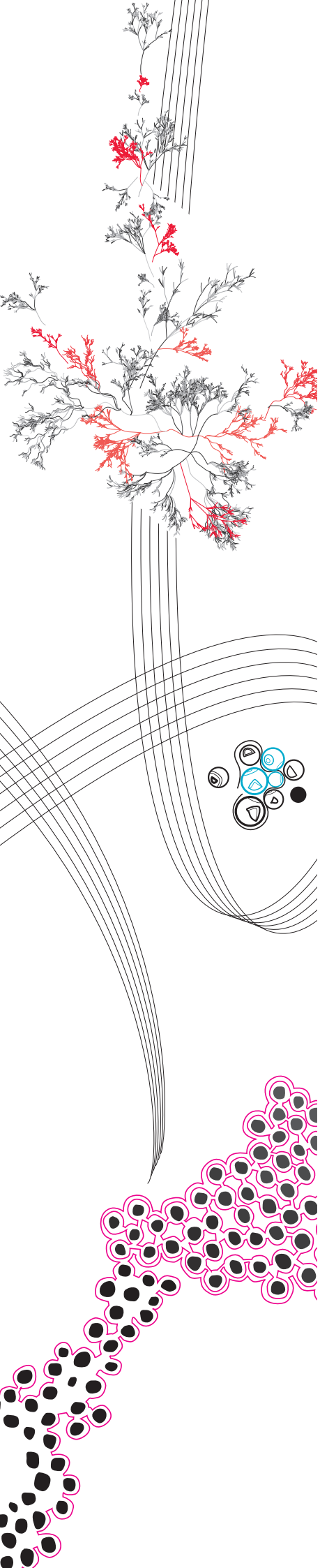


MASTER THESIS

DESIGN OF A GMSK RECEIVER PROTOTYPE ON A HETEROGENEOUS REAL-TIME MULTIPROCESSOR PLATFORM

Daniël van der Veer

Faculty of Electrical Engineering, Mathematics and Computer Science
Computer Architecture for Embedded Systems

EXAM COMMITTEE:
Prof. dr. ir. M.J.G. Bekooij
Dr. ir. A.B.J. Kokkeler
Ir. J. Scholten
G. Kuiper, M.Sc.

15 January 2016

UNIVERSITY OF TWENTE.

ABSTRACT

As computers become faster and smaller, there is an increasing demand for low power computing devices. These devices are used in a broad spectrum of applications; this includes heart rate monitors, home automation and environmental monitoring like air quality measurements. In many cases these devices need to operate for months or years on a small battery. This requires a new generation of ultra low power communication chips.

Bluetooth has an update of the standard, Bluetooth Low Energy Long Range (BLR), which increases the range for the low energy mode of Bluetooth. Bluetooth Low Energy and the long range successor are designed specifically for these ultra low power applications.

An embedded system which does wireless communication processing must combine high processing speed with low power consumption. The multiprocessor system-on-chip (MPSoC) is increasingly used in low power embedded systems. By integrating multiple processors into one system, high computation power can be achieved while keeping power consumption low.

In the CAES research group at the University of Twente research is done on the design of heterogeneous multiprocessor embedded platforms for real-time stream processing applications. One prominent example of these systems is called Starburst.

This thesis describes an implementation of a BLR receiver on the Starburst platform. The receiver performs non-coherent differential detection. The different tasks of the receiver are implemented as separate accelerators, connected to the communication ring. An equalizer, convolutional decoder and repetition decoder were added to the receiver and the bit error rate (BER) performance of different receiver configurations was measured. The receiver configuration with the best BER performance is the equalizer with matched filter, which performs 2.5dB less [1] than the theoretical DQPSK BER curve.

A receiver architecture was designed and implemented where a different filter can be used for frame detection than for decoding. This increases the frame detection rate when the receiver is used with the matched filter and makes the receiver more resilient to frequency offset.

The throughput of the receiver on the Starburst platform was analyzed. The receiver was initially implemented as software tasks running on the Microblaze processors, but was running 6 to 34 times too slow. An implementation of hardware accelerators connected to

1 2.5dB difference between $\mathcal{E}_b/N_0$ values at 10^{-3} BER

the communication ring is fast enough and each task required less hardware than a Microblaze processor.

The fully functional receiver is able to communicate wireless with a transmitter to play an audio stream. It was used as a demonstrator at the University Booth at the Design, Automation and Test in Europe (DATE) 2015 conference[15].

Further research is required on the low BER performance of the repetition decoder. It would also be interesting to investigate increasing the coding gain of the convolutional decoder. Both repetition and convolutional coding are part of the BLR standard, and the gain of the codes is intended to increase the range. However, the gain of both decoders is lower than expected in the current receiver implementation. It would be interesting to investigate to cause of this and research possible improvements.

Another possible way to improve the sensitivity of the receiver is to do coherent detection. There is potentially a significant improvement in sensitivity with coherent detection. It is however not clear how difficult it is to do the phase synchronization required for coherent detection, and how much the increase in hardware costs is.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACRONYMS

ADC analog-to-digital converter

AMBA Advanced Microcontroller Bus Architecture

AWGN additive white Gaussian noise

BER bit error rate

BLE Bluetooth Low Energy

BLR Bluetooth Low Energy Long Range

BPSK binary phase shift keying

CAES Computer Architecture for Embedded Systems

DAC digital-to-analog converter

DPSK differential phase shift keying

DQPSK differential quaternary phase shift keying

DSP48 Xilinx digital signal processing (DSP) processing slice[32]

DSP digital signal processing

FCFS first-come-first-served

FIFO first-in-first-out buffer

FIR finite impulse response

FPGA field-programmable gate array

FSK frequency shift keying

GMSK Gaussian minimum shift keying

GSM global system for mobile communications

IC integrated circuit

I/O input/output

IoT Internet of things

ISI intersymbol interference

MF matched filter

MLSE maximum-likelihood sequence estimation

MPSoC multiprocessor system-on-chip

MSK minimum shift keying

NI network interface

NoC network-on-chip

OQPSK offset quaternary phase shift keying

PAM pulse-amplitude modulation

PLL phased locked loop

PSK phase shift keying

PWM pulse-width modulation

QPSK quaternary phase shift keying

RF radio frequency

SDF synchronous data flow

SDR software-defined radio

SIMD single instruction, multiple data

SNR signal-to-noise ratio

SODA Signal-Processing on Demand Architecture [17]

VA Viterbi algorithm

VHDL VHSIC (very high speed integrated circuit) hardware description language

WGN white Gaussian noise

1

INTRODUCTION

1.1 CONTEXT

Computers are being used in more situations in life. Increasingly devices not normally associated with computers are made 'smart' by adding small computer and wireless communication to them. This so called Internet of things (IoT) requires a new generation of ultra low power communication chips.

Bluetooth is a wireless communication protocol originally conceived as a way to wirelessly connect computer peripherals. Today, people own more and more devices with integrated electronics that communicate wirelessly with each other. Many of these devices are battery-powered. The introduction of Bluetooth Low Energy (BLE) in 2010 focuses more on these low powered devices, by reducing the power and cost of receivers. To increase the range of BLE, there is now being worked on a new version called BLR.

In the past, communication chips were operating for the most part in the analog domain. While in the recent decades more and more processing has moved to the digital domain, only recently have processors become fast and small enough to be able to implement parts of the baseband in software. The appeal of software-defined radio (SDR) is clear; running the processing in software requires less programming effort and increases flexibility of the transmitter/receiver.

Wireless communication chips require computation that is low cost, low power, high throughput and real-time. Low cost means a chip that uses few hardware resources. High throughput and real-time require a platform that not only has enough computing power but for which also suitable throughput analysis techniques exist to derive the minimum throughput. For real-time computing an analysis of the executing times of the tasks in the application is required.

The multiprocessor system-on-chip (MPSoC) is increasingly used in low power embedded systems. By integrating multiple processors into one system, high computation power can be achieved while keeping power consumption low. If these MPSoCs are designed properly, guarantees can be given for the minimum throughput. One challenge in systems with many processing elements is the inter-core communication, which requires a network-on-chip (NoC).

Starburst is a MPSoC platform developed at the Computer Architecture for Embedded Systems (CAES) research group on the University of Twente. The platform is aimed for real-time streaming applications. It is designed to be flexible, where the platform is not designed for one specific application, but rather can support applications not known at design time. Furthermore, the platform should be transparent to the programmer, requiring minimal platform-specific implementation. Starburst features a low-cost interconnect in the form of a unidirectional communication ring. Both processors and hardware accelerators are connected to the ring.

1.2 PROBLEM DESCRIPTION

While a number of improvements have been developed for the Starburst, very few applications are implemented on it. There has been a PAL decoder, which was partially developed in software and partially in hardware. PAL is however a relatively old standard, most countries have phased out terrestrial PAL television broadcasts.

The Bluetooth standard is becoming increasingly relevant with the rise of applications requiring low-powered communication: from wireless headsets to fitness watches. The new Bluetooth Low Energy Long Range (BLR) standard specifies the transmitter, but not the receiver. This thesis discusses the implementation of a Bluetooth Low Energy Long Range-like GMSK receiver with an evaluation of the bit error rate performance.

The research addresses two main questions:

- How can a low-cost and flexible GMSK, BLR-like, receiver be realized by extending the Starburst platform?
- What are suitable reception improvement techniques for such a receiver?

1.2.1 *Contributions*

The main contributions described in this thesis are

- The implementation and evaluation of a real-time GMSK receiver on the Starburst platform using stream processing hardware accelerators.
- The implementation of improvements of a BLR-like receiver that increases sensitivity.
  - An important improvement in the receiver architecture is the case of separate filters for frame detection and decoding which results in a receiver with multiple modes.

1.3 RELATED WORK

In this section we present published work related to our research. Specifically, we look at different GMSK receiver implementations, possible improvements that increase the sensitivity of the receiver and are compatible with the BLR standard. We also look at possible hardware platforms that are suitable for a GMSK receiver.

1.3.1 *Receivers*

In [22] a GMSK receiver is described for use by the European Space Agency (ESA) for communication with probes in deep space. Due to additional requirements related to navigation, a coherent receiver is chosen. This improves sensitivity of the receiver, but requires carrier synchronization that increases the implementation cost.

A non-coherent differential demodulator with decision feedback is described in [34]. A differential demodulator has low implementation cost. In the paper, a GMSK signal with $BT = 0.25$ is used which causes controlled intersymbol interference (ISI) in the signal. The decision feedback logic in the receiver increases sensitivity with the additional ISI. In the Bluetooth standard GMSK with $BT = 0.5$ is used, which has significantly less ISI. Decision feedback will have a reduced benefit in that case.

So called 2-bit differential detection increases sensitivity by detecting symbols over 2 symbol times. This technique is discussed in [23]. By doing differential detection over two symbol times instead of one, the sensitivity is increased and the receiver is less vulnerable to ISI. This does however require a change in the transmitter by adding a differential encoder. In standards like Bluetooth it is not possible to change the transmitter because this is specified by the standard.

1.3.2 *Receiver Improvements*

Equalizers improve the receiver sensitivity be reducing ISI in the signal. In [4] an adaptive equalizer is added to a GSM (GMSK with $BT = 0.3$) receiver with convolutional coding. This leads to a 2dB sensitivity increase when the hard decision equalizer is used. This is for a receiver with an interleaver.

An equalizer with soft output achieves better BER performance than with hard output. A hard output equalizer only outputs the resulting bit, while a soft equalizer outputs additional bits indicating the likelihood of the estimate. In [10] an equalizer with a soft output Viterbi algorithm is described. This results in an advantage of more than 3dB for GMSK with $BT = 0.25$. With a Bluetooth GMSK signal the BT value is higher (0.5) and the sensitivity increase is therefore expected to be lower.

Channel coding can improve sensitivity of the receiver at the cost of a reduced data rate. There are two types of channel coding: convolutional and block coding. The difference between the two is that in block codes the encoded sequence is only dependent on the current input, while with convolutional coding the previous input is also used.

Due to its efficient implementation, convolutional decoding is very often performed with the Viterbi algorithm. However, with larger (>10) constraint lengths the complexity of Viterbi decoding becomes prohibitive [20]. Sequential decoders are then often used, which perform better under good signal-to-noise ratio (SNR) range, but become prohibitively slow at low SNR [8].

Repetition coding is a block code that has limited error correcting ability, but is very easy to decode. In [2] repetition decoding is analyzed for frequency shift keying (FSK) and differential phase shift keying (DPSK). This paper however only discusses majority voting, which only uses the hard decisions of a demodulator. A better BER performance is expected if soft information from the demodulator is used.

Whether a block code or convolutional code is better depends on the application. Sometimes getting a high coding gain requires a convolutional decoder with high constraint length, resulting in a high hardware cost. In this case concatenating a convolutional code with a block code may reach the same coding gain with a lower implementation cost. This is why BLR uses two concatenated codes. The outer code is a rate $^1/_2$ convolutional code and the inner code is a rate $^1/_4$ repetition code.

1.3.3 *Multiprocessor System-on-Chip*

The choice of hardware platform is mainly determined by the application that will run on it. In our case the GMSK receiver will be running as multiple tasks, but it is expected that not all (parts of the) tasks run at the desired speed on processors. It is therefore desirable to be able to move certain parts of the receiver to task-specific accelerators.

CoMPSoC, an architecture for real-time streaming applications, is introduced in [11]. This platform maps tasks to virtualized instances of processors. This approach ensures tasks do not interfere with each other in terms of resource usage. This allows guarantees for performance to be given independent of other tasks on the system. The platform does not support integration of hardware accelerators. The hardware cost of the Æthereal interconnect is high.

In [28] a platform is described with 167 homogeneous processors connected through a mesh interconnect. The processors have both local and long distance links to communicate, where the connections are circuit-switched. There are processing units to speed up FFT, Viterbi

decoding and video motion estimation. All processors have an independent local oscillator, so that unused cores can be disabled individually. The platform is not flexible however: while the previous platform could be implemented on an field-programmable gate array (FPGA), this is a integrated circuit (IC) design. It is therefore also not possible to implement custom accelerators.

Signal-Processing on Demand Architecture [17] (SODA) is an architecture for wireless baseband processing. It features an ARM control processor and 4 DSP cores. The cores have a local memory and there is a global shared memory. The DSP processors support single instruction, multiple data (SIMD) instructions which result in high performance for algorithms with high data-level parallelism. In the SDR domain there is however more task-level parallelism than data-level parallelism [6], limiting the usefulness of SIMD. Furthermore, although the DSP processors have hardware acceleration for certain algorithms (e. g. FFT and Viterbi), there is no support for custom hardware accelerators.

While heterogeneous MPSoCs designed for specific applications can generally reach higher performance, this reduces flexibility of the platform. RAMPSoC [9] is a heterogeneous MPSoC platform that increase the flexibility and programmability by doing application-specific partial reconfiguration of the platform. The workflow for RAMPSoC goes like this; an application is written in C code and profiled to determine the process performance and inter-process communication. Based on this the tasks are mapped to processors and another profile is done to determine the bottlenecks in the application. This profile determines the parts of the application that are best mapped to hardware accelerators. The hardware/software platform is than configured to the FPGA. This approach can potentially improve performance of applications because the processors and NoC are instantiated specifically for the application. While the use case (an image processing application) mentioned in the paper runs on a real-time operating system, no real-time analysis of the inter-processor communication is mentioned.

The Starburst platform has a number of features. The NoC has a high throughput and low hardware cost while still having real-time properties and analyzability. A GMSK receiver is potentially a very suitable application for the Starburst platform. The author in [6] analyzes different MPSoC platforms for their real-time properties and the cost, scalability and performance of their NoCs. The Starburst platform comes out having very favorable properties. The GMSK receiver is not the first receiver implemented on the Starburst platform. A PAL decoder has been developed. This started as a software implementation [5] using 16 Microblaze cores. In [12] support for hardware accelerators on the ring is added and a part of the receiver was implemented with a hardware accelerator. Wevers [30] implemented

accelerator sharing functionality and had a PAL audio decoder as a case study. The audio decoder was done with hardware accelerators, where the accelerators were reused when possible.

1.3.4 *Summary*

A number of receivers is not suitable for the Bluetooth application. The 2-bit differential detection requires modification of the transmitter. The differential demodulator with decision feedback is more suitable for GMSK signals with lower BT values. With a $BT = 0.5$ signal the extra hardware cost may not be worth it. Coherent detection, while achieving high sensitivity, requires complex phase synchronization. Our non-coherent receiver is more suitable for a Bluetooth application, due to the low hardware costs.

For convolutional decoding a sequential decoder is not suitable for real-time applications. The worst case execution time for a sequential decoder, which occurs at low SNR, is not better than a MLSE convolutional decoder. Indeed, a linear decoder has much worse performance at low SNR than a MLSE decoder [8]. A Viterbi decoder therefore appears to be a much better choice.

From the MPSoCs listed above, Starburst is the only real-time system that supports application-specific accelerators. RAMPSoC does support custom accelerators and the software tasks run on a real-time OS, but it is unclear whether real-time throughput guarantees can be provided for the interconnect.

1.4 OUTLINE

This thesis is organized in the following way. The next chapter has background information on GMSK receivers and GMSK modulation. Chapter 3 starts with background information on the Starburst platform and then discusses a baseline implementation of a GMSK receiver. Chapter 4 discusses improvements for the receiver in the form of a Viterbi decoder accelerator which is used for both convolutional decoding and equalization. Chapter 5 introduces a mode switching receiver architecture to resolve limitations of frame detection when the receiver is used with a so called matched filter. Chapter 6 discusses repetition coding as channel coding improvement for the receiver. Finally, in Chapter 7 we present the conclusions and future work.

2

GMSK RECEIVERS

This chapter discusses the background information related to the GMSK receiver. The first section starts with a discussion of the theory of the GMSK modulation technique and the related minimum shift keying (MSK) and QPSK signaling. Section 2.2 introduces the test environment for the GMSK transmitter and receiver. This runs on GNU Radio, which is an open source SDR software package.

2.1 MODULATION

GMSK applies a Gaussian pulse shaping filter to a MSK signal to reduce the bandwidth of the transmitted signal at the expense of more difficult detection. Due to its very efficient bandwidth and power usage, GMSK is a widely used modulation technique, with applications in global system for mobile communications (GSM), Bluetooth and (deep) space communication [22].

This section will look at the theory behind GMSK to determine the theoretical bit error probability of the modulation. To do this we will mainly focus on minimum shift keying (MSK), of which GMSK is a special case. MSK itself is a special case of frequency shift keying (FSK), but can also be viewed as offset quaternary phase shift keying (OQPSK) with a sine pulse wave applied. First we will discuss FSK and OQPSK separately, then MSK is discussed in relation to both FSK and OQPSK.

2.1.1 *Frequency Shift Keying*

In frequency shift keying (FSK) the information is modulated in the frequency of the signal. In this section we will limit the discussion to binary FSK, where the symbols only have two possibilities: $a_n \in \{-1, 1\}$. Each symbol is mapped to a frequency. The modulated signal for one symbol a_n can be written as:

$$s_n(t) = \begin{cases} \sqrt{\frac{2\mathcal{E}}{T}} \cos(2\pi f_0 t) & a_n = -1 \\ \sqrt{\frac{2\mathcal{E}}{T}} \cos(2\pi f_1 t) & a_n = 1 \end{cases} \tag{2.1}$$

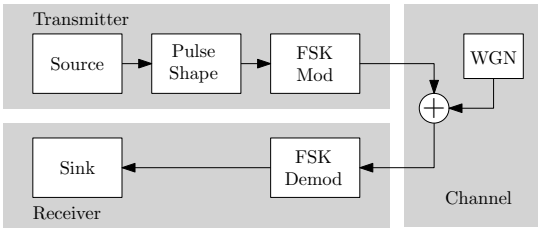


Figure 2.1: Detection of FSK signal transmitted over WGN channel

This can also be written as:

$$s_n(t) = \sqrt{\frac{2\mathcal{E}}{T}} \cos(2\pi f_c t + \phi_n(t)) \tag{2.2}$$

$$\phi_n(t) = 2\pi a_n \Delta f t \tag{2.3}$$

Where f_c is the carrier frequency and $\Delta f = (f_1 - f_0)/2$ is the frequency separation from f_c.

We can generalize the phase signal for transmitting multiple signals. $\phi(t)$ for $nT \leqslant t < (n+1)T$ then becomes:

$$\phi(t) = 2\pi\Delta f T \sum_{k=-\infty}^{n} a_k q(t - kT) \tag{2.4}$$

Where $q(t) = \int_{-\infty}^{t} g(\tau)d\tau$ and $g(t)$ is the transmission pulse. The pulse shape signal $g(t)$ can be used to shape the spectrum of the transmitted signal to reduce spectral width or reduce ISI. In conventional FSK the pulse shape is a rectangular pulse:

$$g(t) = \begin{cases} \frac{1}{2T} & 0 \leqslant t \leqslant T \\ 0 & \text{otherwise} \end{cases} \tag{2.5}$$

2.1.1.1 *Bit Error Probability*

The FSK modulated signal is transmitted over a channel with added white gaussian noise. Figure 2.1 shows a schematic of a FSK transmitter and receiver. Due to delay between the transmitter and receiver, a phase offset ϕ_0 is also present.

First we will assume the phase offset ϕ_0 to be zero. This is true if the receiver synchronizes with the received signal through for example a phased locked loop (PLL). This type of detection is called coherent detection. [20] gives the probability of an error in a transmitted bit for orthogonal FSK:

$$P_b = Q\left(\sqrt{\frac{\mathcal{E}_b}{N_0}}\right) \tag{2.6}$$

The synchronization of the receiver clock with the transmitter can be difficult and costly in terms of implementation cost. It may be advantaguous to leave the phase offset non-zero. In [20] the optimal noncoherent detection is given as a filterbank of matched filters which are

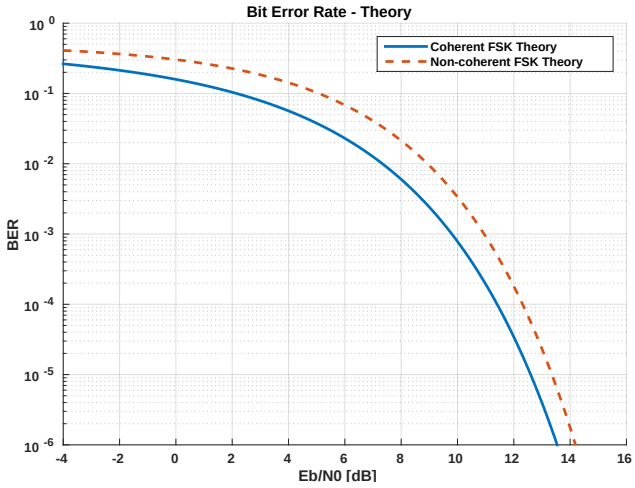


Figure 2.2: Bit Error Rate for coherent and non-coherent FSK detection

integrated over the symbol time. The matched filters are the possible transmitted signals for a 0 or 1 transmitted. The filterbank output with the highest output has the greatest correlation and is therefore the most probable transmitted symbol. The bit error probability is given as:

$$P_b = \frac{1}{2} e^{-\frac{\mathcal{E}_b}{2N_0}} \tag{2.7}$$

Figure 2.2 shows a comparison for the error probabilities of both coherent and non-coherent detection. The difference for higher $\mathcal{E}_b/N_0$ values is less than 0.8dB.

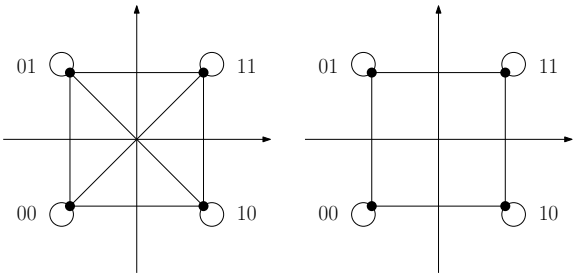
2.1.2 *Phase Shift Keying*

In phase shift keying (PSK) the information is modulated in the phase of the signal. If the signal has a carrier frequency f_c and the symbol has a alphabet of M symbols, the m^{th} symbol can be given as:

$$\begin{aligned} s_m(t) = &\sqrt{\frac{\mathcal{E}_g}{2}} \cos\left(\frac{2\pi}{M}(m-1)\right) \phi_1(t) \\ &+\sqrt{\frac{\mathcal{E}_g}{2}} \sin\left(\frac{2\pi}{M}(m-1)\right) \phi_2(t) \end{aligned} \tag{2.8}$$

$$\phi_1(t) = \sqrt{\frac{2}{\mathcal{E}_g}} h(t) \cos 2\pi f_c t \tag{2.9}$$

$$\phi_2(t) = -\sqrt{\frac{2}{\mathcal{E}_g}} h(t) \sin 2\pi f_c t \tag{2.10}$$



(a) QPSK (b) OQPSK

Figure 2.3: Phase changes for QPSK and OQPSK in IQ constellation

This can be rewritten in quadrature components:

$$s_m(t) = \Re\left\{\left(s_{m,I}(t) + j s_{m,Q}(t)\right) e^{j(2\pi f_c t + \phi_0)}\right\} \tag{2.11}$$

$$s_{m,I}(t) = h(t)\cos\left(\frac{2\pi}{M}(m-1)\right) \tag{2.12}$$

$$s_{m,Q}(t) = h(t)\sin\left(\frac{2\pi}{M}(m-1)\right) \tag{2.13}$$

In Figure 2.3a the mapping of the phases in a IQ constellation are shown for Gray-coding. In quaternary phase shift keying (QPSK) the signal can have a 180° phase change between symbols. In this transition the signal passes through the origin, as can be seen in Figure 2.3a. offset quaternary phase shift keying (OQPSK) prevents this by offsetting the Q signal by a half symbol time. This results in only 0° and ±90° phase changes. Figure 2.3b shows the phase mapping for OQPSK. Note that the phase change lines do not pass through the origin anymore.

2.1.2.1 *Bit Error Probability*

The bit error probability for coherent detection of binary phase shift keying (BPSK) and QPSK is given in [20, p192]:

$$P_b = Q\left(\sqrt{\frac{2\mathcal{E}_b}{N_0}}\right) \tag{2.14}$$

A downside of PSK is that, because the information is in the absolute phase of the signal, any phase offset of the receiver clock with the transmitter is bad for the detection. DPSK encodes the symbols as phase changes instead of absolute phases. In differential QPSK this means a symbol of two bits is mapped to a ±180° or ±90° phase change relative to the current phase. This leads to lower complexity implementations of the detector, but at the cost of higher error probabilities.

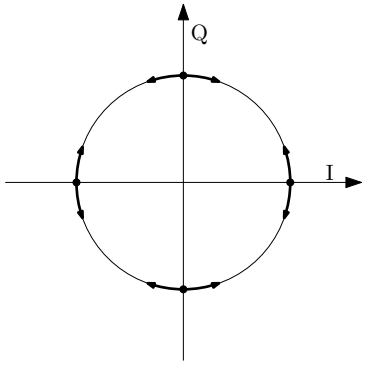


Figure 2.4: Phase changes for MSK in IQ constellation

The bit error probability for differential quaternary phase shift keying (DQPSK) is [20, p225]:

$$P_b = Q_1(a,b) - \frac{1}{2} I_0(ab) e^{-\frac{a^2+b^2}{2}} \tag{2.15}$$

$$a = \sqrt{\frac{2\mathcal{E}_b}{N_0}\left(1 - \sqrt{\frac{1}{2}}\right)} \tag{2.16}$$

$$b = \sqrt{\frac{2\mathcal{E}_b}{N_0}\left(1 + \sqrt{\frac{1}{2}}\right)} \tag{2.17}$$

Where $Q_1(a,b)$ is the Marcum Q-function and $I_0(x)$ is the modified Bessel function of order zero.

2.1.3 *Minimum Shift Keying*

In FSK the signal is orthogonal if the frequency seperation is $\Delta f = l/4T$, where l is some positive integer. The minimum frequency seperation where the signal is still ortogonal is for $l = 1$ and this is called MSK.

Every symbol time the phase of the signal changes $\pm 90°$. The constellation of MSK can be seen in Figure 2.4, where every symbol time the signal is in one of the four dot positions ($(0 \pm j)$ or $(\pm 1 + 0j)$). From Figure 2.3b and Figure 2.4 it can be seen that the constellations of OQPSK and MSK are very similar and indeed MSK can be seen as a special case of OQPSK. Where in conventional OQPSK the pulse shape $h(t)$ is a rectangular shape, in MSK this is a sine wave. Furthermore, the mapping to symbols is a little different; in OQPSK the bits are mapped to absolute phases, while MSK the bits are mapped to phase changes.

The bit error probabilities for coherent and non-coherent detection of FSK are given in Equation 2.6 and Equation 2.7 respectively. Since MSK is a form of FSK these probabilities are still valid for MSK.

Since MSK is also a form of OQPSK the bit error probability for coherent PSK in Equation 2.14 is also valid. However, this assumes the

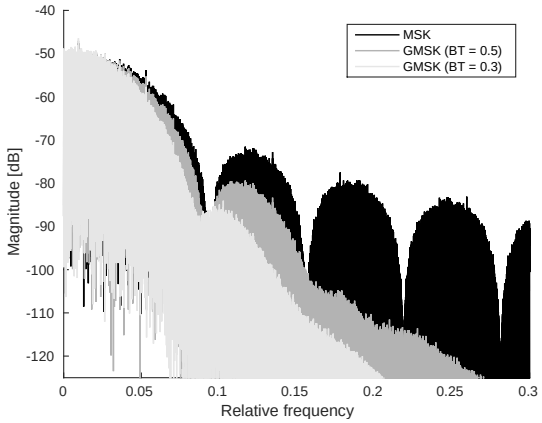


Figure 2.5: Comparison of spectral width for MSK and GMSK modulation

transmitted bits are mapped to absolute phases, while a MSK signal corresponds to phase changes. If a precoder is used before the MSK modulation this equation is valid. If no precoder is used, the bit error probability for differential QPSK can be used as given in Equation 2.15. This equation is valid for both coherent and non-coherent detection.

To summarize, for MSK two detection methods are possible: 'standard' FSK detection, or DQPSK detection. Since DQPSK has better error probabilities we can state that the lower bound for bit error probabilities of coherent and non-coherent MSK detection are that of DQPSK detection.

2.1.3.1 *Gaussian Minimum Shift Keying*

GMSK is a variant of MSK where a Gaussian pulse shaping filter is applied to the signal before modulation. This pulse shaping reduces the spectral width of the modulated signal at the expense of harder detection. Figure 2.5 shows a comparison of the spectral width of MSK and GMSK for different BT values of the Gaussian filter. The BT value of the Gaussian pulse specifies the width of the pulse. A smaller BT value results in a tighter spectral width, but also in more signal leakage in neighboring symbols. Typical values for BT are 0.5 (used in Bluetooth) and 0.3 (used in GSM).

In GMSK the pulse shape function $g(t)$ from Equation 2.4 is a Gaussian pulse given by [1]:

$$g(t) = \frac{1}{2T}\left(Q\left(\gamma\left[t-\frac{T}{2}\right]\right) - Q\left(\gamma\left[t+\frac{T}{2}\right]\right)\right) \tag{2.18}$$

where $\gamma \stackrel{\text{def}}{=} \frac{2\pi B}{\sqrt{\ln(2)}}$, B is bandwidth of Gaussian filter and

$$Q(x) \stackrel{\text{def}}{=} \frac{1}{\sqrt{2\pi}} \int_x^\infty e^{-\frac{u^2}{2}}\, du \tag{2.19}$$

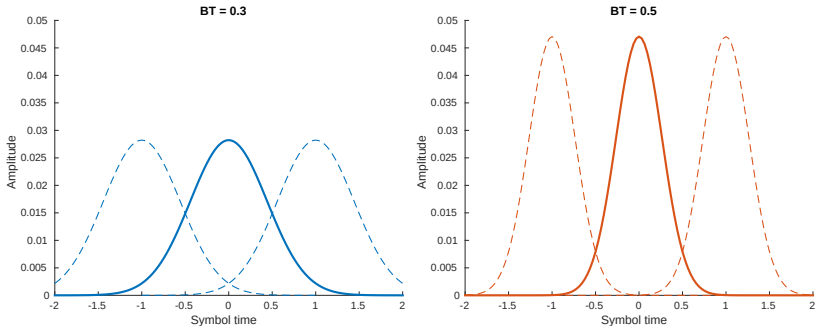


Figure 2.6: Gaussian Pulse Shape for BT = 0.3 (left) and BT = 0.5 (right)

The downside of GMSK is more difficult detection. The Gaussian pulse shape causes symbol energy to leak outside to adjacent symbols. The signal is non-zero outside of the symbol time $0 \leqslant t < T$. This ISI makes detection harder and in general results in higher bit error probabilities. The ISI can be tweaked by changing the BT value of the pulse shape. Figure 2.6 shows the Gaussian pulse shape for two typical BT values. Here it can be seen that there is more ISI with BT = 0.3. Unless otherwise specified the receiver in this thesis will follow the Bluetooth BT value of 0.5.

2.1.3.2 *Matched Filter*

A filter can be applied to the received signal to reduce the noise in the signal. A matched filter is an optimal detection filter in the sense that it maximizes signal-to-noise ratio [20, p181]. If the filter is applied after demodulation, the matched filter (MF) is a Gaussian pulse shape. We get better results if the noise is filtered out before the demodulation. Obviously this requires a different filter that matches the signal before demodulation.

GMSK modulation can be approximated as a sum of linear pulse-amplitude modulation (PAM) signals [14]. In [1] GMSK is approximated with the PAM pulse $h_0(t)$:

$$h_0(t) = C(t-3T)C(t-2T)C(t-T) \quad 0 \leqslant t \leqslant 4T \tag{2.20}$$

$$C(t) = \begin{cases} \sin(\frac{\pi}{2}(1-q(t))) & 0 \leqslant t \leqslant 3T \\ C(-t) & -3T \leqslant t \leqslant 0 \end{cases} \tag{2.21}$$

The modulated signal $s(t)$ then becomes:

$$s(t) = \sqrt{2\mathcal{E}_b} \sum_{n=0}^{N-1} \alpha_{0,n}\, h_0(t-nT) \tag{2.22}$$

$$\alpha_{0,n} = \exp\left(j\frac{\pi}{2}\sum_{k=0}^{n} a_k\right) = \alpha_{0,n-1}\, e^{j\frac{\pi}{2}a_n} \tag{2.23}$$

Where we have a sequence of length N with bits $a_n \in \{-1, 1\}$.

This approximation can be used to create an approximated version of the GMSK signal in the transmitter, but it can also be used in detection. The pulse shape $h_0(t)$ can be used as a matched filter to maximize the SNR before demodulation.

2.2 GNU RADIO

In this section we look at a test environment for radio frequency (RF) transmitters and receivers: GNU Radio.

GNU Radio is an open-source software-defined radio program for Linux and OS X. It enables people to create RF transmitters and receivers by connecting processing functions in what is called in GNU Radio terminology a flow graph.

Multiple manufacturers have RF transmitter and receiver devices that are compatible with GNU Radio. In our setup the USRP N210 by Ettus Research is used, with a SBX RF daughterboard. This can transmit or receive on frequencies between 400 and 4400 MHz.

The research was started with a working GMSK transmitter and receiver in GNU Radio. This flow graph reads a wav audio file, modulates this to GMSK, and on the receiver side it demodulates and plays the audio over the speakers.

Figure 2.7 shows the transmitter and receiver flowgraph. The components will be discussed point for point. The diagram with GNU Radio flow graph is annotated with numbers, which correspond with the points below.

Transmitter

1. The audio file is read with a speed of 44.1k samples per second. While the original wav file contains floating point values, they are converted to bytes.
2. Samples are wrapped in frames, with 128 bytes payload, 64 bit preamble, 1 byte sequence number and 1 tail byte.
3. Bits are converted to $\{-1, 1\}$ symbols, and eight times oversampled.
4. The symbols are passed through a Gaussian pulse shaping filter and frequency modulated (MSK).
5. Optionally, white Gaussian noise can be added to the channel. This is done to test reception at different SNR values.
6. The signal is send to a USRP device, which sends the RF signal to an antenna.

Receiver

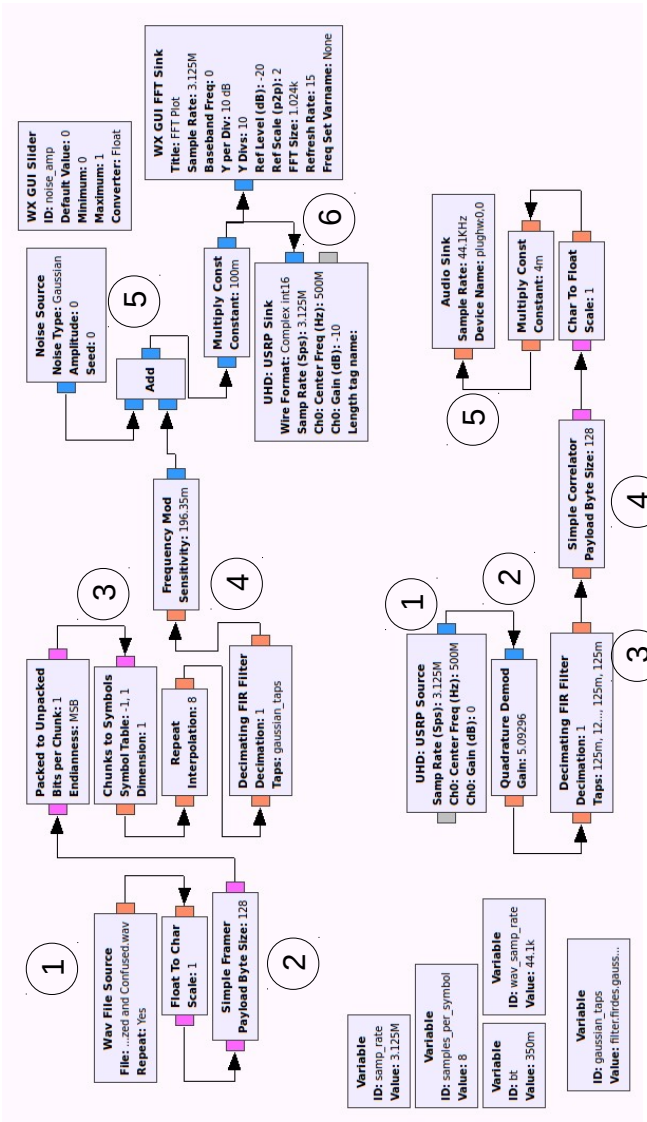


Figure 2.7: GNU Radio flow graph of transmitter and receiver

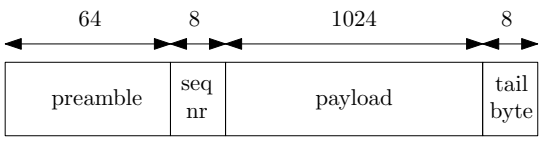


Figure 2.8: *Simple framer* frame layout with length in bits

1. The RF signal is received with another USRP device.
2. The signal is non-coherently demodulated with a quadrature demodulator.
3. The filter averages the signal over the eight subsamples.
4. The packets are detected and the payload is extracted.
5. The data is converted to floating point values and send to the audio output.

We now briefly discuss the most important blocks in the flow graph. A more detailed description of the functions is given in Chapter 3.

SIMPLE FRAMER The samples of the transmitter are wrapped in frames [1] by the GNU Radio block *Simple Framer*. The receiver equivalent of this block is the *Simple Correlator*. The purpose of the frames is mainly symbol synchronization for the receiver.

The frame consists of four parts, which are in order (see Figure 2.8): preamble, sequence number, payload and tail byte. The preamble is a specific sequence of 64 bits. It is a maximum length sequence, which results in a maximum correlation at the receiver side [3]. The sequence number is a byte long and incremented every frame. The payload length is configurable, but in our setup set to 128 bytes. The tail byte is the sequence 01010101 added as padding to the frame. Both the sequence number and tail byte are ignored in the *Simple Correlator*.

SIMPLE CORRELATOR Although the main function of the *Simple Correlator* is to detect frames and extract the payload, it also performs a number of other tasks. The output is bias corrected, synchronized and decimated.

The expected input are floats with soft decision values. Any bias in the signal, for example due to frequency offset of the RF front end, is corrected. The values are then sliced and correlated with the preamble. The optimal sampling moment is determined when a frame is detected. Although not documented in the user interface of GNU Radio (only in the code), it expects a 8 times oversampled signal. The output of the *Simple Correlator* is the payload of the frames, packed in bytes.

1 Throughout this thesis the terms *frame* and *packet* are used interchangeably.

The *Simple Correlator* is discussed in more detail in Section 3.3.4.

QUADRATURE DEMODULATOR The receiver in GNU Radio is non-coherent. The quadrature demodulator is a relatively simple non-coherent differential demodulator. It calculates the phase difference between sequential samples. This GNU Radio block is capable of demodulating FSK, MSK and GMSK.

3

BASELINE IMPLEMENTATION

This chapter discusses the implementation of a baseline GMSK receiver on the Starburst platform. It starts with a detailed discussion of the Starburst platform. A software implementation based on the GNU radio flow graph from the previous chapter is described and the run-time performance is analyzed. From the performance of the software implementation it was determined to implement the receiver in hardware. An accelerator based approach of the same receiver is described with details of the accelerator functions. Finally the BER performance of the receiver implementation is measured and results are presented.

Figure 3.1 shows the hardware setup of the GMSK receiver. On the left is a Xilinx ML-605 development board running the Starburst platform with the receiver. On the right is a USRP transmitter which is used with GNU Radio to transmit a GMSK signal.

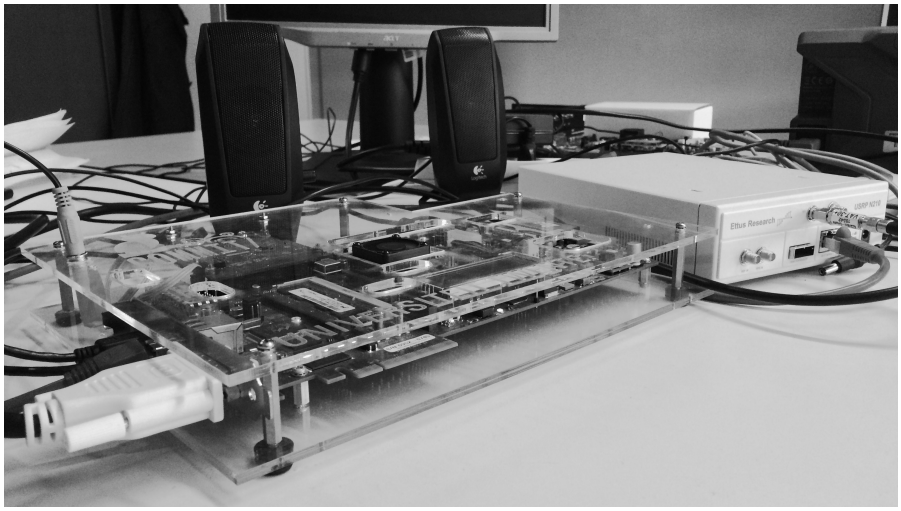


Figure 3.1: Xilinx ML-605 board with GMSK receiver and USRP transmitter

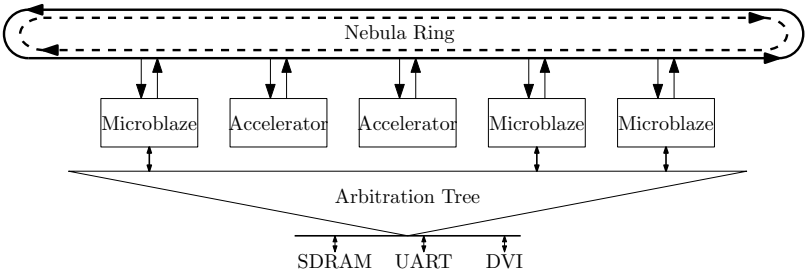


Figure 3.2: Overview of Starburst ring

3.1 STARBURST

Increasingly MPSoCs are used in embedded systems as a way to achieve high performance with low energy costs. However, with the usage of many cores, the communication between cores and other computational elements becomes more important than ever. This is a research topic that is very much active, where different solutions are being proposed. The Starburst platform tries to achieve a number of goals; namely, a communication ring with low hardware costs, even with many cores; real-time guaranteed performance, guaranteed throughput and bounded latencies.

3.1.1 *Overview*

The processors and accelerators on Starburst communicate through an unidirectional ring. The platform is implemented in VHDL for Xilinx Virtex 6 FPGAs, with Microblaze processors. Microblaze is a Xilinx processor architecture specialized for implementation on Xilinx FPGAs [31].

Figure 3.2 shows an overview of the Starburst architecture. A number of Microblaze cores and accelerators are shown. The communication network of Starburst consists of two parts: an arbitration tree and a unidirectional ring. The arbitration tree connects the cores to the shared RAM, DVI and UART. It assigns access to the resources on a first-come-first-served basis. Communication between cores through the SDRAM is not advised because it is significantly slower than the ring and there is no hardware cache coherency between cores. The tree will in general not be used in the scope of this thesis.

The communication ring connects cores and accelerators to each other. The ring is write only and all access is address-based. Nodes can all act as both master and slave and nodes can access all other nodes.

3.1.2 *Ring Communication*

The ring has two communication modes: core-to-core communication and point-to-point streaming. First core-to-core communication is discussed, then point-to-point streaming. The Microblaze cores have a local scratchpad memory. This memory is writable (but not readable) by all nodes on the ring. Communication from core to core is done by writing to the local memory of the destination core. This mode of communication does not have hardware flow control: to ensure write requests do not block on the ring, slaves must always accept the requests. The memory controller is dual port to ensure write actions from both the processor and the ring are always accepted. The configuration of accelerators and network interfaces also falls under core-to-core communication. Configuration values can only be written to, not read from.

3.1.2.1 *Ring slotting*

The arbitration policy on the ring is called *ring slotting*. Each network interface in the ring has a slot, which contains an id, a destination address and a payload. A slot is owned if the id of the slot matches the id of the network interface it currently occupies. Every clock cycle the content of the slot is moved to the next network interface in the ring. A network interface can inject data on the ring with the following two rules: [6]

1. A network interface can always inject data on the ring in its own slot.
2. If a slot is empty and the slot's owner is not reached before the destination of the data, a network interface can transmit data in that slot.

From these two rules bounds can be derived for the throughput on the ring. For a ring with N nodes, a network interface will have it's own slot every N clock cycles. From the first rule follows that the minimum guaranteed throughput for a node on the ring is $1/N$ words per clock cycle. With the second rule we can derive an upper bound for the throughput. If D is the number of hops between the sender and the destination, then in addition to it's own slot there are $N - D$ slots the sender can use to transmit if the slots are empty. This makes the total maximum throughput $B_{max} = B_{min} + \frac{N-D}{N} = \frac{N-D+1}{N}$.

To summarize:

$$B_{min} \leqslant B \leqslant B_{max} \tag{3.1}$$

$$B_{min} = \frac{1}{N} \tag{3.2}$$

$$B_{max} = \frac{N - D + 1}{N} \tag{3.3}$$

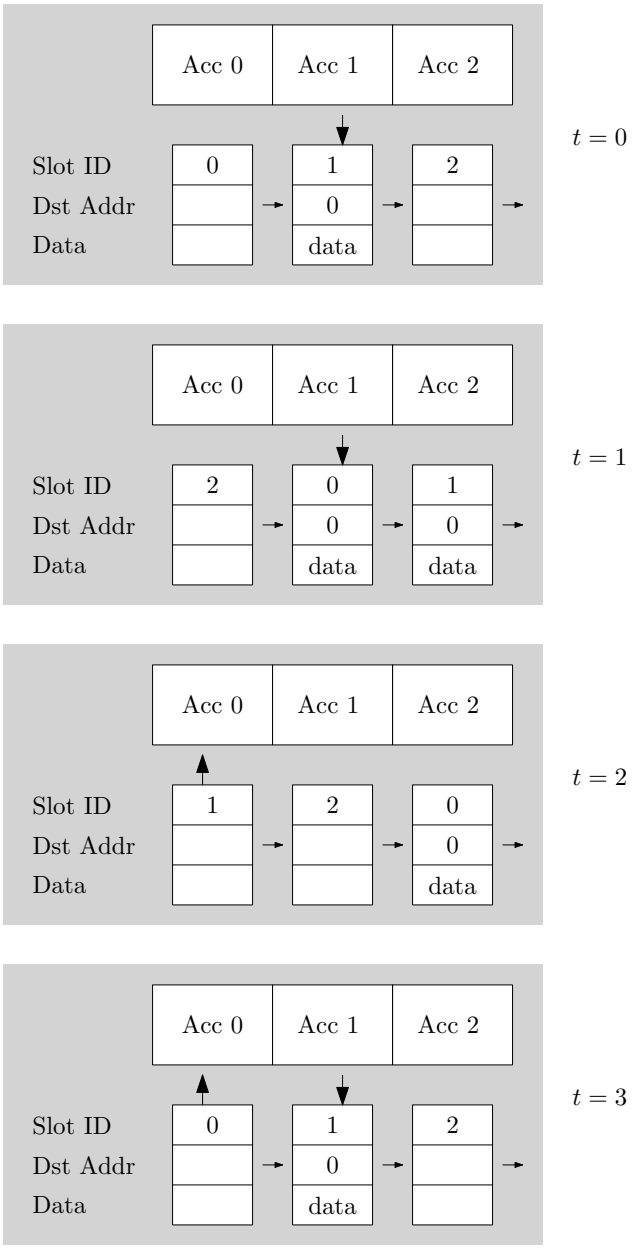


Figure 3.3: Example of *ring slotting* arbitration

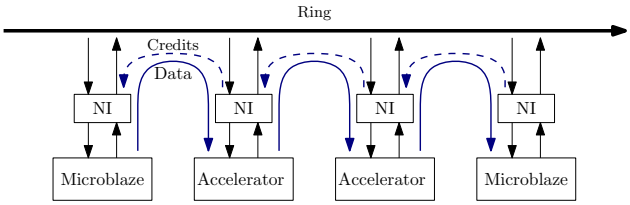


Figure 3.4: Point-to-point streaming on communication Ring

Where B is the throughput of a transmission in words per clock cycle, D is the number of hops to the destination and N is the total number of nodes on the ring.

An example of the *ring slotting* arbitration with a transmitting accelerator is in Figure 3.3. Acc1 transmits data to Acc0. There are three accelerators connected to the ring, making the ring cycle three clock cycles long. Four clock cycles are shown in the figure. At $t = 0$ Acc1 can transmit data due to rule 1. At $t = 1$ it transmits following rule 2, where the slot is empty and the data word can be delivered before Acc1 may need the slot. Acc2 cannot transmit in slot 2 because Acc2 is reached before Acc0. We see the total throughput in this example is $B = 2/3$, which is equal to the maximum throughput from Equation 3.3. This is achievable because slot 0 is not occupied at Acc2.

3.1.2.2 *Flow control*

Figure 3.4 shows a diagram with accelerator communication. It shows a typical flow where data from a Microblaze is send to an accelerator, to another accelerator and finally arrives at a second Microblaze. Unlike in core-to-core communication, there is flow control in point-to-point streaming. This is credit-based, where the credits are sent over a separate ring that runs in the opposite direction of the data ring. In the figure, the data words are sent over the ring with the solid line, while the credits are sent over the dashed line in the opposite direction.

Slaves always accept words sent to them, to ensure the words do not block the ring. To still achieve flow control, the throttling is done by limiting the transmission of words. When a node sends a sample to another node, a credit is spent. The slave sends a credit back to the master when it receives the data and it has space for another sample. This ensures the slave can always store data that the master sends. Because flow control is done on a per sample basis, this allow buffers to stay small in both the accelerators and the network interface.

The buffer in a network interface only has to be as large as the initial number of credits in the sending network interface. [1] To simplify

1 The number of initial credits is the number of credits a network interface (NI) has at startup. This is needed because the NI will not be able send data without credits (in point-to-point streaming mode).

matters the number of initial credits is equal for all network interfaces. The number would ideally be as small as possible to reduce the buffer sizes, but choosing too small may affect throughput of streaming over longer distances.

To determine the effect of the credit flow control we will derive an expression for the worst case throughput. The initial number of tokens of an NI is denoted as C. We are only interested in the throughput limitation due the credit flow control mechanism, so here we will assume that the master can produce and the slave can consume the samples every clock cycle. In [12] the throughput is analyzed and an expression for the minimum, guaranteed throughput due to the flow control is given:

$$B_{min,credit} = \frac{C}{2N + 2D - 1} \tag{3.4}$$

The above expression is derived from a synchronous data flow (SDF) model, but we can also understand this intuitively through a worst case scenario. The master needs to wait $N - 1$ cycles for its own slot to arrive to able to send and then the sample takes D cycles to arrive at the slave. The slave needs one clock cycle to receive the sample and has to wait N cycles to send a credit response back and the response takes D cycles to arrive at the master. The total round-trip time is $2N + 2D - 1$. We get the throughput by dividing the number of tokens by the round-trip time. Note that this is the worst case scenario, so we know the actual throughput [2] will not be lower.

We can now update the throughput bound of Equation 3.1 by redefining Equation 3.2:

$$B_{min} = \min\left(\frac{1}{N}, B_{min,credit}\right) = \min\left(\frac{1}{N}, \frac{C}{2N + 2D - 1}\right) \tag{3.5}$$

From the equation we can also calculate the minimum value for C for which the throughput is not limited by the number of initial credits. The minimum value for $C/(2N + 2D - 1)$ is for $D = N - 1$, the maximum distance to travel of the ring. The expression then is

$$\frac{1}{N} \leqslant \frac{C}{2N + 2D - 1} = \frac{C}{4N - 3} \tag{3.6}$$

$$\implies C \geqslant 4 \tag{3.7}$$

So when the number of initial credits is at least four, the guaranteed throughput is not limited by the credit responses.

3.1.2.3 *Ring shells*

The network interface that handles the communication between ring and accelerator/Microblaze is called *ring shell*. The *ring shell* has small

2 Again, we assume the master and slave can produce and consume respectively every clock cycle. If that is not the case the throughput may be lower.

FIFOs to buffer the samples from and to the ring. Before the streaming of data starts the network interfaces need to be configured. The *ring shells* need to know the forward address and credit return addresses. The forward address is the address where to the data from the accelerator needs to be send on the ring. When a *ring shell* is connected to a Microblaze this is not needed since the Microblaze can set this address herself. The credit return address is the source of the data the *ring shell* receives. It needs the address to be able to send the credits back to the sender. The forward and credit return address imposes a limit on the streaming. A *ring shell* can only accept data from one master, since it only has one credit return address. In a similar argument can a accelerator only stream to one destination because the *ring shell* only has one forward address configured. The latter argument does not hold for Microblazes because they set the forward address themselves.

3.1.2.4 *Accelerator sharing*

Recent work by Gerben Wevers [30] and Oscar Starink [25] looked into accelerator sharing. The idea is that a user may want to use certain accelerators multiple times. This can be done by replicating the accelerator multiple times in the system, but hardware usage can be saved if accelerators are multiplexed. In this approach a gateway is added at the start and end of a chain of accelerators. The gateway configures the forward and return addresses of the accelerators and passes a fixed number of samples through the accelerator chain. When all samples have passed through the accelerators, the gateway saves the state of the accelerators and reconfigures the *ring shells* for a different chain. This way accelerators can be time-shared by multiple streams.

Limitation of current implementation is the large cost of gateways. The hardware savings is limited due to the large cost of a gateway. Also the accelerators need to be modified to support state-saving. The small hardware savings and the additional time needed to modify the accelerators to support state-saving resulted in not using the accelerator sharing feature in this thesis.

3.1.2.5 *Throughput: Examples*

In the previous section the *ring slotting* arbitration was mentioned, in this section the implications for the throughput are discussed with two examples.

Figure 3.5 shows five accelerators in a basic receiver configuration streaming data. The source accelerator produces data that is sent to the filter, demodulator, detection and sink accelerators in that order.

If we use the arbitration rules to determine the throughput of the example in Figure 3.5, we get the following. In the calculation of the

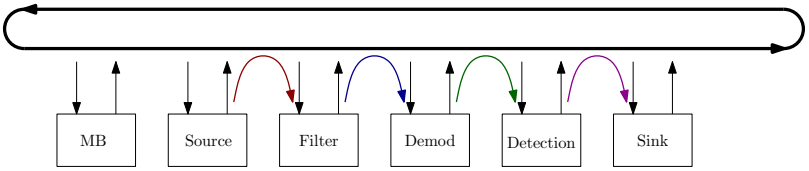


Figure 3.5: Example of accelerator streaming on ring of Starburst

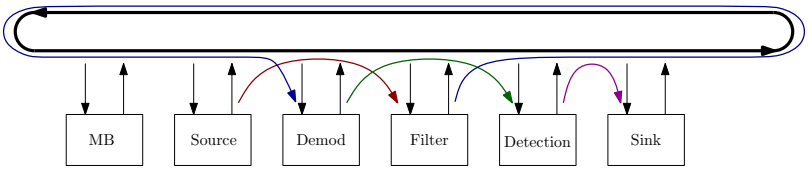


Figure 3.6: Example of accelerator streaming on ring of Starburst with long travel time

throughput we do not take flow control in to account; it is assumed that all accelerators can accept and process data at full speed and that the number of credits of a *ring shell* is sufficient that samples can be send when data is available for transmission. [3]

The throughput of a stream will be B_{max} from Equation 3.3, unless the stream needs to share the link with other streams. In Figure 3.5 the links are not shared and so the throughput can be calculated with Equation 3.3:

$$B = B_{max} = \frac{N - D + 1}{N} = \frac{6 - 1 + 1}{6} = 1 \tag{3.8}$$

In the second example in Figure 3.6 we show a worst case streaming situation where one accelerator streams to it's previous neighbor and so the data needs to travel the complete ring. We will start with the blue flow: filter can always use his own slot and can use demod's slot if this is empty. So the minimum throughput is 1/6, with a maximum of 2/6. Next the red flow: The source NI can always use his own slot, and the slots of filter, detector, sink and MB if empty. Filter's slot may be occupied (the packets from filter contain data to demod), but detection, sink and MB can always be claimed. This makes the throughput between 4/6 and 5/6. The green flow can use his own slot and the empty slots of detection, sink, MB and source. This makes a throughput bound between 1/6 and 5/6. Unfortunately this bound cannot be made tighter, because all empty slots can be used by source before demod. The purple flow has the highest possible throughput of all because of the short distance. The maximum throughput is 6/6 be-

3 This is a reasonable assumption: Most accelerators are designed to work at either clock speed or at the maximum expected data speed and *ring shells* generally have enough credits to cover the round trip time of a credit.

cause it can use all slots (if empty). Only filter's slot can be occupied in that part of the ring, making the minimum throughput 5/6.

$$\frac{4}{6} \leqslant B_{red} \leqslant \frac{5}{6} \tag{3.9}$$

$$\frac{1}{6} \leqslant B_{green} \leqslant \frac{5}{6} \tag{3.10}$$

$$\frac{1}{6} \leqslant B_{blue} \leqslant \frac{2}{6} \tag{3.11}$$

$$\frac{5}{6} \leqslant B_{purple} \leqslant \frac{6}{6} \tag{3.12}$$

In the previous throughput calculations we did not take flow control into account, but in these examples this is not realistic for the stream as a whole. The accelerators are dependent on each other because an accelerator needs the data from the previous one to calculate it's output. It is therefore not possible that one accelerator has a throughput that is twice as high as it's predecessor. Now that we calculated the throughput figures for the stream we can revisit the throughput with flow control. The throughput of the complete stream from source to sink cannot be faster than the slowest part. That makes that the throughput is limited to the blue flow: a throughput of 1/6 is guaranteed, with a maximum of 2/6.

SUMMARY Starburst has a unidirectional ring which supports two communication modes. Core to core communication without flow control and point-to-point streaming with flow control. In core-to-core communication processors write directly in each other's local memory. Accelerator and *ring shell* configuration also work like this. Because there is no flow control slaves must always accept words from the ring. In point-to-point streaming a chain of nodes (processors and/or accelerators) is specified. Every node sends values to the next node in the chain. This chain is fixed at runtime: during streaming nodes can only send data to one destination and receive data from one source.

The throughput of a stream on the ring is guaranteed to be at least $1/N$, if this is not limited by flow control and the number of initial tokens is at least four. The maximum throughput is dependent on the distance D traveled over the ring: $B \leqslant (N - D + 1)/N$.

3.2 SOFTWARE IMPLEMENTATION

The receiver can be implemented as hardware accelerators or as software running on the Microblaze cores of Starburst. A software implementation has more flexibility, because the changing functionality is easier. It is however not clear whether the performance of the Microblaze processors is sufficient to run a real-time GMSK receiver. A basic implementation of the receiver was made in software and the perfor-

mance was measured. The goal is to find out if the Microblaze can run the receiver completely in software and if not, where the performance bottlenecks are. If needed, these bottlenecks could be implemented as hardware accelerators.

Flow graphs in GNU Radio are implemented as a python wrapper script, where the signal processing parts are implemented in C code. To run the receiver on the Microblaze the C code could be reused and modified for the Starburst platform.

The Microblaze processors are not fast with a clock speed of 100MHz and so to improve performance the different components of the receiver all run on different processors. This requires the processors to pass their data to the next processor. This can be done with cFIFOs, which implement a C-HEAP [18] FIFO buffer on top of the core-to-core communication of the ring network [21].

A process works with a local buffer of the data containing a small ($\approx$ 1024) number of samples. When the task has processed the buffer, it sends the result to the next process and starts processing the next buffer at the input. This continues until all buffers are processed.

The receiver implementation starts out reading the baseband signal from file. While the final receiver implementation receives the GMSK signal through the RF front end, for the performance measurement the baseband signal is read from a file. This file is generated in GNU Radio by running the transmitter and writing the GMSK signal to a file. The receiver would then read from this file. The source function that implements this first reads the complete file to memory to speed up subsequent reads. This is done so the source is not a bottleneck of the performance measurement. The sink function at the end of the receiver writes the demodulated data to file. This file can then be compared with the reference implementation in GNU Radio.

3.2.1 *Performance*

The performance of the software implementation was measured by running profiling software along with the receiver processes. This adds some overhead to the running time of the application, but this is good enough for an order of magnitude measurement of both the performance and where the bottlenecks in the application are.

The application consists of five processes. A source process which reads the baseband signal from a file and sends to the next process. The demodulation, filter and frame detection process are the main receiver processes, each passes the result to the next process. The last process is the sink process, which normally writes the result to a file. For the performance measurement this was disabled, because writing to the file system is very slow and this is not used in the final application (where the audio samples are sent to a speaker). The source preloads the file into memory so that this is also not a bottleneck.

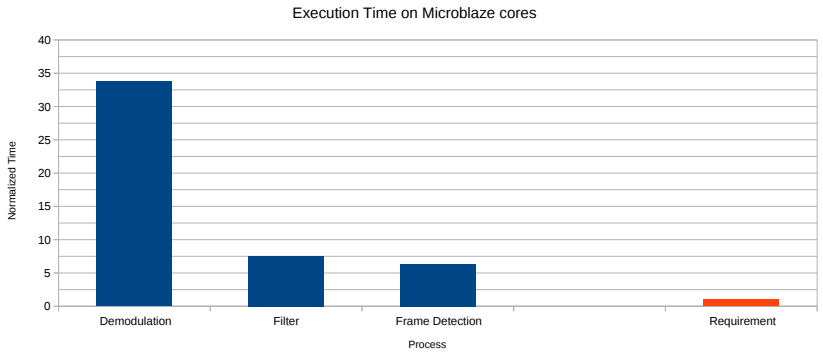


Figure 3.7: Performance measurement for software implementation of receiver.

The results for the three main processes are shown in Figure 3.7. This shows the run time of the processes normalized to the required run time (for real-time audio playback at 44.1kHz). A run time of 1.0 would mean the process is just as fast as required, so any number of 1.0 or below is good enough. In the graph is visible that the demodulation is 33.8 times too slow, the filter operation 7.5 and the frame detector 6.3 times too slow.

The slowest process is the demodulation. This is mainly caused by the arctangent operation. Efforts were made to improve the arctangent calculation by replacing it with a different, lookup table based implementation, but this did not noticeably improve performance. It is also clear from the graph that even if the demodulation can be improved, the other processes are still an order of magnitude too slow. Not only the demodulation but all three processes will need to be implemented as hardware accelerators.

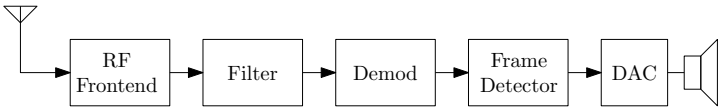


Figure 3.8: Overview of accelerators in baseline receiver architecture

3.3 ACCELERATORS

From the previous section followed that a software implementation of the receiver is not possible. Therefore the receiver components were implemented as hardware accelerators. In Figure 3.8 an overview is shown of the accelerators. Notice that compared to the software implementation, the source and sink are replaced by a RF front end capable of receiving RF signals and a digital-to-analog converter (DAC) speaker output which can play audio through a connected speaker set. Additionally, a pre-demodulation filter is added. This accelerator supports configurable coefficients so that multiple filters can be tested to determine which results in the best reception.

This section will discuss the details of the accelerators. In Section 3.5.2 the sensitivity results will be presented.

3.3.1 *RF Front end*

The RF front end used is a Bitshark FMC-1RX [7] board by Epiq Solutions. The board has a 14-bit analog-to-digital converter (ADC), with a tuning frequency from 300MHz to 4GHz. The ADC chip is a Linear Technology LTC2267-14, which has a SNR of 73.1dB [27]. An accelerator, *bitshark_ctrl*, was already available to do the heavy lifting with regards to interfacing with the board. Configuration of the Bitshark is done through this accelerator. It also does some signal processing including I/Q channel balancing and correction of a DC gap.

3.3.2 *FIR Filter*

A finite impulse response (FIR) filter accelerator developed by Gerben Wevers [30] was used to filter the incoming signal. The filter accelerator has support for a 32-taps filter where the filter coefficients can be configured.

3.3.3 *Quadrature Demodulator*

The demodulation implementation is based on the GNU Radio *quadrature demodulator* block. This is a differential demodulator, which is shown in Figure 3.9.

An incoming sample is multiplied by the conjugated previous sample. The result is a vector where the phase is the phase difference between the two consecutive samples.

$$v(n) = x(n)\, x^*(n-1) = Ae^{j\phi(n)}\, Ae^{-j\phi(n-1)} = A^2 e^{j[\phi(n)-\phi(n-1)]} \tag{3.13}$$

The arctangent outputs the angle of the complex vector $v(n)$:

$$w(n) = \arctan \frac{\mathrm{Im}\{v(n)\}}{\mathrm{Re}\{v(n)\}} = \phi(n) - \phi(n-1) \tag{3.14}$$

The result is a demodulation of the GMSK signal. From Section 2.1 we know the phase rotation is $\pm\pi/2$ radians per symbol. Because of $M = 8$ times oversampling the phase rotation per sample is $\pm\pi/16$. In the implementation the signal is also passed through an averaging filter. By averaging over the last 8 samples of the demodulated signal the phase noise due to white Gaussian noise (WGN) on the channel is reduced. The result is also multiplied by M, so that after decimation (in the packet detector) the phase rotation is $\pm\pi/2$ radians per sample. The netto effect is that the filter calculates the running sum of the last M samples, where M is the oversampling factor:

$$y(n) = M \sum_{k=n-M+1}^{n} \frac{w(k)}{M} = \sum_{k=n-M+1}^{n} w(k) \tag{3.15}$$

NOISE If we assume WGN added to the input, we get a more complicated expression for the demodulation. Assuming the noise is uncorrelated, we rewrite $\eta(n) = \eta_1 = N_1 e^{\theta_1}$ and $\eta^*(n-1) = \eta_2 = N_2 e^{\theta_2}$

$$\begin{aligned}
\bar{v}(n) &= [x(n) + \eta(n)]\ [x^*(n-1) + \eta^*(n-1)] && (3.16)\\
&= [x(n) + \eta_1]\ [x^*(n-1) + \eta_2] && (3.17)\\
&= v(n) + x(n)\eta_2 + x^*(n-1)\eta_1 + \eta_1\eta_2 && (3.18)\\
&= v(n) + AN_2 e^{j(\phi(n)+\theta_2)} + AN_1 e^{j(-\phi(n-1)+\theta_1)} \\
&\qquad + N_1 N_2 e^{j(\theta_1+\theta_2)} && (3.19)
\end{aligned}$$

It is clear from Equation 3.19 that an expression for the output of the arctan is not straightforward anymore. Further analysis of noise at the output of the demodulator is beyond the scope of this thesis; we refer to [19] for the analysis of the phase angle between two vectors with Gaussian noise.

IMPLEMENTATION To reduce the implementation time, standard Xilinx IP blocks were used when possible. These blocks implement basic signal processing functions. For example, the arctangent operation is implemented with a Xilinx CORDIC block. Most new Xilinx IPs support the AMBA AXI4-STREAM protocol [16], which makes connecting different blocks very easy. In Figure 3.10 the blocks of the

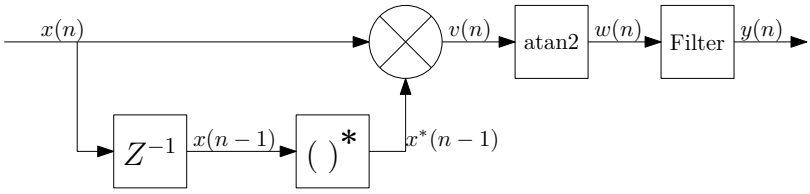


Figure 3.9: Functional diagram of demodulator accelerator

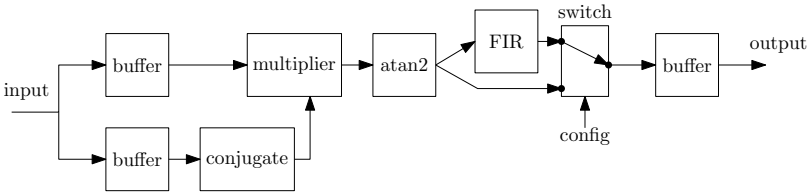


Figure 3.10: Demodulator accelerator implementation

VHDL implementation are shown. The lines between blocks are data connections using AXI4-STREAM (compatible) interfaces. The in- and output connections are not strict AXI4-STREAM, but are compatible with it. The figure is largely the same as the functional diagram in Figure 3.9, so only the differences are discussed. The input and output both have (small) buffers. The output buffer is not strictly necessary if samples can be put on the ring fast enough. The input buffers are implemented so that the conjugate and multiplier inputs do not have to be ready at the same time. This increases the throughput somewhat. The buffers do not need to be large; the minimum possible size for the Xilinx FIFO IP of 16 entries is used.

The switch block in Figure 3.10 is some small logic that selects either the *atan2* or *fir* output. This is dependent on a configuration setting that can be sent to the accelerator. This setting allows the accelerator to circumvent the FIR filter. The default is to use the FIR filter.

The input for the demodulator is complex fixed point numbers. The imaginary and real part are both 16-bit fixed point numbers with 1 integer bits and 15 bits fractional. The imaginary part is in the 16 least significant bits, real in 16 most significant bits. This is inverted from the convention in Xilinx IPs and therefore the order of imaginary and real parts is inverted. The output of the demodulator is 16-bit fixed point number (2-bit integer, 14-bit fractional).

The hardware costs for the demodulator accelerator are shown in Table 3.1. The accelerator originally was designed to handle a sample every clock cycle. However, the sampling frequency for the application is much lower. It was therefore decided to lower the maximum sampling frequency of the FIR filter to save hardware costs. The maximum sample frequency is now 10MHz, which is still well above the sampling frequency of the stream of 3.125MHz.

COMPONENT	SL REGS	LUTS	DSP48S
Input buffer	66	35	0
Conjugate	67	30	0
Multiplication	325	126	3
Atan2	2217	2164	0
Filter (@ 100 MHz)	353	155	5
Filter (@ 10 MHz)	177	104	1
Output buffer	50	35	0
TOTAL [a]	2969	2558	4

Table 3.1: Hardware costs of Demodulator accelerator

a Using FIR filter @ 10 MHz. 1 Slice reg and 29 LUTs used in top level entity.

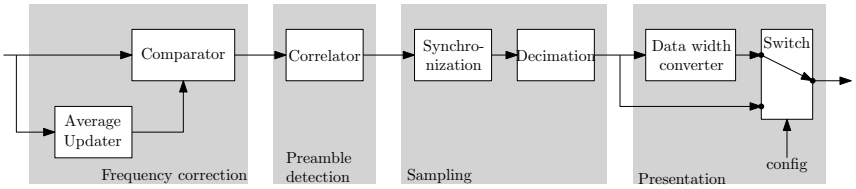


Figure 3.11: Packet detector accelerator implementation

The table lists the cost for the FIR filter at both speeds (100MHz and 10MHz), where the total cost of the accelerator is listed for the 10MHz filter only. Note that this is the speed at which the block can accept new samples, not the clock speed: this is still 100MHz like the rest of the system. In the 100MHz FIR filter the taps are calculated by multiple Xilinx DSP processing slices[32] (DSP48s) in parallel. In the 10MHz FIR filter one DSP48 is reused to calculate the taps, resulting in lower hardware costs but slower throughput.

3.3.4 *Frame Detector*

The frame detector accelerator is an implementation of the GNU Radio Simple Correlator block from Section 2.2. The main purpose of the frame detector is to search the signal for the start of a frame and output the payload part of the frames. There are other functions it also performs. It corrects for any bias in the signal due to frequency deviation of the tuning frequency from the carrier frequency. It also determines the best sampling moment and decimates the signal to remove any oversampling in the signal.

The different components of the packet detector accelerator are shown in Figure 3.11. It consists of three functions: first frequency offset correction and slicing, then preamble detection and finally sam-

pling and decimation. The *average updater* calculates a running average of the signal. This is used in the *comparator* which slices the signal at the average; a signal larger than the average is a one, smaller or equal to the average is a zero. The *correlator* correlates the signal with the preamble by calculating the Hamming distance. *Synchronization* determines from this information when the payload starts and determines the best sampling moment. The *decimator* removes the oversampling. The accelerator can output the packet payload in different formats, depending on configuration. The *data width converter* can pack the bits in 8-bit words, or the *data width converter* can be circumvented and the bitstream is sent to the output.

FREQUENCY OFFSET The *packet detector* corrects the effect of frequency offset on the demodulated signal. We will first discuss the demodulated signal with frequency offset and than look at a method to measure it.

The GMSK modulated signal $s(t)$ is received with carrier frequency f_c. We will not take any WGN on the channel into account.

$$s(t) = \sqrt{\frac{2\mathcal{E}_b}{T}} e^{j2\pi f_c t + j\phi(t)} \tag{3.20}$$

Where $\phi(t)$ is the phase signal from Equation 2.4. The signal is down-converted to baseband with tuning frequency $f_t = f_c - f_o$, where f_o is the error in the tuning frequency. We will assume the frequency offset to be constant in time.

$$r(t) = s(t)e^{-j2\pi(f_c - f_o)t} \tag{3.21}$$

$$= \sqrt{\frac{2\mathcal{E}_b}{T}} e^{j2\pi f_o t + j\phi(t)} \tag{3.22}$$

$$\tag{3.23}$$

If the demodulated signal without frequency offset is $y(n)$, we will now calculate the demodulated signal $y'(n)$ with frequency offset:

$$y'(n) = \frac{M}{\pi}\left(\arg\left[s\left(\frac{nT}{M}\right)\right] - \arg\left[s\left(\frac{(n-1)T}{M}\right)\right]\right) \tag{3.24}$$

$$= \frac{M}{\pi}\left[2\pi f_o nT/M + \phi(n) - (2\pi f_o (n-1)T/M - \phi(n-1))\right] \tag{3.25}$$

$$= \frac{M}{\pi}\left(2\pi f_o T/M + \phi(n) - \phi(n-1)\right) \tag{3.26}$$

$$= y(n) + 2f_o T = y(n) + C \tag{3.27}$$

$$\implies C \stackrel{\text{def}}{=} 2Tf_o = 2M\frac{f_o}{f_{samp}} \tag{3.28}$$

Where we define C as the bias in the demodulated signal due to frequency offset. f_{samp} is the sample frequency, which is M times higher than the symbol frequency $1/T$.

In Equation 3.28 we see that a frequency offset results in a bias in the signal. This directly effects bit errors: if there is a positive bias in the signal there is a larger chance that a transmitted zero is detected as a one. This can be fixed by subtracting a bias estimate from the signal, with the estimate based on the average signal. Let's say the average is calculated over the last P samples. If there are as many zeros as ones transmitted, this should result in an average signal approximately equal to the offset bias.

$$y_{avg}(n) = \sum_{i=0}^{P-1} \frac{y'(n-i)}{P} = \sum_{i=0}^{P-1} \frac{y(n-i)}{P} + C \tag{3.29}$$

The frame header has an equal number of ones and zeros. The bias estimation can therefore best be done by calculating the average over the frame header samples. Obviously this can only be done after the frame header is detected.

The bias correction is implemented by calculating the running average of the last $P = M\ell_S$ samples, which is the length of the frame header (where ℓ_S is number of symbols in header, and M the oversampling factor). This bias estimation is subtracted from the demodulated signal before it is sliced to bits and fed into the *correlator*. When a frame is detected, the *average updater* is stopped for the duration of the payload. This means the *average updater* will still output the last average, but not update it with the new samples. The effect of this is that the payload part of the packet is corrected with the average calculated over the preamble. This results in the best bias correction because the preamble is known and has an equal number of zeros and ones. After the payload is processed and the *packet detector* is in the detection mode again the *average updater* resumes the average calculation.

This approach has two main advantages:

- The average is updated in detection mode, which mean that if the frequency offset changes in time, the bias correction is updated accordingly.
- The average is fixed during the payload part of the packet. This means the bias estimation does not suffer if the number of ones and zeros in the payload part of the packet is not equal.

PREAMBLE DETECTION The preamble detection is done by comparing the incoming signal with the known preamble sequence. This is done by first slicing the incoming phase signal to bits: $\bar{y}(n) = \operatorname{sgn}(y(n))$. The incoming signal is compared bit by bit with the preamble and the Hamming distance of the two sequences is calculated.

$$D(n) = \sum_{i=0}^{\ell_S-1} \bar{y}(Mi+n) \oplus S(i) \tag{3.30}$$

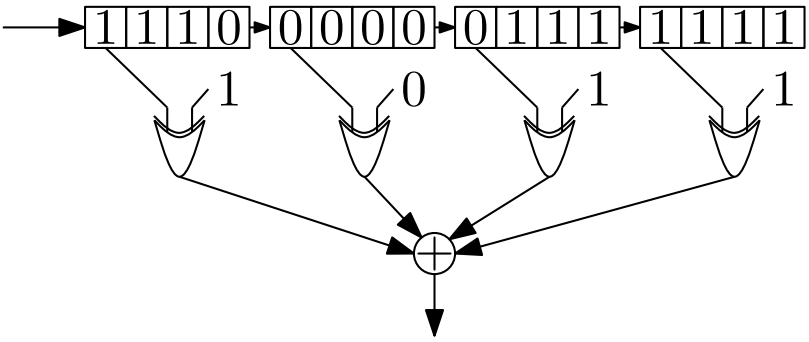


Figure 3.12: Preamble detection by correlation

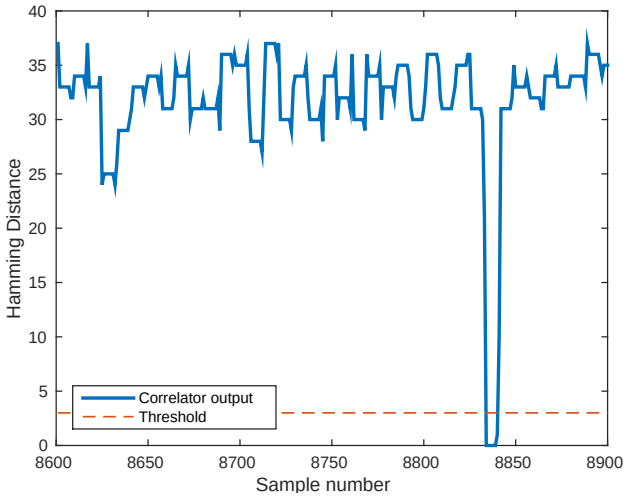


Figure 3.13: Hamming distance output of *preamble detector*

This is shown graphically in Figure 3.12, where the correlation is shown for four times oversampling and a preamble of 4 bits. The entries of the shift register are XOR-ed with the corresponding entries of the preamble. The results of the XORs are summed to get the Hamming distance at the output. A frame detection occurs when the Hamming distance $D(n)$ is lower than a threshold a. In GNU Radio this threshold is set to 3.

SYNCHRONIZATION The synchronization part of the packet detector determines the optimal sampling moment and decimates the signal to remove the oversampling. At the point of frame detection the *preamble detector* outputs a number of consecutive low Hamming distance values. An example of this is shown in Figure 3.13. When the preamble is not aligned, the distance is around 32 (half the bits match the preamble). When the preamble is aligned, the distance drops to 0. Due to the oversampling the Hamming distance is low for a number of consecutive samples. The optimal sampling moment is determined as the middle of the values that drop below the threshold.

COMPONENT	SL REGS	LUTS	LUTRAMS
Avg updater	112	236	128
Comparator	34	20	0
Correlator	33	411	128
Synchronization	33	44	0
Data width converter	544	831	0
Decimator	41	7	0
TOTAL [a]	802	1573	256

Table 3.2: Hardware costs of Packet Detector accelerator

a 5 Slice regs and 24 LUTs used in top level entity.

OUTPUT PRESENTATION The input to the *frame detector* accelerator is 16-bit fixed point numbers (2-bit integer, 14-bit fractional). This format is chosen to correspond with the output of the demodulator. In the original GNU Radio implementation of the frame detector the bitstream of the payload is presented at the output packed in 8-bit integers. This is also practical for our receiver setup since the DAC expects 8-bit integers for audio samples. This is implemented by a *data width converter* component that takes single bit input and outputs 8-bit words. There are however other configurations of the receiver which are not compatible with this output format. For these situations the accelerator can be configured to also output as a bitstream or to output soft information. The bitstream output option is achieved by simply not using the *data width converter*. The soft output is the demodulated signal, which is converted to a single bit in the *comparator*. To get the soft information at the output, the accelerators are modified to pass the soft information along with the original bits. The soft information is output latency matched with the original bit output.

HARDWARE COSTS Table 3.2 shows the hardware cost of the components and the total accelerator. *Data width converter* is the largest component both in terms of LUTs and slice registers though it is not clear why. This is most probably caused by an inefficient implementation in terms of memory access.

During implementation little effort was made to minimize the design in terms of hardware costs. It is therefore very likely that the hardware usage can be reduced by spending some time on the optimization of the accelerator implementations.

3.3.5 *Analog-to-Digital Converter*

Initially the platform could play audio, but using a rather convoluted approach. The audio samples were send over the network to a Linux PC which would play the audio. This method introduced considerable overhead, and more importantly made throughput analysis difficult as a result of the unpredictable behavior of the network stack. A way to directly output audio through some kind of hardware interface was very desirable.

The ADC accelerator needs to play 8-bit samples at a sampling frequency of 44.1kHz. The accelerator generates a pulse-width modulation (PWM) signal at one of the output pins of the ML-605 board. Because the ML-605 board can only drive the pins with very low currents ($<$ 10mA [33]), the output can only be connected to high impedance audio devices like a PC speaker set. The PWM clock is set as close as possible to 44.1kHz, at 100MHz/2268 $\approx$ 44091.7Hz. This results in a PWM resolution of $\lfloor \log_2(2268) \rfloor = 11$ bits.

Unfortunately, the audio signal could not be send with a sample frequency of exactly 44.1kHz, due to limitation in the sample frequency of both the RF transmitter and RF receiver. The closest possible sample frequency is 3.125MHz, which after packet overhead results in an audio sample frequency of $3.125 \cdot 10^6/8/(1024+64+16) \cdot 128 \approx$ 45209Hz. This is slightly higher than the sampling frequency of the ADC. This results in the buffer of the accelerator slowly fulling up over time. To prevent this the accelerator has a 'drop mode'. When this is active the accelerator will always accept new data, but will drop samples if the buffer is full. As long as the incoming and outgoing frequencies are close, this is not noticeable in the audio signal.

To better match the transmitted signal to the sampling frequency of the ADC it is possible to insert 'null' samples at the transmitter between packets. These will be ignored by the *packet detector*, but will slow the speed of the transmitted payload to match the ADC signal. if X is the number of samples to insert, we want:

$$f_{transmitter} = f_{receiver} \tag{3.31}$$

$$\frac{f_{RF}}{M \cdot H_{packet}} = \frac{f_{clock}}{2268} \tag{3.32}$$

$$\frac{3.125 \cdot 10^6}{8 \cdot (1024+64+16+X)/128} = \frac{100 \cdot 10^6}{2268} \tag{3.33}$$

$$\implies X = 30 \tag{3.34}$$

f_{RF} is the sample frequency of the RF signal. This needs to be supported by both the USRP device and the Bitshark. To get the data frequency of the payload we need to divide by the oversample factor M, and the overhead of the packets H_{packet}. The packet overhead is the number of bits in the total packet (1024 bits in payload, 64 in

COMPONENT	SL REGS	LUTS	LUTRAMS	DSP48S
Bitshark	4240	2831	1130	42
Filter	1071	3037	0	33
Mixer	1256	1243	38	0
Demodulator	2969	2558	135	4
Packet detector	802	1573	256	0
DAC	157	137	0	0
Microblaze [a]	2016	2840	208	5
Microblaze w/ peripherals [b]	3045	4449	463	5
Network Interface	321	333	16	0

Table 3.3: Hardware costs of accelerators

a Microblaze core only
b Microblaze including peripherals, as generally instantiated on Starburst

preamble, 16 bits for head and tail bytes, X bits as padding between frames) divided by the number of bytes in the payload (128 bytes).

So if 30 'null' samples were inserted per frame, the number of samples at the receiver would be equal to the sample frequency of the ADC and the accelerator would not need to drop samples.

3.4 HARDWARE COSTS

The hardware costs of the accelerators is summarized in Table 3.3. This has the hardware costs as shown before and the cost of the other accelerators. In the last rows it also shows the costs of Microblaze soft cores and network interfaces to the ring for reference. The first entry for the Microblaze only counts the core alone, while the second entry also counts the peripherals normally added as part of the Microblaze, like the local memory and timer.

3.5 BER MEASUREMENTS

3.5.1 *Measurement setup*

To determine the receiver's performance, BER measurements were done. For this a setup was used that is depicted in Figure 3.14. The GNU Radio transmitter flow graph was used to generate the GMSK signal. The data source is modified from the flow graph in Figure 2.7: instead of an audio file, a periodic signal is generated with the saw-

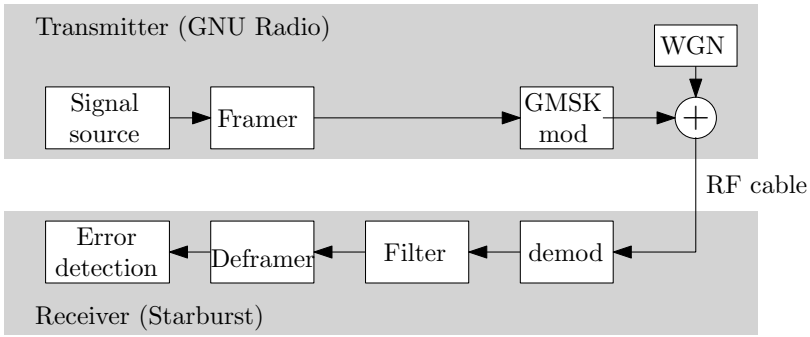


Figure 3.14: BER Measurement setup

tooth signal generator. The signal $x(n)$ is defined by this expression:

$$x(n) = ((n + 96) \bmod 64) + 64 \tag{3.35}$$

Where $x(n)$ is a signed 8-bit integer and the signals for $n = 0, 1, 2, ..., 127$ are in one frame. The signal is chosen such that there is an equal number of zeros and ones transmitted in one frame and such that every frame has the same content. That every frame has the same payload makes error detection easier at lower SNR values, where there is chance of frame drop.

At the receiver side the DAC is not used, but instead a process runs on a Microblaze core that receives the samples from the *packet detector* and determines bit errors. The process will output the number of bits processed and the number of bit errors detected.

The BER measurements are done with an additive white Gaussian noise (AWGN) channel. This noise is added in the GNU Radio script just before transmission with the USRP. The USRP and Bitshark are connected through a coaxial cable instead of antennas to minimize any additional noise. Figure 3.14 has an overview of the BER measurement setup.

The sampling frequency of the USRP and Bitshark is 3.125 MHz. The carrier frequency is set to 434 MHz.

The bit error rate is measured without a filter, and with either a low-pass filter or the matched filter. The low-pass filter is generated with the Matlab *fir1* command, which generates a Hamming window based filter. The normalized cut-off frequency is set to 0.2. This frequency is experimentally determined to achieve the best reception. The matched filter is defined in Equation 2.20.

The measurement results are compared with the theoretical BER performance of DQPSK, which is equivalent to (non)coherent detection of MSK. There are a number of effects that result in worse than optimal BER performance for the receiver. ISI introduced by the Gaussian pulse shape filter is not removed in the baseline receiver and will increase bit errors. Furthermore, there are any number of sources that introduce noise in the signal. This includes noise and errors from:

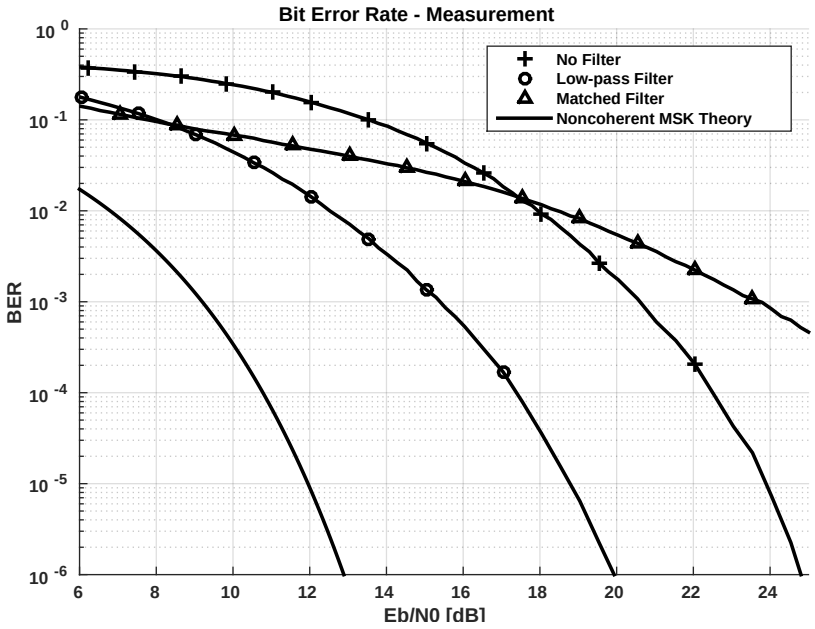


Figure 3.15: BER Measurement without filter, with low-pass filter and with matched filter

- USRP DAC
- USRP RF upconversion
- RF cable
- Bitshark ADC
- Bitshark processing (downconversion, low-pass filtering, gain)
- Processing in accelerators in fixed point representation

3.5.2 *Results*

Figure 3.15 shows BER measurements on the hardware setup without a filter, with a low-pass filter and with the matched filter. For reference the theoretical BER line for DQPSK is shown, which is the theoretical limit for the BER performance. The first thing to notice is that performance is quite a bit worse than theory. The receiver without filter at 10^{-3} BER is 14dB inferior to theory. The receiver with low-pass filter performs significantly better and is 8.6dB worse than theory. This large distance between theory and measurement is caused by the differential detection, which is significantly worse than optimal coherent detection [34].

Also interesting is that the performance with the matched filter, which should be the optimal detection filter, is actually worse than the low-pass and impulse filter. This is caused by extra ISI in the signal introduced by the matched filter. This can be seen in Figure 3.16,

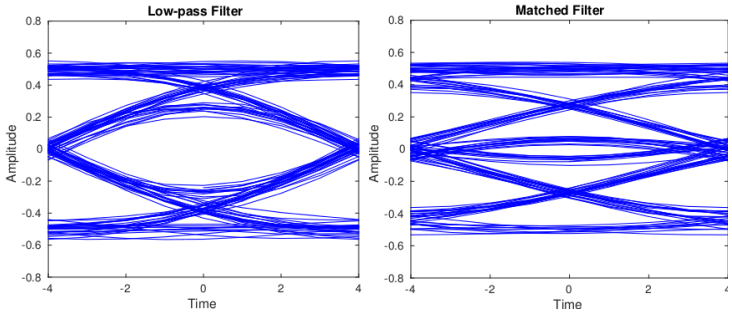


Figure 3.16: Eyediagram of received signal filtered with low-pass filter and matched filter

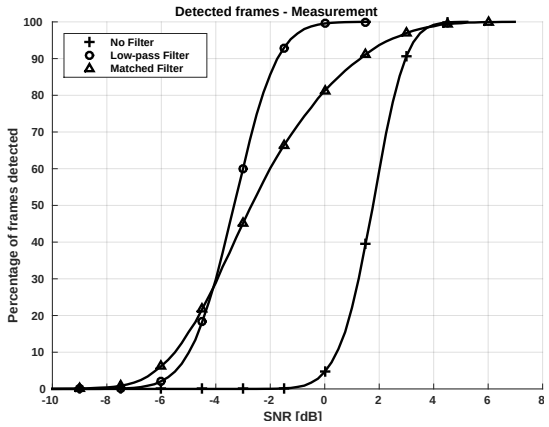


Figure 3.17: Measurement of frame detection without filter, with low-pass filter and with matched filter

where a simulation is shown for the received signal. It is clear from the eyediagrams that the eye is more closed with the matched filter. While not visible in the Figure 3.16, the matched filter does remove more noise than the low-pass filter.

3.5.3 *Frame Detection*

In the previous BER measurements the payload content of each frame is indentical. This is chosen specifically so that a dropped packet (i. e. an undetected frame) does not result in bit errors. This means the bit error rate can be measured independent of the packet detection rate.

The packet detection rate is however an interesting value to measure. This is especially the case when additional channel decoding techniques are added which are located after the *packet detector*. In these cases the BER improves, but the packet detection does not. If one is not careful, the receiver will have good BER values at low SNR, but drops many packets.

The packet detection rate was measured, where the results are in Figure 3.17. The measurement was done by turning on the Bitshark

ADC for a fixed amount of time (45 seconds). The expected number of packets is the sampling frequency divided by the frame length in samples (frame length in bits ℓ_{frame} times the oversampling factor M) multiplied by the capture time:

$$N_{packets} = \frac{f_{samp}}{M\,\ell_{frame}} \cdot t_{capture} = \frac{3.125 \cdot 10^6}{8 \cdot (64 + 8 + 1024 + 8)} \cdot 45 \approx 15922 \tag{3.36}$$

The detected number of frames is measured for different SNR values, where the SNR is set in the same way as in the BER measurements before. The result is shown in Figure 3.17, where the relative packet detection is shown in the vertical axis and the SNR in the horizontal axis. The low-pass filter has best frame detection. At 99% frame detection the low-pass filter is 4.4 dB better than the matched filter and no filter.

There is an analog between this result and the BER measurement in Figure 3.15. Indeed, the number of bit errors directly influences the packet detection. If the number of bit errors in the preamble is higher than the threshold, the frame is not detected.

A way to increase the frame detection rate is to increase the threshold. Currently the threshold is at three, which is quite low. With a threshold the chance that a sequence of 64 bits is wrongfully detected as a frame is $2.4 \cdot 10^{-15}$. Increasing the threshold to 10 will lead to better packet detection rate, but the chance that a frame is detected in error will increase. However with a erroneous detection rate of 10^{-10}, this is still really low.

Increasing the packet detection threshold will lead to a better packet detection rate, but the chance that a frame is wrongfully detection will also increase.

3.6 SUMMARY

In this chapter the basic implementation of a GMSK receiver is presented. The Microblaze processors on the Starburst platform do not have enough performance to run a real-time software implementation, therefore the receiver was implemented with hardware accelerators. The BER was measured with different pre-demodulation filters. The BER performance was not very good, that is why in the next chapter improvements for the receiver are discussed. From the different filters, the low-pass filter has the best BER results, where it is 8.6 dB worse than theory. The frame detection rate is found to be dependent on the choice of filter, where the low-pass filter is 4.4 dB better [4] than the other filters. In Chapter 5 a different architecture will be presented that enables a different filter to be used for frame detection than for the frame payload.

4 at 99% frame detection

4

RECEIVER IMPROVEMENTS

This chapter looks at two improvements to the receiver. Convolutional decoding and equalization are both techniques that can be applied to improve reception. In this chapter the theory and the implementation in the receiver will be discussed. The chapter concludes with BER measurements of a receiver with convolutional decoding, with an equalizer and with both convolutional decoding and equalization.

4.1 CONVOLUTIONAL CODING

Channel coding can be applied to a transmitter to improve detection at the cost of lower channel utilization. Channel codes can be divided in two categories: block codes and convolutional codes. In block coding a block of k bits is mapped to n bits, called the codeword, where $n < k$.

Reed-Solomon codes, the most important block code family, are more resistant to burst errors than convolutional codes. A major drawback of Reed-Solomon codes is the difficulty to do soft decision decoding, which is much easier for convolutional codes with the Viterbi decoder. [13]

Contrary to block codes, convolutional codes have memory. The codeword is not only dependent on the k input bits, but also on the state of the encoder. Convolutional codes are generated with shift registers. The code rate is $R_c = k/n$, where k is the number of input bits and n the number of output bits for each step. Constraint length K is the number of memory slots of the shift register.

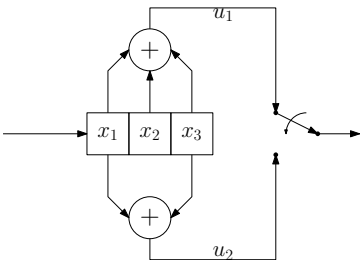


Figure 4.1: Shift register for $[7, 5]$ convolutional code.

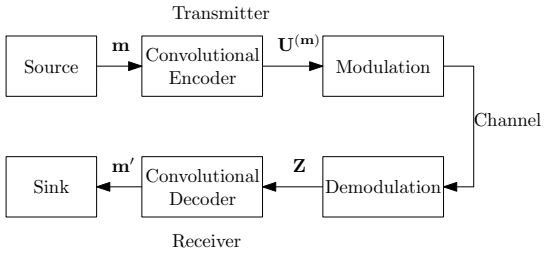


Figure 4.2: Diagram of transmitter and receiver with convolutional encoding

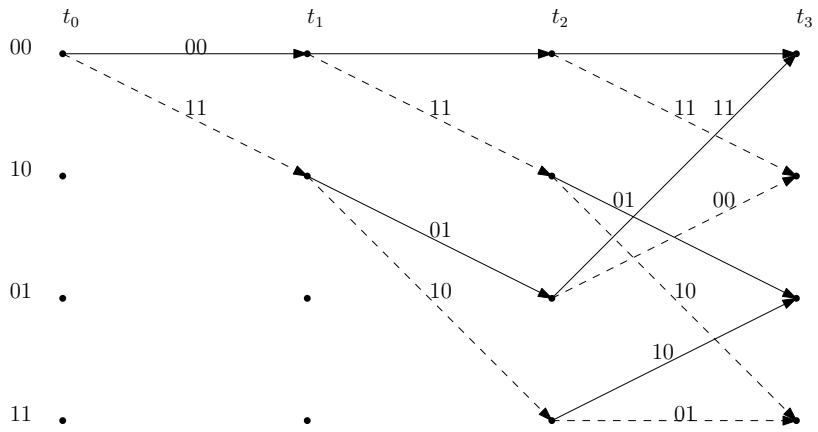


Figure 4.3: Encoder trellis graph of rate 1/2, K = 3 code. The encoder output is written above the edges

Figure 4.1 shows the generation of a rate 1/2 code with $K = 3$ constraint length. Bits are shifted in the register at the left side. Two results are generated: $u_1 = x_1 \oplus x_2 \oplus x_3$ and $u_2 = x_1 \oplus x_3$. The output is the interleaving u_1 and u_2. The generator coefficients are defined as $g_1 = [111]$ and $g_2 = [101]$ which is often written in octal notation as $[7, 5]$.

MAXIMUM LIKELIHOOD The optimal detection method for convolutional coding is maximum-likelihood sequence estimation (MLSE). Say we have an input sequence $\mathbf{m}$ that is encoded to a code sequence $\mathbf{U}^{(\mathbf{m})}$. Figure 4.2 shows a diagram of a simple transmitter and receiver where the sequence is modulated, send over a channel to the receiver and demodulated.

If a sequence $\mathbf{Z}$ is received, the maximum likely sequence $\mathbf{m}'$ is estimated from all possible transmitted sequences $\mathbf{U}^{(\mathbf{n})}$. More formally this can be written as:

$$\mathbf{m}' = \underset{\mathbf{n}}{\operatorname{argmax}}\ P(\mathbf{Z}|\mathbf{U}^{(\mathbf{n})}) \quad \text{over all } \mathbf{n} \tag{4.1}$$

Here $\mathbf{m}'$ is the recovered sequence that is equal to the sequence with the maximum probability $P(\mathbf{Z}|\mathbf{U}^{(\mathbf{n})})$ over all the possible sequences $\mathbf{n}$.

The output of the convolutional code is dependent on both the current input and the last $K - 1$ inputs. This can be modeled in a

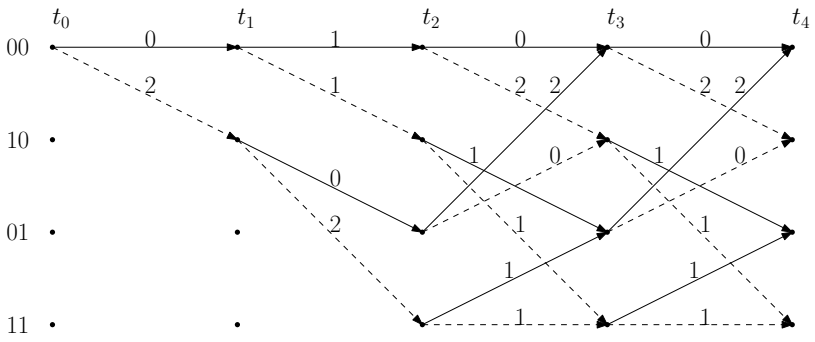


Figure 4.4: Decoder trellis graph of rate 1/2, K = 3 code for received sequence 00010000.

finite state diagram. Trellis graphs show the states and output of a convolutional code in a convenient manner. Figure 4.3 depicts the encoder trellis graph for the [7,5] code from Figure 4.1. The states, or current content of the shift registers, are depicted in the vertical direction. The current input are the edges, which also transition to a new state. By convention, a 0 input is shown as a solid line and a 1 input is a dashed line. The encoder starts in state 00 at t_0 at the top left corner. With a input of 1 the content of the shift register is 100 and the output is (1,1) as shown above the solid line. The 0 on the right is shifted out and the 1 is shifted in, making the new state 10.

4.1.1 *Detection*

We will now look at an example with decoding of a convolutional coded sequence. For this we assume the input sequence $\mathbf{m} = 0000$. If the encoder trellis of Figure 4.3 is followed, the encoded sequence can be found to be $\mathbf{U}^{(\mathbf{m})} = 00\ 00\ 00\ 00$.

Due to noise in the channel, the detector receives the sequence 00010000. The decoder trellis graph is created by noting the distance between the received code word and the possible outputs of the encoder in the edges of the decoder trellis. Figure 4.4 shows the decoder trellis. The decoder starts in state 00. The Hamming distance between the received word (00) and the two possible outputs of the encoder (00 and 11) are written above the corresponding graphs. This is repeated for all the edges. The maximum likely sequence transmitted is the path through the trellis with the minimum distance as noted above the edges of the decoder trellis. From Figure 4.4 it follows that for a received sequence 00010000, the most likely transmitted sequence is 00000000 and the uncoded sequence is $\mathbf{m}' = 0000$.

SOFT INPUT DETECTION The example above is for MLSE detection with hard decisions. This means the input to the detector can only be either a one or a zero. It is possible to have input with extra information to indicate the confidence level: soft decision. For a soft

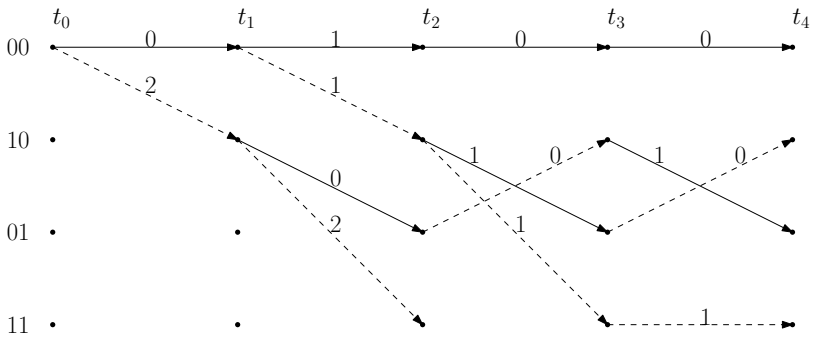


Figure 4.5: Decoder trellis graph of rate 1/2, $K = 3$ code for received sequence 00010000 with Viterbi algorithm.

decision detector with 8 level quantized input the samples have values $-4, -3, -2, -1, 1, 2, 3, 4$, where a -4 is a zero with maximum confidence and 1 a one with minimum confidence. Soft, infinitely quantized decoding has a improvement of 2.2dB over hard decision. Soft decision with 8 level quantization has a improvement of 2.0dB, so doing more than 8 level quantization can only gain an additional 0.2dB [24]. The gain does come at the price of a more complex detector, but this complexity is minimal when using the Viterbi algorithm. The different paths can still be compared with a distance metric, but the metric changes from a Hamming distance to a Euclidean distance. The memory requirements for the soft decoder also increase as it needs to work with 3-bit input compared to 1-bit values for hard decision.

4.1.2 *Viterbi Algorithm*

As was visible in previous discussion about the decoding of the sequence 00010000, even with a short sequence the number of paths involved is quite large. Specifically, for a sequence of nL bits the number of states is 2^L. Clearly this is not feasable for a detector of long sequences. The Viterbi algorithm presented in 1967 by Viterbi [29] greately reduces the number of states needed in a MLSE decoder. The Viterbi algorithm rests on the property of the trellis graphs that the next state and output tuple is only dependent on the current state and input. Any history of previous states or inputs does not matter. This means that during detection when there are two edges merging on a state, one of the paths can be removed. Namely the path with the highest distance, since that path cannot possibly result in a lower distance than the other path. Figure 4.5 shows the decoder trellis from Figure 4.4 again, but with the redundant edges pruned. The Viterbi algorithm ensures that there are only ever as many paths as there are states, since every time two paths converge on a state, only one path survives. This results in a detector which only needs to keep track of K paths, instead of 2^L paths.

TRACEBACK DEPTH In Figure 4.5 it can be seen that for the transition $t_1 \rightarrow t_2$ there are still two options under consideration and so the first bit is not yet recovered. As the sequence moves on, the paths that diverged early will be pruned and the paths will share a common start. The question is however after how many samples this happens. In other words, how many transitions should the Viterbi decoder save for each path before the oldest bit can be recovered? The Viterbi decoder could determine if the first bit is recoverable after every sample, but this is computationally intensive. Therefore the decoder will output after a fixed amount of transitions, which is called the traceback depth. The size of the traceback depth should be minimal to save memory, but maximum to prevent detection errors. [24] states that a traceback depth of 4 or 5 times the constraint length results in a near optimum performance.

FREE DISTANCE The minimum free distance is a property of a code and is a measure of the error correcting ability of the code. As long as the number of errors in a sequence is half of the minimum free distance or less, the MLSE detector will not choose the wrong path in the trellis. As seen in the previous example, the distance of a path is a measure of how likely that particular sequence is transmitted. For a particular sequence, if we know the minimum distance of all paths excluding the transmitted sequence, we know how many errors there can be in the received sequence before the detector will choose the wrong path. For example, if the minimum free distance for a code is five and there are two errors in a sequence, we know the decoder will still follow the correct trellis path and no errors will appear on the output.

Since convolutional codes are linear, it doesn't matter which input sequence we take. We will choose the all-zero code word, since that is easy to work with. Figure 4.3 shows the encoder trellis of the $[7, 5]$ code. We want to know the distance from the all-zero code word for each edge. This is similar to the decoder trellis of Figure 4.4, but now for a sequence of all-zeros. Above each transition the distance to the zero word is written. For the correct path to be rejected, a path needs to divergence from the 00 state and merge back to the 00 state. It turns out that the path with that properties and the minimum distance has a distance of 5. This distance is called the minimum free distance or simply free distance of a convolutional code. To maximize the code correcting ability of convolutional codes, a code with a high free distance should be chosen. The $[7, 5]$ code used in this section has the optimal free distance for constraint length 3. Increasing the constraint length also increases the free distance, but comes at the cost of more states and thus more memory needed for a detector.

4.1.3 *Bit Error Probability*

The goal of convolutional codes is to have better error detection than uncoded signaling, so obviously we expect a lower error probability than modulation techniques without channel coding. Below the bit error probability for coherent BPSK is repeated, along with a lower bound for coherent BPSK detection with soft decision convolutional decoding. [24, p416]

$$P_{b,\text{uncoded}} = Q\left(\sqrt{\frac{2\mathcal{E}_b}{N_0}}\right) \tag{4.2}$$

$$P_{b,\text{conv}} \geqslant Q\left(\sqrt{R_c\, d_{\text{free}} \frac{2\mathcal{E}_b}{N_0}}\right) \tag{4.3}$$

Where R_c is the code rate and d_{free} is the minimum free distance of the code used. This can also be expressed in coding gain, which is the amount of decibels you need to lower the $\mathcal{E}_b/N_0$ to get the same bit error rate for coded signaling as for uncoded signaling. From Equation 4.2 and Equation 4.3 it follows that the coding gain is $10\log_{10}(R_c\, d_{\text{free}})$ dB, or less.

$$\text{coding gain} \leqslant 10\ \log_{10}(R_c\, d_{\text{free}}) \tag{4.4}$$

4.2 EQUALIZATION

intersymbol interference (ISI) in a received signal is generally caused by either distortions in the channel or by the transmitter modulation. ISI due to modulation in transmitter, called controlled ISI, is a result of pulse shaping to limit bandwidth. It is for example present in GMSK, where the Gaussian pulse shaping filter reduces the spectral width of the transmitted signal but also causes signal energy of symbols to leak in the adjacent symbol times.

In our receiver the channel is assumed to only be subject to white Gaussian noise and we therefore do not need an equalizer for channel distortions. However because GMSK is used, controlled ISI is present and an equalizer can help correct this. Because equalization, just like convolutional decoding, is done with maximum-likelihood sequence estimation (MLSE), the Viterbi algorithm can be used.

The Viterbi decoder is configured with the expected channel response for each transmitted sequence. When the equalizer is used for controlled ISI, this channel response is fixed and known before hand. The equalizer only needs to be configured at startup.

4.3 IMPLEMENTATION

Both convolutional decoding and equalization can be performed with the Viterbi algorithm, and so could be implemented in the same ac-

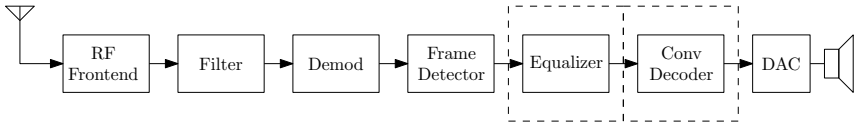


Figure 4.6: Overview of accelerators in receiver architecture, with optional convolutional decoder and equalizer

celerator. Harm te Heneppe [26] implemented a soft input Viterbi decoder accelerator for the Starburst platform. The accelerator can decode convolutional codes or perform channel equalization. Traceback depth and constraint length can be configured at synthesis time. The code rate (where rate 1 means equalization) and trellis coefficients can be configured at run time.

When using a receiver architecture with both an MLSE equalizer and a convolutional decoder, the equalizer can only give hard output decisions to the convolutional decoder. As noted earlier, a Viterbi decoder with hard input values results in a 2dB worse performance than with soft input. It is therefore desirable to have a soft input for the convolutional decoder, but for that we need an equalizer with soft output. In [10] a modification to the Viterbi algorithm is discussed which results in soft output Viterbi decoder. Unfortunately, this was not implemented due to time constraints.

The receiver architecture with equalizer and convolutional decoder now looks like Figure 4.6. Note that the convolutional decoder and equalizer are implemented with the same Viterbi accelerator. Whether the Viterbi accelerator operates as equalizer or convolutional decoder depends on the runtime configuration. The dashed box around the equalizer and convolutional decoder indicate that the inclusion of the accelerator in the receiver is optional. This means that there are four different receiver configurations: the original receiver without equalizer or convolutional decoder, a receiver with only convolutional decoder, a receiver with only equalizer and a receiver with both equalizer and convolutional decoder.

The BER performance of the first option (the baseline receiver) was discussed in the previous chapter. The performance of the other three receiver configurations is presented in the next section.

4.4 BER MEASUREMENTS

The Viterbi decoder accelerator was integrated in the receiver and measurements were performed with the accelerator operating as either convolutional decoder or equalizer. BER measurements were also performed with two Viterbi accelerators, where the first operates as equalizer and the second as convolutional decoder.

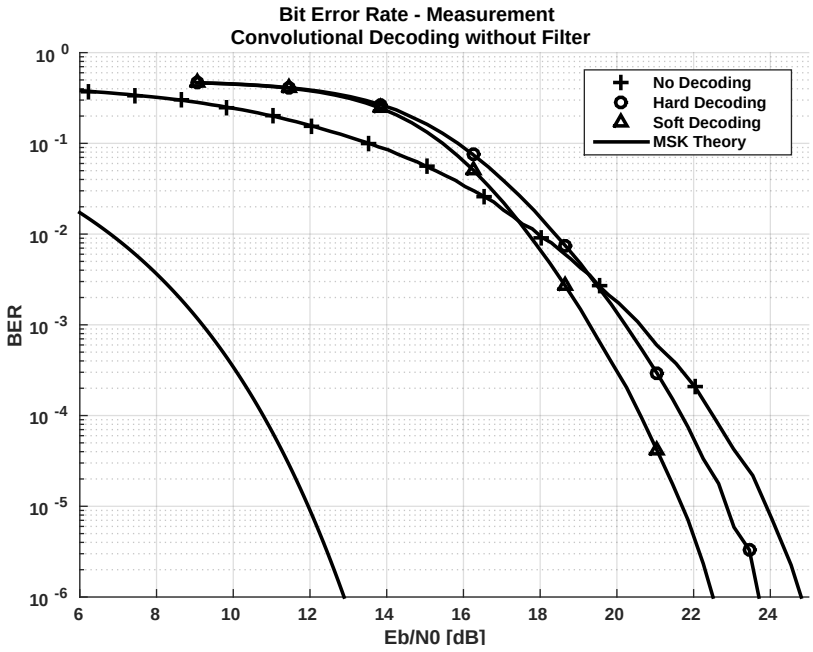


Figure 4.7: BER Measurement with convolutional decoding without filter

The setup for BER measurements is generally the same as in Section 3.5.2. When using the convolutional decoder, a convolutional encoder is added in the transmitter.

4.4.1 *Convolutional Decoding*

The BER was measured with a convolutional code $[7,5]$, which is a rate $^1/_2$ code with constraint length 3. Following Equation 4.4, this has an asymptotic coding gain of 4dB with soft decoding.

Figure 4.7, Figure 4.8 and Figure 4.9 show the measurement results with convolutional decoding for no filter, a low-pass filter and the matched filter respectively. In all figures the theoretical BER line for DQPSK is shown. This is without the 4dB expected coding gain.

The measured coding gain is less than the expected 4dB. The gain without filter and with low-pass filter are about the same, 2.5dB asymptotic with soft decision decoding. This is 1.5dB worse than the theory and there can be multiple effects causing this.

First is that errors are likely to occur in pairs. The demodulator looks at the phase difference between two consecutive samples. If there is a large error in a sample, this will affect two adjacent samples. This means every time there is a (large) error in the input signal this results in two errors in the convolutional decoder. The expression for the theoretical coding gain assumes uncorrelated errors in the decoder and these are not uncorrelated. One modification that can reduce this problem is to interleave the encoded bitstream at the transmitter. In the receiver, the bitstream will need to be de-interleaved

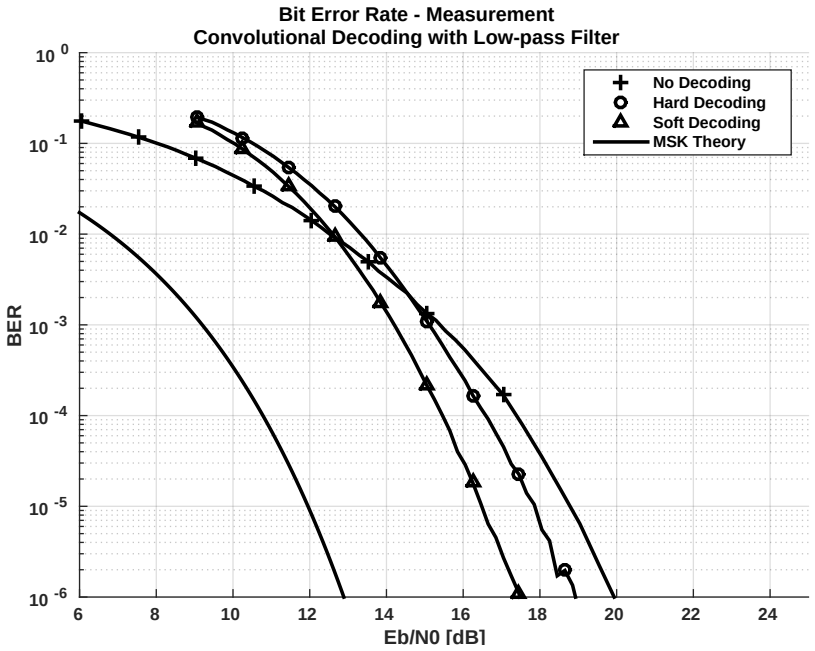


Figure 4.8: BER Measurement with convolutional decoding with low-pass filter

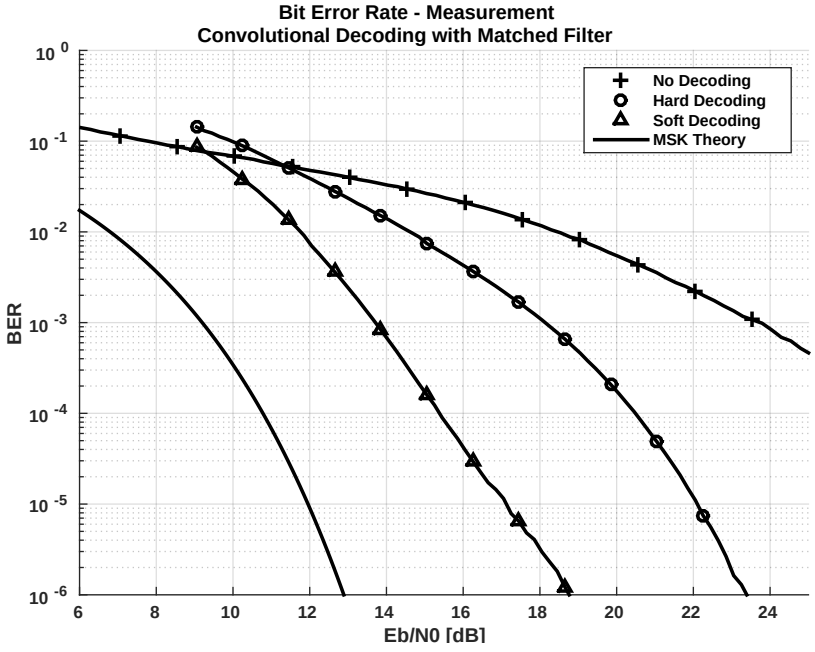


Figure 4.9: BER Measurement with convolutional decoding with matched filter

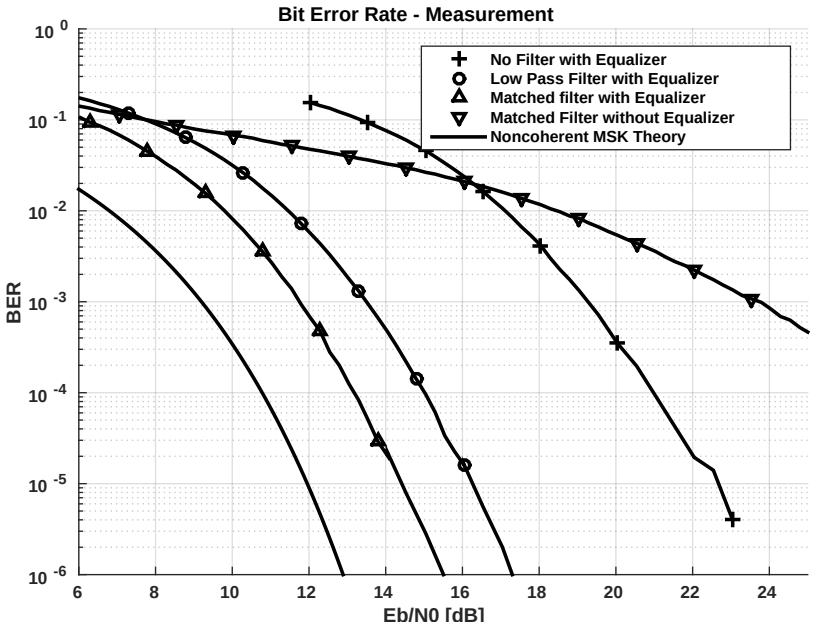


Figure 4.10: BER Measurement with MLSE equalizer without filter, with low-pass filter and with matched filter

before entering the convolutional decoder. This causes error pairs to not be next to each other anymore. Simulations of the receiver indicate that this improves the asymptotic coding gain up to 1dB with soft decoding.

The second possible effect is the output demodulator noise. The Viterbi algorithm (VA) works best with Gaussian noise added to the signal. The differential demodulator does not have Gaussian noise at the output, due the combination of multiplier and arctangent. The measured results are therefore worse than the theoretical limits.

When looking at the receiver with matched filter, adding the convolutional decoder results in a larger coding gain with this receiver configuration than the coding gain with the low-pass filter or without filter. The coding gain is even more than the theoretical asymptotic gain of 4dB. This is however mostly due to the additional ISI in the signal added by the receiver matched filter. The VA partially removes the additional ISI and this results in the large coding gain.

However, not all the ISI is removed from the signal. If you look at the higher $\mathcal{E}_b/N_0$ region, the soft decoding curve is less steep than the theoretical line. The low-pass filter curve does follow the theory line. This results in the fact that for higher $\mathcal{E}_b/N_0$ values ($>$ 16dB) the low-pass filter has better bit error rates than the matched filter.

4.4.2 *Equalizer*

The MLSE equalizer goal is to revert ISI caused by the transmitter pulse shape. In Figure 4.10 it can be seen that the addition of an equalizer

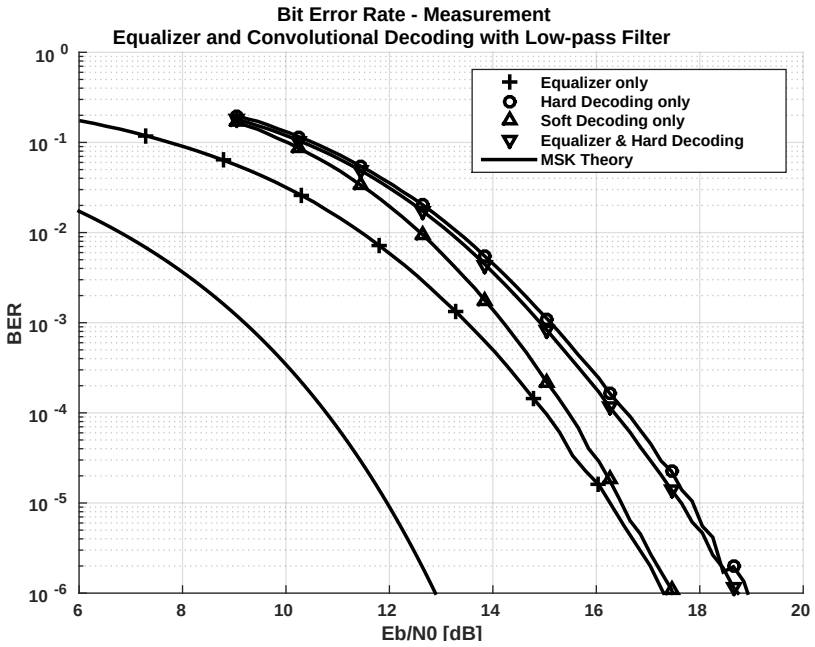


Figure 4.11: BER Measurement with convolutional decoding and low-pass filter, with and without MLSE equalizer

in the receiver improves BER. The improvement is largest with the matched filter. This has two reasons: firstly the equalizer coefficients are tuned to a receiver matched filter, secondly the ISI is worst with the matched filter.

The low-pass filter adds very little ISI to the signal, but also doesn't remove all noise. The matched filter however removes more noise from the signal, at the cost of more ISI. When we look at Figure 3.15 it is clear that without the VA the more noise removal of the matched filter does not weigh against the additional ISI. This is except for very low values of $\mathcal{E}_b/N_0$, where the matched filter is slightly better than the low-pass filter. This is however not a range that the receiver normally will be operating in since the bit errors are already very high.

When the Viterbi accelerator is added however, either as convolutional decoder or equalizer, this removes the additional ISI from the matched filter. In these situations the extra noise removal from the matched filter is greater than the ISI. Since the equalizer is tuned to the specific matched filter response, it removes more ISI than the convolutional decoder. If we compare the BER curve of the matched filter with equalizer with the curve of convolutional decoding, we see that the results with equalizer are about 2dB better than with convolutional decoding.

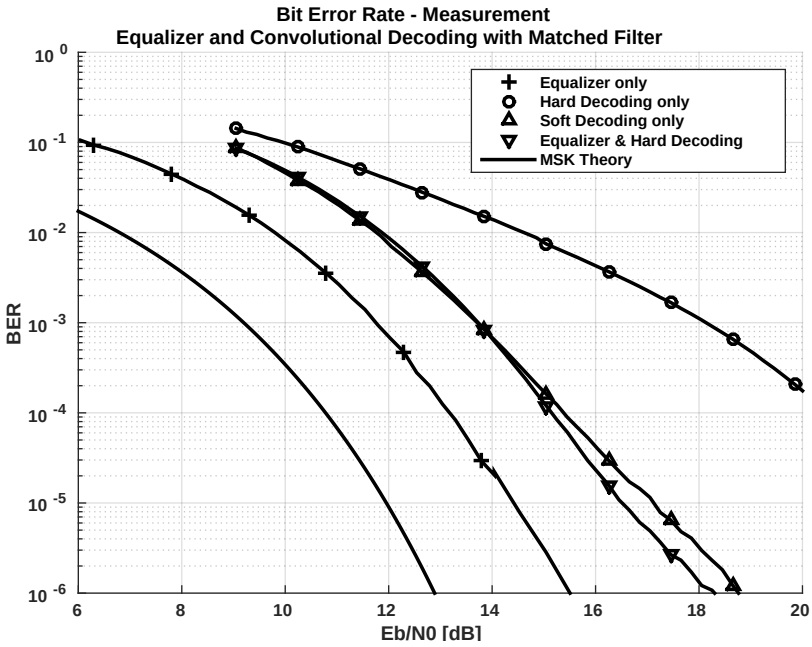


Figure 4.12: BER Measurement with convolutional decoding and matched filter, with and without MLSE equalizer

4.4.3 *Equalizer with convolutional decoding*

Figure 4.11 and Figure 4.12 show BER measurements with both equalizer and convolutional decoding. In this setup, the demodulator output is first passed in the equalizer. The output of the equalizer is then passed in the convolutional decoder. The equalizer can only output hard decisions (bits) and therefore the input of the convolutional decoder are only hard bits.

The plot shows a number of old measurements for reference. The 'hard decoding only' and 'soft decoding only' are the same measurements as Figure 4.8 and Figure 4.9, with the convolutional decoder only and respectively hard or soft input. The 'equalizer only' line shows a measurement with only the equalizer which was shown earlier in Figure 4.10. The new measurement is the fourth line.

With the low-pass filter measurement in Figure 4.11 the equalizer can revert very little ISI and therefore the result is almost the same as with hard convolutional decoding without equalizer. It is clear that in this case it is better to use the convolutional decoder only, in soft decision mode.

With the matched filter the ISI is more severe and the equalizer does lead to an improvement compared to hard convolutional decoding only. The improvement relative to soft decoding is very little however, in high $\mathcal{E}_b/N_0$ range it results in about 0.5 dB improvement. When we compare the equalizer and convolutional decoder combination with the equalizer only, we see that the convolutional decoding has a negative coding gain (a coding loss) of about 2.2dB.

A problem with the Viterbi decoder is that errors occur in bursts. An error in the VA state results in multiple wrong output values. The convolutional decoder, which receives these error values, is not good in correcting burst errors and will likely output errors itself. A solution mentioned in [10] is to interleave the convolutional encoded bits at the transmitter and deinterleave them at the transmitter between the equalizer and convolutional decoder. In this case, if the equalizer causes multiple consecutive errors, these are spread out by the deinterleaver. The convolutional decoder is more likely to still recover the original sequence in this case.

4.4.4 *Summary*

In this section the results of bit error rate measurements of the implemented Viterbi decoder accelerator were presented. The receiver with convolutional decoding has a lower coding gain without a filter and with the low-pass filter than expected. The expected gain is 4dB, while 2.5dB is measured. The matched filter has higher gain because of ISI reduction. The equalizer has similar results where the gain with low-pass filter and without filter is limited, while the ISI reduction for the receiver with the matched filter results in higher gains. The receiver with matched filter is the best performing receiver configuration, being 2dB better than matched filter with convolutional decoding.

The combination of equalizer and convolutional decoder has poor results. The receiver with low-pass filter performs worse than with soft convolutional decoding alone. The receiver with matched filter only performs marginally better than with soft decoding only. With an improvement of less than 0.5dB the hardware cost of an additional Viterbi accelerator is not really worth it.

5

MODE SWITCHING ARCHITECTURE

In Section 4.4.1 we saw that the receiver combination of matched filter with equalizer results in the best bit error rate (BER) performance. However the equalizer is behind the packet detector, and therefore the frame detection does not benefit from the improved bit error rate with equalizer. The frame detection rates measured in Section 3.5.3 are therefore still valid with equalizer, which means that using a matched filter will lead to 4.4dB worse frame detection than a low-pass filter.

So while the matched filter (with equalizer) is the best choice for the BER, it is not the best choice for the frame detection. Additionally, the passband of the matched filter is small, which is a problem when the RF front end has a (large) offset from the carrier frequency.

Frame detection could be improved if the equalizer is placed before the frame detector. This is however not possible in our receiver because the synchronization happens in the frame detector. Also, the signal is not decimated before the frame detector; with eight times oversampling this results in 2^8 more states for the equalizer. [1]

If we cannot use the matched filter with equalizer for frame detection, then the low-pass filter is the best choice for frame detection. The low-pass filter also performs well with frequency offset: it has a wider passband than the matched filter (see Figure 5.1). Unlike the matched filter, the passband of the low-pass filter can also be changed, to accommodate larger frequency offsets if this is necessary. This would be at the cost of less noise removal, but might be desirable if a RF front end is used with a large expected frequency offset.

In short we would like to process the data in two different ways, depending on if it is the packet payload or not. For frame detection we want to do low-pass filtering, while for the payload part we want to switch to matched filtering.

In this chapter we will look at a receiver architecture that combines the good frame detection rates of the low-pass filter with the better BER results of a matched filter with equalizer. This is done by using a different filter for the payload part of a frame than for the frame detection. This receiver architecture is shown in Figure 5.2.

1 There are other ways to implement this, for example by having 8 equalizers parallel. Either way, doing equalization before decimation requires significantly more resources.

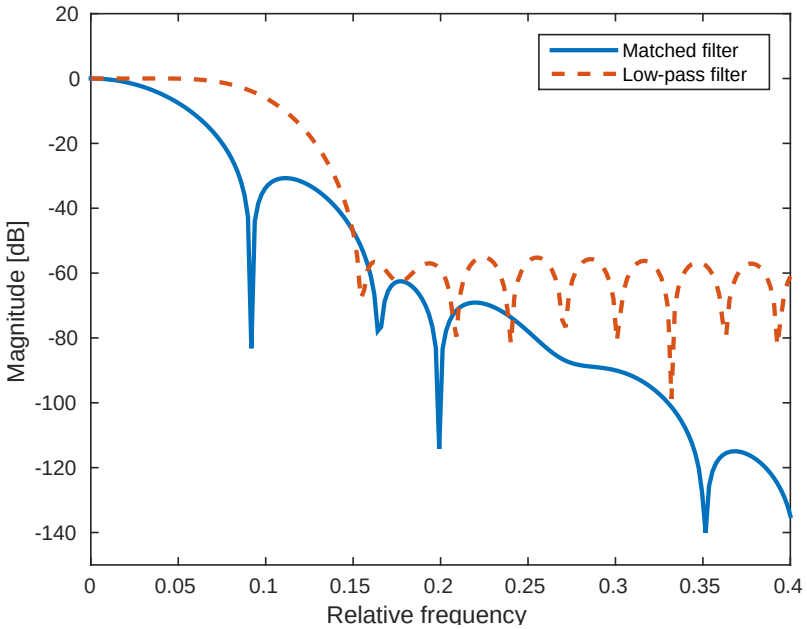


Figure 5.1: Passband of matched filter and low-pass filter with a normalized cutoff frequency of 0.2

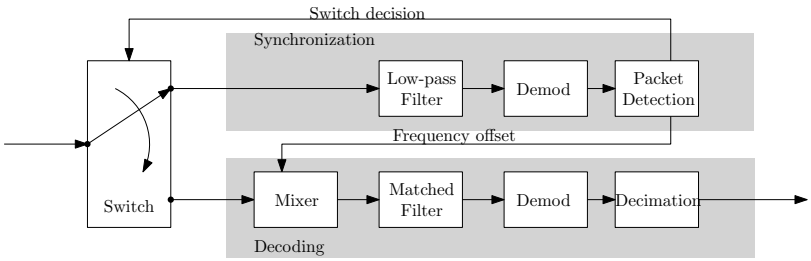


Figure 5.2: Mode switching receiver architecture

5.1 OPERATION

This section will go into the details of the mode switching architecture of Figure 5.2.

The switch starts in detection mode, where samples will first be forwarded to the upper branch for packet detection. The task of the *packet detector* is two-fold: determining the starting point of a frame payload and estimating the frequency offset. The packet detector shown in Figure 5.2 is a modified version of the packet detector in the original receiver. The functions are largely the same, but the output is different. Where the original packet detector outputs the frame payload itself, the modified packet detector needs to send configuration values to the switch so that the payload is sent to the decoding branch.

The packet detector also estimates the frequency offset and needs to configure the mixer so that the samples in the decoding branch are properly frequency corrected. In detection mode there are no samples sent to the decoding branch and so it doesn't make sense for the packet detector to send an update of the frequency estimation every sample. Instead it only sends a frequency estimate right before the switch toggles to the decoding branch.

One of the main difficulties in the mode switching architecture is the pipelining in the detection branch and the problems this presents for the synchronization between *packet detector* and *switch*. The synchronization and possible solutions are discussed in the next section, first we talk about pipelining.

Without pipelining, the switch sends a sample to the synchronization branch and waits for a response from the *packet detector*. If the response is positive (a frame start) the switch will now start outputting to the decoding branch. The critical factor in this is whether the throughput requirements are achievable in this manner. Because the *switch* waits for a response from the *packet detector* before sending the next sample, the critical path is the cycle *switch – packet detector – switch*. The throughput requirement is a sampling frequency of 3.125MHz, or in terms of clock cycles: 100MHz/3.125MHz = 1 sample per 32 clock cycles. Initial estimations indicate that the round-trip time of a packet from the switch to the *packet detector* and back to the *switch* is an order of magnitude larger than 32 clock cycles. It is therefore clear some kind of pipelining is needed to achieve the required throughput requirements. The accelerators already support pipelining, so the *switch* could just send multiple samples before waiting for a response. Some kind of buffering is needed however to store values of which no response is received yet.

5.2 IMPLEMENTATION

The implementation of the mode switching architecture has a number of challenges. The synchronization between the *packet detector* and *switch* is very important. For this two approaches were proposed.

DESIGN 1 In the first design the *packet detector* sends a response for every sample it receives. Due to the pipelining the *switch* doesn't wait for a response from the *packet detector* before sending more sampling to the synchronization path. The samples for which no response have been received are buffered in the *switch*. When the *switch* receives a response for a sample, the action is determined by the value of the response. If it is a negative response the corresponding sample is deleted from the buffer. If it is a positive response (i. e. a frame starts), the buffer read pointer is set to the corresponding sample and the *switch* starts outputting to the decoding branch, starting with this sample.

The length of a packet is known to the *packet detector*, so this is sent along with the response. This way the *switch* knows after how many samples it should switch back to the detection branch. The packet length parameter could also be hard coded in the *switch*, but it was decided to leave information about the packet specifics out of the *switch* to make it more flexible, so it could be used in other use cases.

The main downside of the first design is that for every received sample a response is sent, while at a minimum 512 samples (the length of the preamble) are sent to the *packet detector* before a positive response is sent. This is not a very efficient usage of the ring bandwidth. This in combination with the fact that the *packet detector* responses have to travel a large part of the ring, this could also limit throughput (refer to Section 3.1.2.5 for an example of how long travel distances on the ring affect throughput).

DESIGN 2 The second design tries to improve the throughput by reducing the network traffic from *packet detector* to the *switch*. The packet detector does not send a response for every sample, but only when a frame starts. The detector does this by sending the number of the sample to the *switch*, along with the frame length. The *switch* starts outputting samples to the decoding branch from that sample number onward until the frame length is reached. There are two main challenges in this design. The first and most obvious is synchronizing the internal sample counters in *packet detector* and *switch*. The second problem is buffering and throttling the samples sent to *packet detector*.

At start-up the synchronization is trivial: both counters are initiated at zero and the switch starts outputting to the detection path first. The first sample to be processed by the switch and packet detector are the same sample and both counted as sample 0. It is at the moment when

a frame is detected and the *switch* toggles state that the problem with counter synchronization starts. First, the *switch* outputs the payload (N number of samples) to the decoding branch, but not to detection branch. This mismatch in numbering can be fixed by the *packet detector* increasing the sample counter by N samples after frame detection. Second, due to pipelining, there is a number of samples that is sent to the *packet detector* that belong to payload. In other words, at the moment that a frame is detected, there already are a number of samples belonging to the payload in the detection pipeline. These samples are counted by *packet detector*, but not by *switch*. To solve this, *switch* needs to estimate the number of samples in the pipeline and adjust the sample counter with that amount.

The second problem of design 2 is sample buffering and has to do with buffer space in the switch. The moment that *packet detector* detects a frame, the *switch* probably has sent the first samples of the payload to the synchronization branch. This is because the pipeline depth in the synchronization branch is quite deep (in the order of 100 samples). The *switch* therefore needs to buffer the samples to be able to send the payload to the decoding branch. The buffer space in *switch* is limited and old values need to be overwritten when new samples arrive at the input. However, because the *packet detector* does not send negative responses, the *switch* never knows if it is safe to overwrite an old value. If the buffer space is chosen too small, the first sample of the payload will already have been overwritten when *switch* receives the frame start response. This results in *switch* sending corrupted data to the synchronization branch.

The chance of data corruption is unacceptable. The way to solve this is to send negative responses to *switch* and this returns us to design 1. In theory design 2 could still be correct if you can guarantee that the pipeline depth can never be larger than the buffer space. This is difficult to do however, and will most likely result in an overdimension of the buffer. Another problem with this is that if the accelerators in the detection branch change, the buffer size needs to be reevaluated. This approach collides with the idea of the Starburst's easy configuration of accelerator chains and reusable accelerators.

All these arguments result in the observation that the problem with the data integrity in the buffer in the second design is not worth the reduction of network bandwidth. Therefore the first design is implemented.

HARDWARE COSTS Table 5.1 shows the hardware cost of the components and total accelerator. The *switch* accelerator has no subcomponents and so only the total cost is shown. The *packet detector* accelerator is a modified version of the original *packet detector* from Section 3.3.4. The synchronization block is modified and the *data width converter* and *decimator* are not needed anymore.

COMPONENT	SL REGS	LUTS	LUTRAMS	BRAMS
SWITCH	119	503	176	1
Avg updater	112	216	128	
Comparator	34	23	0	
Correlator	16	388	110	
Synchronization	17	20	0	
PACKET DETECTOR [a]	209	681	238	0

Table 5.1: Hardware costs of modified *Packet Detector* and *Switch* accelerator

a 30 Slice regs and 34 LUTs used in top level entity.

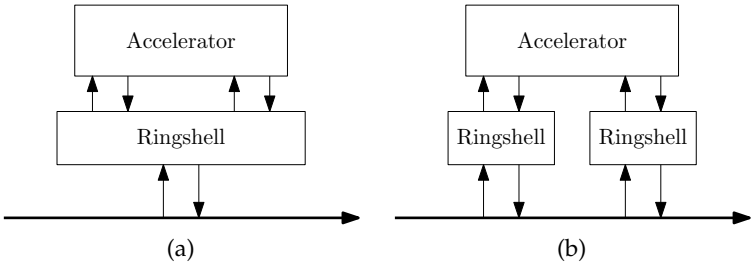


Figure 5.3: Accelerator with 2 I/O, connection to ring

5.2.1 *Ring connection*

Both the *switch* and the modified *packet detector* accelerators in Figure 5.2 have two outputs and *switch* also has two inputs. As discussed in Section 3.1, the *ring shells* only have support for one output or input. This necessitates some modifications in either the *ring shell* or the connection between accelerator and *ring shell*.

Figure 5.3 shows two possible implementations. In the first the *ring shell* is modified to support two in- and outputs from the accelerator. In the second the accelerator is connected to two *ring shells*.

In the first design the *ring shell* needs to share the flow control credits between the two output streams. This means that if the buffer on the first receiving accelerator is full, it also blocks the streaming to the second receiving accelerator. Unless addressed, this could potentially lead to deadlock situations. The second issue is throughput. The throughput of the two output streams is shared because the two outputs share one connection to the ring. Not only the guaranteed throughput of 1 samples per N clock cycles is affected, but it is not trivial to determine which output can use the other slots. A single ring shell with two accelerator outputs is possible, but it requires some significant modifications to the *ring shell*. Special care needs to

be taken to prevent deadlock and starvation of one of the output streams.

The second approach in Figure 5.3b uses two *ring shells*. This has an extra connection to the ring, but requires no modification to the *ring shell*. Indeed, as far as the *ring shells* are concerned there is no difference with it being two separate accelerators. It should be clear that connecting the accelerator to two *ring shells* is superior and this one is therefore used in the implementation.

5.2.2 *Frequency Offset*

The frame detector has one other function besides determining the start of frames. It also corrects for any frequency offset. In the baseline setup of the receiver the frequency offset was determined from a bias in the signal and the correction happened by subtracting the bias directly from the samples. In the mode switching setup this is no longer possible, since the payload data does not pass through the frame detector anymore. The frequency offset also needs to be corrected before the application of the matched filter. This means a bias correction after demodulation is no longer possible. The solution is to add a mixer in the payload path before the filter operation. The mixer multiplies the signal with a complex harmonic such that the signal is shifted in frequency. An implementation of the mixer needs to support a configurable frequency and the *packet detector* needs to send this frequency to the accelerator. The accelerator is implemented with a Xilinx CORDIC IP operating in rotation mode, which needs a normalized frequency as input: i. e. f_o/f_{samp}. The normalized frequency offset can be calculated from Equation 3.28, which is repeated here and rewritten to give the frequency offset from the bias:

$$C = 2M\frac{f_o}{f_{samp}} \tag{5.1}$$

$$\implies f_{o,norm} \stackrel{def}{=} \frac{f_o}{f_{samp}} = \frac{C}{2M} \tag{5.2}$$

Where the mixer performs the operation

$$y(n) = x(n)e^{j2\pi n f_{o,norm}} \tag{5.3}$$

At the start of a frame the frame detector sends the frequency offset to the mixer and then sends a response to the switch. The ring network ensures that the configuration value to the mixer arrives before the switch sends it first samples over the payload branch.

5.3 EVALUATION

We will now look at the throughput performance of the receiver, in an almost complete configuration in the mode switching architecture,

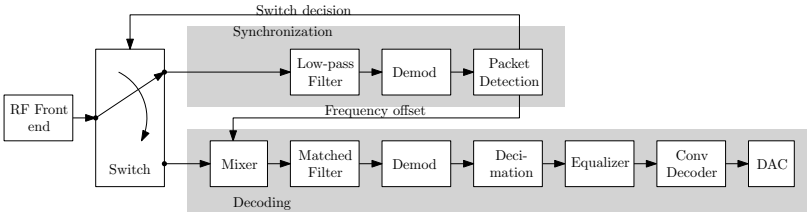


Figure 5.4: Overview of mode switching receiver with equalizer and convolutional decoding

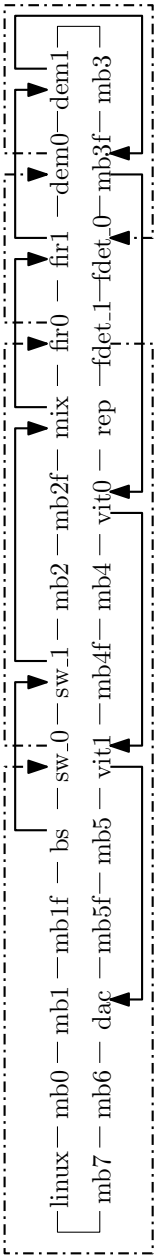


Figure 5.5: Starburst ring with accelerators of mode switching receiver (see Table 5.2 for legend)

Name	Description
linux	Linux Microblaze
mb	Microblaze
mb#f	Microblaze FSL interface
bs	Bitshark accelerator
sw	Switch accelerator (2 interfaces)
mix	Mixer accelerator
fir	Filter accelerator
dem	Demodulation accelerator
fdet	Frame detection accelerator (2 interfaces)
rep	Repetition decoder
vit	Viterbi decoder
dac	digital-to-analog converter (DAC) accelerator

Table 5.2: Abbreviations used in Figure 5.5

with equalizer and convolutional decoding. An overview of the accelerators in the receiver is shown in Figure 5.4. To determine the throughput of samples on the ring the total number of network interfaces is needed. The instantiation of a mode switching receiver with all the necessary accelerators contains:

- 1 Linux Microblaze core
- 8 standard Microblaze cores (5 of which have a FSL buffer)
- 12 accelerators: Bitshark, Switch, Mixer, 2 Filters, 2 Demodulators, Frame detector, 2 Viterbi decoders, Repetition decoder and DAC

The FSL buffer is a first-in-first-out buffer (FIFO) buffer connected to a Microblaze processor that allows it to receive samples from an accelerator. In this context, the only important thing about it is that the FSL buffer requires an extra network interface (NI) on the ring. The switch and packet detector accelerator both have two NIs. This makes the total number of NIs on the ring: $N = 1 + 8 + 5 + 12 + 2 = 28$ NIs.

The ring with all NIs is shown in Figure 5.5, with a legend of the abbreviations used in Table 5.2. The connections that are part of the synchronization branch are shown with a dash-dotted line. In the figure the data flows between nodes on the ring are indicated with arrows. These match the edges between the accelerators in Figure 5.4, with the exception of the configuration of the mixer. This is omitted because it is a configuration value and it is only sent once per frame and therefore does not affect throughput.

The throughput of the receiver is limited by the data flow between two accelerators that needs to travel the most hops on the ring. The longest distance traveled on the ring by any stream is from the packet detector to the switch. This is a number of $D = 16$ hops. The NIs are configured with $C = 4$ initial credits. With these numbers and equations 3.1, 3.3 and 3.5 we can calculate the minimum throughput of the receiver:

$$\min\left(\frac{1}{N}, \frac{C}{2N+2D-1}\right) \leqslant B \leqslant \frac{N-D+1}{N} \tag{5.4}$$

$$\min\left(\frac{1}{27}, \frac{4}{67}\right) = \frac{1}{27} \leqslant B \leqslant \frac{13}{27} \tag{5.5}$$

Due to the choice of 4 initial credits, the minimum throughput is not limited by the sending of credits.

In Section 3.3 we specified the Bitshark sampling frequency as 3.125 MHz. When divided by the clock speed of the system, this translates to 1 sample per 32 clock cycles. From Equation 5.5 follows that the required throughput is guaranteed to be achieved. There can be an additional 5 NIs added to the ring before the required throughput is not guaranteed anymore. In that case a throughput of 1/32 may still be achieved, but this is dependent on which accelerators are active.

IMPROVEMENTS With the current requirement and receiver configuration it is not necessary, but in the future the throughput of the stream may need to be improved. This can be for example because more than five accelerators were added or because the throughput requirement increases. There are still a number of ways that the throughput on the ring can be improved. The easiest to implement are to remove unused accelerators or reorder the accelerators to reduce the longest path. Right now, the critical path is the stream from packet detector to switch. This is due to the large number of hops the data needs to travel. By reordering the NIs on the ring, the distance between these accelerators can be decreased, resulting in a higher guaranteed throughput.

In Figure 5.6 left the throughput of the accelerators is improved compared to that in Figure 5.5 by removing unused accelerators. The throughput is however still limited by the longer distance data needs to travel between accelerators. The two modes (synchronization and decoding) also transmit over the same part of the ring. This does not necessarily limit the actual throughput, since the two modes generally do not transmit at the same time. It is however harder to give real-time guarantees of the throughput in this configuration. This is because the ring bandwidth is shared and the throughput is now dependent on the timing of the accelerators. In the right part of the figure a second possible layout is shown, where the network interfaces of the accelerators that communicate with each other are put directly adjacent. It is now possible to give higher guaranteed throughput

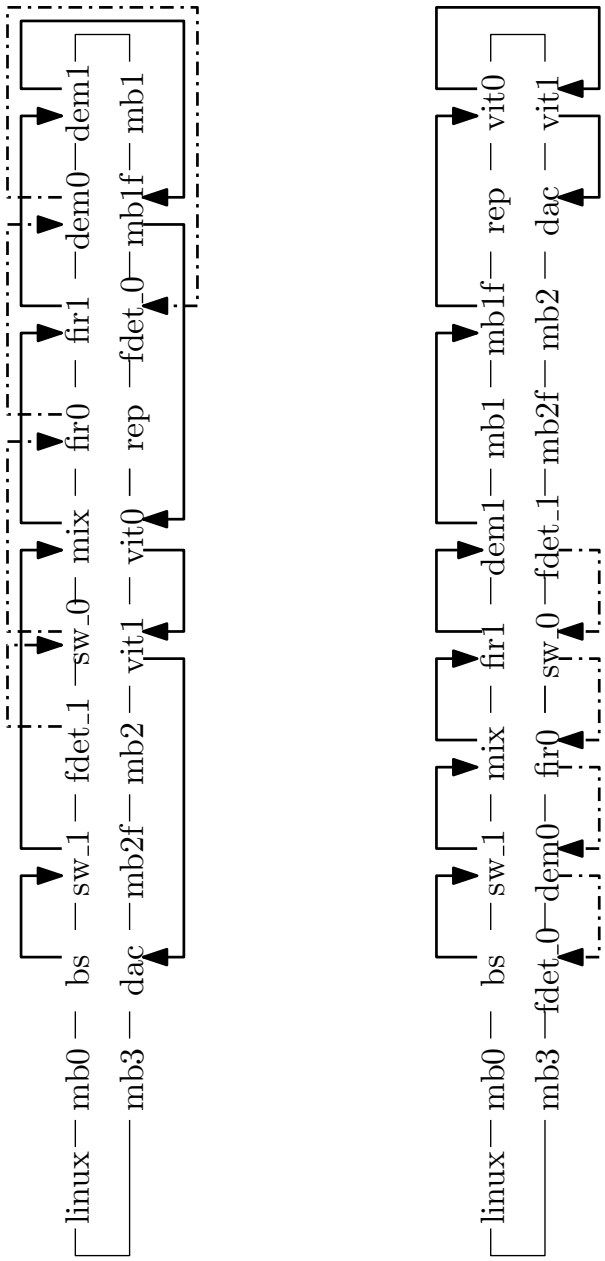


Figure 5.6: Two possible ring orderings to improve accelerator throughput of mode switching receiver (see Table 5.2 for legend)

estimates because the ring connections are not shared. The slowest connection is two hops, with no bandwidth sharing and a total of 21 NIs. The guaranteed throughput is therefore 20 samples per 21 clock cycles, or 3 Gbit/s.

In the future work section in [6] a number of possible future improvements of the ring itself are discussed. Among the suggestions is that by assigning more slots to certain accelerators, throughput can be increased in certain cases. The specific assignment of slots will be application specific however, so this would require some kind of run-time configuration of the slots. Another option that is more promising is 'slot masking', where some NIs are not allowed to use certain slots. In certain situations an NI may use a disproportionate number of slots, leaving few available slots for the next NI. In these situations 'slot masking' can increase throughput for next NI by not allowing the previous NIs to take certain slots.

5.4 SUMMARY

In this chapter a receiver architecture was presented that improves the frame detection rate when used with a matched filter. This leads to a 4.4dB improvement of frame detection and also improves the resilience to frequency offset in the signal.

The mode switching architecture has two branches in the receiver. The first is used for frame detection, where a more frequency offset resistant low-pass filter is used. The second branch is used for the payload part of a frame and performs frequency correction through a mixer and has the matched filter, which has better BER performance in combination with an equalizer.

The throughput of the receiver with all accelerators until now was analyzed. With a minimum throughput of 1 sample per 27 clock cycles, the throughput was determined to be sufficient for the current requirements. A number of improvements were also suggested which could improve the throughput if that is desirable in the future.

6

REPETITION CODING

Repetition coding is a linear block coding scheme and the last reception improvement added to the receiver. While convolutional coding has a higher asymptotic coding gain than repetition coding, it also has worse BER performance in lower SNR region. That is why in Bluetooth Low Energy Long Range a rate $^1/_2$ convolutional code is combined with a rate $^1/_4$ repetition code.

In the first section some theory behind repetition coding will be discussed. Then follows the implementation of a repetition decoder accelerator. BER measurements for different repetition lengths were performed and are also discussed. The chapter finishes with a summary.

6.1 THEORY

Repetition coding is a linear block coding scheme that repeats every input bit R times at the output. R is called the repetition length. The expression for coding gain in Equation 4.4 is also valid for linear block codes. If we fill this in for repetition coding, we get a coding gain of $10\log_{10}(R_c\, d_{min}) = 10\log_{10}(^1/_R \cdot R) = 0$ dB. Repetition coding does not result in a coding gain, because the minimum distance is the same as the code rate. This doesn't mean repetition coding is useless. Because the energy per symbol is increased by repeating it, the SNR per symbol increases. For R times repetition coding the SNR gain is $10\log_{10}(R)$ dB. It is clear that there is a direct trade-off between data rate and reception. By doubling the repetition length the data rate is halved, but the sensitivity should increase by 3dB.

The increase in SNR could also be achieved by raising the transmitter power. This has however a number of downsides. The Bluetooth standard limits the transmitter power (e. g. to reduce interference for other users). Also, the power amplifiers in the transmitter may not be able to handle a larger power level. By transmitting at a lower data rate a higher SNR per bit is achieved with little modification to the transmitter.

When we compare repetition coding and convolutional coding, convolutional coding in general is the superior choice because this has a larger coding gain. The downside of convolutional coding is however

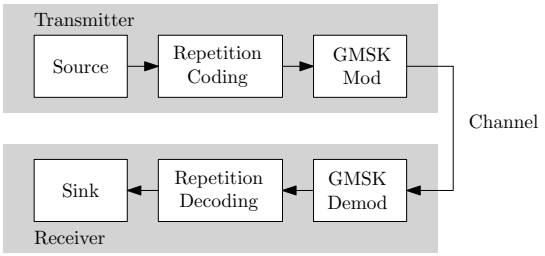


Figure 6.1: Schematic of GMSK transmitter and receiver with repetition coding

the reduced performance in the low SNR range. This is visible in Figure 4.7 and Figure 4.8, where for lower $\mathcal{E}_b/N_0$ values the bit error rate of soft and hard decoding is worse than without convolutional coding. Also, the coding gain in convolutional codes only increases slightly with decreasing code rates. So a smaller code rate will only help with SNR gain, not coding gain. That is why in Bluetooth Low Energy Long Range a rate $^1/_2$ convolutional code is combined with a rate $^1/_4$ repetition code. The repetition coding is the inner coding scheme here, because of the better performance at low SNR. Also, the convolutional decoder causes bursty errors, which would reduce the gain of the repetition coding if the convolutional decoding is done first.

REPETITION PATTERN In [20] repetition coding is defined such that bits are mapped to either the all-one codeword or the all-zero codeword. In this thesis a more general definition of repetition coding is used, where an input bit one is mapped to a pattern G and an input of zero is mapped to the bit inverse of G. This pattern can be the all-one codeword, like the more narrow definition of repetition coding in [20]. The pattern length R is the inverse of the code rate $R_c = 1/R$, and because the zero input bit maps to the bit inverse of the pattern, the Hamming distance between the codewords is also R. In the Bluetooth Low Energy Long Range proposal the repetition pattern is [1100], which should reduce the power of the peaks in the spectrum of the resulting modulated signal. The repetition length in the standard is four, which has a 6 dB theoretical SNR gain.

The difference between repetition coding with the all-one codeword as pattern and increasing the oversampling factor (from Figure 2.7) is that the transmission pulse shape is adjusted to the oversampling factor. Repetition coding does not change the pulse shape, which generally reduces ISI.

DETECTION Figure 6.1 shows a schematic view of a GMSK transmitter and receiver with repetition coding applied. The encoder maps bits to codewords with the repetition pattern. The decoder determines the original bit from the received codeword (containing R sam-

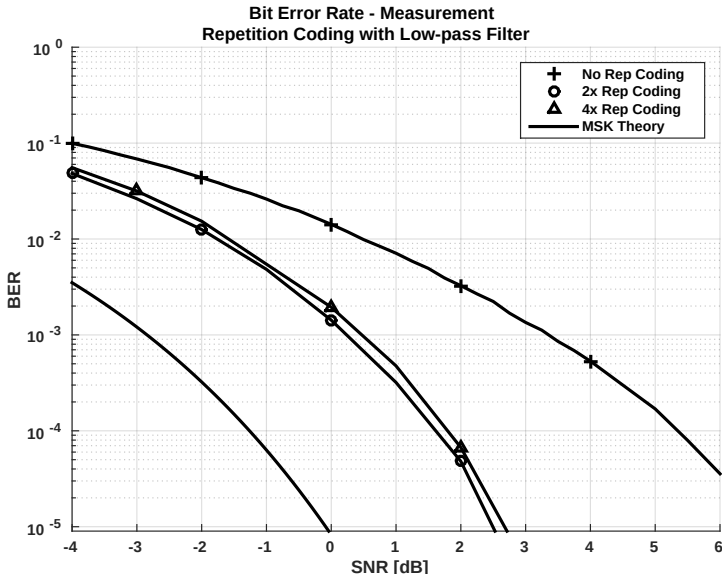


Figure 6.2: BER Measurement of GMSK receiver with repetition decoder for repetition lengths from 1 to 4

ples). This is done by taking the average of the samples, adjusted for the pattern.

$$z(m) = \sum_{i=0}^{R-1} \frac{G_i \, y(mR+i)}{R} \tag{6.1}$$

Where $G_i \in \{-1, 1\}$ is the i^{th} element of the repetition pattern, and $y(n)$ is the demodulated signal. $z(m)$ is the m^{th} output of the repetition decoder, which is a soft decision signal, that can be sliced to bits (with rule $z(m) > 0$). The soft decision information can also be used for a convolutional decoder if the information stream is also convolutional coded.

6.2 IMPLEMENTATION

The repetition decoder is implemented as an accelerator. The accelerator can be configured with a repetition length and repetition pattern, with a maximum repetition length that is chosen at design time. The averaging is implemented as a FIR filter with configurable coefficients, since that is what Equation 6.1 effectively is. The hardware costs of the accelerator are: 325 slice registers, 303 LUTs, 102 LUTRAMs and 1 DSP48 slice.

BER MEASUREMENT A measurement of the bit error rate (BER) was done with the repetition decoder for multiple values of the repetition length. The results of this are shown in Figure 6.2. A low-pass filter is applied before demodulation and the repetition pattern used in the measurements is the 'standard' all-one codeword. With repetition

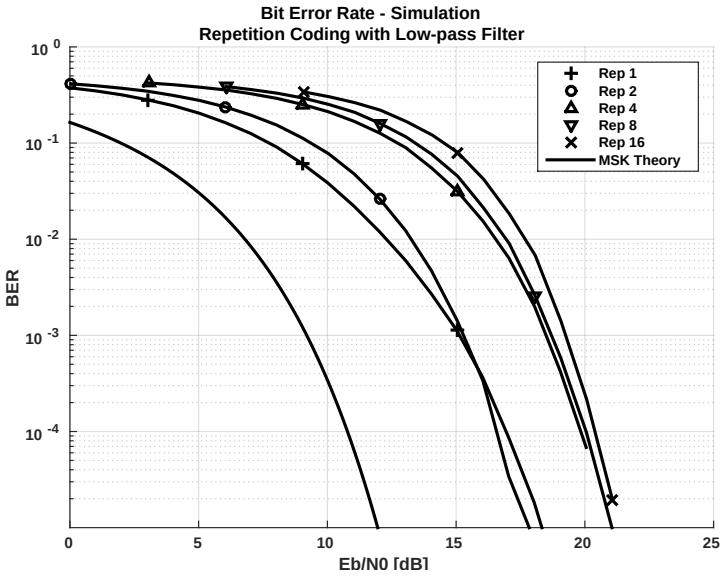


Figure 6.3: BER Simulation of GMSK receiver with repetition decoder for repetition lengths from 1 to 16

length 2 the improvement should be 3dB and 2.8 dB at a BER of 10^{-3} is measured. Reducing the code rate to $^1/_4$ should have another 3dB gain, however the BER measurements show no gain at all. There can be a number of reasons that four times repetition coding has worse performance than expected. However, simulations of the receiver in Matlab show the same results, indicating that this is not a measurement error or an error in the hardware setup.

A plot of these simulations is shown in Figure 6.3. This has $\mathcal{E}_b/N_0$ on the horizontal axis, contrary to Figure 6.2 which has SNR. The simulation is done with a pre-demodulation low-pass filter and repetition rates from 1 to 16. The difference between one and two times repetition coding and four and higher is quite clear, because the graph of Figure 6.3 has the $\mathcal{E}_b/N_0$ on the horizontal axis. Two times repetition coding has almost no coding gain, just as expected. Only for higher $\mathcal{E}_b/N_0$ values is it slightly better due to reduced ISI. With repetition length 4 and higher we expect about the same results as with length 2: a coding gain of about zero dB. The BER curve for length four is however 3.2dB[1] worse than expected. With repetition lengths larger than four the coding gain (relative to no repetition coding) is about the same and only decreases a little. For lengths eight and sixteen the gain is -3.6 dB[1] and -4.1 dB[1] respectively.

6.3 SUMMARY

In this chapter repetition coding was introduced as a way to increase the sensitivity of the receiver, with less coding gain than convolu-

1 Difference with no repetition coding at BER 10^{-3}

tional coding but with simpler detection. The repetition decoder is implemented as an accelerator and can be configured at runtime with a repetition rate and pattern. Measurements of the implementation and of a simulation reveal however that the repetition decoder does not have the SNR gains predicted by the theory. For repetition lengths of four and larger, the decoder has a gain that is 3dB worse than expected. A proper analysis of the effects that cause the reduced gain was not done due to time constraints.

7

CONCLUSION

This thesis discussed the implementation and evaluation of a GMSK receiver on the Starburst platform. The tasks of the receiver were implemented as hardware accelerators. In this chapter we present our conclusions and look at possible future work.

7.1 CONCLUSION

The GMSK receiver was implemented with non-coherent differential detection. Different receiver configurations were tested, where the combination equalizer and matched filter result in the best sensitivity [1]. The channel decoders which are part of Bluetooth Low Energy Long Range (BLR) were also implemented and the bit error rate (BER) performance measured. This has a lower performance than expected. The convolutional decoder with matched filter is 2dB worse than the equalizer. The performance of the repetition decoder is also not good, where the $\mathcal{E}_b/N_0$ point at BER 10^{-3} for rate $1/4$ is 3dB inferior to rate 1 and $1/2$.

The receiver BER performance with equalizer is quite good, but that of the BLR receiver is not. If the receiver is to be used for the BLR standard further research is required to improve the sensitivity of both the convolutional decoder and repetition decoder. It may be necessary to implement a coherent receiver. While this requires possibly complex phase synchronization, the potential sensitivity gain is high.

7.1.1 *Platform*

Initially, the GMSK receiver was implemented as software tasks on Microblaze cores. The performance was not good enough however; the run time of the tasks was between 6 and 34 times too slow. This resulted in the decision to implement all tasks as hardware accelerators. The hardware costs of each accelerator is less than a Microblaze core, making the accelerator implementation smaller and much faster than software.

1 Lowest $\mathcal{E}_b/N_0$ at BER of 10^{-3}

Each task is implemented as a different accelerator. This approach makes it easy to use the receiver in different configurations by only enabling certain accelerators. While the main tasks of the receiver are implemented as hardware accelerators, certain tasks can still run on the Microblaze cores as long as the required throughput in that part of the receiver is not high and the task requires few actions per sample. For example, quantizing the samples for the Viterbi decoder is done as a software task.

A guaranteed throughput and a maximum 'best effort' throughput can be calculated for the data transfers on the communication ring. The limiting factor for the throughput of the receiver is a connection from the packet detector to the switch. The calculated throughput for this connection is between 1 and 13 samples per 27 clock cycles. From this we can conclude that the required throughput for audio streaming of 1 sample per 32 clock cycles is guaranteed to be achieved.

The platform template was modified to allow accelerators to have multiple connections to the ring. Originally accelerators had only one input and output to the ring. For the mode switching architecture two accelerators required multiple in- and outputs. The switch accelerator needs two inputs and two outputs, while the packet detector accelerator requires one input and two outputs. A solution was proposed where two ring shells are connected to the accelerator.

This can be expanded to N NIs, where the accelerator can receive data from up to N sources and send to N destinations. The proposed solution requires very little modification to the platform and no modification of the ring shells itself. A limitation is that this solution does not scale up well to many I/O interfaces. Each NI needs a separate connection to the ring, which can potentially decrease throughput for other accelerators. In the current implementation with only two interfaces this is not an issue, but it is not suitable if many ports are needed.

7.1.2 *Receiver*

In Chapter 3 the baseline receiver is presented, which is a non-coherent differential detector with frame detection and a pre-demodulation filter. The filter is either a matched filter, which maximizes the signal-to-noise ratio, or a low-pass filter, which remove less noise but also suffers less from ISI. In the subsequent chapters a number of improvements for the receiver are implemented and their performance measured.

The mode switching architecture improves the frame detection rate when a matched filter is used for decoding. The mode switching architecture splits frame detection and decoding, so that a different filter can be used for frame detection than for decoding. With this receiver

architecture the frame detection improves by 4.4dB [2] and also is more resilient to frequency offset.

The BLR standard adds a rate $^1/_2$ convolutional code and a rate $^1/_4$ repetition code to the GMSK modulation. The accelerators that are implemented for these codes are the Viterbi decoder and repetition decoder respectively.

The BER performance of the Viterbi decoder was measured for both the low-pass filter and matched filter. The difference between these two in performance is not that large. The matched filter has better bit error rate for $\mathcal{E}_b/N_0$ below 16 dB. For higher $\mathcal{E}_b/N_0$ values the low-pass filter performs better.

The repetition decoder was implemented as an accelerator, just like the Viterbi decoder. The BER performance was however about 3dB worse than expected for repetition lengths of four or higher.

The Viterbi accelerator also can be used as MLSE equalizer. The equalizer performs best with the matched filter by reducing the ISI introduced by the filter. The BER performance with matched filter and equalizer is 2dB better than with convolutional decoding.

The combination of equalizer and convolutional decoding concatenated is not better than convolutional decoder alone. When operating with matched filter the BER performance is only slightly better than convolutional decoding only. With low-pass filter the combination is worse than convolutional decoding only.

From the BER measurements of the different receiver configurations, we can conclude that the receiver with matched filter and equalizer has the best BER curve, which is 2.5 dB worse than the theoretical DQPSK BER curve.

If we limit ourselves to a BLR compliant receiver, which has repetition and convolutional coding, the performance is not as good. Repetition decoding has an issue which causes a 3dB loss for rates of $^1/_4$ and smaller. The convolutional decoder performs about 2dB worse than the equalizer in terms of $\mathcal{E}_b/N_0$. The combination of equalizer and convolutional decoder does not appear to lead to significant gains. The performance improvement is less than 0.5dB, which may not be worth the hardware costs. The concatenation of repetition and convolutional coding has not been measured with the hardware setup, and it is not clear how this will turn out. There are two effects at play, where repetition coding has negative coding gain, but there is also less ISI in the signal.

2 4.4dB improvement at 99% detection rate, when the matched filter is used for decoding and low-pass filter for detection

7.2 FUTURE WORK

The BER performance of the receiver with convolutional and repetition coding was not as good as the theory. It will be interesting to look at the cause of this and investigate possible changes to improve the bit error rate. For example interleaving can be applied, although this is not compatible with the BLR standard. A receiver improvement that is compatible with the standard is using coherent detection. The theoretical BER curves indicate a significant possible improvement by using coherent detection. The potential improvement with the convolutional decoder receiver is 8.5 dB. It is however not clear how difficult phase synchronization is, and how much the increase in hardware costs is.

There are a number of research areas that can be investigated to analyze the performance of the Starburst platform with different applications. By increasing the throughput or size of the current receiver, the performance of the communication ring can be investigated in more demanding situations.

By maximizing the throughput of the receiver bottlenecks in the communication ring can be investigated. Possible optimizations can be researched and implemented to determine ways to improve the performance. Some of these possible optimizations are mentioned in Deken's thesis [6]. These include 'slot masking', where the slots of unused nodes are assigned to other nodes. This can increase the guaranteed throughput of a connection while having minimal effect for others. Another possibility is reordering NIs, or reordering slots.

By increasing the application size, the ring performance with a larger number of accelerators can be investigated. We already know that just increasing the number of accelerators does not decrease the throughput if the streaming is linear (e. g. accelerators stream like A $\rightarrow$ B $\rightarrow$ C $\rightarrow$ D). However, we saw in the mode switching architecture that control loops are a potential problem for the throughput of the application, namely with a large ring. This could probably be solved by moving the NI closer to the receiving NI. [3] Will this solution work with other feedback loops as well? And how does the application perform if it has multiple control structures and feedback loops?

Another possible research could look at a scenario where a wireless receiver processes either multiple channels or multiple protocols (e. g. a receiver for both Bluetooth and WiFi). A single RF front end generates baseband data that are processed by multiple receiver chains. This results in many accelerators, which are not all in the same stream. Multiple real-time applications sharing the ring bandwidth

3 A potential problem with this solution is that the ports of the accelerator are physically located further apart on the FPGA, possibly making timing closure more difficult.

could be a problem if high throughput is required. This research could investigate throughput bottlenecks and possible solutions. Is it possible, desirable and/or necessary to use multiple, separate rings? Because the multiple channels may require partly the same processing, accelerators could be shared between the channels. If this is done in different parts of the receiver and the data needs to travel larger distances on the ring, an interesting question is how this will affect throughput.

While wireless communications is a mature research field, new and exciting things are still being discovered. And with this future work section we can say there is much more to discover.

BIBLIOGRAPHY

[1] N. Al-Dhahir and G. Saulnier. A high-performance reduced-complexity gmsk demodulator. In *Signals, Systems and Computers, 1996. Conference Record of the Thirtieth Asilomar Conference on*, volume 1, pages 612–616 vol.1, Nov 1996. doi: 10.1109/ACSSC.1996.601118.

[2] A.A. Ali and I. Al-Kadi. On the use of repetition coding with binary digital modulations on mobile channels. In *Vehicular Technology Conference, 1987. 37th IEEE*, volume 37, pages 59–65, June 1987. doi: 10.1109/VTC.1987.1623524.

[3] Martin Cohn and A. Lempel. On fast m-sequence transforms (corresp.). *Information Theory, IEEE Transactions on*, 23(1):135–137, Jan 1977. ISSN 0018-9448. doi: 10.1109/TIT.1977.1055666.

[4] R. D'Avella, L. Moreno, and E. Turco. Adaptive equalization and viterbi decoding for digital mobile radio systems. In *Global Telecommunications Conference and Exhibition 'Communications Technology for the 1990s and Beyond' (GLOBECOM), 1989. IEEE*, pages 90–94 vol.1, Nov 1989. doi: 10.1109/GLOCOM.1989.63946.

[5] Berend Dekens. Mapping of a dab radio decoder to homogeneous multi-core soc : a case study to evaluate a nlp based mapping flow, March 2011. URL http://essay.utwente.nl/60148/.

[6] Berend H.J. Dekens. *Low-Cost Heterogeneous Embedded Multiprocessor Architecture for Real-Time Stream Processing Applications*. PhD thesis, University of Twente, Oct 2015.

[7] *Bitshark FMC-1RX Rev C User's Manual*. Epiq Solutions, v2.0 edition, 2011.

[8] J. Feldman, I. Abou-Faycal, and M. Frigo. A fast maximum-likelihood decoder for convolutional codes. In *Vehicular Technology Conference, 2002. Proceedings. VTC 2002-Fall. 2002 IEEE 56th*, volume 1, pages 371–375 vol.1, 2002. doi: 10.1109/VETECF.2002.1040367.

[9] D. Göhringer and J. Becker. High performance reconfigurable multi-processor-based computing on fpgas. In *Parallel Distributed Processing, Workshops and Phd Forum (IPDPSW), 2010 IEEE International Symposium on*, pages 1–4, April 2010. doi: 10.1109/IPDPSW.2010.5470800.

[10] J. Hagenauer and P. Hoeher. A viterbi algorithm with soft-decision outputs and its applications. In *Global Telecommunications Conference and Exhibition 'Communications Technology for the 1990s and Beyond' (GLOBECOM), 1989. IEEE*, pages 1680–1686 vol.3, Nov 1989. doi: 10.1109/GLOCOM.1989.64230.

[11] Andreas Hansson, Kees Goossens, Marco Bekooij, and Jos Huisken. Compsoc: A template for composable and predictable multi-processor system on chips. *ACM Trans. Des. Autom. Electron. Syst.*, 14(1):2:1–2:24, jan 2009. ISSN 1084-4309. doi: 10.1145/1455229.1455231. URL http://doi.acm.org/10.1145/1455229.1455231.

[12] Gerald Hoekstra. Hardware accelerator integration in a connectionless network-on-chip. Master's thesis, University of Twente, June 2013.

[13] R. Johannesson and K. Zigangirov. *Fundamentals of Convolutional Coding*. Wiley-IEEE Press, 1999. doi: 10.1109/9780470544693.

[14] G.K. Kaleh. Simple coherent receivers for partial response continuous phase modulation. *Selected Areas in Communications, IEEE Journal on*, 7(9):1427–1436, Dec 1989. ISSN 0733-8716. doi: 10.1109/49.44586.

[15] G. Kuiper, B.H.J. Dekens, S.J. Geuns, P.S. Wilmanns, J.P.H.M. Hausmans, and M.J.G. Bekooij. Compiler for real-time multi-processor systems with shared accelerators. *DATE Conference*, March 2015.

[16] ARM Limited. Amba 4 axi4-stream protocol version 1.0 specification, 2010.

[17] Yuan Lin, Hyunseok Lee, M. Woh, Y. Harel, S. Mahlke, T. Mudge, C. Chakrabarti, and K. Flautner. Soda: A high-performance dsp architecture for software-defined radio. *Micro, IEEE*, 27(1):114–123, Jan 2007. ISSN 0272-1732. doi: 10.1109/MM.2007.22.

[18] André Nieuwland, Jeffrey Kang, OmPrakash Gangwal, Ramanathan Sethuraman, Natalino Busá, Kees Goossens, Rafael Peset Llopis, and Paul Lippens. C-heap: A heterogeneous multi-processor architecture template and scalable and flexible protocol for the design of embedded signal processing systems. *Design Automation for Embedded Systems*, 7(3):233–270, 2002. ISSN 0929-5585. doi: 10.1023/A:1019782306621. URL http://dx.doi.org/10.1023/A%3A1019782306621.

[19] R.F. Pawula, S.O. Rice, and J. Roberts. Distribution of the phase angle between two vectors perturbed by gaussian noise. *Communications, IEEE Transactions on*, 30(8):1828–1841, Aug 1982. ISSN 0090-6778. doi: 10.1109/TCOM.1982.1095662.

[20] John G Proakis. *Digital Communications*. McGraw-Hill, New York, fifth edition, 2008.

[21] Jochem H. Rutgers. *Programming Models for Many-Core Architectures – A Co-design Approach*. PhD thesis, University of Twente, PO Box 217, 7500 AE Enschede, The Netherlands, may 2014.

[22] Gunther Sessler, Ricard Abello, Nick James, Roberto Madde, and Enrico Vassallo. Gmsk demodulator implementation for esa deep-space missions. *Proceedings of the IEEE*, 95(11):2132–2141, 2007.

[23] Marvin K. Simon and Charles C. Wang. Differential detection of gaussian msk in a mobile radio environment. *Vehicular Technology, IEEE Transactions on*, 33(4):307–320, Nov 1984. ISSN 0018-9545. doi: 10.1109/T-VT.1984.24023.

[24] Bernard Sklar. *Digital communications*, volume 2. Prentice Hall NJ, 2001.

[25] Oscar Starink. State-save overhead reduction techniques for shared accelerators in an mpsoc with a ring noc. Master's thesis, University of Twente, Oct 2015.

[26] Harm te Hennepe. Master's thesis, University of Twente, 2016. Unpublished.

[27] Linear Technologies. Ltc2267-14 datasheet, 2011. URL http://www.linear.com/product/LTC2267-14.

[28] D. Truong, W. Cheng, T. Mohsenin, Zhiyi Yu, T. Jacobson, G. Landge, M. Meeuwsen, C. Watnik, P. Mejia, Anh Tran, J. Webb, E. Work, Zhibin Xiao, and B. Baas. A 167-processor 65 nm computational platform with per-processor dynamic supply voltage and dynamic clock frequency scaling. In *VLSI Circuits, 2008 IEEE Symposium on*, pages 22–23, June 2008. doi: 10.1109/VLSIC.2008.4585936.

[29] Andrew J Viterbi. Error bounds for convolutional codes and an asymptotically optimum decoding algorithm. *Information Theory, IEEE Transactions on*, 13(2):260–269, 1967.

[30] Gerben G.A. Wevers. Hardware accelerator sharing within an mpsoc with a connectionless noc. Master's thesis, University of Twente, September 2014. URL http://essay.utwente.nl/66088/.

[31] *MicroBlaze Processor Reference Guide*. Xilinx, ug081 (v9.0) edition, 2008.

[32] *Virtex-6 FPGA DSP48E1 Slice User Guide*. Xilinx, ug369 (v1.3) edition, 2011.

[33] *Virtex-6 FPGA Data Sheet: DC and Switching Charactistics*. Xilinx, ds152 (v3.6) edition, 2014.

[34] A. Yongacoglu, D. Makrakis, and Kamilo Feher. Differential detection of gmsk using decision feedback. *Communications, IEEE Transactions on*, 36(6):641–649, Jun 1988. ISSN 0090-6778. doi: 10.1109/26.2784.