Hardware implementation of the Front-end module in Distributed Speech Recognition system

by

Ahmad Abdel Moniem Al Sallab

A Thesis submitted to the
Faculty of Engineering at Cairo University
in Partial Fulfillment of the
Requirements for the Degree of
MASTER OF SCIENCE
in
ELECTRONICS AND COMMUNICATIONS

FACTULTY OF ENGINEERING, CAIRO UNIVERSITY
GIZA, EGYPT
April 2009

Hardware implementation of the Front-end module in Distributed Speech Recognition system

by

Ahmad Abdel Moniem Al Sallab

A Thesis submitted to the
Faculty of Engineering at Cairo University
in Partial Fulfillment of the
Requirements for the Degree of
MASTER OF SCIENCE
in
ELECTRONICS AND COMMUNICATIONS

Under supervision of

Mohsen A. Rashwan
Professor
Elec. and Com. Dept.

Hossam A. Fahmy
Assistant Professor
Elec. and Com. Dept.

FACTULTY OF ENGINEERING, CAIRO UNIVERSITY
GIZA, EGYPT
April 2009

Hardware implementation of the Front-end module in Distributed Speech Recognition system

by

Ahmad Abdel Moniem Al Sallab

A Thesis submitted to the

Faculty of Engineering at Cairo University

in Partial Fulfillment of the

Requirements for the Degree of

MASTER OF SCIENCE

in

ELECTRONICS AND COMMUNICATIONS

Approved by the

Examining Committee

Prof. Dr. Mohsen A. Rashwan, Thesis main supervisor

Prof. Dr. Mohamed Waleed Fakhr, Member

Prof. Dr. Ashraf M. El Farghaly, Member

FACTULTY OF ENGINEERING, CAIRO UNIVERSITY

GIZA, EGYPT

April 2009

Acknowledgments

First I would like to thank Allah for supporting me, guiding me, giving me strength during hard times and all other endless favors he has done to me, with which I live and survive. Then I would like to thank my parents for their love, support and trust, which gives meaning to my life. I would like to thank also my advisers, Prof. Dr. Mohsen and Dr. Hossam Fahmy for their valuable advices which always guided me to the right way, they were very kind and flexible with me.

Abstract

From human prehistory to the new media of the future, speech communication has been and will remain the dominant way of human social bonding and information exchange. To understand speech, a human considers not only the specific information conveyed to the ear, but also the context in which the information is being discussed. For this reason, people can still understand the spoken language even if the speech signal is corrupted by noise. Hence, recognition and understanding the context of spontaneous speech remains the goal of speech signal processing research for many years [1].

New system architectures have emerged for *Automatic Speech Recognition* (ASR) systems to deploy speech recognizers in embedded and hand held devices. Modern ASR systems can be structurally decomposed into two main parts; the acoustic *Front-end*, where the process of the feature extraction takes place and the *Back-end*, performing ASR search based on the acoustic and language models. Since most of the portable devices use a communication link, we can classify all the mobile ASR systems in terms of employing wireless communication link and the location of the front-end and back-end parts as; *Embedded Speech Recognition* Systems (ESR), *Network Speech Recognition* (NSR) and *Distributed Speech Recognition* (DSR).
The *European Telecommunications Standards Institute* (ETSI) has formed the STQ Aurora group to work on the standardization of the front end for DSR applications. Four standards emerged for the front-end specifications. All of the four standards use the *Mel-Frequency Cepstral Coefficients* (MFCC) as the features extraction algorithm in the front-end.

The main point of this thesis is the hardware implementation of the basic front end specified in the first Aurora standard (ETSI ES 201 108 V1.1.3) to be deployed in mobile hand-held devices. To meet the tight constraints of an embedded system, FPGA for prototyping and structured ASIC for mass-production style is chosen as the hardware platform to implement the design.

Taking into consideration the tight area usage constraint, some low-resources usage algorithms are used in the design, like the COordinate Rotation DIgital Computer (CORDIC) algorithm, which was used extensively to perform many functions in the system. VHDL coding, synthesis and RTL simulations are done to prove the concept of the design.

The results of this work are presented in two aspects; the first one is to compare the design to other reference hardware designs presented by FPGA manufacturers. Results show that the design presented here outperforms the reference designs in terms of hardware resources usage, with a reduction percentage of 8.9 % in some cases to 58 % in others. The second axis of evaluation was the compliance to the Aurora standard. reference features vectors of about 8 seconds of continuous speech were provided by the ETSI to prove compliance to the specifications. Results show that the final system output matches the reference vectors with average absolute error of 0.002 in some configurations to 0.004 in others.

Contents

List of Figures

List of Tables

Chapter 1

1 Introduction

Speech signal processing refers to the acquisition, manipulation, storage, transfer and output of human utterances by a computer. The main goals are the recognition, synthesis and compression of human speech. Here, we are concerned with the speech recognition part, in addition to some concepts of Speech compression that are mandatory to achieve efficient, lossless speech communication, especially on wireless media.

Speech recognition (also known as automatic speech recognition or computer speech recognition) converts spoken words to machine-readable input (for example, to key presses, using the binary code for a string of character codes). The term "voice recognition" may also be used to refer to speech recognition, but can more precisely refer to speaker recognition, which attempts to identify the person speaking, as opposed to what is being said.

In order to achieve the goal of the above definition, several models to represent the speech signal exist, so that this model can be used as the base of the computer algorithm that will handle speech recognition task. According to the accuracy of the model to represent the real speech signal, the recognition results are determined. Two main directions exist in this area; speech production models, and speech perception models. There exist many ways to analyze the speech system based on the way the speech signal is modeled, among those are: Linear Predictive Coding (LPC) for Speech production models and Mel-Frequency Cepstrum for Speech production models.

In theory, it should be possible to recognize speech directly from the digitized waveform. However, because of the large variability of the speech signal, it is a good idea to perform some form of feature extraction that would reduce that variability. Speech recognition task is originally a pattern recognition problem, where the input speech signal cannot be processed or stored as its raw form in digital samples, because this would require a large

storage for those samples. For these reasons, features extraction must be performed on the input speech signal before further processing.

There are many methods and directions in speech features extraction, according to the way the speech signal is represented. Linear Predictive Coefficients (LPC) is a way to perform speech features extraction that is more suitable in the field of speech coding. For perceptually motivated speech models, two famous speech features extraction methods are presented, which are *Mel-Frequency Cepstral Coefficients* (MFCC) and Perceptual Linear Prediction (PLP). *Mel-Frequency Cepstral Coefficients* (MFCC) is the most popular method utilized nowadays in speech recognition systems.

New system architectures have emerged for *Automatic Speech Recognition* (ASR) systems that are adapted to achieve today's requirements of speech technology applications, where the need arose to deploy speech recognizers in embedded and hand held devices. The recognition task becomes divided between client front-end part and server back-end part. This architecture has many advantages in terms of reducing the processing load on embedded devices and improving the recognition capabilities for speech recognition applications performed over communication networks like GSM or 3G networks. The main three architectures are presented here, which are *Embedded Speech Recognition* (ESR), *Network Speech Recognition* (NSR) and *Distributed Speech Recognition* (DSR).

The *European Telecommunications Standards Institute* (ETSI) has deployed a new series of standards to standardize a set of features and implementations guidelines for the main components of a *Distributed Speech Recognition* (DSR) system. The STQ Aurora group works on the standardization of a front end for DSR applications. Four standards emerged for the front-end specifications, in addition to the features compression algorithm and bit-stream framing algorithm. All of the four standards use the MFCC as the features vector in the front-end. The first standard contains the basic functionality of the Mel-Cepstrum front-end. The second standard works on improving the speech recognition results in a noisy environment; this is

referred to as the *Advanced front-end*. The third one includes a modification to enable speech reconstruction at the back-end and enhance the speech recognition for some tonal languages like Mandarin and Thai; this is referred to as the *Extended front-end*. The last standard is a merge between the second and third ones, where recognition is noise robust and in the same time speech reconstruction and tonal languages support are enabled; this is referred to as the *Extended Advanced front-end*.

The main point of this thesis is the hardware implementation of the basic front-end specified in the first Aurora standard (ETSI ES 201 108 V1.1.3 (2003-09) to be deployed in mobile hand-held devices.

1.1 Organization of the Thesis

The thesis is organized as follows:

CHAPTER 1 is an introduction to the *Automatic Speech Recognition* (ASR) systems in general and *Distributed Speech Recognition* (DSR) systems in particular. In addition, the four Aurora DSR Front-end standards are introduced in brief.

CHAPTER 2 is an overview of the design of VLSI systems, where VLSI design flow and VLSI design styles are introduced, with emphasis on three styles of particular interest; which are Digital Signal Processors (DSP), Field Programmable Gate Arrays (FPGA) and Application Specific Integrated Circuits (ASIC).

CHAPTER 3 is a comparative study between the three VLSI design styles introduced from the point of design time, cost, power consumption, flexibility, re-programmability…etc. A conclusion is drawn at the end of the chapter to choose the best style that suits our design goals.

CHAPTER 4 gives detailed discussion of the design and hardware implementation of the front-end system in the Aurora specifications. First the design constraints are introduced, and then the static and dynamic architectures are discussed. Every module is explained in details. And finally, the overall

system time and hardware utilization of the system is presented. A quick study is made of the effect of run-time configurability on the chip design.

CHAPTER 5 presents the testing and simulation results of the system versus the reference Aurora system to prove standard compliance. Three tests were performed at different levels of the system outputs, and their results are presented.

CHAPTER 6 shows the system performance versus some reference designs and benchmarks in the market.

Appendix A contains some basic information about essential algorithms and concepts used in the thesis, like CORDIC algorithm, radix-2 FFT algorithm and fixed and floating point notations.

1.2 Speech Signal Representation and Modeling

1.2.1 Speech production in human being

In general, speech is a sound wave created by vibration that is propagated in the air. Speech is produced from human being when a source of air flow at the vocal cords passes by the time-varying vocal tract. Manner of articulation describes how the tongue, lips, and other speech organs are involved in making a sound make contact.

Figure 1: Human Vocal Tract [6]

Acoustic theory analyzes the physical laws that govern the propagation of sound in the vocal tract. Such a theory should consider the three-dimensional

wave propagation, the variation of the vocal tract shape with time, losses due to heat conduction and viscous friction at the vocal tract walls, softness of the tract walls, radiation of sound at the lips, nasal coupling, and excitation of sound. While a detailed model that considers all the above is not yet available, some models provide a good approximation in practice, [1] and [6].

1.2.2 Speech digitization

Before speech can be processed by a computer it must be digitally sampled. First, the signal is captured by a microphone (or other transducer) and converted into an electrical signal, where the amplitude of the signal corresponds to the magnitude of the original pressure variation. Second, the signal is sampled at some frequency, so that only a finite number of amplitudes are recorded, stored, or transmitted, for a given period of time. Common sampling rates include 8000 Hz(samples per second) for telephone speech and 44100Hz for compact disk recordings. Shannon's law provides that frequencies up to half the sampling rate can be reconstructed from the sampled signal, so an 8000 Hz telephone signal can reconstruct frequencies up to 4000 Hz. Higher frequencies are subject to aliasing, such that a frequency of 4010 Hz cannot be distinguished from a frequency of 3990 Hz. (This same aliasing makes spinning wheels on stagecoaches appear to spin backwards on old western movies and television shows). To prevent this effect the signal is filtered to remove high frequencies before sampling. Third, the signal is quantized into one of a discrete number of levels so that only a finite number of bits is required to represent each level. This is called Analog-to-Digital (A-to-D) conversion. Thus, a telephone signal will typically carry 8000 speech samples per second, each represented by an 8-bit number, for a total of 64000 bits per second. In contrast, cellular telephone may only employ 4800 or 2400 bits per second, by using Linear Predictive Coding (LPC) or other signal compression techniques [1].

1.2.3 Speech Modeling

A very famous speech model is the *Source-Filter* model, where a speech signal is decomposed into a excitation signal (e[n]) passing by a time varying filter (h[n]) that represents the resonance of the vocal tract which changes over time.

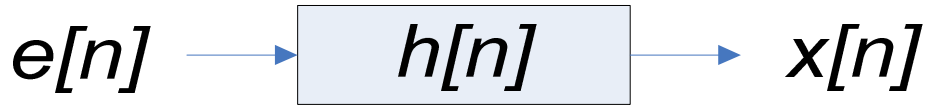


Figure 2: Source-Filter model for speech signals

Separation between the source and filter is one of the most difficult challenges in speech processing. To estimate the filter there are many methods, some of them are inspired by the speech production models (such as the Linear Predictive Coding) and others are inspired by the speech perception models (such as Mel-Frequency Cepstrum). Once the filter has been estimated, the source can be obtained by passing the speech signal through the inverse filter.

Voice is divided into two main categories: *Voiced sounds* and *Unvoiced sounds*. Voiced sounds refer to the articulatory process in which the vocal cords vibrate, while unvoiced sounds describe the pronunciation of sounds when the larynx does not vibrate. According to this classification, the above model can be modified to model the excitation signal as the sum of *Voiced excitation* and *Unvoiced excitation* as shown in the Figure 3.

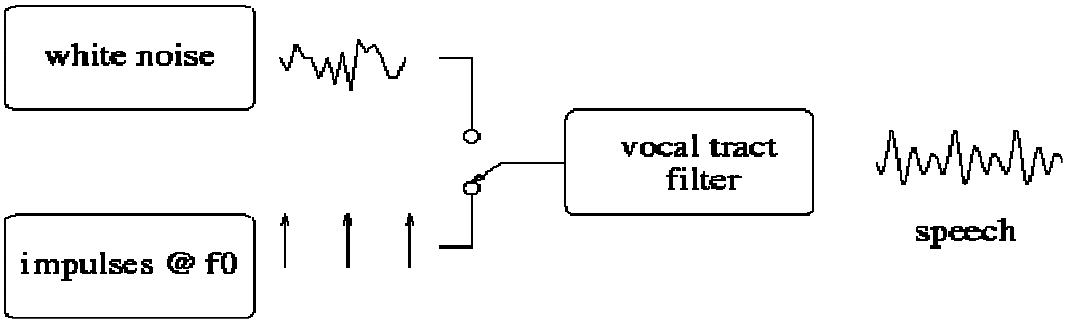


Figure 3: Source-Filter model for Voiced and Unvoiced speech

Where the white noise models the *Unvoiced* speech, while the impulse signal at the sampling frequency is the *Voiced* speech.

In general speech production models are more frequently encountered in speech coding applications, while speech perception models are more utilized in speech recognition applications. That is why Linear Prediction Coding (LPC) is common in speech coding, while *Mel-Frequency Cepstral Coefficients* (MFCC) is more dominant in speech recognition.

1.2.4 Short-term frame based spectral analysis

Due to high speech variability, continuous input speech utterance cannot be processed directly. In addition, the frequency distribution over an entire utterance does not help much more in speech recognition. Instead of processing the speech signal as a whole, a certain time frame should be set, and in which speech signal can be considered stationary.

For short-term analysis the signal must be zero outside of a defined range. This is performed by multiplying the signal with a window. This time window width is usually taken from 20 to 30 ms of speech, and the window shift, which is the time between the start of one window and the next one, is usually taken to be 10 ms. For this window, features can be extracted and processed.

There are many window shapes that can be used, like Rectangular, Hamming, Gauss, Hann and Blackmann window. The choice of certain one of the others is driven by the application and the nature of the input speech signal.

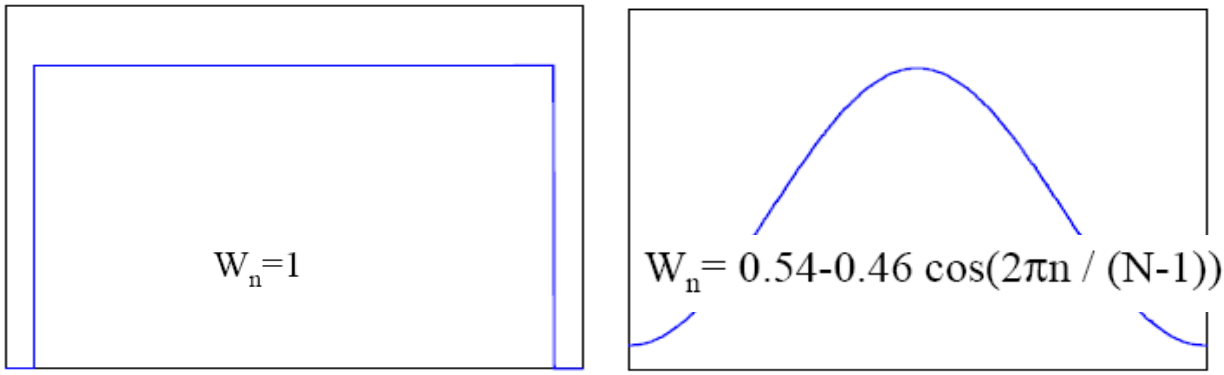


Figure 4: a) Rectangular Window b) Hamming Window

Another reason for computing the short-term spectrum is that the cochlea of the human ear performs a quasi-frequency analysis. The analysis in the cochlea takes place on a nonlinear frequency scale (known as the Bark scale or the Mel scale). This scale is approximately linear up to about 1000 Hz and is approximately logarithmic thereafter. So, in the feature extraction, it is very common to perform a frequency warping of the frequency axis after the spectral computation, [1]. This will be discussed in details in the Mel-Frequency Cepstral Coefficients features description.

1.2.5 Speech Features Extraction

Processing the digital samples of the speech signals is not a good idea, due to many reasons. First, this would require a lot of storage area to store the speech samples. Also, if this speech is to be transmitted over a network, transmitting the whole signal would require a large bandwidth. Second, because of the large variability of the speech signal, it is a good idea to extract some features of the speech that characterizes it, which would reduce that variability.

For the above reasons, some basic features of the speech signal are extracted and stored, processed or transmitted, instead of dealing with the whole speech signal.

Recall the source-filter speech production model discussed section 1.2.3, the filter h[n] can be used to model the speech signal, hence, all what we have to worry about are the coefficients of that filter, which completely model the speech system. This is the main idea of features extraction, and speech compression.

As mentioned before, two famous models exist, which are; speech production models and speech perception models. According to the model used, the filter type and coefficients are determined. For speech production model, Linear Predictive Coding (LPC) coefficients are of the most famous features. While for speech perception models, Mel-Frequency Cepstral

Coefficients (MFCCs) algorithm is the most popular features extraction algorithm used for speech recognition nowadays.

1.2.5.1 Linear Predictive Coding (LPC)

Also know as Auto Regression Coefficients (AR) algorithm (refer to [1] for more information about the algorithm, mathematical derivation and equations presented in this section). The basic idea behind linear predictive analysis is that a specific speech sample at the current time can be approximated as a linear combination of past speech samples. Through minimizing the sum of squared differences (over a finite interval) between the actual speech samples and linear predicted values a unique set of parameters or predictor coefficients can be determined. These coefficients form the basis for linear predictive analysis of speech.

We can predict that the nth sample in a sequence of speech samples is represented by the weighted sum of the p previous samples:

$$\tilde{x}[n] = \sum_{k=1}^{p} a_k x[n-k]$$

The number of samples (p) is referred to as the "order" of the LPC. As p approaches infinity, we should be able to predict the nth sample exactly. However, p is usually on the order of ten to twenty, where it can provide an accurate enough representation with a limited cost of computation. The weights on the previous samples a_k are chosen in order to minimize the squared error between the real sample and its predicted value. Thus, we want the error signal e[n], which is sometimes referred to as the LPC residual, to be as small as possible:

$$e[n] = x[n] - \tilde{x}[n] = x[n] - \sum_{k=1}^{p} a_k x[n-k]$$

We can take the z-transform of the above equation:

$$E(z) = X(z) - \sum_{k=1}^{p} a_k X(z) z^{-k} = X(z)[1 - \sum_{k=1}^{p} a_k z^{-k}] = X(z)A(z)$$

Thus, we can represent the error signal E(z) as the product of our original speech signal S(z) and the transfer function A(z). A(z) represents an all-zero digital filter, where the a_k coefficients correspond to the zeros in the filter's z-plane. Similarly, we can represent our original speech signal S(z) as the product of the error signal E(z) and the transfer function 1 / A(z):

$$\boxed{X(z) = \frac{E(z)}{A(z)}}$$

Where:

$$\boxed{H(z) = \frac{1}{A(z)}}$$

The transfer function H(z) represents an all-pole digital filter, where the a_k coefficients correspond to the poles in the filter's z-plane. Note that the roots of the A(z) polynomial must all lie within the unit circle to ensure stability of this filter.

In reality the actual predictor coefficients are never used in recognition, since they typical show high variance [1]. The predictor coefficients are transformed to a more robust set of parameters known as Cepstral coefficients.

1.2.5.2 Mel-Frequency Cepstral Coefficients (MFCC)

1.2.5.2.1 The Mel Frequency Scale

There are many non-linear frequency scales that approximate the sensitivity of the human ear. For example [6]:

- Constant Q: Q is the ratio of filter bandwidth over centre frequency; hence this implies an exponential form.
- Equivalent Rectangular bandwidth (ERB): The bandwidths of the auditory filters are measured
- Bark: Also derived from perception experiments
- Mel: The engineers solution

The Mel scale is a perceptual scale of pitches judged by listeners to be equal in distance from one another. The reference point between this scale and normal frequency measurement is defined by equating a 1000 Hz tone, 40 dB above the listener's threshold, with a pitch of 1000 Mels. Above about 500 Hz, larger and larger intervals are judged by listeners to produce equal pitch increments. As a result, four octaves on the hertz scale above 500 Hz are judged to comprise about two octaves on the Mel scale. The name Mel comes from the word melody to indicate that the scale is based on pitch comparisons [6]

$$B(f) = 1125 \log_e (1 + f / 700)$$

Where *B(f)* is the frequency in Mels, while f is the frequency in Hz.

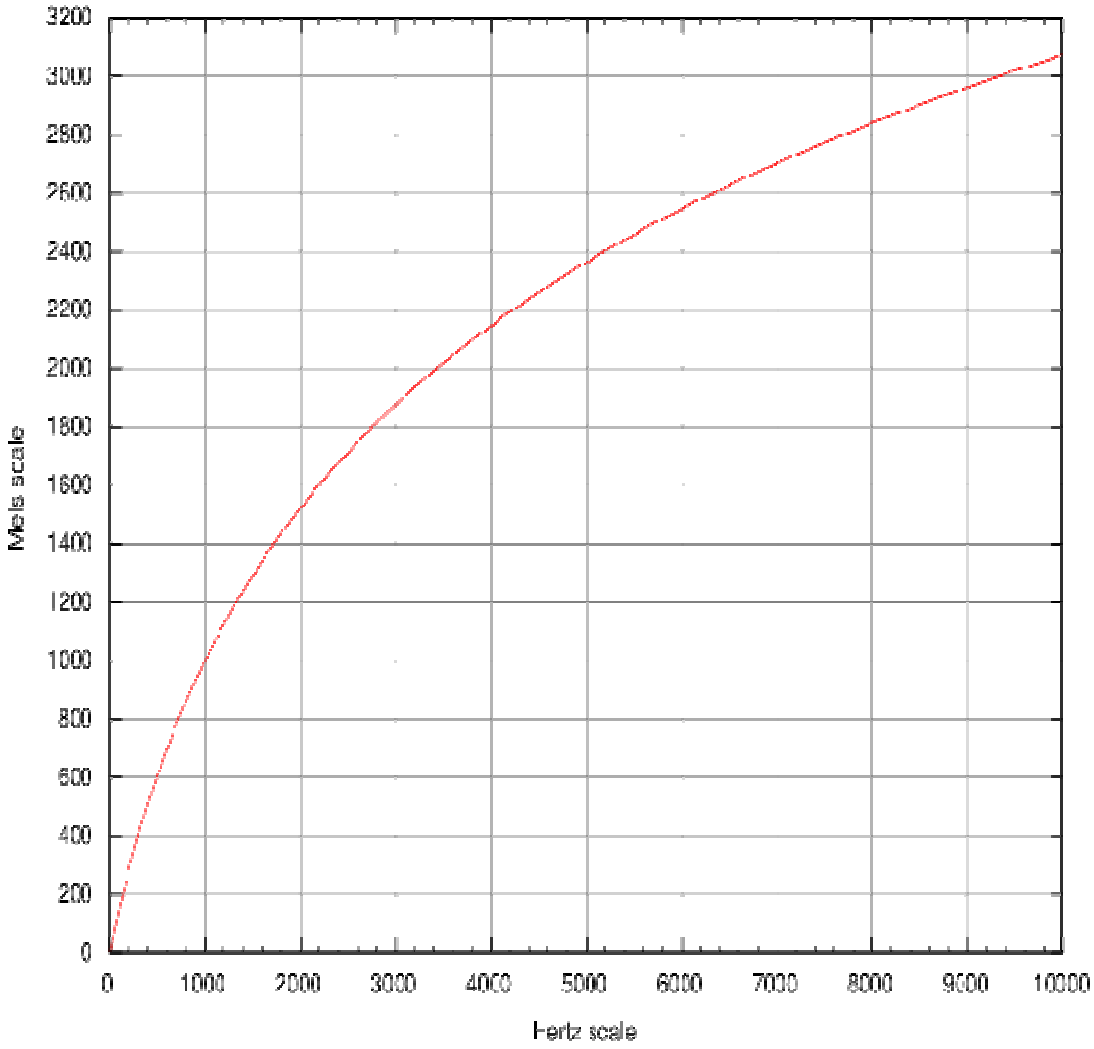


Figure 5: Hertz versus Mel Scales [6]

Experimentally, it was found that the human ear has a set of filter banks that can be perfectly represented on the Mel scale.

1.2.5.2.2 The Cepstrum

The *Cepstrum* is the result of taking the Fourier transform (FT) of the decibel spectrum as if it were a time signal. Its name was derived by reversing the first four letters of "spectrum". There is a complex Cepstrum and a real Cepstrum.

1.2.5.2.3 The Mel-Frequency Cepstrum

The difference between the normal and Mel Cepstrum is that a non-linear scale is used, which simulates the auditory system, which is the Mel-Frequency scale. The block diagram of the MFCC algorithm is shown in Figure 6 [8]. For more information about the algorithm and the underlying mathematics and equations, please refer to [1] and [8].

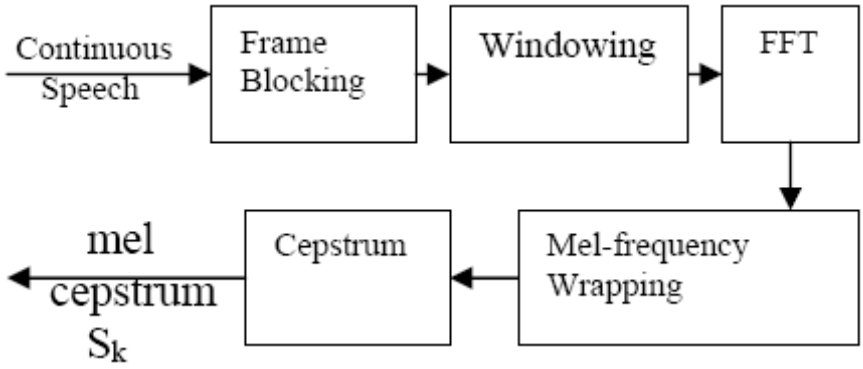


Figure 6: The Mel-Frequency Cepstral Coefficients Algorithm [8]

First, take the DFT of the input speech frame signal:

$$X_a[k] = \sum_{n=0}^{N-1} x[n] e^{-j2\pi nk/N}, 0 \le k \le N-1$$

We define a filter bank with M filters (m=1,2,…M), where m is traiangular filter given by:

$$H_m[k] = \begin{cases} 0 & k < f[m-1] \\ \dfrac{(k - f[m-1])}{(f[m] - f[m-1])} & f[m-1] \le k \le f[m] \\ \dfrac{(f[m+1] - k)}{(f[m+1] - f[m])} & f[m] \le k \le f[m+1] \\ 0 & k < f[m-1] \end{cases}$$

Let's define f_l and f_h as the lowest and highest frequencies of the filter bank in Hz, *Fs* as the sampling frequency in Hz, *M* as the number of filters and *N* as the size of the FFT. The boundary points *f[m]* are uniformly spaced in the Mel scale:

$$f[m] = \left(\frac{N}{F_s}\right) B^{-1}\left(B(f_l) + m\frac{B(f_h) - B(f_l)}{M+1}\right)$$

Where the Mel scale B is given by:

$$B(f) = 1125 \log_e(1 + f/700)$$

And the inverse Mel equation is:

$$B^{-1}(f) = 700(\exp(f/1125) - 1)$$

The filter banks are shown in the Figure 7.

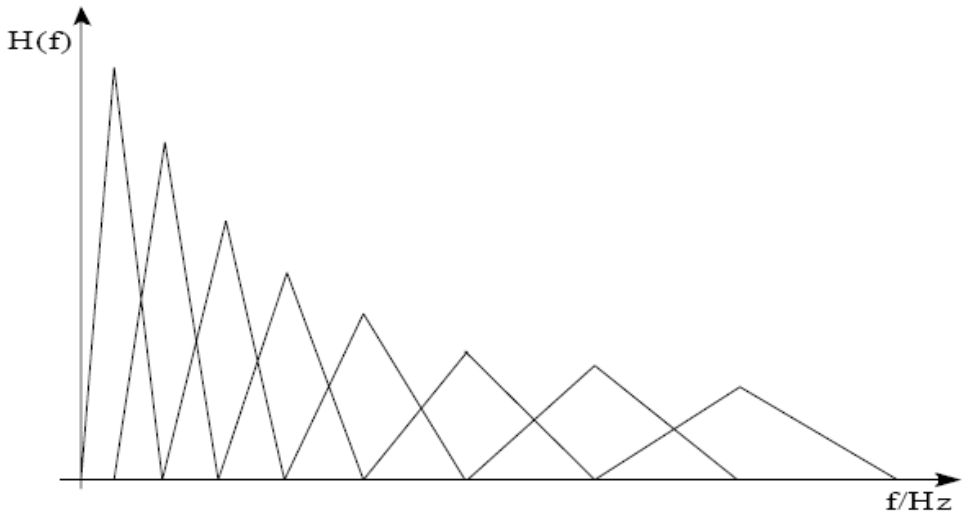


Figure 7: Mel-Filter banks

Then, compute the log-energy at the output of each filter:

$$S[m] = \ln\left[\sum_{k=0}^{N-1} |X_a[k]|^2 H_m[k]\right], \qquad 0 < \mathrm{m} \le \mathrm{M}$$

The Mel-Frequency Cepstrum is then the Discrete Cosine Transform (DCT) of the *M* filter outputs:

$$c[n] = \sum_{m=0}^{M-1} S[m]\cos(\pi n(m-1)/2)/M), \qquad 0 \le n < L$$

Where *M* varies for different implementations from 24 to 40. For speech recognition, typically only the first 13 Cepstrum coefficients are used [1]

1.3 Automatic Speech Recognition (ASR) System

Till now, we have discussed the speech signal representation and models used in speech processing in general. Now we should examine the speech recognition area of speech signal processing in more details. Speech recognition systems are commonly referred to as *Automatic Speech Recognition Systems* (ASR).

Speech recognition is basically a pattern recognition problem. A source-channel model is used to formulate speech recognition problems. As illustrated Figure 8, the speaker's mind decides the word sequence **W** that is delivered through his/her test generator. The source is passed through a noisy communication channel that consists of the speaker's vocal apparatus to produce the speech waveform and the speech signal processing component of the speech recognizer. Finally the speech decoder aims to decode the acoustic signal **O** into the word sequence **W***, which is hopefully close to the original word sequence **W**. **O** is the acoustic observations vector, or the features vector.

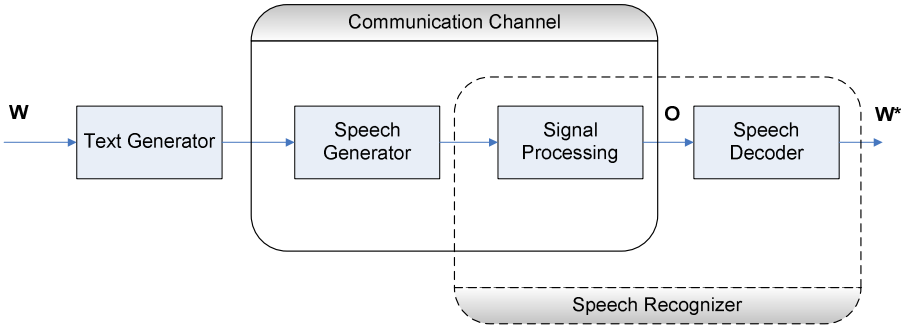


Figure 8: A source-channel model for a speech recognition system

Following the Bayesian approach applied to ASR, the best estimation for the word sequence can be given by:

$$W^* = \arg\max_W P(W|\boldsymbol{O}) = \arg\max_W \frac{P(\boldsymbol{O}|W)P(W)}{P(\boldsymbol{O})}$$

In order to generate an output the speech recognizer has basically to perform the following operations:

- Extract acoustic observations (features) out of the spoken utterance.
- Estimate *P(W)* - the probability of individual word sequence to happen, regardless acoustic observations.
- Estimate *P(O|W)* - the likelihood that the particular set of features originates from a certain sequence of words.
- Find word sequence that delivers the maximum likelihood defined in the equation above.

A typical automatic speech recognition system is shown in the Figure 9. The basic components of the ASR system are the ones in the box.

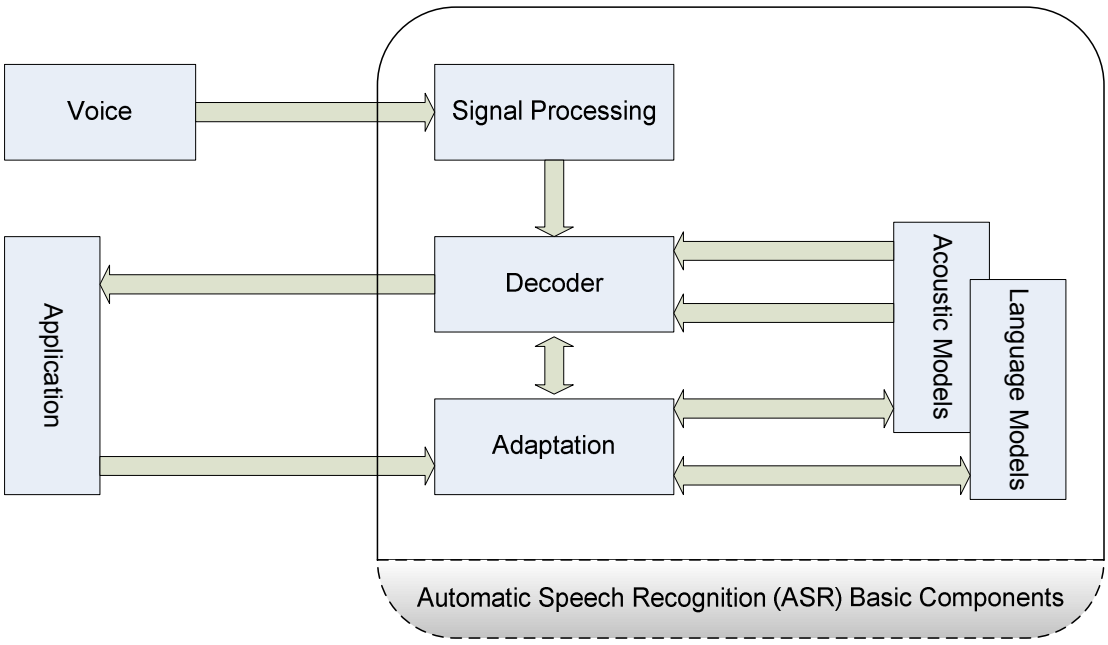


Figure 9: Basic Architecture of Automatic Speech Recognition (ASR) System

Application interface with the decoder to get the recognition results that may be used again to adapt other components in the system through the *Adaptation* component. *Acoustic models* include the representation of knowledge about acoustics microphone and environment variability, gender and dialect differences among speakers, etc.

Language models refer to a system's knowledge of what constitutes a possible word, what words are likely to co-occur, and in what sequence. The semantics and functions related to an operation a user may wish to perform may also be necessary for the language model. Many uncertainties exist in these areas, associated with speaker characteristics, speech style and rate, recognition of basic speech segments, possible words, likely words, unknown words, grammatical variation, noise interference, nonnative accents, and confidence scoring of results. A successful speech recognition system must contend with all of these uncertainties. The acoustic uncertainties of different accents and speaker styles of individual speakers are compounded by the lexical and grammatical complexity and variations of the spoken language, which are all represented in the language model.

The speech signal is processed in the *Signal Processing* module that extracts salient feature vectors for the decoder. The decoder uses both acoustic and language models to generate the word sequence that has the maximum posterior probability for the input feature vectors. It can also provide information needed for the adaptation component to modify either the acoustic or language models so that improved performance can be obtained.

A more detailed representation of the ASR system is shown in Figure 10. Where the function of each module is mentioned according to the maximum likelihood probability equation mentioned earlier in this section. The term *P(W)* is determined by the language model. It can be either rule based or of statistical nature. In the later case the probability of the word sequence is approximated through the occurrence frequencies of individual words (often depending on the previous one or two words) in some predefined database. The likelihoods *P(O|W)* are estimated on most state-of the- art recognizers using

HMM based acoustic models. Here every word *wj* is composed of a set of acoustic units like phonemes, triphones or syllables, i.e. *wj = (u1 ∪ u2…)*. And every unit u_k is modeled by a chain of states s_j with associated emission probability density functions $p(x \mid s_j)$. These densities are usually given by a mixture of diagonal covariance Gaussians, i.e. $p(x \mid s_j) = \sum_{k=1}^{M} b_{mj} N(x, \mu_{mj}, \Sigma_{mj})$. The computation of the final likelihood *P(O|W)* is performed by combining the state emission likelihoods $p(o_t \mid s_j)$ and state transition probabilities. The parameters of acoustic models such as state transition probabilities, means μ_{mj}, variances Σ_{mj} and weights b_{mj} of Gaussian mixtures are estimated on the training stage and also have to be stored. The total number of Gaussians to be used depends on the design of the recognizer.

Finally, armed with both $p(o_t \mid s_j)$ and *P(W)*, we need an effective algorithm to explore all HMM states of all words over all word combinations. Usually modified versions of the Viterbi algorithm are employed to determine the best word sequence in the relevant lexical tree.

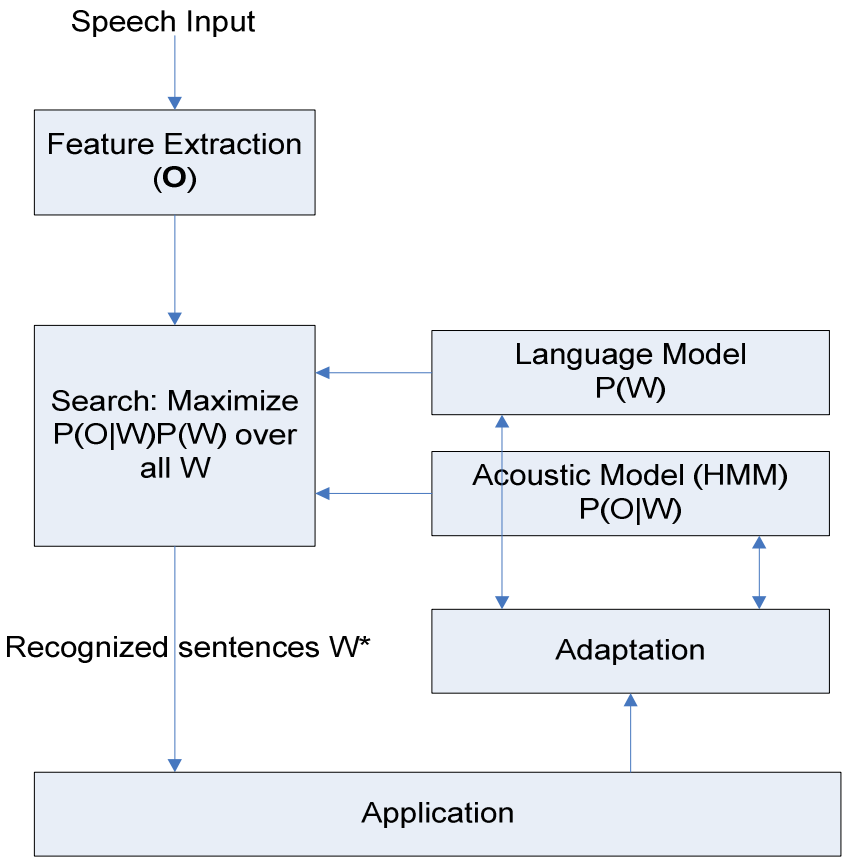


Figure 10: Detailed ASR system

1.4 *Automatic Speech Recognition Systems for Mobile and Embedded Devices*

The past decade has witnessed an unprecedented developing of the telecommunication industry. The today's mobile technologies have far overcome person-to-person communication to *Wide Area Networks* (WAN), such as 3, 3.5, 3.75 and even 4G networks. *Wireless Local Area Networks* (WLAN) based on the IEEE 802.11 specifications also known as Wi-Fi spots became widely available. With its high data rates, Wi-Fi makes possible such applications as *Voice over IP* (VoIP) or video conferencing. Alongside with expansion of the network technologies, the client devices have been developing at the same speed. Also PDAs are getting more and more popular.

Of course such a perfect infrastructure gave rise for the development of many new data services for the handheld devices. However, the user interface, which has definitely improved over the last years, still limits the usability of the

mobile devices. Typing is uncomfortable. In addition, in the case of car driving, it becomes an issue of safety to use the keypad.

Users prefer to use their natural languages to issue an order to these devices. The natural way to solve this problem consists in using speech recognition technology. As a result, new speech recognition systems have been developed. Desktop or PC applications of speech recognition are not suitable to run on embedded devices, due to the highly variable acoustic environment in the mobile domain and very limited resources available on the handheld terminals.

Note that; for more information about this section, please refer to [7].

1.4.1 Main differences between Desktop Speech Recognition and Mobile ASR

Mobile ASR faces the following problems:

- Limited storage and memory on-chip in embedded devices. Recall the discussion of the generic ASR system in section 1.3, where *P(W)* is somehow determined though the *Language Models* of the ASR system, which could be either rule based or statistical model. The main shortcoming of the statistical language models from the mobile ASR viewpoint is the number of parameters to be stored, which may be as gross as hundreds of megabytes for very *Large Vocabulary* (LV) tasks.

  Also, the likelihoods *P(O | W)* are estimated using HMM based *acoustic models*, where the states of the system s_j are associated with emission probability density functions $p(x \mid s_j)$ which are usually given by a mixture of diagonal covariance Gaussians, $p(x \mid s_j) = \sum_{k=1}^{M} b_{mj} N(x, \mu_{mj}, \Sigma_{mj})$. The total number of Gaussians to be used depends on the design of the recognizer. However, even for a digit recognition task ending up with about one thousand 39-dimensional mixtures is a common situation, which also embarrasses a compact implementation of the ASR in a mobile device [7].

- Slow processors clock speeds.
- Limited fixed-point arithmetic, while floating point arithmetic is frequently encountered in such applications.
- High power consumption of such algorithms.

1.4.2 Modern ASR Systems

Modern ASR systems can be structurally decomposed into two main parts [7]:

- The acoustic *Front-end*, where the process of the feature extraction takes place and
- The *Back-end*, performing ASR search (using Viterbi or similar algorithms) based on the *acoustic* and *language models*.

Since most of the portable devices use a communication link, we can classify all the mobile ASR systems upon the location of the *front-end* and *back-end*. This allows us to distinguish three principal system structures [7]:

- ***Client-based*** architecture or ***Embedded Speech Recognition (ESR)***, where both *front-end* and *back-end* are implemented on the terminal.
- ***Server-based*** architecture or ***Network Speech Recognition (NSR)***, where speech is transmitted over the communication channel and the recognition is performed on the remote server.
- ***Client-server*** architecture or ***Distributed Speech Recognition (DSR)***, where the features are calculated on the terminal, while the classification is done on the server side.

Each approach has certain disadvantages. The implementation depends on the application needs and the terminal capabilities. Small recognition tasks are generally recommended to reside on terminals, while the large vocabulary recognition systems take advantage of the server capacities. The following sections present these architectures, the front-end role and the back-end role.

1.4.2.1 Front-end role

The task of the acoustic front-end is to extract characteristic features out of the input speech. Usually it takes in a frame of the speech signal every 10 ms

with length from 20 to 30 ms and performs certain spectral analysis. The features extraction algorithm in most speech recognition systems today is the *Mel-Frequency Cepstral Coefficients*.

1.4.2.2 Back-end role

The main function of the *back-end* part of the ASR system is to perform ASR search using the *acoustic* and *language models* as discussed in 1.3. We have to notice that the feature extraction even if not optimized, takes just about 2% of all processing time in case of the medium vocabulary and even less in large vocabulary recognition tasks[7]. The main computational burden relies in the ASR search, which is governed by two operations: the computation of Gaussians in the emission likelihoods $p(o_t \mid s_j)$ for a given frame and the token propagation, i.e. the maintenance of the information about the survivors (best paths) during the search through the lexical tree [7].

1.4.3 Embedded Speech Recognition System Architecture

In the case of client-based or embedded ASR the entire process of speech recognition is performed on the terminal, or in other words, client side (see Figure 11, refer to [7]). Embedded ASR is often the architecture of choice for PDAs for the following reasons since they have higher capabilities compared to mobile devices.

The main advantage of this architecture relies in the fact that no communication between the server and the client is needed. Thus, the ASR system is always ready for use and does not rely on the quality of the data transmission. On the other hand, the main disadvantage is that the functionality on the terminal needs to be kept to minimum requirements or small tasks due to the limited resources of the client [7].

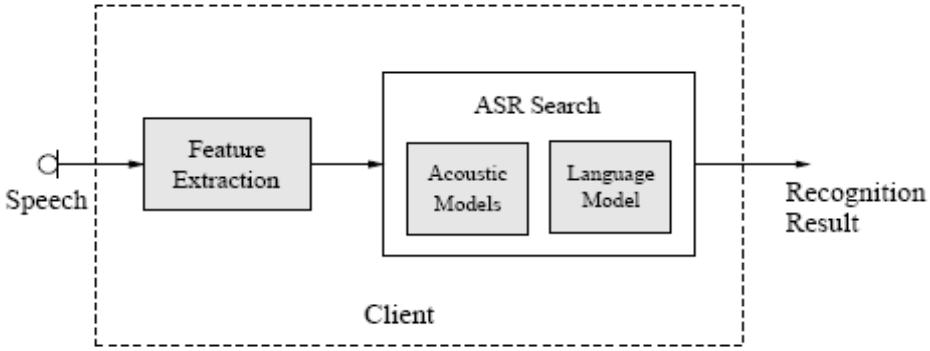


Figure 11: Client-based ASR system- Embedded Speech Recognition (ESR) [7]

1.4.4 Network Speech Recognition System Architecture

Practically all complications caused by the resource limitations of the mobile devices can be avoided shifting both ASR front-end and back-end from the terminal to the remote server. Such a *server-based* ASR architecture is referred in the literature as *Network Speech Recognition* (NSR) (see Figure 12, refer to [7]).

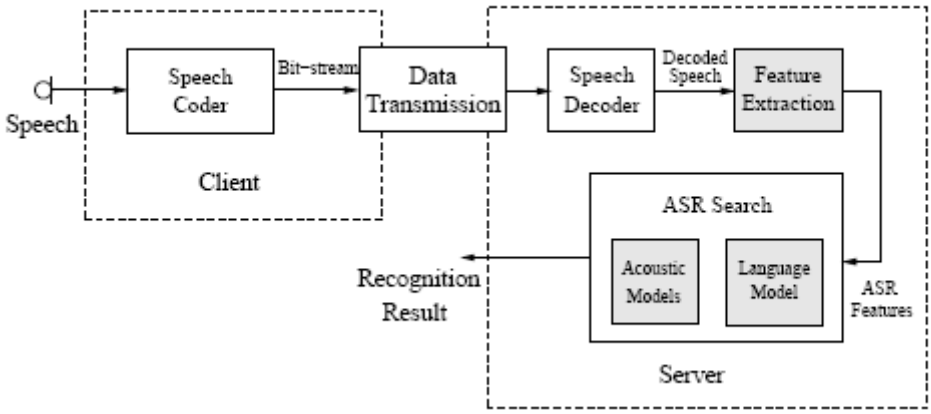


Figure 12: Server-based ASR system - Network Speech Recognition (NSR) [7]

The main advantage of NSR is its light weight terminals, of limited capabilities in contrast to ESR clients. Also, different language recognizers can be used without the need to install them on the client device.

Characteristic drawback of the NSR architecture is the performance degradation of the recognizer caused by using low bit-rate codecs, which

becomes more severe in presence of data transmission errors and background noise.

1.4.5 Distributed Speech Recognition System Architecture

Distributed speech recognition represents the client-server architecture, where one part of ASR system, which is the primary feature extraction, resides on the client, while the computation of temporal derivatives and the ASR search are performed on the remote server (see Figure 13, refer to [7]).

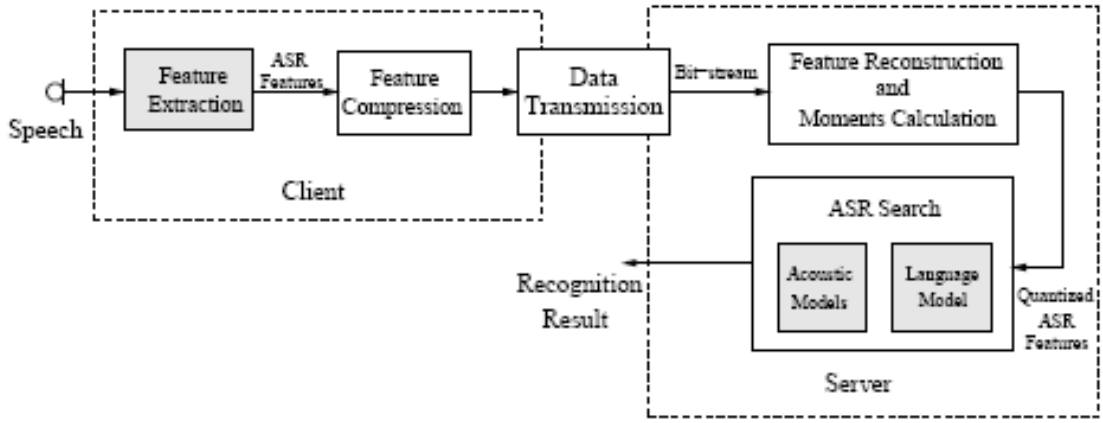


Figure 13: Client-server based ASR system- Distributed Speech Recognition (DSR) [7]

Even though both DSR and NSR make use of the server-based back-end, there are substantial differences in these two schemes favoring DSR [7].

- First of all the speech codecs unlike the feature extraction algorithms are optimized to deliver the best perceptual quality and not for providing the lowest *Word Error Rate* (WER).

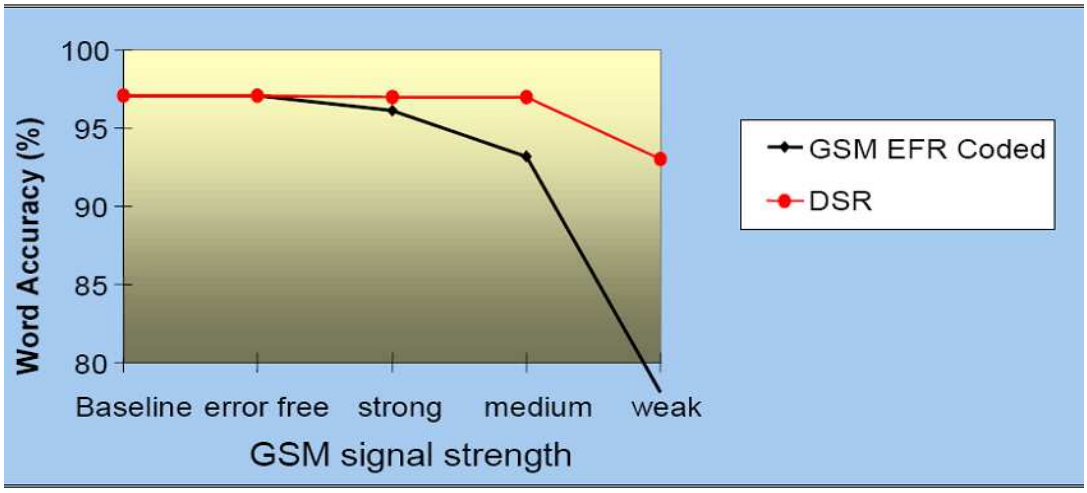


Figure 14: WER degradation in NSR using GSM EFR Coding vs. DSR system [9]

Figure 14 shows performance of a GSM coded speech recognition system (NSR) against that of a DSR over GSM network [9]. The WER is highly degraded as the signal strength becomes weaker in the GSM coded version, while it keeps a reasonable level in DSR system. Note that; unlike NSR, the encoded and transmitted signal are the speech features and not the encoded speech signal itself.

- Second, DSR does not need the high quality speech, but rather some set of characteristic parameters. Thus, it requires lower data rates - 4.8 kbit/s is a common rate for the features transmission.
- Third, since feature extraction is performed place on the client side, the higher sampling rates covering full bandwidth of the speech signal are possible.
- Finally, because in DSR we are not constrained to the error-mitigation algorithm of the speech codec, better error-handling methods in terms of WER can be developed.

The studies within the distributed recognition framework target three aspects indicative for DSR, [7]:

- The development of noise robust and computationally effective feature extraction algorithms.

- The investigation of procedures for feature vectors quantization, permitting compression of the features without losses in recognition quality.
- The elaboration of error mitigation methods.

The composite answer on these entire questions was given by the STQ-Aurora DSR working group established within the *European Telecommunications Standards Institute* (ETSI). The result of the four-year cooperative work of Aurora group members, the world leading ASR companies, has become the ETSI standard ES 201 108 operating at the 4.8 kbit/s data rate, then the ETSI standard ES 202 050 specifying the advanced front-end (AFE) feature extraction, feature compression and back-end error-mitigation algorithms . In 2004 this standard was enriched to the extended advanced front-end (xAFE), allowing for the cost of additional 0.8 kbit/s reconstruction of the intelligible speech signal out of features stream [7]. Overview of this set of standards is given in the next sections.

1.4.5.1 ETSI Aurora Proposal for Distributed Speech Recognition Systems

In *Distributed Speech Recognition* (DSR) architecture the recogniser front-end is located in the terminal and is connected over a data network to a remote back-end recognition server. DSR provides particular benefits for applications for mobile devices such as improved recognition performance compared to using the voice channel with a guaranteed level of recognition performance.

Because it uses the data channel, DSR facilitates the creation of an exciting new set of applications combining voice and data. To enable all these benefits in a wide market containing a variety of players including terminal manufactures, operators, server providers and recognition vendors, a standard for the front-end is needed to ensure compatibility between the terminal and the remote recogniser. The STQ-Aurora DSR working group within ETSI has been actively developing this standard and as a result of this work the first DSR

standard was first published by ETSI in February 2000, featuring the basic front-end specifications. The Mel-Cepstrum was chosen for the first standard because of its widespread use throughout the speech recognition industry. Three next standards have evolved till January 2007.

- The first standard contains the basic functionality of the Mel-Cepstrum front-
  end, ETSI ES 201 108 V1.1.3 (2003-09).
- The second standard works on improving the speech recognition results in a noisy environment; this is referred to as the *Advanced Front End* (AFE), ETSI ES 202 050 V1.1.5 (2007-1).
- The third standard includes a modification to enable speech reconstruction at the back end and enhance the speech recognition for some tonal languages like Mandarin and Thai; this is referred to as the *Extended Front End* (xFE), ES 202 211 V1.1.1 (2003-11).
- The last standard is a merge between the second and third ones, where recognition is noise robust and in the same time speech reconstruction and tonal languages support are enabled; this is referred to as the *Extended Advanced Front End* (xAFE), ES 202 212 V1.1.2 (2005-11).

The Aurora proposal to DSR architecture is given in Figure 15, refer to [10].

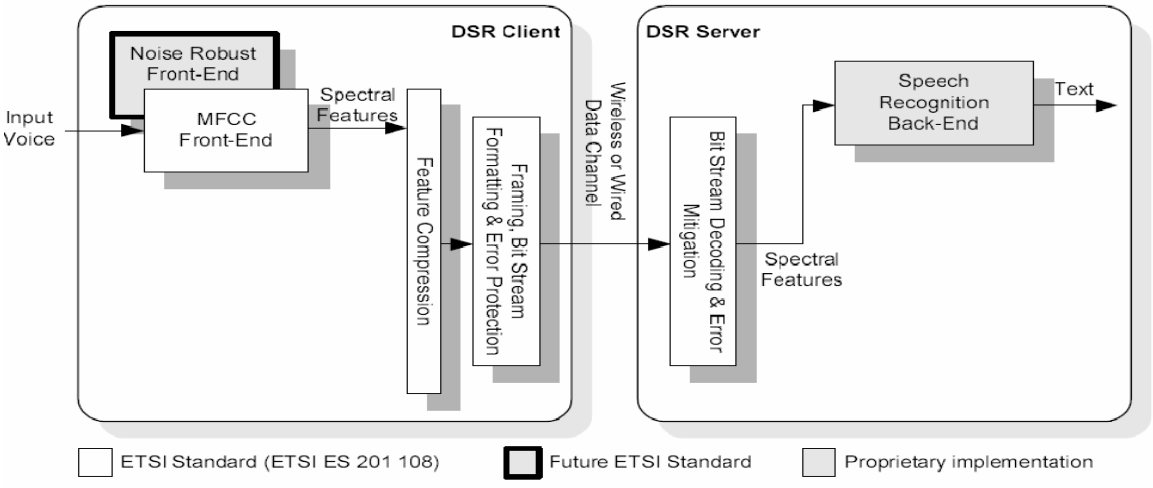


Figure 15: Aurora proposal for DSR system [10]

Reference high level C-code implementations for the algorithms present in the specifications are provided for the four standards, so that proprietary implementations of the standards can compare their performance to that obtained by the reference C-code.

The first standard (ETSI ES 201 108), featuring the basic Front end specification, is the main issue of this thesis, where a hardware implementation of this front-end is to be developed to be deployed in mobile hand-held devices. In the following sections, a brief overview is given on each of the four standards.

1.4.5.1.1 Basic Front End Specifications, ETSI ES 201 108

This standard specifies algorithms for front-end feature extraction and their transmission which form part of a system for distributed speech recognition. Also, it presents a standard for a front-end to ensure compatibility between the terminal and the remote recognizer. The specification covers the following components (more details about this section can be found in [2]):

- The algorithm for front-end feature extraction to create Mel-Cepstrum parameters.
- The algorithm to compress these features to provide a lower data transmission rate.
- The formatting of these features with error protection into a bitstream for transmission
- The decoding of the bitstream to generate the front-end features at a receiver together with the associated algorithms for channel error mitigation.

The standard does not cover the "back-end" speech recognition algorithms that make use of the received DSR front-end features.

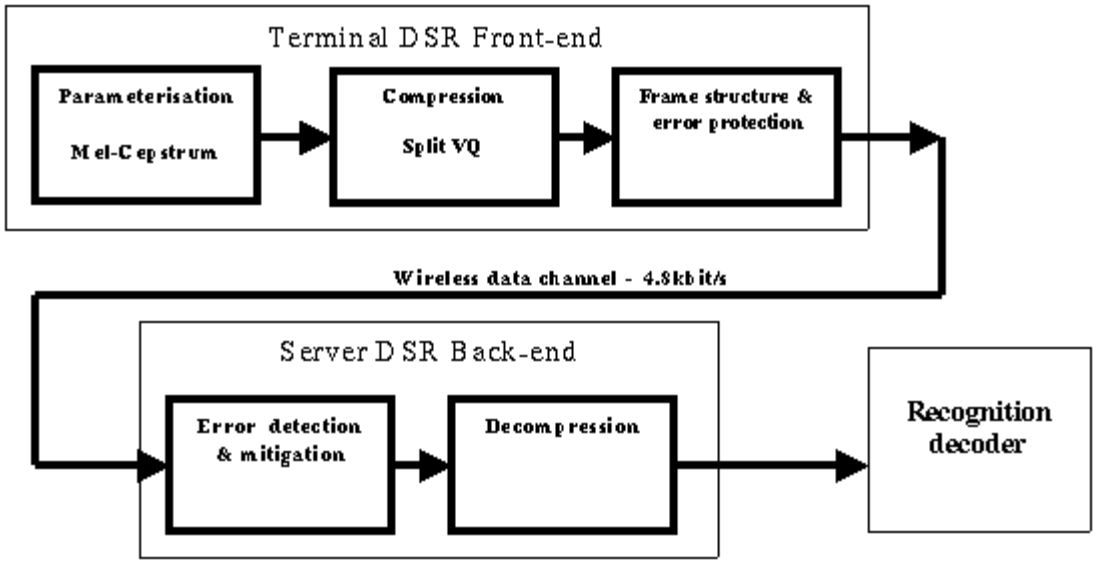


Figure 16: DSR system defined in the Basic front end standard, ETSI ES 201 108 [9]

The specification covers the computation of feature vectors from speech waveforms sampled at different rates (8 kHz, 11 kHz, and 16 kHz). The feature vectors consist of 13 static Cepstral coefficients and a log-energy coefficient.

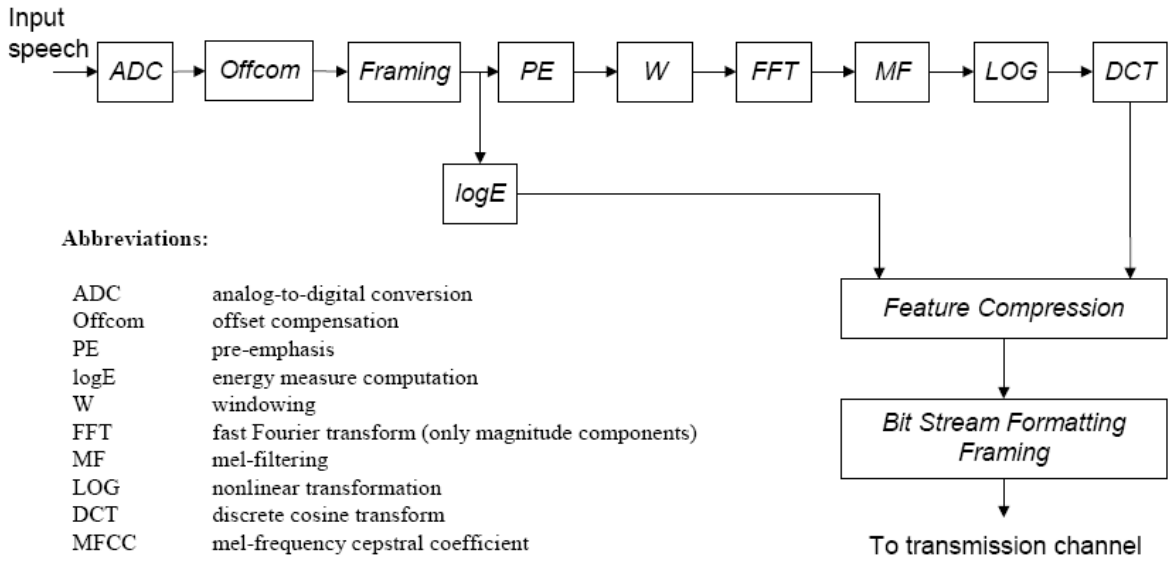


Figure 17: Block diagram of the Front end algorithm specified in the Basic standard, ETSI ES 201 108 [2]

1.4.5.1.2 Advanced Front End Specifications, ETSI ES 202 050

This standard is for an advanced DSR front-end (AFE) that provides substantially improved recognition performance in background noise.

Evaluation of the performance during the selection of this standard showed an average of 53 % reduction in speech recognition error rates in noise compared to ES 201 108. The specification covers the following components (more details about this section can be found in [4]):

- The algorithm for advanced front-end feature extraction to create Mel-Cepstrum parameters.
- The algorithm to compress these features to provide a lower data transmission rate.
- The formatting of these features with error protection into a bit stream for transmission.
- The decoding of the bit stream to generate the advanced front-end features at a receiver together with the associated algorithms for channel error mitigation.

The standard does not cover the "back-end" speech recognition algorithms that make use of the received DSR advanced front-end features.

The advanced DSR standard is designed for use with discontinuous transmission and to support the transmission of voice activity information. The *Voice Activity Detection* (VAD) algorithm is presented in the specification [4], however it is not mandatory for the implementer to use this one, instead he can use any alternative algorithm.

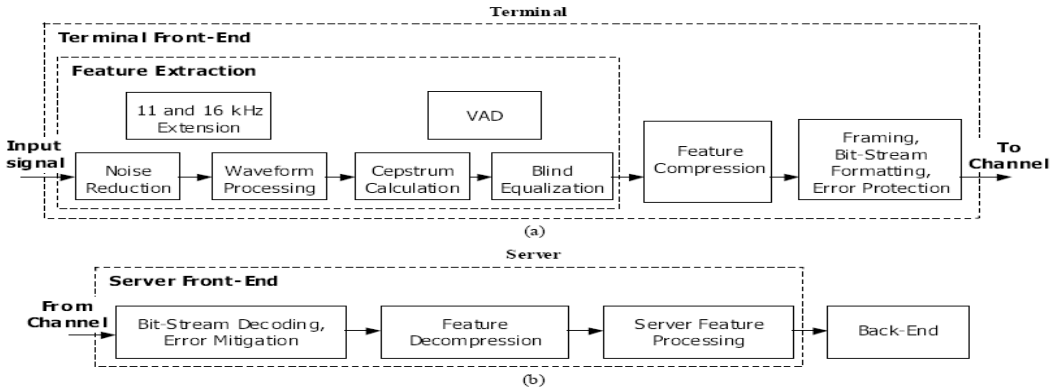


Figure 18: Block scheme of the proposed front-end in specification ETSI ES 202 050. Figure (a) shows blocks implemented at the terminal side and (b) shows blocks implemented at the server side [4]

In the features extraction part, noise reduction is performed first. Then, waveform processing is applied to the de-noised signal and Cepstral features are calculated. At the end, blind equalization is applied to the Cepstral features. The features extraction part also contains an 11 and 16 kHz extension block for handling these two sampling frequencies. Voice activity detection (VAD) for the non-speech frame dropping is also implemented in features extraction.

At the server side (see Figure 18 b), bit-stream decoding, error mitigation and decompression are applied. Before entering the back-end, an additional server feature processing is performed.

1.4.5.1.3 Extended Front End Specifications, ETSI ES 202 211

This standard is for an extended DSR front-end (xFE). It specifies a proposed standard for an extended front-end (XFE) that extends the Mel-Cepstrum front-end with additional parameters, viz., fundamental frequency F0 and voicing class. It also specifies the back-end speech reconstruction algorithm using the transmitted parameters. The specification covers the following components (more details about this section can be found in [3]):

- The algorithm for front-end feature extraction to create Mel-Cepstrum parameters.
- The algorithm for extraction of additional parameters, viz., fundamental frequency F0 and voicing class.
- The algorithm to compress these features to provide a lower data transmission rate.
- The formatting of these features with error protection into a bitstream for transmission.
- The decoding of the bitstream to generate the front-end features at a receiver together with the associated algorithms for channel error mitigation.
- The algorithm for pitch tracking and smoothing at the back-end to minimize pitch errors.

- The algorithm for speech reconstruction at the back-end to synthesize intelligible speech.

For some applications, it may be necessary to reconstruct the speech waveform at the back-end. Examples include, [3]:

- *Interactive Voice Response* (IVR) services based on the DSR of "sensitive" information, such as banking and brokerage transactions. DSR features may be stored for future human verification purposes or to satisfy procedural requirements.
- Human verification of utterances in a speech database collected from a deployed DSR system. This database can then be used to retrain and tune models in order to improve system performance.
- Applications where machine and human recognition are mixed (e.g. human assisted dictation).

In order to enable the reconstruction of speech waveform at the back-end, additional parameters such as fundamental frequency (F0) and voicing class need to be extracted at the front-end, compressed, and transmitted. The availability of tonal parameters (F0 and voicing class) is also useful in enhancing the recognition accuracy of tonal languages, e.g. Mandarin, Cantonese, and Thai, [3].

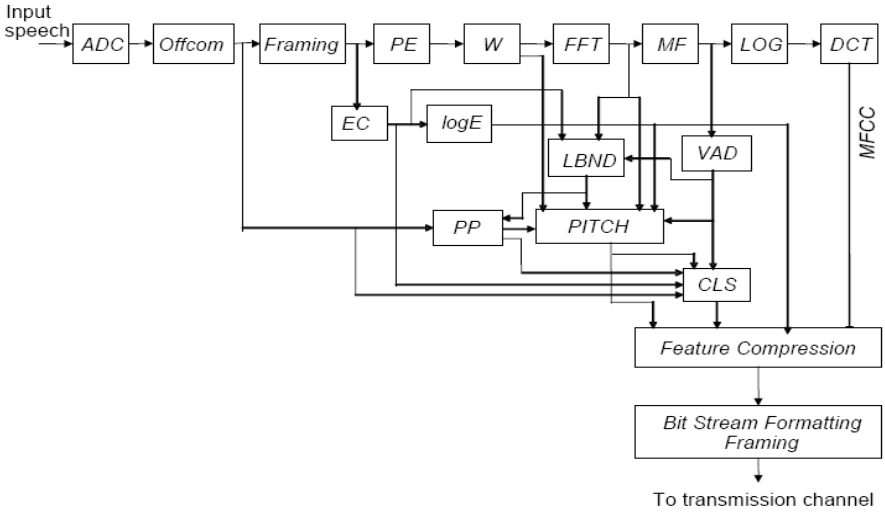


Figure 19: Block diagram of the front-end algorithm specified in specification ETSI ES 202 211 [3]

1.4.5.1.4 Extended Advanced Front End Specifications, ETSI ES 202 212

This standard is for an extended advanced DSR front-end (xAFE). This standard simply comprises the advanced and extended front end features together, where background noise enhancements are included (see 1.4.5.1.2) in addition to tonal language and back-end reconstruction support (see 1.4.5.1.3). The specification covers the following components (more details about this section can be found in [5]):

- The algorithm for advanced front-end feature extraction to create Mel-Cepstrum parameters.
- The algorithm for extraction of additional parameters, viz., fundamental frequency F0 and voicing class.
- The algorithm to compress these features to provide a lower data transmission rate.
- The formatting of these features with error protection into a bit stream for transmission.
- The decoding of the bit stream to generate the advanced front-end features at a receiver together with the associated algorithms for channel error mitigation.
- The algorithm for pitch tracking and smoothing at the back-end to minimize pitch errors.

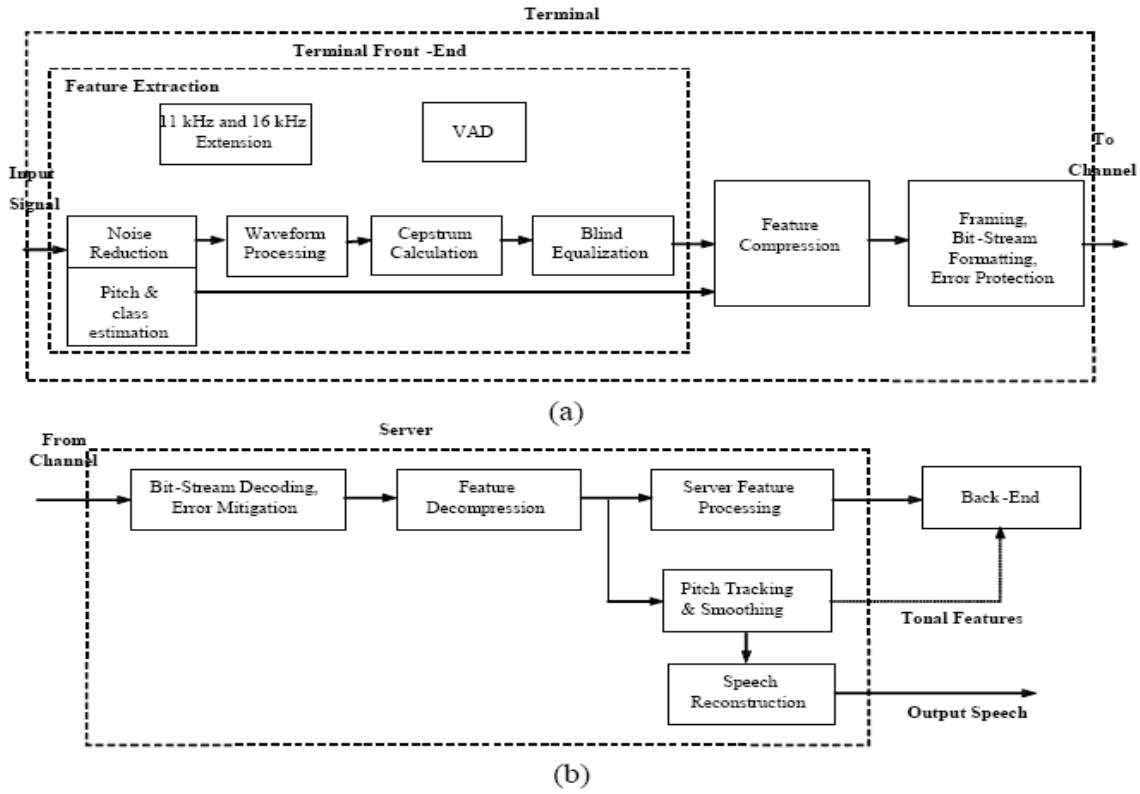


Figure 20: Block scheme of the proposed extended front-end in specification ETSI ES 202 212

(a) shows blocks implemented at the terminal side and

(b) shows blocks implemented at the server side [5]

Chapter 2

2 Design of VLSI Systems

In this chapter, the Very Large Scale Integration (VLSI) digital design styles for *Digital Signal Processing* (DSP) applications are introduced. Digital signal processing systems are required to perform intensive arithmetic operations such as multiplication, division, trigonometric operations, non-linear operations like natural logarithms calculations, error-protection and correction calculations like *Cyclic Redundancy Check* (CRC), *Fast Fourier Transform* (FFT),.. etc. These tasks may be implemented on general purpose processors or custom integrated circuits. Also, DSP applications are required to be performed in real-time, that is, it has a certain deadline to end before. Finally, DSP applications are usually deployed nowadays in hand-held mobile and embedded devices, so power consumption, cost and area usage efficiency factors are essential to DSP applications. The selection of appropriate hardware is determined by many factors, like the application domain, cost, power consumption, or combination of all of these. This chapter introduces different digital hardware implementations for DSP applications.

First, the general design flow and hierarchy of VLSI systems are introduced. Then, different hardware options for DSP applications are presented; which are:

- *Digital Signal Processors* (DSP)
- *Field Programmable Gate Arrays* (FPGA)
- *Application Specific Integrated Circuit* (ASIC)

For each design style of the above, its different types, different architectures and classifications, design flow are presented.

2.1 VLSI Design Flow

The design process, at various levels, is usually evolutionary in nature. It starts with a given set of requirements. Initial design is developed and tested against the requirements. When requirements are not met, the design has to be improved. If such improvement is either not possible or too costly, then the revision of requirements and its impact analysis must be considered. The Y-chart (first introduced by D. Gajski) shown in Figure 21 illustrates a design flow for most logic chips [12], using design activities on three different axes (domains) which resemble the letter Y.

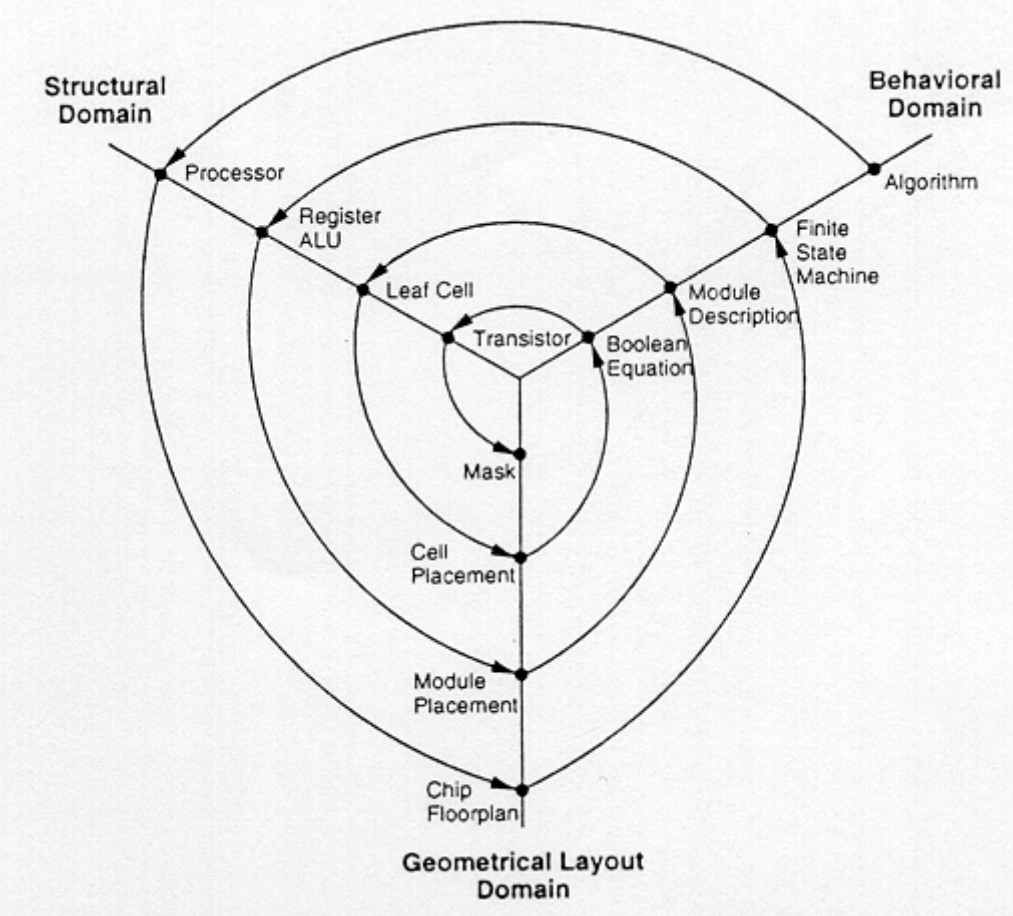


Figure 21: Typical VLSI design flow in three domains (Y-chart representation) [12]

The Y-chart consists of three major domains [2], namely:

- Behavioural domain,

- Structural domain,
- Geometrical layout domain.

The design flow starts from the algorithm that describes the behaviour of the system. The corresponding architecture of the processor is first defined. It is mapped onto the chip surface by floor planning.

The next design evolution in the behavioural domain defines *Finite State Machines* (FSM) which are structurally implemented with functional modules such as registers and *Arithmetic Logic Units* (ALU). These modules are then geometrically placed onto the chip surface using *Computer Aided Design* (CAD) tools for automatic module placement followed by routing, with a goal of minimizing the interconnect area and signal delays.

The third evolution starts with a behavioural module description. Individual modules are then implemented with leaf cells or logic gates. At this stage the chip is described in terms of logic gates (leaf cells), which can be placed and interconnected by using a cell placement & routing program.

The last evolution involves a detailed boolean description of leaf cells followed by a transistor level implementation of leaf cells and mask generation. In standard-cell based design, leaf cells are already pre-designed and stored in a library for logic design use [2].

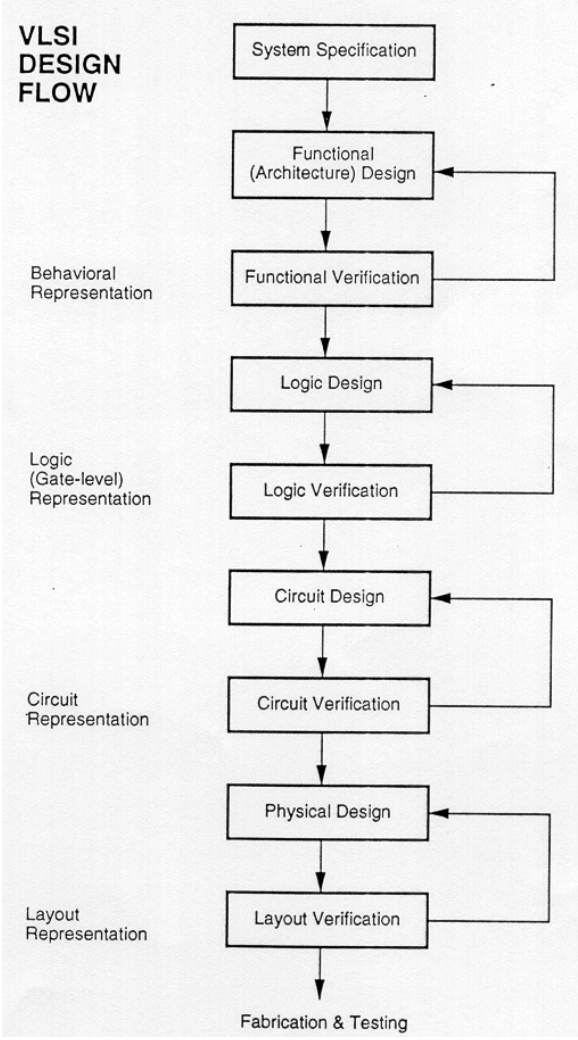


Figure 22: A more simplified view of VLSI design flow [2]

Figure 22 provides a more simplified view of the VLSI design flow, taking into account the various representations, or abstractions of design - behavioural, logic, circuit and mask layout. Note that the verification of design plays a very important role in every step during this process. The failure to properly verify a design in its early phases typically causes significant and expensive re-design at a later stage, which ultimately increases the time-to-market [2].

Although the design process has been described in linear fashion for simplicity, in reality there are many iterations back and forth, especially between any two neighbouring steps, and occasionally even remotely separated pairs. The transition step from level to the lower is called *Synthesis*, and from

lower to higher is called *Verification* or *Simulation*. The transition from the Algorithmic to the FSM level is called *High Level Synthesis*. The transition from the Modular (usually called RTL) level to the gate level is called *Logic Synthesis*. The later is now automated to be done by the modern synthesisers for *Hardware Descriptive Languages* like VHDL or Verilog.

Although top-down design flow provides an excellent design process control, in reality, there is no truly unidirectional top-down design flow. Both top-down and bottom-up approaches have to be combined. For instance, if a chip designer defined architecture without close estimation of the corresponding chip area, then it is very likely that the resulting chip layout exceeds the area limit of the available technology.

2.2 Hardware Design Styles for Digital Signal Processing Applications

Several design styles can be considered for chip implementation of specified signal processing algorithm. The different design styles vary from General Digital Signal Processor, Programmable Device, or Application Specific Integrated Circuit. Each design style has its own merits and shortcomings, and thus a proper choice has to be made by designers in order to provide the functionality at low cost.

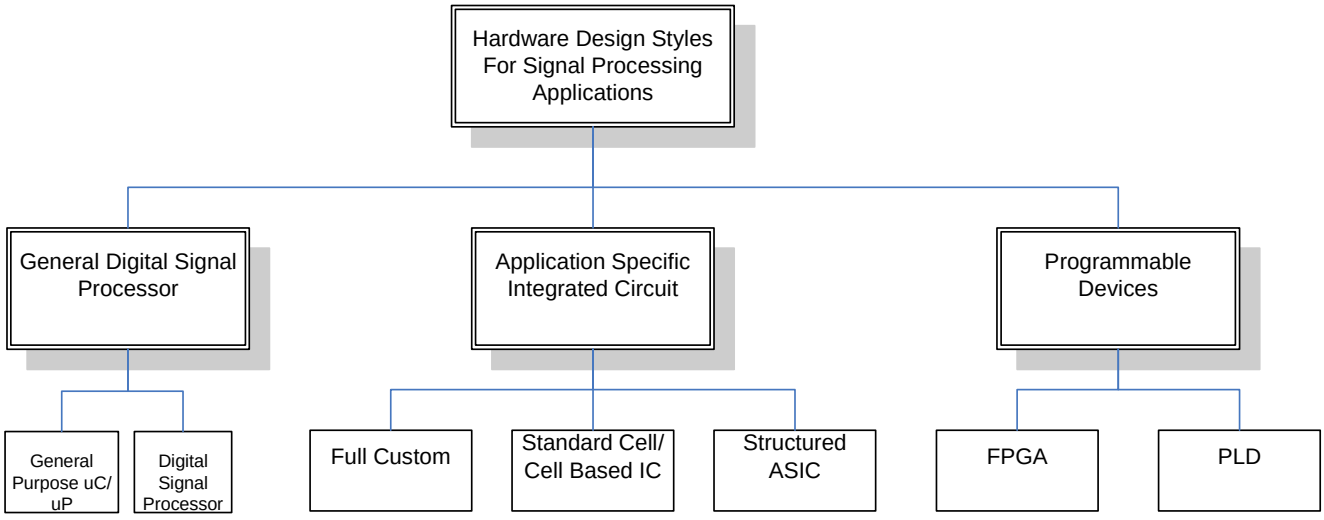


Figure 23: Hardware Design Styles for Signal Processing Applications

The above design styles spans the spectrum from general purpose platform (DSP), passing by Semi- Custom Platform (FPGA), to full custom platform (full custom ASIC). In the following sections each style is presented.

In the following sections, the following design styles are discussed in more details:

- *Digital Signal Processors* (DSP)
- *Field Programmable Gate Arrays* (FPGA)
- *Application Specific Integrated Circuits* (ASIC)

2.2.1 Digital Signal Processors

Digital signal processors differ from genera purpose processors in that they are customized to certain application domain, which is digital signal processing. Its architecture is very different from a general purpose Von Neumann architecture to accommodate the demands of real-time signal processing. When first developed in the beginning of the 80's, the main application was filtering. Since then, the architectures have evolved together with the applications.

DSP processors were originally developed to implement traditional signal processing functions, mainly filters, such as FIR's and IIR's. These applications decided the main properties of the programmable DSP architecture: the inclusion of a multiply- accumulate unit (MAC) as separate data path unit and Harvard or modified Harvard memory architecture instead of Von Neumann architecture as will be discussed later.

2.2.1.1 Classification of Digital Signal Processors Architectures

The fundamental property of a DSP processor is that it uses Harvard or modified Harvard architecture instead of Von Neumann architecture. The main operation in most DSP application is the multiply and accumulate (MAC) operation done on the data and the coefficient of the filter. The different DSP architectures try to reduce the time needed to finish this operation. The DSP architectures according to the MAC unit time are:

- *Harvard Architecture:* uses different buses for program and data memories, which reduces the MAC time.
- *Modified Harvard Architecture (Conventional DSP):* The "Fetch" phase of the MAC instruction is kept in cache, which reduces the access to the instruction memory, and gets the MAC time to only one cycle.
- *Super Harvard Architecture (SHARC):* uses two data buses. SHARC is a trade name of Analog Devices.
- *Enhanced DSP Architecture:* it tries to reach two MAC operations in one cycle. This can be accomplished in many ways; for example, pipelining between the multiply and accumulate operations could achieve *one MAC at double speed*, by performing the multiplication of the current MAC in the same time of performing the addition of the next one.

2.2.1.2 DSP Design Flow

A generalized DSP system design process is illustrated in **Figure 24** (refer to [11]). For a given application, the theoretical aspects of DSP system specifications such as system requirements, signal analysis, resource analysis, and configuration analysis are first performed to define system requirements.

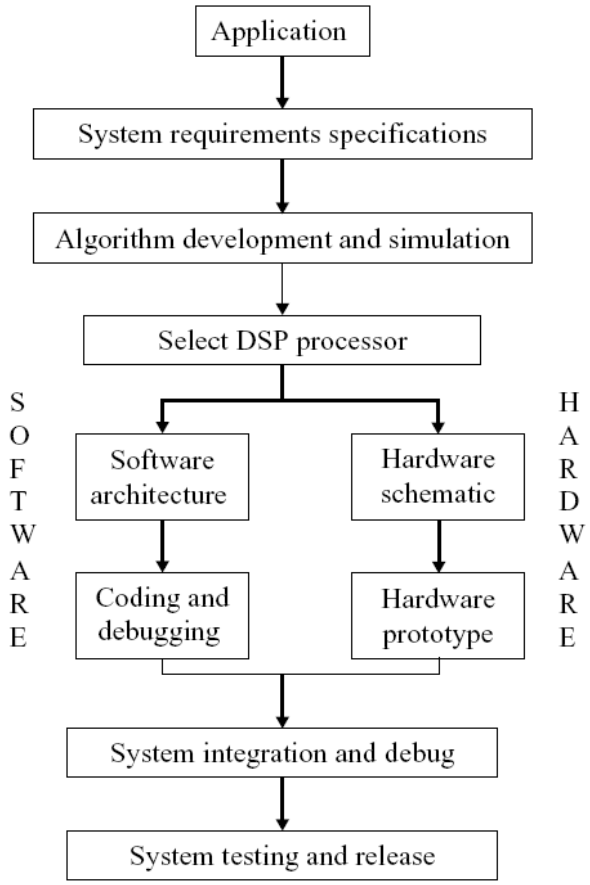


Figure 24: DSP Design Flow [11]

The DSP design flow starts with analysing the system requirements specifications. Then the system is characterized by the embedded algorithm derived from the system requirements analysis step, which specifies the arithmetic operations to be performed. The algorithm for a given application is initially described using difference equations or signal-flow block diagrams with symbolic names for the inputs and outputs. In documenting an algorithm, it is sometimes helpful to further clarify which inputs and outputs are involved by means of a data-flow diagram which specifies the required steps in order to derive the outputs. There are two methods of characterizing the sequence of operations in a program: flowcharts or structured descriptions. High-level languages DSP tools (such as MATLAB, Simulink, or C/C++) are used at the algorithm level, since they are capable of algorithmic-level system simulations. We then implement the algorithm using software, hardware, or both, depending

on specific needs. After the system components are ready they are integrated and tested, then the whole system is validated to be released.

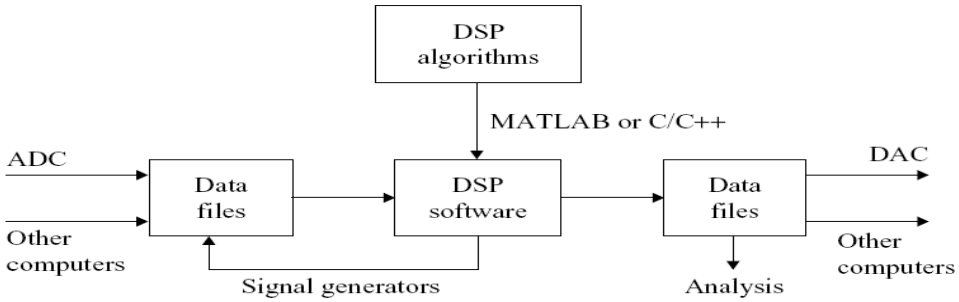


Figure 25: DSP simulation environment [11]

A DSP algorithm can be simulated using a general-purpose computer so that its performance can be tested and analyzed. A block diagram of general-purpose computer implementation is illustrated in Figure 25 (refer to [11]). The test signals may be internally generated by signal generators or digitized from a real environment based on the given application or received from other computers via the networks. The simulation program uses the signal samples stored in data file(s) as input(s) to produce output signals that will be saved in data file(s) for further analysis [11].

Advantages of developing DSP algorithms using a general-purpose computer are [11]:

1. Using high-level languages saves algorithm development time and facilitates testing and debugging. In addition, the prototype C programs used for algorithm evaluation can be ported to different DSP hardware platforms.
2. Input/output operations based on disk files are simple to implement and the behaviours of the system are easy to analyze.
3. Floating-point data format and arithmetic can be used for computer simulations, thus easing development.
4. Further, fixed point simulation tool boxes exist in MATLAB or Simulink.

2.2.2 Field Programmable Gate Arrays

A *Programmable Logic Device* (PLD) is digital circuit that performs reconfigurable or programmable logic function. Unlike a logic gate, which has a fixed function, a PLD has an undefined function at the time of manufacture. Before the PLD can be used in a circuit it must be programmed or configured. PLDs exist in many forms:

- *ROM as PLD:* where logic functions are stored in ROM.
- *Programmable Array Logic (PAL):* where logic functions are obtained by "sum-of-product" fashion, with fixed-OR plane and programmable-AND plane.
- *Generic Array Logic (GAL):* same as PAL, but can be reprogrammed and erased.
- *Complex Programmable Logic Device (CPLD):* same as PAL and GAL but with larger size (few hundreds logic gates).
- *Field Programmable Gate Arrays (FPGA):* they are two dimensional arrays of logic blocks and flip-flops with electrically programmable interconnections between logic blocks. The interconnections consist of electrically programmable switches; which is why FPGA differs from Custom ICs which have hard-wired interconnections which cannot be re-programmed.

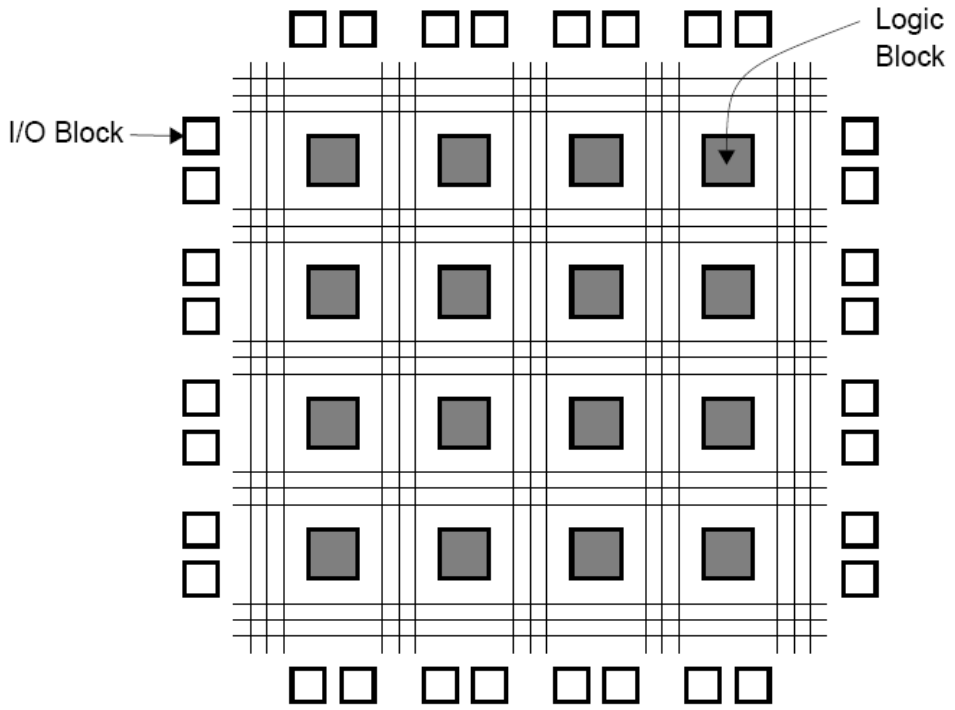


Figure 26: FPGA general internal structure

Routing in FPGAs consists of wire segments of varying lengths which can be interconnected via electrically programmable switches. Density of logic blocks used in an FPGA depends on length and number of wire segments used for routing. Number of segments used for interconnection typically is a trade off between density of logic blocks used and amount of area used up for routing.

FPGAs were introduced as an alternative to custom ICs for implementing entire system on one chip and to provide flexibility of re-programmability to the user. It reduces the time to market and significantly reduces the cost of production. Another advantage of FPGAs over Custom ICs is that with the help of *Computer Aided Design* (CAD) tools circuits could be implemented in a short amount of time (no physical layout process, no mask making, no IC manufacturing).

2.2.2.1.1 Classification of FPGA

Field programmable gate arrays can be classified according to three different criteria detailed in the next sections.

2.2.2.1.1.1 Main Logic Block Type Classification

This is the main building block that performs the logic functions. Logic blocks of an FPGA can be implemented by any of the following:

- Transistor pairs
- Combinational gates like basic NAND gates or XOR gates
- N-input Lookup tables
- Multiplexers
- Wide fan-in AND-OR structure

Size of the block decides the density and utilization of the FPGA resources (smaller size means higher density and better utilization).

2.2.2.1.1.2 FPGA Architecture Classification

Basic structure of an FPGA includes logic elements, programmable interconnects and memory. Arrangement of these blocks is specific to particular manufacturer. On the basis of internal arrangement of blocks FPGAs can be divided into three classes:

2.2.2.1.1.2.1 Symmetrical Array

This architecture consists of logic elements (LE) arranged in rows and columns of a matrix and interconnect laid out between them. This symmetrical matrix is surrounded by I/O blocks which connect it to outside world. Interconnects provide routing path. Direct interconnects between adjacent logic elements have smaller delay compared to general purpose interconnect.

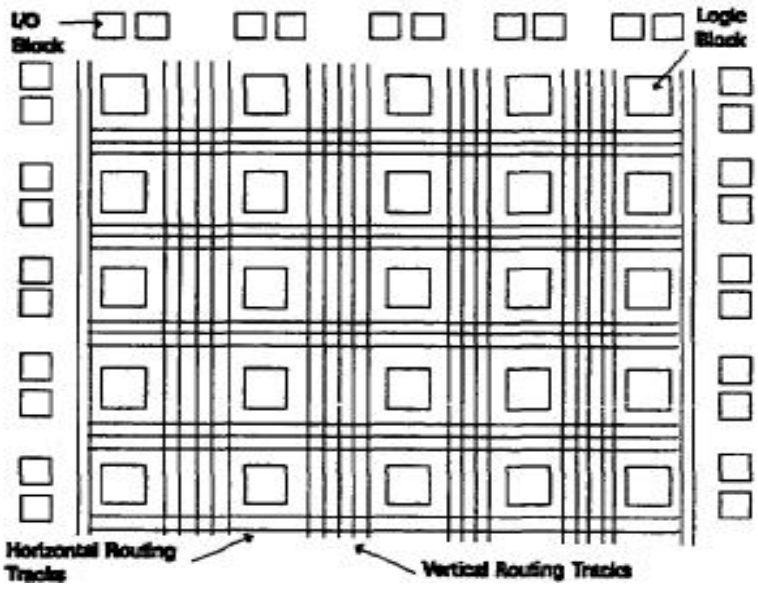


Figure 27: Symmetrical Array

2.2.2.1.1.2.2 Row Based Architecture

Row based architecture consists of alternating rows of logic modules and programmable interconnect tracks. Input and output blocks are located in the periphery of the rows. One row may be connected to adjacent rows via vertical interconnect.

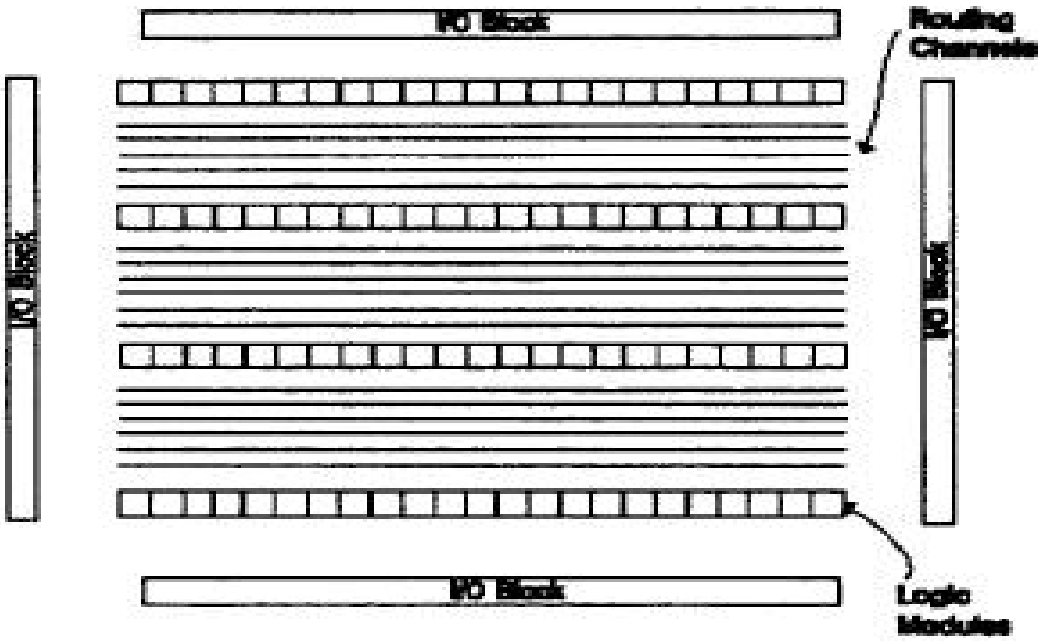


Figure 28: Row Based Architecture

2.2.2.1.1.2.3 Hierarchal PLD

This architecture is designed in hierarchical manner with top level containing only logic blocks and interconnects. Each logic block contains number of logic modules. Each logic module has combinatorial as well as sequential functional elements. Communication between logic blocks is achieved by programmable interconnects arrays. Input output blocks surround this scheme of logic blocks and interconnects.

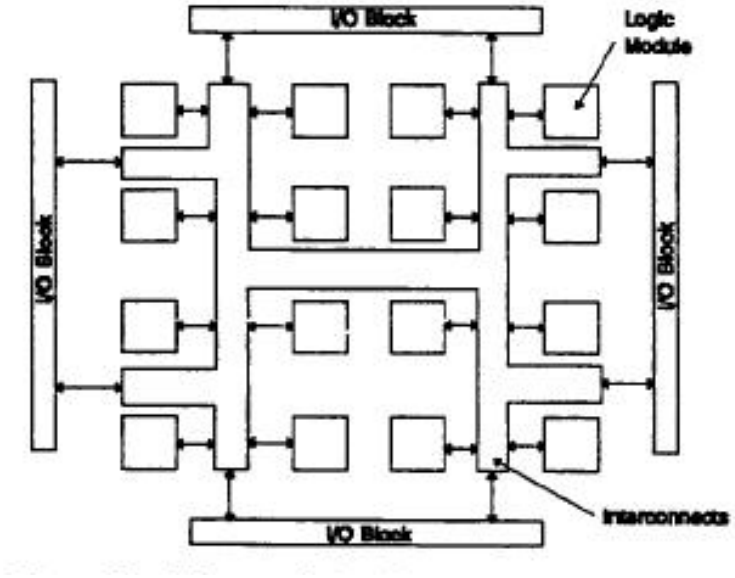


Figure 29: Hierarichal PLD

2.2.2.1.1.3 Programming Technology Classification

The first type of user-programmable switch developed was the *fuse* (still used in some smaller devices). For higher density devices, where CMOS dominates the IC industry, different approaches to implementing programmable switches have been developed. Three major programming technologies are used nowadays:

- Floating Gate Programming Technology
- SRAM Programming Technology
- Anti-Fuse Programming Technology

2.2.2.1.2 Design Flow of FPGA Systems

Figure 30 shows the general FPGA design flow.

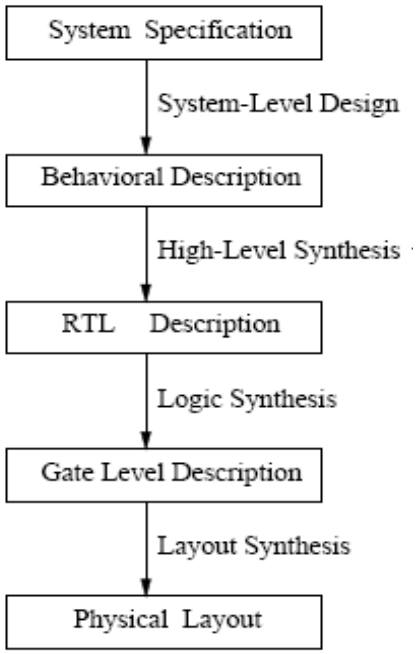


Figure 30: FPGA Design Flow [13]

Generic design flow of an FPGA includes following steps:

- **System Specification:** in this step, the designer analyses the system requirements and make the hardware-software distribution; that is, to decide which parts of the system shall be done in hardware (FPGA) and which shall be done in software. Having hardware part specified, the system requirements for hardware should be clear to design the chip according to those requirements.
- **RTL/ HDL Description:** at this step, the hardware is described, either using schematic representation or Hardware Description Languages (HDLs) like Verilog or VHDL.
- **Logic Synthesis:** this is to transfer to the gate-level from the *Register Transfer Language* (RTL) level. In other words, to implement the design on a given FPGA. CAD tools automate this step.

- **Place and Route:** Implementation includes partition, place and route. The output of design implementation phase is bit-stream file. This step is automated with CAD tools.
- **Circuit Verification:** Bit stream file is fed to a simulator which simulates the design functionality and reports errors in desired behavior of the design. Timing tools are used to determine maximum clock frequency of the design. Now the design is loading onto the target FPGA device and testing is done in real environment.

As appears from the above steps, following *Logic Synthesis* step, all steps are automated, which is a major advantage of FPGA design flow, where CAD tools are available to do most of the work, leaving the design burden on the designer. This is not the case of Custom IC development.

2.2.3 Application Specific Integrated Circuits

An Application-Specific Integrated Circuit (ASIC) is an Integrated Circuit (IC) customized for a certain target application, rather than intended for general-purpose use. This definition includes FPGA too. To differentiate ASIC from FPGA, most designers use ASIC only for non field programmable devices. ASIC usually provides less cost, less power consumption than FPGA and DSP, in addition to their high Intellectual Property (IP) design security, where it is much harder to reverse-engineer ASIC design. This comes on the cost of hard and long development cycle.

2.2.3.1 Classification of ASIC

2.2.3.1.1 Standard Cell/ Cell Based IC

Every ASIC manufacturer creates ready made functional blocks with known electrical characteristics, which the designer can use directly. The RTL code is mapped to these pre-defined standard cells defined by the manufacturer at the Logic Synthesis step.

Standard cell design is the utilization of these functional blocks. Standard cell design fits between gate array and full custom design in terms of both its NRE (Non-Recurring Engineering) and recurring component cost.

2.2.3.1.2 Gate Array ASIC

In gate-array-based ASIC, transistors are predefined on the silicon wafer, where:

- Base cell is the smallest element that is replicated.
- Base array is the predefined pattern of transistors.

It is called *Masked Gate Array* (MGA) when only layers which define the interconnect between transistors are defined by the designer using custom masks. Designer chooses from a gate-array library pre designed and pre characterized logic cells (often called macros). There are three types of this style:

2.2.3.1.2.1 Channeled Gate Array

In this type, we leave space between the rows of transistors for wiring. Its characteristics are as follows

- Only interconnect is customized
- The interconnect uses predefined spaces between rows
- Manufacturing lead time is between 2 days and 2 weeks

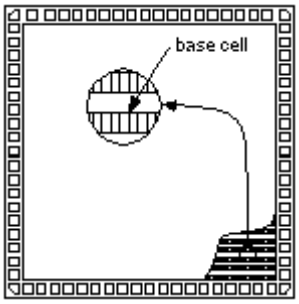


Figure 31: Channeled Gate Array

2.2.3.1.2.2 Channel less Gate Array

In this style, there are no predefined areas set aside for routing between cells. We customize the contact layer that defines the connections between metal1 and transistors. The characteristics of this style are:

- Only some (the top few) mask layers
  are customized – the interconnect
- Manufacturing lead time is
  between 2 days and 2 weeks

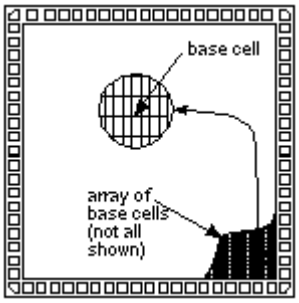


Figure 32: Channel-less Gate Array

2.2.3.1.2.3 Structured/Platform ASIC

Structured (also referred to as Platform) ASIC design is a relatively new term in the industry. The motivation to this style is that other gate arrays have only fixed gate-array base cell; which is difficult and inefficient implementation, so we set aside some IC area and dedicate it to a specific function (which can contain different cells, more suitable for building memory cells, for example, or complete block, such as a microcontroller). This technology is seen as bridging the gap between field-programmable gate arrays and standard-cell ASIC design.

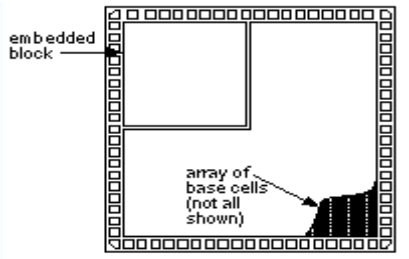


Figure 33: Structured ASIC

Structured ASIC design has the following advantages:

- Small non-recurring expenditures (NRE) due to less custom-produced metal layers.
- Other gate arrays focus on lowering the turnaround time and mask set cost by making predefined metal layers, in addition to that, structured ASIC reduces the design time by having blocks of predefined characteristics. For example, in a cell-based or gate-array design the user often must design power, clock, and test structures themselves; these are predefined in most structured/platform ASICs and therefore can save time and expense for the designer compared to other gate-array techniques.
- Structured ASIC encourages Intellectual Property (IP) cores re-use by embedding them in the reserved wafer area. For example, a complete ARM processor, USB driver,..etc can be embedded and reused with no extra effort.

The Altera technique of producing a structured cell ASIC where the cells are the same design as the FPGA, but the programmable routing is replaced with fixed wire interconnect is called HardCopy. The Xilinx technique of producing a customer specific FPGA, that is 30% - 70% less expensive than a standard FPGA and where the cells are the same as the FPGA but the programmable capability is removed, is called EasyPath [6]

Modern VLSI design flow consists of FPGA prototype and then automatic migration (through EDA and CAD tools) to structured ASIC device that corresponds to the FPGA device used in the prototype. Migration effort is often small or negligible, since FPGA manufacturers provide free migration services and physical verification of the final ASIC versus the prototype in case of large production volume requested.

2.2.3.1.3 Full Custom IC

Full-custom ASIC design defines all the photo lithographic layers of the device. The benefits of full-custom design usually include reduced area (and therefore recurring component cost), performance improvements and also the ability to integrate (include) analog components and other pre-designed (and thus fully verified) components such as microprocessor cores that form a System-On-Chip (SoC) [6].

The disadvantages of full-custom can include increased manufacturing and design time, increased non-recurring engineering costs, more complexity in the Computer Aided Design (CAD) system and a much higher skill requirement on the part of the design team. However for digital only designs, "standard-cell" cell libraries together with modern CAD systems can offer considerable performance/cost benefits with low risk[6].

2.2.3.2 Cell libraries, IP-based design, hard and soft macros

Cell libraries of logical primitives are usually provided by the device manufacturer as part of the service. Although they will incur no additional cost, their release will be covered by the terms of a Non Disclosure Agreement (NDA) and they will be regarded as intellectual property by the manufacturer. Usually their physical design will be pre-defined as so they could be termed hard macros [6].

But what most engineers understand as "intellectual property" are IP cores, designs purchased from a third party as sub-components of a larger ASIC. They may be provided as an HDL description (often termed a Soft Macro), or as a fully routed design that could be printed directly onto an ASIC's mask (often termed a Hard Macro). Many organizations now sell such pre-designed IP, and larger organizations may have an entire department or division to produce such IP for the rest of the organization. For example, one can purchase CPUs, Ethernet, USB or telephone interfaces. Soft Macros are

often process independent; i.e., they can be fabricated on a wide range of manufacturing processes and indeed different manufacturers. Hard Macros are process limited and usually further design effort must be invested to migrate (port) to a different process or manufacturer [6].

2.2.3.3 Design Flow of ASIC

Broadly used ASIC design flow can be divided into following:

- **System Requirements Analysis and Specification:** This is the same as in FPGA design flow.
- **RTL Description:** This is the same as in FPGA design flow.
- **Functional Simulation/Verification:** Here the RTL description is tested for functional correctness.
- **Logic Synthesis:** This is the same as in FPGA design flow.
- **Design Verification:** Formal verification methods are used to test the functional correctness of gate-level netlist. Testing functional correctness involves testing an optimized design against a golden design description.
- **Layout:** This phase involves floor planning. Placement of cells on the chip area. Placement of Input/Output pads on the chip area. Clock tree synthesis is performed in order to minimize space and power consumed by clock signal. Placement and routing is carried out on this design.

Chapter 3

3 Comparative Study of VLSI Design Styles for Front End Speech Processor

In this chapter, a comparative study is made between three suggested hardware design styles for the speech front end processor, which are:

- Digital Signal Processors
- Field- Programmable Gate Arrays
- Application Specific Integrated Circuits.

The target of the comparison is to reach the best hardware platform for the front end speech processor.

The points of comparison are as follows:

- The *Non- Recurring Engineering* (NRE) Cost is the cost paid once for the first design to be accomplished. This is different from the production cost, which is paid every time a unit is produced. A comparison is made to target this important point.
- The re-programmability is ability to modify or add new features to the design after it is being downloaded to hardware platform. The cost of this modification varies form one style to another. This is an important point to be addressed while choosing a certain hardware platform.
- The production cost in each style.
- The available hardware resources in each style and its suitability to the required resources for the front end speech processor.
- The available hardware internal memory in each style and its suitability to the required memory needs for the front end speech processor.
- The speed limits in each style and its suitability for the required processing time for front end speech processor.

- The power consumption in each style is also an important point of comparison, where it gives a good indication on the suitability of this style to be used with hand–held or battery powered devices.
- Also, it is important to consider the required production volume while choosing a platform; hence, a comparison is made from the required production volume perspective between the suggested platforms.

The comparison is held on two dimensions. The first is between different designs styles mentioned above. The second dimension is between different models and manufacturers of each style alone, this is done only when there is difference between various models concerning the corresponding point of comparison. The comparison is supported with real figures from the available platforms in the market today. At the end of the comparison, a brief comparison table is made, and a conclusion is drawn about the best design style for the front end processor.

3.1 Design Time and Non- Recurring Expenditures Cost (NRE) Comparison

The *Non- Recurring Expenditures* (NRE) Cost is the cost paid once for the first design to be accomplished. This is different from the production cost, which is paid every time a unit is produced. To address this point of comparison, the design flow of every style should be revisited, and compared to the design flows of the other styles; this is done to be able to identify extra or time consuming steps of each design style.

Digital signal processors offer the least design time and shortest design flow. In general, moving to custom designs (like full-custom ASICs) will increase the design performance, but on the other hand, it will increase the design time and effort. This is shown in Figure 34, which makes FPGA better in terms of shorter design Time than ASIC design. The gap between FPGA and ASIC design times can be reduced by using FPGA for prototyping and then migrating to structured ASIC in production, where migration requires less effort and time than full-custom ASIC design.

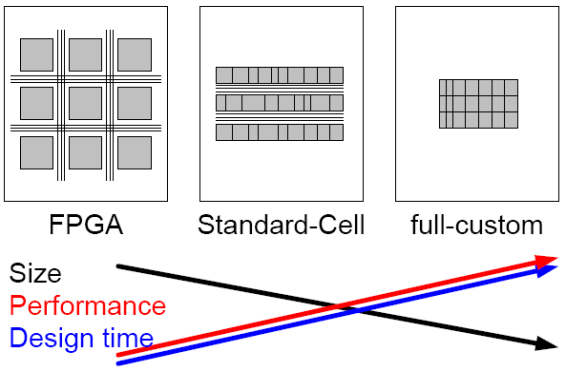


Figure 34: Customization increases the design time

3.2 *Re-Programmability Comparison*

This feature is highly required for designs that are not stable, or subject to modifications all the time. Also, upgrading a design, adding new features, or embedding a design in a larger requires having high re programmability at the lowest possible cost.

General purpose DSPs are the most flexible design style to modifications and updates, since its development nature is software. FPGAs are more flexible than ASICs. So, it is recommended that if the design is subject to modifications, addition of features, or extension to higher versions, then use DSPs or FPGAs rather than ASICs. In case of migration to structured ASIC, the design must be highly stable in the FPGA prototype stage before put in production.

3.3 *Resources Comparison*

In most DSP applications, the main operation is the *Multiply- And – Accumulate* (MAC), hence, the main resource in the DSP processor is the MAC unit. Other resources are also important, like barrel shifter, ALU…etc.

In case of FPGA or ASIC, we mean by resources the total number of gates or logic elements, the available dedicated multipliers, the available registers…etc.

The following sections will show some samples of the available platforms in the market today, and their corresponding features.

3.3.1 Digital Signal Processor

Table 1 shows sample DSPs capabilities in the market, together with their manufacturers:

Manufacturer	Family	Peak MMACs	Total RAM (Program + Data RAM)	ROM	Frequency
Texas Instruments	TMS320C54x	50	16 KB	8 KB	50 MHz
Texas Instruments	TMS320C55x	320	32 KB	32 KB	300 MHz
Analog Devices	SHARK	300	125 KB	375 KB	150 MHz
Analog Devices	ADSP-218x	NA	256 KB	NA	80 MHz
Free Scale	DSP56300	80 - 100	24 KB	NA	80-100 MHz

Table 1: Sample Digital Signal Processors and their features

Study of porting the front end processor to DSP platform was done, where only *one* available MAC unit was assumed with no pipelining between stages. Conventional DSP was assumed (one MAC per clock cycle). From the Table 1, it is very clear that any DSP processor contains more than one MAC unit, which exceeds the needs of the front end processor. This enables addition of new features or modules from a larger speech recognition system on the same DSP processor.

3.3.2 Field Programmable Gate Arrays

Table 2 shows sample FPGAs and their corresponding capabilities:

Manufacturer	Model	Number of Gates/ Cells	Total RAM (distributed and Block)	Dedicated Multipliers	Frequency
Xilinx	Virtex-5	330,000 Cells	150 KB – 1 MB	32 to 640 * (25 X 18 Multipliers)	550 MHz
Xilinx	Virtex-E	58 K – 4 M Gates 1728-73,000 Logic Cell	10 KB – 800 KB	NA	130- 240 MHz
Xilinx	SPARTAN-3A	50K-1400K	7 KB-72 KB	3 to 32 * (18 X 18 Multipliers)	5 – 250 MHz
Altera	Stratix III	47,000-338,000 Logic Cells	330 KB-2.8 MB	216 to 576 * (18 X 18 Multipliers)	600 MHz
Altera	Cyclone III	5,000-119,000 Logic Cells	52 KB-486 KB	23 to 288 * (18 X 18 Multipliers)	260 MHz

Table 2: Sample FPGAs and their features

Rough estimation of the required resources for the front end algorithm showed that the total required resources can easily fit in a 10-20K gates FPGA chip. From Table 1 and Table 2, it is clear that these requirements are met.

Some of the above FPGAs have much higher capabilities than required, which can used to implement extra features or modules.

3.3.3 Application Specific Integrated Circuits

For full-custom and cell-based designs, the resources are customized by the designer to the application needs. For structured ASICs, usually the resources of the FPGA used in the prototype design will limit the final design resources.

3.3.4 Conclusion

From the above results, it is clear that the required resources for the front end processor can be met in the three design styles easily. However, some styles capabilities might exceed the required resources, in which case adding extra modules and extension to larger parts of speech recognition system done.

3.4 Processing Time Requirements Comparison

The timing requirements of the front end specified in the Aurora standard are relaxed, where the effective frame rate is 9.16 ms as will be discussed later in the System Design and Implementation chapter. Hence, this can be easily achieved in any of the three design styles.

3.5 Memory Requirements Comparison

Table 1 and Table 2 show the available internal RAM and ROM for DSP and FPGA styles. For ASIC style, it will be assumed to be the same as FPGA. From Table 2, the total internal RAM in the shown FPGAs range from 16 to 256 KB (including program and data memories). And for ROM, the range is from 8 to 375. From Table 1, the total internal RAM in the DSP processors shown range from 7KB to 1 MB.

Initial estimation of the required RAM and ROM needs for the front end processor showed that about 4 to 6 KB of RAM and 4 KB of ROM are needed. It is clear that these requirements are met easily in any of the three design

styles, and extra memory is available for extension or addition of extra speech recognition modules from a larger speech recognition system.

3.6 Power Consumption Comparison

The power consumption issue is very vital, especially when addressing hand-held or battery powered devices. Types of power consumption are:

- Static power is the power consumed by a device when it is in its quiescent condition with no input signals being exercised. It is also referred to as steady-state or standby power. In today's 90 nm technology devices, leakage currents in the transistors are the biggest contributors to static power. This is usually the key parameter of concern to designers of portable equipment because of its effect on battery life, especially for devices that spend large amounts of time in a standby condition waiting for input from the outside world.
- Dynamic power is the power consumed during normal operation. It is also referred to as operating power. Dynamic power is dependant on operating signal frequency; interconnect capacitance, and operating voltage. Because the voltage dependency is a square function, the reduction in voltage when moving to 90 nm devices has substantially reduced operating power in many devices. However, for large, high-performance systems with high operating frequencies, dynamic power is still a significant component of total system power.
- In-rush power is the power required at device power-up. It is also referred to as power-up or start-up power, or power-on surge power (or current). Some devices require many times more power to begin operation than they do during normal operation, thereby placing demands on system power supplies. In a consumer system with very tightly controlled power supply size and cost, ensuring that in-rush power is not more than normal operating power is a key design goal.

It is of no doubt that moving towards customization improves performance and reduces power consumption, hence, full-custom ASICs will be at the top

most side of the spectrum, while general purpose DSP will lie at the other side of the spectrum.

3.6.1 Digital Signal Processor

General purpose DSPs are in general consuming higher power than FPGAs and ASICs. Hence, most of the applications that utilize DSP processors use chargeable batteries, and suffer from less battery life. Power dissipation of an FPGA design is typically about 20% of a microprocessor based design working at the same sample rate.

3.6.2 Field Programmable Gate Arrays

The main sources of power consumption in FPGA are:

- Inrush - power-up consumption, which is very high for SRAM based FPGAs. No Inrush power-up consumption for Anti-fuse.
- Standby – no switching activity but power is on. It is large for SRAM FPGAs due to large number of SRAM cells.
- Dynamic – consumption during normal operation. It is proportional to the frequency of charging and discharging of internal parasitic capacitances.

3.6.3 Application Specific Integrated Circuits

One of the most important motivations towards customization is to reduce power consumption. In full-custom ASIC design, the designer designs his own cells, with the required power characteristics. In cell-based, the standard cells with power characteristics that match the system power consumption requirements are chosen carefully to achieve the minimum level of power consumption.

Structured ASIC designs are estimated to lower the power consumption by around 50 % compared to the FPGA that was used in prototyping.

3.6.4 Conclusion

It is of no doubt that moving towards customization improves performance and reduces power consumption, hence, full-custom ASICs will be at the top most side of the spectrum, while general purpose DSP will lie at the other side of the spectrum.

3.7 Production Volume and Unit Cost Comparison

We mean here by production cost: the cost of one unit after design is stable and finished. This cost varies from one design style to another, and in some design styles, the cost of the design prototype is the same as the produced unit cost, like FPGA and DSP styles. In the next sections, a sample of the unit cost of the available units in each design style in the market today is presented. Two important notes are to be considered:

- The prices given are restricted only to the date of writing this document.
- Only the units that meet the required needs for the speech front end processor are given here.

Also, for ASIC style, the production volume is a main point while talking about cost, so, a detailed comparison between the minimum business size (i.e. production volume) and the corresponding cost in different ASIC manufacturers is made.

3.7.1 Digital Signal Processor

Table 3 shows a sample of the prices of the available DSP processors in the market today, together with their manufacturers:

Manufacturer	Model	Price
Texas Instruments	TMS320C54x	$4.00
Texas Instruments	TMS320C55x	$5.25
Analog Devices	ADSP-218x	$ 33.00
Analog Devices	SHARK	$7.22
Free Scale	DSP56300	$45.00

Table 3: DSP Processors Prices

3.7.2 Field Programmable Gate Arrays

Table 4 shows a sample of the prices of the available FPGAs in the market today, together with their manufacturers:

FPGA	Model	Manufacturer	Supplier	Price
Virtex-5	XC5VLX30	Xilinx	Avnet Electronics www.em.avnet.com	$250.000
Virtex-5	XC5VSX95T	Xilinx	Avnet Electronics www.em.avnet.com	$2,735.000
Virtex-E	XCV50E-6	Xilinx	Avnet Electronics www.em.avnet.com	$26.000
Virtex-E	XCV200E	Xilinx	Avnet Electronics www.em.avnet.com	$83.000
SPARTAN 3AN	XC3S50AN	Xilinx	Avnet Electronics www.em.avnet.com	$14.000
SPARTAN 3AN	XC3S1400AN	Xilinx	Avnet Electronics www.em.avnet.com	$91.000
SPARTAN 3AN	XC3S400AN	Xilinx	Avnet Electronics www.em.avnet.com	$45.000
SPARTAN 3A	XC3S400A	Xilinx	Avnet Electronics www.em.avnet.com	$31.000
SPARTAN 3A	XC3S200A	Xilinx	Avnet Electronics www.em.avnet.com	$22.000

FPGA	Model	Manufacturer	Supplier	Price
SPARTAN 3A	XC3S50A	Xilinx	Avnet Electronics www.em.avnet.com	$12.000
SPARTAN-3A DSP	XC3SD1800A	Xilinx	Avnet Electronics www.em.avnet.com	$147.00
SPARTAN-3A DSP	XC3SD3400A-4CS484LI	Xilinx	Avnet Electronics www.em.avnet.com	$202.00
Stratix III	EP3SL150	Altera	Altera www.altera.com	$2,184.000-$3,352.000
Stratix II	EP2S3	Altera	Altera www.altera.com	$258.000-$339.000
Cyclone III	EP3C5	Altera	Altera www.altera.com	$12.000-$17.000

Table 4: FPGA Unit Prices

From Table 4, it is clear that Cyclone III from Altera, and SPARTAN 3A from Xilinx are the most cost effective FPGAs.

However, migration to ASIC using Hardcopy devices from Altera, or EasyPath devices from Xilinx can reduce the unit cost by 10-90%.

3.7.3 Application Specific Integrated Circuits

Table 5 shows the details of production cost and volume when converting a design from prototype FPGA/ PLD to structured ASIC:

Company	Minimum Business Size	Estimated per-unit price savings over FPGA/PLD	Time to complete conversion	Time to first prototype	Time to production units
Altera	NA	10-90%	NA	NA	NA
Xilinx	NA	30-70%	NA	NA	8-12 weeks

Atmel	$250,000.000	50-80%	3-4 weeks	3-4 weeks	8 weeks
Orbit Semiconductor	$40,000.000	$5- $50	1- 4 weeks	2-4 weeks	6-8 weeks
S-MOS Systems Inc	10,000 units		2-3 weeks	20 days	8-12 weeks

Table 5: FPGA/PLD to ASIC conversion cost [14]

Table 6 shows the cost of Mask Programmable Gate Arrays (without FPGA prototype):

Company	**Minimum Business Size**	**Estimated per-unit price savings over FPGA/PLD**	**Time to complete conversion**	**Time to first prototype**	**Time to production units**
Altera	10,000 units	50-75%	2 weeks	4-5 weeks	6-8 weeks
Xilinx	3000- 10,000 units	20-80%	2-6 weeks	3 weeks	4-8 weeks
Lucent Technologies	$250,000.000	10-90%	4-2 weeks	2-6 weeks	0-6 weeks
AMI	25,000 units	25- 45 %	2 days	7 weeks	6 weeks

Table 6: MPGA Cost [14]

The minimum business size is the minimum number of units or the minimum production cost. Altera device technique from FPGA to structured ASIC is called HardCopy. In Xilinx a similar device is called EasyPath, while in Atmel; the similar device is called ULC.

3.7.4 Conclusion

For DSP case, the unit cost is nearly the same as the production cost. While for FPGA, the unit cost is nearly the same as production cost, only when we do not

consider migration to structured ASIC, in which case production unit cost reduces by 10-90%.

3.8 Brief Overall Comparison

Table 7 gives a summary of the points of comparison between Digital Signal Processors, Field Programmable Gate Array, Structured Application Specific Integrated Circuit and full-custom Application Specific Integrated Circuit:

	Digital Signal Processor	**Field Programmable Gate Array**	**Structured - Application Specific Integrated Circuit**	**Full-Custom Application Specific Integrated Circuit**
Design Time and NRE cost	Short time and low cost	Medium time and cost	Medium time and cost	Long time and high cost
Re-programmability and flexibility	High	Limited	None	None
Hardware Resources	Met and exceeding	Met	Met	Met
On-chip memory	Met and exceeding	Met	Met	Met
Processing Time	Met and exceeding	Met and exceeding	Met and exceeding	Met and exceeding
Power Consumption	Low-Medium	Low-Medium	Low	Low
Suitability for extension or addition of new modules	Suitable	Suitable if high capability FPGA is used	Suitable if high capability FPGA is	Not suitable

			used	
Production cost	Medium	Medium-High	Low if high production volume	Low if high production volume
Production volume	Can be small	Can be small	Required to be high	Required to be high

Table 7: Brief overall comparison between the three design styles

3.9 Conclusion

From the results obtained in the previous sections, two design approaches are recommended for the front end processor system:

3.9.1 Using DSP Processor

This option has the following advantages:

- Short development time and less design effort.
- Flexibility to include extra modules on the same processor to utilize the extra resources.

And it has the following disadvantages:

- High power consumption, which reduces the possibility of battery powered solution, such that, the chip shall take its power from a rechargeable battery (like the one in PDA or mobile device or even a lap-top). In this case, it will not be possible to place the front end processor in the microphone piece, as it will require high power source.

3.9.2 Migration from FPGA to Structured ASIC

This option has the following advantages:

- The same design steps as developing an FPGA prototype are done with no extra effort for migration.
- FPGA prototyping gives extra design flexibility before migrating to structured ASIC. Also, the RTL prototype can be used as an

independent IP core (soft macro) that can be included as an embedded block (System on Chip - SoC) in a larger system if needed.

- Lower unit Cost (10-90 % reduction over FPGA).
- Lower power consumption (around 50% power reduction than the corresponding FPGA).
- Higher design security features of Systems on chip (SoC).

And has the following disadvantages:

- Requires high production volume to be worth the cost of migration.
- Less design flexibility, especially after the migration step.

The decision to take this option will be limited with the required production volume. However, this design style will be adopted in this thesis because it has the lowest power consumption, cost, and reliability and design security. In addition, no extra effort or risk is added to migrate to the ASIC solution of the prototyped, well-tested design on the FPGA. And finally, migration technologies are available at most of the FPGA manufacturers (like Altera HardCopy and Xilinix EasyPath devices), so no matter the choice of the FPGA platform, a corresponding migration technology is easily available.

Chapter 4

4 System Design and Implementation

In this chapter, the system design of the front end speech processor chip is presented. First, the system limitations and constraints are listed, like time, cost and power consumption constraints. The system static and dynamic architectures are then presented, followed by the detailed design of each module in the architecture, featuring the basic functionality of the module, the internal architecture, the configuration parameters, signal widths justification for the module internal signals, the module state machine, the memory requirements of the module and finally the chip usage of that module.

Then the overall system performance is discussed, where the resources utilization and memory requirements on different FPGA platforms are presented. Also, the time and speed performance of the system is described.

Finally, a theoretical study of the effect of run-time configuration of different parameters instead of the static configuration of those parameters on the system is presented, with the modified architecture and the required modifications in every component of the system.

Note that; this chapter contains information that depends on the system specifications for the front end processor described earlier in the thesis and as specified in [2].

4.1 Design Constraints

In this section the time, cost and power constraints on the design are discussed in details.

4.1.1 Time Constraints

The number of samples per frame is N samples. The frame shift interval (difference between the starting points of consecutive frames) is M samples.

The parameter M defines the number of frames per unit time. For more information about this section, please see [2].

The specific values of N and M depend on the sampling rate according to Table 8. The frame length is 25 ms for 8 and 16 kHz sampling rates, and 23,27 ms for 11 kHz.

Sampling rate (kHz)	f_{s3} = 16	f_{s2} = 11	f_{s1} = 8
Frame length *N* (samples)	400	256	200
Shift interval *M* (samples)	160	110	80

Table 8: Supported Configurations Supported options [2]

The 3 supported options provide different shift samples (M), different frame length in samples (N) and different frame interval. However, the frame shift interval is constant in the three cases and equal 10 ms. For example, for the 8 kHz case the sample duration will be:

$$SampleDuration = \frac{FrameLengthInTime}{FrameLengthInSamples} = \frac{25ms}{N(200)} = 0.125\ ms = 1/(8\ kHz)$$

$$ShiftInterval = M \times SampleDuration = 80 \times 0.125\ ms = 10\ ms$$

The above formulae hold for 11 and 16 kHz sampling rates. Hence;

$$FrameRate = \frac{1}{ShiftInterval} = 100\ Frame/Sec$$

The major constraint is on the processing time of each speech frame; that is, it shall end in less than 9.16 ms, where the time between two frames is 10 ms (for the 3 supported sampling rates), however, for every 2 frames (88 bits) there is additional 4 bits of CRC, and every 24 frames (144 byte) we have 6 bytes of overhead (header and sync. sequence), which makes the effective frame rate = 10 ms * (144 – 0.5 * 12 - 6)/144 = 9.16 ms. However, compared to today's chip frequencies, this constraint is very relaxed, hence, most optimizations were directed towards hardware resources rather than processing time optimization.

4.1.2 Memory Constraints

Memory usage should be optimized as much as possible, especially it is not planned to use any external memories other than that on chip, since this will degrade the performance of the system. This constraint implies that most of the calculations will be done at run time whenever possible, and no pre computed constants will be used, unless necessary (e.g. the quantization tables should be stored and cannot be computed).

4.1.3 Power Consumption

The front end processor system is intended to be deployed in a stand alone chip, for example, it could be placed in the microphone piece. This means that the final system could be battery powered, which implies that the power consumption should be hold at its minimum values as much as possible. This constraint will affect the choice of the hardware platform to be used, where FPGA or structured ASIC choices are preferred to Digital Signal Processors.

4.1.4 Cost and Resources Constraints

Since the final system is meant to be placed in a separate chip that could be part of the microphone, so the cost should be kept at its minimum. This constraint makes the choice of the target hardware platform moves towards FPGA or structured ASIC rather than DSP. Also, the hardware resources on that chip are expected to be very limited. Hence, numerical algorithms (like CORDIC, see Appendix A) were used to compute complicated DSP (like FFT, see Appendix A) and trigonometric functions, which highly reduced the resources usage with good accuracy. Also, reuse of resources between serial operating components was used in the design.

4.2 System Architecture

This section describes the high level architecture of the front end processor system. The block diagram of the system as described in the Aurora standard in [2] is shown in Figure 35:

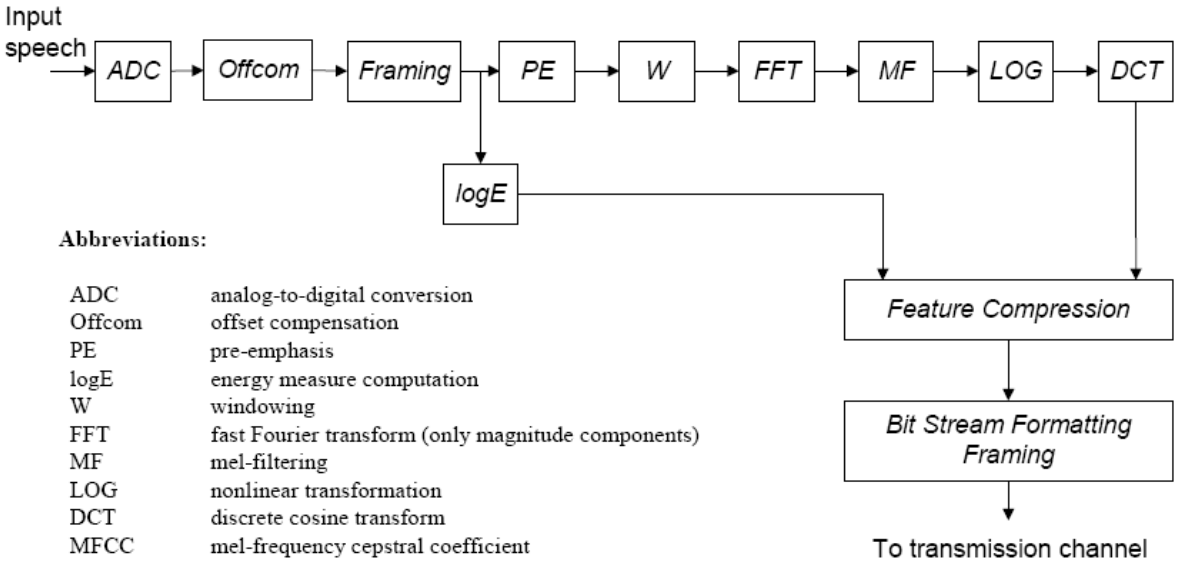


Figure 35: Block diagram of the system [2]

The following two sections show the static and dynamic architectures of the system.

4.2.1 Static Architecture

This section describes the internal components of the front end processor without specifying the interaction between them nor the inputs or outputs of each block. The main component that manages the state machine of the system is the Buffer Manager. The Buffer Interface component is the interface between the ADC and the rest of the system, it could be implemented as a shared memory, or direct link between the ADC and the system, in either case the implementation of this component is outside the scope of the design.

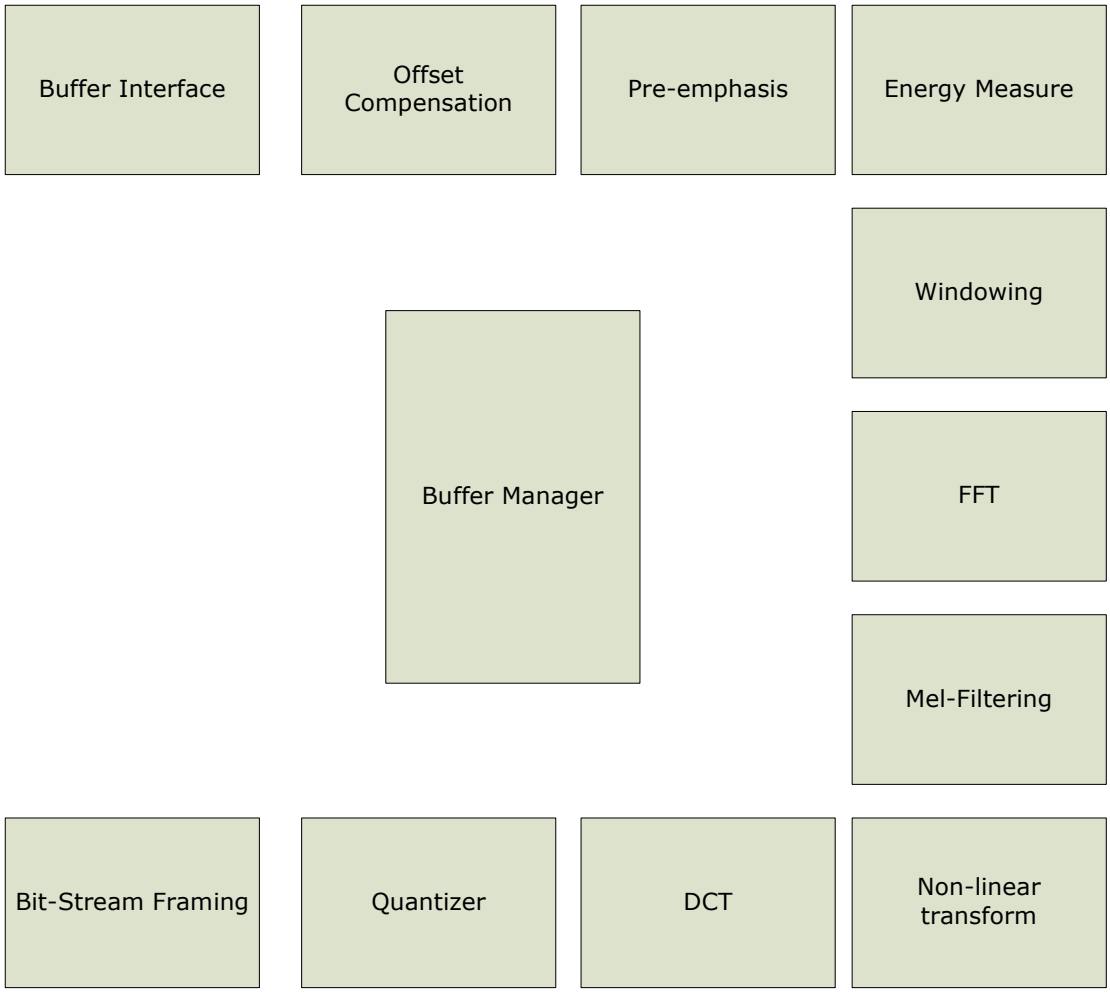


Figure 36: Static Architecture of the system

4.2.2 Dynamic Architecture

This section describes the dynamic behavior of the system. There are many points that can be extracted from the dynamic architecture:

1. The communication between internal components.
2. The widths of the signals exchanged between components.
3. The sequence of events and flow of data in the system.
4. The dependence between the components, i.e. some components can be running in parallel and others are dependent on each other.

It should be noted that: the names of the signals mentioned here are not necessarily the same in the actual design. Also, only the main signals are shown here, which means that more signals could exist in the actual design and

not mentioned here. And finally, some signals shown here could be exchanged between components indirectly, for example, the bink coefficients coming out of the FFT component to the Mel-Filter are not directly exchanged between them in the actual design, actually, they are written by the FFT in a shared memory, and then read by the Mel-filter form the same memory under the control of the buffer manager.

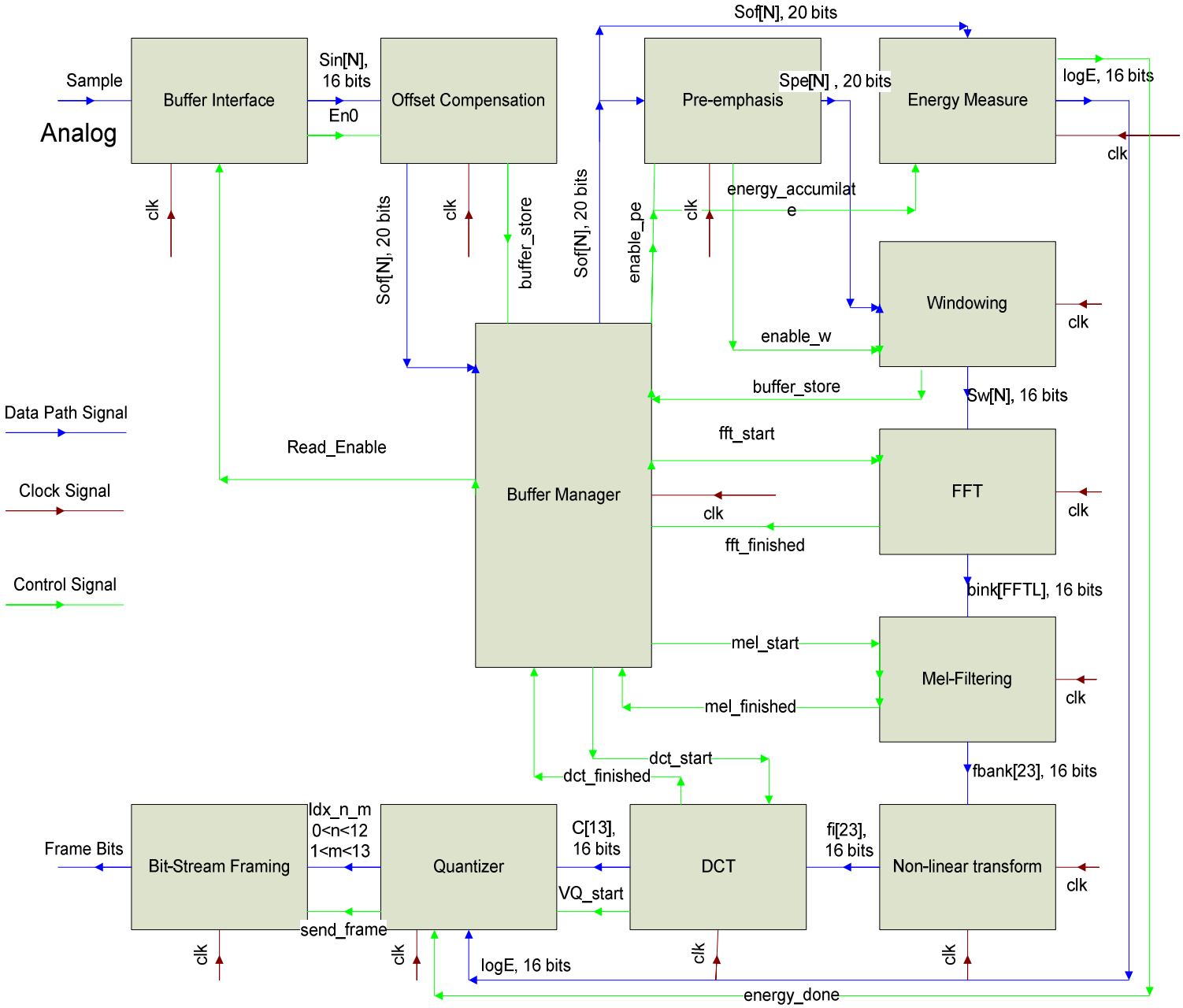


Figure 37: Dynamic Architecture of the system

The system can be divided into three main parts:

1. The first part includes the following components: Offset Compensation, Pre-emphasis, Energy Measure and Windowing. The components of this part can run in parallel, however, since the rate of change of the input samples (which is the sampling rate) is much slower than the time needed to process the sample these modules will never run in parallel. The processing in this part is done per sample, which means that, whenever a new sample is present, processing is made on it, and the result is stored in the data buffer under the control of the Buffer Manager. So, these components are driven by the input samples existence.
2. The second part consists of the rest of the system except the Buffer Manager. The components of this part are dependent on each other, that is; every component should wait the result of the previous component to be ready to start its function. This operation is controlled by the Buffer Manager, where a *finished* signal is generated by each component when it finishes its job, and a *start* signal is generated by the buffer manager to trigger every component to start working.

   The state machine of the sequence of activating those components is managed by the Buffer Manager as will be discussed in details in its corresponding detailed design section.

   Also, this part operation is done on a block of N samples, that means; in the very beginning of the system operation, the first N samples (first frame) should be stored in the buffer first to start the first operation of that part. For the consecutive frames, every M samples, the Buffer Manager will start the operation of this part again according to its state machine sequence of operation.

   The only exception to the serial operation of the components of this part is the Non-Linear Transformation and the Mel-Filter, where the Log is computed for every coefficient of the filter at the moment it is generated, in parallel with the computation of the next coefficient.

Although the components of this part run serially, however, there is a high degree of parallelism inside some of these components. For example the magnitude of the final output of the FFT component will be computed in parallel with the final stage computations of the FFT algorithm.

A final note on the second part of the system is that, since the components of this part should run in serial, so some components, like CORDIC cores or multipliers can be reused among them.

3. The third part is the Buffer Manager component itself, which coordinates the operation of the first and second part, in addition to managing the state machine of the second part. Also, this component will be responsible of managing the access to the memory shared between the first and second parts, and also the memories shared between the components of the second parts.

4.2.3 Modules Detailed Design

In this section the details of the internal design of each module are presented. Every section contains the module basic functionality, the internal architecture of the module, the configuration parameters to configure the module, the signal widths justification, the state machines definition in the module, the memory (RAM/ROM) requirements, the actual chip usage of the available resources and finally the Processing time taken by the module to finish its task. Some of these sections do not exist for some modules, in which case they will be described as *None.*

4.2.3.1 Offset Compensation

4.2.3.1.1 Basic functionality

Prior to the framing, a notch filtering operation is applied to the digital samples of the input speech signal *Sin* to remove their DC offset, producing the offset-free input signal *Sof.* The main function of the module is to calculate the following equation:

$$s_{of}(n) = s_{in}(n) - s_{in}(n-1) + 0{,}999 \times s_{of}(n-1)$$

This calculation is done whenever a new sample *Sin* is present.

4.2.3.1.2 Internal Architecture

This section shows the data flow graph of the module. This graph is just for design purpose and does not mean that the final synthesized hardware on the chip will look like that, yet it should be very near to it. At the very beginning of the system operation, Sin(n-1) and Sof(n-1) are zeros, this is done by a MUX activated by the reset signal. Then with every new sample, the basic equation is calculated in one clock, then the values of Sin and Sof are stored, which will be used in the next time as Sin(n-1) and Sof(n-1).

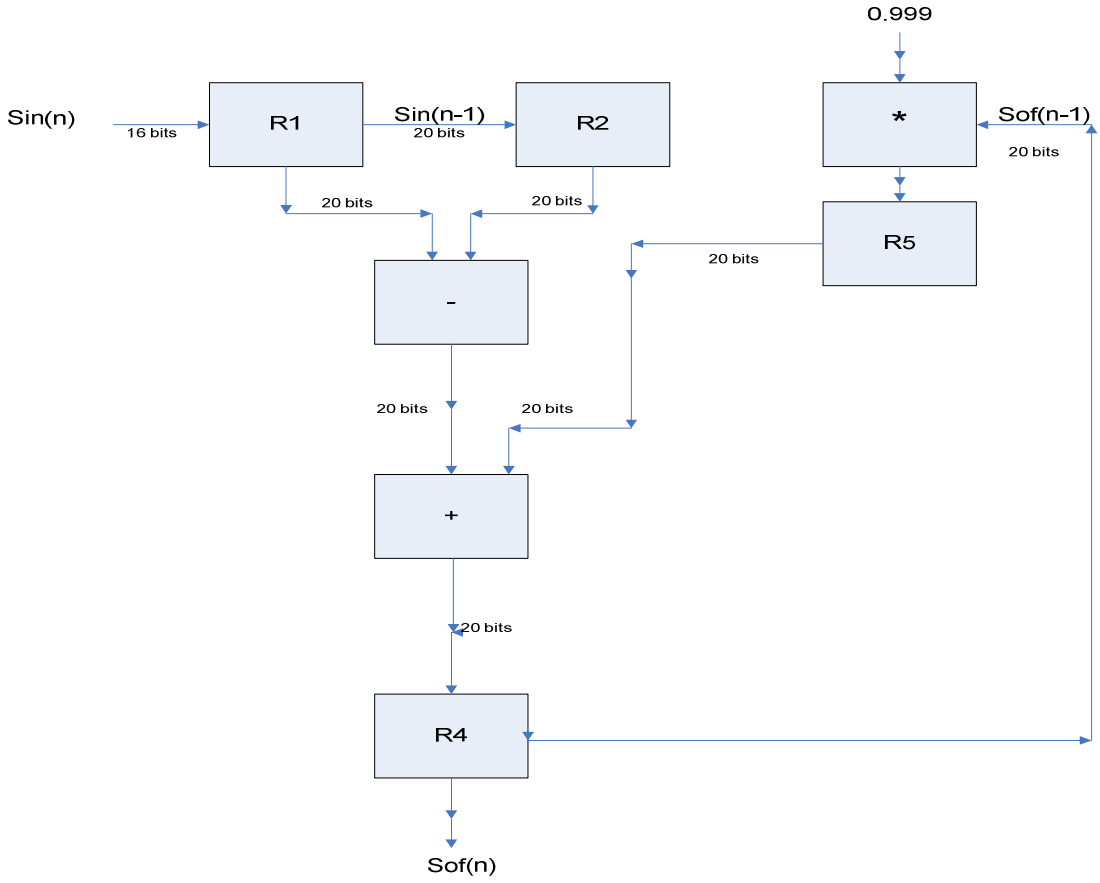


Figure 38: Offset Compensation data flow graph

4.2.3.1.3 Configuration

None

4.2.3.1.4 Signal width justification

The input signal is 16 bits as generated by the A/D. This 16 bits input will be treated as a fixed point number, such that the actual value of the input signal is considered between 0 and 1. This is equivalent to dividing the number by 65536. This division will be compensated later in the system after the calculation of the DCT, which can be done in one of two fashions:

- This division by 65536 will propagate through the whole system till the output of the Mel-Filter, after that at the stage of the Non-Linear transformation, the division will be converted to subtraction of Ln(65536), which can be compensated by adding this constant to the result, or waiting to the DCT stage, and adding similar constant to every coefficient of the resulting 13 coefficients of the DCT, where every compensation constant will be calculated as:

$$Const(i) = \sum_{j=1}^{23} Ln(65536)\cos(\frac{\pi \times i}{23}(j-0.5)), 0 \le i \le 12$$

- The other way is to do this compensation in the quantization table values once, and store the modified tables, i.e. store the tables after subtracting the above calculated constants from the values of each coefficient quantization table, and store the result.

Solution 2 will be adopted, since the tables are statically stored in ROM, so, storing the adjusted tables saves the processing time and resources required to compensate the division constant with every DCT coefficient.

Note that: similar manipulation will be done on LogE feature, where the following constant should be subtracted from its quantization table entry:

$$LogE\_Const = \ln(\sum_{i=1}^{N}(65535)^2)$$

The rest of signals in that module are of width 20 bits, where:

- The 20^{th} bit is the sign bit.
- The next 4 bits (19 to 16) represent the Integer part of the number.
- The rest of bits represent the fraction part.

The above widths were decided based on numerical tests on real frames of speech, which showed that the dynamic range of the Integer part of the resulting *Sof* requires 4 bits to avoid overflow at this early stage of the system. The alternate solution to adding those 4 bits was to reduce the fraction part by 4 bits (shift right *Sin* 4 locations) to be just 12 bits; however this was not done due to the following reasons:

- This would reduce the accuracy of the system at a very early stage of processing.
- Since this filter depends on the previous output (IIR filter), hence, any error due to fixed point calculation will propagate in all the next frames and will be magnified, so it is highly recommended to be as accurate as possible in this calculation.
- This part of the system (till the windowing component) operates on a sample-by-sample basis, which means that it requires only a storage of the previous sample only, so it will not be a big loss to add 4 bits to single internal register that holds the previous sample. This is unlike the modules that operate on the whole N samples of the frame, in which case it requires to add these 4 bits to the whole samples of the buffer, which would add 4*N bits to the total memory requirement of the system.

4.2.3.1.5 State Machines

None

4.2.3.1.6 Memory requirements

Memory	**Size**	**Description**
sof_prev	20 bits	To hold Sof(n-1)
sin_prev	20 bits	To hold Sin(n-1)

Table 9: Memory requirements of the Offset Compensation component

4.2.3.1.7 Actual Chip Usage

The following is a summary of the chip usage as generated by the Quartus II software:

Flow Status Successful - Wed Oct 01 08:52:49 2008 *Quartus II Version 7.2 Build 203 02/05/2008 SP 2 SJ Web Edition* *Revision Name source_tb* *Top-level Entity Name Offset_Compensation_1* *Family Cyclone III* *Device EP3C10U256C8* *Timing Models Preliminary* *Met timing requirements N/A* *Total logic elements 169 / 10,320 (2 %)* *Total combinational functions 167 / 10,320 (2 %)* *Dedicated logic registers 2 / 10,320 (< 1 %)* *Total registers 2* *Total pins 40 / 183 (22 %)* *Total virtual pins 0* *Total memory bits 0 / 423,936 (0 %)* *Embedded Multiplier 9-bit elements 4 / 46 (9 %)* *Total PLLs 0 / 2 (0 %)*

Figure 39: Summary of resources usage of Offset Compensation module

4.2.3.1.8 Processing time

Let:

1. Number of clocks taken by adder = n = 1.
2. Number of clocks taken by multiplier = m = 1.

Therefore:

ProcessingTime = max(n, m) = 1

4.2.3.2 Pre-Emphasis filter

4.2.3.2.1 Basic functionality

A pre-emphasis filter is applied to the framed offset-free input signal:

$$s_{pe}(n) = s_{of}(n) - 0{,}97 \times s_{of}(n-1)$$

Here *Sof* and *Spe* are the input and output of the pre-emphasis block, respectively.

4.2.3.2.2 Internal Architecture

This section shows the data flow graph of the module. This graph is just for design purpose and does not mean that the final synthesized hardware on the chip will look like that, yet it should be very near to it. Every time the module is enabled, the basic calculation is performed on the input Sof(n), then the result is stored as Sin(n-1) to be used in the next time.

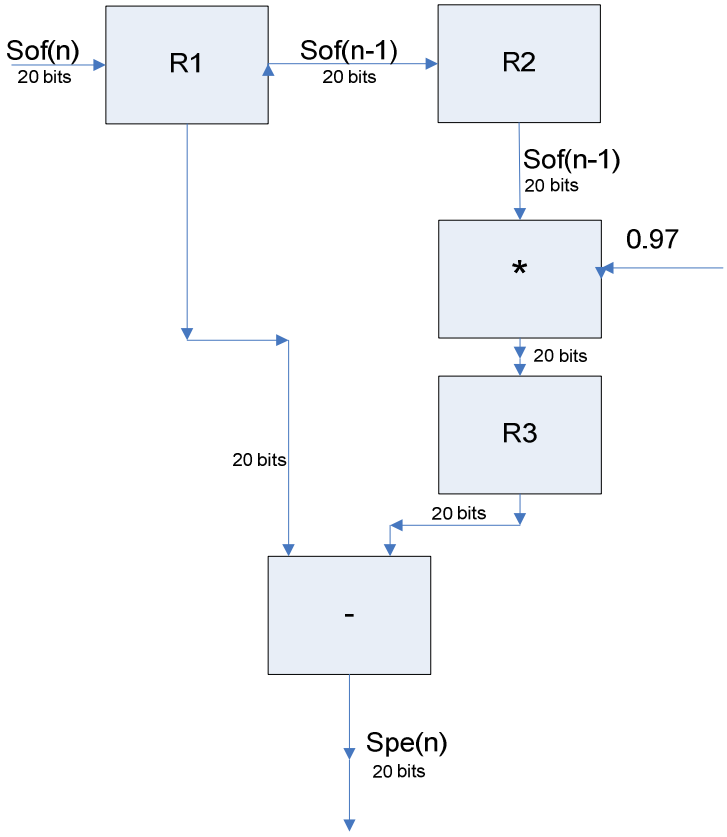


Figure 40: Pre-emphasis data flow graph

4.2.3.2.3 Configuration

None.

4.2.3.2.4 Signal width justification

The 20 bits width choice follows the same justification as that of the Offset Compensation component.

4.2.3.2.5 State Machines

None.

4.2.3.2.6 Memory requirements

Memory	Size	Description
sof_prev	20 bits	To hold Sof(n-1)

Table 10: Memory requirements of the Pre-emphasis component

4.2.3.2.7 Actual Chip Usage

The following is a summary of the chip usage as generated by the Quartus II software:

Flow Status Successful - Wed Oct 01 09:04:30 2008
Quartus II Version 7.2 Build 203 02/05/2008 SP 2 SJ Web Edition
Revision Name source_tb
Top-level Entity Name Pre_Emphasis
Family Cyclone III
Device EP3C10U256C8
Timing Models Preliminary
Met timing requirements N/A
Total logic elements 152 / 10,320 (1 %)
Total combinational functions 150 / 10,320 (1 %)
Dedicated logic registers 2 / 10,320 (< 1 %)
Total registers 2
Total pins 44 / 183 (24 %)
Total virtual pins 0
Total memory bits 0 / 423,936 (0 %)
Embedded Multiplier 9-bit elements 4 / 46 (9 %)
Total PLLs 0 / 2 (0 %)

Figure 41: Summary of resources usage of Pre-emphasis module

4.2.3.2.8 Processing time

Let:

1. Number of clocks taken by adder = n = 1.
2. Number of clocks taken by multiplier = m = 1.

Therefore:

$$\text{ProcessingTime} = \max(n, m) = 1$$

4.2.3.3 Energy Measure

4.2.3.3.1 Basic functionality

The logarithmic frame energy measure (logE) is computed after the offset compensation filtering and framing for each frame:

$$\text{logE} = \ln\left(\sum_{i=1}^{N} s_{of}(i)^2\right)$$

Here N is the frame length and *Sof* is the offset-free input signal.

4.2.3.3.2 Internal Architecture

This section shows the data flow graph of the module. This graph is just for design purpose and does not mean that the final synthesized hardware on the chip will look like that, yet it should be very near to it. First Sof(i)^2 is calculated, then accumulated to the old energy measure. After the N samples are accumulated, the resulting energy is placed as an input to a CORDIC core that is configured to calculate the Logarithm function,(for more information about CORDIC algorithm, please see Appendix A). Note that, the CORDIC core is only enabled after the Nth sample is accumulated, and the final energy is ready.

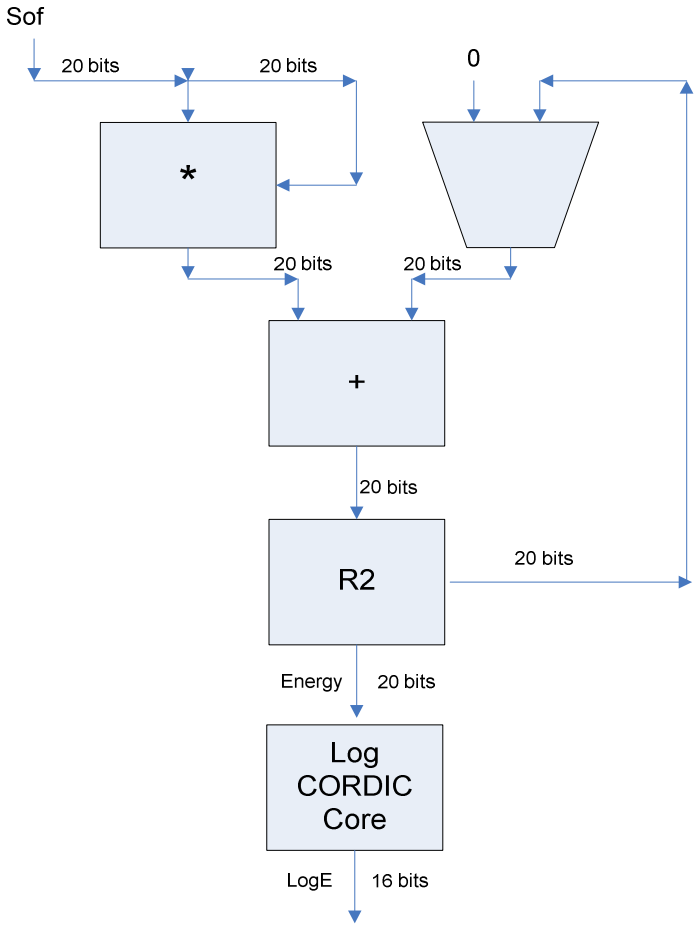


Figure 42: Energy Measure data flow graph

4.2.3.3.3 Configuration

Configuration Parameter	Possible values	Default value	Description
N	200/256/400 (for SamplingRate = 8/11/16)	200	The frame length in samples.

Table 11: Energy Measure module configuration parameters

4.2.3.3.4 Signal width justification

The accumulated energy signal width is 20 bits, with 10 bits as the integer part and 10 bits as the fraction part in case of 8/11 kHz sampling rate,

and 11 bits as the integer part and 9 bits as the fraction part in case of 16 kHz sampling rate. This is because analysis of the quantization tables used to quantize the LogE feature shows that the maximum energy value can be put in 9 bits for 8/11 kHz and 10 bits for 16 kHz, so that the dynamic range of the energy feature can be hold in 9/10 bits; however 10 bits for 8/11 kHz and 11 bits for 16 kHz were taken for safety measures. Also, the minimum distance between two entries in the quantization table show that the minimum resolution required to quantize the LogE feature only requires 7 bits, that is the fraction part should not be hold in less than 7 bits, otherwise the quantization resolution will not be maintained, leading to wrong quantization, hence 10 bits for the fraction part are enough.

4.2.3.3.5 State Machines

None

4.2.3.3.6 Memory requirements

Memory	Size	Description
Energy	20 bits	To hold accumulated energy of the current frame.

Table 12: Memory requirements of the Energy Measure component

4.2.3.3.7 Actual Chip Usage

The following is a summary of the chip usage as generated by the Quartus II software:

Flow Status Successful - Wed Oct 01 09:21:49 2008
Quartus II Version 7.2 Build 203 02/05/2008 SP 2 SJ Web Edition
Revision Name source_tb
Top-level Entity Name Energy_Measure
Family Cyclone III
Device EP3C10U256C8
Timing Models Preliminary
Met timing requirements N/A
Total logic elements 739 / 10,320 (7 %)
Total combinational functions 736 / 10,320 (7 %)

Dedicated logic registers	*168 / 10,320 (2 %)*
Total registers	*168*
Total pins	*40 / 183 (22 %)*
Total virtual pins	*0*
Total memory bits	*0 / 423,936 (0 %)*
Embedded Multiplier 9-bit elements	*6 / 46 (13 %)*
Total PLLs	*0 / 2 (0 %)*

Figure 43: Summary of resources usage of Energy Measure module

4.2.3.3.8 Processing time

Let:

1. Number of clocks taken by multiplier = m = 1.
2. Number of clocks taken by adder = n = 1.
3. Number of clocks taken by Log calculator circuit = M = 16.

Therefore,

$$ProcessingTime = max(m , n) * N * M + M = (N + 1)*16$$

Assuming that the minimum time between two samples is M clocks.

4.2.3.4 Hamming Window

4.2.3.4.1 Basic functionality

A Hamming window of length N is applied to the output of the pre-emphasis block:

$$s_w(n) = \left\{ 0{,}54 - 0{,}46 \times \cos\left(\frac{2\pi(n-1)}{N-1} \right) \right\} \times s_{pe}(n),\ 1 \le n \le N$$

Here N is the frame length and *Spe* and *Sw* are the input and output of the windowing block, respectively.

4.2.3.4.2 Internal Architecture

The module is enabled by the Pre-emphasis module every new sample, accordingly, the following constant is calculated as shown in Figure 44. Then, the result is multiplied by the input *Spe(n)*. The constant circuit uses CORDIC core similar to the one in Appendix A. The cosine argument depends on the

sample number, so, a modulo-9 bits counter is used to indicate the sample number (n) in case of N=512, or modulo-8 in case of N=256. This counter is not kept internally in the module, however, it is kept in the Buffer Manager, and the sample count is provided as an input to the Window component.

$$\left\{0{,}54 - 0{,}46 \times \cos\left(\frac{2\pi(n-1)}{N-1}\right)\right\}$$

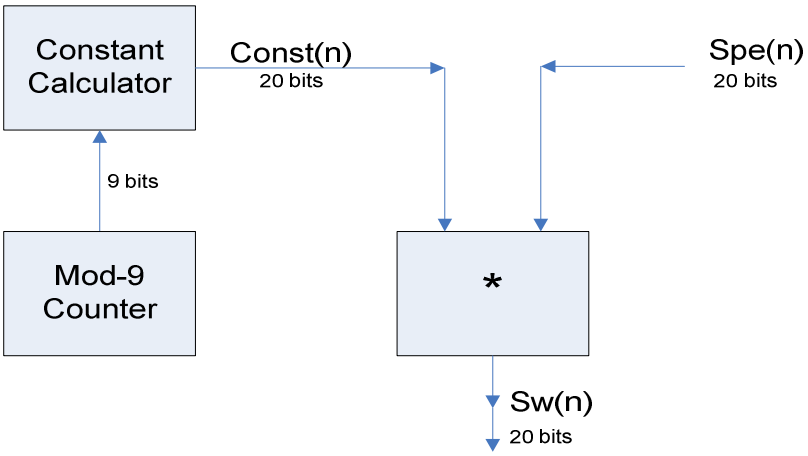


Figure 44: Hamming Window data flow graph

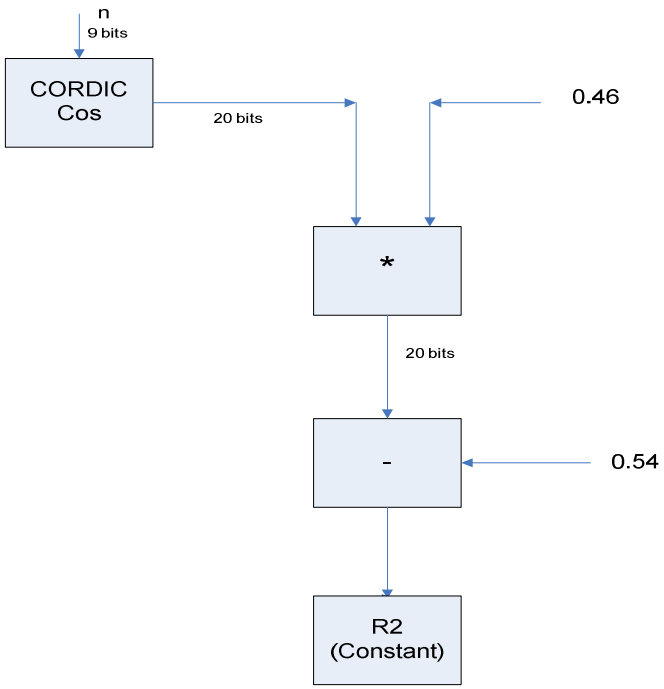


Figure 45: Constant calculation of Hamming Window Filter

4.2.3.4.2.1 Look-up table implementation

The basic Hamming window filtering described in 4.2.3.4.1 could be implemented in one of two ways; a) to calculate the Hamming window factor, using CORDIC core for example, or b) storing the Hamming window factors in a *Look-up table* (LUT). The first method was already described in 4.2.3.4.2. In the LUT method, only half of the window factors are stored in a ROM of length equals N/2, where N is the frame length in number of samples, and the width of the stored factors is chosen to be 20 bits.

4.2.3.4.3 Configuration

Configuration Parameter	Possible values	Default value	Description
N	200/256/400 (for SamplingRate = 8/11/16)	200	The frame length in samples.

Table 13: Hamming Window module configuration parameters

4.2.3.4.4 Signal width justification

The 20 bits width choice follows the same justification as that of the Offset Compensation component.

4.2.3.4.5 State Machines

The module has an internal state machine to follow the CORDIC calculation state as shown in **Figure 46**:

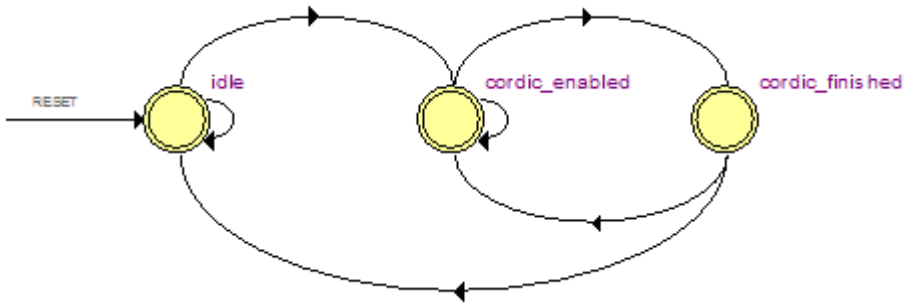


Figure 46: State machine of the Window component

Source State	Destination State	Condition
Idle	idle	(!ENABLE)
Idle	cordic_enabled	(ENABLE)
cordic_enabled	cordic_enabled	(!cordic_done)
cordic_enabled	cordic_finished	(cordic_done)
cordic_finished	idle	(!ENABLE)
cordic_finished	cordic_enabled	(ENABLE)

Table 14: State transition of the Window state machine

In general, the module is initially in the IDLE state unless enabled. When enabled, it goes to the CORDIC_ENABLED state, at which the CORDIC module is enabled to calculate the cosine part of the constant. The module will remain in this state till the cordic_done signal is raised by the CORDIC processor. When the CORDIC is finished, the module goes to the CORDIC_FINISHED state, where the final constant is calculated and multiplied by *Spe* to get the final *Sw*.

4.2.3.4.6 Memory requirements

Memory	Size	Description
Sample Counter	9 bits	To hold the current sample index. The size is calculated on the maximum needed number of samples (N=512).

Table 15: Memory requirements of the Window component

In case of LUT implementation, extra ROM of size equals (N/2 X 20) bits is needed.

4.2.3.4.7 Actual Chip Usage

The following is a summary of the chip usage as generated by the Quartus II software, it is based on the CORDIC implementation:

Flow Status	*Successful - Wed Oct 01 09:37:00 2008*
Quartus II Version	*7.2 Build 203 02/05/2008 SP 2 SJ Web Edition*
Revision Name	*source_tb*
Top-level Entity Name	*window_1*
Family	*Cyclone III*
Device	*EP3C10U256C8*
Timing Models	*Preliminary*
Total logic elements	*781 / 10,320 (8 %)*
Total combinational functions	*778 / 10,320 (8 %)*
Dedicated logic registers	*71 / 10,320 (< 1 %)*
Total registers	*71*
Total pins	*53 / 183 (29 %)*
Total virtual pins	*0*
Total memory bits	*0 / 423,936 (0 %)*
Embedded Multiplier 9-bit elements	*14 / 46 (30 %)*

Figure 47: Summary of resources usage of Window module

4.2.3.4.8 Processing time

Let:

- Number of clocks taken by multiplier = m =1.
- Number of clocks taken by adder = n = 1.
- Number of clocks taken by cosine calculator circuit = M = 12.

Therefore:

$$ProcessingTime = min(m,n) + M = M + 1 = 13$$

This is the processing time to apply hamming window filter to one sample only.

In case of LUT implementation mentioned in 4.2.3.4.2.1, only *min(m,n)* clocks are needed.

4.2.3.5 Buffer Manager

4.2.3.5.1 Basic functionality

This module is responsible of:

- Managing the state machine of the second part of the system, which includes
  - FFT
  - Mel-Filter

- Non-Linear Transformation
- DCT
- Vector Quantization
- Bit-Framing

This is done by activating these modules in sequence, such that each module is not activated unless the previous one has finished.

- Padding zeros to the frame length before activating the FFT module.
- Managing the frame overlapping of samples, where it controls the read and write addresses from the 2N RAM, and provides them to the FFT to operate on them. This is done by managing the access to the 2N RAM as a circular buffer.
- Managing the access to the memory shared between the components.

4.2.3.5.2 Internal Architecture

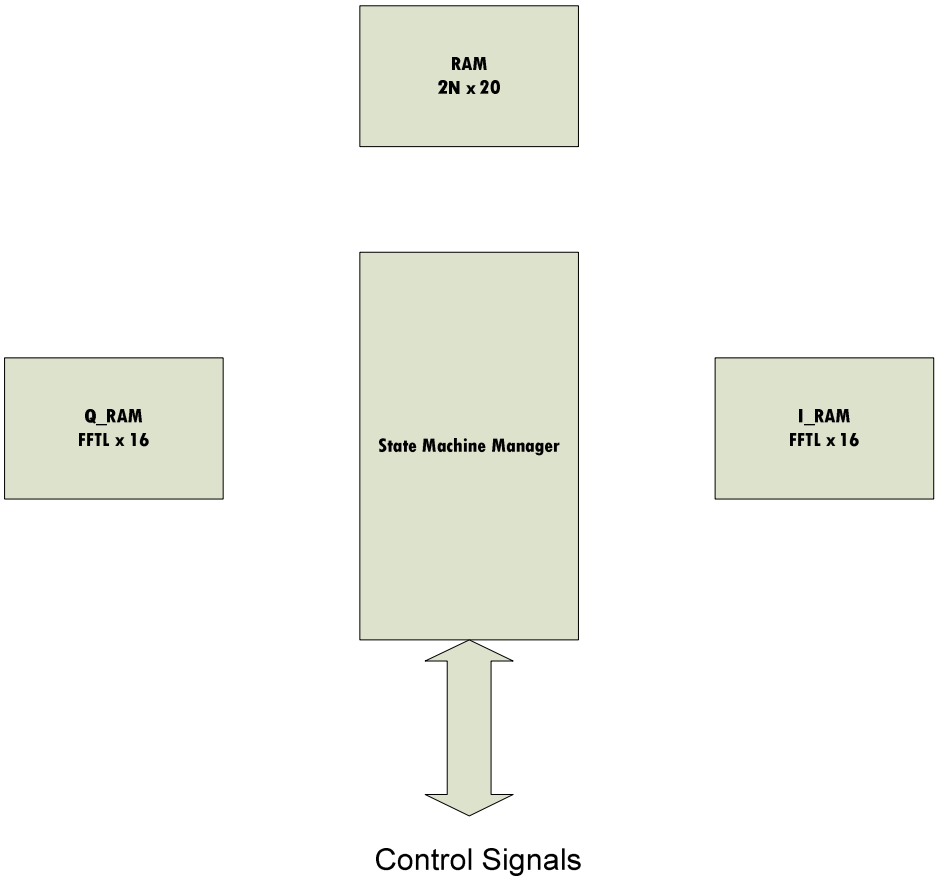


Figure 48: Internal Architecture of Buffer Manager

The module contains three memories:

- *2N RAM:* this stores every sample after being processed by the Window module. The accesses to this memory is managed by the Buffer Manager itself, where it provides the read/ write signals, and manages the read/ write addresses. This memory is managed as a circular buffer, where a write address is advanced till it reaches the end of the buffer, and then it rolls over to the start of the buffer again. Write to this memory is triggered with every new sample after it is process with Window component.

  The read address from this memory is advanced with M samples every new frame. The read signal is generated every new sample, and the read data is input to the Pre-Emphasis filter, then the result is passed by the Hamming Window filter, and at the end the final result is stored in the *I RAM* buffer in a bit reversed order, till N samples are accumulated and then the next stages of the FFT operation can start. Alternatively, the read data is passed to the Energy Measure module to be accumulated to calculate the LogE feature of the frame. In this way, frame overlapping specified in the standard is managed.
- *I RAM and Q RAM:* these are used as real and imaginary memories to serve as the in-place buffer that is used in the FFT algorithm. At the end of the FFT operation, the *I RAM* should contain the magnitude of the real and imaginary FFT coefficients, and the *Q RAM* is free.

  The two RAMs are then reused with the Mel-Filter and DCT modules, where one of the two memories is used as the input buffer to the module, and the other one is used as the output buffer, then these roles are exchanged, where the memory that was acting as input buffer in the previous module will act as output buffer with the next module, and so on.

  The access to I and Q RAMs is given to the module that is currently active according to the state machine mentioned in 4.2.3.5.5. This state machine is managed by the State Machine Manager block shown in the Figure 48, and the control signals are generated in accordance to the

current state. The control signals are mainly divided into the following groups:

- Memory read and writes signals.
- Activation and deactivation signals to the modules in the second part of the system defined in 4.2.2.

Note that: the FFT module needs a two entries access simultaneously to the *I RAM* and *Q RAM* in the butter fly operation, this is why two read/ write signals, data and addresses are provided to the FFT module.

Before a new frame can be processed, the *I RAM* and *Q RAM* must be reset, in this way zero padding before FFT is accomplished.

4.2.3.5.3 Configuration

Parameter	**Possible values**	**Default value**	**Description**
N	200/256/400 (for SamplingRate = 8/11/16)	200	The frame length in samples.
M	80/110/160 (for N = 200/256/400)	80	The frame shift in samples.
Awidth	8/9 (for FFTL= 256/512)	8	The address width of the I,Q memories. The 2N RAM address width is 1 bit more than that width.
FFTL	256 (for N=200/256)/ 512 (for N=400)	256	The FFT frame length in samples after padding.

Parameter	Possible values	Default value	Description
N_MEL	23	23	The number of Mel-Filter banks.
N_CEPSTRAL	13	13	The number of Cepstral coefficients.
Dwidth	16	16	The data width of the I and Q RAM's memory. This width will be the fixed width in the FFT, Mel-Filter, DCT, Vector Quantizer modules.
Parameter	Size	Size	Description
Iwidth	8/9 (for FFTL= 256/512)	8	The FFT Integer part width used in fixed point calculations
Fwidth	7/6 (for FFTL= 256/512)	7	The FFT Fraction part width used in fixed point calculations.

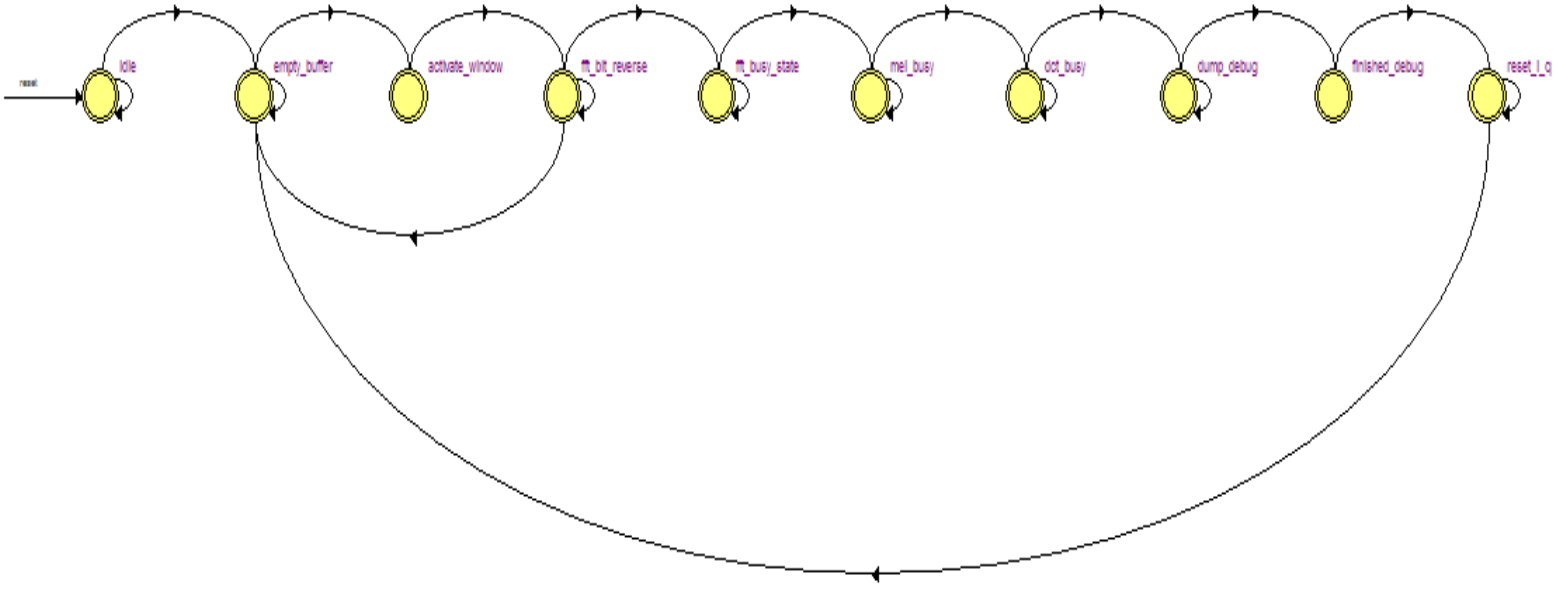
Table 16: Configuration parameters of the Buffer Manager

4.2.3.5.4 Signal width justification

The input to the Buffer Manager is 20 bits width. After the window component this width will be fixed to 16 bits, by truncating the 4 least significant bits from the fraction part, and keeping the integer part as it is.

The 16 bits of data in this part of the system is divided as follows:

- In case of FFTL=256
  - 16^{th} bit as the sign bit.
  - 8 bits (15^{th} to 8^{th} bits) as Integer part.
  - 7 bits (7^{th} to 0^{th} bits) as Fraction part.
- In case of FFTL=512
  - 16^{th} bit as the sign bit.
  - 9 bits (15^{th} to 7^{th} bits) as Integer part.
  - 6 bits (6^{th} to 0^{th} bits) as Fraction part.



4.2.3.5.5 State Machines

Figure 49: State machine of the Buffer Manager module

Source State	Destination State	Condition
idle	idle	(!start)
idle	empty_buffer	(start)
empty_buffer	empty_buffer	(buffer_empty)

empty_buffer	activate_window	(!buffer_empty)
activate_window	fft_bit_reverse	
fft_bit_reverse	empty_buffer	(!fft_busy_1).(window_1:window)
fft_bit_reverse	fft_bit_reverse	(!window_1:window)
fft_bit_reverse	fft_busy_state	(fft_busy_1).(window_1:window)
fft_busy_state	fft_busy_state	(!fft_done)
fft_busy_state	mel_busy	(fft_done)
mel_busy	mel_busy	(!mel_finished)
mel_busy	dct_busy	(mel_finished)
dct_busy	dct_busy	(!dct_finished)
dct_busy	dump_debug	(dct_finished)
dump_debug	dump_debug	debug_counter < N_CEPSTRAL
dump_debug	finished_debug	debug_counter = N_CEPSTRAL
finished_debug	reset_i_q	None
reset_i_q	empty_buffer	reset_I_Q_counter < FFTL
reset_i_q	reset_i_q	reset_I_Q_counter = FFTL

Table 17: State transition of the Buffer Manager state machine

The description of these states is as follows:

- *IDLE:* the module remains in this state until a start signal is triggered.
- *EMPTY_BUFFER:* the module remains in this state as long as the *2N RAM* is empty.
- *ACTIVATE_WINDOW:* in this state the read sample form the *2N RAM* is input to the Pre-Emphasis filter then to the Hamming Window filter, and the result is fixed to 16 bits and stored in the *I RAM* in a bit-reversed order.
- *FFT_BIT_REVERSE:* in this state the samples are fed to the FFT module. Also, the read address of the *2N RAM* is advanced by M samples in this state, because this state indicates the start of a new frame.

- *FFT_BUSY:* the FFT module is running, the module remains in this state until the FFT module finishes. The access to *I RAM* and *Q RAM* is given to the FFT module in this state. At the end of this state the *I RAM* contains the magnitude of the FFT coefficients and the *Q RAM* is free.
- *MEL_BUSY:* the Mel-Filter is activated, and the module remains in this state until the Mel-Filter finishes processing. In this state the Non-Linear Transformation module is running in parallel with the Mel-Filter. The *I RAM* is the input buffer to the Mel-Filter, and the output coefficients are placed in the *Q RAM.*
- *DCT_BUSY:* the DCT module is activated, and the module remains in this state till DCT finishes. The *Q RAM* is the input buffer. The output Cepstral coefficients are supplied directly to the Vector Quantization module to be quantized.
- *DUMP_DEBUG:* this state is used for testing and debugging purposes, where the memory is dumped to provide its contents as outputs.
- *FINISHED_DEBUG:* this is a transient state to reset internal variables after debugging is finished.
- *RESET_I_Q:* this state is where the *I RAM* and *Q RAM* are reset by writing N*Dwidth zeros in the two memories during N clocks. This state is important, where in this way padding from N to FFTL length is done in the I and Q RAM before activating the FFT component, where the samples are stored in the I RAM in a bit-reversed order in N locations, the remaining locations (FFTL - N) are zeros due to the reset operation, so zero padding is accomplished.

4.2.3.5.6 Memory requirements

Memory	**Size**	**Description**
2N RAM	2*N*20 = 8000 for N= 200 10240 for N= 256 16000 for N= 400	The input samples memory.

I RAM	2*FFTL*Dwidth = 4096 for N=256, Dwidth =16 8192 for N=256, Dwidth =16	The real coefficients for FFT module. Used alternatively as input or output buffer for the next modules.
Q RAM	2*FFTL*Dwidth = 4096 for N=256, Dwidth =16 8192 for N=256, Dwidth =16	The imaginary coefficients for FFT module. Used alternatively as input or output buffer for the next modules.

Table 18: Memory requirements of the Buffer Manager module

4.2.3.5.7 Actual Chip Usage

The following is a summary of the chip usage as generated by the Quartus II software:

Flow Status Successful - Wed Oct 01 09:52:11 2008
Quartus II Version 7.2 Build 203 02/05/2008 SP 2 SJ Web Edition
Revision Name source_tb
Top-level Entity Name Buffer_Manager_2
Family Cyclone III
Device EP3C40F780C8
Timing Models Preliminary
Met timing requirements N/A
Total logic elements 2,415 / 39,600 (6 %)
Total combinational functions 2,386 / 39,600 (6 %)
Dedicated logic registers 257 / 39,600 (< 1 %)
Total registers 257
Total pins 309 / 536 (58 %)
Total virtual pins 0
Total memory bits 16,192 / 1,161,216 (1 %)
Embedded Multiplier 9-bit elements 18 / 252 (7 %)
Total PLLs 0 / 4 (0 %)

Figure 50: Summary of resources usage of Buffer Manager Module: N = 200, M=80

Flow Status	*Successful - Thu Oct 02 10:44:52 2008*
Quartus II Version	*7.2 Build 203 02/05/2008 SP 2 SJ Web Edition*
Revision Name	*source_tb*
Top-level Entity Name	*Buffer_Manager_2*
Family	*Cyclone III*
Device	*EP3C40F780C8*
Timing Models	*Preliminary*
Met timing requirements	*N/A*
Total logic elements	*2,072 / 39,600 (5 %)*
Total combinational functions	*2,057 / 39,600 (5 %)*
Dedicated logic registers	*211 / 39,600 (< 1 %)*
Total registers	*211*
Total pins	*309 / 536 (58 %)*
Total virtual pins	*0*
Total memory bits	*18,432 / 1,161,216 (2 %)*
Embedded Multiplier 9-bit elements	*18 / 252 (7 %)*
Total PLLs	*0 / 4 (0 %)*

Figure 51: Summary of resources usage of Buffer Manager Module: N=256, M=110

Flow Status	*Successful - Thu Oct 02 10:53:06 2008*
Quartus II Version	*7.2 Build 203 02/05/2008 SP 2 SJ Web Edition*
Revision Name	*source_tb*
Top-level Entity Name	*Buffer_Manager_2*
Family	*Cyclone III*
Device	*EP3C40F780C8*
Timing Models	*Preliminary*
Met timing requirements	*N/A*
Total logic elements	*2,406 / 39,600 (6 %)*
Total combinational functions	*2,382 / 39,600 (6 %)*
Dedicated logic registers	*258 / 39,600 (< 1 %)*
Total registers	*258*
Total pins	*316 / 536 (59 %)*
Total virtual pins	*0*
Total memory bits	*32,384 / 1,161,216 (3 %)*
Embedded Multiplier 9-bit elements	*18 / 252 (7 %)*
Total PLLs	*0 / 4 (0 %)*

Figure 52: Summary of resources usage of Buffer Manager Module: N=400, M=160

4.2.3.6 FFT

4.2.3.6.1 Basic functionality

Each frame of N samples is zero padded to form an extended frame of 256 samples for 8 and 11 kHz sampling rate, and 512 samples for 16 kHz. An FFT of length 256 or 512, respectively, is applied to compute the magnitude spectrum of the signal.

$$bin_k = \left| \sum_{n=0}^{FFTL-1} s_w(n) e^{-jnk\frac{2\pi}{FFTL}} \right|, \quad k = 0,\ldots,FFTL-1.$$

Here *Sw(n)* is the input to the FFT block, FFTL is the block length (256 or 512 samples), and *bink* is the absolute value of the resulting complex vector. Radix-2 algorithm is used in the design to calculate the FFT. For more information about Radix-2 algorithm please see Appendix A.

The twiddle factor calculation is translated into vector rotation with the same angle. Hence, CORDIC algorithm was used to do the twiddle factor operation. For information about CORDIC algorithm please refer to Appendix A

4.2.3.6.2 Internal Architecture

First the *Sw(n)* is input to the Bit Reversal module, where the bit-reversed index is calculated, and the sample is stored back in the *I RAM* in the bit-reversed index.

When FFTL (256/512) samples are complete after bit reversal, the Butterfly is enabled, which enables the Address Generator.

For the consecutive FFT stages, the Butterfly reads the two input samples (each with real and imaginary parts) from the *I RAM* and *Q RAM* according to the addresses generated by the Address Generator. Thanks to the dual-port RAM implementation, reading and writing is done in single clock cycle for the whole butterfly operation.

After the two samples are read, the Twiddle factor calculator is enabled, which uses a CORDIC processor to calculate the rotated vector (see Appendix A). When the calculation finishes (after 16 clocks), the Butterfly calculates the two output samples, and stores them in the *I RAM* and *Q RAM*. These two operations (Twiddle factor calculation and Butterfly operation) are repeated for the FFTL (256/512) samples of the input buffer.

The above operation (after bit reversal) is repeated in every stage of the algorithm. The algorithm needs $\log_2 N$ stages to finish. In the last stage, the magnitude of the result of the Butterfly is calculated, and stored in the *I RAM*, while the *Q RAM* is free.

CORDIC algorithm is also used to compute the magnitude of the last stage of the algorithm, for more details about the usage of CORDIC in magnitude calculation refer to Appendix A. Note that; the magnitude calculation in the last stage is done in parallel with the butterfly operations of the last stage, which enhances the time performance.

The read/ write signals and addresses are controlled by the Butterfly in all the stages except the last one. In the last stage only, these signals are under control of the Magnitude Calculator. It is the role of the RAM Manager to control the access to these memories.

Note that; Bit reversal storage is actually performed in the context of Buffer Manager module before starting the FFT Butter fly.

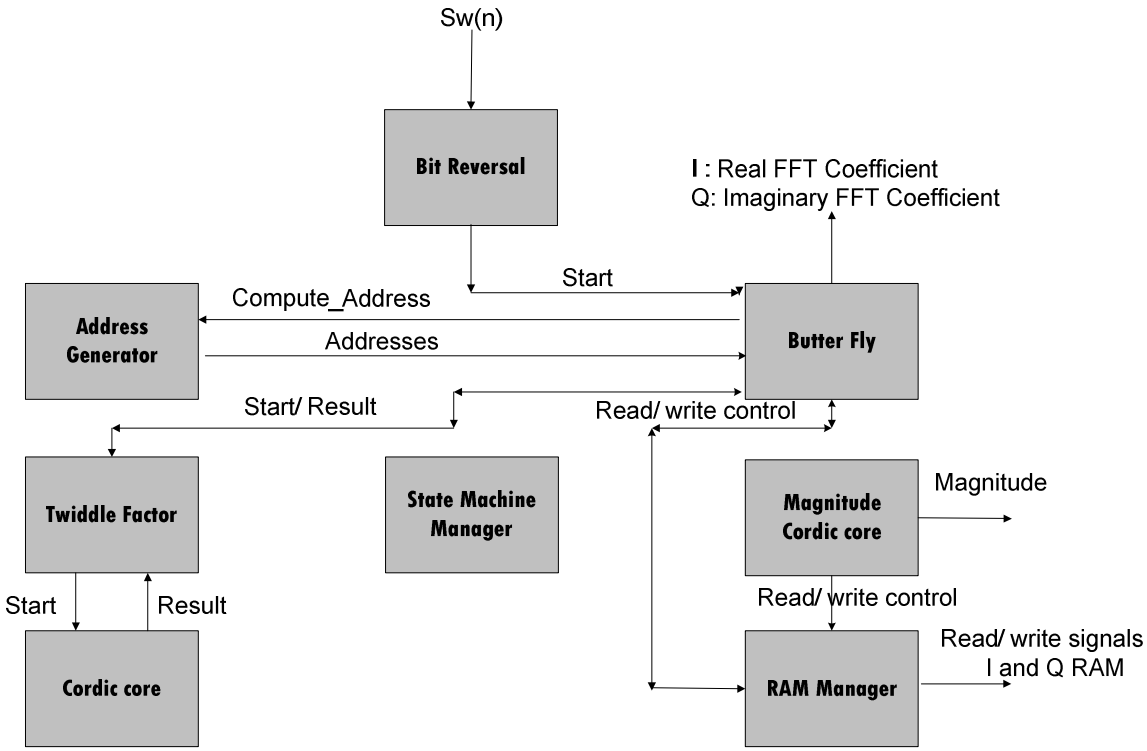


Figure 53: FFT Internal architecture

Note that: the Butterfly operation requires read/ write operation to two entries in the *I* and *Q RAMs*, so, the corresponding RAMs are dual-port to read or write in two locations simultaneously in one clock cycle, which reduces the required time for performing the butterfly by 50%.

4.2.3.6.2.1 Look-up table implementation

The implementation described in 4.2.3.6.1 relies on calculating the twiddle factors using a CORDIC core, another implementation can be done by storing the cosine values in a LUT, and use them to deduce the sine values. In addition, only ¼ of the cosine wave need to be stored, and the rest of the wave can be obtained from this first quadrant. Note that; the LUT is dual-port, such that, reading the factors for the whole butterfly operation (two multiplications) is performed in single cycle.

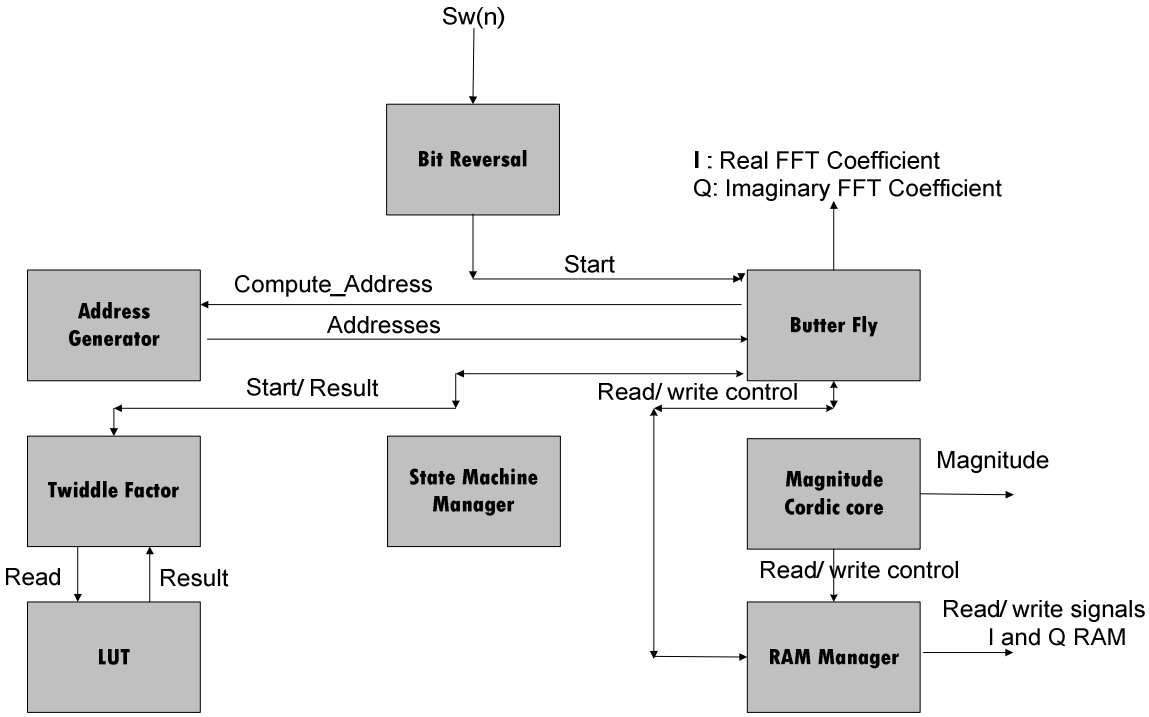


Figure 54: FFT Internal architecture for LUT implementation

4.2.3.6.3 Configuration

Parameter	Possible values	Default value	Description
depth	256/ 512 for FFTL = 256/512	256	The frame length in samples
Dwidth	16	16	The data width of the I and Q data.
Iwidth	8	8	The integer part of the data
Fwidth	7	7	The fraction part of the data
Awidth	8/9 bits for FFTL = 256/512	8	The address width of the I,Q memories. The 2N RAM address width is 1 bit more than that width.

Table 19: Memory requirements of the FFT component

4.2.3.6.4 Signal width justification

The data width is 16 bits. The Integer part width is 8 for FFTL = 256 and 9 for FFTL = 512. The Fraction part width is 7 for FFTL = 256 and 6 for FFTL = 512. This choice was based on run time results of real test vectors to obtain the dynamic range of the signals, so that overflow or underflow is completely avoided in any stage of the calculation. This analysis was done at the algorithm level, using high level code of the algorithm, were the fixed point behavior was tested to obtain the right signal widths.

Note that; limiting the signal widths to 16 bits optimizes the number of multipliers required for the butterfly operation, since the available embedded multipliers on the FPGA are 18x18 multipliers, which makes t possible to use the on-chip multipliers instead of implementing them.

4.2.3.6.5 State Machines

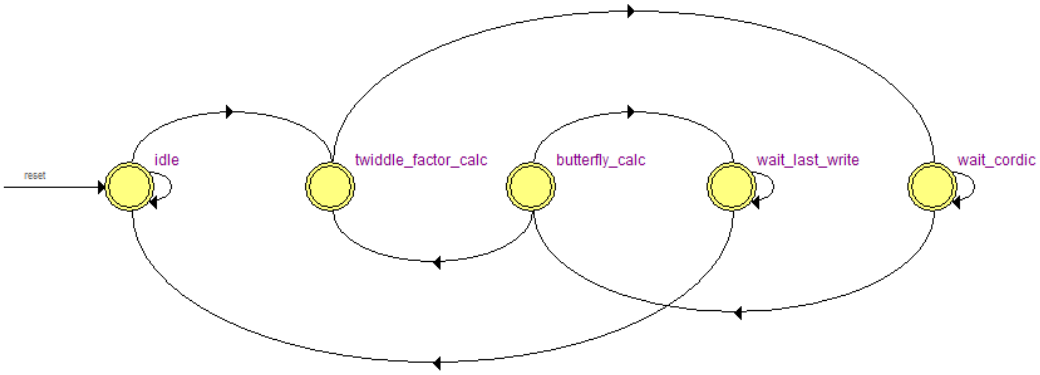


Figure 55: State machine of the FFT component

The description of these states is as follows:

- *IDLE:* the module remains in this state until a start signal is triggered.
- *TWIDDLE_FACTOR_CALC:* this is a transient state to calculate the angle of rotation of the current vector to be used as an input to the CORDIC module.
- *WAIT_CORDIC:* the module remains in this state until the CODIC finishes the vector rotation operation.
- *BUTTERFLY_CALC:* in this state the final Butterfly calculation is done, and the result is written back in the *I* and *Q RAM*s, except for the

last stage, where the magnitude of the real and imaginary parts of the result is written back to the *I RAM*. If this is not the last magnitude calculation of the algorithm, the next state will be TWIDDLE_FACTOR_CALC, otherwise it will be WAIT_LAST_WRITE.

- *WAIT_LAST_WRITE:* during the last stage of the algorithm, the magnitude of the result of the Butterfly is calculated in parallel with the next Butterfly operation. However, for the last magnitude calculation, there is no next Butterfly operation; so, we should wait till the magnitude is calculated to announce the end of the whole algorithm and to write the last magnitude result.

4.2.3.6.6 Memory requirements

The memory required for *I RAM* and *Q RAM* is kept in the Buffer Manager component, as described in section 4.2.3.5.6, so no memory is required for the FFT component itself.

In case of *Look-up table* (LUT) implementation mentioned in 4.2.3.6.2.1, an extra ROM of length FFTL/4 is needed.

4.2.3.6.7 Actual Chip Usage

The following is a summary of the chip usage as generated by the Quartus II software based on CORDIC implementation:

Flow Status	*Successful - Thu Oct 02 10:59:23 2008*
Quartus II Version	*7.2 Build 203 02/05/2008 SP 2 SJ Web Edition*
Revision Name	*source_tb*
Top-level Entity Name	*FFT_6*
Family	*Cyclone III*
Device	*EP3C40F780C8*
Total logic elements	*2,314 / 39,600 (6 %)*
Total combinational functions	*2,313 / 39,600 (6 %)*
Dedicated logic registers	*271 / 39,600 (< 1 %)*
Total registers	*271*
Total pins	*169 / 536 (32 %)*
Total virtual pins	*0*
Total memory bits	*0 / 1,161,216 (0 %)*
Embedded Multiplier 9-bit elements	*12 / 252 (5 %)*

Figure 56: Summary of resources usage of FFT module: FFTL = 256

Flow Status	*Successful - Thu Oct 02 11:27:18 2008*
Quartus II Version	*7.2 Build 203 02/05/2008 SP 2 SJ Web Edition*
Revision Name	*source_tb*
Top-level Entity Name	*FFT_6*
Family	*Cyclone III*
Device	*EP3C40F780C8*
Total logic elements	*2,252 / 39,600 (6 %)*
Total combinational functions	*2,248 / 39,600 (6 %)*
Dedicated logic registers	*271 / 39,600 (< 1 %)*
Total registers	*271*
Total pins	*173 / 536 (32 %)*
Total virtual pins	*0*
Total memory bits	*0 / 1,161,216 (0 %)*
Embedded Multiplier 9-bit elements	*12 / 252 (5 %)*

Figure 57: Summary of resources usage of FFT module: FFTL = 512

4.2.3.6.8 Processing time

Let:

- Number of clocks taken by CORDIC calculator circuit = C = 12.
- Number of clocks taken for Magnitude calculator = L = 11.
  Magnitude Calculation is performed in the last stage only, in parallel with the Butterfly operation, so the last calculation only should be added to the total required time.
- The number of times that the Twiddle factor is changed is dependent on the current stage of the algorithm, where we have $\log_2$FFTL stages. Let the stage order be i, where $1 \leq i \leq \log_2 FFTL-1$, then the number of times the Twiddle factor changes during this stage is 2^i.
  Hence;

$$\textit{The total number of times the Twiddle factor changes} = \sum_{i=1}^{i=\log_2 FFTL-1} 2^i$$

Every time the Twiddle factor changes, this requires a new CORDIC operation, so;

$$\textit{The number of clocks for CORDIC operations} = C \times \sum_{i=1}^{i=\log_2 FFTL-1} 2^i$$

The above calculation should be added to the total number of clocks required by the algorithm which is FFTL /2 * $\log_2$ FFTL.

Therefore:

$$Processing\ time = FFTL/2 \times \log_2 FFTL + C \times \sum_{i=1}^{i=\log_2 FFTL-1} 2^i + L =$$

$$= 4095\ for\ FFTL = 256$$

$$= 8447\ for\ FFTL = 512.$$

In case of LUT implementation, the processing time will be:

$$Processing\ time = FFTL/2 \times L + FFTL =$$

$$= 1664\ for\ FFTL = 256$$

$$= 3328\ for\ FFTL = 512.$$

4.2.3.7 Mel-Filter

4.2.3.7.1 Basic functionality

This module is responsible of calculating the 23 Mel coefficients. The centre frequencies of the channels in terms of FFT bin indices (cbini for the ith channel) are calculated as follows:

$$Mel\{x\} = 2595 \times \log_{10}\left(1 + \frac{x}{700}\right),$$

$$f_{c_i} = Mel^{-1}\left\{Mel\{f_{start}\} + \frac{Mel\{f_s/2\} - Mel\{f_{start}\}}{23+1} i\right\}, \quad i = 1,...,23,$$

$$cbin_i = round\left\{\frac{f_{c_i}}{f_s} FFTL\right\},$$

$$cbin_0 = round\left\{\frac{f_{start}}{f_s} FFTL\right\},$$

$$cbin_{24} = round\left\{\frac{f_s/2}{f_s} FFTL\right\} = FFTL/2.$$

The 25 cbin coefficients are pre-computed and stored. Those stored values are dependent on the FFTL configuration.

The output of the mel filter is the weighted sum of the FFT magnitude spectrum values (bini) in each band. Triangular, half-overlapped windowing is used as follows:

$$fbank_k = \sum_{i=cbin_{k-1}}^{cbin_k} \frac{i - cbin_{k-1} + 1}{cbin_k - cbin_{k-1} + 1} bin_i + \sum_{i=cbin_k+1}^{cbin_{k+1}} \left(1 - \frac{i - cbin_k}{cbin_{k+1} - cbin_k + 1}\right) bin_i$$

Where k = 1... 23.

Following to this step, the Non-linear transformation is calculated:

$$f_i = \ln(fbank_i), i = 1,\ldots,23$$

4.2.3.7.2 Internal Architecture

This section shows the data flow graph of the module. This graph is just for design purpose and does not mean that the final synthesized hardware on the chip will look like that, yet it should be very near to it. Define:

LOW_PART_CONST(k) = cbin(k) – cbin(k-1) + 1

HIGH_PART_CONST(k) = cbin(k+1) – cbin(k) + 1

The data flow graph of calculating these constants is shown in Figure 58:

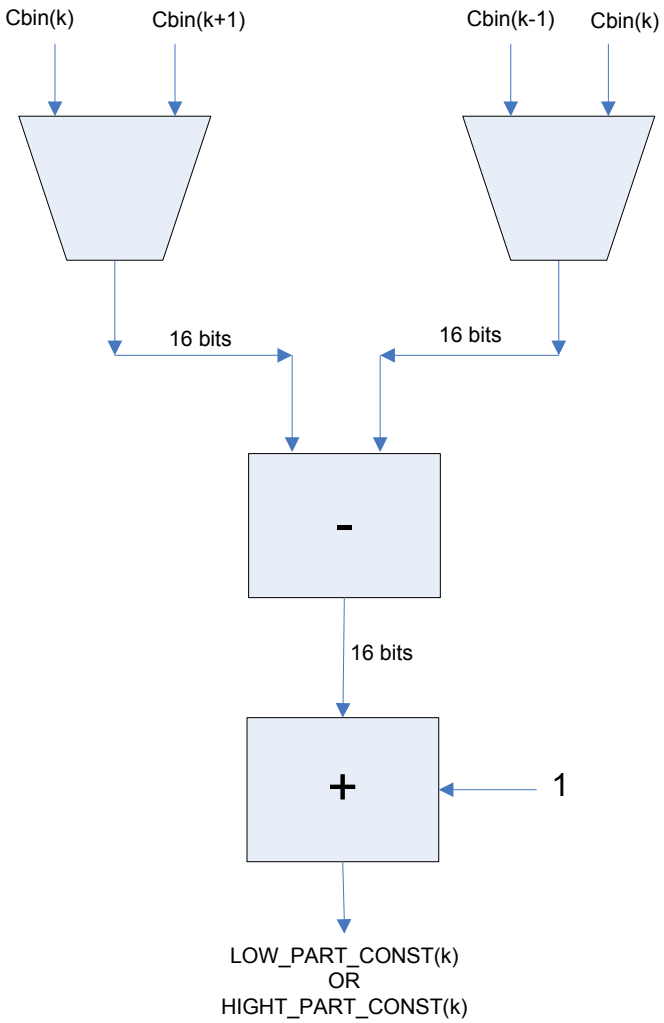


Figure 58: Data flow graph of calculating the LOW_PART_CONST and the HIGH_PART_CONST

The above circuit is used to calculate both constants according to a configuration parameter, which act as the select of the MUX's shown in the figure. These constants are then used to calculate the following:

$$CL(i) = (i+1)/(LOW\_PART\_CONST(k))$$

$$CH(i) = (HIGHT\_PART\_CONST(k) - i)/(HIGHT\_PART\_CONST(k))$$

Where i = cbin(k-1)…cbin(k), for the low frequency part calculation and

= cbin(k) + 1…cbin(k+1), for the high frequency part calculation.

Figure 59 shows the data flow graph of the above equation.

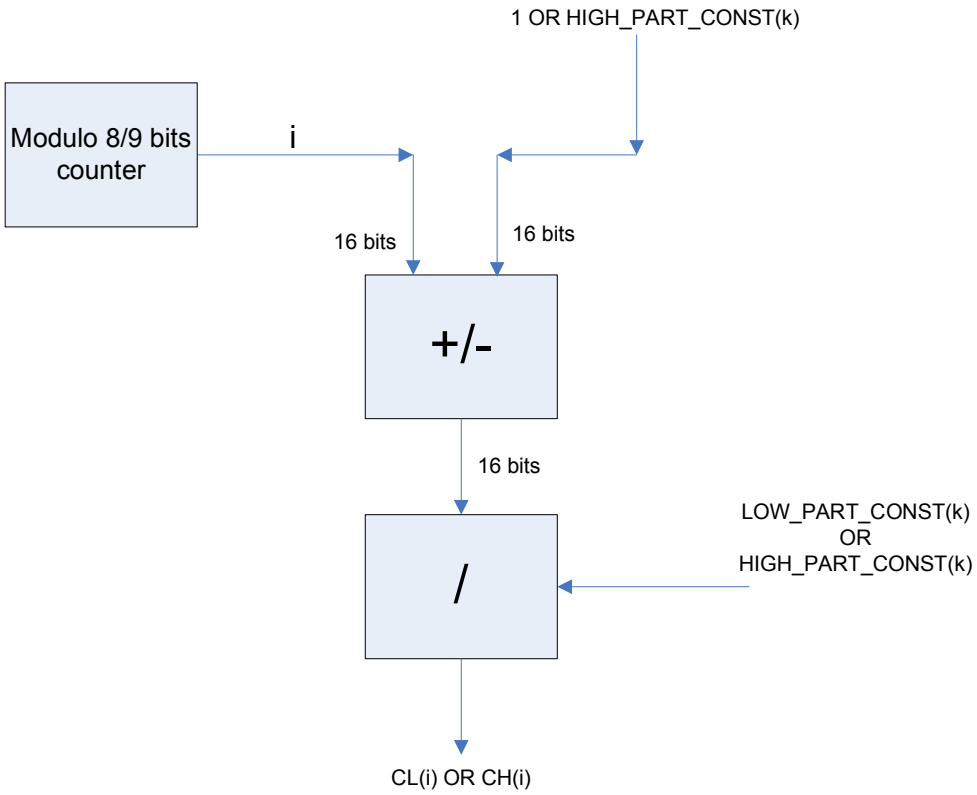


Figure 59: Data flow graph of CL(i) and CH(i)

The division operation is done using a CORDIC processor (please refer to Appendix A). The division operation takes 11 clock cycles.

The final step to calculate fbank(k) is to multiply the above calculated constants by the corresponding bin(i) coefficients of the FFT, and accumulate the result, for i in the range

$cbin(k-1) \leq i \leq cbin(k)$, for the low frequency calculation, then

$cbin(k) + 1 \leq i \leq cbin(k+1)$, for the high frequency calculation.

First, fbank_low(k) is calculated, then fbank_high(k) is calculated next, and finally the two results are added to get fbank(k). The calculation of fbank_low(k) or fbank_high(k) is done in a *core* circuit that can be configured to calculate either of them. The data flow graph of this calculation is shown in Figure 60:

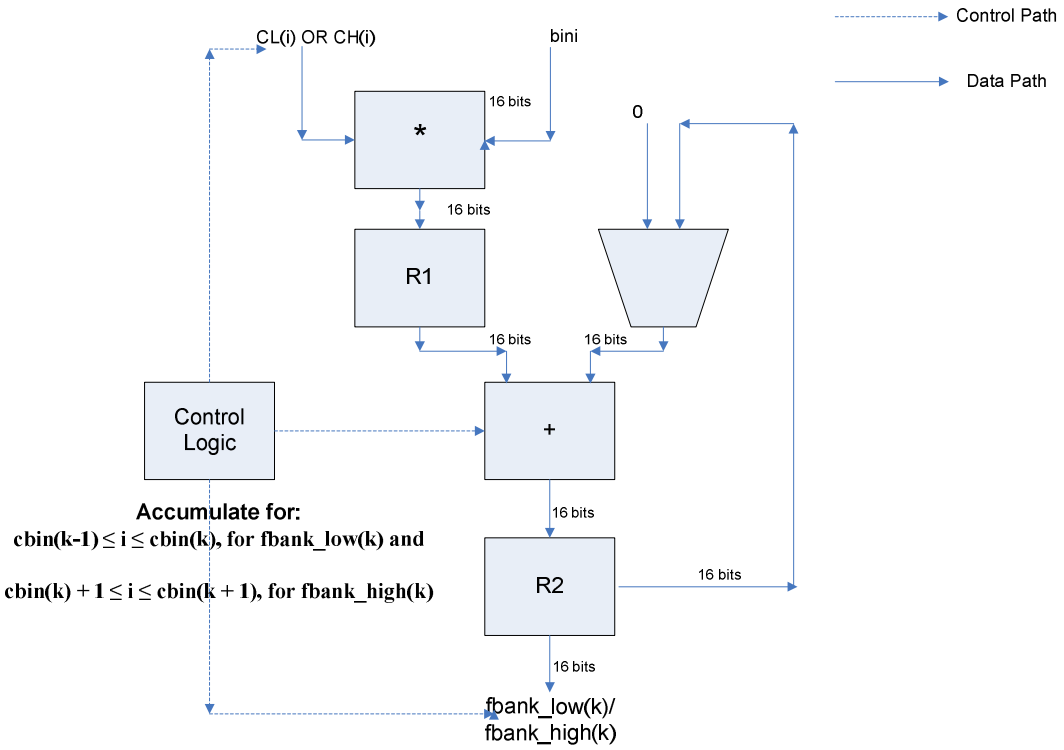


Figure 60: Data flow graph of Mel-Filter

The data flow graph in Figure 60 is to calculate one fbank(k) coefficient. The whole operation is repeated 23 times to get fbank(1) to fbank(23).

Every time a coefficient fbank(k) is generated, its natural logarithm is calculated. The natural logarithm is calculated using a CORDIC processor, please refer to Appendix A. The calculation of the natural logarithm is done in parallel with calculating the next fbank(k). The natural logarithm calculation takes 16 clock cycles.

4.2.3.7.3 Configuration

Parameter	Possible values	Default value	Description
N_MEL	23	23	Number of mel-filter coefficients
Dwidth	16	16	The data width of

Parameter	Possible values	Default value	Description
			the I and Q data.
Iwidth	8 for FFTL = 256 9 for FFTL = 512	8	The integer part of the input data
Fwidth	7 for FFTL = 256 6 for FFTL = 512	7	The fraction part of the input data
Awidth	8 for FFTL = 256 9 for FFTL = 512	8	The address width of the I,Q memories.
Shift	3 for FFTL = 256 and 512	3	The Iwidth and Fwidth of the input data is different than those of the internal signals, so the Shift parameter defines the number of bits needed to adapt the input to the internal signals fixed point widths. This parameter is used to shift the input signal right with this number of bits, so

Parameter	Possible values	Default value	Description
			that the Iwidth of the internal signals (Iwidth_internal) is Iwidth of the input + Shift
Shift_output	16 for FFTL = 256 15 for FFTL = 512	16	The accumulated signal Fwidth is made wider to obtain more accurate result, and at the end, the result need to be fixed to the external world width, so this parameter defines the number of bits to fix from the Fwidth part of the accumulated signal before connecting it to the external fbank(k) output. So, the Fwidth of the internal signals will be Fwidth of the input + (Shift_output - Shift)

Table 20: Memory requirements of the Mel-Filter module

The values of cbin(k) are stored in ROM based on the FFTL configuration.

4.2.3.7.4 Signal width justification

The input and output data widths are Dwidth (=16) bits. The internal signal widths are Dwidth + Shift_output = 32 bits.

The integer part width Iwidth of the internal signals is increased by the Shift parameter to accommodate the accumulated fbank(k) to be 8 bits in case of FFTL = 256, and 9 bits in case of FFTL = 512. The final fbank(k) after taking the natural logarithm has a dynamic range that needs only 3 bits for integer part, hence the Iwidth of the final fbank(k) is fixed to only 3 bits after the natural logarithm is calculated using the CORDIC module.

Also, the fraction part width Fwidth of the internal signals is increased by Shift_output – Shift, to be 23 in case of FFT L = 256 and 22 in case of FFTL = 512. This is to increase the accuracy of the accumulated signal before it is fixed in both cases to Fwidth = 12 when connected to the final fbank(k) after the natural logarithm is taken using the CORDIC module.

The above choice of signal widths was based on run time results of real test vectors to obtain the dynamic range of the signals, so that overflow or underflow is completely avoided in any stage of the calculation. This analysis was done at the algorithm level, using high level code of the algorithm, were the fixed point behavior was tested to obtain the right signal widths.

4.2.3.7.5 State Machines

4.2.3.7.5.1 Mel-Filter

This part defines the general state machine that controls the low and high part sub-filters. The low part sub-filter is applied first, then the high part.

After the low and high part are calculated the results are added to get fbank(k) as follows:

$$fbank_k = \sum_{i=cbin_{k-1}}^{cbin_k} \frac{i - cbin_{k-1} + 1}{cbin_k - cbin_{k-1} + 1} bin_i + \sum_{i=cbin_k+1}^{cbin_{k+1}} \left(1 - \frac{i - cbin_k}{cbin_{k+1} - cbin_k + 1}\right) bin_i$$

The outer state machine defined here controls the operation of the *core* sub-filter that calculates the low and high parts filtering. The following is the definition of the states:

- *IDLE*: the module remains in this state till a start signal comes, and then it goes to *ACTIVATE_MEL_LOW* state.
- *ACTIVATE_MEL*: this is a transitional state, the module remains in it for one clock to activate the low part sub-filter, and then it goes to the state *WAIT_MEL_LOW*.
- *WAIT_MEL_LOW*: in this state, the module waits the low part sub-filter or finish operation, then it goes to *ACTIVATE_MEL_HIGH* state.
- *ACTIVATE_MEL_HIGH*: this is a transitional state, the module remains in it for one clock to activate the high part sub-filter, and then it goes to the state *WAIT_MEL_HIGH*.
- *WAIT_MEL_HIGH*: in this state, the module waits the high part sub-filter or finish operation, then it goes to *INCREMENT_COUNTER* state.
- *INCREMENT_COUNTER*: if all Mel coefficients (N_MEL = 23) were calculated, the module goes to *WAIT_LOG_CORDIC*, otherwise it goes to *ACTIVATE_MEL_LOW* state to start calculating the next Mel coefficient.
- *WAIT_LOG_CORDIC:* in this state the module waits for the natural logarithm of the last coefficient to be calculated by the CORDIC module to announce that all coefficients have been calculated. This waiting is done only in the last coefficient, since the natural logarithm of the rest of the coefficients was calculated in parallel with the calculation of the next one, however for the last one, there no next calculation, so we should for the CORDIC module to finish calculating the natural logarithm of the last coefficient.
- *MEL_FINISHED_ALL*: reaching this state means that all 23 Mel coefficients were calculated.

4.2.3.7.5.2 Low/ High Frequency part

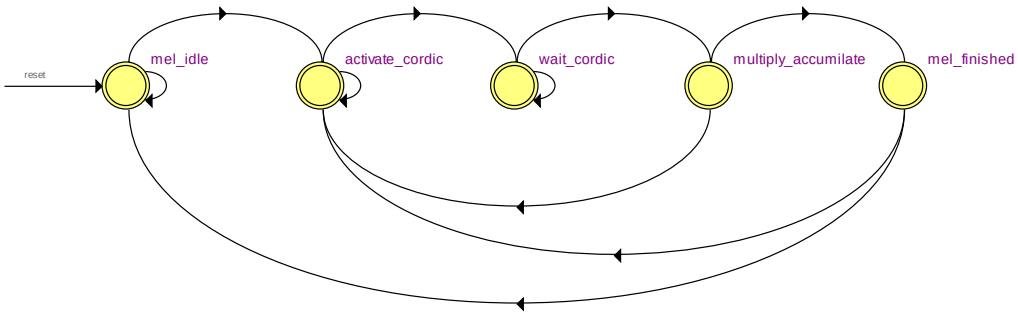


Figure 61: State machine of the Low/ High part Mel-Filter

This is the state machine of the *core* subfilter. This sub-filter is responsible of calculating the multiply-accumilate operation of the low or high parts of the filter. The low part operation is as follows:

$$\sum_{i=cbin_{k-1}}^{cbin_k} \frac{i - cbin_{k-1} + 1}{cbin_k - cbin_{k-1} + 1} bin_i$$

And the high part operation is:

$$\sum_{i=cbin_k+1}^{cbin_{k+1}} \left(1 - \frac{i - cbin_k}{cbin_{k+1} - cbin_k + 1}\right) bin_i$$

The module start in *MEL_IDLE* state. With the start signal it goes to the *ACTIVATE_CORDIC* state, in which the CORDIC divider is activated, then it goes to the *WAIT_CORDIC* state, till the CORDIC finishes. When the CORDIC finishes, the module goes to *MULTIPLY_ACCUMILATE* state, where the bini coefficient is multiplied by the result of the CORDIC divider and accumilated. If the accumilation counter reached its limit, the module goes to *MEL_FINISHED*, otherwise it goes to *ACTIVATE_CODIC* state.

4.2.3.7.6 Memory requirements

The input values of bin(i) of the magnitude of the final FFT coefficients are stored in the *I RAM* managed by the Buffer Manager. The resulting Mel-

Filtered coefficients are stored in the *Q RAM*. These memories are managed in the Buffer Manager component.

The cbin(k) coefficients are stored in ROM.

Memory	Size	Description
cbink_rom	25 * 16	Pre-computed center frequencies of the bands of the Mel-Filter. These are stored in ROM.

Table 21: Memory requirements of the Mel-Filter component

4.2.3.7.7 Actual Chip Usage

The following is a summary of the chip usage as generated by the Quartus II software:

Flow Status Successful - Fri Oct 03 13:53:51 2008
Quartus II Version 7.2 Build 203 02/05/2008 SP 2 SJ Web Edition
Revision Name source_tb
Top-level Entity NameMel_Filter
Family Cyclone III
Device EP3C10U256C8
Timing Models Preliminary
Total logic elements 2,076 / 10,320 (20 %)
Total combinational functions 2,043 / 10,320 (20 %)
Dedicated logic registers 275 / 10,320 (3 %)
Total registers 275
Total pins 54 / 183 (30 %)
Total virtual pins 0
Total memory bits 0 / 423,936 (0 %)
Embedded Multiplier 9-bit elements 8 / 46 (17 %)

Figure 62: Summary of resources usage of Mel-Filter module: FFTL = 256

Flow Status Successful - Fri Oct 03 13:54:45 2008
Quartus II Version 7.2 Build 203 02/05/2008 SP 2 SJ Web Edition
Revision Name source_tb
Top-level Entity NameMel_Filter
Family Cyclone III
Device EP3C10U256I7
Timing Models Preliminary
Total logic elements 2,157 / 10,320 (21 %)
Total combinational functions 2,116 / 10,320 (21 %)
Dedicated logic registers 269 / 10,320 (3 %)
Total registers 269
Total pins 56 / 183 (31 %)

Total virtual pins 0 *Total memory bits 0 / 423,936 (0 %)* *Embedded Multiplier 9-bit elements 8 / 46 (17 %)*

Figure 63: Summary of resources usage of Mel-Filter module: FFTL = 512

4.2.3.7.8 Processing time

Let:

- Number of clocks taken by the divider = D = 11.
- Number of clocks taken by natural logarithm CORDIC = NL = 16
- Largest difference between any two center frequencies cbin(k-1) and cbin(k+1) = K =
    - 21 in case of sampling frequency 8 kHz ,and
    - 23 in case of 11 kHz, and
    - 51 in case of 16 kHz.

Therefore:

$$Processing\ time = N\_MEL \times K \times D + NL$$

- 5560 clocks for sampling frequency = 8 kHz
- 6088 clocks for sampling frequency = 11 kHz
- 13480 clocks for sampling frequency = 16 kHz

4.2.3.8 DCT

4.2.3.8.1 Basic functionality

13 cepstral coefficients are calculated from the output of the Non-linear Transformation block.

$$C_i = \sum_{j=1}^{23} f_j \times \cos\left(\frac{\pi \times i}{23}(j-0{,}5)\right), \qquad 0 \le i \le 12$$

4.2.3.8.2 Internal Architecture

This section shows the data flow graph of the module. This graph is just for design purpose and does not mean that the final synthesized hardware on the chip will look like that, yet it should be very near to it.

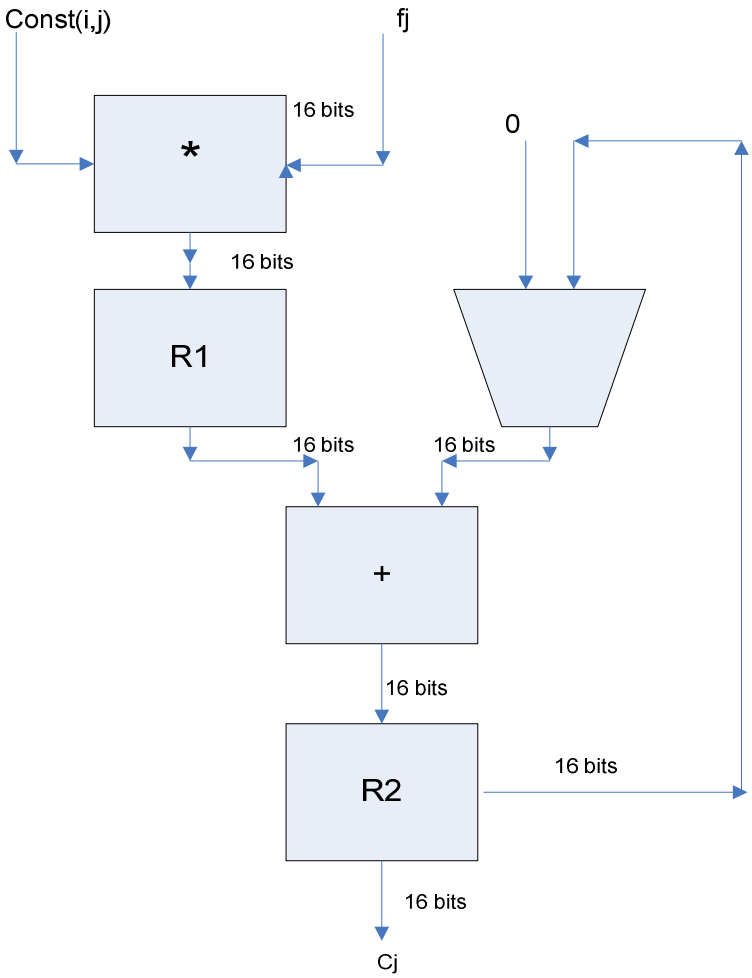


Figure 64: Data flow graph of DCT component

Const(i,j) is as shown below:

$$\cos\left(\frac{\pi \times i}{23}(j-0{,}5)\right), \qquad 0 \le i \le 12$$

This can be calculated with the CORDIC Core like the one described in Appendix A.

The data flow graph in Figure 64 is to calculate one Ci coefficient. The whole operation is repeated 13 times to get C0 to C12.

4.2.3.8.3 Configuration

Parameter	Possible values	Default values	Description
N_MEL_COEFF	23	23	Number of mel-filter coefficients

Parameter	Possible values	Default values	Description
N_CEPSTRAL	13	13	Number of cepstral coefficients
Dwidth	16	16	The data width of the I and Q data.
Iwidth	7	7	The integer part of the data
Fwidth	8	8	The fraction part of the data
Awidth	8 bits for FFTL = 256 9 bits for FFTL = 512	8	The address width of the I,Q memories. The 2N RAM address width is 1 bit more than that width.
Shift_sum	4 bits	4 bits	The internal accumulated signal is made wider than the input and output signals bu Shift_sum bits to increase the accuracy of accumulation. The final result is eventually fixed to the Dwidth of the input signal.

Table 22: Memory requirements of the DCT component

4.2.3.8.3.1 Look-up table implementation

The argument of the cosine factor in the basic DCT equation has a resolution of 0.5, which means that, 46 values are required to be stored to represent the whole cosine wave. Having those factors stored in a LUT, there is no need for the CORDIC core, which reduces the processing time. On the other hand, extra ROM of 24 entries is needed.

4.2.3.8.4 Signal width justification

The input and output data signals widths are 16 bits. However, the accumulator signal used internally is wider by Shift_sum bits to increase the accuracy of accumulation, then the final result is fixed again to Dwidth.

The above choice of signal widths was based on run time results of real test vectors to obtain the dynamic range of the signals, so that overflow or underflow is completely avoided in any stage of the calculation. This analysis was done at the algorithm level, using high level code of the algorithm, were the fixed point behavior was tested to obtain the right signal widths.

4.2.3.8.5 State Machines

The module is divided into two parts; the first part (*inner state machine*) is responsible of calculating the inner multiply accumulate operation to calculate every Cepstal coefficient. The multiply accumulate operation of this state machine is as shown:

$$\sum_{j=1}^{23} f_j \times \cos\left(\frac{\pi \times i}{23}(j-0{,}5)\right)$$

The second part (*outer state machine*) is responsible of managing the overall state machine of the DCT, where it controls the trigger of the inner state machine to calculate next coefficient, until all 13 coefficients are calculated.

The definition of the states of the *outer state machine* is as follows:

- *IDLE* : if start signal is raised, the DCT operation is triggered, and the module goes to the state *ACTIVATE_DCT* to start calculating the cepstral coefficients.
- *ACTIVATE_DCT* : in this state the *inner state machine* is activated to calculate the next cepstral coefficient. The module then goes to the *WAIT_DCT* state.
- *DCT_ACTIVATED:* this is a transient state to reset internal signals and activate the *inner state machine*. The module unconditionally goes to *WAIT_DCT* state.
- *WAIT_DCT* : the module remains in this state until the *inner state machine* finishes calculating the current cepstral coefficients.
- *INCREMENT_COUNTER* : in this state a counter is incremented, if it reached 13, which means that all coefficients were calculated, the module goes to *DCT_FINISHED_ALL*, otherwise it goes to the *ACTIVATE_DCT*.
- *DCT_FINISHED_ALL:* the finished signal is generated in this state indicating the end of DCT filtering.

The definition of the states of the *inner state machine* is as follows:

- *DCT_IDLE*: the system remains in this state until the *inner state machine* is activated, then the system goes to the *ACTIVATE_CORDIC* state.
- *ACTIVATE_CORDIC*: the CORDIC processor that calculates the following constant is activated in this state, and then the system goes to the *WAIT_CORDIC* state:

$$\cos\left(\frac{\pi \times i}{23}(j-0{,}5)\right)$$

- *WAIT_CORDIC*: the system remains in this state till the CORDIC finishes, then it goes to the *MULTIPLY_ACCUMILATE* state
- *MULTIPLY_ACCUMILATE*: in this state the following multiplication is performed and the result is accumulated.

$$f_j \times \cos\left(\frac{\pi \times i}{23}(j-0{,}5)\right)$$

A counter is incremented till it reaches 23, then the system goes to the *DCT_FINISHED* state, otherwise it goes to the *ACTIVATE_CORDIC* again.

- *DCT_FINISHED:* reaching this state means that the Ci DCT coefficient was successfully calculated.

4.2.3.8.6 Memory requirements

The memory required for *I RAM* and *Q RAM* is kept in the Buffer Manager component, as described in section 4.2.3.5.6, so no memory is required for the DCT component itself. In case of LUT implementation, extra ROM is required to store the 24 DCT factors.

4.2.3.8.7 Actual Chip Usage

The following is a summary of the chip usage as generated by the Quartus II software based on CORDIC implementation:

Flow Status	*Successful - Fri Oct 03 14:29:35 2008*
Quartus II Version	*7.2 Build 203 02/05/2008 SP 2 SJ Web Edition*
Revision Name	*source_tb*
Top-level Entity Name	*DCT_1*
Family	*Cyclone III*
Device	*EP3C10U256C8*
Timing Models	*Preliminary*
Met timing requirements	*N/A*
Total logic elements	*1,153 / 10,320 (11 %)*
Total combinational functions	*1,153 / 10,320 (11 %)*
Dedicated logic registers	*87 / 10,320 (< 1 %)*
Total registers	*87*
Total pins	*46 / 183 (25 %)*
Total virtual pins	*0*
Total memory bits	*0 / 423,936 (0 %)*
Embedded Multiplier 9-bit elements	*2 / 46 (4 %)*
Total PLLs	*0 / 2 (0 %)*

Figure 65: Summary of resources usage of DCT module

4.2.3.8.8 Processing time

Let the number of clocks taken by cosine calculator circuit = C = 12. Therefore:

*Processing time = C * N_MEL * N_CEPSTRAL = 3744.*

In case of LUT implementation:

*Processing time = N_MEL * N_CEPSTRAL = 299*

4.2.3.9 Split-Vector Quantization

4.2.3.9.1 Basic functionality

The feature vector y(m) is directly quantized with a split vector quantizer. Coefficients are grouped into pairs, and each pair is quantized using its own VQ codebook. The resulting set of index values is then used to represent the speech frame. Coefficient pairings (by front-end parameter) are shown in table 5.1, along with the codebook size used for each pair.

	Size	Weight Matrix		
Codebook	$(N^{l,l+1})$	$(W^{l,l+1})$	Element 1	Element 2
$Q^{0,1}$	64	I	c_1	c_2
$Q^{2,3}$	64	I	c_3	c_4
$Q^{4,5}$	64	I	c_5	c_6
$Q^{6,7}$	64	I	c_7	c_8
$Q^{8,9}$	64	I	c_9	c_{10}
$Q^{10,11}$	64	I	c_{11}	c_{12}
$Q^{12,13}$	256	Non-identity	c_0	log[E]

Figure 66: Split Vector Quantization Features Pairings [2]

Two sets of VQ codebooks are defined; one is used for speech sampled at 8 kHz or 11 kHz while the other for speech sampled at 16 kHz. The weights used (to one decimal place of numeric accuracy) are:

8 kHz or 11 kHz sampling rate $W^{12,13} = \begin{bmatrix} 1446,0 & 0 \\ 0 & 14,7 \end{bmatrix}$

16 kHz sampling rate $W^{12,13} = \begin{bmatrix} 1248,9 & 0 \\ 0 & 12,7 \end{bmatrix}$

The closest VQ centroid is found using a weighted Euclidean distance to determine the index:

$$d_j^{i,i+1} = \begin{bmatrix} y_i(m) \\ y_{i+1}(m) \end{bmatrix} - q_j^{i,i+1}$$

$$idx^{i,i+1}(m) = \underset{0 \le j \le (N^{i,i+1}-1)}{\operatorname{argmin}} \left\{ \left(d_j^{i,i+1}\right) W^{i,i+1} \left(d_j^{i,i+1}\right) \right\}, \quad i = 0,2,4 \ldots 12$$

Where $q_j^{i,i+1}$ denotes the jth code vector in the codebook $Q^{i,i+1}$, $N^{i,i+1}$ is the size of the codebook, $W^{i,i+1}$ is the (possibly identity) weight matrix to be applied for the codebook $Q^{i,i+1}$, and $idx^{i,i+1}(m)$ denotes the codebook index chosen to represent the vector $[y_i(m), y_{i+1}(m)]^T$. The indices are then retained for transmission to the back-end.

4.2.3.9.2 Internal Architecture

The code books centroids are calculated and stored in the ROM, according to the configuration of the chip, where the there are different tables for 8 kHz, 11 kHz and 16 kHz. These values are stored in 7 tables., hence we need 7 ROMs to store the tables. Every pair of input features (ci, ci+1) are quantized using the proper quantization table. The distance between the input vector and each entry in the proper table is calculated as follows:

$$(Dist)^2 = (Ci - Q(i,j))^2 + (Ci+1 - Q(i+1,j))^2$$

Then the index (j) of min(Dist ^{2}) is chosen, and put in the output frame as a 6/ 8 bits value. Note that: the operation of this module is triggered when every new feature (LogE, C0… C13) becomes ready, so, quantization is done in parallel with the DCT module. Every time two features are ready, they are input to the Vector Quantizer module to be quantized. According to the current features being quantized, the access is given to the core Quantizer to the proper ROM that contains the proper quantization table.

4.2.3.9.3 Configuration

Parameter	Possible values	Default values	Description
Awidth	8 bits for FFTL = 256 9 bits for FFTL = 512	8	The address width of the I,Q memories.
Dwidth	16	16	The data width of the features
Iwidth	7	7	The integer width
Fwidth	8	8	The fraction part
Shift_energy	4	4	The Iwidth of the LogE feature is 3 bits, while the Iwidth of the internal signals is 7 bits, so the LogE feature needs to be fixed by shifting it right by 4 bits.

Table 23: Memory requirements of the Vector Quantization component

4.2.3.9.4 Signal width justification

The width of the input features is 16 bits, with the same width as the output of the previous module. This choice of signal widths was based on run time results of real test vectors to obtain the dynamic range of the signals, so that overflow or underflow is completely avoided in any stage of the calculation. This analysis was done at the algorithm level, using high level code of the algorithm, were the fixed point behavior was tested to obtain the right signal widths.

4.2.3.9.5 State Machines

The module starts in the IDLE state, until the start signal comes (which indicates that the LogE feature is ready), so the module goes to the wait_C0 state. Every time a store signal is triggered the module goes to the next state. When a feature pair is ready (like LogE-C0, C1-C2, C3-C4,…C11-C12), the core Quantizer is activated, and the access of the ROM is given to the proper quantization table ROM.

4.2.3.9.6 Memory requirements

Memory	Size	Description
Q_0	64 * 16	Quantization table of feature C0
Q_1	64 * 16	Quantization table of feature C1
Q_2	64 * 16	Quantization table of feature C2
Q_3	64 * 16	Quantization table of feature C3
Q_4	64 * 16	Quantization table of feature C4
Q_5	64 * 16	Quantization table of feature C5
Q_6	64 * 16	Quantization table of feature C6
Q_7	64 * 16	Quantization table of feature C7
Q_8	64 * 16	Quantization table of feature C8
Q_9	64 * 16	Quantization table of feature C9
Q_10	64 * 16	Quantization table of

Memory	Size	Description
		feature C10
Q_11	64 * 16	Quantization table of feature C11
Q_12	256 * 16	Quantization table of feature C12
Q_13	256 * 16	Quantization table of feature LogE

Table 24: Memory requirements of the Split-Vector Quantization module

4.2.3.9.7 Actual Chip Usage

The following is a summary of the chip usage as generated by the Quartus II software:

Flow Status Successful - Fri Oct 03 15:53:55 2008
Quartus II Version 7.2 Build 203 02/05/2008 SP 2 SJ Web Edition
Revision Name source_tb
Top-level Entity Name Vector_Quantization
Family Cyclone III
Device EP3C10U256C8
Timing Models Preliminary
Met timing requirements N/A
Total logic elements 729 / 10,320 (7 %)
Total combinational functions 675 / 10,320 (7 %)
Dedicated logic registers 132 / 10,320 (1 %)
Total registers 132
Total pins 89 / 183 (49 %)
Total virtual pins 0
Total memory bits 20,480 / 423,936 (5 %)
Embedded Multiplier 9-bit elements 4 / 46 (9 %)

Figure 67: Summary of resources usage of the Split-Vector Quantization module

4.2.3.9.8 Processing time

This module runs in parallel with the DCT, where every coefficient is quantized once it is produced by the DCT. Hence, the processing time is taken as the maximum quantization time of the 14 features, which is 256 clock cycles

needed to search for the minimum distance in the quantization tables of C0 and LogE.

In case of LUT implementation of DCT, the Vector Quantization will take longer time than DCT, and hence will not be masked by the DCT time, in this case the Vector Quantization processing time will be:

*Processing time = 256+64*6 = 640 clocks*

4.2.3.10 Bit Stream Framing

4.2.3.10.1 Basic functionality

This module forms the bitstream used to transmit the compressed feature vectors, using the defined frame structure and the error protection mechanism defined in the standard.

In order to reduce the transmission overhead, each multiframe message packages speech features from multiple short-time analysis frames. A multiframe, as shown in **Figure 68**, consists of a synchronization sequence, a header field, and a stream of frame packets.

Sync Sequence	Header Field	Frame Packet Stream
<- 2 octets ->	<- 4 octets ->	<- 138 octets ->
		<- 144 octets ->

Figure 68: Multiframe format [2]

In order to improve the error robustness of the protocol, the multiframe has a fixed length (144 octets). A multiframe represents 240 ms of speech, resulting in a data rate of 4 800 bits/s.

According to the standard, octets are transmitted in ascending numerical order; inside an octet, bit 1 is the first bit to be transmitted. When a field is contained within a single octet, the lowest-numbered bit of the field represents the lowest-order value (or the least significant bit). When a field spans more than one octet, the lowest-numbered bit in the first octet represents the lowest-order value (LSB), and the highest-numbered bit in the last octet represents the highest-order value (MSB). An exception to this field mapping convention is

made for the cyclic redundancy code (CRC) fields. For these fields, the lowest numbered bit of the octet is the highest order term of the polynomial representing the field. In simple stream formatting diagrams, fields are transmitted left to right.

Each multiframe begins with the 16-bit synchronization sequence 0 × 87B2. Following the synchronization sequence, a header field is transmitted. Ordering of the message data and parity bits is shown in Figure 69, and definition of the fields appears in Figure 70. The 4 bit multiframe counter gives each multiframe a modulo-16 index. The counter value for the first multiframe is "0001". The multiframe counter is incremented by one for each successive multiframe until the final multiframe. The final multiframe is indicated by zeros in the frame packet stream

Bit	8	7	6	5	4	3	2	1	Octet
	EXP1	MframeCnt				feType	SampRate		1
	EXP9	EXP8	EXP7	EXP6	EXP5	EXP4	EXP3	EXP2	2
	P8	P7	P6	P5	P4	P3	P2	P1	3
	P16	P15	P14	P13	P12	P11	P10	P9	4

Figure 69: Header field format [2]

Field	No. Bits	Meaning	Code	Indicator
SampRate	2	sampling rate	00	8 kHz
			01	11 kHz
			10	undefined
			11	16 kHz
FeType	1	Front-end specification	0	standard
			1	noise robust
MframeCnt	4	multiframe counter	xxxx	Modulo-16 number
EXP1 - EXP9	9	Expansion bits (TBD)	0	(zero pad)
P1 - P16	16	Cyclic code parity bits	(see below)	

Figure 70: Header field definition [2]

The generator polynomial used to generate P1-P16 is:

$$g_1(X) = 1 + X^8 + X^{12} + X^{14} + X^{15}$$

The parity bits of the codeword are generated using the calculation:

$$\begin{bmatrix} P_1 \\ P_2 \\ P_3 \\ P_4 \\ P_5 \\ P_6 \\ P_7 \\ P_8 \\ P_9 \\ P_{10} \\ P_{11} \\ P_{12} \\ P_{13} \\ P_{14} \\ P_{15} \\ P_{16} \end{bmatrix} = \begin{bmatrix} 1&0&0&0&0&0&0&0&1&0&0&0&1&0&1&1 \\ 1&1&0&0&0&0&0&0&1&1&0&0&1&1&1&0 \\ 1&1&1&0&0&0&0&0&1&1&1&0&1&1&0&1 \\ 0&1&1&1&0&0&0&0&0&1&1&1&0&1&1&1 \\ 1&0&1&1&1&0&0&0&1&0&1&1&0&0&0&0 \\ 0&1&0&1&1&1&0&0&0&1&0&1&1&0&0&0 \\ 0&0&1&0&1&1&1&0&0&0&1&0&1&1&0&0 \\ 0&0&0&1&0&1&1&1&0&0&0&1&0&1&1&0 \\ 1&0&0&0&1&0&1&1&0&0&0&0&0&0&0&1 \\ 0&1&0&0&0&1&0&1&1&0&0&0&0&0&0&1 \\ 0&0&1&0&0&0&1&0&1&1&0&0&0&0&0&1 \\ 0&0&0&1&0&0&0&1&0&1&1&0&0&0&0&1 \\ 0&0&0&0&1&0&0&0&1&0&1&1&0&0&0&1 \\ 0&0&0&0&0&1&0&0&0&1&0&1&1&0&0&1 \\ 0&0&0&0&0&0&1&0&0&0&1&0&1&1&0&1 \\ 0&0&0&0&0&0&0&1&0&0&0&1&0&1&1&1 \end{bmatrix}^T \times \begin{bmatrix} SampRate1 \\ SampRate2 \\ feType \\ MFrameCnt1 \\ MFrameCnt2 \\ MFrameCnt3 \\ MFrameCnt4 \\ EXP1 \\ EXP2 \\ EXP3 \\ EXP4 \\ EXP5 \\ EXP6 \\ EXP7 \\ EXP8 \\ EXP9 \end{bmatrix}$$

Each 10 ms frame from the front-end is represented by the codebook indices. The indices for a single frame are formatted for a frame according to Figure 71. The exact alignment with octet boundaries will vary from frame to frame.

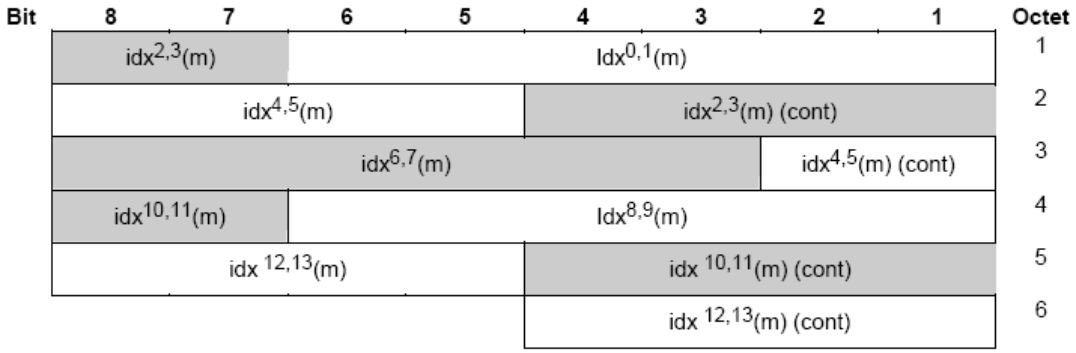


Figure 71: Frame information for mth frame [2]

Two frames worth of indices, or 88 bits, are then grouped together as a pair. A 4-bit CRC with generator polynomial

$$(g(X) = 1 + X + X^4)$$

It is calculated on the frame pair and immediately follows it, resulting in a combined frame pair packet of 11,5 octets. Twelve of these frame pair packets are combined to fill the 138 octet feature stream. Figure 72 illustrates the format of the protected feature packet stream. When the feature stream is combined with the overhead of the synchronization sequence and the header, the resulting format requires a data rate of 4800 bits/s.

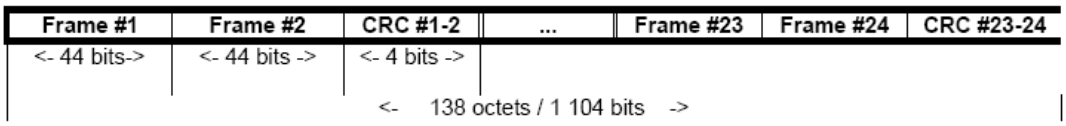


Figure 72: CRC protected feature packet stream [2]

4.2.3.10.2 Internal Architecture

The module functionality is handled through a 4-state State machine. A CRC engine is used to generate the header CRC, and another one is used to generate the frame pair CRC's, both are controlled and activated according to the state machine. Another core module is used to form the frame and assign the frame length according to the current state of the state machine.

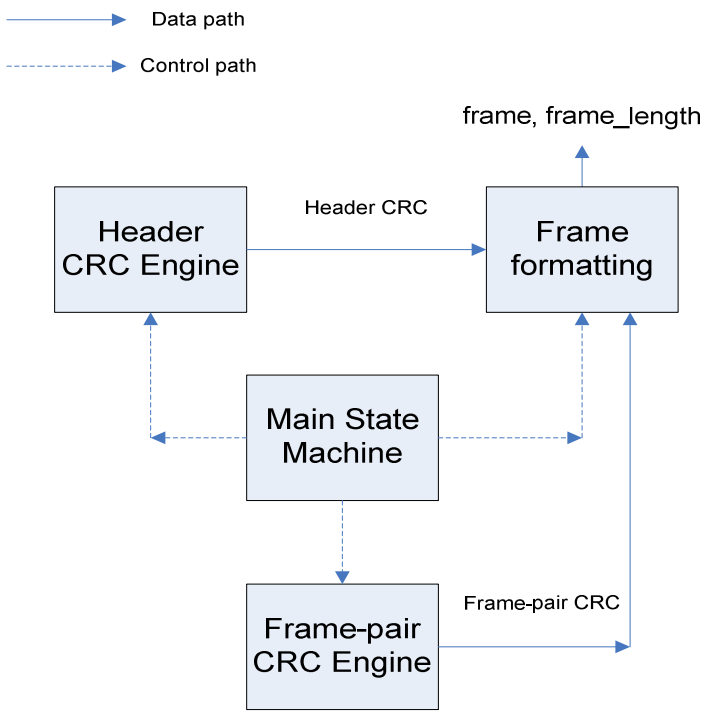


Figure 73: Internal architecture of the Bit Stream Framing

4.2.3.10.3 Configuration

Parameter	Possible values	Default value	Description
Sampling_Rate	“00”	“00” : for 8 kHz “01” : for 11 kHz “11” : for 16 kHz	The configured sampling frequency. This will be included in the header field of the Multi-frame.

Table 25: Configuration table of the Bit framing module

4.2.3.10.4 Signal width justification

None

4.2.3.10.5 State Machines

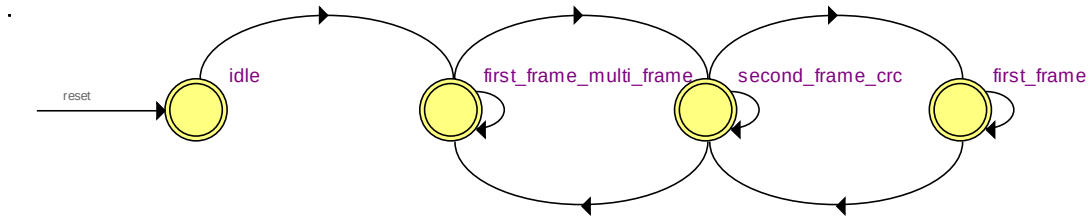


Figure 74: State machine of the Bit stream framing component

Source State	Destination State	Condition
idle	first_frame_multi_frame	
first_frame_multi_frame	first_frame_multi_frame	(!send_frame)
first_frame_multi_frame	second_frame_crc	(send_frame)
first_frame	first_frame	(!send_frame)
first_frame	second_frame_crc	(send_frame)
second_frame_crc	first_frame_multi_frame	frame_counter = 24
second_frame_crc	first_frame	frame_counter < 24

Table 26: State transition of the Bit stream framing state machine

The description of the states in Table 26 is as follows:

- *IDLE:* this transitional state just to reset the internal counters of the component.
- *FIRST_FRAME_MULTI_FRAME:* being in this state indicates that this is the first frame in the current multi frame. The module remains in this state until a send_frame command is triggered, and then it goes to *SECOND_FRAME_CRC* state. In this state the header field is formed and appended before the data frame.
- *FIRST_FRAME:* this state indicates that the frame is the first of a frame pair, but not the first of a multi frame, so no header or CRC fields are added to the frame. The module will go to the *SECOND_FRAME_CRC* when a send_frame command is triggered, otherwise is remains in its state.

- *SECOND_FRAME_CRC:* this state indicates that this is the second frame of a frame pair, which means that 4 bits frame CRC should be calculated and appended to the formatted frame. If 24 frames were sent, then this multi frame is terminated, so the module goes to *FIRST_FRAME_MULTI_FRAME* state, otherwise it goes to *FIRST_FRAME* state.

4.2.3.10.6 Memory requirements

None.

4.2.3.10.7 Actual Chip Usage

The following is a summary of the chip usage as generated by the Quartus II software:

Flow Status Successful - Fri Oct 03 17:11:07 2008
Quartus II Version 7.2 Build 203 02/05/2008 SP 2 SJ Web Edition
Revision Name source_tb
Top-level Entity Name Bit_Framing
Family Cyclone III
Device EP3C10U256C8
Timing Models Preliminary
Met timing requirements N/A
Total logic elements 474 / 10,320 (5 %)
Total combinational functions 474 / 10,320 (5 %)
Dedicated logic registers 4 / 10,320 (< 1 %)
Total registers 4
Total pins 147 / 183 (80 %)
Total virtual pins 0
Total memory bits 0 / 423,936 (0 %)
Embedded Multiplier 9-bit elements 0 / 46 (0 %)
Total PLLs 0 / 2 (0 %)

Figure 75: Summary of resources usage of Bit framing module

4.3 Overall System Performance

4.3.1 Actual Resources Utilization

The overall usage for the whole chip based on the configuration of the frame length is shown below for the FPGA device Cyclone III

EP3C10U256C8. Note that, the following results are based on LUT table implementation of Hamming Window, FFT and DCT.

Flow Status Successful - Fri Mar 13 06:39:21 2009
Quartus II Version 7.2 Build 203 02/05/2008 SP 2 SJ Web Edition
Revision Name source_tb
Top-level Entity Name Front_End_Processor
Family Cyclone III
Device EP3C10U256C8
Timing Models Preliminary
Met timing requirements No
Total logic elements 7,844 / 10,320 (76 %)
Total combinational functions 7,724 / 10,320 (75 %)
Dedicated logic registers 1,179 / 10,320 (11 %)
Total registers 1179
Total pins 138 / 183 (75 %)
Total virtual pins 0
Total memory bits 39,712 / 423,936 (9 %)
Embedded Multiplier 9-bit elements 46 / 46 (100 %)
Total PLLs 0 / 2 (0 %)

Figure 76: Actual resources usage: Sampling Rate = 8 kHz, N=200, FFTL=256, Cyclone III EP3C10U256C8

Flow Status Successful - Fri Mar 13 06:39:21 2009
Quartus II Version 7.2 Build 203 02/05/2008 SP 2 SJ Web Edition
Revision Name source_tb
Top-level Entity Name Front_End_Processor
Family Cyclone III
Device EP3C10U256C8
Timing Models Preliminary
Met timing requirements No
Total logic elements 7,844 / 10,320 (76 %)
Total combinational functions 7,724 / 10,320 (75 %)
Dedicated logic registers 1,179 / 10,320 (11 %)
Total registers 1179
Total pins 138 / 183 (75 %)
Total virtual pins 0
Total memory bits 42,512 / 423,936 (10 %)
Embedded Multiplier 9-bit elements 46 / 46 (100 %)
Total PLLs 0 / 2 (0 %)

Figure 77: Actual resources usage: Sampling Rate = 11 kHz, N=256, FFTL=256, Cyclone III EP3C10U256C8

Flow Status	*Successful - Fri Mar 13 06:08:27 2009*
Quartus II Version	*7.2 Build 203 02/05/2008 SP 2 SJ Web Edition*
Revision Name	*source_tb*
Top-level Entity Name	*Front_End_Processor*
Family	*Cyclone III*
Device	*EP3C10U256C8*
Timing Models	*Preliminary*
Met timing requirements	*N/A*
Total logic elements	*8,575 / 10,320 (83 %)*
Total combinational functions	*8,447 / 10,320 (82 %)*
Dedicated logic registers	*1,186 / 10,320 (11 %)*
Total registers	*1186*
Total pins	*138 / 183 (75 %)*
Total virtual pins	*0*
Total memory bits	*58,928 / 423,936 (14 %)*
Embedded Multiplier 9-bit elements	*46 / 46 (100 %)*
Total PLLs	*0 / 2 (0 %)*

Figure 78: Actual resources usage: Sampling Rate = 16 kHz, N=400, FFTL=512, Cyclone III EP3C10U256C8

4.3.2 Processing time and Speed limitations

The processing time performance discussed here is the time taken to perform:

- MFCC Features Extraction Algorithm,
- Split-Vector Features Quantization and Compression,
- And Bit-Stream Frame Formatting.

The calculations mentioned here assume that the speech frame (N-samples) is already buffered and ready. In other words, the pipelining delay till the frame is buffered is not considered, since it depends on the input sampling rate and not on the system performance. In general, this pipelining time is calculated as (in clock cycles):

$$N \times \frac{SamplingFrequency}{InternalChipFrequency}(clocks)$$

Where N is the number of samples in a speech frame, which depends on the configured sampling frequency as in **Table 8**. The sampling frequency can be 8, 11 or 16 kHz. The internal chip frequency depends on the hardware platform used.

4.3.2.1 Frame Processing Time

The *Frame Processing Time* is the summation of the time required by the Offset Compensation, Pre-emphasis, Hamming Window, FFT, Mel-Filter, DCT, Vector Quantization and Bit-Stream Framing modules to process a speech frame of N-samples, from the instant they are ready in the input samples buffer till the output frame bit-stream is ready at the output ports. In other words, it is the time between the rising of the input signal "store" to the time of the rising of the output signal "frame_ready".

After the whole frame is processed, the *I RAM* and *Q RAM* should be reset again, which requires FFTL (FFT length) clocks (256/512). This is done in parallel with the last feature quantization, which requires 64 clocks, so it is less than the memory reset time, hence it is not considered in the total time. In general, the Vector Quantization module runs in parallel with the DCT module, so its time is masked by the DCT processing time, and hence not included in the total time calculation.

For the very first frame, the input samples buffer is empty, so the FFT and the consecutive modules should wait till N-samples are ready. Hence, for the very first frame the Offset Compensation, Pre-Emphasis and Hamming Window times will be added to the Total time.

$$\textit{First Frame Processing Time} =$$

$$\textit{Offset Compensation Time} + \textit{Pre-Emphasis Time} + \textit{Hamming Window Time} + \textit{FFT Time} + \textit{Mel-Filter Time} + \textit{DCT Time} + \textit{Memory Reset Time} + \textit{Bit-Stream Framing Time} =$$

$$N + N + (C + 1) \times N + FFTL/2 \times \log_2 FFTL + C \times \sum_{i=1}^{i=\log_2 FFTL} 2^i + L + N\_MEL \times K \times D + NL + C \times N\_MEL \times N\_CEPSTRAL + FFTL$$

However, for consecutive frames, the operation of the Offset Compensation, Pre-Emphasis and Hamming Window modules will be done in parallel with the operation of the FFT and consecutive modules, so the processing times of the first three modules is not considered.

$$\textit{Next Frames Processing Time} =$$

$$\textit{FFT Time} + \textit{Mel-Filter Time} + \textit{DCT Time} + \textit{Memory Reset Time} + \textit{Bit-Stream Framing Time} =$$

$$FFTL/2 \times \log_2 FFTL + C \times \sum_{i=1}^{i=\log_2 FFTL-1} 2^i + L + N\_MEL \times K \times D + NL + C \times N\_MEL \times N\_CEPSTRAL + FFTL$$

Where:

- The number of samples per frame = N = 200/256/400.
- The FFT length = FFTL = 256/512.
- The number of Mel-Filter Banks = N_MEL = 23.
- The number of Cepstral Coefficients = N_CPESTRAL = 13.
- Time taken by the CORDIC Sine/Cosine Calculator = C = 12.
- Time taken by the CORDIC Magnitude Calculator = L = 11.
- Time taken by the CORDIC Divider Calculator = D = 11.
- Time taken by the CORDIC Natural Logarithm Calculator = NL = 16.
- Largest difference between any two center frequencies cbin(k-1) and cbin(k+1) = K =
    - 21 in case of sampling frequency 8 kHz ,and
    - 23 in case of 11 kHz, and
    - 51 in case of 16 kHz.

The Energy Measure operation is always performed in parallel with other modules operation, so its processing time is always masked and hence not considered in the total time calculation.The configuration of these parameters is show in Table 8 and Table 38.

The numerical value of the processing time will be different according to the configured sampling rate and the corresponding configuration parameters that follow it. The following table shows the different processing times with the sampling rate:

	Fs = 8 kHz	Fs = 11 kHz	Fs = 16 kHz
First Frame	15124 clock cycles	16492 clock cycles	29644 clock cycles
Next Frames	12124 clock cycles	12652 clock cycles	23644 clock cycles

Table 27: Frame processing time with different sampling frequencies

The maximum allowed processing time for each frame is 9.16 ms as discussed in 4.1.1. Now, we wish to get the ratio (in %) between the allowed and actual consumed time, which will follow the following equation:

$$\frac{Frame\,\mathrm{Pr}oces\sin gTimeInClocks / InternalChipFrequency}{MaximumAllowed\,\mathrm{Pr}oces\sin gtime}\times 100\%$$

The following table shows this percentage for Internal Chip Frequency of 100 MHz:

	Fs = 8 kHz	Fs = 11 kHz	Fs = 16 kHz
First Frame	1.7284%	1.8847%	3.3878%
Next Frames	1.3856%	1.4459%	2.7021%

Table 28: Frame processing time as a percentage of the allowed time for 100 MHz chip frequency

4.3.2.1.1 Look-up table implementation

In case of Look-up table implementation of Hamming Window, FFT and DCT mentioned in 4.2.3.4.2.1, 4.2.3.6.2.14.2.3.8.3.1, the frame processing time will be:

First Frame Processing Time =

Offset Compensation Time + Pre-Emphasis Time + Hamming Window Time + FFT Time + Mel-Filter Time + Vector Quantization time + Bit-Stream Framing Time =

$$N + N + (C + 1) \times N + FFTL/2 \times L + FFTL + L + N\_MEL \times K \times D + NL + 640$$

Next Frames Processing Time =

FFT Time + Mel-Filter Time + Vector Quantization time + Bit-Stream Framing Time =

$$FFTL/2 \times L + FFTL + L + N\_MEL \times K \times D + NL + 640$$

Note that; in case of LUT implementation, the Vector Quantization time is larger than the DCT time; hence, the DCT time is masked.

	Fs = 8 kHz	**Fs = 11 kHz**	**Fs = 16 kHz**
First Frame	10633 clock cycles	11979 clock cycles	22887 clock cycles
Next Frames	7633 clock cycles	8139 clock cycles	16887 clock cycles

Table 29: Frame processing time with different sampling frequencies for LUT implementation

	Fs = 8 kHz	**Fs = 11 kHz**	**Fs = 16 kHz**
First Frame	1.1608 %	1.3078 %	2.4986 %
Next Frames	0.8333 %	0.8885 %	1.8436 %

Table 30: Frame processing time as a percentage of the allowed time for 100 MHz chip frequency for LUT implementation

4.3.2.2 Minimum Internal Chip Frequency

The internal chip speed is limited by the input frame rate, where the internal chip processing should be faster than this rate, otherwise some input samples will be missed, and the input samples buffer will overflow. The frame processing time was calculated in the previous section, hence, this time (in seconds) should be less than or equal to the input frame rate. In this calculation we will consider the next frames processing time and not the first frame, since the first frame time only occurs in the very beginning of the system operation. The minimum internal chip frequency can be deduced from the following equation:

$$\frac{NextFrameTimeInClocks}{MinimumInternalChipFrequency} = \frac{M\,(FrameShiftIntervalInSamples)}{SamplingFrequency} = 10ms$$

For this equality, the percentage of the maximum allowed frame processing time consumed by the system is 100%.

The frame shift interval (M) configuration is shown in **Table 8**. The following table shows the minimum internal chip speed for different sampling frequencies:

	Fs = 8 kHz	**Fs = 11 kHz**	**Fs = 16 kHz**
Minimum Internal Chip Frequency (MHz)	1.2124	1.2652	2.3644

Table 31: Minimum Internal Chip Speed for different Sampling Frequencies

In case of Look-up table implementation the results will be:

	Fs = 8 kHz	**Fs = 11 kHz**	**Fs = 16 kHz**
Minimum Internal Chip Frequency (MHz)	0.7633	0.8139	1.6887

Table 32: Minimum Internal Chip Speed in case of LUT implementation

4.3.3 Memory

The only RAM memory required is in the Buffer Manager module mentioned in 4.2.3.500. The ROM memory exists in Mel-Filter and Split-Vector Quantization module 4.1.7 and 4.1.9.

The overall memory requirements are shown in the following table:

SamplingFrequency	**Size in bits**	**Type**
8 kHz	16192	RAM
	20880	ROM
11 kHz	18432	RAM
	20880	ROM
16 kHz	32384	RAM

	20880	ROM

Table 33: Memory requirements of the System

4.4 *Effect of Run-time configurability of the chip*

The configurations mentioned in the modules detailed are done statically, which means that; once the chip is manufactured these configurations cannot be modified anymore. This section discusses the required modifications to the system to enable making this configuration process at run-time, such that modifications can be done dynamically during the operation of chip. To enable this modification, the architecture of the system is modified, such that a new module is added to manage different configurations, which is the *Configuration Manager*.

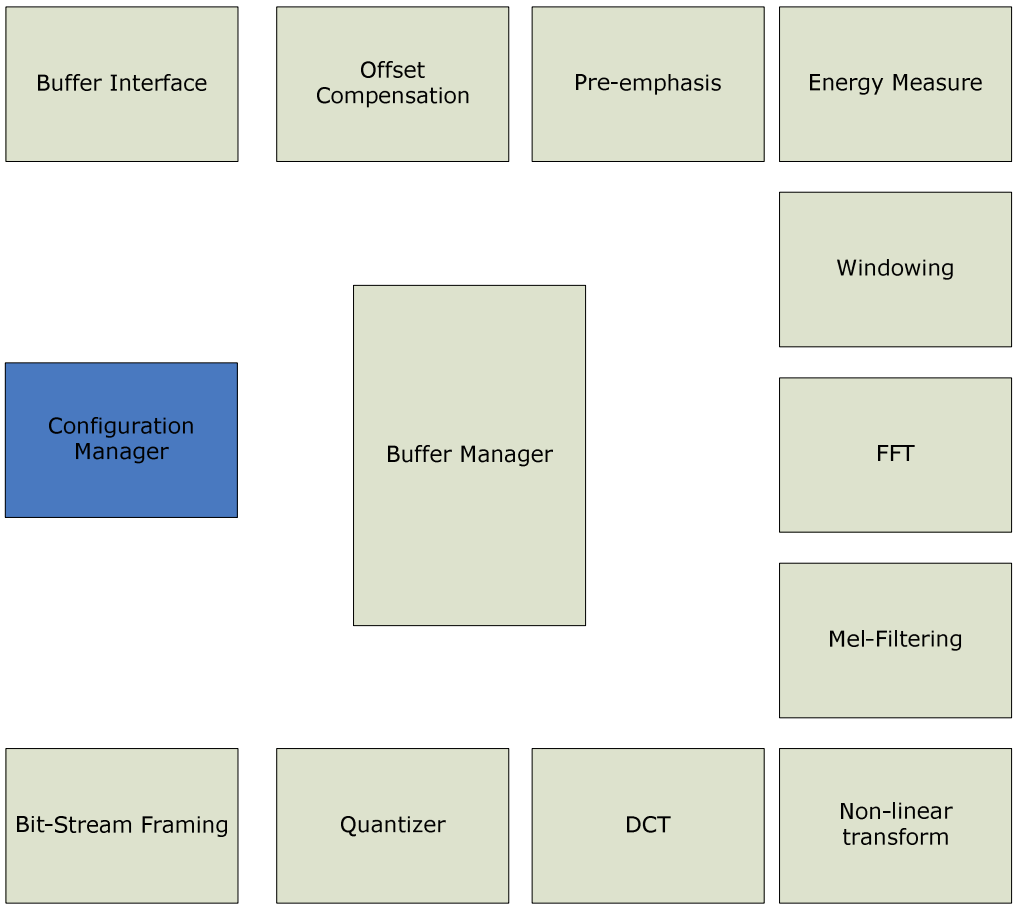


Figure 79: Modified Static Architecture for Run-time configurability

This will certainly affect many modules of the system in different ways; this effect will be mentioned for each module- if affected- independently in the next sections.

4.4.1 Energy Measure

This module has a configuration parameter N as mentioned in 4.2.3.3.3. This configuration parameter will be declared as an input variable that is modified by the *Configuration Manager*. This reading frequency of this variable will be every frame, such that it will be ineffective to change it during the processing of the current frame.

Parameter	Update frequency
N	Every frame

Table 34: Update frequency of the configuration paramters of the Energy Measure component

The *Configuration Manager* will be in charge of reading this variable from the RAM area dedicated for chip run-time configuration.

4.4.2 Windowing

This module has a configuration parameter N as mentioned in section 4.2.3.4.3. This configuration parameter will be declared as an input variable that is modified by the *Configuration Manager*. This reading frequency of this variable will be every frame, such that it will be ineffective to change it during the processing of the current frame.

Parameter	Update frequency
N	Every frame

Table 35: Update frequency of the configuration paramters of the Windowing component

The *Configuration Manager* will be in charge of reading this variable from the RAM area dedicated for chip run-time configuration.

4.4.3 Buffer Manager

This is the most affected component of the system, where the memories managed by this component shall be declared to their maximum, assuming 512 FFT points, however, the actual used portion of these memories will be controlled by the configuration parameters managed by the *Configuration Manager*.

This module has the configuration parameters mentioned in section 4.2.3.5.3. The run-time configurable parameters are mentioned in the following table. These configuration parameters will be declared as input variables that are modified by the *Configuration Manager*. The update frequency of these parameters is shown in the following table.

Parameter	Update frequency
N	Every frame
FFTL	Every frame
M	Every frame

Table 36: Update frequency of the configuration paramters of the buffer manager component

The *Configuration Manager* will be in charge of reading these variables from the RAM area dedicated for chip run-time configuration.

4.4.4 FFT

This module has the configuration parameters mentioned in section 4.2.3.6.3. The run-time configurable parameters are mentioned in the following table. These configuration parameters will be declared as input variables that are modified by the *Configuration Manager*. The update frequency of these parameters is shown in the following table.

Parameter	Update frequency
depth	Ever frame

Table 37: Update frequency of the configuration paramters of the FFT component

The *Configuration Manager* will be in charge of reading these variables from the RAM area dedicated for chip run-time configuration.

4.4.5 Mel-Filter

The cbink coefficients will be stored in two groups in ROM, one for FFTL of 256 and the other for FFTL of 512. The proper group will be used according to the current configuration. Using a different group than the current one is not allowed during the processing of the current frame. Apparently, this will double the needed ROM space to 50×16 bits for the cbink coefficients.

The *Configuration Manager* will be in charge of indicating which group to be used according to the required configuration.

4.4.6 Split-Vector Quantization

This module is affected in the quantization tables that it uses, where every sampling frequency will use different quantization table. In case of compile-time configuration, the proper quantization table was loaded in ROM according to the sampling frequency configured, which cannot be the case for run-time configuration of the sampling frequency.

One solution to this problem is to load the three tables (for 8, 11 and 16 kHz sampling rates) in ROM and using the proper one according to the required rate. However this solution would require triple the ROM area used before, which is about 7.5 Kbytes.

The other solution is to put the three tables in an External ROM, and load the proper one to the On-chip ROM based on the required rate. The disadvantage of this solution is that it requires long copying time from External to On-chip ROM. However, it is not expected that the rate of changing the sampling rate configuration to be high, so this long copy operation will be performed rarely during system operation.

4.4.7 Configuration Manager

This is the new component added to manage run-time configurability. A dedicated RAM area will be declared for run-time configurations, which will

be accessible by the external entity (the user) for writing, and by the *Configuration Manager* for reading, this will be referred as the *Configuration RAM* area. The only configurable parameter by the external entity is the sampling rate. The correspondence between the sampling rate and other configurations parameters is mentioned in Table 38:

Sampling rate (kHz)	**N(Frame length)**	**M(Frame shift)**	**FFTL (FFT length)**
8	200	80	256
11	256	110	256
16	400	160	512

Table 38: Relation between sampling rate and other configuration parmaters

The responsibilities of this module are as follows:

- Read the sampling rate configuration from the *Configuration RAM* and communicate the proper configuration parameters to the modules that need them, like Mel-Filter for example.
- Copy the proper quantization tables from external ROM to the on-chip ROM whenever the sampling rate configuration is modified.

Chapter 5

5 Compliance to the Aurora Standard Test Vectors

Hardware testing is usually a hard task. This is because hardware debugging is hard, and locating the problem takes longer time that testing a Software program. Also, fixed point errors need to be tested carefully to ensure that the hardware implementation of a reference software algorithm is not deviating away to give results that are far from being correct. In modern HDL's, test benches facilitates the task of hardware testing to some extent, together with modern simulation and validation tools that are integrated with the development tools and synthesizers to form an integrated development and testing environment.

In order to test the validity of the design, the fixed point results of simulating the system using ModelSim software is validated against standard results of floating point implementation provided by the ETSI with the Aurora standard, so that compliance to the standard is proved.

In this chapter we will present the test bench setup used to test and validate the front end hardware. The types of test cases performed will be clearly explained. All kinds of tools used to develop, simulate or test the system will be mentioned and explained in details. And finally, the simulation and testing results of the test cases mentioned will be presented.

5.1 Test Bench Setup

When writing a design, it is important to verify its functionality. The most common method of doing this is to create a test bench, i.e., instantiating a *Device Under Test* (DUT), generate test vectors (a set of inputs), and monitor the output, as shown in Figure 80. Common test bench tasks are to generate clock and reset signals, and read/write information to a file. Writing the output values to a file makes it possible to verify the result using test scripts written in high level language like C-language.

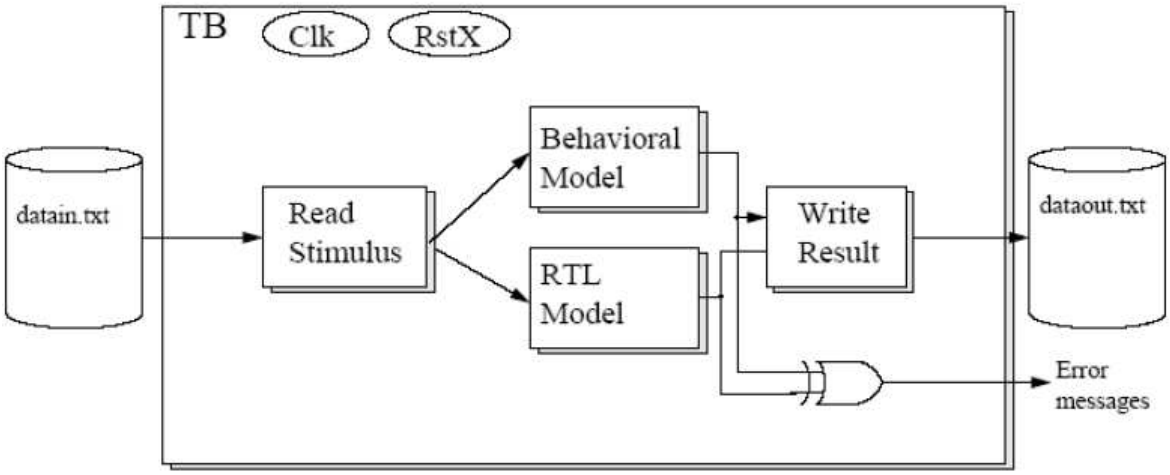


Figure 80: Typical VHDL Test Bench [16]

5.2 *Performed Test Cases*

The front end system was tested and validated against the reference result vectors generated by the high level C-code provided by ETSI with the Aurora standard ETSI ES 201 108. The same input stimulus vector of samples that is used with the reference C-code was applied to the front end system designed in hardware and coded in VHDL. The VHDL code will be referred from now on as the *Device Under Test* (DUT). The outputs of both systems (the reference C-code and the DUT) are then compared to detect the DUT performance against the reference high level code.

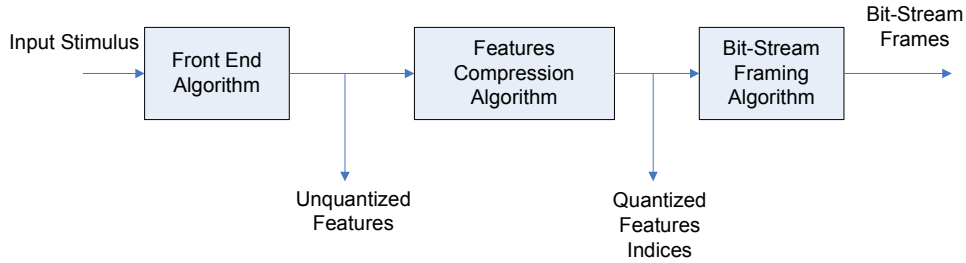


Figure 81: Observed System Outputs

Here we are interested in two observed outputs from both systems, shown in Figure 81:

- *The Unquantized Features Vector:* consists of the 13 Cepstral Coefficents resulting after the DCT operation, plus the natural logarithm of the frame energy.
- *The Quantized Features Indices:* consists of 7 indices representing the quantized features.

Using the above observations from reference and DUT systems, the following tests can be performed:

5.2.1 Unquantized Features Error Test- Test 1

Output features before quantization and after the DCT operation of both the reference and DUT systems are compared to each others. The *Average Absolute Error* is the result of this test, and is defined by the following equation:

$$AverageAbsoluteError = \frac{\sum_{AllTestedFramesFeatures} | DUTFeature - \mathrm{Re}\, ferenceFeature |}{TotalNumberOfFeatures}$$

The above number is simply the deviation between the DUT and reference features. In other words, the correct features are in the range the DUT features ± The *Average Absolute Error*.

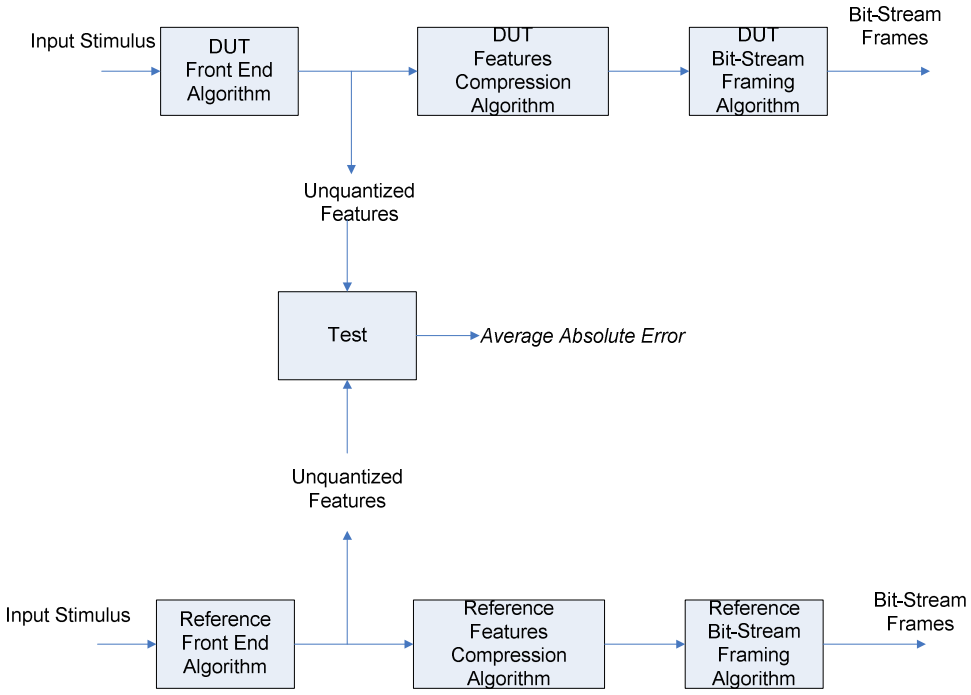


Figure 82: Unquantized Features Error Test

5.2.2 Quantized Features Error Test- Test 2

In this test, the two output quantized features indices are considered, decoded to get the corresponding features, then the two decoded features of the DUT and the reference systems are compared to get two results:

1. *The Average Absolute Error:* calculated exactly as in 5.2.1.
2. *The Error Rate:* calculated as follows:

$$ErrorRate = \frac{NumberOfIncorrectIndices}{TotalNumberOfTestedIndices}$$

Again, the correct features are in the range the DUT features ± The *Average Absolute Error*.

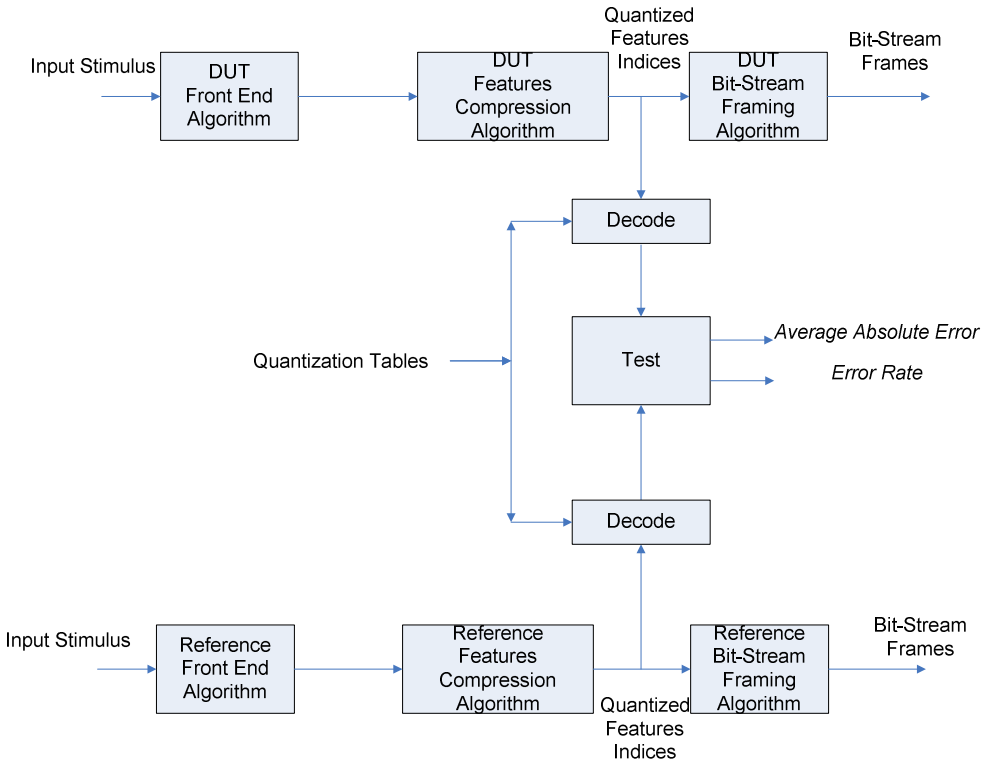


Figure 83: Quantized Features Error Test

The *Error Rate* represents the number of times by which the DUT quantized indices deviates from the reference ones through the whole tested features of all the tested frames. Normally, this rate should be low, as the DUT quantized indices are usually the same as the reference ones, except for very few features that the fixed point error (represented in the *Average Absolute Error*) makes the Quantiser mis-classify the features pair to a wrong index. The following figure is a snap-shot of the results file of the difference between the DUT and reference decoded features from the quantized indices using the quantization tables. It is clear that most of the resulting difference is”0”, which means the DUT and reference indices are exactly the same.

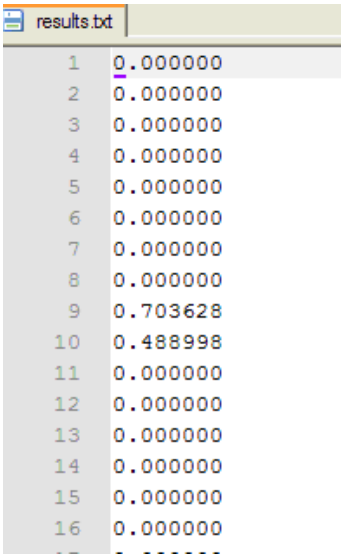


Figure 84: Snap-Shot of the difference between decoded DUT and Reference Features

5.2.3 Quantization Error Test- Test 3

Since the last stage of the front end system is the Split-Vector Quantization, a quantization error must exist for both reference and DUT systems. This test aims at observing the difference in the *Average Quantization Error* between the DUT and reference systems. The *Average Quantization Error* for both DUT and reference systems is defined as:

$$AverageQuantizationError = \frac{\sum_{AllTestedFrameIndices} |\, QuantizedFeature - UnquantizedFeature \,|}{TotalNumberOfTestedFramesFeatures}$$

The *Quantized Features* are the decoded features from the result quantized indices using the quantization tables. The output of this test is the *Difference In Quantization Error*, which is the difference between the DUT and reference quantization errors. The number should be small.

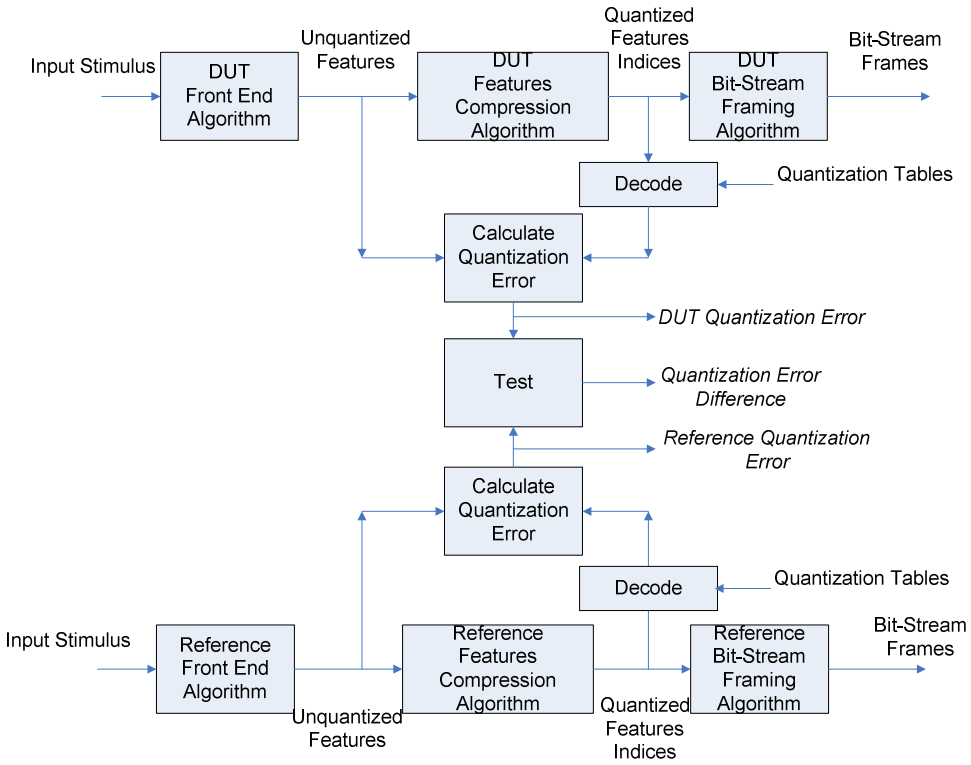


Figure 85: Quantization Error Difference Test

5.3 *Environment and Tools*

Quartus II 7.2 IDE is used for development, synthesis, module simulation, FPGA programming bitmap generation and net list writing. ModelSim PE Student Edition 6.4 a tool is used to perform full simulation and module integration tests. Finally, Visual Studio 6.0 IDE was used to develop, compile and link the test scripts used with the test batch files to perform the required test cases.

5.4 *Testing and Simulation Results*

Two configurations of the front end were tested, which are:

- Configuration A: sampling frequency = 8 kHz, frame length = 200 sample.
- Configuration B: sampling frequency = 16 kHz, frame length = 400 sample.

The tests explained in 5.2 were executed on the two chip configurations above, and the results are summarized in the following tables. The FPGA chip used is Cyclone III EP3C10U256C8. For more information about the performed test see 5.2.

	Test 1	**Test 2**		**Test 3**		
	Average Absolute Error	*Average Absolute Error*	*Error Rate*	*DUT Quantisation Error*	*Reference Quantisation Error*	*Quantisation Error Difference*
Configuration A	0.033725	0.041925	0.052188	1.902172	1.899551	0.002621
Configuration B	0.049192	0.210935	0.082411	1.670316	1.665433	0.004883

Table 39: Testing and Simulation results

The following observations can be drawn from the results in Table 39:

- For Test 1: the *Average Absolute Error* seems to increase slightly for Configuration B than Configuration A.
- For Test 2: the *Average Absolute Error* increases notably for Configuration B than Configuration A. However, the *Error Rate* experiences a slight increase in Configuration B than Configuration A.
- For Test 3: the *Quantisation Error* seems to remain unaffected in the DUT than the reference system in both configurations. However, the difference in Configuration B is nearly double that of Configuration A.

The general conclusion that can be drawn from these results is that:

1. As appears from Test 3 results, the DUT do not add significant error to the already existing quantization error in the reference system, which means that the performance of the whole *Distributed Speech Recognition* system will remain unaffected by the fixed point approximations done in the hardware implementation of the front end part. Even in Test 1 and 2 results, the *Average Absolute Error* remains small. This error-as mentioned before- means that the correct features are in the range

the DUT features ± The *Average Absolute Error*, hence, the DUT is correct to 2 decimal places in most cases, and to 1 decimal place in only one case.

2. As the frame length increases, which means more iterations and steps in summations, this increases the fixed point errors significantly (nearly the double in most cases), though remains reasonable even for the longest frame length in Configuration B.

Chapter 6

6 System Benchmarks

In this chapter, we try to find a way to evaluate the front end processor hardware design presented in this thesis by comparing it to other implementations and reference designs.

Benchmarking is a way to measure performance of a computer system. More specifically, benchmark is a reference algorithm or program used to quantitatively evaluate computer hardware and software resources. To get a better picture of a computer system, engineers define benchmark suites - sets of benchmarks. By choosing a suitable benchmark for a system it is possible to test if it behaves the way we expect.

The above definition of a benchmark is more suitable to software programs and algorithms, however, we will try to alter it a little to suite the hardware custom designs like-in our case- the front end speech processor. Following the above definition of a benchmark, we consider the Front end processing together with the vector quantization algorithms defined in the Aurora standard as the reference algorithm that is used to evaluate a certain design. The reference hardware platform will be Altera FPGAs (Cyclone III, Stratix II... etc). Evaluation is to be done based on the FPGA resources utilization and processing time.

Since the system is custom in its nature, there are no available complete hardware designs for the front end processor to be referred to as a benchmark. So, comparing the whole system to another reference one will not be possible. However, some of the main components constituting the system have reference hardware designs provided by the FPGA manufacturer itself (like Altera, Xilinix… etc), which are optimized for their target FPGAs. These components are:

- The Fast Fourier Transform (FFT) processor.
- The CORDIC processor.
- The hardware divider.

The above mentioned components are the most expensive resources usage and area consuming components in the system, like FFT. Also, the CORDIC processor, is used extensively in many parts of the system to do many functions, like Sine and Cosine calculations, Magnitude calculation, Logarithm calculation,… etc. So, comparing those main components individually to their reference designs provided by the FPGA manufacturer shall give a good indication of how optimized is the design presented compared to already existing related hardware designs in the area of digital signal processing.

In brief, the benchmark here will be based on the above components. What to be compared to reference designs will be FPGA resource utilization and processing time required. In addition, the whole system can be considered as a new benchmark for the front end speech processor designs in future related works. In the following sections, this comparison will be held, with their results clarified in tables.

6.1 Individual Components Comparison

In this section, individual comparison will be made between some chosen components of the front end processor, and their corresponding reference designs provided by the FPGA manufacturer. Throughout this study, Altera FPGA's reference designs will be referred to. Comparison is done based on more than one FPGA family, like Cyclone III, Stratix II,…etc.

6.1.1 Fast Fourier Transform Processor (FFT)

All the information in this section on the reference design features is extracted from [19] and [20].

As mentioned before, our design is area optimized rather than time optimized.

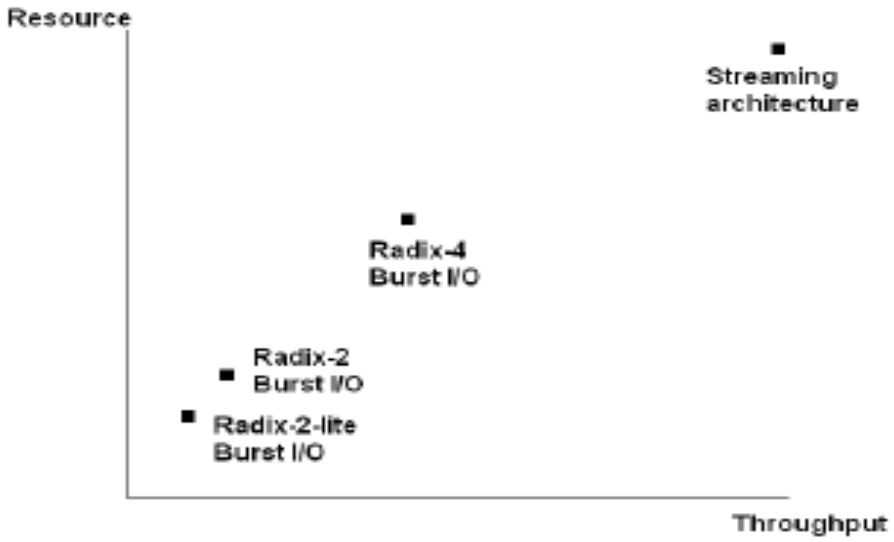


Figure 86: Resources versus throughput for Architectural options of FFT implementation [20]

Figure 86 (refer to [20]) illustrates the trade-off of throughput versus resource usage for four architectures

- Radix-2 lite Burst I/O
- Radix-2 Burst I/O
- Radix-4 Burst I/O
- Streaming architecture

For more information about the above architectures, see [19] and [20].

As a rule of thumb, each architecture offers at least a factor of "2" difference in resource from the next architecture. The most suitable architecture to compare our FFT to it is the radix-2 lite Burst I/O, since it is the resource optimized architecture among the four mentioned architectures. Next, comparison will be held for Radix-2 lite Burst I/O architecture versus the FFT presented in this thesis. Radix-2 lite Burst I/O architecture will be referred to as Burst Data Flow architecture with single-output, for more information about the details of this architecture type [20].

Table 40 shows a comparison between the front end FFT design and the Altera reference FFT design with Burst Data Flow architecture, Single output, 16 bits

signal width. For more information about the reference design architectures see [20].

	Device	Point	Points	Combinational LUTs	Logic Registers	Memory (M9K)	Memory (Bits)	Multipliers	Clock Cycle Count
Reference Design- Burst Data Flow(1)	EP3C10F256C6	Fixed	256	1,463	1,476	3	9,472	4 (18x18)	1628
Front End Design	EP3C10F256C6	Fixed	256	1,212	235	3	9,232	4 (18x18)	256
Result				17.1%(2)	84% (2)	0 %(2)	2.5%(2)	0 %(2)	6.3 (3)

Table 40: Comparison of FFT on Cyclone III Devices- Burst Data Flow Architecture, Single Output [19]

(1) The Reference Design Architecture Type.

(2) Difference (in %) between the Front End Design figure and the Reference Design figure of the corresponding feature. Where

Difference = (Reference Designs figure - Front End Design figure)/ Reference Designs figure

If this percentage is positive, then it means that the Front End Design outperformed the reference design in the corresponding feature, and vice versa. This number will represent the reduction (if positive) or increase (if negative) in resources introduced by the Front End Design over the Reference Design.

(3) The ratio between the times taken by the reference design to the time taken by the front end design.

Note that; the reference design does not provide the magnitude of the FFT output; hence, the resources utilization and time performance shown in the results do not take in consideration the magnitude calculation of the final FFT output.

The above presented comparisons show that the front end design of FFT in general outperformed the reference design in the FPGA resources utilization and time performance.

6.1.1.1 Analysis of the FFT benchmarking results

The detailed design aspects of the reference FFT design is not publicly available, since it is an IP core of Altera, however the improved performance of the local design over the reference one can be referred to the following reasons:

- In terms of time performance, the bit reversal operation of the local design is performed on the fly with every new input sample, where the bit reversed address is calculated immediately after every new sample is stored in the input buffer, this saves the time to re-order the samples after they are completely buffered, in addition this implicitly performs the zero padding operation.
- Using dual port memories (ROM and RAM) makes it possible to read and write two butterfly inputs or outputs in one clock cycle, which reduces the butterfly time by 50%.
- In terms of hardware utilization, making use of the embedded multipliers saves the need to implement them.
- Thanks to the fixed point width of 16 bits of all internal signals, it is possible to use the 18x18 on-chip embedded multipliers.
- In terms of memory resources, the LUT implementation is highly optimized by the memory algorithm used to store the sine/cosine values of the twiddle factors, where only ¼ the cosine wave is stored and the rest of the values are deduced from it.

6.1.2 CORDIC Processor

All the information in this section on the reference design features is extracted from [17].

Table 41 shows a comparison between reference and front end designs for CORDIC processor on Cyclone devices.

	Clocks	Logic Elements
Reference Design	16	963
Front End Design	16	399
Difference (%)(1)	0	58.5%

Table 41: Comparison between Reference and Front End Designs for CORDIC processor on Cyclone Devices [17]

(1) Difference (in %) between the Front End Design figure and the Reference Design figure of the corresponding feature. Where

Difference = (Reference Designs figure - Front End Design figure)/ Reference Designs figure

If this percentage is positive, then it means that the Front End Design outperformed the reference design in the corresponding feature, and vice versa. This number will represent the reduction (if positive) or increase (if negative) in resources introduced by the Front End Design over the Reference Design.

The results show that the time required for both designs is the same. However, the front end design offers 58.5 % reduction in resources utilization over the reference design.

6.1.3 Hardware divider

All the information in this section on the reference design features is extracted from [18]. Table 42 shows a comparison between the reference and front end designs for the hardware divider.

	FLEX® EP10K100E	APEX™ EP20K100E	ACEX® EP1K100
Reference Design	3,338	3,428	3,338
Front End Design	244	248	244
Difference (%)(1)	92.6 %	92.7%	92.6%

Table 42: Comparison between the Reference and Front End Designs for the hardware divider [4]

(1) Difference (in %) between the Front End Design figure and the Reference Design figure of the corresponding feature. Where

Difference = (Reference Designs figure - Front End Design figure)/ Reference Designs figure

If this percentage is positive, then it means that the Front End Design outperformed the reference design in the corresponding feature, and vice versa. This number will represent the reduction (if positive) or increase (if negative) in resources introduced by the Front End Design over the Reference Design.

From the Table 42, it is clear that the front end design offers more than 92 % reduction in resources utilization over the reference design.

The time required for the reference design of a hardware divider to finish is 15 clocks, while it is only 11 clocks for the front end design.

6.1.3.1 Analysis of the hardware divider benchmarking results

The local implemented hardware divider is enhanced by the use of CORDIC core in its linear version, which highly reduced the hardware resources. Also, thanks to the fast convergence of the CORDIC algorithm, 4 cycles saving in time performance is achieved.

6.2 Overall System Benchmark

As mentioned in the introduction of this chapter, since the front end processor system is custom by nature, so no overall benchmark exist for the

whole system to compare the design to it. Hence, the current design will be considered a reference for future works to compare to it. In the following, the system features (FPGA resources utilization and processing time) will be presented for all system configurations.

6.2.1 System performance on Cyclone III FPGA Family

In this section the front end design performance is presented when Cyclone III FPGA family devices are used.

	Total Logic Elements	**Total Registers**	**Total memory bits**	**Total multipliers**
EP3C10U256C8	7,844 (76 %)	1,179 (11 %)	39,712 (9%)	46(100%)
EP3C55F780C8	7,871 (14%)	1,179 (2%)	39,712 (2%)	46 (15%)

Table 43: Front End Processor Performance on Cyclone III Devices- Frame length configuration = 200 samples

	Total Logic Elements	**Total Registers**	**Total memory bits**	**Total multipliers**
EP3C10U256C8	7,844 (76 %)	1,179 (11 %)	42,512 (9%)	46(100%)
EP3C55F780C8	7,871 (14%)	1,179 (2%)	42,512 (2%)	46 (15%)

Table 44: Front End Processor Performance on Cyclone III Devices- Frame length configuration = 256 samples

	Total Logic Elements	Total Registers	Total memory bits	Total multipliers
EP3C10U256C8	8,575 (83%)	1,186 (12%)	58,928 (14%)	46(100%)
EP3C55F256C8	8,603 (15%)	1,186 (2%)	58,928 (2%)	46 (15%)

Table 45: Front End Processor Performance on Cyclone III Devices- Frame length configuration = 400 samples

6.2.2 System performance on Stratix II FPGA Family

In this section the front end design performance is presented when Stratix II FPGA family devices are used.

	Total Logic Elements	Total Registers	Total memory bits	Total multipliers
EP2S15F484C3	6,395 (51%)	1,169(9%)	40,096(10%)	54(56%)

Table 46: Front End Processor Performance on Stratix II Devices- Frame length configuration = 200 samples

	Total Logic Elements	Total Registers	Total memory bits	Total multipliers
EP2S15F484C3	6,395 (51%)	1,169(9%)	42,896(9%)	54(56%)

Table 47: Front End Processor Performance on Stratix II Devices- Frame length configuration = 256 samples

	Total Logic Elements	Total Registers	Total memory bits	Total multipliers
EP2S15F484C3	6,414 (51%)	1,176 (9%)	59,312(14%)	54(56%)

Table 48: Front End Processor Performance on Stratix II Devices- Frame length configuration = 400 samples

6.2.3 System time performance

Table 49 shows the frame processing time with different sampling frequencies

	Fs = 8 kHz	Fs = 11 kHz	Fs = 16 kHz
First Frame (clock cycles)	10633	11979	22887
Next Frames (clock cycles)	7633	8139	16887

Table 49: Frame processing time with different sampling frequencies

Chapter 7

7 Conclusions

In this thesis, the front-end part of the Distributed Speech Recognition system specified in the Aurora standard (ETSI ES 201 108 V1.1.3) is implemented in hardware. The VLSI design cycle and styles were presented, and a brief comparison was made between three of them to choose the proper one to implement the system. Based on this comparison, FPGA was chosen for prototyping the design, with consideration of migration to structured ASIC design fashion in case of mass production.

The constraints on the design were presented. Time constraint (10 ms frame processing time) is relatively relaxed compared to nowadays chip frequencies. On the other hand, area constraint and limited hardware resources are the major constraints, hence, the design criteria was directed towards hardware optimization.

Based on the above constraints, the system static and dynamic architecture were designed, where hardware optimized algorithm like CORDIC was used to implement non-linear computationally intensive operations in the system, like natural logarithm, magnitude, trigonometric function,...etc. Also, in some cases, two options of implementation were available; memory optimized solution and time optimized solution, like in case of Hamming window, FFT and DCT components. CORDIC algorithm was used in the memory optimized solution, and look-up tables were used in the time optimized solution.

The proposed design was synthesized and tested on 10 K gates low-cost Cyclone III FPGA. Finally, performance was evaluated based on compliance of the system output to the reference test vectors provided by ETSI. Also, some system components, like FFT, CORDIC and hardware divider were compared to reference designs provided by Altera.

7.1 Contributions

The first contribution of this thesis is the complete VLSI implementation of the front-end client of the DSR system in VHDL using FPGA design style, in contrast to software implementations, which were intended merely for the sake of DSR system simulation. The design presented was tested to the RTL simulation level, and benchmarked against Altera reference designs.

The second contribution made in this thesis is the optimization of the hardware implementation of the front-end system, and respecting the time constraint as well, such that, the complete system fits in 10 K gates FPGA utilizing 83% of its resources based on maximum system configuration. Using hardware optimized algorithm like CORDIC reduced the resources utilization of the numerous non-linear operation in the system, especially in non-linear transformations like natural logarithms in many points in the system, the magnitude calculation of the FFT output, and the trigonometric functions in many parts of the system. Also, the re-use of many components in the system optimized the hardware resources utilization. Using single computational core and iterating on it (like the cases of FFT, DCT, Mel-filter, and Vector Quantization), improved the hardware usage, keeping in mind the relaxed time constraint on the design.

Fixed point implementation constrained the signal widths in most of the system parts to be less than 18 bits; this was done to make use of the on-chip embedded multipliers and DSP MAC units (18 bits wide) instead of implementing them.

Memory resources usage was highly optimized in the design. For example, in case of FFT twiddle factors storage, only ¼ the cosine wave was stored in a ROM, which saved about 82.5% of the memory resources required for FFT twiddle factors. The same concept was repeated in Hamming window factors and DCT factors in case of LUT implementation. Also, RAM buffers were re-used between components, like FFT, Mel-filter and DCT, to exploit the serial nature of their operation.

Finally, time performance was also improved by using the inherent feature of the on-chip memories, which is the dual-port memory operation, which reduce the access time by 50% in case of FFT butterflies. Also, time was improved by parallel operation of some modules, like the FFT last stage and the magnitude calculation of the FFT output, and the operation of the Mel-filter the non-linear (LOG) operation following it, and finally the DCT operation and the Vector Quantizer. This time optimization makes the time taken by the front-end algorithm between 0.8 to 1.8% of the allowed frame processing time, leaving the rest of the time to the remaining back-end recognition task.

7.2 Recommendations for Future Work

The next versions of the Aurora standard can be implemented. These versions use the same basic core implemented here, so implementing any of these version will be a feature addition to the current design. It is highly recommended to implement the noise robust feature in the Advanced Front-end (AFE) system, and test its performance in noisy environments; this is because the DSR system is intended to be deployed in mobile devices, which are operated in noisy environments.

The design presented was verified to the RTL level only, which can be extended to be tested to the gate level, and downloaded to real FPGA chip. Also, the design could be ported to other styles, like DSP processors for example to compare the performance on both platforms. Migration to ASIC style is highly recommended for future implementations to reduce cost and power consumption.

Power consumption measurement and optimization is still needed to be completed for the current design.

8 References

[1] Xudding Huang, Alex Acero, Hsiao-Wuen Hon, " Spoken Language Processing, A guide to Theory, Algorithm, and System Development ", Prentice Hall, 2001.

[2] European Telecommunications Standard Institute, ETSI, " Speech Processing, Transmission and Quality Aspects (STQ); Distributed speech recognition; Front-end feature extraction algorithm; Compression algorithms ", Aurora Standard, ETSI ES 201 108 V1.1.3 (2003-09).

[3] European Telecommunications Standard Institute, ETSI, " Speech Processing, Transmission and Quality Aspects (STQ); Distributed speech recognition; Extended Front-end feature extraction algorithm; Compression algorithms; Back-end speech reconstruction algorithm ", Aurora Standard, ETSI ES 202 211 V1.1.1 (2003-11).

[4] European Telecommunications Standard Institute, ETSI, “Speech Processing, Transmission and Quality Aspects (STQ); Distributed speech recognition; Advanced front-end feature extraction algorithm; Compression algorithms”, Aurora Standard, ETSI ES 202 050 V1.1.5 (2007-1).

[5] European Telecommunications Standard Institute, ETSI, “Speech Processing, Transmission and Quality Aspects (STQ); Distributed speech recognition; Extended advanced front-end feature extraction algorithm; Compression algorithms; Back-end speech reconstruction algorithm”, Aurora Standard, ETSI ES 202 212 V1.1.2 (2005-11).

[6] *www.wikipedia.org*, 12/31/2008 9:29 PM.

[7] Dmitry Zaykovskiy, " Survey of the Speech Recognition Techniques for Mobile Devices ", in Proc. SPECOM 2006, 11-th International Conference on Speech and Computer, St. Petersburg, Russia, 25-29 June 2006.

[8] Md. Rashidul Hasan, Mustafa Jamil, Md. Golam Rabbani Md. Saifur Rahman, “Speaker Identification using Mel-Frequency Cepstral Coefficients”, Proceedings of the Third international Conference on Electrical and Computer Engineering, ICECE 2004, Dhaka, Bangladesh, December, 2004.

[9] David Pearce, " Enabling New Speech Driven Services for Mobile Devices: An overview of the ETSI standards activities for Distributed Speech Recognition Front-ends ", in Proc. AVIOS 2000, The Speech Applications Conference, San Jose, CA, USA, May 22-24, 2000.

[10] European Telecommunications Standard Institute, ETSI, "New Aurora Activity for Standardization of a Front-End Extension for Tonal Language Recognition and Speech Reconstruction", ETSI DSR Applications and Protocols Working Group Notes, June 2001, http://portal.etsi.org/stq/kta/DSR/Au33501_DSR_reconstruction.pdf, 4/20/2009 2:24:53 PM.

[11] Sen M Kuo, Bob H Lee and Wenshun Tian, " Real-Time Digital Signal Processing, Implementations and Applications ", Second Edition, John Wiley and Sons, 2001.

[12] Signal Processing Laboratory of Swiss Federal Institute of Technology, *"Design of VLSI Systems"*, webcourse; http://lsiwww.epfl.ch/LSI2001/teaching/webcourse/toc.html (12/28/2008 8:14 PM).

[13] Namballa, R.; Ranganathan, N.; Ejnioui, A, "Control and Data Flow Graph Extraction for High-Level Synthesis", Proceedings. IEEE Computer society Annual Symposium on VLSI, 2004, 19-20 Feb. 2004.

[14] Brian Dipert, "Moving beyond programmable logic: if, when, how?", EDN Access Magazine, November 20, 1997, http://www.edn.com/archives/1997/112097/24df_02.htm, (4/20/2009 2:29 PM).

[15] Dmitry Zaykovskiy, " Survey of the Speech Recognition Techniques for Mobile Devices ", in Proc. SPECOM 2006, 11-th International Conference on Speech and Computer, St. Petersburg, Russia, 25-29 June 2006.

[16] Teemu Pitkänen, "VHDL Test Benches", Tampere University of Technology, Institute of Digital and Computer Systems.

[17] Altera, " CORDIC reference design ", Altera Application Note, AN: 263, www.altera.com/products/ip/dsp/ipm-index.jsp, June 2005

[18] Altera, “DFPDIV Floating-Point Pipelined Divider Unit”, Application Note, www.altera.com/products/ip/dsp/ipm-index.jsp, 01/01/2009 16:37

[19] Altera, " FFT MegaCore function User Guide ", Altera MegaCore documentation and user guides, UG-FFT-7.0, www.altera.com/products/ip/dsp/ipm-index.jsp, November 2008

[20] Xilinx, “Fast Fourier Transform v5.0”, Xilinx application notes, Digital Signal Processing, Xilinx product specification, DS260 October 10, 2007, www.xilinx.com 01/01/2009 17:16.

[21] Shousheng He and Mats Torkelson, “A New Approach to Pipeline FFT Processor”, Parallel Processing Symposium, 1996., Proceedings of IPPS '96, The 10th International, 15-19 April 1996.

9 Appendix

Some useful algorithms and concepts

In this Appendix, some useful algorithms and concepts that were used in the design, like CORDIC algorithm, Radix-2 FFT and Fixed and Floating point concepts and notations.

1 CORDIC Algorithm

It is an efficient hardware algorithm based on iterative numerical method to compute a wide range of functions including certain trigonometric, hyperbolic, linear and logarithmic functions. For more information about CORDIC algorithm, mathematical derivations, equations mentioned in this section, please refer to [15].

The trigonometric functions are based on vector rotations, while other functions such as square root are implemented using an incremental expression of the desired function. The trigonometric algorithm is called CORDIC; CORDIC is an acronym of COordinate Rotation DIgital Computer. The incremental functions are performed with a very simple extension to the hardware architecture, and while not CORDIC in the strict sense, are often included because of the close similarity.

The CORDIC algorithm generally produces one additional bit of accuracy for each iteration. The main advantage of the algorithm is that it permits to compute those functions, that are widely used in DSP application through a set of shift-add operations, which permits to implement them only using shift registers, adders and Look-up tables (LUT), which highly reduces the hardware complexity of the implementation.

The trigonometric CORDIC algorithms were originally developed as a digital solution for real-time navigation problems. The original work is credited to Jack Volder. Extensions to the CORDIC theory based on work by John

Walther and others provide solutions to a broader class of functions. The CORDIC algorithm has found its way into diverse applications including the 8087 math coprocessor, the HP-35 calculator, radar signal processors and robotics. CORDIC rotation has also been proposed for computing Discrete Fourier, Discrete Cosine, Discrete Hartley and Chirp-Z transforms, filtering, Singular Value Decomposition, and solving linear systems [15].

1.1 Basic Theory of the Algorithm

All of the trigonometric functions can be computed or derived from functions using vector rotations, as will be discussed in the following sections. Vector rotation can also be used for polar to rectangular and rectangular to polar conversions, for vector magnitude, and as a building block in certain transforms such as the DFT and DCT. The CORDIC algorithm provides an iterative method of performing vector rotations by arbitrary angles using only shifts and adds. The algorithm, credited to Volder [15], is derived from the general rotation transform:

$$\begin{aligned} x' &= x\cos\phi - y\sin\phi \\ y' &= y\cos\phi + x\sin\phi \end{aligned}$$

Which rotates a vector in a Cartesian plane by the angle Φ. These can be rearranged so that:

$$\begin{aligned} x' &= \cos\phi \cdot \left[x - y\tan\phi\right] \\ y' &= \cos\phi \cdot \left[y + x\tan\phi\right] \end{aligned}$$

So far, nothing is simplified. However, if the rotation angles are restricted so that $\tan(f)=\pm 2^{-i}$, the multiplication by the tangent term is reduced to simple shift operation. Arbitrary angles of rotation are obtainable by performing a series of successively smaller elementary rotations. If the decision at each iteration, i, is which direction to rotate rather than whether or not to

rotate, then the cos(δi) term becomes a constant (because cos(δi) = cos(-δi)). The iterative rotation can now be expressed as:

$$x_{i+1} = K_i\left[x_i - y_i \cdot d_i \cdot 2^{-i}\right]$$
$$y_{i+1} = K_i\left[y_i + x_i \cdot d_i \cdot 2^{-i}\right]$$

Where:

$$K_i = \cos\left(\tan^{-1} 2^{-i}\right) = 1/\sqrt{1+2^{-2i}}$$
$$d_i = \pm 1$$

Removing the scale constant from the iterative equations yields a shift-add algorithm for vector rotation. The product of the Ki's can be applied elsewhere in the system or treated as part of a system processing gain. That product approaches 0.6073 as the number of iterations goes to infinity. Therefore, the rotation algorithm has a gain, An, of approximately 1.647. The exact gain depends on the number of iterations, and obeys the relation

$$A_n = \prod_n \sqrt{1+2^{-2i}}$$

The angle of a composite rotation is uniquely defined by the sequence of the directions of the elementary rotations. That sequence can be represented by a decision vector. The set of all possible decision vectors is an angular measurement system based on binary arctangents. Conversions between this angular system and any other can be accomplished using a look-up. A better conversion method uses an additional adder-subtractor that accumulates the elementary rotation angles at each iteration. The elementary angles can be expressed in any convenient angular unit. Those angular values are supplied by a small lookup table (one entry per iteration) or are hardwired, depending on the implementation. The angle accumulator adds a third difference equation to the CORDIC algorithm:

$$z_{i+1} = z_i - d_i \cdot \tan^{-1}\left(2^{-i}\right)$$

The CORDIC rotator is normally operated in one of two modes. The first, called rotation rotates the input vector by a specified angle (given as an argument). The second mode, called vectoring, rotates the input vector to the x-axis while recording the angle required making that rotation.

1.1.1 Rotation mode

In rotation mode, the angle accumulator is initialized with the desired rotation angle. The rotation decision at each iteration is made to diminish the magnitude of the residual angle in the angle accumulator. The decision at each iteration is therefore based on the sign of the residual angle after each step. Naturally, if the input angle is already expressed in the binary arctangent base, the angle accumulator may be eliminated. For rotation mode, the CORDIC equations are:

$$\begin{aligned} x_{i+1} &= x_i - y_i \cdot d_i \cdot 2^{-i} \\ y_{i+1} &= y_i + x_i \cdot d_i \cdot 2^{-i} \\ z_{i+1} &= z_i - d_i \cdot \tan^{-1}\left(2^{-i}\right) \end{aligned}$$

Where:

$$d_i = -1 \text{ if } z_i < 0, \ +1 \text{ otherwise}$$

This provides the following result:

$$\begin{aligned} x_n &= A_n\left[x_0 \cos z_0 - y_0 \sin z_0\right] \\ y_n &= A_n\left[y_0 \cos z_0 + x_0 \sin z_0\right] \\ z_n &= 0 \\ A_n &= \prod_n \sqrt{1 + 2^{-2i}} \end{aligned}$$

1.1.2 Vectoring mode

In the vectoring mode, the CORDIC rotator rotates the input vector through whatever angle is necessary to align the result vector with the x axis. The result of the vectoring operation is a rotation angle and the scaled magnitude of the original vector (the x component of the result). The vectoring function works by seeking to minimize the y component of the residual vector at each rotation. The sign of the residual y component is used to determine which direction to rotate next. If the angle accumulator is initialized with zero, it will contain the traversed angle at the end of the iterations. In vectoring mode, the CORDIC equations are:

$$x_{i+1} = x_i - y_i \cdot d_i \cdot 2^{-i}$$
$$y_{i+1} = y_i + x_i \cdot d_i \cdot 2^{-i}$$
$$z_{i+1} = z_i - d_i \cdot \tan^{-1}\left(2^{-i}\right)$$

Where:

$$d_i = +1 \text{ if } y_i < 0, \text{ -1 otherwise.}$$

Then:

$$x_n = A_n \sqrt{x_0^2 + y_0^2}$$
$$y_n = 0$$
$$z_n = z_0 + \tan^{-1}\left(y_0 / x_0\right)$$
$$A_n = \prod_n \sqrt{1 + 2^{-2i}}$$

The CORDIC rotation and vectoring algorithms as stated are limited to rotation angles between -pi/2 and pi/2. This limitation is due to the use of 2^0 for the tangent in the first iteration. For composite rotation angles larger than pi/2, an additional rotation is required. Volder describes an initial rotation ±pi/2. This gives the correction iteration:

$$x' = -d \cdot y$$
$$y' = d \cdot x$$
$$z' = z + d \cdot \pi/2$$

Where:

$$d = +1 \text{ if } y<0, -1 \text{ otherwise.}$$

There is no growth for this initial rotation. Alternatively, an initial rotation of either pi or 0 can be made, avoiding the reassignment of the x and y components to the rotator elements. Again, there is no growth due to the initial rotation:

$$x' = d \cdot x$$
$$y' = d \cdot y$$
$$z' = z \text{ if } d = 1, \text{ or } z - \pi \text{ if } d = -1$$
$$d = -1 \text{ if } x<0, +1 \text{ otherwise.}$$

The CORDIC rotator described is usable to compute several trigonometric functions directly and others indirectly. Judicious choice of initial values and modes permits direct computation of sine, cosine, arctangent, vector magnitude and transformations between polar and Cartesian coordinates. The following sections present some of these functions that are utilized in the design.

1.1.2.1 Sine and Cosine

The rotational mode CORDIC operation can simultaneously compute the sine and cosine of the input angle. Setting the y component of the input vector to zero reduces the rotation mode result to:

$$x_n = A_n \cdot x_0 \cos z_0$$
$$y_n = A_n \cdot x_0 \sin z_0$$

By setting x0 equal to 1/An, the rotation produces the unscaled sine and cosine of the angle argument, z0.

1.1.2.2 The Fast Fourier Transform (FFT)

A DFT with N input values s can be described as the matrix vector multiplication

$$\mathbf{S} = \mathbf{V} \cdot \mathrm{s} \qquad \text{with} \quad \mathbf{V}_{nk} = e^{-\mathrm{j}\frac{2\pi}{N}nk}$$

By exploiting the properties of V the operations can be greatly reduced, and the well known Fast Fourier Transformation (FFT) is derived. An eight point FFT leads to the network shown in Figure 1. The twiddle factors are derived as:

$$\omega_y^x = \mathrm{e}^{-\mathrm{j}\frac{2\pi x}{y}}$$

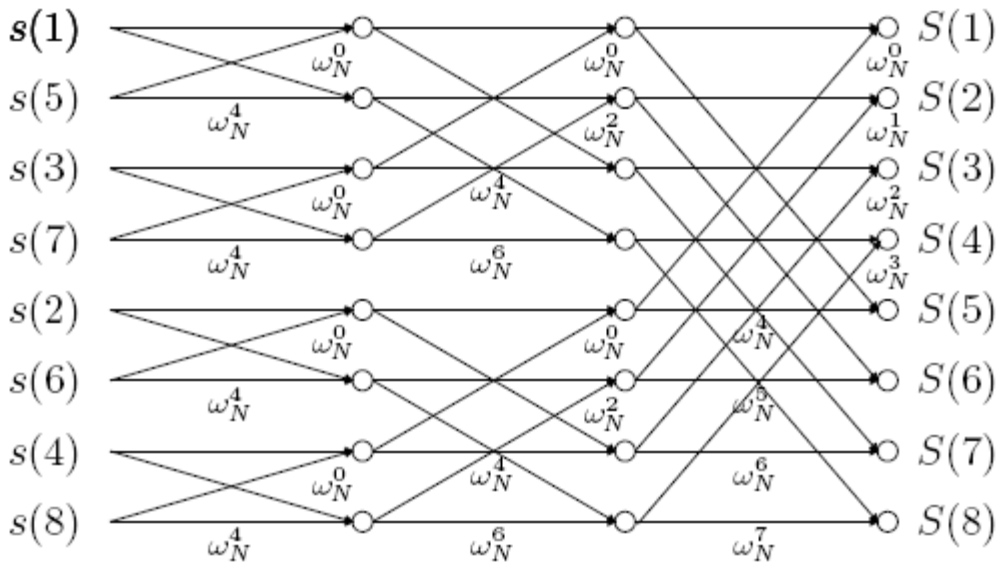


Figure 87: 8-point FFT Network

The multiplication by the twiddle factor is equivalent to rotation by the angle (2*pi*x/y). This can be easily implemented using the CORDIC algorithm in its basic vector rotation mode.

1.1.2.3 Vector magnitude

The vectoring mode CORDIC rotator produces the magnitude of the input vector as a byproduct of computing the arctangent. After the vectoring mode rotation, the vector is aligned with the x-axis. The magnitude of the vector is therefore the same as the x-component of the rotated vector. This result is apparent in the result equations for the vector mode rotator:

$$x_n = A_n \sqrt{x_0^2 + y_0^2}$$

The magnitude result is scaled by the processor gain, which needs to be accounted for elsewhere in the system. This implementation of vector magnitude has a hardware complexity of roughly one multiplier of the same width. The accuracy of the magnitude result improves by 2 bits for each iteration performed. The same vectoring can be directly used to get the Cartesian to Polar transformation of a vector, where final xn is the magnitude of the vector multiplied by An, and zn is the angle between x0 and y0, such that tan(zn) = y0/x0.

1.1.3 Extension to Linear functions- Multipliers and Dividers:

A simple modification to the CORDIC equation permits the computation of linear functions:

$$x_{i+1} = x_i - 0 \cdot y_i \cdot d_i \cdot 2^{-i} = x_i$$
$$y_{i+1} = y_i + x_i \cdot d_i \cdot 2^{-i}$$
$$z_{i+1} = z_i - d_i \cdot \left(2^{-i}\right)$$

For rotation mode (di= -1 if zi < 0, +1 otherwise) the linear rotation produces:

$$x_n = x_0$$
$$y_n = y_0 + x_0 z_0$$
$$z_n = 0$$

This operation is similar to the shift-add implementation of a multiplier, and as multipliers go is not an optimal solution. The multiplication is handy in applications where a CORDIC structure is already available.

The vectoring mode (di= +1 if yi < 0, -1 otherwise) is more interesting, as it provides a method for evaluating ratios (CORDIC Divider):

$$x_n = x_0$$
$$y_n = 0$$
$$z_n = z_0 - y_0 / x_0$$

The rotations in the linear coordinate system have a unity gain, so no scaling corrections are required.

1.1.4 Extension to Hyperbolic functions- Natural Logarithm:

The close relationship between the trigonometric and hyperbolic functions suggests the same architecture can be used to compute the hyperbolic functions. While, there is early mention of using the CORDIC structure for hyperbolic coordinate transforms, the first description of the algorithm is that by Walther [15]. The CORDIC equations for hyperbolic rotations are derived

using the same manipulations as those used to derive the rotation in the circular coordinate system. For rotation mode these are:

$$x_{i+1} = x_i + y_i \cdot d_i \cdot 2^{-i}$$
$$y_{i+1} = y_i + x_i \cdot d_i \cdot 2^{-i}$$
$$z_{i+1} = z_i - d_i \cdot \tanh^{-1}\left(2^{-i}\right)$$

Where:

$$d_i = -1 \text{ if } z_i < 0, \ +1 \text{ otherwise.}$$

Then:

$$x_n = A_n\left[x_0 \cosh z_0 + y_0 \sinh z_0\right]$$
$$y_n = A_n\left[y_0 \cosh z_0 + x_0 \sinh z_0\right]$$
$$z_n = 0$$
$$A_n = \prod_n \sqrt{1 - 2^{-2i}} \approx 0.80$$

In vectoring mode (di= +1 if yi < 0, -1 otherwise) the rotation produces:

$$x_n = A_n \sqrt{x_0^2 - y_0^2}$$
$$y_n = 0$$
$$z_n = z_0 + \tanh^{-1}\left(y_0 / x_0\right)$$
$$A_n = \prod_n \sqrt{1 - 2^{-2i}}$$

The elemental rotations in the hyperbolic coordinate system do not converge. However, it can be shown that convergence is achieved if certain iterations (I=4, 13, 40... k, 3k+1...) are repeated.

The hyperbolic equivalents of all the functions discussed for the circular coordinate system can be computed in a similar fashion. Additionally, as

Walther points out, the following functions can be derived from the CORDIC functions:

$\tan\alpha = \sin\alpha/\cos\alpha$

$\tanh\alpha = \sinh\alpha/\cosh\alpha$

$\exp\alpha = \sinh\alpha + \cosh\alpha$

$\ln\alpha = 2\tanh^{-1}[y/x]$ where $x=\alpha+1$ and $y=\alpha-1$

$(\alpha)^{1/2} = (x^2-y^2)^{1/2}$ where $x=\alpha+1/4$ and $y=\alpha-1/4$

This will be useful when calculating the Natural Logarithm (Ln), which is encountered twice in the system, the first time, is calculating the Log of the Energy. The second time is when calculating the Natural logarithm of the output of the Mel-Filter.

1.2 General Hardware Implementation of the CORDIC Processor:

Figure 88 shows general hardware architecture of the CORDIC processor in its basic form:

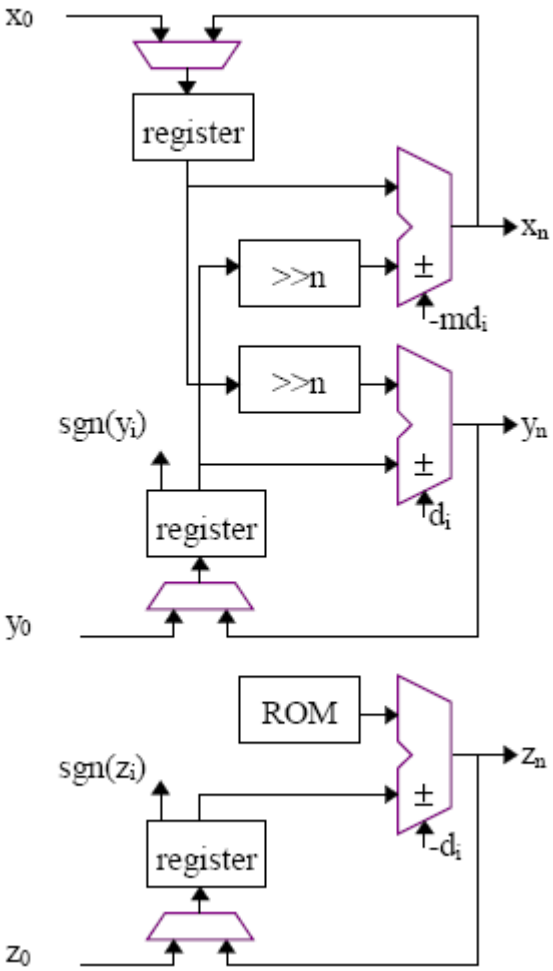


Figure 88: Iterative CORDIC Processor [15]

2 The Fast Fourier Transform (FFT)

There is a family of fast algorithms to compute the Discrete Fourier Transform (DFT), which is called Fast Fourier Transform (FFT). Direct computation of DFT follows the equation:

$$X[k] = \sum_{n=0}^{N-1} x[n] e^{-j2\pi nk/N} \qquad 0 \le k \le N$$

Which requires N^2 operations, assuming that the trigonometric functions have been pre-computed. The FFT algorithm only requires $N \log_2 N$ operations, so it is widely used for speech recognition tasks.

2.1 Radix-2 FFT

There are many algorithms to compute the Fast Fourier Transform. In our design, we adopted the Radix-2 FFT algorithm, due to its simplicity and suitability to the timing and resources requirements of the design. For more information about the algorithm, mathematical derivations, equations mentioned in this section, please refer to [1].

2.1.1 Mathematical derivation

Let's express the discrete Fourier transform of x[n] as:

$$X[k] = \sum_{n=0}^{N-1} x[n] e^{-j2\pi nk/N} = \sum_{n=0}^{N-1} x[n] W_N^{nk} \qquad 0 \le k \le N$$

Where we have defined the Twiddle Factor W_N^k as:

$$W_N^k = e^{-j2\pi k/N}$$

Let's suppose that N is even, and let f[n] = x[2n] represent the even-indexed samples of x[n], and g[n] = x[2n+1] the odd-indexed samples of x[n], so:

$$X[k] = \sum_{n=0}^{N-1} f[n] W_{N/2}^{nk} + W_N^k \sum_{n=0}^{N-1} g[n] W_{N/2}^{nk} = F[k] + W_N^k G[k]$$

Where F[k] and G[k] are the N/2 point DFTs of f[n] and g[n], respectively. Since both F[k] and G[k] are defined for $0 \le k \le N/2$, we need to also evaluate them for $N/2 \le k \le N$, which is straight forward, using the periodicity and symmetric properties of the DFT:

$$F[k + N/2] = F[k]$$

$$G[k + N/2] = G[k]$$

If N/2 is also even, then both f[n] and g[n] can be decomposed into sequences of even and odd indexed samples and therefore its DFT can be computed using the same process. Furthermore, if N is an integer power of 2, this process can be iterated and it can be shown that the number of multiplies and adds is N $\log_2$N, which is a significant saving from N^2 . This is called *decimation in time*. A dual algorithm called *decimation in frequency* can be derived by decomposing the signal into its first N/2 and last N/2 samples.

2.1.2 Algorithm Implementation

A graphical representation of the radix-2 algorithm is shown in Figure 89. This algorithm optimizes the memory usage by using only one buffer of depth equal N in all the steps of the algorithm, where calculations are done and restored in their places again in the buffer. At the end of the N $\log_2$N operations of the algorithm, the same input buffer that contained the input samples will have the result samples. This is why the algorithm is called *in-place* radix-2 algorithm.

The algorithm is composed of $\log_2$N stages, with N operations taking place at each stage, as shown in Figure 89.

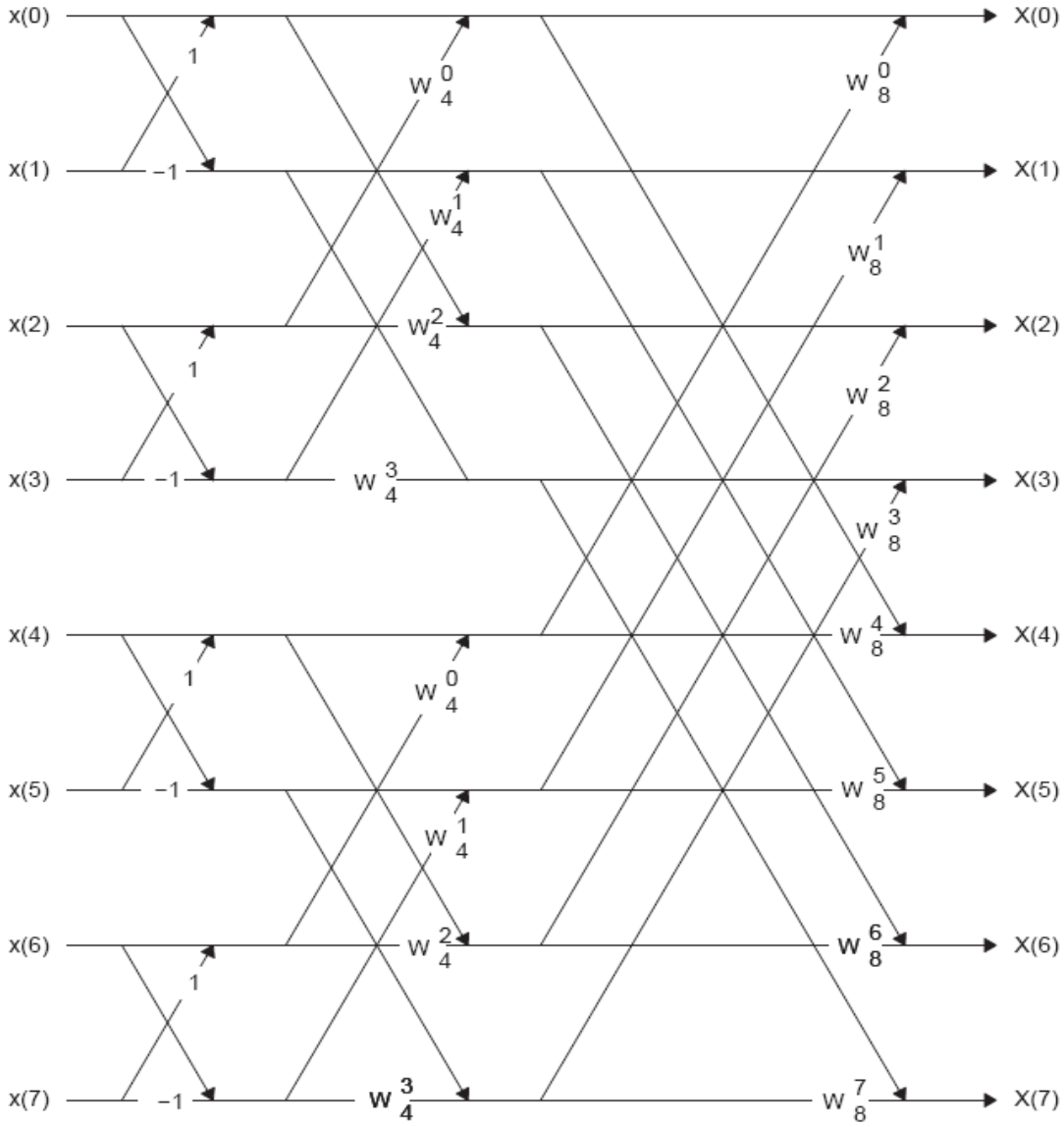


Figure 89: the 8 point Decimation In Time (DIT) Radix-2 FFT algorithm

There are three four basic operations that take place in the algorithm:

1. Bit reversal
2. Butter fly
3. Twiddle factor calculation

This will be presented in the following sections

2.1.2.1 Bit Reversal

The first step of the algorithm is to order the samples in the buffer in a certain order called bit-reversed order, where the destination index of the input sample in the buffer is the result of reversing the binary equivalent representation of the source index. The following figure illustrates this operation:

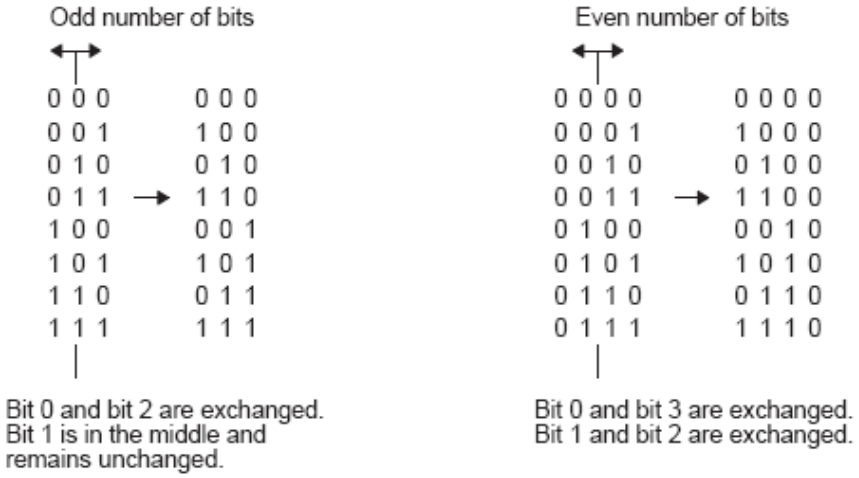


Figure 90: Bit Reversal operation

2.1.2.2 Butterfly

This operation is repeated N/2 times in every iteration of the $\log_2 N$ stages of the algorithm. The basic butter fly operation is shown in **Figure 91**, where it takes 2 samples as an input, and produces 2 new result samples to be placed in the same location of the input samples in the buffer. The addresses of the input samples are generated according to a given pattern that depends on which stage the algorithm is in, as illustrated in **Figure 90**. The twiddle factors

(W_N^i) used in the butter fly operation depend on the addresses of the input samples and the stage of the algorithm.

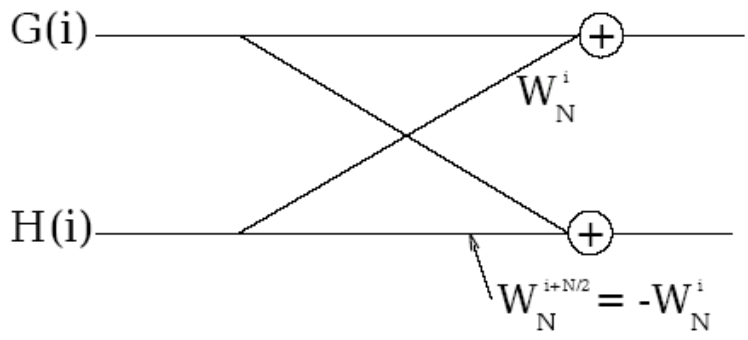


Figure 91: Basic Butter fly operation

The above operation can be reordered as:

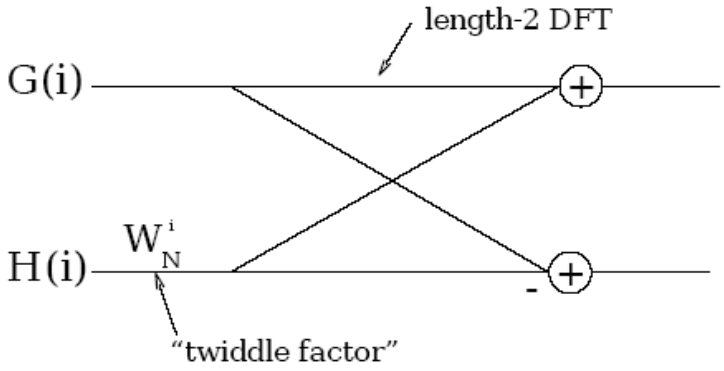


Figure 92: Reordered Butter fly operation

Which enables the usage of basic 2-points FFT, after multiplying H(i) by the proper twiddle factor as shown. Hence, the new shape of the algorithm will be:

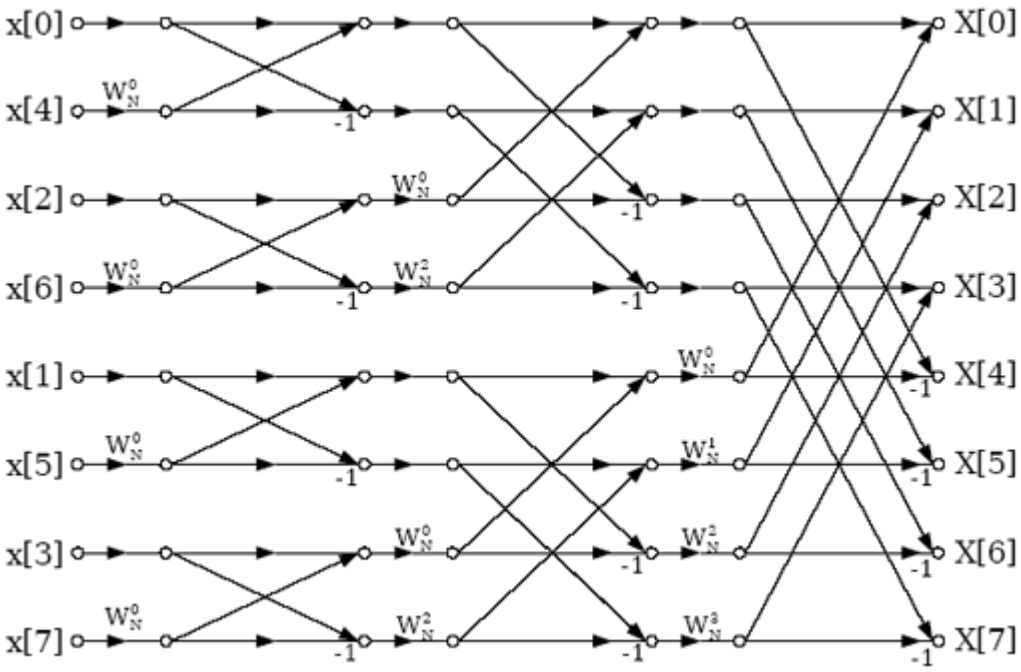


Figure 93: Reordered radix-2 DIT In place FFT algorithm

2.1.2.3 Twiddle factor calculation

The twiddle factor W_N^k is defined as:

$$W_N^k = e^{-j2\pi k/N}$$

Which is equivalent to vector rotation with angle $2\pi k/N$, which enables to use the CORDIC algorithm in its rotation mode as described earlier in this Appendix.

2.2 Other FFT Algorithms

Although radix-2 FFT is the best known algorithm, there are other variants. Among those are the radix-4, radix-8, split-radix and prime factor algorithm. Next information is obtained from [1].

The same process used in the derivation of the radix-2 decimation in time algorithm applies if we decompose the sequences into four sequences: f1[n] = x[4n], f2[n] = x[4n+1], f3[n] = x[4n+2], and f4[n] = x[4n+3]. This is the radix-4 algorithm, which can be applied when N is a power of 4.

Similarly, there are radix-8 and radix-16 algorithms for N being powers of 8 and 16 respectively. These algorithms use fewer adders and multiplies then the famous radix-2 algorithm, however, they add extra constraints and additional control logic, which makes them not necessarily faster than the radix-2 equivalent, and need to be customized to a given processor.

Some values of N cannot use radix-4, radix-8 or radix-16. A combination of radix-2 and radix-4 is called split-radix can be applied to N being a power of 2.

Finally, another possible decomposition is N=p1p2…pl with pi being prime numbers. This leads to the prime-factor algorithm. While this family of algorithms offers a similar number of operations as the algorithms above, it offers more flexibility in the choice of N.

3 Concept of Fixed and Floating Point Arithmetic

The basic element in digital hardware is the binary device that contains one bit of information. A register (or memory unit) containing B bits of information is called a B-bit word. There are several different methods for representing numbers and carrying out arithmetic operations. The most famous among those ways are the fixed and floating point representations. Both representations are given in the following sections, with more emphasis on the fixed point notation. Information in this section is obtained from [6]

3.1 Floating Point

The term floating point refers to the fact that the radix point (decimal point, or, more commonly in computers, binary point) can "float": that is, it can be placed anywhere relative to the significant digits of the number. This position is indicated separately in the internal representation, and floating-point representation can thus be thought of as a computer realization of scientific notation.

In scientific notation, the given number is scaled by a power of 10 so that it lies within a certain range – typically between 1 and 10, with the radix point

appearing immediately after the first digit. The scaling factor, as a power of ten, is then indicated separately at the end of the number. For example, the revolution period of Jupiter's moon is 152853.5047 seconds. This is represented in standard-form scientific notation as 1.528535047×105 seconds. oating-point representation is similar in concept to scientific notation. Logically, a floating-point number consists of:

1. A signed digit string of a given length in a given base (or radix). This is known as the significand, or sometimes the mantissa or coefficient. The radix point is not explicitly included, but is implicitly assumed to always lie in a certain position within the significand – often just after or just before the most significant digit. The length of the significand determines the precision to which numbers can be represented.
2. A signed integer exponent, also referred to as the characteristic or scale, which indicates the actual magnitude of the number.

The significand is multiplied by the base raised to the power of the exponent, equivalent to shifting the radix point from its implied position by a number of places equal to the value of the exponent — to the right if the exponent is positive or to the left if the exponent is negative. Using base-10 (the familiar decimal notation) as an example, the number 152853.5047, with ten decimal digits of precision, is represented as the significand 1528535047 together with an exponent of 5. To recover the actual value, a decimal point is placed after the first digit of the significand and the result is multiplied by 105 to give 1.528535047 × 105, or 152853.5047.

Symbolically, this final value is

$$s \times b^e$$

Where s is the value of the significand (after taking into account the implied radix point), b is the base, and e is the exponent. Equivalently, this is:

$$\frac{s}{b^{p-1}} \times b^e$$

Where s here means the integer value of the entire significand, and p is the precision: the number of digits in the significand. The significand always stores the most significant digits in the number: the first non-zero digits. When the significand is adjusted in this way so that its leftmost digit is nonzero, it is said to be normalized, and its value obeys $1 \leq s < b$, given that the radix point is assumed to follow the first digit. Zero is a special case and is normally represented as $s = 0$, $e = 0$. (Subnormal numbers and certain other cases also need special treatment; see dealing with exceptional cases.)

Floating point arithmetic has always been very costly in terms of resources needed to implement or processing time required, specially when dealing with hardware implementations, despite the simplicity of developing applications using this type of mathematical representation. This leads to the fixed point notation.

3.2 Fixed Point

The most commonly used fixed-point representation of a fractional number x is illustrated in Figure 94. The word length is B(= M + 1) bits, i.e., M magnitude bits and one sign bit. The most significant bit (MSB) is the sign bit, which represents the sign of the number as follows:

$$b_0 = \begin{cases} 0, & x \geq 0 \quad \text{(positive number)} \\ 1, & x < 0 \quad \text{(negative number)} \end{cases}.$$

In the following figure, the fixed point number representation is illustrated.

$$(x)_{10} = \sum_{m=1}^{15} 2^{-m} = 2^{-1} + 2^{-2} + \cdots + 2^{-15}$$
$$= 1 - 2^{-15} \approx 0.999969.$$

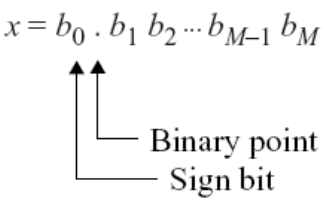


Figure 94: Fixed point representation of binary fractional numbers [11]

The remaining M bits give the magnitude of the number. The rightmost bit bM is called the least significant bit (LSB), which represents precision of the number.

As shown in the following figure, the decimal value of a positive (b0 = 0) binary fractional number x can be expressed as:

$$(x)_{10} = b_1 \cdot 2^{-1} + b_2 \cdot 2^{-2} + \cdots + b_M \cdot 2^{-M}$$
$$= \sum_{m=1}^{M} b_m 2^{-m}.$$

In general, the decimal value of a B-bit binary fractional number can be calculated as:

$$(x)_{10} = -b_0 + \sum_{m=1}^{15} b_m 2^{-m}$$

In general, according to the dynamic range of the number, there can be more than one bit (b0) to represent the non-fractional part of the number in addition to the sign bit, this will be pointed to as the *Integer part* of the number

through out this thesis, and the bits after the decimal point (b1 to bM in the above example) will be noted as the *fractional part.*

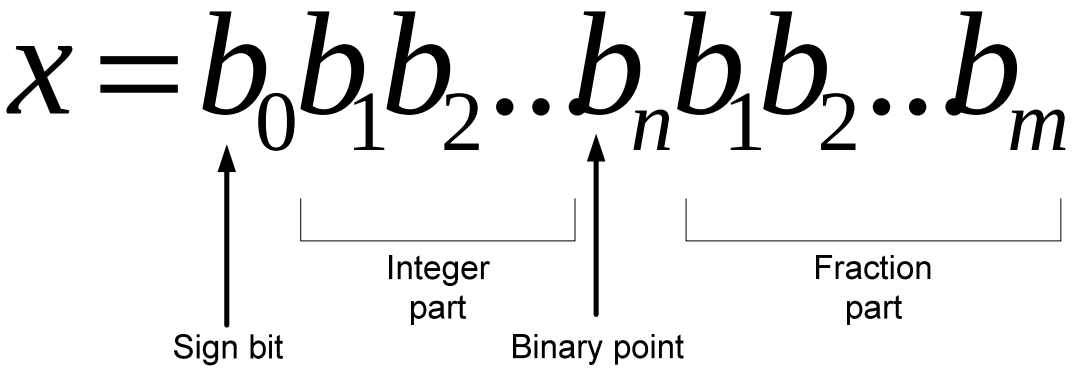


Figure 95: A general binary fractional number

In this case, the conversion to decimal equation should be modified such that the final number is the sum of the decimal equivalent of the integer part and the fraction part, where the integer part bits will be multiplied by 2 raised to positive powers according to the index of the bit, while the fraction part bits will be multiplied by 2 raised to negative powers according to their index too.

In general, conversion can be easily done by dividing the decimal equivalent of the binary number by the maximum of the fraction part. For example, if the fraction part is 10 bits, then its maximum value is 1024, then the decimal equivalent is obtained by dividing the fixed point number in decimal by 1024, which will give the original number.