Turbo Coding

Hardware Acceleration of an EGPRS-2 Turbo Decoder on an FPGA

Master Thesis, AAU,
Applied Signal Processing and Implementation
Spring 2009

Group 1041
Jesper Kjeldsen

AALBORG UNIVERSITET

Aalborg University
Institute for Electronic Systems
Fredrik Bajers Vej 7
DK-9100 Aalborg
Phone: (+45) 9635 8080
http://es.aau.dk

Title:

Turbo Coding
Hardware Acceleration of an EGPRS-2 Turbo Decoder on an FPGA

Theme:
Master Thesis

Project period:
E10, Spring term 2009

Project group:
09gr1041

Participants:
Jesper Kjeldsen

Supervisors:
Peter Koch
Ole Lodahl Mikkelsen

Copies: 4

Number of pages: 120

Appendices hereof: 30

Attachment: 1 CD-ROM

Completed: 18-06-2009

Abstract:

This report presents a hardware implementation of an EGPRS-2 turbo decoder algorithm called soft-output Viterbi algorithm (SOVA), where techniques for optimizing the implementation has been used to establish an Finite State Machine with Datapath (FSMD) design. The report is developed in cooperation with Rohde & Schwarz Technology Center A/S (R&S). EGPRS-2 is the second evolution of GPRS, a standard for wireless transmission of data over the most widespread mobile communication network in the world, GSM. The technologies utilized by EGPRS-2 to improve Quality of Service is investigated in this report to determine an application with high complexity. Turbo coding is chosen and its encoder and decoder structures are analyzed. Based on the interest of R&S, a SOVA implemented in Matlab is analyzed and through profiling a bottleneck, that takes up 70 % of the decoders execution time, is found. This bottleneck is mapped to an FSMD implementation, where the datapath is determined through cost optimization techniques and a pipeline is also implemented. XILINX Virtex-5 is used as an implementation reference to estimate a decreased execution time of the hardware design. It shows that a factor 1277 improvement over the Matlab implementation can be achieved and that it is able to handle the maximum EGPRS-2 throughput speed of 2 Mbit/s.

Preface

This report documents the work for the master thesis; Turbo Coding - Hardware Acceleration of an EGPRS-2 Turbo Decoder on an FPGA. It represents the work done on a semester long project at Applied Signal Processing and Implementation master specialization at Department of Electronic Systems, Aalborg University, Denmark.

The project proposal was presented by Rohde & Schwarz Technology Center A/S and a special thanks goes out to M. Sc. E.E. Ole Lodahl Mikkelsen, SW developer at Rohde & Schwarz Technology Center A/S, for his help throughout the course of this project.

For notation some basic rules are set. *Italic* notation is used for Matlab functions, while a `typewriter` font is used for variables inside these Matlab functions. Furthermore "()" indicates if a variable contains more than one value, so `variable1()` would contain a set of values, whereas `variable2` would only consist of one value.

Citations are written in square brackets with a number, e.g. [3]. The citations are listed in the bibliography on page 90.

The report is composed of three parts: the main report, appendix, and an enclosed CD-rom. The CD-rom contains Matlab code and a digital copy of this report.

Aalborg University, June 18^{th} 2009

Jesper Kjeldsen

Contents

List of Acronyms

3GPP Third Generation Partnership Project

ACK Acknowledge

ASIC Application-Specific Integrated Circuit

AWGN Additive White Gaussian Noise

BCJR Bahl, Cocke, Jelinek and Raviv

BER Bit Error Rate

BSC Base Station Controller

BRAM Block Random-Access Memory

BTS Base Transceiver Station

BUF Buffer

CLB Configurable Logic Block

CMT Clock Management Tiles

EDGE Enhanced Data rates for GSM Evolution

EGPRS Enhanced GPRS

FPGA Field-Programmable Gate Array

GERAN GSM/EDGE Radio Access Network

GGSN Gateway GPRS Support Node

GMSC Gateway Mobile Switching Center

GPRS General Packet Radio Service

GSM Global System for Mobile communications

I/O Input/Output

HLR Home Location Register

HSPA High Speed Packet Access

ITU International Telecommunication Union

LTE Long-Term Evolution

LUT Look Up Table

LLR Log-Likelihood Ratio

MMS Multimedia Messaging Service

MSC Mobile Switching Center

NACK Not Acknowledge

NW Network

PCCC Parallel Concatenated Convolutional Code

PoC Push to Talk Over Cellular

PSTN Public Switched Telephone Networks

QoS Quality of Service

RLC Radio Link Control

RTTI Reduced TTI

SGSN Serving GPRS Support Node

SISO Soft-Input Soft-Output

SNR Signal-to-Noise Ration

SOVA Soft-Output Viterbi Algorithm

TDMA Time Division Multiple Access

TTI Transmission Time Interval

VLR Visitor Location Register

VoIP Voice over IP

WCDMA Wideband Code-Division Multiple Access

Chapter 1

Introduction

Right now wireless communication systems are on the verge to a change in generation from 3G to 4G. An indication of this generation change is illustrated by the amount of research done in the two different generations. Figure 1.1 shows how the number of articles about 3G have declined the last couple of years, while articles about 4G have increased.

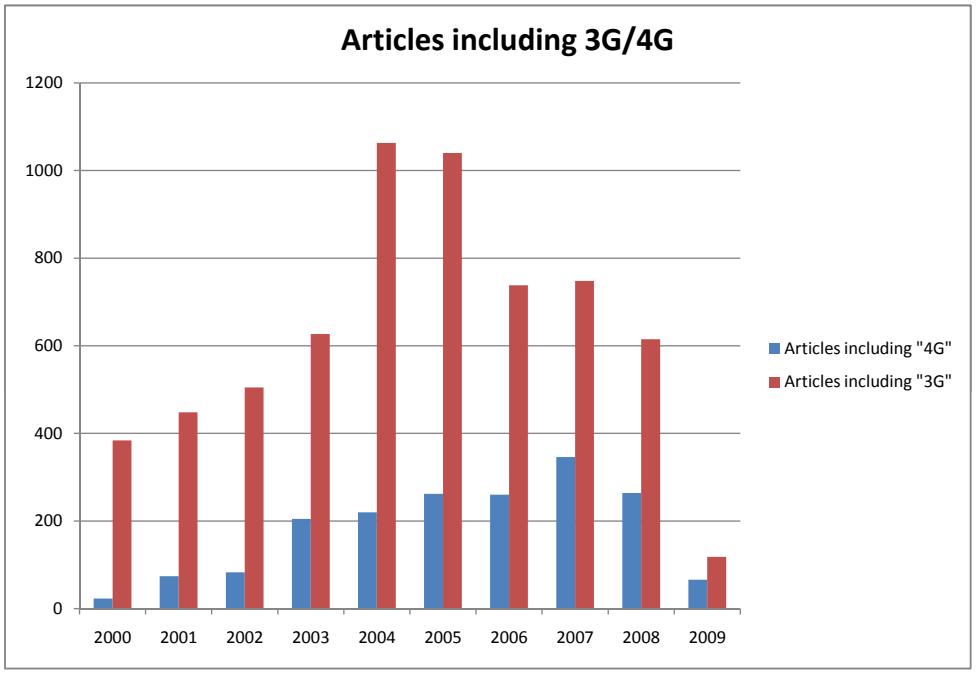


Figure 1.1: Numbers of articles about 3G and 4G wireless communication [1].

There are two reasons why a new generation is of interest. One is because the demand for fast and stable connections to the Internet has increased with the introduction of Apple's new 3G iPhone and phones alike. Here music, games, movies and social networking (such as facebook)

is an essential part of the phone, which all are applications that require fast data connection for streaming or downloading. Another reason is that mobile communication service providers wants to stay competetive by introducing new services, but also wants to provide old services at a lower cost [2, p. 16]. Upgrading the mobile communication infrastructure to support these services is a costly and time-consuming task. As for today, not even 3G is completely deployed in Denmark as illustrated in figure 1.2. Therefore the 3GPP - a collaboration of telecommunication associations that manages the specifications for the 3G and GSM based mobile phone systems - wants to evolve the already existing infrastructure in small steps. At this point 3GPP has finished the specifications for Evolved EDGE (Enhanced Data Rates for GSM Evolution), an intermediate step to LTE. This should make a smooth transition to LTE-Advanced that is going to be 3GPP submission to the ITU as their suggestion for 4G (also called IMT-Advanced) [2, p. 540]. Evolved EDGE (also called EGPRS-2 which will be used in the following) is as the name indicates a further step in enhancing the data rate for GSM based mobile systems.

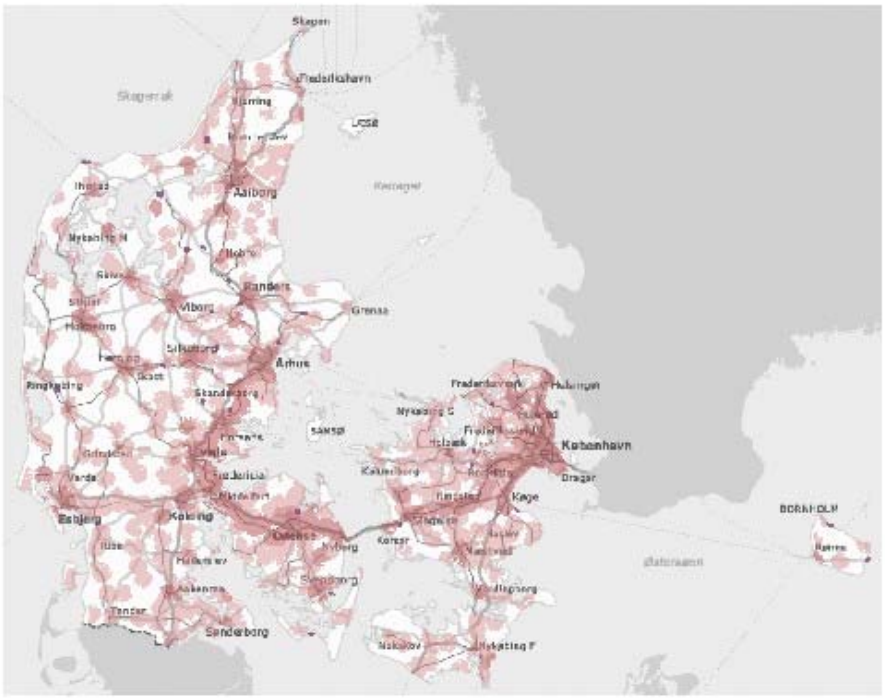


Figure 1.2: 3G/Turbo 3G coverage provided by TDC over Denmark, where red symbolizes Turbo 3G coverage and pink symbolizes regular 3G [3].

With EGPRS-2, 3GPP wants to improve mobile phone connectivity by introducing a variety of applications that cut down latency, improves BER and increase data speed with only minor changes to the already deployed infrastructure [2]. This provides the mobile communication service providers with a cheap addition to their infrastructure, that will allow them keep up with public demand and the possibility of introducing new services.

1.1 Scope of Project

The minor changes introduced by the implementation of EGPRS-2 puts a demand for high performance systems, which are capable of coping with the increased data rate and the more advanced error correction and latency reduction algorithms. Therefore it is of interest to find a platform that provides high performance. Moreover it is interesting to investigate possible optimization of the new algorithms introduced by EGPRS-2 for faster execution. As the scope of optimizing an entire EGPRS-2 receiver on different platforms is too large for this project, it will instead revolve around a single application implemented on a specific platform. Rohde & Schwarz Technology Center A/S has proposed that implementation on the Virtex-5 FPGA platform would be of interest, as it consists of an high-speed FPGA with an embedded Power PC (PPC). This platform makes it possible to compare three different kinds of implementations - an all hardware (HW) on the FPGA, an all software (SW) on PPC and a hybrid of HW and SW. This results in the following tasks for this project:

- EGPRS-2
  - Investigate the receiver structure to identify an application for implementation on the FPGA.
  - Define a set of metrics that is of interest for the implementation of this application based on industry demands.
  - Investigate the algorithm of the chosen application with special interest in bottlenecks and other delimiting factors of the application.
  - Analyze the algorithm or piece hereof and find suitable techniques for optimizing an implementation.
  - Synthesize a design based on metrics and optimization techniques for FPGA implementations.
- Platform:
  - Investigate Virtex-5 FPGA platform and describe components necessary in the implementation of the synthesized design.
  - Map the synthesized design so it utilizes the features of the platform.
  - Establish whether or not the synthesized design is capable of executing at a sufficient speed for the implementation to be applicable.

1.2 Delimitation

EGPRS-2 introduces several interesting features for optimizing peak bit rates and spectrum efficiency, as will be explained in chapter 3. The scope of the project is therefore, already at this point, limited to an investigation of these features and is thoroughly explained in other literature. This is done, since the goal for this project is not to invent a new receiver architecture or parts of it. Instead the goal is to investigate the effects of optimizing a specific application while mapping it to a given platform and try to develop a framework for for this process.

Before investigating the different applications introduced by EGPRS-2, a short explanation of why the industry (in this case R&S) is interested in this project, is presented. Establishing which products that could be affected of the work done in this project also gives an idea of which metrics are of interest for R&S.

1.3 The A^3 Paradigm

A simple model is used for structuring this project. With the help of the A^3 paradigm it should be easier to establish the next step in the design flow. Furthermore it will provide the reader with a guideline making the report easier to read, and the process of the design easier to follow. It will also be a help in establishing a framework for the process of this project, as stated in the delimitation.

The A^3 paradigm is illustrated in figure 1.3, and it will be presented in the start of the following chapters. The object and/or transition between objects that is investigated in a given chapter, will be emphasized in the A^3 model introducing the chapter. The A^3 paradigm consists of the

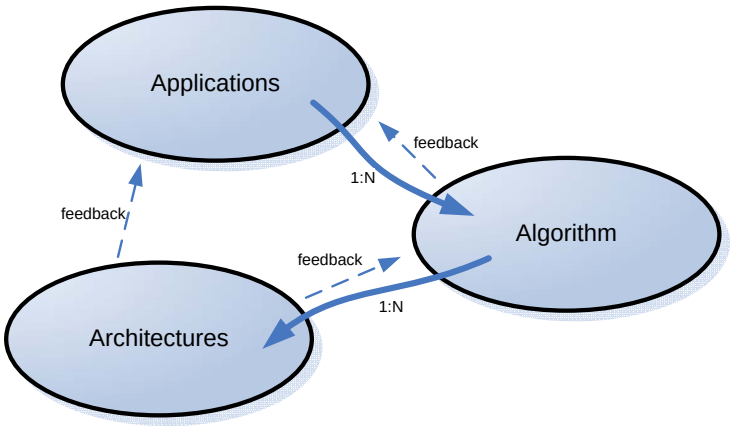


Figure 1.3: This paradigm is used throughout the entire design phase in this report [4].

three domains: Applications, Algorithms and Architectures. As indicated in fig. 1.3 the model

describes the mapping of one application to many algorithms (1:N) and one algorithm to many platforms. In the Application domain a or system is specified, analyzed and a main application or set of tasks is derived from this analysis. There may be several algorithms capable of solving the tasks specified by the system, but only the algorithm that best fits the requirements specified for the system is chosen. In the Algorithm domain a number of algorithms appropriate algorithms are analyzed and one specific algorithm is chosen. If no algorithm is capable of solving the tasks of the application domain in a sufficient way, then this domain has to be revised, as indicated by the feedback arrow. The last domain is Architecture, where a number of platforms are investigated to find the one best suited for the algorithm. The algorithm is then implemented and the results of this implementation is held up against the requirements set by Application. If the architecture can not comply the these requirements a revision of these may be necessary. The same goes for the algorithm domain.

In figure 1.4 the A^3 is applied to this project. It is illustrated how a turbo decoder application for EGPRS-2 leads to two decoding algorithms - SOVA and BCJR/Log-MAP algorithm - where SOVA is profiled and a bottleneck is established. This bottleneck is then mapped to two different pipeline designs for implementation on a Virtex-5 FPGA. Finally the two designs are compared with the requirements set by the specification for EGPRS-2 and an architecture is chosen.

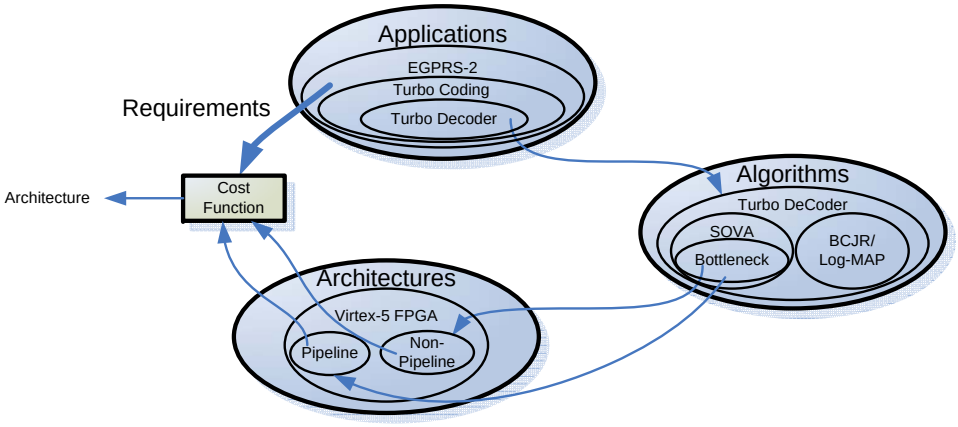


Figure 1.4: A^3 applied to this project.

Chapter 2

Interests of R&S (Industry)

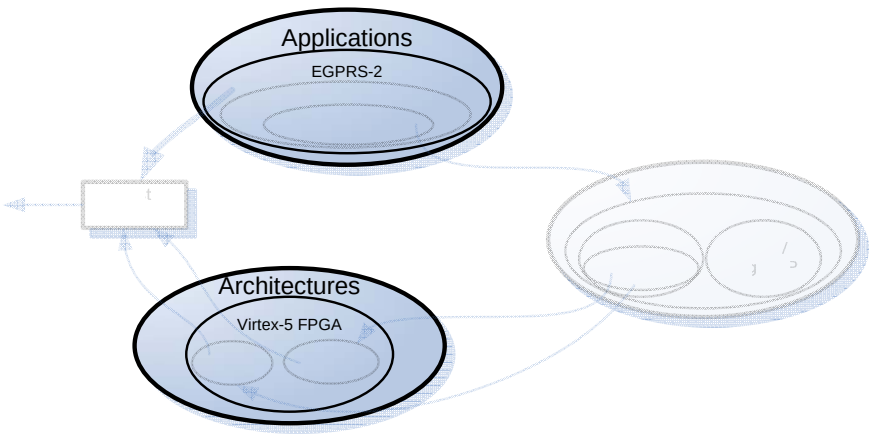


Figure 2.1: As this chapter explains why R&S is interested in EGPRS-2 and its applications, the Application object is emphasized. Some attention to choice of platform is also put in this chapter, thus the architecture object is not totally faded.

Rohde & Schwarz Technology Center A/S proposed this project, and this chapter will concern R&S' interest in the subject. It will show why R&S is interested in EGPRS-2 and its applications. This chapter should also help in further delimitation to the scope of this project.

R&S was started by the two doctors - Dr. Hermann Schwarz and Dr. Lothar Rohde. They where both physicists who met while studying under the same professor at the university in Jena, Germany. Their first product was an instrument for measuring interference over a large wavelength range and was to be used in laboratories. Since then R&S have kept on making measurement and laboratory equipment, but also added broadcasting and radio communication systems to their portfolio.

As described in the delimitation, this project will concern the implementation considerations of some application in the EGPRS-2 receiver part on an FPGA with an embedded Power PC (PPC).

R&S interest in this is connected to their test and measurement equipment. Their product portfolio consist of equipment capable of protocol testing 2G/3G mobile radio standards. The product of interest for this EGPRS-2 project is the R&S®CRTU seen in fig. 2.2. This equipment is used in the research and development of newly developed systems, for a wide variety of wireless communication concepts used in the 2G/3G mobile radio standards. It also provides the user the opportunity to test if certain services, such as MMS, PoC and video telephony is plausible for a given system. The flexibility of the R&S®CRTU provides the user with a test bed for almost any kind of application, as well as the ability to program one's own test scenarios. Testing of mobile phone systems is time consuming, as several hundred of tests are necessary to establish that a mobile phone satisfy the requirements set by 3GPP. To make up for this, it is possible to make completely automatic test sequences on the R&S®CRTU, it is even capable of controlling external equipment [5].

Figure 2.2: R&S®CRTU protocol tester for 2G/3G mobile radio standards [6].

Looking at these features shows two underlying demands. First of all it puts a demand for high performance, not only for reducing test time, but also for testing peak performances of a given communication equipment or system. The list of features also shows a high demand for flexibility. This is especially necessary for further research of communication standards, as new methods may suddenly show themselves beneficial for implementation. As R&S' interest for EGPRS-2 is in the development of test and measuring equipment and not in the development of new handsets, it is safe to conclude that area cost is not a parameter of much concern. Power consumption is always a concern in a environmental aspect, but compared to how significant it is in wireless components, it is not a big issue for this design.

The high performance and flexibility demands make the FPGA with embedded PPC a platform

of special interest, as it provides a huge amount of computational power as well as the inherently great amount of flexibility provided by an FPGA. In the project proposal, R&S recommends the newest platform from XILINX Virtex-5 FXT, which consists of the Virtex-5 FPGA and an embedded PPC. With such a powerful platform, the application that should be investigated for implementation, should also be the part of the EGPRS-2 receiver that puts the highest demand for computational power.

The following chapter investigates the new technologies introduced by EGPRS-2 for improving the peak bit rates, spectrum efficiency, latency and many other parameters. Through this investigation it is decided which technology should be chosen for further analysis and implementation.

Chapter 3

EGPRS-2; a Short Review

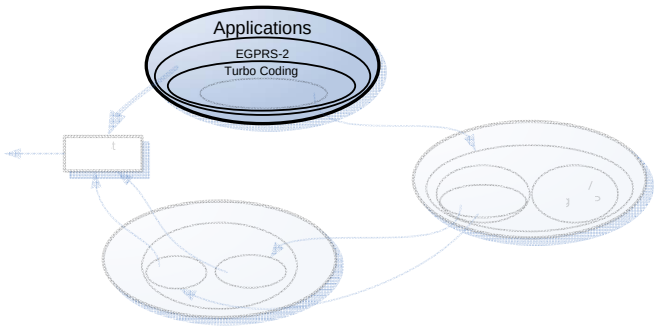


Figure 3.1: In this chapter EGPRS-2, its applications and their properties are investigated.

This chapter will investigate some of the most important new technologies introduced by EGPRS-2 and explain how EGPRS-2 benefits from these technologies. It is based on [2, chapter 24.4] and [7]. The survey done in this chapter leads to a selection of an application used in EGPRS-2 for further investigation and analysis.

3.1 Origin and Goals of EGPRS-2

EGPRS-2 originates from GSM or rather its packet-switched service GPRS. GSM is the most widely used cellular standard in the world with more than 2.6 billion users. Only three countries in the world does not use GSM [2, chapter 1.1]. The first data service introduced in 2G was SMS and circuit-switched data services for e-mail and other applications. 2.5G introduced GPRS, which showed the great potential of packet data in mobile systems. The evolution towards EGPRS-2 started out with the standardization of GPRS as EDGE. This was done to enhance the data transfer rate of GSM and GPRS by introducing higher order modulation. EGPRS originates from the work of this, as several additions, such as advanced scheduling techniques

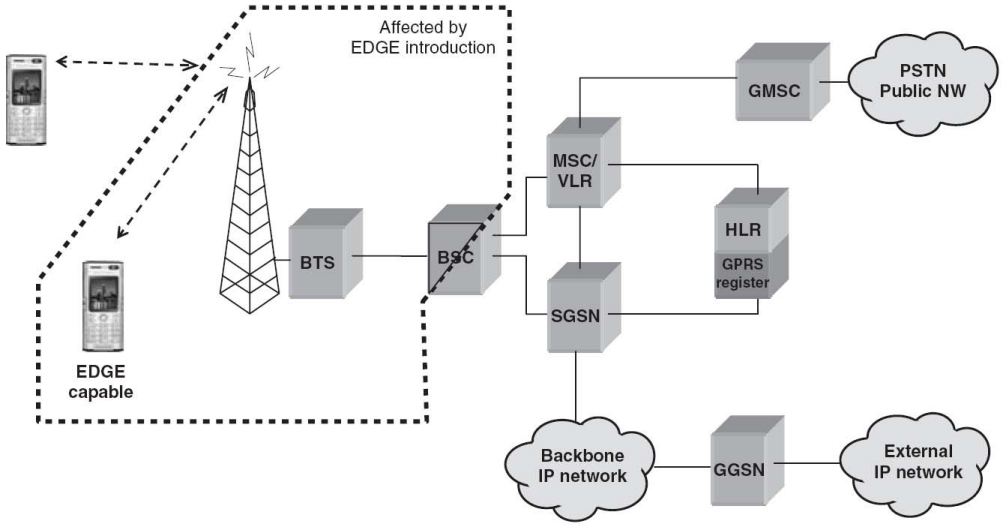


Figure 3.2: GSM network architecture where the part affected by the implementation of EDGE is encircled. It is 3GPPs goal to incorporate EGPRS-2 in the existing GSM/EDGE infrastructure by the same minor effect as was the case for the incorporation of EDGE [2, chapter 24.4]. For explanation of acronyms please locate the acronym list in the beginning of this report p. 1.

were added to the standard. To keep up with new services such as VoIP, 3GPP evolved further on GSM/EDGE based on the study of Evolved GERAN (also called Evolved EDGE or EGPRS-2). Figure 3.2 illustrates the part of the GSM network infrastructure affected by the incorporation of EDGE. 3GPP wants to implement EGPRS-2 with only a minor impact on Base Transceiver Station (BTS), Base Station Controller (BSC) and core network hardware as was the case for EDGE. In this way the already existing GSM/EDGE network architecture is reused and the cost of implementation is minimized. This was also stated in the introduction as a main goal for catching the service provider's interest.

Besides implementing EGPRS-2 in an already existing architecture, 3GPP five essential goals for the new standard, which are listed below:

- 50% increasement of spectrum efficiency.
- 100% increase in data transfer rate in both down- and uplink.
- Improve the coverage by increasing the sensitivity for downlink by 3 dB.
- Improve the service at cell edges planned for voice with 50% in mean data transfer rate.
- Reduce round trip time (RTT) and thereby latency to 450 ms for initial access and less than 100 ms hereafter.

3.2 Technology Improvements for EGPRS-2

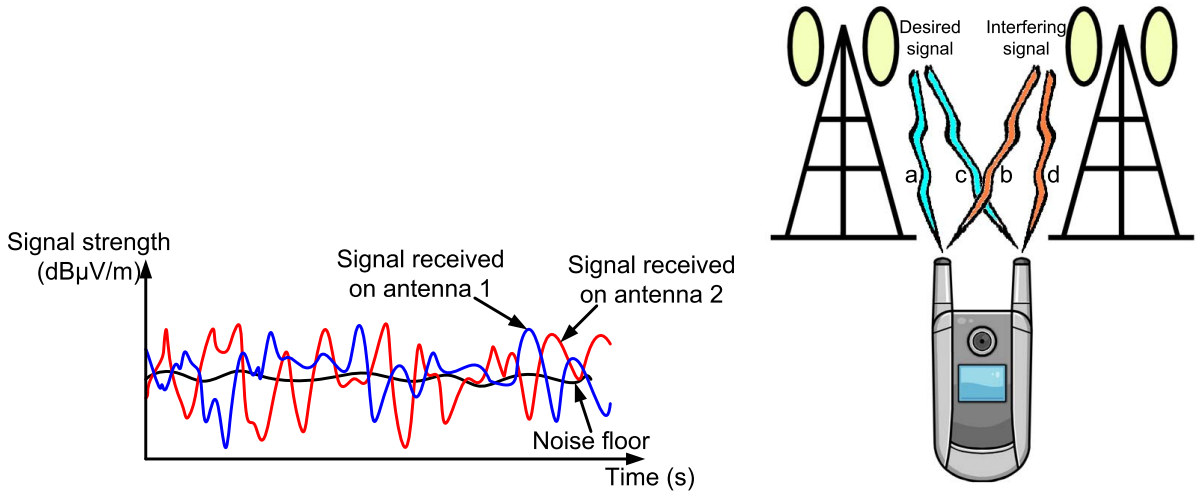
To obtain these goals 3GPP incorporated several technologies in their new standard. These technologies and their benefits are investigated below.

3.2.1 Dual-antenna terminals

Dual-antenna terminals makes it possible to do interference cancellation and reduce the effect of multipath fading. Multipath fading is caused by objects that scatter the signal from transmitter to receiver, which may cause the received signal strength to vary rapidly. In a worst-case-scenario it may even be to weak to be captured. Introducing two antennas with different polarization and/or separated in space, increases the probability that at least one of the received signals, from the same transmission, is above the receivers noise floor as illustrated in fig. 3.3a. Moreover with two antennas it is possible to combine the received signal from both antennas and thereby increase its power, retrieving signals that would have been too weak for one antenna to retrieve. Combining signals is also beneficial when it comes to canceling out an interfering signal. There are different ways of doing so, one is to take the instantaneous attenuation caused by fading into account, which is different for the desired and the interfering signal see fig. 3.3b. Another way is to use the cyclostationary properties of a signal, which differs from transmitter to transmitter. This is investigated in [8]. A paper by the mobile phone manufacturer Ericsson [7], states that dual-antenna terminals could increase the coverage with 6 dB and cope with almost 10 dB more interference compared to EDGE. Eventhough this is an interesting subject, no further investigation will be done in this subject. Implementing dual antennas will impact the terminal itself but will not affect the hardware or software in the BSC.

3.2.2 Multiple Carriers

The idea behind multiple carriers is to increase the data rate by adding carriers to the up- and downlink, so the obtainable speed is increased proportional to the number of carriers. GSM/EDGE uses TDMA with a maximum of eight time slots and a carrier frequency of 200 kHz. With an 8-PSK modulation scheme this would lead to a peak data rate of almost 480 kbps, but due to design and complexity issues, a terminal usually receives on five time slots, while using the last three for transmission and measurement of neighboring cell's signal strength. Still though, by adding e.g. four carriers, it is possible to achieve a theoretical peak bit rate close to 2 Mbps. Figure 3.4 shows how the additional carriers could be perceived. Introducing this technology puts some cost and complexity on the terminal, as it would need multiple receivers and transmitters or a wideband transmitter and receiver. Otherwise the implementation only has a minor effect on the BTS.

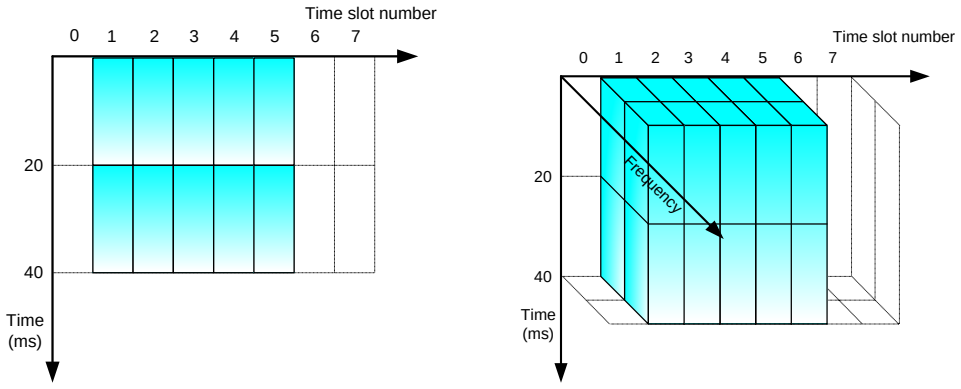


(a) (b)

Figure 3.3: (a) Variation of signal strength in the received signal at two antennas. (b) The received signals are weighted with different fading factors a, b, c and d to illustrate the different paths of the received signals. The desired and interfering signal are combined for each receiver, which makes it possible to cancel the interferer [7].

3.2.3 Reduced transmission time interval and fast feedback

RTTI and fast feedback reduces the latency and in doing so improves the user experience, especially for services such as VoIP, video telephony and gaming. Low delay is crucial for all these services for satisfying the user's demand for quality. With 3GPPs EGPRS-2 standard, the TTI of GSM/EDGE is reduced from 20 ms to 10 ms. As explained earlier, GSM/EDGE uses TDMA to send data via radio blocks on parallel time slots. Each time slot consists of four consecutive bursts over which the radio blocks are transmitted. There are two ways of reducing TTI: One way is by reducing the number of bursts, which brings down the size of radio blocks. The other way is by spreading the bursts out on two carriers with parallel time slots. Latency can also be



(a) (b)

Figure 3.4: (a) Single carrier sending five radio blocks in parallel. (b) Two carriers sending ten radio blocks in parallel. [7].

reduced by faster feedback of information about the radio environment. This is done by the RLC protocol, which sends ACK or NACK from receiver to transmitter depending on whether a data block is received or lost respectively. Faster feedback is obtained by harsher requirements to reaction times, and by immediate response when a radio block is lost. This reduces the latency by ensuring that a lost block is sent as fast as possible. It is also possible to add ACK/NACK to the user data, thereby reducing the overhead. Figure 3.5 illustrates the difference between GSM/EDGE and EGPRS-2 with RTTI and faster feedback.

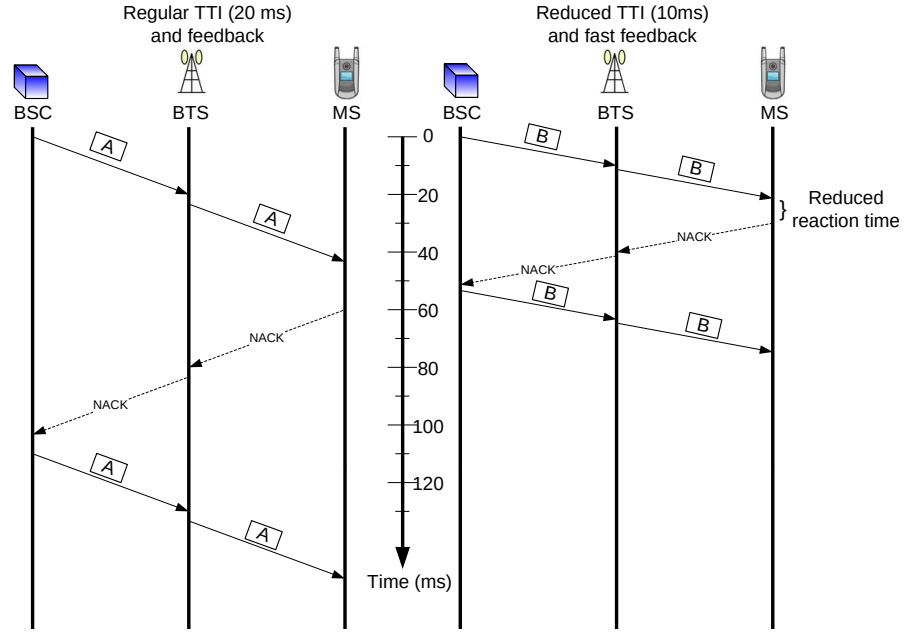


Figure 3.5: Comparison of regular TTI and feedback with reduced TTI and faster feedback [7].

3.2.4 Improved modulation and turbo coding

Improved modulation such as 16QAM and 32QAM is possible due to improved error correcting code, and [7] states an increase around 30% - 40% for user bit rates by this addition. From GSM to EDGE the modulation was changed from Gaussian minimum-shift keying (GMSK) to octonary phase shift keying (8-PSK), which increased the peak transfer rate from approx 20 kbps to almost 60 kpbs. For both modulation schemes convolutional coding was used to recover lost data. EGPRS-2 introduces 16QAM and 32QAM, going from 8-PSKs 3 bits/symbol to 4 and 5 bits/symbol for 16QAM and 32QAM respectively (for higher symbol rates it will also use QPSK). This reduces the distance between signal points as it is illustrated in figure 3.6, indicating that 16QAM and 32QAM are more susceptible to noise and interference. However, by introducing turbo coding, these new modulation schemes becomes feasible for even low SNR levels, accomplishing near Shannon limit data rates [2]. Claude Shannon showed that a com-

munication channel has a capacity (C) in bits/s and when transmitting at a rate (R) lower than this capacity, error free transmission should be obtainable with the right coding. He found the capacity explicitly for the AWGN channel, where the normalized capacity is given by:

$$\frac{C}{W} = \log_2\left(1 + \frac{R}{W}\frac{E_b}{N_0}\right) \quad \text{[bit/s/Hz]} \tag{3.1}$$

where:	W is bandwidth of the channel	[Hz]
	R/W is the normalized transmission rate	[bit/s/Hz]
	E_b/N_0 is the signal bit energy to noise power spectral density ratio	[dB]

Simulations done by Berrou and Glavieux [9] showed turbo codes that could achieve BER of 10^{-5} at $E_b/N_0 = 0.7$ dB, meaning that turbo codes are only 0.7 dB from the Shannon limit (optimum) coding performance, as 10^{-5} is regarded as close to zero. The above equation and explanation of Shannon limit, is taken from [10].

Turbo coding needs large code blocks to perform well and is therefore applicable for high data rate channels such as 8-PSK, 16QAM and 32QAM. This is also the reason QPSK is only applicable for use when having higher symbol rates. Turbo coding more than makes up for the increased noise and interference susceptibility of 16QAM and 32QAM. Turbo coding is more complex and computational heavy than the convolutional code used by GSM and EDGE, however, turbo coding is already in use by WCDMA and HSPA. As many GSM/EDGE terminals already support WCDMA/HSPA, it is possible to reuse this turbo coding circuitry for EGPRS-2 as well.

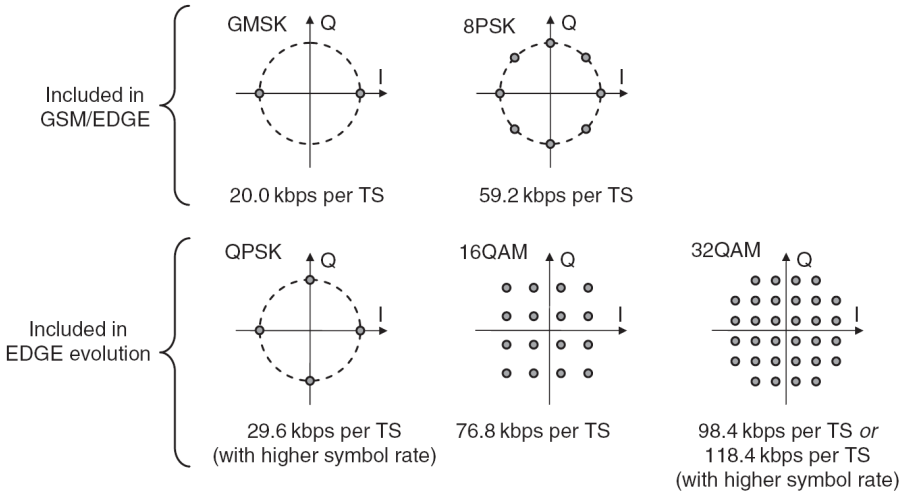


Figure 3.6: Constellation diagrams for GSM (GMSK), EDGE (8-PSK) and evolved EDGE (QPSK, 16QAM and 32QAM) [2, chapter 24.4].

Terminal capability	Symbol rate	Modulation schemes
Uplink Level A	271 ksymbols\s	GMSK, 8-PSK, 16QAM
Uplink Level B	271 ksymbols\s	GMSK
Uplink Level B	325 ksymbols\s	QPSK, 16QAM, 32QAM
Downlink Level A	271 ksymbols\s	GMSK, 8-PSK, 16QAM, 32QAM
Downlink Level B	271 ksymbols\s	GMSK
Downlink Level B	325 ksymbols\s	QPSK, 16QAM, 32QAM

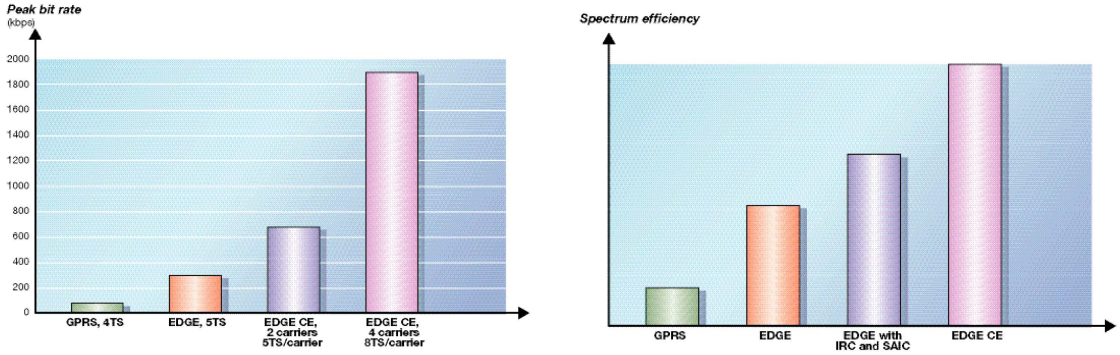
Table 3.1: Listing of which modulation schemes are used at different symbol rates in EGPRS-2. [2, fig. 24.6]

3.2.5 Higher symbol rates

3GPP has increased the symbol rates for EGPRS-2 by 20% for some modulation schemes. Table 3.1 shows which modulation schemes are used for a given symbol rate. Note that QPSK is only used for cases where the symbol rate is increased by 20 %. This is due to the necessity of large data blocks for turbo coding to work properly.

3.3 Conclusion on EGPRS-2 Improvements

As described above many additions can be made to the already existing infrastructure that increases throughput and reduces latency, with only minor changes to the hardware and software at the terminals and base stations. This makes EGPRS-2 easy and cheap to implement and therefore interesting for the service providers that want to keep a competitive edge. Figures 3.7a and 3.7b illustrate the gain of EGPRS-2 compared to the old GPRS and EDGE technologies in peak bit rate and spectrum efficiency respectively. Furthermore table 3.2 displays which features are affected by the new technologies presented in this chapter.



(a) (b)

Figure 3.7: (a) Theoretical peak bit rates for GPRS, EDGE and EGPRS-2 for 2 and 4 carriers. Note the different number of TS for the two EGPRS-2 cases (EGPRS-2 is denoted as EDGE CE) [7]. (b) Spectrum efficiency for GPRS, EDGE, EDGE with single antenna interference cancellation (SAIC) and interference rejection combining(IRC), and EGPRS-2 [7].

Feature/ Technologies	**Mean data rate**	**Peak data rate**	**Latency**	**Coverage**	**Spectrum efficiency**
Dual-antenna terminals	x	-	-	x	x
Multicarrier EDGE	x	x	-	-	x
RTTI and fast feedback	-	-	x	-	-
Improved modulation and coding	x	x	-	-	x
Higher symbol rate	x	x	-	-	x

Table 3.2: The new technologies - introduced by 3GPP to meet the goals stated for EGPRS-2 - have a positive effect on the features marked by x [7].

Turbo coding was introduced as a complex and computationally advanced addition to the EGPRS-2 standardization, making it ideal for investigating the benefits of implementing this technique on different platforms. Furthermore Rohde & Schwarz is interested in possibility of hardware-accelerating the computational heavy turbo code [11]. In the next chapter turbo coding is examined throughly with implementing issues in mind.

Chapter 4

Turbo Coding

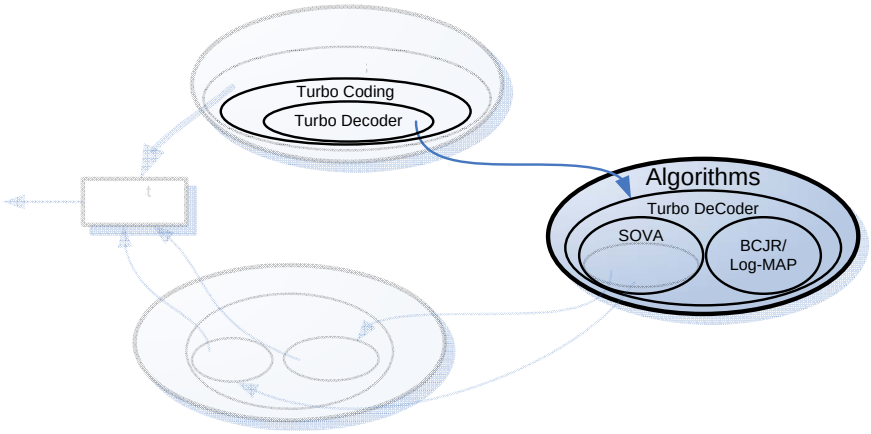


Figure 4.1: A deeper analysis of the turbo code application, which in the end leads to a choice of algorithms for further analysis.

Before an in depth analysis of the turbo decoder algorithm is undertaken, a short survey of both the turbo encoder and decoder is outlined in this chapter. This is done to display the different functions of turbo coding and to provide knowledge that will help in the understanding of the turbo decoder algorithm analyzed in the next chapter. The material concerning the turbo encoder is taken from [12, 5.1a].

4.1 Turbo Encoder

Figure 4.2 illustrates the turbo encoder structure as specified by 3GPP. It consists of two 8-state constituent encoders and an interleaver and it produces a recursive Parallel Concatenated Convolutional Code (PCCC). Here the first constituent encoder receives input bits directly, whereas the second constituent encoder is fed with input bits through the interleaver. Each

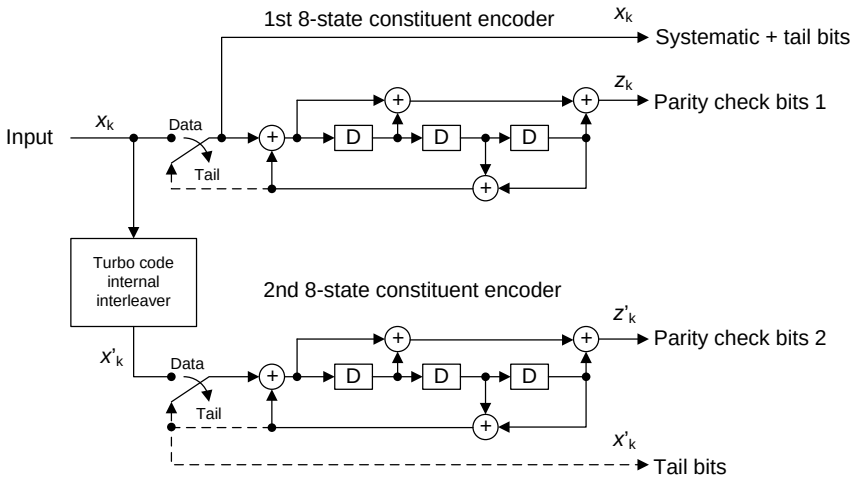


Figure 4.2: Structure of turbo encoder specified by 3GPP for EGPRS-2, the boxes labeled D is the encoders shift registers [12, 5.1a].

constituent encoder is build around a shift register of length = 3 resulting in $2^3 = 8$ different states. Which state each constituent encoder is in depends on the input bits. Furthermore the figure shows that the entire turbo encoder is an 1/3 rate encoder, thus for each input, three outputs are generated. This rate is however altered depending on possible puncturing of bits and tail bits from the second constituent encoder at termination.

The encoder outputs are labeled as seen in equation 4.1, but for emptying the shift registers (termination), the labeling changes as another output is introduced, which is stated in equation 4.2. This labeling is for when the switches of figure 4.2 are in the lower position, meaning trellis termination is activated and the shift registers are brought back to an all zero state. Termination is done by first passing three tail bits into the shift registers of the upper encoder of figure 4.2 and hereafter three tail bits into the lower encoder. While one encoder is being terminated, no bits are put into the other encoder.

$$\begin{aligned} C(3\mathrm{i}-3) &= x_\mathrm{i} \\ C(3\mathrm{i}-2) &= z_\mathrm{i} \qquad\qquad \text{for i} = 1, ..., \mathrm{K} \\ C(3\mathrm{i}-1) &= z'_\mathrm{i} \end{aligned} \tag{4.1}$$

$$\begin{aligned} C(3\mathrm{K}+2\mathrm{i}-2) &= x_{\mathrm{K+i}} \\ C(3\mathrm{K}+2\mathrm{i}-1) &= z_{\mathrm{K+i}} \qquad\qquad \text{for i} = 1, 2, 3 \\ C(3\mathrm{K}+2\mathrm{i}+4) &= x'_{\mathrm{K+i}} \\ C(3\mathrm{K}+2\mathrm{i}+5) &= z'_{\mathrm{K+i}} \end{aligned} \tag{4.2}$$

where:	$C(\mathrm{i})$ represents the output bit stream	[-]
	x_i is the input bit stream	[-]
	z_i is the output from 1st constituent encoder	[-]
	z'_i is the output from 2nd constituent encoder	[-]
	x'_i is the output from the interleaver block	[-]

The generator matrix for one constituent encoder - of the turbo encoder illustrated in figure 4.2 - is given by equation 4.3 where the second entry is the transfer function for the parity check bit generating part. The nominator of this function is the feed forward part of the constituent encoder and the denominator is the feedback part, which besides from the figure, shows that the turbo encoder is recursive. The constituent encoders are made recursively so their shift registers depend on previous outputs. This increases the performance of the encoder, as one error in the systematic input bits (x_i) would result in several errors in parity check bits.

$$\mathbf{G}(D) = \left[1, \frac{1+D+D^3}{1+D^2+D^3}\right] \tag{4.3}$$

4.1.1 Internal Interleaver

In an interleaver the input bits are permuted in a matrix, such that the interleaver output is a pseudo-random string of the input bits. This means that the input and output bits of the interleaver is the same just in a different temporal order. Interleaving helps by linking easily error-prone code and burst errors together with error free code. A simple example of the properties of an interleaver is given in figure 4.3. Without the interleaver it is impossible to derive the message when a burst error corrupts a chunk of letters in the middle of the text. Interleaving the text before sending it through a noisy channel spreads out the burst error and the text is much easier to derive after deinterleaving. In 3GPPs standardization of internal interleaver for turbo coding [12, section 5.1a.1.3.4] the permutation matrix is based on the number of bits per block. Padding and pruning is used to fill up and remove possible empty spaces in the permutation matrix respectively. Functions based on prime numbers and modulus are used to make intra- and inter-row permutation to assure that each input bit is allocated to different indexes in the permutation matrix. For specific knowledge about these functions please refer to [12, section 5.1a.1.3.4].

4.1.2 Puncturing

Puncturing is a way of removing or adding parity bits based on the channel properties. Turbo coding introduces puncturing for two reasons. One is rate matching where bits are punctured so the number of coded bits fit the available bits in the physical channel. Another reason is to make different redundancy versions, adding more parity bits when the decoder fails to decode the

Original Message:
"Prediction is very difficult, especially if it's about the future"
-Niels Bohr, Danish physicist (1885 - 1962)

Received message with burst error:
PredictionIsVeryDifficult,XXXXXXXXXXIfIt'sAboutTheFuture

Original message interleaved:
PioVDitpaI'ohtrcneic,elfsueuetIrfuEclIAtFrdisyflsiytbTue

Interleaved message with burst error:
PioVDitpaI'ohtrcneic,elfsuXXXXXXXXXXAtFrdisyflsiytbTue

De-interleaved message with burst error:
PrXdicXionXsVeXyDiXficXlt,XspeXiallyIfIt'sAboutThXFutXre

Figure 4.3: Interleaver example on a quote from the famous Danish physicists Niels Bohr. X represents an error.

transmitted bits. Puncturing can be used to prioritize between systematic bits and parity bits. An example of prioritizing could be seen between the first transmission and a retransmission. For the first transmission systematic bits should be prioritized so the decoder receives a minimum of redundancy. An error in the decoding of the first transmission implies a need for redundancy and therefore priority bits should be prioritized. Which priority to use can be established by the use of ACK/NACK in the RLC [2, chapter 9].

This concludes the basic functions of the encoder. Next up is an investigation of the functionalities of the turbo decoder and hereinafter an analysis of an specific decoder algorithm.

4.2 Turbo Decoder

The job of the turbo decoder is to reestablish the transmitted data from the received systematic bitstream and the two parity check bitstreams, even though these are corrupted by noise. Or said more specific; the systematic bits x_k are encoded resulting in two sets of parity check bits z_k and z'_k. Then x_k, z_k and z'_k is send through a channel where it is corrupted by noise such that x_k, z_k and z'_k at the decoder may differ from what was originally send. It is now the decoders job to make an estimate $\hat{x_k}$ of x_k based on an estimate of z_k and z'_k ($\hat{z_k}$ and $\hat{z'_k}$).

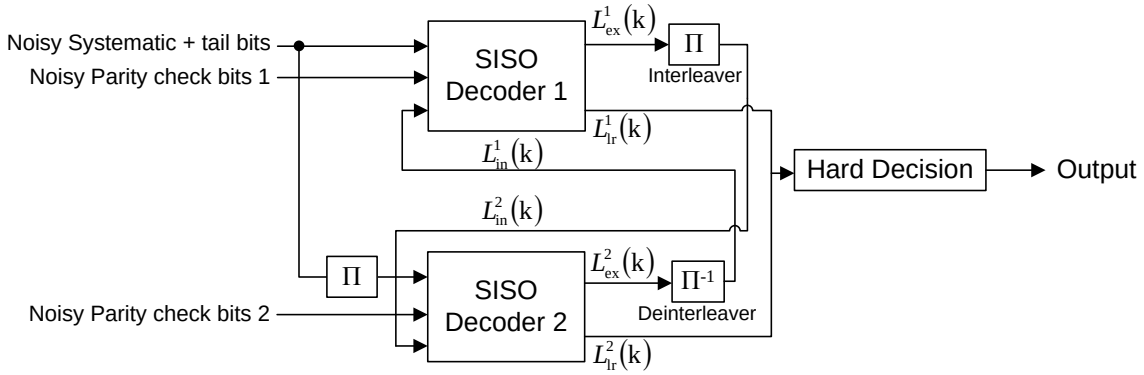


Figure 4.4: Structure of a turbo decoder where the two decoder blocks are soft-input soft-output (SISO) decoders, based on either SOVA or the BCJR algorithm [13, chapter 10].

There are two different well known decoding algorithms, developed for turbo coding: The soft-output Viterbi algortihm (SOVA) invented by Hagenauer and Hoher based on the Viterbi algorithm by Andrew J. Viterbi. The other algorithm is the BCJR algorithm invented by Bahl, Cocke, Jelinek and Raviv. Even though the performance of BCJR is slightly better than that of the Viterbi (never worse), BCJR - which is a maximum *a posteriori* probability (MAP) algorithm - also introduce a backward recursion and is therefore more complex than Viterbi [13, chapter 10]. Figure 4.5 shows that the gain of using the Log-MAP based decoder compared to new SOVA based decoders is only around 0.1 dB for an E_b/N_0 around 2.0 dB [14]. Comparison of complexity for the two algorithms, shows that Log-MAP has a complexity of $O_C(n^2)$, $O_S(2n^2)$ and SOVA has a complexity of $O_C(0.5n^2)$, $O_S(0.5n^2)$, where n is the number of bits for decoding, C stands for comparisons and S stands for summations [15]. The small BCJR decoder gain does not make up for increased complexity cost and is therefore not as interesting for the industry as SOVA [11]. This is the reason for investigating SOVA thoroughly instead of the BCJR algorithm. However some comments on BCJR and differences between the two decoding algorithms is mentioned.

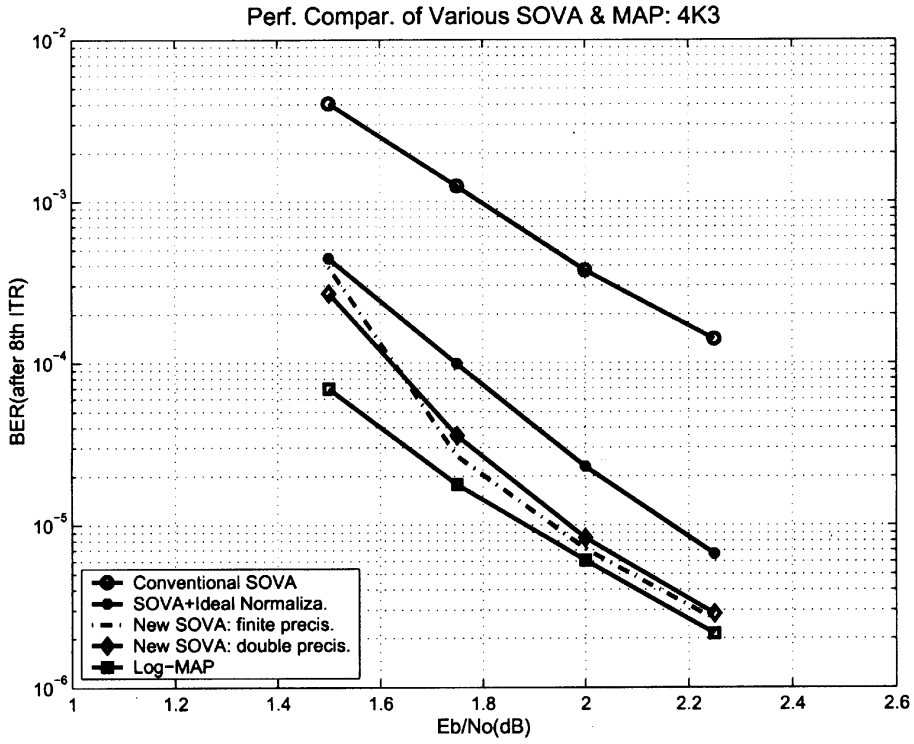


Figure 4.5: Comparison of SOVA based and Log-MAP based turbo decoders [14].

Identically for both SOVA and BCJR is the way that the internal decoders (SISO Dec. 1 and 2) interchange the extrinsic information that is used as *a priori* information in an iterative manner. A descriptions of this process follows here, use figure 4.4 as reference.

From the front-end receiver the received bit stream is demultiplexed into three parts; systematic bits, and parity check bits 1 and 2. SISO Decoder 1 uses the systematic bits, parity check bits 1 and *a priori* information from SISO Decoder 2 (zero at the first iteration) to estimate a bit sequence by use of SOVA. This results in two outputs; extrinsic information L^1_{ex} and a log-likelihood ratio (LLR) L^1_{lr}. The extrinsic information is interleaved on its way to the SISO decoder 2 where it is used as *a priori* information. It is then, together with an interleaved version of the systematic bits and the parity check bits used to form a decoding. SISO decoder 2 - also based on SOVA - outputs extrinsic information L^2_{ex} and a LLR L^2_{lr}. L^2_{ex} is deinterleaved and is used as *a priori* information in SISO decoder 1 for a second iteration on the same systematic and parity check bits as for the first iteration. After a number of iterations the two LLR outputs L^1_{lr} and L^2_{lr} are used to make a hard decision on the bit sequence. The number of iterations needed to provide a good estimate of bit sequence depends on the encoder that is used. E.g. in [13, chapter 10.9] 8 to 10 iterations are suggested for BCJR decoding of a convolutional turbo encoder with generating functions [1, 1, 1] for encoder 1 and [1, 0, 1] for encoder 2. In [16, chapter 9.8.5] 18 iterations is needed to reach a performance just 0.7 dB from the Shannon limit. Please refer to the stated sources for further details. The decoder structure illustrated in figure 4.4 is well

known by the industry and is also the one used by R&S.

4.2.1 Viterbi Decoding Algorithm

The Viterbi algorithm works as a maximum likelihood sequence estimator utilizing the encoders trellis diagram, hence it looks at all possible sequences through a trellis diagram and chooses the sequence or ”path” with the highest likelihood. It uses the Hamming distance between incoming bits and possible transitions in the encoder (or trellis) as a metric to establish which path has the highest likelihood. An example of this is shown in figure A.1 in appendix A on p. 91 for decoding of a simple convolutional code. This is different from the BCJR algorithm, which also uses the encoders trellis diagram but treats the incoming bits as a maximum a posteriori probability (MAP) detection problem and produces a soft estimate for each bit. Viterbi on the other hand finds the most likely *sequence* and thereby estimates several bits at once instead of maximizing the likelihood function for each bit. This is the reason why Viterbi does not perform as well as BCJR [13, chapter 10].

Looking at the trellis diagram for one of the two constituent encoders in figure 4.6, it becomes clear that there are several possible paths through this diagram. To calculate all possible paths would require an immense amount of memory, so instead Viterbi does two things to reduce the amount of memory needed. Viterbi introduces something called survivor paths, which are the paths through the trellis diagram with smallest Hamming distance. Viterbi only keeps 2^{K-1} survivor paths where K is the constraint length of the encoder - the encoders memory plus one ($K = M + 1$). This means for the sake of one single encoder in figure 4.2 a total of 8 paths have to be stored in which the path with the maximum likelihood choice is always contained [13, chapter 10]. To decrease the amount of memory needed even further, a window of length *delta* is used. This lets the Viterbi algorithm work on small frames of the trellis diagram, where a decision of the best path is made for each iteration, outputting the resulting symbol for the first branch of the trellis. The decoding window is then moved forward one time interval (branch) and a new decision is made based on the code encapsulated by this frame. This means that the decoding no longer is truly maximum likelihood, but keeping the window length $delta = 5 \cdot K$ or more has shown satisfying results [13, chapter 10]. For a good illustration of the Viterbi decoding algorithm, check out http://www.brianjoseph.com/viterbi/workshop.htm (Java is required).

4.3 Conclusion on Turbo Coding

The SOVA presented above is the de facto standard for industry turbo decoding, as it performs nearly as well as the Log-MAP algorithm at a much lower cost. In the following chapter further analysis of SOVA is done to find potential bottlenecks.

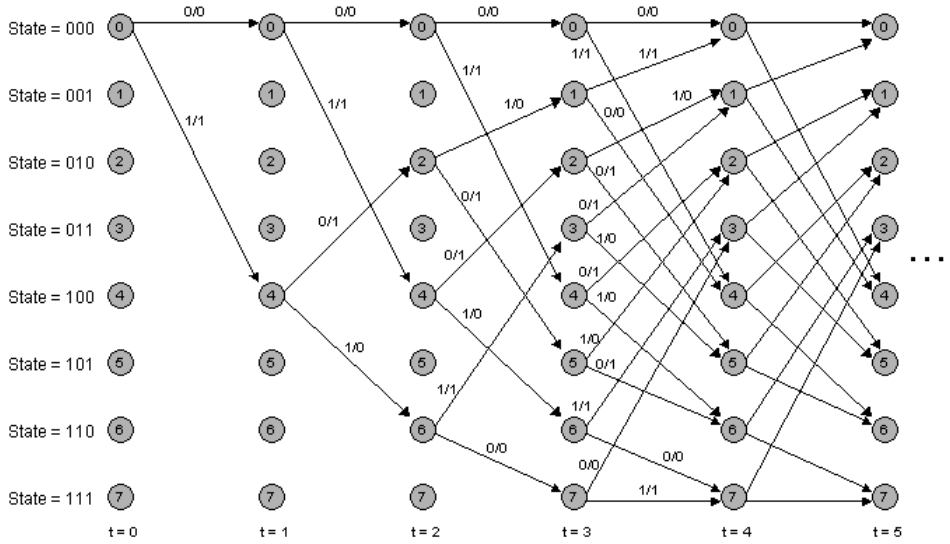


Figure 4.6: Trellis diagram for one 8-state constituent encoder as illustrated in figure 4.2 [17].

Chapter 5

Algorithm Analysis of SOVA

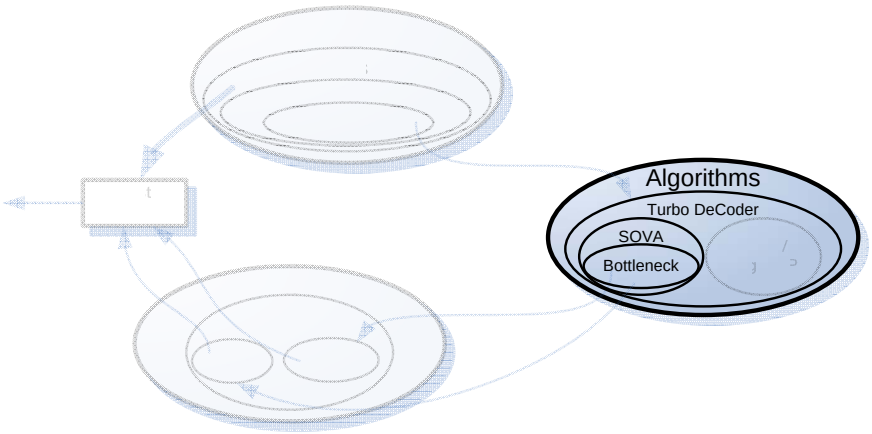


Figure 5.1: Based on the selection of SOVA in the previous chapter, this algorithm is further analyzed mainly by profiling. Based on the results of this profiling a part of SOVA is chosen for in-depth analysis and implementation.

In this chapter the turbo decoding algorithm discussed in the previous chapter is analyzed and profiled to establish possible bottlenecks that later on will be mapped to an architecture. Soft-output Viterbi algorithm (SOVA) implemented as the SISO decoders illustrated in 4.4 is the algorithm of choice. The SOVA decoder algorithm used for this in-depth analysis is part of Yufei Wu's [18] turbo encoding and decoding simulation script written in Matlab. Besides incorporating a SOVA decoder it also has a Log-MAP decoder and gives the user the opportunity to specify the generating function of the encoder, which also determines the trellis diagram used by the SOVA decoder. Yufei Wu's SOVA decoder have been used and referred to in other works such as [19], [20] and the book [21]. As this is a simulation tool it is not optimized to achieve the same performance as the improved SOVA decoders in figure 4.5. It will, however, be possible by profiling to identify bottlenecks in the SOVA decoder, bottlenecks being parts of the code that reduce the performance of the entire decoder algorithm.

After profiling and a description of the algorithms functions and subfunctions, the part of the algorithm identified as a bottleneck is mapped to an FPGA as an ASIC architecture. During this mapping optimizing techniques for area and performance is applied in an effort of reducing the cost and execution time of the bottleneck. The Matlab code for Yufei Wu's entire encoder/decoder simulation script is located on the enclosed CD.

5.1 Profiling

Matlab's own profiler is used for profiling four different setups on two different systems - a desktop and a laptop. This is done to investigate if different setups introduce different workloads. The profiling is done on the entire script, but further analysis will revolve around the decoder algorithm.

5.1.1 Setup

A few modification to the Matlab code is made to make it run automatically. The unmodified code asks for parameters as it is initialized, which will show up as a time consuming task in the profiler. Therefore the Matlab code is modified to contain the necessary parameters upon initialization. Both the modified (*ModifiedSOVAtest_turbo_sys_demo.m*) and unmodified (*turbo_sys_demo.m*) Matlab code is located on the enclosed CD. The parameter settings are the following (the brackets indicate the parameter setting):

- Log-MAP or SOVA decoder: (SOVA decoder).
- Frame size, the number of information bits + tail bits per frame and also the size of the internal interleaver: (400 bits default by Matlab script).
- Code generator matrix **G** where $\mathbf{G}[1,:]$ is feedback and $\mathbf{G}[2,:]$ is feedforward: (Two different code generators is tested: specific for EGPRS-2 $\mathbf{G1} = [0011; 1101]$ and default for the Matlab code $\mathbf{G2} = [111; 101]$).
- Punctured or unpunctured: (Both are tested resulting in a rate 1/3 test and 1/2 test respectively).
- Number of decoder iterations: (5 default by Matlab script).
- Number of frame errors used as stop criteria: (15 default by Matlab script).
- SNR for a AWGN channel as given by E_b/N_0: (2 dB default by Matlab script).

Components	Desktop	Laptop
Operating System	Windows 7 Ultimate 32-bit	Windows Vista Business 32-bit SP1
Processor	Intel Core 2 Duo E6400 2.13 GHz	Intel Core 2 Duo T9600 2.80 GHz
Motherboard	Intel 945G Express (Gigabyte)	N/A
Memory	3 GB PC2-5300 667 MHz	3 GB PC3-8500 DDR3 1067MHz

Table 5.1: Components of test systems.

SOVA decoder is chosen as it is the decoder of interest. The rest of the parameters with a constant value is based on the given default value of the Matlab script. The different generating functions is to identify if any change in workload is detectable when changing the encoder and thereby the trellis for the decoder. The shift in between punctured and unpunctured is to see if the workload of the decoder is changed by an increase of redundancy provided by the encoder. Unpunctured coding should reduce the BER, increasing the number of iterations needed to reach the stop criteria, which should be accounted for when inspecting the profiling results.

The four test setups was tested on a desktop and a laptop with the specifications stated in table 5.1. This is done to see if different profiling results is produced on different systems. In both systems Matlab version 7.8.0.347 (R2009a) is used.

Both systems support dual core processing, hence they are able to use two processors for execution of the Matlab code. However, to perform the most efficient and accurate profiling, Mathworks recommends that only one CPU is activated, during profiling [22].

5.1.2 Profiling Results

Profiling on the two different systems only resulted in a change of execution time, the laptop being twice as fast as the desktop for all four test setups. Looking at the execution time spend in each function percentage wise, shows neglectable deviations between the two systems. Therefore only the results for the desktop profiling is given in this section. The results of the profiling is stated in table 5.2. The changes in parameters implies four different tests: **G1** rate=1/2, **G1** rate=1/3, **G2** rate=1/2, and **G2** rate=1/3.

Function Name	G1 rate=1/2	G1 rate=1/3	G2 rate=1/2	G2 rate=1/3
sova0()	94.2279 %	93.6555 %	94.1342 %	93.8641 %
encoderm()	1.9468 %	2.2488 %	2.1410 %	2.3735 %
rsc_encode()	1.6652 %	1.7876 %	1.8351 %	1.8746 %
encode_bit()	1.1945 %	1.3455 %	1.2661 %	1.2851 %
Functions with < 1 % of total time	0.9657 %	0.9626 %	0.6236 %	0.6026 %

Table 5.2: Results from profiling Yufei Wu's turbo encoder/decoder simulation script.

As expected the overall profiling of the Matlab script stated that the by far biggest amount of computation time is located in the decoder function *sova0()*. For every setup, *sova0()* takes up more than 93 % of the total computation time as seen in table 5.2. The Matlab profiler also states the amount of time used in each line of code inside the *sova0()* function. Inspection shows that one *for* loop in *sova0()* stands for almost 70 % of the entire execution time of the *sova0()* function, indicating a bottleneck. Remember that the number of frame errors was set as stop criteria and depending on the decoder and the rate of the code (punctured/unpunctured) the execution time will differ from setup to setup. However these different runtimes does not have much effect on the position of the workload. A small increase is spotted for rate 1/3 at the *encoderm()* function, which is due to an increase in making unpunctured mapping. A cut out of the profiling results is illustrated in figure 5.2 to confirm this.

A description of the SOVA decoder is given in the next section, before further investigation into the bottleneck of the *sova0()* function is undertaken. This will provide an overview of the entire decoder, while most effort is put into explaining the *sova0()* function.

time	calls	line	
	12	53	if puncture > 0 % unpunctured
	12	54	for i = 1:L_total
< 0.01	4800	55	for j = 1:3
0.15	14400	56	en_output(1,3*(i-1)+j) = y(j,i);
< 0.01	14400	57	end
< 0.01	4800	58	end

(a)

time	calls	line	
	12	59	else % punctured into rate 1/2
	12	60	for i=1:L_total
0.03	4800	61	en_output(1,n*(i-1)+1) = y(1,i);
< 0.01	4800	62	if rem(i,2)
		63	% odd check bits from RSC1
0.02	2400	64	en_output(1,n*i) = y(2,i);
	2400	65	else
		66	% even check bits from RSC2
0.02	2400	67	en_output(1,n*i) = y(3,i);
	2400	68	end
< 0.01	4800	69	end
< 0.01	12	70	end

(b)

Figure 5.2: A cut out from the profiling of *encoderm()* with unpunctured mapping (a) showing an time increase of almost twice that of punctured mapping in (b).

5.2 Decoder Algorithm Structure

To get an overview of the algorithm structure for the SOVA decoder please refer to figure 5.3, which besides from showing the blocks of the decoder and its functions (written in italic), also shows the flow of the decoder algorithm. The reader is encouraged to compare this block diagram with the decoder structure given in fig. 4.4 in chapter 4.2. The Matlab code on which the following is derived is found in appendix B, and should be used as an assistance in grasping the algorithm.

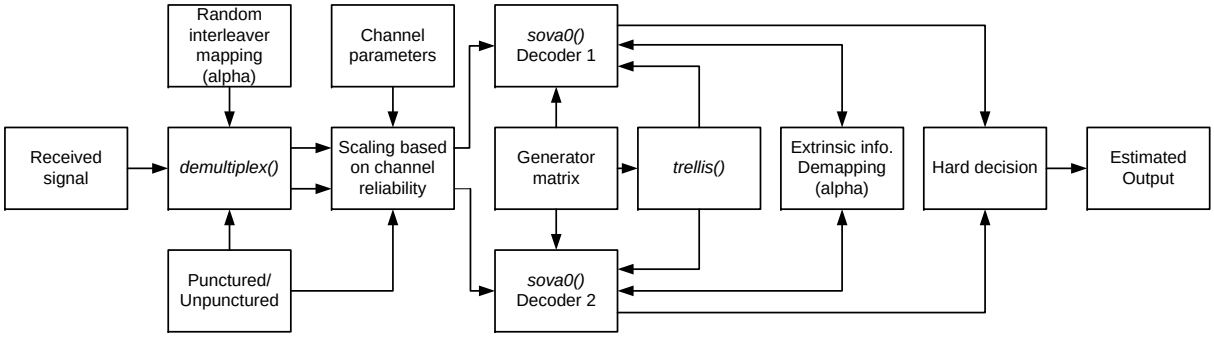


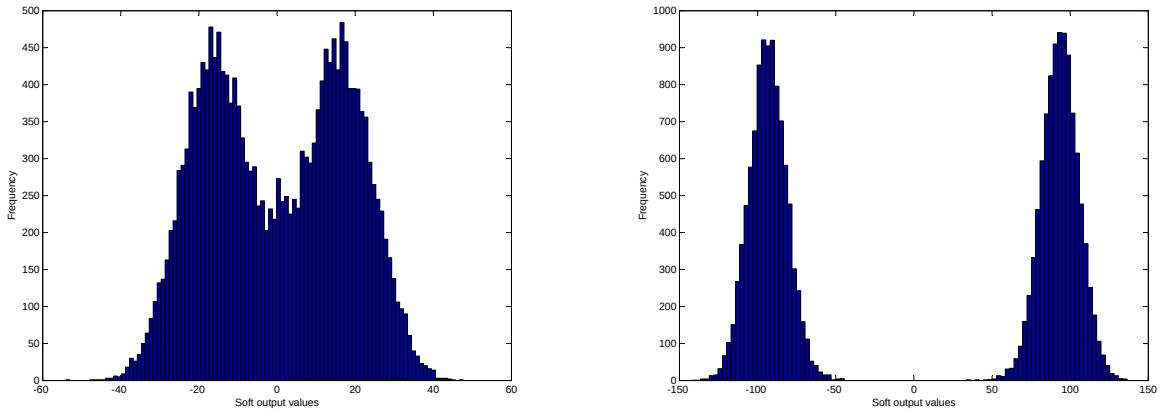
Figure 5.3: Block diagram of the decoder algorithm with Matlab functions written in italic.

The input (**Received signal**) to the decoder is the encoded bits send through an AWGN channel. The parity check bits and systematic bits are extracted from this input string of bits. The systematic bits that is going to the second decoder is interleaved using the same mapping as the interleaver in the encoder. Extraction and interleaving is done by the *demultiplex()* function. To do this it needs to know the **Random interleaver mapping** given by the interleaver mapping **alpha** and if the code is **Punctured or Unpunctured**. The next step is **Scaling**, where every bit of the demultiplexed signal is multiplied with a channel reliability constant. This constant is based on **Channel parameters**, such as E_b/N_0 measurements and fading. As specified in chapter 5.1, E_b/N_0 for the AWGN channel is set to 2 dB. Channel fading based on for example Rayleigh fading, however, is neglected in this case and fading is therefore a constant set to 1, meaning it does not affect the reliability constant. The reliability constant is also based on the rate of the turbo code, which is 1/2 for punctured and 1/3 for unpunctured. This decoder algorithm does not provide dynamic puncturing, meaning that each simulation runs with either rate 1/2 or 1/3, not both or any other rate. Dynamic puncturing is a feature usually provided by turbo coding [11].

The interleaved and deinterleaved bit streams are fed into the appropriate SISO decoder in the *sova0()* decoder function. In figure 5.3 two decoders are depicted even though only one *sova0()* function is used. The reason for this, is to clarify the decoding process of *sova0()* and show that **Extrinsic information** is utilized by both decoders. The extrinsic information is also demapped based on the interleaver mapping **alpha**, just as illustrated by deinterleaving and

interleaving blocks between SISO decoder 1 and 2 in figure 4.4. The two decoders needs to know the **Generator matrix** to derive the number of states used in the encoder. Whereas the trellis diagram needed to measure Hamming distance between the received signal and possible paths through the trellis, is given by the *trellis()* function. This function is therefore also provided with information about the **Generator matrix**.

Finally, after a specified number of iterations through the decoder, the soft-output from the decoders are used to estimate the output based on a **Hard decision**. **Hard decision** means that the algorithm does not distinguish between e.g. a strong zero or a weak zero. The histogram plots in figure 5.4 illustrates the distribution of the soft-output from the SOVA decoders after 5 iterations. In the case of the hard decision decoder every soft-output above zero will be estimated as a "1" and every soft-output below zero as a "0". For the case of $E_b/N_0 = 0.5$ dB in 5.4(a) the variance of the signal is so big that a lot of wrong estimations are bound to happen. The case of $E_b/N_0 = 3.0$ dB in fig. 5.4(b) on the other hand, shows that the smaller variance gives a low/no frequency around zero, resulting in a much better estimation, as expected. The difference in mean value is due to the scaling factor, which is based on the E_b/N_0 of the channel.



(a) (b)

Figure 5.4: (a) Histogram of the soft-output data at a E_b/N_0=0.5 dB test scenario with a frame size of 20.000 bits. (b) Histogram of the soft-output data at a E_b/N_0=3 dB test scenario with a frame size of 20.000 bits.

Knowing the overall code structure of the decoder it is easier to understand how the three functions *demultiplex()*, *trellis()*, and *sova0()* interact. These functions will now be explained more thoroughly. For the *sova0()* function, emphasis will be put on the *for* loop that takes up almost 70 % of the *sova0()* functions execution time.

demultiplex()

This function is best described by figure 5.5, as this function just maps the input bit vector to a two row matrix, where parity check bits and systematic bits are stored in even and odd columns

respectively. The interleaver mapping **alpha** is used to map the systematic bits into the row appointed for the second decoder, so the interleaved systematic bits fits the encoded parity bits. The Matlab code for the *demultiplex()* function is found in appendix C

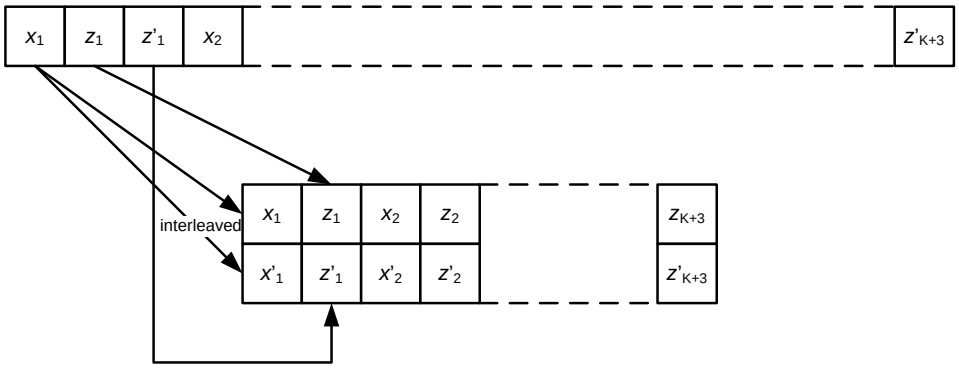


Figure 5.5: Mapping of input bit string to decoder input matrix.

trellis()

This function uses the functions *bin_state()*, *encode_bit()*, and *int_state()* to generate four matrices that is necessary for the *sova0()* function to calculate the Hamming distance between the received signal and possible paths through the trellis. The only parameter needed to construct this trellis is the generator matrix. The Matlab code for the *trellis()* function is found in appendix D. The reader is encouraged to use this appendix with the description below to understand its functionality.

In the *trellis()* function, *bin_state()* converts a vector of integers into a matrix where each row now corresponds to the integers value in binary form. A depiction of this is given in figure 5.6. Each row of this matrix is a state vector (binary combinations of the shift registers in the encoder, see fig. 4.2), which is used by *trellis()* to calculate an input to *encode_bit()*.

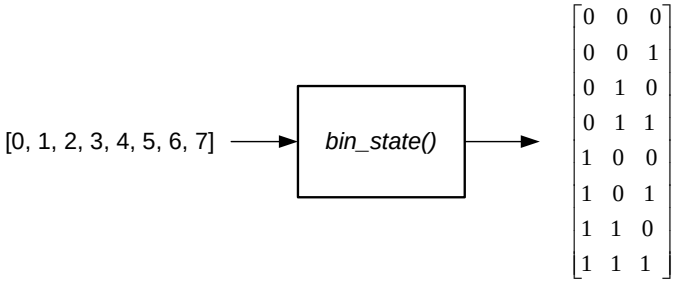


Figure 5.6: *bin_state()* transform a vector with integers to a matrix of corresponding binary values.

encode_bit() uses the input value from *trellis()* (”0” or ”1”), as well as the state vectors from

bin_state(), to construct a matrix consisting of next state vectors. It also calculate outputs resulting from the transition between the state given in *bin_state()* and the next state of *encode_bit()*. Figure 5.7a illustrates the two resulting matrices, **Output** and **Next state** of the *encode_bit()* function, which is based on results from *bin_state()* (**Present state**) and the **Input** value calculated in *trellis()*. Figure 5.7b shows that the result from *encode_bit()* can be viewed upon as one time interval of a trellis diagram. But for *sova()* to use the results of *encode_bit()* for trellis decoding, the results needs to be stored in a next state and a last state matrix corresponding to right and left side of fig. 5.7b respectively. This is explained shortly. The results in figure 5.7 is based on the same convolutional encoder as used in the Viterbi decoding example of appendix A.

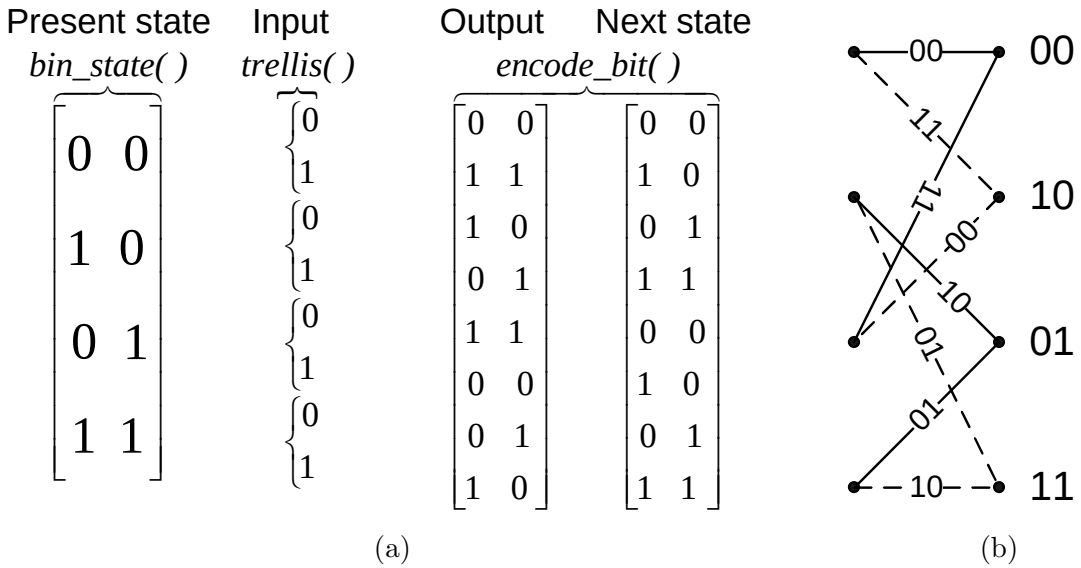


Figure 5.7: (a) The two matrices for output and next state created in *encode_bit()* is based on the present state matrix generated in *bin_state()* and the input bits calculated in *trellis()*. (b) Combining the results listed in (a) gives the information needed to construct one time interval of the trellis diagram.

In *trellis()* the binary output values of *encode_bit()* are converted to ”-1” for binary ”0” and ”1” for binary ”1”. Furthermore the *int_state()* function is then used to convert the *binary states* of *encode_bit()* into a next state matrix of *integer states*. This next state matrix is then used to find two states that leads to one given next state called last states. The outputs resulting from transition between last state and next state, and vice versa is also store in two matrices. To summarize, *trellis()* gives four matrices; next_state, next_out, last_state, and last_out, which describes a time interval in a trellis diagram based on the given generator function. An example of this is the trellis time interval in figure 5.7b.

sova0()

As illustrated in figure 5.3, *sova0()* uses the scaled received bits, the generator matrix and the *trellis()* function as well as extrinsic information to establish a soft-output, which is the most likely sequence of bits. The Matlab code for the *sova0()* function is listed in appendix E and the reader is encouraged to use this listing for ease in comprehending the functionality of *sova0()*.

First of all the generator matrix is used to let the decoder know the number of states, as this is used for iterating through the matrices provided by *trellis()*. Then based on the scaled received bits, *trellis()*, and the extrinsic information two metric values are calculated - one for the binary ”0” (`Mk0`) and one for the binary ”1” (`Mk1`). These two values are compared, and the highest value is stored as well as its corresponding binary value, and the difference between the two (`Mdiff`). This is done for an entire frame block (usually the size of the interleaver), which corresponds to going forward in the trellis diagram while calculating the metric of each path.

Now the decoders are almost ready to start their traceback to find the most likely sequence of bits. But first the most likely state for each decoder has to be found. This can be depicted as standing at the end of the trellis diagram looking at each of the final states, trying to find the one with the highest metric of likelihood. This is easily done for decoder 1, as its trellis is terminated to an all zero state in the encoder, causing the final state in the decoder to be the all zero state as well.

Due to the interleaver it is not possible to terminate the trellis of decoder 2 without some additions to the decoder that is not included in Yufei Wu' code. Instead the decoder compares the metric for all the states at the end of the trellis, yielding the highest metric as the most likely.

With a starting point stated for each decoder, it is now possible to start the traceback through the trellis to find the most likely sequence. A sequence is estimated based on the stored binary values and their corresponding metric values (`Mk0`), and (`Mk1`) as explained above. This estimation is then compared with a competition path based upon possible paths inside a frame of the trellis of length $delta = 5 \cdot K$ as explained in chapter 4.2.1. Should the bits of this competition path and the estimated sequence differ from each other, a log likelihood ratio (LLR) is calculated. This LLR is based upon the difference `Mdiff` between the two values described above; (`Mk0`), and (`Mk1`). The estimated sequence is then converted to ”-1” for binary ”0” and ”1” for binary ”1” and multiplied with the LLR. Resulting in a negative LLR value when the estimated bit value is zero, and a positive LLR when the estimation is one. This is the so called soft-output result of the decoder, but the decoding process does not stop here. As seen in figure 4.4 the extrinsic information of one decoder is fed back to its counterpart. The extrinsic information is found by subtracting the received signal specified for the given decoder, and the *a priori* information fed to the decoder in the beginning. Using this extrinsic information as *a priori* information,

another decoding is undertaken on the same received signal, and the code keeps this up for a number of iterations, specified by the user. After the last iteration a hard decision is made as explained in chapter 5.2.

5.3 Conclusion on SOVA Analysis

As the profiling shows that the last part of *sova0()* - calculating competition path and comparing its bit sequence with the estimated bit sequence - takes up the most computation time by far, this part is chosen for optimization. Results from profiling showed that this part took up 20.66 seconds for 150 iteration. Each iteration consists of 400 bits, giving this part of the SOVA decoder the following throughput:

$$\text{Execution time per bit} = \frac{20.66 \text{ s}}{150 \cdot 400 \text{ bit}} = 344.33 \qquad [\mu\text{s/bit}] \qquad (5.1)$$

$$\text{Throughput} = \frac{1}{344.33 \ \mu\text{s/bit}} = 2.9 \qquad [\text{kbit/s}] \qquad (5.2)$$

Given that this part of the SOVA decoder is the biggest bottleneck and the fact that it takes up 70 % of the entire runtime of the SOVA decoder, the total throughput of the decoder is:

$$\text{Throughput for SOVA} = 0.7 \cdot 2.90 \text{ kbit/s} = 2.03 \qquad [\text{kbit/s}] \qquad (5.3)$$

The throughput based on the total runtime of the SOVA decoder is calculated to establish that the above bottleneck assumption is true:

$$\text{Execution time per bit for SOVA} = \frac{29.714 \text{ s}}{150 \cdot 400 \text{ bit}} = 495.23 \qquad [\mu\text{s/bit}] \qquad (5.4)$$

$$\text{Throughput for SOVA} = \frac{1}{495.23 \ \mu\text{s/bit}} = 2.02 \qquad [\text{kbit/s}] \qquad (5.5)$$

The above assumption is proven, as the two result for SOVA throughput is only 0.01 bit/s apart. This throughput is almost a factor of 1000 lower than the necessary throughput of 2 Mbit/s stated in chapter 3.3. The goal of implementing this algorithm is therefore to achieve an 1000 times acceleration of this SOVA decoder algorithm part. In figure 5.8 the flowchart for this specific part of code is illustrated. Based on this flowchart the code in listing 5.1 is optimized and mapped to the Virtex-5 FPGA as an ASIC implementation.

```
1  for t=1:L_total
2     llr = Infty;
3     for i=0:delta %delta is the traceback search window which should be between 5 and
            9 times K
4        if t+i<L_total+1
5           bit = 1-est(t+i); %inverting the estimated bits 0=1 1=0
6           temp_state = last_state(mlstate(t+i+1), bit+1); %temp_state = the most
                likely last_state based on bit
7           for j=i-1:-1:0 %traceback in the window of size delta
8              bit = prev_bit(temp_state,t+j+1); %competition bit values
9              temp_state = last_state(temp_state, bit+1); %competition path
10          end
11          if bit~=est(t) %if estimated and compition bit deviates
12             llr = min( llr,Mdiff(mlstate(t+i+1), t+i+1) ); % the llr is updated if a
                   lower Mdiff is obtainable
13          end
14       end
15    end
16    L_all(t) = (2*est(t) - 1) * llr; %llr is negative for est(t)=0 and positive for est(t)=1
17 end
```

Listing 5.1: The Matlab code used for calculating a competition path of length delta.

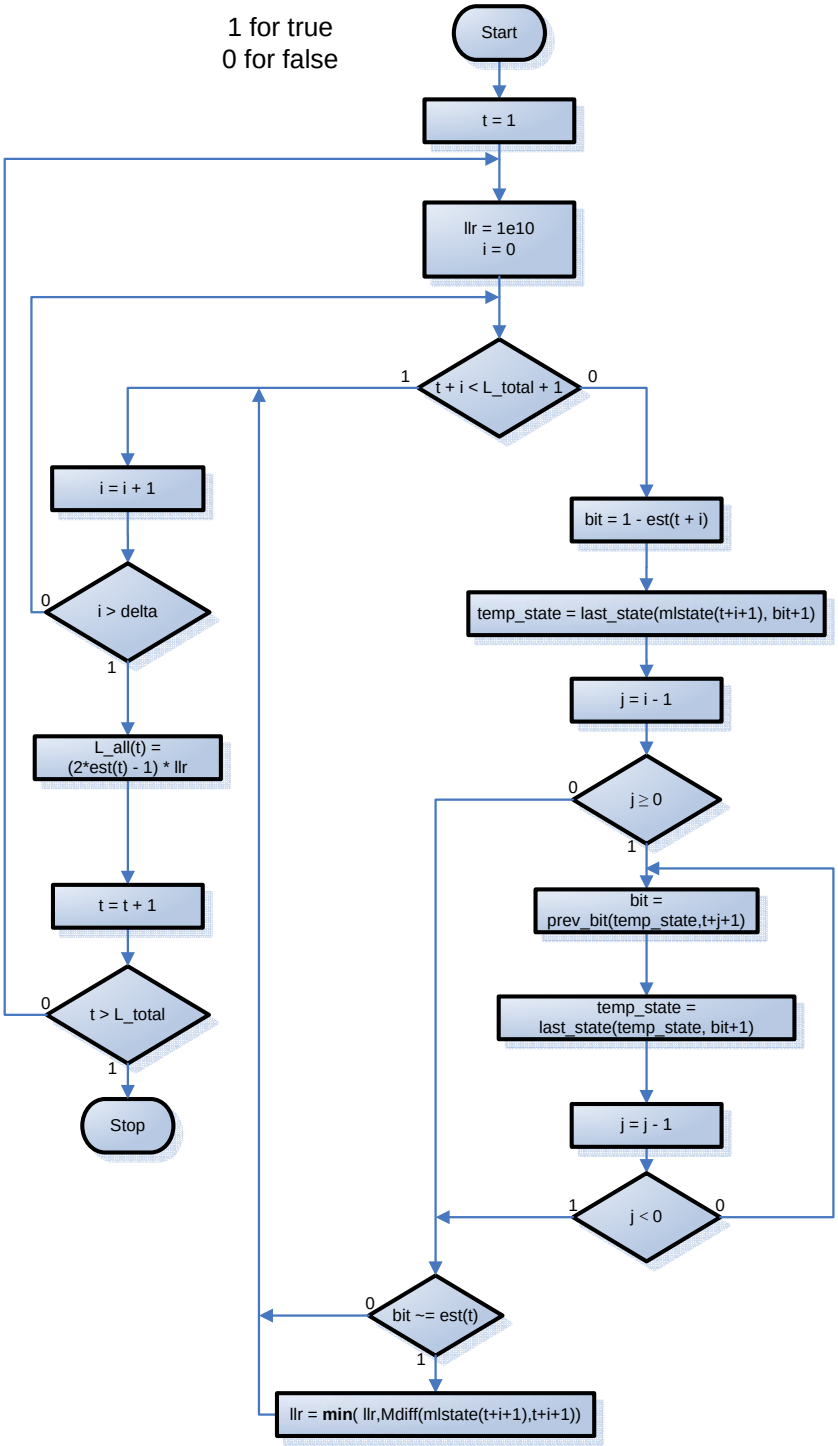


Figure 5.8: Flowchart of the code seen in listing 5.1.

Chapter 6

Algorithmic Design and Optimization

Having determined the algorithm for implementation in the previous chapter, it is now time to look on possible techniques for optimizing the algorithm as it is implemented. This chapter illustrates techniques for optimizing both cost and performance as well as an architectural design of the datapath and its control logic.

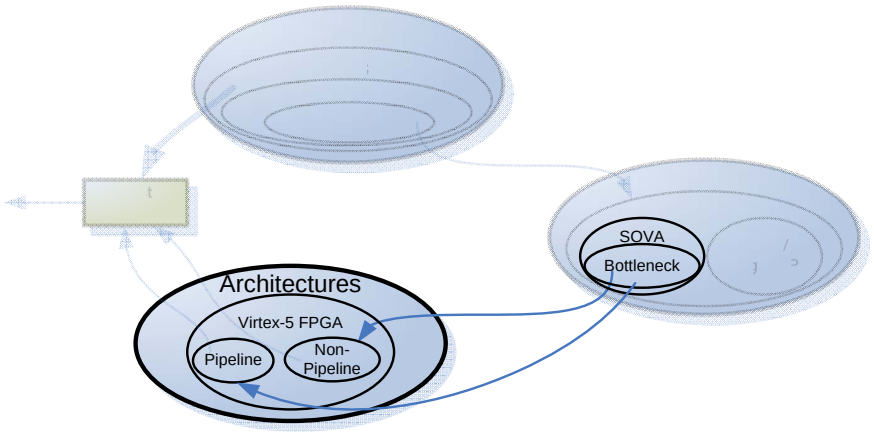
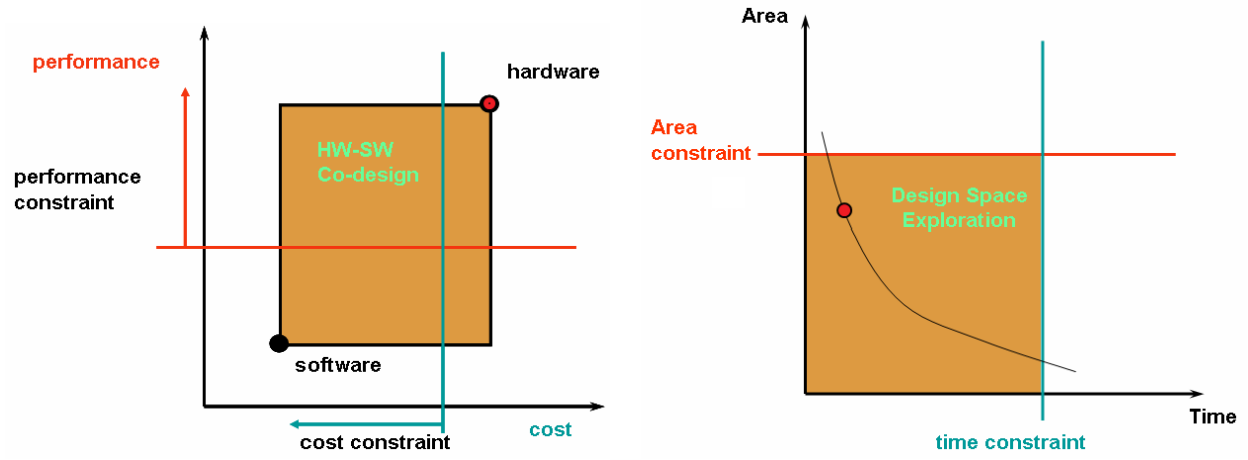


Figure 6.1: Mapping the algorithm to an architecture leads to changes in the algorithm. This chapter ends out with a design optimized for cost and a design optimized for performance.

The first thing to establish when implementing an algorithm is which architecture should be used for the implementation. Should it be a single architecture or a co-design of different architectures. In the case of this project R&S suggested the Virtex-5 FPGA platform, which includes both an PowerPC architecture and an FPGA architecture. This provides the opportunity for a hardware-software co-design, with software implementation in the PowerPC and a hardware implementation on the FPGA. Figure 6.2a shows how a HW/SW co-design affect performance and

cost constraints. As described in chapter 2, R&S design interest for their product R&S®CRTU was located in performance optimization. It is therefore decided to concentrate on a pure hardware implementation, as specified by the red dot in fig. 6.2a. This leads to the best performance, but also the highest cost.



(a) (b)

Figure 6.2: (a) Design space exploration (DSE) for a HW/SW co-design . (b) Area vs. Time DSE, the dot indicating the desired trade-off [23].

Optimizing the pure hardware implementation of the algorithm leads to further DSE, as a reduction in area often directly implies a decrease in performance and vice versa. In the following techniques for optimizing both area and performance are presented and since increasing the performance metric is the most essential task, some choices are made in favor of performance. This leads to the design space of figure 6.2b, where again the red dot illustrates, that even though most effort is put in increasing performance, some area optimization techniques are introduced.

6.1 Finite State Machines

There are several ways of doing optimization when implementing an algorithm, but Daniel D. Gajski presents well explained methods in [24], where optimization by pipelining and merging of functional units, registers, and connections into buses, is based upon ASM charts. Besides pointing out parts for optimization, this method also gives an in depth understanding of the logic and arithmetic units needed for an implementation. Figure 6.3 illustrates a basic register transfer level block diagram for a Mealy machine, which shows some of the blocks that is going to be optimized in this chapter.

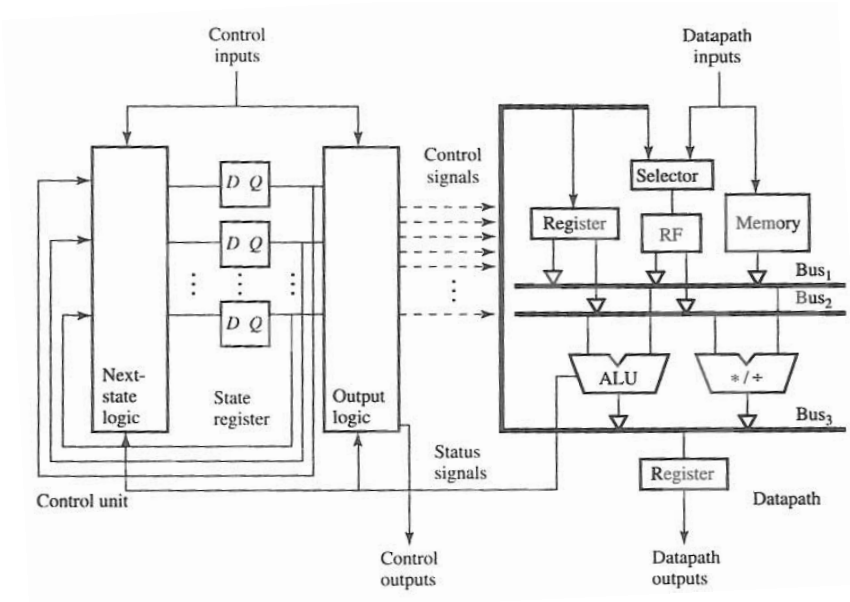


Figure 6.3: Block diagram of a basic register-transfer-level design.

In the following the algorithm from 5.1 is put into two different algorithmic state machine (ASM) charts. The first ASM chart is based on a Moore finite state machine (FSM) and the second is based on a Mealy FSM. This is done to compare if the reduction of clock cycles in the Mealy structure makes up for the increased complexity of its output logic. Later on the selected ASM chart and its corresponding state action table is used for optimizing purposes.

6.1.1 Moore Machine

A Moore machine is a state based machine, meaning that its output only depends on the current state the machine is in. It changes states based on the input to its next state logic, and its state and thereby output logic may be changed at each clock cycle. This means that the output control signals from the output logic of the Moore machine (G) to the datapath is always synchronized with the clock. The structure of a Moore machine control unit is illustrated in figure 6.4. Based

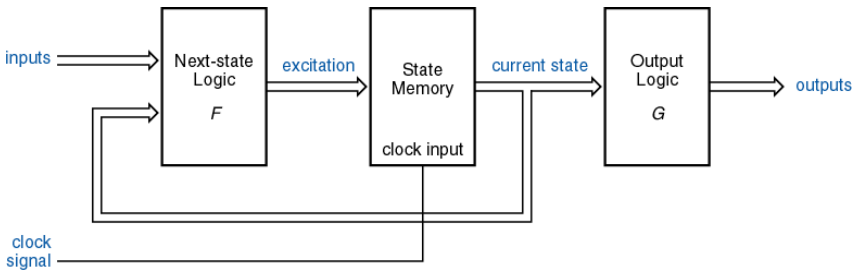


Figure 6.4: Moore machine control unit [4].

on this control unit structure a ASM chart is derived from the flowchart in figure 5.8. The overall structure for this flowchart is kept in the new ASM chart and only some minor modifications are necessary for a state based implementation of the algorithm. The main difference between the flowchart and the ASM chart is the introduction of states. In the Moore ASM chart each square box represents a state, indicating that one clock cycle is necessary to execute the operation in this *state box*. It is, however, possible to group a state box together with a *decision box*, making this grouping an *ASM block*. An entire ASM block also needs one clock cycle to execute the operations within. The changes made from flowchart to ASM chart are listed underneath and the Moore ASM chart is illustrated in figure 6.5:

- A new box and thereby state is implemented for saving `t + i` to a new variable `x`. This is done to save the calculations of `t + i` in following states.
- The variable `x` is compared with `L_total` and a decision is made based on whether `x` $\leq$ `L_total`. As both `x` and `L_total` are integers the `+1` can be saved by using the $\leq$ operation instead.
- State box `x = t + i` and decision box `x` $\leq$ `L_total` is grouped together in an ASM block, meaning that for one clock cycle `x` and `L_total` are saved in two registers and a decision is made based upon their values.
- A false result of the above decision box leads to state s_3 followed by s_4. As the result of s_4 (`temp_state`) depends on the value of `bit` in s_3, the two states needs to be separated.
- A new ASM block is implemented for assigning a value to `j` and to decide if `j` $\geq$ `0`, all done in s_5. Should this be false the algorithm jumps to an empty state (s_9). Is it on the other hand true, then `bit` and `temp_state` are assigned new values in s_6 and s_7.
- In s_8 `j` is decreased by one and in the same ASM block it is checked if `j < 0`. If true the algorithm jumps to empty state s_9 and if false it loops back to s_6.
- s_9 is an empty state due to the principals of the Moore machine. Here a decision box cannot directly follow another decision box. A new state is necessary to setup the correct control signals in the output logic of the Moore machine before the new decision can be made.
- Based on the comparison of `bit` and `est(t)` the ASM chart either jumps to s_{10} or s_{11}, where in s_{10} a addition is saved by using the variable `x` instead of `t+i`.
- An ASM block is created around state box s_{11} and the following decision box. Remembering once again that one decision box cannot be followed by another decision box, it is clear that s_2 has another advantage besides creating `x`. It also ensures the separation of the `i > delta` and `x` $\leq$ `L_total` decision boxes.

- In s_{12} the variable `L_all(t)` is computed and this is followed by one last ASM block, for incrementing `t` and checking if the stop conditions are met.

From the ASM chart it is possible to derive a `state action tabel`, which is later on used to find the next state equations for the control unit. With these next state equations it is possible to set up a control unit for implementation.

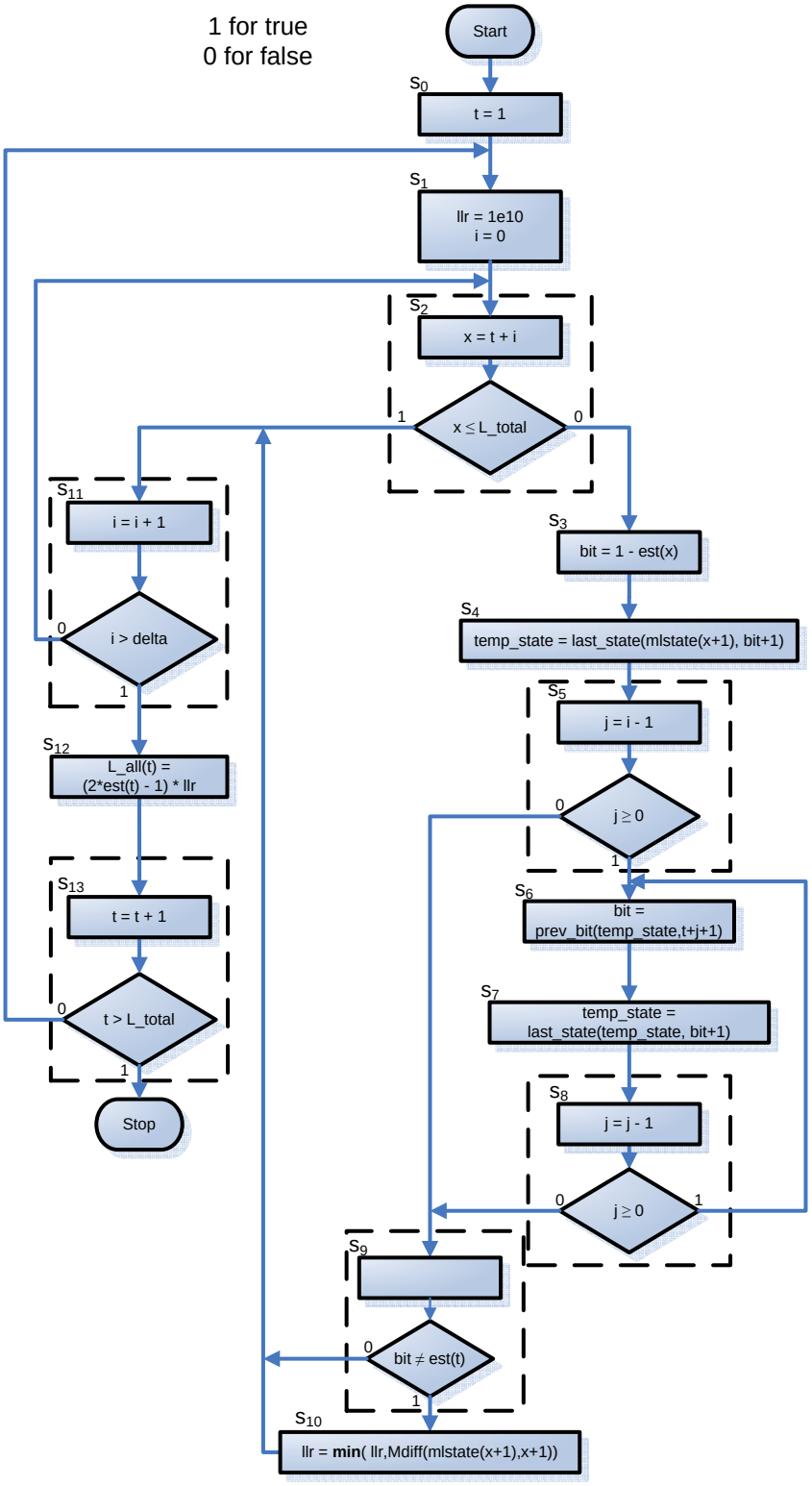


Figure 6.5: State based ASM chart. This chart is based on a Moore finite state machine with a datapath (FSMD).

Present State					Next state		Datapath actions	
Q_3	Q_2	Q_1	Q_0	Name	Condition	State	Condition	Operations
0	0	0	0	s_0		s_1		t = 1
0	0	0	1	s_1		s_2		llr = 1e10 , i = 0
0	0	1	0	s_2	x ≤ L_total x > L_total	s_3 s_{11}		t + i = x
0	0	1	1	s_3		s_4		bit = 1 - est(x)
0	1	0	0	s_4		s_5		temp_state = last_state(mlstate(x+1), bit+1)
0	1	0	1	s_5	j ≥ 0 j < 0	s_6 s_9		j = i - 1
0	1	1	0	s_6		s_7		bit = prev_bit(temp_state,t+j+1)
0	1	1	1	s_7		s_8		temp_state = last_state(temp_state, bit+1)
1	0	0	0	s_8	j ≥ 0 j < 0	s_9 s_6		j = j - 1
1	0	0	1	s_9	bit ≠ est(t) bit = est(t)	s_{10} s_{11}		
1	0	1	0	s_{10}		s_{11}		llr = min(llr,Mdiff(mlstate(x+1),x+1))
1	0	1	1	s_{11}	i > delta i ≤ delta	s_{12} s_2		i = i + 1
1	1	0	0	s_{12}		s_{13}		L_all(t) = (2*est(t) - 1) * llr
1	1	0	1	s_{13}	t > L_total t ≤ L_total	Stop s_1		t = t + 1

Table 6.1: State action tabel for the state based (Moore) ASM chart in figure 6.5. Note that after state s_2 where `x = t + i`, the variable `x` is used throughout the table instead of t + i which differs from the algorithm in 5.1.

Based solely on this state action table the next state equations are derived for a Moore FSM implementation of the algorithm. The state action table also shows that 4-bit state registers are necessary for storing these 13 states as well as how the encoding of these 13 states should be done. This is shown by the next state equations derived below. Note that these next state equations uses the same decision statements as in the ASM chart as well as their opposite. As a false argument equaled a 0 and a true argument a 1 in the ASM chart, the opposite arguments of these next state equations should be seen as a 0 and vice versa when constructing the control unit. Or said more specific for the case of the next state condition $\texttt{x} \leq \texttt{L\_total}$; $\texttt{x} \leq \texttt{L\_total} = \overline{\texttt{x > L\_total}}$ and $\overline{\texttt{x} \leq \texttt{L\_total}} = \texttt{x > L\_total}$.

$$
\begin{aligned}
D_3 &= Q_3(next) = s_2(\texttt{x > L\_total}) + s_5(\texttt{j < 0}) + s_7 + s_8(\texttt{j} \geq \texttt{0}) + s_9 + s_{10}... \\
&\quad + s_{11}(\texttt{i > delta}) + s_{12} \\
D_3 &= Q_3(next) = Q_3'Q_2'Q_1Q_0'(\texttt{x > L\_total}) + Q_3'Q_2Q_1'Q_0(\texttt{j < 0}) + Q_3'Q_2Q_1Q_0... \\
&\quad + Q_3Q_2'Q_1'Q_0'(\texttt{j} \geq \texttt{0}) + Q_3Q_2'Q_1'Q_0 + Q_3Q_2'Q_1Q_0' + Q_3Q_2'Q_1Q_0(\texttt{i > delta}) + Q_3Q_2Q_1'Q_0' \\
D_2 &= Q_2(next) = s_3 + s_4 + s_5(\texttt{j} \geq \texttt{0}) + s_6 + s_8(\texttt{j < 0}) + s_{11}(\texttt{i > delta}) + s_{12} \\
D_2 &= Q_2(next) = Q_3'Q_2'Q_1Q_0 + Q_3'Q_2Q_1'Q_0' + Q_3'Q_2Q_1'Q_0(\texttt{j} \geq \texttt{0}) + Q_3'Q_2Q_1Q_0'... \\
&\quad + Q_3Q_2'Q_1'Q_0'(\texttt{j < 0}) + Q_3Q_2'Q_1Q_0(\texttt{i > delta}) + Q_3Q_2Q_1'Q_0' \\
D_1 &= Q_1(next) = s_1 + s_2 + s_5(\texttt{j} \geq \texttt{0}) + s_6 + s_8(\texttt{j < 0}) + s_9 + s_{10} + s_{11}(\texttt{i} \leq \texttt{delta}) + s_{12} \qquad (6.1) \\
D_1 &= Q_1(next) = Q_3'Q_2'Q_1'Q_0 + Q_3'Q_2'Q_1Q_0' + Q_3'Q_2Q_1'Q_0(\texttt{j} \geq \texttt{0}) + Q_3'Q_2Q_1Q_0'... \\
&\quad + Q_3Q_2'Q_1'Q_0'(\texttt{j < 0}) + Q_3Q_2'Q_1'Q_0 + Q_3Q_2'Q_1Q_0' + Q_3Q_2'Q_1Q_0(\texttt{i} \leq \texttt{delta}) + Q_3Q_2Q_1'Q_0' \\
D_0 &= Q_0(next) = s_0 + s_2 + s_4 + s_5(\texttt{j < 0}) + s_6 + s_8(\texttt{j} \geq \texttt{0}) + s_9(\texttt{bit = est(t)}) + s_{10}... \\
&\quad + s_{12} + s_{13}(\texttt{t} \leq \texttt{L\_total}) \\
D_0 &= Q_0(next) = Q_3'Q_2'Q_1'Q_0' + Q_3'Q_2'Q_1Q_0' + Q_3'Q_2Q_1'Q_0' + Q_3'Q_2Q_1'Q_0(\texttt{j < 0})... \\
&\quad + Q_3'Q_2Q_1Q_0' + Q_3Q_2'Q_1'Q_0'(\texttt{j} \geq \texttt{0}) + Q_3Q_2'Q_1'Q_0(\texttt{bit = est(t)}) + Q_3Q_2'Q_1Q_0'... \\
&\quad + Q_3Q_2Q_1'Q_0' + Q_3Q_2Q_1'Q_0(\texttt{t} \leq \texttt{L\_total})
\end{aligned}
$$

Before implementing the control unit, the next state equations above could be simplified with some algebraic manipulation, which would reduce the size and number of gates necessary for the implementation of the control unit. One way of doing this by hand is using Karnaugh maps and in the following a description of this procedure is explained via an example. The the theory behind Karnaugh maps is based on [25]. These maps are based on the truth table for each next state equation, mapping the output from the next state equations into minterm boxes (see fig. 6.6), so they corresponds to the binary values of the next state equation variables. A four variable Karnaugh map example for next state equation D_3 is illustrated in fig. 6.6. Here the next state conditions have been ignored to simplify the illustration. The truth table on which this Karnaugh map is based can be found in appendix F, fig. F.1. Truth tables for the remaining states (D_2, D_1, and D_0) is also found in appendix F.

In the Karnaugh map of fig. 6.6 the next state equation of D_3 without it next state conditions

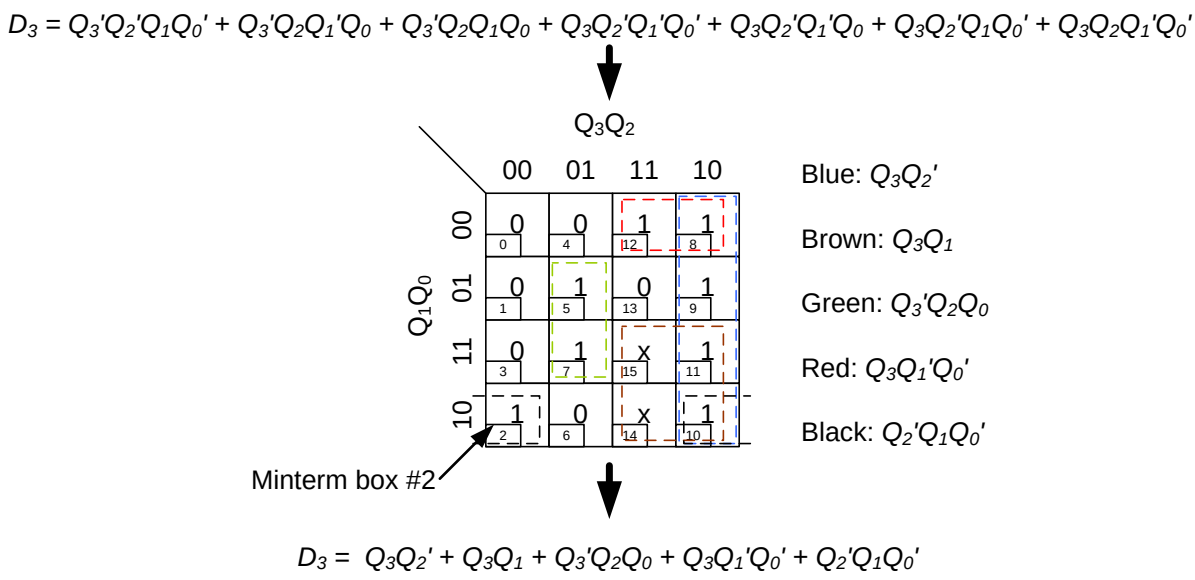


Figure 6.6: Modified D_3 next state equation simplified by Karnaugh map.

is reduced by the use of Karnaugh mapping. This is done by grouping the minterm boxes containing 1's and "don't cares" (labeled by x) together. These groupings have to be rectangular and their area has to be in the power of two. With these rules in mind, the 1's and x's are grouped into as big rectangles as possible. Note that minterm 2 and 10 are grouped together. This is done as it is possible to wrap around edges of the Karnaugh map. Another possible wrap around grouping would be of minterms 8, 10, 12, and 14. With the 5 groupings in fig. 6.6 it is fairly easy to see how D_3 can be reduced. Taking the blue box as an example, shows that Q_3 and Q_2 are constant for all 1's in this box, while Q_1 and Q_0 are both 0 and 1 for this box. This means that Q_1 and Q_0 can be excluded and as Q_3 is always 1 and Q_2 is always 0 for the blue box, the boolean term for this box would be Q_3Q_2'. Same method is used for the remaining boxes and the result is a reduction of D_3 as indicated in the bottom of fig. 6.6. Should this next state equation be implemented as natural binary encoding, the reduction provided by Karnaugh mapping would save 2 AND gates and only 13 AND gate inputs would be needed compared to 28 of the modified D_3 equation. Finally two OR gate inputs is also saved by this reduction.

Constructing a Karnaugh map for the next state equations including all variables would produce a map with 2^6 minterms for D_2 and D_1, while 2^7 minterms are necessary for D_3 and D_0. This would a tedious and very time consuming tax to do by hand, so instead a Quine-McCluskey (QM) algorithm is used. The QM algorithm works similar to the Karnaugh maps where the main difference is that, where Karnaugh maps benefits pattern recognition properties of the human mind, the QM algorithm uses systematic tables for deriving simplified expressions. The tabular method of the QM algorithm makes it suitable for a computer implementation. One of

Gates used	AND gates				OR gates					Total number	
# of inputs pr gate	2	3	4	5	6	7	8	9	10	of gates	of inputs
Unmodified	0	0	20	14	0	1	1	1	1	38	184
QM minimized	1	7	18	1	2	1	1	0	0	27	127

Table 6.2: Comparison between unmodified and Quine-McCluskey minimized next state equations.

such was developed by Antonio Costa and can be found in [26]. Using this code the next state equations was reduced to the following:

$$
\begin{aligned}
D_3 &= Q_3(next) = Q_2'Q_1Q_0'(\texttt{x > L\_total}) + Q_3'Q_2Q_0(\texttt{j < 0}) + Q_3'Q_2Q_1Q_0 \ldots \\
&\quad + Q_3Q_2'Q_0'(\texttt{j} \geq \texttt{0}) + Q_3Q_2'Q_1'Q_0 + Q_3Q_2'Q_1Q_0' + Q_3Q_2'Q_1(\texttt{i > delta}) + Q_3Q_2Q_1'Q_0' \\
D_2 &= Q_2(next) = Q_3'Q_2Q_0' + Q_2Q_1'Q_0' + Q_2'Q_1Q_0(\texttt{i > delta}) + Q_3'Q_2Q_1'(\texttt{j} \geq \texttt{0}) \ldots \\
&\quad + Q_3'Q_2'Q_1Q_0 + Q_3Q_1'Q_0'(\texttt{j < 0}) \\
D_1 &= Q_1(next) = Q_2'Q_1'Q_0 + Q_3'Q_1Q_0' + Q_3'Q_1'Q_0(\texttt{j < 0}) + Q_3Q_2Q_1'Q_0' \ldots \\
&\quad + Q_3Q_2'Q_1(\texttt{j < 0}) + Q_3Q_2'Q_0(\texttt{i} \leq \texttt{delta}) \\
D_0 &= Q_0(next) = Q_3'Q_0' + Q_2'Q_0'(\texttt{j < 0}) + Q_2'Q_1Q_0' + Q_2Q_1'Q_0' + Q_3Q_2Q_1'(\texttt{t} \leq \texttt{L\_total}) \ldots \\
&\quad + Q_3'Q_2Q_1'(\texttt{j < 0}) + Q_3Q_2'Q_1'Q_0(\texttt{bit = est(t)})
\end{aligned}
\tag{6.2}
$$

Comparing these next state equations with the ones first derived in eq. 6.1, it is possible to see how much combinational logic is saved by using the Quine-McCluskey algorithm. The results are listed in table 6.2 and from these it is possible to derive that 11 AND gates and 57 inputs are saved by QM minimization. In the next section it is explained how Karnaugh maps are also a good aid in detecting and removing race hazards. This is not a problem in a Moore FSM, as the control outputs are based solely on the present state of the state register, which is controlled by a clock signal. This ensures that the control signals from the output logic are based on the present state and synchronized with the clock. The same cannot be said about the Mealy Machine, where special attention to race conditions may be necessary [4].

6.1.2 Mealy Machine

Using a Mealy structure gives the capability to produce several control outputs for each state. As long as the information for the present state is maintained, the datapath may generate inputs to the Mealy machines output logic. This will cause the control output to change in-between present and next state, which means that the control outputs can be asynchronous with respect to the clock signal. These intermediate changes in the control output via the output logic between states are called conditional outputs, as the control outputs are based on the condition of the datapath. Utilizing this FSM structure the designer is able to reduce the amount of states with the cost of an increased complexity of the output logic. A Mealy machine structure

is illustrated in figure 6.7 which only differs from the Moore machine in 6.4 in the input fed directly to the output logic.

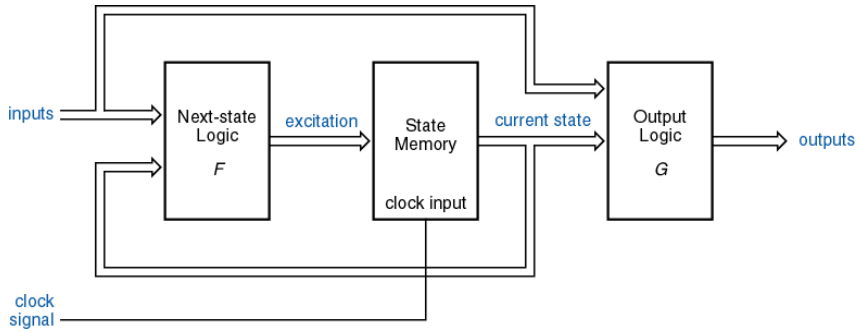


Figure 6.7: Control unit for a Mealy machine. Notice that inputs are not only fed to the next state logic, but the output logic as well [4].

An ASM chart utilizing the conditional control outputs of the Mealy machine is derived based on the Moore ASM chart. The considerations done in the construction of this ASM chart is listed below and the ASM chart itself is illustrated in figure 6.8.

- The first two states resembles the Moore ASM chart, as the loop back to s_0 and s_1 constrains the modification of this part of the ASM chart.
- The first ASM block is encountered at s_2 and differs from the Moore ASM chart by including two conditional boxes. This is possible as the information for the present state is kept and each conditional output can be used for generating a new control output. This modification introduces a reduction of states by two.
- An empty state is introduced after s_3 due to the removal of a decision box and the loop back of s_5. Besides the benefit of removing one decision block, this modification also reduces the number of states by one compared to the Moore ASM chart.
- The third ASM block (starting with s_6) resembles the Moore ASM chart, but now it includes the **min** function removing yet another state.
- The last states of the Mealy ASM chart is the same as for the Moore ASM chart. It is necessary to keep s_8 a state of its own because this state creates the final output of the datapath. This means the state cannot be a conditional output of the ASM block prior to this state.

The Mealy structure allows the algorithm to be executed in 9 clock cycles compared to the 13 clock cycles of the Moore structure. The Mealy ASM chart is found in figure 6.8 and its corresponding state action table is located in table 6.3.

The state action table for the input based Mealy implementation is a bit more complex than the

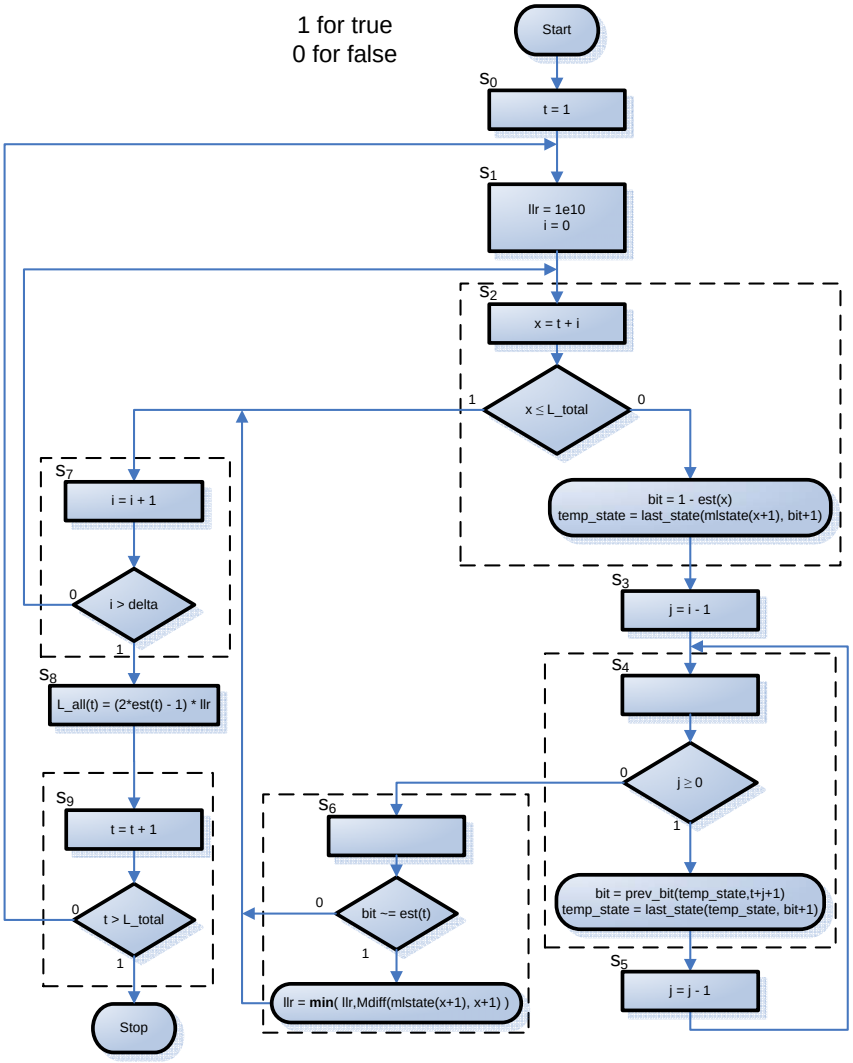


Figure 6.8: Mealy ASM chart. The input based output (conditional boxes) gives a reduction of states by 4 compared to the Moore ASM chart.

one for Moore. In this table, conditions are added to the operations of the datapath, denoting that the condition of the decision boxes not only specify the next state, but also the operation that is executed in the present state. This increases the complexity of the output logic compared to the output logic of a Moore FSM [24]. As were the case for state based next state equations, it is also possible to derive next state equations for the input based Mealy machine, based solely on the state action table. The next state logic can be reduced a bit by noticing that Q_3 is only '1' for s_8 and s_9, making it possible to define the states as follows: $s_0 = Q_2'Q_1'Q_0'$, $s_1 = Q_2'Q_1'Q_0$, $s_2 = Q_2'Q_1Q_0'$, $s_3 = Q_2'Q_1Q_0$, $s_4 = Q_2Q_1'Q_0'$, $s_5 = Q_2Q_1'Q_0$, $s_6 = Q_2Q_1Q_0'$, $s_7 = Q_2Q_1Q_0$, $s_8 = Q_3Q_0$', and $s_9 = Q_3Q_0$. Using this in the next state equations for the Mealy machine gives the

Present State					Next state		Datapath actions	
Q_3	Q_2	Q_1	Q_0	Name	Condition	State	Condition	Operations
0	0	0	0	s_0		s_1		t = 1
0	0	0	1	s_1		s_2		llr = 1e10 , i = 0
0	0	1	0	s_2				x = t + i
					x ≤ L_total	s_3	x ≤ L_total	bit = 1 - est(x) temp_state = last_state(mlstate(x+1), bit+1)
					x > L_total	s_7		
0	0	1	1	s_3		s_4		j = i - 1
0	1	0	0	s_4	j ≥ 0	s_5	j ≥ 0	temp_state = last_state(temp_state, bit+1) bit = prev_bit(temp_state,t+j+1)
					j < 0	s_6		
0	1	0	1	s_5		s_4		j = j - 1
0	1	1	0	s_6		s_7	bit ≠ est(t)	llr = min(llr,Mdiff(mlstate(x+1),x+1))
0	1	1	1	s_7				i = i + 1
					i > delta	s_2		
					i ≤ delta	s_8		
1	0	0	0	s_8		s_9		L_all(t) = (2*est(t) - 1) * llr
1	0	0	1	s_9	t > L_total t ≤ L_total	Stop s_1		t = t + 1

Table 6.3: State action tabel for the input based (Mealy) ASM chart in figure 6.8. Note that by the use of Mealy FSM the state action table is reduced to 9 states compared to the 13 of Moore FSM.

following:

$$D_3 = Q_3(next) = s_7(i \leq delta) + s_8$$
$$D_3 = Q_3(next) = Q_2Q_1Q_0(i \leq delta) + Q_3Q_0'$$
$$D_2 = Q_2(next) = s_2(x > L\_total) + s_3 + s_4 + s_5 + s_6$$
$$D_2 = Q_2(next) = Q_2'Q_1Q_0'(x > L\_total) + Q_2'Q_1Q_0 + Q_2Q_1'Q_0' + Q_2Q_1'Q_0 + Q_2Q_1Q_0'$$
$$D_1 = Q_1(next) = s_1 + s_2 + s_4(j < 0) + s_6$$
$$D_1 = Q_1(next) = Q_2'Q_1'Q_0 + Q_2'Q_1Q_0' + Q_2Q_1'Q_0'(j < 0) + Q_2Q_1Q_0'$$
$$D_0 = Q_0(next) = s_0 + s_2 + s_4(j \geq 0) + s_6 + s_8 + s_9(t \leq L\_total)$$
$$D_0 = Q_0(next) = Q_2'Q_1'Q_0' + Q_2'Q_1Q_0' + Q_2Q_1'Q_0'(j \geq 0) + Q_2Q_1Q_0' + Q_3Q_0' + Q_3Q_0(t \leq L\_total)$$

Further reduction of the next state equations can be achieved by algebraic manipulations through the aid of Karnaugh maps or the QM algorithm. But more importantly the Karnaugh maps and the QM algorithm can also aid in the detection and elimination of race hazards [27].

Race hazards occur in combinational logic circuits because of different signal paths with different propagation delays, resulting in unwanted output "glitches". Elimination of race hazards becomes an issue when dealing with Mealy machines, as status signals are fed from the datapath directly to the output logic. Had these signals, as in Moore, been fed through the next state logic and state registers, the state registers would have worked as a latch. In the synchronous Moore system these state registers look at their input signals for each clock signal where all signals have reached steady state, which removes the issue of race hazards [25].

Looking at the Karnaugh map example in fig. 6.6 of section 6.1.1 as a map for a Mealy implementation, possible race hazards are detected as a transition between adjacent, disjoint groupings of this map. This means that a possible race hazard exist between minterm 8 and 10. Here $Q_3Q_2Q_0$ are static (1 0 0) while a transition of Q_1 should still produce a consistent 1 output. There is, however, no term that ensures this, therefore a new term should be added to the reduced next state equation D_3. The method for this is just the same as for creating the other terms for the boxes in the Karnaugh map. In this case $Q_3Q_2Q_0$ where static, while Q_1 could be both 1 or 0, resulting in the term $Q_3Q_2'Q_0'$ which should be added to D_3 in fig. 6.6. Another potential race hazard exist between minterm 2 and 8. The race hazards removed by this method are called *static-1* hazards.[25]

6.1.3 Conclusion on FSM

The Moore FSM needs 13 states to execute the algorithm in question, whereas a Mealy FSM can execute the same algorithm in 9 states. However, due to the more complex design of the Mealy output logic and the special care needed for eliminating the race hazards from the next state

equations, the further work will revolve around optimizing the Moore structure. A concetual finite state machine with datapath (FSMD) design is illustrated in fig. 6.9. In this figure f denotes the next state equations from 6.2, which derives the next state based on control inputs (I_C), datapath status signals (I_D) and the current state (S) of the state register. $h : S \rightarrow O$ denotes the mapping performed in output logic and as it is a Moore FSM, the output (O) is solely based on the current state (S). At this point the next state logic has been specified,

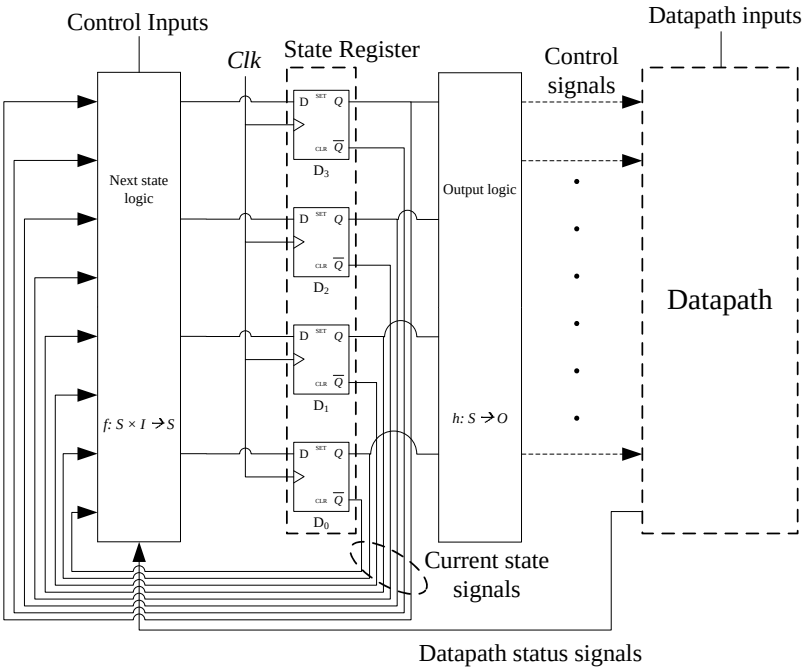


Figure 6.9: State based (Moore) FSMD design necessary for implementing the Moore adapted algorithm of fig. 6.5. Based on fig. 6.18 and 8.1 in [24].

but it is necessary to design the datapath before the output logic can be determined. Output logic controls the selection of operands from memory elements, the operation of functional units and where a result should be stored. It is therefore essential to establish parameters of the datapath, such as number of functional units, their operations, interconnects in the datapath and memory elements, before it is possible to design the output logic. The design of the datapath is undertaken in the following section and throughout this design phase, several optimization techniques are used for reducing cost and to boost the performance of the datapath. Before optimization techniques are utilized in the design of the Moore adapted algorithm, the necessary word length and size of memory elements is established and one of three memory elements is chosen for each variable.

6.2 Data Structures

The flow chart in fig. 5.8 shows that 15 variables are used throughout the entire algorithm. The data structure of these variables are mainly based on the specifications stated in [12] for the turbo coder in EGPRS-2. The variables `t`, `x`, and `L_total` are based on the size of the turbo coders internal interleaver block, where `L_total` is the specific size of this interleaver. Remember that `x = t + i` and that the maximum value of `i` is `delta`. In [12] the maximum block size for any *packet data block type* to be interleaved, is that of DBS-10. This block type interleaves 2500 bits at a time. Meaning that for this maximum block size `L_total` would be assigned the value 2501 (the additional 1 is due to constraints of indexing in Matlab), `t` would be any integer between 1 and 2501 and `x` would be any integer between 1 and 2501 + `delta` $\geq 5 \cdot K$ (see p. 27 for further explanation). With the turbo encoder constraint length $K = 4$ in EGPRS-2 `x` would have a maximum value of 2521 or a little bit above. A word length of of 12 bit is therefore necessary to represent the values of `t`, `x`, and `L_total` if 2's complement arithmetic is used. Furthermore since `delta` $\geq 5 \cdot 4$ it is established that `i` and `delta` should have a word length of 6 bits. With a 12-bit word length and 2's complement arithmetic it is possible to represent integers from $[-2^{12-1}:2^{12-1} - 1] = [-4096:4095]$ and a 6-bit word length from [-32:31].

The variables `bit` and `est()` can only take the values '1' and '0', meaning that these variables can be seen as binary values. The memory element containing `est()` should be able to save `L_total` number of values (2500 in when DBS-10 is used), whereas the memory element containing `bit`, should only be able to save one value. The memory element for `mlstate()` is also based on `L_total`, but its word length is based on the values of `last_state()` as seen in listing E.1 of *sova0.m* line 71, which is found in appendix E. The values of `last_state()` is based on the number of states in the trellis for the turbo code, which is given by 2^{K-1}. As mentioned earlier $K = 4$ for the EGPRS-2 turbo coder (states from 0 to 7 or 1-8 in Matlab), so the memory elements for `mlstate()` and `last_state()` should have a word length of 4 bits as 2's complement arithmetic is used. Investigating the Matlab code for the states where `bit` is used as an operand, s_4 and s_5, the following turns out. `last_state`, in which `bit` is used for indexing, is a matrix with row length given by the number of constituent encoders and a column length given by 2^{K-1}. This means that for the EGPRS-2 turbo coding, `last_state` would be an 8-by-2 matrix, as EGPRS-2 consists of two constituent encoders with a constraint length of $K = 4$. `last_state` memory element should therefore have a word length of 4 bits and a length of 16 needing 4 bit for addressing. As it is the values of `mlstate()` and `bit` that decides the value of `last_state`, it would be preferable to use these as indirect adressing of `last_state`. This could be done by letting the value of `mlstate()` be the three LSBs and the value of `bit` the MSB for the memory element containing `last_state`.

The variable `temp_state` takes on the same values as `last_state` and its register should therefore also have a word length of 4 bits, but it only needs to store one value at a time. `j` is given

by `i-1` and its register should therefore have the same 6-bit word length as `i`. Only one value is stored for `j`.

Listing E.1 in appendix E shows in line 56 and 60 that `prev_bit()` can take on the values "1" and "0" and a word length of 1 bit is therefore sufficient. As indicated in s_6 of 6.5 `prev_bit()` is indexed by `temp_state` and `t+j+1`. This means `prev_bit()` is a matrix of the size 2^{K-1}x(interleaver block length). In this case `prev_bit()` could be as big as an 8x2500 matrix which means the memory element storing `prev_bit()` should be able to contain 20000 values. This means that the memory element for `prev_bit()` should have a word length of 1 bit and needs 15 bits for addressing. `Mdiff()` uses `mlstate()` and `x+1` for indexing and as these values resembles those of `temp_state` and `t+j+1`, so should `Mdiff()` be able to contain 20000 values and therefore use 15 bits for addressing. Establishing the word length of `Mdiff()` requires some considerations and Matlab simulations, which is done in the following.

`L_all()` can take on values ranging from `-llr` to `llr` and should be able to contain up to 2500 values. In the flowchart in fig. 5.8 `llr` is set to 1e10 and an implementation would require a 34 bit register to represent such high a value. It is therefore investigated if this value can be reduced to lower the word length of the memory elements for `llr` and `L_all()`. The variable `llr` is mainly used for comparison with `Mdiff()` in s_{10} and its initial value should always be bigger than the maximum value of `Mdiff`. Simulation of the Matlab code shows that `Mdiff()` is only affected by the SNR (E_b/N_0) of the received signal and the encoders transfer function. As the transfer function for EGPRS-2 is constant, simulations for different SNR was undertaken. The received signal strength for EGPRS-2 is in the range from -115 dBm to -38 dBm, leading to a maximum SNR of 77 dB [11]. The plot illustrated in figure 6.10 shows the maximum value for `Mdiff()` at increasing SNR. From this plot it is possible to derive that the maximum value for `Mdiff()` at a SNR of 77 dB is around $1.003 \cdot 10^8$. The initial value for `llr` should therefore be equal or exceed this value. Using 2's complement fixed point arithmetic the minimum word length for representing a value of $1.003 \cdot 10^8$ would be 27 bit, making it possible to represent values from [-2^{27}:2^{27}-1] = [-134.217.728:134.217.727]. The word length will increase if a fraction part is needed for representing more precise values of `Mdiff()` and `llr`. An example of a 2's complement arithmetic representation is illustrated in figure 6.11a. It is possible to bring down the size of the memory element containing these variables by use of floating point arithmetic instead. Floating point arithmetic consist of a mantissa and an exponent. The mantissa is a fraction part which is multiplied with an exponent to form the decimal value [28]. An example of how the floating point arithmetic can be used to represent a value of at least $1.003 \cdot 10^8$ is illustrated in figure 6.11b.

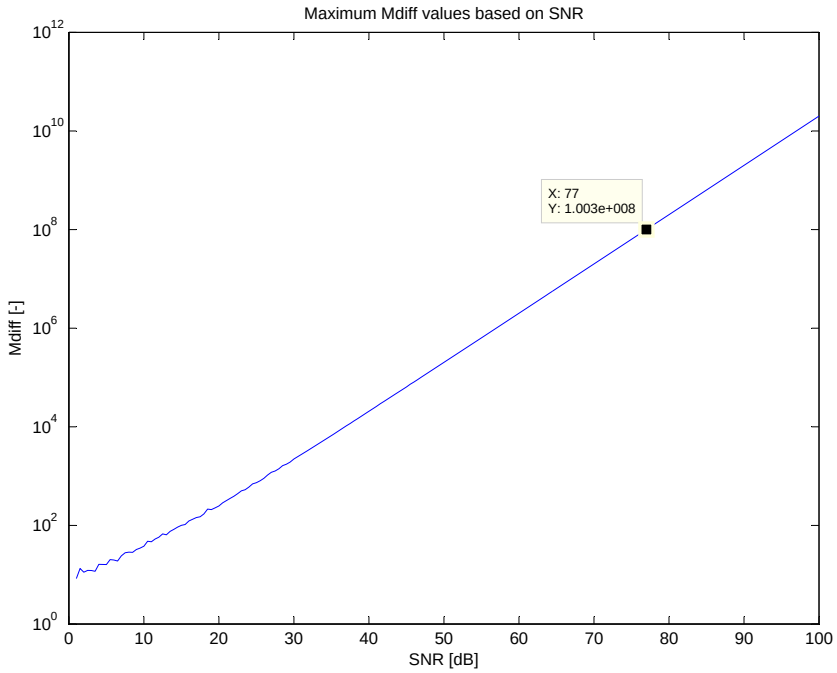


Figure 6.10: Mdiff increases exponentially as E_b/N_0 is increased, it is therefore important to establish the dynamic range of the received input as to determine the size of the memory elements containing the values for `llr`, `Mdiff()` and `L_all()`.

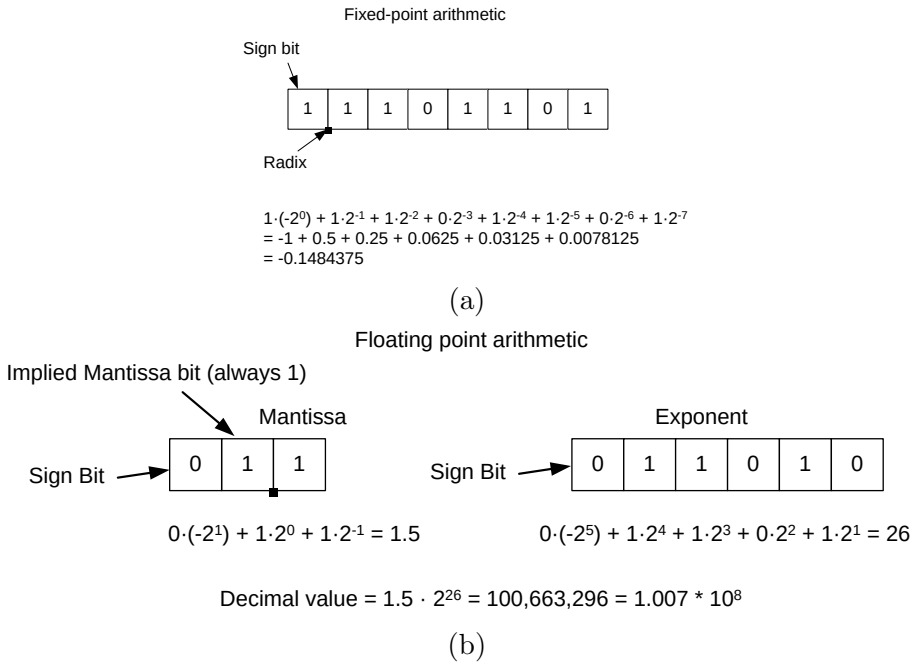
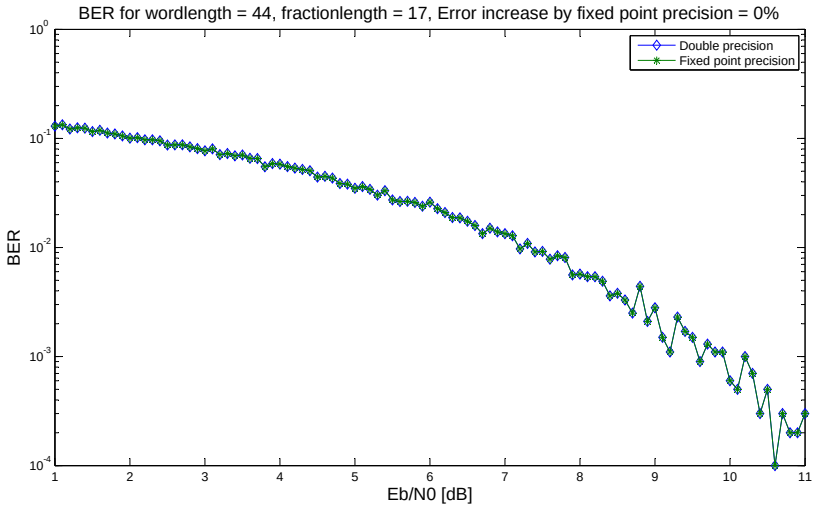


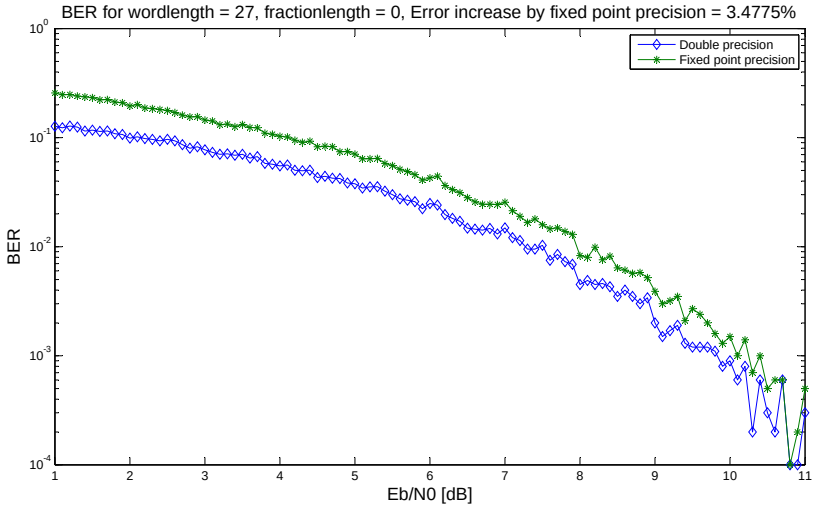
Figure 6.11: (a) 8-bit fixed point example with 2's complement (note the - sign in the MSB). The radix is moved to the right for representing larger values, which however, leads to a reduction of precision. (b) Example showing that an 9-bit floating point representation is sufficient to reach the maximum value of the 77 dB SNR requirement.

As the variables `L_all()` and `llr()` can take on both positive and negative non-integer values, fixed-point 2's complement arithmetic is chosen. This will ease the in the execution of state 12, where sign change is done based on the value of `est()`. A 2's complement representation also simplifies addition and subtraction, as it is unnecessary to treat sign bits separately from the rest of the bits [24].

To establish the minimum fraction length of this 2's complement fixed point representation, so the BER of the turbo encoder is not increased, Matlab simulations are done. `Mdiff()` in *sova0.m* is quantized using fixed-point arithmetic and a minimum word length of 27 bits. The fractional part of this quantization is then increased until BER of the quantized `Mdiff()` version reaches the BER of Matlabs double precision `Mdiff()`. Figures G.2a to G.2f in appendix G shows selected results from these simulations. The wanted precision is obtained at a word length of 44 bits giving a 17-bit fraction length, which is illustrated in figure 6.12a. Here there is no increase of BER between the fixed point and double precision versions. To illustrate the need for a fractional representation of `Mdiff()`, `L_all()`, and `llr()` the results for a word length of 27 and a fraction length of 0 is seen in figure 6.12b.



(a)



(b)

Figure 6.12: (a) A 0 % error increase is acheived at a word length of 44 and a fraction length of 17. (b) Without a fractional representation the quantized version introduces a 3.47 % increase in decoding errors.

As the results plotted in figure 6.12 is based on random sequences of ones and zeros, the 0 % increase of errors might be acheived at a slightly higher or lower word length. However several simulations showed no errors occured while using a word length of 44 bits. The observant reader might have noticed that the BER plots of figure 6.12 deviates from its decreasing slope as E_b/N_0 increases. The reason for this lies in the simulation algorithm where "only" 10.000 samples were processed for each value of E_b/N_0. A consequence of this is that almost no errors are detected as E_b/N_0 reaches 10-11 dB, which increases the variance of the error probability and makes the BER less reliable. For a more detailed explanation on how the simulation should be done instead, see appendix G. In this appendix the setup of the quantizer in Matlab is also explained.

For storing the variables mentioned above three slightly different memory elements are available. These are illustrated in figure 6.13. The simplest memory element is the register, which consist of n number of D flip-flops as in fig. 6.13a, which illustrates a 4-bit register. Registers are limited to contain one value if more values needs to be stored, register files comes in handy. Basically a register file consists of 2^n register rows and some control logic. Each row is capable of saving an m-bit value, where m is the number of D flip-flops, just like in the case of the register. The control logic in a register file is used for determining the row (address) on which a value is either written to or read from. Register files are usually used for storing 2, 4, 8 or 16 values. Finally there is the random access memory (RAM), which resembles the register file, but differs as its input and output is combined, illustrated by the *data* arrow in fig. 6.13c. Otherwise it also consists of 2^n rows each capable of storing an m-bit value. RAM usually stores around 2^{16} to 2^{32} values with a word length in the power of two ranging from 1 to 32.[24]

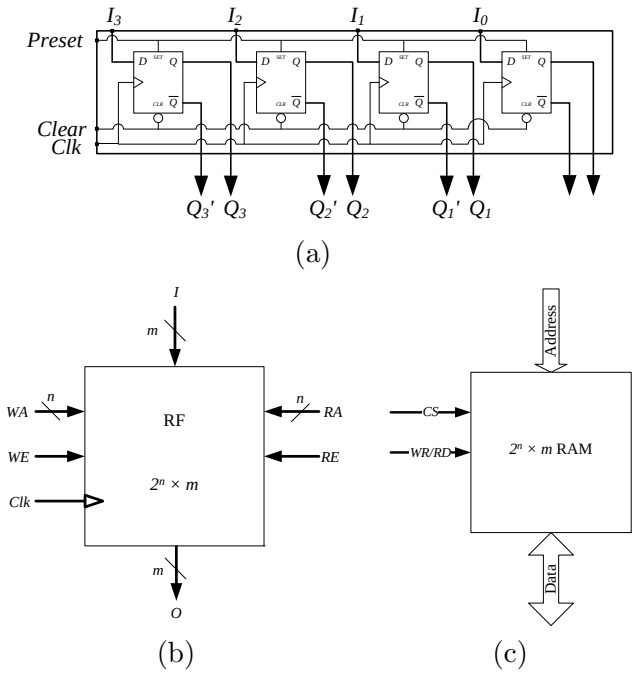


Figure 6.13: (a) 4-bit register with preset and clear. This register also provides an inversion of the input. (b) Register file with m-bit word length and capable of storing 2^n different values. (c) RAM memory with a m-bit word length and capability of storing 2^n values.[24]

6.2.1 Conclusion on Data Structures

With the knowledge established in this section it is possible to determine the size of the memory elements needed for each variable. The word length and memory size for each variable is listed in table 6.4 as well as the memory element that fits the variable the best. These distributions of variables into memory elements may be subdued to change when merging of variables, as an optimization of cost, is done in the following section.

6.3 Cost Optimization Techniques

The ASM charts derived in section 6.1.1 is a good aid in the creation of a datapath that fits the Moore FSM, which was also presented in that section. In [24] several techniques for cost and performance optimizing a datapath design is presented and many of them are based on the ASM chart. Some of the techniques used for cost optimization is presented in this section, starting out with optimization of memory allocation. This will reduce the cost of the implementation, as it reduces the number of memory elements needed, but also the number of connections needed between memory elements and functional units.

Table 6.5 is a variable usage table that shows the lifetime of the variables in the Moore ASM

Variable	Word length	# of values	Memory type
`t`	12-bit	1	Register
`llr`	44-bit	1	Register
`i`	6-bit	1	Register
`x`	12-bit	1	Register
`L_total`	12-bit	1	Register
`bit`	1-bit	1	Register
`est()`	1-bit	2500	RAM
`temp_state`	4 bit	1	Register
`last_state()`	4 bit	16	Register file
`mlstate()`	4-bit	2500	RAM
`j`	6-bit	1	Register
`prev_bit()`	1-bit	20000	RAM
`delta`	6-bit	1	Register
`Mdiff()`	44-bit	20000	RAM
`L_all()`	44-bit	2500	RAM

Table 6.4: Word length and number of values for all variables used in the algortihm of listing 5.1.

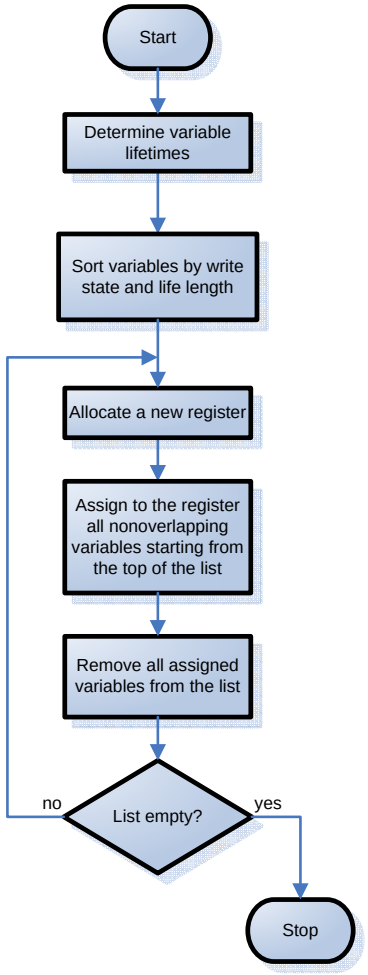
chart illustrated in figure 6.5. A variable's lifetime starts at the state following the variable's write state - the state in which a variable is assigned a value for the first time. An 'x' is put in the table for the state following this write state to indicate the start of the variables lifetime. Furthermore an 'x' is assigned the states where the variable is used on the right hand side, known as a read state. Finally the states in between the first 'x' and the last read state of the variable, are marked with an 'x'.

6.3.1 Left Edge Algorithm

The variable usage table in 6.5 is used for grouping non-overlapping variables together in as few memory elementss as possible. One way of grouping variables together is with the use of the *left edge algorithm* [24, chapter 8]. Here the variables are first prioritized based on their start state and then their lifetime span. Should two or more variables have the same start state and lifetime span, a random prioritizing is used. This sorting of the variables have already been done in table 6.5 and it shows that 12 memory elements are needed to contain all the variables. The result of allocating memory elements by use of the left edge algorithm is shown in figure 6.14a and a flowchart for the left edge algorithm is illustrated in figure 6.14b. Note that in figure 6.5 several variables are only stated on the right side of the equations and it would therefore seem like no value were ever assigned to this variable. However in tab. 6.5 this is interpreted as these variables are assigned their values i the state previous to the state it is first used in. These variables are actually supplied by the first part of the *sova0.m* algorithm, which was ignored as no bottlenecks showed up in the profiling.

Variable/State	s_0	s_1	s_2	s_3	s_4	s_5	s_6	s_7	s_8	s_9	s_{10}	s_{11}	s_{12}	s_{13}
t		x	x	x	x	x	x	x	x	x	x	x	x	x
llr			x	x	x	x	x	x	x	x	x	x	x	
i			x	x	x	x	x	x	x	x	x	x	x	
L_total				x	x	x	x	x	x	x	x	x	x	x
x				x	x	x	x	x	x	x	x			
est()					x	x	x	x	x	x	x	x	x	
mlstate()					x	x	x	x	x	x	x			
bit					x	x	x	x	x	x				
last_state()					x	x	x	x						
temp_state()						x	x	x	x					
j							x	x	x	x				
prev_bit()							x							
Mdiff()											x			
delta												x		
L_all(t)														x
Number of live variables	0	1	3	5	9	10	12	11	10	9	8	6	5	3

Table 6.5: Sorted variable usage table which shows the variables lifetime span.



(a)

ME_1 = [t]
ME_2 = [llr, L_all(t)]
ME_3 = [i]
ME_4 = [L_total]
ME_5 = [x, delta]
ME_6 = [est()]
ME_7 = [mlstate()]
ME_8 = [bit, Mdiff()]
ME_9 = [last_state()]
ME_{10} = [temp_state]
ME_{11} = [j]
ME_{12} = [prev_bit()]

(b)

Figure 6.14: (a) Flowchart for the left edge algorithm used for memory allocation [24, fig. 8.13]. (b) Allocation of variables into memory elements based on the left edge algorithm. Note that even though L_all(t) and llr are used in the same state (s_{12}), they can still be allocated in the same memory element, as L_all(t) is the destination of s_{12} and therefore first starts its lifetime in the following state.

6.3.2 Operator Merging and Graph Partitioning Algorithm

Another way to allocate variables is by use of compatibility graphs [24]. These graphs link variables together with incompatible and priority edges. An incompatible edge is put between two variables with overlapping lifetime, whereas a priority edge is put between variables with non-overlapping lifetimes and which shares a source or destination with each other. A weight metric is assigned to each priority edge describing how many functional units that uses the two connected variables as source and/or destination (s/d). Here s is the number of functional units that uses both connected variables as left or right side operands. d indicates the number of functional units that generates values for both connected variables. The higher the weight of s/d the higher priority of merging the variables. By merging variables based on these priorities it is possible to reduce the cost of memory elements as well as the connectivity cost.

Merging of Operators

Before establishing the weight of priority edges, it is a good idea to see if any operations can be merged into a single functional unit. This will further reduce the number of connections in the datapath and the cost of the functional units. But before the merging of functional units is commenced, the operations of the Moore ASM chart is modified to take advantage of the resources a hardware implementation provides. First the `1-est(x)` function is simplified to an inverter, as `est()` can only take on the values "1" and "0". This also affects s_{12}, where `llr` changes sign if `est(t)` = 0. Changing sign of a binary value is fairly simple if a 2's complement representation is used. Instead of using the multiplier, the 2's complement is found by inverting the bits representing the value of `llr` and adding 1 to this value. This method is feasible as long as `llr` is not the highest representable negative value for its word length. The reason for this lies in the range of 2's complement; $[-2^{n-1}:2^{n-1}-1]$ which shows that there does not exist a 2's complement for -2^{n-1} where n is the word length of the variable. An easy way to perform this operation is explained later on when an ALU design is determined.

Another interesting aspect of the algorithm is the "`bit+1`" operation in s_4 and s_7. The reason why "1" is added to `bit`, which only takes on the values "0" and "1", is because Matlab can not index with zero. As this is not a limitation of a hardware implementation, the "`bit+1`" is neglected. This would of course require some additional modification of the surrounding code providing the variables for `last_state()`. These additional changes are however not in the aspects of this report.

With these modification the different operations are listed in an operation usage table 6.6 based on the states they are used in. This table shows the maximum number of any given operation in any state and thereby also the maximum number of functional units needed. E.g. two additions are needed in state s_4 and s_6, which means two adders are needed in the datapath. The table also shows that many operators are only used in a single state and is therefore idling in the rest. This indicates that a merging of operations into functional units may be beneficial.

Note from the operation usage table 6.6 that only one "+"-operation is done in s_4 and none are done in s_7 contrary to what is stated in fig. 6.5. As previous explained these increment `bit` by one operations were neglected, as they originated from the index constraints of Matlab.

In tab. 6.6 it is also stated that two "+"-operations are necessary in s_6. It will be explained later on, how the ALU can add two variables and still increment by one. This removes the need for a second functional unit, which would otherwise be necessary to conduct two "+"-operations in one state.

Merging operations into functional units based on how they are computed, results in two func-

Operand /State	s_0	s_1	s_2	s_3	s_4	s_5	s_6	s_7	s_8	s_9	s_{10}	s_{11}	s_{12}	s_{13}	Max # units
+			1		1		2				1	1		1	2
≤			1												1
inv				1											1
-						1			1						1
≥						1			1						1
≠										1					1
min											1				1
>												1		1	1
sign													1		1
Number of operations			2	1	1	2	2		2	1	2	2	1	2	

Table 6.6: Operation usage table. Note that only **sign** is used in s_{12} indicating that this operation can be merged into both functional units.

tional units, each containing the operators as specified: FU1 = [+, -, **inv**, **sign**] and FU2 = [≤, ≠, ≥, >, **min**]. The operations conducted by FU1 are all standard ALU operations. In [24] an ALU is presented which is capable of conducting all of the operations needed in FU1.

ALU Design for FU1

Since all arithmetic operations are based on the adder, the main building blocks of the ALU is n number of full adders (FA) linked together as a ripple-carry adder or a carry look ahead adder. Modifying logic called arithmetic and logic extenders (AE and LE) is connected to each input of these adders. With only three control signals it is possible to conduct all the operations needed for FU1. Table 6.7 and 6.8 lists the operations available by the partial ALU design illustrated in figure 6.15. The AE, LE, and FA blocks of fig. 6.15 consists of simple combinational logic.

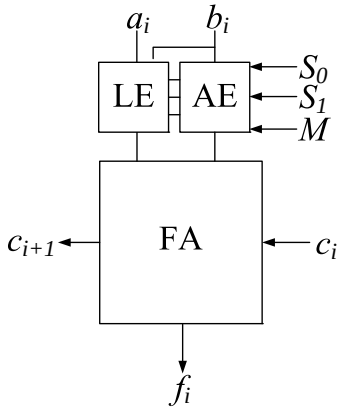


Figure 6.15: One section of an ALU based on the full adder ripple carry design.

M	S_1	S_0	Function Name	F	X	Y	c_0
1	0	0	Decrement	A-1	A	all 1's	0
1	0	1	Add	A + B	A	B	0
1	1	0	Subtract	A + B' + 1	A	B'	1
1	1	1	Increment	A + 1	A	all 0's	1

Table 6.7: Control signals and how the input signals are modified based on these control signals in the arithmetic extender. [24]

M	S_1	S_0	Function Name	F	X	Y	c_0
1	0	0	Complement	A'	A'	0	0
1	0	1	AND	A AND B	A AND B	0	0
1	1	0	Identity	A	A	0	0
1	1	1	OR	A OR B	A OR B	0	0

Table 6.8: List of logic operations provided by the logic extender. [24]

With this ALU design it becomes easy to conduct the increment by one and decrement by one operations needed in s_5, s_6, s_8, s_{10}, and s_{13} as illustrated in fig. 6.5. The `t+j+1` operation in s_6 is accomplished by choosing the ALU's add function and setting the carry c_0 to "1".

It takes a little effort to perform a sign change of `llr` based on the value of `est()`, but here is how it is done. The A input to the ALU should be all 0's while the B input is the value of `llr`. `est()` is used as an internal datapath control signal, setting the values of S_1, S_0 and c_0, whereas M should be set by the output logic of the control unit. Figure 6.16 illustrates these modifications. The additional logic added to this FU1 design could be seen as additional logic added to the output logic of the control unit, where operation is changed based on the value of `est()`. This leads to a design that resembles the Mealy FSM a bit.

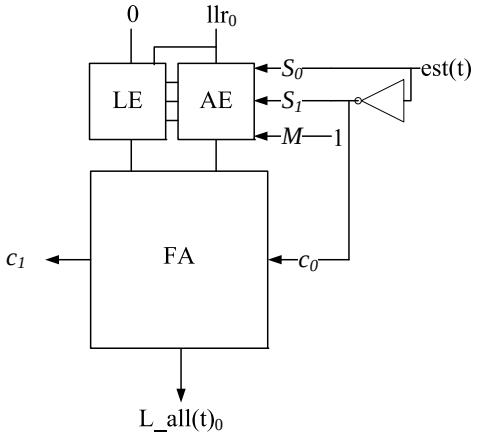


Figure 6.16: Implementation of the operation in s_12. As `llr` is 44 bit value the FA in this figure should be followed by 43 other resembling circuits (leaving out the control logic from `est(t)`

Of the logic operations stated in table 6.8 only the complement and identity functions are used.

Complement is used to invert `est(x)` in s_3 and identity is used to pass the A input through to the full adder. This ALU implementation is of course an overkill to provide a single logic inverter function, so instead the LE block could be replaced with an inverter separated from the ALU. In the following however, the inverter operations is treated as part of FU1. Combinational logic and truth tables for AE, LE and FA is given in appendix H.

Comparator Design for FU2

FU2 mainly consists of comparison operations used for making decisions throughout the algorithm, the exception being the **min**-operation in s_{10}. As FU2 consists of a wide variety of comparators, a universal comparator would be a good way of realizing FU2. The problem with a universal comparator as the one given in [24, chaptef 5.10] is that it only works for positive integers. This is an issue for s_5 and s_8 as the variable `j` at some point becomes negative. Another way of implementing a comparator is with the use of an ALU, but such an implementation is left for future work.

Memory Merging by Graph Partitioning Algorithm

With the merging of operators into functional units given above, all the necessary information for designing a compatibility graph for merging variables is obtained. The graph is illustrated in figure 6.17b and a list of how the weights were calculated is given below. Variables which are incompatible due to overlapping lifetimes or does not share a source or destination, are not mentioned in the following.

- `x` is the left side operand of FU2 as it uses the "$\leq$" operation in s_2. `delta` is the right side operand of FU2 in s_{11}, which results in a 1/0 weight between `x` and `delta`.
- `j` shares weight 1/0 with `Mdiff()` and `delta` due to comparator operation in FU2.
- Both `Mdiff()` and delta are sources for a comparator and therefore share weight 1/0.

This concludes the first step of the graph partitioning algorithm shown in figure 6.17a, resulting in the initial compatibility graph illustrated in figure 6.17b. The next step is to merge the nodes with highest priority (variables that share the highest weight) into supernodes. This is done until only incompatibility edges are left, and the supernodes should then be the optimum memory allocation based on functional units available in the datapath.

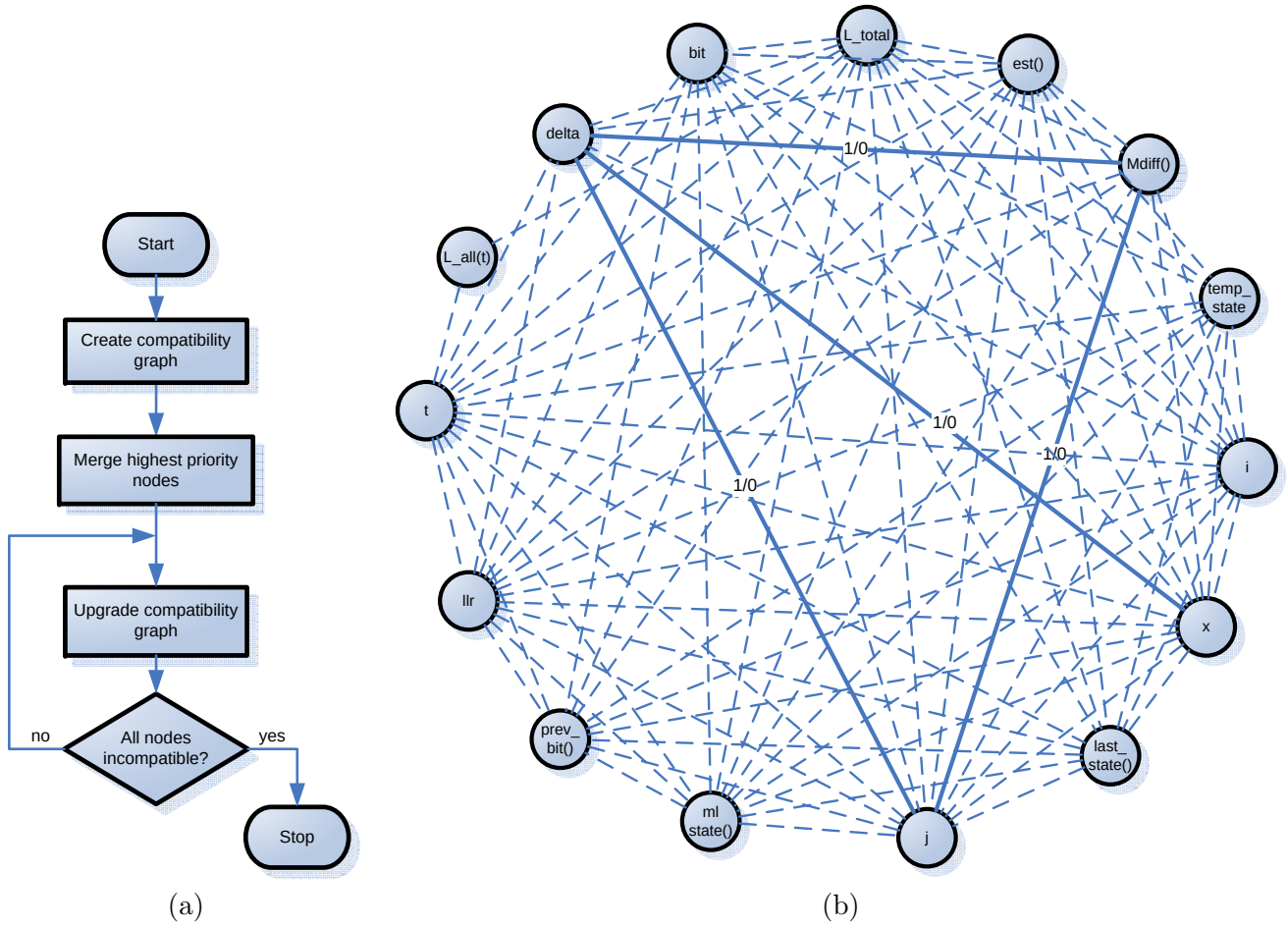
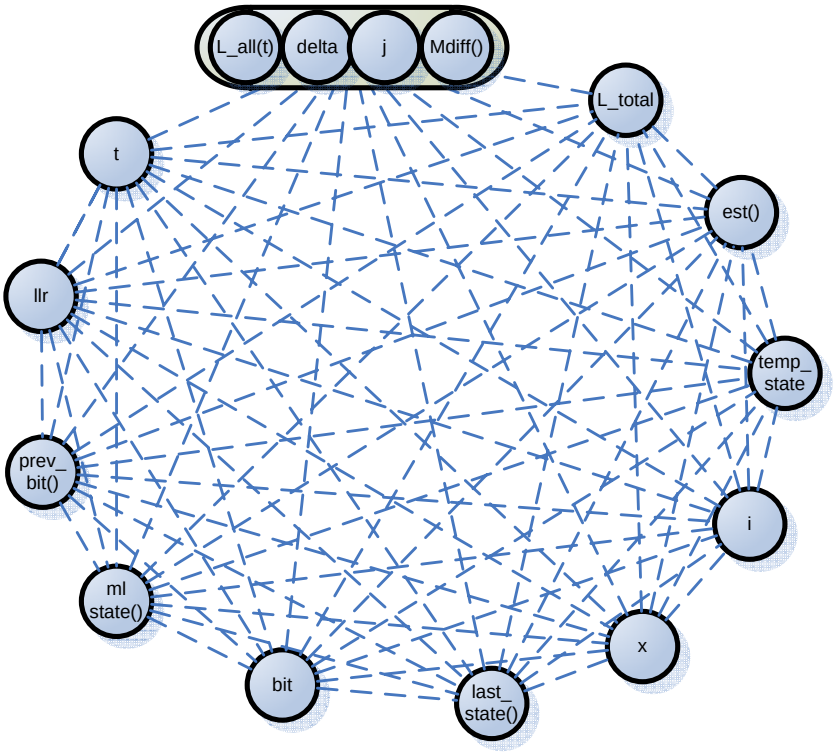


Figure 6.17: (a) Flowchart for the graph partitioning algorithm [24, fig. 8.16]. (b) Initial compatibility graph for a Moore FSM and the functional unit grouping given by FU1 and FU2.

In figure 6.17b it is seen that all weights are equal and in this case [24] states that the variable with the highest number of priority edges should be merged first. This leads to merging of `delta`, `Mdiff()`, and `j` since they all have more compatibilty edges than `x`. Lastly `L_all(t)` is merged with `delta`, `j`, and `Mdiff()` to reduce the amount of memory elements needed. This leads to the final compatibility graph illustrated in figure 6.18a and the memory allocation listed in figure 6.18b.



(a)

ME_1 = [L_all(), delta, j, Mdiff()]
ME_2 = [t]
ME_3 = [llr]
ME_4 = [i]
ME_5 = [L_total]
ME_6 = [x]
ME_7 = [est()]
ME_8 = [mlstate()]
ME_9 = [bit]
ME_{10} = [last_state()]
ME_{11} = [temp_state]
ME_{12} = [prev_bit()]

(b)

Figure 6.18: (a) Final compatibility graph with one supernode and only incompatibility edges left. (b) Optimum memory allocation of variables based on FUs. Still 12 memory elements are needed.

With the allocation given in 6.18b it is possible to find the size of the 12 memory elements needed in the hardware implementation. These sizes are given in table 6.9. Based on this allocation of variables a datapath for each state is derived and illustrated in figure 6.19a and b. Note that each connection in this figure has a short abbreviation attached to it. These specifies if a connection is an output to a register (O-), a input to a functional unit (I-) or a indirect addressing connection. Based on these notations several buses are derived with the use of the graph partition algorithm. Note in 6.19b that a red arrow indicating indirect addressing is points to the side of FU1 ([+, -, **inv**, **sign**]). This is done to indicate the sign change operation illustrated in 6.16. Furthermore its connection indicator (AKX) is marked with an "X" to illustrate that this connection should not be merged together with others into a bus. The rest of the connections are merged in the following section.

The representation of the datapath operations needed at each state, eases the job of identifying the state with the worst register to register delay (or memory element to memory element delay). Even though no exhaustive knowledge has been established for the computation speed of the functional units at this point, it would be a fairly good assumption that state 10 would have

Variables	Memory element	Word length	# of values	Memory type
`L_all(), delta, j, Mdiff()`	ME_1	44-bit	22502	RAM
`t`	ME_2	12-bit	1	Register
`llr`	ME_3	44-bit	1	Register
`i`	ME_4	6-bit	1	Register
`L_total`	ME_5	12-bit	1	Register
`x`	ME_6	12-bit	1	Register
`est()`	ME_7	1-bit	2500	RAM
`mlstate()`	ME_8	4 bit	2500	RAM
`bit`	ME_9	1 bit	1	Register
`last_state()`	ME_{10}	4-bit	16	Register file
`temp_state`	ME_{11}	4-bit	1	Register
`prev_bit`	ME_{12}	1-bit	20000	RAM

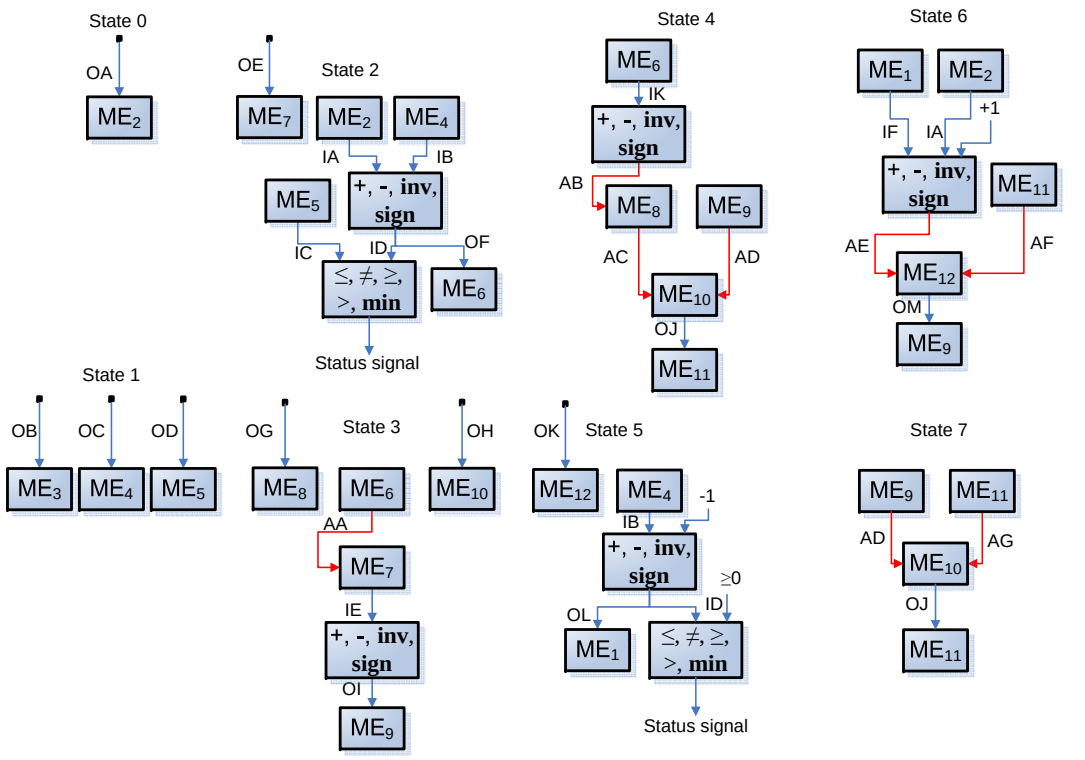
Table 6.9: Word length and number of values each memory element needs to contain for all variables used in the algortihm of listing 5.1.

the worst register to register delay. In state 10 both functional units are used sequentially with two intermediate memory reads. Assuming that the register to register delay is the main contributer to the critical path compared to the contribution of the propagation delay through the control unit. It is possible to establish the execution time for a hardware implementation of the algorithm. As the clock signal and thereby period time for each state is determined by the propagation delay of the critical path, the total execution time is established based on the propagation delay from status register to status register in state 10. First though a merging of connections is done and pipelining of the functional unit is undertaken.

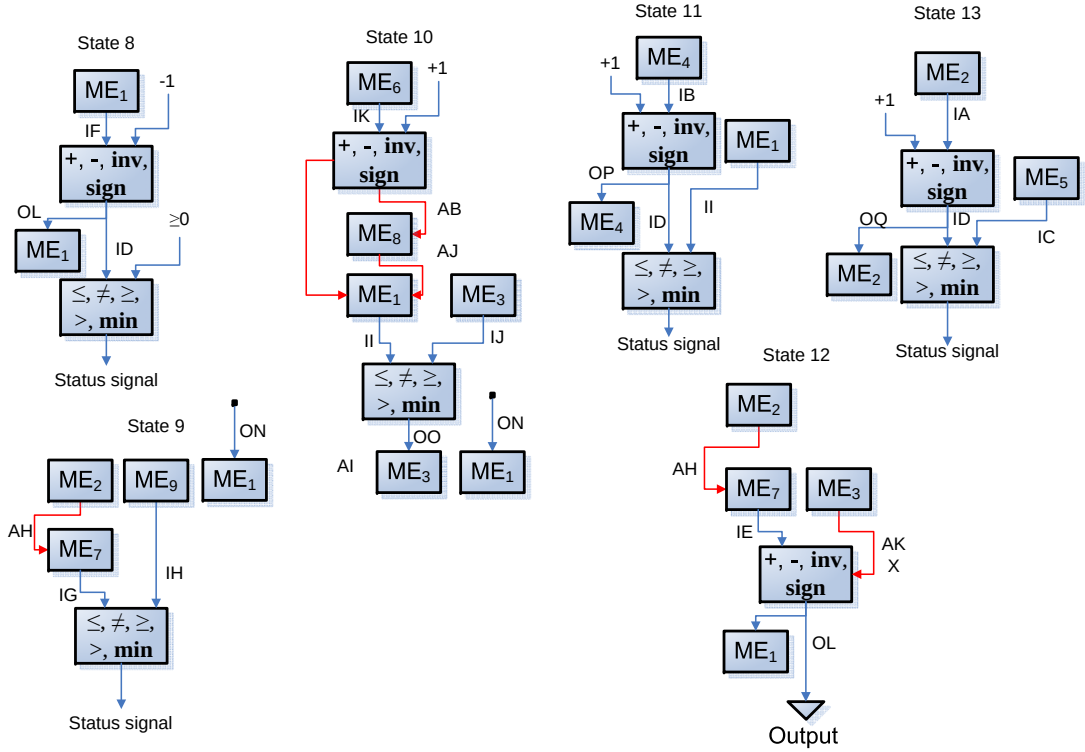
6.3.3 Connection Merging

The final cost optimization technique is applied to the datapath schematics mentioned above. As explained earlier each individual connection is marked with an abbreviation starting with A, I or O, indicating if a connection is for indirect addressing, input to a FU or input for to a memory element (usually output from a FU) respectivly. This is done to help identifying which connections should be merged together as Address Input or Output bus.

Once again the graph partitioning algorithm is used, but first a connection use table is made that indicates which states a connection is active in. Connections that are active in the same state cannot be merged into the same bus and is indicated with an incompatibility edge. Priority edges are made between connections with the same source or destination. All this was done for the 11 input connections, resulting in 4 separate input buses illustrated in figure 6.20a. The same was done for the 17 output connections with the 3 output buses in fig. 6.20b as the result. Finally the 10 indirect addressing connections are merged into the 3 buses seen in fig. 6.20c.



(a)



(b)

Figure 6.19: (a) Datapath operations needed for state 1 to 7. Indirect addressing is indicated by red arrows that enters the side of memory elements. (b) Datapath operations for state 8 to 13.

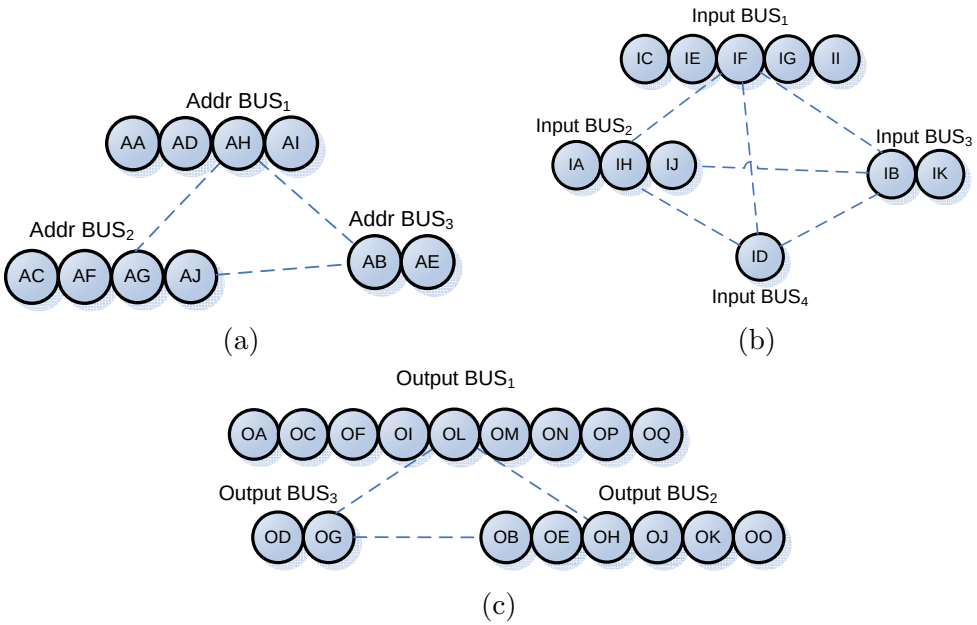


Figure 6.20: (a) Compatibility graph for indirect addressing resulting in 3 separate buses. (b) Compatibility graph for connection merging of input connections resulting in 4 separate buses. (c) Compatibility graph for connection merging of output connections resulting in 3 separate buses.

Note that in fig. 6.20b connection ID is not merged with any other connections. The reason for this is not only because it is incompatible with the other constellations of buses. It is also because this connection feeds the output of FU1 directly to FU2 and is therefore an essential part of the datapath. These buses greatly decreases the cost of the datapath implementation. Figure 6.21 shows the datapath with these 10 buses. The memory elements and functional units connected to these buses, drives the bus through a tristate bus driver, which activates the bus in the necessary states. By using these buses the necessity of selectors in front of functional units is removed, further reducing the cost of this implementation.

6.3.4 Conclusion on Cost Optimization

This concludes the use of techniques for optimizing the cost of an datapath implementation for the algorithm listed in listing 5.1. It has been demonstrated how based on the ASM chart for a Moore FSM, it was possible to merge variables, operators, and connections together in ways that decreased the cost of the implementation. In the following pipelining will be used for optimizing the performance of the datapath.

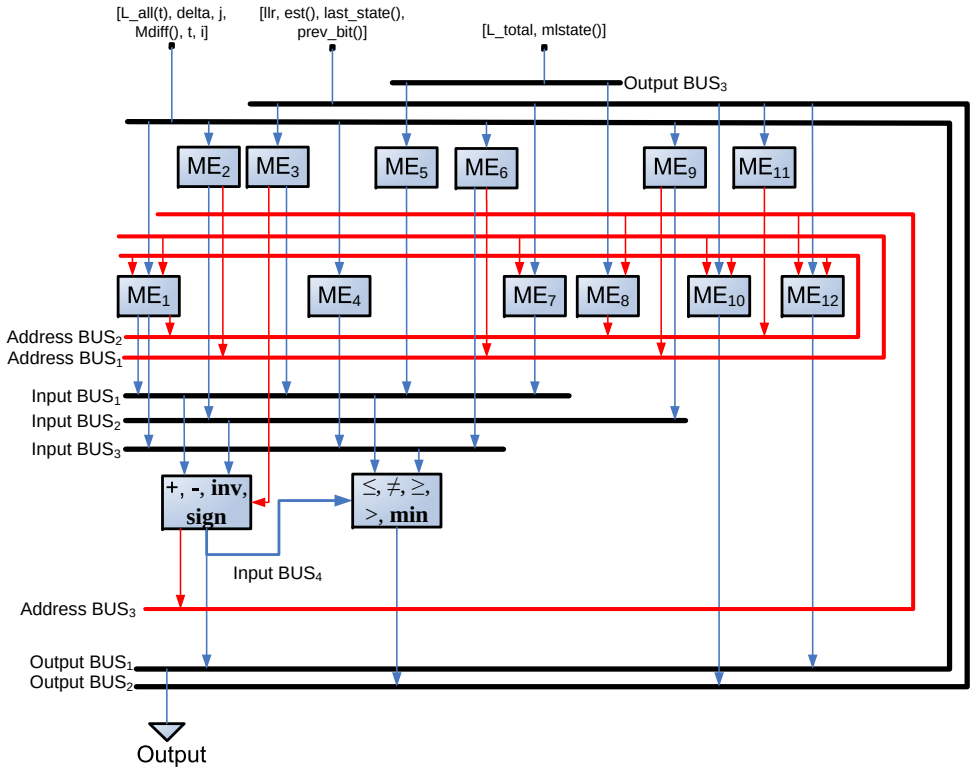


Figure 6.21: Datapath with connections merged into 10 buses. Note that only three inputs for loading the variables is necessary in this bus design.

6.4 Performance Optimization

There are several ways to optimize performance of a design and some has already been done at this point. Removing the need for an extra adder in chapter 6.3.2 by ignoring the "`bit+1`" as a Matlab constraint, is one way of optimizing the performance. Another is the reduction of next state equations using the Quine-McCluskey Algorithm. In the following pipelining is used to optimize the performance of FSMD design.

6.4.1 Functional Unit Pipelining

The pipelining method used for the datapath is called functional unit pipelining, where the the FU is divided into stages. Each stage of the FU is then capable of running simultaneously with the other stages on different parts of a given operation. This means that while one following pipeline stage of an FU is performing one part of an operation, the preceding stage can take on a new operation. Other forms of pipelining is datapath and control unit pipelining. A good metaphor for a pipeline is a assembly line at a factory. [24, chapter 8]

Pipelining of a FU optimizes both latency and throughput of the FU and thereby also the datapath. There is no time improvements for processing the first set of operands that is put through a pipelined FU. But for the next set of operands the latency is brought down with a rate equal to the number of pipeline stages. E.g if a non-pipelined design has a delay of 30 ns, a 3-stage pipeline would still use 30 ns to process the first set of operands, but for the following sets of operands a new result is obtained for every 10 ns. When implementing a pipeline the main goal is to cut the critical path into equal sizes. Two-stage pipelining of a functional unit is done by inserting latches in the middle between its inputs and outputs. This means that two clock cycles (or states) are necessary to perform one result, but if the pipelining is done in the exact middle of the critical path, it is possible to increase the clock signal by two.

A 2-stage pipeline is implemented in FU1 as FU1 is located near the middle of datapath for nearly every state, illustrated in fig. 6.19a and 6.19b. This does not introduce immediate changes to the overall datapath design, but the number of states required to execute the algorithm is increased as expected. Timing diagrams for the non-pipeline and pipeline execution of the algorithm is given in table I.1 and I.2 in appendix I. Here the read and write operations as well as the operations executed by the functional units are listed for each state. Note from the pipeline timing diagram, that one of the two FU1 stages is idling in several states. This is due to several decisions made throughout the algorithm. It is not possible to merge a state where a decision is made together with the following state, because the decision state does not always jump to its following state. For this reason there are only three parts of the algorithm that is capable of utilizing both stages of FU1. One part is from s_3 to s_5, the second part is from s_6 to s_8, and the third is from s_{12} to s_{13}, as no decision blocks interrupts the data flow for these states.

6.4.2 Conclusion on Performance Optimization

The pipeline of FU1 increases the number of states by six, but as mentioned earlier if the pipeline of FU1 cuts the critical path in two equal sizes, it is possible to increase the clock frequency by two. As a result it is possible to execute the 19 states of the pipeline in the same time it takes to execute 9.5 states in the non-pipeline design. If the pipeline does not separate the critical path in two equal parts, the clock frequency is based on the biggest part of the critical path. In the next chapter a description of the Virtex-5 FPGA from Xilinx is given and based on the performance of this FPGA an estimate for the execution time for the algorithm is established. Finally this is used to see if the pipeline hardware implementation of the algorithm is capable of executing at a sufficient rate.

Chapter 7

Virtex-5 Analysis and Implementation

This analysis will mainly revolve around the parts of the Virtex-5 architecture that is of interest for the hardware design given in the previous chapter. Based on the analysis components are chosen for a hardware implementation and the performance of these components is used to determine the execution time and throughput of the FSMD design from the previous chapter. These results is held up against the EGPRS-2 requirements to turbo coding, specified in chapter 3. For the following description [29] and [30] is used as reference.

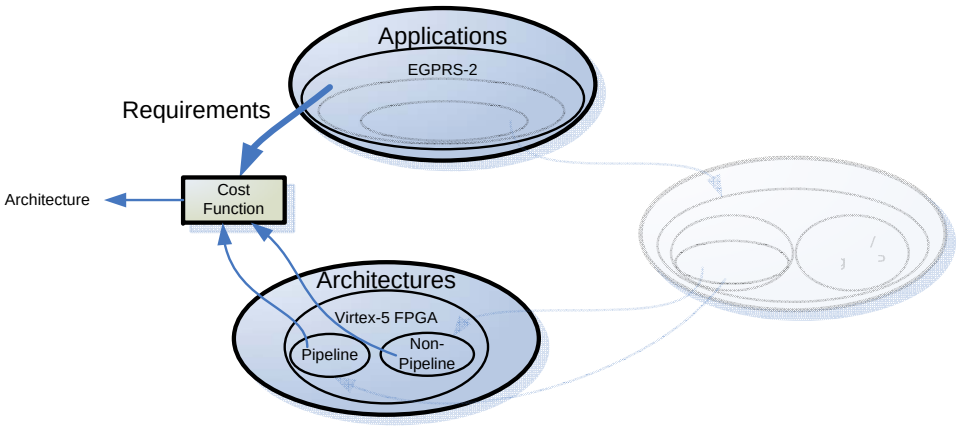


Figure 7.1: Analysis of the FPGA chosen for hardware implementation results in estimations of execution time and throughput. These are compared for the two designs with the specifications for throughput in chapter 3 and a architectural design is chosen based on this comparison.

The Virtex-5 FPGA consist of 6 main building blocks. A simple diagram of these components is illustrated in figure 7.2. Starting from left to right the first components are high-speed I/O's that incorporates ASIC circuits for improved performance. Next up is a set of buffers (BUF)

followed by configurable logic blocks (CLB), which can be used for implementing combinational an sequential logic. Then an array of block RAM (BRAM) is depicted followed by yet another set of CLBs before a set of high performance DSP48E digital signal processors. The DSP48E provides arithmetic functions such as adders and multipliers, but it can also perform logic operations. Virtex-5 uses block RAM for building memory arrays and clock management tiles (CMT) for reducing clock skew and jitter. One important part is left out of fig. 7.2, this is the switch matrix that provides interconnects between the components depicted in the figure.

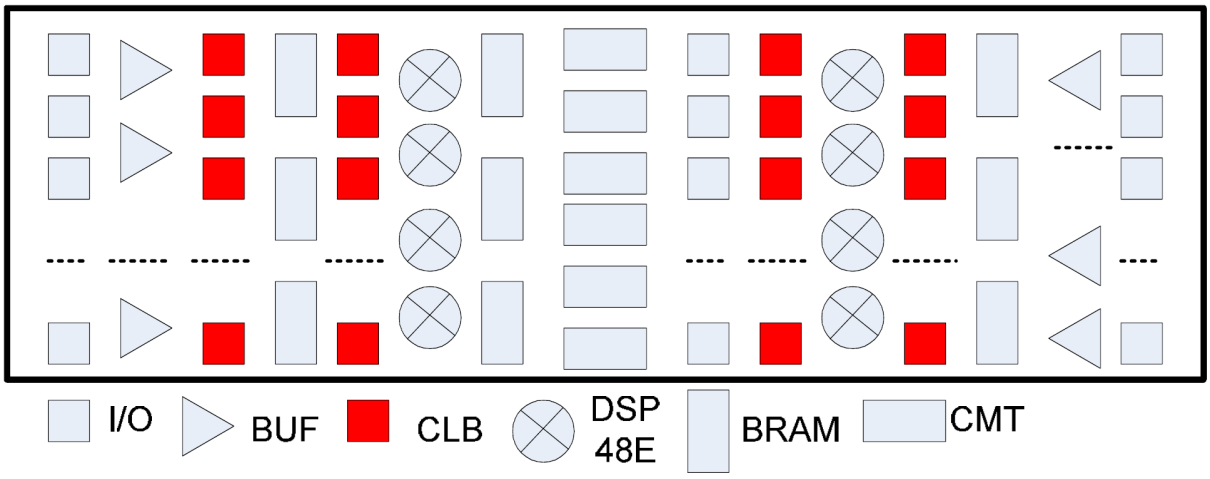


Figure 7.2: Simple diagram of the Virtex-5 FPGA's components [29].

As described in the previous chapter both the control unit of the FSM and the functional units of the datapath could be implemented as sequential and combinational logic respectively. This makes the CLBs of the FPGA ideal for implementing the control unit and the arithmetic operations of the functional units. Furthermore the CLBs may be used for implementing memory elements as well.

A CLB consists of two slices to which the switch matrix is capable of connecting other CLBs so designs that exceeds the capacity of one CLB can be implemented. Each CLB slice consists of four look up tables (LUT) and four flip-flops (FF). With a LUT is is possible to implement any truth table as long as it meets the constrains set by number of in- and outputs. The LUT of the Virtex-5 has 6 inputs and 1 output, but can be combined with other LUTs to increase the number of outputs. This makes the LUT ideal for implementing the combinational logic of the functional units. The FF acts like memory elements and together with the LUT it can be used for implementing sequential logic, such as that of the control unit.

Propagation delay through interconnects between CLBs may account for more than 50 % of the delay through the critical path. To reduce this, Virtex-5 incorporates *a diagonally symmetric interconnect patern, [30]* . This allows for a faster interconnect pattern as more connections can be made with only a few hops. It also allows the Xilinx ISE software to derive the optimum

Functional Blocks	Execution time
6-Input Function	0.9 ns
Adder, 64-bit	2.5 ns
Ternary Adder, 64-bit	3.0 ns
Barrel Shifter, 32-bit	2.8 ns
Magnitude Comparator, 48-bit	1.8 ns
LUT RAM, 128 x 32-bit	1.1 ns

Table 7.1: Execution time for a variety of CLB implemented functional blocks.

routing between CLBs. The routing delay from one CLB to its surrounding CLBs is 665 ps, and for the second ring CLBs the delay is 723 ps. Figure 7.3 illustrates the CLB design described above.

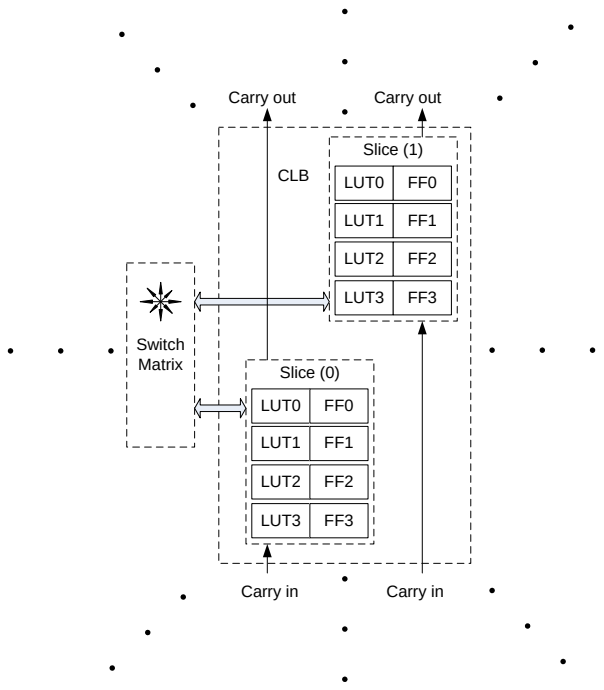


Figure 7.3: Arrangement of LUTs, FFs, slices, switch circuit and CLBs in the Virtex-5 FPGA. Note that the dots represents the CLB being surrounded by other CLBs in all directions. Note also the arrow arrangement in the switch matrix block, indicating that connections can be made directly to adjacent CLBs in all directions [29].

Table 7.1 states the time it takes to execute different functional blocks implemented by CLBs. Furthermore it is stated that the block RAM of the FPGA can operate up to speeds of 550 MHz. Interfaces to external memory may operate up to speeds of 333 MHz (DDR2 and RLDRAM II) [31]. With these specifications it is possible to compute the execution time of the pipeline hardware implementation based on state 10 being the state with the longest state register to state register delay. The results are listed in table 7.2 and the operations for state 10 is illustrated in fig. 7.4.

Operation	Execution time
Control Unit	3×0.9 ns $= 2.7$ ns
Read ME_6	1.1 ns
Increment ME_6	0.47 ns
Read ME_8	1.82 ns
Read ME_1 and ME_3	3 ns
min	1.8 ns
Write ME_1 and ME_3	3 ns
Interconnects	$2 \cdot 13.89$ ns
Total time for s_{10}	27.78 ns

Table 7.2: Execution time for each operation in state 10. Note that it is assumed that interconnects make up for half of the entire execution time.

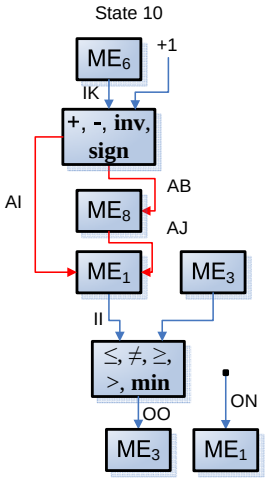


Figure 7.4: Datapath for state 10.

The execution time for the increment ME_6 operation is based on the execution time of the 64-bit adder operation. It assumed that this adder operation is implemented as an ripple carry adder, where the propagation delay is proportional to the word length. This means that the propagation delay for the 12-bit value of ME_6 is given by $\frac{12}{64} \cdot 2.5$ ns $= 0.47$ ns. It should be noted that ME_6 is implemented in LUT RAM. ME_8 and ME_3 is implemented as block RAM, because their base is 36 Kbits and may contain bit widths as high as 72 bits. ME_1 is stored in external RAM as it contains 22502 values with a word length of 44 bit, resulting in almost 1 Mb of memory usage. The latency for block RAM and external RAM is calculated as 1/(maximum operation frequency) assuming that only one clock cycle is necessary for reading and writing to these memory elements. Note also that the latency of the control unit is set at $3 \times$ (latency of 6-input function). The reason for this assumption is caused by the fact that the reduced next state equation for D_0 in 6.2 uses seven variables and two 6-input LUT is therefore combined in an implementation of this function. Furthermore the output logic is combinational logic based on the four outputs from the state register and can therefore be implemented as a 6 input function.

It may however, be necessary to implement several 6-input functions in parallel to provide the needed amount of control signals. This should not increase the propagation delay through the output logic, as the 6-input functions is coupled in parallel to supply enough control signals.

With a total execution time for state 10 of 27.78 ns and 13 states in the non-pipeline implementation, clock frequency and execution time is given by:

$$\text{Clock frequency} = \frac{1}{27.78 \text{ ns}} \approx 35 \quad [\text{MHz}] \tag{7.1}$$

$$\text{Total execution time} = 13 \cdot \frac{1}{35 \text{ MHz}} = 371.43 \quad [\text{ns}] \tag{7.2}$$

Note that the clock frequency is rounded down to the nearest integer MHz value. If instead the clock frequency was rounded to the nearest integer (in this case 36 MHz), then state 10 would not be able to execute at this rate. Remembering from chapter 5.1 about profiling, that this Moore adapted algorithm took up 70 % of the total execution time, and from fig. 6.5 that one bit is calculated for each full iteration of the ASM chart, the thoughput of the entire SOVA algorithm is given by:

$$\text{Total execution time for SOVA} = \frac{1}{70\%} \cdot 371.43 \text{ ns} = 530.6143 \quad [\text{ns}] \tag{7.3}$$

$$\text{Throughput} = \frac{1 \text{ bit}}{530.6143 \text{ ns}} = 1.88 \quad [\text{Mbit/s}] \tag{7.4}$$

Note that these results are only true if a hardware implementation on the remaining algorithm delivers a performance increase similar to the one of this non-pipeline implementation.

Assuming that the pipeline of FU1 cuts the critical path in two equal sizes, the clock frequency may be doubled, giving a clock frequency of 70 MHz. As the pipeline implementation consists of 19 states, its total execution time and throughput for the pipeline implementation would be:

$$\text{Total execution time} = 19 \cdot \frac{1}{70 \text{ MHz}} = 271.43 \quad [\text{ns}] \tag{7.5}$$

$$\text{Throughput} = \frac{1 \text{ bit}}{\frac{1}{70\%} \cdot 271.43 \text{ ns}} = 2.58 \quad [\text{Mbit/s}] \tag{7.6}$$

7.1 Conclusion on Implementation

In chapter 3.3 figure 3.7a a peak bit rate close to 2 Mbit/s was stated for EGPRS-2 with 4 carriers and 8 time slots. In chapter 5.3 eq. 5.5 the throughput of the Matlab SOVA decoder was computed to be 2.02 kbit/s. It was shown in the previous section that a non-pipeline implementation can not handle the required speed of EGPRS-2. However, the pipeline implementation is capable of reaching a sufficient bit rate to comply with the requirements set by EGPRS-2 and it seen that this implementation improves the throughput with a factor of $\frac{2.52 \text{ Mbit/s}}{2.02 \text{ kbit/s}} = 1277$

compared to the Matlab implementation. With these results it has been illustrated that a SOVA decoder implemented on the Virtex-5 FPGA is feasible. Further comments on the results of this report is summarized in the following chapter.

Chapter 8

Conclusion and Future Work

In this chapter the essential parts of the subconclusions drawn throughout this report is summarized. A general conclusion is also presented based on the goals set in the start of this report. Finally this report ends with suggestions for future work, which will revolve round further improvements to the already accomplished work.

8.1 Conclusion

The demand for fast and stable wireless Internet connections has never been bigger, and it is now more a rule than an exception that mobile phones support Internet access. This strains the mobile communication infrastructure, and as upgrading the entire infrastructure is an expensive and time consuming affair, new methods for better utilization of the existing infrastructure is developed. One of these is EGPRS-2, which improves both peak bit rate and spectrum utilization of the most widespread wireless communication standard in the world, GSM.

EGPRS-2 introduces a wide variety of new technologies to enhance the QoS of wireless data transfers. The most computational heavy of these technologies is turbo coding, an error correcting code for reaching near Shannon limit (optimum) coding performance. Due to the computational complexity of this feature, it is chosen for further analysis with regards to optimizing cost and performance through hardware implementation. An investigation of turbo code algorithms shows, that the soft-output Viterbi algorithm (SOVA) achieves the same BER at only 0.1 dB higher SNR than the optimum Log-MAP algorithm. The Log-MAP algorithm, however, introduces a backward recursion that increases its complexity to $O_C(n^2)$, $O_S(2n^2)$ compared to the $O_C(0.5n^2)$, $O_S(0.5n^2)$ complexity of SOVA. For this reason Log-MAP decoding is not as interesting to the industry as SOVA and consequently the SOVA algorithm was selected for further analysis.

Yufei Wu's turbo coding simulation script for Matlab [18], provides the SOVA algorithm used in this report. Profiling of this script revealed bottlenecks in the code and it shows that the SOVA decoder takes up over 93 % of the entire runtime of the simulation script. A part of the algorithm that compares possible paths through a trellis diagram with the received bit stream is responsible for 70 % of the SOVA decoders execution time. Based on this outcome, this part of the SOVA decoder is chosen for optimization.

Through DSE, based on the interest of Rohde & Schwarz Technology Center A/S, it is decided to do an all hardware implementation of the bottleneck in the SOVA decoder. Furthermore it is decided to put the main focus on performance optimization, but utilize cost optimization techniques in the procedure of determining the hardware design. The bottleneck part of the SOVA decoder is adapted to a state based (Moore) ASM chart, which is used throughout the process of optimizing the datapath design. In the procedure of constructing this datapath, techniques for optimizing memory allocations, functional units and connections throughout the datapath, is utilized. Finally a pipeline is implemented in a functional unit to increase the performance of the hardware implementation.

One concern when mapping the SOVA decoder algorithm to a hardware implementation, is in the process of determining the necessary word length for the variables of the algorithm. As they are mapped to hardware, they are assigned a finite word length. This showed through simulations to be 44 bits for some variables, before no increase in BER was obtained. Furthermore the mapping of these variables to hardware revealed that one of them (`Mdiff()`) contained up to 20.000 values. It would be interesting to see if this 20.000-by-44 bit allocation of memory could be brought down by reducing either the number of values or the word length, this will further reduce the cost of the design. One way could be adaptive memory allocation based on the parameters of the turbo coder and utilizing the flexibility of the FPGA.

The XILINX Virtex-5 FPGA is used as a design reference for implementing a Moore based FSMD design. A critical path delay for the FSMD implementation is found based on performance specifications of the combinational and sequential logic available on the Virtex-5. It shows that the non-pipeline implementation can run with a clock frequency of 35 MHz and that the pipeline implementation can run at a rate twice of that, namely 70 MHz. At these rates, the non-pipeline implementation can achieve a throughput of 1.88 Mbit/s and the pipeline version can go as high as 2.58 Mbit/s.

One of the interesting aspects regarding the implementation, is the time spend fetching data. The error correcting probabilities of turbo coding is build upon the availability of large amounts of data. This becomes a problem in the FPGA design as showed in chapter 7, as the availability of fast memory elements in the FPGA is limited by the number and size of its CLBs. In the case of this designs critical path, it was necessary to use memory interfaces to DDR2 RAM twice. This resulted in an increased execution time of the critical path by 6 ns, over twice the amount

of time spent in the control unit, and almost three times the amount of time spent in the two functional units. Hence it is another reason for investigating the possibilities of bringing down the necessary memory to store the variables.

For the hardware implementations to be applicable in the testing of EGPRS-2 communication systems, a throughput of nearly 2 Mbit/s is necessary. This is only achieved by the pipeline hardware implementation, but even without the pipeline, a hardware implementation provides almost sufficient throughput. However, as performance is essential in this design, the pipeline implementation is selected over the non-pipeline. Only one optimization technique was applied to the design in this report, so it should be possible to achieve even higher throughput rates. Other optimization techniques are discussed in future work.

Throughout the report the A^3 paradigm illustrated in fig. 8.1 is used as a tool to organize the report into interrelated domains, as well as illustrating the mapping from one domain to another. One issue with this paradigm is its lack of different abstraction levels in the architecture design phase. A meta-model that provides several abstraction levels is the Rugby meta-model. This helps the designer with a consistent design flow, and links different domains of design together at the same abstraction levels. The Rugby meta-model would therefore be a helpful tool in reaching the goal of establishing a framework for the process of mapping an application to a given platform. A more thorough explanation of the Rugby model is given in Future Work and the model is illustrated in fig. 8.2.

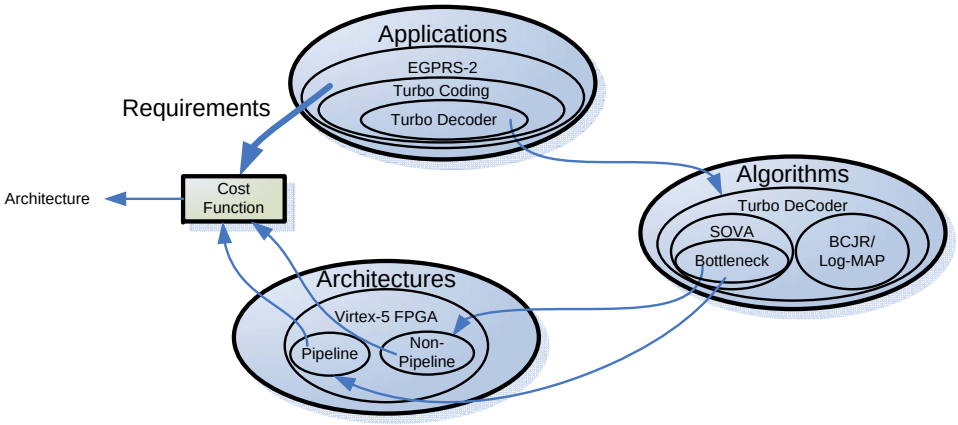


Figure 8.1: A^3 used throughout the report as a guideline in the design flow, resulting in the selection of a pipeline implementation.

This report showed that a hardware implementation, optimized with one method of performance optimization, is capable of providing a factor 1277 throughput improvement over a Matlab implementation and this is at a clock rate of 70 MHz compared to the 2.8 GHz of the Laptop running the Matlab code. These results are, however, based on rough estimates, and to assure that the synthesized design of this report is really capable of achieving these speeds, such that R&S may find it usable, some further work is needed. The immediate work necessary is described

in the next section, with a discussion of some additional improvements to the design, but also the design process.

8.2 Future Work

The throughput results of this report, is an estimate on whether or not the optimized hardware implementation of the SOVA algorithm, is capable of decoding at a rate of 2 Mbit/s. To establish if this hardware implementation really can achieve throughput as high as the estimated results, some additions to the already synthesized design are necessary.

First of all the output logic for the Moore FSM has not been established at this point. This should be fitted to each state of the datapath, so it provides the correct control signals in each state. Also a comparator design for FU2 is missing, and it should be specified before a implementation in VHDL can be done. With this VHDL code implemented on the Virtex-5 FPGA, it would be possible to test and find the real execution time for the Moore adapted algorithm. Should it reveal that the execution time is insufficient, there are some enhancements described below, that would be of interest.

As described in chapter 6.1 the input based (Mealy) FSM, reduced the number of necessary states by four compared to the Moore FSM. This reduction of states, is a trade-off for an increase in output logic and critical path of the datapath. It would, however, be interesting to see if this reduction of states, can match the increase of critical path and thereby provide a higher throughput. Other pipeline techniques as the one described in this report, would also be of interest to further increase the performance. [24] presents techniques for datapath and control unit pipelining. A pipeline implementation in both control unit and datapath will however, imply new states in the ASM chart and therefore affect the optimization techniques applied in this report, as these all originates from the state based ASM chart.

In the conclusion, memory allocation is mentioned as an issue in the critical path. To reduce the affect that reading and writing to external memory has on the critical path, some scheduling method might be of help. Instead of reading the variable from these slow memory elements in the state where it is used, it could be written to a fast register in one of the previous states. This is for example possible in the case of the critical path. Here the variable `Mdiff()` is read from an external memory, due to its size, but the exact value needed in `Mdiff()` could actually be written to an intermediate register several states before it is used. This may however, also increase the execution time of the algorithm, because whether or not `Mdiff()` is actually used in the algorithmic flow, depends on a decision box.

The A^3 model was used as a method for structuring the report, going from application and algorithmic analysis to hardware synthesis. For a one man project, concerning the implementation

of a small, but complex part of an SOVA decoder algorithm, this model showed to be helpful in organizing the project. One issue with this model, however, was the lack of abstraction levels. Had the project concerned an entire SOVA decoder algorithm, being implemented by a group of people in a HW/SW co-design, it would be advisable to use more complex meta-models. One example, that is developed specifically for HW/SW co-design, is the Rugby meta-model, proposed by A. Jantsch et al. [32]. The Rugby model starts out with an idea that spreads into four domains; Computation, Communication, Data, and Time, ending with a physical system. Each domain is divided into different abstraction levels, with the highest abstraction level at the start of a design process and lowest at the end. These different abstraction levels reminds the developer about abstraction levels that are necessary to consider through a design phase. The process of deciding the word length of the memory element containing the value of `Mdiff()`, is a good example of different abstraction levels in the data domain. Going from the decimal value of the variable in Matlab, to how it is affected by the SNR of the input. Figure 8.2 is an overview of the Rugby meta-model, where different abstraction levels would be vertical slices through this model. A specific framework for optimizing a HW/SW co-design should incorporate this meta-model to ensure all abstraction levels are considered under the design development.

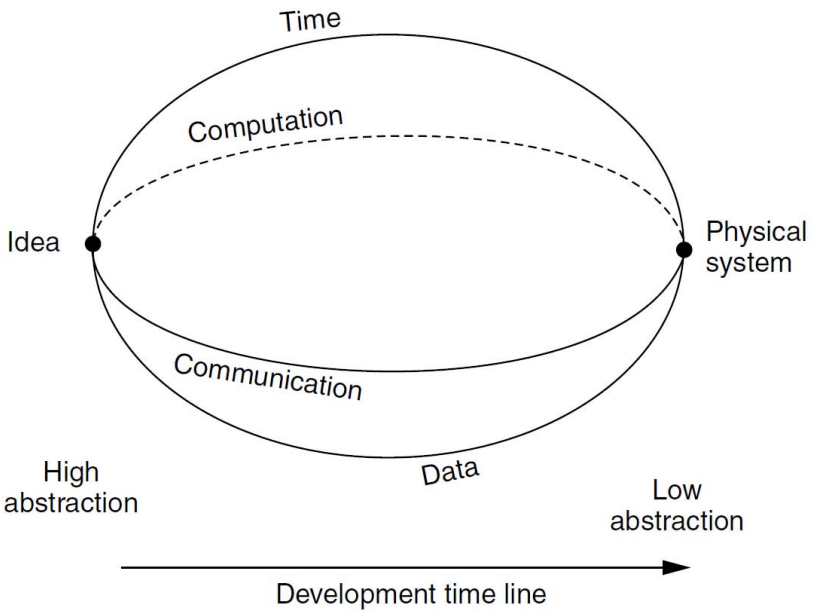


Figure 8.2: Rugby meta-model overview used in the a design phase from idea to physical system [32].

Bibliography

[1] http://www.engineeringvillage2.org, June 2009. Search for 3G and 4G.

[2] E. Dahlman, S. Parkvall, J. Sköld, and P. Beming, *3G Evolution HSPA and LTE for Mobile Broadband*. Elsevier Ltd., 2 ed., 2008.

[3] TDC, "Dækningskort." internet, 2009.

[4] A. B. Olsen, "DSP Algorithms and Architectures Minimodule 1 and 2." Slides, 2008.

[5] Rohde & Schwarz, *Protocol Test Platform R&S® CRTU*, 2008.

[6] Rohde & Schwarz, "R&S®CRTU Protocol Test Platform." internet, 2009.05.12.

[7] H. Axelsson, P. Björkén, P. de Bruin, S. Eriksson, and H. Persson, "GSM/EDGE continued evolution," 2006.

[8] J. Kjeldsen and M. Lauridsen, "Specific Emitter Identification Approach to Adaptive Interference Cancellation." Wireless@VT, 2009.

[9] C. Berrou and A. Glavieux, "Near optimum error correcting coding and decoding: turbo-codes," *Communications, IEEE Transactions on*, vol. 44, pp. 1261–1271, Oct 1996.

[10] M. Schwartz, *Mobile Wireless Communications*. Cambridge University Press, 2005.

[11] Ole Mikkelsen, et al., "Meetings with R&S." Oral, 2009.

[12] 3GPP, "3GPP TS 45.003 V7.5.0." Internet, 2008.

[13] S. Haykin, *Communication Systems*. John Wiley and Sons, Inc., 2001.

[14] Z. Wang and K. Parhi, "High performance, high throughput turbo/SOVA decoder design," *Communications, IEEE Transactions on*, vol. 51, pp. 570–579, April 2003.

[15] L. Sabeti, "New Design of a MAP Decoder." Slides, 2004.

[16] J. G. Proakis and M. Salehi, *Communication Systems Engineering*. Prentise Hall, 2002.

[17] J. G. Harrison, "Implementation of a 3GPP Turbo Decoder on a Programmable DSP Core." Internet, October 2nd, 2001.

[18] Y. Wu, "Copyright Nov 1998, Yufei Wu." internet, 1998.

[19] K. Popovski, T. Wysocki, and B. Wysocki, "Combined User Multiplexing and Data Modulation Through Non-Binary Turbo Codes for UWB," pp. 1038–1043, 31 2008-April 3 2008.

[20] M. Hata, E. Yamaguchi, Y. Hamasuna, T. Ishizaka, and I. Takumi, "High performance error correcting code of the high-dimensional discrete torus knot," pp. 547–552, Apr 2001.

[21] H. Abut, *DSP for In-Vehicle and Mobile Systems*. Springer, 1st ed., 2004.

[22] The Mathworks helpdesk, "profile." internet.

[23] Y. L. Moullec, "ASPI S2 11 HW/SW Co-design." Slides, 2008.

[24] D. D. Gajski, *Principles of Digital Design*. Prentice Hall International, Inc., 1997.

[25] J. F. Wakerly, *Digital Design Principles and Practices*. Prentice Hall International, Inc., 2001.

[26] http://www.dei.isep.ipp.pt/ ACC/bfunc/, June 2009. Download bfunc.zip.

[27] S. Nowick and D. Dill, "Exact two-level minimization of hazard-free logic with multiple-input changes," *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, vol. 14, pp. 986–997, Aug 1995.

[28] P. Lapsley, J. Bier, A. Shoham, and E. A. Lee, *DSP Processor Fundamentals*. Berkely Design Technology, Inc., 1994.

[29] M. Long, "Implementing Skein Hash Function on Xilinx Virtex-5 FPGA Platform," 2009.

[30] A. Cosoroaba and F. Rivoallon, "Achieving Higher System Performance with the Virtex-5 Family of FPGAs," 2006.

[31] XILINX, "Virtex-5 FPGA Data Sheet: DC and Switching Characteristics," 2009.

[32] A. Jantsch, S. Kumar, and A. Hemani, "A metamodel for studying concepts in electronic system design," *Design & Test of Computers, IEEE*, vol. 17, pp. 78–85, Jul/Sep 2000.

[33] W. H. Tranter, K. S. Shanmugan, T. S. Rappaport, and K. L. Kosbar, *Communication Systems Simulation with Wireless Applications*. Prentcie Hall PTR, 2004.

Appendix A

Viterbi Decoding Example

Figure A.1 illustrates a Viterbi decoding through trellis iteration of a simple convolutional code. The encoders state diagram is also depicted to show the possible transitions in the encoder and its output based on the encoder input. A punctured line represents a binary ”1” as input and a solid line represents a binary ”0” as input in both state and trellis diagram. Looking at the state diagram it should be noted that the numbers located on the transition edges are the outputs for one given input bit. This shows that the encoder is a rate 1/2 encoder, meaning for each input bit, two output bits are encoded. The numbers located at each dot in the state diagram corresponds to a state which is the same states located at the right end of the trellis diagram.

The following is a description of decoding the encoded bit sequence 0000000000 which after transmission is received as 0100010000, meaning noise i the transmission channel have corrupted two bits. The description follows the example illustrated in figure A.1 which is taken from [13, figure 10.17].

As explained in 4.2.1 on p. 27 Viterbi decoding works by calculating the Hamming distance between a bit sequence made possible by different paths through a trellis diagram and the received bit sequence. The Hamming distance between different paths are then compared and the path with lowest Hamming distance is chosen as the most likely bit sequence.

Before it is possible to exclude any paths, the Viterbi algorithm needs to reach a state in the trellis diagram where paths conjoin, giving the possibility to compare the Hamming distance for the paths in this state. In this case it is assumed that the encoder was terminated before transmission meaning that the decoding trellis should start at state ”00”. Note that in the encoder state diagram possible transitions from state ”00” leads to either ”11” or ”00” depending if the input to the encoder is ”1” or ”0” respectively. Note further that the first two received bits are ”01” indicating an error and increasing the Hamming distance for the two possible paths by 1, this value is noted at each state point. The two next received bits are ”00” which leads to the

Received bits	Start state	Transition edge	Output bits	End state	Increment in HD	Resulting HD
00	00	0	00	00	0	1
		1	11	10	2	3
	10	0	10	01	1	2
		1	01	11	1	2

Table A.1: Results from Viterbi decoding for second transition in figure A.1b.

following Hamming distances (HD) shown in table A.1.

This procedure is also done for the next step before paths can be excluded. Looking at the state points furthest right in figure A.1b two HDs are noted for each point. The paths leading to the highest HD is removed, in this case the HD = 3 for state ”00”, the HD = 3 for state ”10”, the HD = 5 for state ”01”, and the HD = 4 for state ”11”. The remaining paths are called survivor paths. The HD between the next set of received bits and the bits resulting from the next transitions in the trellis is calculated. Once again the highest HD at each state is removed and the Viterbi algorithm keeps on iterating in this way through the entire received block of bits. At the end all the survivor paths are compared and the one with lowest HD is chosen. Should the HD of two competing paths be of the same size at some point, a path is chosen randomly. The final survivor paths in figure A.1f shows that the bit sequence 0000000000 has the lowest HD = 2. This sequence is also the original encoder output showing that the Viterbi algorithm worked successfully. The Viterbi algorithm may however fail if additional errors are located in the bit sequence.

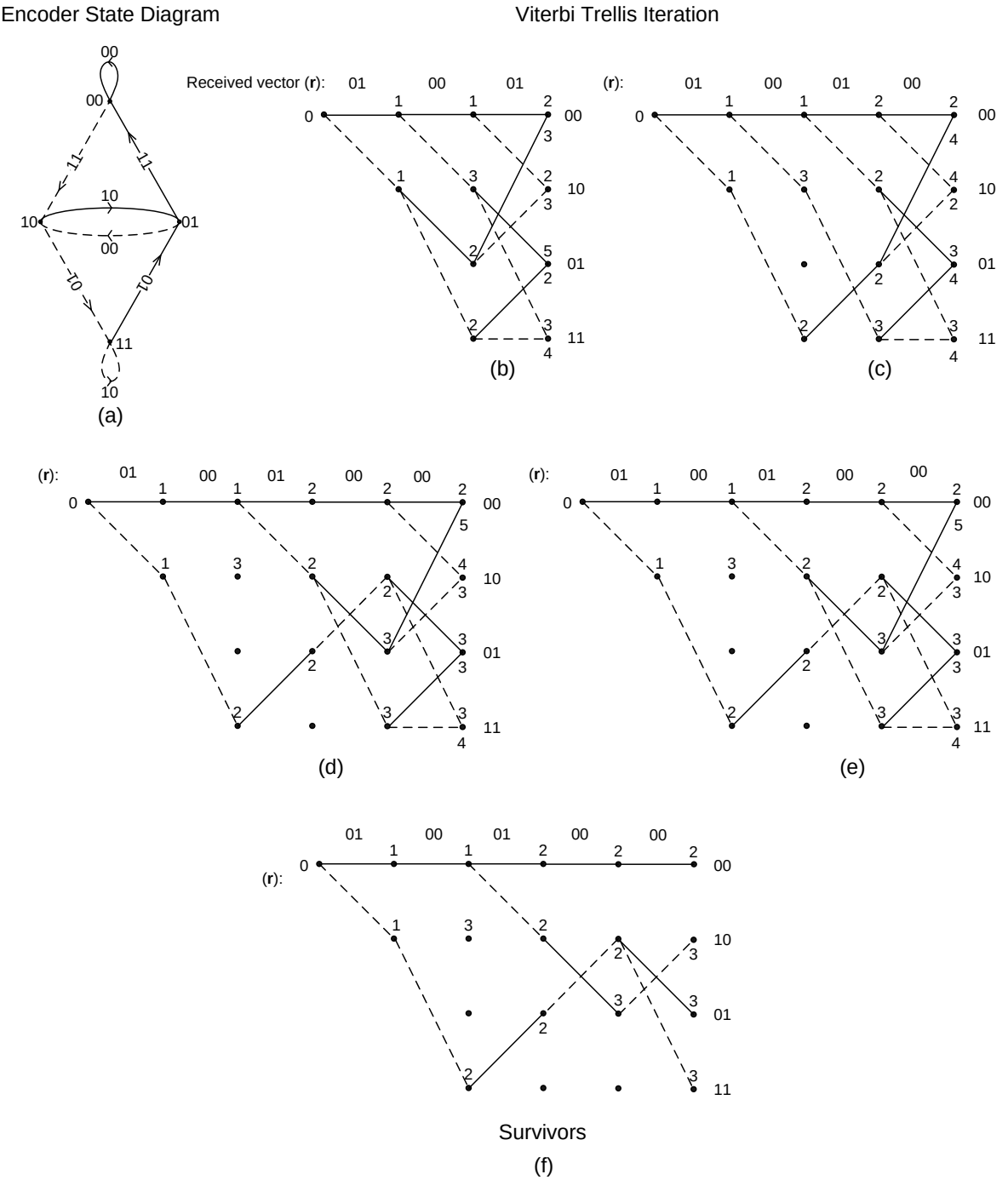


Figure A.1: Iterations done by the Viterbi algorithm to establish survivor paths based on a given encoder that leads to a decision for a bit sequence. Edited version of fig. 10.17 in [13]

Appendix B

SOVAturbo_sys_demo.m

The Matlab code listed in this appendix is a modified version of Yufei Wu's original code. Some explanatory text is added as well as making the code run automatically upon initialization.

```
% This script simulates the classical turbo encoding-decoding system.
% It simulates parallel concatenated convolutional codes.
% Two component rate 1/2 RSC (Recursive Systematic Convolutional) component
    encoders are assumed.
% First encoder is terminated with tails bits. (Info + tail) bits are scrambled and
    passed to
% the second encoder, while second encoder is left open without tail bits of itself.
%
% Random information bits are modulated into +1/-1, and transmitted through a
    AWGN channel.
% Interleavers are randomly generated for each frame.
%
% Log-MAP algorithm without quantization or approximation is used.
% By making use of ln(e^x+e^y) = max(x,y) + ln(1+e^(-abs(x-y))),
% the Log-MAP can be simplified with a look-up table for the correction function.
% If use approximation ln(e^x+e^y) = max(x,y), it becomes MAX-Log-MAP.
%
% Copyright Nov 1998, Yufei Wu
% MPRG lab, Virginia Tech.
% for academic use only

clear all

% Write display messages to a text file
diary turbo_logmap.txt

% Choose decoding algorithm
%dec_alg = input(' Please enter the decoding algorithm. (0:Log-MAP, 1:SOVA) default
    0 ');
```

```
%if isempty(dec_alg)
   dec_alg = 1;
%end

% Frame size
%L_total = input(' Please enter the frame size (= info + tail, default: 400) ');
%if isempty(L_total)
   L_total = 400;    % infomation bits plus tail bits
%end

% Code generator
%g = input(' Please enter code generator: ( default: g = [1 1 1; 1 0 1] )      ');
%if isempty(g)
   g = [1 1 1; 1 0 1];
%end
%default: g = [1 1 1; 1 0 1]
%g = [0 0 1 1 ; 1 1 0 1]; Turbo Encoder for EGPRS-2 1st row feedback 2nd
%row feedforward
%g = [1 1 0 1; 1 1 1 1];
%g = [1 1 1 1 1; 1 0 0 0 1];

[n,K] = size(g);
m = K - 1;
nstates = 2^m;

%puncture = 0, puncturing into rate 1/2;
%puncture = 1, no puncturing
%puncture = input(' Please choose punctured / unpunctured (0/1): default 0 ');
%if isempty(puncture)
    puncture = 1;
%end

% Code rate
rate = 1/(2+puncture);

% Fading amplitude; a=1 in AWGN channel
a = 1;

% Number of iterations
%niter = input(' Please enter number of iterations for each frame: default 5     ');
%if isempty(niter)
   niter = 1;
%end
% Number of frame errors to count as a stop criterior
%ferrlim = input(' Please enter number of frame errors to terminate: default 15     ');
%if isempty(ferrlim)
   ferrlim = 12;
%end
```

```
74
75  %EbN0db = input(' Please enter Eb/N0 in dB : default [2.0] ');
76  %if isempty(EbN0db)
77      EbN0db = [2];
78  %end
79
80  fprintf('\n\n--------------------------------------------------\n');
81  if dec_alg == 0
82      fprintf(' === Log-MAP decoder === \n');
83  else
84      fprintf(' === SOVA decoder === \n');
85  end
86  fprintf(' Frame size = %6d\n',L_total);
87  fprintf(' code generator: \n');
88  for i = 1:n
89      for j = 1:K
90          fprintf( '%6d', g(i,j));
91      end
92      fprintf('\n');
93  end
94  if puncture==0
95      fprintf(' Punctured, code rate = 1/2 \n');
96  else
97      fprintf(' Unpunctured, code rate = 1/3 \n');
98  end
99  fprintf(' iteration number = %6d\n', niter);
100 fprintf(' terminate frame errors = %6d\n', ferrlim);
101 fprintf(' Eb / N0 (dB) = ');
102 for i = 1:length(EbN0db)
103     fprintf('%10.2f',EbN0db(i));
104 end
105 fprintf('\n--------------------------------------------------\n\n');
106
107 fprintf('+ + + + Please be patient. Wait a while to get the result. + + + +\n');
108
109 for nEN = 1:length(EbN0db)
110     en = 10^(EbN0db(nEN)/10);      % convert Eb/N0 from unit db to normal numbers
111     L_c = 4*a*en*rate;  % reliability value of the channel
112     sigma = 1/sqrt(2*rate*en);  % standard deviation of AWGN noise
113
114 % Clear bit error counter and frame error counter
115     errs(nEN,1:niter) = zeros(1,niter);
116     nferr(nEN,1:niter) = zeros(1,niter);
117
118     nframe = 0;     % clear counter of transmitted frames
119     while nferr(nEN, niter)<ferrlim
120         nframe = nframe + 1;
121         x = round(rand(1, L_total-m)); % info. bits
```

```
122       [temp, alpha] = sort(rand(1,L_total));        % random interleaver mapping
123       en_output = encoderm( x, g, alpha, puncture ) ; % encoder output (+1/-1)
124
125       r = en_output+sigma*randn(1,L_total*(2+puncture)); % received bits
126       yk = demultiplex(r,alpha,puncture); % demultiplex to get input for decoder 1 and
              2
127
128   % Scale the received bits
129       rec_s = 0.5*L_c*yk;
130
131   % Initialize extrinsic information
132       L_e(1:L_total) = zeros(1,L_total);
133
134       for iter = 1:niter
135   % Decoder one
136           L_a(alpha) = L_e; % a priori info.
137           if dec_alg == 0
138              L_all = logmapo(rec_s(1,:), g, L_a, 1);   % complete info.
139           else
140              L_all = sova0(rec_s(1,:), g, L_a, 1);   % complete info.
141           end
142           L_e = L_all - 2*rec_s(1,1:2:2*L_total) - L_a; % extrinsic info.
143
144   % Decoder two
145           L_a = L_e(alpha); % a priori info.
146           if dec_alg == 0
147              L_all = logmapo(rec_s(2,:), g, L_a, 2);   % complete info.
148           else
149              L_all = sova0(rec_s(2,:), g, L_a, 2);   % complete info.
150           end
151           L_e = L_all - 2*rec_s(2,1:2:2*L_total) - L_a; % extrinsic info.
152
153   % Estimate the info. bits
154           xhat(alpha) = (sign(L_all)+1)/2;
155
156   % Number of bit errors in current iteration
157           err(iter) = length(find(xhat(1:L_total-m)'=x));
158   % Count frame errors for the current iteration
159           if err(iter)>0
160              nferr(nEN,iter) = nferr(nEN,iter)+1;
161           end
162       end   %iter
163
164   % Total number of bit errors for all iterations
165       errs(nEN,1:niter) = errs(nEN,1:niter) + err(1:niter);
166
167       if rem(nframe,3)==0 | nferr(nEN, niter)==ferrlim
168   % Bit error rate
```

```
169         ber(nEN,1:niter) = errs(nEN,1:niter)/nframe/(L_total-m);
170 % Frame error rate
171         fer(nEN,1:niter) = nferr(nEN,1:niter)/nframe;
172
173 % Display intermediate results in process
174         fprintf('************** Eb/N0 = %5.2f db **************\n', EbN0db(nEN));
175         fprintf('Frame size = %d, rate 1/%d. \n', L_total, 2+puncture);
176         fprintf('%d frames transmitted, %d frames in error.\n', nframe, nferr(nEN,
                niter));
177         fprintf('Bit Error Rate (from iteration 1 to iteration %d):\n', niter);
178         for i=1:niter
179            fprintf('%8.4e   ', ber(nEN,i));
180         end
181         fprintf('\n');
182         fprintf('Frame Error Rate (from iteration 1 to iteration %d):\n', niter);
183         for i=1:niter
184            fprintf('%8.4e   ', fer(nEN,i));
185         end
186         fprintf('\n');
187         fprintf('************************************************\n\n');
188
189 % Save intermediate results
190         save turbo_sys_demo EbN0db ber fer
191       end
192
193    end      %while
194 end         %nEN
195
196 diary off
```

Listing B.1: The main Matlab code for Yufei Wu's turbo encoder/decoder simulation script.

Appendix C

demultiplex.m

The Matlab code listed in this appendix is Yufei Wu's original code for the *demultiplex()* function. Some explanatory text is added.

```
function subr = demultiplex(r, alpha, puncture);
% Copyright 1998, Yufei Wu
% MPRG lab, Virginia Tech.
% for academic use only

% At receiver end, serial to paralle demultiplex to get the code word of each
% encoder
% alpha: interleaver mapping
% puncture = 0: use puncturing to increase rate to 1/2;
% puncture = 1; unpunctured, rate 1/3;

% Frame size, which includes info. bits and tail bits
L_total = length(r)/(2+puncture);

% Extract the parity bits for both decoders
if puncture == 1        % unpunctured
  for i = 1:L_total
      x_sys(i) = r(3*(i-1)+1); %the systematic bits are stored
      for j = 1:2 %1 for decoder 1 and 2 for decoder 2
          subr(j,2*i) = r(3*(i-1)+1+j); %the parity check bits are put in every even
              column
      end
   end
else              % punctured
   for i = 1:L_total
       x_sys(i) = r(2*(i-1)+1);
       for j = 1:2
          subr(j,2*i) = 0;
       end
```

```
29          if rem(i,2)>0
30              subr(1,2*i) = r(2*i);
31          else
32              subr(2,2*i) = r(2*i);
33          end
34      end
35  end
36
37  % Extract the systematic bits for both decoders
38  for j = 1:L_total
39  % For decoder one
40      subr(1,2*(j-1)+1) = x_sys(j); %systematic bits are put in every odd column
41  % For decoder two: interleave the systematic bits
42      subr(2,2*(j-1)+1) = x_sys(alpha(j)); %systematic bits are interleaved and put in
            every odd column
43  end
```

Listing C.1: The Matlab code for Yufei Wu's demultiplex function.

Appendix D

trellis.m

The Matlab code listed in this appendix is Yufei Wu's original code for the *trellis()* function. Some explanatory text is added.

```
1  function [next_out, next_state, last_out, last_state] = trellis(g)
2  % copyright Nov. 1998 Yufei Wu
3  % MPRG lab, Virginia Tech
4  % for academic use only
5
6  % set up the trellis given code generator g
7  % g given in binary matrix form. e.g. g = [ 1 1 1; 1 0 1 ];
8
9  % next_out(i,1:2): trellis next_out (systematic bit; parity bit) when input = 0, state = i
       ; next_out(i,j) = -1 or 1
10 % next_out(i,3:4): trellis next_out (systematic bit; parity bit) when input = 1, state =
       i;
11 % next_state(i,1): next state when input = 0, state = i; next_state(i,i) = 1,...2^m
12 % next_state(i,2): next state when input = 1, state = i;
13 % last_out(i,1:2): trellis last_out (systematic bit; parity bit) when input = 0, state = i;
        last_out(i,j) = -1 or 1
14 % last_out(i,3:4): trellis last_out (systematic bit; parity bit) when input = 1, state =
       i;
15 % last_state(i,1): previous state that comes to state i when info. bit = 0;
16 % last_state(i,2): previous state that comes to state i when info. bit = 1;
17
18 [n,K] = size(g);
19 m = K - 1;
20 max_state = 2^m;
21
22 % set up next_out and next_state matrices for systematic code
23 for state=1:max_state
24    state_vector = bin_state( state-1, m );
25
```

```
26      % when receive a 0
27      d_k = 0;
28      a_k = rem( g(1,:)*[0 state_vector]', 2 );
29      [out_0, state_0] = encode_bit(g, a_k, state_vector);
30      out_0(1) = 0;
31
32      % when receive a 1
33      d_k = 1;
34      a_k = rem( g(1,:)*[1 state_vector]', 2 );
35      [out_1, state_1] = encode_bit(g, a_k, state_vector);
36      out_1(1) = 1;
37      next_out(state,:) = 2*[out_0 out_1]-1; %conversion 0 = -1 and 1 = 1
38      next_state(state,:) = [(int_state(state_0)+1) (int_state(state_1)+1)];
39      %conversion of binary states into integer states
40  end
41
42  % find out which two previous states can come to present state
43  last_state = zeros(max_state,2);
44  for bit=0:1
45      for state=1:max_state
46          last_state(next_state(state,bit+1), bit+1)=state;
47          last_out(next_state(state, bit+1), bit*2+1:bit*2+2) ...
48              = next_out(state, bit*2+1:bit*2+2);
49      end
50  end
```

Listing D.1: The Matlab code for Yufei Wu's *trellis* function.

Appendix E

sova0.m

The Matlab code listed in this appendix is Yufei Wu's original code for the *sova()* function. Some explanatory text is added.

```
function L_all = sova(rec_s, g, L_a, ind_dec)
% This function implememts Soft Output Viterbi Algorithm in trace back mode
% Input:
%       rec_s: scaled received bits. rec_s(k) = 0.5 * L_c(k) * y(k)
%              L_c = 4 * a * Es/No, reliability value of the channel
%              y: received bits
%       g:  encoder generator matrix in binary form, g(1,:) for feedback, g(2,:) for
    feedforward
%       L_a: a priori information about the info. bits. Extrinsic info. from the previous
%            component decoder
%       ind_dec: index of the component decoder.
%              =1: component decoder 1; The trellis is terminated to all zero state
%                  =2: component decoder 2; The trellis is not perfectly terminated.
% Output:
%       L_all: log ( P(x=1|y) ) / ( P(x=-1|y) )
%
% Copyright: Yufei Wu, Nov. 1998
% MPRG lab, Virginia Tech
% for academic use only

% Frame size, info. + tail bits
L_total = length(L_a);
[n,K] = size(g); %generator matrix from demo file
m = K - 1;       %shiftregisters in encoder
nstates = 2^m;   %number of states
Infty = 1e10;

% SOVA window size. Make decision after 'delta' delay. Decide bit k when received bits
% for bit (k+delta) are processed. Trace back from (k+delta) to k.
```

```
delta = 30;

% Set up the trellis defined by g.
[next_out, next_state, last_out, last_state] = trellis(g);

% Initialize path metrics to -Infty
for t=1:L_total+1
   for state=1:nstates
      path_metric(state,t) = -Infty;
   end
end

% Trace forward to compute all the path metrics
path_metric(1,1) = 0;
for t=1:L_total
   y = rec_s(2*t-1:2*t);
   for state=1:nstates
      sym0 = last_out(state,1:2);
      sym1 = last_out(state,3:4);
      state0 = last_state(state,1);
      state1 = last_state(state,2);
      Mk0 = y*sym0' - L_a(t)/2 + path_metric(state0,t);
      Mk1 = y*sym1' + L_a(t)/2 + path_metric(state1,t);

      if Mk0>Mk1
         path_metric(state,t+1)=Mk0;
         Mdiff(state,t+1) = Mk0 - Mk1;
         prev_bit(state, t+1) = 0;
      else
         path_metric(state,t+1)=Mk1;
         Mdiff(state,t+1) = Mk1 - Mk0;
         prev_bit(state,t+1) = 1;
      end

   end
end

% For decoder 1, trace back from all zero state,
% for decoder two, trace back from the most likely state
if ind_dec == 1
   mlstate(L_total+1) = 1; %As the first decoder always starts from zero state
else
   mlstate(L_total+1) = find( path_metric(:,L_total+1)==max(path_metric(:,L_total
        +1)) );
end

% Trace back to get the estimated bits, and the most likely path
for t=L_total:-1:1
```

```
76    est(t) = prev_bit(mlstate(t+1),t+1);
77    mlstate(t) = last_state(mlstate(t+1), est(t)+1);
78 end
79
80 % Find the minimum delta that corresponds to a compitition path with different info.
       bit estimation.
81 % Give the soft output
82 for t=1:L_total
83    llr = Infty;
84    for i=0:delta %delta is the traceback search window which should be between 5 and
          9 times K
85       if t+i<L_total+1
86          bit = 1-est(t+i); %inverting the estimated bits 0=1 1=0
87          temp_state = last_state(mlstate(t+i+1), bit+1); %opposite state is chosen
                compared to line 77
88          for j=i-1:-1:0
89             bit = prev_bit(temp_state,t+j+1);
90             temp_state = last_state(temp_state, bit+1);
91          end
92          if bit~=est(t)
93             llr = min( llr,Mdiff(mlstate(t+i+1), t+i+1) );
94          end
95       end
96    end
97    L_all(t) = (2*est(t) - 1) * llr; %converted to -1 and 1 and multiplied with Mdiff
98 end
```

Listing E.1: The Matlab code for Yufei Wu's *sova0()* function.

Appendix F

Truth Tables for Next State Equations

The following four tables are truth tables for the state based next state equations in eq. 6.1. These are used in the Quine-McCluskey (QM) algorithm to reduce the next state equations to the ones stated in eq. 6.2.

D_3	Q_3	Q_2	Q_1	Q_0	x ≤ L_total	j ≥ 0	i > delta	bit ≠ est(t)	t > L_total
0	0	0	0	0	x	x	x	x	x
0	0	0	0	1	x	x	x	x	x
1	0	0	1	0	0	x	x	x	x
0	0	0	1	1	x	x	x	x	x
0	0	1	0	0	x	x	x	x	x
1	0	1	0	1	x	0	x	x	x
0	0	1	1	0	x	x	x	x	x
1	0	1	1	1	x	x	x	x	x
1	1	0	0	0	x	1	x	x	x
1	1	0	0	1	x	x	x	x	x
1	1	0	1	0	x	x	x	x	x
1	1	0	1	1	x	x	1	x	x
1	1	1	0	0	x	x	x	x	x
0	1	1	0	1	x	x	x	x	x

Table F.1: Truth table for next state equation D_3.

D_2	Q_3	Q_2	Q_1	Q_0	x ≤ L_total	j ≥ 0	i > delta	bit ≠ est(t)	t > L_total
0	0	0	0	0	x	x	x	x	x
0	0	0	0	1	x	x	x	x	x
0	0	0	1	0	x	x	x	x	x
1	0	0	1	1	x	x	x	x	x
1	0	1	0	0	x	x	x	x	x
1	0	1	0	1	x	1	x	x	x
1	0	1	1	0	x	x	x	x	x
0	0	1	1	1	x	x	x	x	x
1	1	0	0	0	x	0	x	x	x
0	1	0	0	1	x	x	x	x	x
0	1	0	1	0	x	x	x	x	x
1	1	0	1	1	x	x	1	x	x
1	1	1	0	0	x	x	x	x	x
0	1	1	0	1	x	x	x	x	x

Table F.2: Truth table for next state equation D_2.

D_1	Q_3	Q_2	Q_1	Q_0	x ≤ L_total	j ≥ 0	i > delta	bit ≠ est(t)	t > L_total
0	0	0	0	0	x	x	x	x	x
1	0	0	0	1	x	x	x	x	x
1	0	0	1	0	x	x	x	x	x
0	0	0	1	1	x	x	x	x	x
0	0	1	0	0	x	x	x	x	x
1	0	1	0	1	x	0	x	x	x
1	0	1	1	0	x	x	x	x	x
0	0	1	1	1	x	x	x	x	x
1	1	0	0	0	x	0	x	x	x
1	1	0	0	1	x	x	x	x	x
1	1	0	1	0	x	x	x	x	x
1	1	0	1	1	x	x	0	x	x
1	1	1	0	0	x	x	x	x	x
0	1	1	0	1	x	x	x	x	x

Table F.3: Truth table for next state equation D_1.

D_0	Q_3	Q_2	Q_1	Q_0	x ≤ L_total	j ≥ 0	i > delta	bit ≠ est(t)	t > L_total
1	0	0	0	0	x	x	x	x	x
0	0	0	0	1	x	x	x	x	x
1	0	0	1	0	x	x	x	x	x
0	0	0	1	1	x	x	x	x	x
1	0	1	0	0	x	x	x	x	x
1	0	1	0	1	x	0	x	x	x
1	0	1	1	0	x	x	x	x	x
0	0	1	1	1	x	x	x	x	x
1	1	0	0	0	x	1	x	x	x
1	1	0	0	1	x	x	x	0	x
1	1	0	1	0	x	x	x	x	x
0	1	0	1	1	x	x	x	x	x
1	1	1	0	0	x	x	x	x	x
1	1	1	0	1	x	x	x	x	0

Table F.4: Truth table for next state equation D_0.

Appendix G

Evaluation of Fixed Point Precision

To determine the required word length for the variables `Mdiff()`, `L_all()`, and `llr()` in a hardware implementation of the *sova0.m* algorithm, a quantizer was build in Matlab. In Matlab the function `quantizer()` constructs a quantizer object with the parameters set in the parentheses. An example would be `quantizer('fixed', 'floor', 'saturate', [27 0])`, where `'fixed'` means the quantizer object will use a signed fixed point arithmetic representation (such as that of figure 6.11b on p. 58). `'floor'` sets the round mode of the quantizer object to round towards minus infinity resembling a truncation of the LSB in hardware. The quantizers overflow mode is set to `'saturate'`, which means the quantizer will represent a value higher than what is possible to represent with the given word length, as the highest possible value of the given word length. The last property of `'quantizer'` sets the word length and fraction length for the quantizer object. [22]

The Monte Carlo technique is used to estimate the BER of the different setups. Here N number of samples are passed through the simulation model, comparing each sample at the end to determine the number of errors N_e. This leads to an estimation of the BER given by:

$$\hat{P}_E = \frac{N_e}{N} \tag{G.1}$$

Where the true error probability is:

$$P_E = \lim_{N \to \infty} \frac{N_e}{N} \tag{G.2}$$

Showing that the estimated error probability reaches the true error probability as N goes to infinity. For the plots in fig. G.2 number of samples was set to N = 10,000. It turns out, however, that this is not a sufficiently high number of samples for large E_b/N_0 values. This is seen in the plots as the increased variance of the BER curve as E_b/N_0 is increased.

Instead of increasing the number of samples processed by the simulation to one arbitrary extremely high value, there is another way around the problem of a sufficient number of samples. In [33, chapter 10] N is based on the theoretical error probability (P_T) of an AWGN channel. Here the N is calculated as a K/P_T, where K is the number of observed errors. E.g. if $P_T = 0.5$ - which is based directly on the value of E_b/N_0 - and 20 errors is needed to provide a reliable BER estimate, 40 samples have to be processed. As P_T is based on the theoretical error probability, the number of errors in the simulation achieved by setting $N = K/P_T$ will be higher than K, as decoding errors will add to the number of errors.

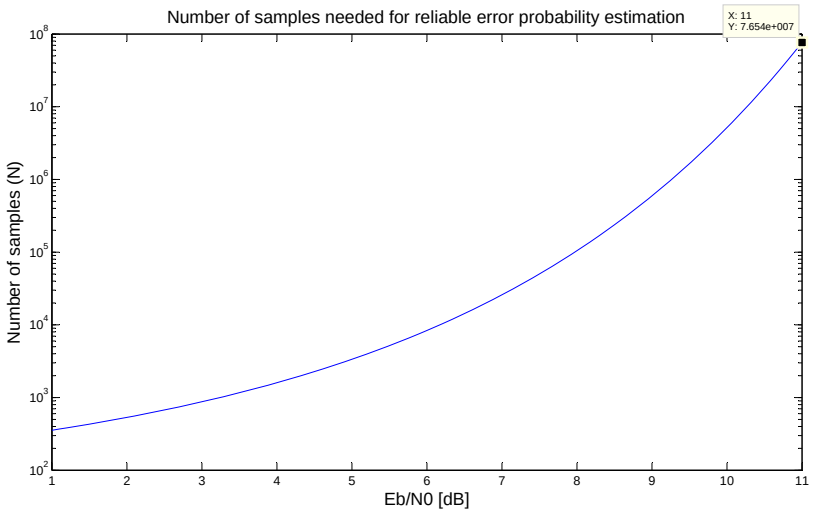


Figure G.1: The number of samples needed to obtain a reliable BER increases exponentially as E_b/N_0 is increased. But note also how few samples are needed for low E_b/N_0 values.

So why was this method not used for the simulations investigating the necessary word length? The main reason is time. The number of samples necessary to obtain a reliable BER estimate increases exponentially as E_b/N_0 increases, as illustrated in fig. G.1. If 20 errors is set as the necessary amount of errors for each value of E_b/N_0, 76,538,385 samples would have to be simulated at 11 dB. E_b/N_0 ranges from 1 to 11 dB in steps of 0.1 in the plots of fig. G.2a - G.2f and would require a total of 317,254,892 samples for each simulation instead of the 1,010,000 used. Should this method be used it is advised to lower the range and increase the stepsize, as this will greatly reduce N. Another reason is that these simulations where not done to establish the performance of the SOVA decoder, but to determine the size of some internal registers for a hardware implementation. The number of samples does not affect the different error rates, between the double and fixed point precision results and it is therefore save to base the decision of word length from the results illustrated in the following figures.

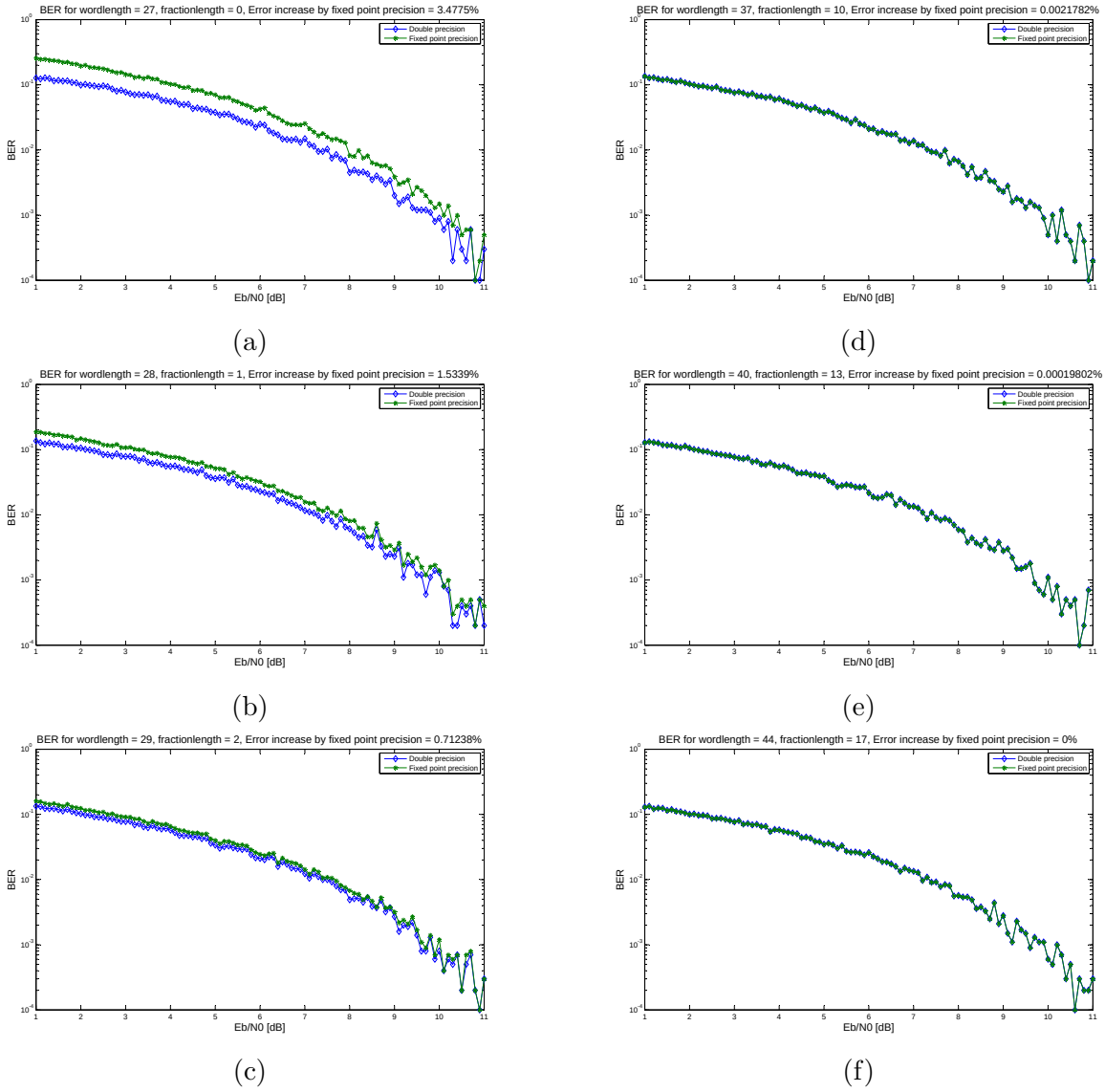
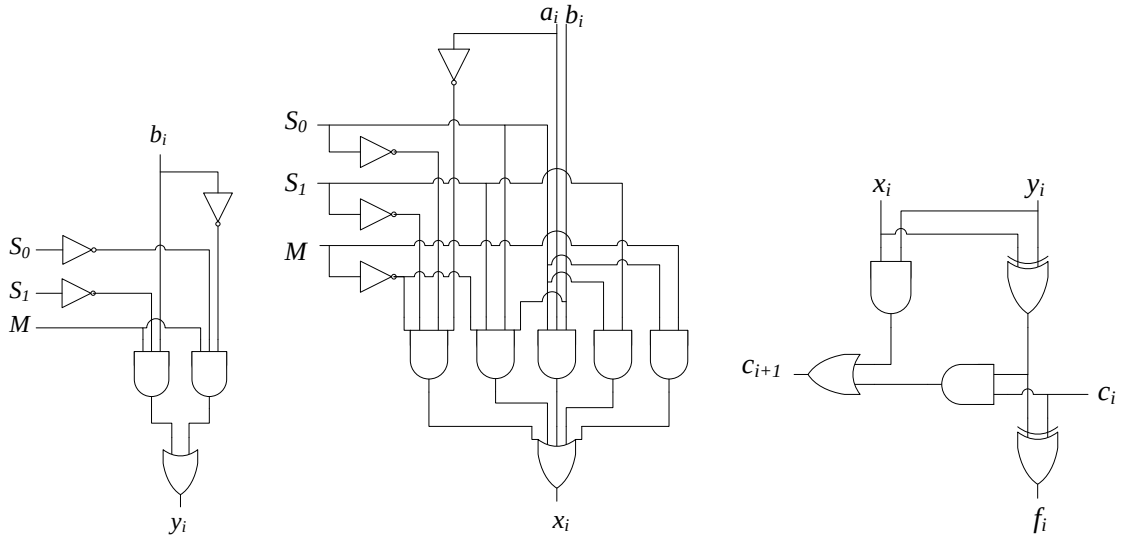


Figure G.2: (a) A word length of 27 bit resulted in 35,123 errors out of 1,010,000 samples compared to double precision. (b) Here the quantized simulation caused 15,492 more errors reducing the number of errors by more than one half. (c) Once again an increase of the fraction length by 1 more than halved the number of errors = 7,195. (d) The word length is now 37 leading to a fraction length of 10 and the quantizer now only introduces 22 errors. (e) Here the quantizer only produce 2 errors more than Matlabs double precision. (f) To reach the goal of no error increase, a word length 44 and fraction length of 17 is needed. A fairly big increase of 4 bits to remove 2 errors.

Appendix H

Combinational Logic for ALU Design

The combinational circuits necessary for implementing an ALU capable of performing the operations of FU1 given in chapter 6.3.2, is illustrated in figure H.1a to H.1c. Their corresponding truth tables are listed in table H.1 to H.3.



(a) (b) (c)

Figure H.1: (a) Combinational logic for the arithmetic extender (AE). (b) Combinational logic for the logic extender (LE). (c) Combinational logic for the full adder (FA). [24, chapter 5]

M	S_1	S_0	b_i	y_i
1	0	0	0	1
1	0	0	1	1
1	0	1	0	0
1	0	1	1	1
1	1	0	0	1
1	1	0	1	0
1	1	1	0	0
1	1	1	1	0

Table H.1: Arithmetic extender truth table [24, chapter 5].

M	S_1	S_0	y_i
0	0	0	a_i'
0	0	1	$a_i b_i$
0	1	0	a_i
0	1	1	$a_i + b_i$
1	X	X	a_i

Table H.2: Arithmetic extender truth table [24, chapter 5].

x_i	y_i	c_i	c_{i+1}	f_i
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

Table H.3: Full adder truth table [24, chapter 5].

Appendix I

Pipeline Timing Diagrams

In this appendix the timing diagram for the non-pipeline design is presented with the timing diagram with the pipeline design. Each row in these diagrams represents a read or write of a specific memory element or an operation done by the functional units. Each column represents the state of the control unit and indicates the operation done by the datapath in a given state.

Timing diagram without pipelining

	s0	s1	s2	s3	s4	s5	s6	s7	s8	s9	s10	s11	s12	s13
Read ME1							j		j		Mdiff()	delta		
Read ME2			t				t			t			t	t
Read ME3											llr		llr	
Read ME4			i			i						i		
Read ME5			L_total											L_total
Read ME6			x	x	x						x			
Read ME7				est()						est()			est()	
Read ME8					mlstate()						mlstate()			
Read ME9					bit			bit		bit				
Read ME10					last_state()			last_state()						
Read ME11							temp_state	temp_state						
Read ME12							prev_bit()							
FU1			t+i	**inv(est())**	x+1	i-1	t+j+1		j-1	bit≠est	x+1	i+1	**sign**	t+1
FU2			x≤L_total			j≥0			j≥0		**min**	i>delta		t>L_total
write ME1						j			j	Mdiff()	delta		L_all()	
write ME2	t													t
write ME3		llr									llr			
write ME4		i										i		
write ME5		L_total												
write ME6			x											
write ME7			est()											
write ME8				mlstate()										
write ME9				bit			bit							
write ME10				last_state()										
write ME11					temp_state			temp_state						
write ME12						prev_bit()								

Figure I.1: Timing diagram for the non-pipeline hardware implementation.

The timing diagram for the non-pipeline design is just a tabular representation of the Moore ASM chart in 6.5. This table is inserted here to ease the comparison of the two designs.

Note from table I.2 the representation for each state. E.g. one state is marked (s6-1 s-7 s8-0). This is done to illustrate that the last part of state 6, the entire state 7, and the first part of state 8, is executed in one state. A possibility as the operation of state 7 does not rely on the result of state 6 and state 8 is independent of the results from both state 6 and 7. With this merging of states there is actually no increase of states from the non-pipeline design for state 6 to 8.

Other parts of state merging is seen for state 3, 4, and 5, as well as for state 12 and 13. Here however, the pipeline causes an increase of one state, so it takes four states to compute state 3-5 and three states to compute state 12 and 13.

Overall the number of states is increased by six, but if the pipelining of FU1 is done in the exact middle of the critical path, dividing this in two paths of equal latency. It is possible to increase the clock frequency by two and thereby bring down both the latency and throughput of hardware implementation.

Timing diagram with pipelining

	s0	s1	s2-0	s2-1	s3-0	s3-1 s4-0	s4-1 s5-0	s5-1	s6-0	s6-1 s-7 s8-0
Read ME1									j	j
Read ME2			t						t	
Read ME3										
Read ME4			i				i			
Read ME5				L_total						
Read ME6				x	x	x	x			
Read ME7					est()					
Read ME8							mlstate()			
Read ME9							bit			bit
Read ME10							last_state()			last_state()
Read ME11										temp_state
Read ME12										prev_bit()
FU 1 stage 1			t+i		**inv(est())**	x+1	i-1		t+j+1	j-1
FU 1 stage 2				t+i		**inv(est())**	x+1	i-1		t+j+1
FU2				x≤L_total				j≥0		
write ME1								j		
write ME2	t									
write ME3		llr								
write ME4		i								
write ME5		L_total								
write ME6				x						
write ME7				est()						
write ME8						mlstate()				
write ME9						bit				bit
write ME10						last_state()				
write ME11							temp_state			temp_state
write ME12								prev_bit()		prev_bit()

	s8-1	s9	s10-0	s10-1	s11-0	s11-1	s12-0	s12-1 s13-0	s13-1
Read ME1				Mdiff()		delta			
Read ME2		t					t	t	t
Read ME3				llr			llr		
Read ME4					i				
Read ME5									L_total
Read ME6			x						
Read ME7		est()					est(t)		
Read ME8				mlstate()					
Read ME9		bit							
Read ME10									
Read ME11									
Read ME12									
FU 1 stage 1			x+1		i+1		**sign**	t+1	
FU 1 stage 2	j-1			x+1		i+1		**sign**	t+1
FU2	j≥0	bit≠est(t)		**min**		i>delta			t>L_total
write ME1	j	Mdiff()		delta					t
write ME2				llr				L_all()	
write ME3						i			
write ME4									
write ME5									
write ME6									
write ME7									
write ME8									
write ME9									
write ME10									
write ME11									
write ME12									

Figure I.2: Timing diagram for the FU1 pipeline hardware implementation.