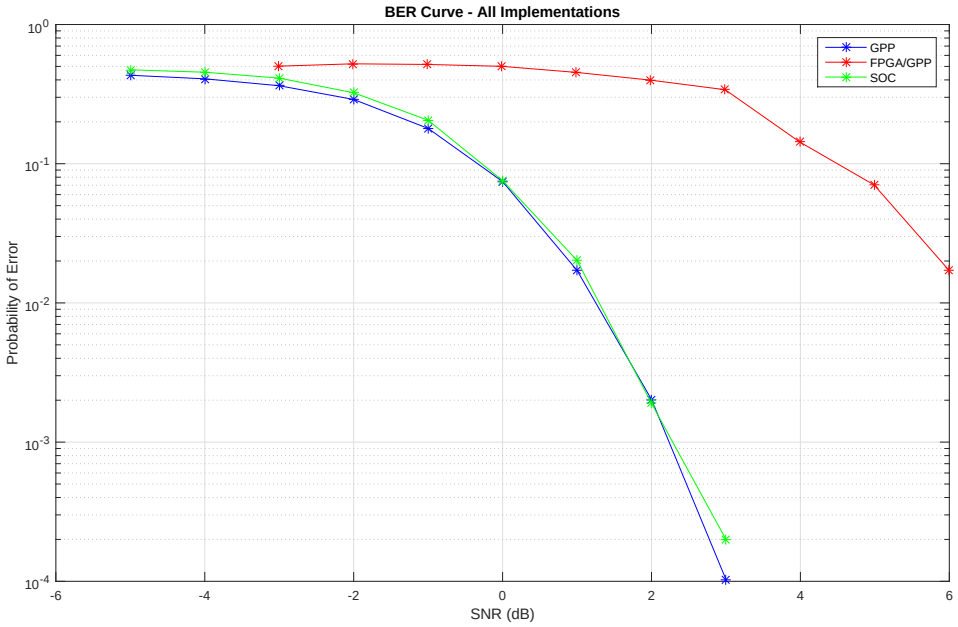

Figure 5.6: BER curves of all three implementations: GPP, FPGA/GPP, and SOC.

5.2.4 Performance Comparison

This subsection aims to compare the three implementations discussed in this chapter in both their BER performance and speed. These comparisons are made with the caveat that the FPGA/GPP implementation is not operating properly, and therefore that particular implementation is not an accurate portrayal of the platform's abilities. In terms of speed and throughput, the SOC implementation seemed to do the best by using the chip's included MAPLE. When it came to the BER curves, using identical parameters it can be seen in Figure 5.6 that the GPP and SOC were both very similar in terms of performance. The FPGA/GPP implementation once again not behaving similarly due to an error.

5.3 Summary

This chapter covered the outputs of each implementation and examined the behavior of each decoder, afterwards comparing the three using their BER curves for identical parameters. It was found that the FPGA/GPP implementation was unsuccessful, but that the GPP and SOC versions worked as expected. The next chapter will draw conclusions from these results. In addition it will suggest potential paths for future work to follow.

Chapter 6

Conclusion

This chapter summarizes the work performed for this project as well as examining the outcomes from this research. It then recommends future work that can lead to improved results as well as new research topics. The research outcomes include the working implementations on the multiple platforms and the comparison of the implementations' behaviors against one another.

6.1 Outcomes

While completing this research, the following outcomes were achieved:

- Multiple SDR platforms studied, setup, and initialized so as to be viable for implementations.
- Simulations of Viterbi decoders performed through MATLAB, Simulink, and the use of IT++ libraries in order to set a baseline for their behavior.
- Hardware implementations of Viterbi decoder successful on both GPP and SOC platforms.
- Hardware implementation of FPGA/GPP of Viterbi decoder, while not successful, laid the groundwork for future work to make adjustments.

- Comparison between the simulations and actual hardware implementations provided for each implementation where applicable.
- Timing measurements performed for both SOC and GPP implementations.
- Potential issues and roadblocks for each implementation recorded and discussed.

6.1.1 Flexibility

Flexibility is very important when it comes to development as it gives the developer options, allowing him or her to pick the path he feels most comfortable with. Doing this also helps reduce costs, as the development time will be less if the user does not need to learn how to use a new environment. In terms of flexibility, although the actual implementation was not successful, the FPGA/GPP platform performs the best. This stems from the inclusion of the GPP. The option of using a model-based design provides an alternative to those who do not know HDL or another low level language. For those that are comfortable with those languages, they are still a viable option. A sole GPP also does very well due to the ability to code in high level languages or use applicable software such as Simulink to program an SDR. It may not perform as well or as quickly though without the inclusion of an FPGA.

6.1.2 Recommendations

Based on the outcomes of this project, it was determined that specific SDR technologies are more accessible to different technical communities due to their knowledge base and skill sets. For instance, focusing on the three SDR technologies studied in this work, the following can be stated:

- SOC
  - Computer scientists and computer engineers that understand how the computer hardware operates and look at communication algorithms strictly as generic processes and do not need to deal with the hardware specifics of the SDR technology.
- Hybrid FPGA/GPP

  - Embedded hardware technologists who thoroughly understand the low-level hardware and SDR technology, but who do not completely understand the communication algorithms at the higher layers.

- GPP
  - Communication technologists can use high-level tools such as Matlab and Simulink in order to implement communication system models without needing to understand the corresponding computing hardware and algorithm development.

In order to enable collaboration of these different technical communities, this proposed testbed would allow for side by side analysis of different platforms. At the same time, in terms of exchanging models and for different communities to share implementation details, a software development framework is needed, such as Open Service for Lifecycle Collaboration (OSLC). OSLC is an open community that allows for the integration of software development by defining certain specifications [43].

6.2 Future Work

This section will discuss possible avenues for future work to take. This work can aim to either complete additional research or simply improve the testing foundation laid here. The first goal of any future work directly based off this research may be to adjust the FPGA/GPP implementation, but there are other directions that could be areas of focus. An example is increasing the accuracy and resolution of the timing measurements taken. Currently the GPP implementation simply uses the standard clock function and the FPGA/GPP timing was unable to be measured. As such work here could allow for a more in depth comparison of the speed and performance of each platform.

6.2.1 Improvements to Implementation

Possible improvements to the implementation are:

- For the FPGA/GPP implementation, use the RTDEx connection in order to incorporate the GPP into the design. As well as provide an easier testing environment.

- For the SOC implementation, include the BER calculations in the actual code. This would prevent the calculations needing to be done in post processing and possibly increase prototyping speed.
- For the GPP implementation the resolution of the timing calculations can be increased. Also an alternative to the basic clock function used so as to increase accuracy.

6.2.2 Single Interface

A major area for future work would be developing a single interface for these three platforms to be tested on. Current implementations are separated and have varying interfaces, making it difficult to test many algorithms at once. An approach akin to using a class wrapper around the decoder functions, with a parameter to select which platform to create inputs for, could be used here to bring all three implementations together. The wrapper would operate by accepting certain inputs and input types and then converting them into a form that can be understood by each individual platform. An example is that if the input to the encoder was defined as a vector of random boolean values. If one platform could only operate on a vector of integers, while another platform could only understand an input of a single integer at a time, the wrapper would adjust the input to accommodate the platform's construction. In the first platforms case it would simply cast each element in the array from a boolean to an integer value. For the second platform, it would first recast the inputs and then feed them into a type of First In First Out (FIFO) buffer, allowing the inputs to be fed to the platform one at a time. The use of an identical interface also provides the advantage of the same BER curve generating code being able to be used for each test, assuring that the calculations are the same for both.

Bibliography

[1] M. Khurram and S. Mirza, "A General Purpose Processor Based IEEE802.11a Compatible OFDM Receiver Design," in *GCC Conference (GCC), 2006 IEEE*. IEEE, 2006, pp. 1 – 5. [Online]. Available: http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=5686202

[2] *FPGA Fundamentals*, National Instruments Corporation, 2012. [Online]. Available: http://www.ni.com/white-paper/6983/en/

[3] J. Proakis, *Digital Communications*. McGraw-Hill, 1995.

[4] W. J. Ebel, "Turbo Code implementation on the C6x." [Online]. Available: http://focus.ti.com/pdfs/univ/3-Wireless.pdf

[5] Q. Li, X.-Z. Li, H.-H. Jiang, and W.-H. He, "A High-Speed Viterbi Decoder," in *Natural Computation, ICNC '08. Fourth International Conference on*. IEEE, 2008, pp. 313 – 316. [Online]. Available: http://ieeexplore.ieee.org/xpl/articleDetails.jsp?tp=&arnumber=4667991

[6] M. Valenti and R. I. Seshadri, "Turbo and LDPC Codes: Implementation, Simulation, and Standardization." [Online]. Available: http://www.csee.wvu.edu/~mvalenti/documents/TurboLDPCTutorial.pdf

[7] Y. Lin, S. Mahlke, T. Mudge, C. Chakrabarti, A. Reid, and K. Flautner, "Design and implementation of turbo decoders for software defined radio," *Signal Processing Systems Design and Implementation*, pp. 22 – 27, 2006. [Online]. Available: http://ieeexplore.ieee.org/xpl/articleDetails.jsp?tp=&arnumber=4161819

[8] *Nutaq PicoSDR: FPGA-based, MIMO-Enabled, Tunable RF SDR Solutions*, Nutaq, 2014. [Online]. Available: http://nutaq.com/sites/default/files/PicoSDR_V1.4_02_14_2014_web.pdf

[9] "Model-based design kit (mbdk)," Nutaq.

[10] *BSC9131 QorIQ Qonverge Multicore Baseband Processor*, Freescale Semiconductor Inc., 2014. [Online]. Available: http://cache.freescale.com/files/32bit/doc/data_sheet/BSC9131.pdf

[11] *LogiCORE IP Viterbi Decoder v7.0*, XILINX, 2010.

[12] M. S. Safadi and D. L. Ndzi, "Digital Hardware Choices For Software Radio (SDR) Baseband Implementation," *Information and Communication Technologies*, vol. 2, pp. 2623 – 2628, 2006. [Online]. Available: http://ieeexplore.ieee.org/xpl/articleDetails.jsp?arnumber=1684823

[13] R. Joost and R. Salomon, "Advantages of FPGA-Based Multiprocessor Systems in Industrial Applications," in *Industrial Electronics Society 31st Annual Conference of IEEE*. IEEE, 2005, pp. 445 – 450. [Online]. Available: http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=1568946

[14] A. M. Wyglinski and D. Pu, *Digital Communication Systems Engineering with Software-Defined Radio*. Artech House, 2013.

[15] T. Collins, "Implementation and Analysis of Spectral Subtraction and Signal Separation in Deterministic Wide-Band Anti-Jamming Scenarios," Master's thesis, Worcester Polytechnic Institute, 2013.

[16] J. Glossner, D. Iancu, M. Moudgill, S. Jinturkar, G. Nacer, S. Stanley, A. Iancu, H. Ye, M. Schulte, M. Sima, T. Palenik, P. Farkas, and J. Takala1, "Implementing Communications Systems on an SDR SoC," in *Acoustics, Speech and Signal Processing, 2008. ICASSP 2008. IEEE International Conference on*. IEEE, 2008, pp. 5380 – 5383. [Online]. Available: http://ieeexplore.ieee.org/xpl/articleDetails.jsp?arnumber=4518876

[17] P. Mackenzie, L. Doyle, K. Nolan, and D. O'Mahony, "Software Radio on General-Purpose Processors," in *Proceedings of the Irish Signals and Systems Conference, ISSC*, 2001. [Online]. Available: https://www.cs.tcd.ie/~omahony/iei-swr.pdf

[18] V. G. Bose, "Design and Implementation of Software Radios Using a General Purpose Processor," Ph.D. dissertation, Massachusetts Institute of Technology, 1999.

[19] J.-O. Jeong, "Hybrid FPGA and GPP Implementation of IEEE 802.15.4 Physical Layer," Master's thesis, Virginia Polytechnic Institute and State University, 2012.

[20] A. S. Rodriguez, M. C. M. Jr., I. S. Ahn, and Y. Lu, "Model-based Software-defined Radio(SDR) Design Using FPGA," in *Electro/Information Technology (EIT), 2011 IEEE International Conference on.* IEEE, 2011, pp. 1 – 6. [Online]. Available: http://ieeexplore.ieee.org/xpl/articleDetails.jsp?arnumber=5978602

[21] A. I. Mecwan and N. P. Gajjar, "Implementation of Software Defined Radio on FPGA," in *Engineering (NUiCONE), 2011 Nirma University International Conference on.* IEEE, 2011, pp. 1 – 5. [Online]. Available: http://ieeexplore.ieee.org/xpl/articleDetails.jsp?arnumber=6153271

[22] D. J. Greaves, "System on Chip Design and Modeling." [Online]. Available: http://www.cl.cam.ac.uk/teaching/1011/SysOnChip/socdam-notes1011.pdf

[23] F. Vahid, *Digital Design with RTL Design, VHDL, and Verilog.* Wiley, 2011.

[24] *About IT++*, 2013. [Online]. Available: http://itpp.sourceforge.net/4.3.1/

[25] *Simulink: Simulation and Model-Based Design*, MathWorks, 2014. [Online]. Available: http://www.mathworks.com/products/simulink/

[26] *Xilinx System Generator and HDL Coder*, MathWorks, 2014. [Online]. Available: http://www.mathworks.com/solutions/fpga-design/simulink-with-xilinx-system-generator-for-dsp.html

[27] *CodeWarrior Embedded Software Development Tools*, Freescale Semiconductor Inc.,

2014. [Online]. Available: http://www.freescale.com/webapp/sps/site/homepage.jsp?code=CW_HOME

[28] B. Stern, *QorIQ Qonverge Portfolio: Next-Generation Wireless Network Bandwidth and Capacity Enabled by Heterogeneous and Distributed Networks*, Freescale.

[29] J. Gao, "Viterbi." [Online]. Available: http://www3.cs.stonybrook.edu/~jgao/CSE590-fall09/viterbi.pdf

[30] R. Ktter, A. Singer, and M. Tchler, "Turbo Equalization," *IEEE Signal Processing Mag*, vol. 20, p. 6780, 2004. [Online]. Available: http://www.comm.csl.uiuc.edu/~koetter/publications/spmag.pdf

[31] "Error detection and correction," MathWorks.

[32] S. ten Brink, J. Speidel, and R.-H. Yan, "Iterative Demapping and Decoding for Multilevel Modulation," in *Global Telecommunications Conference, 1998. GLOBECOM 1998. The Bridge to Global Integration. IEEE.* IEEE, 1998, pp. 579 – 584. [Online]. Available: http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=775793

[33] L. Wei, "Iterative Viterbi Algorithm: Implementation Issues," *Wireless Communications, IEEE Transactions on*, vol. 3, pp. 382 – 386, 2004. [Online]. Available: http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=1271231

[34] C.-Y. Chu and A.-Y. Wu, "Power-Efficient State Exchange Scheme for Low-Latency SMU Design of Viterbi Decoder," *Journal of Signal Processing Systems*, vol. 68, p. 233245, 2012).

[35] C. Herzet, N. Noels, V. Lottici, H. Wymeersch, M. Luise, M. Moeneclaey, and L. Vandendorpe, "Code-Aided Turbo Synchronization," *Proceedings of the IEEE*, vol. 95, pp. 1255–1271, 2007. [Online]. Available: http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=4282126

[36] W. E. Ryan, "A Turbo Code Tutorial." [Online]. Available: http://vada.skku.ac.kr/ClassInfo/digital-com2000/slides/turbo2c.pdf

[37] *PCI Express An Overview of the PCI Express Standard*, National Instruments, 2014. [Online]. Available: http://www.ni.com/white-paper/3767/en/

[38] *System Generator for DSP*, XILINX, 2010.

[39] "Viterbi decoder," MathWorks.

[40] *RTDEx Programmer's Reference Guide*, Nutaq, 2014.

[41] *BSC9131 QorIQ Qonverge Multicore Baseband Processor Reference Manual*, Freescale Semiconductor Inc., 2013.

[42] *Multi Accelerator Platform Engine, Baseband 2 for Femto (MAPLE-B2F) Reference Manual*, Freescale Semiconductor Inc., 2012.

[43] "Open Services for Lifecycle Collaboration." [Online]. Available: http://open-services.net/