Aalborg University

Institute of Electronic Systems
Department of Communication Technology
Fredrik Bajers Vej 7
DK–9220 Aalborg Ø
Phone +45 98158522

TITLE: Design and Implementation of an MPEG/Audio Layer III Bitstream Processor

SUBJECT: Applied Signal Processing and Implementation

PROJECT PERIOD: S10   97.02.02 – 97.06.03

PROJECT GROUP: S-1014

PARTICIPANTS:
Kent Salomonsen

Sten Søgaard

Eddie Proft Larsen

SUPERVISOR:
Steen Ditlev Sørensen

ABSTRACT:

The topic for this masters thesis is the design and implementation of a microprocessor to be applied in an MPEG-1/Audio Layer III decoder. The Layer III decoding scheme is a complex compound of algorithms ranging from digital signal processing to data unpacking and interpretation. It is shown that by dedicating the latter algorithm types to a Bitstream Processor, only a small general purpose DSP will be required for a Layer III decoder.
In order to find the operational demands to the Bitstream Processor, a complexity analysis of the Layer III decoding algorithm was conducted. Also, an architectural survey was performed to set up a set of design guidelines for the Bitstream Processor.
The Bitstream Processor is based on a traditional Von Neumann architecture. A wide variety of arithmetic, logic and Layer III specific operations are supported. Both internal and external nested interrupts have been implemented, and bus arbitration is supported by the processor.
The Bitstream Processor was implemented in VHDL, and the description is fully synthesizeable. The algorithms to be executed in the Bitstream Processor was written in an easy to use assembly programming language. An assembler was constructed to translate the language into machine code.
Testing was conducted on the Bitstream Processor executing the Layer III algorithms. Based on an existing reference decoder, both the Bitstream Processor and the software was successfully validated. A synthesis in Mentor Graphics AutoLogic has shown that all the timing requirements are met for an implementation in 0.7μ standard cells.

NUMBER PRINTED: 9

NUMBER OF PAGES: 237

Synopsis

Målet for dette afgangsprojekt er at designe og implementere en microprocessor til brug i en MPEG-1/Audio Layer III dekoder. Layer III dekodningsprincippet består af en kompleks samling af algoritmer strækkende ligefra digital signal behandling til data udpakning og fortolkning. Det er vist at en flytning af de sidstnævnte algoritmer til en bitstrøms processor, er det muligt at sænke kravene til en digital signal processor, og derved opnå et billigt dekoderdesign.

For at opstille kravene til bitstrømsprocessoren, er der blevet foretaget en kompleksitets analyse. Forskellige processorarkitekturer er blevet analyseret, og udfra disse analyser er der blevet sat nogle retningslinier til brug under udviklingen af bitstrømsprocessoren.

Bitstrømsprocessoren er baseret på en traditionel "Von Neumann" arkitektur, og er i stand til at udføre aritmetiske, logiske og Layer III specifikke operationer. Bitstrømsprocessoren understøtter både interne og eksterne interrupts, og det er muligt for processoren at fungere sammen med andre processorer på en fælles bus.

Bitstrømsprocessoren er blevet implementeret i VHDL, og beskrivelsen er fuld syntetiserbar. Algoritmerne som bliver udført på bitstrøms processoren er skrevet i et brugervenligt assembler sprog, og en assembler er konstrueret til at oversætte assemblerprogrammet til maskinkode. Bitstrøms processoren blev testet ved at lade den udføre de dedikerede Layer III algoritmer, og ved at sammenholde resultaterne med en reference dekoder. Bitstrømsprocessoren blev syntetiseret ved hjælp af Mentor Graphics AutoLogic, og resultatet viste at processoren opfylder alle timingskrav ved en implementation i 0.7μ standard celler.

Preface

This report is the result of the work performed by project group S1014 on the 10th semester of system construction at the Department of Communication Technology, at the Institute of Electronic Systems – Aalborg University.

The report is divided into three parts, one containing the main report, one containing the appendices and one containing source code. All figures, equations, tables and algorithms are referenced by the number of the chapter or the letter of the appendix, followed by consecutive numbers. Citations are made as the last name of the author and the year of publishing in square brackets, e.g. [Pan, 1995]. These references correspond to a bibliography listing placed at the end of the main report.

The following is a short description of how the report is laid out:
Chapter 1, *Introduction*. This chapter is brief presentation of the MPEG/Audio compression standard, and its applications.
Chapter 2, *MPEG/Audio Layer III.*The MPEG/Audio Layer III compression method is described. The focus of this description is to give an understanding of which computations are required in the Layer III decoder algorithms. This presentation includes both encoder and decoder, though emphasis has been put on the decoder.
Chapter 3, *Design Specification*. The algorithms in Layer III are divided into typical DSP algorithms and algorithms requiring bitstream handling. When the bitstream part has been defined, the demands to the bitstream processor are defined.
Chapter 4, *Computations Types in the BSP*. This chapter describes the algorithms of the bitstream part of Layer III in detail. This chapter serves as a foundation for the later implementation of the Layer III algorithms on the BitStream Processor (BSP).
Chapter 5, *Algorithmic Solutions to Layer III*. Layer III involve Huffman decoding and CRC calculations. In this chapter specific solutions to these calculation are evaluated.
Chapter 6, *Complexity Analysis*. This chapter presents the exact demands to the BSP, in terms of the number of instruction count.
Chapter 7, *Overall Design of the BSP*. Some overall considerations concerning the processor architecture are presented. The result is some guidelines for the design of the bitstream processor.
Chapter 8, *BSP Hardware*. The hardware of the bitstream processor is designed.
Chapter 9, *BSP Assembly Language*. The instruction set is designed, and a algebraic assembly language with assembler is constructed.
Chapter 10, *BSP Software*. The assembler program required to decode a MPEG/Audio Layer III bitstream is presented.
Chapter 12, *Evaluation of the Layer III Decoder*. The bitstream processor is tested and profiled by letting it decode some real life bitstreams.

Enclosed with the report is a folder titled "User's Manual for the BSP Assembler" and a floppy disk. The floppy disk contains the VHDL code describing the bitstream processor, and assembly code to the bitstream processor in order to decode a MPEG/Audio Layer III bitstream. The VHDL and assembly code is also listed in Part III. The floppy disk also contains the BSP assembler used for translating the assembly code into machine code.

Contents

Part I

Main Report

1 Introduction

In radio and television broadcasting the trend is toward replacing analogous transmission technology by digital technology. Digital Audio Broadcast (DAB) is expected to replace the current mode of FM radio within the next twenty years [Shlien, 1994]. High Definition TeleVision (HDTV) has already been taken into use in Japan, and in USA the "Digital HDTV Grand Alliance" is completing a digital HDTV standard to fulfill the requirements for North America.

The reason for the ongoing change in transmission technology is found in benefits like better noise immunity, potentially better performance at low transmission bandwidth and a reduction in transmission power. Though a change in transmission technology can be very advantageous, an important problem will arise. Raw digitized video and sound data requires high bandwidth and therefore are not suitable for long distance transmission. In order to regain the above mentioned benefits some preprocessing resulting in a major data reduction must be applied to the raw video and sound data.

1.1 Introducing Data Reduction

The need for data reduction arises in many applications, and therefore various methods have been developed. In the following a small survey on some of the commonly applied methods will be conducted.

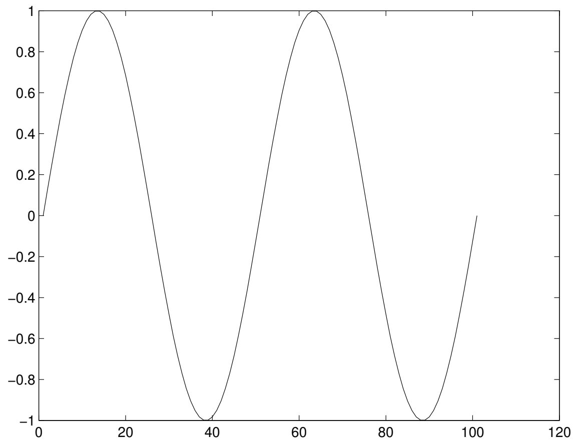
A crude but simple way to reduce sound and video data is to requantize every sample linearly. Considering sound only, requantization from typically 16 bit down to 8 bit will reduce the amount of data to only 50 percent. The prize payed for the data reduction is a substantial degradation of the signal to noise ratio, making transmission of high fidelity sound impossible. In terms of video a similar requantization will result in a reduction in the color depth, leading to an unnatural color reproduction. Linear requantization cannot provide any significant data reduction without causing too severe degradation on the signal.

An enhancement of linear requantization would be to apply either A-law or μ-law amplitude compression prior to quantization. In short, both methods amplify small values and attenuate larger values. In Figure 1.1 an example of a μ-law compression is shown for a sinusoidal signal.

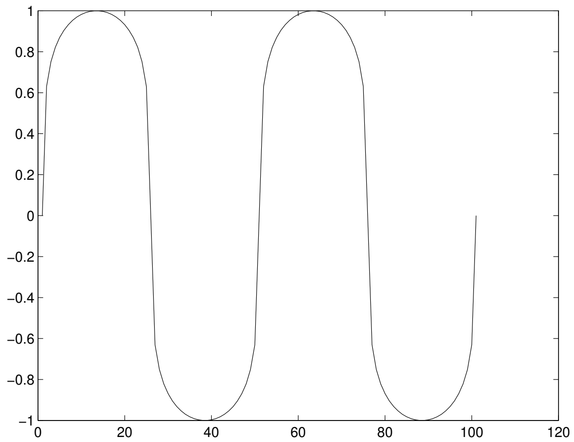
For a signal normalized to 1, a bigger share of the signal will be concentrated in the vicinity of 1, and thereby benefit from the finer quantization large values are offered.

Performing a non-linear quantization using a rate of 8 bit per sound sample will result in a better signal to noise ratio than the corresponding linear quantized signal using 8 bit per sample. Using non linear quantization only, is not suited for the above described applications because only a negligible data reduction is obtainable.

A third method that could be applied in conjunction with one of the above quantization methods is to resample data down to a lower sampling frequency. An example would be to resample music from a CD from a standard sampling frequency at 44.1 kHz down to 22.05 kHz, after having applied an appropriate low pass filtering. Performing a reduction in this manner will halve the initial data amount. The data reduction is obtained at the expense of halving



(a)



(b)

Figure 1.1: Illustration of a sinusoid **(a)** and the compressed μ-law representation **(b)**.

the sound signal frequency range. For some music genres, like classic for instance, important information in the music will therefore be missing in the reproduction. Resampling a picture by a factor 2 will lead to pixel sizes 4 times bigger than initially. Resampling can obviously not provide any major data reductions without introducing severe quality degradation.

The above reduction methods are classified as lossy data reduction methods. Algorithms performing lossless data reduction, i.e. data compression where no quality degradation is introduced, also exist and might therefore be suitable for reducing the amount of sound and video data for transmission. Some of the most commonly used algorithms are the Huffman algorithm and the Shannon Fano algorithm. In short, both algorithms obtain an overall data compression by substituting frequently appearing bit patterns by short bit patterns and infrequently appearing bit patterns by longer bit patterns. If a big share of the data, to be reduced, is concentrated among only few bit patterns both algorithms can provide a high reduction ratio. Unfortunately this is seldom the case for music, speech and video, and therefore lossless reduction alone will not apply for reduction of sound and video data.

From the above it is evident that the solution to the data reduction problem is not found by means of simple algorithms. Some attempts has been conducted in order to combine the above methods with psychoacoustics, i.e. heuristic models of the human ear. From the more successful attempts, audio compression schemes like "Audio Spectral Perceptual Entropy Coding" (ASPEC) and "Masking pattern adapted Universal Subband Integrated Coding And Multiplexing" (MUSICAM) has proven to offer fairly good compromises between compression rate and quality preservation. In the next section a more competent solution based on a refinement of the better properties from ASPEC and MUSICAM will be introduced.

1.2 The MPEG-1 Algorithm

The upcoming need for a highly efficient and quality preserving data reduction algorithm led the International Organization for Standardization (ISO) to initiate the development of a common standard for compressing digital audio and video signals. A work group within ISO called the Moving Pictures Expert Group (MPEG) was established to develop an appropriate compression scheme. The first standard called MPEG-1 was accomplished in November 1992.

The MPEG-1 encoder achieves a high compression rate for audio signals by exploiting the perceptual limitations of the human ear, and thereby removes perceptual irrelevant information in the audio signal. The limitations of the human ear arise if a dominating tone is present for a period of time. Frequencies in the vicinity of this tone are less audible if their energy is below a certain threshold called a masking threshold. The tone so to speak masks out the nearby frequencies for the period the tone is present. This effect allows for a more coarse quantization of the nearby masked frequency components, without introducing audible degradation. In Figure 1.2 an illustration of the tonal masking effect is shown. The curve denoted “Masking Threshold” illustrates the frequency band within which other frequencies below the Masking Threshold are masked out.

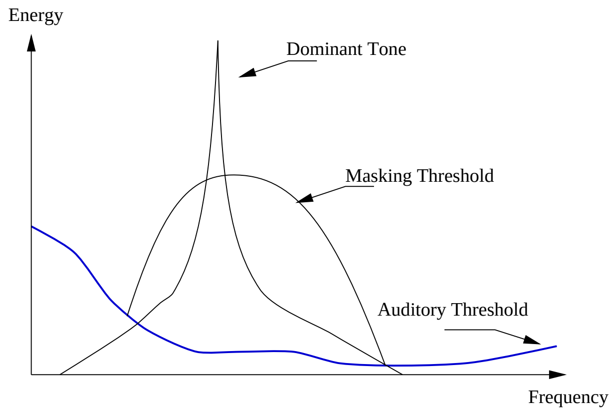


Figure 1.2: *Illustration of a dominant tone masking out the signal energies in the vicinity with an energy below the masking threshold. The Masking threshold curve shown is normalized to an arbitrary hearing threshold.*

Perceptual limitations will also be introduced when a transient appears in an audio signal. If for instance a piano tone is stroked firmly, a small period will arise where the human ear is less sensible to the frequencies in the transient and below. The term masking is also applied in this case. In the period the masking effect is present a more coarse quantization of the masked frequency components is allowed without introducing perceptual quality degradation.

compressed with the MPEG-1 algorithm cannot be restored to the original signal, but to a signal which is perceptually identical, to the human ear.

The MPEG-1 standard developed consists of three parts. An audio part (ISO/IEC 11172-3) was defined to cover compression and formatting of high quality sound [ISO/IEC 11172-3, 1993]. The audio part allows for compression ratios ranging from 2.4 to 24. A video part (ISO/IEC 11172-2) was defined to cover compression and formatting of low resolution video, that is pictures within 320 times 200 pixels. Finally, a system part (ISO/IEC 11172-1) describing how to transmit multiple audio and video signals on a single distribution media was defined. The three parts in conjunction allow for transmission of video and associated sound at a bitrate of approximately 1.5 Mbit/s.

Figure 1.3 illustrates an example of how the three parts in the MPEG-1 standard can be combined. The configuration in the illustration is a conjunction of two combined audio and video sources broadcasting on a shared media. Both sources generates an audio bitstream and a video bitstream to be multiplexed into one single system bitstream. In the decoding end, the System Demultiplexer passes on an audio and a video bitstream for decoding, depending on which source has been selected by the Source Selector. The system bitstream is not restricted to combine one audio and one video bitstream only. Generally, the system bitstream supports all combinations of audio only, video only and combined audio and video bitstreams.

MPEG-1 is a generic compression standard. This ensures that any decoder compliant with the MPEG-1 standard is capable of decoding a bitstream generated by an arbitrary encoder compliant with the MPEG-1 standard. Though an encoder or decoder must comply with the

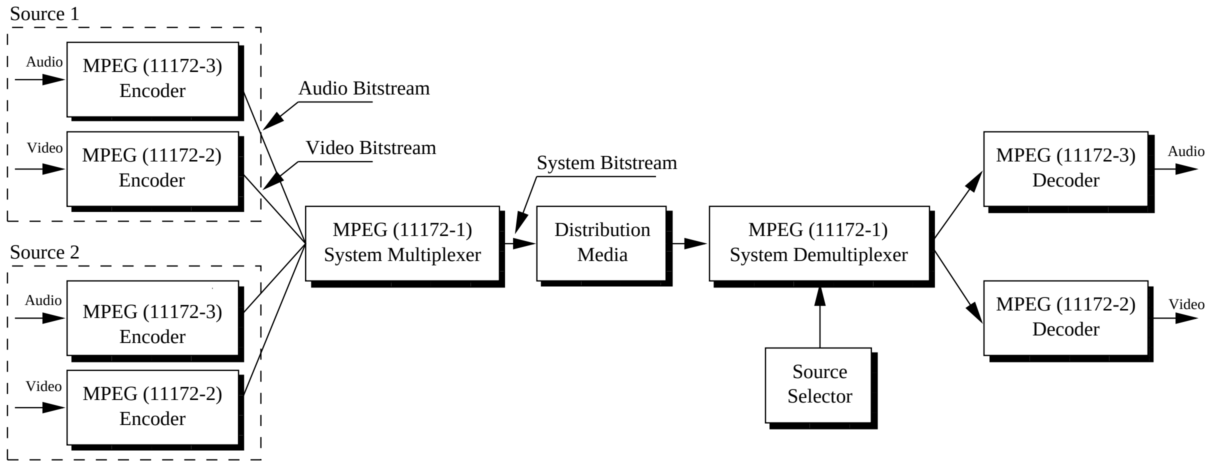


Figure 1.3: *An MPEG encoder/decoder system. The audio and video signal from both sources are compressed and formatted into a single bitstream, which can be either broadcasted or stored on a digital media. From this media a source is chosen by the Source selector, and the associated bitstreams are decoded into an audio and a video signal.*

standard, much freedom is left for the designer to choose from. An extension of MPEG-1, called MPEG-2 was developed in November 1994. MPEG-2 is a consequent extension from two to five audio channels providing backward compatibility.

Within the MPEG-1 standard, three different compression schemes called Layers are offered. The difference between the Layers are in the complexity and thereby the respective Layers capability to obtain high compression ratios without introducing perceptual degradation. An introduction to the three Layers will be conducted in the following section.

Both the video and the audio part can function as stand alone units and can therefore be developed independently. In this project only the audio part is dealt with, and therefore focus will be on the (ISO/IEC 11172-3) part of the MPEG-1 standard. In the following chapters the (ISO/IEC 11172-3) part will be referred to as the MPEG/Audio standard.

1.2.1 Applications for the Three Compression Layers

The MPEG/Audio standard offers three levels of compression, each with an increasing complexity. The three levels are denoted Layer I, Layer II and Layer III. Layer I has the most simple compression scheme, and therefore requires the highest transmission bandwidth to achieve high-fidelity quality. Layer I is used in Philips' Digital Compact Cassette (DCC) with 192 kbit/s per channel. Layer II has an intermediate complexity, and is able to achieve high-fidelity audio quality at 128 kbit/s per channel. Layer II is well suited for DAB and storage on CD-ROM. Layer III has the most complex compression scheme, and targets bitrates around 64 kbit/s per channel [Pan, 1995]. The existence of the superior Layer III scheme does not make the Layer I and II schemes less attractive. The increased complexity in Layer III makes codec implementations, i.e. combined coder-decoders, considerably more expensive than Layer I and II codecs. Therefore Layer III codecs are economically attractive in broadcasting systems where transmission bandwidth is small or expensive, whereas Layer I and II codecs are economically attractive in closed systems (e.g. DCC) where a wide and inexpensive bandwidth is present. An example of the Layer III complexity can be found in some of the real–time Layer III software decoders written for Intel x86 compatible computers. A real time decoder, which is significantly more simple than the corresponding real time encoder, demands at least a Pentium™ 75 MHz processor in order to function properly.

Layer III has proven itself useful for a large spectrum of applications when it comes to transmission of audio. An actual example of where the MPEG/Audio Layer III has been employed is the private broadcasting station *Radio FFN* in Niedersachsen, Germany. They replaced their leased phone lines with ISDN and Layer III codecs, to transmit 8 local programs 20 minutes per day to the central broadcasting studio. This move saved them transmission fees of more than 300.000 US$ per year. Another example of the usage of MPEG/Audio Layer III, was during the Winter Olympics in Albertville. During the games, all private radio stations in Germany used Layer III codecs as reporter links between the various sporting events and their central studio in Meribel [Popp, 1997].

The German publisher, Bertelsmann, is providing a new CD-ROM encyclopedia "Discovery 97" providing information for approximately 100.000 key words and rich multimedia information with 150 minutes of sound samples coded with MPEG/Audio Layer III [Popp, 1997].

1.3 Motivation

The complexity of the algorithms required in an MPEG/Audio decoder depends on the layer chosen for implementation. The complexity increases with the number of layer so that Layer III includes the most complex set of algorithms. In fact, a quite big share of the Layer III algorithm involves bitstream handling, which in essence is an interpretation of the bitstream in which many data dependencies exist. Having performed the bitstream handling, what remains is typical digital signal processing, that is multiplications and accumulations.

The general purpose Digital Signal Processor (DSP) is developed for implementation of a wide variety of algorithms. Algorithms well suited for DSP implementation are characterized by multiply-accumulate operations (MAC) and linear data access. Hence for applications involving special DSP unfriendly algorithms, e.g. algorithms with many data dependencies and bit-wise operations, the result will be a poor utilization of the processor, i.e. big parts of the processor features are kept idle for long periods. The poor hardware utilization is caused by the fact that these algorithms cannot exploit the properties offered by the MAC unit, the extended bus architecture and the advanced addressing options available in most DSP processors.

Literature describes examples considering different DSP implementations of MPEG/Audio decoders, for instance [Papamichalis et al., 1995] who implemented a combined MPEG/Audio Layer I and II decoder using a TMS320C3x floating point processor. This implementation seems reasonable because the Layer I and II algorithms are dominated by DSP friendly operations like multiplication, addition and deterministic control loops.

The combination of the bitstream handling and DSP friendly operations in the MPEG/Audio Layer III algorithm increases the demands on a general purpose DSP. The demands can however be met by high performance DSP's such as Texas TMS320C80 and Analog Devices AD21060 SHARC, but at the expense of a poor processor utilization.

An implementation using high performance DSP's is expensive, and thus the alternative of designing an ASIC becomes more attractive. Literature presents a number of ASIC solutions to the MPEG/Audio decoder problem e.g. PHILLIPS and SGS THOMSON, who both present combined Layer I and II decoders and [Witte et al., 1994] who have implemented an fully ISO compliant MPEG/Audio Layer III decoder in an ITT MASC 3500 processor. A combination of DSP and ASIC is seen in [Stabernack et al., 1994], who use two ITT MASC 3500 processors and a Texas TMS320C40 to implement an MPEG-2/Audio decoder. Though the latter implementation utilizes two ITT MASC processors, the implementation seems appropriate compared to [Witte et al., 1994] because MPEG-2/Audio considers five audio channels compared to maximally 2 audio channels in MPEG-1/Audio.

From study of literature on the MPEG/Audio Layer III decoding algorithm, a solution consisting of both a bitstream processor, i.e. a processor dedicated for bitstream handling, and a low cost DSP seems to be a cost efficient solution. This hypothesis stems from the fact that when excluding the bitstream handling, the demand for computation power is approximately the same for the Layer III and Layer II algorithms [Pan, 1995]. Therefore, a TMS320C30 or a similar inexpensive DSP must be capable of handling the DSP friendly part of the Layer III algorithm. If an inexpensive bitstream processor is constructed, a low cost Layer III decoder can be designed. The Layer III decoder in concern is illustrated in Figure 1.4.

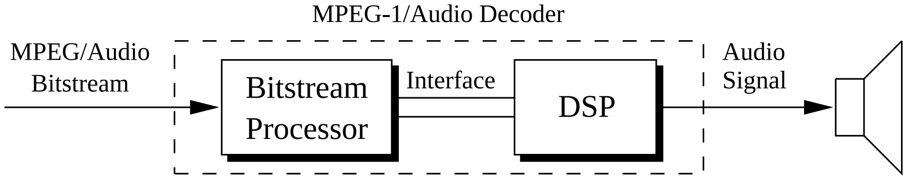


Figure 1.4: *A sketch of an MPEG/Audio Layer III decoder system.*

1.4 Objectives and Strategies

The overall objective of this project is to design a processor, referred to as a BitStream Processor (BSP). The BSP is intended to process the sub algorithms of the Layer III algorithm, belonging to the bitstream handling. The thesis is that the presence of a BSP in an MPEG/Audio Layer III decoder as sketched in Figure 1.4 will require only a small and inexpensive DSP.

In the project, emphasis will be on the design and software implementation of the bitstream processor, that is a fully synthesizeable VHDL description of the BSP. The goal will be to obtain a VHDL simulation of the BSP and the software written for processing an MPEG/Audio Layer III bitstream. Having obtained the goal, the first steps toward an implementation can be initiated. The BSP part of the project terminates at this point with concluding remarks on the cost for an implementation in standard cell technology.

In order to accept the thesis saying that only a small and inexpensive DSP will be required for the remaining Layer III processing, a complexity analysis on the DSP part of Layer III will be conducted.

The strategy to obtain the objectives for this project is summarized in the paragraphs below.

- Obtain a precise description of the algorithms to be performed in the BSP.
- Perform a complexity analysis on algorithms to be executed in the BSP.
- Perform a complexity analysis of the algorithms to be executed in a DSP, in order to estimate the demands to be fulfilled by the DSP.
- Analyze possible architectures for the BSP.
- Analyze if there is a need for special hardware units within the BSP.
- Analyze possible interfaces to a DSP.
- Write a VHDL description for the BSP.
- Write the software to be executed in the BSP.
- Perform a profiling on the software running on the BSP in order to evaluate the architecture.
- Perform a synthesis in Mentor Graphics AutoLogic in order to perform a cost estimate.

2

MPEG/Audio Layer III

This chapter describes the algorithms in the MPEG/Audio Layer III compression standard. The chapter is an initial survey used as a foundation for a later definition of which algorithms within Layer III belong to the bitstream handling and which algorithms belong to the DSP part.

Although, only parts of the decoder will be implemented in this project, the following description also covers the audio encoding part in order to form a general view of the encoding/decoding process. The encoder description will be held at a level where only the functionality will be considered, whereas the description of the decoder will contain enough information to perform a classification of the algorithms involved.

2.1 MPEG/Audio Layer III Encoder Overview

In this section the MPEG/Audio Layer III encoder will briefly be described with emphasis put on the functionality. The description of the encoding process is based on the block diagram in Figure 2.1. Here the audio signal is assumed to be a single channel Pulse Code Modulated (PCM) signal sampled with a rate of 48 kHz and quantized to 16 bit. The MPEG/Audio encoder compresses the input signal to a coded bitstream and thereby reduce the bit rate from 768 kbit/s to some arbitrary bitrate kbit/s.

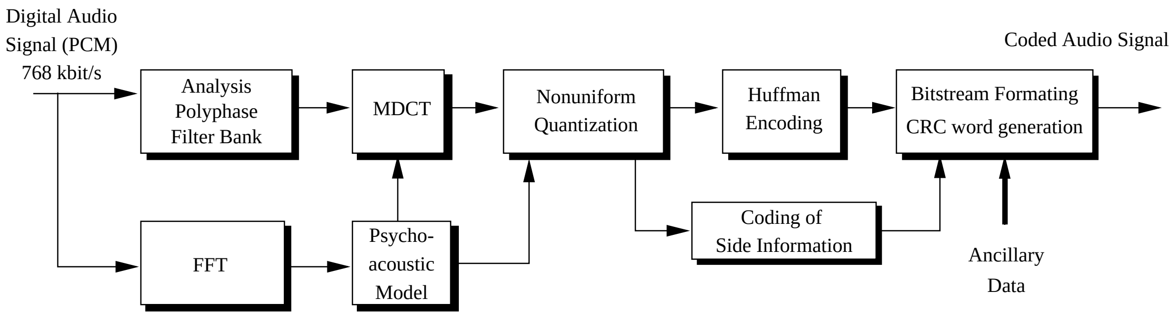


Figure 2.1: *Block diagram of MPEG/Audio encoder.*

Analysis Polyphase Filter Bank: The first step in the encoding process is the filtering of the audio signal through a filter bank. In this process a sequence of 1152 PCM audio samples are filtered by a parallel structure of bandpass filters into 32 equally spaced subbands and decimated by a factor 32. Each subband will thereby contain 36 subband samples. Assuming perfectly square bandpass filters, the Nyquist theorem guarantees that the original 1152 PCM samples can be reconstructed, by interpolating the subbands to their original sampling frequency followed by a summation of the 36 subbands. Since it is not possible to construct bandpass filters with a perfectly square frequency response, some aliasing will be introduced by the decimation. The alias contribution will be taken care of at a later point in the decoder.

The samples in each subband are still in the time domain, but calculating the total energy within each frequency band results in a energy distribution in the frequency domain. The output from the analysis polyphase filter bank can therefore be considered as a coarse spectral representation of the time samples, using only 36 spectral values.

MDCT: In this process the 32 subbands are mapped into a Modified Discrete Cosine Transform (MDCT) representation. Performing this transformation will enhance the spectral resolution to 36 frequency lines per subband. Prior to the transformation a windowing of the subband samples is performed. The windowing can be performed using either long windows or short windows depending on the dynamics within each subband. If the subband samples within a given subband shows a stationary behavior, a long windows is chosen in order to enhance the spectral resolution in the following MDCT. If the subband samples contains transients, three consecutive short windows are applied in order to enhance the time resolution in the following MDCT. In order to obtain better adaption when window transitions are required, two windows referred to as start windows and stop windows, are defined. In Figure 2.2 the four applicable window types are illustrated. 18 frequency lines output by the **MDCT** block after applying either a long window or three short windows are referred to as a "block", that is a start block, an end block and a short block, corresponding to the start, short and stop windows respectively.

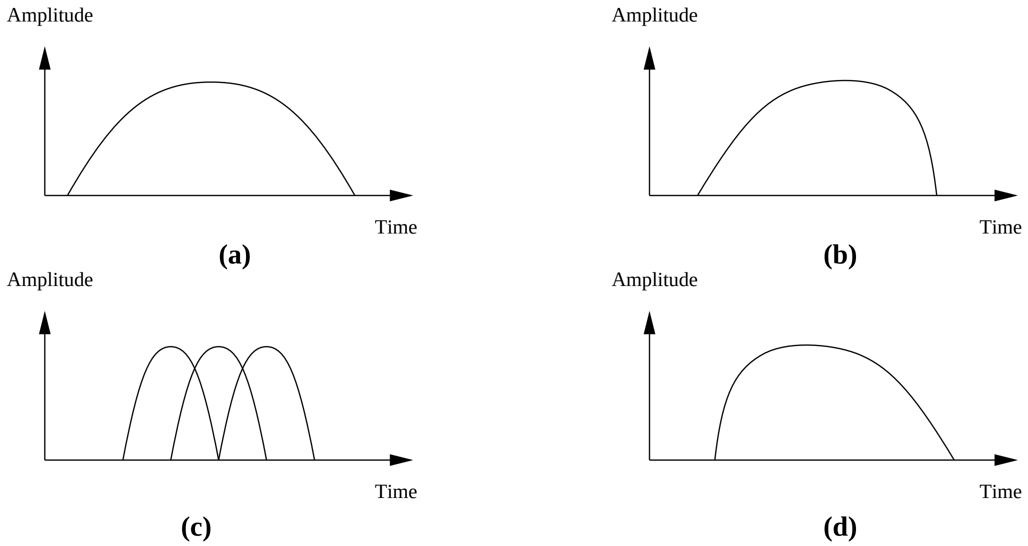


Figure 2.2: *Illustration of the four applicable window types. (a) corresponds to normal window type, (b) corresponds to start window type, (c) correspond to short windows type and (d) correspond to stop window type.*

The decision on which window types to apply is controlled by the Psycho Acoustic Model introduced later. Figure 2.3 shows an example of a sequence of windows applied to a subband.

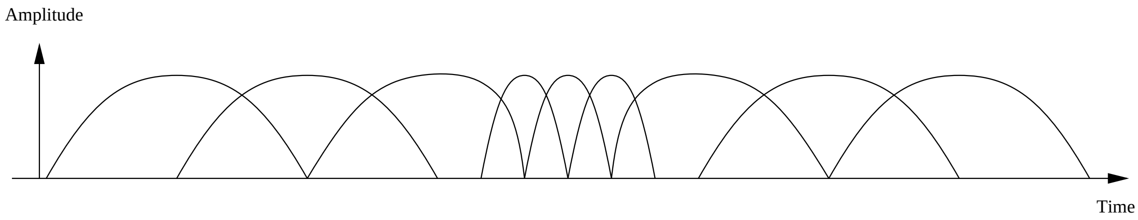


Figure 2.3: *Illustration of a typical sequence of windows to be applied to a subband.*

Performing an MDCT on a subband windowed with either of the long windows will produce 18 frequency lines due to 50 percent overlap. Using 3 short windows with 50 percent overlap will produce 3 groups of 6 frequency lines, each group belonging to different time intervals. Having performed one MDCT transformation will therefore produce 576 frequency lines referred to as a granule. The MDCT block produces 2 granules when transforming 32 subbands.

Before passing on the frequency lines a reduction of the aliasing introduced in the Analysis Polyphase Filter Bank is removed. The aliasing is removed at this early stage in order to reduce the amount of information for transmission. The reduction is obtained by means of a series of butterfly computations, see Figure 2.11.

In Figure 2.4 all the processing applied to the 1152 PCM samples are illustrated in one figure. At this point the signal processing model has transformed the initial PCM samples into 32 equally spaced subbands within the audio signal.

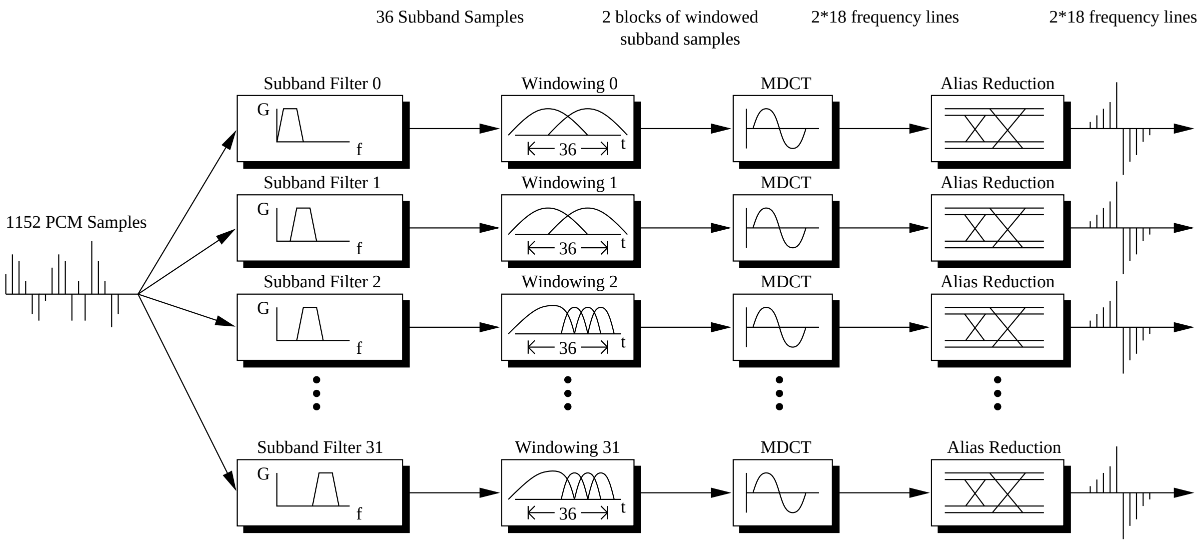


Figure 2.4: *Summary of the signal processing applied to the 1152 PCM samples.*

FFT: Concurrent to the Analysis Polyphase Filter bank calculations, two Fast Fourier Transformations (FFT) of the 1152 PCM samples are performed. Both a 1024 and a 256 point FFT are performed in order to provide a fairly high spectral resolution and information on the spectral changes over time. Obviously 1024 point will not give an optimal frequency domain representation of the 1152 samples in consideration. The number is chosen because an FFT, demands a power of two samples. Compared to an ordinary 1152 point Discrete Fourier Transform less than one percent computation power is required [Oppenheim and Schafer, 1989].

Psycho Acoustic Model: This block contains a set of algorithms that constitutes the Psycho Acoustic Model. In short, the Psycho Acoustic Model is a model of the human sound perception. The model is used in the encoder only, in order to decide which parts of the audio signal have perceptual relevance and which parts have not. The results of the psycho acoustic evaluations are utilized in the MDCT block and in the Nonuniform Quantization block.

The Psycho Acoustic Model supplies the MDCT block with information about which window type to apply. The decision is based on a measure of the differences between the two presently calculated FFT spectra and the two previous spectra. If a certain difference is present the Psycho Acoustic Model will call for a transition to short windows. When the differences fade away, a transition back to long windows will be called for.

The Psycho Acoustic Model also supplies the Nonuniform Quantization block with information on how to quantize the frequency lines. The quantization of the frequency lines is adapted to the limitations of the human ears perception of audio. Heuristic experiments have shown that the human ear has 24 frequency bands, called critical bands [Furui, 1989], in which the human ear is less frequency selective. When a dominant tonal component is present in an audio signal, frequencies in the associated critical band are not perceived very well. The dominant tonal component introduces a masking threshold below which frequencies in the same critical band are masked out. This effect allows for a more coarse quantization of the nearby masked frequency components, without introducing audible degradation. In the MPEG/Audio standard critical bands are approximated by scalefactor bands. The Psycho Acoustic Model analyzes the FFT spectrum to detect whether dominant tonal components are present. When dominant tones are present a masking threshold is calculated. Based on this threshold an upper limit for the quantization level required in the individual scalefactor bands is determined. Two psycho acoustic models applicable to the Layer III encoder are described in [ISO/IEC 11172-3, 1993], but other psycho acoustic models optimized toward certain properties can be implemented as well.

Nonuniform Quantization: In this block a nonlinear quantization of the frequency lines is performed. The nonlinearity is introduced by first raising each sample to the power of 3/4. In order to further reduce the quantization noise, a scaling of the frequency lines in each scalefactor band is performed prior to the nonuniform quantization. Hence the output from this block is quantized frequency lines and scalefactors applying for each of the scalefactor bands. The scaling and a grouping of the frequency lines into scalefactor bands is performed in accordance with the psycho acoustic model.

Huffman Encoding: In this block a coding of the scaled frequency lines is performed using the Huffman coding algorithm based on 32 static Huffman tables. The Huffman coding scheme is one of the major reasons, why the MPEG/Audio Layer III algorithm can retain a high audio quality at low transmission bit rates. The coding scheme provides lossless compression and thereby reduces the amount of data to be transmitted without degrading the quality.

The reduction of data is obtained through a coding of the entropy, e.g. the likeness, inherently present in most data. In Huffman coding the entropy is based on a statistic distribution of the group of data values considered. From the data statistics a substitution table covering all data values is established. In this table data values with a high probability of being present in the data are associated with short code words and data rarely present are associated with longer code words. Encoding a block of data will produce a new representation of the exact same data, but using fewer bits.

In order to enhance the compression rate both granules are subdivided into smaller partitions and encoded in pairs or quadruples. The partitioning enables the use of different Huffman tables to compress the quantized frequency lines. How to perform the partitioning is further explained in Section 4.2.

Coding of Side Information: In order to enable the decoder to reproduce the audio signal all parameters generated in the encoder must be provided. The Coding of Side Information block orders all the parameters used through the coding process. e.g. boundaries for certain data blocks, quantizer step sizes, which windows are used for MDCT etc.

Bitstream Formatting CRC Word Generation: In this block the Huffman coded frequency lines, the side-information and a frame header are assembled to form the bitstream. The bitstream is partitioned into frames each representing 1152 PCM samples. A Cyclic Redundancy Code (CRC) can optionally be included for data validation in the decoder. In Subsection 2.2.1 the frame format is further explained.

Ancillary Data: This data is typically used for features like informing on artist name or music category. The Ancillary Data area is user defined and optional for the designer to implement. The feature often appears in digital radio broadcasting.

2.1.1 Stereo Encoding

The introduction of the Layer III encoder presented so far only applies for encoding of single channel audio. The MPEG/Audio standard also defines a method for encoding a dual channel audio signal and two methods for stereo redundancy encoding. In the following only a short introduction to the three methods will be conducted in order to describe how the signal processing model in Figure 2.1 can be applied for dual channel and stereo encoding. More information on the stereo modes will be presented in the next section describing the Layer III decoder.

Encoding of a dual channel signal, i.e. a bilingual or non processed stereo audio signals, is accomplished by time sharing the processing model in Figure 2.1. Encoding a dual channel signal does not introduce extra complexity, because the two channels are encoded independently of each other.

The two stereo redundancy encoding methods supported in MPEG/Audio are Middle Side stereo (MS) and Intensity stereo. The first method transmits the stereo signal as the sum and the difference of the two channels. The two new channels are both transmitted as described for the dual channel model.

Intensity stereo requires only one channel for transmission. In this channel the sum of the two audio signal channels is transmitted.

2.2 MPEG/Audio Layer III Decoder Overview

Next the Layer III decoder will be described at functionality level. The description will be based on Figure 2.5.

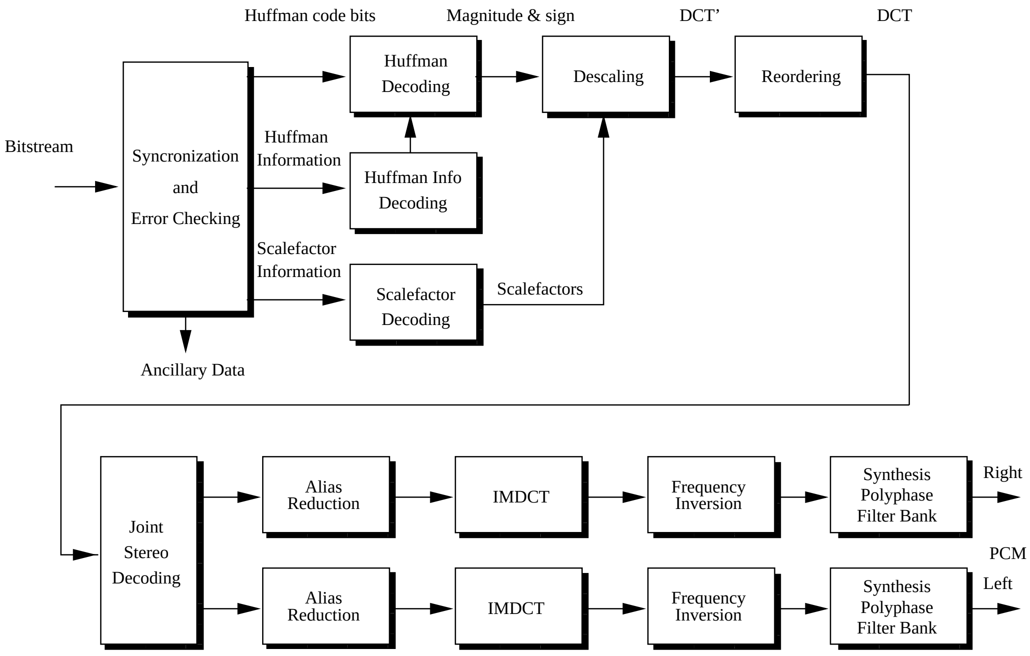


Figure 2.5: *Block diagram of MPEG/Audio decoder.*

2.2.1 Synchronization and Error Checking

The purpose of the **Synchronization and Error Checking** block is to receive the incoming bitstream, identify the contents of the bitstream and pass the information on to the succeeding blocks in the decoder. To form a foundation for what is needed in this block, this section begins with a description of the bitstream format. Later it is described how to synchronize to the bitstream, i.e. to identify where each frame begins in the bitstream and how to perform the error check.

The Format of the Bitstream

The contents of an MPEG/Audio bitstream is organized into frames, each containing information to reconstruct the initial PCM samples, 1152 for the example used in this chapter. A frame consist of five parts: header, CRC, side information, main data, and ancillary data, as shown in Figure 2.6. In the following the contents of the frames will be described.

Header (32)	**CRC** (0,16)	**Side Information** (136,256)	**Main Data**	**Ancillary Data**

Figure 2.6: *Layout of a Layer III frame. The values in parenthesis are the width of the part in bits, e.g. the Side Information block is 136 bits in single channel, and 256 bits in dual channel. The size of the main data area varies with the bitrate and sampling frequency. The size of ancillary data is user defined.*

Header: The header part of the frame contains a synchronization word and system information. To be able to detect the beginning of a new frame, each frame starts with a 12 bit synchronization word. The rest of the header describes the type of frame, that is, which layer is used in the frame, which bitrate is used for transmission, the sampling frequency, whether the audio signal coded was a single channel signal or dual channel signal, and some additional information which can be found in Appendix B. Some effort has been made to avoid spurious synchronization words to appear in other parts of the frame, but it is not guaranteed. To gain a more stable synchronization, all the bits in the header which are known in advance by the decoder, should be used as synchronization word.

CRC: An optional feature in each frame is the 16 bit CRC check word which provides some error detection facility to the decoder. Only the most sensitive information in each frame is included in the CRC check word, and is defined by the MPEG/Audio standard to be bit 16 to 31 in the header and the side information. The reason why only these data are included in the CRC word is that any error in these values can corrupt an entire frame, whereas an error in the main data only distorts a part of the frame. The Hamming distance of the CRC check word is $d = 4$, which allows for detection of up to 3 single bit errors or for detection of one error burst of up to 16 bit. If a frame is corrupted, the normal procedure is either to repeat the previous frame or mute the corrupted frame [ISO/IEC 11172-3, 1993].

Side Information: The side information section in the frame contains the necessary information to decode the main data. The side information contains information concerning which Huffman tables to use during the Huffman decoding process, and information for reconstruction of scalefactors. The side information section also contains information about where the main data begins due to the “bit reservoir” technique used in Layer III. The “bit reservoir will be described in the following paragraph. The width of the side information is 136 bits in single channel mode, and 256 bits in dual channel mode. A more thorough description of the side information format can be found in Appendix B and [ISO/IEC 11172-3, 1993].

Main Data: The main data section contains the coded scalefactor values and the Huffman coded data. In the following there will be referred to the coded scalefactor values and Huffman coded data as main data, and the section in each frame which is reserved to hold these data as main data area. For the Layer III bitstream a "bit reservoir" technique is applied. Here the size of the main data is allowed to vary depending on time-varying demands on Huffman code bits. When space is free in the main data area of a frame, it is allowed for succeeding frames to place main data in this area. Main data for one frame can thereby extend over multiple main data area's. An example of how the main data can be distributed is illustrated on Figure 2.7. The

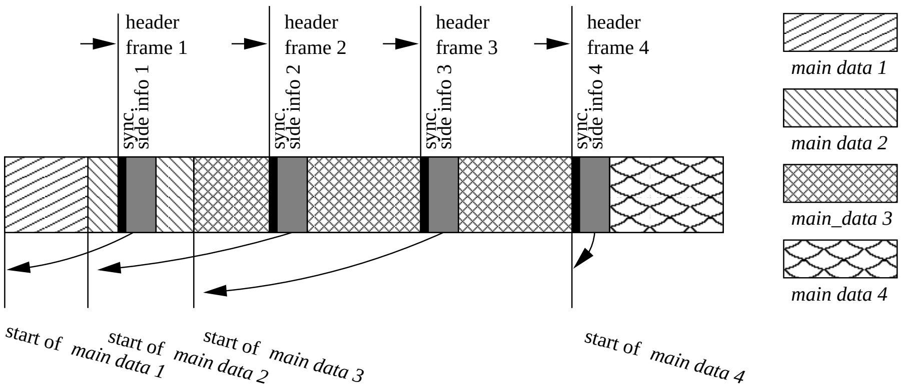


Figure 2.7: *An example Layer III bitstream organization. It is shown how the main data belonging to one frame can be placed in another frame. Frame 3 requires more code bits, hence it consumes the accumulated bit reservoir. Ancillary bits have been excluded from the figure for clarity.*

beginning of the main data can be found by reading a pointer placed in the side information. This pointer can be interpreted as the number of bytes in the bit reservoir. The width of this pointer is 9 bits, hence the bit reservoir cannot exceed $8 \cdot (2^9 - 1) = 4088$ bits. More information concerning the "bit reservoir" technique can be found in Appendix B.

Ancillary Data: It is possible to include an ancillary data section in each frame. The format of this ancillary data is user defined and can be used for, e.g. transmitting the title of the song in digital audio broadcasting applications. In Layer III the ancillary data is placed between the end of the main data bits in one frame and the start of the main data bits in the next frame, not including bits used for header and side information.

Synchronization:

The **Synchronization and Error Checking** block receives the incoming bitstream and identifies the location of each frame. When the location of a frame is known it is possible, for the succeeding blocks in the decoder, to extract all information contained in the frame, by using the specifications in the MPEG/Audio standard.

After startup the synchronization is conducted by searching for the synchronization word in the bitstream. The validity of the synchronization word can be tested by reading the header information and perform the CRC check on the header and side information, if available. If the CRC check fails, or if the information in the header does not correspond to what was expected, the synchronization word is discarded, and the search for a new synchronization word is started.

2.2.2 Huffman Decoding

In this block the decoding of the Huffman code bits is performed. The incoming Huffman code bits can be classified as Variable Length Code (VLC), that is, no attempt has been put into separating the individual code words in the Huffman code bits. Therefore, a single code word in the middle of the Huffman code bits cannot be identified without starting to decode from a point in the Huffman code bits known to be the start of a code word. Consequently, if an error occurs somewhere in a string of Huffman code bits, the remaining code bits cannot be decoded correctly.

To expand the dynamics for the Huffman coded frequency lines, ESCAPE values are introduced in the encoder. When quantized frequency lines exceed 15, only the value 15 is encoded. The remainder is represented in the Huffman code bits as an uncoded ESCAPE value. The Huffman decoder must detect the presence of ESCAPE values, and sign bits too, in order to reconstruct the quantized frequency lines.

The Huffman decoding calls for an algorithm that both decides when an incoming codeword terminates and which pair or quadruple of values to substitute for a code word. A variety of implementations, each emphasizing different benefits, e.g. speed and memory consumption, has been suggested for Huffman decoding. In Section 5.1 three different implementations of the **Huffman Decoding** are suggested.

2.2.3 Huffman Info Decoding

The **Huffman Info Decoding** block can be conceived as a controller for the **Huffman Decoding** block. The block serves to setup all the parameters necessary for the **Huffman Decoding** block to perform a correct Huffman decoding.

The first task to perform is collecting data in the side information describing the characteristics of the Huffman code bits. This includes where to find the Huffman code bits in the bitstream, to decide which Huffman tables are used and whether ESCAPE values are present in the Huffman code bits. Moreover, the **Huffman Info Decoder** block must insure that 576 frequency lines are generated regardless of how many frequency lines are described in the Huffman code bits. When fewer than 576 frequency lines appear the **Huffman Info Decoding** block must initiate a zero padding to compensate for the lack of data.

2.2.4 Scalefactor Decoding

The purpose of the **Scalefactor Decoding** block is to decode the coded scalefactors, i.e. the first part of the main data. Input to the **Scalefactors Decoding** block is scalefactor info and coded scalefactors. The output of the block is the decoded scalefactors, to be used in the **Descaling** block.

Figure 2.8 shows the organization of the scalefactors within each frame. It is shown how a frame is divided into two granules and for each granule scalefactors for the left and right channel respectively are presented. The side information contains information on how the scalefactors were coded. The side information for each frame is basically organized in the same way. The only difference is that the side information contains information common to each channel in each granule. This information must be considered when decoding the scalefactors.

The scalefactors transmitted, each applies for one scalefactor band. The number of scalefactor bands and hence the number of transmitted scalefactors depends on the type of window used in the MDCT in the encoder. The maximum number of scale factors sent for a channel within a granule is 36. This situation occurs when short windows are used prior to the MDCT. The actual number of scalefactors for a channel must be determined prior to the decoding process. In order to decode the scalefactors the side information must be used. The side information

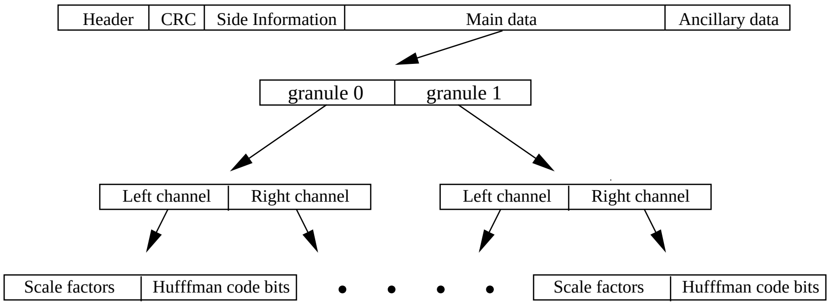


Figure 2.8: *Organization of the scalefactors in granules and channels. A dual channel frame is used for this illustration*

contains information about how the scalefactors are compressed. Two different compression schemes can be applied and the side information is used to select the one to use for a particular scalefactor band. In Layer III the number of scalefactors transmitted can vary between frames. The situation of less scalefactors occurs when scalefactors, belonging to a channel in the first granule, are reused for the corresponding channel in the second granule.

2.2.5 Descaling

The purpose of this block is to reestablish a perceptually identical copy of the frequency lines generated by the MDCT block in the encoder. The descaling is based on the scaled quantized frequency lines is_i reconstructed by the **Huffman Decoder** and the scalefactors reconstructed in **Scalefactor Decoding**. The calculation of the frequency lines belonging to one channel within a granule is shown in Equation (2.1) (for long windows only).

$$
\begin{aligned}
xr_i \;=\;& \mathrm{sign}(is_i) \cdot \mathrm{abs}(is_i)^{\frac{4}{3}} \\
&\cdot \frac{2^{\frac{1}{4}(\texttt{global\_gain}[gr][ch]-210)}}{2^{(scalefac\_multiplier \cdot (\texttt{scalefac\_l}[gr][ch][sfb]+\texttt{preflag}[gr][ch] \cdot pretab[sfb]))}}
\end{aligned}
\tag{2.1}
$$

For short windows Equation (2.2) must be applied in order to calculate the frequency lines for one channel within a granule.

$$
\begin{aligned}
xr_i \;=\;& \mathrm{sign}(is_i) \cdot \mathrm{abs}(is_i)^{\frac{4}{3}} \\
&\cdot \frac{2^{\frac{1}{4}(\texttt{global\_gain}[gr][ch]-210-8 \cdot \texttt{subblock\_gain}[gr][ch][sfb][window])}}{2^{(scalefac\_multiplier \cdot \texttt{scalefac\_s}[gr][ch][sfb][window])}}
\end{aligned}
\tag{2.2}
$$

In both Equation (2.1) and Equation (2.2) the quantized frequency lines is_i are first raised to the power of $\frac{4}{3}$, which is the inverse power of that used in the non-uniform quantizer in the encoder. Afterwards the sign of is_i is applied.

The `global_gain` variable, used in both Equation (2.2) and Equation (2.1), defines the quantization step size used for one channel within a granule. The number 210 is a system constant which ensures an appropriate scaling of the output [ISO/IEC 11172-3, 1993]. The scalefactors are logarithmically quantized in the encoder with a step size of either 2 or $\sqrt{2}$ which can be identified by the value of the `scalefac_scale` flag. If `scalefac_scale`=0 then *scalefac_multiplier*=0.5 else *scalefac_multiplier*=1. The variables *pretab* and `preflag` are only used in the case that long windows were applied in the MDCT. A table in [ISO/IEC 11172-3, 1993] specifies a single value per scalefactor band for the *pretab* variable, which is used for preemphasis. The `preflag` variable is added to the scalefactors `scalefac_l` and is used for additional amplification of the higher frequencies. The scalefactors `scalefac_s` and `scalefac_l` correspond to the scalefactors decoded in the **Decoding of Scalefactors** block. When using short windows in Equation (2.2), a gain factor for each of the short windows is specified by `subblock_gain` in the side information. `subblock_gain` is given for each channel in each granule, see Appendix B.

2.2.6 Reordering

The frequency lines generated by the **Descaling** block are not always ordered in the same way. In the **MDCT** block the use of long windows prior to the transformation, would generate frequency lines ordered first by subband and then by frequency. Using short windows instead, would generate frequency lines ordered first by subband, then by window and at last by frequency. In order to increase the efficiency of the Huffman coding the frequency lines for the short windows case were reordered into subbands first, then frequency and at last by window. The purpose of the **Reordering** block is to detect whether short windows has been applied to any of the 36 subbands, and if – then reorder the values. Whether a reordering is necessary can be tested on the `window_switching_flag` and `block_type` variable in the side information block. An algorithm for reordering of a subband is shown in Algorithm 2.1.

Algorithm 2.1 *Algorithm for reordering of the frequency lines in a subband. n denotes the subband and is in the interval $\{0, 1, 2, \ldots, 31\}$. F_d denotes the disordered frequency lines and F_o the ordered frequency lines.*

$Reorder(n)$
for $window = 0$ to 2 **do**
  for $index = 0$ to 5 **do**
    $F_o(n * 18 + 6 * window + index) = F_d(n * 18 + window + 3 * index)$
  end for
end for

2.2.7 Stereo Processing

Besides supporting single and dual channel mode, the MPEG/Audio Layer III algorithm also supports two modes of stereo redundancy coding: middle/side stereo coding (MS-stereo) and intensity stereo coding. These modes exploits the redundancy inherently present in most stereo audio signals. The purpose of the **Stereo Processing** block is to perform the necessary processing to convert the encoded stereo signal into separate left/right stereo signals.

In MS stereo mode the left/right channels are transmitted as the sum of the two channels and the difference between the two channels, called middle (M) and side (S), respectively. The MS stereo model is suitable for stereo transmissions because the two channels in an uncoded stereo signal are highly correlated most of the time. Also, it should be noted that MS stereo performs lossless stereo encoding.

The middle signal will contain much information and the side channel will contain a much smaller amount of information. The stereo signal can therefore be compressed more efficiently compared to transmitting the two stereo channels independently of each other. In the decoder the left/right channels can be reconstructed using Equation (2.3) where i is the frequency line index.

$$L_i = \frac{M_i + S_i}{\sqrt{2}} \quad \text{and} \quad R_i = \frac{M_i - S_i}{\sqrt{2}} \tag{2.3}$$

In intensity stereo mode the upper frequency subbands are encoded into a single summed signal with corresponding intensity positions `is_pos` for the scalefactor bands encoded. In this mode the stereo information is contained within the intensity positions, because only a single channel is transmitted. Obviously, stereo inconsistencies will appear for this model, because audio restricted to one channel will be present in both channels. The inconsistencies will not be conceivable by the human ear if they are kept small.

The intensity position is coded for each intensity stereo encoded scalefactor band. As a result the decoded left/right channels have same spectral shape within each scalefactor band, but differ in magnitude. In the decoder the left/right channels can be reconstructed by using Equation (2.4) where *sfb* denotes scalefactor band.

$$is\_ratio_{sfb} = \tan\left(\texttt{is\_pos}_{sfb} \cdot \frac{\pi}{12}\right) \tag{2.4}$$

$$L_i = L_i' \cdot \frac{is\_ratio_{sfb}}{1 + is\_ratio_{sfb}} \quad \text{and} \quad R_i = L_i' \cdot \frac{1}{1 + is\_ratio_{sfb}},$$

The `is_pos` values are placed in the scalefactor region of the right channel. Which upper frequency subbands are encoded with intensity stereo is controlled by the zero partition appearing during the Huffman encoding, see Subsection 4.3. Figure 2.9 illustrates the four different modes for encoding stereo signals, each with an increasing complexity.

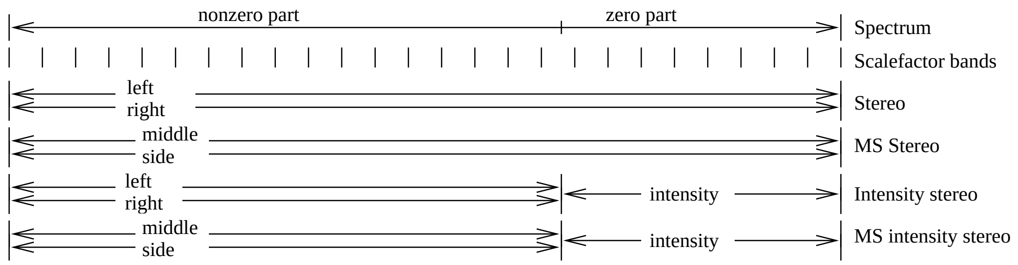


Figure 2.9: *Different modes for encoding stereo signals. These modes are controlled by* `mode` *and* `mode_extension`.

The method used for encoding the stereo signal can be read from the `mode` and `mode_extension` in the header of each frame. A flow chart of the stereo processing is illustrated in Figure 2.10.

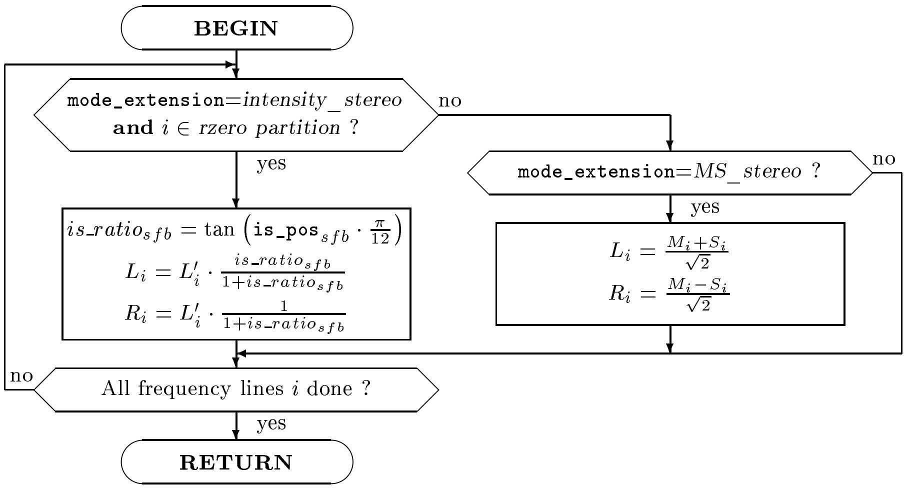


Figure 2.10: *Flow chart of the Stereo Processing of the frequency lines.* $i \in \{0, 1, \ldots, 575\}$ *denotes the frequency line number, and sfb denotes the scalefactor band.*

2.2.8 Alias Reconstruction

In the **MDCT** block within the encoder it was described that an alias reduction was applied. In order to obtain a correct reconstruction of the audio signal in the algorithms to come, the aliasing artifacts must be added to the signal again. The alias reconstruction calculation consists of eight butterfly calculations for each subband, as illustrated in Figure 2.11

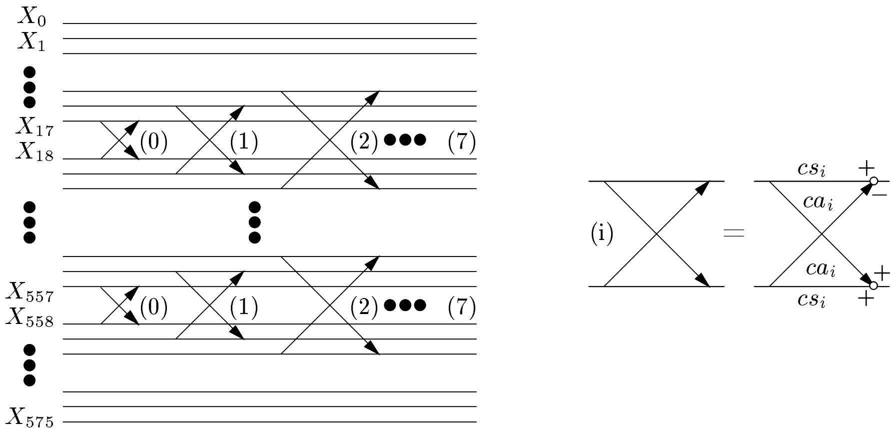


Figure 2.11: *Layer III alias reduction butterflies. The butterflies are performed before the IMDCT transformation in the decoder. The cs_i and ca_i constants are tabulated in [ISO/IEC 11172-3, 1993].*

2.2.9 Inverse Modified Discrete Cosine Transformation

The frequency lines from the **Alias Reduction** block are mapped to polyphase filter subband samples in the **IMDCT** block. The frequency lines are denoted X_k and the polyphase filter samples S_i (This S_i should not be confused with the S_i used during the Stereo Processing.). The analytical expression of the IMDCT is shown in Equation (2.5), where n is 12 for short blocks, and 36 for long blocks.

$$x_i = \sum_{k=0}^{\frac{n}{2}-1} X_k \cos\left(\frac{\pi}{2n}\left(2i+1+n/2\right)\left(2k+1\right)\right), \quad \text{for } i = 0 \text{ to } n-1 \tag{2.5}$$

Depending on the `block_type` value, x_i is multiplied with one of the following windows.

`block_type`= 0 (normal block)

$$w_i = \sin\left(\frac{\pi}{36}\left(i+\frac{1}{2}\right)\right), \quad \text{for } i = 0 \text{ to } 35 \tag{2.6}$$

`block_type`= 1 (start block)

$$w_i = \begin{cases} \sin\left(\frac{\pi}{36}\left(i+\frac{1}{2}\right)\right) & \text{for } i = 0 \text{ to } 17 \\ 1 & \text{for } i = 18 \text{ to } 23 \\ \sin\left(\frac{\pi}{12}\left(i-18+\frac{1}{2}\right)\right) & \text{for } i = 24 \text{ to } 29 \\ 0 & \text{for } i = 30 \text{ to } 35 \end{cases} \qquad (2$$

`block_type`= 3 (stop block)

$$w_i = \begin{cases} 0 & \text{for } i = 0 \text{ to } 5 \\ \sin\left(\frac{\pi}{12}\left(i-6+\frac{1}{2}\right)\right) & \text{for } i = 6 \text{ to } 11 \\ 1 & \text{for } i = 12 \text{ to } 17 \\ \sin\left(\frac{\pi}{36}\left(i+\frac{1}{2}\right)\right) & \text{for } i = 18 \text{ to } 35 \end{cases} \qquad (2.$$

`block_type`= 2 (short block)

$$w_i = \sin\left(\frac{\pi}{12}\left(i+\frac{1}{2}\right)\right) \quad \text{for } i = 0 \text{ to } 11 \tag{2.9}$$

When `block_type!=2` the result x_i is obtained with Equation (2.10).

$$z_i = x_i w_i \tag{2.10}$$

When `block_type=2` the window function Equation (2.9) is multiplied to each of the three blocks in the subband, and these windowed short blocks are then overlapped and concatenated as follows

$$y_i^{(j)} = x_i^{(j)} \sin\left(\frac{\pi}{12}\left(i+\frac{1}{2}\right)\right) \quad \text{for } i = 0 \text{ to } 11, \text{ for } j = 0 \text{ to } 2 \tag{2.11}$$

$$z_i = \begin{cases} 0 & \text{for } i = 0 \text{ to } 5 \\ y_{i-6}^{(1)} & \text{for } i = 6 \text{ to } 11 \\ y_{i-6}^{(1)} + y_{i-12}^{(2)} & \text{for } i = 12 \text{ to } 17 \\ y_{i-12}^{(2)} + y_{i-18}^{(3)} & \text{for } i = 18 \text{ to } 23 \\ y_{i-18}^{(3)} & \text{for } i = 24 \text{ to } 29 \\ 0 & \text{for } i = 30 \text{ to } 35 \end{cases}.$$

The result of these calculations is a vector z_i for each subband. These vectors are then overlapped with half a window and added, resulting in 32 new subband vectors S_i.

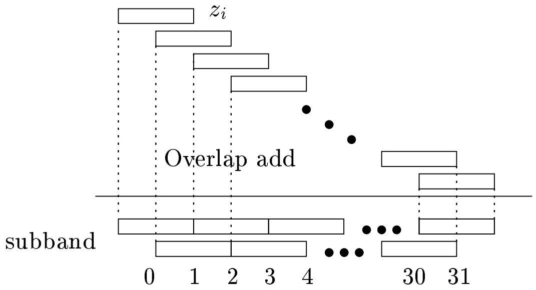


Figure 2.12: *Overlap add of the result of the IMDCT.*

2.2.10 Frequency Inversion

In order to compensate for frequency inversions in the synthesis polyphase filter bank, every odd time sample of every odd subband is multiplied with -1, when denoting the subband $n \in \{0, 1, \ldots, 31\}$ and the time samples $i \in \{0, 1, \ldots, 575\}$.

2.2.11 Synthesis Polyphase Filter Bank

Each time a sample, from each of the 32 subbands, S_i has been calculated, they are applied to the synthesis polyphase filter bank and 32 consecutive audio samples are calculated. Algorithm 2.2 shows an algorithm for computation of the polyphase filter bank. The N variable can be calculated using Equation (2.12).

$$N_{ik} = \cos\left((16+i)(2k+1)\frac{\pi}{64}\right). \tag{2.12}$$

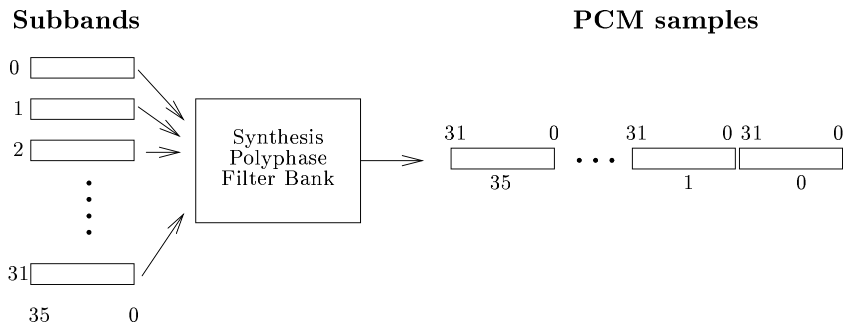


Figure 2.13: *The synthesis filter transform a sample from each subband into 32 consecutive PCM samples.*

Algorithm 2.2 *Algorithm for calculation of the synthesis polyphase filter bank*

for $i = 1023$ **downto** 64 **do** //Shifting
  $V[i] = V[i-64]$
end for
for $i = 0$ **to** 63 **do** //Matrixing
  $V[i] = \sum_{k=0}^{31} N_{ik} S_k$
end for
for $i = 0$ **to** 7 **do** //Build a 512 values vector U
  for $j = 0$ **to** 31 **do**
    $U[64i + j] = V[128i + j]$
    $U[64i + 32 + j] = V[128i + 96 + j]$
  end for
end for
for $i = 0$ **to** 511 **do** //Window by 512 coefficients
  $W[i] = U[i]D[i]$
end for
for $j = 0$ **to** 31 **do** //Calculate 32 PCM samples
  $p[i] = \sum_{i=0}^{15} W[j + 32i]$
end for

3

Design Specification

In Chapter 2 the algorithms needed in order to decode an MPEG/Audio Layer III bitstream were described. From this description a subset of algorithms is chosen to constitute the Bitstream Part, i.e. the algorithms to be executed on the BSP. When the algorithms to be performed are defined, the interfaces to the BSP will be specified. Having defined the interfaces, the demands to the BSP will be provided. This includes demands on both area and time as well as demands on functionality. The section ends with specifying how the design could be tested and evaluated.

3.1 Definition of the Bitstream Part

In Chapter 2 each block in the MPEG/Audio Layer III algorithm was introduced. From the introduction it appears that differences in the operation types required in the individual blocks were present.

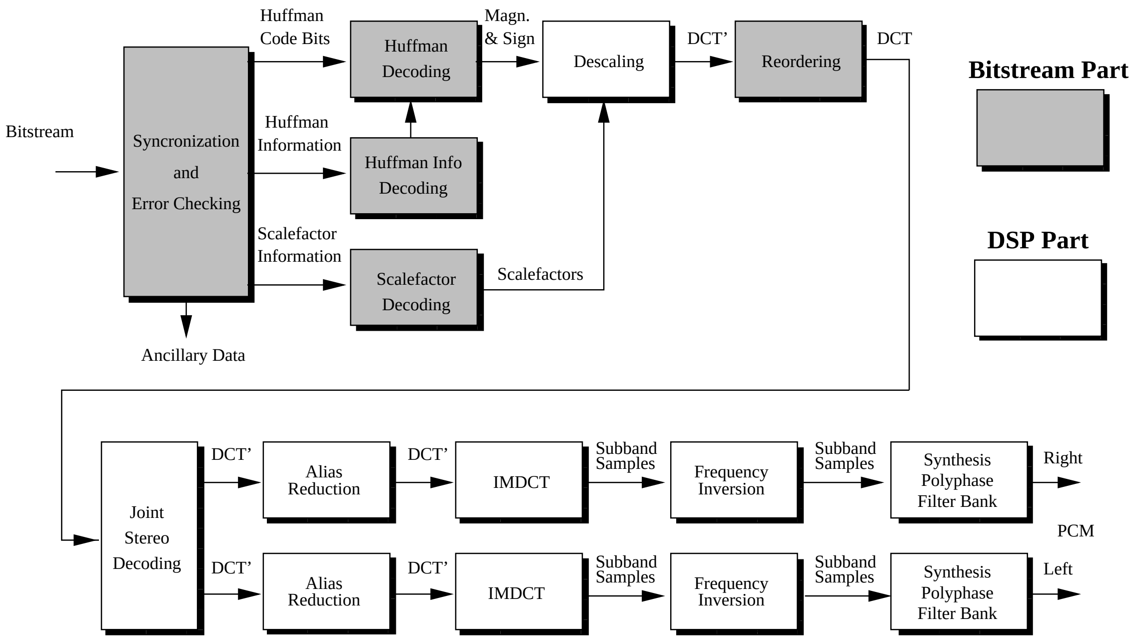


Figure 3.1: *Block diagram of MPEG/Audio decoder. The shaded boxes in the figure are executed on a BSP and the remaining blocks are executed in a DSP.*

The algorithms to be performed in the **Synchronization and Error Checking**, **Huffman Decoding**, **Huffman Info Decoding**, **Scalefactor Decoding** and **Reordering** blocks in the decoder cannot be characterized as typical DSP algorithms. The algorithms to be performed in these blocks are very data dependent and only involve simple arithmetic operations, as e.g. address calculations. The **Descaling**, **Stereo Processing**, **Alias Reduction**, **Invers Modified Discrete Cosine Transformation**, **Frequency Inversion** and the **Synthesis Polyphase**

Filter Bank blocks represent typical DSP algorithms including multiply accumulate operations, linear memory access and data independent loops. These blocks can relatively easy be implemented on most general purpose DSP's. The distinct characteristics of the different algorithms included in the decoder, motivates a grouping of algorithms into the two parts shown in Figure 3.1. The shaded boxes in Figure 3.1 represent algorithms that could be executed on a BSP. The Bitstream Processor must be optimized toward handling the selected blocks efficiently. The BSP should then assist a general purpose DSP in performing the algorithms needed in order to decode a Layer III audio bitstream. Assembling the algorithms highlighted with shaded boxes in Figure 3.1 to a single unit, provides the exact definition of the Bitstream Part, as illustrated in Figure 3.2. Compared to Figure 3.1 a minor change has been made. The **Reordering** block has been moved forward. The movement is a consequence of the design strategy chosen. The design strategy is to lower the computational demands to the DSP. Consequently as many of the computations required in the Layer III algorithm as possible, should be performed on the BSP. The task performed in the **Reordering** is basically to load the descaled values from memory and store them on reordered addresses, which involve operation such as address calculations and data movements. The reordered addresses cannot be generated by the data address generator (DAG) provided in most DSP's. This is the reason for moving the **Reordering** block forward. By moving the **Reordering** block forward some of the data dependent computations in the **Descaling** block must be carried out by the BSP instead and hence they are removed from the DSP part of the algorithms.

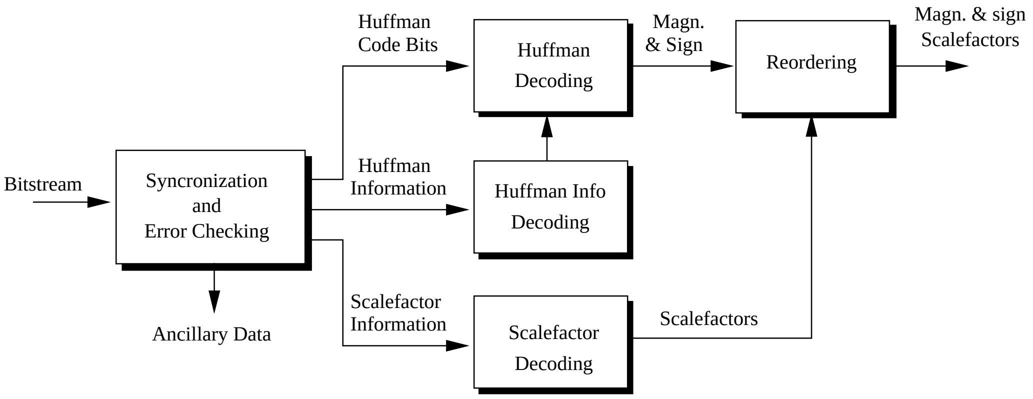


Figure 3.2: *Definition of the Bitstream Part of MPEG/Audio Layer III decoder algorithm.*

The demands to the DSP after the **Reodering** has been moved forward is calculated in Appendix A. The result is a computational demand on 11 MIPS. This requirement can be fulfilled by e.g. an ADSP 210x, which is considered as a inexpensive DSP.

3.2 System Interfaces

Having defined the algorithms to be performed on the BSP the interfaces to other parts must be defined. Figure 3.3 shows the interconnections of the BSP with the remaining parts of the decoder.

System Decoder: The purpose of the System Decoder is to demultiplex the MPEG system bitstream into an audio and a video part and to select the desired source if more than one is present. The input to the BSP must always be a single audio bitstream. The System Decoder is out of the scope for this project and will therefore considered further more.

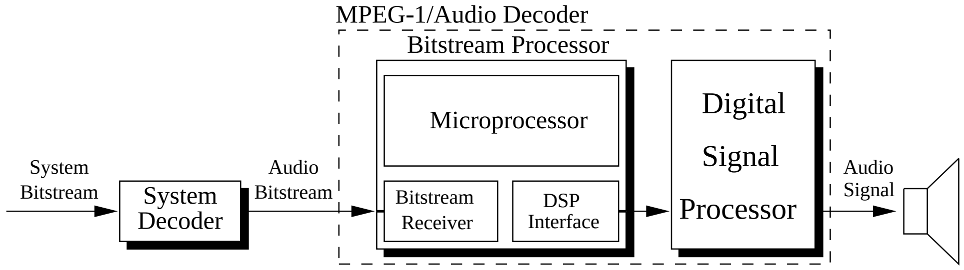


Figure 3.3: *The system interfaces to the BSP.*

Bitstream Receiver: An interface between a System Decoder and the BSP must be considered. This interface is referred to as a bitstream Receiver. The bitstream receiver must handle the synchronization of the incoming audio bitstream. In order to perform the synchronization some buffering must be present in the receiver. An appropriate solution capable of reading a bitstream and storing in a data buffer will therefore be required. The amount of RAM and the specific configuration appropriate will be left for further investigation.

DSP Interface: An interface between the BSP and the DSP is necessary to enable transmission of the decoded bitstream. At this point the actual implementation is not considered, but only the demands. For the highest sampling frequency allowed i.e. $48kHz$ the DSP interface must be able to transmit the decoded data required in order to reconstruct 96000 samples.

3.3 Design Objectives for the BSP

In the project, emphasis will be put on design and software implementation of the BSP. The purpose of the BSP is to execute the algorithms belonging to the Bitstream Part, defined in Figure 3.2. The BSP to be designed and implemented is defined in Figure 3.3. As indicated in Figure 3.3 the Bitstream Receiver and the DSP interface is included in the design.

Layer: In order to designate the designed decoder an ISO/IEC MPEG/Audio Layer III decoder, it must support all three layers, i.e. Layer I, II and III. In the design process only Layer III will be considered, meaning that only software for the Layer III algorithm will be written. Software supporting Layer I and II can be written later on to make the decoder ISO/IEC compatible, if desired. The Layer III decoder will be designed to support all the features specified in the Layer III part in [ISO/IEC 11172-3, 1993].

Real–Time Demands: The BSP must be designed to operate in real time. In order to fulfill this demand at all times, the BSP must be capable of processing 320 kbit/s and passing on data, required to reconstruct 96.000 samples per second.

Area Consumption: To fulfill the objectives of designing a low cost Layer III decoder, the BSP must be designed toward low area consumption.

Target Technology: The project considers the design and test phase of the Layer III BSP. The designed BSP will be described in VHDL, and the VHDL description is required to be fully synthesizeable. To be able to provide time and area estimates, a target technology must be chosen. In this project area and time estimates will be given using standard cells from ES2.

Test: When the design of the BSP has been completed, the description in VHDL can be initiated. The test and validation of the processor is divided into three stages. First, the functionality of each functional block must be validated. At this stage the complexity is low, which allows for an in-depth test. The next stage is an iterative process, where the functional blocks, one at the time, are added to the architecture chosen for the BSP. Having assembled all the functional blocks constituting the BSP, the final functionality test can be initiated. A testing method providing absolute comfort would be to excite the BSP with a set of test vectors that covers all possible states and combinations of states the BSP can attain. The test vectors are unfortunately very difficult to define because of the high complexity inherently present in a microprocessor. An appropriate alternative is to write the Layer III program, perform the bitstream decoding on a large variety of bitstreams and obtain a reliable reference for validation. The confidence in this test method depends on the selection of bitstreams. As both a reliable reference and a very large set of test bitstreams have been obtained, the test and validation of the BSP will be conducted using the latter method. Besides a functional test, it must be verified that the timing requirements are met.

Design Evaluation: When the functional test has been successfully performed and the timing requirements are met, the overall cost of the design must be evaluated. This is done by using a performance function. The performance function represents an evaluation of the measurable parameters in the design. The overall performance of a given design in an ASIC can be expressed in terms of the cost function, shown in Equation (3.1).

$$C = f(N, P, A, T) \tag{3.1}$$

In Equation (3.1) N represents "noise", i.e., a reflection of the accuracy of the algorithm, P is the power consumption, A represents the area consumed by the ASIC and T is the time factor. When using the parameters in the cost function for a measurement of performance it should be noted that the primary goal is to fulfill the demands derived from the application. Minimization of the cost function is a secondary task. Having a real–time demand for the application makes the time factor less interesting and hence the processor utilization will be used as a measure instead. The utilization measure is defined in Equation (3.2).

$$U = \frac{T_{comp}}{T_{comp} + T_{idle}} \tag{3.2}$$

In Equation (3.2), T_{comp} is the time the processor spends on computation and T_{Idle} is the period in which the processor is idle.

When designing the BSP the primary goal, after the functionality has been validated, is to fulfill the real–time demands. In order to determine how expensive the BSP would be in a commercial implementation an area estimate is necessary. Therefore an area estimate for an implementation of the BSP in standard cells from ES2 will be conducted. Also, it will be estimated which general purpose DSP is appropriate for the DSP part of the decoder.

Besides the directly measurable parameters described above, a set of subjective parameters can be used to evaluate the "goodness" of the design. Three appropriate parameters to be considered are listed below.

- Flexibility
- Re-programmability
- Expandability

The flexibility parameter indicates the ease, with which the design can be applied to a different application.

The re-programmability is a design parameter that affects the cost of performing a reprogramming of the software used in the processor. In this project the BSP would be designated as re-programmable, if the processor facilitates the programming of all three MPEG/Audio decoding algorithms, i.e. Layer I, II and III.

Adding new features to the BSP may result in a completely new design. In the case that the processor was design with re-usability in mind, the adding of new features would not require a complete redesign of the processor, but only few minor changes. In the latter case the design can be characterized as expandable.

Contrary to parameters such as time and area, the parameters described above are not directly measurable. Thus the evaluation of these parameters will be performed in terms of a discussion.

4

Computation Types Required in the BSP

To optain an efficient architecture in the Bitstream Processor, it is crucial to know the types of computations required to perform the parts of the Layer III decoder designated for the Bitstream Processor. The purpose of this chapter is to analyse the required types of computaions and the frequency of these. To be able to conduct such an analysis it is necessary to make a more thorough description of the blocks to be executed in the Bitstream Processor. This is done in the following.

4.1 Synchronization and Error Checking

The **Synchronization and Error Checking** block receives the incoming bitstream and identifies the start location of each frame. When the start of a frame is known, it is possible for the succeeding blocks to extract all the information contained in the frame, by using the specifications in the MPEG/Audio standard. The start of a frame is found by searching for a synchronization word, "`1111 1111 1111`". However, even though a synchronization word is found in the bitstream, it does not guarantee that a start of a frame is found. The reason for this is that Layer III does not guarantee that the synchronization word will not appear in the main data area or in the ancillary data area. To improve the stability of the synchronization some extra checks are performed before accepting a synchronization word as valid.

The number of checks needed to retain an error free audio signal, depends on the error probability in the bitstream. If the error probability is high, a trade-off between assigning processing power for validity checks and accepting distortions in the audio signal, must be made. For most applications it can be argued that rarely occurring distortions are acceptable.

Most of the bits in the header are held constant from the startup of the decoder, e.g. sampling frequency and layer number. The MPEG/Audio standard states that changes in these values may require a reset of the decoder, hence they can be used to check the validity of the header during decoding. From the information in the header it can be tested whether a 16-bit CRC check word is appended after the header. If this is the case, the validity of the the side information and the last 16 bits of the header can be checked. These checks will of cause also improve the stability of the system if the bitstream is error prone, caused by faulty transmission.

The simplest method of synchronization is a continuously search for synchronization words. The downside of this method is that it might try to synchronize on a spurious synchronization word, even though it appears in a frame, which already has been accepted. A method to enhance the stability of the synchronization, is to skip the contents of the current frame, if it is proven valid, before starting to search for the next synchronization word. This is possible because the length of a frame can be calculated from the information in the header. An example of the difference between these methods can be seen in Figure 4.1. Search method (a), in Figure 4.1 continues the search immediately after the synchronization word has been found, thus it tries to synchronize to the spurious synchronization word. Search method (b) does not try to synchronize on the spurious synchronization word, because it reside in a frame which is already verified, thus the contents of the frame is skipped.

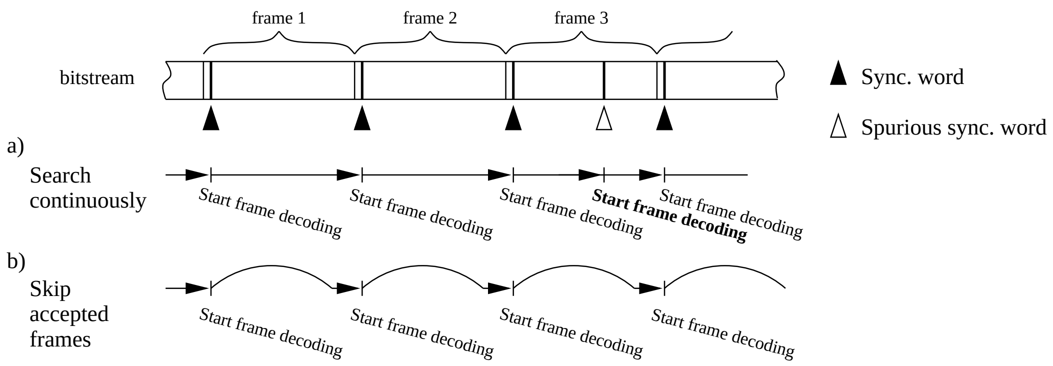


Figure 4.1: *Two methods for searching for the synchronization word in the bitstream. The first (a) searches continuously, and if a spurious synchronization word occurs in a frame, it will try to synchronize to it. The second method (b) skips the contents of valid frame, before starting to search for the next synchronization word.*

The length of a Layer III frame can be calculated with Equation (4.1).

$$\text{frame length} = 8 \left\lceil 144 \frac{\text{bitrate}}{\text{sampling frequency}} \right\rceil \; bits \tag{4.1}$$

When reaching a synchronization word, the position of the consecutive synchronization word can be calculated from the information in the header. The frame length is allowed to vary in Layer III and therefore it is necessary to calculate the position of the next synchronization word in every frame. In one case the position of the synchronization words cannot be calculated, this is in the free bitrate mode. However, the free bitrate mode also implies fixed bitrate, hence the frame length is constant. If the frame length is constant it is possible to estimate the frame length as the most frequently distance between synchronization words.

A flow chart of the **Synchronization and Error Checking** can be seen of Figure 4.2. This flow chart describes the decoding of a single frame in the bitstream, and if it is possible to calculate the lenght of the frame, it skips the contents of the frame before returning.

4.1.1 Computational Types in Synchronization and Error Checking

Because of the bit reservoir technique employed in Layer III, it is necessary for the decoder to have a buffer where the incomming bitstream can be temporarily stored. The MPEG/Audio standard specifies the size of this buffer to be the length of a frame when using the highest possible bitrate (320 kbit/s) and a sampling frequency at 48 kHz. When using Equation (4.1) the frame length can be calculated to 7680 bits.

The computations required to perform the **Synchronization and Error Checking** is restricted, mainly because it is performed only once every frame. When searching for a synchronization word, the worst case scenario is that an entire frame has to be scanned before finding a synchronization word. This happens if the length of a frame can not be calculated because the bitrate is in free mode. The rest of the operations needed in this block is a few conditional branches based on comparisons between values in the bitstream and stored values, see Figure 4.2.

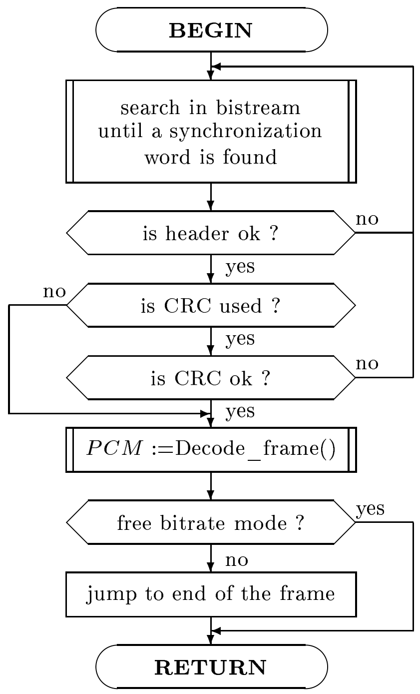


Figure 4.2: *Flow chart of the block for handling* **Synchronization and Error Checking**. *The Decode() function includes all the succeding blocks in the decoder, Figure 2.5.*

4.2 Huffman Decoding

In this section the Huffman coding scheme and conventions used in the MPEG/Audio Layer III bitstream will be presented. Also, the task of the Huffman decoder will be specified. Emphasis will be put on presenting the rather complex partitioning of the Huffman coded data and relevant data for decoding.

In the MPEG/Audio Layer III bitstream, Huffman coding is only applied to the quantized frequency lines. The Huffman coding is based on the assumption that frequency lines in the higher frequency regions have lower absolute values than frequency lines in the lower frequency regions, and that the absolute value of adjacent quantized frequency lines not differs significantly. In the following it will be explained how data are handled in the Huffman encoder to establish a foundation for which operations have to be processed in the Huffman decoder. Without any loss of generality, the explanation is conducted for only one channel within a granule.

In order to maximize the coding efficiency, the quantized frequency lines are Huffman coded in smaller partitions. First each granule is partitioned into three partitions as shown in Figure 4.3. Starting from right, i.e. the higher frequencies, adjacent pairs of quantized frequency lines equal to zero are counted. This number is named rzero. Next adjacent quadruples of quantized frequency lines with absolute value less than or equal to 1 are counted, and the number is named count1. The remaining quantized frequency lines are paired and the number of pairs is called big_values. For pairs of quantized frequency lines belonging to the rzero partition, no bits are transmitted because the contents of this partition is fully predictable. The quadruples from the count1 partition are coded with one code word using one of two Huffman tables. Values in the big_value partition are coded in pairs using one of thirty Huffman tables. The big_value partition is further divided into two or three regions and the values are coded using two or three separate Huffman tables. The boundaries for the division of the big_value partition and the Huffman tables used for coding is specified in the side information.

```
xxxxxxxxxxxxxxxx-----------------------00000000000000000000
|                |                      |                  |
1          big_values*2        big_values*2+count1*4      576
```

Figure 4.3: *Partitioning of a granule into three partitions each requiring different coding tables.*

Prior to the Huffman coding a formatting of the quantized frequency lines is applied. First, the absolute value for each frequency line is calculated. Secondly, all values exceeding 15 are represented with the number 15 and a remainder called an ESCAPE value. Depending on the size of the ESCAPE value a number of bits, called *linbits*, are assigned to represent the ESCAPE value, see Equation (4.3).

$$\text{ESCAPE value} = \text{Quantized frequency line - 15, for Quantized frequency line} \geq 15 \quad (4.2)$$

$$\mathit{linbits} = \text{Word length(ESCAPE value)} \quad (4.3)$$

The number of *linbits* required for a code word can be identified by the Huffman table used for coding. For nonzero values a sign bit applied. For quadruples, ESCAPE values are never required. Here only the code word for the formatted quadruple and corresponding sign bits are transmitted. An example of six Huffman coded frequency lines is illustrated in Figure 4.4. Here the transition from the big_values region into the count1 region is illustrated by the double line succeeding the first appearing `sign_y`.

	`Code word(x,y)`	`linbits_x`	`sign_x`	`sign_y`	`Code word(v,w,x,y)`	`sign_v`	`sign_y`	

Figure 4.4: *Extract of the Huffman code bits showing a pair (x,y) from the big_value partition where x is appended with linbits. Also a quadruple (v,w,x,y) from the count1 region is shown where w and x is equal to zero, which can be seen by the absence of sign bits for w and x.*

For a further illustration one could assume the following for the first code word in Figure 4.4:

`Code word(x,y)='0000110'`

`linbits_x='0011'`

`sign_x = '1'`

`sign_y = '1'`

Huffman table 24 applies.

Using this table implies that if x or y equals 15, four *linbits* are present. Decoding the code word using table look up reveiles that x equals 15 and y equals 9. The ESCAPE value for x equals 3 and the sign bits `sign_x` and `sign_y` indicates a minus which results in $x = -18$ and $y = -9$.

In the decoding process the Huffman decoder is supplied with control information from the **Huffman Info Decoding** block. The Huffman decoder needs to know whether the next code word belongs to the big_value partition or the count1 partition in order to produce the correct number of decoded frequency lines. Also the Huffman decoder must be supplied with information on which Huffman table to select, and how many *linbits* are present. The decoding process can be described by the flow chart in Figure 4.5.

The flow chart in Figure 4.5 is called once for every code word to be decoded or for performing a zero stuffing in the rzero partition. The latter is in effect a Huffman decoding using Huffman

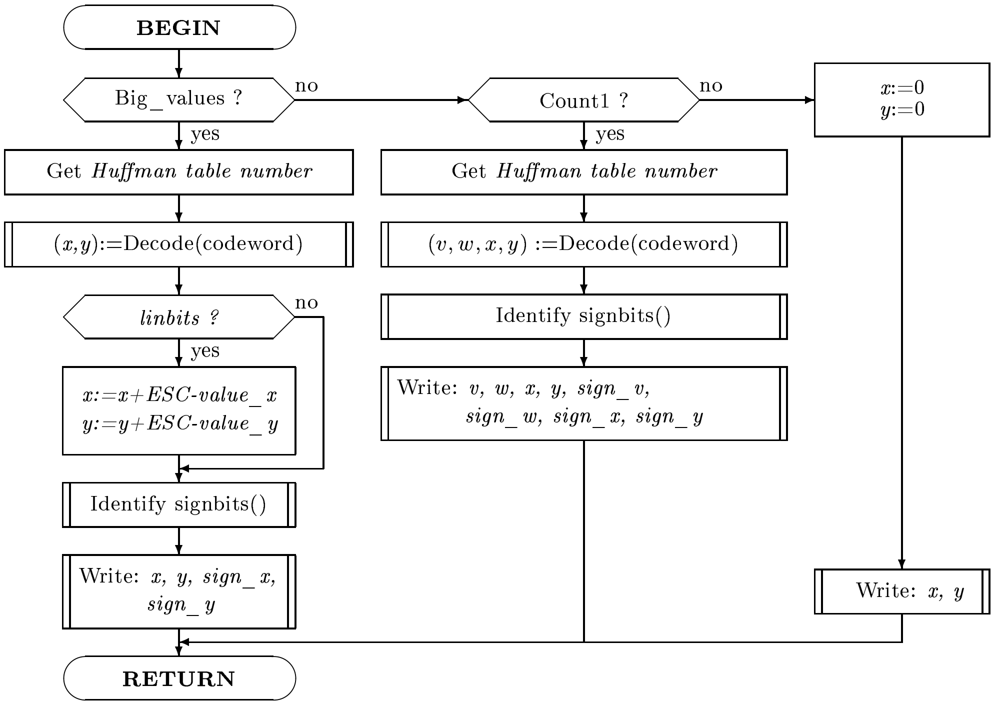


Figure 4.5: *Flow chart showing the steps and decisions to make in decoding the Huffman code bits.*

table 0, which indicates that two zeroes must be generated. In Figure 4.5 four subroutines are used. The subroutine Decode() is performing the actual Huffman decoding which in essence is an identification of a codeword and a table look up that will produce two or four values. The specific technique used for both codeword identification and table look up is optional for the system designer. The subroutine Identify sign bits() decides if sign bits exist for the two or four decoded frequency lines.

4.2.1 Computation Types in Huffman Decoding

In the following the computational types needed in the **Huffman Decoding** block will be presented. The estimate is restricted to apply for the worst case situation only, because the complexity of the **Huffman Decoder** is very data dependent. What is referred to as "worst case" should not be conceived as a provable worst case situation but rather as an estimated upper bound. Also, it should be noted that the worst case situation to be estimated only will appear very seldom in an application. However, the worst case estimate is performed because it is found to be the best guideline for designing the BSP. If the worst case estimate shows to present unreasonable processing demands a decrease in processing demands may be required. This, of course, should be performed while ensuring that the probability of lack in processing power is kept at a reasonable level. If a situation occurs, though very unlikely, where the BSP cannot keep up with the time demands for Huffman decoding, no damage requiring a restart or equally drastic events will be induced. The frame or sequence of frames causing the problems will simply be reproduced erroneously.

In Figure 4.5 the most complex path traverses the leftmost branch, because only here the Huffman codeword length can vary and only here *linbits* can be present. The branch contains two sub branches each requiring peak performance. In the first situation, no *linbits* are present which will leave most bits for Huffman decoding. In the second case, all Huffman codewords are appended with two *linbits*, one *linbit* per quantized frequency line, which will be the case leading to most frequently handling of ESCAPE values.

When decoding a bitstream containing a stereo signal sampled at 48 kHz and transmitted at 320 kbit/s, the maximum amount of data for decoding will be present. In the two worst case situations to present, all Huffman codewords belong to the big_value partition. This will lead to the worst case situation because the maximum number of Huffman decodings in the leftmost branch will occur.

When transmitting 320 kbit/s, the bitreservoir technique will never be taken into use which will bound main data belonging to one frame to reside within the main data area of that frame [ISO/IEC 11172-3, 1993]. Therefore, the amount of Huffman code bits to be decoded per second can be calculated straightforward.

Every second $41\frac{2}{3}$ frames are transmitted, when the sampling frequency is set to 48 kHz. For every frame 32 bits are used for the header, 256 bits for the side information and approximately 500 bits are used for the coded scalefactors. In Equation (4.4) the number of bits per second to be decoded into frequency lines is calculated.

$$\#\text{ Huffman code bits} = 320\ kbit/s - 41\frac{2}{3}(32\ bits + 256\ bits + 500\ bits) = 287,167\ bits \quad (4.4)$$

When assuming that all frequency lines belong to the big_value partition approximately all frequency lines in the big_value partition will be appended with a sign bit. In the first worst case situation where no *linbits* are present, 96.000 sign bits will be present. This will leave 191.167 bits for Huffman decoding, resulting in 48.000 Huffman codewords with a mean length of 4.0 bits. In the second situation where *linbits* always are present, the Huffman codeword length is 8 bits in worst case, which can be identified in the Huffman tables in [ISO/IEC 11172-3, 1993]. Data to reconstruct two frequency lines will therefore consist of 8 Huffman code bits, 2 *linbits* and 2 sign bits, resulting in 23.931 Huffman codewords. Obviously, Huffman codewords are lacking in order to reconstruct 96.000 frequency lines. Therefore the remainder must belong to the rzero partition. The latter will require zero stuffing for reconstruction. Common to both worst case situations is that the two values x and y and their corresponding sign bits must be written into memory after decoding. For this purpose a generic operation Mem() is defined. All the operations necessary to perform the two worst case situations are summarized in Table 4.1.

	Situation 1	Situation 2
# ESCAPE value handling	0	48,000
# Huffman Decodings, codeword length = 8	0	24,000
# Huffman Decodings, mean codeword length = 4.0	48,000	0
# Read (Sign)	96,000	48,000
# Zero stuff (x, y)	0	24,000
# Mem (x_sign, x)	48,000	48,000
# Mem (y_sign, y)	48,000	48,000

Table 4.1: Summary of all the operations to perform in the two worst case situations

In Table 4.1 the operation count is performed for general operations that most likely resides at a higher abstraction level than the operations available in the bitstream unit to implement. At this instance, the table does therefore not tell which situation actually is the worst case situation, because a translation of the two situations operation count into comparable operations is depending on the bitstream unit.

4.3 Huffman Info Decoding

In this section all the operations to be executed in the **Huffman Info Decoding** block will be presented. Without any loss of generality the explanation to be conducted will apply for only one channel within a granule.

Huffman Info Decoding supplies the **Huffman Decoding** block with the information necessary to decode the incoming Huffman code bits. In the following it will be explained how the decoding information can be extracted from the side information, and how to select the correct Huffman table, granule partition and regions in the big_value partition.

In the side information the value `part2_3_length` informs on the length of the main data, i.e scalefactors and Huffman code bits. In the **Scalefactor Decoding** block the value *part2_ length*, i.e. the number of bits used to represent the scalefactors, is calculated. Based on this value and `part2_3_length` the number of Huffman code bits, called *part3_ length*, can be calculated, see Equation (4.5) .

$$\mathit{part3\_length} = \texttt{part2\_3\_length} - \mathit{part2\_length} \tag{4.5}$$

The side information provides the number of codewords, that is pairs of frequency lines, denoted `big_values`, belonging to the big_value partition of the granule. Knowing this value and *part3_ length* it can be decided on run time when the Huffman code bits belonging to the count1 partition starts and ends. The interpretation of the two parameters is: When big_value codewords have been decoded, the remaining code bits belong to the count1 partition.

The last parameters to be determined are the boundaries for the two or three regions within the big_value partition. The region boundaries given by `region0_count` and `region1_count` are given in terms of scalefactor bands which can be used as index into a number of tables given in [ISO/IEC 11172-3, 1993]. The table look up will provide the number of frequency lines within the particular region.

If the parameter `window_switching_flag` is set, predetermined region boundaries are set depending on the value of `block_type` and `mixed_block_flag`. If `block_type` = 2 and `mixed_block_flag` = 0 then region0_count is set to 8 and and region1_count is set to 36 by the decoder. Else region0_count and region1_count are set to 7 and 36 respectively. By performing a table look up in the appropriate tables it turns out that region0_count for the predetermined case always will be 36 frequency lines. The region1_count value of 36 scalefactor bands simply states that the remaining codewords within the big_value partition belong to region1. For the predetermined region boundaries case two Huffman table indexes `table_select [0]` and `table_select [1]` for region0 and region1 respectively are given in the side information.

If `window_switching_flag` is not set, meaning only normal windows are used, the two region boundaries `region0_count` and `region1_count` given in the side information and the sampling frequency are used as index into one of three tables given in [ISO/IEC 11172-3, 1993]. Thereby the number of frequency lines within region0 and region1 can be determined. The remaining codewords within `big_value` codewords will belong to region2. In this case three Huffman table indexes for the three regions are given in the side information. The algorithm for determining both region boundaries and Huffman tables for the whole granule is illustrated in Figure 4.6.

The **Huffman Info Decoding** block provides control information for the block **Huffman Decoding**, but the block can also be viewed as a controller. In Figure 4.7 a flow chart illustrates how the block **Huffman Info Decoding** calls the block **Huffman Decoding**. Here the granule partition, the regions in the big_value partition and the Huffman tables are determined, and the block Huffman Decoding(), presented in previous section, is called like any subroutine. This process continues while codewords for the big_value partition and the count1 partition are present. The last subroutine in Figure 4.7, Determine Next Partition Determine Next Region(), is basically a counter keeping track of what granule partition and what region in the big_value

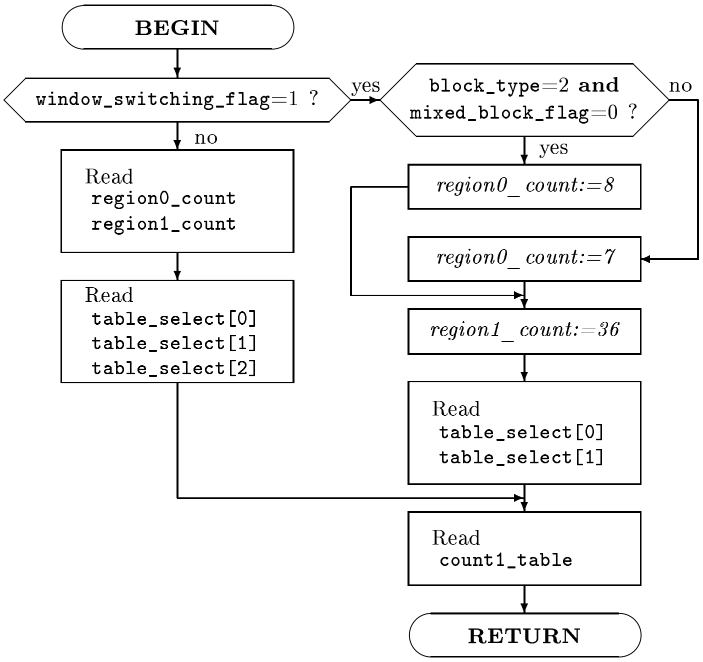


Figure 4.6: *Flow chart showing how the regions in the big_ value partition are found and which Huffman tables applies.*

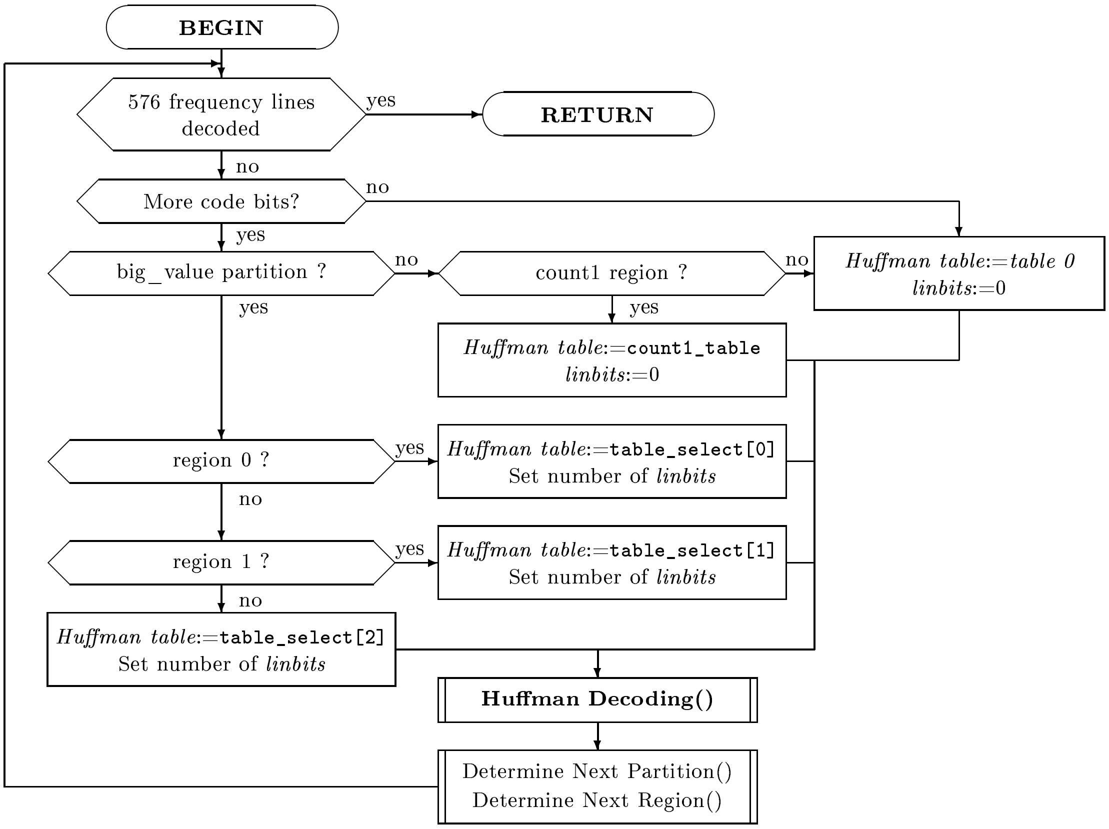


Figure 4.7: *Flow chart showing the decision of valid parameters and the call to the block* **Huffman decode**.

partition the next codeword to be decoded is placed in. It should be noted that the flow diagram can terminate not having read all code bits. This situation is valid in case stuffing bits are present in the bitstream, and is handled like illustrated because *part3_ length* includes stuffing bits.

4.3.1 Computation types in Huffman Info Decoding

All the tasks to perform in **Huffman Info Decoding** are summarized in Figure 4.6 and Figure 4.7. While Figure 4.6 is a linear process to be executed once per channel per granule and Figure 4.7 is one big loop to traverse 288 times in worst case per channel per granule, it can be presumed that the complexity of **Huffman Info Decoding** can be estimated by the tasks in Figure 4.7 only.

Traversing Figure 4.7 once will in worst case require three decisions, one call to **Huffman Decoding** and two arithmetic operations to perform Determine Next Partition() and Determine Next Region(). In order to produce 576 frequency lines this set of operations must be performed 288 times per granule in the worst case situation. In Table 4.2 an estimate for the number of operations per second required to perform **Huffman Info Decoding** is presented.

# Decisions	144.000
# Call	48.000
# Arithmetic operations	96.000

Table 4.2: Estimate of all the operations to perform in **Huffman Info Decoding** per second

4.4 Scalefactor Decoding

The purpose of the **Scalefactor Decoding** block is to decode the coded scalefactors, i.e. the first part of the main data. Input to the **Scalefactors Decoding** block is Scalefactor info and coded scalefactors, as shown in Figure 4.8. The output of the block is the scalefactors `scalefac_s` and `scalefac_l` which are processed in the **Reordering** block.

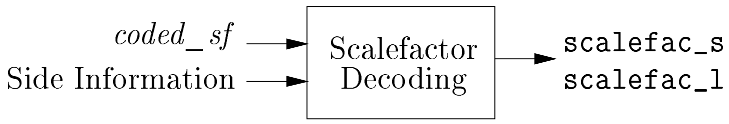


Figure 4.8: *The input and output of the* ***Scalefactor Decoding*** *block. The meaning of* `scalefac_s` *and* `scalefac_l` *will be explained later.*

The section is organized as follows: First the main flow containing all the necessary calls to subalgorithms will be presented. Next, all the subalgorithms will be described in turn.

In Figure 4.9 the main flow of the **Scalefactor Decoding** block is shown. First the two variables *slen1* and *slen2*, which will be explained later, are determined. Afterwards the total number of bits used on coded scalefactors, *part2_ length*, is calculated. This number is calculated for each channel in each granule. Then the scalefactors belonging to each channel in each granule are decoded by one of three algorithms. In the case of short windows decode_ scalefac_s() is used and for long windows decode_calefac_l() is used. In the case of mixed windows the algorithm decode_scalefac_n&s() is used.

The number of coded scalefactors transmitted can vary within each channel in a granule. The actual number of coded scalefactors depends on which window type is used in the MDCT.

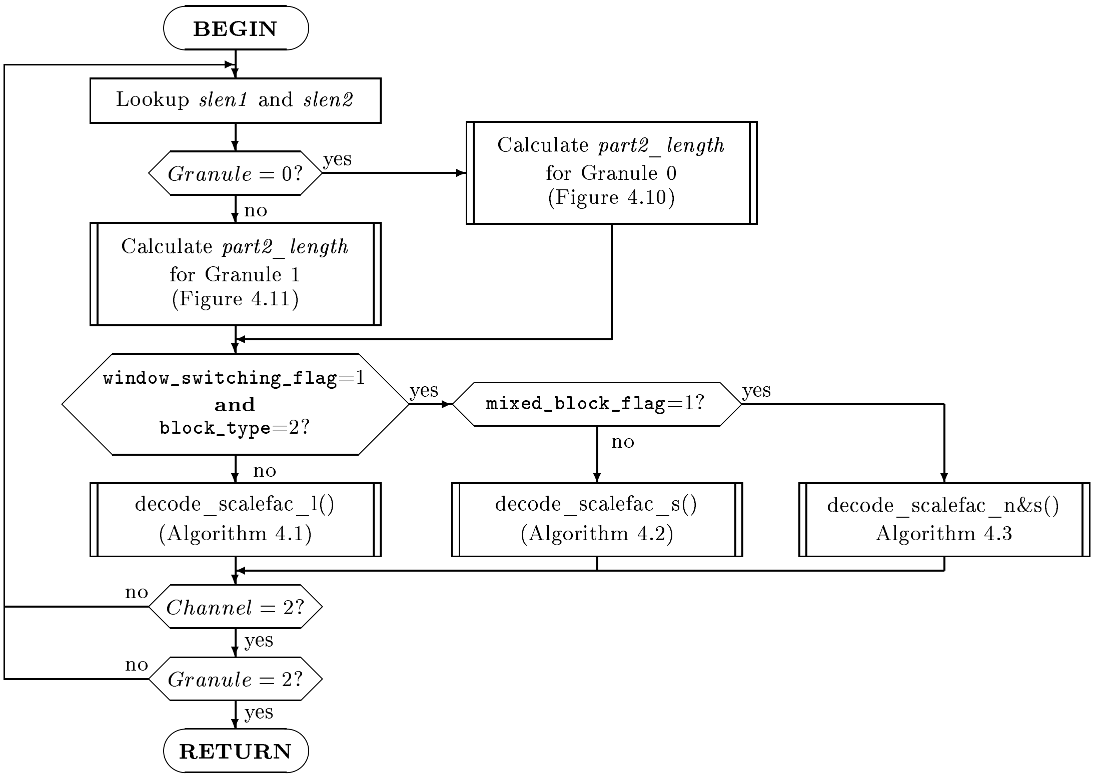


Figure 4.9: *The main flow of the **Scalefactor Decoding** block.*

Each channel in a granule is divided into 12 or 21 scalefactor bands for short and long windows respectively. These scalefactor bands are again arranged into two groups of scalefactor bands depending on the value of `block_type`, i.e. 0-10 and 11-20 in the case of long windows and 0-5 and 6-11 for short windows. The *slen1* and *slen2* values represent the number of bits used to encode each of the groups of scalefactor bands.

The first job in the process of decoding the scalefactors is to determine *slen1* and *slen2*. This is done using `scalefac_compress`, which is a 4 bit variable to be used as an index into a table specified in [ISO/IEC 11172-3, 1993], which returns the two desired variables *slen1* and *slen2*. In order to control the decoding algorithm, the total number of bits used to code the scalefactors for each channel in each granule, i.e. *part2_length*, must be calculated. The calculation of *part2_length* for the first granule is shown in Figure 4.10.

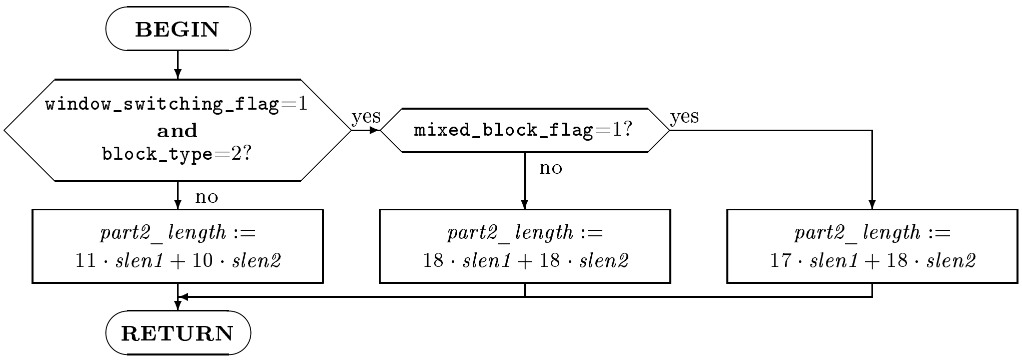


Figure 4.10: *Calculation of part2_length for the first granule, i.e. granule* $= 0$.

When calculating *part2_length* for the second granule, side information common to each channel in both granules must be considered. The `scfsi` variable influences the number of bits used for scalefactors for the second granule in a frame. When using other than short type windows the `scfsi` variable specifies whether the scalefactors sent for the first granule also apply to the second granule, see Appendix B for further explanation. Figure 4.11 shows the calculation of *part2_length* for the second granule.

When intensity stereo mode is enabled the stereo positions *is_pos* are transmitted for the via the scalefactors of the right channel is coded. One *is_pos* value is transmitted per scalefactor band.

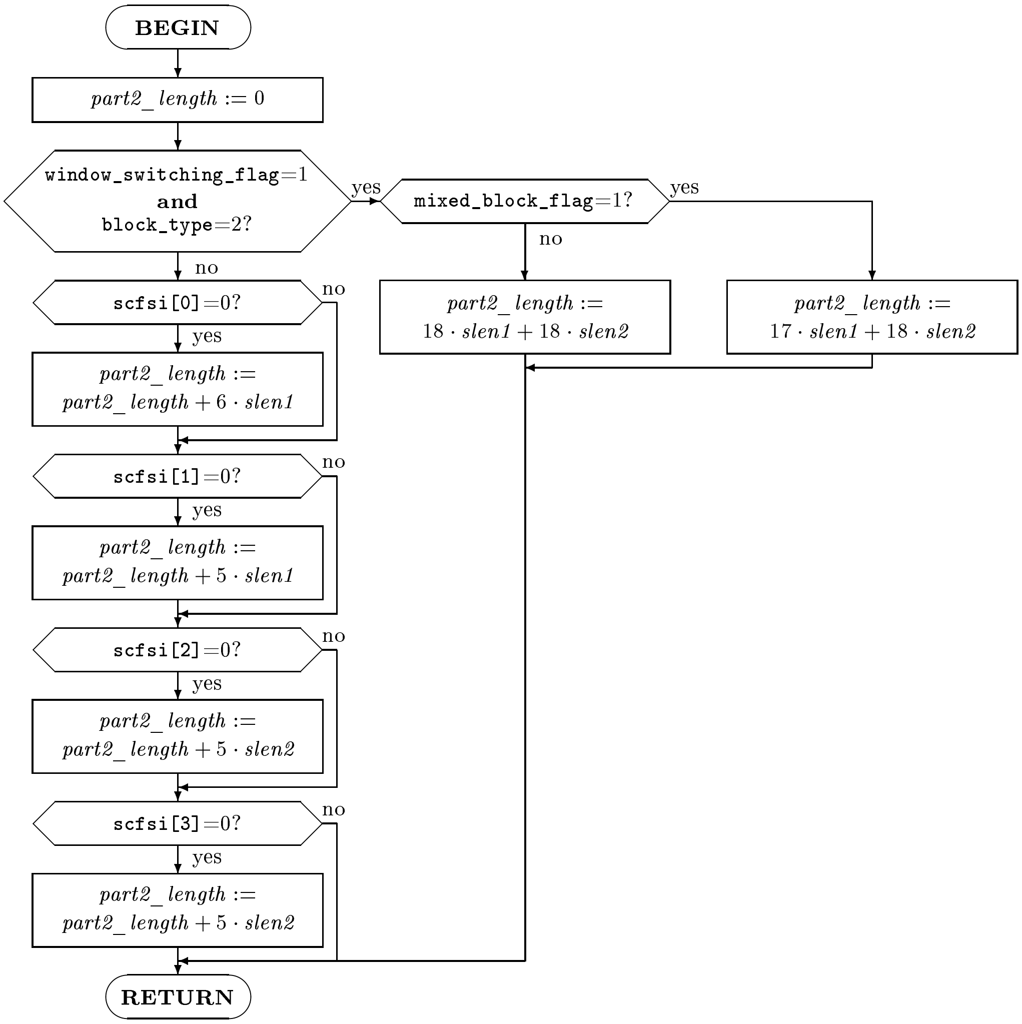


Figure 4.11: *Calculation of part2_length for the second granule, i.e. granule* = 1.

Algorithm 4.1 shows how the coded scalefactors are decoded when using long windows. The algorithm is separated into four independent parts (one **for** loop for each part). When using long windows the frequency spectrum is divided into 21 scalefactor bands. These 21 scalefactor bands are then arranged into groups of scalefactor bands. Four groups are constructed with the following ranges in scalefactor bands 0-5, 6-10, 11-15 and 16-20. The first two groups are coded with *slen1* bits per scalefactor band and the two last groups are coded using *slen2* bits per scalefactor band. When decoding scalefactors for the first granule all four parts are executed, but for the second granule the side information common to both granules must be considered. When using long windows Layer III enables the reuse of scalefactors for both granules. For this purpose a four bit variable `scfsi` is placed in the side information. The value of the `scfsi` variable determines whether the scalefactors sent for granule 0 are to be reused for granule 1. Each of the four bits in the variable `scfsi` facilitates the reuse of scalefactors for one of the four groups of scalefactor bands. For instance if `scfsi[0]=0` it means that, for scalefactors included in group 1, scalefactors are transmitted for each granule. Otherwise the scalefactors used in the first granule also applies to the second granule and hence no scalefactors are transmitted for the second granule, for the first group of scalefactors.

The bodies of the **for** loops in Algorithm 4.1 must be interpreted in the following way: *scalefac_l*[gr][ch][sfb] indicates a scalefactor either *slen1* or *slen2* bits long, belonging to scalefactor band number *sfb*, granule number *gr* and and channel number *ch*. The *l* in *scalefac_l* indicates that long windows are used.

Algorithm 4.1 decode_scalefac_l(). *Algorithm for decoding of scalefactors for long window types. coded_sf represent the coded scalefactors for both left and right channel.*

```
index := 0   //Initialization of pointer used to index the coded scalefactors coded_sf
for sfb = 0 to 20 do   //For all scalefactor bands
  if (sfb < 6) then   //For first group of scalefactor bands
    if (gr = 0) or (scfsi[ch][0] = 0) then   //First granule or scalefactors not reused
      scalefac_l[gr][ch][sfb] := coded_sf[index : (index + slen1 − 1)]   //Get slen1 bits
      index := index + slen1   //Increase index to point to the succeeding slen1 bits
    else
      scalefac_l[gr][ch][sfb] := coded_sf[gr − 1][ch][sfb]   //Copy scalefactors from granule 0
    end if
  else if (sfb < 11) then   //For second group of scalefactor bands
    if (gr = 0) or (scfsi[ch][1] = 0) then   //First granule or scalefactors not reused
      scalefac_l[gr][ch][sfb] := coded_sf[index : (index + slen1 − 1)]   //Get slen1 bits
      index := index + slen1   //Increase index to point to the succeeding slen1 bits
    else
      scalefac_l[gr][ch][sfb] := coded_sf[gr − 1][ch][sfb]   //Copy scalefactors from granule 0
    end if
  else if (sfb < 16) then   //For third group of scalefactor bands
    if (gr = 0) or (scfsi[ch][2] = 0) then   //First granule or scalefactors not reused
      scalefac_l[gr][ch][sfb] := coded_sf[index : (index + slen2 − 1)]   //Get slen2 bits
      index := index + slen2   //Increase index to point to the succeeding slen2 bits
    else   //For fourth group of scalefactor bands
      scalefac_l[gr][ch][sfb] := coded_sf[gr − 1][ch][sfb]   //Copy scalefactors from granule 0
    end if
  else
    if (gr = 0) or (scfsi[ch][3] = 0) then   //First granule or scalefactors not reused
      scalefac_l[gr][ch][sfb] := coded_sf[index : (index + slen2 − 1)]   //Get slen2 bits
      index := index + slen2   //Increase index to point to the succeeding slen2 bits
    else
      scalefac_l[gr][ch][sfb] := coded_sf[gr − 1][ch][sfb]   //Copy scalefactors from granule 0
    end if
  end if
end for
```

Algorithm 4.2 shows the algorithm used in the case of short window types. In this case 36 scalefactors are sent. For short windows the number of scalefactor bands equals 21 and they are grouped in two groups: 0-5 and 6-11. To each scalefactor band three short windows are applied, and one scalefactor per window per scalefactor band is transmitted. Again scalefactors belonging to the first group are coded with *slen1* bits per window and scalefactors in the second group are coded using *slen2* bits per window.

Algorithm 4.2 decode_scalefac_s(). *Algorithm for decoding of scalefactors for short window types. coded_sf represent the coded scalefactors for both left and right channel.*

```
index := 0   //Initialization of pointer used to index the coded scalefactors coded_sf
for sfb = 0 to 36 do   //For all scalefactor bands
  if (sfb < 18) then   //For first group of scalefactor bands
    scalefac_s[gr][ch][sfb] := coded_sf[index : (index + slen1 − 1)]
    index := index + slen1  //Increase index to point to the succeeding slen1 bits
  else   //For second group of scalefactor bands
    scalefac_s[gr][ch][sfb] := coded_scalefac[index : (index + slen2 − 1)]
    index := index + slen2  //Increase index to point to the succeeding slen2 bits
  end if
end for
```

Algorithm 4.3 is the algorithm for decoding scalefactors using mixed window types. In this case 12 scalefactor bands are applied. Two types of scalefactor bands exist, short and long scalefactor bands. Each of the types is to be used with the corresponding type of window. Consequently when changing the window type, the type of scalefactor band changes as well. This must be considered when decoding the scalefactors. The frequency ranges of the different scalefactor bands in long and short window mode depend on the sampling frequency and are specified in tables in [ISO/IEC 11172-3, 1993]. The scalefactor bands 0-7 (long window scalefactor band) and scalefactor band 3-5 (short window scalefactor band) are coded with *slen1* bits. The scalefactor bands 6-11 (short window scalefactor band) are coded with *slen2* bits.

Algorithm 4.3 decode_scalefac_n&s(). *Algorithm for decoding of scalefactors for mixed window types. coded_sf represent the coded scalefactors for both left and right channel*

```
index := 0   //Initialization of pointer used to index the coded scalefactors coded_sf
for sfb = 0 to 35 do   //For all scalefactor bands
  if (sfb < 8) then   //For 8 long scalefactor bands
    scalefac_l[gr][ch][sfb] := coded_sf[index : (index + slen1 − 1)]
    index := index + slen1  //Increase index to point to the succeeding slen1 bits
  else if (sfb < 18) then   //For first group of short scalefactor bands
    scalefac_s[gr][ch][sfb] := coded_sf[index : (index + slen1 − 1)]
    index := index + slen1  //Increase index to point to the succeeding slen1 bits
  else
    scalefac_s[gr][ch][sfb] := coded_sf[index : (index + slen2 − 1)]
    index := index + slen2  //Increase index to point to the succeeding slen2 bits
  end if
end for
```

Computation types in the Scalefactor Decoding Block

In this section the computation types needed in the **Scalefactor Decoding** block will be presented. The operation types needed in to decode the scalefactors are control operations, arithmetic operations and memory operations such as load and store. To control the flow of the decoding algorithm according to Figure 4.9 operations to decision making such as conditional jumps and calls. These operation can also be used to implement **for** and **if** loops, requuired in the all the algorithms included in decoding the scalefactors.

4.5 Reordering

The motivation for performing the reordering prior to the descaling is to remove the data dependencies from the algorithms included in the DSP part. Normally the operations performed in the descaling block depend on the type of window used and by scalefactor bands. How these dependencies arise will shortly be described. The dependency on window type is present because the window type must be determined prior to the descaling. This is necessary in order to choose between Equation (2.1) and Equation (2.2). Having selected either Equation (2.1) or Equation (2.2) the dependency on scalefactor band arise. The descaling is performed by applying the scalefactor and other values to the quantized frequency lines within each scalefactor band. The frequency index of the different scalefactor bands are tabulated in [ISO/IEC 11172-3, 1993]. When the number of frequency lines in a given scalefactor band have been rescaled, the number of frequency lines within the succeeding scalefactor band must be looked up. The scalefactor band indexes again depend on the window type and sampling frequency.

By moving the reordering block these dependencies are removed, meaning that the only variable required the frequency index, as it will be shown in the end of this section. Moving the **Reordering** block requires some preprocessing of the data dependent variables used in Equation (2.1) and Equation (2.2), i.e. scalefactors, `subblock_gain`, `pretab` and the quantized frequency line $is[i]$.

Reordering of the Quantized Frequency Lines

The quantized frequency lines output from the **Huffman Decoder** are not always ordered by increasing frequency. When using long window types the quantized frequency lines are ordered by subband and by increasing frequency within each subband. Using the short types of windows the ordering of the quantized frequency lines is different. In this case the quantized frequency lines are ordered by frequency and then by window. The purpose of the reordering algorithm is, in the case of short windows, to reorder the quantized frequency lines by subband and then by frequency. The reordering is a rather simple operation which just consists of reordering of the in individual frequency lines within each subband, as shown in Figure 4.12. The reordering algorithm is shown in Algorithm 4.5

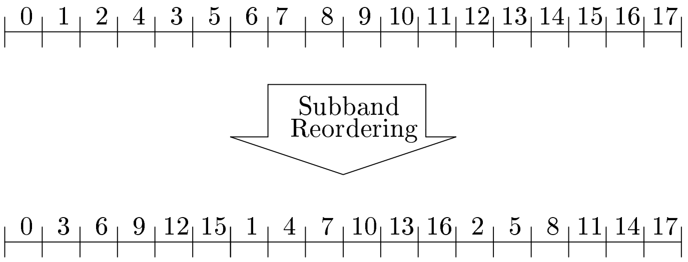


Figure 4.12: *Reordering of 18 frequency lines corresponding to a single subband*

Reordering of Variables Depending on Scalefactor Bands

A reordering of the quantized frequency lines requires a reordering of the scalefactors and other variables depending on scalefactor band such as `subblock_gain` and `pretab`. Prior to the reordering, the values of these variables for each of the frequency lines must be determined. In order to determine the variables depending on scalefactor bands for each frequency line, the indexes of the different scalefactor bands must be considered. The indexes of the different scalefactor bands depend on the sampling frequency *fs* and whether long or short types of window are used.

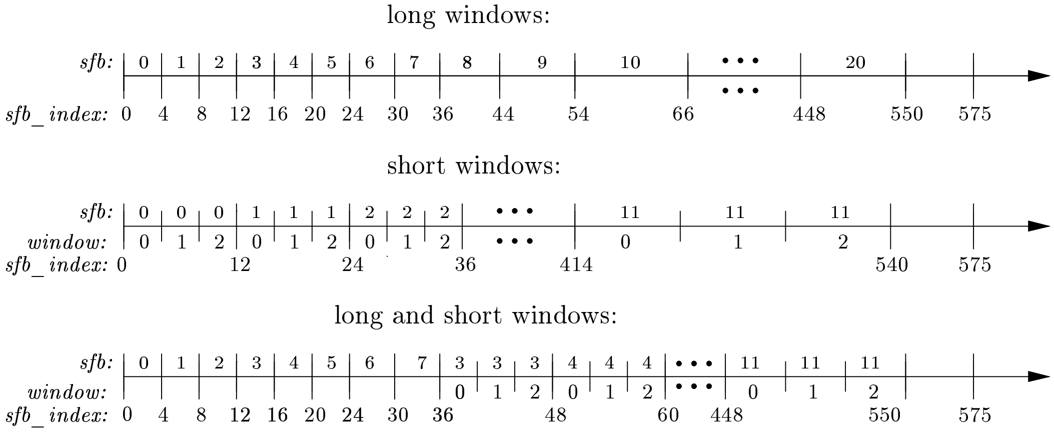


Figure 4.13: *The distribution of scalefactors, windows and scalefactor bands for fs=32 kHz. sfb_index indicates the starting index of each individual scalefactor band.*

The distribution of scalefactors throughout the frequency spectrum is explained in the following. When only long windows are used scalefactors are present for each of the 21 scalefactor bands. For short windows only, a single scalefactor for each of the three windows per scalefactor band is present. Using mixed window a single scalefactor is available for each of the first 8 long scalefactor bands. After the first 8 long scalefactor bands a switch between long and short types of windows occurs. Consecutively, a single scalefactor for each of the three windows per scalefactor band is present, when using short scalefactor bands. From now on following abbreviations for preprocessed scalefactors, `subblock_gain` and `pretab` respectively will be used: $sf[i]$, $sbg[i]$ and $pt[i]$. i is the frequency index ranging from 0 to 575. To determine the values of $sf[i]$, $sbg[i]$ and $pt[i]$. The values for a given scalefactor band, and window if present, must be copied to all frequencies within the particular scalefactor band. The number of frequency lines within a scalefactor band depends on the samplings frequency and the window type used. The number of scalefactors and stop index of each scalefactor band are specified in [ISO/IEC 11172-3, 1993] for both short and long window types and for the three possible sampling frequencies, i.e. 32, 44,1 and 48 kHz. Figure 4.13 shows the distribution of scalefactors throughout the frequency spectrum for fs=32 kHz.

The determination of $sf[i]$, $sbg[i]$ and $pt[i]$ is accomplished using Algorithm 4.4. Note that in Algorithm 4.4 an additional scalefactor band for each of the three mode exist. This last scale factor band covers the frequencies from the end of the last scalefactor band up to 575. The values of $sf[i]$, $sbg[i]$ and $pt[i$ should be set to zero for these frequencies, as in the last **for** loop in Algorithm 4.4

To realize what has been achieved by performing the **Reordering** block before the **Descaling**, a rewriting of Equation (2.2) and Equation (2.1) to Equation (4.6) is carried out.

$$xr[i] \quad = \quad \mathrm{sign}(is_{ro}[i]) \cdot \mathrm{abs}(is_{ro}[i])^{\frac{4}{3}} \cdot \frac{2^{\frac{1}{4}(\texttt{global\_gain}-210-8\cdot sbg_{ro}[i])}}{2^{(scalefac\_multiplier \cdot sf_{ro}[i] + \texttt{preflag} \cdot pt_{ro}[i])}} \tag{4.6}$$

From the rewriting shown in Equation (4.6) it can be seen that the only variable is the frequency index i. Compared to Equation (2.1) and Equation (2.2), the descaling can now be performed with a single equation and the dependencies on window type and scalefactor band are removed. Equation (4.6) must be applied to each channel in each granule.

Algorithm 4.4 *Algorithm for preprocessing the scalefactors,* `subblock_gain` *and* `pretab` *from the data belonging to a single channel.* $sf[i]$, $sbg[i]$ *and* $pt[i]$ *are the scalefactors,* `subblock_gain` *and* `pretab` *respectively for the i'th frequency line where* $i \in \{0, 1, 2, \ldots, 575\}$. $sfb\_index\_l[fs][sfb]$ *and* $sfb\_index\_s[fs][sfb]$ *correspond to the starting index of the long and short scalefactor band respectively using the sampling frequency fs and scalefactor band sfb.*

```
if windows_switching_flag=1 and block_type=2 then
  if mixed_block_flag=1 then  //For short and long windows
    for sfb=0 to 7 do  //For the first 8 long scalefactor bands
      for i=sfb_index_l[fs][sfb] to sfb_index_l[fs][sfb+1]-sfb_index_l[fs][sfb] do
        sf[i] = scalefac_l[sfb]  //Copy scalefactors
        sbg[i] = 0  //Gain for subblocks not used for long windows
        pt[i] = pretab[sfb]  //Copy pretab values
      end for
    end for
    for sfb=3 to 11 do  //For the last 9 short scalefactor bands
      for window=0 to 2 do  //For each window within the scalefactor band
        for i=sfb_index_s[fs][sfb] to sfb_index_s[fs][sfb+1]-sfb_index_s[fs][sfb] do
          sf[i] = scalefac_s[sfb + window]  //Copy scalefactors
          sbg[i] = subblock_gain[window]  //Copy gain for subblocks
          pt[i] = 0  //Pretab values not used for short windows
        end for
      end for
    end for
  else  //For only short windows
    for sfb=0 to 11 do  //For all 12 short scalefactor bands
      for window=0 to 2 do  //For each window within the scalefactor band
        for i=sfb_index_s[fs][sfb] to sfb_index_s[fs][sfb+1]-sfb_index_s[fs][sfb] do
          sf[i] = scalefac_s[sfb + window]  //Copy scalefactors
          sbg[i] = subblock_gain[window]  //Copy gain for subblocks
          pt[i] = 0  //Pretab values not used for short windows
        end for
      end for
    end for
  end if
else  //For only long windows
  for sfb=0 to 20 do  //For the all 21 long scalefactor bands
    for i=sfb_index_l[fs][sfb] to sfb_index_l[fs][sfb+1]-sfb_index_l[fs][sfb] do
      sf[i] = scalefac_l[sfb]  //Copy scalefactors
      sbg[i] = 0  //Gain for subblocks not used for long windows
      pt[i] = pretab[sfb]  //Copy pretab values
    end for
  end for
end if
for i'=i to 575 do  //Zeropad for the rest of the spectrum
  sf[i'] = 0  //Zeropad scalefactors
  sbg[i'] = 0  //Zeropad subblock gains
  pt[i'] = 0  //Zeropad pretab values
end for
```

Algorithm 4.5 *Algorithm for reordering 576 samples corresponding to the data belonging to a single channel. The samples are ordered into 32 subbands, each containing 18 subbands samples.* $X \in \{is, sf, sbg, pt\}$.

```
if windows_switching_flag and block_type then
  if mixed_block_flag=1 then  //For short and long windows
    for sb=0 to 1 do  //For the first 2 subbands
      for ss=0 to 17 do  //For all 18 subband samples
        X_ro[18·sb + ss] = X_dis[18·sb + ss]  //No reordering
      end for
    end for
    for sb=2 to 31 do  //For the last 30 subbands
      for window=0 to 2 do  //For each of the 3 windows
        for ws=0 to 5 do  //For each sample within a window
          X_ro[18·sb + ss] = X_dis[18·sb + window + 3·ws]  //Reorder
        end for
      end for
    end for
  else  //For only short windows
    for sb=0 to 31 do  //For all 32 subbands
      for window=0 to 2 do  //For each of the 3 windows
        for ws=0 to 5 do  //For each sample within a window
          X_ro[18·sb + ss] = X_dis[18·sb + window + 3·ws]  //Reorder
        end for
      end for
    end for
  end if
else  //For only long windows
  for sb=0 to 31 do  //For all 32 subbands
    for ss=0 to 17 do  //For all 18 subband samples
      X_ro[18·sb + ss] = X_dis[18·sb + ss]  //No reordering
    end for
  end for
end if
```

Computation Types Used in the Reordering Block

In the following, the computations types required in the **Reordering** block will be presented. In this section two algorithms were presented, Algorithm 4.4 and Algorithm 4.5. The preprocessing algorithm is simply the process of copying the values to the frequency lines within each window in each scalefactor band. When this has been performed, all 576 frequency lines for each of the three preprocessed variables $sf[i]$, $sbg[i]$ and $pt[i]$ must be stored in memory. This scheme is repeated for each channel in each granule. The reordering algorithm simply loads the input values from memory and saves them on "reordered" addresses. Hence the basic operations needed to perform these algorithms is the memory move operation, i.e. load and store operations, combined with some arithmetic operations. The **for** loops and the **if** sentences requires some control operations, such as jumps, in order to make decisions. Algorithm 4.5 can be performed by the MEM operation, since the reordering process basicly consists of moving a value from one place in memory to another. When using only short windows all the 576 frequency lines must be reordered. The reordering process is the same for the three preprocessed variables $sf[i]$, $sbg[i]$ and $pt[i]$ and for the quantized frequency lines $is[i]$. The reordering process must be executed for each channel in each granule. In the calculations it is assumed that the three variables $sf[i]$, $sbg[i]$ and $pt[i]$ are contained within a single word. This means that the preprocessing algorithm only has to perform one instead of three MEM operations per frequency line. In addition the amount of data needed to be transmitted to the DSP part is halved.

5

Algorithmic Solutions to Layer III

In this chapter specific solutions for Huffman decoding and CRC check is presented. The topics have to be considered more thoroughly because [ISO/IEC 11172-3, 1993] does not describe or recommend any particular algorithmic solutions for these topics.

5.1 Solutions to the Huffman Algorithm

For the Huffman algorithm presented, no specific solution to the decoding has been presented yet. In this section three solutions to the Huffman decoding will be presented. The solutions are direct table lookup, binary tree search, and clustered Huffman decoding.

When conducting a Huffman decoding, the basic task is to use the Huffman codeword as a guide in order to traverse the correct branches from the root down to an end node in a binary tree. The binary tree illustrated in Figure 5.1 will be used to present examples in the introduction of all three decoding methods.

Codeword	Symbol
1	Symbol 1
01	Symbol 2
001	Symbol 3
0001	Symbol 4
00001	Symbol 5
00000	Symbol 6

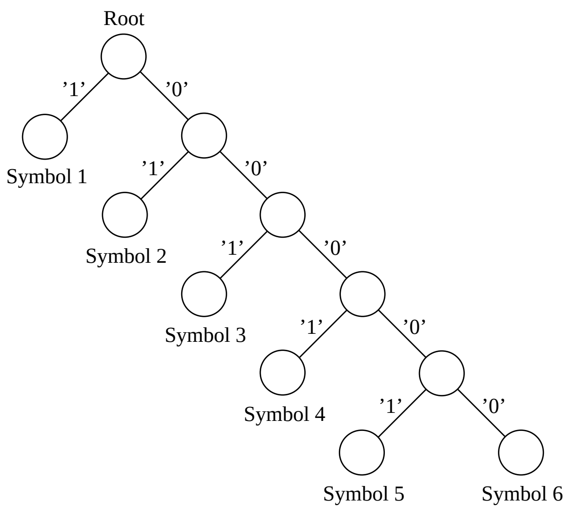


Figure 5.1: Illustration of the codeword table and the associated Huffman tree to be used for the three solutions for Huffman decoding

5.1.1 Direct Table Lookup

Direct table lookup is the fastest and most simple way to perform a Huffman decoding. The decoding technique can be implemented by using a ROM as decoder. When performing a decoding, a frame of the code stream is selected as an address to the ROM. The frame must be large enough to contain the longest codeword, and a smaller codeword must start at the most significant bit in the frame. At every entry in the ROM's address space, two fields must be present. The first field contains the decoded symbol, and the second field contains the length of the associated codeword. The second field is necessary in order to identify the length of the codeword. Having performed a decoding, the length field is used to decide where in the code stream the next code word starts.

When decoding a codeword smaller than the frame, the address to the ROM decoder will depend on the part of the frame not belonging to the codeword itself. For instance, a codeword one bit smaller than the frame length can point to two locations in the ROM, depending on the bit following the codeword. In this case it is necessary to have the associated symbol and the corresponding codeword length field placed at both addresses. A codeword two bit smaller can point to four locations and etc. A small example of a table lookup decoder is illustrated in Table 5.1.

Address	Symbol	Codeword length
00000	Symbol 6	5
00001	Symbol 5	5
00010	Symbol 4	4
00011	Symbol 4	4
00100	Symbol 3	3
⋮	Symbol 3	3
00111	Symbol 3	3
01000	Symbol 2	2
⋮	Symbol 2	2
01111	Symbol 2	2
10000	Symbol 1	1
⋮	Symbol 1	1
11111	Symbol 1	1

Table 5.1: The table shows how to organize the entries in the ROM based table lookup decoder.

The direct table lookup method is only suited for Huffman decoding when the codewords are short, because long codewords require a large memory space. For the Huffman tables used in Layer III, some tables contain codewords nineteen bits large. One table only will therefore require 512 kilobytes ROM. Because of the huge memory demands, the direct table lookup method is rejected.

5.1.2 Binary Tree Search

Binary tree search is almost the opposite of direct table lookup. Here a bitwise examination of the code stream is performed in order to determine when a codeword has been detected. When a codeword has been detected the associated symbol can be found by a simple table lookup.

Binary tree search can be implemented by use of a small lookup table. Starting with the first bit in a codeword, a table lookup is performed to detect if the codeword size is one bit. If a codeword is found, the table lookup must provide that information and also the associated symbol. If the codeword contains more bits, the table look up must provide that information and two addresses for the next table lookup. One address specifies the address if the next code bit is '1' and the other address specifies the next address for a '0'. A small example of a lookup table to be used with a binary tree search decoder is illustrated in Table 5.2. In the example, the first bit in the lookup table tells whether the table lookup provided a code word hit or a miss. If a codeword is found the associated symbol is provided. If not, two address offsets are provided for the cases next code bit is '1' or '0'. All Huffman decodings start at address '000'.

Address	Status	Symbol or offset	
0000	miss	#0001	#0010
0001	hit	Symbol 1	
0010	miss	#0001	#0010
0011	hit	Symbol 2	
0100	miss	#0001	#0010
0101	hit	Symbol 3	
0110	miss	#0001	#0010
0111	hit	Symbol 4	
1000	miss	#0001	#0010
1001	hit	Symbol 5	
1010	hit	Symbol 6	

Table 5.2: *An example of a binary search tree implementation. The table is an illustration of the memory layout for the lookup table. The numbers marked with # are address offsets to the next entry depending on the next bit in the codeword stream.*

An advantage in choosing the binary tree search algorithm is a small memory consumption compared to direct table lookup. An obvious disadvantage is that binary tree search is a slow process demanding many memory accesses to decode codewords of even modest size. Therefore the binary tree search is rejected.

5.2 Clustered Huffman Decoding

In literature clustered Huffman decoding algorithms can be found in slightly differing versions. The clustered Huffman decoding algorithm to be presented in the following is based on proposals by [Hashemian, 1995], [Hashemian, 1994] and [Hsieh and Kim, 1996].

Clustered Huffman decoding is a compromise between direct table lookup and binary tree search. The algorithm provides the benefits of fast Huffman decoding and modest memory consumption. Also, the algorithm is adjustable, and thereby gives the designer freedom to optimize the algorithm toward a specific application.

When initiating a nth order clustered Huffman decoding, an n-bit frame is read from the code stream. n can be chosen as wished, but must remain fixed. The n-bit frame value is used as an address offset into an initial lookup table. The entry read in the lookup table must provide information on whether a hit or a miss has occurred. In case of a hit, the table entry must contain the decoded symbol and information on the length p of the associated codeword. Based on this entry the decoded symbol can be read and the beginning of the next codeword in the code stream can be identified. If the first table lookup turns out to be a miss, an offset to the start of a new lookup table must be provided. Also, information on how many bits m are necessary to obtain a hit within the next table must be present. If a hit is not obtainable within the next table, m will attain the value n. The decoding proceeds by first discarding n bits from the code bits and then reading m bits. The following table lookup is performed in the table just advised using the value of the m bits as an offset. If the table lookup turns out to be a miss the decoding proceeds as described above. If a hit occurs the table entry must contain the decoded symbol and the number of bits p necessary to obtain the hit within the table only. Having read the decoded symbol, the decoding can proceed p bits from the last read frame. In Figure 5.2 the algorithm is illustrated.

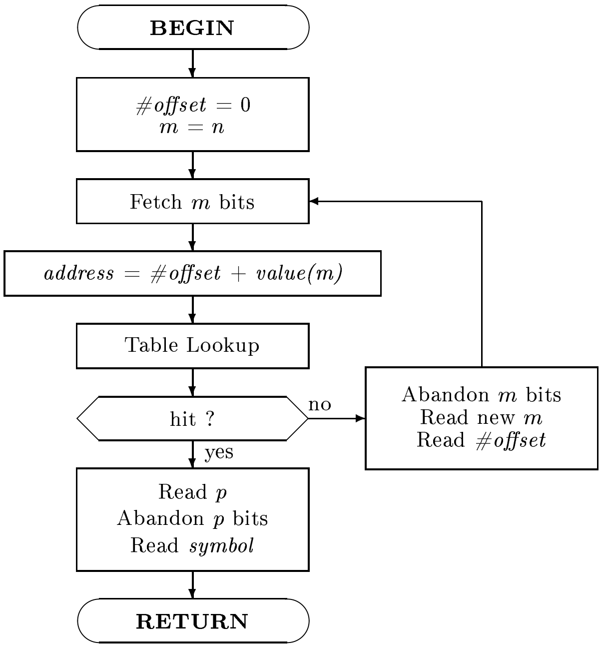


Figure 5.2: *The algorithm to traverse in order to conduct a decoding of one Huffman codeword.*

An example of a clustered Huffman decoding implementation using $n = 2$ is illustrated in Table 5.3. In the right table, i.e. the lookup tables, the double lines indicates the boundaries between the three tables present. The first column within each lookup table tells whether a codeword hit or miss has occurred. In case a hit has occurred, the table entries provide information on how many bits p out of m or n are necessary to decode the associated symbol within that particular table. In case of a miss, an offset address (marked with #) indicates where the next table appropriate starts relative to the initial tables start address, i.e. address 0000. Finally, a value indicating how many bits were necessary to perform the decoding within the particular lookup table is given.

For a further illustration the decoding of the codeword 00001 will be demonstrated. First, the two leftmost bits 00 is read. Performing a table lookup at the address 0000 + 00 reveiles that a miss has occurred, that the next lookup table starts at address 0000 + 0100, and that two bits from the code stream applies for the next table look up. The two bits just read from the code stream are rejected and replaced by the following two bits. From these bits a the next table lookup will be at address 0000 + 0100 + 00, that is address 0100. Again, it turns out to be a miss. From the table entry it is seen that the third table lookup is at address 0000 + 1000 and that only one bit applies for the third table lookup. Reading that last bit reveiles the bit 1. The table lookup is at address 0000 + 1000 + 1 and a hit is found. At the address Symbol 5 is read and it is also seen that only one bit was necessary to get the hit within that table.

The clustered Huffman decoding using a frame size of eight or four bits will apply very well for the Huffman decoding present in the Layer III algorithm. This assumption is based on the fact that a large percentage of the codewords are smaller than eight bit, and a decoding thereby can be accomplished by only one or two table lookup 's.

According to Table 4.1 the two worst case situations implies either 48,000 or 23,931 Huffman decodings with a mean codeword length of four or eight respectively. Choosing a frame size of eight will result in half the table lookups in the first of the worst case situations compared to choosing a frame size of four. It is worth to note that this does not imply the double performance, simply because processing a hit requires more post processing than a miss, i.e. setting sign bit. For the latter worst case situation the same number of table lookups is required, and no gain in

Address	Status	codeword size	Symbol or offset
0000	miss	2	#0100
0001	hit	2	Symbol 2
0010	hit	1	Symbol 1
0011	miss	1	Symbol 1
0100	miss	1	#1000
0101	hit	2	Symbol 4
0110	hit	1	Symbol 3
0111	hit	1	Symbol 3
1000	hit	1	Symbol 6
1001	hit	1	Symbol 5

Table 5.3: *An example of a clustered Huffman decoding implementation. The left table shows six symbols and their associated codewords, and the right table is an illustration of the memory layout for the three lookup tables necessary. The numbers marked with # are address offsets to the next lookup table to be applied.*

choosing a frame length of eight will be obtained here. Choosing a frame size of eight will also require approximately twice the memory space for lookup tables, compared to a solution using a frame length of four. Based on these performance comparisons a frame length of four seems to be the better choice, and is therefore chosen for implementation in the Layer III decoder.

5.3 Operation Count for Huffman Decoding

Having chosen the clustered huffman decoding algorithm it can be estimated how many operations one Huffman decoding will require. Two estimates applying for the worst case situations mentioned above and in Section 4.2, will be made. These implies that both an operation count for a hit and a miss followed by a hit must be estimated. Starting with the hit, it seems reasonable to assume the following. One operation to copy four bits from the code stream, one operation to calculate an address for the lookup table, one load operation, one operation to test for a hit, three operations to extract codeword length and the variables x and y from the lookup table entry, and one operation to update code stream. For the sequence of a miss followed by a hit the following operation count is estimated. One operation to copy four bits from the code stream, one operation to calculate an address for the lookup table, one load operation, one operation to test for a hit, one operation to extract next codeword length from the lookup table entry and one operation to update code stream . For the following hit the above estimate will apply. The operation count is summarized in Table 5.3.

Operation type	Worst case situation 1	Worst case situation 2
Address calculations	1	2
Code stream operations	2	4
Load operations	1	2
Tests	1	2
Extract operations	3	4

Table 5.4: Summary on operations to perform in two worst case situations.

5.4 CRC Calculations

The MPEG/Audio Layer III includes the possibility of including CRC check word into each frame, and thereby making it possible to verify the integrity of the frame. Not all data in the frame are included into the CRC calculations, but only the most critical data, i.e. parts of the header and the side information. Depending on whether the frame contains a single or dual channel signal, the number of bytes included into the CRC calculations is 19 or 34, respectively. In the following section, different methods for implementing the CRC check will be presented. These methods includes both hardware and software implementations.

To be able to describe the CRC calculations mathematically, it is necessary to introduce binary fields and binary polynomials. A binary field is an algebraic system in which additions, subtractions, multiplications and divisions can be performed. The binary field only contains two elements, '0' and '1'. Additions and subtractions of binary elements are conducted using the XOR operator, and multiplications are done by AND'ing the elements. Divisions are only legal when the denominator equals '1', thus leaving the numerator unchanged. Binary polynomials are polynomials where the coefficients all belongs to the binary field, e.g. $0, 1, x, x+1, x^2, x^3+1$, where x is a dummy variable. Binary polynomials are often described as a sequence of bits where the position of each bit corresponds to the degree of the dummy variable. For instance, the bit sequence 100011 corresponds to a fifth degree polynomial $1 + x^4 + x^5$. Operations on binary polynomials can be performed in same manner as ordinary polynomials, except the coefficents are manipulated using the rules of the binary field, which is also known as modulo-2 calculations. By describing a bitstream as binary polynomials, multiplications with x^n corresponds to n right shifts of the bistream.

$$
\begin{array}{ll}
\text{addition} & 1 + x + x^3 + x^6 \\
\text{(or subtraction)} & \phantom{1 +{}} x + x^2 + x^4 + x^6 \\
\hline
 & 1 + x^2 + x^3 + x^4 \\
\hline\hline
\end{array}
$$

$$
\text{Multiplication} \quad (1 + x^2 + x^4) \cdot (x + x^3) = \underline{\underline{x + x^7}}
$$

$$
\begin{array}{lll}
 & & x^2 + x + 1 \quad \leftarrow \text{Quotient} \\
\hline
\text{Division} & x^4 + x^2 + 1 \;: & x^6 + x^5 + x + 1 \\
 & & x^6 + x^4 + x^2 \\
\hline
 & & x^5 + x^4 + x^2 + x + 1 \\
 & & x^5 + x^3 + x \\
\hline
 & & x^4 + x^3 + x^2 + 1 \\
 & & x^4 + x^2 + 1 \\
\hline
 & & x^3 \quad \leftarrow \text{Remainder} \\
\hline
\end{array}
$$

Table 5.1: *Examples of operations involving binary polynomials.*

A given bitstream which is to be guarded by a CRC check word, can be decribed by a message polynomial, e.g. if the bitstream is '101011' then the message polynomial is $u(x) = 1 + x^2 + x^4 + x^5$. To each CRC check word a CRC generator polynomial, is associated. In the case of MPEG/Audio, this generator polynomial is $g(x) = 1 + x^2 + x^{15} + x^{16}$. The CRC check word is the remainder resulting from divison of $x^{16}u(x)$ by $g(x)$. The remainder, $s(x)$, will in the following be written as Equation (5.1).

$$
s(x) = R_{g(x)}\left[x^{16}u(x)\right] \tag{5.1}
$$

The literature describes several methods for calculating the CRC check word in either hardware or software, [Ramabadran and Gaitonde, 1988, Hasan and Sain, 1995]. The hardware method evaluated in this section is a bit-wise implementation, and the software methods evaluated are a bit-wise implementation and two byte-wise implementations.

5.4.1 Hardware Implementations

The simplest method for solving the problem of calculating the remainder from division of $x^{16}u(x)$ by $g(x)$, is by using a linear feedback shift register LFSR, see Figure 5.3. This circuit is able to simulate the division operation, by feeding the message $u(x)$ into the input of the circuit, one bit at the time. In the case of a stereo MPEG/Audio bitstream, the $8 \cdot 34 = 272$ bits to be verified are fed to the LFSR circuit one bit at the time, and if the contents of the registers equals the CRC check word, the information within these 272 bits can be presumed correct.

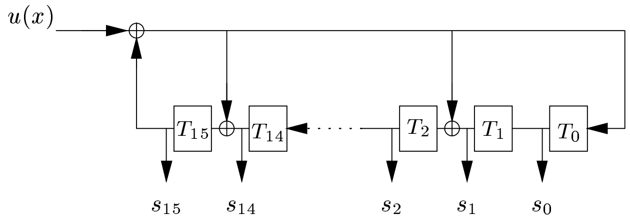


Figure 5.3: *Linear Feedback Shift Register (LFSR). Circuit for calculating the remainder from a division between to binary polynomials. T_n symbolizes shift registers and $\oplus$ symbolizes* XOR *functions. $s(x)$ is the remainder from the division.*

5.4.2 Software Implementations

It is possible to emulate the operations of the LFSR hardware implementation in software, however it is possible to speed up the calculations by processing more than a single bit at the time. By rewriting the algorithm it is possible to calculate the CRC check word, by processing a byte of the message at the time. In the following it will be shown how Equation (5.1) can be rewritten to a byte-wise calculation.

The basis of the CRC calculation is Equation (5.2) where $u(x)$ is a given message, and $g(x)$ is a 16th degree generator polynomial.

$$s(x) = R_{g(x)}\left[x^{16}u(x)\right] \tag{5.2}$$

For some message polynomial Equation (5.2) can be rewritten to Equation (5.3), where $q(x)$ is the quotient of the division between $x^{16}u(x)$ and $g(x)$, and $s(x)$ is the remainder.

$$x^{16}u(x) = q(x)g(x) + s(x) \tag{5.3}$$

Let $u(x)$ represent a message, and $s(x)$ represent the corresponding CRC check word. Let $b(x)$ represent the succeeding byte of the message, which has to be included into the CRC check word, $s(x)$.

$$\begin{aligned} u(x) &= u_0 + u_1x + u_2x^2 + \ldots && (5.4)\\ s(x) &= s_0 + s_1x + s_2x^2 + \ldots + s_{15}x^{15} && (5.5)\\ b(x) &= b_0 + b_1x + b_2x^2 + b_3x^3 + b_4x^4 + b_5x^5 + b_6x^6 + b_7x^7 && (5.6) \end{aligned}$$

The message polynomial $u'(x)$ after adding $b(x)$ to the message, becomes Equation (5.7), where the original message $u(x)$ has been shifted 8 places to the right and the new byte is inserted on the vacant byte.

$$u'(x) = b(x) + x^8u(x) \tag{5.7}$$

The new CRC check word $s'(x)$ can be calculated in normal manner by Equation (5.8).

$$s'(x) = R_{g(x)}\left[x^{16}u'(x)\right] \tag{5.8}$$

Equation (5.8) can be rewritten to Equation (5.11) by using Equation (5.3)

$$s'(x) = R_{g(x)}\left[x^{16}b(x) + x^8x^{16}u(x)\right] \tag{5.9}$$

$$s'(x) = R_{g(x)}\left[x^{16}b(x) + x^8\left(q(x)g(x) + s(x)\right)\right] \tag{5.10}$$

$$s'(x) = R_{g(x)}\left[x^{16}b(x) + x^8q(x)g(x) + x^8s(x)\right] \tag{5.11}$$

By exploiting that the remainder of a sum of two polynomial, equals the sum of the remainder of the individual polynomials, Equation (5.11) can be rewritten to Equation (5.12)

$$s'(x) = R_{g(x)}\left[x^{16}b(x)\right] + R_{g(x)}\left[x^8q(x)g(x)\right] + R_{g(x)}\left[x^8s(x)\right] \tag{5.12}$$

since $g(x)$ evenly divides $x^8q(x)g(x)$, Equation (5.12) can be reduced to Equation (5.13)

$$s'(x) = R_{g(x)}\left[x^{16}b(x)\right] + R_{g(x)}\left[x^8s(x)\right] \tag{5.13}$$

$$= R_{g(x)}\left[x^{16}b(x) + x^8s(x)\right] \tag{5.14}$$

The numerator of Equation (5.14) can now be rewritten to Equation (5.15), and by collecting the coefficients of the two polynomials and substitute $s_{i+8} + b_i$ by t_i; this becomes Equation (5.16).

$$x^{16}b(x) + x^8s(x) = \sum_{i=0}^{7} b_i x^{16+i} + \sum_{i=0}^{15} s_i x^{8+i} \tag{5.15}$$

$$x^{16}b(x) + x^8s(x) = \sum_{i=0}^{7} t_i x^{16+i} + \sum_{i=0}^{7} s_i x^{8+i} \tag{5.16}$$

The new CRC check word after including $b(x)$ can now be calculated with Equation (5.17).

$$s'(x) = R_{g(x)}\left[\sum_{i=0}^{7} s_i x^{8+i}\right] + R_{g(x)}\left[\sum_{i=0}^{7} t_i x^{16+i}\right] \tag{5.17}$$

Since the degree of the first expression is smaller than $g(x)$, Equation (5.17) can be reduced to Equation (5.18).

$$s'(x) = \sum_{i=0}^{7} s_i x^{8+i} + R_{g(x)}\left[\sum_{i=0}^{7} t_i x^{16+i}\right] \tag{5.18}$$

Equation (5.18) consists of two expressions, where the first expression is the lower byte of the previous CRC, shifted 8 places to the right. The second expression demands the calculation of the remainder of the higher byte added to the new byte. Note that addition in this context refers to addition in a binary field, which is a modulo-2 addition, and is easiest calculated using xor. Different algorithms exist for efficient calculation of $s'(x)$ from $s(x)$ and $b(x)$, in the following two of these methods will be described, namely two methods called CRC with lookup table, and CRC with reduced lookup table.

CRC with Lookup Table: The second expression in Equation (5.18) is a calculation of the remainder of a 7th degree polynomial, multiplied with x^{16}. As the coefficients of the 7th degree polynomial only attain '0' or '1', only 256 possible polynomials exist and can therefore be precomputed. Thus for every new byte added to the CRC check word, only a single memory reference is required.

This algorithm can calculate the CRC check word of a n bytes long message in n steps, compared to the bit-wise implementation, this is an 8 fold speedup. The cost of this speedup is memory usage, the CRC lookup table requires 256×16 bit memory.

Algorithm 5.1 *CRC calculation with lookup table. CRCLUT() is a lookup table with the pre-computed values of* $R_{g(x)}[t_7x^{23} + t_6x^{22} + \ldots + t_0x^{16}]$.

Initialize:
1: $\mathbf{s} = (0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0)$
Body:
2: **for** bytes in buffer **do**
3: $\mathbf{b}$ =new byte
4: $(t_7,t_6,t_5,t_4,t_3,t_2,t_1,t_0) = (b_7,b_6,b_5,b_4,b_3,b_2,b_1,b_0)$ XOR $(s_{15},s_{14},s_{13},s_{12},s_{11},s_{10},s_9,s_8)$
5: $\mathbf{s} = (s_7,s_6,s_5,s_4,s_3,s_2,s_1,s_0,0,0,0,0,0,0,0,0)$
6: $\mathbf{s} = \mathbf{s}$ XOR CRCLUT($\mathbf{t}$)
7: **end for**
8: $\mathbf{s}$ CRC check word

CRC with Reduced Lookup Table: A method to reduce the memory usage in the CRC calculation is to dissolve the second expression of Equation (5.18) into Equation (5.19).

$$R_{g(x)}\left[\sum_{i=0}^{7} t_i x^{16+i}\right] = \sum_{i=0}^{7} R_{g(x)}\left[t_i x^{16+i}\right] \tag{5.19}$$

Only eight possible remainders exist for the right hand side of Equation (5.19), thus the lookup table can be reduced to eight entries. By assuming that '0' and '1' are evenly distributed, $4n$ table references are required for an n-byte message.

5.4.3 Evaluation of CRC Implementations

Choosing between a hardware and a software solution, is often guided by a tradeoff between choosing high performance – the hardware solution, and low complexity of the hardware – the software solution. This is also the case when choosing which CRC solution to use. The hardware solution is very simple, it only requires 16 registers and a few logic gates. The critical path in the LFSR circuit is also very short, which makes it able to work at a high clock frequency. When evaluation performance with the demands in the BSP, it is clear that the LFSR will deliver a substantial higher performance than needed for the BSP. The worst-case demands in the BSP, appears when decoding a 48 kHz stereo signal at 320 kbit/s. At 48 kHz the BSP is receiving ≈ 42 frames per second, and in each frame $8 \cdot (32+2)$ bits have to be verified. This results in a total of only ≈ 1500 bytes per second, which have to be verified. Besides of delivering an excessive performance, the LFSR will increase the complexity of the datapath. The software solution has several advantages in this case: *i)* It can easily be reprogrammed to handle other generator polynomials. *ii)* It does not complicate the datapath. *iii)* It can easily fulfill the requirements of the BSP. The disadvantage of using the software solution (the version with lookup table) is more memory usage. When evaluation the pros and cons of the hardware solution and the software solution with the demands of the BSP, the software solution is the better choice.

6

Complexity Analysis

From the algorithms listed in Chapter 4 and Chapter 5, it is possible to estimate how many instructions per second the BSP must perform in order to process a Layer III bitstream.

These estimations are necessary to guide the design process of the BSP. The complexity analysis is a part of the hardware-software tradeoff performed when deciding which operations to perform in hardware and which part to perform in software. By making hardware support for a operations, the operation becomes faster, at the cost of more complexity in the hardware.

The estimates in this chapter are based on assumptions, which later in the design process might prove to be inaccurate. Nevertheless these assumptions serves their purpose as an indication of the number of instructions the BSP must deliver, per second, in order to perform the Layer III decoding in real time. Some of the assumptions are listed below:

- Both register operations, memory operations and program modifications, i.e. arithmetic operations, load/store and jumps, respectively, can be accomplished in a single instruction cycle.
- Conditionals and loops, i.e. `if` and `for` loop requires two instructions, one for evaluating the statement, and one for performing the program modification.
- All operations which involves the bitstream requires two instructions. These assumptions originates from the fact that all accesses to the bitstream is on bit level, meaning that a specific number of bits has to be extracted from the bitstream before they can be used. If such an operation is performed on a typical ALU by using logic operations, the demands will typical be higher than two instructions, but as most of the Layer III decoding involves access to a bitstream, it is at this point assumed that the BSP will contain some kind of hardware support for bit-wise access to a bitstream.
- The bitstream is received using interrupts, at the cost of four instructions per byte. This approximation of four instructions per byte is based on the following assumptions: *i)* The bitstream is received byte-wise and stored in a circular buffer – this costs two operations. *ii)* The reception is performed in an interrupt, and an interrupt call is assumed to cost two instructions, to store and reload status of the internal registers of the BSP.
- The result from the decoding in the BSP is send to the DSP via interrupt. As in the case of the bitstream receiver it is assumed that this operation costs four instructions, however in this case the result is send word-wise.

In the following sections it will be estimated how many instructions per second the BSP must be able to perform in order to process a Layer III bitstream. These estimations are based on worst-case, e.g. sampling frequency at 48 kHz, bitrate at 320 kbit/s and short windows. Not all calculations will be presented here, but only the results, as they are rather trivial.

Complexity of the Bitstream Receiver

At the highest bitrate the BSP receives 320,000 bit/s. These bits are processed byte-wise resulting in 40,000 bytes per second. By using the assumption from above which states that each byte costs four instructions, the total cost of the bitstream receiver becomes: 160,000 instructions per second.

Complexity of Communication with DSP

The amount of data to be transmitted to the DSP depends of the sampling frequency, and is in worstcase 48 kHz. At 48 kHz stereo, 96,000 frequency lines and 96,000 scalefactor values are transmitted to the DSP. In addition to the frequency lines and the scalefactors, some system information is also transfered to the DSP, but the amount of system information is negligible in comparison. By using the assumption that each word costs four instructions, this results in a total of approximately:

$$4 \cdot 2 \cdot 96,000 = 768,000 \text{ instructions per second.} \tag{6.1}$$

Complexity of the Synchronization and Error Checking

At most, the calculations involved in the synchronization and error checking is a few operations once per frame – these operations are not included in this calculation. If CRC is enabled, approximately 1500 bytes have to be verified per second, see Section 5.4. By using the CRC table lookup algorithm, this results in:

$$1500 \cdot (2 + 3 + 1 \cdot 2) \approx 17,000 \text{ instructions per second.} \tag{6.2}$$

Complexity of Huffman Info Decoding

In the case that all frequency lines belong to the big_values partition, 3 decisions, 1 call to the Huffmann decoding algorithm and two arithmetic operations must be performed 288 times for each channel in each granule. When the number of frames per second is approximately 42, the total number of instructions per frame can be calculated as follows:

$$42 \cdot 4 \cdot 288(3 \cdot 2 + 1 + 2) \approx 436,000. \tag{6.3}$$

Complexity of Huffman Decoding

In Table 4.1 two worst case situation were analyzed and estimates for computational complexity were presented for two situations. Common to both situations is the number of memory operations required to store the decoded values. With a sampling frequency of 48 kHz this yields 96,000 instructions per second. The additional operations to be performed depend on the particular worst-case situation.

Situation 1 In this case the Huffman decoding routine is called 48000 times. Choosing the clustering algorithm as Huffman decoding algorithm, the number of operations to perform, listed in Table 5.3 can be used to calculate the total number of instructions required:

$$96000 + 48000(1 + 2 \cdot 2 + 1 + 2 + 3) = 624,000. \tag{6.4}$$

Situation 2 In this situation approximately 25,000 zero stuffings, 24,000 Huffman decoding calls and 48,000 ESCAPE_value handling have to be performed. The zero stuffing is assumed to require only one memory operation per value. Each ESCAPE_value handling takes a single test operation, a single addition and one bitstream operation. Using the clustering algorithm for Huffman decoding again, the total number of instructions needed in this situation can be calculated as follows:

$$96,000 + 24,000(2 + 4 \cdot 2 + 2 + 2 \cdot 2 + 4) + 25,000 + 48,000(2 + 1 + 2) \approx 840,000 \tag{6.5}$$

Complexity of the Decoding of Scalefactors

In the case of short windows, 36 coded scalefactors for each channel in each granule must be decoded. This results in a total of 144 scalefactor decodings per second. Each decoding requires one memory reference, one bitstream reference and is done within a `for` loop. At 48 kHz approximately 42 must be decoded per second, resulting in a total of:

$$42 \cdot (2 + 1 + 2)(144) \approx 30,000. \tag{6.6}$$

Complexity of Preprocessing

This preprocessing is necessary for copying the scalefactor from each scalefactor band out to each frequency line. This operation requires approximately 36 load and 576 store for each channel in each granule, including approximately two loop control instructions per frequency line. This results in:

$$42 \cdot 4(576 + 36 + 2 \cdot 576) \approx 300,000. \tag{6.7}$$

Complexity of Reordering

The reordering is only necessary when using short windows. The cost of reordering is approximately three memory references and two loop control instructions, for each scalefactor and two memory references and two loop control instructions for each Huffman value. Totally this results in: $4((576(3+2)) + (576(2+2))) \approx 21,000$ instructions per frame. This results in approximately $900,000$ instructions per second.

6.1 Summary of Complexity Analysis

Table 6.1 summarizes the results from the previous calculations. It can be seen that the worst-case demands of a Layer III decoding demands approximately 3.5 million instructions per second (MIPS). This number will be used in the following chapter to guide the design of the BSP architecture.

Part of Layer III	**instructions per second**
Bitstream Receiver	160,000
Communication with DSP	768,000
Synchronization and Error checking	17,000
Decoding of scalefactors	30,000
Huffman info decoding	436,000
Huffman decoding	*840,000
Preprocessing	300,000
Reordering	900,000
Total	3,451,000

Table 6.1: *Approximations of the number of instructions required to perform the Layer III decoding.*) Situation 2 is used.*

7

Overall Design of the BSP

In this chapter some overall considerations are made concerning the internal architecture of the BSP, and it is decided which essential features and units to implement. The chapter will cover RISC and CISC processors, and different bus architectures, such as Von Neumann and Harvard architectures. The chapter concludes with a short description of the internal architecture of the BSP and its interfaces to other units.

7.1 Discussion of Architectures

This section introduces some of the architectures resulting from four decades of constructing computers. The architectures of computers have evolved much since the first computers were announced in the fifties and sixties. A major factor in this evolution, is the refinements of the silicon technology which makes it possible to produce processors with millions of transistors on relatively small dies. Because of the advances in the silicon technology it is possible to make sophisticated microprocessors, which were only theoretical ideas, few years ago.

The architecture of a microprocessor is defined as the functional capabilities of the processor, and the interconnection scheme of the internal units in the processor. By functional capabilities is referred to which types of operations the processor is capable of performing, e.g. if it can perform complex calculations in single instructions. In the following two of the most prominent architectures will be described shortly, and the characteristics of these architectures will be emphasized. Later this description will be used as a guideline for choosing an architecture for the BSP.

Data and Address Busses

The data and address busses are the communication links in the microprocessor, and are used for transferring data between memory and processor. Several bus architectures exist, and one of the most simple is known from the "Von Neumann" computer. The term Von Neumann computer covers one of the simplest and oldest concepts, where the program and data is placed in the same memory space, and thereby only needs one data bus and one address bus. All computations in a Von Neumann computer are done on a single arithmetic logical unit (ALU), and the program execution is carried out by a control unit, see Figure 7.1.
The simplicity of the "Von Neumann" computer is also its vulnerable point. The shared program and memory bus, is likely to become the bottleneck in the system, and will therefore put an upper limit on the program execution speed of the computer.

Another concept, which is just as old as the Von Neumann computer, is the "Harvard Architecture". The Harvard Architecture is an extension of the Von Neumann computer, where data and program are placed in separate memories, and accessed by separate busses. This enables the processor of loading a data operand and a program instruction simultaneously. The Harvard Architecture has been taken a step further, where it is provided with two data memory spaces

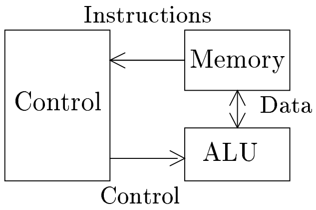


Figure 7.1: *The original Von Neumann machine.*

and one program memory. This concept is know as the "Modified Harvard Architecture", and has primarily been used in DSP's because of its ability to perform repetetive load of two data operands, as needed in *sum of products* calculations. The Von Neumann, the Harvard and the Modified Harvard architecture are sketched in Figure 7.2.

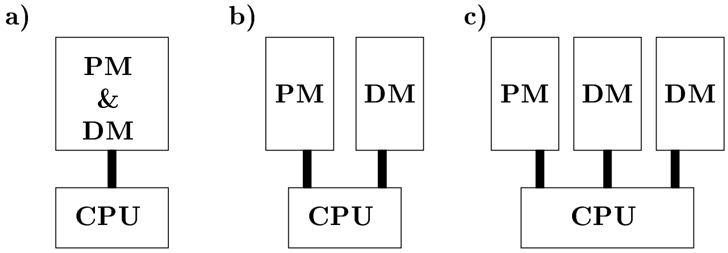


Figure 7.2: ***a)*** *The Von Neumann computer.* ***b)*** *The Harvard architecture, with separate program memory and data memory.* ***c)*** *Modified Harvard architecture, with one program memory and two data memories.*

Although clearly different from the original concepts, the Harvard and Modified Harvard concepts, are today used on processors with a single memory space, but seperate internal program and data caches, which enables the processor of loading data operand(s) and program instruction simultaneously. This is a modification of the original concept, because it requires that the data is already in the cache. Examples of such processors are the MC68040 from Motorola and Nx586 from AMD, [AMD, 1997, Clements, 1992].

Instruction Set

Another method for classification of microprocessors, is by the instruction set. The instruction set is traditionally divided into two groups, the CISC group (Complex Instruction Set Computer) and the RISC group (Reduced Instruction Set Computer). The CISC/RISC terms basically concern some philosophies of whether arithmetic operations should be performed directly on data in memory, and whether each instruction is performed directly in hardware, or it is translated into a sequence of microcode operations.

The first microprocessors developed, had only a few internal registers for holding data, and had to spend a significant amount of time on loading and storing data. These processors were then equipped with more complex instructions which enabled the processor to perform operations directly on operands residing in memory, and thereby reducing the time spend on loading several instructions. An example of a complex operation is the `add` operation in the Motorola 68000 processor. This operation is able to perform an addition of the contents of two memory cells, and store the result in a third memory cell, [Clements, 1992].

`ADD D0,D1` ; *[D1]* $\leftarrow$ *[D0]* + *[D1]*

One of the costs of implementing CISC instructions, is long instruction code, which typically cannot be fetched in a single load operation.

In a RISC instruction set, all instructions have fixed length, typically 32 bits. The RISC computer cannot perform arithmetic operations directly on operands in memory, but has to load the operands into internal registers, before performing the operation. The simplicity of the RISC operations makes it possible to shorten the critical path, and thereby achieve a higher clock frequency, and that way compensate for the simplicity of the operations. Because of the simplicity of the RISC instructions, they can be implemented directly in hardware. When taking the lid of a RISC processor, it should be possible to actually see the hardware which is responsible for the individual instructions. The CISC processors have taken another angle to this, because of the complexity of its instructions. By dividing each instruction into a sequence of microinstructions, a more simple set of microinstructions is obtained, and these microinstructions can then be implemented directly in hardware.

As the architectures of microprocessors have evolved, it has become more difficult to classify them by the RISC or CISC terms. The reason is that most modern processors have features, which in their original form were reserved for only one of the two philosophies. This is also the reason that the RISC and CISC definitions are used more loosely nowadays, for example are some processors referred to as a "CISC processors with RISC features", [AMD, 1997].

Pipelining

Pipelining is an implementation technique allowing multiple instructions to be overlapped in execution. Basically, pipelining is used for reducing the critical path in the processor, and thereby allow it to run at a higher clock frequency. In the following, pipelining will be shortly explained, and some of the techniques which exist for making pipelining more efficient will be introduced.

By pipelining the datapath in a processor, the instructions can typically be divided into five stages, in which the processor performs the following jobs:

Instruction Fetch: The instruction is loaded from memory, and the program counter is incremented.

Instruction Decode: The instruction loaded in the previous state, is decoded, in order to apply the correct control signals to the datapath.

Execution: The operation is performed.

Memory Access: If the instruction is a memory access, this is the stage where it is performed.

Write Back: In the write back stage, the result from either execution or memory access, is written into the internal register.

If each of the above stages is completed within one clock cycle (cc), and all stages were executed sequentially, then n instructions would require $5 \cdot n\ cc$. The problem with executing each stage sequentially, is that only one stage is active at the time, thus making the utilization of the processor poor, see Figure 7.3. Each of the five stages can be executed independently, which makes it possible to use each of the stages as an assembly line, where each stage completes a part of an instruction. Each time the clock ticks, the instruction moves one stage forward, and after five ticks the instruction is completed. The advantage introduced by pipelining is that when the instruction has completed the first stage, and moved on to the second stage, the first stage is ready to start fetching the consecutive instruction. By using pipelining, the latency for a single instruction is maintained at 5 cc, but the throughput is increased with a factor five (n instructions requires, $(n+4)\ cc \approx n\ cc$, for $n >> 1$).

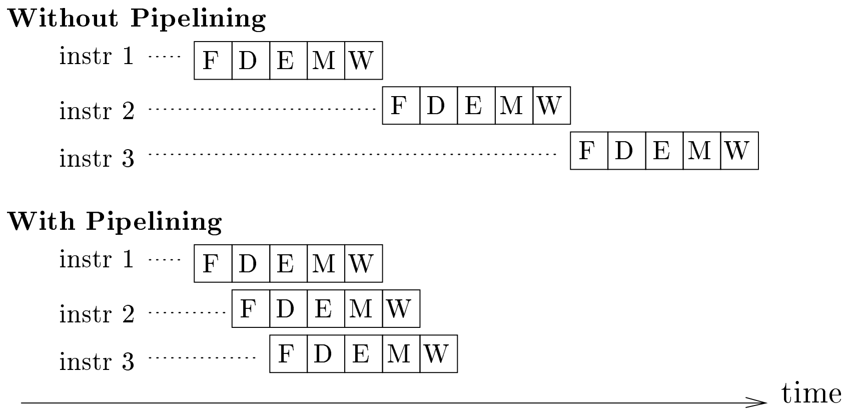


Figure 7.3: *Illustration of gain by pipelining. Without pipelining, three instructions takes 15 cc, with pipelining this is reduced to 7 cc. When the number of instructions is high, the pipelined can execute five times as many instructions, than without pipeline.*

Pipelining introduces some problems, in terms of data hazards and control hazards. Data hazards occur when the result from one instruction, is used in the succeeding instruction. An example of such a data hazard is shown on Figure 7.4, where the result from the first instruction is needed in the following instruction. Because of the latency in the pipeline, the result from the first instruction is not ready in time to be used by the second instruction.

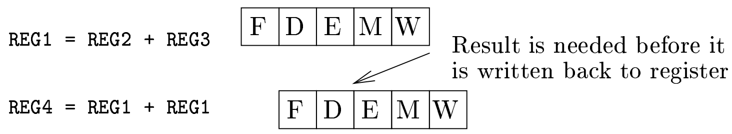


Figure 7.4: *Illustration of a data hazard, the result from the first instruction is needed in the following instruction before it is written back to the registers.*

Basically, the problem can be solved using one of three methods: *i)* Simply by avoiding such data hazards, when writing the assembly program, by inserting `nop`'s and by reordering the assembly program. *ii)* The second method is to stall the pipeline until the result is written back into the register. Both of these methods guarantees correct execution, but the cost of correctness is lower performance. *iii)* The third method is called forwarding, and covers a method of transferring data between stages, e.g. in the case shown in Figure 7.4 the result can be transfered from the execution stage of the first instruction directly into the execution stage of the next instruction. This method of avoiding data hazards is far more efficient than the previous methods because it does not stall the pipeline. But, even though forwarding ensures correct results when operating on registers, it cannot "save the day", when an instruction tries to load a register with data from memory, and then tries to use it in the following instruction.

Just as there exists data hazards, there exists control hazards. Control hazards appears when the assembly program contains data dependent loops. When reaching a condtional branch in the assembly program the processor cannot know whether it should be taken, before the branch instruction has reached the execution stage. Control hazards will not be discussed further here, but just be pointed out that control hazards cannot be avoided. However, sophisticated methods of minimizing the penalty of control hazards, such as *branch prediction*, have been developed, [Hennessy and Patterson, 1994].

When pipelining a microprocessor, the microprocessor must be able to perform all five stages simultaneously, e.g. it must be able to perform an “Instruction Fetch” and an “Execution” simultaneously. In the instruction fetch stage the program counter has to be incremented, hence it requires some kind of addition. The execution unit must clearly also contain the ability of adding two numbers, hence the two stages both requires an adder. A method for reducing the hardware requirements in the processor, is by time-sharing such functional units between the stages, hence move away from pipelining. This solution will in the following be referred to as a time shared processor.

Time Shared Implementation

The time shared processor is an alternative to the pipelined processor. Where the pipelined processor is able to achieve a high throughput, at the expense of additional hardware, the time shared processor is able to reduce the hardware usage, at the expense of throughput.

The term “time shared processor” covers the strategy of sharing the functional units in the microprocessor, during an instruction cycle. Besides reducing the amount of hardware required, the time shared processor allows the individual instructions to take different numbers of clock cycles, without introducing further complexity, as illustated in the following example.

> The contents of register1 is saved at the address contained in register2 plus an offset of 5, in assembly language this can be written as:
>
> `STO 5(REG2),REG1` ; *memory(REG2+5)* $\leftarrow$ *REG1*
>
> This operation does not perform any operation in the “write back” stage, and in the pipelined processor this will just make the hardware, which performs the write back operation, idle. In the time shared processor it will be possible to break off the execution of the instruction after the “memory access” stage, and initiate the following instruction immediately.

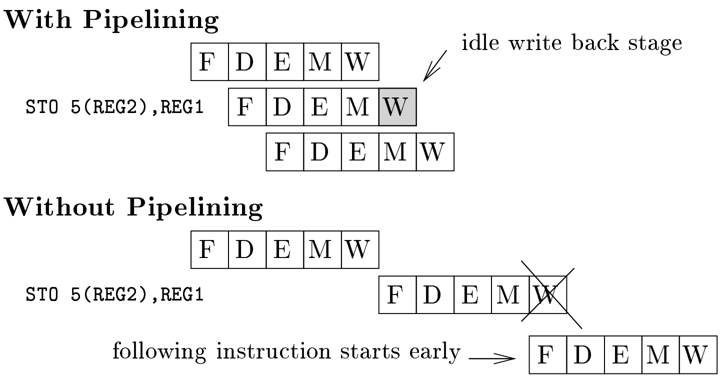


Figure 7.5: *Illustration of the time shared processors ability of handling instructions with different length.*

Architecture Chosen for the BSP

In Chapter 6 it was concluded that if the BSP processor could execute approximately 3.5 million instructions per second (MIPS), then it would be able to decode a Layer III bitstream in real time. The MIPS count and the fact that the algorithms, in Layer III, contain a great part of control loops and conditional jumps, are used for deciding which architecture to use in the BSP.

First of all, the time shared processor seems best suited for Layer III because of the moderate MIPS count. Assuming that each instruction can be executed in a time shared processor with an average of four clock cycles, then 3.5 MIPS will result in a clock frequency at approximately 15 MHz. At 15 MHz the processor can have 67 ns latency in the datapath, which seems very reasonable when targeting an implementation in standard cells. With a performance of 3.5 MIPS it is not necessary to use pipelining. Pipelining would introduce more complexity in the datapath. If pipelining were to be used, it would require an efficient handling of data and control hazards because of the many short data dependent loops in the Layer III algorithms.

The algorithms to be implemented on the BSP does not contain operations which repetitively needs two operands from memory, hence there are no special need for employing a Harvard bus architecture. This and the fact that multiple busses increases the complexity of the datapath, is the reason for chosing a Von Neumann bus architecture.

Each instruction is defined to consist of three stages, a **fetch** stage, a **decode** stage and an **execution** stage. The execution stage is allowed to have varying length, depending on the instruction, e.g. an internal register-register operation can be executed faster, than operations requiring memory access. By allowing the execution length to vary, it is possible to achieve a higher effective MIPS count than if all instructions had the same execution time as the worst-case instruction.

7.2 Interface to BSP

As it was stated in Chapter 3, the BSP is unable of decoding a Layer III bitstream as a stand alone unit, but must be able to communicate with other units, first of all for receiving the bitstream, and secondly for transmitting the unpacked and preprocessed data to a DSP. Both interface ports are described separately in the following paragraphs. To provide the BSP with the possibility of working on a shared bus, together with other processors, the BSP is provided with features for bus arbitration.

Bitstream Receiver

The bitstream receiver is the interface responsible for receiving the MPEG/Audio bitstream to be decoded. The receiver is simply a synchronous serial receiver, capable of receiving a bitstream at 320 kbit/s. If the bitstream is sent as a part of an MPEG-1 audio/video bitstream, bitrates at 1.5 Mbit/s are allowed. Even though the MPEG-1 bitstream is demultiplexed into seperate audio and video bitstreams before the audio bitstream is received by the BSP, the bitrate is unchanged. As a consequence, the bitstream receiver must be capable of receiving a bitstream with a mean bitrate at 320 kbit/s, but with burst bitrates at 1.5 Mbit/s.

Bus Arbitration Interface

If the BSP is to allow other processors to access its memory directly, it must support bus arbitration. By allowing a master processor to access the BSP memory it can reprogram the BSP at runtime. This facility will be necessary if the BSP's program is placed in static RAM, and has to be reloaded at every power up. The BSP is designed as a slave and therefore always sends a bus request. If the bus is granted, it starts executing the program, until a central bus arbiter, decides to reclaim the bus. The BSP must only be denied the bus for shorter periods, if the bitstream receiver is to receive the incoming bitstream flawless. If the BSP is denied the bus in longer periods, overflow will occur in the bitstream receiver, because the BSP cannot move the bitstream to memory.

Interface to DSP

After each frame has been unpacked and processed in the BSP, the data is sent to a DSP, which then performs the last algorithms of the Layer III decoding. The worst-case amount of data, to be transmitted, occurs when decoding a stereo signal, sampled at 48 kHz. Together with each quantized frequency line, a scalefactor value is sent to the DSP, giving a total of $2 \cdot 2 \cdot 48,000 \cdot 16\ bit/s \approx 3.1$ Mbit/s which has to be transmitted to the DSP. The interface between the BSP and the DSP could be constructed by letting the DSP reading directly in the BSP's memory. This is possible because of the bus arbitration facilities implemented in the BSP. By using the bus arbitration, the BSP will be denied the bus in approximately 200,000 clock cycles per second, not including overhead caused by handshaking during the bus arbitration. Another possibility is to let the BSP use dual ported memory. In this case the DSP will be capable to read data directly from the BSP's memory without stalling the BSP. Both these methods put low demands on the BSP, and the problem of reading from memory and determining when new data is ready is left over to the DSP. Most DSP's are able of receiving serial data with a minimum of overhead, and at a bitrate of 3.1 Mbit/s both the ADSP-21xx and the TMS320C3x series will be able to receive the serial data. One major advantage by using serial communication between the BSP and the DSP, is the simplicity of the connection. Using serial communication only three signals are required, a serial clock, a frame signal and a serial data line. If the quantized frequency lines were to be transmitted in parallel, it would cost at least 16 data lines, plus a few control signal.

User Defined Ports

It is chosen to implement three user defined ports on the BSP. These ports are memory mapped, and can be used for e.g. connecting a UART for debugging or LCD display for displaying information about the bitstream. The ports could also be used for a D/A-converter, if the BSP was expanded to also handle the DSP part of Layer III.

7.3 Interrupts

It is chosen to implement interrupt handling in the BSP. The reason for this decision is mainly based on considerations about the two serial interfaces. Interrupts are a comfortable method of servicing I/O ports, especially when larger amounts of data is to be received/transmitted. It is chosen to implement seven prioritized nested interrupts. The exact characteristics of each of these interrupts will be described in Chapter 8. Six of the seven interrupts are used internally, and the seventh interrupt is an external interrupt. The Layer III algorithm implemented in the BSP does not need an external interrupt, but an external interrupt is included to keep the BSP open for future expansions. The external interrupt is implemented as a vectored interrupt with eight possible vectors. All interrupts are implemented as vectored interrupts, meaning that each interrupt has reserved a specific word in memory space where the address of the interrupt service routine is placed.

7.4 Word Width in the Datapath

It is chosen to design the BSP as a 16-bit architecture, meaning that the wordwidth of the datapath, databus and address bus all are 16 bits. A 16-bit architecture was chosen as the better solution, when comparing it with an 8-bit alternative. By choosing a 16-bit architecture a much simpler memory interface was archived, and it is possible to perform all calculations needed in the algorithms, without having to include extended precision operators in the BSP, e.g. both the Huffman Decoding and CRC calculations require more than 8-bit precision in their calculations.

7.5 Testability

It is necessary to provide some sort of testability in the BSP. Even though the hardware in the BSP might work perfectly, it is necessary to provide the processor with some debugging facilities in order to debug the software execution the BSP. This is done by dedicating one of the internal interrupts for a *trace*-interrupt. The trace interrupt is to be called once for every instruction cycle. The actual function of the trace interrupt is left to the programmer, but it can for instance be used to print out the contents of the internal registers, or to trace the execution of the assembly program, by printing out the program counter. If a UART is connected to a user port, the trace interrupt can also be used in connection with a monitor.

7.6 Summary of Features to be Implemented in the BSP

The findings and decisions in the previous sections are summarized here to a set of guidelines for designing the BSP.

- The BSP is implemented as a time shared microprocessor, with three stages in each instruction, i.e. fetch, decode, execution. These stages does not have the same length, as the length of the execution stage are allowed to vary.
- The BSP is designed as a load/store architecture, meaning that only arguments residing in internal registers can be used in operations on the ALU. If arguments are placed in memory, they first have to be loaded into internal registers before they can be used.
- The BSP is designed as a pure 16-bit architecure, meaning that the datapath, registers, databus and addressbus all are 16 bits wide.
- The BSP is designed with a Von Neumann architecture.
- Seven prioritized nested interrupts are implemented. One external interrupt with eight associated interrupt vectors, and six internal interrupts. The vector of the external interrupt is transfered via the databus.
- The BSP has bus arbitration logic, for making it able to function as a slave on a shared bus.
- The BSP receives the bitstream on a serial port. The required speed of the bitstream receiver is 320 kbit/s.
- The quantized frequency lines and associated scalefactors are transmitted to the DSP via serial port at approximately 4 Mbit/s.
- The BSP is equipped with three user ports.

Figure 7.6 shows a sketch of an environment where the BSP could function. This could for example be a DAB application where a master processor controlled a user interface and the overall control of the decoding.

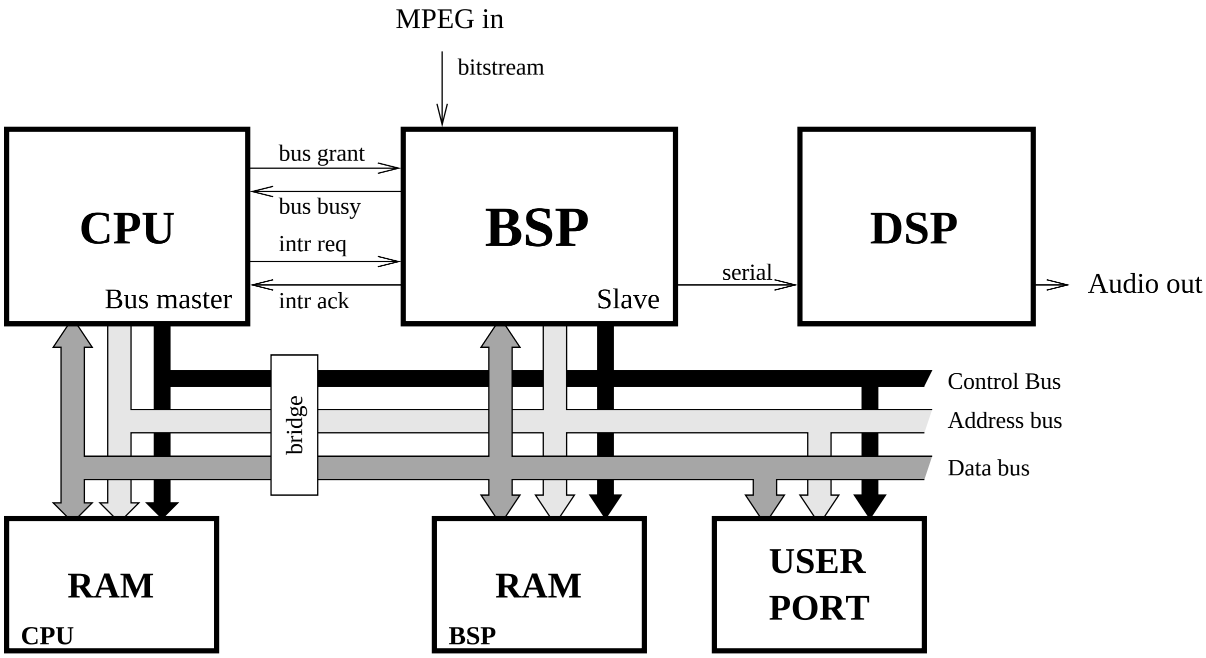


Figure 7.6: *Sketch of an environment where the BSP can be used.*

8

BSP Hardware

In this chapter the design of the BSP will be presented. The design will be performed with the starting point from the BSP guide lines presented in Chapter 7. An overall structural view of the BSP to design is illustrated in Figure 8.1.

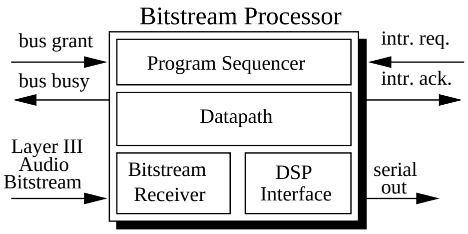


Figure 8.1: *A structural view of the Bitstream Processor.*

In short the overall BSP can be interpreted as a traditional microprocessor offering the features tabulated below.

- A program sequencer in that interprets program instructions and sets up control signals for the remaining units in the processor. The program sequencer is also dedicated to handle interrupts and bus arbitration.
- A datapath in which all arithmetic and logical operations are performed. The datapath is controlled by the program sequencer.
- A bitstream receiver which initially acts like a serial input port.
- A DSP interface which will be implemented as a serial output port.

In the sections to come, the overall BSP will be developed into the Layer III specific BSP. The development will reflect the strategy applied through the design process. The strategy takes a start in the design of a datapath skeleton flexible for addition of special purpose units. These units will then be designed in turn and added to the architecture. Finally the program sequencer, the DSP interface and an address decoder will be designed. The chapter concludes with a summary of the processor designed.

8.1 Datapath

In this section a datapath skeleton for the BSP will be designed. The purpose of the datapath is to make the execution of the necessary operations possible. Before beginning the design of the datapath the decisions regarding architecture will shortly be summarized. In Chapter 7 it was decided to design a time shared processor. It was further decided that the architecture of the processor should have some of the typical RISC characteristics. One of these, which is used in this section, is the following: All operations can only be performed using the internal registers in the processor. This directly affect the architecture of the datapath in the following way:

- Consequently when using a Von Neumann architecture a single memory block is used to hold both program and data.

- There is only a single execution unit, i.e. an ALU.

The time shared implementation requires that the functional units in the datapath are used for different purposes. Therefore multiplexors must be added in order to cope with these different situations.

The datapath is designed the bottom up way, using a datapath skeleton as a basis, to be expanded as the demands to meet the requirements for more functionalities arise. The rest of this section will be organized as follows. First the major datapath elements, i.e. registers, program counter, instruction register, ALU etc. needed in order to execute an instruction will be presented. These major datapath elements required to perform the individual types of instructions will then be connected together in the end of this section.

Instruction Execution: The basic set of operations needed can be grouped into three classes, i.e. arithmetic–logic, control and memory–reference operations.

- Arithmetic–logic class: add, sub, shifts, and, or, xor etc.

- Memory–reference class: load and store.

- Control class: jump, conditional jump and call.

Before looking at what is necessary to implement the individual instruction types, the needs for executing an instruction in general is considered. To execute an instruction, the first thing to be done is to fetch the instruction from memory. To prepare for fetching the next instruction the program counter (PC) must be incremented to point at the next instruction in memory. The basic elements used to perform the first step in executing the instruction and the connections between them are shown in Figure 8.2.

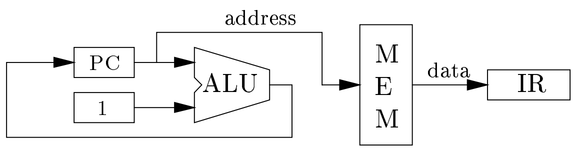


Figure 8.2: *Datapath elements that make an instruction fetch, IR ← MEM(PC) and PC ← PC + 1 possible. It is assumed that an instruction only occupies a single memory location.*

Arithmetic–logic Instruction Class: Instructions belonging to the arithmetic–logic class are all similar in their execution. They all perform two register reads, an ALU operation and then writes the result of the operation back into the destination register. The datapath elements needed for this type of instructions are an ALU and registers. The registers could be implemented as a register file. A register file is a collection of registers, where it is possible to read from or write to any of the registers in the register file by specifying the index of the register. A way to implement a register file is using an array of registers and multiplexers, as shown in Figure 8.3

With the datapath elements shown in Figure 8.4 all arithmetic–logic operations supported by the ALU can be executed within a single clock cycle. The actual number of registers is not determined at this point.

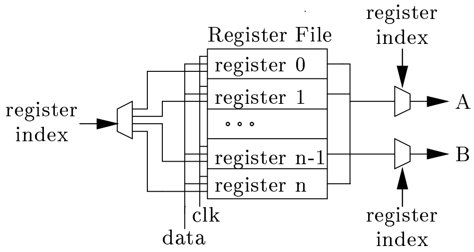


Figure 8.3: *A register file with two read ports A and B, and a single write port. The contents of a register can be read by specifying the register index at the appropriate multiplexer. To write data to a register in the register file, the register index must also be applied, but in a addition to read procedure, a clock signal must also be applied, in order to ensure that the demands on setup and hold times are meet.*

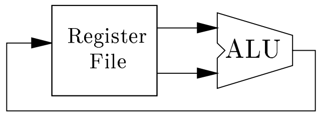


Figure 8.4: *A datapath consisting of a register file and an ALU. This datapath enables the execution of: REG ← REG alu_op REG, where alu_op can be any of the operations supported by the ALU.*

Memory–reference Instruction Class: To be able to perform memory reference operations, memory and additional multiplexers must be added in Figure 8.4. The BSP only supports operations using the internal registers. Therefore memory references can only be performed using the internal registers as addresses, i.e. indirect addressing. In indirect addressing the memory address is contained in the registers, and hence addressing in memory can only be done by specifying a register, which holds the actual address, in the instruction. The address which is calculated by the ALU can then expressed as REG alu_op REG, where alu_op can be any of the operations supported by the ALU. Figure 8.5 shows the datapath elements needed to perform a load operation and Figure 8.6 shows the datapath elements needed to perform a store operation.

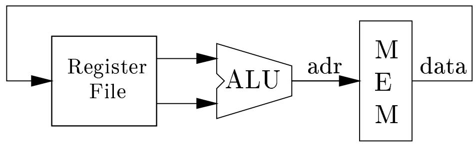


Figure 8.5: *Datapath elements the make the load operation REG ← MEM(REG alu_ op REG) possible.*

Control Class: Until now only instructions used in sequential programs have been considered, i.e. the case where the succeeding instruction is always carried out. To make high–level operations such as IF–THEN–ELSE and FOR possible, decision making must be introduced in the datapath. The decision making consists of deciding whether to execute the next instruction or to perform a program counter modification. The decision is made on the basis of a test. The typical test situation is the subtraction of two registers and checking the status register for some condition, e.g. zero Z or less than zero LZ.

To be able to perform the operations belonging to the control class, where an immediate

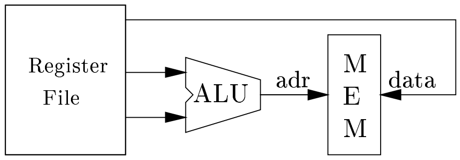


Figure 8.6: *Datapath elements that make the store operation MEM(REG alu_op REG) $\leftarrow$ REG possible.*

value is a part of the instruction, the immediate value must be accessible from the datapath architecture. When deciding whether to execute a conditional jump instruction, it must be possible to test the result from the ALU. For this purpose a status register (SR) is applied. The status register holds the condition flags from the ALU, as shown in Figure 8.7. The chosen condition flags are: less than zero (LZ), and equal to zero (Z).

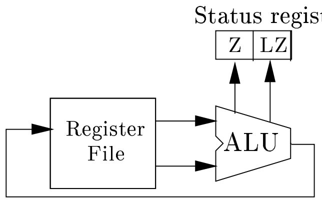


Figure 8.7: *Introducing decision making in the datapath. The condition flags from the ALU are placed in a status register.*

These two ALU status flags can be used to implement a jump less than zero (jumplz) and a jump not zero (jumpnz) instruction respectively. The immediate value is embedded in the instruction and must in the case of immediate instructions be accessible from datapath.

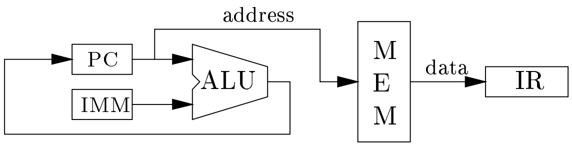


Figure 8.8: *Execution of relative jump, jumpnz and jumplz by performing PC $\leftarrow$ PC + IMM.*

The last operation belonging to the control class is call and its related return. The call operation makes procedure calls possible and hence code sharing becomes easy. Alternatively, the call operation could be done by manually inserting jumps, but compared to the call feature this is not a very attractive alternative. Executing a relative call in general involves the following steps:

1. Save return address.
2. Goto subroutine. (PC $\leftarrow$ PC + Immediate).
3. Execute subroutine.
4. Set program counter to return address.

Three possibilities exist for saving the return address: in memory, in a register or on a stack. The most elegant and common used solution is the stack. The stack solution requires a minimum of management and it is guaranteed that the return address never is overwritten. In addition a stack, facilitates an arbitrarily number of nested calls to subroutines. Placing the stack internal would increase the number of registers in the register file and thereby increase datapath area significantly, and hence it has been decided to place the stack in the memory.

Using a stack requires a stack pointer, which always point to the top of the stack. An execution of a subroutine then involves:

- Push the program counter PC on the stack. (MEM(SP) $\leftarrow$ PC, SP $\leftarrow$ SP + 1).
- Set program counter to address of the subroutine.(PC $\leftarrow$ PC + immediate.)
- Execute subroutine.
- When the subroutine has finished the return address is popped back into the program counter.(SP $\leftarrow$ SP - 1, PC $\leftarrow$ MEM(SP)).

8.1.1 Assembling the Datapath Skeleton

At this point the basic datapath elements required to implement the different types of operations have been presented. As one would expect, many of the required elements can be reused in the different instructions. The only problem is that they are used for different purposes. For instance when executing a load operation, the data on the data bus must be written into a register in the register file. But when executing an arithmetic operation the result from the ALU must be written to a register in the register file. Problems like this are solved using multiplexers.

Register Placement: It must be decided where to place the three registers that were needed throughout the instructions, i.e. PC, SR and SP. It is decided to place them in the register file. One motivation for placing the program counter internal is the immediate addressing mode. By placing the PC internal the execution of immediate operations become much like a register–register operations, which all can be performed within a single clock cycle. Meaning the operation PC $\leftarrow$ PC + IMM can be performed in one clock cycle. The placement is also a consequence of one the architectural decisions listed in the beginning of the section, i.e. all operations must be performed using the internal registers. Placing the PC external would require that it should be loaded into the register file prior to each instruction, which would not be very efficient. In addition the decision contributes to the regularity an simplicity of the datapath, which are keywords when building a datapath. The last fact mentioned is also the reason for placing the status register SR and the stack pointer SP, internal. The reason for placing the status register internal might not be clear at this point, but later the description of the interrupt handling will contribute to the understanding of this decision. The instruction register is placed in the program sequencer.

The assembling of the datapath skeleton is visualized in sub figures a) through f) in Figure 8.9. In a) the datapath elements for fetching an instruction and incrementing the program counter is shown. b) introduces a register file with two read ports, implemented with multiplexers and a single write port. With this all the register–register instructions can be executed. The control signals and the demultiplexer from Figure 8.3 is left out for the sake of simplicity, these will be presented in the end of the section. When the PC is placed in the register file it must be possible to write the contents of the PC to the address bus, this is solved adding a multiplexer, as shown in b). In c) the datapath is improved to execute a load operation. For this purpose a multiplexer is placed on the write port of the register file, in order to be able to select either the output from the ALU or the data bus. In d) the datapath is expanded to be able to execute store instructions. As the ALU is used to calculate the destination address an extra read port is added to the register file. In e) condition flags are added to the ALU and placed in a status register. To be able to execute relative jumps an immediate from the instruction register is led into the datapath, as shown in e). Finally, in f) the status register (SR) and the stack pointer (SP) is placed in the register file.

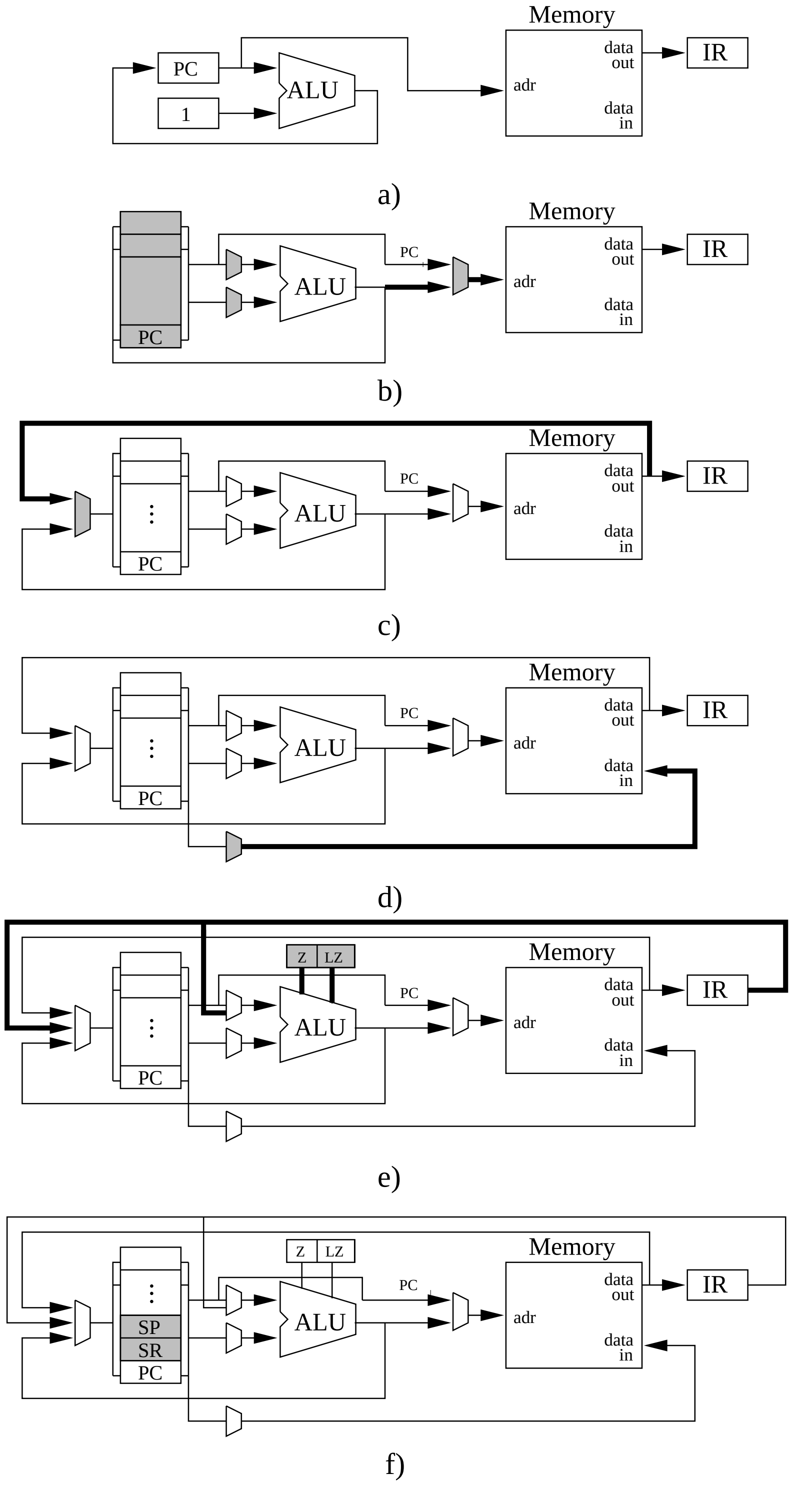


Figure 8.9: *Visualization of the assembling process of the datapath skeleton. The assembling is performed using the basic datapath elements that were necessary in order to execute the desired instruction types. a) shows the initial datapath. a) is then stepwise improved to meet the requirements of the individual instructions. The changes made from step to step is indicated by shaded boxes and bold lines.*

In Figure 8.10 the complete datapath skeleton is shown. Opposite to the previous presented figures, control signals are added. The datapath skeleton consists of the following datapath elements: A memory block, containing both program and data. A register file, with 3 reads ports and one write port, implemented with multiplexers and registers. An ALU, with the condition flags *zero* an *less than zero*. Three additional multiplexers to be able to reuse the datapath elements for different purposes.

The Number of Registers in the Register File By designing the BSP as a load/store architecture where all variables have to be loaded into internal registers before they can be used in calculations, the number of internal registers becomes a critical design parameter. If the BSP is constructed with too few registers, the processor becomes inefficient because it has to spend time in critical loops on loading and storing variables. If the BSP is constructed with too many registers, the area increases and the delay through multiplexers increases, thus making the processor inefficient. An estimate of the number of registers needed in the datapath will be based on the requirements of the Huffman decoding. The flow chart of the Huffman decoding is shown in Figure 4.7. This flow chart is traversed minimum 576 times per frame, hence will contribute significantly to the computational load on the BSP. Using the flow chart in Figure 4.7 will therefore yield a reasonable estimate of the number of registers needed in the register file. The number of registers necessary in Figure 4.7 can be estimated to 5 registers, i.e. a register for holding: the number of bits decoded, the number of reconstructed frequency lines, the table number, the number of linbits in the particular table and a partition counter. 2 additional registers are required to hold each of the decoded values x and y plus a single register to hold the sign. To the 8 registers needed above 2 additional registers should be added for special purpose units, one for the program counter, one for the stack pointer and one for the status register. A register file with 13 entries therefore seems as a reasonable estimate.

The Size of the Multiplexers The size of the multiplexers chosen to be 16–1. The reason for this choice is explained in the following. Its an advantage to have a power of 2 number of registers. Considering the register estimate above a register file containing of 16 registers is an attractive solution. The ES2 library does not include larger than 4–1 multiplexers. This means that a 9–1 multiplexer will contribute with the same combinatorial delay as a 16–1 multiplexer and using 16–1 multiplexers in the datapath seems like reasonable choice. Note that the registers all are 16–bit registers, meaning that the 16–1 multiplexers are all 16 bits wide.

Having defined this datapath skeleton, the next task to be performed is to design the controller which generates all the control signals described above.

8.2 Bitstream Receiver

Functional Demands to the Bitstream Receiver: The purpose of the bitstream receiver is to read and synchronize to the incoming bitstream. To avoid complex data management when accessing maindata, the data will be stored byte-wise and byte-aligned in memory. Hence the bitstream receiver is also assigned the task of ensuring that the data received from the incoming bitstream is stored byte-aligned in memory. The reasons for putting these demands on the bitstream receiver are explained in the following.

The maindata area in a frame transmitted at any bitrate is guaranteed to contain an integer number of bytes. In addition the pointer `main_data_begin`, used in the conjunction with the bit reservoir technique, refers a number of bytes back from the header start. Storing the received data directly in memory without performing synchronization and would require the synchronization to be conducted by the BSP. The synchronization performed by the BSP could be done by loading the received data word-wise into the internal registers and perform comparisons until

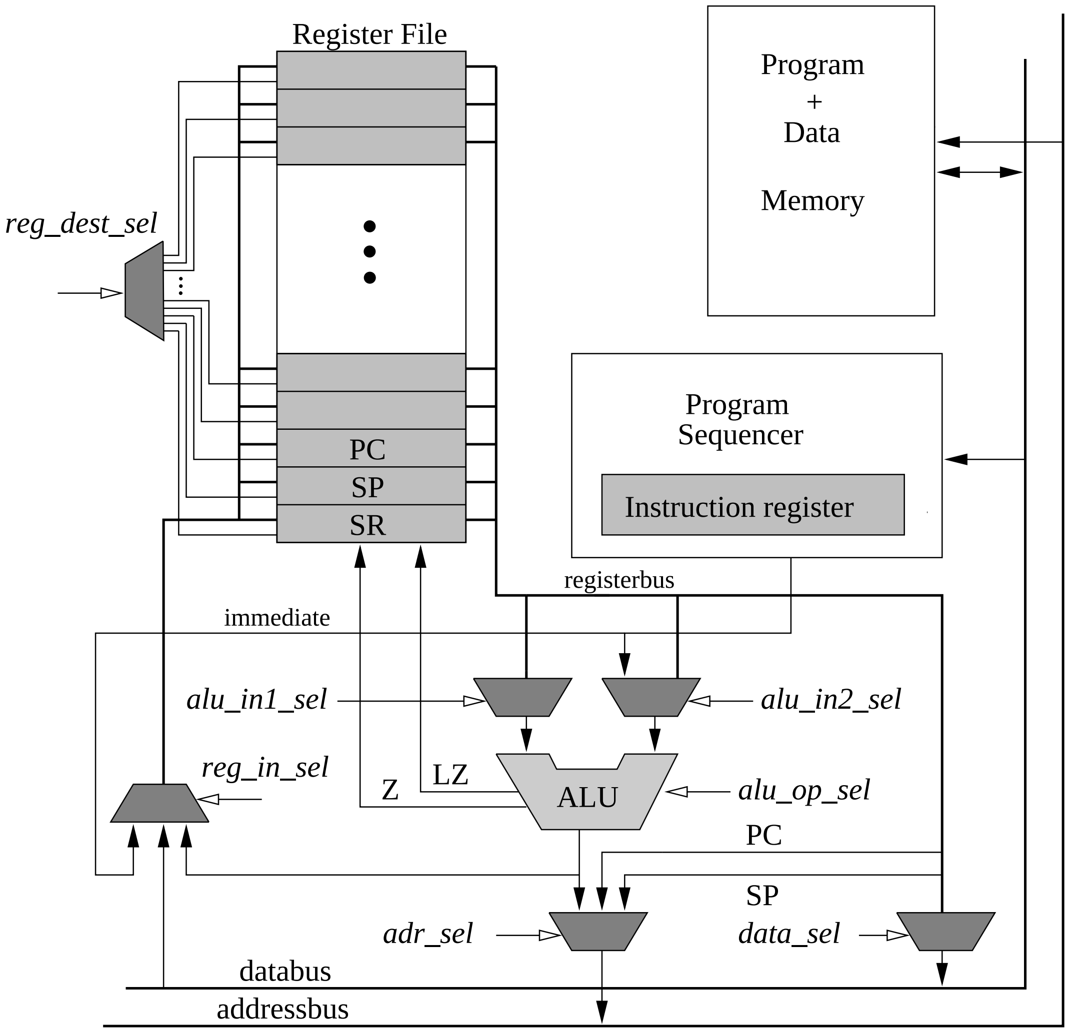


Figure 8.10: *The complete datapath skeleton architecture of the Bitstream Processor. The Program Sequencer is not considered as a part of the datapath, but is included for visualizing the placement of the instruction register and the distribution of the immediate value.*

the synchronization word is found. When the synchronization word has been found the data, in order to be byte-aligned, should then be copied to another place in the memory. If the received data was not byte-aligned, a pointer to the address where the synchronization word starts plus an eventual offset within word, must be used to access the data. Considering the overhead introduced by searching for the synchronization using the datapath and the overhead in keeping pointers to data updated, a synchronization and byte-aligning performed by the bitstream receiver seems like a better solution.

Design of the Bitstream Receiver: The synchronization can be obtained by searching through the bitstream bit-wise until a synchronization word is found. Because spurious synchronization words may appear in the bitstream a continuous search through the bitstream is not a secure way to perform synchronization. Instead it must be possible to stop searching when a header is found and then start again after the frame has been received. This is the reason for making the bitstream receiver programmable. By doing this the actual mode of the bitstream receiver can be controlled from software. Before designing the controller for the bitstream receiver buffering must be considered. To be able to both receive the incoming bitstream and search for the synchronization word simultaneously it is decided to have two buffers, one incoming buffer in which the serial bitstream is bit-wise clocked into and a second buffer containing the outgoing data. The synchronization is performed on the data in the outgoing buffer. The problem now is to determine the size of the buffers. The size of the buffers becomes a tradeoff between area consumption and timing. Because the bitstream receiver is interrupt driven, the size of the buffer directly affects the time interval with which the buffer must be emptied. A large buffer will result in longer intervals between the interrupts from the bitstream receiver, but at the expense of larger area consumption.

The interrupt routine must be able to read the contents of the outgoing buffer before the incoming buffer again becomes full. An interrupt routine has been written for the bitstream receiver, from which the critical path has been estimated. This worst-case situation occurs when the bitstream receiver starts receiving a new frame. In this situation the interrupt routine must first indicate to the main program that the previous frame is completed, synchronize to the new bitstream, read the data from the outgoing buffer into the appropriate buffers, interpret the information in the header and calculate the length of the frame. The number of instructions to perform during this interrupt call are approximately 230 instructions.

The worst-case bitrate of the incoming bitstream is 1.5 Mbit/s. Assuming that the average instruction length is 4 clock cycles and the clock frequency is $15MHz$, the required buffer length can be calculated as:

$$\frac{4\ [cc/instr] \cdot 230[instr/interrut]}{15\ MHz} = 61.3\ \mu s/interrupt \tag{8.1}$$

$$61.3\mu[s/interrupt] \cdot 1.5\ [Mbit/s] = 92\ [bit/interrupt]. \tag{8.2}$$

Hence, the bitstream buffer must be able to hold 92 to bit. In order to get an integer number of bytes in the buffer, this value is rounded up to 96 bits.

It has also been decided to register map the bitstream receiver, which means that it is addressed as any other register in the register file. The two buffers, i.e. the incoming and the outgoing buffer, are shown in Figure 8.11.

To ensure that the synchronization word is not lost some memory must be present in one of the buffers. If the two buffers were equal in size the synchronization word could be lost and hence the outgoing buffer is made 8 bits wider than the incoming buffer, as it can be seen in Figure 8.11.

The control of the bitstream receiver is handled by the two state machines shown in Figure 8.12, i.e. Incoming state machine (ISM) and Outgoing state machine (OSM). The first thing

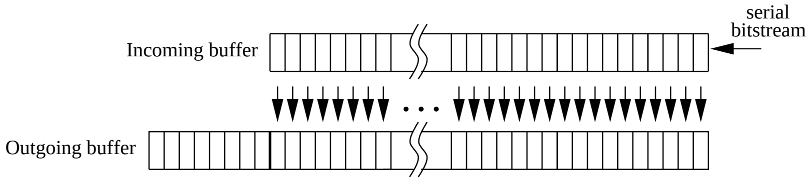


Figure 8.11: *The multilevel buffering in the bitstream receiver consisting of an incoming and an outgoing buffer.*

to be done is to fill up the incoming buffer with bitstream data. This is handled by the ISM. The initial state for the ISM is the state Empty. Being in this state the bitstream data is bit-wise clocked into the incoming buffer. When the Incoming buffer is full, indicating that is 96 new bits are ready, the ISM enters the Full state and signals the OSM that the incoming buffer is full and ready to be loaded into outgoing buffer. The OSM then takes over the control. The OSM initiates in the Idle state. When the incoming buffer is full it enters the Reload state. In this state the contents of the incoming buffer is copied into the outgoing buffer. When the reloading is done it returns to the Idle state and forces the ISM back to the Empty state, indicating that it can start reading new bits from the bitstream.

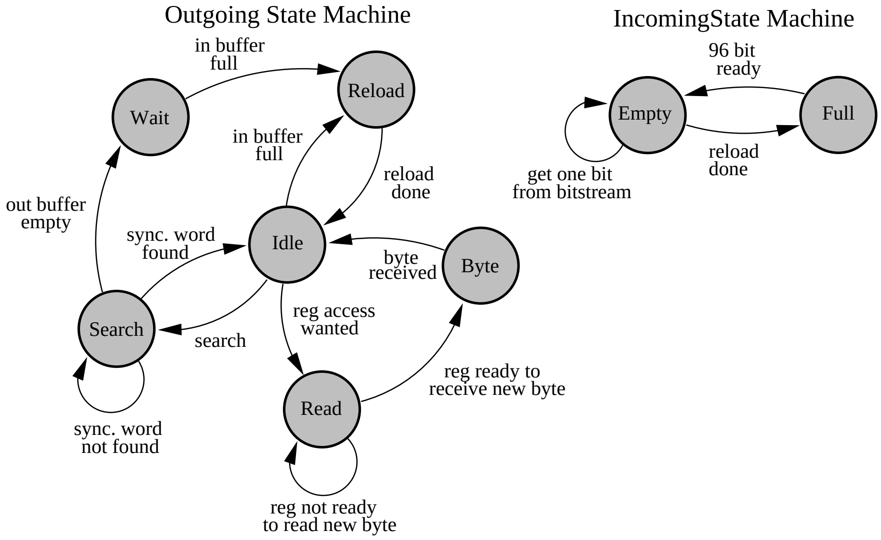


Figure 8.12: *The two state machines that make out the control part of the bitstream receiver.*

Communication with the Bitstream Receiver: Having loaded new data into the outgoing buffer, the control of the bitstream receiver is left to software. Communication with software is done with interrupt signals and via a memory mapped control port. The two interrupts that can occur are: Overflow and NewDataReady. The overflow interrupt indicates if the data in the incoming buffer overwrites unread data in the outgoing buffer and the second interrupt indicates that new data is ready to be read.

Data communication is done using a memory mapped control port. By addressing the memory mapped control port dedicated to the bitstream receiver a chip select signal is generated. Whether data is written to or read from the bitstream receiver, is determined by the control signals (we and oe), generated by the program sequencer.

The state of the bitstream receiver can be read from a status word in the receiver. The rightmost bit SyncFound tells whether a synchronization word has been detected or not. The next 4 bits in the status word indicates the number of bytes ready to be read. An example is shown in Figure 8.14, where the SyncFound is 1 and the next 4 bits equal 0101. SyncFound =

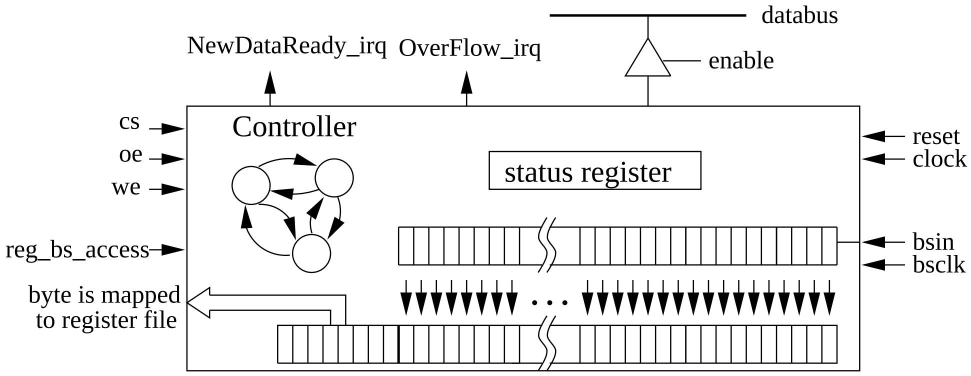


Figure 8.13: *A structural view of the Bitstream Receiver, consisting of a 96-bit incoming buffer, a 104-bit outgoing buffer, a controller and a status register. The leftmost 8 bits of the outgoing buffer are directly mapped to the a register in the register file in the datapath. bsin is the serial input from where the bitstream is received and bsclk is a clock used to shift the bitstream bit-wise into the incoming buffer.*

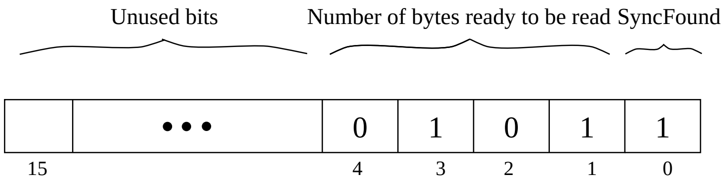


Figure 8.14: *The status word in the bitstream receiver, containing information about the number of bytes ready to be read and a single bit telling whether the synchronization word was found or not.*

1 indicate that a synchronization word has been fund and the bit pattern in the next four bits indicate that 5 bytes are ready in the outgoing buffer.

Searching for the Synchronization Word: When 96 new bits have been loaded into the outgoing buffer, the next step to be performed is to search for a synchronization word in the outgoing buffer. Entering the Search state of the OSM is done by using the communication method described above. The search for a synchronization word is done searching for the word "1111111111111" in the leftmost 16 bits of the outgoing buffer. Having the 16 leftmost bits in the outgoing buffer valid all the time, it is ensured that the synchronization word is not lost. If the 13 leftmost bits in the outgoing buffer do not match the synchronization word, the outgoing buffer is left shifted by one and the matching procedure is repeated. The procedure described above is repeated until a synchronization word is found or until data in the outgoing buffer been used (only the 16 leftmost bits are valid). If a synchronization word is found the SyncFound bit in the status word is asserted, indicating that a frame has been found. After the SyncFound bit has been asserted the OSM returns to the Idle state. If a synchronization word is not found the OSM enters the Wait state, waiting for new data. The OSM remains in the Wait state until the ISM signals that new data are ready in the incoming buffer. When this happens, the Reload state is entered and the data in the incoming buffer is copied into the outgoing buffer. After reloading the OSM enters the Idle state. The action carried out from here depends on whether the synchronization word was found or not. If a synchronization word was found the OSM leaves the Idle state and enters the Read state when access to the register is wanted.

Reading From the Bitstream receiver: Being in the Read state of the OSM, means that a byte is ready in the register, mapped into the register file. From the Read state the OSM awaits a signal from software that indicates it has read the first byte. When this is signalled, the OSM enters the Byte state. In the Byte state the contents of the outgoing buffer is left shifted eight times, which corresponds to placing the most recent unread byte in the dedicated register in the register file. Consecutively the OSM returns to the Idle state.

Byte Aligning: The demand on byte-aligned data from the bitstream receiver can cause some trouble. The problem can occur when searching for a synchronization word in the outgoing buffer. In the case that the number of shifts, performed in order to find the synchronization word, divided with eight yields a nonzero remainder, a special case is present. Which means that a number (8-15) of valid data bits is left in the outgoing buffer when the last byte has been read. Such a situation is described in the following example.

The example describes a situation where the synchronization word, at some point the search procedure, is placed as shown in Figure 8.15.

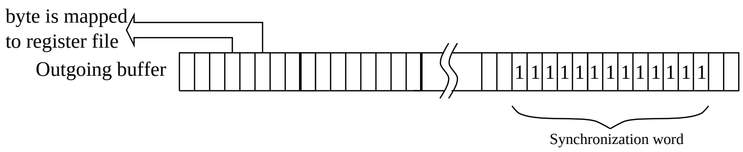


Figure 8.15: *The placement of the synchronization word in the outgoing buffer.*

The search procedure then bit-wise shifts the contents of the outgoing buffer. The procedure stops when last 16 valid data bits are placed leftmost in the outgoing buffer, which means that 88 shifts will be performed. At this point 96-8 = 88 new data bits are reloaded from the incoming buffer and the contents of the outgoing buffer will be as shown in Figure 8.16.

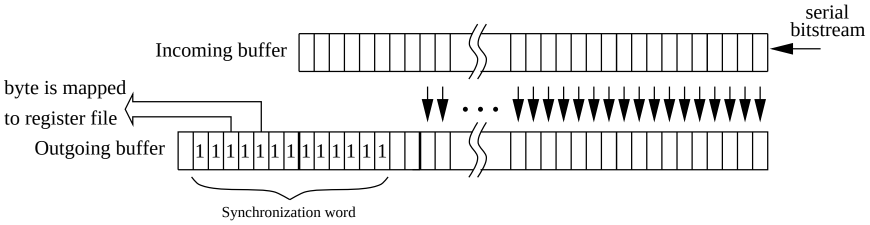


Figure 8.16: *The placement of the synchronization word in the outgoing buffer after 88 shifts and a reload of new 88 bits.*

After the reload of new data, the search for synchronization word is continued. The synchronization word is then found after a single shift of the outgoing buffer. A consequence of shifting the contents of the outgoing buffer one bit left is that the rightmost bit will be invalid, as indicated in Figure 8.17.

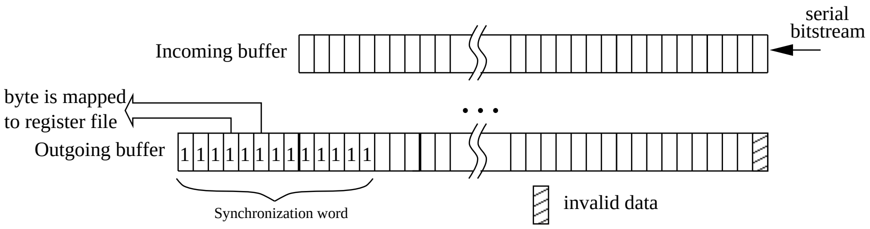


Figure 8.17: *The contents of the outgoing buffer after the synchronization word has been detected.*

Having found the synchronization word the SyncFound bit in the status word is asserted and the number of bytes present in the outgoing buffer are is written in the 4 bits next to the

SyncFound bit. In this case 11 bytes are ready in the outgoing buffer. When the 11 bytes has been read the contents of the buffers will be as shown in Figure 8.18.

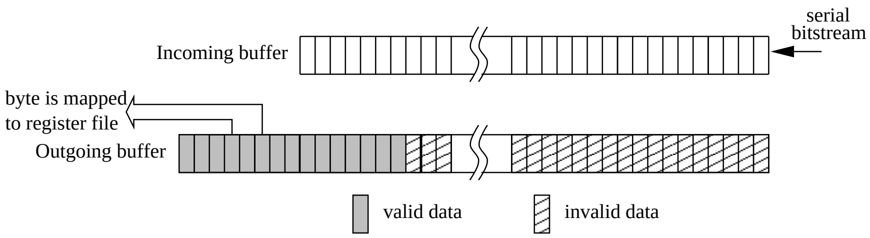


Figure 8.18: *The contents of the incoming and outgoing buffer when 11 bytes has been read.*

To ensure that the invalid bit is not read, the last byte is not read. But to ensure that the seven valid bits are not discarded special handling must be performed. If 96 bits from the bitstream was shifted into the incoming buffer and afterward loaded into the outgoing buffer, the remaining valid bit from Figure 8.18 would be overwritten. What is desired, is only to get the next 89 bit placed to the right of the remaining seven valid bits. In order to achieve this, the contents of the incoming buffer can be exploited, see Figure 8.19 a). From Figure 8.19 it appears that the rightmost 7 bits in the incomning buffer are the same data bit as the bits 8-14 from left in the outgoing buffer. By shifting the incoming buffer 89 times the leftmost 7 bits in the incoming buffer will be equal to the remaining valid bit in the outgoing buffer, as shown in Figure 8.19 b). As a consequence of this the all 96 bits in the incoming buffer can be loaded into the outgoing buffer without overwriting the the valid data bit with new data, as shown in Figure 8.19 c).

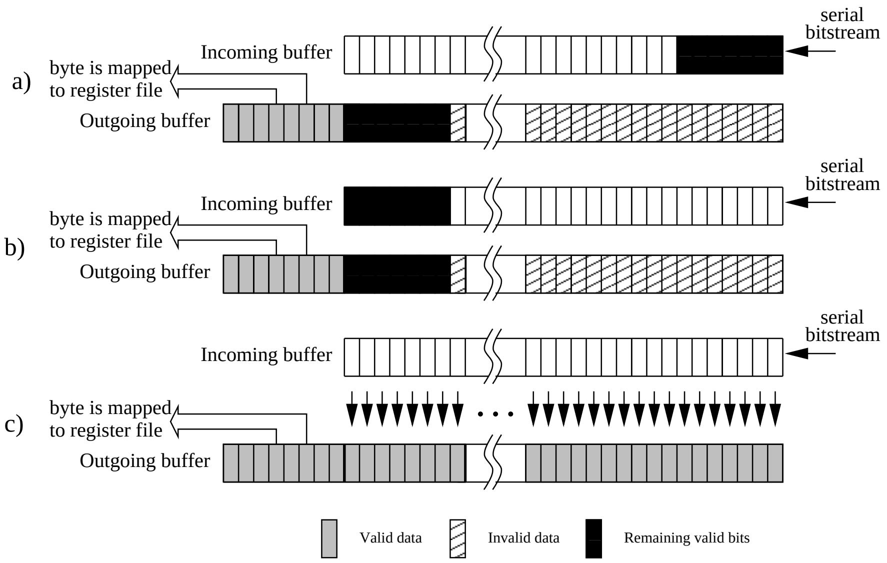


Figure 8.19: *An example of how the byte-aligning is performed.*

In the example described above 7 remainder bits were present, but its possible to have from 1 to 7 remainder bits. The number of remainder bits must be calculated each time a synchronization word is detected, and used to control the number of new bits to be read from the incoming bitstream.

8.3 Fifo Register

The purpose of this section is to design a proper interface between maindata residing in memory and the BSP. The importance of the topic is seen by summing up the accesses to maindata to be performed per second. Looking at the worst case situations sketched for the Huffman decoding in Section 4.2, 192.000 accesses to main data are required.

The processing of a bitstream frame, is rarely performed byte wise or word wise, i.e. 16 bits at the time. When calculating a CRC word or decoding scalefactors and Huffman data, a varying number of bits from the maindata is required at the time. The number can vary from 1 to 16 bits. In Section 8.2 it was stated that the maindata was stored byte–wise in memory. Therefore the interface must be designed toward this storage implementation.

One way to work around the problem of accessing a varying number of bits is to read one main data byte from memory at the time and then decide if all the bits required are present within that byte. If this condition is met, the remainder of the word can be shifted or masked out, leaving exactly the bits required. If the bits required are distributed over multiple bytes, all must must be read and the bits required can then be identified and assembled. Obviously, a word wise access to maindata will be a tedious and inefficient solution, requiring a complex management of what has been read from the maindata.

A superior solution to the above would be to introduce a fifo register within the BSP to contain a few bytes of maindata. When n maindata bits are required they can be accessed in the fifo register. Subsequently to the reading, the fifo register initiates a shift routine in order shift out the recently read data and thereby prepare the next maindata in the fifo register for an access.

The fifo register yields a considerable advantage over the above solution. When n bits are required for processing, they can be accessed from the fifo register in only one instruction cycle. Also, when the fifo register requires new data, these can be accessed with one byte at the time.

8.3.1 Functional Requirements to the Fifo Register

The fifo register to be designed will be accessed very often during decoding of a frame, as stated above. It will therefore be reasonable to place the fifo register within the datapath in order to ensure fast access to maindata. According to Section 8.1, the register file in the datapath skeleton was flexible for changes in the contents. Therefore, the fifo register will be mapped into the register file.

When reading n bits from the fifo register, it will be convenient for the programmer if the n bits were right aligned in the destination register. Only unsigned values resides in maindata, and therefore a zeropadding to the left of the n bits will be required. The ALU is employed to perform the right shifting every time n bits are requested. Thereby the zeropadding will also be performed. In Figure 8.20 it is shown how the fifo register, an arbitrary destination register within the register file and the ALU are interconnected. The figure also illustrates how data from an n bit fifo register access is stored in the destination register.

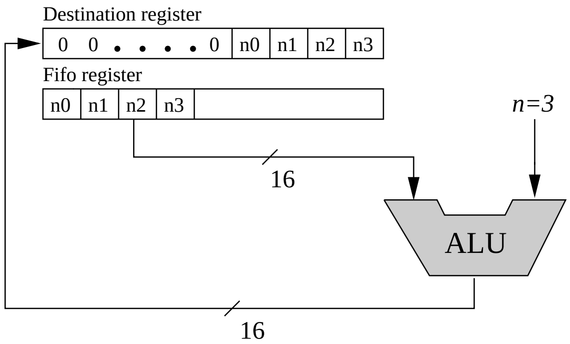


Figure 8.20: *An extract of the datapath showing the fifo register incorporated in the datapath. Applying n = 3 will result in 4 maindata bits right shifted 12 times.*

The fifo registers interface toward the databus will differ from the interface used for the registerfile. As stated above, maindata are stored byte–wise in memory and therefore the fifo register will only utilize the lower eight bits in the interface to the databus.

A single fifo register access migth require up to 16 bits. In order to maintain a single cycle access to maindata it must be ensured that 16 bits always are present in the fifo register. An implementation ensuring this is shown in Figure 8.21. Here a 24 bit fifo register and an 8 bit buffer is implemented. The contents of the 8 bit buffer is copied into the fifo register for every 8 shifts followed by an updating of the buffer.

Using a 24 bit register guarantees that at least 17 bits in the fifo register always are valid, whereby single instruction cycle access to maindata is ensured. The number 17 is made evident by the fact that when only 16 bits are valid, a register update is performed leaving 24 valid bits. An advantage of using the buffer is that the fifo register shifting routine never has to be stopped in order to perform a memory access. The memory access is only of concern to the fifo buffer.

8.3.2 Design of Fifo Controller

In order to make the fifo register and buffer behave properly a controller is required. The fifo controller is also assigned the task of ensuring that the fifo buffer is loaded when required. A programmer designed interrupt routine is found appropriate for the fifo buffer updating. Placing the routine in the main program is impossible because it generally cannot be predicted when an updating is required. The software driven solution will require no extra hardware, i.e. a data address generator (DAG) or a Direct Memory Access (DMA) controller, because the architecture presented so far is capable of loading data into any entry in the registerfile. The consequence of the solution will be that the fifo controller must be capable of interrupting, and the program sequencer, to be presented later, must support interrupts. Prior to the design of the fifo controller, the operations the fifo controller must support will be specified. The names for the operations and the exact interpretation in the fifo controller will be defined in the following.

popfifo(n) The first operation to be supported will be designated the name popfifo(n). For this operation the fifo controller first waits until the contents has been read. Then a sequence of n fifo register shifts is initiated while ensuring the contents of the fifo buffer is copied into the fifo register for every 8 shifts. Every time the latter situation arises the fifo buffer must be updated which is done by setting an interrupt.

resetfifo(): The second operation to be supported is designated resetfifo(). When the operation is employed, the fifo controller initiates four maindata accesses in order to replace the entire contents of both fifo register and fifo buffer. The operation is required prior to the first fifo register access after a system startup. The operation also makes the programmer capable of jumping between more than only one data area in memory and still be confident in that the new contents of the fifo register is what was intended.

When programming the BSP, situations may arise where only a reading of the fifo register is required. This situation will not affect the fifo controller as no shifting will be required.

Figure 8.21 illustrates how the fifo controller interacts with the fifo register and the fifo buffer, and which control signals are required. From the illustration it is seen that the register mapping is conducted so that data is loaded into the 8 bit fifo buffer and data is read from the leftmost 16 bits in the fifo register. Both the fifo register and the fifo controller are driven by a clock called clk_fast. The fifo control and register are clocked four times faster than the rest of the datapath in order to accomplish the shifting task fast.

The fifo register is controlled by register_enable and the latch/toggle signal. The first signal is an indication of whether a change in the fifo register contents is to be performed or not. If a change is to be conducted the latch/toggle signal is read. The latch condition indicates that the fifo buffer should be latched into the fifo register, and toggle indicates that the register must shift its contents, once for every rising edge of clk_fast. The signal reg_sel is activated when the register file index selects the fifo buffer for storage.

The fifo controller is driven by four control signals. The reset signal simply puts the fifo controller into an initial state. The signal reset_fifo indicates that the operation resetfifo() must be initiated. The signals shift_signal and num_shift indicates that a shifting routine must be initiated and the number n to shift the contents of the fifo register. fifo_irq is an fifo controller generated interrupt signal used for signaling the program sequencer that the fifo buffer requires an updating.

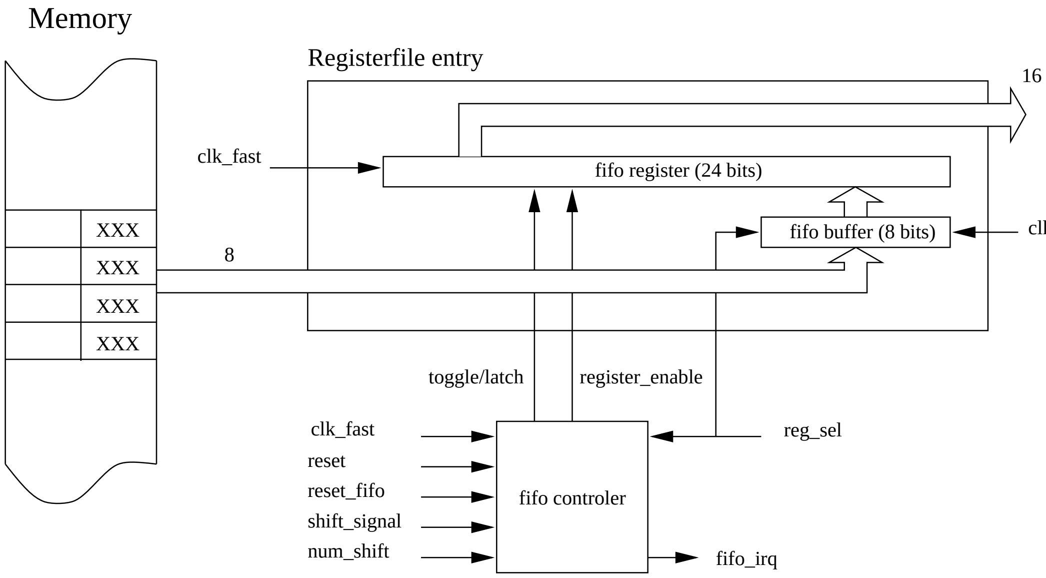


Figure 8.21: *Implementation of the fifo register. The illustration shows the interconnection between the elements constituting the fifo register. The interfaces to both memory and the register busses within the datapath are also shown.*

The fifo controller is a state machine responsible for setting up the control signals for the fifo register and fifo buffer, based on the control inputs described above. Figure 8.22 illustrates this state machine. In the following each of the states and the conditions for transitions between states will be described.

Fifo Ready: Prior to any access to the fifo register the state machine must be reset, that is, the state machine must be set to an initial state. When reset, the state machine is waiting in the state Fifo Ready. The state machine can perform a transition into either Fifo Wait or Reset Register depending on which of the control signals reset_fifo or shift_fifo is activated.

Fifo Wait: The state machine enters this state when the control signal shift_fifo is activated, that is the popfifo() operation has been selected. The state machine is kept in this state until the shift_fifo signal changes to inactive. This assures that the contents of the fifo register has been read, shifted in the ALU and latched into another register in the register file. The state machine changes state into Fifo Toggle when shift_fifo is inactivated.

Fifo Toggle: In this state the fifo register is shifted n times. When accomplished it is tested if a reload of the fifo register is required. If not, the state machine jumps back to the Fifo Ready state. When a reload is required it is tested whether the fifo buffer is full or empty. If it is full, the state machine jumps into the state Reload Register. If the fifo buffer is empty a transition into Wait Reg_sel is conducted. The latter will appear if an

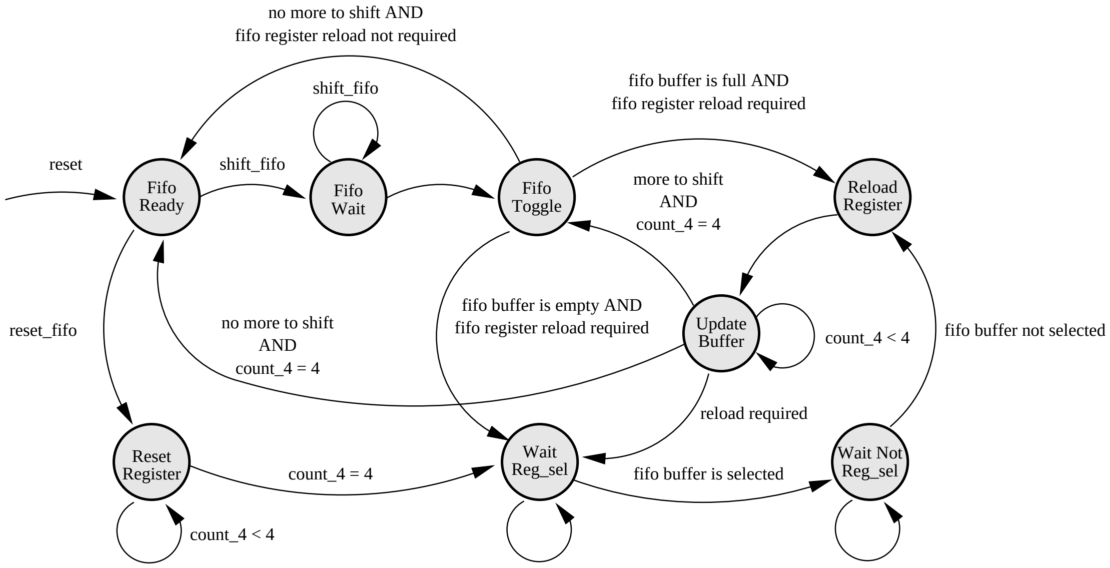


Figure 8.22: *Illustration of the state machine within the fifo controller. The conditions for a state transition are all placed above or to the right of the associated arrow.*

interrupt has been requested but not accomplished yet. Both state transitions will appear even if the fifo register has not finished shifting its contents yet. The shifting process will be stalled until entering the Fifo Toggle state again.

Reload Register: In this state the contents of the fifo buffer is latched into the fifo register. When accomplished, a transition into the state Update Buffer will be performed.

Wait Reg_sel: This state is entered when the fifo controller cannot continue without an updated buffer. The state machine is held in this state until the register file index selects the fifo buffer for storing. Next a transition into the state Wait Not Reg_sel will follow.

Wait Not Reg_sel: In this state the fifo buffer latches 8 bits from the databus into the fifo buffer. The state machine is held in this state for 4 clk_fast cycles in order to synchronize the fifo buffer to the databus.

Update Buffer: This state is used for interrupting the program sequencer. The state machine is held in this state for 4 clk_fast cycles in order to keep the fifo_irq signal active for an entire processor clock cycle. If the state machine has not accomplished all n shifts required a transition into Fifo Toggle will follow. A transition into Wait Reg_sel is used if the fifo register requires a reload, which will be the case when the operation reset_fifo is executed. If none of the above situations are valid the state machine returns to the State Fifo Ready.

Reset Register: When reset_fifo is activated the state machine jumps into this state. Here an interrupt is generated and held active for 4 clk_fast cycles. Also, internal parameters indicating that both the fifo register and fifo buffer is empty are set up. Next the state machine jumps into the Wait Reg_sel state. In order to fill up both the fifo register and fifo buffer the sequence: Wait Reg_sel, Wait Not Reg_sel, Reload Register and Update Buffer is run through three times.

8.4 Program Sequencer

Given a program – the purpose of the program sequencer is to read the program one instruction at the time, and issue all the necessary control signals to the datapath in order to execute the instruction correctly. In Chapter 7 some basic functionalities were discussed. These functionalities puts some demands on the program sequencer. These demands are listed below.

- Read the program, instruction by instruction and issue the corresponding signals to the datapath.
- Resign to a sleep state when the BSP is denied access to the bus.
- Being capable of handling seven prioritized nested interrupts.
- The BSP must be able to execute the following instruction types.
  - Register-register operations. Operations which solely operate on internal registers in the datapath, e.g. arithmetic and logical operations.
  - Register-register operations, where one of the arguments is immediate data, i.e. the argument is a part of the instruction code.
  - Load and store operations, either from memory address placed in register or by the sum of a immediate value and memory address in a register. The latter is for example used for performing memory access relatively to the stack pointer or program counter.
  - Perform calls and unconditional jumps. The BSP defines jumps as jumps in the program where it is not possible to return to the jumping point. Calls on the other hand, stores the jump point on the stack, and can later return to this point.
  - Perform conditional jumps in the program code, for example if the last operation the ALU resulted in a non-zero result.
  - The BSP is equipped with a stack, and must therefore be able to perform push and pop operations. The push operation stores the contents of a register on the stack, and updates the stack pointer. The pop operation writes the most resently stored value on the stack into a register and updates the stack pointer.

Each instruction cycle is basically divided into three stages, a fetch stage, a decode stage and an execution stage. These three stages is by tradition called **F**, **D** and **E** stage. A more precise definition of these three stages will be given in the following.

The Fetch stage: is used for loading the next instruction into the program sequencer. From which memory address to load is determined by the program counter (PC). Afterwards, the PC is incremented to point at the next instruction in memory.

The Decode stage is reserved for the program sequencer to decode the information in the instruction. This decoding includes determining the instruction type, e.g. arithmetic operation, memory reference or program modification.

The Execution stage: is the stage where the instruction is executed. The length of this stage depends on the instruction type, e.g. an arithmetic operation is shorter than a stack operation, which demands both a memory reference and an increment of the stack pointer.

During normal execution, the program sequencer will repeat executing these three stages. As the last part of each instruction cycle, it is tested, on the `bus_grant` signal, whether the bus is granted the BSP. If the bus is not granted the BSP, it will jump into a sleep stage where it stays until it is granted the bus again. To indicate to the bus arbiter that the BSP has gone into sleep mode, the `bus_busy` signal is released, see Figure 8.23.

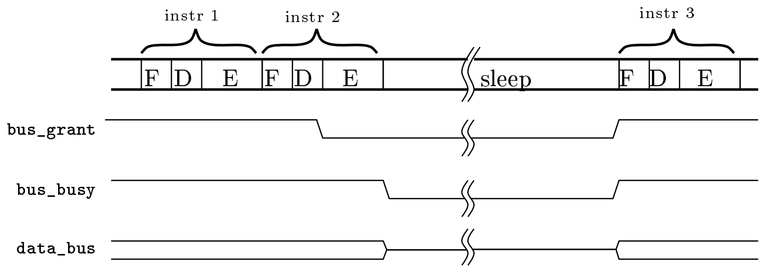


Figure 8.23: *An example of the bus arbitration facilities included in the BSP. When the central bus arbiter denies the BSP access to the databus, the BSP enters a sleep stage and releases the* `bus_busy` *signal. When the data bus is granted the BSP, the BSP leaves the sleep stage and asserts the* `bus_busy` *signal.*

8.4.1 State Machines in the Program Sequencer

The program sequencer is constructed as three state machines, connected hierarchically. These state machines are denoted "Master state machine", "Sub state machine" and "Memory state machine". The master state machine controls the overall functionality of the BSP, the sub state machine is used for controlling the number of clock cycles in each Master state, and the memory state machine is used for timing access to memory. An example of the interactions between these state machines is given by showing the state transactions in the execution state of a write instruction.

`mem(R1+R2)=R0` *; store R0 at address R1+R2*

After the instruction has been fetched and decoded, the master state machine is in its execution state. In this example, the address to be written at, is first calculated in the ALU, and afterwards `R0` can be written to the resulting address. Because of these two consecutive operations, the execution state consumes two sub state cycles. Below the sub state machine, a memory state machine controls the setup of control signals to memory. Figure 8.24 illustrates how the memory state machine is used to time the control signals.

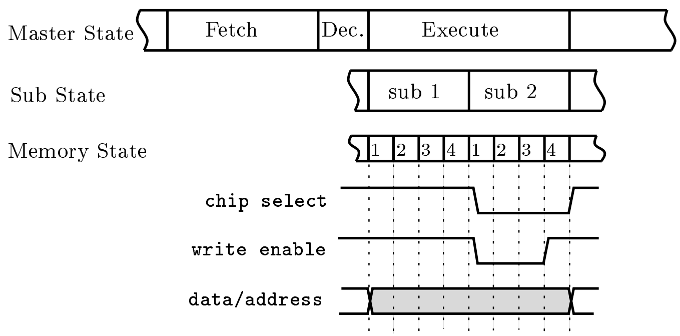


Figure 8.24: *Illustration of the interaction between the hierarchical state machines, when execution a write operation.*

By utilizing a memory state machine it is possible to make very accurate timing of control signal to memory. This is desirable because it enables the BSP to be reprogrammed to access a wide variety of memories.

The master state machine is the backbone of the BSP. During normal execution it cycles between three states, a Fetch state, a Decode state and an Execution state. The master state machine diverts from the Fetch-Decode-Execute cycle in two situations, if it is denied the bus or if an interrupt occurs. As the last step in each Fetch-Decode-Execute cycle, it is checked whether the BSP is still granted the bus. If it is denied the bus, it resigns to a Sleep state, where it stays until it is granted the bus again. Another situation where the master state machine diverts from its normal Fetch-Decode-Execute cycle is in the case of an interrupt. As the last step before initiating a new Fetch-Decode-Execute cycle it is checked whether an interrupt is waiting to be serviced. If an interrupt is pending, the program sequencer first stores the present status of the program counter and status register, and then starts executing the interrupt service routine. A flow chart describing the state flow in the program sequencer can be seen in Figure 8.25.

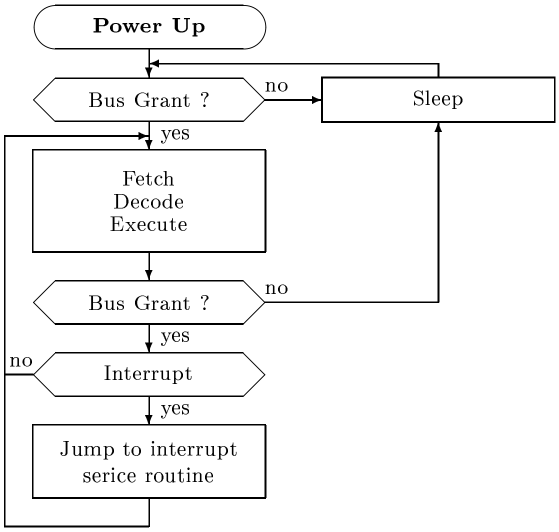


Figure 8.25: *Flow chart describing the state flow in the program sequencer*

Figure 8.26 shows the state diagram corresponding to the master state machine. When the BSP is executing normal instructions without being disturbed by interrupts or bus arbitration, it will cycle in the Fetch-Decode-Execute stages. In case of an interrupt the PushPC-PuchSR-UpdateSR states are executed, and depending on whether it is an external interrupt or an internal interrupt LoadPC or GetVector is executed. The exact nature of these interrupt states will be described more thoroughly in Subsection 8.4.2.

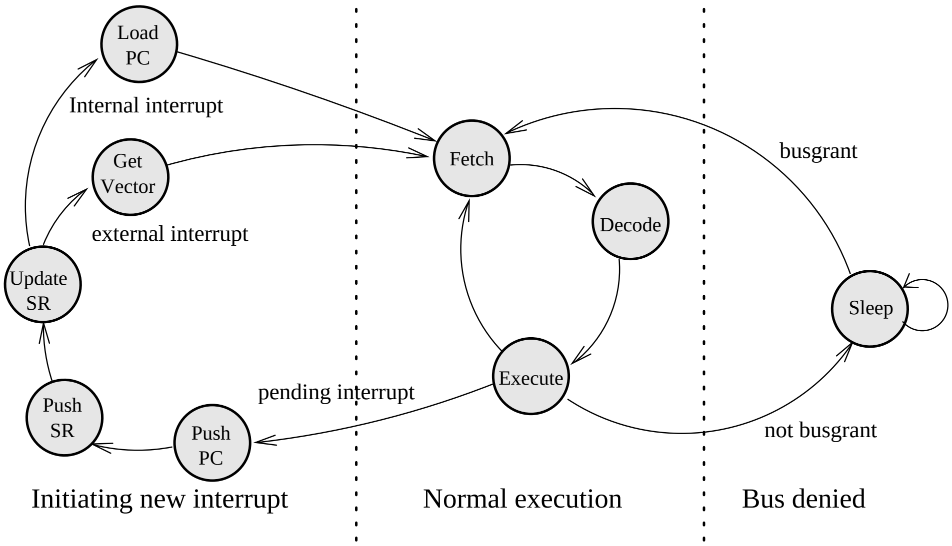


Figure 8.26: *State diagram of the program sequencer.*

The sub state machine is very simple. Basically it functions as a counter, counting down towards zero. When the counter reaches zero it indicates that the master state machine must shift state. The sub state machine is implemented as a state machine with 6 states, (*sub0,sub1,sub2,...,sub5*). In each clock cycle it unconditional shift state from *sub*(n) to *sub*(n+1), and when sub state *sub5* has completed, it indicates that the current master state is finished. The length of the master state is controlled by the starting state, e.g. if the sub state machine starts in *sub4*, the master state machine stays in the current state for two clock cycles. Figure 8.27 illustrates the sub state machine.

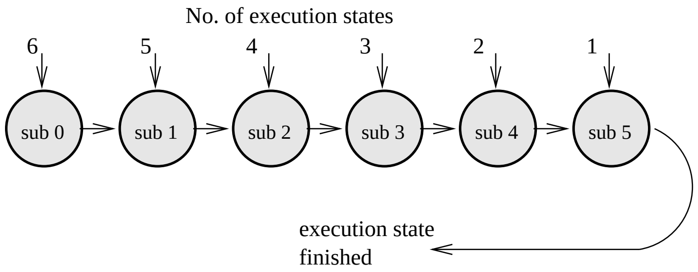


Figure 8.27: *Illustration of the sub state machine used to determine the length of the execution stage in the program sequencer.*

8.4.2 Interrupts

The BSP has seven prioritized nested interrupts. They are prioritized in the sense that, when an interrupt with a higher priority is running, interrupts with lower priority are suspended until the interrupt with higher priority has ended. If an interrupt with low priority occurs during an interrupt routine with higher priority, the low priority interrupt is stored in a pending register. The interrupts can be enabled and disabled at runtime.

The status register is used to hold information concerning which interrupts are enabled, and at which interrupt level the processor is currently running. The status register is shown in Figure 8.28, where the 7 lower bits control which interrupts are enabled, the next 7 bits are used to store the current interrupt level, and the two MSB's are used to hold status flags from the ALU. In order to enable the program sequencer to return to the program execution when

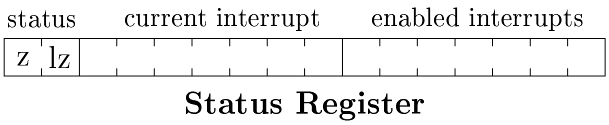


Figure 8.28: *Status Register of the BSP. The lower seven bits are used to control which interrupts are enabled. The seven next bits are used to hold the current interrupt level, and the two MSB's are used to hold the two status flags from the ALU.*

an interrupt has been serviced, the status of the program sequencer is stored before servicing interrupts. This is done by pushing the program counter and the status register on a stack. By pushing the status register on the stack, the status flags from the ALU is stored. This is necessary in the case that an interrupt occurs between an operation used to test on a variable and a conditional jump. By pushing the current interrupt level on the stack it is secured that when the program sequencer returns from an interrupt it always returns to the level from where it was interrupted. In order to make it possible to disable interrupts from within an interrupt service routine, the “enabled” part of the status register is not restored when popping the old status register contents from the stack into the status register.

The seven prioritized interrupts are described in the following. The described interrupts are ordered according to their priority, with the lowest priority first.

Fifo: The interrupt with the lowest priority is the fifo interrupt. This interrupt is asserted whenever the fifo register needs to load a new byte from memory.

Overflow: The overflow interrupt is asserted when an overflow or underflow occurs in the ALU. This feature is not used in the Layer III decoder, but has been included as an extra feature.

Serial: This interrupt is asserted when the serial port are waiting for new data. The serial interrupt will be further described in the following section.

Bitstream receiver: When the bitstream receiver has received new data, it asserts this interrupt.

Overflow in bitstream receiver: If the bitstream receiver interrupt is not serviced in time, before new data arrive, an overflow will occur in the bitstream receiver. By asserting this interrupt it is possible for the BSP to take the appropriate step whenever bits is lost in the receiver. This interrupt can only occur in two situations, if an interrupt with a higher priority is hogging the BSP, or if the BSP is denied the memory bus in longer periods at the time.

Trace: The trace interrupt is included for debugging purpose. If the trace interrupt is enabled, a trace interrupt is generated after each instruction has been executed. The interrupt service routine called by the trace interrupt can be altered by the programmer to fit a specific need, e.g. write the contents of internal registers to a UART. By giving the trace interrupt a high priority, it is possible to trace the program execution through lower priority interrupts.

External: The external interrupt is implemented as eight vectored interrupts. When an external unit asserts the external interrupt in the BSP, it must apply a 3 bit word on the databus, signaling which interrupt service routine to execute. The external interrupt is implemented with a handshake signal to indicate to the interrupting unit that the BSP is servicing the interrupt. Figure 8.29 illustrates how the BSP uses the handshake signal to indicate to the interrupting unit that it is ready to receive the vector on the data bus.

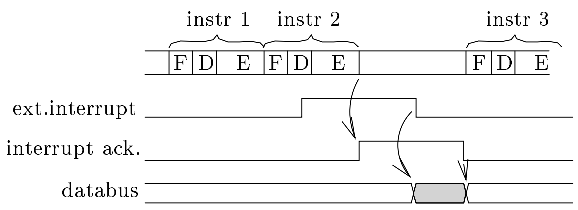


Figure 8.29: *Control signals during an external interrupt. The interrupt acknowledge is used to indicate to the interrupting unit that the BSP is ready to receive a vector on the databus.*

When an interrupt occur, the BSP stores the state of the program sequencer, and updates the program counter to point at the address of the first instruction of the interrupt service routine. All addresses of the $8 + 6 = 14$ interrupt service routines are stored in a table with a fixed placement in the memory space. In the BSP this table is placed from address \$0000h to \$000Dh, as shown on Table 8.1.

1st ext.	\$0000h
2nd ext.	
3rd ext.	\$0002h
4th ext.	
5th ext.	\$0004h
6th ext.	
7th ext.	\$0006h
8th ext.	
trace	\$0008h
overflow i bs rec.	
bs receiver	\$000Ah
serial	
overflow	\$000Ch
fifo	

Table 8.1: *Memory layout for address \$0000h to \$000Dh. These memory cells hold the address of each of the 14 interrupt service routines.*

8.5 Serial Port

In this section the serial port within the BSP will be presented. The serial port designed will be designed toward use with both the ADSP 21xx and TMS320C3x processor series. Though, the term serial port is used it must be noted that the BSP serial port only supports transmission of data. In the following a discussion on the requirements to put on the serial port will be conducted.

Serial ports placed on-chip are conventionally placed outside the core. A reasonable argument for this design choice is that a serial port is an optional feature and therefore is not allowed to add more complexity to the processor core. All access to the serial port is conducted using the data bus, and therefore an address within the memory space is assigned the serial port. Writing to a serial port in this configuration will not be different from writing to memory, from the programmers point of view.

In Chapter 6 it was stated that the Layer III application will require to transmit approximately 192,000 words per second, i.e. 3.1 Mbit/s. In order to obtain a high utilization level and thereby keep the bitrate reasonably low at the serial port, an interrupt was reserved for the serial port in Section 8.4. Every time the serial port is ready to transmit, a new word can be requested by setting the interrupt. An interrupt routine writing the new word to the serial port will also be required. An alternative software solution will be only to allow the programmer to poll the serial port. This solution was rejected, because polling is time inefficient when transmitting larger amounts of data.

The complexity analysis in Chapter 7 is based on a usage of approximately four instructions per word to transmit. If the interrupt service routine only writes one word to the serial port for every interrupt, this assumption cannot be met. The overhead introduced by executing an interrupt and setting up pointers for data to be transmitted will exceed the four instructions estimate. An alternative solution to the interrupt driven software solution would be to utilize a DMA controller. In short the DMA controller is an autonomous unit capable of claiming both the databus and addressbus to load data directly from memory. Though this solution will be appropriate, it is rejected because extra complexity is introduced in the bus arbitration, and a move away from the time shared processor core thereby will be taken.

A second solution is to equip the serial port with a buffer and thereby allow the interrupt service routine to write more words at a time. Increasing the buffer size will decrease the average instruction usage per word to be transmitted. Using a buffer size of 16 words has shown to yield an usage of approximately four instructions per word to transmit. Alternatively, a DMA controller could be used for loading the serial port.

Both the ADSP 21xx and TMS320C3x support word wise synchronous serial communication. The transfer mode chosen for the BSP serial port is referred to as Variable Burst Mode for the TMS320C3x and Alternate Framing for the ADSP 21xx. Both DSP's allow for an externally generated synchronization clock, which will be left for the BSP to generate.

In Figure 8.30 the timing diagram for the transmission mode is illustrated. The key notes for this mode is that the synchronization frame is set high during transmission of data and low elsewhere. Also, the data must be stable during negative edges on the synchronization clock.

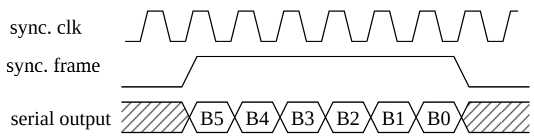


Figure 8.30: *Protocol for serial transmission with the BSP serial port. In the illustration 6 bits are transmitted, B0 first.*

8.5.1 Design of the Serial Port

Having sketched the functionality of the serial port an actual design will be accomplished. In short the serial port will require a 16 word buffer, a shift register for serial transmission, a ripple counter for generation of a synchronization clock and a controller. In Figure 8.31 the serial port design is illustrated. A description of the individual elements is conducted in the following paragraphs.

fifo buffer: The buffer has been implemented as a fifo buffer. Thereby data can be written to the serial port using only one port address, and data can be read from the buffer using no mux to select from the 16 data words.

shift register: The shift register performs the serial transmission by right shifting the register contents towards msb. The shift register is clocked by the synchronization clock, and the mode of operation is controlled by the serial controller, i.e. whether to right shift or read the next entry in the fifo buffer.

serial controller: The serial controller produces the control signals to the fifo buffer and the shift register. When data are transmitted the serial controller is also raising the synchronization frame.

In the serial controller, a programmable clock generator has been included. First of all, this extra feature is included in order to provide a synchronization clock. Using a second clock different from the system clock, i.e. clock serial in Figure 8.31, as input to the clock generator a bitrate can be obtained making the serial port compatible with asynchronous serial ports present in PC's. The clock generator allows division of clock serial by 2^n where n can attain integer values from 0 to 15, see Equation (8.3). Programming the clock is conducted by writing a 4 bit value to the fifo register using a port address different from the fifo buffer address.

$$f_{\text{synchronization}} = f_{\text{serial}} \times 2^{-n}, \quad n = 0, 1, 2, ..., 15 \tag{8.3}$$

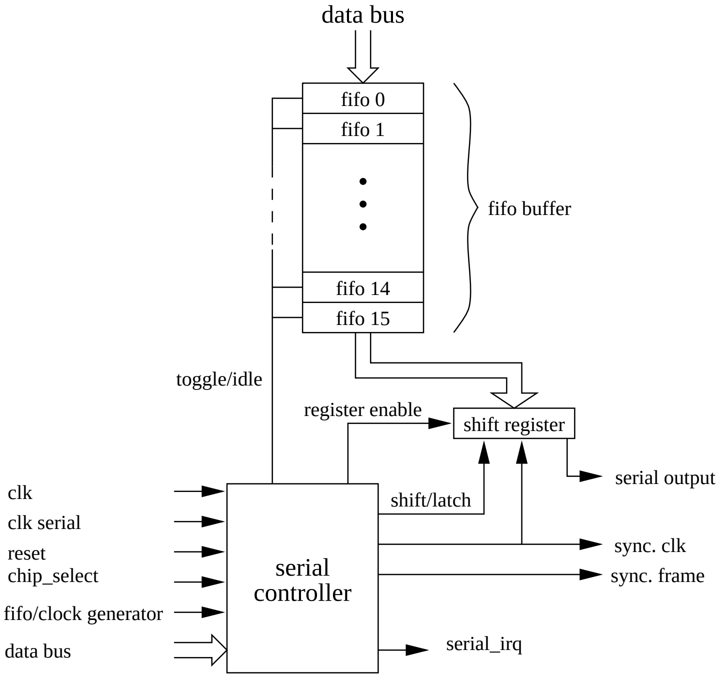


Figure 8.31: *Design of the serial port*

The timing of the control signals is managed by two state machines within the serial controller. The first state machine, called master, is driven by the processor clock and therefore only controls the fifo buffer and the interrupt request. The second, called slave, is driven by the synchronization clock and therefore controls the shift register and the synchronization frame. Both state machines are illustrated in Figure 8.32. In the following the individual states in the master state machine will be explained.

Ready When a processor reset arise, the master controller jumps into this state. Also, the programmable clock generator is set to clock serial to ensure that also the slave state machine can reset. When entering this state the interrupt request is made. The state machine jumps to the state Wait on Data on the next positive clock edge.

Wait on Data: In this state the master state machine is waiting for 16 data words to arrive. When the last word has arrived the master state machines sets a flag to indicate that the fifo is full and then jumps into the state Wait on Slave.

Wait on Slave: In this state the master is monitoring the slower slave state machine. When the slave enters the state Register Latch, a transition into the state Toggle Fifo is made. When all 16 words within the fifo buffer has been read by the shift register, the flag indicating the fifo buffer status is set to empty and the state machine jumps into the state Ready.

Toggle Fifo: When the master is in this state the slave is still being monitored. When the slave has performed a transition back to Shift Register, the master jumps back into Wait on Slave. At the transition the fifo contents is toggled once. Controlling the fifo buffer in this manner ensures that the fifo buffer only toggles the shift register after the next content to be transmitted has been read.

The status flag indicating whether the fifo buffer is full or empty is the only input the slave state machine responds to. In the following the individual states in the slave will be explained.

Register Idle: When reset is applied the slave state machine jumps into this state. Here it is waiting while watching the fifo buffer state. When the fifo buffer is full, a transition into the state Register Latch is performed.

Register Latch: In this state the contents of the last entry in the fifo buffer is read into the shift register. If the last state in the slave was idle register a transition into the state register shift is conducted.

Register Shift: In this state the contents of the shift register is shifted 15 times and the synchronization frame is set high. When 14 shifts have been accomplished the status flag for the fifo buffer is tested. If the fifo buffer is empty the remaining shift is conducted and the slave performs a transition into the state idle register. If the fifo buffer is full the last shift is conducted while a transition into the state Register Latch is performed. When entering the state Register Latch from Register Shift the synchronization frame must remain high. This requirement ensures a completely continuous and thereby faster serial transmission.

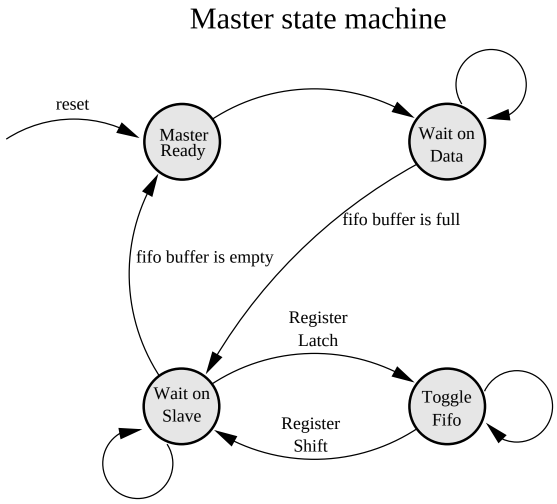


Figure 8.32: *Illustration of the master and slave state machines within the serial controller.*

8.6 Address Decoding

In Chapter 7 it was decided use memory–mapped I/O and hence some address decoding is necessary.

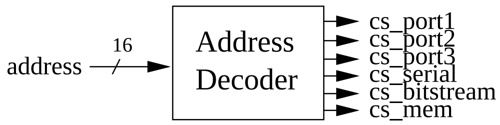


Figure 8.33: *A overview of the address decoder, which on the basis of the 16–bit input address generates chip select signals to all the memory–mapped units. Full address decoding is used.*

The purpose of the address decoding is, on the basis of the address written on the addressbus, to generate chip select signals to the different I/O units. The I/O units that are memory mapped are: the Bitstream Receiver, the serial interface to the DSP and the three user defined ports. It has been chosen to allocate 16 for each unit. The reason for choosing 16 addresses is to provide for some flexibility. For instance a UART 16550 has 8 internal registers that can be programmed, to get a desired functionality. By using full address decoding it is easy to expand with other future units. The addresses that generates chip selects for the I/O units is shown in Table 8.1.

Unit	Address
CS_port1	FFFF ⋮ FFF0
CS_port2	FFEF ⋮ FFE0
CS_port3	FFDF ⋮ FFD0
CS_serial	FFCF ⋮ FFC0
CS_bitstream	FFBF ⋮ FFB0
CS_mem	FFAF ⋮ 0000

Table 8.1: *The addresses allocated for the individual I/O units.*

8.7 Summary of the hardware design

The preceding sections, presented the design of the individual blocks in the BSP. To put the BSP in perspective as a self-contained processor, a summary on the BSP and the features offered will be conducted. All the features presented will be based on the capabilities of the processor only. The limitations usually imposed by the instruction set, e.g. restricted immediate value range, will not be taken into consideration at this point.

Figure 8.34 illustrates the architecture of the BSP designed. In the architecture tri-state drivers have been included for proper sharing of the databus between the datapath, memory and the bitstream receiver's status word. In the following the features implemented in the individual blocks in Figure 8.34 will be summarized

Datapath: The BSP is based on a traditional microprocessor core offering a 16 bit databus and a 16 bit addressbus. The datapath contains an ALU supporting addition, subtraction, AND, OR, XOR, and arithmetic left and right shifts. Detection of the status after an ALU operation has been provided. When the result of an ALU operation is zero or less than zero, it is detected and saved in a status register. If the ALU operation causes an overflow or underflow an interrupt is generated. The datapath supports all the arithmetic and logical operations for 10 general purpose registers and a immediate value, which enables the use

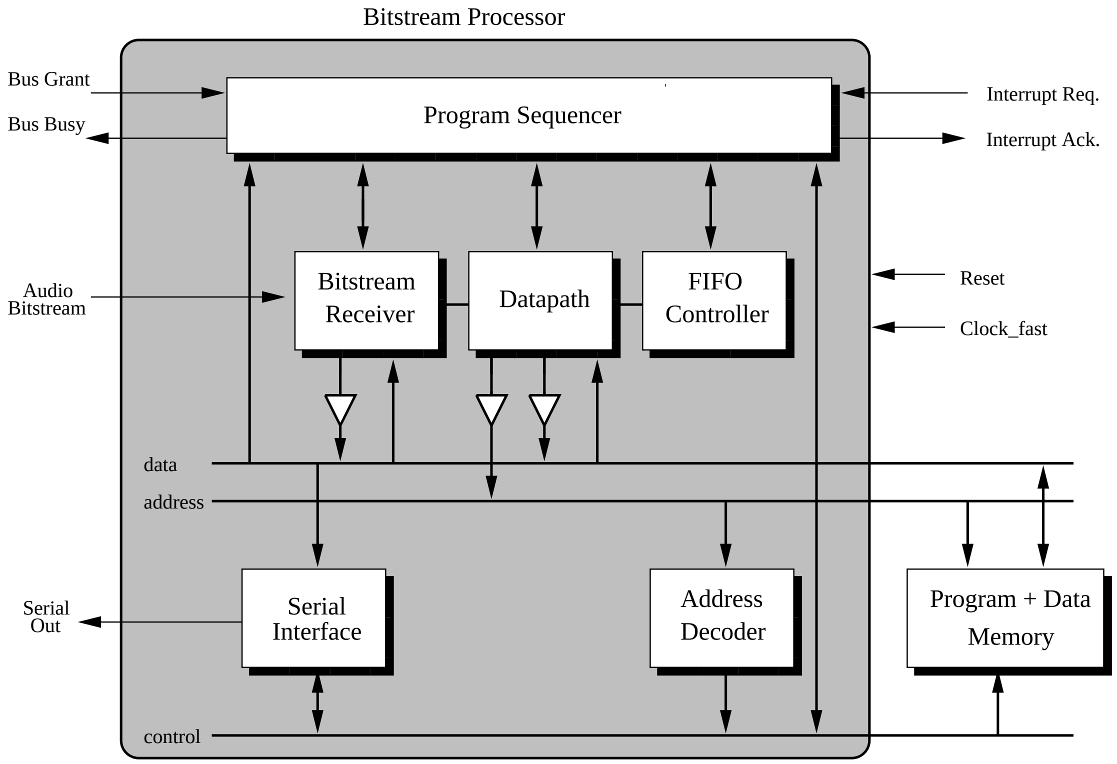


Figure 8.34: *The proposed architecture for a Layer III BSP*

of both register-register and register-immediate operations. The registers in the datapath are loaded using either an immediate operand or indirectly using a register for addressing an operand in memory. Using a stack for the latter addressing mode is supported by indivisible push and pop operations.

Program Sequencer: Program modifications are supported in terms of procedure calls, jumps and conditional jumps. The conditional jump is implemented in conjunction with the ALU status which leads to the operations jump less than zero and jump not zero. The BSP supports prioritized nested interrupts. Six internal and one external interrupt are offered. The external interrupt has been implemented as a vectored interrupt, which enables the use of multiple interrupt service.

The processor has been equipped with two Layer III specific features, a bitstream receiver and a fifo register. The bitstream receiver constitutes a serial interface to the incoming bitstream.

Bitstream Receiver: An 96 bit register has been applied for buffering the incoming bitstream. When the buffer is full an interrupt is set to indicate the presence of 96 bit data, ready to be read. The bitstream receiver can be programmed to search for a synchronization word. The programming is conducted by writing to a memory mapped port on the bitstream receiver. When a synchronization word is found, a status register will apply information on the synchronization word occurrence and the number of valid bytes ready to be read. The port to read data from is mapped into the register file.

Fifo Register: The fifo register has been included in the processor core to provide the programmer with instant access to an arbitrary number of bits from maindata. Thereby, a very simple management of data accessed bit-wise is introduced.

Serial Port: The BSP is equipped with an internal serial port with programmable bitrate. A

16 word buffer has been included to keep down the total instruction overhead when writing data to the serial port.

Address Decoder: The address decoder is used for selecting the memory mapped ports. Besides providing signals for selecting the internally placed blocks, three external ports are provided for optionally use.

At this stage the BSP will require an instruction set and the corresponding instruction decoder, in order to function properly. The development of both parts is a co-design process in which it is attempted to utilize the processor capabilities as well as possible. In the following chapter the design of the instruction set will be conducted.

9

BSP Assembly Language

This chapter is divided into two sections. The first describes the format of the instruction set, and the second describes the assembly language constructed, which is used to program the BSP.

9.1 The Instruction Set of the BSP

A program consists of a sequence of instructions, each specifying some particular action to be performed in the processor. A part of an instruction is used for opcode and a part for data. The opcode part is used by the program sequencer to detect the type of operation to perform, and when the type of operation is known, the program sequencer also known the format of the data part of the instruction code.

When designing an instruction set, a number of factors must be considered. Short instructions are better that long instructions. By having long instructions the amount of data needed to be transfered from memory to program sequencer increases, and is likely to become the bottleneck in the processor, thus setting an upper limit on the program execution speed. By having short instructions, e.g. 16-bit instructions on an 16-bit architecture, it is possible to load an instruction in a single memory access and thereby gaining a faster processor. While short instructions is desirable for minimizing the amount of data to be transfered from memory to program sequencer, they must have sufficient room for expressing the operations to perform. If for example a processor has 16 internal registers, and an operation requires to select three of these registers simultaneous – then the data part of the opcode will require $3 \cdot 2^4 = 12$ bits, leaving only 4 bits for the opcode. Another factor when designing the instruction set is simplicity. If the opcodes are compressed very efficiently they will typically set higher demands to the decoding in the program sequencer and thus increase the complexity.

It was desired to make a 16 bit wide instruction set for the BSP. The fact that the datapath and the busses between the BSP and memory are 16-bit wide makes 16 bit instruction set desirable. By setting the instruction width to 16 bit, some restrictions are imposed upon the instruction, e.g. immediate values are often restricted to a rather narrow range. But it is appraised that the instruction overhead introduced by the limited range of the immediate values, is of less consequence compared to the cost of making longer instructions. The instruction set of the BSP is designed with an expanding opcode, meaning that instructions which demand many bits as argument, have short opcodes, and instructions with low demands on argument bits have long opcodes. By using expanding opcodes it is possible to make trade-offs between assigning bits to arguments or opcode.

Making the actually instruction set of a microprocessor is like a big puzzle, where different combinations are tried, until a reasonable result is obtained. This design process is not described in this chapter, but instead the initial demands to the instruction set are given together with result of the design process.

Operational Demands to the Instruction Set

The following operations are needed in order to control the datapath

Arithmetic Operations: The BSP must be able to perform the arithmetic operations, add, subtract and left/right shift. These operations are performed between two source registers, and the result is store in a destination register.

Arithmetic Operations with Immediate Values: The BSP must be able to perform arithmetic operations on registers where one of the arguments is an immediate value. The range of the immediate value when performing left and right shift operations is between 0 and 15. The range of the immediate value when performing additions and subtractions does not have as specific requirements, but are to be maximized.

Logical Operations: The BSP must be able to perform AND, OR and XOR operations on registers in the same manner as the arithmetic operations.

Load/Store: It must be possible to load data from memory into an internal register. The address to be loaded is specified as the sum of two address, e.g. load(R1+R2). The store operation must have same addressing abilities as the load operation.

Load/Store with Immediate Values: It must be possible to specify an immediate value as one of the arguments in a load operation. The store operation must have same addressing abilities as the load operation.

Set: The set operation is necessary to write immediate constants into the internal registers.

Jump: It must be possible to perform both unconditional and conditional jumps in the BSP.

Call: The BSP must be able to perform unconditional calls. This operation is used for procedure calls.

Fifo Operations: The BSP must have some operations specially dedicated for handling the fifo register. These operations are denoted popfifo(arg) and resetfifo. The popfifo(arg) operation reads the number of bits specified by the argument into a register, and afterwards it shifts the recently read bits out of the fifo register. The argument can either be a register or an immediate value. The resetfifo operation is used for reseting the fifo controller.

Stack Operations: The BSP also needs operations for pushing and popping the contents of registers onto the stack. A push operation writes the contents of the specified register to be pushed onto the stack and updates the stack pointer. Pop updates the stack pointer and reads the newest value on the stack into a register. From the operations previously described, the BSP is actually capable of performing these operations in software. For example a push operation can be done by a store operation relative to the stack pointer and the stack pointer can afterwards be updated with an immediate addition. But the push operation must be an indivisible operation which interrupts cannot disturb between data has be written to the stack and the stack pointer has be updated. This is the reason for making specific push/pop operations.

The Instruction Set of the BSP

The demands listed in the previous section together with the goal of making a 16 bit instruction set led to the instruction set shown in Table 9.1. The “binary” column describes the format of the instruction, where the part written in boldface is the opcode part. The argument part of the instruction is divided into sections where 'x', 'y' and 'z' symbolizes register indexes and 'i' symbolizes that the space is reserved for immediate values. Don't-cares are denoted by '-'. The rightmost column describes the range of the immediate value, when immediate values are applied.

Instruction set						
Binary				Mnemonic	Meaning	range of imm. value
0000	xxxx	yyyy	zzzz	add	$r_x = r_y + r_z$	
0001	xxxx	yyyy	zzzz	sub	$r_x = r_y - r_z$	
0010	xxxx	yyyy	zzzz	and	$r_x = r_y$ AND r_z	
0011	xxxx	yyyy	zzzz	or	$r_x = r_y$ OR r_z	
0100	xxxx	yyyy	zzzz	xor	$r_x = r_y$ XOR r_z	
0101	xxxx	yyyy	zzzz	sll	$r_x = r_y << r_z$	
0110	xxxx	yyyy	zzzz	slr	$r_x = r_y >> r_z$	
0111	xxxx	yyyy	zzzz	load	$r_x = \text{mem}(r_y + r_z)$	
1000	xxxx	yyyy	zzzz	store	$\text{mem}(r_y + r_z) = r_x$	
1001	xxxx	yyyy	iiii	slli	$r_x = r_y << \textit{immediate}$	$[0;15]$
1010	xxxx	yyyy	iiii	slri	$r_x = r_y >> \textit{immediate}$	$[0;15]$
1011	xxxx	yiii	iiii	addi	$r_y = r_x + \textit{immediate}$	$[-63;63]$
1100	xxxx	yiii	iiii	loadi	$r_y = \text{mem}(r_x + \textit{immediate})$	$[-63;63]$
1101	xxxx	yiii	iiii	storei	$\text{mem}(r_x + \textit{immediate}) = r_y$	$[-63;63]$
1110	xxxx	iiii	iiii	set	$r_x = \textit{immediate}$	$[0;255]$
1111	**000**i	iiii	iiii	jump	$PC = PC + \textit{immediate}$	$[-256;255]$
···	**001**i	iiii	iiii	jumplz	if lz then; $PC = PC + \textit{immediate}$	$[-256;255]$
···	**010**i	iiii	iiii	jumpnz	if nz then; $PC = PC + \textit{immediate}$	$[-256;255]$
···	**011**i	iiii	iiii	call	$PC = PC + \textit{immediate}$	$[-256;255]$
···	**1000**	yyyy	zzzz	readfifo	$r_y = \text{readfifo}(r_z)$	
···	· **001**	yyyy	iiii	readfifoi	$r_y = \text{readfifoi}(\textit{immediate})$	$[0;15]$
···	· **010**	yyyy	zzzz	popfifo	$r_y = \text{popfifo}(r_z)$	
···	· **011**	yyyy	iiii	popfifoi	$r_y = \text{popfifoi}(\textit{immediate})$	$[0;15]$
···	· **100**	**00**ii	zzzz	mask1++	$r_z = (r_z + 1)\%(64, 128, 256, 512)$	$[0;3]$
···	· · **00**	**01**ii	zzzz	mask2++	$r_z = (r_z + 1)\%(1k, 2k, 4k, 8k)$	$[0;3]$
···	· · **00**	**10**--	zzzz	push	$\text{push}(r_z)$	
···	· · **00**	**11**--	zzzz	pop	$\text{pop}(r_z)$	
···	· · **01**	**00**--	zzzz	inc	$r_z = r_z + 1$	
···	· · **01**	**01**--	zzzz	dec	$r_z = r_z - 1$	
···	· · **01**	**10**--	----	resetfifo	resetfifo	
···	· · **01**	**11**--	----	nop	no operation	
···	· · **10**	**00**--	----	rti	return from interrupt	
···	· · **10**	**01**--	----	lcall	PC=mem(PC), PC=PC+1	

Table 9.1: *Instruction set of the BSP.*

The first 11 instruction on Table 9.1 are straight forward. Because of the limited range addi, loadi and storei are restricted only to operate on two of the registers. These registers are denoted g-registers. This restriction is not at all desirable, but is necessary in order to maximize the range of the immediate value. By choosing only two registers, it is possible to have an immediate value in the range from -63 to 63. As a result of not being able to add immediate into all registers, it is not possible to use others than the two g-registers as loop counters. To mend this, an increment and decrement command is added, which is able to increase/decrease the contents of all registers. The jump and call operations are restricted only to be capable of jump 255 addresses back or forth in the program. This does not result in any problem in most cases, but to manage the cases where it does, a lcall operation is added. The argument of the lcall operation is stored on the succeeding address in memory, and is therefore able to make a call to any address in the program memory. This is the only operation requiring 2×16 bits.

Another operation not yet introduced is the mask operation. This operation is specially designed for incrementing pointers in circular buffers. The mask operation is given two arguments, the register to perform the operation on, and the size of the circular buffer. The mask operation increments the register and then truncates the result to stay within the buffer size. An example of how the mask operation works is given in the following example.

A circular buffer with 256 words is placed from address 2048 to $2048 + 255$. The register indexing this buffer is incremented by using mask(reg,256). Each time the mask(reg,256) is executed the register is incremented by one, and bit-wise AND'ed with $2^{16} - 1 - (2 \cdot 256)$. When the register reaches $2048 + 255$, the following happens.

```
Register:          0000 0100 1111 1111  ; Address 2048+255
Register+1:        0000 0101 0000 0000  ; Address 2048+256
Truncation mask:   1111 1110 1111 1111  ; Buffsize=256
Result:            0000 0100 0000 0000  ; Address 2048+  0
```

The truncation mask keeps the buffer pointer within the circular buffer. By implementing circular buffers in this manner, certain restrictions are imposed on the size and the base address of the circular buffer. The size of the circular buffer must be a power of 2. If the size of the buffer is b, the the buffer can only be placed at the following base addresses:

$$base\ adr = (2b)n, \qquad n \in \{1, 2, \ldots\} \tag{9.1}$$

Because of the limited number of bits in the instruction only the following buffer sizes are possible: 64, 128, 256, 512, 1024, 2048, 4096, 8192 words.

The Rti instruction is used when returning from interrupts. Besides popping the program counter, the status register is also restored.

9.2 Assembler

Handling instructions as sequences of binary numbers is natural and efficient on computers. Humans, however, is more comfortable when instructions are written with symbols and labels. This is basical the reason for spending time and effort on making an assembler to the BSP. If the algorithms to be implemented were simple algorithms with only a few variables and conditional branches, it might have been more efficient to have invested the man-hours otherwise, but as the Layer III algorithms are large and require many temporary variables and conditional branches, making an assembler is evidently a good idea.

Instead of making a traditional assembler, where the instructions are closely matched to the actual instruction opcode, it has been chosen to make an algebraic assembler, meaning that the syntax in the assembler matches to the way algorithms normally are written. An example of the differences between traditional assemblers and an algebraic assembler is given by the following:

Traditional Assembler	Algebraic Assembler	Comment
ADD R1,R2,R3	R1 = R2 + R3	Add R2 and R3 and place result in R1
SLR R1,R2,2	R1 = R2 « 2	Shift logically R2 to bits to the left and place result in R1
LDA R3,[R1,R2]	R3 = mem(R1+R2)	load the contents of address R1+R2 into R3
STO [R1,R2],R3	mem(R1+R2)=R3	Store the contents of R3 into address R1+R2

It is clear from these examples that the syntax of an algebraic assembler is much closer how algorithms are normally written.

The following section contains a short description of the features and commands implemented in the assembler. The chapter concludes with a few examples of how an assembly program can be written. This chapter does not serve as a user manual of how to use the assembler, for information about this see the enclosed folder "User's Manual for the BSP Assembler". For a description of the assembler in Bachus Naur Form see Appendix F.

9.3 Features in the Assembler

General : All operations are written in algebraic form, as illustrated by following examples

```
G0 = 100       ; set G0 equal 100
R0 = G1 and R2 ; Bit-wise and of G1 and R2. Result stored in R0
R0 = R1 << R2  ; Shift R1 with contents of R2 and store in R0
R0 = R1 + R2   ; Add R1 to R2 and place result in R0
G0 = R1 - 10   ; Subtract 10 from R1 and store in G0
```

The memory is indexed as one large vector denoted *mem*(), where load and store is distinguish by whether the *mem*() is placed to the left or the right in the expression.

```
R1 = mem(PC)       ; Load next instruction into R1
mem(R2) = R1       ; Store contents of R1 on address R2
R2 = mem(R0 + R1)  ; Load the contents at address R0+R1 into R2
G0 = mem(SP - 2)   ; Load the contents at address SP-2 into G0
mem(R1 + 10) = G0  ; Store G0 at address R1+10
```

The programmer need not to distinguish between register-register operations and operations including immediate values. The operator used for register-register operations is the same as the operator used when operating on immediate values, because the assembler is able to detect whether it is an immediate operation out of the context.

Labels: It is allowed to perform jumps to labels, thereby leaving it to the assembler to determine the exact address. Examples of how to use labels are given in the following:

```
          R0 = R1 - R0        ; Update status flags in alu
          jumpnz OneBiggest   ; If R1>R0 jump to OneBiggest
          jump ZeroBiggest    ; If R0>=R1 jump to ZeroBiggest
OneBiggest:
          ;; instructions to perform if R1>R0
ZeroBiggest:
          ;; instructions to perform if R0>=R1
```

In the same manner it is possible to load variables placed in the assembly code.

```
var:      .word 10        ; This address is reserved for a variable
                          ; and is initial set to 10
          G0 = mem(var)   ; Load var into G0
          G0 = G0 << 1    ; Left shift G0 one bit
          mem(var) = G0   ; Store the result in var
```

These references will automatically be substituted with references relative to PC, e.g. the first reference will be substituted with `mem(PC-2)`, and the second reference will be substituted with `mem(PC-4)`. Note that the program counter has already been incremented to point at the following instruction when the references are made.

Include files: The assembler is capable of including files into an assembly file. By this feature, it is possible to make a more structured design where different procedures and tables used in the program are included from separate files.

```
#include filename        ; include file named 'filename.asm' into code
```

Number formats: The assembler is able to handle both hexadecimal and decimal numbers. The hexadecimal numbers are recognized by a leading $.

In-line calculations: It is possible to express constants as expressions, e.g. the bit pattern '1100' can be expressed as:

```
R0 = (2^2)+(2^3)      ; Examples of inline calculations
R0 = ($F)-3           ; "
R0 = ((2^3)+(2^4))/2  ; "
```

Constants: The assembler is able to handle constants, which can be used throughout the assembly program. For example if the labels `maxcount` is defined to the constant 10, the all occurrences of `maxcount` throughout the assembly code will be substituted with 10 when translated.

```
#define maxcount = 10
    R0 = maxcount         ; constant used directly
    R0 = maxcount-1       ; constant used in expression
    R0 = mem(R1-maxcount) ; constant used as offset in load operation
```

Huffman: The assembler is provided with three additional commands for generating the Huffman tables. The three commands are: #huftable, jumpnode() and endnode().

Syntactic and Semantic check: The assembler has both syntactic and semantic error check. The syntactic check verifies that each instruction is syntactic correct with respect to the defined BNF statements, see Appendix F. The semantic check verifies that the range of the immediate value is within the specified range for the individual instructions.

9.4 Assembly Instructions

In this section, each of the instructions available in the assembler will be described. The purpose of this description is to establish a foundation for describing the software in the succeeding chapter. Each of the assembly instructions are described with a near BNF syntax, where $<$ *register* $>$ is an abbreviation for all registers in the BSP, and $<$ *g-register* $>$ is a abbreviation for G0 and G1. Immediate values are denoted as the interval, in which the immediate value must be. For a precise definition of the syntax for an assembly program written in BNF, see Appendix F.

Addition

Additions can be conducted between registers only, and between registers and immediate values. When addition is done between a register and an immediate value, the destination register must be G0 or G1.

$$
\begin{array}{lllllll}
\text{Addition} := & & <\textit{register}> & = & <\textit{register}> & + & <\textit{register}> \\
& | & <\textit{g-register}> & = & <\textit{register}> & + & [-63;63]
\end{array}
$$

Subtraction

Subtractions can only be conducted between registers.

$$
\text{Subtraction} := \quad <\textit{register}> \; = \; <\textit{register}> \; - \; <\textit{register}>
$$

Shift left

Shift the contents of the register towards MSB, and insert zeros on LSB.

$$
\begin{array}{lllllll}
\text{Shift left} := & & <\textit{register}> & = & <\textit{register}> & \ll & <\textit{register}> \\
& | & <\textit{register}> & = & <\textit{register}> & \ll & [0;15]
\end{array}
$$

Shift right

Shift the contents of the register towards LSB, and insert zeros on MSB.

$$
\begin{array}{lllllll}
\text{Shift right} := & & <\textit{register}> & = & <\textit{register}> & \gg & <\textit{register}> \\
& | & <\textit{register}> & = & <\textit{register}> & \gg & [0;15]
\end{array}
$$

Logic operations

Logic operations are written as follows, where $<$ *logic operator* $>$ is a abbreviation for AND,OR and XOR.

$$
\text{Logic} := \quad <\textit{register}> \; = \; <\textit{register}> \; <\textit{logic operator}> \; <\textit{register}>
$$

Set

The set operation is used to load a constant into a register. The constant must be in the range between 0 and 255.

$$
\text{Set} := \quad <\textit{register}> \; = \; [0;255]
$$

Load

Data can be loaded from memory into registers by using the mem() instruction. If mem() is given one register argument, it loads the memory cell using the register as address. If it is given two registers as arguments, the contents of the registers are added, and the result is used as an address. If the arguments are a register and an immediate value, the immediate values are added to the contents of the register and the result is used as an address. When using an immediate value in the mem() operation, the destination must be either G0 or G1.

$$\begin{array}{lllll} \text{Load} := & <\mathit{register}> & = & \text{mem}(<\mathit{register}> \; + \; <\mathit{register}>) \\ | & <\mathit{register}> & = & \text{mem}(<\mathit{register}>) \\ | & <\mathit{g\text{-}register}> & = & \text{mem}(<\mathit{register}> \; + \; [-255;255]) \end{array}$$

Store

Data can be stored from a register into memory by using the same mem() instruction, as in the load operation. The only difference is that the mem() operation is used on the left side of the equality sign.

$$\begin{array}{llll} \text{Store} := & \text{mem}(<\mathit{register}> \; + \; <\mathit{register}>) & = & <\mathit{register}> \\ | & \text{mem}(<\mathit{register}>) & = & <\mathit{register}> \\ | & \text{mem}(<\mathit{register}> \; + \; [-255;255]) & = & <\mathit{g\text{-}register}> \end{array}$$

Jump and Call

Both jumps, unconditional jumps and calls are done by specifying the label of the destination address. All jumps are performed relatively to the program counter, meaning that the assembler substitutes the label with the relative offset to the address where the label is defined. This relative offset must be in the range between -255 and 255. If the offset exceed this range, the assembler reports an error, and it is up to the programmer to fix the problem.

$$\begin{array}{ll} \text{Jump} := & \text{jump } \mathit{label} \\ | & \text{jumplz } \mathit{label} \\ | & \text{jumpnz } \mathit{label} \end{array}$$

$$\begin{array}{ll} \text{Call} := & \text{call } \mathit{label} \\ | & \text{lcall } \mathit{label} \end{array}$$

Fifo operations

The assembler has four instruction specially dedicated for operating on the fifo register. Actually the BSP only has two instructions specially designed for handling the fifo, namely popfifo() and resetfifo. The other instructions, shiftfifo() and readfifo() is substituted with popfifo() by the assembler. The only difference between shiftfifo() and popfifo() is that shiftfifo() does not return an argument. The difference between popfifo() and readfifo() is that readfifo() does not update the fifo register. Shiftfifo(), readfifo() and popfifo() are given either a register or an immediate value as argument. This argument is used to determine the number of bits to read from the fifo register. Because of the limited number of bits in the opcode, this value is encoded as "the number of bits to read" minus one, i.e. popfifo(0) reads one bit and popfifo(15) reads sixteen bits.

$$\begin{array}{llll} \text{Popfifo} := & <\mathit{register}> & = & \text{popfifo}(<\mathit{register}>) \\ | & <\mathit{register}> & = & \text{popfifo}([0;15]) \end{array}$$

$$\begin{array}{lrcl} \text{Readfifo} := & < register > & = & \text{readfifo}(< register >) \\ | & < register > & = & \text{readfifo}([0;15]) \end{array}$$

$$\begin{array}{ll} \text{Shiftfifo} := & \text{shiftfifo}(< register >) \\ | & \text{shiftfifo}([0;15]) \end{array}$$

resetfifo := resetfifo

Mask

The mask instruction is specially designed for handling circular buffers. Although both mask instructions exist in the instruction set, these are hided behind one instruction in the assembler.

$$\text{Mask} := \quad \text{mask}(< register >, \{64, 128, 256, 512, 1024, 2048, 4096, 8192\})$$

Stack Operations

These two instructions are used for pushing and popping registers to and from the stack.

$$\begin{array}{lrcl} \text{Stack} := & < register > & = & \text{pop}(< register >) \\ | & \text{push}(< register >) & & \end{array}$$

Counter Operations

The BSP is equipped with two operations specially designed for increasing and decreasing the contents of a register. The operations can for example be used to update loop counters.

$$\begin{array}{ll} \text{Counter} := & < register > ++ \\ | & < register > -- \end{array}$$

Return

The assembler has two instructions for terminating normal program calls, and interrupt routines. Only the Rti instruction is implemented directly supported by the BSP. The Rtn instruction is substituted by PC=pop().

$$\begin{array}{ll} \text{Return} := & \text{rtn} \\ | & \text{rti} \end{array}$$

Directives		Z	LZ	CC
#define	#define *label*=*constants*	–	–	–
#include	#include *filename*	–	–	–
#set address	#set address = *constant*	–	–	–
.word	.word	–	–	–
Instructions		Z	LZ	CC
register copy	*reg*=*reg*	•	•	3
and	*reg* = *reg* and *reg*	•	•	3
or	*reg* = *reg* or *reg*	•	•	3
xor	*reg* = *reg* xor *reg*	•	•	3
addition	*reg* = *reg* + *reg*	•	•	3
	reg = *reg* + [-63;63]	•	•	3
subtraction	*reg* = *reg* - *reg*	•	•	3
shift left	*reg* = *reg* « *reg*	•	•	3
	reg = *reg* « [0;15]	•	•	3
shift right	*reg* = *reg* » *reg*	•	•	3
	reg = *reg* » [0;15]	•	•	3
popfifo	*reg*=popfifo(*reg*)	–	–	3
	reg=popfifo([0;15])	–	–	3
readfifo	*reg*=readfifo(*reg*)	–	–	3
	reg=readfifo([0;15])	–	–	3
shiftfifo	shiftfifo(*reg*)	–	–	3
	shiftfifo([0;15])	–	–	3
resetfifo	resetfifo	–	–	3
push	push(*reg*)	–	–	5
pop	pop(*reg*)	–	–	5
mask	mask(reg++,{64,128,256,512,1k,2k,4k,8k })	–	–	4
jump	jump *label*	–	–	3
jumpnz	jumpnz *label*	–	–	3
jumplz	jumpmz *label*	–	–	3
call	call *label*	–	–	6
lcall	lcall *label*	–	–	6
rtn	rtn	–	–	5
rti	rti	–	–	7
increment	*reg*++	•	•	3
decrement	*reg*- -	•	•	3
nop	nop	–	–	3
Huffman		Z	LZ	CC
#hufftable	#hufftable	–	–	–
jumpnode	jumpnode({1,2,3,4},*label*)	–	–	–
endnode	endnode({1,2,3,4},[0;15],[0;15])	–	–	–

Table 9.2: *Summary of directives and instructions availeble in the BSP. Z and LZ indicates whether the status flags are updated. CC is the required clock cycles in order to execute the instruction*

9.5 Example of an Assembly Program

The following section demonstates how an assembly program can be written, and how the BSP assembler is used.

By translating a file with the BSP assembler, two files are generated – a symbolic file contaning the assemblers interpretation of the assembly program, and a binary file containing the binary code to be programmed into the BSP. An example of an assembly program, and its resulting files are demonstated in the following.

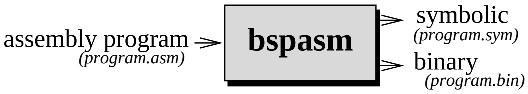


Figure 9.1: *BSP assembler*

Assembly program

The assembly program used in this example is a small routine used for calculating CRC-16 check words. The routine calculates the CRC check word byte-wise, using pre-calculated values placed in a table. This table is included as a file last in the assembly program. The assembly program is placed from address $\$1000_{hex}$ in memory.

```
;;;--------------------- Begin Bytewise CRC generation -------------------
;;; Subroutine crcb
;;; Calculates a CRC check word on data read on the fifo register.
;;; The routine take one argument in R0, specifying the number of bytes
;;; to check.
;;; The routine returns the CRC checkword in G0

#set address = $1000
crclut: .word crcLookupTable
        ;; Calling point for the crc calculation
crcb:
        push(r1)                ; store the state of the register
        push(r2)                ; "
        push(r3)                ; "

        g0 = mem(crclut)        ; Load pointer to crc table
        r1 = 0

crcloop:
        r3 = popfifo(7)         ; Read new byte
        r2 = r1 >> 8            ; 8 MSB's of CRC (x^15 + ... + X^8)
        r3 = r3 xor r2          ; Sum between CRC og new byte in binary field
        r2 = mem( g0 + r3 )     ;
        r1 = r1 << 8            ; s0*x^8+s1*x^9+...+s7*x^15
        r1 = r1 xor r2          ;
        r0--                    ; update counter
        jumpnz crcloop          ; if counter!=0 then jump
        g0 = r1                 ; move crc to g0

        r3 = pop()
        r2 = pop()
        r1 = pop()
        rtn

#include crc-lookup-tables
;;;---------------------- End Bytewise CRC generation --------------------
```

Symbolic file

The symbolic file contains the BSP assemblers interpretation of the assembly code. The hexadecimal values between square brackets are the addresses of the instructions. All label references are substituted with addresses relative to the program counter, e.g. in address $1004 is `crclut` substituted with `PC-5`.

```
;;;--------------------- Begin Bytewise CRC generation -------------------
;;; Subroutine crcb
;;; Calculates a CRC check word on data read on the fifo register.
;;; The routine take one argument in R0, specifying the number of bytes
;;; to check.
;;; The routine returns the CRC checkword in G0
#SET ADDRESS = 4096
crclut:
     [1000] .WORD    4115  [crcLookupTable]        ;; Calling point for the crc calculation
crcb:
     [1001]          PUSH ( R1 )                   ; store the state of the register
     [1002]          PUSH ( R2 )                   ; "
     [1003]          PUSH ( R3 )                   ; "
     [1004]          G0 = MEM(PC+-5 [ crclut ])    ; Load pointer to crc table
     [1005]          R1 = 0
crcloop:
     [1006]          R3 = POPFIFO ( 7 )            ; Read new byte
     [1007]          R2 = R1 >> 8                  ; 8 MSB's of CRC (x^15 + ... + X^8)
     [1008]          R3 = R3 XOR R2                ; Sum between CRC og new byte in binary field
     [1009]          R2 = MEM(G0 + R3)             ;
     [100A]          R1 = R1 << 8                  ; s0*x^8+s1*x^9+...+s7*x^15
     [100B]          R1 = R1 XOR R2                ;
     [100C]          R0--                          ; update counter
     [100D]          JUMPNZ PC + -8 [ crcloop ]    ; if counter!=0 then jump
     [100E]          G0 = R1                       ; move crc to g0
     [100F]          R3 = POP()
     [1010]          R2 = POP()
     [1011]          R1 = POP()
     [1012]          RTN
#INCLUDE crc-lookup-tables
crcLookupTable:
     [1013] .WORD    0
     [1014] .WORD    0                             ; end of include
;;;---------------------- End Bytewise CRC generation --------------------
```

Binary file

The binary code to be programmed into the BSP's memory.

```
0000000000010011
1111110010000011
1111110010000100
1111110010000101
1100110001000101
1110001100000000
1111101101010111
1010010000111000
0100010101010100
0111010000000101
1001001100111000
0100001100110100
1111110101000010
1111010100001000
0010000000110011
1111110011000101
1111110011000100
1111110011000011
1111110011001100
0000000000000000
0000000000000000
```

10

Software for Bitstream Decoding

This chapter provides a description of the assembler program used to decode an MPEG/Audio Layer III bitstream.

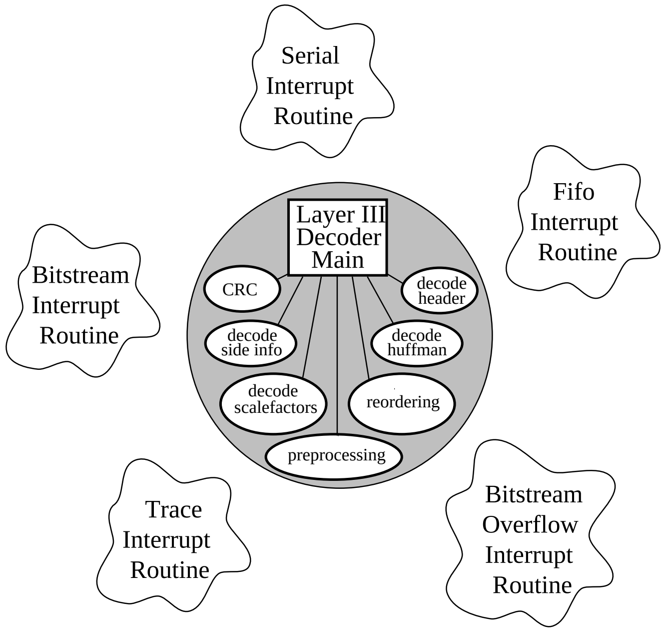


Figure 10.1: *An overview of the Layer III decoder program. The decoder consists of a main program and a number of interrupt routines.*

Figure 10.1 provides an overview of the software written for executing the BSP–part of the MPEG/Audio Layer III algorithm. The main program contains the main loop of the program. In addition a number of interrupt routines exist, six internal and up to eight external. One of these interrupt routines is executed whenever an interrupt occur. The processor facilitates prioritized nested interrupts. Three interrupts are enabled at startup of the main program, i.e. the overflow in incoming bitstream interrupt , the bitstream receiver interrupt and the fifo interrupt. The two first interrupts are received from the bitstream receiver. The bitstream receiver interrupt indicate that new data is ready in the outgoing buffer of the bitstream receiver. The interrupt routine Handle Incoming Bitstream is executed when this interrupt occur. The Handle Incoming Bitstream must be able to perform synchronization and save the received data byte-aligned in the memory. How the data belonging to the individual frames can be organized in memory is considered next.

Buffering: The data received from the bitstream receiver is stored in a circular buffer in memory. The buffer must be large enough to hold the data belonging to two frames and to ensure that data is not overwritten before it has been used. The maximum size of a frame transmitted at 320 kbit/s is 960 bytes. A circular buffer of 2K bytes therefore seems reasonable. However it is convenient to place the header+side information and main data in separate buffers. This is due to the bit reservoir technique used in the Layer III algorithm. In short, the bit reservoir makes it possible to "borrow" unused bits from the previous frames. Hence a 9–bit `main_data_begin` pointer was introduced, pointing back from the Header start of the actual frame to the beginning byte of the main data belonging to the frame. The `main_data_begin` pointer does not include bytes consumed by the previous headers and side information. Placing the entire frames consecutively in the same circular buffer will require some unnecessary and unpleasant calculations. By placing the header and side information and the main data in separate buffers, the `main_data_begin` pointer can be used directly without any extra calculations.

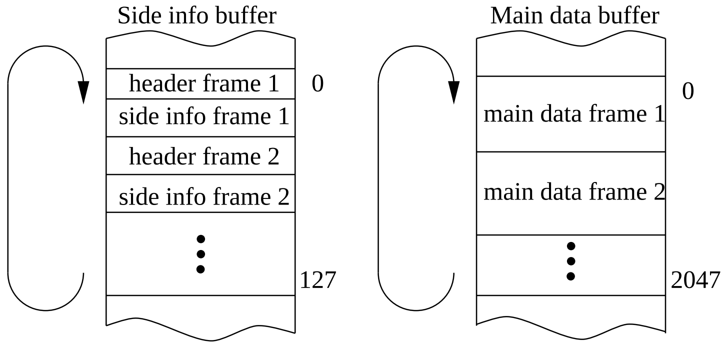


Figure 10.2: *The two circular buffers holding the header plus side information and main data respectively of the received frames*

The size of the buffer containing the headers and side information is chosen to 128 bytes and the size of the main data buffer is selected to 2K bytes, the circular buffers are shown in Figure 10.2.

In the following sections the individual routines involved in the BSP part of the decoder will be described. The description will start with a short description of the main loop of the decoder. The individual procedures will then be explained in turn, in the same order as they are used in the main loop.

10.1 Main Loop of the Layer III Decoder Algorithm

The main loop of the assembler program is shown in Figure 10.3. The program remains idle if a new frame is not ready. When a new frame has arrived the header and side information is decoded. When the header and side information is decoded, the header is validated. If an error was returned, it is reported to the DSP that the the current frame was erroneous and the program returns to its idle state, where it will wait for a new frame to be ready. If the header was OK the program continues. Next it is checked whether a CRC check word was present in the frame. If a CRC word is not available the calculation of CRC word is not performed and the program jumps to the decoding of main data procedure. Otherwise the CRC word for the header and side information is calculated. The calculated CRC word is then compared to the CRC check word. If they are not equal it is reported to the DSP that the frame was erroneous. If the calculated CRC word is identical with the CRC check word sent for the frame, the decoding of main data is performed, which will be explained more thoroughly later. In the assembly code, the main loop is a part of the code in Assembly Code B.1.

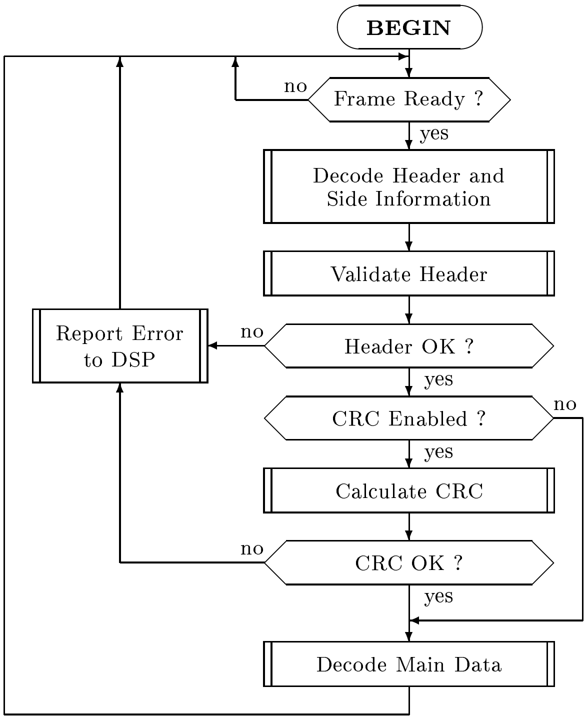


Figure 10.3: *The main loop of the Layer III decoder algorithm.*

10.2 Handle Incoming Bitstream

The Handle Incoming Bitstream interrupt routine is called every time data is ready in the bitstream receiver. The purpose of the interrupt routine is to ensure synchronization to the bitstream, identify the length of the side information and main data, and to place these data in the two circular buffers, as shown in Figure 10.4.

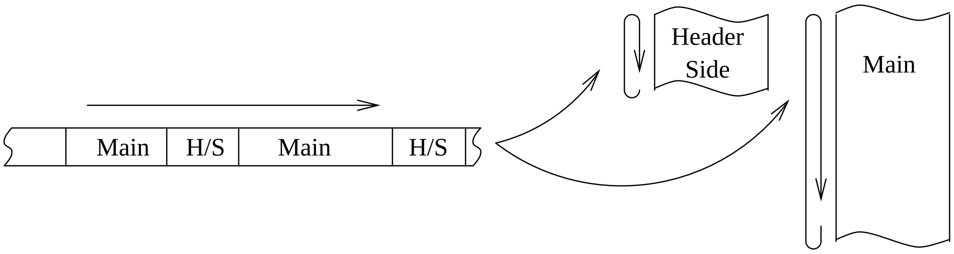


Figure 10.4: *The Handle Incoming Bitstream interrupt divides the incoming frame between a header and side buffer and a main data buffer.*

The transitions between the Header and Side part and the Main part of a bitstream is not know in advance, but has to be determined during the reception. The length of the header and side part is determined from whether it is a single channel or a dual channel bitstream, and whether a CRC check word is applied, see Table 10.1. The total frame length can vary from 104 bytes to 960 bytes.

Each interrupt from the bitstream receiver indicates that 12 bytes is ready to be moved into buffer. The fact that the size of the buffer in the bitstream receiver is only 12 bytes causes some trouble. The Handle Incoming Bitstream interrupt must remember, from call to call, how far is in receiving the frame. The interrupt service routine can be divided into four different tasks, which are performed in turn.

Length	Without CRC	With CRC
Mono	19 bytes	21 bytes
Stereo	36 bytes	38 bytes

Table 10.1: *Length of Header and Side information*

- Synchronize to the bitstream, and byte-align the frames.
- Place header in Side and Header buffer, and decode the information in the header which is needed in order to calculate the length of the side info and main data.
- Place side information in Side and Header buffer.
- Place main data in Main buffer.

A flow chart for the Handle Incoming Bitstream interrupt routine is shown in Figure 10.5.

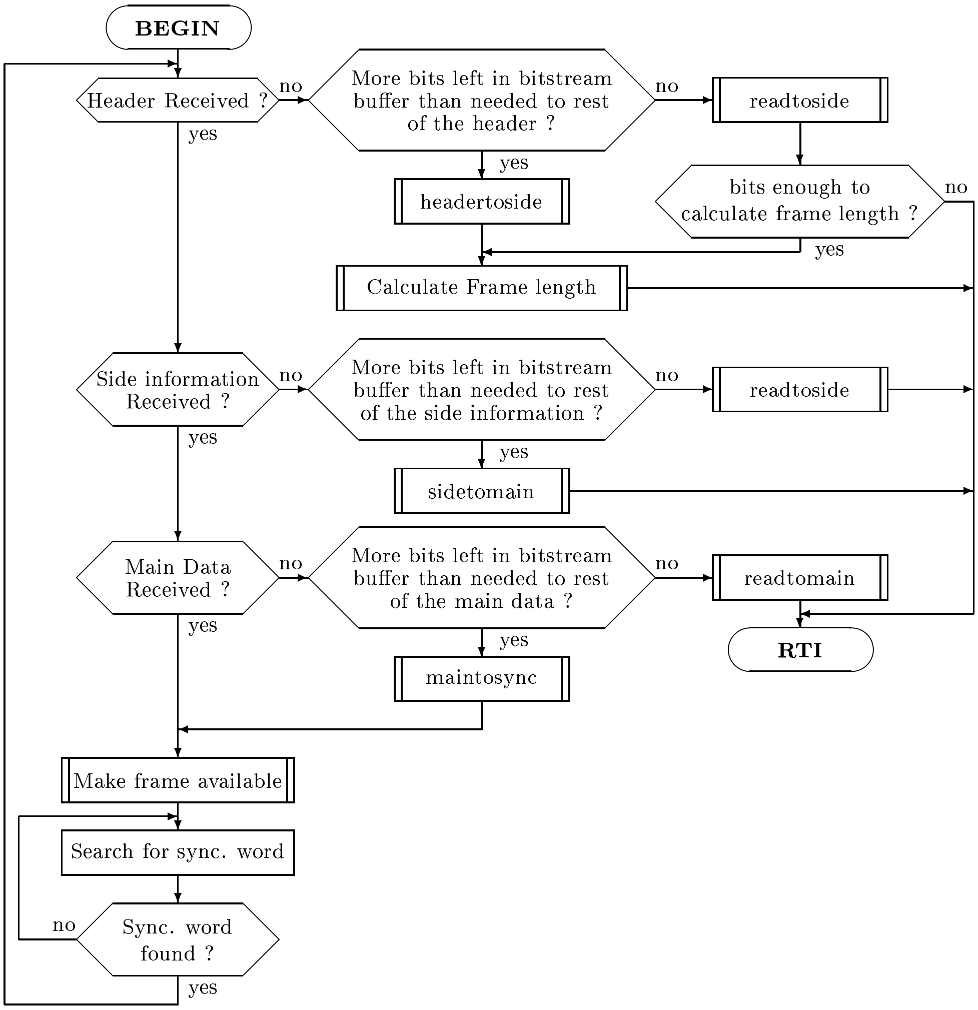


Figure 10.5: *The interrupt routine that handles the incoming bitstream.*

At startup no frame has been received yet and the routine must program the bitstream receiver to search for synchronization word in the outgoing buffer. This is done by using the memory mapped port dedicated the bitstream receiver. Whether a synchronization word has been found or not, can be read from the status register of the bitstream receiver. The routine then goes busy waiting until the synchronization word is found. When the synchronization word has been found the reading of data can begin. Each time an interrupt from the bitstream receiver arrives new data is ready. How many bytes are actually present in the outgoing buffer can be read from the upper 4 bits in the status register of the bitstream receiver. The routine must then read the status from where it was left after the termination of the previous interrupt. When the bitstream receiver interrupt occurs again, a decision must be made whether it was reading header, side information or main data in the previous interrupt. When a synchronization word has just been found the routine must read the next bits from the bitstream receiver as header bits. When a interrupt occur it is checked, via the status register of the bitstream receiver, how many bit are available in the outgoing buffer of the bitstream receiver. For example if the header has not been read yet, it is check whether the number of bits in the bitstream receiver is sufficient to read the entire header. Which of the two subroutines readtoside and headertoside is executed next depends on the result of this check. If there is not bits enough to read the whole header, the bits available are read as header bits. If enough bits has been read, the length of the frame can be calculated, else the interrupt routine returns. Otherwise if more bits were available in the outgoing buffer of the bitstream receiver than needed to read the header, the first four bytes are read as header bits and the rest as side information. Both the header and side information is placed in the same 128-bit circular buffer. This procedure is repeated, when reading the side information and the main data. Most times the readtomain routine is executed. When all data from a frame has been read into the respective circular buffers Make Frame Available is called. Making a new frame available is done by setting a flag, that indicates to the main loop that a new frame is ready and by copying the two pointers to each of the circular buffers. These two pointers are used in the succeeding procedures in the main loop. The source code for the Handle Incoming Bitstream interrupt routine can be seen in Assembly Code B.2.

10.3 FIFO Interrupt Routines

The purpose of the fifo interrupt routines is to feed the fifo register with data from memory. Two interrupt routines exist for the fifo, one for each of the circular buffers, i.e. fifointerrupt128 and fifointerrupt2048 respectively. The reason for having two interrupt routines is the fact that the two circular buffers do not have the same size. The two routines are shown below.

```
fifointerrupt128:
        fifo = mem( r7 )
        mask(r7++,128)
        rti

fifointerrupt2048:
        fifo = mem( r7 )
        mask(r7++,2048)
        rti
```

The two pointers generated by Handle Incoming Bitstream is used as base addresses for the two fifo routines. Both interrupts are vectored interrupts, which means that a memory location is allocated for each of the interrupts (the collection of these addresses are denoted vector table), The vector table is specified in the top of Assembly Code B.1. To select one of the two fifo interrupt routines two things must be done:

- Write the address of the wanted routine into the vector table, at the entry dedicated for the fifo.
- Set r7 to the base address of the buffer in memory.

The interrupt routines described above are a direct listing from Assembly Code B.1 line 173–181.

10.4 Decode Header and Side Information

When a new frame has been received and made available, the header and the side information must be decoded. The decoding simply copies all the variables present in the header and side information to separate locations in memory. In the top of Assembly Code B.4 all the variables are defined using the assembler directive #DEFINE. These define statements are global and can be used throughout the Layer III decoder. The procedure then returns five pointers. One pointing to the start address of the decoded system information and one pointing to the beginning of the decoded side information belonging to each of the channels in each of the granules. By using these pointers and having defined all the variables once, makes references to values in the header and side information very pleasant. An example is shown below.

```
g0 = mem(system_info_adr) ;set pointer to system info
g1 = mem(g0 + Layer)
```

In the example the Layer variable from the system information is put into g1. The only thing needed in this case, is the pointer system_info_adr. Figure 10.6 shows how the header and side information is decoded and how they are placed in memory.

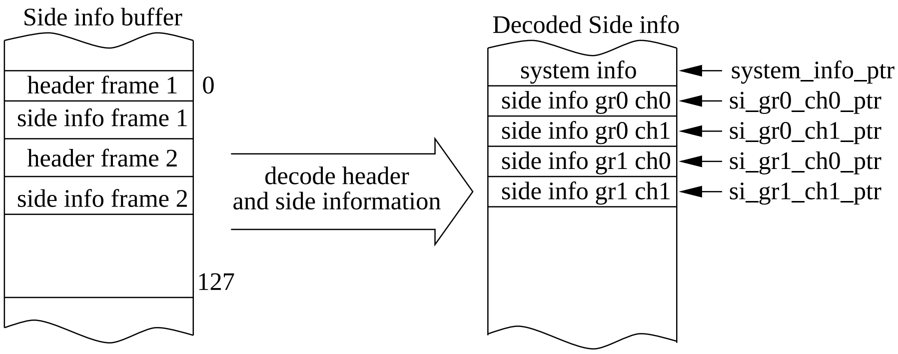


Figure 10.6: *Visualization of the task performed by the decode header and side information routine.*

10.5 Validation of header

The validation of the header procedure checks if the header of the incoming frame is valid. First the routine checks if the bitrate is one of the bitrates specified in [ISO/IEC 11172-3, 1993]. If the bitrate equals 15 an error is returned. The second check performed on the header is the sampling frequency. If the sampling frequency equals the reserved value 3 an error is returned. The assembly code for validating the header can be seen in Assembly Code B.5. In general this validation can be extended to include all variables in the header which are held constant, e.g. if only Layer III is supported.

10.6 CRC

If the header was valid the system info is used to find out, if a CRC check word was present in the frame. If a CRC check word is available the CRC word for the header and side information is calculated using the byte–wise method described in Section 5.4. By comparing the CRC check word and the calculated CRC word, it is decided whether to continue in the main loop or to produce an error message to the DSP. The source code is shown in Assembly Code B.6.

10.7 Decode Main Data

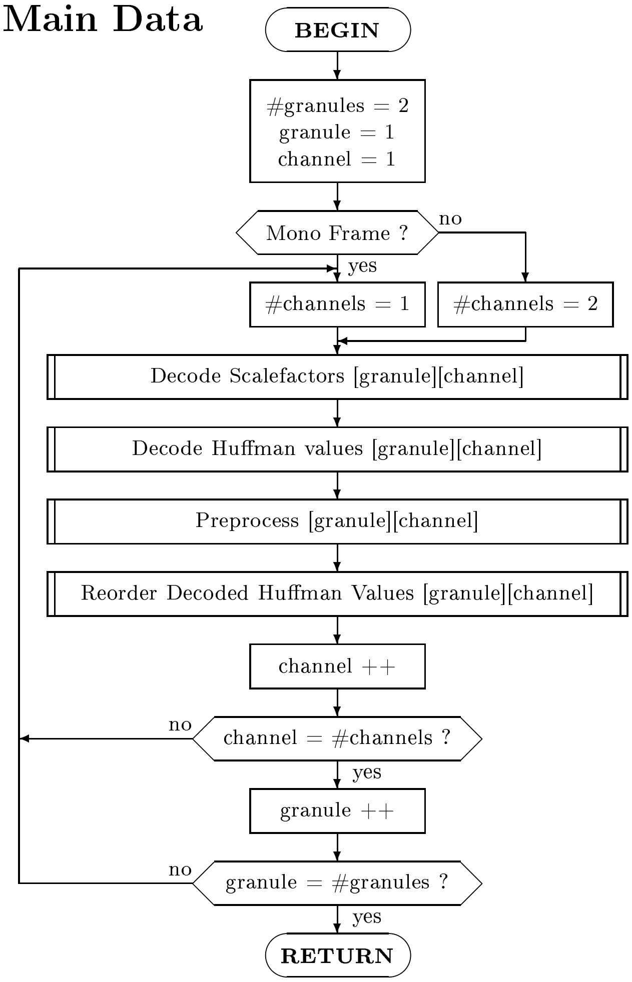


Figure 10.7: *The flow diagram for decoding the main data of an MPEG/Audio Layer III frame.*

The flow chart for the decoding of main data can be seen in Figure 10.7. The decoding of main data calls four procedures either two or four times depending on whether the frame is a mono or a stereo frame. If its a mono frame only one channel exist in each granule and hence the procedures is carried out twice, one time for each granule. When a stereo frame is present, two channels are present in each granule and therefore the procedures is carried out four times, once for each of the two channels in each of the two granules. The source code can be seen in Assembly Code B.7. In the following each of the four procedures included in the decoding of maindata algorithm will be explained shortly.

10.7.1 Decoding of Scalefactors

The decoding of scalefactors is done according to the algorithms specified in Subsection 4.4. The source assembly code can be seen in Assembly Code B.8. The procedure returns 36 decoded scalefactors and the number of bits used on scalefactors, i.e. *part2_lengt* for each channel in each granule

10.7.2 Huffman Decoding

Decoding of Huffman values is conducted in accordance with the description of Huffman Info Decoding and Huffman Decoding presented in Chapter 4, and the algorithm for clustered Huffman decoding has been invoked in the Huffman decoding. Two Huffman decoding algorithms have been implemented, one for the pairs within big_values partition and a second for the quadruples in the count1 partition. Zero padding of the rzero partition has been implemented directly in Huffman Info Decoding.

In Figure 10.8 the interaction between Huffman info decoding and the two Huffman decoding routines, Huffman Decoding Big_values and Huffman Decoding Count1, is shown. Prior to a call, Huffman Info Decoding reads the side information to determine the boundaries for the three partitions and to setup the correct Huffman tables and the associated number of linbits. When calling Huffman Decoding Big_values, a pointer to the proper Huffman table and the number of linbits associated with the Huffman table are passed. When calling Huffman Decoding Count1 only the Huffman table is passed. Both Huffman decoding routines returns a value indicating how many Huffman code bits have been used for decoding. The assembly code for Huffman Info Decoding is listed in Assembly Code B.9.

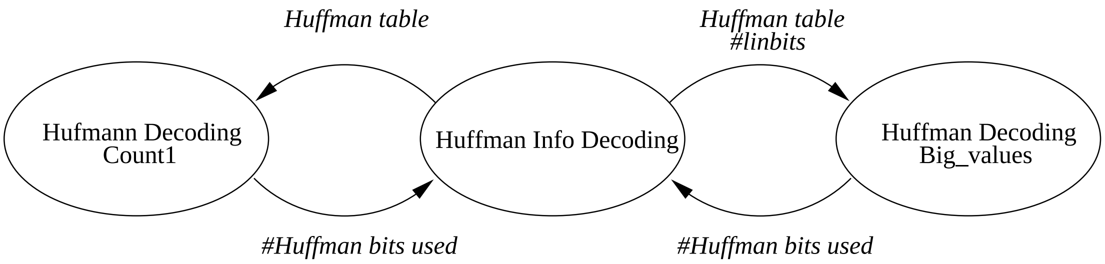


Figure 10.8: *Illustration of the interaction between the three routines that constitutes Huffman decoding.*

In Figure 10.9 the program flow for Huffman Decoding Big_values is shown. The upper half of the flow chart including the first decision is the implementation of the 4th order clustered Huffman decoding routine. The latter part illustrates how the quantized frequency lines X and Y are reconstructed and how their respective sign bits are concatenated in order to represent X and Y in sign extended notation. When returning from the routine the accumulated number of Huffman bits discarded are passed to Huffman Info Decoding. The assembly code for Huffman Decoding Big_values can be found in Assembly Code B.10.

Figure 10.10 illustrates the program flow for Huffman Decoding Count1. The implementation of clustered Huffman decoding does not differ from the implementation in Huffman Decoding Big_values. The latter part shows the sign concatenation of the quantized frequency lines and the storage of the sign extended representations of V, W, X and Y. The assembly code implementation is listed in Assembly Code B.10

10.7.3 Preprocessing

The preprocessing procedure follows Algorithm 4.4. However some changes have been made from Algorithm 4.4 to the assembly code version. The preprocessing of the values used in the descaling block is combined with the reordering algorithm. This means that the preprocessed values are stored on reordered memory locations on the fly. The reordered addresses are calculated in advance and tabulated. By doing this 576 memory operations per channel per granule is saved. The assembly code for the preprocessing and reordering of values used in the descaling block can be seen in Assembly Code B.11.

10.7.4 Reorder Huffman data

The decoded Huffman values are reordered as described in Algorithm 4.5. The reordering is performed by subband and the procedure returns 576 reordered decoded Huffman values for

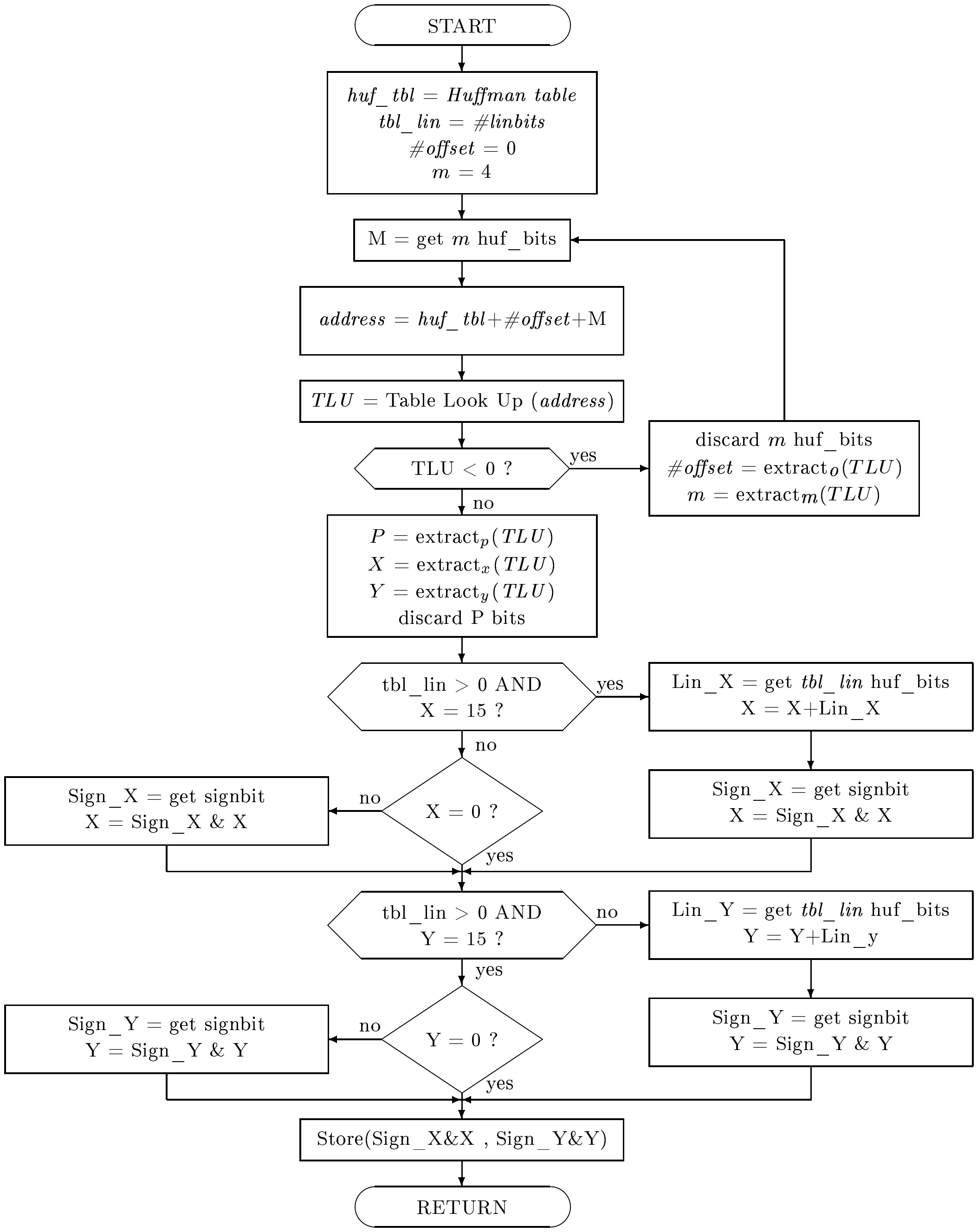


Figure 10.9: *Illustration of the program flow in Huffman Decoding Big_values. The variables m, M, #offset and P correspond to the number of codebits in next table look up, the actual value of m, the address offset to next table entry and the number of codebits in last table entry, respectively.*

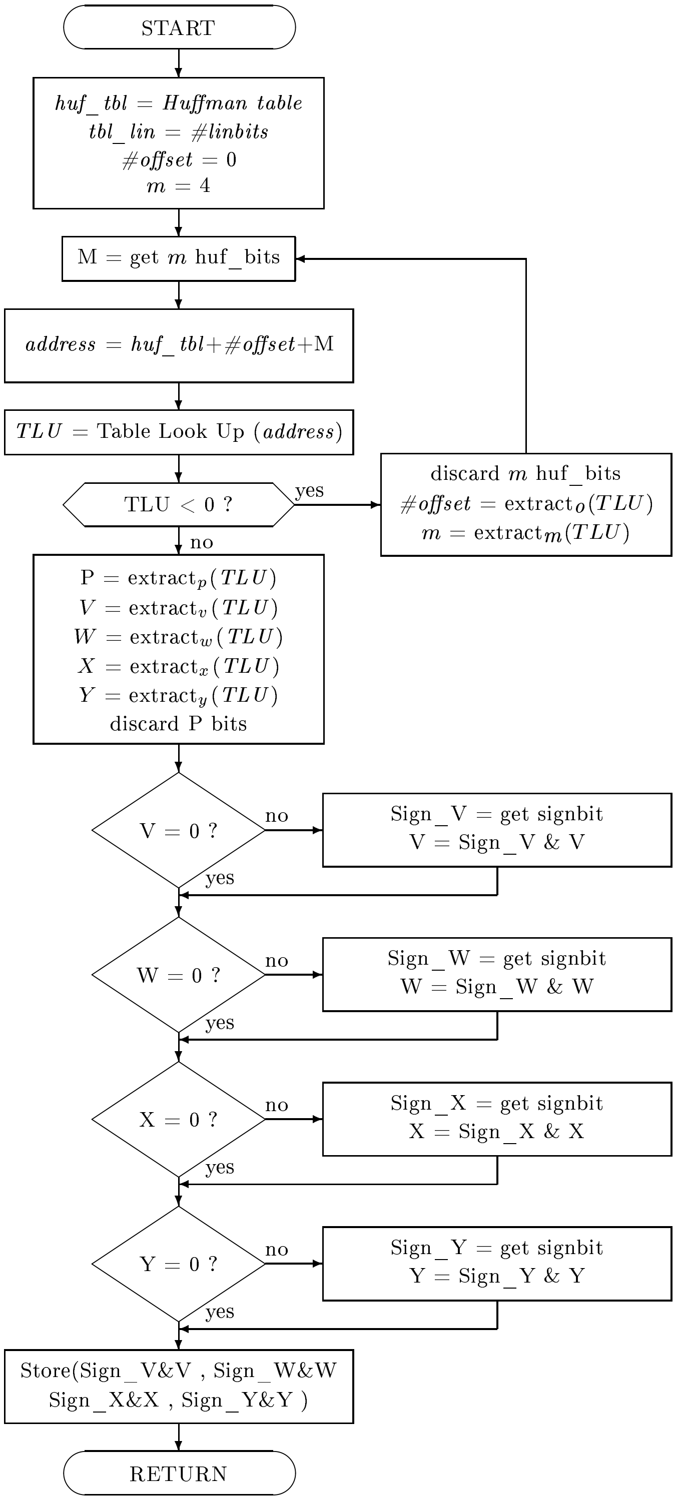


Figure 10.10: *Illustration of the program flow in the Huffman Decoder Count1. The variables m, M, $\#offset$ and P correspond to the number of codebits in next table look up, the actual value of m, the address offset to next table entry and the number of codebits in last table entry, respectively.*

each channel in each granule. The source code can be seen in Assembly Code B.12.

10.8 Serial Transmission of the Output from the Decoder.

When the four procedures in Main Data Decode have been executed, the decoding task for a frame has completed. The memory layout for the output data, ready to be transmitted to the DSP is shown in Figure 10.11. What remains is an appropriate way to copy these data to the serial port. For this purpose an interrupt routine to has been written.

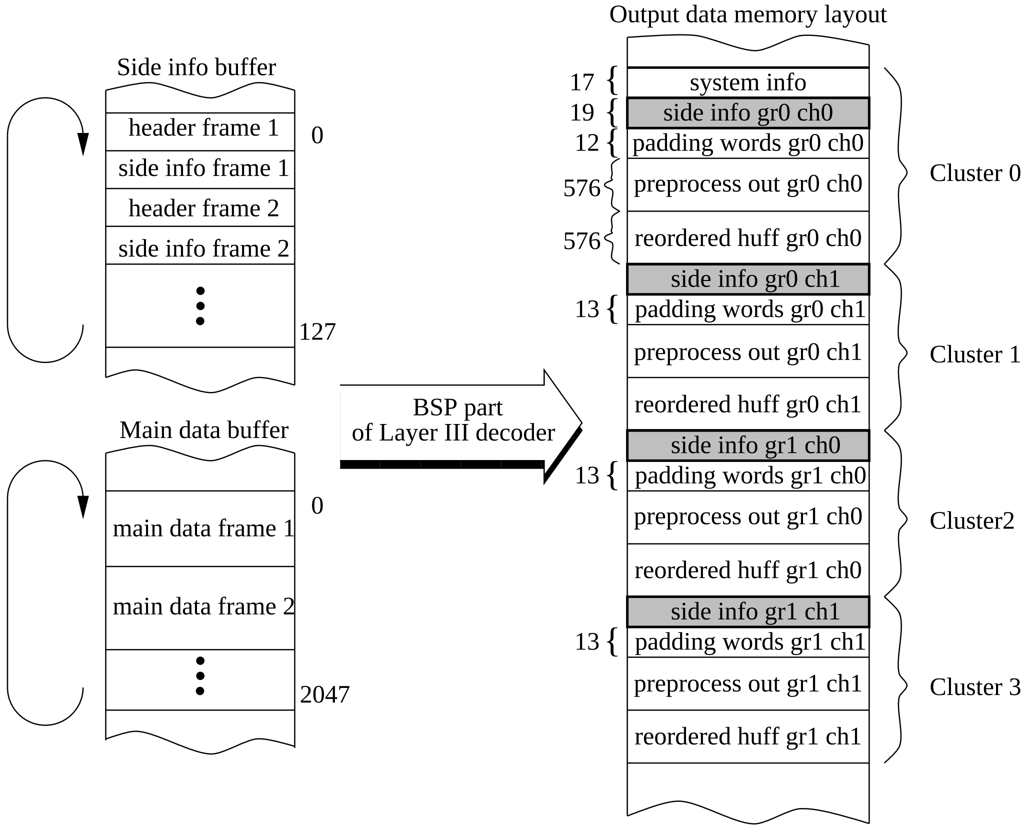


Figure 10.11: *Input to the BSP part of the Layer III decoder and the resulting memory layout.*

The resulting data from the decoding of a frame is placed in the same clusters as the previous frame in memory. This means that the transmission of data for a particular cluster must be completed before new decoded data can be written into the same cluster. How the decoding of frames and the transmission of decode data are schedule is shown in Figure 10.12. The transmission of data is enabled as soon as the data of the first cluster is ready, which gives the serial interrupt routine the maximum amount of time to complete its transmission. Hence the demands to the speed of the serial transmission is lowered. Alternatively the data transmission could be performed after the entire frame was decoded. Contrary to the method first mentioned, this method put the highest demands to the serial transmission.

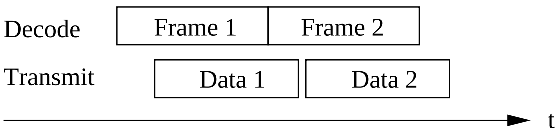


Figure 10.12: *The scheduling of the decoding of frames and the transmission of decoded data.*

The amount of data to be copied to the serial port depends on whether a mono or a stereo frame has been decoded. If a mono frame was decoded the only Cluster 0 and Cluster 2 are transmitted. Otherwise all four clusters in Figure 10.11 are transmitted. When Cluster 0 has been decoded the serial transmission is initiated. First, a start address and an end address for the cluster is copied into three locations in memory as shown in Figure 10.13. The end address has been copied into two locations, end_write and end_read as shown in Figure 10.13. Next the serial interrupt is enabled whereby data transmission can begin. As the remaining three clusters are decoded the end address in end_write is updated.

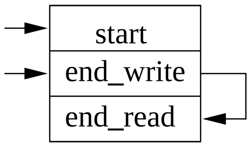


Figure 10.13: *Communication with the serial port. Where end_write and start is written by the main data decode procedure and read by the serial interrupt routine.*

In Figure 10.14 the interrupt routine for the serial port is shown. When a service is initiated it is first tested if 16 words are ready for transmission. If the term is met the 16 words following the start pointer is copied to the serial port buffer. Next the start pointer is updated, and it is tested if 16 words are ready for the next time the interrupt routine is entered. If data is ready for another transmission the interrupt routine returns.

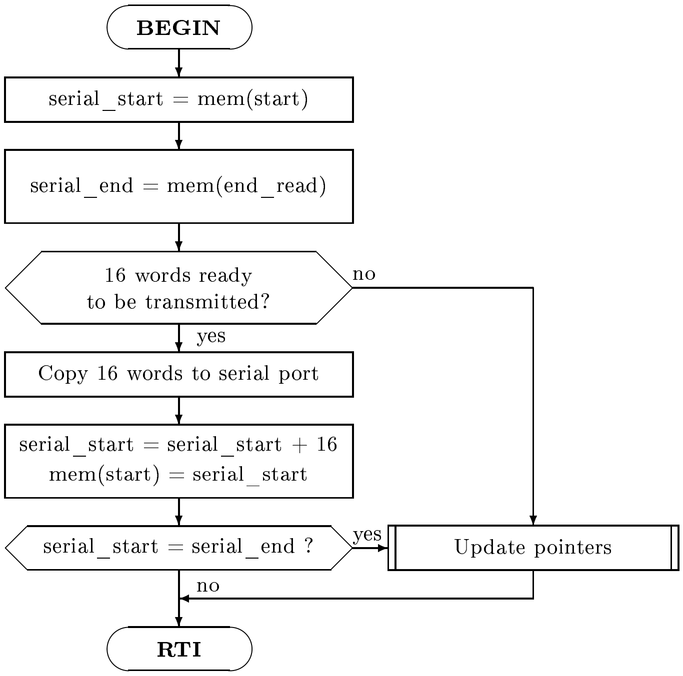


Figure 10.14: *A flowchart of the interrupt routine for transmitting the output data from the decoding algorithm.*

When no data are ready it is tested whether a new cluster has been decoded and is ready for transmission. The test is conducted as shown in Figure 10.15. First it is tested if the end_write pointer has been updated, meaning new clusters are ready for transmission. If no new clusters are ready the interrupt routine disables it self and returns. If new clusters have been added, it is tested whether a mono or a stereo frame has been decoded in order to determine how the start pointer and the end_read pointer should be updated. For a stereo frame the end_read pointer substituted with end_write. For a mono frame the start pointer is set to the start of Cluster 2 and the end_read pointer is substituted with end_write.

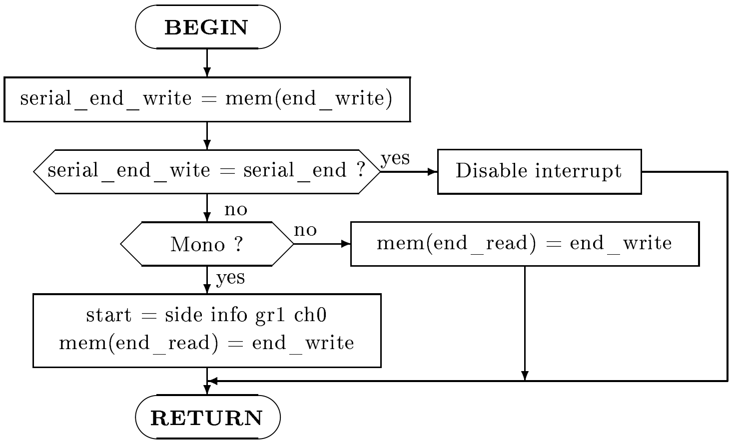


Figure 10.15: *Sub routine that updates the pointers used to transmitting the output data.*

11

Hardware-Software Test

In Chapter 3 the set of operations to be performed on the BSP were defined together with the interfaces from the BSP to the remaining parts of the decoder. In Chapter 7 a set of guidelines to the design of the BSP were stated. Having established these guidelines the design of the BSP could be initiated. In this chapter the design, implementation and test phase of the BSP will be presented. Focus will be on how the hardware and software is tested. The test was performed in the three stages specified in Chapter 3.

11.1 Hardware Implementation and Test

The design, implementation and test of hardware is an iterative process, which will be described in the following. An overview of the process is shown in Figure 11.1. The BSP could be divided into a number of elements, i.e. datapath, program sequencer, bitstream receiver, fifo and serial port. Each of these elements were then designed separately. Having designed the individual parts of the BSP, the next step was to implement the design in VHDL.

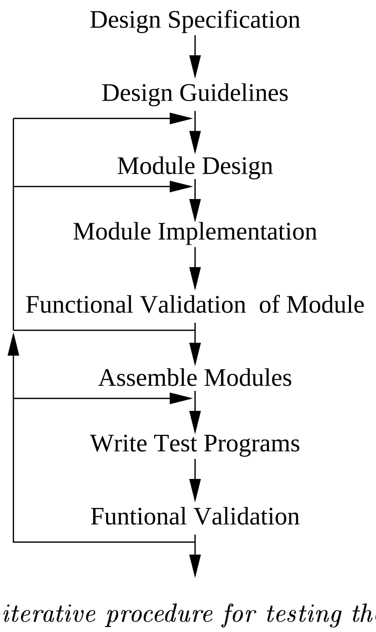


Figure 11.1: *The iterative procedure for testing the designed hardware.*

Dividing the processor into modules makes it possible to implement more than one part at the time. Implementation of the BSP was conducted by implementing the individual modules of the processor independently. After the individual modules were designed a functional test was performed by use of visual debugging in QHsim. QHsim is the simulator part of QuickVHDL. Having validated the functionality of the individual modules, the BSP was assembled by clustering the individual modules hierarchically. At this point a Bitstream Processor that could be synthesized, was present. The individual modules in the processor were tested separately and next step was to validate the functionality of the assembled BSP. For this purpose a method for simulating memory was required in order to be able to execute programs on the processor. A behavioral model of SRAM memory was used to simulate memory. With this memory model it was possible to load the contents of a file into the "memory". The memory model has the normal input and output pins of a memory, i.e. cs, oe, we, address bus and data bus, but provided no timing. A testbench was constructed by supplying a clock generator to the BSP, as shown in Figure 11.2

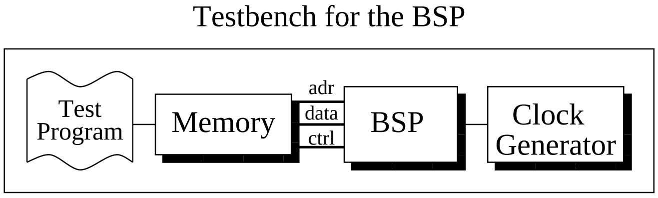


Figure 11.2: *The testbench used to test the BSP.*

Small routines were generated using the assembler and then executed on the processor using the testbench in Figure 11.2. The test programs were small routines used to test specific parts of the processor.

Debugging: When the functionality of all the instruction types were validated, and the test programs produced the expected results, the algorithms included in the BSP part of the Layer III decoder could be programmed. To ease the debugging of the programmed algorithms, a debug port was constructed. The debug port was used together with the trace interrupt facilitated by the processor. By writing interrupt routines for the trace interrupt, that writes the desired status from the processor to a user port, it was possible to display information from the BSP, e.g. the contents of registers and the program counter, in the simulator window of QHsim. An example of the use of the trace interrupt is shown in Appendix D. The trace interrupt could be enabled any where in the program, where the debugging should start. To send information to the debug unit other routines that simply writes a message to the display were written, an example of this can also be seen in Appendix D.

Test of Software: To validate the output of the algorithms written in assembly code a reference is required. For this purpose an existing decoder was used. The decoder used, was a shareware ISO–compliant MPEG/Audio Layer III Audio decoder provided by Frauenhofer–Institute for Integrated Circuits. The decoder provided different information about the bitstream being decoded. In addition some compliance bitstreams was supplied. The compliance bitstreams test specific parameters in the decoder. Examples of information that can be displayed with the existing decoder are: Header and side information, decoded scalefactors and decoded Huffman values. In addition the compliance bitstreams contains frames that tests the sampling frequency, bitrate, `main_data_begin` and other parameters. An example display from the existing decoder can be seen in Appendix D. For all of the algorithms included in the decoder the relevant parameters can be tested. For instance the decoding of scalefactors, frames with long, short and normal+short window types are available.

A validation of the decoder could be performed by decoding the compliance bitstreams, display the output after the respective algorithms by use of the debug port and then compare these results with the output from the existing decoder. If the result of decoding the compliance bitstreams equals the result from the reference then the functionality of the decoder has been validated. Because the decoder is capable of decoding the compliance bitstreams this does not mean that the decoder will work for all bitstreams and hence no guarantee can be given, without performing an exhaustive test of all the parameters involved in the Layer III decoder.

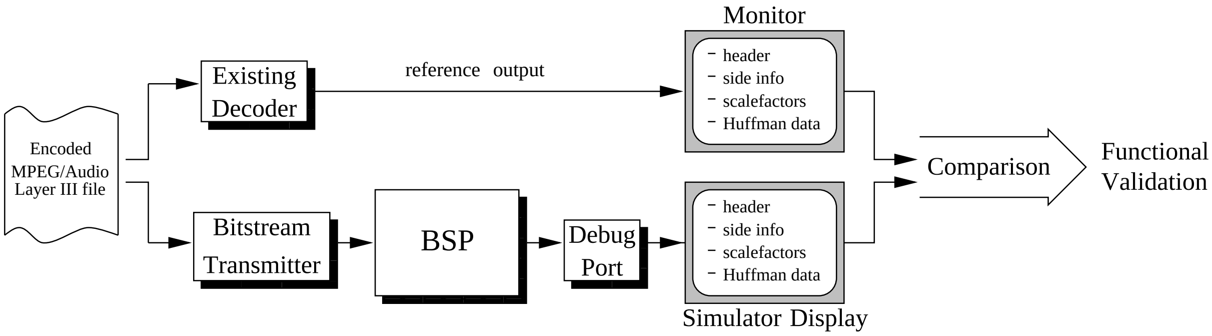


Figure 11.3: *An overview of the test procedure carried out in order to perform a functional validation of the decoder.*

In Figure 11.3 the test procedure is visualized. The Bitstream Transmitter serves for test purposes only and its function is to translate the audio file into a serial bitstream. The worst case situation, specified the design phase, was the decoding of a stereo frame with a bitrate of 320 kbit/s, a sampling frequency at $48kHz$ and short windows only. Ensuring that the decoder will work in worst case could the be performed, by exciting the decoder with a worst case frame. However a frame representing the worst case has been unobtainable and it was not among the frames in any of the bitstreams used for compliance testing. A frame with bitrate 320 kbit/s and $48kHz$ with short windows only was found among the compliance bitstreams but unfortunately it was coded in mono. This fact led to the encoding of real audio bitstreams using the encoder supplied from Frauenhofer, but even here the highest bitrate achieved in a frame, that had the remaining characteristics, was 256 kbit/s.

The final accept test would be a listening test, but this requires that the DSP part of the decoder is implemented.

12

Evaluation of the Layer III Decoder

Although it is possible to estimate the demands to the BSP in advance, as it was done in Chapter 6, this chapter will provide a more precise estimate. The BSP is profiled during decoding of real MPEG/Audio bitstreams. The memory demands of the BSP is calculated. The area of the processor is estimated, and it is estimated if ES2-0.7μ standard cells can meet the timing requirements. The chapter concludes with an appraisal of some DSP's which will be able to perform the remanding algorithms of Layer III.

12.1 Profiling of the BSP

In order to evaluate the required clock frequency of the BSP processor, a profiling has been performed when execution the Layer III decoding program. The profiling is done by tracing the program counter during decoding of some test frames. The program counter can be traced using two methods, either by sampling the program counter with a constant frequency, or by sampling the program counter in each fetch cycle. The difference between the two methods stems from the possibility of having variable instruction length in the BSP. The first method gives a profile where the time spend in each program line can be determined, the second gives a profile where the number of times each program line has been executed can be determined. The latter method of profiling is used in this chapter as it is the easiest obtainable from the VHDL simulation.

The decoding was done by simulating the VHDL description of the BSP in QHsim. The program counter is extracted from the simulation by letting QHsim log the program counter to a file, each time the program sequencer is in the fetch state. By tracing the program counter in this manner, it is possible to simulate "real time" execution. The program counter could be traced using the trace interrupt instead, but by using this method the trace interrupt is executed between each instruction line, thus the simulation will not be in "real time". The latter method of tracing the program counter is though very valuable because it opens the possibility for performing profiling on the processor when it has been implemented on a chip. When profiling the BSP on a chip, the trace interrupt routine could for example write the program counter to a serial port.

The decoder is profiled when decoding real audio bitstreams. The bitstreams have been specially selected for setting high demands to the decoder. Whether a bitstream sets high demands to the decoder is determined from a number of factors. *i)* The size of the rzero partition of the bitstream. When the rzero region is small it indicates that the audio signal is broadbanded and requires code bits for the entire frequency spectrum. *ii)* If a frame is coded with short windows, it requires reordering of both scalefactors and quantized frequency lines. This is also an indication of the complexity of a frame. *iii)* If the bitrate is high, more bits have to be processed. *iv)* If the sampling frequency is high, more data has to be transmitted from the BSP to the DSP.

The Layer III standard allows bitrates at 320 kbit/s, but because bitstreams coded with this bitrate have been unobtainable, bitstreams coded with 256 kbit/s have been used. A short description of the test streams used in this profiling is tabulated in Table 12.1.

Name	Duration	No of Frames	SF	Bitrate	Long/Short windows
Stream1	1.2 s.	50	44.1 kHz	256,000	44/16
Stream2	0.026 s.	1	44.1 kHz	256,000	0/1
Stream3	0.36 s.	15	48 kHz	256,000	7/8
Stream4	1.07 s.	40	44.1 kHz	128,000	36/14

Table 12.1: *Test frames used when profiling the BSP.*

The frame in Stream2 is a close to worst-case frame as it has the following features:

- 6868 bits in the frame including bits in reservoir. (7680 is the maximum allowed frame length in MPEG/Audio)
- Entire frame is coded in big_values partition, hence no rzero partition.
- Short windows only, thus reordering is demanded.

During the profilings, the BSP is clocked with 20 MHz. The output from the profiling is a file containing a complete trace for the program counter during execution. This file is then translated into a histogram, where the frequency of each program line is counted. The results from the profilings are the number of times each instruction in the program has been executed in order to decode a bitstream. The instruction count belonging to each of the algorithms, are then summarized and represented as a bar graph, see Figure 12.1. From the profilings it is found that Stream2 sets the highest demands to the BSP. Graphs of the results from profiling Stream1 and Stream3 can be seen in Appendix C.

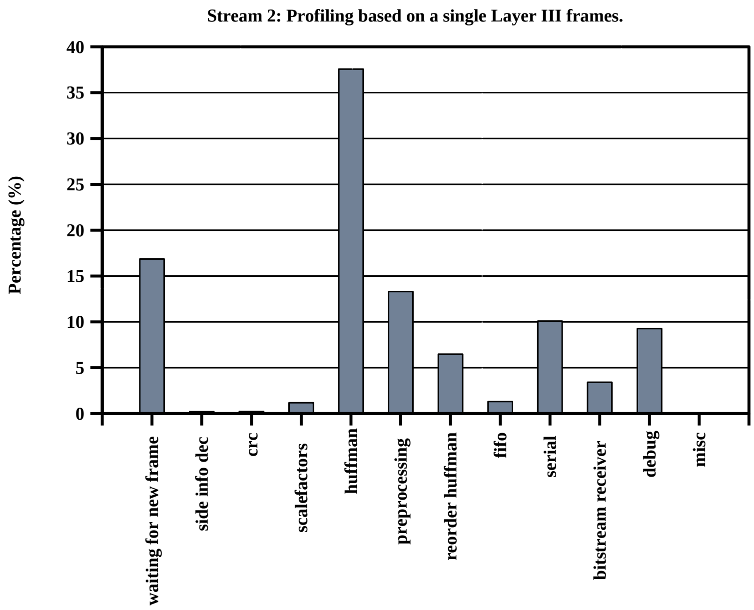


Figure 12.1: *Profiling result from decoding Stream1.*

From the results of profiling the decoding of Stream2, it can be seen that the BSP spends 17 % of execution on waiting for new data to decode. In addition to the 17 % waiting for new frames, the BSP spends 9 % on printing debug information, giving a total of 26 % idle time in the processor. If it is assumed that the increase in computation demands increases linearly from

256 kbit/s to 320 kbit/s, then the BSP will be able to decode 320 kbit/s as well. But because the frame is only a near worst-case frame, this has to be tested further with other frames in order to guarantee that the processor will be able to decode every Layer III bitstream when clocked at 20 MHz.

The major consumer on Figure 12.1 is the Huffman decoding. This is a natural consequence of being the algorithm that has to process most bits. If the assembly code is to be furter optimized, the natural place to start is in the Huffman decoding. In order to get a more thorough knowledge where in the Huffman decoding, the BSP spends 37 % of its instructions, the responsible code is listed below. The percentage on the beginning of each line is the number of times the instruction line is executed compared to the entire program execution. Note that the reason that only some lines has a percentage, is that all percentages below 0.01 % have been removed. The assembly code is a part of Assembly Code B.10.

```
         :  HUFFMAN_DECODE:
  0.82 % :       [ 690]          PUSH ( R3 )                   ; push frequency line saving address on the stack
  0.82 % :       [ 691]          R1 = 3                        ; r1 = #code bits to be read(4 bits), modify later if hole in one!
  0.82 % :       [ 692]          R0 = READFIFO ( R1 )          ; copy 4 bits from fifo into r0
  0.82 % :       [ 693]          R3 = MEM(R6 + R0)             ; r3 = r0'th entry in Huffman table pointed to by r6
  0.82 % :       [ 694]          R3 = R3                       ; test if possitive (msb = 0) meaning a endnode hit
  0.82 % :       [ 695]          JUMPLZ PC + 53 [ NXT_CLUSTER ]
  0.53 % :       [ 696]          R1 = R3 >> 13                 ; r1 = #codebits in endnode
  0.53 % :       [ 697]          SHIFTFIFO ( R1 )              ; discard the r1 bits in fifo just used
         :  START_DEC:
  1.03 % :       [ 698]          G0 = 240                      ; g0 = 0000.0000.1111.0000
  1.03 % :       [ 699]          R0 = G0 AND R3                ; r0 = 0000.0000.   x.0000
  1.03 % :       [ 69A]          R0 = R0 >> 4                  ; r0 = x
  1.03 % :       [ 69B]          G0 = G0 >> 4                  ; g0 = 0000.0000.0000.1111
  1.03 % :       [ 69C]          R3 = G0 AND R3                ; r3 = y
  1.03 % :       [ 69D]          G0 = R0 - 15                  ; g0 = x - 15, test if x = 15
  1.03 % :       [ 69E]          JUMPLZ PC + 10 [ NO_LINBTSX ]
         :       [ 69F]          R5 = R5                       ; test if #linbits = 0
         :       [ 6A0]          JUMPLZ PC + 8 [ NO_LINBTSX ]
         :       [ 6A1]          G0 = R5 + 2                   ; g0 = #signbit_x + #linbits_x + 1
         :       [ 6A2]          R1 = R1 + G0                  ; r1 = #used codebits is updated
         :       [ 6A3]          G0 = POPFIFO ( R5 )           ; pop r5 linbits_x of the fifo
         :       [ 6A4]          R0 = R0 + G0                  ; x = x + linbits_x
         :       [ 6A5]          G0 = POPFIFO ( 0 )            ; g0 = signbit_x
         :       [ 6A6]          G0 = G0 << 15                 ; signbit_x is moved to the left
         :       [ 6A7]          R0 = R0 OR G0                 ; r0 = signbit_x & 00..0 & (x + linbits_x)
         :       [ 6A8]          JUMP PC + 6 [ TEST_Y ]
         :  NO_LINBTSX:
  0.82 % :       [ 6A9]          G0 = R0 - 1                   ; test if x = 0
  0.82 % :       [ 6AA]          JUMPLZ PC + 4 [ TEST_Y ]
  0.59 % :       [ 6AB]          G0 = POPFIFO ( 0 )            ; g0 = signbit_x
  0.59 % :       [ 6AC]          G0 = G0 << 15                 ; signbit_x is moved to the left
  0.59 % :       [ 6AD]          R0 = R0 OR G0                 ; r1 = signbit_x & 00..0 & x
  0.59 % :       [ 6AE]          R1++                          ; r1 = r1 + #codebits in signbit_x
         :  TEST_Y:
  0.82 % :       [ 6AF]          G0 = R3 - 15                  ; g0 = y - 15, test if y = 15
  0.82 % :       [ 6B0]          JUMPLZ PC + 10 [ NO_LINBITSY ]
         :       [ 6B1]          R5 = R5                       ; test if #linbits = 0
         :       [ 6B2]          JUMPLZ PC + 8 [ NO_LINBITSY ]
         :       [ 6B3]          G0 = R5 + 2                   ; g0 = #signbit_y + #linbits_y + 1
         :       [ 6B4]          R1 = R1 + G0                  ; r1 = #used codebits is updated
         :       [ 6B5]          G0 = POPFIFO ( R5 )           ; pop r5 linbits_y of the fifo
         :       [ 6B6]          R3 = R3 + G0                  ; y = y + linbits_y
         :       [ 6B7]          G0 = POPFIFO ( 0 )            ; g0 = signbit_y
         :       [ 6B8]          G0 = G0 << 15                 ; signbit_y is moved to the left
         :       [ 6B9]          R3 = R3 OR G0                 ; r3 = signbit_y & 00..0 & (y + linbits_y)
         :       [ 6BA]          JUMP PC + 6 [ END_HUF_DEC ]
         :  NO_LINBITSY:
  0.82 % :       [ 6BB]          G0 = R3 - 1                   ; test if y = 0
  0.82 % :       [ 6BC]          JUMPLZ PC + 4 [ END_HUF_DEC ]
  0.64 % :       [ 6BD]          G0 = POPFIFO ( 0 )            ; g0 = signbit_y
  0.64 % :       [ 6BE]          G0 = G0 << 15                 ; signbit_y is moved to the left
  0.64 % :       [ 6BF]          R3 = R3 OR G0                 ; r3 = signbit_y & 00..0 & y
  0.64 % :       [ 6C0]          R1++                          ; r1 = r1 + #codebits in signbit_y
         :  END_HUF_DEC:
  0.82 % :       [ 6C1]          G0 = POP()                    ; g0 = frequency line saving address
  0.82 % :       [ 6C2]          MEM(G0) = R0                  ; save signbit_x & 00..0 & x  in memory
  0.82 % :       [ 6C3]          G0++                          ; increment saving address
  0.82 % :       [ 6C4]          MEM(G0) = R3                  ; save signbit_y & 00..0 & y  in memory
  0.82 % :       [ 6C5]          G0++                          ; increment saving address
  0.82 % :       [ 6C6]          R3 = G0                       ;
  0.82 % :       [ 6C7]          R1++                          ; r1 is updated with the 1 bit always missing !!
  0.82 % :       [ 6C8]          RTN                           ; return to HUFFMAN_INFO_DECODING
         :  NXT_CLUSTER2:
  0.02 % :       [ 6C9]          G0 = R1 + 4                   ; #code bits used = #codebits used  + 4 for every cluster jump
  0.02 % :       [ 6CA]          R1 = G0                       ; g0 is copied into r0
         :  NXT_CLUSTER:
  0.45 % :       [ 6CB]          SHIFTFIFO ( 3 )               ; discard 4 bits in fifo
  0.45 % :       [ 6CC]          R0 = R3 << 3                  ; 3 msb (test bit and next cluster length) in huffman table entry is removed
  0.45 % :       [ 6CD]          R0 = R0 >> 3                  ; r0 = offset to next cluster in huffman table
  0.45 % :       [ 6CE]          G0 = R3 << 1                  ; g0 = max cluster size & offset to next cluster in huffman table
  0.45 % :       [ 6CF]          G0 = G0 >> 14                 ; g0 = max cluster size
  0.45 % :       [ 6D0]          R3 = READFIFO ( G0 )          ; g0 = copy g0 bits from fifo into r3
```

```
0.45 % :     [ 6D1]     R3 = R3 + R0                  ; r3 = the total offset to table adress in r6
0.45 % :     [ 6D2]     R3 = MEM(R3 + R6)             ; r3 = r3'th entry in huffman table
0.45 % :     [ 6D3]     R3 = R3                       ; test if possitive (msb = 0) meaning endnode hit
0.45 % :     [ 6D4]     JUMPLZ PC + -12 [ NXT_CLUSTER2 ]
0.31 % :     [ 6D5]     G0 = R3 >> 13                 ; r3 = #codebits in endnode
0.31 % :     [ 6D6]     R1 = R1 + G0                  ; r1 = r1 + #code bits in endnode --might miss 1
0.31 % :     [ 6D7]     R1++
0.31 % :     [ 6D8]     SHIFTFIFO ( G0 )              ; discard the r3 bits in fifo just used
0.31 % :     [ 6D9]     JUMP PC + -66 [ START_DEC ]   ; end of include
```

Part of Huffman decoding responsible for using 37 % of the availeble instructions.

As it can be seen from the profiling, the time spend in Huffman decoding is distributed over several instructions. If the time was spend on a few instructions repeatedly called in the same order, it would be possible to obtain a better performance by equipping the BSP with instructions capable of performing these instructions in one instruction. As it can be seen from the profiling, the Huffman decoding has no such loops, and even though a few instructions were saved, it would only result in a minor performance increase.

Because of the possibility of having variable execution length in the BSP, it is not possible to calculated the resulting MIPS count in advance. The resulting MIPS count is dependent of the program to be performed on the BSP, e.g. a program with a majority of register-register operations will provide a higher MIPS count than a program dominated by load/store operations. The length of the instructions in the BSP varies from three sub cycles, to seven sub cycles, yielding a MIPS count between 2.9 and 6.7 MIPS. If the BSP was executing register-register operations exclusively, it would be able to perform 6.7 MIPS, and if it was performing load/store operations exclusively it would perform 5 MIPS.

The profiling reports that the BSP executes 138,000 instructions during the decoding of Stream2. The BSP receives $44100/1152 = 38.28$ frames per second, when decoding a 44.1 kHz signal. By using this information it can be calculated that the BSP performes 5.3 MIPS when decoding Stream2.

Although the MPEG/Audio Layer III standard supports 320 kbit/s, Layer III targets bitrates at 128 kbit/s [Shlien, 1994]. This is the reason for profiling the BSP when decoding a 128 kbit/s bitstream. The result of this profiling can be seen in Figure 12.2. The profiling reveals that the BSP has over 50 % idle time when decoding a 128 kbit/s bitstream.

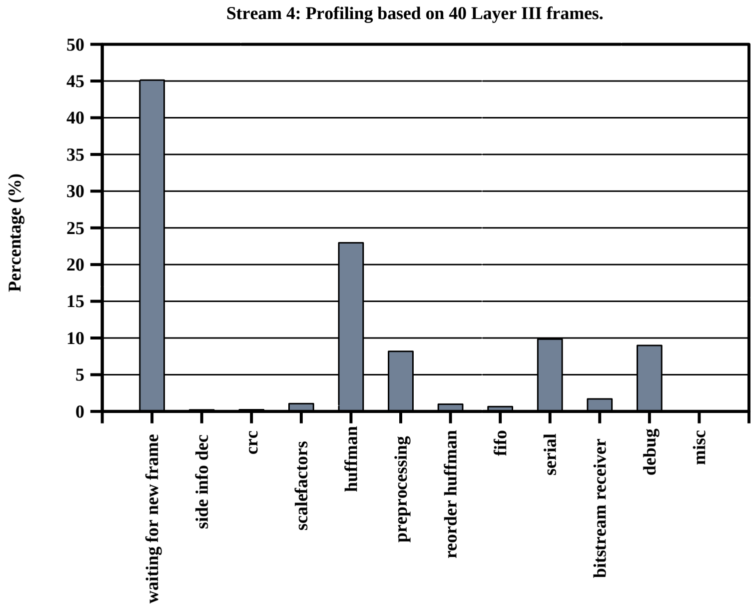


Figure 12.2: *Profiling result from decoding Stream4.*

12.2 Memory Usage

The memory usage in the BSP is illustrated in Figure 12.3. The total memory usage of the BSP is approximately 16kwords. Approximately 3.5 kword of this memory usage is used for lookup tables, 2 kword for program and the rest is used for data buffers.

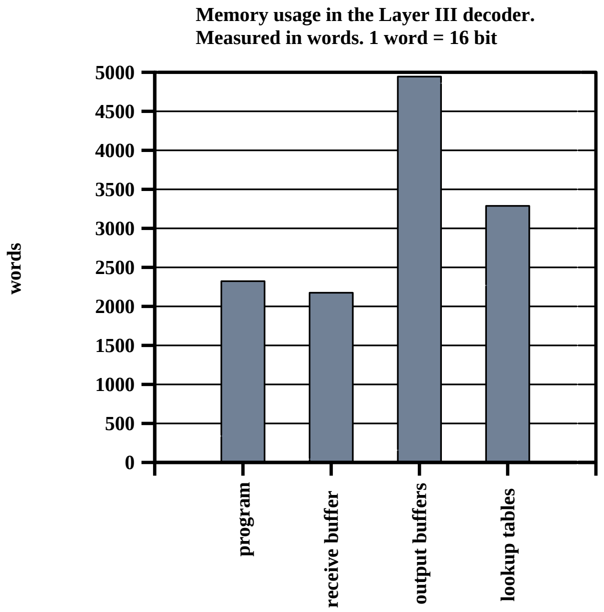


Figure 12.3: *Memory usage in the BSP.*

12.3 Timing Analysis

Whether the BSP is able to actually running at 20 MHz depends on the technology it is implemented in, and the length of the critical path. The critical path is determined by the longest delay through combinatorial circuits, placed between registers plus propagation delay, setup and hold time for the register, as illustrated in Figure 12.4.

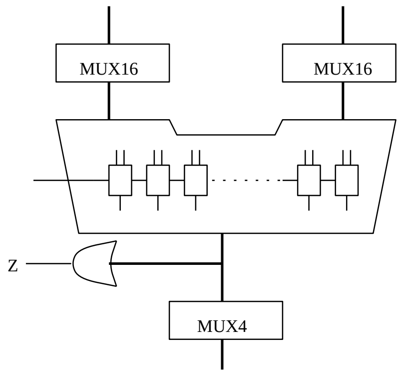


Figure 12.4: *Critical Path: Delay through combinatorial circuit between registers.*

The critical path can be assumed to be the datapath. This is a reasonable assumption based on the fact that both the program sequencer, fifo controller, bitstream receiver and serial port have little combinatorial logic, and are primarily state machines. In a single clock cycle the datapath must be able to perform an arithmetic operation between the contents of two registers and make the result stable on the input of the destination register. If the arithmetic operation to be performed is add, and the logic for addition in the ALU is realized as a ripple-carry adder, the delay through the datapath is as illustrated in Figure 12.5. The total delay in this datapath is through a 16:1 MUX, the carry chain of the adder and an or-gate to generate z-flag.

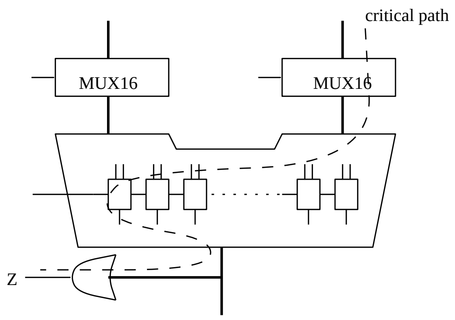


Figure 12.5: *Critical Path: Delay through combinatorial circuit from mux to zero flag.*

When synthesizing the BSP to ES2-0.7μ technology using AutoLogic II, the program reports the critical path is from the mux-control signal to the LZ flag. The total combinatorial delay through this path to be 36.5 ns. When synthesizing the program sequencer, AutoLogic II reports that the total combinatorial delay from rising clock edge to output is ≈ 4 ns. By adding the delay in the program sequencer to the delay in the datapath, the BSP will be able to support a clock frequency of 25 MHz.

Required Access Time to Memory

The required access time to memory can be calculated as follows. A memory write cycle is performed in two clock cycles. The first clock cycles is used to calculate the address to memory, and the second is used to write the data into memory. The calculation of an address is basically a register-register operation, hence the address will be stabile at the rising edge of the second clock cycle. Most memory requires that the write enable signal is released before the address and data becomes invalid. This can be secured by using the memory state machine in the program sequencer. The write enable signal is asserted in the first three memory state cycles during the second execution state, giving the memory 3/4 clock cycle to latch the contents of the data bus. This is illustrated in Figure 12.6. If the BSP is clocked at 20 MHz, the memory access time must be under 37.5 ns. Similar calculations can be done for memory reads.

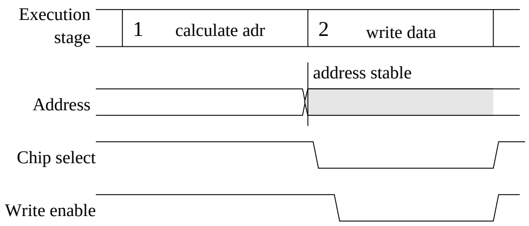


Figure 12.6: *Memory Access timing. In the first execution state the address is calculated, and in the second the data is written to memory.*

12.4 Area

After having synthesized the BSP in AutoLogic II with 0.7μ ES2 standard cells as target technology, some area estimates are accomplished. These reports describes the area consumption of the cells used in the design, not including routing area. The area consumption of the individual units in the BSP is tabulated in Table 12.2. As it can be seen the total area consumption of the BSP is estimated to approximately 4 mm^2.

Unit	**Cost** mm^2
Address Decoder	0.007
Datapath	1.487
Fifo Controller	0.078
Program Sequencer	0.387
Bitstream Receiver	0.898
Serial Port	0.639
Bus Drivers	0.056
BSP	3.558

Table 12.2: *Area estimates of BSP.*

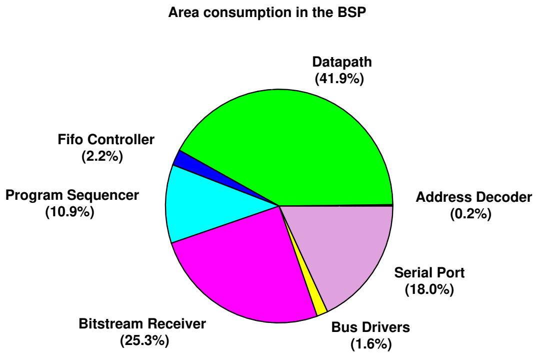


Figure 12.7: *Pie chart of area comsumption in BSP. Based on Table 12.2.*

The bitstream receiver contributes with approximately 1/4 of the total area in the BSP. The 2×96-bit buffering in the bitstream receiver is responsible for this significant area consumption. By decreasing the length of the buffers to for example 2×48-bit, the Handle Incoming bitstream interrupt has to be serviced twice often. In worst-case the Handle Incoming bitstream interrupt consumes approximately 5% of the total execution time, see Figure 12.1. Whether to use 2×96-bit or 2×48-bit buffers becomes a tradeoff between area consumption and clock frequency. Similar assessments can be performed for the serial port. The detailed area report from AutoLogic is tabulated in Appendix E.

In addition to the 4 mm^2 required for the BSP itself, some memory is needed to hold the program and data. The BSP is designed with a 16 bit address and 16 bit databus, enabling it for addressing 64 kword data. Even though it is possible to address 64 kword memory, it is not necessary to equip the BSP with more that 16 kword memory in order to decode a Layer III bitstream.

Depending on the clock frequency of the BSP this memory can be either static memory or dynamic memory. If the BSP is clocked at 25 MHz, which is the upper bound, a write operation to memory must be performed in 30 ns, at this speed most external SRAM can be used. In ES2 it is possible to place memory on-chip. The cost of including the memory on-chip is of cause a larger die area. As an example of ES2 memory, a 1k$\times$16 bit memory has been made using the ES2 megacell generator, [ES2, 1994].

ES2 0.7μ **1k×16 bit SRAM**	
Access time	26.12 ns
Area	4.76 mm^2

If 16 kword RAM was to be placed on-chip using ES2 megacells, it is clear that it will contribute with a significant increase in area.

If the BSP is clocked at 25 MHz it is clear that static memory (SRAM) is the only feasible solution. If it was chosen to loosen the demands to the BSP, by for example chosing to accept that the BSP only should be able to decode bitstreams with bitrates at 196 kbit/s and below, the clock frequency could be lowered. If the BSP was clocked with 12 MHz instead, the access time to memory would be 62 ns, at this rate it is possible to use dynamic memory (DRAM) instead, which is far more area efficient.

12.5 Demands to DSP

In Appendix A an operation count for the algorithms to be performed on a DSP is conducted. It is stated that approximately 11 MIPS are required, and that MAC operations counts for 58 % of the total. The requirements to put on a DSP are moderate and therefore an inexpensive DSP can be chosen. If the Layer III algorithm and the application allow for a fixed point implementation, an ADSP 2101 (20 MIPS) will be an appropriate choice. The ADSP 2101 provides single cycle execution of all required instructions and will therefore offer a fair margin, in terms of MIPS, for an implementation. If a floating point implementation is required, the TMS320C30 (16.5 MIPS), offering single cycle execution of the required instructions, will be adequate. A small drawback is that the complexity will increase a little because a conversion from integers to IEEE floating point format must be conducted on all the incoming data. In spite of the complexity increase, the TMS320C30 seems able to offer the performance required.

13 Conclusion

13.1 Summary

The topic for this project was to design and implement a microprocessor to be used in an MPEG-1/Audio Layer III decoder. The microprocessor was intended to process a Layer III bitstream and leave the remaining algorithms for a general purpose DSP. From this definition the term "Bitstream Processor" (BSP) was designated for the microprocessor.

The motivation to design a BSP was based on a survey of existing hardware implementations of Layer III decoders. The design strategy generally applied for these was to base the decoder on a powerful and expensive DSP. In this project the thesis was that the design and implementation of an inexpensive BSP would only require a small and inexpensive general purpose DSP, and thereby reduce the cost of designing a Layer III decoder.

In the project, the Layer III decoding scheme was studied in order to obtain a precise definition of the algorithms to be executed on the BSP. From the study it was evident that the Layer III scheme is a complex compound of DSP friendly algorithms and data dependent interpretation and unpacking algorithms. A reordering of the decoding scheme was performed in order to obtain a proper partitioning of the algorithms. This solution offered a separation, leaving only DSP friendly algorithms for the DSP. A complexity analysis conducted for the DSP part had proven the partitioning appropriate. The demands to put on the DSP could be estimated to approximately 11 MIPS. This demand can be met by inexpensive general purpose DSP's.

Based on an exact definition of the algorithms to be executed on the BSP, a design specification for the Bitstream Processor was established. A complexity analysis of the algorithms to be performed on the BSP was conducted in order to estimate the processing requirements to put on the BSP. The demands to put on the Bitstream Processor was estimated to approximately 3.5 MIPS.

In order to establish a set of guidelines for the design of the Bitstream Processor an architectural survey was performed. It was chosen to base the Bitstream Processor on a traditional Von Neumann architecture offering a 16 bit databus and a 16 bit addressbus. In order to keep down the complexity of the hardware a time shared datapath utilizing 12 internal registers was chosen. The datapath was designed with an ALU offering the following operations: addition, subtraction, OR, AND, XOR, left shifts and right shifts. The program sequencer designed offers jumps, conditional jumps, procedure calls, handling of nested prioritized interrupts, bus arbitration and indivisible push and pop operations for stack use. For debugging the BSP was equipped with a trace interrupt that allowed for constantly updating the programmer with the register contents.

Two Layer III specific features were added to the architecture in order to obtain a processor efficient for executing the designated algorithms. First a bitstream receiver was designed. This unit provided the processor with the capability of receiving data bursts at 1.5 Mbit/s, detect synchronization words and perform byte aligning on data before saving in memory. A fifo register was mapped into the datapath to provide instant access to an arbitrary number of data bits. For interfacing to a DSP a serial port capable of transmitting 4 Mbit/s was designed. In order to perform 3.5 MIPS on the architecture chosen a clock frequency of 15 MHz was estimated.

It was chosen to design an assembly programming language for the BSP in order to ensure fast and correct programming of the designated algorithms. The assembly language was designed for algebraic notation, which provided an easy to use language.

The BSP and the algorithms were tested in conjunction using Mentor Graphics QHsim. The test was conducted using both compliance test bitstreams and bitstreams based on music recordings. For validation, an existing PC based Layer III decoder was used as reference. From the tests it could be concluded that both the BSP and the algorithms written were capable of processing a bitstream correctly. It could also be concluded that the BSP was capable of managing bitstreams presenting close to worst-case characteristics. An actual worst-case test could not be conducted, because frames applying for the worst-case situation were not obtainable.

The project concluded with a synthesis in 0.7μ standard cells from ES2. Analysis of the critical path in the synthesized BSP showed that a clock frequency of 25 MHz was supported. It was also concluded that the BSP would require approximately 3.6 mm^2 die area.

13.2 Conclusion

The result of the project is a functional validated VHDL implementation of the proposed BSP. The BSP running the intended software has proven to be capable of processing both compliance test bitstreams and bitstreams based on music recordings. Though this testing method cannot guarantee against the presence of errors, it allows for confidence in the functionality of both VHDL description and the algorithms implemented.

During the design of the BSP, a strategy initially placing all complexity in software was followed. The first goal was to obtain a general purpose microprocessor capable of performing basic operations. Next, Layer III specific units were considered for implementation in the architecture in order to decrease the high software complexity. This strategy, of starting with a general purpose microprocessor and secondly approach an actual ASIC design, resulted in the final design of the BSP. An illustration of the design process is illustrated in Figure 13.1. The design approach initiated in the lower left corner and ended in the shaded area. The design strategy was found successful in that a reasonable compromise between hardware and software complexity was found in a structured manner. Using the strategy made the effect of adding Layer III specific units measurable.

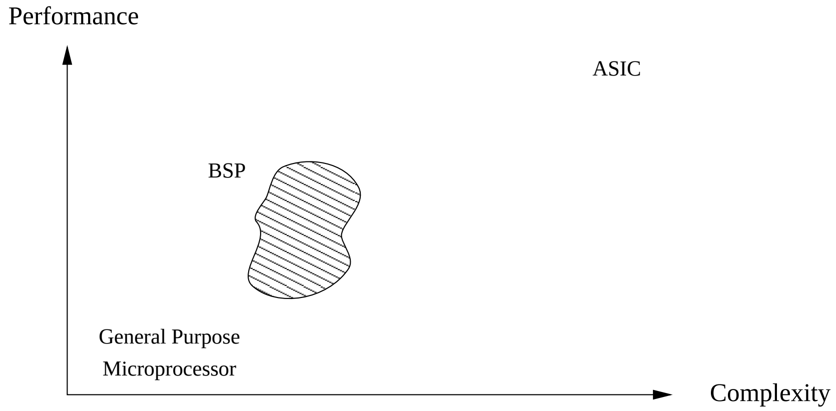


Figure 13.1: *An illustration showing how the BSP fits in between a general purpose microprocessor solution and an ASIC*

It was chosen to base the BSP on a time shared Von Neumann architecture. In short the concept covers an architecture utilizing only one data and addressbus, and a time shared datapath for arithmetics and data and program address calculations. The architecture has proven it self adequate for the application, because the Layer III algorithms are incapable of efficiently exploiting the benefits from one or two separate databusses. Another indication of the adequacy is found in the fact that the architecture and the technology chosen were capable of meeting timing the demands.

The architecture chosen offers much freedom for implementing the designated instructions. It was chosen to implement variable length instructions, because long and short instructions thereby could be mixed without introducing time penalties for shorter instructions. An example of the gain obtained was shown by conducting an instruction count for the BSP decoding a frame. An effective MIPS count of 5.3 was found, which was to be compared to 2.9 MIPS, if equal length instructions were implemented. In effect a gain of approximately 75% was obtained by implementing variable length instructions.

Variable length instructions made implementation of complex instructions attractive. Complex instructions, i.e. instructions compounded by more than one operation, can easily be implemented in the BSP without introducing the time penalties for shorter instructions. An example was the introduction of the mask instruction, which is compounded by an addition and an AND operation. The implementation produced a speedup of 50% compared to executing two separate instructions, and only required a minor modification of the program sequencer in order to work. The penalty introduced by using variable length instructions is a complexity increase of the program sequencer and a small increase of the die area. The latter is however acceptable because the program sequencer only counts for approximately 10% of the total BSP die area.

The BSP was designed to support interrupts. This feature makes the processor easy to program when considering non-predictable events. The use of interrupts in the BSP is considered the better choice for servicing I/O ports and updating of the fifo register. Alternatively, polling could be implemented, but this solution would entail much instruction overhead. An alternative could be an implementation of complex hardware to handle the specific events. This solution was rejected because a substantial increase in die area consumption would follow. Interrupt handling implies a minor increase of the die area required, but this drawback is well compensated for in that a good compromise between hardware and software complexity is made obtainable.

One of the steps away from the general purpose microprocessor was the implementation of a fifo register in the datapath. The fifo register provides instant access to an arbitrary number of bits. The fifo register implementation is considered an elegant solution to accessing maindata in that a minimum of maindata pointer handling is required, and both fifo register reloading and data pointer handling is placed in an interrupt routine. When a specific number of bits is required the fifo register provides them using only one instruction. Profiling of the interrupt routine has shown that only a small amount of processing time is spent on updating the fifo register though the fifo register is utilized very often. Synthesis shows that the register requires only a small percentage of the total die area.

The designed bitstream receiver is capable of receiving an incoming bitstream at the highest allowable bitrate within MPEG-1, i.e. 1.5 Mbit/s. The bitstream receiver is a good example of the hardware–software design strategy followed in the project. The bitstream receiver is a slave like hardware unit which is software controlled. The significant cost of the bitstream receiver is caused by the die area consumed by the two buffers present. The size of the buffers depends on the bitrate and clock frequency. Hence the demand on buffer size can be loosened if the BSP is applied in a specific application having a lower bitrate or a higher clock frequency. The buffers were designed for a clock frequency at 15 MHz, but if the BSP is clocked with 25 MHz, the area consumption of the bitstream receiver would decrease by a factor 25/15.

A serial port has been constructed for transmitting the decoded data to the DSP. The reason for designing a serial port is that it provides a simple interface from the BSP to the DSP. A serial port capable of meeting a bitrate demand of 3 Mbit/s was however obtained at the expense of a die area consumption corresponding to 20 % of the die area consumed by the BSP. In addition profiling has shown that the serial port takes 10 % of the total execution time. Comparing this performance to e.g. the performance of the serial port provided in an ADSP 21xx, where a word can be transmitted using only a single instruction cycle, the 10 % is too much. Hence an alternatively solution should be considered. An appropriate way to reduce the instruction overhead, if required, will be by dedicating a Direct Memory Access (DMA) controller for the serial port. By using a DMA controller the area consumption would decrease significantly.

A proper method for evaluating the BSP design is by the instruction set implemented. For this purpose a series of profilings of the BSP processing a variety of bitstreams was conducted. Two results from this profiling call for attention. The BSP uses a big share of the processing time on processing Huffman code bits. The processing demands set by Huffman decoding were predictable because a large amount of data is processed in this algorithm, but the prominence gives reason for reevaluating the instruction set applied. Study of the assembly program written for Huffman decoding reveiles no extensive use of certain instruction sequences and no particular capers have been applied to work around simple operations. A considerable reduction of the time spend on Huffman decoding can therefore not be obtained by means of an instruction set optimization. A noticeable time is spend on servicing the serial port. The time spend is a consequence of transmitting 200,000 words per second and that each word demands 4 instructions for transmission.

An algebraic assembly language was designed for the BSP. To translate the assembly programs to machine code executable on the BSP, an algebraic assembler was constructed. Contrary to traditional assembly language the algebraic assembly language is easier to understand and hence faster to produce assembly programs in. The construction of the assembler therefore made it possible to write the programs to the entire set of algorithms belonging to the bitstream part, within the time allocated for the project. Because of the waste amount of assembly code needed to implement the algorithms performed by the BSP, this would not have been the case if the algorithms should have been implemented directly in machine code.

The fact that the instruction set constructed for the BSP covers a broad spectrum of ordinary instructions and do not contain any dedicated instructions, provides for some reprogrammability. This ensures that if any of the sub algorithms would change, it is possible to perform a reprogramming of the BSP using the existing instructions set.

The flexibility of the BSP can be evaluated by considering the ease with which the BSP would work in a different application. Several features in the BSP provides for flexibility. For instance, it is possible to exclude the constructed serial port if it was not required in a specific application. The flexibility is also improved by supporting bus arbitration, which makes it possible for the BSP to operate as slave in a shared bus architecture. In addition the BSP supports different memory speed grades. This feature is facilitated by the program sequencer. Because the program sequencer is implemented as a hierarchal state machine, it is possible to adapt the timing to the specific memory used. Besides providing for flexibility, the BSP is capable of adapting to future expansions. By facilitating external interrupts the BSP supports for future expansions. The BSP is also supplied with three user defined ports, which could be used for connecting e.g. a display to the processor.

The BSP is synthesized in 0.7μ standard cells, and will require a die area of approximately 3.6 mm^2 in the technology chosen. When considering that the project only has involved the first phases of the BSP development, and an actual processor implementation could be expected to take place at least a year from now, 0.7μ standard cells will be considered obsolete. 0.35μ is presumably the De facto technology a year from now and will allow for an implementation only requiring 0.9 mm^2 die area and a considerable increase of the upper bound clock frequency.

Though I/O paddings and routing has not been considered in the die area estimate, the BSP can be considered as an inexpensive processor, demanding a low die area.

The thesis, saying that an inexpensive MPEG-1/Audio Layer III decoder can be constructed using a BSP and an inexpensive general purpose DSP has been accepted. The 11 MIPS demand for the DSP can be met by the ADSP 2101 for an fixed point implementation and by the TMS320C30 for a floating point implementation. Both DSP's are considered inexpensive processors today.

Bibliography

[ADSP-21xx, 1995] ADSP-21xx (1995). *ADSP-2100 Family User's Manual*. Analog Devices, third edition.

[AMD, 1997] AMD, I. (1997). Advanced fifth generation technologies for 586-class performance. World Wide Web. http://www.amd.com/products/cpg/nx586/nx586brf.html.

[Burd, 1997] Burd, T. (1997). CPU Information Center. Department of Electrical Engineering and Computer Science at Berkeley University. WWW page – http://infopad.eecs.berkeley.edu/CIC/.

[Clements, 1991] Clements, A. (1991). *The Principles of Computer Hardware*. Oxford Science Publications, second edition.

[Clements, 1992] Clements, A. (1992). *Microprocessor Systems Design*. PWS-Kent Publishing Company, Boston, second edition.

[ES2, 1994] ES2 (1994). *ES2 Library Databook*. European Silicon Structures.

[Furui, 1989] Furui, S. (1989). *Digital Speech Processing, Synthesis, and Recognition*. Marcel Dekker.

[Hasan and Sain, 1995] Hasan, W. and Sain, S. M. (1995). Hardware Design and VLSI Implementation of a Byte-Wise CRC Generator Chip. *IEEE Transactions on Consumer Electronics*, 41(1):195–.

[Hashemian, 1994] Hashemian, R. (1994). Design and hardware implementation of a memory efficient huffman decoding. *IEEE Transactions on Consumer Electronics*, 40(3):345–351.

[Hashemian, 1995] Hashemian, R. (1995). Memory Efficient and High-speed Search Huffman Coding. *IEEE Transactions on Communications*, 43(10):2576–2581.

[Hennessy and Patterson, 1990] Hennessy, J. L. and Patterson, D. A. (1990). *Computer Architecture – A Quantitative Approach*. Morgan Kaufmann Publishers, Inc.

[Hennessy and Patterson, 1994] Hennessy, J. L. and Patterson, D. A. (1994). *Computer Organization & Design, The Hardware/Software Interface*. Morgan Kaufmann Publishers, Inc.

[Hsieh and Kim, 1996] Hsieh, C.-T. and Kim, S. P. (1996). A Concurrent Memory-Efficient VLC Decoder for MPEG Applications. *IEEE Transactions on Consumer Electronics*, 42(3):439–446.

[ISO/IEC 11172-3, 1993] ISO/IEC 11172-3 (1993). *Information technology – Coding of moving pictures and associated audio for digital storage media at up to about 1,5 Mbit/s–Audio part.*, 1 edition.

[Oppenheim and Schafer, 1989] Oppenheim, A. V. and Schafer, R. W. (1989). *Discrete-Time Signal Processing*. Prentice-Hall International, Inc., Englewood Cliffs, N.J. 07632, international edition.

[Pan, 1995] Pan, D. (Summer 1995). A Tutorial on MPEG/Audio Compression. *IEEE Multimedia*, pages 60–74.

[Papamichalis et al., 1995] Papamichalis, P., Reimer, J., and Rowlands, J. (1995). System and Algorithm Implementation Techniques on the TMS320 Family. In *ICASSP IEEE*, volume 5, pages 2845–2848.

[Popp, 1997] Popp, H. (1997). Frequently Asked Questions about MPEG Audio Layer III. World Wide Web. http://www.iis.fhg.de/departs/amm/layer3/sw/index.html.

[Ramabadran and Gaitonde, 1988] Ramabadran, T. V. and Gaitonde, S. S. (1988). A Tutorial on CRC Computations. *IEEE Micro*, 8(4):62–75.

[Schiøning and Simonsen, 1996] Schiøning, J. and Simonsen, K. (1996). Design of mpeg-1 audio layer iii decoder. Master's thesis, Aalborg University.

[Shlien, 1994] Shlien, S. (1994). Guide to MPEG-1 Audio Standard. *IEEE Transactions on Broadcasting*, 40(4):206–217.

[Stabernack et al., 1994] Stabernack, B., Heinelt, C., Talmi, M., and Sinnhöfer, W. (1994). A Flexible MPEG-2 Audio Decoder and Transport Stream Interface Based on TMS320C40 and ITT MASC3500C. 225-229.

[Tanenbaum, 1990] Tanenbaum, A. S. (1990). *Structured Computer Organization*. Prentice-Hall International, third edition.

[Witte et al., 1994] Witte, F.-O., Dietz, M., and Sinnhöfer, W. (1994). Single Chip Implementation of an ISO/MPEG Layer III Decoder. In *96th Convention of the Audio Engineering Society (AES)*. Preprint.