Universidade de Aveiro 2008

Departamento de Electrónica, Telecomunicações e Informática.

André Tavares Coutinho

Análise de DRM num Simulador de Sistema Embutido Multiprocessador

DRM Analysis Using a Simulator of a Multiprocessor Embedded System

Universidade de Aveiro 2008

Departamento de Electrónica, Telecomunicações e Informática

André Tavares Coutinho

Análise de DRM num Simulador de Sistema Embutido Multiprocessador

Dissertação apresentada à Universidade de Aveiro para cumprimento dos requisitos necessários à obtenção do grau de Mestre em Análise de DRM num Simulador de Sistema Embutido Multiprocessador, realizada sob a orientação científica do Dr. José Nuno Panelas Nunes Lau, Professor Auxiliar do Departamento de Electrónica e Telecomunicações da Universidade de Aveiro.

o júri

presidente

Prof. Dr. António Manuel de Brito Ferrari Almeida
Professor Catedrático da Universidade de Aveiro

Prof. Dr. Luís Filipe Santos Gomes
Professor Associado do Departamento de Engenharia Electrotécnica da Faculdade de Ciências e Tecnologia da Universidade Nova de Lisboa

Prof. Dr. José Nuno Panelas Nunes Lau (Orientador)
Professor Auxiliar da Universidade de Aveiro

Acknowledgements

This report is the result of my internship at *NXP Semiconductors research*. I would like to thank the Hijdra project leader Dr Marco Bekooij and my advisor Orlando Moreira for all the support given to me, and I would like to give special thanks to Jan Willen which helped me a lot and spent a lot of his time during my entire stay at NXP and Maarten Wiggers which also helped me a lot on the last month of the project.

palavras-chave

resumo

Os sistemas multiprocessador são uma tecnologia emergente. O projecto Hijdra, que está a ser desenvolvido na "NXP semiconductors Research" é um sistema multiprocessador de tempo real que corre aplicações com constrangimentos do tipo "hard" e "soft". Nestes sistemas, os processadores comunicam através de uma rede de silício. As aplicações que correm no sistema multiprocessador consistem em múltiplas tarefas que correm em processadores embutidos. Achar soluções para o mapeamento das tarefas é o maior problema destes sistemas. Uma aplicação para este sistema que tem vindo a ser estudada é o "Car Radio".

Esta dissertação diz respeito a uma aplicação de rádio digital (DRM) na arquitectura Hijdra. Neste contexto, uma aplicação de um receptor de DRM foi estudada. Um modelo de análise de "Data Flow" foi extraído a partir da aplicação, foi estudada a latência introduzida na rede de silício pela introdução de um novo processador (acelerador de Viterbi) e foi estudada a possibilidade do mapeamento das várias tarefas da aplicação em diferentes processadores a correr em paralelo.

Muitas estratégias ainda ficaram por definir a fim de optimizar o desempenho da aplicação do receptor de DRM de modo a esta poder trabalhar de uma forma mais eficaz.

keywords

abstract

Multiprocessor systems are an emerging technology. The Hijdra project being developed at *NXP semiconductors Research* is a multiprocessor system running with both hard and soft real time streaming media jobs. These jobs consist of multiple tasks running on embedded multiprocessors. Finding good solutions for job mapping is the main problem of these systems. One application which has being studied for Hijdra is the “Car Radio”.

This thesis concerns the study of a digital radio receptor application (DRM) in Hijdra architecture. In this context, a data flow model of analysis was extracted from the application, the latency introduced by the addition of a new tile (Viterbi accelerator) and eventual speed gains were studied and the possibility of mapping the different tasks of the application in different processors was foreseen.

Many strategies were yet to be defined in order to optimize the application performance so it can work more effectively in the multiprocessor system.

Index

List of Figures

Glossary

AAC	Advanced Audio Coding
AM	Amplitude Modulation
BER	Bit Error Rate
CELP	Code Excited Linear Prediction for speech
CA	Communication assist
CPU	Central Process Unit
DRM	Digital Radio Mondiale
DSP	Digital Signal Processing
EEP	Equal Error Protection
FAAC	Freeware Advanced Audio Decoder
FAC	Fast Access channel
FFT	Fast Fourier Transform
FM	Frequency Modulation
GSM	Global System for Mobile Communications
IFFT	Inverse Fast Fourier Transform
ITU	International Telecommunications Union
MF	Medium-frequency
MPEG	Moving Picture Experts Group
MPSoC	Multi-Processor System-on-Chip
MSC	Main service channel
NI	Network interface
NoC	Network-on-Chip
OFDM	Orthogonal Frequency Division Multiplex
QAM	Quadrature Amplitude Modulation
RDS	Radio Data System
SBR	Spectral Band Replication
SDC	Service Description channel
TCM	Trellis Coded Modulation
UEP	Unequal Error Protection

1 Introduction

1.1 Background and Context

Hijdra project being developed at *NXP Semiconductors Research* (formerly Philips Research) has the objective of developing design and temporal analysis techniques for embedded multiprocessors systems based on silicon networks with guarantees of real-time performance [16]. Silicon networks allow establishing communication channels with maximum latency and available bandwidth guarantees between processes loaded on different processors. One of the objectives of the project consists on the development of a system simulator in a software environment and on the analysis of the performance of Car Radio applications. One of the applications running on the Car Radio simulator is DRM (Digital Radio Mondiale) receptor. The DRM software receiver was provided by Catena, and it is a complex application which accepts as inputs OFDM (Orthogonal frequency-division multiplexing) [18] symbols and produces AAC (Advanced Audio Coder) frames as output. To do that, a multistage decoder [8] which includes a Viterbi decoder [19] is used. Hijdra Architecture [1], explained on Chapter 2 is a multiprocessor which includes several tiles. DRM application runs on a TriMedia [20] processor. Through cycle counting, it was able to verify that Viterbi decoding is the most cycle consuming task on DRM Receiver. The gain of having Viterbi decoding running on a different tile (Viterbi accelerator) can be substantial because TriMedia can be running other tasks while Viterbi is executing. The addition of a new tile can be also profitable due to the fact that many other digital signal processing applications also use Viterbi decoder. The Hijdra Architecture is based on Æthereal NoC (Network-on-Chip) [2] which is a Silicon network, that has latency and throughput constrains. Although speed gain can be achieved by parallelizing the software code, this may not be enough to compensate the time wasted on passing the information through the silicon network.

The principal aim of this dissertation is to study the viability of the implementation of a Viterbi accelerator in the Car Radio Simulator.

1.2 Scope and Objectives

One of the problems to be resolved in a multiprocessor system is the choice of processing units. One of the applications that can be implemented in these kinds of systems is a Digital Radio Mondiale

(DRM) receptor. Although there is already one implementation of this application running on TriMedia, there are important issues that need answers:

1 - What is the application performance when communication between processor caches and system memories is performed through a silicon network? Does the extra latency introduced by the silicon network affect too much the performance?

2 - Will it be worth, from the performance of the system point of view, to use a hardware accelerator for Viterbi decoder, taking into account the communication time between the TriMedia and the accelerator?

3 – As the simulator has real-time constrains, the worst case performance is as much or even more relevant than the medium case observed in the simulator. What is the performance worst case (in different mapping schemes) and what is the difference between the worst case and medium case?

1.3 Achievements

To answer the questions above it was necessary to develop an extension of the simulator with a Viterbi accelerator, to re-write the DRM application to allow multiple mapping solutions, and to obtain a temporal analysis model to compare worst case with simulator performance. In this context, Hijdra architecture and the code of DRM application were studied. The following text resumes the work done:

- *Advanced Audio Coder* (AAC) decoder was attached to DRM application;
- Viterbi decoder accelerator was added to the multiprocessor system simulator;
- DRM application code was re-written in Khan Processes Network allowing the exploration of mapping parallelism and the extraction of a data flow model necessary to proceed to the temporal analysis of the system;
- A *Cyclo-Static data flow* (CSDF) analysis model of the Khan Processes Network implementation was extracted;
- Simulations were made with DRM application assuming different mapping schemes of Viterbi decoder task.

1.4 Overview of Dissertation

The organization of this dissertation is as follows. A chapter of theory background (chapter 2) was included, with the most relevant information of Multiprocessor systems, the Car Radio Simulator, Cyclo-Static and Synchronous data flow, DRM and AAC decoder. The most relevant work is described on Chapter 3, results are presented on chapter 4 and conclusions on chapter 5.

Detailed information on DRM encoding/decoding and multilevel encoding and multistage decoding can be found on Appendix A. Information of Viterbi decoder can be found in Appendix B. Appendix C and D are the user and installation guide. On the user guide, information about the code developed and respective folders, how results were measured and information on the files containing these measurements can be consulted.

2 Background

Consumer demand on the latest in-car infotainment applications ranging from stereo, navigation, MP3 players to video entertainment, is increasingly driving the use of electronics in cars. Future car radio systems will be able to process different types of data streams, and receive a number of terrestrial and satellite data channels simultaneously.

Modern multimedia applications are becoming more complex and demanding more processing power. In many cases it can be more efficient to combine several average-speed processor cores and divide the workload between them to achieve a high performance. A Multiprocessor System on Chip (MPSoC) is an approach that combines several tiles on one single chip. Tiles can contain general purpose processors, digital signal processors, application domain processors and memories. Communications between tiles are made by a Network-on-Chip (NoC). These networks contain routers, network interfaces and network links. NoC's can deliver a scalable and flexible communication infrastructure. Hijdra Project [1], developed at NXP Eindhoven, is a MPSoC built over Æthereal NoC [2] with the objective to design a scalable and predictable system. Within the Hijdra and Æthereal projects, a car radio simulation environment which allows cycle accurate simulation was studied.

2.1 Multiprocessor System on Chip and Network on Chip

MPSoC allows the integration of a heterogeneous mix of processing and memory components on a single chip. On a MPSoC, communications between processors are implemented on a NoC. NoC is an emerging paradigm for communications within systems implemented on a single silicon chip. Modules such as processor cores, memories and specialized IP blocks (Intellectual Property Core), exchange data using a network as a "public transportation" sub-system for the information traffic. A NoC is constructed from multiple point-to-point data links interconnected by switches (routers), such that messages can be sent from any source module to any destination module over several links by making routing decisions at the switches. It is similar to a modern telecommunications network, using digital bit-packet switching over multiplexed links. Wires in the links are shared by many signals and a high level of parallelism is achieved because all links in the NoC can operate simultaneously on different data packets. As the complexity of integrated systems keeps growing, a NoC provides enhanced performance such as throughput and scalability in comparison with previous communication architectures (e.g., dedicated point-to-point signal wires, shared buses, or segmented buses with bridges). Algorithms must be designed in a way that

they offer large parallelism maximizing the potential of the NoC. In this way it can provide a flexible, scalable, and predictable on-chip communication infrastructure.

Hijdra MPSoC architecture template is illustrated on Figure 1.

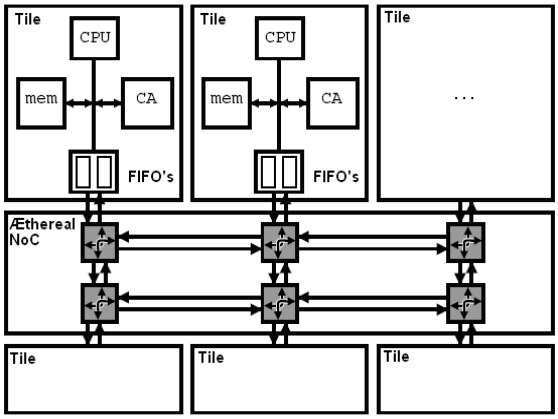


Figure 1. Hijdra architecture template

2.2 Car radio simulator

The car radio simulator is based on Hijdra architecture template and it is represented on Figure 2. It contains three programmable processors, a Viterbi accelerator, peripheral interfaces and a SDRAM memory controller.

DSP (Digital Signal Processor) is used for audio post processing and sample rate conversion. These tasks are small assembly coded programs that work on small data structures. DSP has small local memories to store the program and data structures.

TriMedia [17] is a VLIW (Very Long Instruction Word) media processor from NXP Semiconductors (formerly Philips Semiconductors). TriMedia is a Harvard architecture CPU that features many DSP and SIMD (Single Instruction Multiple Data) operations to efficiently process audio and video data streams. This processor is suitable for large programs because an optimizing compiler is available. On this processor we execute the DRM reception. The program is stored in the SDRAM because the code size of DRM is larger than 200 Kbytes.

Processing of control code is made by ARM9. This processor can execute an operating system and system configuration software. The system is configured before a job starts. The code for this processor is usually large, so it is stored in SDRAM.

Processors have uncached access to local memories in which the input and output buffers of the tasks are stored.

Memory controller has ports for low latency (LL) and latency tolerant (LT) streams. Each cache is connected to its LL port. A processor can use posted writes to write data directly in to the memory via a LT port.

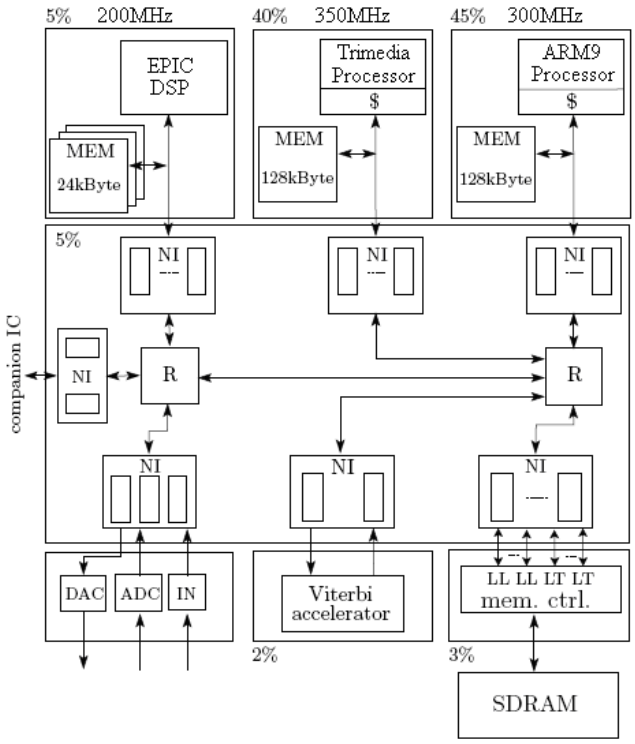


Figure 2. Car radio multiprocessor system

The Hijdra Architecture provides an important attribute. It can be designed to be composable. A highly composable system provides recombinant components that can be selected and assembled in various combinations to satisfy specific user requirements. One essential attribute that makes a system composable is that it is self contained. Other essential attribute is that it treats each request as an independent transaction, unrelated to any previous request, so, the system is stateless. This means that the temporal behaviour of one job is independent from another.

2.3 Khan Process Networks, Synchronous and Cyclo-Static Data Flow

Khan Process Networks [21] [22] are a model of communication where concurrent autonomous processes communicate through a FIFO queue and synchronization is made with a blocking/read primitive. Processes are connected by a connection channel, forming a network. These communication channels are the only way that processes can exchange information. To make an exchange of data, there has to be a process producing data elements called tokens, and a process that consumes those tokens. When a process attempts to get data from an input channel, the execution of this process needs to be suspended. A process cannot test the presence or absence of data in a channel. A process is either enabled or blocked waiting for data on only one of its input channels. A producer process can always write to a channel, and does not stall the process. A consumer process can be stalled reading from a channel. Reading from a channel is a blocking process. The process can only continue when there are enough tokens on the channel to be able to complete the process. Not every process needs to have a reading channel. They can be a pure data source. Khan Process Networks are deterministic. For a certain sequence of inputs, there is only one possible sequence of outputs. A simple example of a Khan Process Network is represented on Figure 3.

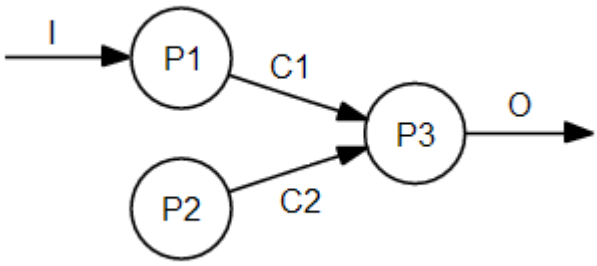


Figure 3. Khan Process Network

A pseudo-code with syntax similar to C can be used to define this Khan Process Network:

```
Channel int I, O, C1, C2;
P1(I, C1);
P2(C2);
P3(C1, C2, O);
Process P1 (in int x, out int y)
{
    int k;
    while(1)
    {
        x.read(k);
        if(k % 2 == 0)
            y.write(k)
    }
}
Process P2 (out int x)
{
    int k=20;

    while(1);
        x.write(k);
}

Process P3 (in int x, in int y, out int z)
{
    int k;
    int n;
    while(1)
    {
        x.read(k);
        y.read(n);
        n = n+k;
        z.write(n);
     }
}
```

In this case, if P1 reads as input an odd number, P3 will block. If the input is even, the process network will not block.

Khan Process Networks cannot be scheduled statically. In more complex networks there are situations were processes will block due to static schedule. It is not possible to derive, at compile time, a sequence of process activations such that the Network does not block under any circumstance. Instead of static schedule, Khan Process Networks have to be dynamically scheduled. The process to be activated at a certain time has to be decided during execution time, based on the current situation. This causes a huge overhead in implementing Khan Process Networks. They are too general, and cannot be implemented efficiently.

Data flow networks are a particular case of Khan Process Networks. A particular kind of Data Flow Networks that can be implemented efficiently, are Synchronous Data Flow [3] (SDF) Networks. These Data Flow models add some restrictions to Khan Process Networks. A process produces and consumes a fixed number of tokens on each of its outgoing and incoming channels. For a process to fire, it must have at least as many tokens on its input channels as it has to consume. For a correct SDF Network, a static schedule can be derived, and there is a partial order of events.

Applications can be divided in to independently operational parts called jobs (sometimes one application is one job). Each job consists of several tasks, which are the smallest functional units of an application. To ensure quality on multimedia applications, stream processing has real-time requirements, demanding that the time it takes to process a data packet is bounded. The number of output packets per second a job produces (throughput) is at least predefined. Tasks have worst execution times, and in this time, they consume a fixed number of tokens on every output. A token is defined as a container in which a fixed amount of data can be stored. During every execution of a task, the same amounts of tokens are produced. Another important constrain of real time systems is the time a job takes to start producing output packets (latency).

A job can be specified as a SDF and mapped in to a multiprocessor platform. Data flow is a natural paradigm for describing DSP (Digital Signal Processing) applications for current implementation on parallel hardware. Data flow programs for signal processing are directed graphs where each node represents a function and each arc represents a signal path. In SDF, the number of data packets consumed and produced by each node on each invocation is specified a priori (note that a node correspond to a task). An example of a simple SDF graph is shown on Figure 4

Figure 4. Simple SDF graph

In this example, actor (node) A produces 2 tokens and actor B consumes 3 tokens for each firing. In a valid SDF schedule, the first in/first out (FIFO) buffers on each arc return to their initial state after one schedule period.

Cyclo-Static Data Flow (CSDF) [15] is a generalization of SDF which each actor can have many phases. In CSDF, actors have cyclic changing firing rules. The number of tokens produced and consumed by an actor can vary from one firing to another one in a cyclic predictable pattern. It is then possible to construct periodic schedules using techniques based on those developed for SDF [23].

An example of a CSDF is represented on Figure 5

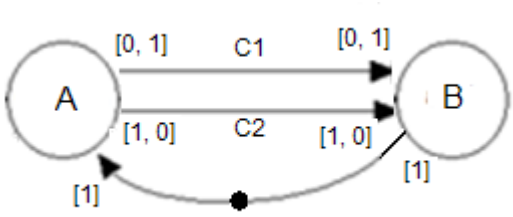


Figure 5. CSDF Graph

In this example, actors A and B have two distinct phases. On the first phase, channel C2 is used, and on the second phase channel C1.

With CSDF model it is possible to analytically derive the cycle that determines the throughput after conversion to a SDF. This model is well suited for multirate signal processing applications.

SDF model makes it possible to derive minimal throughput and maximum latency with analytical techniques [1]. HAPI [4] (Hijdra Application Programmer's Interface) developed at Philips Research simulates SDF graphs. HAPI is built on YAPI [5] (Y-chart Application Programmer's Interface), a SystemC library also developed at Philips Research. HAPI simulates communication between processes with worst-case timing. The worst-case time that data can arrive is made visible, making it possible to reason about the throughput and the latency. The main advantages between HAPI and the Car Radio Simulator are the visibility of worst case temporal behaviour, and simulation speed. Simulations with the Car Radio simulator based on Æthereal can take some hours while in HAPI it takes some seconds.

3 Digital Radio Mondiale

A DRM software receptor is to be implemented on the Car Radio simulator. Understanding DRM was essential in order to put it working on the Car Radio Simulator. An even more deep understanding was necessary to develop and attach functions to the software in order to have the application running on the simulator and producing an audible audio file. Data dependencies of the code were also studied. For these reasons it is important to include a Chapter about DRM.

3.1.1 General information

Since the first hour of radio broadcast, almost all Medium Wave (MW) and Long Wave (LW) transmitters use Amplitude Modulation (AM) for the transmission of audio signals. The modulation scheme as well as the small channel bandwidth of 10 KHz highly limits the audio quality which is the reason why commonly speech signals are transmitted in these bands. In the end of 2003 the European Telecommunications Standards Institute (ETSI) published the specification for digital radio broadcast below 30 MHz using a multi carrier technique called Orthogonal Frequency Division Multiplex (OFDM) [18]. The system was named Digital Radio Mondiale (DRM) and the audio quality of the streams reaches or exceeds the quality of FM mono transmissions. This results from advanced coding techniques. The advanced audio coding standard (AAC) combined with Spectral Band Replication (SBR) and parametric stereo provide high audio quality at very low bit rates. Beside AAC the DRM standard defines the HVXC and CELP codec to be used for transmitting speech signals. Bandwidth can be either 10 KHz or 20 KHz depending on the desired quality of transmission.

ETSI ES 201 980 [6] defines DRM Standards. It is the upcoming successor of AM radio and provides a flexible and efficient audio and data broadcasting standard. The intention of the DRM standard is to combine FM-like sound-quality on the AM frequency bands below 30 MHz with a large national or international coverage area by a small number of transmitting sites.

3.1.2 System overview

The DRM system is designed to be used at any frequency below 30 MHz, i.e. within the long, medium and short wave broadcasting bands, with variable channelization constraints and propagation conditions throughout these bands. In order to satisfy these operating constraints, different transmission modes are available. A transmission mode is defined by transmission parameters classified in two types:

- Signal bandwidth related parameters;
- Transmission efficiency related parameters.

The first type of parameter defines the total amount of frequency bandwidth and the structure used for one transmission. The current channel widths for radio broadcasting below 30 MHz are 9 kHz and 10 kHz. The DRM system is designed to be used within these nominal bandwidths in order to satisfy the current planning situation, and within channels with a bandwidth multiple of 4.5 kHz (half of 9 kHz) or 5 kHz (half of 10 kHz) to allow for simulcast with analogue AM signals or to provide for larger transmission capacity where and when the planning constraints allow for such facility.

It may additionally provide for channel bandwidths which are not strictly included within the ITU (International Telecommunications Union) channelization, but which would allow increased system capacity under relaxed interference condition.

For any value of the signal bandwidth parameter, transmission efficiency related parameters are defined to allow a trade off between capacity (useful bit rate) and ruggedness to noise or multipath and Doppler. These parameters are of two types:

- Coding rate and constellation parameters, defining the code rate and constellations which are used to convey data,

- OFDM symbol parameters, defining the structure of the OFDM symbols to be used as a function of the propagation conditions, called “Ground Wave mode”, “Sky Wave mode” and “Highly Robust modes”.

3.1.3 DRM Frame Transmission

In the current analogue broadcast systems, every radio channel contains one audio service and maybe some service data via Radio Data System (RDS). DRM transmission chain is characterized by three channels, the Main Service channel (MSC), the Fast Access Channel (FAC) and the Service Description Channel (SDC). A DRM super frame is represented on Figure 6.

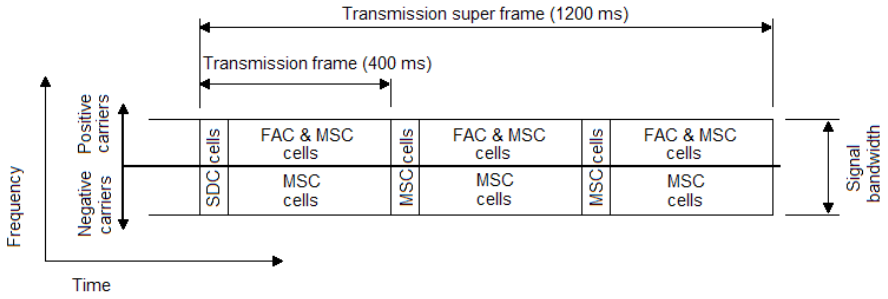


Figure 6. DRM super frame

The Main Service Channel (MSC) contains the data for all services contained in the DRM multiplex. The multiplex may contain between one and four services. Each service may carry audio or audio and data. The gross bit-rate of the MSC is dependent upon the DRM channel bandwidth and transmission mode. MSC consists of two parts which are each assigned a protection level. This way unequal protection error can be provided for one or more channels. Equal error protection is possible assigning the same level of protection for both parts. The MSC is divided into logical frames of 400ms.

The Fast Access Channel (FAC) is used to provide service selection information for fast scanning. It contains information about the channel parameters (for example the spectrum occupancy) such that the receiver is able to begin to decode the multiplex effectively. It also contains information about the services in the multiplex to allow the receiver to either decode this multiplex or change frequency and try again. The periodicity of the FAC frame is 400ms. The service parameters are carried in successive FAC frames, one service per frame. The service based information is carried in every FAC and each FAC contains the information for one service. The broadcaster may choose the way in which the FAC for each service is repeated to suit his requirements. For example, if he transmits an audio service and two data services, he may chose to send the FAC for the audio service on alternate frames to reduce the scan time for audio at the expense of increased time for data.

Service description channel (SDC) gives information on how to decode the MSC, how to find alternate sources of the same data, and gives attributes to the services within the multiplex. Alternative frequency checking may be achieved, without loss of service, by keeping the data carried in the SDC quasi-static. Therefore the data in the SDC frames has to be carefully managed. The SDC frame periodicity is 1200ms. The data capacity of the SDC frame varies with the spectrum occupancy of the multiplex and other parameters. The capacity can be further increased where necessary by varying the repetition pattern of the SDC frames.

3.1.4 General system architecture

Figure 7 describes the general encoder (transmitter) side of the DRM system. Different audio encoders can be used to encode the audio input stream, depending of the transmission capacity and content (audio/speech).

The source encoder and pre-coders ensure the adaptation of the input streams into an appropriate digital transmission format. For the case of audio source encoding, this functionality includes audio compression techniques. Different audio encoders can be used to encode the audio input stream, depending on the transmission capacity and content (audio/speech). DRM uses Advanced Audio Coding (AAC) extended by Spectral Band Replication (SBR) for audio and Code Excited Linear Prediction (CELP) for speech. The output of the source encoder(s) and the data stream pre-coder may comprise 2 parts requiring 2 different levels of protection within the subsequent channel encoder. All services have to make use of one or both of the same 2 levels of protection. The Multiplexer combines the protection levels of all data and audio services.

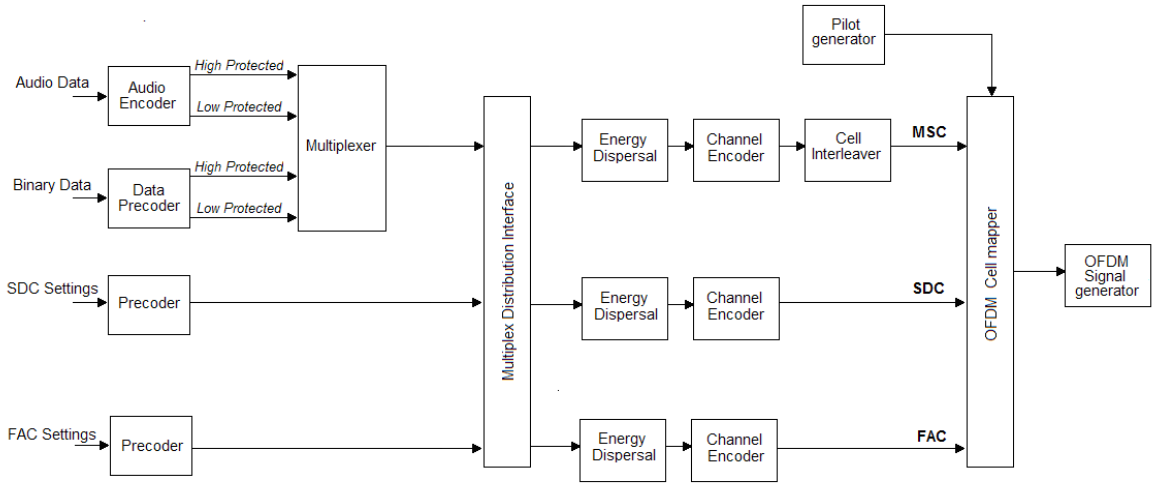


Figure 7. DRM transmitter scheme

After the pre-encoding the data stream and source encoding the audio stream, the bits are encoded by the Multilevel Coding (MLC) scheme using energy dispersal, convolutional encoding and bit interleaving. The energy dispersal scrambler provides a deterministic selective complementing of bits in order to reduce the possibility that systematic patterns result in unwanted regularity in the transmitted signal. At the receiver bits are decoded with a multistage decoder using Viterbi Decoder [8].

The channel encoder adds redundant information for error correction, and defines the mapping of the digitally encoded information onto QAM (Quadrature Amplitude Modulation) cells. The parameters for the MLC encoder depend on the desired error protection levels of the information. Encoded bits are mapped with a 4-QAM, 16-QAM or 64-QAM modulation scheme. MSC cells are cell interleaved to prevent burst errors at the receiver side. Cell interleaving can be short or long. Short interleaving only uses cells from one frame while long interleaving uses cells from five frames. The receiver will have to wait for at least five frames to start decoding.

The pilot generator provides a means to derive channel state information in the receiver, allowing for a coherent demodulation of the signal. The OFDM cell mapper collects the different classes of cells and places them on the time frequency grid. The inverse Fast Fourier Transform (IFFT) maps the signal back from the frequency domain into the time domain. After the IFFT a guard time is added to prevent inter symbol interference and perform time synchronization at the receiver. After the modulator, the OFDM symbol is ready to be transmitted on the desired carrier frequency.

Resuming all above, digital sound broadcasting system comprises conceptually distinct transmission stages:

- The audio signal must first be converted to digital form. Since the raw bit-rate that results is impracticably high, a form of bit-rate reduction tailored to the signal properties is then applied. This is referred to as source coding.

- The source-coded data is then multiplexed together with any other data that forms part of the pay-load.

- The multiplexed data of the payload is subjected to channel coding to increase its ruggedness.

- The channel-coded data is modulated onto the RF signal for transmission.

Source coding reduces the data rate while channel coding increases it.

At the receiving end, the receiver first acquires synchronization with the signal, and then reverses the transmission stages by means of the following processes:

- Demodulation;

- Channel decoding (*correcting the transmission errors*);

- Demultiplexing the transmitted data into component streams;

- Source decoding (*to obtain an audio signal from the audio stream*).

More information on DRM encoding/decoding can be found on Appendix A.

3.2 Advanced Audio Coder

ISO/IEC 14496-3 [12] defines the MPEG-4 Audio standard. The audio coding standard MPEG-4 AAC is part of the MPEG-4 Audio standard. DRM uses the Advanced Audio Coding (AAC), supplemented by Spectral Band Replication (SBR).

3.2.1 General overview

Waveform coders like AAC, work by analyzing the content of each part of the audio spectrum and describing each one no more accurately than is needed in order to satisfy the ear of the listener. Sounds that are masked by nearby louder sounds are discarded altogether. AAC follows in the tradition of MPEG-1 Layer 2 and MP3 in this regard, and forms part of the MPEG-4 standard. However, even with the advances made, it is difficult to deliver an "FM-like" 15 kHz bandwidth using AAC alone at the very low bit-rates envisaged, without introducing audible artefacts. The answer lies in the combination of AAC with the SBR technique. The SBR technique synthesizes the sounds which fall within the highest frequency octave-and-a-bit. Sounds in this range are usually either:

- Noise-like (sibilance, percussion instruments such as shakers, brushed cymbals etc.), or

- Periodic and related to what appears lower in the spectrum (overtones of instruments or voiced sounds).

At the sender, the highest-frequency band of the audio signal is examined to determine the spectral distribution and whether it falls into one of the categories above. A small amount of side information is then prepared for transmission to help the decoder. The highest-frequency band is then removed before the remaining main band of the audio signal is passed to the AAC coder, which codes it in the conventional way. At the receiver, the AAC decoder first decodes the main band of the audio signal. The SBR decoder then adds the synthetic upper band, helped by the instructions sent in the side information. Overtones are derived from the output of the AAC decoder, while noise-like sounds are synthesized using a noise generator with suitable spectral shaping.

The possibility of stereo operation is foreseen, although this would only be sensible if it was possible to use a double-width channel of 18 or 20 kHz RF bandwidth.

3.2.2 AAC Super Audio Frame

An AAC super audio frame (Figure 8) is 400ms long (equal to one DRM frame). One MSC frame carries one AAC super audio frame. An AAC super audio frame has two parts. One higher protected part and one lower protected part.

3.2.2.1 Higher protected part

The higher protected part contains one header followed by higher protected blocks. The number of higher protected blocks equals the number of AAC frames in the audio super frame.

The header contains the absolute position of the frame borders that are used to recover the length of each AAC frame. The frame borders are stored consecutively in the header occupying 12 bits each (*unsigned integer, most significant bit first*). Frame borders are measured in bytes from the start of the AAC bit stream sequence. Header does not contain information about the last frame border since it is possible to calculate it by subtracting the previous frame border to the audio payload length. In the case of 10 audio frames header size will be 12 bits x 9 Frame borders = 108 bits plus 4 padding bits that are added to make the value divisible by 8. Header size in this case is 112 bits.

One higher protected block contains a certain amount of bytes from the start of each AAC frame, dependent upon the UEP (*Unequal Error Protection)* profile. One 8-bit CRC check derived from the CRC-bits of the corresponding AAC frame follows. For a mono signal, the CRC-bits cover (*mono1, mono2*). For a stereo signal, the CRC-bits cover (*stereo1, stereo2, stereo3, stereo4, stereo5*).

3.2.2.2 Lower protected part

The lower protected bytes (the remaining bytes not stored in the higher protected part) of the AAC frames are stored consecutively in the lower protected part.

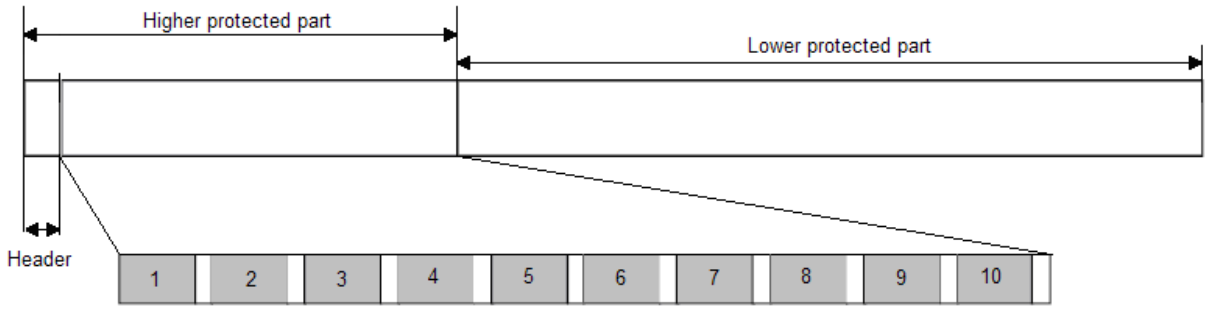


Figure 8. AAC Super Audio Frame

4 DRM on Hijdra

DRM reception software runs on TriMedia. The aim is to divide the code and “pull out” Viterbi decoding so it can run on the Viterbi accelerator at the same time that other tasks run on the TriMedia. Two different approaches can be considered. Construct the task graph and fill the tasks with different pieces of code, or divide the code and then construct the task graph with it. The first approach seemed to be reasonable, but several complications arisen due to problems sending data over the FIFOS. The second approach showed to be easier to implement. This chapter describes how the code was divided and implemented.

4.1 Implementation of the DRM receiver

The first thing was to add an audio decoder to DRM receiver because it was only producing non audible AAC frames. In order to decode them, functions were developed and were put together along with Dream (Free DRM software receptor) and FAAD (Free Audio Advanced Decoder) functions. As soon as the simulator was producing an audible file (.wav) dividing the code was the next step.

DRM receptor is a complex code, and the function performing Viterbi decoding was deep inside many “if” and “for” statements. At this stage the code can be represented by the task graph on figure 9. One can observe that it is necessary to make 25 reads in order to produce one AAC frame. Reads are made from a text file containing OFDM symbols. Each read corresponds to the amount of 4*768 floats. Drm_task decodes the symbols and produces AAC frames. Viterbi function is inside Drm_task and needs to be “pulled out”. The problem here is that Viterbi decoder is deep inside many functions and cyclic functions performed in Drm_task. To be able to “pull out” Viterbi, it is necessary to know the behaviour in each conditional statement. Depending on the input data, a determined conditional statement could or could not take part, or the number of iterations on a “for” statement for example, could vary. The behaviour of the code is different for MSC and for SDC data, and it is even different between MSC data of DRM frame 1, and MSC data from DRM frame 2. Knowing every detail of code behaviour allowed writing the code in a straight forward way, which in turn made it easier to study data dependencies.

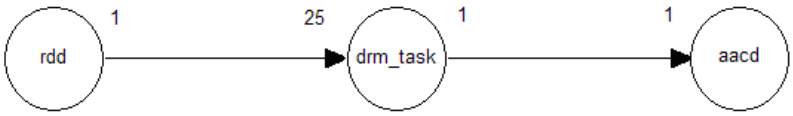


Figure 9. DRM receiver task graph

4.2 The Viterbi decoder

DRM transmitter uses a multilevel encoder. Multilevel encoders join channel encoding with modulation producing powerful transmission schemes. DRM uses an iterative multistage decoder (for OFDM symbols) performing Viterbi decoding at the receiver. As the number of iteration gets higher, a higher reliability on the decoded symbol can be achieved. A 64-QAM Multilevel encoder and multistage decoder are represented on Figure 10. Multilevel encoder would have only 2 levels for 16-QAM and 1 for 4-QAM. More information on Multilevel encoding/decoding can be found in Appendix 1.

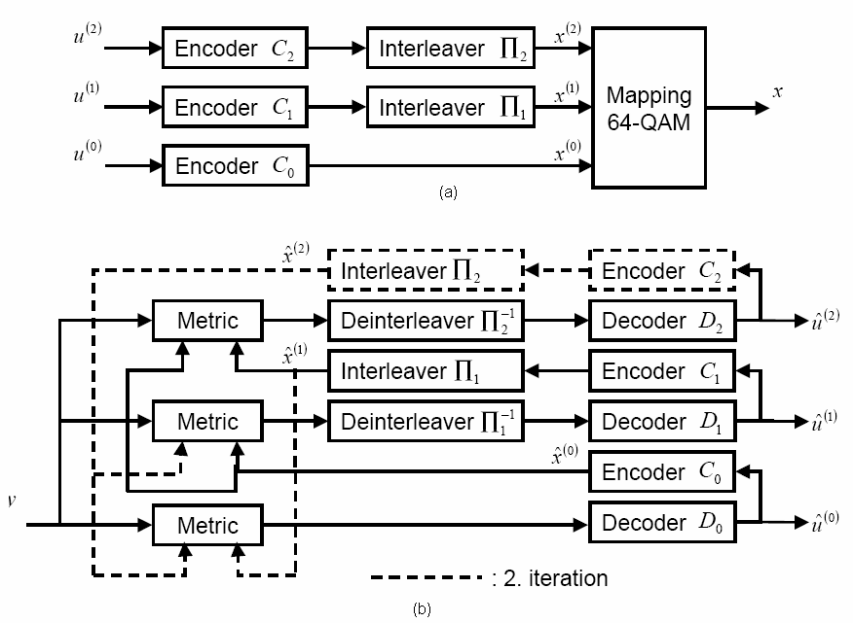


Figure 10. (a) - Multilevel encoder 64-QAM; (b) – Multistage decoder 64-QAM

In a DRM super frame, as said on clause 3.1.3, there are three channels. Each channel is transmitted with a different modulation scheme according to their purposes. For example, data needed to synchronize frames need to be reliable, and so, a more powerful scheme is needed to assure that.

Depending on the transmission modes SDC and MSC can be either modulated with 16-QAM or 64-QAM. We used mode A, QPSK for FAC, 16-QAM for SDC and 64-QAM for MSC. Multistage decoders for each one of these modulation schemes (explained on [11]) are represented on Figures 11, 12 and 13. These figures are represented with the names of variables used in the code. For simplicity, feedback of multistage decoder was removed.

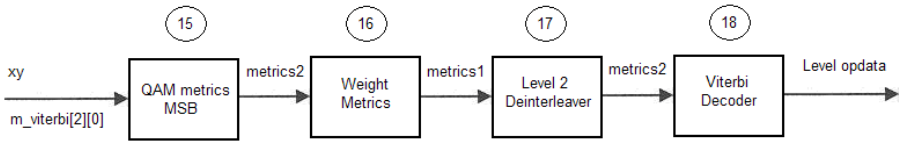


Figure 11. Multistage decoder for 4-QAM

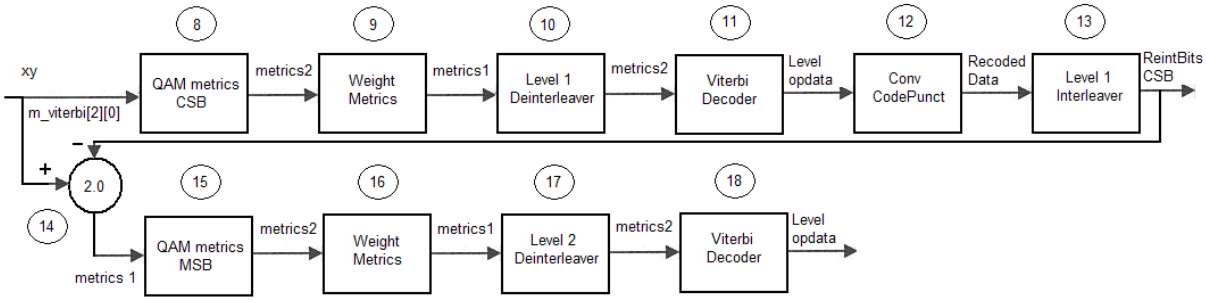


Figure 12. Multistage decoder for 16-QAM

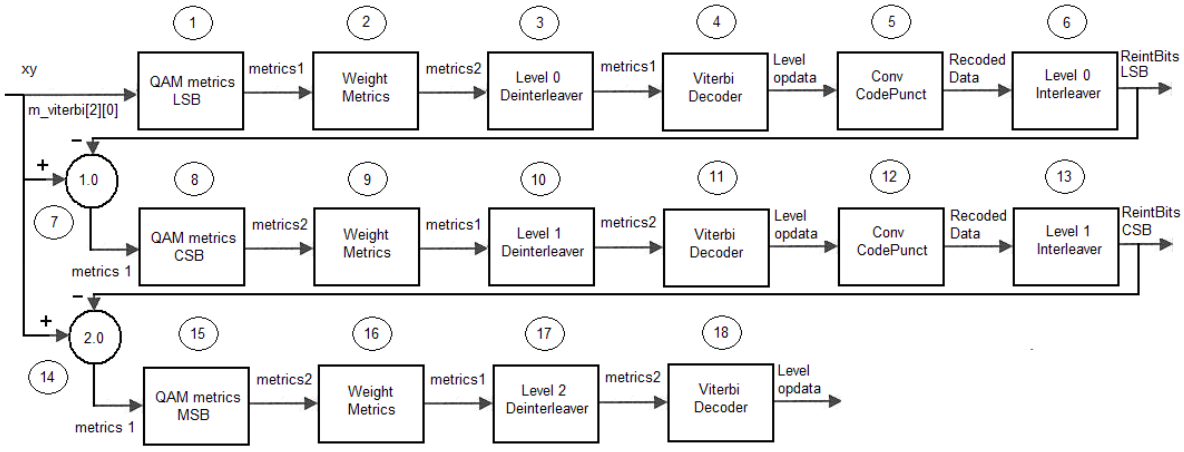


Figure 13. Multistage decoder for 64-QAM

As said before, the code repeats itself on each DRM Super Frame. Each DRM Super frame has 3 DRM frames. Although the number of OFDM symbols is equal in every FAC channel, the number of symbols read before FAC of DRM frame one is different from number of symbols read before decoding FAC channel from DRM frame 2. This means that the code used to decode channel SDC for DRM frame 1 is different from the one used to decode DRM frame 2. This is possible because the data is stored in buffers before being decoded allowing some flexibility between number of reads and the production of one AAC frame. It was observed that the code makes 28 reads to produce the AAC frame corresponding to the first DRM frame in a DRM Super Frame, 23 for the second and 24 for the third.

The first channel to be decoded on a DRM Super Frame is SDC which is only present on the first DRM frame. The code first reads five tokens, stores that information on buffers and then starts the decoding it with the multistage decoder previously showed on figure 12. One can see on the figure that Viterbi is used two times. FAC channel is to be decoded next. 22 reads are made, and multistage decoder on figure 11 is used. Viterbi decoder is used 1 time. For MSC multistage decoder on figure 13 is used after 3 reads. Viterbi is performed three times. “Pulling out” Viterbi decoding from multistage decoder will lead to the task graph represented on figure 14.

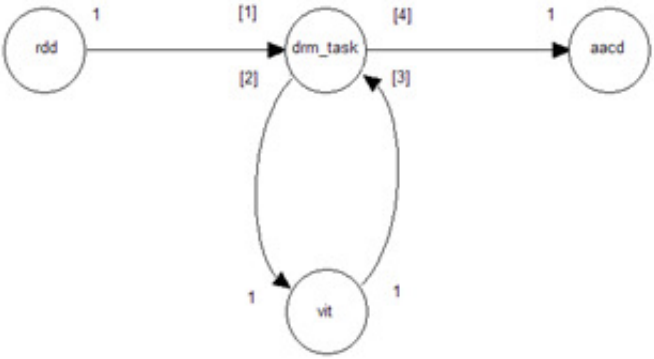


Figure 14. Task graph with Viterbi task

Drm_task phases are represented on the table from figure 15. For example in SDC in phase 1, drm_task needs 5 tokens. After getting these tokens, one token is sent to Viterbi task. When Viterbi task has data available, drm_task reads one token from it, corresponding to phase 3. Phases 4 and 5 correspond to sending/receiving another token to Viterbi decoder. For the other phases and channels the logic is made in an analogue way.

Frame 0																
Phase	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
	SDC					FAC			MSC							
[1]	5	0	0	0	0	20	0	0	3	0	0	0	0	0	0	0
[2]	0	1	0	1	0	0	1	0	0	1	0	1	0	1	0	0
[3]	0	0	1	0	1	0	0	1	0	0	1	0	1	0	1	0
[4]	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1

Frame 1											
Phase	17	18	19	20	21	22	23	24	25	26	27
	FAC			MSC							
[1]	22	0	0	1	0	0	0	0	0	0	0
[2]	0	1	0	0	1	0	1	0	1	0	0
[3]	0	0	1	0	0	1	0	1	0	1	0
[4]	0	0	0	0	0	0	0	0	0	0	1

Frame 2											
Phase	28	29	30	31	32	33	34	35	36	37	38
	FAC			MSC							
[1]	24	0	0	0	0	0	0	0	0	0	0
[2]	0	1	0	0	1	0	1	0	1	0	0
[3]	0	0	1	0	0	1	0	1	0	1	0
[4]	0	0	0	0	0	0	0	0	0	0	1

Figure 15. Drm_task phases

Establishing a connection between these phases and figures 11, 12 and 13 we get to table on figure 14.

FAC 1	Necessary reads, demodulation and data storing till begginig of multistage decoder
FAC2	15; 16; 17;
FAC3	From the end of multistage decoder to the end of Drm Receiving Process
SDC1	5 reads, demodulation and data storing
SDC2	8, 9, 10;
SDC3	12, 13
SDC4	15, 16, 17;
SDC5	19, 20, to the end of Drm Receiving Process
MSC1	Necessary reads, sync ok?, cells ready?
MSC2	1, 2, 3
MSC3	5, 6
MSC4	8, 9, 10
MSC5	12, 13,
MSC6	15, 16, 17
MSC7	19, 20
MSC8	Prepare and send data to AAC decoder

Figure 16. Drm_task phases

DRM behaviour is now known, and code can be written in a straightforward way. Task graph previously showed on figure 14 can represent the code. This task graph represents DRM code after synchronization. Before synchronization, DRM reception behaviour is explained on [14]. The behaviour is only predictable assuming no failure in DRM receiver, otherwise it will have to synchronize and behaviour will not be the same. When DRM receiver starts receiving OFDM symbols, the first channel that is need to be decoded is FAC. FAC has information needed to decode SDC and MSC. That is why FAC is encoded with 4-QAM. Decoding needs to be reliable and fast. If SDC or MSC parameters change in time, this change is alerted in FAC, so, although SDC is the first channel to be decoded in a DRM Super Frame, DRM receiver needs FAC information from a previous DRM Super Frame in order to decode the first SDC channel.

Now that Viterbi was “pulled out”, a new tile Viterbi Accelerator was created. Connections through the network were established and many simulations with different execution times were made. Results are in Chapter 4.

4.3 DRM data dependencies between phases

After sending data to Viterbi accelerator, TriMedia (where DRM reception application is running) is stopped wasting processing time. DRM data dependencies were studied in order to plan a solution to load some work in TriMedia while Viterbi accelerator is processing information.

Figure 17 represents the different phases grouped by channels (FAC, SDC and MSC) on a DRM Super frame with a distinct separation between drm_task and Viterbi task. There is always a data dependency on the code in each channel, and for that reason, the figure shows not all the phases, but group of phases for each channel.

Data dependencies between two consecutive DRM Super Frames are showed in the block diagram on figure 18. Phases in the same channel have always dependencies as shown in figure. After synchronization, the first channel that needs to be decoded is FAC (it does not depend on any other channel). SDC and MSC depend on FAC parameters. The receiver starts storing symbol data after acquiring synchronization and after CRC check for FAC of frame two passes. Synchronization is done and the receiver has necessary information to decode SDC. As phase SDC 1 is only for reading and storing information it does not need FAC information. It will only need FAC information on SDC2. The same happens with MSC with the difference that MSC needs FAC and SDC to be decoded.

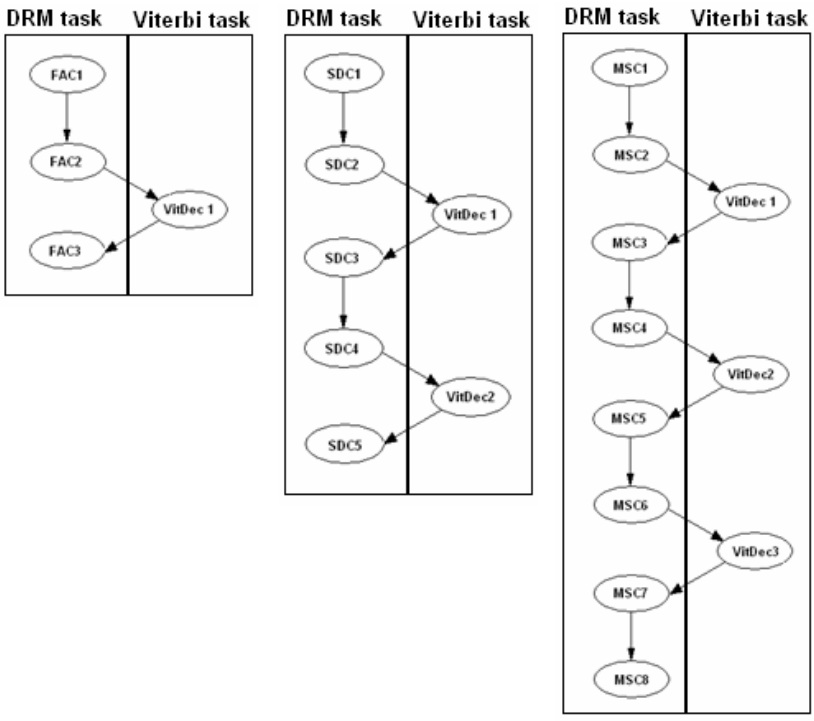


Figure 17. Drm_task phases and Viterbi task

Note that FAC parameters may change in time. When that happens, there is a FAC parameter (clause 6.3.3 on ETSI TS 101 980 v1.1.1), Reconfiguration Index, that is set to a number different from 0 (when 0 means FAC is not going to change). If the number is for instance 5, this number will be decreased by one in each Super Frame. When that parameter is 1, FAC will change on the next Super Frame. Changed Parameters are signalled on advance in SDC data entity type 10 (clause 6.4.3.11 on ETSI TS 101 980 v1.1.1). This way the receiver has more time to make necessary adjustments in order to cope with the change of parameters.

One possible solution on running the code in parallel could be for example, running SDC1 from Super Frame 1 between FAC2 and FAC3 while Viterbi is being performed. Many other approaches like this can be made. Execution times of these tasks are important to be able to implement the best solution. Values are presented on chapter 5.

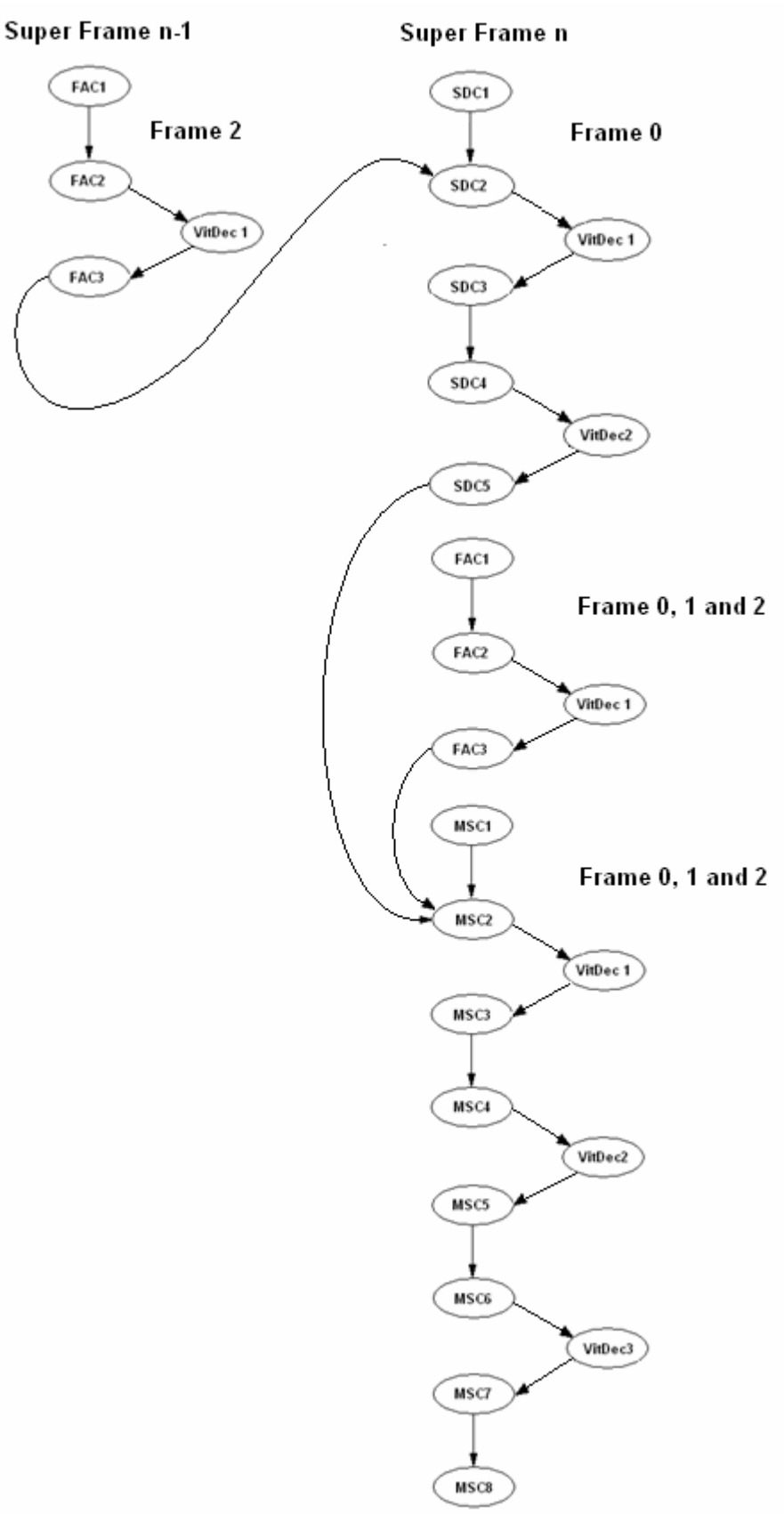


Figure 18. DRM data dependencies between phases

5 Results

5.1 The Viterbi accelerator

Viterbi accelerator was implemented on the Car Radio simulator. To obtain results, several simulations were realized with different execution times for the Viterbi accelerator. The simulator was reading input data from a .txt file with OFDM symbols in the form of floats. These symbols were obtained with a Spark DRM transmitter. This is a freeware software DRM transmitter that has the option to store the OFDM symbols in a text file. This tool accepted as input .wav files. Several different configurations were possible. Configurations were made in accordance with the Car Radio configuration, already mentioned on Chapter 4.

Figure 19 shows the number of cycles used to produce AAC Super frames. An AAC super frame is produced per DRM frame. On the first DRM frame of one DRM super frame, SDC has to be decoded besides MSC and FAC. That is the reason of the peaks of cycles observed on the figure for AAC super frame 1, 4 and 7.

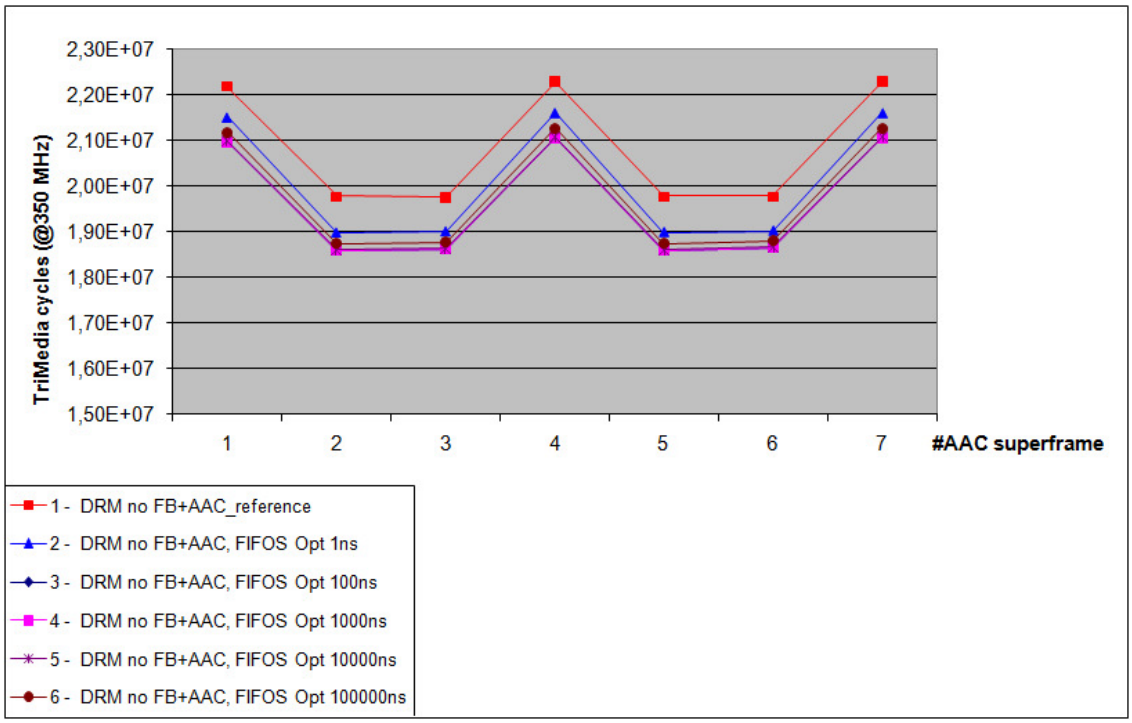


Figure 19. TriMedia cycles at 350 MHz

Simulation number 1 (DRM no FB + AAC_reference) is a simulation with DRM receiver running on TriMedia. FB stands for Feedback. In simulation 2, Viterbi decoding was already being decoded on Viterbi accelerator. On this first stage of the Viterbi accelerator testing data was being sent in more amounts than actually needed. This is because, at this time, the amount of data needed for each stage decoding (regarding to FAC, SDC or MSC), was not precisely known. Viterbi decoding on FAC needs much less data than Viterbi decoding on MSC, but exact values were not known. On this simulation, enough data was sent for each channel decoding, to guarantee the correct decoding. This is the reason why cycle consuming in this simulation with only 1ns of execution time, is higher than number of cycles used on simulations 3, 4, 5 and 6 which have execution times higher. From simulation 3 to simulation 6, the amount of data needed for each stage of decoding was already known, thus, data sent over the NoC was substantially reduced when compared to data sent over NoC in simulation 2. The only parameter that varies in simulations 3, 4, 5 and 6 is the execution time of Viterbi accelerator. One can observe that simulation 6 has the higher results.

This graphic relates the number of cycles with the number of the frame produced. Frame 0 is discarded due to the fact that the first AAC super frame produced contains the synchronization and so the value is very high.

Figure 20 shows a detailed table of values obtained. Last column shows the percentage of gain in number of cycles in comparison with the number of cycles used with Viterbi decoding being performed on TriMedia. Viterbi decoding number of cycles was estimated to be 10 % of DRM Receiver. From the table one can see that gains of 5% were achieved. Even with an execution time of 100000ns gains achieved were 4%. Gains could be much higher parallelising the code and the Viterbi Accelerator could be also useful and maximize efficiency if another application besides DRM Receiver uses Viterbi decoding. One can say that Viterbi Accelerator can be useful in achieving a better efficiency in a multiprocessor system. Execution times of a hardware Viterbi accelerator are unknown.

1 - DRM no FB + AAC	DRM	AAC	Total	Gain		
	116144661	7082753	123227414			
	15539913	6642900	22182813			
	13033102	6743266	19776368			
	13004947	6748678	19753625			
	15549078	6739883	22288961			
	13029511	6746258	19775769			
	13006749	6768743	19775492			
	15530173	6760553	22290726			
2 - DRM no FB+AAC, FIFOS					Gain related to 1	% Compared to 1
	117529064	7149910	124678974	-1451560	-1451560	-1,18
	15312769	6697165	22009934	172879	172879	0,78
	12489404	6777948	19267352	509016	509016	2,57
	12507454	6783253	19290707	462918	462918	2,34
	15330241	6772637	22102878	186083	186083	0,83
	12499355	6780106	19279461	496308	496308	2,51
	12516208	6801099	19317307	458185	458185	2,32
	15308055	6794375	22102430	188296	188296	0,84
3 - DRM no FB+AAC, FIFOS QPSK opt					Gain related to 1	% Compared to 1
	116186610	7153228	123339838	1339136	-112424	-0,09
	15184156	6690492	21874648	135286	308165	1,39
	12365054	6765002	19130056	137296	646312	3,27
	12371738	6768051	19139789	150918	613836	3,11
	15204675	6765657	21970332	132546	318629	1,43
	12370611	6771152	19141763	137698	634006	3,21
	12377017	6796617	19173634	143673	601858	3,04
	15181660	6790020	21971680	130750	319046	1,43
4 - DRM no FB+AAC, FIFOS opt, 1ns viterbi					Gain related to 1	% Compared to 1
	115497798	7152586	122650384	689454	577030	0,47
	14812863	6700026	21512889	361759	669924	3,02
	12193159	6780733	18973892	156164	802476	4,06
	12214119	6786113	19000232	139557	753393	3,81
	14838377	6772067	21610444	359888	678517	3,04
	12204182	6782373	18986555	155208	789214	3,99
	12209493	6806496	19015989	157645	759503	3,84
	14811194	6796737	21607931	363749	682795	3,06
5 - DRM no FB+AAC, FIFOS opt + opadata opt, 1ns viterbi, big bw					Gain related to 1	% Compared to 1
	112334250	7155221	119489471	3160913	3737943	3,03
	14261000	6702622	20963622	549267	1219191	5,50
	11810521	6782920	18593441	380451	1182927	5,98
	11844435	6788595	18633030	367202	1120595	5,67
	14269306	6778095	21047401	563043	1241560	5,57
	11819074	6785567	18604641	381914	1171128	5,92
	11849706	6807283	18656989	359000	1118503	5,66
	14256502	6799844	21056346	551585	1234380	5,54
7 - DRM no FB+AAC, FIFOS ipdata and opdata opt, Vit 100 ns					Gain related to 1	% Compared to 1
	112335332	7155222	119490554	-1083	3736860	3,03
	14261239	6702622	20963861	-239	1218952	5,50
	11810621	6782921	18593542	-101	1182826	5,98
	11844586	6788595	18633181	-151	1120444	5,67
	14269488	6778095	21047583	-182	1241378	5,57
	11819269	6785568	18604837	-196	1170932	5,92
	11849894	6807284	18657178	-189	1118314	5,66
	14256970	6799843	21056813	-467	1233913	5,54
6 - DRM no FB+AAC, FIFOS opdata ipdata opt, Vit 1000 ns					Gain related to 1	% Compared to 1
	112343895	7155220	119499115	-8561	3728299	3,03
	14263212	6702622	20965834	-1973	1216979	5,49
	11811881	6782921	18594802	-1260	1181566	5,97
	11845814	6788595	18634409	-1228	1119216	5,67
	14271327	6778095	21049422	-1839	1239539	5,56
	11820530	6785567	18606097	-1260	1169672	5,91
	11851136	6807283	18658419	-1241	1117073	5,65
	14258658	6799843	21058501	-1688	1232225	5,53
7 - DRM no FB+AAC, FIFOS opdata ipdata opt, Vit 10000 ns					Gain related to 1	% Compared to 1
	112428800	7155220	119584020	-84905	3643394	2,96
	14282100	6702621	20984721	-18887	1198092	5,40
	11824501	6782919	18607420	-12618	1168948	5,91
	11858389	6788595	18646984	-12575	1106641	5,60
	14290259	6778095	21068354	-18932	1220607	5,48
	11833124	6785567	18618691	-12594	1157078	5,85
	11863704	6807283	18670987	-12568	1104505	5,59
	14277697	6799843	21077540	-19039	1213186	5,44
8 - DRM no FB+AAC, FIFOS opdata ipdata opt, Vit 10000 ns					Gain related to 1	% Compared to 1
	113279314	7155221	120434535	-850515	2792879	2,27
	14471010	6702622	21173632	-188911	1009181	4,55
	11950456	6782920	18733376	-125956	1042992	5,27
	11984396	6788595	18772991	-126007	980634	4,96
	14479310	6778095	21257405	-189051	1031556	4,63
	11959155	6785567	18744722	-126031	1031047	5,21
	11989730	6807283	18797013	-126026	978479	4,95
	14466519	6799844	21266363	-188823	1024363	4,60

Figure 20. TriMedia cycles table

5.2 DRM data dependencies and phase cycles

As seen before, DRM data dependencies allow the load of some work in TriMedia while Viterbi accelerator is processing data. The measurements presented on this chapter are to see if the amount of data or possibly an entire phase can be processed in the TriMedia while Viterbi accelerator is working. For this, we will need the number of cycles of each phase.

First, the number of cycles was computed for two entire super frames. Table on Figure 21 shows the number of instructions and cycles used. Measurements were made with debugging messages at the beginning and end of the reception code and AAC decoder.

Number of inst. of entire super frames			
Super Frame 0	Frame 0	Frame 1	Frame 2
Total inst.	15843697	14102999	14078935
AAC inst.	5642494	5482497	5408735
Inst. No AAC	10201203	8620502	8670200
Super Frame 1	Frame 0	Frame 1	Frame 2
Total inst.	15614088	14031882	14085779
AAC inst.	5415275	5411844	5416115
Inst. No AAC	10198813	8620038	8669664
Number of cycles of entire super frames			
Super Frame 0	Frame 0	Frame 1	Frame 2
Total cycles	29456073	26178247	26292073
AAC cycles	9257628	9373420	9379511
Cycles No AAC	20198445	16804827	16912562
Super Frame 1	Frame 0	Frame 1	Frame 2
Total cycles	29562262	26193287	26302305
AAC cycles	9369799	9381967	9393863
Cycles No AAC	20192463	16811320	16908442

Figure 21. Instructions and cycles used in two DRM super frames

Next step was to measure cycles/instructions of each phase. Measurements were made with debugging messages at beginning and end of each phase (see Figure 16). Figure 20 shows the results. These values represent the number of cycles/instructions discarding cycles used in reading, storing and in the code after channel decoding to the end of DRM receiving process function. Cycles used on these parts of the code are in Figure 22 and 23.

Frame 0			
	Phase	Cycles	Instructions
SDC	1	22629	11237
	2	26832	18887
	Viterbi	238005	63403
	3	19735	14636
	4	23326	16906
	Viterbi	276845	101223
	5	41500	31272
FAC	6	7457	2106
	7	5555	3079
	Viterbi	206761	35795
	8	6487	2520
MSC	9	614594	309946
	10	206555	112744
	Viterbi	445078	236979
	11	63113	46001
	12	298117	181336
	Viterbi	684232	445624
	13	167588	150258
	14	258912	144361
	Viterbi	923861	655425
	15	143061	133431
	16	11229	298
SUM		4691472	2717467

Frame 1			
	Phase	Cycles	Instructions
FAC	17	7587	2106
	18	5559	3079
	Viterbi	206421	35795
	19	6487	2520
MSC	20	616593	309946
	21	204939	111411
	Viterbi	445065	46001
	22	63122	236979
	23	298092	181336
	Viterbi	684258	445624
	24	167593	150258
	25	258894	144361
	Viterbi	923831	655425
	26	143062	133431
	27	11354	3013
SUM		4042857	2461285

Frame 2			
	Phase	Cycles	Instructions
FAC	28	7323	2106
	29	5680	3079
	Viterbi	207446	35795
	30	6479	2520
MSC	31	619462	309946
	32	204918	111411
	Viterbi	442498	236979
	33	63118	46001
	34	297893	181336
	Viterbi	684249	445624
	35	167559	150258
	36	258811	144361
	Viterbi	923915	655425
	37	143062	133431
	38	11141	2981
SUM		4043554	2461253

Figure 22. DRM super frame cycles/instructions without readings and storing

	Reads	Cycles	Instructions
SDC	5	2673941	1304115
FAC	20	10898622	5277297
MSC	3	1726667	849189
FAC	22	11848697	5733973
MSC	1	770633	391499
FAC	24	12800398	6191707
MSC	0	0	0

Figure 23. Number of cycles/instructions used on reading and storing

After	Cycles	Instructions
SDC	70566	16708
FAC	77107	17232
MSC	71325	19519
FAC	77099	77099
MSC	71669	71669
FAC	0	0
MSC	71247	19519

Figure 24. Cycles/instructions used in the code after the channel decoding

Adding these values to the ones on Figure 22, we get the final values. These are represented on Figure 23. Summing the number of cycles/instructions of each phase and comparing these values to measurements obtained on Figure 21 (precise values of cycles/instructions in each super frame), we can see that the error is minor corresponding to a precision on the values of 99.4%, and so, measurements are valid. All these values can be consulted more in detail in the CD provided with the dissertation.

Frame 0					Frame 1			
	Phase	Cycles	Instructions			Phase	Cycles	Instructions
SDC	1	2696570	1315352		FAC	17	11856284	5736079
	2	26832	18887			18	5559	3079
	Viterbi	238005	63403			*Viterbi*	206421	35795
	3	19735	14636			19	83586	79619
	4	23326	16906		MSC	20	1387226	701445
	Viterbi	276845	101223			21	204939	111411
	5	112066	47980			*Viterbi*	445065	46001
Sum			**1578387**			22	63122	236979
FAC	6	10906079	5279403			23	298092	181336
	7	5555	3079			*Viterbi*	684258	445624
	Viterbi	206761	35795			24	167593	150258
	8	83594	19752			25	258894	144361
Sum			**5338029**			*Viterbi*	923831	655425
MSC	9	2341261	1159135			26	143062	133431
	10	206555	112744			27	83023	74682
	Viterbi	445078	236979		SUM		16810955	**8735525**
	11	63113	46001		Frame 2			
	12	298117	181336			Phase	Cycles	Instructions
	Viterbi	684232	445624		FAC	28	12807721	6193813
	13	167588	150258			29	5680	3079
	14	258912	144361			*Viterbi*	207446	35795
	Viterbi	923861	655425			30	6479	2520
	15	143061	133431		MSC	31	619462	309946
	16	82554	19817			32	204918	111411
Sum		20209700	**3285111**			*Viterbi*	442498	236979
						33	63118	46001
TOTAL			**10201527**			34	297893	181336
						Viterbi	684249	445624
						35	167559	150258
						36	258811	144361
These tables include the reads and the code						*Viterbi*	923915	655425
after mlcpDecode for each channel						37	143062	133431
						38	82388	22500
					SUM		16915199	**8672479**

Figure 25. Cycles used in DRM phases

This tables show that it is possible to put some load in to the TriMedia while Viterbi accelerator is performing. Time used in Viterbi decoding is comparable to some phases. With this and the information on the chapter of DRM data dependencies it is possible to map tasks in to the TriMedia while Viterbi decoding is being performed.

6 Conclusion

6.1 Summary

The most important issue on this work was to know the advantages and disadvantages of having a specific processor performing Viterbi decoding. Although this processor is faster than TriMedia performing this task, because it is designed specifically to perform Viterbi decoding, time wasted transferring data through the silicon network between processors has to be taken in account. On this context, the initial scope of this work was to answer the following questions:

What is the application performance when communication between processor caches and system memories are made through a silicon network? Does the extra latency introduced by the silicon network affect too much the performance?

Will it be worth, from the performance of the system point of view, to use a hardware accelerator for Viterbi decoder, taking in account the communication time between the TriMedia and the accelerator?

As the simulator has real-time constrains, the worst case performance is as much or even more relevant than the medium case observed in the simulator. What is the performance worst case and what is the difference between the worst case and medium case?

6.2 Evaluation

DRM receiver application was the main subject around this dissertation. Most of the time spent on studying DRM and Catena's software. Working around DRM receiver code was a big time consuming job. The code was deeply rewritten, altered and new functionalities were added to the application. In the early stage of the dissertation, DRM receiver did not produce any audible file, which made the task of checking if a right demodulation had taken a part more difficult. Functions from FAAD were used to be able to produce audible outputs. As these functions accepted as input AAC frames, and DRM receiver was producing AAC super frames, new functions to transform AAC super frames in frames were created. Some functions from Spark DRM Receiver (developed by Volker Fischer) were also used. In the end, DRM Receiver was producing an audible file. This would make the DRM application easier to test and experiencing changes in the code. The production of an audible audio file equal to the inputted file would prove that everything was gone well.

“Pulling out” the Viterbi task was revelled to be much more difficult than predicted. Same behaviour was expected when decoding FAC or MSC channel for every DRM frame because one AAC Super Frame is produced per DRM frame, and that was not at all the real behaviour of the Receiver. Different behaviour in each channel and in each frame was not easy to observe, but after knowing the complete behaviour, “pulling out” Viterbi decoding was fast and simple. The code was rewritten in a straight forward way, and the new tile Viterbi decoder was created. After these jobs were done, simulations and experiences started. Results were obtained, a Data Flow graph was obtained and data dependencies were studied.

Results show that the insertion of the tile Viterbi decoder brought some benefit. Speed gains on DRM receiving around 5 % were achieved. This is an acceptable value considering that Viterbi decoding represents around 10 % of the DRM receptor. Although real execution times of a Viterbi accelerator are not known, simulations with really high execution times were made and demonstrated to have high speed gains also.

From the study made on data dependencies, one can conclude that there are always data dependencies when demodulating the same channel. It is not possible to shift phases in the same channel (FAC, SDC, MSC). It is possible though, to shift phases from different channels between them because they do not depend on each other’s unless the receiver is in synchronization mode. In this case, channels may depend on the information carried on other channels to be decoded. When not in synchronization, there are many possibilities to shift phases because Viterbi decoding consumes enough time to be able to map a task in TriMedia while Viterbi accelerator is working. The results show that latency introduced in the silicon network do not compromise speed gains obtained with the addition of a Viterbi accelerator. The possibility of shifting phases reinforces this idea.

6.3 Future Work

Design, test and implement solutions with phases shifted in many ways would be the next step.

Results show that reading, demodulating and storing phases consumes much more cycles than any other phase. It was verified that these read/storing phases have no data dependencies with other phases. The DRM application may take advantage on running these phases on other processor in an asynchronous way. For example, this processor could be reading, demodulating and storing FAC information (were most cycles are used), and perform MSC multistage decoding, producing and decoding an AAC

super frame at the same time. These two pieces of code use almost the same number of cycles, and this solution could decrease significantly the processing speed of the application.

As this was the first step regarding to DRM application on Hijdra Architecture, this work is left still with many other possible solutions in order to get even more efficient performances.

References

[1] M. Bekooij, O. Moreira, P. Poplavko, B. Mesman, M. Pastrnak, and J. van Meerbergen, "*Predictable embedded multiprocessor system design*", Procedings International Workshop on Software and Compilers for Embedded Systems (SCOPES), September 2004.

[2] K Goossens, J. Dielissen and A. Radulesco. *The AEthereal network on chip: Concepts, Architecture, and Implementations.* In IEEE Design and Test of Computers, Vol22 (5):21-31, Sept-Oct 2005

[3] E.A. Lee and D.G. Messerschmitt. "*Synchronous data flow*". Proceedings of the IEEE 1987.

[4] A. Moonen, *Modelling and simulation of guaranteed throughput channels of a hard real-time multiprocessor system*, Master's thesis, Eindhoven University of Technology, January 2004.

[5] E. de Kock and G. Essink, *Y-chart Application Programmer's Interface*, version 1.1, Philips Research 2000.

[6] European Telecommunication Standard Institute (ETSI), Sophia Antipolis, France, *Digital Radio Mondiale (DRM); System Specification*, ETSI ES 201 980 edition, April 2003.

[7] http://home.netcom.com/%7Echip.f/viterbi/algrthms2.html - A Tutorial on Convolutional Coding with Viterbi Decoding by Chip Fleming of Spectrum Applications. Updated 2006-11-02 19:35.

[8] G. Ungerboeck, "*Channel Coding with Multilevel / Phase Signals*", IEEE Trans. Inf. Theory, vol. JT-28, pp. 55-67, January 1982.

[9] *Presentation on Multilevel Codes and Iterative Multistage Decoding* by M. Jaber Borran and Behnaam Aazhang Rice University

[10] Digital Radio Mondiale: key technical features by Jonathan Stott

[11] *Presentation on Improved Multistage Decoding of Multilevel Codes for Digital Radio Mondiale (DRM)* by Volker Fischer, Alexander Kurpiers and Florian Kulla from Institute for Communication Technology Darmstadt University of Technology

[12] International Organization for Standardization (ISO), *Information technology – Coding of audio-visual objects* - Part 3: Audio, ISO/IEC 14496-3 edition, 2001

[13] *ON DECODING OF MULTI-LEVEL MPSK MODULATION CODES* Technical Report to NASA Goddard Space Flight Center Greenbelt , Maryland 20771 Grant Number NAG 5-931 Report Number NASA 90-003 Shu Lin Principal Investigator Department of Electrical Engineering University of Hawaii at Manoa Honululu, Hawaii 96822 May 20,1990

[14] *Partitioning of a DRM Receiver* by Pascal T. Wolkotte, Gerard J.M. Smit, Lodewijk T. Smit University of Twente, Department of EEMCS P.O. Box 217, 7500AE Enschede, The Netherlands

[15] Efficient Computation of Buffer Capacities for Cyclo-Static Real Time Systems with Back-Pressure - Maarten Wiggers 1, Marco Bekooij 2 , Pierre Jansen 1, Gerard Smit University of Twente, Enschede, The Netherlands NXP Semiconductors Corporate Research, Eindhoven, The Netherlands

[16] A Multi-Core Architecture for In-Car Digital Entertainment - Arno Moonen[1,2,3], Ren´e van den Berg[2], Marco Bekooij[3], Harpreet Bhullar[2] and Jef van Meerbergen[1,3]

[17] The TM3270 Media-processor - Jan-Willem van de Waerdt

[18] "The how and why of COFDM" Jonathan Stott. EBU: *EBU Technical Review* 278 (winter 1998).

[19] G. D. Forney. The Viterbi algorithm. *Proceedings of the IEEE* 61(3):268–278, March 1973

[20] van de Waerdt, J.-W.; Vassiliadis, S.; Sanjeev Das; Mirolo, S.; Yen, C.; Zhong, B.; Basto, C.; van Itegem, J.-P.; Dinesh Amirtharaj; Kulbhushan Kalra; Rodriguez, P.; van Antwerpen, H. Microarchitecture, 2005. MICRO-38. Proceedings. 38th Annual IEEE/ACM International Symposium on Volume , Issue , 12-16 Nov. 2005 Page(s): 12 pp. - Digital Object Identifier 10.1109/MICRO.2005.35

[21] Gilles Kahn, "The Semantics of a Simple Language for Parallel Programming," in *Proc. of the IFIP Congress 74*. 1974, North-Holland Publishing Co.

[22] Edward A. Lee and Thomas M. Parks, "Dataflow process networks," *Proceedings of the IEEE*, vol. 83, no. 5, pp. 773–799, May 1995.

[23] G. Bilsen, M. Engels, R. Lauwereins, and J. A. Peperstraete. Cyclo-static data flow. In *IEEE Int. Conf. ASSP*, pages 3255– 3258, Detroit, Michigan, May 1995.

Appendix A – Multilevel encoding/decoding

Introduction

Conventionally in digital communication, channel coding is designed and performed separately from modulation. In the cases of both block codes and convolutional codes, error control is achieved by replacing k-tuple message with a well-structured n-tuple codeword, where n > k. Transmission of these (n-k) redundant symbols requires either a bandwidth expansion or a reduction of data rate. Either case results in lowering the information rate per channel bandwidth, known as the bandwidth efficiency. This type of channel coding is suitable for power limited channels without bandwidth constraints, where bandwidth efficiency is traded for increased power efficiency and coding gain or reliability is achieved at the expense of bandwidth expansion or reduction of data rate. However, using a combined modulation and coding scheme known as coded modulation or bandwidth efficient coding, that achieves coding gain with little or no bandwidth expansion. At first, it may seem that this statement violates some basic power-bandwidth-error probability trade-off principle. However, there is still a trade-off at work. Coded modulation achieves coding gain at the expense of increased decoder complexity. In 1982, Ungerbock [5] showed that by combining coding and modulation properly, significant coding gain over uncoded modulation systems can be achieved without compromising bandwidth efficiency. Since that, a great deal of research effort has been expended in bandwidth efficient coded modulation for achieving reliable communication on band limited channels [4].

Multilevel encoding and modulation

The channel encoding process on DRM is based on a multilevel coding scheme. The principle of multilevel coding is to join optimization of coding and modulation to reach the best transmission performance. This denotes that more error prone bit positions in the QAM mapping get a higher protection. The different levels of protection are reached with different component codes which are realized with punctured convolutional codes, derived from the same mother code.

The decoding in the receiver can be done by a multistage decoder, either straightforwardly or through an iterative process. Consequently the performance of the decoder with errored data can be increased with the number of iterations and hence is dependent on the decoder implementation.

Depending on the signal constellation and mapping used, five different schemes are applicable. For simplicity we will consider a less complex and more general case

Ungerboeck rules

The Euclidean distance between the valid paths on the trellis diagram must be the biggest possible. With this objective Ungerboeck established some rules to obtain better TCM (Trellis coded Modulation) schemes [8]. With Ungerboeck rules we establish a biunivoc correspondence between paths on the trellis diagram and the sequence of the transmitted signal. As said above decoding is done using the Viterbi decoder. Combining sequence of transmitted bits and the modulation helps us getting a stronger code. For that, Ungerboeck established a mapping systematic procedure called "set partitioning". Considering an 8PSK (simpler than QAM 64 used by DRM) constellation with radius $(E_s)^{1/2}$, let's divide it in 2, 4 and 8 constellations as shown in Figure 26

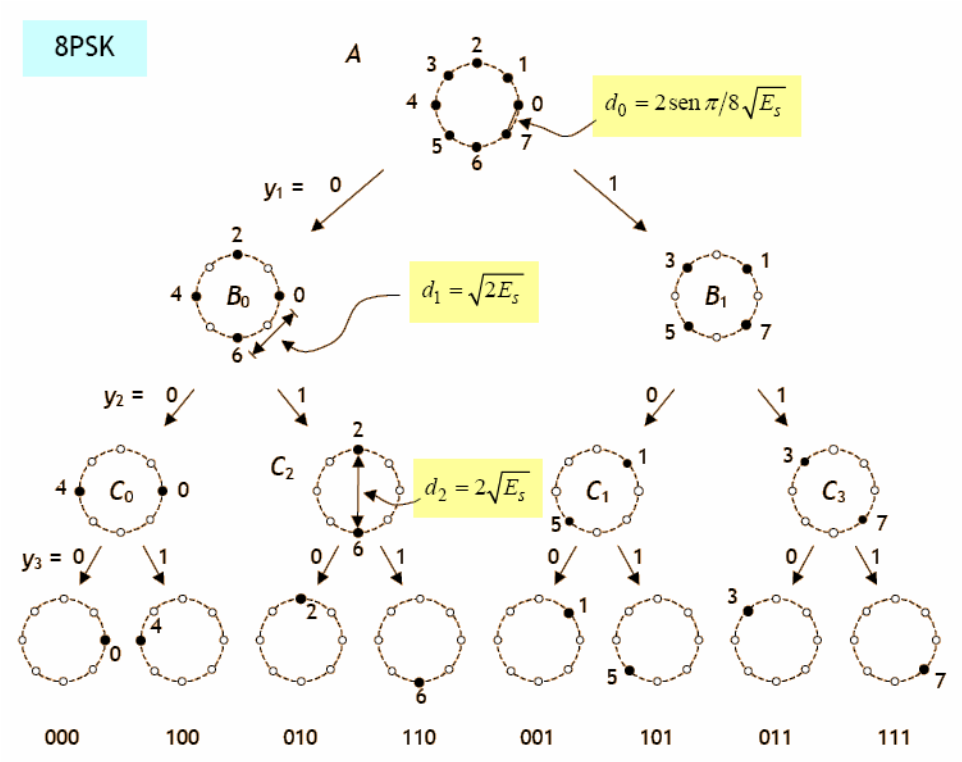


Figure 26. Set partitioning of 8PSK

In the original constellation the minimum distance between dots is d0 = 2sen (π / 8) $(E_s)^{1/2}$ = 0,765$(E_s)^{1/2}$. In the next groups the distance between dots gets significantly larger.

The 8PSK was considered to simplify the understanding. DRM uses 4QAM, 16QAM or 64, QAM. The partitioning is analogue. The important thing is to make Euclidian distances grow when dividing the constellation. Notice that in rectangular constellations like QAM, the Euclidean distance growth is made by a regular pattern, characteristic that does not verify in non rectangular constellations like MPSK.

The idea of multilevel coding is to protect each address bit of the signal point by an individual binary code at that same level. At the receiver, each code is decoded individually starting from the lowest level and taking into account decisions of prior decoding stages. But what does this mean? Figure 27 [9] illustrates a general multilevel encoder and the corresponding multistage decoder.

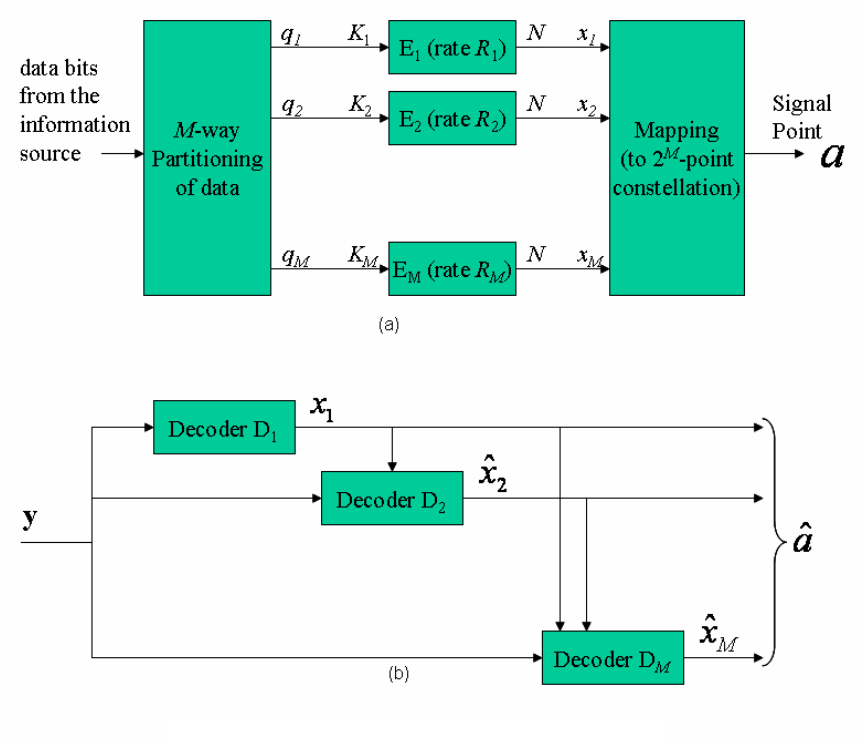


Figure 27. (a) – Multilevel encoder; (b) – Multistage decoder

As said above the most significant bits are coded at a higher rate, i.e. less strongly coded. So the multistage decoder first decodes the LSB from the constellations of a block of received data cells. This gives the corresponding stream of data originally sent. The stream can then be re-encoded, so that on revisiting the same received constellation points, the receiver knows (to a certain reliability) which LSB were mapped onto them. This simplifies the decision as to which the MSB must have been sent. In effect the distance between a 0 and a 1 for the MSB has been increased, and decoding thus more reliable. It is possible to perform multiple iterations of this decoding process, in each case using the recently decoded results to improve the reliability of the next step of decoding. In doing this, a modest performance benefit can be obtained [10].

For better understanding, consider for example that the mapper output is a 6. That corresponds on the dot illustrated on Figure 28.

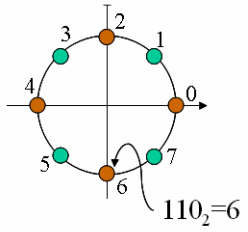


Figure 28. 8PSK constellation

Decoder D1 will decode the LSB first (LSB was highly coded), and if correct, X1 will correspond to 0. Knowing the LSB = 0, dots 1, 3, 5 and 7 are obviously excluded from the constellation. When the decoder gets to the MSB he only has to chose between two dots that have a big distance between them, making the decoding more reliable even with the MSB at a high rate. Resuming, combining channel coding with modulation, gives us a powerful transmission scheme.

Interleaving

Figure 29 [8] illustrates a multilevel encoder/multistage decoder for DRM.

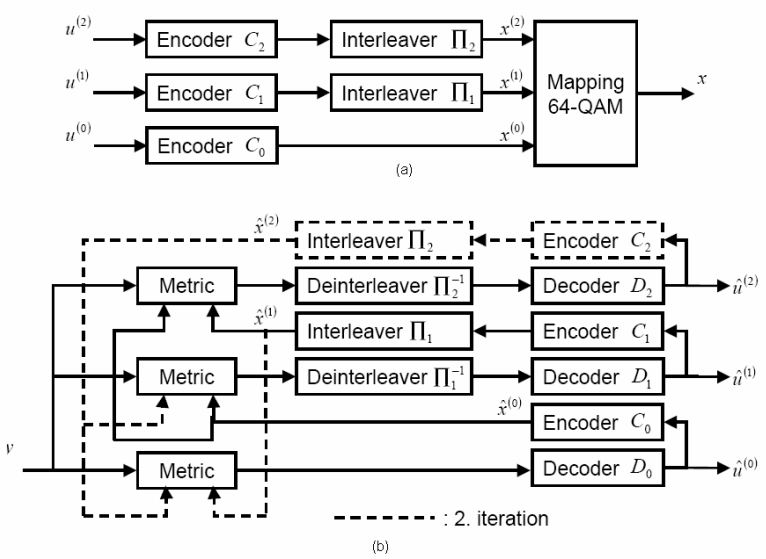


Figure 29. (a) – Multilevel encoder; (b) – Multistage decoder

Although the encoder provides us the capability to detect and correct errors (to a certain limit), when interference occurs, it usually occurs in a certain time span affecting many bits in the same place, mak-

ing impossible to correct errors at the receiver. In order to get around this problem, an interleaver is used to shuffle bits before they are sent in a wireless connection. When an error burst occurs, it will then affect bits from many frames and not just one or two, making possible the detection and correction of the error. The situation is illustrated on Figure 30.

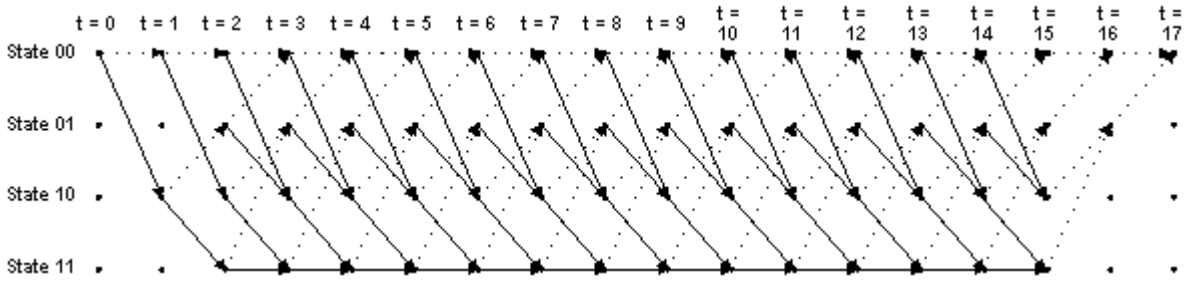
Figure 30. Interleaving

A cell-wise interleaving shall be applied to the QAM symbols (cells) of the MSC after multilevel encoding with the possibility to choose low or high interleaving depth (denoted here as short or long interleaving) according to the predicted propagation conditions. The basic interleaver parameters are adapted to the size of a multiplex frame which corresponds to *NMUX* cells. For propagation channels with moderate time-selective behaviour (typical ground wave propagation in LF and MF) the short interleaving provides sufficient time- and frequency diversity for proper operation of the decoding process in the receiver (spreading of error bursts). The same block interleaving scheme as used for bit interleaving in the multilevel encoder is always applied to the *NMUX* cells of a multiplex frame.

Appendix B – Viterbi decoder

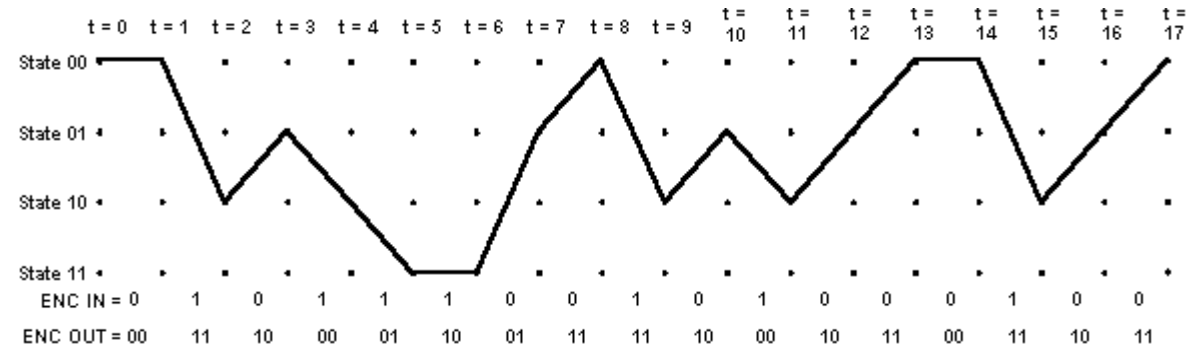
Performing Viterbi Decoding

Perhaps the single most important concept to aid in understanding the Viterbi algorithm is the trellis diagram. The figure below shows the trellis diagram for our example rate 1/2 K = 3 convolutional encoder, for a 15-bit message:



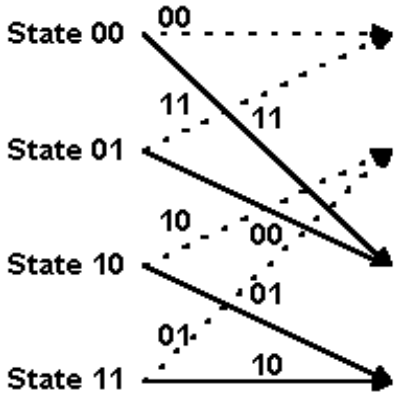
The four possible states of the encoder are depicted as four rows of horizontal dots. There is one column of four dots for the initial state of the encoder and one for each time instant during the message. For a 15-bit message with two encoder memory flushing bits, there are 17 time instants in addition to t = 0, which represents the initial condition of the encoder. The solid lines connecting dots in the diagram represent state transitions when the input bit is a one. The dotted lines represent state transitions when the input bit is a zero. Notice the correspondence between the arrows in the trellis diagram and the state transition table discussed above. Also notice that since the initial condition of the encoder is State 00_2, and the two memory flushing bits are zeroes, the arrows start out at State 00_2 and end up at the same state.

The following diagram shows the states of the trellis that are actually reached during the encoding of our example 15-bit message:



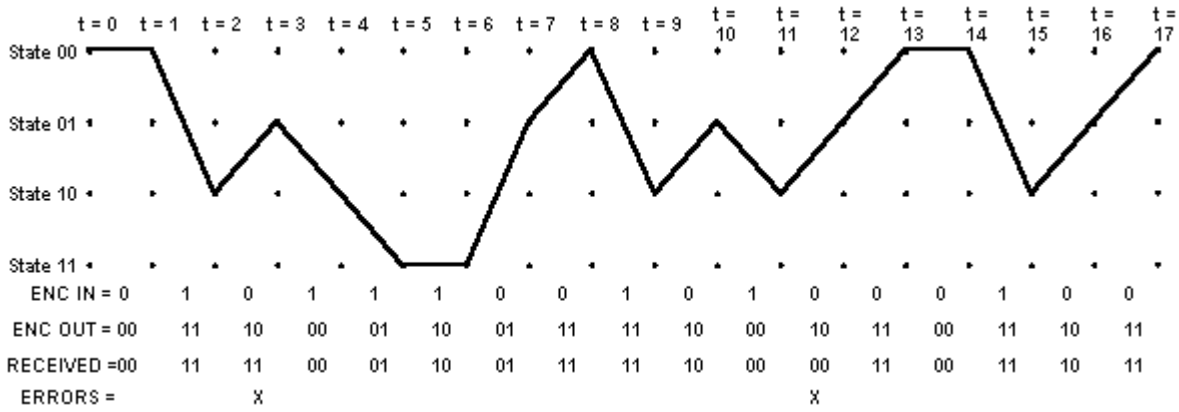
The encoder input bits and output symbols are shown at the bottom of the diagram. Notice the correspondence between the encoder output symbols and the output table discussed above. Let's look at that

in more detail, using the expanded version of the transition between one time instant to the next shown below:



The two-bit numbers labeling the lines are the corresponding convolutional encoder channel symbol outputs. Remember that dotted lines represent cases where the encoder input is a zero, and solid lines represent cases where the encoder input is a one. (In the figure above, the two-bit binary numbers labeling dotted lines are on the left, and the two-bit binary numbers labeling solid lines are on the right.)

OK, now let's start looking at how the Viterbi decoding algorithm actually works. For our example, we're going to use hard-decision symbol inputs to keep things simple. (The example source code uses soft-decision inputs to achieve better performance.) Suppose we receive the above encoded message with a couple of bit errors:

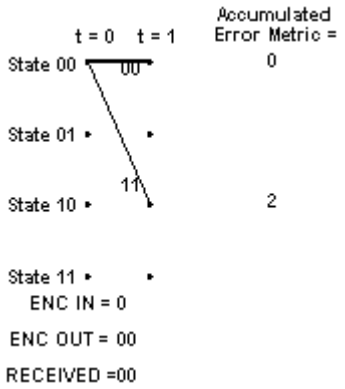


Each time we receive a pair of channel symbols, we're going to compute a metric to measure the "distance" between what we received and all of the possible channel symbol pairs we could have received. Going from t = 0 to t = 1, there are only two possible channel symbol pairs we could have received: 00_2, and 11_2. That's because we know the convolutional encoder was initialized to the all-zeroes state, and given one input bit = one or zero, there are only two states we could transition to and two possible outputs of the encoder. These possible outputs of the encoder are 00_2 and 11_2.

The metric we're going to use for now is the Hamming distance between the received channel symbol pair and the possible channel symbol pairs. The Hamming distance is computed by simply counting

how many bits are different between the received channel symbol pair and the possible channel symbol pairs. The results can only be zero, one, or two. The Hamming distance (or other metric) values we compute at each time instant for the paths between the states at the previous time instant and the states at the current time instant are called branch metrics. For the first time instant, we're going to save these results as "accumulated error metric" values, associated with states. For the second time instant on, the accumulated error metrics will be computed by adding the previous accumulated error metrics to the current branch metrics.

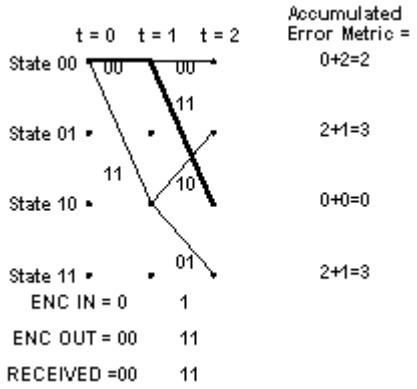
At t = 1, we received 00_2. The only possible channel symbol pairs we could have received are 00_2 and 11_2. The Hamming distance between 00_2 and 00_2 is zero. The Hamming distance between 00_2 and 11_2 is two. Therefore, the branch metric value for the branch from State 00_2 to State 00_2 is zero, and for the branch from State 00_2 to State 10_2 it's two. Since the previous accumulated error metric values are equal to zero, the accumulated metric values for State 00_2 and for State 10_2 are equal to the branch metric values. The accumulated error metric values for the other two states are undefined. The figure below illustrates the results at t = 1:



Note that the solid lines between states at t = 1 and the state at t = 0 illustrate the predecessor-successor relationship between the states at t = 1 and the state at t = 0 respectively. This information is shown graphically in the figure, but is stored numerically in the actual implementation. To be more specific, or maybe clear is a better word, at each time instant t, we will store the number of the predecessor state that led to each of the current states at t.

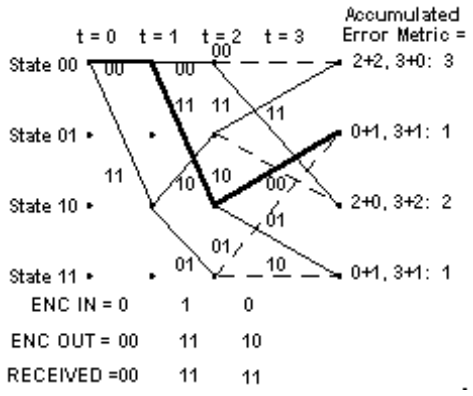
Now let's look what happens at t = 2. We received a 11_2 channel symbol pair. The possible channel symbol pairs we could have received in going from t = 1 to t = 2 are 00_2 going from State 00_2 to State 00_2, 11_2 going from State 00_2 to State 10_2, 10_2 going from State 10_2 to State 01_2, and 01_2 going from State 10_2 to State 11_2. The Hamming distance between 00_2 and 11_2 is two, between 11_2 and 11_2 is zero, and between 10_2 or 01_2 and 11_2 is one. We add these branch metric values to the previous accumulated error metric values associated with each state that we came from to get to the current states. At t = 1, we could only be at State 00_2 or State 10_2. The accumulated error metric values associated with those states

were 0 and 2 respectively. The figure below shows the calculation of the accumulated error metric associated with each state, at t = 2.



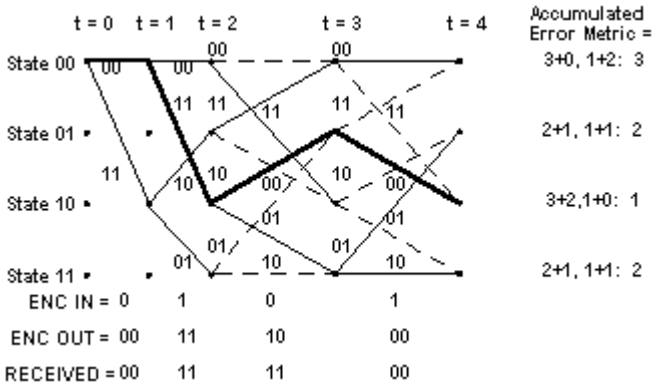
That's all the computation for t = 2. What we carry forward to t = 3 will be the accumulated error metrics for each state, and the predecessor states for each of the four states at t = 2, corresponding to the state relationships shown by the solid lines in the illustration of the trellis.

Now look at the figure for t = 3. Things get a bit more complicated here, since there are now two different ways that we could get from each of the four states that were valid at t = 2 to the four states that are valid at t = 3. So how do we handle that? The answer is, we compare the accumulated error metrics associated with each branch, and discard the larger one of each pair of branches leading into a given state. If the members of a pair of accumulated error metrics going into a particular state are equal, we just save that value. The other thing that's affected is the predecessor-successor history we're keeping. For each state, the predecessor that survives is the one with the lower branch metric. If the two accumulated error metrics are equal, some people use a fair coin toss to choose the surviving predecessor state. Others simply pick one of them consistently, i.e. the upper branch or the lower branch. It probably doesn't matter which method you use. The operation of adding the previous accumulated error metrics to the new branch metrics, comparing the results, and selecting the smaller (smallest) accumulated error metric to be retained for the next time instant is called the add-compare-select operation. The figure below shows the results of processing t = 3:

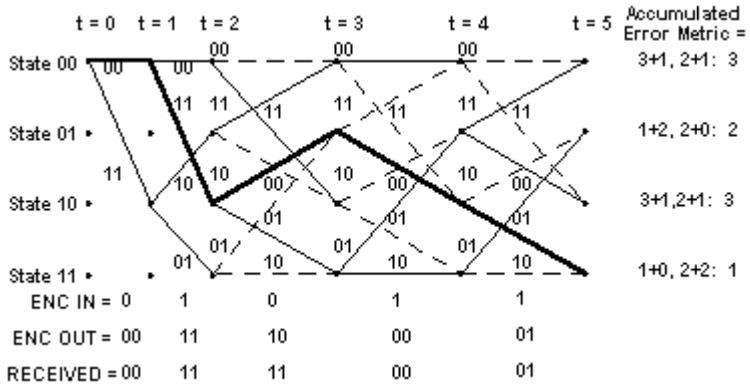


Note that the third channel symbol pair we received had a one-symbol error. The smallest accumulated error metric is a one, and there are two of these.

Let's see what happens now at t = 4. The processing is the same as it was for t = 3. The results are shown in the figure:



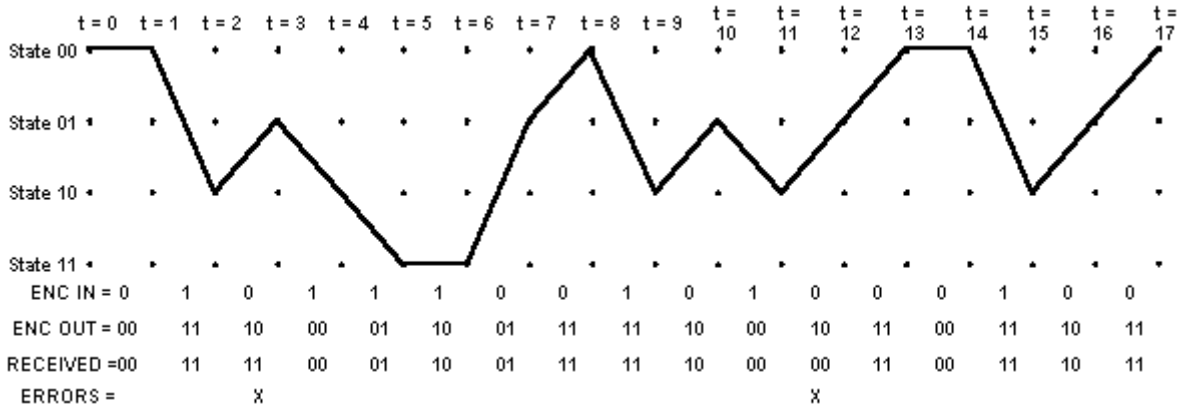
Notice that at t = 4, the path through the trellis of the actual transmitted message, shown in bold, is again associated with the smallest accumulated error metric. Let's look at t = 5:



At t = 5, the path through the trellis corresponding to the actual message, shown in bold, is still associated with the smallest accumulated error metric. This is the thing that the Viterbi decoder exploits to recover the original message.

Perhaps you're getting tired of stepping through the trellis. I know I am. Let's skip to the end.

At t = 17, the trellis looks like this, with the clutter of the intermediate state history removed:



The decoding process begins with building the accumulated error metric for some number of received channel symbol pairs, and the history of what states preceded the states at each time instant t with the smallest accumulated error metric. Once this information is built up, the Viterbi decoder is ready to recreate the sequence of bits that were input to the convolutional encoder when the message was encoded for transmission. This is accomplished by the following steps:

First, select the state having the smallest accumulated error metric and save the state number of that state.

Iteratively perform the following step until the beginning of the trellis is reached: Working backward through the state history table, for the selected state, select a new state which is listed in the state history table as being the predecessor to that state. Save the state number of each selected state. This step is called trace back.

Now work forward through the list of selected states saved in the previous steps. Look up what input bit corresponds to a transition from each predecessor state to its successor state. That is the bit that must have been encoded by the convolutional encoder.

The following table shows the accumulated metric for the full 15-bit (plus two flushing bits) example message at each time t:

t =	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
State 00_2		0	2	3	3	3	3	4	1	3	4	3	3	2	2	4	5	2
State 01_2			3	1	2	2	3	1	4	4	1	4	2	3	4	4	2	
State 10_2		2	0	2	1	3	3	4	3	1	4	1	4	3	3	2		
State 11_2			3	1	2	1	1	3	4	4	3	4	2	3	4	4		

It is interesting to note that for this hard-decision-input Viterbi decoder example, the smallest accumulated error metric in the final state indicates how many channel symbol errors occurred.

The following state history table shows the surviving predecessor states for each state at each time t:

t =	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
State 00_2	0	0	0	1	0	1	1	0	1	0	0	1	0	1	0	0	0	1
State 01_2	0	0	2	2	3	3	2	3	3	2	2	3	2	3	2	2	2	0
State 10_2	0	0	0	0	1	1	1	0	1	0	0	1	1	0	1	0	0	0
State 11_2	0	0	2	2	3	2	3	2	3	2	2	3	2	3	2	2	0	0

The following table shows the states selected when tracing the path back through the survivor state table shown above:

t =	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
	0	0	2	1	2	3	3	1	0	2	1	2	1	0	0	2	1	0

Using a table that maps state transitions to the inputs that caused them, we can now recreate the original message. Here is what this table looks like for our example rate 1/2 K = 3 convolutional code:

	Input was, Given Next State =			
Current State	$00_2 = 0$	$01_2 = 1$	$10_2 = 2$	$11_2 = 3$
$00_2 = 0$	0	x	1	x
$01_2 = 1$	0	x	1	x
$10_2 = 2$	x	0	x	1
$11_2 = 3$	x	0	x	1

Note: In the above table, x denotes an impossible transition from one state to another state.

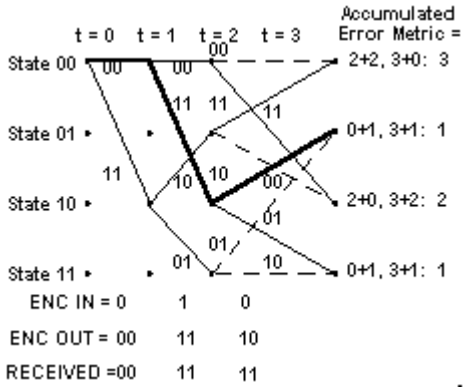
So now we have all the tools required to recreate the original message from the message we received:

t =	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
	0	1	0	1	1	1	0	0	1	0	1	0	0	0	1

The two flushing bits are discarded.

Here's an insight into how the trace back algorithm eventually finds its way onto the right path even if it started out choosing the wrong initial state. This could happen if more than one state had the small-

est accumulated error metric, for example. I'll use the figure for the trellis at t = 3 again to illustrate this point:



See how at t = 3, both States 01_2 and 11_2 had an accumulated error metric of 1. The correct path goes to State 01_2 -notice that the bold line showing the actual message path goes into this state. But suppose we choose State 11_2 to start our trace back. The predecessor state for State 11_2 , which is State 10_2 , is the same as the predecessor state for State 01_2! This is because at t = 2, State 10_2 had the smallest accumulated error metric. So after a false start, we are almost immediately back on the correct path.

For the example 15-bit message, we built the trellis up for the entire message before starting trace back. For longer messages, or continuous data, this is neither practical nor desirable, due to memory constraints and decoder delay. Research has shown that a trace back depth of K x 5 is sufficient for Viterbi decoding with the type of codes we have been discussing. Any deeper trace back increases decoding delay and decoder memory requirements, while not significantly improving the performance of the decoder. The exception is punctured codes, which I'll describe later. They require deeper trace back to reach their final performance limits.

To implement a Viterbi decoder in software, the first step is to build some data structures around which the decoder algorithm will be implemented. These data structures are best implemented as arrays. The primary six arrays that we need for the Viterbi decoder are as follows:

A copy of the convolutional encoder `next state` table, the state transition table of the encoder. The dimensions of this table (rows x columns) are $2^{(K-1)}$ x 2^k. This array needs to be initialized before starting the decoding process.

A copy of the convolutional encoder `output` table. The dimensions of this table are $2^{(K-1)}$ x 2^k. This array needs to be initialized before starting the decoding process.

An array (table) showing for each convolutional encoder current state and next state, what input value (0 or 1) would produce the next state, given the current state. We'll call this array the `input` table. Its dimensions are $2^{(K-1)}$ x $2^{(K-1)}$. This array needs to be initialized before starting the decoding process.

An array to store state predecessor history for each encoder state for up to K x 5 + 1 received channel symbol pairs. We'll call this table the state history table. The dimensions of this array are $2^{(K - 1)}$ x (K x 5 + 1). This array does not need to be initialized before starting the decoding process.

An array to store the accumulated error metrics for each state computed using the add-compare-select operation. This array will be called the accumulated error metric array. The dimensions of this array are $2^{(K - 1)}$ x 2. This array does not need to be initialized before starting the decoding process.

An array to store a list of states determined during traceback (term to be explained below). It is called the `state sequence` array. The dimensions of this array are (K x 5) + 1. This array does not need to be initialized before starting the decoding process.

Before getting into the example source code, for purposes of completeness, I want to talk briefly about other rates of convolutional codes that can be decoded with Viterbi decoders. Earlier, I mentioned punctured codes, which are a common way of achieving higher code rates, i.e. larger ratios of k to n. Punctured codes are created by first encoding data using a rate 1/n encoder such as the example encoder described in this tutorial, and then deleting some of the channel symbols at the output of the encoder. The process of deleting some of the channel output symbols is called puncturing. For example, to create a rate 3/4 code from the rate 1/2 code described in this tutorial, one would simply delete channel symbols in accordance with the following puncturing pattern:

1	0	1
1	1	0

One indicates that a channel symbol is to be transmitted, and a zero indicates that a channel symbol is to be deleted. To see how this make the rate be 3/4, think of each column of the above table as corresponding to a bit input to the encoder, and each one in the table as corresponding to an output channel symbol. There are three columns in the table, and four ones. You can even create a rate 2/3 code using a rate 1/2 encoder with the following puncturing pattern:

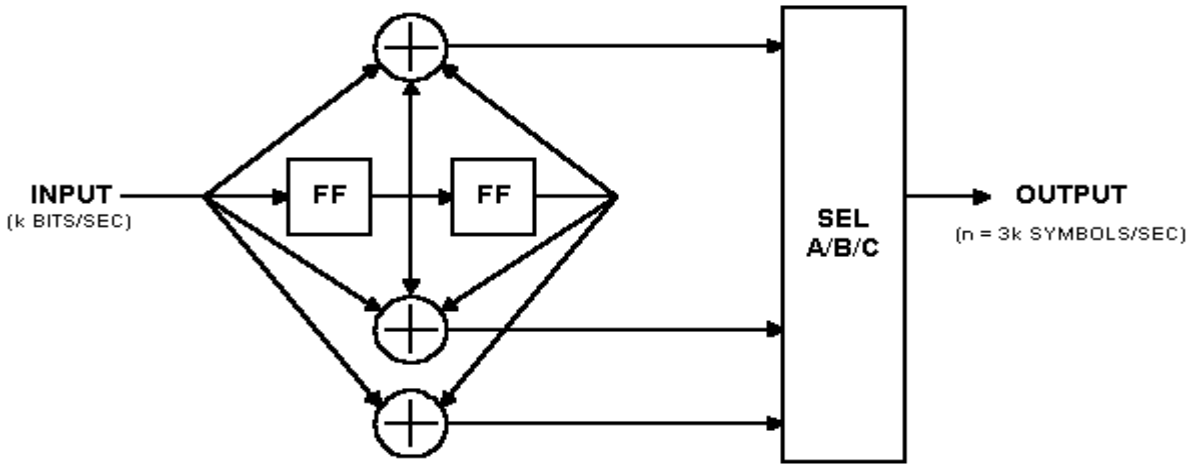
1	1
1	0

which has two columns and three ones.

To decode a punctured code, one must substitute null symbols for the deleted symbols at the input to the Viterbi decoder. Null symbols can be symbols quantized to levels corresponding to weak ones or

weak zeroes, or better, can be special flag symbols that when processed by the ACS circuits in the decoder, result in no change to the accumulated error metric from the previous state.

Of course, n does not have to be equal to two. For example, a rate 1/3, K = 3, (7, 7, 5) code can be encoded using the encoder shown below:



This encoder has three modulo-two adders, so for each input bit, it can produce three channel symbol outputs. Of course, with suitable puncturing patterns, you can create higher-rate codes using this encoder as well.

Note: All information in this Appendix is from http://home.netcom.com/~chip.f/viterbi/tutorial.html.

Appendix C – User guide

Software

The software developed on this work can be found on “drm”, “drm_vitsplit”, “drm_graph”, “drm_sdf” and “drm_csdf” folders.

Drm

Catena’s original code, AAC decoder and functions to produce an audible wav file.

Drm_vitsplit

Viterbi was separated from original code and it is running on a separated task.

Drm_graph

Synchronization is being performed in a different task (drm_sync.cc).

Drm_sdf

The code on this folder was an attempt to first design the SDF graph implement it, and then put the respective code inside each task. This method revelled to create many problems.

Drm_csdf

This folder contains code written in a straightforward way. Writes and reads for/from Viterbi task are out of for loops and if statements. This code permits a big flexibility in shifting phases.

Results

Measurements can be consulted on CSDF_DRM_graph.xls and ViterbiAccelerator.xls.

ViterbiAccelerator.xls

This file contains measurements made on Viterbi.

CSDF_DRM_graph.xls

This file contains three worksheets (Phases, Graph, SDC, MSC, FAC and Reads AAC dbget).

Phases have a detailed description on the relation between the code and phases. It also has some figures illustrating data dependencies.

Reads AAC dbget worksheet show measurements of the cycles needed for read/store on phases SDC1, FAC1 and MSC1 for each frame of a super frame. These values are in table named “Reads”. The table on the top-left has cycles/instruction measurements of two super frames. These measurements where made with debugging messages at the beginning and the end of DRM receiving function (number of instructions is precise). Measurements of cycles after channel decoding (mlpDecode) are in table pos-mlcpDecode. Cycles regarding to the inside instructions of channel decoding are discarded. “Confirm Data” table assures that values are correct (compare to table on top-left).

Worksheet Phases contain all the most important data measured. On the left, there are three tables named frame 0, frame 1 and frame 2. These tables have the values of each phase discarding reading cycles and cycles corresponding to the code after channel decoding (the opposite situation of Reads AAC dbget). Tables below these ones have the number of cycles of the read/store action and cycles corresponding to the code after channel decoding. Tables on right show the final result (sum of the rest of the tables). Data values can be confirmed with values on Reads AAC dbget.