Hardware Optimization of the Front End Engine in an Automatic Speech Recognition System

A THESIS

submitted by

RADHIKA R

for the award of the degree

of

MASTER OF TECHNOLOGY

DEPARTMENT OF ELECTRICAL ENGINEERING
INDIAN INSTITUTE OF TECHNOLOGY MADRAS.

MAY 2019

THESIS CERTIFICATE

This is to certify that the thesis titled **Hardware Optimization of the Front End Engine in an Automatic Speech Recognition System**, submitted by **RADHIKA R**, to the Indian Institute of Technology, Madras, for the award of the degree of **Master of Technology**, is a bona fide record of the research work done by her under our supervision. The contents of this thesis, in full or in parts, have not been submitted to any other Institute or University for the award of any degree or diploma.

Dr. Janakiraman Viraraghavan
Project Guide
Assistant Professor
Department of Electrical Engineering
IIT Madras, 600 036

Place: Chennai

Date: 5th May 2019

ACKNOWLEDGEMENTS

First and foremost, I sincerely acknowledge the guidance given by my Project Guide - Dr.Janakiraman Viraraghavan, without which, I could not have envisaged this project. The constant encouragement and support given by him, as also the timely inputs and suggestions made by him, helped me stay positive and complete the project successfully.

I am also grateful to Dr.Nitin Chandrachoodan for the invaluable guidance given by him throughout the project. His timely advice helped me resolve the problems faced during the project work.

Further, I would also like to thank my project mates, Mr.Prithvi Raj, Mr.Akhil Reddy and Mr. Arjun for the constant support and encouragement given by them.

Last but not the least I would like to thank my parents who, have always stood by me and provided me with the strength needed for completing this onerous task.

ABSTRACT

The objective of this thesis is to present a fixed point FPGA implementation of the front end of an automatic speech recognition system.

As the front end receives samples which contain redundant information like noise, variation in accent, speaking styles etc, the task is to detect such variations and extract relevant features necessary to perform accurate decoding.

While carrying out its task, the front end does a lot of computations and in the process consumes a lot of power. But, as the front end takes negligible amount of time to execute its task when compared to the back end, it can be slowed down by serializing a few operations or by running it on a slower clock so as to save on the power consumption.

This project seeks to realize the front end of an ASR system in hardware with minimal use of resources and reduced power consumption. Various optimization techniques were experimented to reduce the computational complexity and power consumption. Many operations were serialized and signal processing algorithms were modified to exploit the real nature of the input data. Also, computationally less intensive alternatives to existing signal processing techniques were explored without compromising on the accuracy of detection.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ABBREVIATIONS

IITM	Indian Institute of Technology, Madras
RTFM	Read the Fine Manual
ADC	Analog to Digital Converter
FFT	Fast Fourier Transform
DCT	Discrete Cosine Transform
CMVN	Cepstral Mean and Variance Normalization
FIFO	First In First Out
PSD	Power Spectral Density
TI	Texas Instruments
HMM	Hidden Markhov Model
GMM	Gaussian Mixture Model
WFST	Weighted Finite-State Transducer
FF	Flip Flop
LUT	Look Up Table
FPGA	Field Programmable Gate Arrays

CHAPTER 1

INTRODUCTION

Speech is the most effective channel of communication. It is the expression of thoughts and feelings to other(s) by articulate sounds. Just as proper initiating of the sounds is essential it is also important to receive them in a manner that facilitates its understanding as intended by the originator.

In modern times, there is need to have 'Automated Speech Recognition' systems. They would be of help in aiding visually challenged or hearing impaired people or in communicating with 'Google Maps' while driving or in communicating with digital personal assistants like 'Alexa' or in controlling robots and so on.

In this context, an ASR system can be defined as 'an independent, computer-driven device to transcribe spoken language into readable text in real time.

The ASR seeks to convert sound waves created by our vocal chords, in the process of speaking, to electrical signals, using a microphone. These signals are then processed, using advanced signal processing techniques, to isolate syllables and words. Over time, with the help of machine learning computers learn to understand speech and help us decode spoken words to text.

One of the main challenges in speech recognition is to reduce speech signal variability due to accent, dialect, speaking style and so on. Thus the aim of an ASR system is to recognize in real time all words that are intelligibly spoken, independent of vocabulary size, noise, accent, speaker characteristics etc. This is achieved by reducing the speech signal variability and by converting the ASR relevant spectral information into speech acoustic features.

The speech acoustic features for ASR, such as the Mel Frequency Cepstral Coefficients (MFCCs), use the spectral envelope of the speech short-term spectrum as a starting point for all the succeeding calculations. However, even with the speech variability reduction achieved by such standard speech signal analysis, ASR performance is still adversely affected by noise and other sources of acoustic variability.

The HMM-based recognition system provides much better accuracy and can be made fast by designing it appropriately. It involves a lot of computations and tend to perform better in hardware as compared to software.

A speech Recognition system can either be cloud-based or circuit based. In areas with good internet connectivity, cloud-based systems can be used whereas, in case of slow internet connectivity, hardware speech recognition systems are necessary.

1.1 Motivation

Speech Recognition in hardware necessitates a lot of hard core computations and the requirements increase as researchers identify improved modelling techniques.

Even though a lot of useful accurate algorithms have been developed, the excessive memory requirements and the difficulties in reprogramming digital circuits to keep up with changing algorithms and models, has limited the widespread adoption of hardware speech decoders.

In this project, the attempt is to devise and implement a high-speed speech recognition system in hardware with high accuracy, optimal power consumption and memory usage. It is based on the paper Price *et al.* (2015)

CHAPTER 2

Basic Architecture of Speech Recognition System

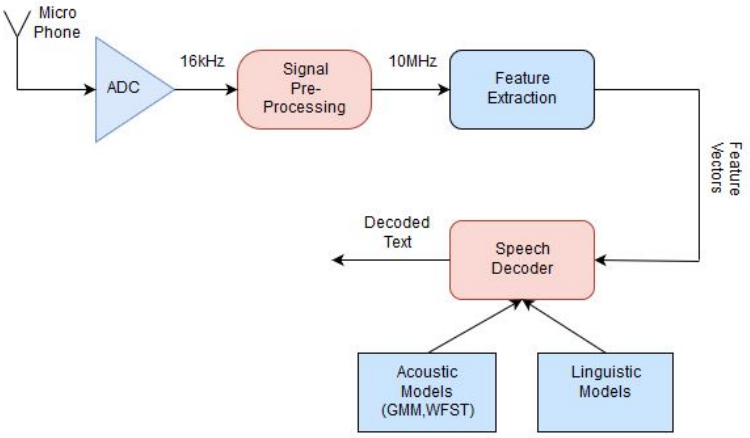

Figure 2.1: A Basic Speech Recognition System

The front end of a speech recognition system is basically a features extraction device. It facilitates removal of the redundancies in the input signal and converts the data into a set of features. The features so extracted by the front end are used by the back end to decode the spoken words into text.

2.1 Front End of an ASR system

The incoming speech signal is sampled using an ADC at a sampling rate of 16KHz. As speech signals more or less occupy a frequency of less than 8kHz, the ADC's sampling rate is fixed at 16kHz. 25ms of data constitute one frame, out of which the first 15ms of data is overlapped from the previous frame for better accuracy in detection.

Each frame is then sent through a series of blocks like windowing, FFT, Mel-bank filtering, Discrete Cosine Transform(DCT), Cepstral liftering, Cepstral mean and variance normalization (CMVN) and so on.

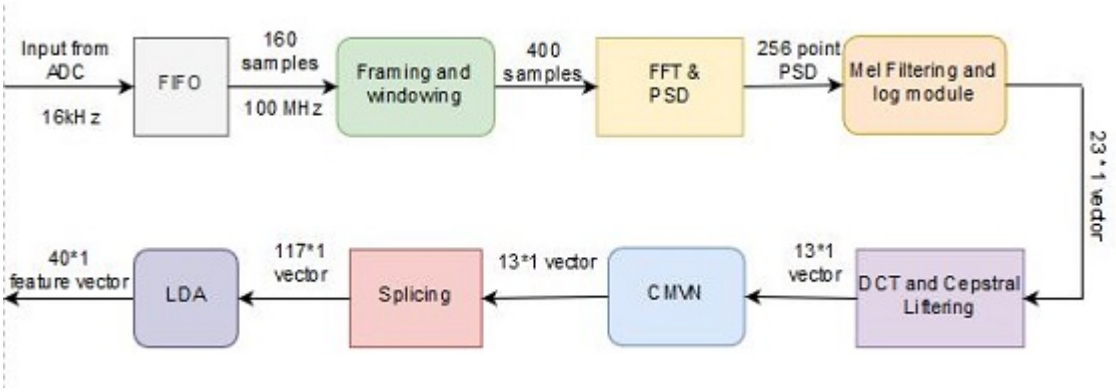

Figure 2.2: Front End of an ASR system

2.1.1 FIFO Block

The function of the FIFO block is to collect samples coming from ADC at 16 kHz and transmit them at 100 MHz.

As mentioned before, 25ms of data coming at 16 kHz constitute one frame which works out to 400 samples in each frame. Out of these 400 samples, the first 240 samples of data are overlapped from the previous frame and the FIFO module facilitates collection of the new 160 samples and transmits them to the framing and windowing module.

2.1.2 Framing and Windowing

In the framing part, the incoming 160 samples are appended to the last 240 samples of the previous frame, to get 400 samples. Each sample is then multiplied by its corresponding Povey window coefficient. To these windowed samples, 112 zeros are padded to get 512 samples. They are then passed on to the FFT Block.

The reason behind the overlap between the frames is that speech is a rapidly changing signal. So for better detection, a few samples from the previous frame are overlapped into the next frame. It also helps to smoothen the transition from one frame to the next frame.

The samples that we get in speech recognition are not periodic signals. But, FFT assumes its input to be one period of a periodic signal, then creates an infinite periodic sequence by using multiple copies of input and further acts on it. But in most cases,

the input won't be periodic due to which there will be discontinuities when an infinite sequence is constructed. As a result, the FFT output will have higher frequency components which, were not actually present in the input signal.

Windowing reduces both ends of the input sequence to almost zero, so that, when an infinite sequence is constructed, it will look like a continuous signal. This will facilitate the reduction of the undesired higher frequency components.

2.1.3 FFT and PSD

After initial signal pre-processing like framing, windowing, etc, are completed, the power spectrum of the frame is calculated. This is done because hearing is based on the vibration of the frequencies on the human cochlea and for detecting speech we have to know how much energy exists at different frequencies.

512 samples are received and a 512 point FFT is performed. FFT exploits the real valued nature of input data to operate on half many points (the FFT output of real data is symmetric). Thus only 256 samples are required to represent the PSD. These 256 points are then passed on to mel-filter banks.

2.1.4 Mel Filter banks

MFCC is a popular feature extraction technique used in speech recognition in frequency domain using the Mel scale, which is based on the human audio scale. It uses audio feature extraction technique to extract parameters from the speech and is similar to those that are used by humans to hear speech. In the process, it, additionally, attenuates all other redundant information like noise, etc.

The human ear cannot differentiate between two closely spaced frequencies, especially at higher frequencies. The concern then is only about how much energy exists in various frequency ranges.

The mel-filter bank which consists of triangular overlapping windows sums up clumps of periodogram bins and gives us an idea of how much energy exists in each frequency region.

The formula for converting from frequency to Mel-Scale is:

$$M(f) = 1125ln(1 + f/700)$$

To compute mel filtering, a set of 23 triangular filters are applied to the periodogram power spectral estimate.

The filter bank comes in the form of 23 vectors of length 257. The power spectrum is multiplied with each filter bank and the coefficients are added up. Once this task is done, we are left with 23 numbers which indicate how much energy is present in each filter bank.

After filtering, logarithm of the energies is taken as the the human perception of sound is non-linear.To perceive a doubling in loudness, the energy that has to be spent might be quadrupled or increased many more times. By taking the logarithm of the energies we actually compress the energies and make them match closely with what humans actually hear. The resultant 23 log energies are then passed on to a DCT module.

2.1.5 Discrete Cosine Transform and Cepstral Liftering

Since the mel-filter banks are overlapping in nature, the filter energies are quite correlated. In order to de-correlate them, a DCT is performed.

DCT ranges coefficients according to their relative significance. The zero'th coefficient is excluded since it is unreliable. It is replaced by the total energy of the signal in the frame, before windowing. The higher DCT coefficients represent fast changes in the data and these can be discarded. Thus from 23 coefficients, only the first 13 are passed on to the CMVN module. Before passing on to CMVN module, cepstral liftering is done to smoothen the log magnitude spectrum. It is a kind of windowing in the cepstral domain.

2.1.6 Cepstral Mean and Variance Normalization

After DCT and cepstral liftering, a channel normalization technique called CMVN is used to minimize the noise in the system. Here the cepstral coefficients are transformed to have the same statistics (zero mean, unit variance).

As this method doesn't require prior knowledge of noise statistics it adapts quickly to changing noise conditions.

The mean, variance and cepstral output are calculated as follows :

$$\text{mean[i]} = ((1 - \text{w}) \times \text{mean[i]}) + (\text{w} \times \text{feat\_in[i]})$$

$$\text{variance[i]} = \text{(1-w)*variance[i]+w*(feat\_in[i]-mean[i])*(feat\_in[i]-mean[i])} \quad (2.1)$$

$$\text{feat\_out[i]} = \frac{\text{(feat\_in[i] - mean[i])}}{\text{variance[i]}} \quad (2.2)$$

where w=0.002, feat_in is the input to the CMVN module and feat_out is the output from the CMVN module

2.1.7 Splicing

In splicing, the output array is a vector of length 117.

At first, it is initialized to zero. As new 13 incoming values come, the first 13 values of the array are shifted out. These incoming values get appended as the last 13 values of the output vector. Thus a vector of 117 values are sent out to the LDA module.

2.1.8 Linear Discriminant Analysis

LDA is used in pattern recognition, machine learning etc, to find a linear combination of features that separates or characterizes two or more classes of objects or events. In simple terms, LDA is classification - the act of distributing things into classes or categories groups, of the same type.

In simple terms, LDA is classification - the act of distributing things into classes or categories groups, of the same type.

The incoming vector of 1 *117 samples is multiplied by the LDA matrix of dimensions 117 * 40 to get a vector of 40 features. These vectors are then sent to the back end of the ASR system for further processing.

2.2 Back End of an ASR system

The back end mainly constitutes of a Viterbi decoder. The decoder performs a forward movement across states followed by a trace back to determine the actual words spoken.

The Viterbi search begins with an empty hypothesis and then incorporates a stream of information from the obtained features to develop a set of active hypotheses, which are modelled as states by the HMM. The search algorithm maintains a list of active states at each point in time and the likelihood of all reachable states at the next time step is calculated. HMM makes use of two models viz. WFST and GMM to perform decoding.

WFST provides a state machine representation wherein all transitions between states are represented using weighted, labelled arcs. The transition probability of reaching a state from the current state is the weight of WFST arcs leading from current state to another state. The GMM model adds an acoustic likelihood to these weights. At the end of an utterance, a trace back is done to evaluate the most likely sequence considering the path with the lowest costs.

CHAPTER 3

Hardware Implementation of Front End in Xilinx Vivado Design Suite

A few blocks of the front end viz. FIFO module, framing, windowing, FFT-PSD calculation were synthesized in Verilog Hardware Description Language.

Fixed-point representation was used to implement the system so as to minimize hardware. The optimal number of bits was decided by ensuring that the word error rate of the fixed point version doesn't deviate much from the existing floating point software version.

3.1 FIFO Module

The FIFO Module collects 10ms worth of incoming data (160 samples arriving at 16kHz) and then transmits them at 100MHz to the framing block.

The module is implemented using Xilinx FIFO Generator IP. The details for the module can be found in Xilinx FIFO Guide (2017). The FIFO was configured in AXI4, Packet FIFO mode with independent clocks for read and write. The read clock (the slave clock in AXI mode) is set at 16kHz and the write clock (the master clock in AXI mode) is set at 100MHz. The module has been configured to accept 16bit input.

In process Integrated systems, IP's communicate via AXI bus through an AXI interconnect. Currently, the standard is AXI4. It has the highest performance. When an address is supplied, it can do burst data transfer up to 256 data words. AXI standards follow a master-slave configuration. An AXI master can connect to many AXI slaves through AXI interconnect using a write data channel (or a read data channel from slave to master).

Packet FIFO mode delays the start of packet transmission until the end of packet is received. This ensures continuous availability of data once master-side transfer begins, thus avoiding source end stalling of the AXI data channel.

The functionality of the ports in the FIFO module is as explained below:

Table 3.1: Port Description of Xilinx FIFO Generator IP

Port	Functionality
wr_rst_busy	A Low signal on this port indicates that the FIFO is ready for a write operation
rd_rst_busy	A Low signal on this port indicates that the FIFO is ready for a read operation
m_aclk	Input port for master clock signal
s_aclk	Input port for slave clock signal
s_aresetn	An active low signal at this port causes the reset of entire core logic
s_axis_tvalid	A high signal applied on this port indicates the slave that the data being driven to it is a valid input
s_axis_tready	A high signal on this port indicates that the slave is ready to accept data
s_axis_tdata[m-1:0]	Its the input primary payload used to provide the data passing through the interface
s_axis_tlast	A high signal applied on this port indicates the slave that the last sample of the packet is being received
m_axis_tvalid	A high signal on this port indicates that the master is driving a valid output
m_axis_tready	A high signal on this port indicates that the master is ready to accept data
m_axis_tlast	A high signal on this port indicates that the last sample of the packet is being sent out by the master
m_axis_tdata[m-1:0]	Its the output primary payload used to provide the data passing through the interface
axis_wr_data_count[d:0]	This bus indicates the number of words written into FIFO
axis_rd_data_count[d:0]	This bus indicates the number of words read out of FIFO

The AXI interface protocol uses a two way valid and ready handshake mechanism. The data transfer takes place only if both ready and valid signals are high.

3.1.1 Synthesis report

LUT	FF	BRAMs	DSP	LUTRAM
81	186	0.5	0	3

Figure 3.1: Xilinx FIFO Generator IP Synthesis Report

3.2 Framing and Windowing

The FIFO module transmits 160 samples at 100MHz to the framing and windowing module. Framing is done by appending these 160 values to the last 240 values of the previous frame, thus obtaining a frame of 400 samples. These 400 samples are multiplied by their corresponding window coefficients. They are then zero padded with 112 zeros to get an array of 512 values and these 512 samples are sent to the FFT module. All the modules communicate with each other using the valid and ready handshake mechanism.

The window function used here is the Povey window function. It looks mostly like the Hamming window but goes to zero smoothly at the edges (to stop the high-energy low-frequency components bleeding into the high-frequency parts of the spectrum).

The windowing coefficients were pre-calculated using the following formulae:

$$a = 2 * M\_PI/(frame\_length - 1).$$
$$w(i) = pow(((cos(a * i) * (-0.5) + 0.5)), (0.85)).$$

where frame_length=400, M_PI =3.1415926 and i varies from 0 to 399. These coefficients are stored in BRAM inside the FPGA and accessed as and when required. The state diagram for the framing and windowing module is shown below :

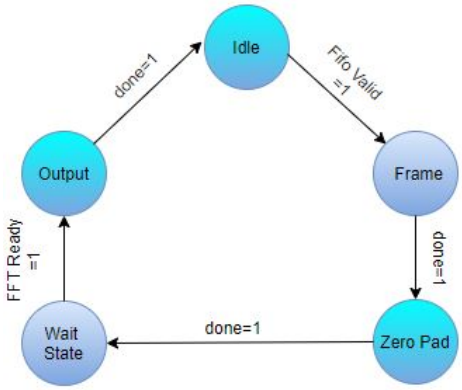

Figure 3.2: State Diagram of Framing & Windowing Module

Initially, the system will be in idle state. As soon as a valid data comes from FIFO module (indicated by valid high signal), the system transits from idle state to state: First Frame. As and when each sample is collected, it is multiplied by its corresponding windowing coefficient. When 400 samples are collected, frame ready signal is made low, so that, no new data will be sent out by FIFO. The data is then zero padded with 112 zeros and sent out to the next module and the frame ready signal is made high again. After this, the system enters wait state and waits for FIFO valid to go high again. As soon as the valid signal goes high, the system enters into the state: Next Frame. In the Next Frame 160 new samples are collected from the FIFO module and then Frame ready is made low again. The last 240 samples are shifted to the front end of the array and the new 160 samples are appended to the end. The system then goes to state: ZeroPad, then to wait state and back to state: Next Frame. This process continues until the detection is done.

3.2.1 Synthesis report

LUT	FF	BRAMs	DSP	LUTRAM	IO	BUFG
906	177	0.5	2	308	71	1

Figure 3.3: Synthesis Report of Framing & Windowing Module

3.3 FFT Module

The FFT module has been implemented using Xilinx inbuilt FFT IP. The details for the module can be found in Xilinx FFT Guide (2017)

The IP has been configured to accept 24bit fixed point input data at the rate of 100MHz and calculate 512 point FFT.

To minimize resources the FFT was used in Radix-2 Lite Burst I/O mode. The architecture uses only one adder/subtractor, thereby reducing resources at the expense of an additional delay per butterfly calculation.

It takes around 5670 clock cycles to calculate the output. Its interface is also configured in AXI4 mode.

3.3.1 Synthesis report

LUT	FF	BRAMs	DSP	LUTRAM
739	1628	2	4	302

Figure 3.4: Xilinx FFT IP Synthesis Report

CHAPTER 4

Hardware Implementation of Front End in Xilinx SDSoC

The SDSoC provides an embedded C/C++/OpenCL software platform including an easy to use Eclipse IDE and a comprehensive design environment for heterogeneous Zynq® SoC.

An application is written in C/C++/OpenCL code. The programmer has to identify a target platform and the set of the functions within the application to be compiled into the hardware. The SDSoC system then compiles the application into hardware and software to realize the complete system onto a Zynq device, including a boot image with firmware, operating system, and application executable. The target platform used in this project is a ZedBoard. It is a low-cost development environment.

The features of ZedBoard has been tabulated below :

Table 4.1: Technical Specifications of ZedBoard

Core	Dual-core ARM Cortex A9
Clock Frequency	667MHz
Memory	512MB DDR3 RAM 256MB Quad-SPI Flash External SD Card slot
Power	12v DC @ 3.0A(Max)
User Inputs	Slide switches Push button switches
Connectivity	10/100/1000 Ethernet USB OTG (Device/Host/OTG) USB UART
Analog	Xilinx XADC header Supports 4 analog inputs 2 Differential / 4 Single-ended

The architecture of an SDSoc system is shown in Fig: 4.1. In SDSoC, the hardware functions are compiled into hardware accelerators and are accessed through C run time calls. For each hardware function call, the arguments of the function are transferred between the CPU (Arm Processor)and accelerator. Data transfer between memory and

Table 4.2: Hardware Specifications of ZedBoard

DSPs	220
BRAMs	140
LUTs	53200
FFs	106400

accelerator is accomplished through data movers. Data transfer can either be done by direct memory access(DMA), or by the hardware accelerator itself.

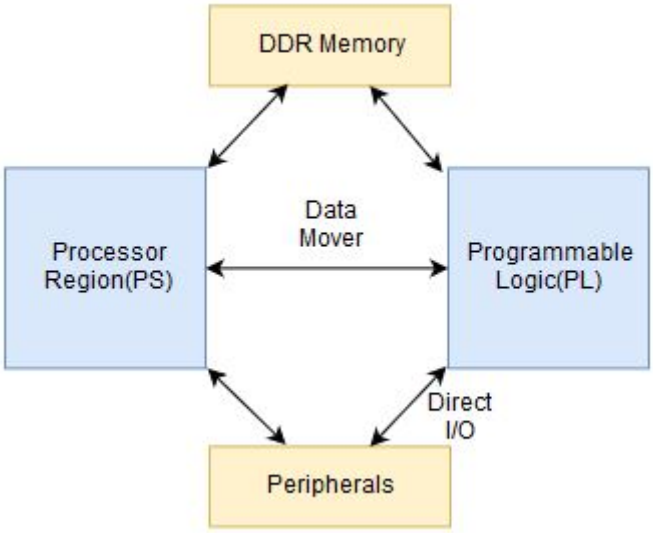

Figure 4.1: Architecture of SDSoC System

In this project, we have used a hardware accelerator data mover called the zero copy data mover. The zero copy mover helps to use array arguments that can function as both input and output. It tells the compiler that the array has to be kept in shared memory. To use zero copy data mover, the arrays should be stored in physically contiguous memory locations. Hence these arrays are allocated using sds_alloc and released using sds_free. The hardware function can directly access data from shared memory through the AXI bus interface.

An inbuilt performance counter class is present in SDSoC, which can be used to accurately measure the execution time of hardware functions based off the free running clock of the Arm processor. The class includes functions like sds_clock_counter and sds_clock_frequency. The function sds_clock_frequency returns the speed in ticks of the Arm Processor. The function avg_cpu_cycles can be used to find the average

number of CPU cycles taken by the hardware function.

The front end of the speech recognition system takes only negligible amount of time to execute as compared to the back end. Hence the task is to serialize front end as much as possible thereby reducing the hardware and power consumption. To reduce resources various HLS pragmas can be used. Pragmas are directives added to the source code to reduce latency, area, increase throughput and so on. SDSoC supports the use of Vivado HLS pragmas.

In this project we mainly made use of the following HLS pragmas namely::

$$\#pragma\ HLS\ inline$$
$$\#pragma\ HLS\ allocation\ instances = \ mul\ limit = \ 1\ operation$$
$$\#pragma\ HLS\ array\_partition\ variable = AB\ block\ factor = 10$$
$$\#pragma\ HLS\ array\_map\ variable = A\ instance = AB\ horizontal$$
$$\#pragma\ HLS\ array\_map\ variable = B\ instance = AB\ horizontal$$

A function to which HLS Inline pragma has been applied dissolves into its calling function and no longer can be treated as a separate entity. Inlining functions allow to share operations with the surrounding inlined functions and the caller function. Thus hardware can be effectively optimized. The HLS allocation pragma helps to limit resources. It limits the number of multiplications happening at a time to one. Various HLS and SDx pragmas can be found at SDx reference Guide (2017).

The number of bits used to represent input is 32 bit. Each 32-bit multiplication requires 4 DSPs. The entire front end part was coded in C++ and then hardware accelerated using SDSoC.

Initially, the entire blocks in the front end were inlined and resource allocation pragma was applied such that it will do only one multiplication at a time. It resulted in timing error since sharing one multiplier across the entire front end hardware requires a lot of multiplexers and long routing lines are required (huge capacitance) resulting in the delay of signals. Hence a total of 3 multipliers were required (12 DSPs) to resolve the timing error. Thus an initial code was successfully built and run in hardware and the output was verified.

The next step was to see if any further optimizations can be done by modifying the algorithms used.

4.1 Optimization Techniques

In this section, the various optimization techniques that were tried are described in short details. The main aim was to see if the number of multiplications, BRAMs, etc can be reduced.

4.1.1 Modified Window Function

Instead of Povey windowing, the following window function was tried:

$$w(i) = A/pow(2, 199 - i)\ 0 \leqslant i \leqslant 199$$
$$w(i) = A/pow(2, i - 200)\ 200 \leqslant i \leqslant 399$$
$$A\ is\ a\ scale\ factor\ in\ power\ of\ 2's.$$

The motivation to try this kind of windowing is that multiplication and division by power of 2 require only shifiting of bits. Tried varying A to minimize WER. However, the minimum %WER was found to be about 5% worse for this method and hence had to be discarded.

4.1.2 Texas Instrument FFT Algorithm for Real Inputs

The existing Cooley-Turkey algorithm for FFT was replaced using Texas Instruments FFT algorithm for real numbers(See TI real FFT paper (2001)). In this method 2N point FFT is calculated from N point FFT. This results in saving a few multiplications.

Assume g(n) to be a real valued sequence of 2N points.

$$\text{Let } x_1(n) = g(2n) \tag{4.1}$$

$$x_2(n) = g(2n+1)\ ,\ 0 \leqslant n \leqslant N-1 \tag{4.2}$$

$$\text{and } x(n) = x_1(n) + j * x_2(n) \tag{4.3}$$

$x_1(n)$ and $x_2(n)$ can be expressed in terms of x(n) as follows:

$$x_1(n) = \frac{x(n) + x^*(n)}{2} \tag{4.4}$$

$$x_2(n) = \frac{x(n) - x^*(n)}{2j} \tag{4.5}$$

where * is the complex conjugate operator.

The DFT of $x_1(n)$ and $x_2(n)$ viz. $X_1(k)$ and $X_2(k)$ can be expressed in terms of DFT of x(n) i.e X(k) as shown below:

$$X_1(k) = DFT[\frac{x(n) + x^*(n)}{2}] = \frac{X(k) + X^*(N-k)}{2} \tag{4.6}$$

$$X_2(k) = DFT[\frac{x(n) - x^*(n)}{2j}] = \frac{X(k) - X^*(N-k)}{2j} \tag{4.7}$$

Let G(k) be the DFT of g(n).

$$G(k) = DFT[g(2n) + g(2n+1)] \tag{4.8}$$

$$= \sum_{n=0}^{N-1} g(2n) W_{2N}^{2nk} + \sum_{n=0}^{N-1} g(2n+1) W_{2N}^{(2n+1)k} \tag{4.9}$$

$$= \sum_{n=0}^{N-1} x_1(n) W_N^{nk} + W_{2N}^k \sum_{n=0}^{N-1} x_2(n) W_N^{nk} \tag{4.10}$$

Thus,

$$G(k) = X_1(k) + W_{2N}^k X_2(k) \tag{4.11}$$

Using equations(4.6) and (4.7), we can write G(k) in terms of X(k) as :

$$G(k) = \frac{[X(k) + X^*(N-k)]}{2} + W_{2N}^k \frac{X(k) - X^*(N-k)]}{2j} \tag{4.12}$$

$$= X(k)\frac{[1 - jW_{2N}^k]}{2} + X^*(N-k)\frac{[1 + jW_{2N}^k]}{2j} \tag{4.13}$$

Let

$$A(k) = \frac{[1 - jW_{2N}^k]}{2} \tag{4.14}$$

$$B(k) = \frac{[1 + jW_{2N}^k]}{2j} \tag{4.15}$$

Thus G(k) can be expressed as follows:

$$G(k) = X(k)A(k) + X^*(N-k)B(k) \quad k = 0, 1, ..., N-1 \tag{4.16}$$

where,

$$X(N) = X(0) \tag{4.17}$$

We just need N points of G(k) to calculate PSD. Thus DFT of a 2N point real sequence can be calculated using one N point DFT and additional computations. 512 point normal FFT requires 9216 real multiplications. 512 point FFT calculated using TI FFT method requires 4096+256*8 =6144 multiplications. Thus we can save 3072 multiplications.

For bit reversing, the array indexes are also pre-calculated and stored in BRAMs. Since this method requires only 256 memory locations for input, the rest of 256 locations can be used to store A(k) or B(k). To store bit reversal coefficients only 256 locations are required as opposed to 512 locations in the normal FFT method. Twiddle factors also take up less space for a 256 point FFT. These freed memory locations can be used to store the other set of coefficients. Thus the number of multiplications can be saved without any additional memory usage.

The requirement in the front end is to reduce hardware as much as possible. The front end takes up only a small amount of time to execute. Hence TI real FFT algorithm was found to be best for the purpose.

4.2 Performance Estimate of Front End in SDSoC

Three implementations of the front end were run on hardware and the performance was estimated. In the first two implementations, coefficients were stored in on-chip BRAMs. In the third method, coefficients were stored inside the main function and accessed as and when required by the hardware function. This consumes far fewer BRAMs compared to the other two methods.

Method 1: Uses Cooley Turkey FFT, all coefficients are stored in BRAM. It takes around 62k clock cycles to execute

Method 2: Uses TI real FFT, all coefficients are stored in BRAM. It takes around 50k clock cycles to execute

Method 3: Uses TI real FFT, all coefficients are accessed from memory in burst mode. It takes around 150k clock cycles to execute

Method 3 was found to be the best since it utilizes lesser BRAMs and does fewer number of multiplications

Resource	Used	Total	% Utilization
DSP	12	220	5.45
BRAM	18	140	12.86
LUT	8092	53200	15.21
FF	5593	106400	5.26

Figure 4.2: Estimate for Method 1

Resource	Used	Total	% Utilization
DSP	12	220	5.45
BRAM	19	140	13.57
LUT	6964	53200	13.09
FF	5896	106400	5.54

Figure 4.3: Estimate for Method 2

Resource	Used	Total	% Utilization
DSP	12	220	5.45
BRAM	8	140	5.71
LUT	7938	53200	14.92
FF	6401	106400	6.02

Figure 4.4: Estimate for Method 3

4.3 Speeding up the MFCC module

4.3.1 Modified Cooley Turkey FFT algorithm to Mirror the Effect of Windowing and Mel-filtering

First, we tried to bring in the windowing coefficients inside twiddle factors. The following figure shows a 16-bit FFT structure into which the windowing coefficients were brought in. The same can be expanded for a 512 point FFT.

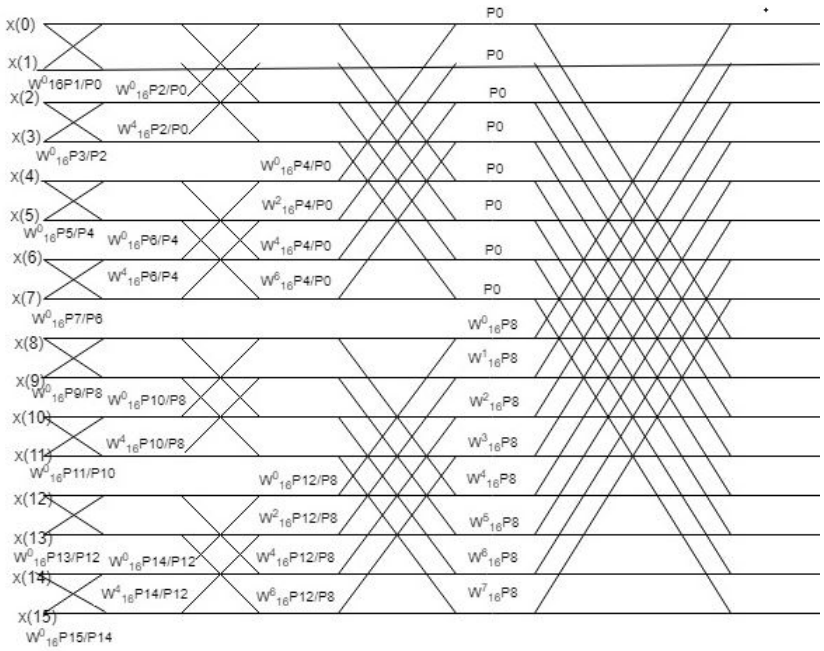

Figure 4.5: Incorporating Windowing Coefficients into Twiddle Factors.

In the figure W's represent twiddle factors and P's represent corresponding windowing coefficient of x(n).

This method will take a lot of BRAMs to store the twiddle factors since each stage in FFT has a different set of coefficients. This method doesn't have any multiplicative advantage over the former method of doing windowing first and then FFT. However, when Mel-coefficients are brought in, a few operations can be parallelized as explained below:

The square root of the 256 left and right Mel-coefficients were taken. It was then multiplied by coefficients for the last stage (obtained after incorporating windowing coefficients into FFT) and stored separately. Now in the last stage of FFT simultaneous

calculation of psd_left and psd_right can be done(psd_left and psd_right denote PSD multiplied with mel_left and mel_ right coefficients respectively).

The number of multiplications in this method is obviously higher than finding PSD first and then multiplying with Mel coefficients. This is because when we bring in mel coefficients into FFT, we have to do complex multiplications twice to get both left and right coefficients whereas we have only 512 real multiplications if it is done after finding PSD.

4.3.2 Comparison of Modified Cooley Turkey FFT Algorithm with TI real FFT algorithm

The blocks of front end starting from framing to PSD calculation were coded in C and hardware accelerated in SDSoC. A performance estimate was also done to evaluate the hardware that is being used up. The results are shown in the figure 4.7.

The hardware acceleration has been resource limited using the resource allocation HLS pragma to limit the number of parallel multiplications to a maximum of four. Each input is 32 bit in size and one 32-bit multiplication requires four DSPs. That is why the performance estimate is showing a count of 16 DSPs for both the methods. The BRAMs are used to store coefficients like window function, twiddle factors and so on.

The first method takes more BRAMs because it has to store twiddle factors for each stage of FFT separately. The first method takes up more FFs and LUTs too since it has to parallelize more operations. But we can speed up the process by around 1500 clock cycles using the first method.

Resource	Used	Total	% Utilization
DSP	16	220	7.27
BRAM	16	140	11.43
LUT	4492	53200	8.44
FF	3856	106400	3.62

Figure 4.6: Performance Estimate of method using Modified Cooley Turkey FFT algorithm

Resource	Used	Total	% Utilization
DSP	16	220	7.27
BRAM	9	140	6.43
LUT	3803	53200	7.15
FF	3591	106400	3.38

Figure 4.7: Performance Estimate of method using TI real FFT algorithm

CHAPTER 5

Conclusion

The entire front end was optimized on hardware to use as less resources as possible.

It takes around 150k clock cycles (1.5ms) to process one frame. Approximately every 10ms we get a new frame. So the front end can take 10ms to process one frame of data. Hence the front end can be run on slower clocks and thus power consumption can be reduced.

REFERENCES

1. **Price, M.**, **J. Glass**, and **A. P. Chandrakasan** (2015). A 6 mw, 5,000-word real-time speech recognizer using wfst models. *IEEE Journal of Solid-State Circuits*, **50**(1), 102–112. ISSN 0018-9200.

2. **SDx reference Guide** (2017). Sdx pragma reference guide. `https://www.xilinx.com/support/documentation/sw_manuals/xilinx2017_1/ug1253-sdx-pragma-reference.pdf`.

3. **TI real FFT paper** (2001). Implementing fast fourier transform algorithms of real-valued sequences with the tms320 dsp platform. `http://www.ti.com/lit/an/spra291/spra291.pdf`.

4. **Xilinx FFT Guide** (2017). Fast fourier transform v9.0. `https://www.xilinx.com/support/documentation/ip_documentation/xfft/v9_0/pg109-xfft.pdf`.